

1996

## Analysis of (iso)surface reconstructions: Quantitative metrics and methods

Tracey Allen Beauchat  
*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Beauchat, Tracey Allen, "Analysis of (iso)surface reconstructions: Quantitative metrics and methods" (1996). *Dissertations, Theses, and Masters Projects*. Paper 1539623885.  
<https://dx.doi.org/doi:10.21220/s2-k33s-h070>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# UMI

A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA  
313/761-4700 800/521-0600



ANALYSIS OF (ISO)SURFACE RECONSTRUCTIONS :  
QUANTITATIVE METRICS AND METHODS

---

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Tracey Allen Beauchat

1996

**UMI Number: 9720972**

**Copyright 1997 by  
Beauchat, Tracey Allen**

**All rights reserved.**

---

**UMI Microform 9720972  
Copyright 1997, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized  
copying under Title 17, United States Code.**


---

**UMI**  
**300 North Zeeb Road**  
**Ann Arbor, MI 48103**

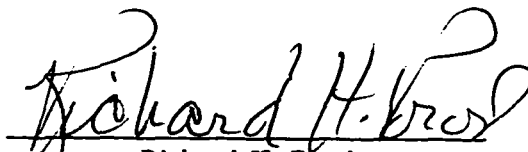
## APPROVAL SHEET


This dissertation is submitted in partial fulfillment of  
the requirements for the degree of

Doctor of Philosophy

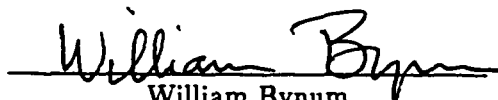
  
Tracey A. Beauchat

Approved, October 1996

  
Richard H. Prosl  
Thesis Advisor

  
Weizhen Mao

  
Stephen K. Park

  
William Bynum

  
Joseph S. McDonnell  
E-OIR Measurements, Inc.

*To my parents ...*

# Contents

<b>Acknowledgements</b>	ix
<b>List of Tables</b>	x
<b>List of Figures</b>	xi
<b>Abstract</b>	xiii
<b>1 Introduction</b>	2
1.1 Software/Visualization Verification and Validation . . . . .	2
1.2 Related Work . . . . .	3
1.2.1 Magnusson, et al., 1988 . . . . .	3
1.2.2 Pommert, et al, 1989 . . . . .	5
1.2.3 Marschner and Lobb, 1994 . . . . .	7
1.2.4 Patel, et al., 1996 . . . . .	9
1.3 Research Contribution – Beauchat, 1996 . . . . .	10
1.4 Thesis Overview . . . . .	11



<b>2</b>	<b>Background</b>	<b>13</b>
2.1	Volumetric Data . . . . .	13
2.2	Visualization System . . . . .	14
2.2.1	Grid Structures . . . . .	16
2.2.2	Volumetric Data Interpretations . . . . .	18
2.3	Volumetric Rendering Techniques . . . . .	19
2.4	Error . . . . .	20
<b>3</b>	<b>Surface Reconstruction Techniques</b>	<b>22</b>
3.1	Marching Cubes . . . . .	22
3.1.1	Configuration Table . . . . .	23
3.1.2	Algorithm . . . . .	24
3.2	Surface Tracking . . . . .	25
3.2.1	Background . . . . .	25
3.2.2	Algorithm . . . . .	27
3.3	Contour Triangulation . . . . .	27
3.3.1	Definitions . . . . .	28
3.3.2	Algorithm . . . . .	29
3.4	Contour Lofting . . . . .	30
3.4.1	Formalization - B-Splines . . . . .	30
3.4.2	Formalization - Cardinal Splines . . . . .	32
3.4.3	Algorithm . . . . .	33
<b>4</b>	<b>Metrics for Quantitative Analysis</b>	<b>35</b>
4.1	Background . . . . .	35

4.2	Metric Definitions . . . . .	37
4.2.1	GSAP . . . . .	37
4.2.2	VP . . . . .	38
4.2.3	PDP . . . . .	39
4.2.4	IVP . . . . .	40
4.3	Metric Computation . . . . .	42
<b>5</b>	<b>Reconstruction Quantification (sub)System</b>	<b>44</b>
5.1	RQS . . . . .	44
5.1.1	Property Computations . . . . .	45
5.1.2	Metric Computations . . . . .	47
5.1.3	Analysis . . . . .	48
5.2	An Implementation . . . . .	49
5.2.1	Tcl/Tk Language . . . . .	50
5.2.2	RQS Extensions . . . . .	51
5.2.3	RQS Example . . . . .	54
<b>6</b>	<b>Results</b>	<b>57</b>
6.1	Introduction . . . . .	57
6.1.1	Data Generation . . . . .	58
6.1.2	Analytical Data Functions . . . . .	59
6.1.3	Resampled Data Functions . . . . .	59
6.2	Analytical Data . . . . .	60
6.2.1	GSAP <sub>abs</sub> . . . . .	61
6.2.2	PDP . . . . .	64

6.2.3	IVP <sub>mse</sub> - Grid Variance	65
6.2.4	IVP <sub>mse</sub> - Threshold ( $\tau$ ) Variance	66
6.3	Resampled Data	69
6.3.1	Resampled Analytical Data	69
6.3.2	Resampled CFD Data	70
<b>7</b>	<b>Conclusion/Future Work</b>	<b>76</b>
7.1	Review	76
7.2	Guidelines	77
7.3	Topics for Further Study	78
<b>A</b>	<b>Volumetric Data Sets</b>	<b>80</b>
A.1	Analytical Functions	80
A.2	CFD Data	85
A.3	Volumetric Data File Format	87
<b>B</b>	<b>Definite Integral Approximation</b>	<b>90</b>
B.1	Background	90
B.2	Monte Carlo Integration	93
B.3	Interval Estimation	94
B.4	Implementation Issues	96
<b>C</b>	<b>RQS Extension to Tcl/Tk</b>	<b>98</b>
C.1	Data Structures	98
C.2	Objects	100
C.2.1	Functions	102

C.2.2	Volumes	108
C.2.3	Surfaces	111
C.3	Reconstruction Algorithms	115
C.4	Metrics	120
C.5	Rendering	124
C.5.1	OpenGL Raster Widget	124
C.5.2	OpenGL Rendering Commands	127
C.5.3	GLU Rendering Commands	144
C.5.4	RQS Rendering Commands	148
<b>Bibliography</b>		<b>152</b>

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Richard H. Prosl, for his guidance throughout my educational endeavors at the College of William & Mary. Dr. Prosl allowed me the flexibility to explore several avenues before pursuing the research presented in this dissertation. These additional avenues provided me with a deeper knowledge of other areas in the vast domain of computer graphics, and this knowledge provided me with greater insights into the problems presented by this dissertation. Dr. Prosl's guidance extended well beyond my scholastic education, and made my experience at the College of William & Mary more enjoyable. I would also like to thank Dr. Stephen Park, Dr. Weizhen Mao, Dr. William Bynum, and Dr. Joseph McDonnell for serving on my dissertation committee.

I must also acknowledge the friends and acquaintances that I obtained at The College of William and Mary. Without them, life in Williamsburg would have meant a less enjoyable existence. Among these, there are three that I must mention by name. First, Dr. Tracy Camp and Dr. (ahem) Clayton Johnson are two friends that have been with me from the beginning. We have shared many times, good and bad, and they will always be in my thoughts. It also must be stated that this may be the only time I mention Matt with "Dr." attached to his name. This remark should be received by all with the humor intended. Finally, I must mention Felipe Perrone. He is the truest of friends, and could quite possibly be a long lost brother if it weren't for the fact that he is a native Brazilian.

Finally, I am indebted to my parents for their love and support. They have always supported me in whatever endeavors I chose to pursue. It was their undying support and enthusiasm that kept me wanting to do more. Without them, I would not have accomplished all that I have.

# List of Tables

1.1 Reconstruction Filters . . . . .	7
6.1 Statistics for IVP on CFD Data with $\tau$ Varied . . . . .	74

# List of Figures

1.1	Ringing . . . . .	8
2.1	A Simplified End-to-end Visualization System . . . . .	14
3.1	15 Possible Voxel Configurations . . . . .	23
3.2	Vertex and Edge Numbering . . . . .	24
3.3	A Boundary Face . . . . .	26
3.4	Face Configurations . . . . .	26
3.5	$\delta\gamma$ Tracking Direction . . . . .	26
3.6	Planar contours . . . . .	28
5.1	The Reconstruction Quantification System . . . . .	45
6.1	Trilinear Interpolation . . . . .	59
6.2	GSAP <sub>abs</sub> on Sphere Data . . . . .	61
6.3	GSAP <sub>abs</sub> on Cone Data . . . . .	61
6.4	Contour Triangulations of the Sphere . . . . .	62
6.5	Circle Sampling . . . . .	63
6.6	PDP on Sphere Data . . . . .	64
6.7	PDP on Cone Data . . . . .	64

6.8	IVP on Sphere Data with Grid Varied . . . . .	65
6.9	IVP on Cone Data with Grid Varied . . . . .	65
6.10	IVP on Sphere Data . . . . .	66
6.11	IVP on Cone Data . . . . .	66
6.12	IVP on Sphere Data with $\tau$ Varied . . . . .	67
6.13	IVP on Hyperboloid Data with $\tau$ Varied . . . . .	67
6.14	Surface Tracking Reconstruction of a Hyperboloid of Two Sheets . . . . .	68
6.15	IVP on Volume Sphere Data (Grid Varied) . . . . .	69
6.16	IVP on Volume Cone Data (Grid Varied) . . . . .	69
6.17	IVP on Volume Sphere Data ( $\tau$ Varied) . . . . .	70
6.18	IVP on Volume Hyperboloid Data ( $\tau$ Varied) . . . . .	70
6.19	IVP on Ellipsoid CFD Data with Grid Varied . . . . .	71
6.20	IVP on Ship CFD Data with Grid Varied . . . . .	71
6.21	Reconstructions of Ellipsoid CFD Data . . . . .	73
A.1	Derivation of the Sphere Distance Function . . . . .	81
A.2	Derivation of the Cylinder Distance Equation . . . . .	82
A.3	Derivation of the Cone Distance Equation . . . . .	84
A.4	VDF Format File Header . . . . .	89
B.1	The Uniform(a,b) pdf . . . . .	91



## ABSTRACT

Due to sampling processes volumetric data is inherently discrete and most often knowledge of the underlying continuous model is not available. Surface rendering techniques attempt to reconstruct the continuous model, using isosurfaces, from the discrete data. Therefore, it natural to ask how accurate the reconstructed isosurfaces are with respect to the underlying continuous model. A reconstructed isosurface may look impressive when rendered (“photorealism”), but how well does it reflect reality (“physical realism”)?

The users of volume visualization packages must be aware of the short-comings of the algorithms used to produce the images so that they may properly interpret, and interact with, what they see. However, very little work has been done to quantify the accuracy of volumetric data reconstructions. Most analysis to date has been qualitative. Qualitative analysis uses simple visual inspection to determine whether characteristics, known to exist in the real world object, are present in the rendered image. Our research suggests metrics and methods for quantifying the “physical realism” of reconstructed isosurfaces.

Physical realism is a many faceted notion. In fact, a different metric could be defined for each physical property one wishes to consider. We have defined four metrics — Global Surface Area Preservation (GSAP), Volume Preservation (VP), Point Distance Preservation (PDP), and Isovalue Preservation (IVP). We present experimental results for each of these metrics and discuss their validity with respect to those results.

We also present the Reconstruction Quantification (sub)System (RQS). RQS provides a flexible framework for measuring physical realism. This system can be embedded in existing visualization systems with little modification of the system itself. Two types of analysis can be performed; *reconstruction analysis* and *algorithm analysis*. Reconstruction analysis allows users to determine the accuracy of individual surface reconstructions. Algorithm analysis, on the other hand, allows developers of visualization systems to determine the efficacy of the visualization system based on several reconstructions.

**ANALYSIS OF (ISO)SURFACE RECONSTRUCTIONS :  
QUANTITATIVE METRICS AND METHODS**

# Chapter 1

## Introduction

### 1.1 Software/Visualization Verification and Validation

An increasing trend is the use of scientific visualization in fault intolerant domains. For example, visualization systems are now used for surgical planning, analyzing the flow field around aircraft, and the research of biological entities. These applications require accurate depictions of the observed phenomena. If a visualization system does not yield accurate depictions of the phenomena, the results could be disastrous. Therefore, there is a growing need for the verification and validation of visualization software. End users of visualization systems must understand the software's strengths and weaknesses so that they may properly interpret and interact with what they see.

Several questions arise when visualizations are produced. Is the visualization valid? Does it show what truly exists in the data? The visualization process should do more than just produce a pretty picture. In scientific applications, the visualization must provide insight. In providing this insight, the software user must be able to distinguish between the characteristics of the data and the artifacts produced by the visualization process.

Is the visualization software correct or verifiable? In other words, does the software do what it is suppose to do, or what the software designer claims it does? To this end, methods are needed to measure how much results vary between software systems, and which results are more nearly correct. There is no reason to believe that any results are correct without software verification and validation. Typical accuracies should be known, and made available, in an easily interpreted way, to users of the system. Moreover, users should be given the means to perform their own verification and validation analysis.

Verifying and validating visualization software poses an interesting challenge. Visualization systems usually begin with incomplete information about a function, and through their processes, attempt to recreate that which has been lost. Reconstruction is a difficult process, and prone to error. Methods are needed for quantifying the error, and these methods must become part of the verification and validation process.

## 1.2 Related Work

Methods for validating the reproductions of scientific phenomena are beginning to emerge. In this section, we present some of the work that has emerged in this area, and in the next section, we discuss the new contributions of our work as presented in this dissertation.

### 1.2.1 Magnusson, et al., 1988

One of the earliest studies of realism in the visualization processes was performed by Magnusson, et al. [53]. Magnusson studied the artifacts produced by shaded surface display techniques in volume visualization. Shaded surface display techniques attempt to model how light is reflected from the surfaces in a scene. The intensity of light reflected at a surface location can be simulated using Phong's formula (see [19]).

Magnusson did not consider ambient light and specular reflectivity because the authors felt the characteristics they wished to study were present when only diffuse reflection was used. The authors also used a point light source located infinitely far away and a viewer located infinitely far away. Finally, the diffuse reflectivity coefficient was assumed constant across the entire surface. With these assumptions, Phong's equation only depends on the precision of the surface normals.

Volumetric data is inherently discrete and most often the only attribute known about an object is the value of the object's defining function at discrete locations. Therefore, methods are required for approximating the ideal surface normals from the functions values given at several discrete locations. Magnusson considered two normal generation schemes. The first scheme approximates the normals using the gradient vector

$$\nabla z = \left( \frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}, 1 \right) \quad (1.1)$$

where the function  $z$  is given by depth values in the zbuffer. The partial derivatives in Equation 1.1 were estimated using two dimensional Sobel operators. The second approach approximates the normals using the gradient vector

$$\nabla g = \left( \frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial z} \right) \quad (1.2)$$

The partial derivatives were approximated using three dimensional Sobel operators applied to the greyscale values of the original volumetric data.

Normals approximated using the above methods are dependent on the segmentation process used to define the object of interest because different segmentations will yield different depth buffers. Magnusson considered three segmentation techniques — grey value

thresholding, the magnitude of the gradient (first derivative) transition, and Laplacian zero-crossings (second derivatives). For better accuracy within the segmentation, they also investigated two localization schemes. The first was grid point accuracy (or nearest neighbor). The second used linear interpolation between the nearest neighbors.

Considering all combinations of the above techniques yields at least twelve possible combinations <sup>1</sup>. Using mostly qualitative judgments, Magnusson compared the resulting images with respect to artifacts appearing in the final image. Their quantitative analysis used graphs of the intensity distribution to explain the false ring structures that appeared in various images.

The most significant aspect of this research was its design. By comparing different combinations of the above schemes, the authors could analyze how one stage of the visualization process was affected by prior stage(s). The analysis not only considered how artifacts appeared in the rendered images, but also considered the cost/rendering quality tradeoff.

### 1.2.2 Pommert, et al, 1989

Pommert, et al. [69] had an objective similar to Magnusson in their research. Their objective was “to assess the quality of different surface rendering algorithms using both quantitative and qualitative measures of image quality.” Similar to Magnusson, Pommert defined image quality as being dependent on various components — accuracy of object segmentation, accuracy of the computed surface normals, and quality of the shading. Accuracy of the object segmentation was not regarded as pertinent to their research because it was not directly related to computer graphics. Phong’s equation was used to control shading quality.

---

<sup>1</sup>More actually exist because of available parameters for the various schemes.

Like Magnusson, Pommert made several simplifying assumptions so that image quality was only dependent on the accuracy of the computed surface normals.

Pommert considered four normal approximation methods in this research: zbuffer gradient, gray level gradient, adaptive grey level gradient, and marching cubes. Zbuffer gradient shading approximates normals from the depth values given in the zbuffer using the gradient vector shown in Equation 1.1. Pommert approximated the partial derivatives using a weighted sum of the forward and backward differences.

Gray level gradient shading uses central differences of the six nearest neighboring gray level values in the volumetric data to approximate the partial derivatives in Equation 1.2. For objects with thin surfaces (e.g., a hollow ball), the gray level gradient may yield poor results. Adaptive gray level shading attempts to correct this problem by varying the size of the neighborhood.

The first three normal approximation approaches render volumetric data directly. Marching cubes (see Section 3.1) differs from these approaches by combining traditional surface representations, via triangles, with normals at the grid points approximated by gray level gradients. Linear interpolation is used to compute normals at the triangle vertices from the grid point normals.

To measure the accuracy of the above normal approximation techniques, Pommert determined the angle between the computed surface normals and the ideal surface normals in the range from 0 to 90 degrees. To visualize this metric, Pommert rendered the objects as a pseudo-colored image with each pixel colored according to the above angle(s) at the location(s) of the object which projected into that pixel. Their analysis also included qualitative judgments based on whether known characteristics of the data appeared in the rendered image.

The authors acknowledged the fact that all aspects of image quality were not covered. However, the major drawback of their research is that they neglected to consider how various stages affected others. Various normal generation algorithms may perform better/worse under different segmentation and shading techniques, but using their research methods this cannot be determined.

### 1.2.3 Marschner and Lobb, 1994

Marschner and Lobb [54] presented a unique approach to analyzing the reconstruction of volumetric data. Since surface reconstruction is a filter, it can be performed and analyzed in the frequency domain. Marschner and Lobb used three dimensional extensions of traditional image processing reconstruction filters (see Table 1.1), and analyzed the results with respect to some traditional image processing errors.

Table 1.1: Reconstruction Filters

- trilinear interpolation
- windowed sinc
- two parameter ( $B, C$ ) cubic
- rotated cosine bell
- truncated Gaussian
- windowed 3-sinc
- cosine bell

Mitchell and Netravali [60] identified three common errors due to imperfect reconstruction; postaliasing, smoothing, and ringing (overshoot). Postaliasing occurs when the reconstruction filter is non-zero beyond the Nyquist frequency; even when the signal is sufficiently band-limited, the filter will incorrectly reconstruct parts of the aliased spectrum as part of the baseband spectrum. Smoothing occurs when high frequencies in a signal



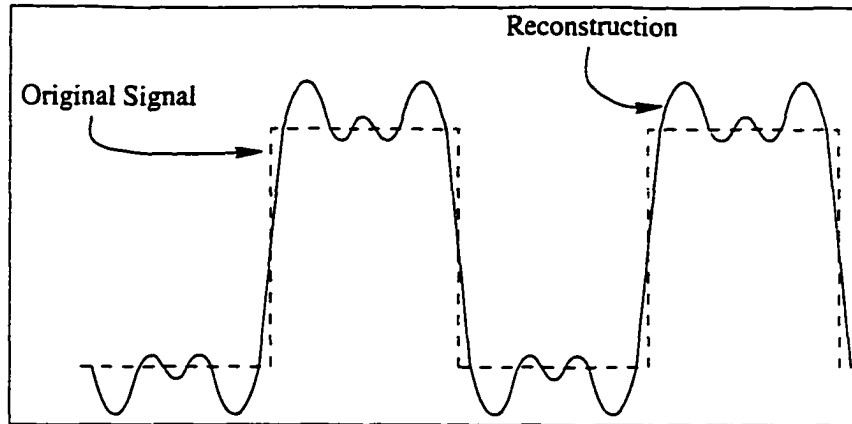


Figure 1.1: Ringing

are suppressed by low pass filtering. Finally, ringing occurs when high frequencies are abruptly truncated by a low pass filter. This results in oscillations in the neighborhood of a discontinuity as shown in Figure 1.1.

Marschner and Lobb developed a metric for each of the errors described above. They defined their smoothing metric to measure the difference between the filter in question and the ideal low pass filter with the same amount of energy within the Nyquist region. Similarly, the postaliasing metric was defined to measure the amount of energy outside of the Nyquist region. Finally, the overshoot metric measured how much overshoot occurs when the reconstruction filter is applied to the unit step function.

Marschner and Lobb pointed out that choosing an appropriate reconstruction filter requires considering the nature of the scene, how it was sampled, cost of using the filter, and what rendering algorithm will be used to display the reconstruction. Some filters are better at reducing certain errors at the cost of increasing others. For this reason, the authors chose to develop a metric for each error of interest. This allows a filter to be described by its abilities to reduce/increase particular errors. Thus, depending on the characteristics of the signal the most appropriate filter can be selected.

#### 1.2.4 Patel, et al., 1996

Visualization systems are being used increasingly in the simulation of surgical processes. These simulations are used for planning surgeries and as a tool for medical students to practice surgical methods non-invasively. Patel, et al. [68] report that the processes involved in these simulations must be validated with respect to their accuracy and measurability. The authors evaluated craniofacial surgical simulation methods with the goal "to define and test a methodology for comparing surgical simulations to postoperative outcomes."

The authors noted several possible sources of error in the simulation process: scanning, image reconstruction, image manipulation, and landmark identification and measurement. Since CT scanners have been validated, this source was not evaluated. To quantify the error in the other sources, the authors devised several tests based on objects with known (or easily computed) properties (e.g., cubes) and phantoms<sup>2</sup>. Among these were tests for the accuracy of linear measurements and landmark position measurements, a surface match rotation test, and a mass properties test.

In addition to the individual processes, the authors considered the surgical process as a whole. Phantom objects were scanned and rendered using a CT scanner and commercial software packages. The resulting computer models and phantom objects were manipulated using the same surgical procedures. The postoperative phantoms were rescanned and rendered, and compared to the postoperative computer models. Finally, comparisons were made with respect to predefined landmark positions, mass properties (e.g., volume and centers of mass), boolean differences, and depth-coded topographic maps.

---

<sup>2</sup>A *phantom* is a physical object designed to test some characteristic of a CT measurement system.

### 1.3 Research Contribution – Beauchat, 1996

Unfortunately, the methods presented by Magnusson and Pommert use qualitative analysis by asking the question, “do characteristics known to exist in the real phenomena exist in the reproduction?” This approach to verification and validation is inexact, and depends on the judgment and/or expertise of the viewer. Qualitative techniques are based primarily on the notion of “photorealism.” Photorealism is concerned with creating computer generated pictures which are indistinguishable from photographs.

The efforts of Magnusson and Pommert concentrated on the precision of the computed surface normals and their effect on the quality of the final image. It is important that characteristics of the data (e.g., sutures) appear in the final image because these characteristics attract the user’s attention to particular areas of interesting phenomena. However, there is a deeper issue that these efforts do not consider — the underlying representations. When a user manipulates the reconstructed object via cross-sections, for example, that user interacts with the underlying representation. If errors exist in the representation, then the visualizations which result from the manipulations may produce incorrect or unexpected results regardless of the quality of the surface normals.

Verification and validation techniques must move away from photorealism, and move towards the notion of “physical realism.” Physical realism is more concerned with how well the underlying structures that are used to represent the phenomena reflect the phenomena’s physical properties. This dissertation offers quantitative methods for analyzing the physical realism of a given reconstruction and the ability of visualization systems to preserve physical realism.

Similar to Pommert's metric for measuring the error between computed and ideal surface normals, in this dissertation, we formalize metrics based on physical properties. We also present methods for performing physical realism analysis. Like the design of Magnusson, our methods allow developers of visualization systems and users of those systems to determine the degree of physical realism, and investigate how various stages of the visualization process depend on prior stages and affect those which follow.

We take the view of Marschner and Lobb that having a single reconstruction metric may not be satisfactory because a single metric may miss interests of the user and characteristics of the data. For example, a doctor analyzing the reconstruction of a tumor might be interested in the ability of a reconstruction algorithm to preserve the surface area and/or volume of the tumor. A single value representing the overall preservation of physical realism may not reflect this ability because it could be tainted in some fashion by the algorithm's ability to preserve some other physical characteristic.

## 1.4 Thesis Overview

In Chapter 2, we present the background which lays the framework for the rest of this thesis. Volumetric data is described along with the grid structures on which it can be redefined, and the possible interpretations of the entities produced by the new definition. This chapter also describes a simplified end-to-end pipeline of the processes in the typical visualization system.

Chapter 3 describes four popular surface reconstruction algorithms. These algorithms are the marching cubes technique presented by Lorensen and Cline [51], a surface tracking algorithm presented by Shu and Krueger [77], a contour triangulation method presented by Ganapathy and Dennehy [22], and a contour method utilizing a modified spline lofting tech-

nique as presented by Sunguroff and Greenberg [80]. Each of these algorithms is analyzed with the validation techniques defined in this thesis.

Chapter 4 formalizes the notion of metrics based on physical properties, and defines new metrics for verifying and validating isosurface reconstructions and visualization systems. This chapter concludes with a discussion about how to ensure that the metrics are implemented correctly.

Chapter 5 defines a new framework in which the metrics presented in Chapter 4 should be applied. This framework defines two types of analysis, and describes a new (sub)system that can be directly and easily inserted into most visualization software. This gives the software the ability to produce verification and validation results.

In Chapter 6 we present the results obtained by applying the metrics of Chapter 4 to surface reconstructions produced by the techniques described in Chapter 3. We discuss each result presented with respect to what it tells us about the validity of the reconstruction, and what a collection of results tell us about the validity of the visualization system.

Finally, Chapter 7 concludes with a summary of the results, and some ideas for future work. This future work includes other possible metrics for investigation, and the creation of new reconstruction techniques which attempt to minimize the error with respect to a particular metric (or metrics).

## Chapter 2

# Background

### 2.1 Volumetric Data

Volumetric data typically consists of scalar or vector values specified throughout a continuous multi-dimensional space<sup>1</sup> according to some function ( $\hat{\mathcal{F}} : \mathfrak{R}^3 \rightarrow \mathfrak{R}$ ). This function, representing nature, may be as simple and regular as the analytical equation of a sphere, or it may be as complex as the density of tissue within a human head.

We can define a new function  $\mathcal{F} : \mathcal{G} \rightarrow \mathfrak{R}$  by sampling  $\hat{\mathcal{F}}$  on a superfine grid  $\mathcal{G}$ .  $\mathcal{F}$  represents an approximation to  $\hat{\mathcal{F}}$  to the best possible precision of the equipment being used to acquire the measurement. The domain of  $\mathcal{F}$  is no longer continuous and the range is no longer infinitely precise. However, with ever-changing technologies and proper software design, we assume that the measurement can be taken at any predetermined precision.

---

<sup>1</sup>We will only be concerned with scalar data specified throughout a three dimensional space in this research.

## 2.2 Visualization System

Volume visualization is concerned with the representation, manipulation, and rendering of volumetric data [41]. Figure 2.1 shows a simplified model of the typical end-to-end surface visualization system. This model takes as input the function  $\mathcal{F}$  defined throughout a three dimensional space. Unfortunately, in most cases, the computer is incapable of handling the complexity of  $\mathcal{F}$ . Therefore, before  $\mathcal{F}$  can be used by the visualization system it must be sampled. The first step of the visualization system samples  $\mathcal{F}$  on a grid. A new function,  $F$ ,

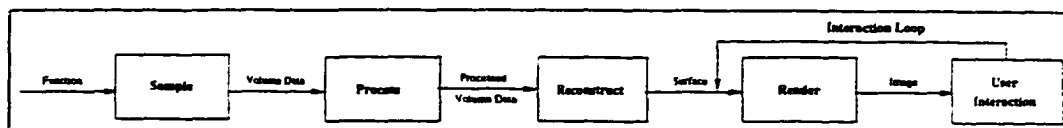


Figure 2.1: A Simplified End-to-end Visualization System

is defined by this sampling process. The precision of  $F$  is restricted by the storage capacities of the computer and visualization system.

The next stage is to process the sampled data, usually by quantization. Quantization reduces the amount of memory and disk space required to store the data by mapping the floating point values of  $F$  to a small subset of integers. If  $\max_F$  and  $\min_F$  are the maximum and minimum values of  $F$ , then a value  $v$  in the range of  $F$  can be uniformly quantized to the values  $0, 1, \dots, N$  using

$$v' = \left\lfloor \frac{v - \min_F}{\max_F - \min_F} N \right\rfloor$$

Other processing may include noise filtering and segmentation. Regardless of the processing, we still use the symbol  $F$  to represent the function after this stage.

Rendering volumetric data represents an interesting challenge because most of the graphics methods and hardware today are oriented toward surface representations. In the third stage, the system “converts” user specified parts of the volumetric data to a traditional

surface representation. Recall that the volumetric data is only defined on a discrete set of points. Therefore, this third stage must use interpolation to obtain unknown values at other locations within the domain of  $\mathcal{F}$ . This conversion process is known as reconstruction. Some of the popular reconstruction techniques can be found in [80, 51, 11, 77, 59]. Four of these techniques will be described in Chapter 3 of this dissertation.

Once the (iso)surface has been reconstructed it can be rendered in the fourth stage. This stage encompasses many separate processes, and each process influences the output by providing a number of controlling parameters. For example, a view transform must be defined to map the surface from its world coordinate space to an image coordinate space. The parameters of the viewing transform may include linear transformations (e.g., scaling and rotation), the view point, type of projection, and the viewport. The output from this stage is an image that can be viewed by the user.

As a fifth stage, the visualization system often provides some mechanisms for user interaction. This stage takes as input the rendered image from the prior stage, and provides mechanisms for manipulating that image. The mechanism often comes in the form of a graphical user interface (GUI) that allows the user to alter the parameters of stage four. For example, after the user views this image, s/he may want to rotate the object. If the visualization system allows this operation, it would give the user some means to change the appropriate view transform parameters in stage four. The visualization system may also provide more complex means of user interactions. For example, the visualization system might be used for surgical planning. In this case, the system would provide the user with a means of taking cross-sections through the reconstruction to view interior structures.



### 2.2.1 Grid Structures

As stated above, the second stage of the visualization system reduces the amount of data by sampling  $\mathcal{F}$  on a grid. Grids are generated by choosing a discrete subset of points in the domain ( $\mathcal{G}$ ) of  $\mathcal{F}$ . Grids can be classified by the regularity of their appearance (or alternatively by the regularity of the processes used to generate them). Grid generation processes often partition space into small volume units called voxels, and typically there is a one-to-one correspondence between voxels<sup>2</sup> and grid points.

In this section, we describe four classifications of grids (see [90][79]), and where appropriate their associated voxels. Because the research in this dissertation is concerned with three dimensional data, we describe the grids below with this in mind.

**Regular** This is the simplest grid. The nodes are generated as the intersections of three orthogonal sets of planes given by;

$$x = n\delta$$

$$y = n\delta$$

$$z = n\delta;$$

where  $n = \pm 0, \pm 1, \dots$  and  $\delta$  is some small, real-valued number fixed throughout the grid generation process.

Voxels are cubes with sides of length  $\delta$ .

---

<sup>2</sup>Voxels are analogous to pixels in two dimensions.

**Rectilinear** *Rectilinear* grids generalize *regular* grids by allowing each set of planes to be defined by a different  $\delta$ . The planes are given by;

$$x = n\delta_x$$

$$y = n\delta_y$$

$$z = n\delta_z;$$

where  $n = \pm 0, \pm 1, \dots$  and  $\delta_x, \delta_y, \delta_z$  are some small, real-valued numbers fixed throughout the grid generation process.

Voxels, in this case, are rectangular parallelepipeds with sides of length  $\delta_x, \delta_y$ , and  $\delta_z$ .

This grid occurs most often in data defined as a series of two dimensional slices, where the  $\delta$  between data slices is lesser than that on each slice.

**Structured** Structured grids, also known as curvilinear grids, occur when the voxels are allowed to become warped. Structured grids are usually generated by the intersection of three families of doubly parameterized surfaces given by

$$x = x_n(s, t)$$

$$y = y_n(s, t)$$

$$z = z_n(s, t);$$

where  $n = \pm 0, \pm 1, \dots$  and  $(s, t)$  comes from some fixed subset of  $\mathfrak{R}^2$ .

*Structured* grid voxels usually resemble warped parallelepipeds, and the association of voxels with grid points must be made with respect to the generating surfaces.

These grids are typically used in computational fluid dynamics (CFD) where the grid is wrapped around some object.

**Unstructured** This is the most general grid, and occurs when grid points may be specified individually. Topology is not necessarily implied in the point specification. Though this grid is also known as *random*, the point specification is never really random.

It may not be easy to associate voxels with the grid points in unstructured grids.

Unstructured grids are becoming more common in CFD because they allow a more dense specification of points in areas where interesting behavior occurs, and a less dense specification in areas of low interest.

The data sets used in this research have been sampled on either a regular grid or a rectilinear grid. Appendix A of this thesis describes the data sets used in this research.

### 2.2.2 Volumetric Data Interpretations

Given a grid specified on the domain of  $\mathcal{F}$ , the values of  $F$  are defined at each point of the grid by sampling  $\mathcal{F}$  in some way. When point sampling is used, the value of  $F$  at a grid point is precisely the value of  $\mathcal{F}$  at that grid point. The values of  $F$  can also be obtained through some averaging of the values of  $\mathcal{F}$  near the grid point in question. This process is known as area sampling.

Regardless of the sampling method used, it is often desired and usually necessary to use the values of  $F$  to assign values to all points in the domain of  $\mathcal{F}$ . Several approaches are common. One approach assigns each point in a voxel the same value as the grid point associated with it. In others, adjacent grid points define a volume called a computational cell; values are assigned to all points within the computational cell by some weighted average

of values at the vertices. It is common practice to use the term *vozel* to refer to both interpretations.

## 2.3 Volumetric Rendering Techniques

Since the representation of volumetric data differs from that of traditional computer graphics representations (e.g., lines and polygons), new approaches must be used to visualize the data on today's hardware. Currently, there are two complementary approaches; *volume rendering* and *surface rendering*.

Volume rendering uses methods which directly display voxels. Some methods, [84, 20, 9, 7, 32, 14], treat each voxel as a set of six orthogonal faces, and project each voxel onto the viewplane in a back-to-front order relative to the viewpoint. These methods typically rely on "good" normal generation methods to reduce the stair step effects inherent in the resulting surfaces. Other techniques [73, 17, 47] use ray-tracing combined with averaging and integration methods to obtain opacities and intensities which are used to determine the color for pixels in the view plane.

Surface rendering approaches, [80, 51, 11, 76, 91, 77, 59], reconstruct isosurfaces which exist in the volumetric data according to a user specified threshold or contours defined on each data slice. The reconstructed isosurfaces are stored in one of the more traditional computer graphics representations. The converted data can be rendered and displayed using traditional graphics algorithms, and can take advantage of available graphics hardware. The research in this dissertation is only concerned with surface rendering techniques, and Chapter 3 describes those techniques which this author investigated.

## 2.4 Error

There are many sources of error in a visualization system. In fact, each stage in Figure 2.1 introduces some form of error. The first stage computes samples of the underlying function at discrete locations in the domain of the function. This process introduces error by passing a new function, whose range is a discrete subset of the original function, onto the next stage of the visualization pipeline. This form of error is known as sampling error.

In the second stage, the sampled data from stage one is processed in any of a number of ways (e.g., uniform quantization and low pass filtering). This stage alters some, or all, of the data values. For example, if quantization is applied, the dynamic range of values is rounded and truncated as presented in Section 2.2. This processing increases the error present in the data.

The reconstruction process introduces error when the isosurface is incorrectly interpolated. This source of error may appear in a number of ways. The reconstruction algorithm may miss important features of the isosurface such as a sutures or crevasses. It also can introduce features that do not exist in the data. For example, contour reconstruction algorithms often produce holes in solid objects. An example of this behavior will be illustrated in Chapter 6.

The next stage involves rendering the isosurface. Rendering requires the computation of surface normals. There are a number of ways these normals can be computed, and each contains some form of error. This topic was studied by a number of authors, as presented in Chapter 1. The rendering process involves changing a continuous three dimensional surface representation into a discrete two dimensional image. One source of error from this process manifests itself as aliasing artifacts. One particular instance of aliasing is known as

stairstepping, and is seen as jagged edges in the image. These sources of error have been well studied in the field of image processing. Other sources of error in the rendering process may include the incorrect display of hidden surfaces, improper shading, etc.

## Chapter 3

# Surface Reconstruction Techniques

This chapter will describe a collection of surface reconstruction techniques which fall under the surface rendering class of algorithms. We use these techniques for our results and discussion in Chapter 6. These methods were chosen because they represent some of the main approaches to the problem of surface rendering from volumetric data.

### 3.1 Marching Cubes

Lorenson and Cline [51] present a divide and conquer approach to surface reconstruction which consists primarily of two steps. First, the topology of the surface within a given voxel is first determined by classifying it as one of a small, finite number of possible configurations. Each configuration is uniquely defined by the edges intersected by the surface. Once the configuration is known, an approximation to the actual surface is generated as a triangular mesh using linear interpolation of the values at the vertices adjacent to the intersected edges.

### 3.1.1 Configuration Table

The classification of surface topologies inside a voxel can be implemented as a lookup table where each entry in the table contains the edges which are intersected by the surface. A voxel is defined by eight vertices, and there are two possibilities for each of the eight vertices: inside and outside the isosurface. Therefore, there are  $2^8 = 256$  possible configurations for an isosurface intersecting a voxel. However, if complementary configurations are considered

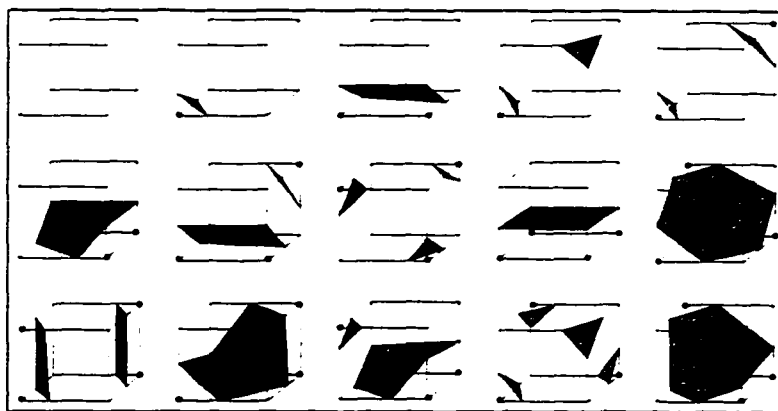


Figure 3.1: 15 Possible Voxel Configurations

equivalent, only four or less vertices have to be considered. This reduces the number of configurations to 128. Using linear operations, such as rotations, the number of distinct cell configurations can be further reduced to fifteen. These configurations are shown in Figure 3.1. In this figure, a black dot indicates that the vertex is inside the isosurface.

Each entry in the table is indexed by a bitcode. If the vertices of a voxel are numbered as shown in Figure 3.2, then a bitcode can be assigned to a voxel by assigning a one to the bit corresponding to each vertex that is inside the isosurface and a 0 to each outside vertex. For each bitcode, the table contains the edges intersected for the corresponding configuration. The edges are numbered as shown in Figure 3.2.



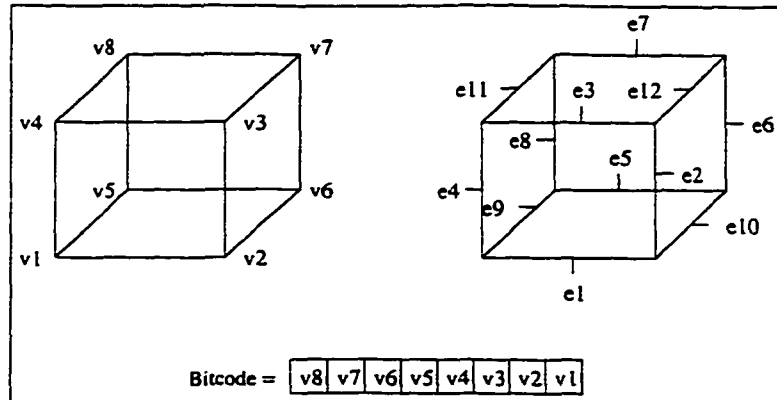


Figure 3.2: Vertex and Edge Numbering

### 3.1.2 Algorithm

*Marching Cubes* processes the data voxel by voxel, and determines where the isosurface intersects each voxel by classifying the vertices that define the computational cell as being inside or outside a user specified threshold ( $\tau$ ). If the value at a vertex is less than or equal to the threshold, it is classified as inside the isosurface. The isosurface will intersect an edge of the cell when the classification of two adjacent vertices differ. That is, when one vertex is inside ( $\leq \tau$ ) and one vertex is outside ( $> \tau$ ) the threshold. This classification determines the topology of the isosurface intersecting the voxel.

Once the topology for a given voxel is known, the intersection of the isosurface  $\mathcal{F}(x, y, z) = \tau$  with an edge can be approximated using linear interpolation on the values of  $\mathcal{F}$  at adjacent vertices. The interpolated point  $\mathbf{P}$  on edge  $e$  defined by voxel vertices  $\mathbf{V}_a$  and  $\mathbf{V}_b$  is expressed by

$$\mathbf{P} = (1 - t)\mathbf{V}_a + t\mathbf{V}_b \quad (3.1)$$

where

$$t = \frac{\tau - F(\mathbf{V}_a)}{F(\mathbf{V}_b) - F(\mathbf{V}_a)}$$

## 3.2 Surface Tracking

A solid in volumetric data sets can be identified as a connected set of voxels, and the surface of that solid as a connected set of boundary faces. Boundary faces exist between two voxels having differing values. *Surface tracking* algorithms construct the surface by traversing the voxels face by face beginning with a "seed" which is known to be a part of the surface. Each adjacent face is considered using tracking functions. If it is a boundary face on the desired surface, then it is included in the surface. The algorithm developed by Shu and Krueger [77] is rather complex. Therefore, only their terminology and the basic idea of the algorithm will be described here.

### 3.2.1 Background

Surface tracking algorithms require some method of determining the boundary faces. Thus, a segmentation process, typically, must be performed. The segmentation process should produce a binary scene where each voxel which is a part of the solid is assigned a 1 (called 1-voxels), and all other voxels are assigned 0's (called 0-voxels).

As shown in Figure 3.3, a *boundary face*, in this scene, will exist between two adjacent voxels if one is a 1-voxel and the other is a 0-voxel. Each voxel can have six boundary faces. Figure 3.4 shows these faces and their names;  $X$ ,  $\bar{X}$ ,  $Y$ ,  $\bar{Y}$ ,  $Z$ , and  $\bar{Z}$ . The name assigned to a face reflects the direction of its outward-pointing normal.  $X$  is the name of the face at which the outward-pointing normal points in the direction of increasing  $x$ .  $\bar{X}$  is the name

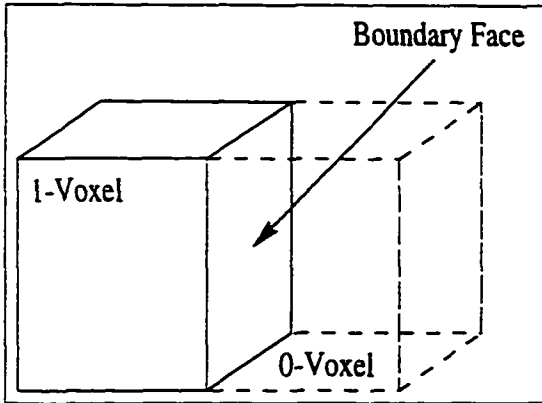


Figure 3.3: A Boundary Face

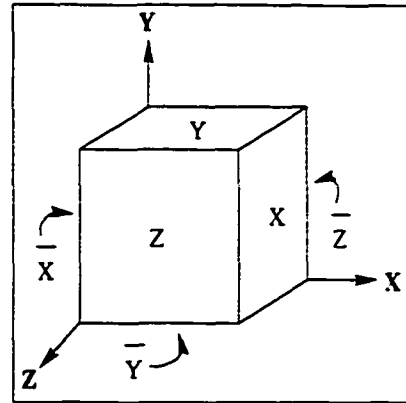


Figure 3.4: Face Configurations

of the face at which the outward-pointing normal points in the direction of decreasing  $x$ .  $Y$ ,  $\bar{Y}$ ,  $Z$ , and  $\bar{Z}$  are defined similarly.

Each voxel also has six *tracking directions*,  $\delta_a$ , for  $a = X, \bar{X}, Y, \bar{Y}, Z, \bar{Z}$ . The tracking direction is given by placing the thumb of the right hand in the direction of the normal for a boundary face. The direction that the fingers curl gives the tracking direction. Figure 3.5 shows the tracking direction  $\delta_Y$ . A *tracking function*  $T_a$  is associated with each tracking

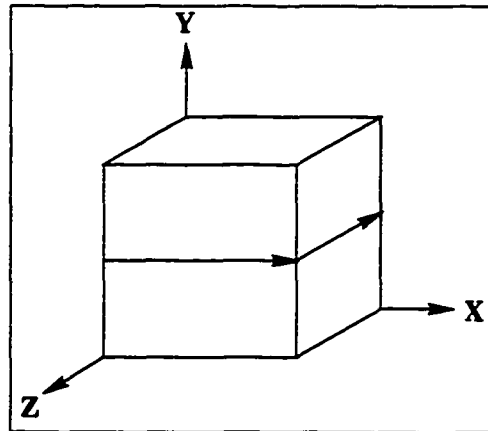


Figure 3.5:  $\delta_Y$  Tracking Direction

direction.  $T_a$  takes a boundary face as input and returns the boundary face adjacent to it in direction  $\delta_a$ . The process of tracking all faces in a particular direction is known as clearing the track.

### 3.2.2 Algorithm

The surface tracking algorithm uses three data structures.  $BF$  is the set of boundary faces comprising the surface. Faces are only added to this set when they are not already present. Therefore,  $BF$  must be optimized for the *is\_member* operator.  $Q_Y$  is a queue containing boundary faces to which the  $T_Y$  tracking function will be applied. Likewise,  $Q_Z$  is a queue of boundary faces to which the  $T_Z$  tracking function will be applied.

With this background, Shu and Krueger's *surface tracking* algorithm can be described simply as follows. The volumetric data is first conceptually divided into one voxel wide slices parallel to the  $x,z$ -plane. The  $T_Y$  tracking function is used to find all  $X, \bar{X}, Z,$  and  $\bar{Z}$  boundary faces in each slice, and the  $T_Z$  tracking function is used to find all  $Y$  and  $\bar{Y}$  boundary faces in each slice. Note that these two tracking functions are sufficient to find all boundary faces on the surface of the solid.

As stated earlier, the tracking process begins with a boundary face which is known to belong to the resulting surface. This face is called the seed face,  $f_{seed}$ . For simplicity, assume  $f_{seed}$  belongs one of the  $X, \bar{X}, Z,$  or  $\bar{Z}$  class of faces.  $Q_Y$  is initialized to contain  $f_{seed}$ , and the other two structures are initially empty. If  $Q_Y$  is not empty, the next boundary face in this queue is removed, and  $T_Y$  is used to clear the track. If  $Q_Y$  is empty,  $Q_Z$  is used. The algorithm continues to clear tracks until  $Q_Y$  and  $Q_Z$  are empty. Clearing the tracks is a complex process and, for the sake of brevity will not be described here.

## 3.3 Contour Triangulation

The general triangulation approach can be stated as follows. Given a set of points scattered throughout some three dimensional space, how can we best model the surface defined by

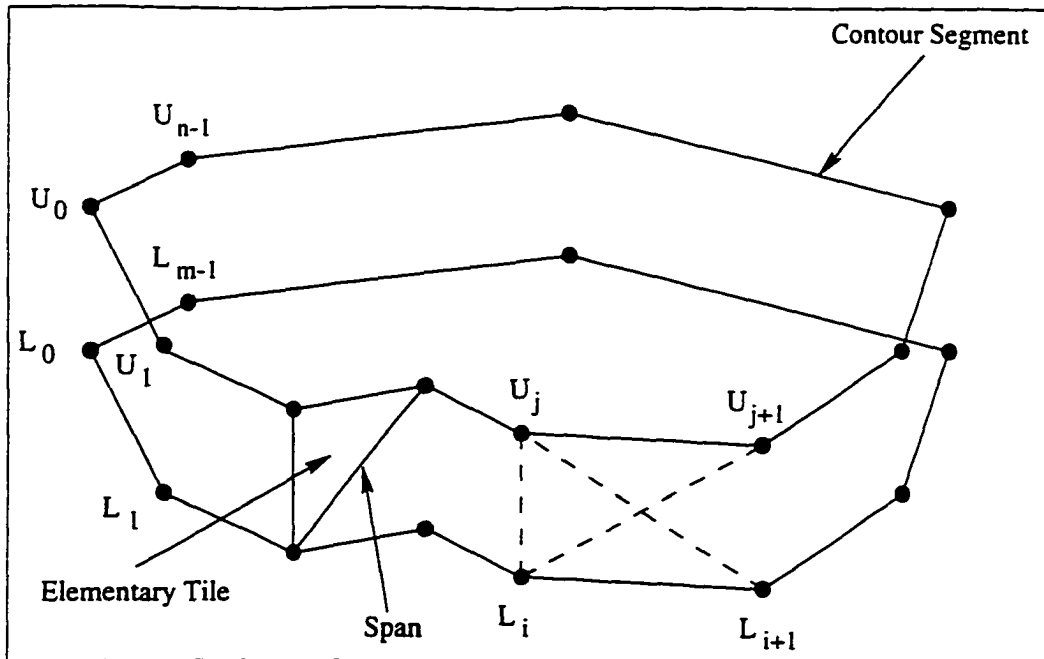


Figure 3.6: Planar contours

these points using standard computer graphics primitives? The simplest approach would be to model the surface using triangular surface patches. Approaches to this general problem are rather difficult [18]. A simpler problem arises when the set of points are distributed on several parallel planar contours as shown in Figure 3.6. Ganapathy, et al. [22] presented a heuristic approach as a solution to this problem. Their approach takes advantage of the coherence between adjacent contours. Adjacent contours typically have similar shapes, and are made up of relatively the same number of segments. Before the algorithm is described, a few definitions must be presented (refer to Figure 3.6).

### 3.3.1 Definitions

**definition 3.1** A **contour segment** is a linear approximation of the curve connecting two adjacent points on a contour.

**definition 3.2** A **span** is a segment connecting two points on adjacent contours.

**definition 3.3** *An elementary tile is the triangular facet defined by a contour segment and two spans connecting the end points of the segment with a common point on an adjacent contour.*

Given these definitions, it has been shown [21] that an “acceptable” surface must satisfy two constraints. First, a contour segment must appear in only one elementary tile between two adjacent contours. Second, if a span occurs as the left (right) span of an elementary tile, then it will appear as the right (left) span of exactly one other elementary tile in the set of tiles defining the surface.

A contour can be “redefined” in such a way that its perimeter is normalized. In these new contours, a weight  $\phi_i$  is defined for each elementary tile containing a contour segment from the lower contour.  $\phi_i$  is equal to the length of the  $i^{\text{th}}$  contour segment divided by the length of the entire contour. Thus,  $\phi_i$  represents the normalized length of the  $i^{\text{th}}$  contour segment. A weight  $\phi_j$  is similarly defined for each elementary tile containing a contour segment from the upper contour.

### 3.3.2 Algorithm

Elementary tiles are added to the surface in such a way that the absolute difference between the sum of the upper weights  $\phi_j$  and the sum of the lower weights  $\phi_i$  is minimized at all times. Define  $\Phi_l$  to be the normalized distance traveled thus far along the lower contour, and similarly define  $\Phi_u$ , for the upper contour. Given that we are at point  $L_i$  in the lower contour, and at point  $U_j$  in the upper contour, as shown in Figure 3.6,  $L_i L_{i+1} U_j$  and  $L_i U_j U_{j+1}$  are the only tiles which can be considered for addition into the set of elementary tiles defining the surface reconstruction.  $L_i L_{i+1} U_j$  is chosen if

$$|\Phi_l + \phi_i - \phi_j| < |\Phi_u + \phi_j - \phi_i| \quad (3.2)$$

Otherwise,  $L_i U_j U_{j+1}$  is chosen. In the first case,  $i$  is incremented by one and  $\Phi_i$  is incremented by  $\phi_i$ . The algorithm proceeds. In the second case,  $j$  is incremented by one and  $\Phi_{u_j}$  is incremented by  $\phi_j$  before the algorithm proceeds.

### 3.4 Contour Lofting

The contour method developed by Sunguroff and Greenberg [80] uses a modified lofting technique to interpolate the sectional curves with splines. The contour on each data slice is first fitted by uniform B-splines, and then the surface is interpolated between curves using Cardinal splines. For the discussion which follows we assume the reader is familiar with some spline theory terminology.

#### 3.4.1 Formalization - B-Splines

The first step in the contour lofting algorithm is to fit a B-spline to each set of contour points. To simplify the algorithm, the points defining a contour are assumed to be uniformly distributed with respect to arc length.

**definition 3.4** *Given a closed set  $\mathbf{V} = \{V_i : i = 0, 1, \dots, m\}$  of points defining a closed polygon, we can generate points on an approximating curve for the  $i^{\text{th}}$  segment of the polygon using the following matrix equation:*

$$P_i(U) = [U^3 \ U^2 \ U \ 1][B_{b-spline}][V_{i-1} \ V_i \ V_{i+1} \ V_{i+2}]^T \quad (3.3)$$

$$[B_{b-spline}] = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad (3.4)$$

*Curves defined by this equations are known as **periodic uniform B-splines**, and the set  $\mathbf{V}$  is known as a **control polygon**.*

In **definition 3.4**,  $U = (u - u_i)/(u_{i+1} - u_i)$  is the normalized parametric coordinate.  $u$  is the curve parameter, and  $u_i$  represents the value of this parameter at the  $i^{\text{th}}$  control point. Because the set of points is closed the following end conditions are imposed  $V_{-1} = V_m$ ,  $V_{m+1} = V_0$ , and  $V_{m+2} = V_1$ .

Using Equation 3.3, we can obtain any point on the contour given an appropriate value for the parameter  $U$ . Unfortunately, the set of points  $V$  defining the control polygon are unknown. From the contour generation process, we obtained a small subset of the points  $P_i(U)$ . Therefore, an inversion process can be used to obtain  $V$ . Given that there are  $m + 1$  points in the control polygon, we require  $m + 1$  independent conditions. These conditions will uniquely define a system of linear equations which, when solved, will yield the  $m + 1$  points defining the control polygon.

The necessary conditions can be obtained from  $m + 1$  points on the contour and Equation 3.3. The simplest way to obtain the conditions is by stipulating  $P_i(0)$  for the  $m + 1$  contour segments. From Equation 3.3 these points are given by

$$P_i(0) = (1/6)(V_{i-1} + 4V_i + V_{i+1}) \quad (3.5)$$

Because the coefficients for each point are constant we can conveniently write the system of linear equations as:





**definition 3.5** Given a set of points  $\{P^j(u) : j = 0, 1, \dots, n\}$  defining the control polygon, a Catmull-Rom spline interpolating the  $j^{\text{th}}$  segment is defined by

$$Q_j(v) = [V^3 \ V^2 \ V \ 1][B_{\text{catmull-rom}}][P^{j-1}(u) \ P^j(u) \ P^{j+1}(u) \ P^{j+2}(u)]^T \quad (3.7)$$

$$[B_{\text{catmull-rom}}] = \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \quad (3.8)$$

Again we normalize the curve parameter  $v$ ,  $V = (v - v_j)/(v_{j+1} - v_j)$ . Using  $P^{-1}(u) = P^0(u)$  and  $P^{n+2}(u) = P^{n+1}(u) = P^n(u)$  as the end conditions provides a “bending free” curve at the end points.

### 3.4.3 Algorithm

A given data set contains  $n + 1$  planar slices of data. For each slice, the first step is to generate the contour which defines the cross section of the object on the given plane. From the above discussion, it is assumed that each contour is described by  $m + 1$  points distributed uniformly with respect to arc length. Therefore the second step is to either raise or reduce the number of points defining the contour to the predetermined number  $K$ , and to ensure they are uniformly distributed. Next, the inversion technique described above is applied to fit a B-spline to this new contour. Once B-splines have been fitted to all contours, they are interpolated between contours using Catmull-Rom splines.

The original algorithm from [94] allowed user interaction at several stages of the surface generation process. For example, after a B-spline was fitted to the contour, the user could remove/add points to the control polygon to force the B-spline to better fit the contour. We desired each reconstruction method to be free from user intervention. Therefore, we removed those steps from our implementation.

A surface patch  $\mathcal{I}'_{i,j}$ , in this approach, is defined parametrically in the usual way by

$$\mathcal{I}'_{i,j}(u, v) = \sum_{n=-1}^2 \Phi_n(v) \sum_{m=-1}^2 \Psi_m(u) P_{i+m, j+n} \quad (3.9)$$

with

$$\Phi(v) = \frac{1}{2} [v^3 \ v^2 \ v \ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \quad (3.10)$$

and

$$\Psi(u) = \frac{1}{6} [u^3 \ u^2 \ u \ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad (3.11)$$

$P_{i,j}$  is the geometry matrix for the  $i^{th}$ ,  $j^{th}$  surface patch.

## Chapter 4

# Metrics for Quantitative Analysis

### 4.1 Background

The idea of “physical realism” is a many faceted notion, and thus measures of “physical realism” can be many. There can be a metric for each physical or topological property we may wish to consider. Metrics motivated from these physical and topological properties can be *global* or *local*. Global metrics measure the difference between properties of an object as a whole, for example, its surface area, or its volume. Local metrics measure the difference between properties of an object at points, for example, the displacement between a point on a reconstructed isosurface and the corresponding point on the ideal isosurface, or the discrepancy between the actual value of a function at a point on a reconstructed isosurface and the value intended.

The distinction can be clarified further by considering how each class of metrics is computed. For a local metric, the error is measured at points on the surface, and integration averages these error values across the whole surface. In this context, local metrics often

require the evaluation of a surface integral having the following form

$$\int_{v \in V} \int_{u \in U} H(u, v) \left\| \frac{\partial I(u, v)}{\partial u} \times \frac{\partial I(u, v)}{\partial v} \right\| du dv \quad (4.1)$$

where  $H(u, v)$  represents the function that measures the error at points on the surface. In this equation,  $\times$  represents the cross product operator,  $\| \cdot \|$  represents magnitude, and evaluation of this surface integral yields the average value of  $H(u, v)$  across  $I$ .

For a global metric, a property is computed for infinitesimal pieces of the surface and integration adds these values together. The metric is defined as the error between two such integrations. For example, surface area of a surface  $S$  can be measured using

$$\int_{v \in V} \int_{u \in U} \left\| \frac{\partial S(u, v)}{\partial u} \times \frac{\partial S(u, v)}{\partial v} \right\| du dv.$$

If  $I$  is the ideal surface and  $J$  is a reconstruction of  $I$ , the surface area properties would be measured using

$$Area_I = \int_{v \in V} \int_{u \in U} 1 \left\| \frac{\partial I(u, v)}{\partial u} \times \frac{\partial I(u, v)}{\partial v} \right\| du dv$$

and

$$Area_J = \int_{n \in N} \int_{m \in M} 1 \left\| \frac{\partial J(m, n)}{\partial m} \times \frac{\partial J(m, n)}{\partial n} \right\| dm dn,$$

and a metric could be defined using the following relative error

$$\frac{Area_I - Area_J}{Area_J}.$$

## 4.2 Metric Definitions

In this thesis we define four metrics. The first two are global metrics, and the other two are local metrics. We call these metrics global surface area preservation (**gsap**), volume preservation (**vp**), point distance preservation (**pdp**), and isovalue preservation (**ivp**). In the definitions that follow, we use  $\mathcal{I}$  to represent the real surface, and  $I$  to represent the surface reconstruction of  $\mathcal{I}$ .

### 4.2.1 GSAP

This metric measures the difference between the surface areas of  $\mathcal{I}$  and  $I$ . This is a global metric. Given the surface area of  $\mathcal{I}$  ( $\text{Area}_{ideal}$ ) and the surface area of  $I$  ( $\text{Area}_{est}$ ), we define two measures of interest;

$$\mathbf{gsap}_{signed} = \frac{\text{Area}_{est} - \text{Area}_{ideal}}{\text{Area}_{ideal}} \quad (4.2)$$

and

$$\mathbf{gsap}_{abs} = \frac{|\text{Area}_{est} - \text{Area}_{ideal}|}{\text{Area}_{ideal}} \quad (4.3)$$

The first measure gives a signed error. It tells how far the estimated surface area is from the ideal surface area, and it also tells in which direction the general error occurs. That is, for example, if  $\mathbf{gsap}_{signed} > 0$  then we have a fair idea that the “size” of the isosurface reconstruction, in general, exceeds that of the actual isosurface.  $\mathbf{gsap}_{abs}$  simply gives the magnitude of the error with no indication of direction.

We have defined these metrics as a relative error with respect to the ideal surface area. The relative error allows us to consider several such measures for determining the overall ability of a reconstruction technique to preserve surface area. Without a relative measure this would only be possible when the surfaces are described by the same units.

As a global metric, **gsap** has some drawbacks. It is not sensitive to local deviations between the reconstruction and the surface. Analysis based on **gsap** merely states whether the surface area of the reconstruction exceeds the surface area of the real surface. Furthermore, even if analysis states the surface areas are equal, this does not necessarily imply the reconstruction is perfect. Consider the simple case of the surface area of a golf ball (e.g., a sphere with divots). It is easy to construct another surface with the same surface area as the golf ball by replacing the divots with bumps having the same dimensions as the divots.

Another drawback of **gsap** is that it is not easily applied to general surface representations such as the spline surfaces of Section 3.4. The surface area of a parametrically defined surface  $I(u, v)$  is given by Equation 4.1 with  $H(u, v) = 1$ . Evaluation of this surface integral typically requires numerical integration, and these methods contain their own sources of error. An additional measure must be provided which quantifies this additional error.

Finally, **gsap** is not realistically applicable to data sets, such as flow fields, where physical surfaces do not exist. The **gsap** metric, however, is simple to understand, illustrates a global metric, demonstrates the application of global metrics to analyze “physical realism” in isosurface reconstructions, and may be an important metric in some medical applications (e.g., analysis of tumors).

#### 4.2.2 VP

A logical extension of **gsap** is volume preservation (**vp**). Volume preservation is another global metric which measures the difference between the volumes enclosed by two surfaces. We define two volume metrics analogous to the metrics defined for **gsap**. Given the volume enclosed by the reconstructed surface (  $\text{Volume}_{est}$  ) and the volume enclosed by the surface

being reconstructed (  $\text{Volume}_{ideal}$  ) the two metrics are defined by

$$\mathbf{vp}_{signed} = \frac{\text{Volume}_{est} - \text{Volume}_{ideal}}{\text{Volume}_{ideal}} \quad (4.4)$$

and

$$\mathbf{vp}_{abs} = \frac{|\text{Volume}_{est} - \text{Volume}_{ideal}|}{\text{Volume}_{ideal}} \quad (4.5)$$

As with **gsap**, the first metric gives a signed error, and the second gives the magnitude of the error. Again, we define these metrics as relative measures.

Since **vp** is also a global metric, it suffers from many of the same drawbacks as **gsap**. In addition to those, volume is more difficult to calculate for surface reconstructions. The computation of volume for polygonal mesh representations requires finding volumes of pyramids, and for general surface representations, triple integrals are required.

### 4.2.3 PDP

Provided that we are reconstructing the surface of an actual object (e.g., that of a sphere), we can measure the error between a reconstruction and an actual surface by computing the distance from each point on the reconstruction  $I$  to a “corresponding” point on the surface of  $\mathcal{I}$ . We define **pdp** using the surface integral given above (Equation 4.1)

$$\mathbf{pdp}(I(u, v)) = \int_{v \in V} \int_{u \in U} H(u, v) \left\| \frac{\partial I(u, v)}{\partial u} \times \frac{\partial I(u, v)}{\partial v} \right\| du dv \quad (4.6)$$



where

$$H(u, v) = \text{dist}(I_x(u, v), I_y(u, v), I_z(u, v))$$

and  $\text{dist}(\cdot)$  is a function which yields the distance between an arbitrary point and the “corresponding” point on the surface  $\mathcal{I}$ . Implicit in  $H(u, v)$  is a mapping of points on the reconstructed surface to points on the real surface. A perfect reconstruction arises when the distance from every point on the reconstructed surface to the “corresponding” point on the real surface is equal to zero, and the mapping between “corresponding” points is one-to-one.

The greatest difficulty in computing **pdp** is finding the proper point mapping. For surfaces defined by symmetric convex analytic functions, such as those described in Appendix A.1, finding this mapping is often a matter of finding a line segment from the point on the reconstructed surface to a point on the real surface which yields a normal to the real surface. When surfaces are not defined by this class of functions, this mapping may be difficult, or impossible, to find. In these cases, we turn to the metric defined in the next section.

#### 4.2.4 IVP

The **pdp** metric is one of the best measures of physical realism. However, in many cases surfaces do not actually exist, or we do not have the means to find the necessary point mapping. In these instances, **pdp** is not applicable. A solution to this problem is isovalue preservation (**ivp**).

$I(u, v)$  is assumed to approximate the isosurface of constant value given by  $\mathcal{F}(x, y, z) = \tau$ . Based on this assumption,  $\mathcal{F}(I_x(u, v), I_y(u, v), I_z(u, v))$  should equal  $\tau$  for all  $u$  and  $v$  in the domain of  $I$ . In practice, however,  $I(u, v)$  is never a perfect reconstruction, and so  $\mathcal{F}(I_x(u, v), I_y(u, v), I_z(u, v))$  is only approximately equal to  $\tau$ . **ivp** provides a measure of this error. Using Equation 4.1, we define **ivp** as an error measure relative to the threshold  $\tau$ .

$$\mathbf{ivp}_{mse}(\tau, I(u, v)) = \int_{v \in V} \int_{u \in U} H(u, v) \left\| \frac{\partial I(u, v)}{\partial u} \times \frac{\partial I(u, v)}{\partial v} \right\| du dv; \quad (4.7)$$

where

$$H(u, v) = \left( \frac{\mathcal{F}(I_x(u, v), I_y(u, v), I_z(u, v)) - \tau}{\tau} \right)^2 \quad \tau \neq 0.0$$

The **ivp** metric rectifies many of the drawbacks present in the other metrics defined above. For example, it does not require a specific class of surface representations, and point mappings are not needed. Finally, it can be computed for reconstructions of any surfaces. The ideal surfaces are not required. The **ivp** metric measures an algorithm's ability to reconstruct a particular subset of a function (the isosurface given by  $\tau$ ). Performing this analysis for all possible  $\tau$ 's describes an algorithm's ability to reconstruct the function  $\mathcal{F}$  itself.

Unfortunately **ivp** is not the "cure all" for quantitative analysis. A drawback is that **ivp** is not well behaved for functions whose values do not vary continuously throughout  $\mathcal{F}$ 's domain. CT data provides an example of this problem. In a CT scan of a human head there is often a discontinuous change in the values between surfaces of significantly different densities (e.g., between skull and soft tissue). When points on the reconstruction

lie in such regions of discontinuity, the value of  $\mathcal{F}(x, y, z) - \tau$  may be large even when the actual distance of that point from the ideal location on the real (iso)surface is small.

### 4.3 Metric Computation

Global metrics are typically straightforward to compute. For example, the output of the reconstruction techniques in Section 3.3 is a surface described by several polygonal facets: triangles in the case of 3.1 and 3.3, and rectangles in 3.2. If  $N$  is the number of triangles or rectangles describing the surface, then the surface area is given by  $\sum_{i=1}^N \text{Area}_i$ ; where  $\text{Area}_i$  is the area of the  $i^{\text{th}}$  facet. The area of each three and four sided facet is easily computed. Most global metrics can be computed in this fashion.

The computation of local metrics defined in terms of Equation 4.1 is not as straightforward. Most often a closed form solution will not exist, and the metric will have to be approximated using numerical approximation techniques. The technique used for the results in Chapter 6 is Monte Carlo integration. The Monte Carlo method computes the expected value of  $H(u, v)$  across the surface by computing

$$H(u, v) \left\| \left\| \frac{\partial I(u, v)}{\partial u} \times \frac{\partial I(u, v)}{\partial v} \right\| \right\|$$

at points on  $I(u, v)$ .

Proper selection of the points is critical. Improper selection can bias the result towards a particular part of the surface. Computation of the metric across a “good” part of the surface can yield a false sense of security. Likewise, computation of the metric across a “bad” part of the surface would yield the same lack of security. A proper selection of points should work towards preventing this bias from occurring. One way to reduce this bias is

to approximate the metric using an ensemble average. To further reduce bias, the set of points should be randomly selected.

Another pitfall inherent in the Monte Carlo method, and all approximation techniques, is uncertainty. Uncertainty is always implied by the term "approximation." Two desirable characteristics of the Monte Carlo method are that the amount of uncertainty present in the approximation can be bounded, and to some extent it can be controlled. The interested reader can read Appendix `refapp:monte` for a brief description of the Monte Carlo method for numerical approximation of integrals.

## Chapter 5

# Reconstruction Quantification (sub)System

### 5.1 RQS

A direct and inevitable outcome of implementing the metrics from Chapter 4 was the definition of the **Reconstruction Quantification (sub)System (RQS)**. **RQS** provides a flexible framework for measuring physical realism. This system can be embedded in existing visualization systems with little modification of the system itself, and the addition of **RQS** to current visualization systems can yield valuable insight into the phenomena described by the renderings of surface reconstructions produced by them.

Figure 5.1 shows **RQS** as part of the end-to-end visualization system shown in Section 2.2. As shown in this figure, quantification analysis can be conceptually viewed as a three step process; property computation, metric computation, and analysis. In most cases, it would be efficient to combine the property computations with the metric computations. In the following sections, we describe the process(es) performed at each step in the **RQS**

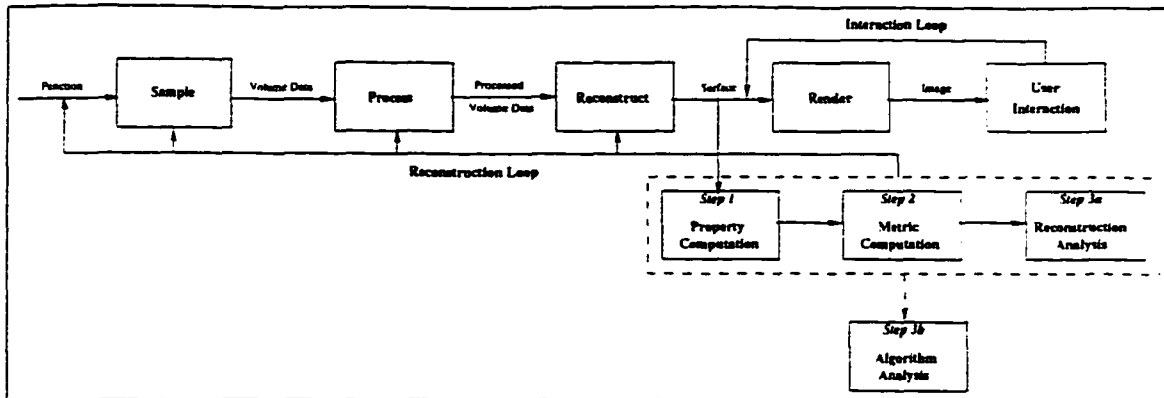


Figure 5.1: The Reconstruction Quantification System

system, and at the end of this chapter, we discuss our particular implementation of the system shown in Figure 5.1.

### 5.1.1 Property Computations

In the first step, we compute the property (or properties) measured by the metric(s) in step two. Step one requires three or more inputs. The first two inputs are the function which describes the ideal reconstruction ( $\mathcal{I}$ ) and the function which describes the reconstructed isosurface ( $I$ ). For example, if the isosurface of a sphere was reconstructed, the function describing the ideal surface would be

$$\hat{\mathcal{F}}(x, y, z) = (x - a)^2 + (y - b)^2 + (z - c)^2 - r^2,$$

where  $r$  is the radius of the sphere. Assuming, the surface was reconstructed using the Marching Cubes algorithm, the function describing the reconstructed isosurface would be polygonal mesh consisting of several triangular patches.

For each metric we are considering, step one also requires the corresponding property functions as input. For global properties (e.g., surface area), the property function will

take a single input. This input is a function describing the entity for which the property is being measured ( $\mathcal{I}$  or  $I$ ). If the metric of interest is **gsap**, then the property function for the ideal reconstruction would be

$$\mathbf{gsap}(\mathcal{I}) = 4\pi r^2$$

The property function, in pseudocode, for the marching cubes reconstruction of  $\mathcal{I}$  is

```

define function gsap(  $I$  )
begin
    area = 0.0
    foreach triangle  $i$  do
         $P_0 = I_i(0)$ 
         $P_1 = I_i(1)$ 
         $P_2 = I_i(2)$ 
        area = area + 0.5 * |(  $P_1 - P_0$  )  $\times$  (  $P_2 - P_0$  )|
    end
end

```

$I_i(j)$  for  $j = 0, 1, 2$  represents the  $j^{\text{th}}$  vertex of the  $i^{\text{th}}$  triangle.

For local metrics, however, the computation of the property requires two inputs. The first is, again, either  $\mathcal{I}$  or  $I$ . The second is a vector of sampling locations. These locations represents the points on  $\mathcal{I}$  (or  $I$ ) where the property will be measured. For example, if  $N$  samples are going to be used to approximate the integral in Equation 4.7, then the second input would be a vector of  $N$  randomly generated points (e.g.,  $(u, v)$ ). As stated in the previous chapter, care should be taken during the point generation process so the final result(s) will not be biased.

For each global metric being computed, the output of step one will be two scalar values. These values are the result of measuring the property for  $\mathcal{I}$  and  $I$ . For each local metric, the output will be two vectors that represent the measurements of the property at the generated sampling locations.

### 5.1.2 Metric Computations

Given the property computations from step one, we compute the metric(s) in step two. As stated earlier, coupling the metric computations with the property computations is usually more efficient for local metrics. For example, if coupling is not used, and if the ivp metric is to be approximated using  $N$  samples, step one must allocate memory to store  $N$  computations of the property given by

$$H_i(u, v) = \left( \frac{\mathcal{F}(I_x(u_i, v_i), I_y(u_i, v_i), I_z(u_i, v_i)) - \tau}{\tau} \right)^2$$

for  $i = 1 \dots N$ . Step one would also have to store the  $N$  values of  $u$  and  $v$  used above. Step two would then calculate the following average

$$\frac{1}{N} \sum_{i=1}^N H_i(u_i, v_i) \left\| \frac{\partial I(u_i, v_i)}{\partial u} \times \frac{\partial I(u_i, v_i)}{\partial v} \right\|$$

These two steps can obviously be combined to allow the metric computation to be a one pass process, and will require less computer memory. Note that coupling is not an issue with most global metrics.

Besides computing the metrics, we also organize the results in an informative manner for analysis. Thus, the output from this stage can take many forms. For most global metrics the output from this step will be a single scalar value. For example, **gsap** might be equal to the difference between two surface areas, or it might express the relative error in the reconstruction. Due to the global nature of this metric, it may be hard to conceive other forms for output.



Local metrics have a greater range of possibilities: a single scalar value such as that from Equation 4.1, a vector of values, a pseudocolored image, etc. For example, using *ivp*, we might consider two obvious options. First, we can give a numerical measure such as the relative error given in Section 4.1 by Equation 4.7. This single measure describes the overall “goodness” of the reconstruction. However, suppose we want to know if there are particular areas of the surface where the reconstruction is poor. We can then portray *ivp* graphically using an error image such as that used by Pommert, et al [69][81]. Whatever the form, the output should describe the information from step two in a meaningful and easily interpreted way.

### 5.1.3 Analysis

In this final step, *physical realism* analysis is performed based on the metric(s) computed in step two. We have defined two forms of analysis as indicated in Figure 5.1 by Step 3a and Step 3b. These are called *reconstruction analysis* and *algorithm analysis*.

#### Reconstruction Analysis

Reconstruction analysis allows the user/system-designer to make judgments about the quality of individual reconstructions because it is based on single reconstructions. Very often, the results from Step 2 would be used with the actual isosurface rendering to better interpret what is seen. For example, suppose we are viewing the reconstruction of a spherical object, and we notice an abnormal “bump” in an area which is otherwise smooth. To determine if this bump is an artifact produced by the reconstruction we could turn our attention to an error image for the *pdp* metric. If this image, shows a relative constant value across the area where the bump was seen, then there is a good chance the bump is not an artifact.

### Algorithm Analysis

Algorithm analysis is performed using the *reconstruction loop* shown in Figure 5.1. As indicated by the loop and the dashed box around steps 1-3a, algorithm analysis requires performing Step 1 and Step 2 (and maybe Step 3a) on some predetermined number of different reconstructions. For each reconstruction the function  $\mathcal{F}$ , the grid  $G$  (and hence the function  $F$ ), how the volumetric data is processed, and/or the (iso)surface threshold  $\tau$  can be altered. By altering the above variables and performing RQS analysis, the optimal system for the visualization task may be obtained.

When properly performed, algorithm analysis can be used to describe the overall "physical realism" tolerance of the visualization system. For instance, we might be interested in an algorithm's behavior when a minimal amount of data is available. To investigate this phenomena, we could systematically vary the density of the grid  $G$ , and analyze the trend which arises. As the grid becomes less dense we would expect the algorithm to perform worse, but what might be more important is how quickly the algorithm's performance degrades. An example of this analysis will be shown in Chapter 6.

## 5.2 An Implementation

In this section, we describe our implementation of the visualization system described in this thesis. Though not intended for professional use, this software provided a strong, flexible arena for evaluating the metrics and the methods defined in this thesis. We designed our system to be easily modified, enhanced, and embellished with other reconstruction methods, metrics, and rendering platforms through the use of dynamically loadable, independent program modules called dynamic storage objects (DSOs).

Modular programming and DSOs are relatively new concepts for software creation. Typically, libraries are created from modular (independent), reusable blocks of code, and visualizations are performed by choosing and linking only those modules required for the task at hand. These ideas allow programmers to create easily extendible and dynamically changeable environments.

### 5.2.1 Tcl/Tk Language

We chose to implement our system using the Tcl scripting language. The Tcl environment provides an application independent command interpreter, a small set of base data types, a group of built in commands and control structures for manipulating those data types, and a set of C routines for adding new structures and commands to the existing interpreter. Tcl's interpretive shell is similar to other shell languages (e.g., csh) in that it contains variables, control structures, and substitution mechanisms. Tcl, however, provides a more structured and easily extendible environment.

The interpretive nature of Tcl allows a user to enter commands and immediately see the results. If the intended results are not achieved the user can modify the commands immediately without the need to recompile the program. Once the desired set of commands have been obtained, they can be placed in a file. This file can be executed through the shell using a single command, thus, relieving the user from having to remember and retype the entire set of commands.

Tcl is also the basis of a X11 toolkit, called Tk. Tk adds a set of widgets to the Tcl interpreter. Buttons, labels, and text entry areas are just a few of the widgets present in Tk. In the Tk environment, the interpretive nature of Tcl, allows users and developers to quickly design and redesign graphical user interfaces for their applications. Tk contains a

set of similar C routines to easily incorporate additional widgets containing functionality not already present in the base set of widgets. For our research, we added an OpenGL drawing area widget to Tk with a minimum of difficulty.

### 5.2.2 RQS Extensions

Implementing the visualization system in Tcl/Tk required several extensions to the base language. We have grouped our extensions into five classes; data types, module loading, reconstruction, metric, and rendering. The data type class contains data structures for surface reconstruction and visualization. The module loading extension provides the mechanisms for dynamically loading the desired pieces of the system into memory. The reconstruction extension contains DSOs for the reconstruction algorithms (e.g., Marching Cubes). Likewise, the metric extension contains a single DSO containing the commands for the RQS subsystem. Finally, the rendering module contains Tk commands for rendering reconstructions and displaying images.

#### Data Structures

Since the only true data structures for Tcl are scalar values, strings, and lists, the language required several new entities for volumetric data storage, isosurface storage, and for performing isosurface reconstruction and visualization efficiently. We group these entities into two broad classes; data types and objects. The two classes differ in how they are defined and implemented.

Datatypes are created from the existing Tcl structures (e.g., lists), and merely define new interpretations for those structures. For example, the RGB color datatype is defined as a Tcl list containing three entries corresponding to the red, green, and blue components

of the color. Additionally, the entries are constrained to be integer values in the range 0...255.

Objects, on the other hand, are more complicated, and are implemented in C or C++. The object hierarchy includes three subclasses; functions, volumes, and surfaces. Many of these objects have corresponding Tcl commands and/or have Tcl commands which operate on them. These entities were added directly to the base system because they were required by the other modules. See Section C.1 of Appendix C, for a complete description of the new data structures and objects.

### Module Loading

The module loading extension was made part of the base shell. This extension provides two commands for loading and unloading DSOs

```
load_module    < DSO-filename > < init-function >
```

```
unload_module < module-name >
```

As indicated by its name, the first command loads a DSO into memory. It has two arguments. The first is the name of the file containing the DSO. This file must exist in one of the paths given by the *LD\_LIBRARY\_PATH* environment variable. The second argument is the name of the function within the DSO that initializes the Tcl/Tk commands. **load\_module** returns a unique string identifying the module. This string should be used as the argument to the **unload\_module** command to remove the DSO from memory when it is no longer going to be utilized.

### Reconstruction Modules

In our RQS package, there exists a separate module for each reconstruction algorithm implemented. This allows the user to only load those algorithms needed for the current

visualization task. We have implemented each of the algorithms described in Chapter 3 as Tcl commands. These commands have the following general form:

```
command volumeObject [options]
```

Each command takes as its input a volume object and a number of options. Many of the options are the same for each command. For example, each reconstruction command accepts *-low low\_threshold* and *-high high\_threshold* to specify the threshold range defining the isosurface. Each command may also take a small number of options unique to that command. The output of the reconstruction commands is a string representing the handle to the newly created surface object. See Section C.3 of Appendix C for a complete description of the Tcl commands in the reconstruction modules.

## Metric Module

The metric module contains implementations of the RQS metrics. In our implementation, each metric command has the following general form

```
command functionObject surfaceObject [options]
```

The first argument is a function describing the ideal surface/function reconstruction. This argument can be one of the Tcl analytical function objects described in C.2.1 of Appendix C, or a Tcl volume object (see Section C.2.2 of the same Appendix). The second argument is a surface object returned by one of the reconstruction commands described above. Finally, a metric command may take some number of metric specific options which modify the command's behavior, and alter its output.

Like the other modules, this module is only loaded when RQS analysis is going to be performed. Implementing the metrics as a separate module accomplishes our idea that the

RQS system should be easily embeddable into any visualization system. Our Tcl implementation of the metrics in Chapter 4 are described in Section C.4 in Appendix C.

## Rendering Modules

Rendering modules are used to generate graphic images and provide user interaction with those images. Thus, a complete rendering module implements the **Render** and **User Interaction** stages in Figure 5.1. We created one such module based on the portable graphics language, OpenGL. The OpenGL rendering module defines a new Tk widget called *glRaster*. This widget is very similar to Tk's canvas widget. However, it only accepts OpenGL rendering commands. Our module is not a complete implementation of OpenGL for Tk. The *glRaster* widget supports most of the OpenGL framebuffer modes (e.g., double buffering, rgba, color index, overlays, underlays, and depth buffer), and several of the OpenGL library commands are supported. We gave our Tcl implementation of the OpenGL commands the same names to ease the transition to the RQS environment. We provide a complete description of the *glRaster* widget, and an example of some of the Tcl implementations for the OpenGL library functions in Appendix C.

### 5.2.3 RQS Example

In this final section, we provide and discuss a simple example using Tcl and our extensions. This example performs RQS analysis on a reconstruction of a sphere. In the example below, *rqs.tcl*> is the command interpreter's shell prompt where commands are typed. Commands typed by the user are given in boldface, and responses returned by the shell are italicized.

```
rqs.tcl> load_module march.so InstallMarch  
module0  
rqs.tcl> load_module rqs.so InstallRQS  
module1
```

These first two lines load and initialize the necessary modules: the Marching Cubes algorithm module and the metrics module. The result for each of these commands is a string. These strings represent a handle which can be used later as input to `unload_module` to remove the module from memory.

```
rqs_tcl> fsphere -radius 1.0 -point { 0.0 0.0 0.0 } \  
-world { -1.5 -1.5 -1.5 } { 1.5 1.5 1.5 }  
sph0
```

Next, we create the object's function. In this example, we have created a function which represents a sphere having its center located at (0.0,0.0,0.0) and having radius 1.0. The domain of this object has been constrained to the range  $-1.5 \leq x, y, z \leq 1.5$ . This function is used in the next command to create the volumetric data.

```
rqs_tcl> volume byte 64 64 64 -function sph0  
vol0
```

The first argument represents the type of data to store in the volume. The next three arguments for the volume command are the number voxels along the  $x$ -axis, the number of voxels along the  $y$ -axis, and the number of voxels along the  $z$ -axis, respectively. The volume created above contains  $64 \times 64 \times 64$  voxels.

```
rqs_tcl> march vol0 -low 0 -high 212  
surf0
```

The volumetric data object is used as input to the Marching Cubes command. In this example, the isosurface corresponding to the threshold range  $0 \leq \tau \leq 212$  was reconstructed. This range corresponds to the surface of the sphere object created in line three of this example. The output of the `march` command is a surface object. This object, with the sphere function object, can now be used to perform RQS analysis.

```
rqs_tcl> gsap sph0 surf0 -relative  
gsap 1.591412e-01
```



In this example, we find the relative surface area error using the **gsap** metric. The final argument to the **gsap** command computes **gsap** relative to the sphere object using Equation 4.2. The result states that this is the output from the **gsap** metric, and shows that the reconstruction's surface area exceeds the surface area of the sphere by 0.159.

# Chapter 6

## Results

### 6.1 Introduction

This chapter presents results based on the metrics defined in Chapter 4. We have divided this chapter into two sections based on the class of volumetric data being reconstructed. These classes can be described as follows. The volumetric data from the first class is defined directly by the analytical functions described in Section A.1 of Appendix A. When computing metrics, the analytical function becomes the reference function. For example, when computing a sample value of *ivp*, we compare the isosurface value ( $\tau$ ) with the value of the analytical function at some location in the domain of the function.

The second class is based on resampled data. As in the first class, volumetric data is generated on a rectangular grid according to some function. The reconstruction process, however, is not applied to this data. It is applied to a volumetric data set generated by sampling the original volumetric data on a less dense rectangular grid. The original volume is used as the reference function when computing metrics. When computing a sample value of *ivp*, for example, the isosurface value  $\tau$  at a point  $P$  is compared to the value

of the original volume at  $P$ . Note, when  $P$  does not occur at a vertex of a voxel, linear interpolation would be used to find the value of the volume at  $P$ .

### 6.1.1 Data Generation

The data generation process for each class can be described by the following simple pseudocode.

```

volumeWidth = number of samples along X axis
volumeHeight = number of samples along Y axis
volumeDepth = number of samples along Z axis

worldMinX = minimum world X coordinate
worldMinY = minimum world Y coordinate
worldMinZ = minimum world Z coordinate

worldMaxX = maximum world X coordinate
worldMaxY = maximum world Y coordinate
worldMaxZ = maximum world Z coordinate

voxelWidth = (worldMaxX - worldMinX) / (volumeWidth - 1)
voxelHeight = (worldMaxY - worldMinY) / (volumeHeight - 1)
voxelDepth = (worldMaxZ - worldMinZ) / (volumeDepth - 1)

foreach point (i, j, k) on the grid G do
    x ← worldMinX + i * voxelWidth
    y ← worldMinY + j * voxelHeight
    z ← worldMinZ + k * voxelDepth
    volume(i, j, k) ← function(x, y, z)

```

**end**

Note that the data generation process for each class only differs in the *function* being used. Thus for each point on the lattice, we first map a point on the grid into the domain of the function (e.g., world  $(x, y, z)$  coordinates), and then compute the value of the function at that world coordinate.

### 6.1.2 Analytical Data Functions

Volumetric data for the first class is defined by point sampling analytical functions on a rectangular grid. Therefore, *function* is replaced by one of the analytical functions described in Section A.1 of Appendix A. For example, if

$$\text{function}(x, y, z) = (x - a)^2 + (y - b)^2 + (z - c)^2 - r^2$$

the volume represents data which describes spherical isosurfaces.

### 6.1.3 Resampled Data Functions

The last class involves resampling a volumetric dataset. Several methods exist for resampling volumetric data. Since our volumetric data is defined on rectangular grids, we chose trilinear interpolation of the values found at adjacent grid points. As shown in Figure 6.1, this can be accomplished by linear interpolation along each coordinate axis. We first linearly

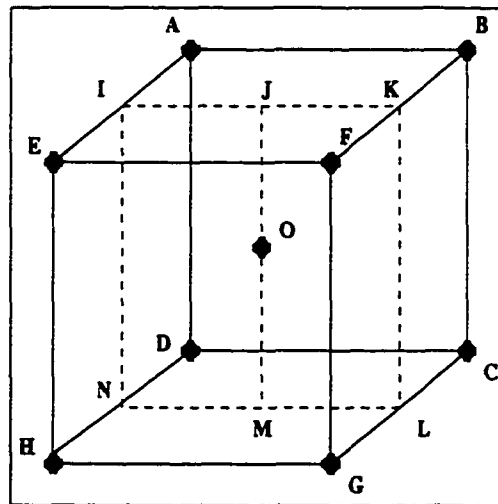


Figure 6.1: Trilinear Interpolation

interpolate along the Z axis to obtain values at I, K, L, and N using an equation having the

following form

$$V(a) = V(b) + \frac{Z(a) - Z(b)}{Z(c) - Z(b)} [V(c) - V(b)]$$

where  $V(p)$  is a function which returns the value at point  $p$ , and  $Z(p)$  is a function which returns the  $Z$  world coordinate of point  $p$ . Similar functions exist using  $X(p)$  and  $Y(p)$ . To calculate a value at  $I$ , we would set  $a, b$ , and  $c$  equal to  $I, A$ , and  $E$  respectively. Next we interpolate between  $I$  and  $K$  along the  $X$  axis to obtain a value at  $J$ , and between  $N$  and  $L$  to obtain a value at  $M$ . Finally, we interpolate linearly along the  $Y$  axis between  $J$  and  $M$  to obtain a value at  $O$ .

## 6.2 Analytical Data

As stated earlier, this section presents results based on a class of data sets where the underlying function is known, and becomes the reference function when computing metrics. We chose to present results for two analytical functions; the ones for which the isosurfaces are spheres and hyperboloids. These functions are describe in Appendix A.1 of this thesis. The domain of these functions are unbounded; we focused on the range  $(-1.5, -1.5, -1.5) \leq (x, y, z) \leq (1.5, 1.5, 1.5)$  during the sampling stage. To reduce the possible sources of error, no processing was performed in the second stage of the visualization pipeline.

The reconstruction process extracted isosurfaces given by the threshold  $\tau = 0.0$ . For the spherical data set, this threshold corresponds to a sphere with radius  $r$ , and for the hyperboloid data set, it represents a right circular cone having radius  $r$  at height  $h$ . Given the reconstructions, we analyze physical realism with respect to three of the metrics presented in Chapter 4; **gsap**, **pdp**, and **ivp**.

The density of the sampling grid was varied for the first three sets of results. Varying the density of the sampling grid varies the amount information about the underlying function which is available to the reconstruction process. Analysis of the resulting trends indicates an algorithm's ability to reconstruct isosurfaces with limited knowledge about the underlying function. Each plot is drawn with the average value drawn as a horizontal line with the appropriate symbol at the end points.

### 6.2.1 GSAP<sub>abs</sub>

Figure 6.2 and Figure 6.3 represent the  $gsap_{abs}$  metric applied to reconstructions <sup>1</sup> of a sphere and a cone. The first characteristic that is apparent from these figures is that, in

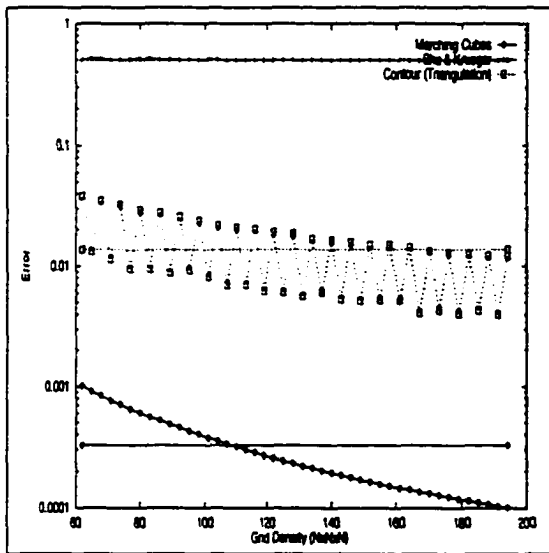


Figure 6.2: GSAP<sub>abs</sub> on Sphere Data

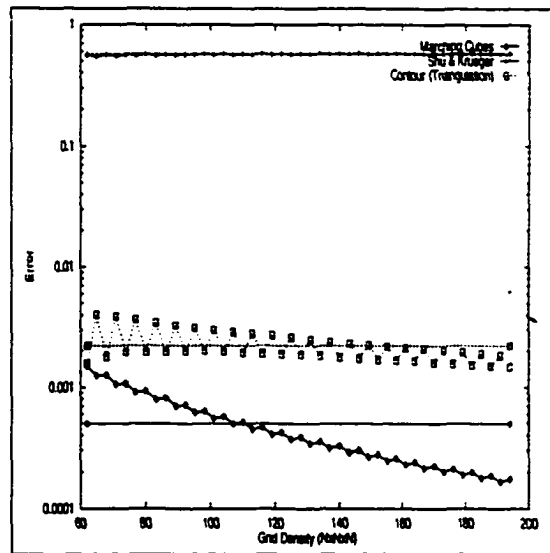


Figure 6.3: GSAP<sub>abs</sub> on Cone Data

general, as the density of the grid increases the algorithms perform better with respect to  $gsap_{abs}$ . In fact, we would expect this characteristic to persist no matter what property is

<sup>1</sup>Recall from Section 4.2.1 that  $gsap$  is not appropriately applied to spline surfaces. Therefore, we will not present any results for the spline lofting surface reconstruction algorithm in this section.

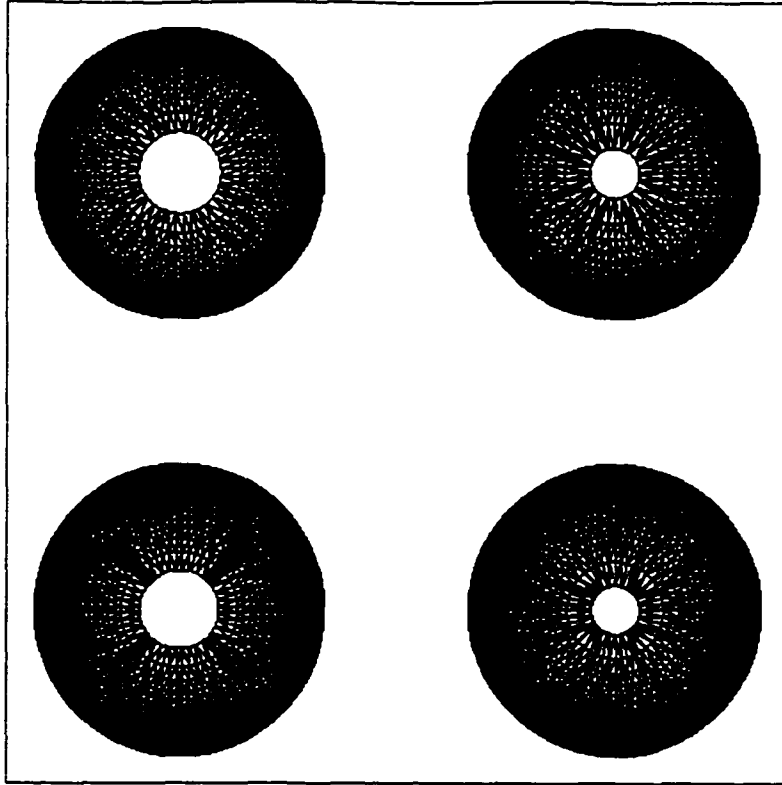


Figure 6.4: Contour Triangulations of the Sphere

being analyzed. As the sampling grid becomes more dense,  $F$  becomes a better approximation of  $\mathcal{F}$ . Since, reconstruction algorithms are essentially interpolation schemes, the better approximation to the underlying function provides the algorithms with more knowledge about that function, and, intuitively, a better reconstruction should be obtained.

We see, however, that the value of  $\text{gsap}_{ab_s}$  oscillates up and down for the contour triangulation algorithm. This illustrates the fact that intuition is not always correct. A denser sampling grid does not imply necessarily a better reconstruction of a particular instance (a particular value of  $\tau$ ) of a function. Figure 6.4 illustrates how this unusual phenomena can occur. This figure shows reconstructions of the sphere for grids  $62 \times 62 \times 62$ ,  $65 \times 65 \times 65$ ,  $68 \times 68 \times 68$ , and  $71 \times 71 \times 71$  (left to right, top to bottom). Each of these reconstructions are missing the “north and south poles” of the sphere. Recall, the contour triangulation algo-

rithm first generates contours slice by slice, and then interpolates between the contours it finds using triangular facets. The first and last contours occur on the slices immediately after and immediately before the voxel containing the poles. That is, the poles of the sphere do not occur on a data slice. Because inter-slice interpolation is not used, the poles are not discovered by the contour algorithms, and the open-ended reconstructions occur. The contour algorithms, as implemented, rely heavily on a proper sampling of the underlying function.

How does this effect **gsap** for these algorithms? We will consider the problem in two dimensions. The analogous metric in two dimensions would measure the relative difference between the circumferences. Figure 6.5 shows a circle being sampled with two different grid densities, and the associated reconstructions. Like the spheres shown in Figure 6.5, we see

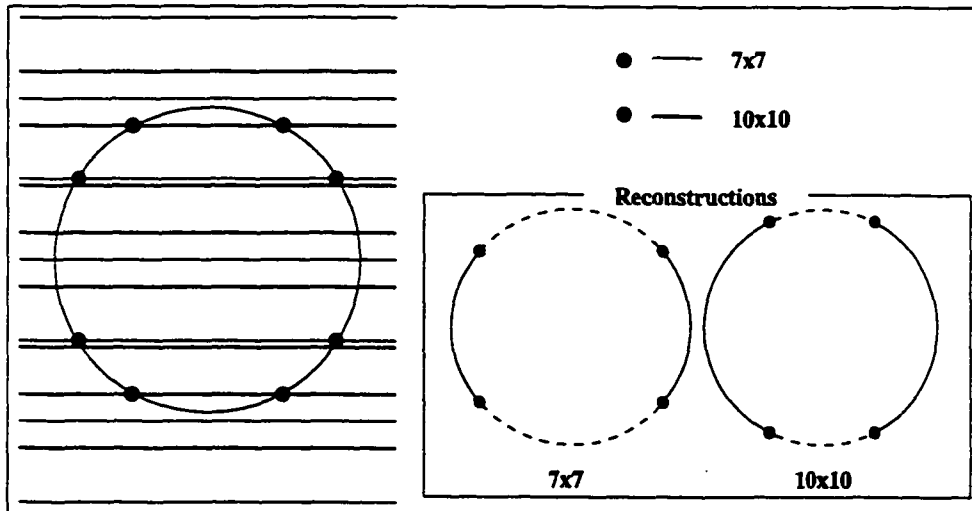


Figure 6.5: Circle Sampling

that the reconstructions are missing the top and bottom of the circle (e.g., the part of the circle shown as a dashed line). Assuming the two arcs composing the reconstruction are perfectly reconstructed, we see that the circumference of the reconstruction will be less than the circumference of the original circle by the sum of the lengths of the two dashed line



arcs. As we alter the sampling density, the size of these arcs will fluctuate, and therefore so will the amount of error with respect to the circumference.

### 6.2.2 PDP

In this section, we present **pdp** applied to the same two data sets used in the previous section. Again, we see that as the sampling grid becomes more dense, the algorithms perform better with respect to the metric being investigated. We also see that the following order

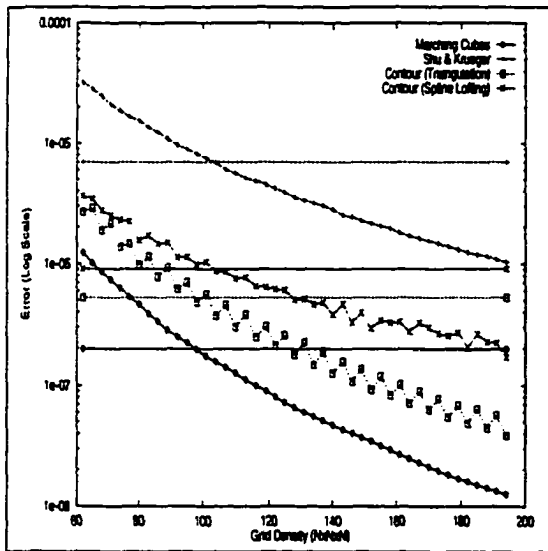


Figure 6.6: PDP on Sphere Data

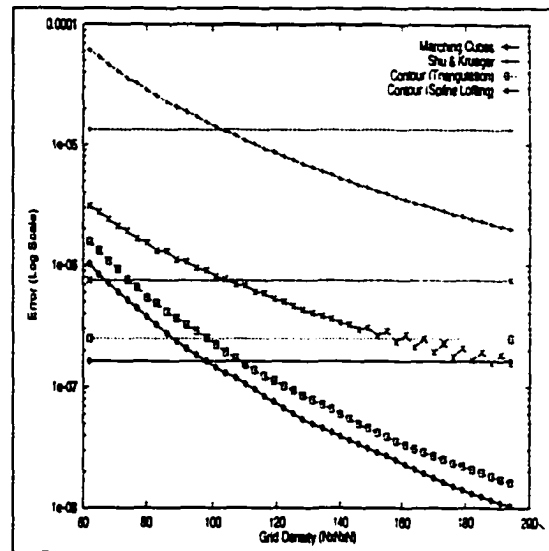


Figure 6.7: PDP on Cone Data

(with respect to decreasing performance) is maintained in both figures; marching cubes, contour triangulation, spline lofted contour, and surface tracking. The reason behind this ordering can be understood by considering the topology of the underlying representations. A reconstruction resulting from the surface tracking algorithm consists of rectangular faces which belong to voxels that the desired isosurface is believed to intersect. These faces can only have six configurations; one parallel to each positive and negative coordinate plane. In comparison, the facets for a reconstruction resulting from the marching cubes algorithm

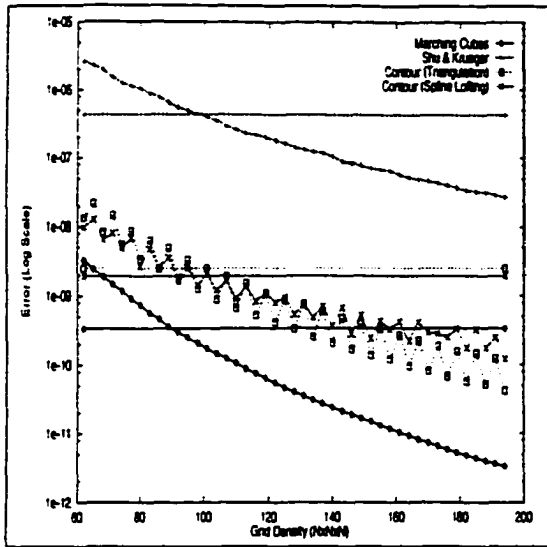


Figure 6.8: IVP on Sphere Data with Grid Varied

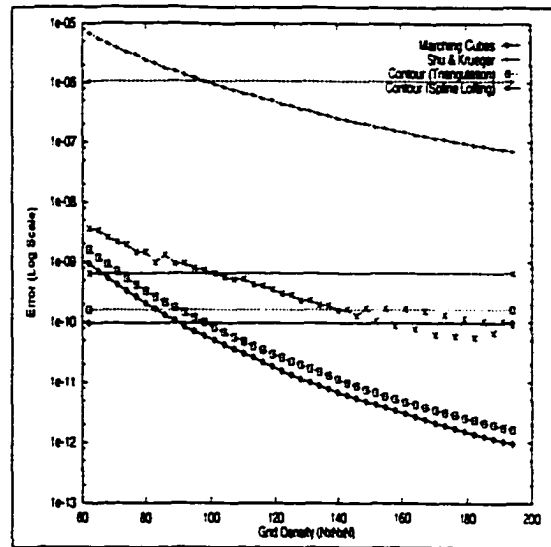


Figure 6.9: IVP on Cone Data with Grid Varied

can have 256 configurations. This added flexibility allows the marching cubes algorithm to construct a more accurate representation. The contour algorithms' performance is hindered by the fact that the reconstruction is initially defined by the planar contours on each data slice. Subvoxel accuracy is not considered.

### 6.2.3 IVP<sub>mse</sub> – Grid Variance

Recall from Section 4.2.4, that **pdp** realistically is not applicable when reference surfaces do not exist, or when the point distance function is unobtainable. The **ivp** metric is intended to provide a solution to that problem. As shown in Figure 6.8 and Figure 6.9 the results are very similar to those shown in Figure 6.6 and Figure 6.7. When considering analytical data sets, the **ivp** metric is essentially **pdp** cast to a new domain. The metric **pdp** measures the differences in the distances between corresponding points on two surfaces. The **ivp** metric, on the other hand, measures the differences in the distances between isovalues. The domain

of *pdp* is surfaces defined by a set of three dimensional points, and the domain of *ivp* is surfaces defined by an isovalue.

We see in Figure 6.6 and Figure 6.9, that the contour triangulation algorithm performs better than the spline lofted contour algorithm. This may seem counter-intuitive because in most cases spline surfaces are believed to better interpolate data than planar facets. Recall from Section 3.4, that the number of points defining a contour is first reduce to some

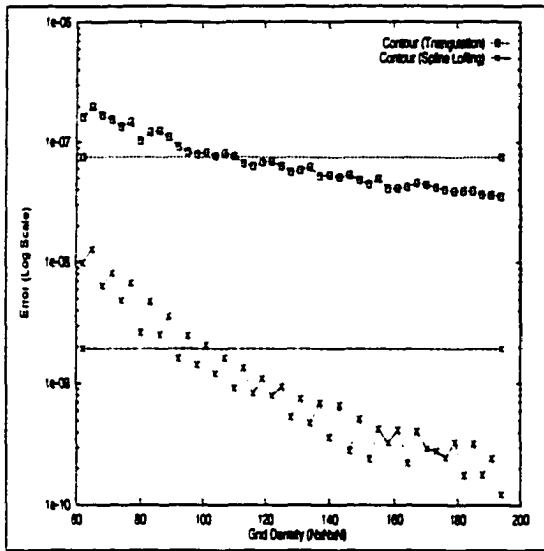


Figure 6.10: IVP on Sphere Data

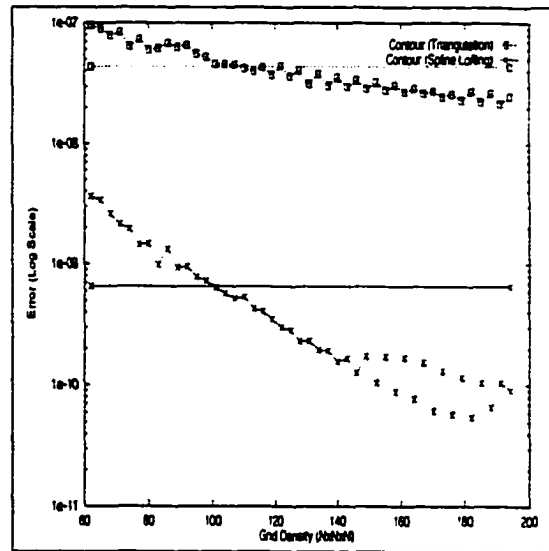


Figure 6.11: IVP on Cone Data

predetermined number  $K$  before being fitted with a B-spline curve. As shown in Figure 6.10 and Figure 6.11, if we also reduce the number of points in the contour for the triangulation algorithm it now performs worse than the spline method.

### 6.2.4 IVP<sub>mse</sub> - Threshold ( $\tau$ ) Variance

The previous sections presented results which gave insight into the performance of the reconstruction algorithms as the density of the sampling grid was varied. Another trend which might be of interest is the performance of the algorithms when the isovalue used

for the threshold  $\tau$  is varied. Analysis of the resulting trends will provide insight into

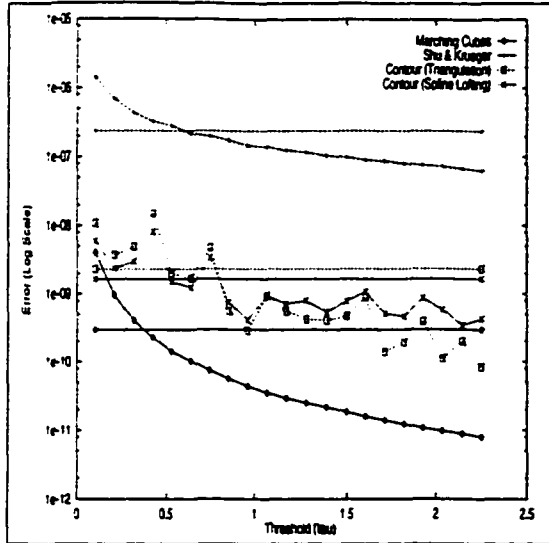


Figure 6.12: IVP on Sphere Data with  $\tau$  Varied

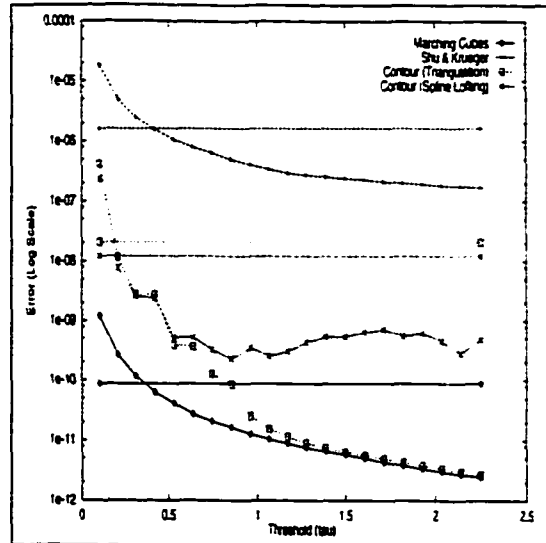


Figure 6.13: IVP on Hyperboloid Data with  $\tau$  Varied

an algorithm's ability to reconstruct the underlying function. For regular analytical data sets, such as those described in Section A.1 of Appendix A, this also yields insight into an algorithm's ability to reconstruct objects of varying sizes. These results are shown in Figure 6.12 and Figure 6.13.

In general, as the size of the objects increase, the reconstruction algorithms perform better. Small objects are hard to represent discretely. They require a denser sampling grid to represent their properties. For example, consider a  $64 \times 64 \times 64$  sampling of a  $3.0 \times 3.0 \times 3.0$  world. In the worst case, a sphere with radius approximately 0.04 units centered inside a voxel can be missed completely by the sampling. The same sphere centered on a grid point would be reconstructed incorrectly as a tetrahedron.

For smaller objects, we see that the spline lofted contour algorithm performed better than the triangulated contour algorithm. With smaller objects, limiting the number of

contour points is less detrimental to the performance of the spline lofted contour algorithm. This helps validate our previous statement in Section 6.2.3 about why the triangulated contour algorithm performed better than the spline lofted contour algorithm.

Unfortunately, what the figures above do not show is that the surface tracking algorithm only reconstructs one sheet of the hyperboloid of two sheets as shown in Figure 6.14. This example helps illustrate the fact, that one metric might not be able to yield insight into

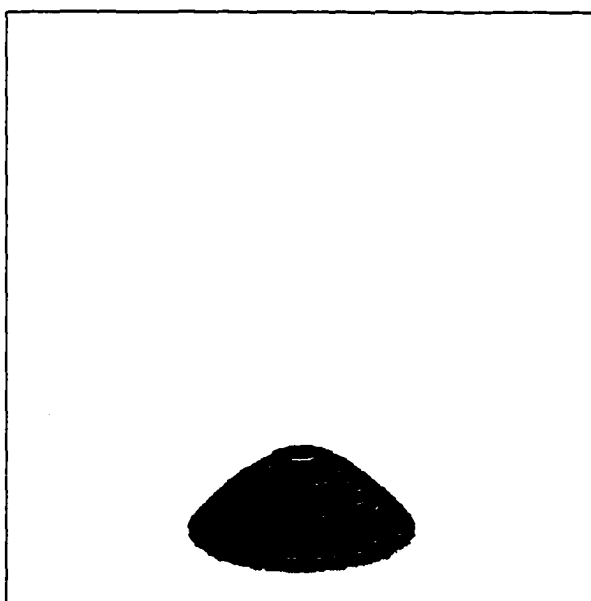


Figure 6.14: Surface Tracking Reconstruction of a Hyperboloid of Two Sheets

all problems of a reconstruction algorithm. Sometimes multiple metrics are required. The *gsap* metric, for example, would have shown that the surface tracking algorithm produced a reconstruction of the hyperboloid of two sheets whose surface area was always approximately half that of the ideal reconstruction. Note that this flaw could have been discovered also by having knowledge of the underlying data and viewing a reconstruction as shown in Figure 6.14.

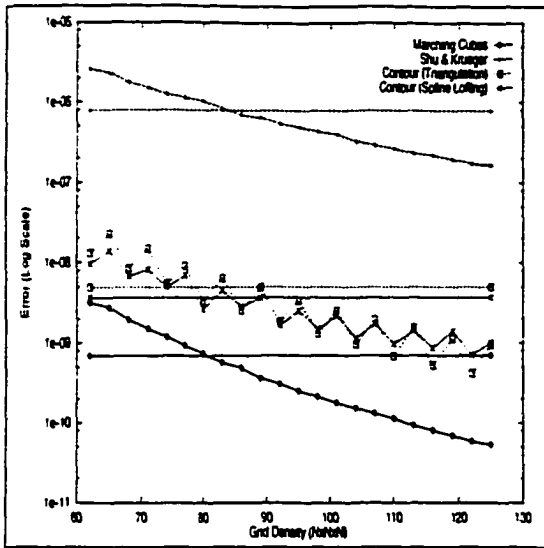


Figure 6.15: IVP on Volume Sphere Data (Grid Varied)

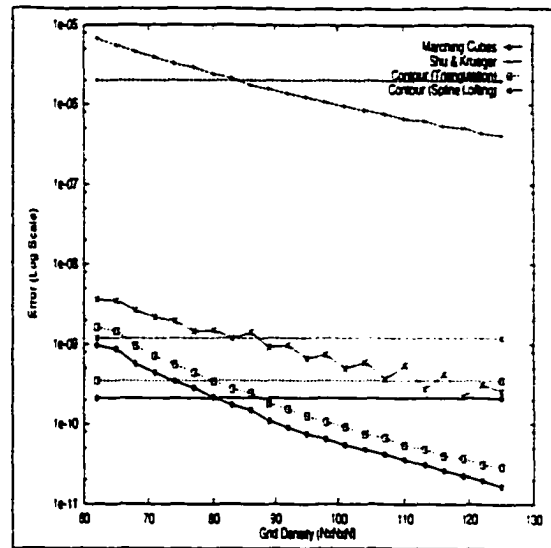


Figure 6.16: IVP on Volume Cone Data (Grid Varied)

### 6.3 Resampled Data

Section 6.2 presented results for data sets where the underlying functions were analytical. For these functions, all of their properties are known. From this, a question naturally arises. What if the underlying function is not known? If the underlying function is not known, then most of its properties will also be unknown. Since the metrics defined in Chapter 4 are based on physical properties, how can we analyze performance? This section attempts to help answer that question.

#### 6.3.1 Resampled Analytical Data

The first set of results, shown in Figure 6.15 and Figure 6.16, were generated by varying the resampling density of a  $128 \times 128 \times 128$  ( $128^3$ ) grid. The *ivp* metric was computed using the values in the original  $128^3$  volume. That is,  $\mathcal{F}(x, y, z)$  in Equation 4.7 is given by the original volume. The actual threshold value for each point on the surface of the

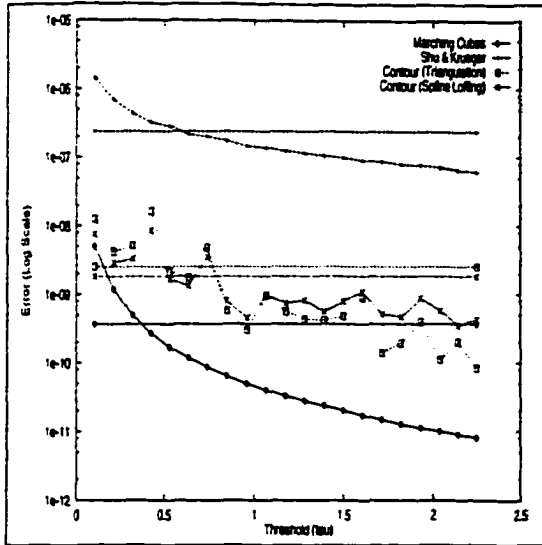


Figure 6.17: IVP on Volume Sphere Data ( $\tau$  Varied)

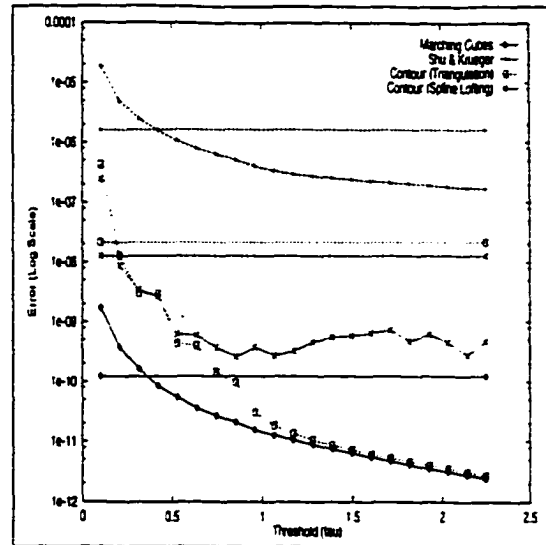


Figure 6.18: IVP on Volume Hyperboloid Data ( $\tau$  Varied)

reconstruction was computed using trilinear interpolation. If we compare these results with those shown in Figure 6.8 and Figure 6.9, we see that they are nearly identical.

In the next set of results, an initial  $256^3$  volume was generated from the analytical functions for the sphere and hyperboloid. These volumes were resampled on a  $128^3$  grid. Reconstructions were performed on the  $128^3$  with the threshold  $\tau$  being varied. The ivp metrics was computed by comparing the reconstructions with the original  $256^3$  volume. The results are shown in Figure 6.17 and Figure 6.18. Again, we see that the results are nearly identical to the corresponding analytic results shown in Figure 6.12 and Figure 6.13.

### 6.3.2 Resampled CFD Data

Up to this point, all of our results were generated from data originating from analytical functions. These data sets are well behaved. For example, they are symmetric through each coordinate plane, and they don't contain any irregularities such as bumps or holes.

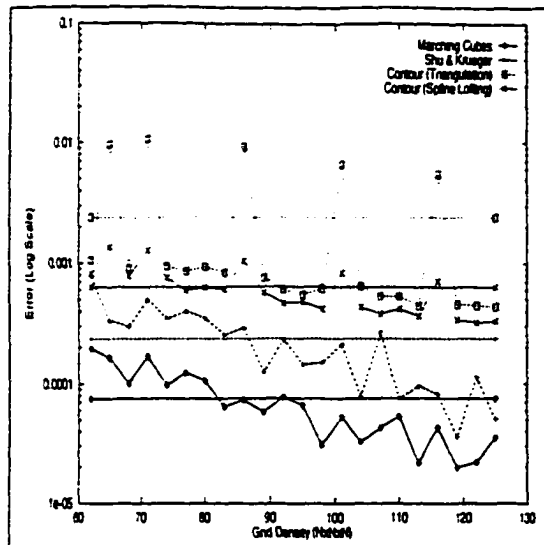


Figure 6.19: IVP on Ellipsoid CFD Data with Grid Varied

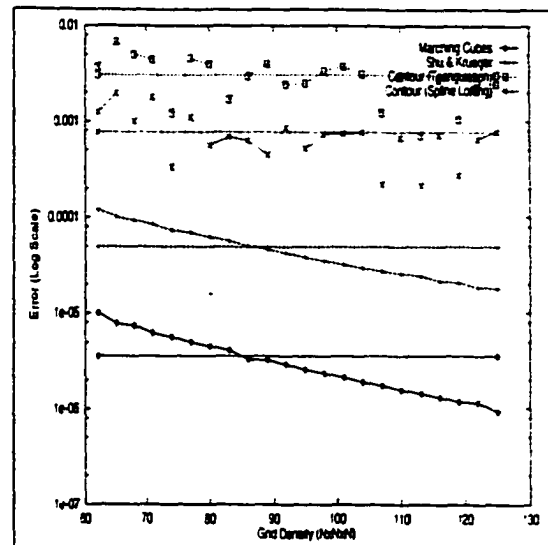


Figure 6.20: IVP on Ship CFD Data with Grid Varied

The next sets of resampled data results were generated using two data sets generated from the field of Computational Fluid Dynamics (see Appendix A.2). These data sets do not share the niceness of the analytical data sets. For example, an isocurve on a single slice of the analytical datasets is convex (e.g., a circle). This may not be the case in the CFD datasets. The CFD datasets may also contain holes or tunnels. For reasons such as these, the CFD datasets present a greater challenge to reconstruction algorithms. This is apparent in the results shown in Figure 6.19 and Figure 6.20.

The first characteristic that becomes apparent is that a majority of the plots oscillate more dramatically than those for analytical or resampled data. These oscillations are due to naive resampling of the original data sets. The grids were systematically changed without concern for retaining characteristics (e.g., bumps or holes) present in the data. When the necessary characteristics were present in the resampled data, the algorithms performed better, and vice versa.



Two other characteristics are evident in Figure 6.19 and Figure 6.20 when we compare them to results obtained from the analytical data sets. First, the contour lofting algorithm performs better than the contour triangulation algorithm. In the previous results, the contour triangulation algorithm out performed the contour lofting algorithm. The second characteristic is that for the first time, the surface tracking algorithm out performed both contour algorithms. Figure 6.21 will help illustrate why these characteristics can occur in irregular data sets such as the CFD data sets.

Figure 6.21 shows four reconstructions of an isosurface in the CFD ellipsoid data set. The upper left reconstruction was obtained using the marching cubes algorithm. The lower left was obtained using the surface tracking algorithm. The upper and lower right reconstructions were obtained using the contour lofting algorithm and the contour triangulation algorithm, respectively.

The CFD data sets have been quantized to the range  $0 \dots 255$ , and are stored in byte format. The analytical data sets, however, were computed and stored as floating point values. This quantization process makes the function's values discrete and the difference of the values between neighboring voxels larger. As seen in Figure 6.21, the contour triangulation algorithm is strongly affected by this characteristic, and produces a reconstruction with strong aliasing artifacts. An example of these artifacts is the circular pattern on the large front isosurface of the reconstruction in the lower left figure. Since the contour lofting algorithm uses splines as its underlying representation, it has the ability to more smoothly interpolate the isosurface, and better performance with respect to the *ivp* metric.

Surface Tracking performs better than the contour methods because they are unable to capture the concavity of the isosurface appropriately. In order to capture effectively a concave isosurface, a reconstruction algorithm must be able to detect multiple isocurves on

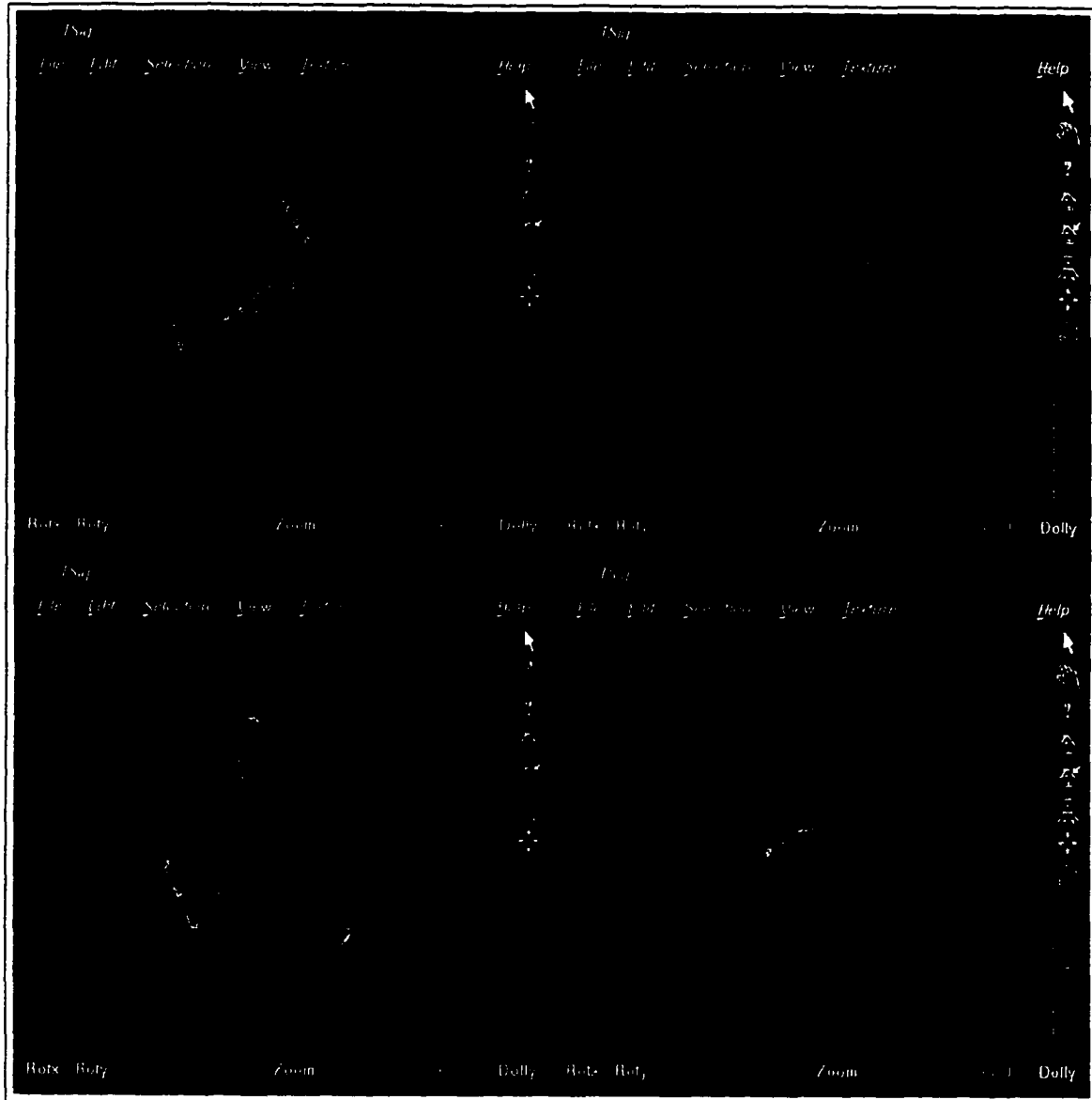


Figure 6.21: Reconstructions of Ellipsoid CFD Data

Table 6.1: Statistics for IVP on CFD Data with  $\tau$  Varied

		Marching Cubes	Surface Tracking	Contour Triangulation	Contour Lofting
Ellipsoid	Mean	$4.6632e^{-5}$	$1.1042e^{-3}$	$1.0904e^{-3}$	$2.0731e^{-3}$
	StdDev	$1.2462e^{-4}$	$1.8176e^{-3}$	$2.6346e^{-3}$	$6.1192e^{-3}$
Ship	Mean	$1.3093e^{-6}$	$2.0457e^{-5}$	$1.5782e^{-3}$	$1.8293e^{-3}$
	StdDev	$1.9784e^{-6}$	$2.7020e^{-5}$	$4.0878e^{-3}$	$5.7724e^{-3}$

a single slice of data. The contour algorithms we implemented are incapable of performing this task. This fact is illustrated in Figure 6.21 by the large flat bottomed isosurface in the right two figures. As seen in the left figures, both the marching cubes algorithm and the surface tracking algorithm were able to capture the concavity of the isosurface appropriately, and therefore perform better than the contour methods with respect to the *ivp* metric.

Our final experiment compared the reconstruction algorithms' ability to reconstruct isosurfaces within the CFD data sets with various values of  $\tau$ . Unlike the situation with analytical data sets, the reconstruction for one value of  $\tau$  is most likely unrelated to for another. For this reason, we only present the average error and standard deviation with respect to the *ivp* metric for each reconstruction algorithm.

As shown in Table 6.1, on average the ordering (e.g., in order of decreasing performance; marching cubes, contour triangulation, contour lofting, surface tracking) that we saw in previous figures is generally preserved. An exception to this is seen in Table 6.1 where the contour lofting algorithm performs worse than the surface tracking algorithm for

the ellipsoid data set. Recall that the contour lofting algorithm's performance can decline for large isosurfaces due to the contour point reduction step. This characteristic, along with the contour algorithms' inability to reconstruction concave isosurfaces, accounts for this discrepancy.

As shown in Table 6.1, the following order is maintained with respect to the standard deviation; marching cubes, surface tracking, contour triangulation, contour lofting. Though the contour algorithms, on average, performed better than the surface tracking algorithm, their standard deviations were worse. The standard deviation yields an indication of the stability of each reconstruction algorithm, and though the contour algorithms performed better on average, their averages are worse predictors of their performance. Given a particular reconstruction, a user can predict better the amount of error that may be present for the surface tracking algorithm than for the contour algorithms.

## Chapter 7

# Conclusion/Future Work

### 7.1 Review

This thesis presented metrics for analyzing the error in reconstructions of physical phenomena. Recall that physical phenomena are described by a number of physical properties, and metrics can be defined for each of these properties. Chapter 4 of this thesis presented four possible metrics of interest; surface area preservation (**gsap**), volume preservation **vp**, point distance preservation (**pdp**), and isovalue preservation (**ivp**), and Chapter 6 presented results based on three of these metrics.

These metrics illustrated some important attributes of reconstruction algorithms. Our results showed that the more information about the underlying function an algorithm had available to it, the better it was able to perform. This information can be provided in a number of ways. For example, one can simply provide a denser sampling of the underlying function. However, as we saw in Section 6.2.1, this is not necessarily a solution. One must also ensure a “good” sampling of the underlying function. Characteristics of the function must be accurately captured in the sampled volumetric data.

We also saw that algorithms (e.g., marching cubes) which utilize subvoxel accuracy require less knowledge of the underlying function. Algorithms (e.g., the contour algorithms) which did not consider subvoxel accuracy were unable to reconstruct features, such as the poles of the sphere, which occur between data slices.

This thesis also presented a framework for performing reconstruction error analysis. The methods of this framework manifest themselves in the RQS visualization (sub)system presented in Chapter 5, and its implementation as described in Appendix C. Each stage (sampling, processing, and reconstruction) in the visualization pipeline introduces error into the resulting reconstruction and the final rendering. The RQS (sub)system offers, to the developers and users of visualization systems, a structured arena for analyzing how these errors compound, cancel, and otherwise interact with each other. The analysis presented in this thesis only considered the error introduced by the reconstruction process. The other sources of error and how they interact provides a large source of topics for future research.

## 7.2 Guidelines

The analysis presented in this thesis provides a few guidelines to consider when designing a visualization system. First reconstruction algorithms can rely on an appropriate sampling of the underlying function. Care must be taken to guarantee that characteristics present in the underlying function are represented by the volumetric data. An example was seen in Section 6.2.1, where the triangulation contour algorithm was unable to reconstruct the poles of the sphere because they occurred between data slices. Fortunately, the sampling process is one part of the visualization pipeline over which we have a great deal of control.

A second problem was illustrated in Section 6.2.4. A single metric will most likely not show all problems of a reconstruction algorithm. For instance, the surface tracking

algorithm only reconstructed one sheet of the hyperboloid of two sheets. The *ivp* metric failed to catch this. However had surface area been considered, *gsap* would have shown the surface area of the reconstruction to be approximately half of the ideal surface's surface area.

The primary purpose of reconstructing volumetric data is to display, interact with, and interpret the phenomena contained there in. The display of these isosurfaces may show other hidden problems which applied metrics had not discovered. Visualizing an instance of the surface tracking reconstruction of the hyperboloid of two sheets showed immediately that only one sheet was being reconstructed. A user cannot view and blindly interpret the reconstructions presented to them. Some knowledge of the underlying functions is required.

### 7.3 Topics for Further Study

Physical realism is a many faceted notion, and it would be very short sighted to believe the four metrics defined in this thesis were all that were needed. More work can be done on defining and verifying other metrics. One can consider, for example, the notion of local curvature [15]. In the medical domain, one might also consider mass as the global property of interest. Metrics for properties, like mass, would require the use of the "generalized voxel model" [32]. This model allows other attributes (e.g., mass, membership to an organ, etc.) to be stored in a voxel.

As stated above, another topic for future research involves studying other sources of error (e.g., quantization) and how errors passed between stages are compounded or canceled. It is important to understand each source of error, and how it contributes to the final rendered image. These studies will yield a better understanding of the artifacts present in the rendered images, and allow the user to interact more properly with them.

Given a better understanding of the sources of error, new reconstruction algorithms can be developed which strive to reduce the amount error by minimizing one or more of the metrics defined to measure the error. Some work has been done by McAllister, et al. [55, 56, 57] which considered shape preserving spline interpolation of two dimensional data.

Finally, this thesis did not consider how well quantitative results correspond to qualitative judgments. In other words, future research can consider the question, "does physical realism imply photorealism?"



# Appendix A

## Volumetric Data Sets

### A.1 Analytical Functions

In this section, we describe the analytical functions which we used to generate our test data.

For each of the functions, the surface of the object is given by the threshold value  $\tau = 0.0$ .

**Object :** Sphere

**Description :** A sphere centered at  $(a, b, c)$  with radius  $r$ .

**Density Function**

$$\hat{\mathcal{F}}(x, y, z) = (x - a)^2 + (y - b)^2 + (z - c)^2 - r^2$$

**Surface Area**

$$4\pi r^2$$

**Distance Function**

As shown in Figure A.1, the distance any given point  $P$  is from the surface

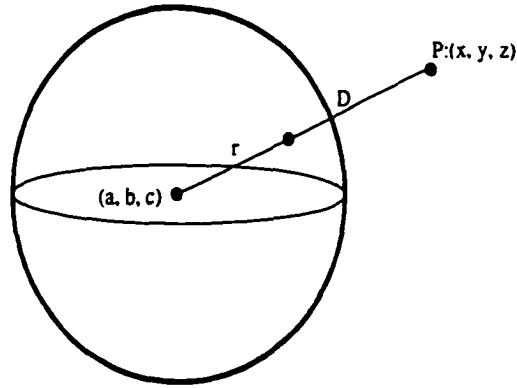


Figure A.1: Derivation of the Sphere Distance Function

of a sphere can be calculated as the difference between the distance from  $P$  to the sphere's center and the sphere's radius  $r$ .

$$D = \left| \sqrt{(x-a)^2 + (y-b)^2 + (z-c)^2} - r \right|$$

We take the absolute value of the difference because  $P$  might be "inside" the surface of the sphere yielding a negative distance.

**Object:** Right Circular Cylinder

**Description :** A cylinder with center at  $(a, b, 0)$  having radius  $r$ .

**Density Function**

$$\hat{\mathcal{F}}(x, y, z) = (x-a)^2 + (y-b)^2 - r^2$$

**Surface Area**

$$2\pi rh$$

**Distance Function**

As shown in Figure A.2, finding the distance between a point  $P$  and the

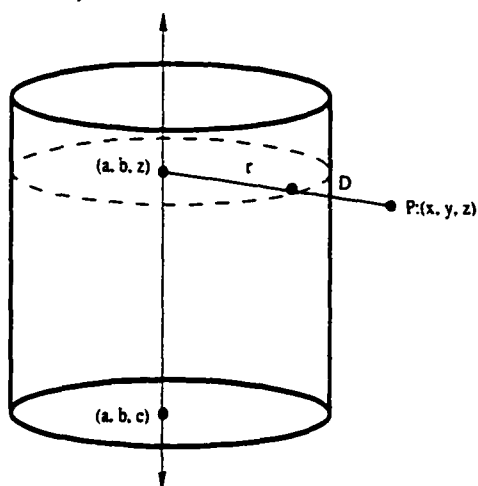


Figure A.2: Derivation of the Cylinder Distance Equation

isosurface of a cylinder can be simplified to finding distance between a 2D circle and the point  $(x, y)$ . The circle is defined by intersecting the plane  $Z = z$  with the cylinder. The center of this circle will be at  $(a, b)$ , and it will have radius  $r$ . Similar to the case of a sphere, the distance  $D$  is given by the difference between the distance from  $(x, y)$  to  $(a, b)$  and the circle's radius,  $r$ .

$$D = \left| \sqrt{(x-a)^2 + (y-b)^2} - r \right|$$

Again, the absolute value is used to constrain  $D$  to positive distances.

**Object :** Hyperboloid

**Description :** This data represents hyperboloids.

**Density Function**

$$\hat{\mathcal{F}}(x, y, z) = (x - a)^2 + (y - b)^2 - \left( \frac{r(z - c)}{h} \right)^2$$

The hyperboloids can be classified into three categories based on the value of  $\tau$

$$\left\{ \begin{array}{l} \tau < 0.0 \quad \text{Hyperboloid of two sheets} \\ \tau = 0.0 \quad \text{Right circular cone} \\ \tau > 0.0 \quad \text{Hyperboloid of one sheet} \end{array} \right.$$

The special case cone whose apex is at  $(a, b, c)$  will have radius  $r$  at  $c + h$  and  $c - h$ .

**Surface Area of Cone**

$$2\pi r \sqrt{r^2 + h^2}$$

**Distance Function for Cone**

The distance function for the analytical cone is more difficult than the previous two cases. We need to find the point on the surface of the cone closest to  $P$  to define the distance function. However, this is a difficult problem to solve using the current 3D configuration.

As shown in Figure A.3a, the center axis ( $Z = c$ ) of the cone, and the point  $P$  defines a plane, and we can define a new (two dimensional) coordinate system in this plane centered at  $(a, b, c)$  having the line  $Z = c$  as the ordinate axis and

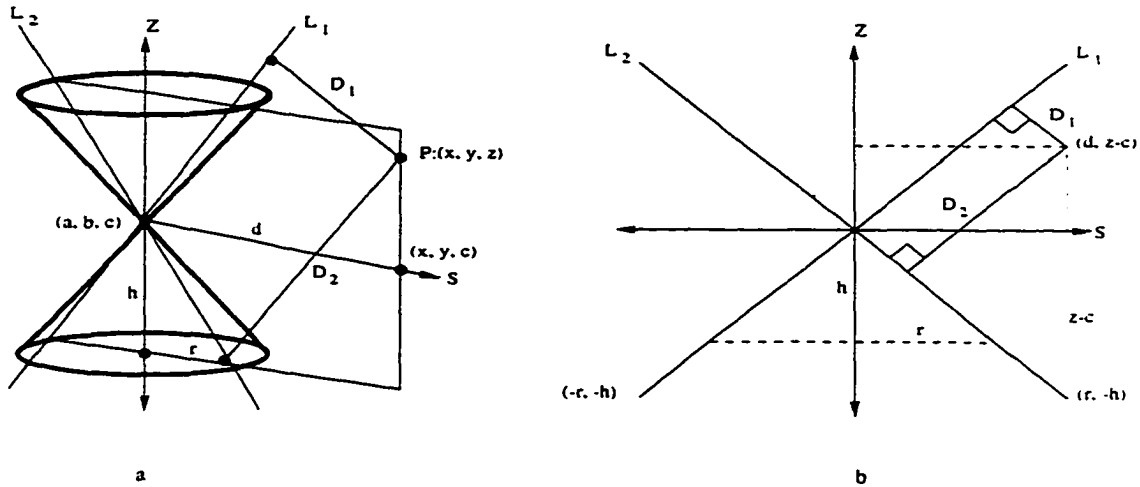


Figure A.3: Derivation of the Cone Distance Equation

the projection of  $P$  onto the plane  $Z = c$  as the abscissa.

Using this plane and coordinate system, we can redefine the 3D problem as a much simpler 2D problem. The trace of the cone in this plane consists of the two lines:

$$L_1 : ry - hx = 0$$

$$L_2 : ry + hx = 0$$

Projecting  $P$  onto the plane  $Z = c$ , we see that it is at a distance of  $z - c$  from the  $S$ -axis, and a distance of  $\sqrt{(x - a)^2 + (y - b)^2}$  from  $(a, b, c)$  (the center of the new coordinate system). Figure A.3b shows the new 2D configuration: where  $d = \sqrt{(x - a)^2 + (y - b)^2}$ .

The derivation of the cone distance function is now reduced to the simpler problem of finding the distance between a point  $(x_0, y_0)$  and a line  $(Ax + By + C =$

0); which is given by

$$D = \frac{|Ax_0 + By_0 + C|}{\sqrt{A^2 + B^2}}$$

Since, however, the cone becomes two lines in the two dimensional problem. we must find the distance to each line. Taking the minimum of the two distances yields the desired distance function.

$$\min\{D_1, D_2\} = \min \left\{ \frac{|-hd + r(z - c)|}{\sqrt{h^2 + r^2}}, \frac{|hd + r(z - c)|}{\sqrt{h^2 + r^2}} \right\}$$

## A.2 CFD Data

Section A.1 defined three analytic data sets. These data sets were used for initial testing of the metrics defined in this thesis. They truly represent fabricated data, and don't arise very often in the domain of scientific visualization. To illustrate the abilities and validity of our metrics in more realistic situations, we turned to data sets generated by Computational Fluid Dynamics (CFD).

Unfortunately, CFD data sets are most often generated on irregular grids. The reconstruction algorithms described in Chapter 3 require data defined on rectilinear grids. Therefore, it was necessary to resample these datasets. For simplicity, we used first degree Taylor polynomials for resampling.

**definition A.1** *The value of a function  $f(\cdot)$  at a point  $x$  close to a point  $c$  can be approximated by the following  $n^{\text{th}}$  degree polynomial*

$$f(x) \approx P_n(x) = f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 + \cdots + \frac{f^n(c)}{n!}(x - c)^n \quad (\text{A.1})$$

provided the first  $n$  derivatives exist and are continuous on some interval close to  $c$  and  $x$  is located in that interval.  $P_n(x)$  is called a **Taylor polynomial of degree  $n$** . Note. Equation A.1 can be generalized to any dimension.

CFD datasets are most often generated by sampling some function  $\mathcal{F}$  at a discrete subset of points located in the domain of  $\mathcal{F}$ . Users and developers of visualization systems typically do not have access to  $\mathcal{F}$ , but often require knowledge of  $\mathcal{F}$  at points other than those defined at the given grid points. Given a function  $F(x, y, z)$  defined on some arbitrary three dimensional grid with a known connectivity (e.g. an irregular grid), we can approximate  $F(x + r, y + s, z + t)$  with the first degree Taylor polynomial  $P_1$  given by

$$P_1(x + r, y + s, z + t) = F(x, y, z) + r \frac{\partial F}{\partial x}(x, y, z) + s \frac{\partial F}{\partial y}(x, y, z) + t \frac{\partial F}{\partial z}(x, y, z) \quad (\text{A.2})$$

where

$$\frac{\partial F}{\partial x} = \frac{\partial F}{\partial i} \frac{\partial i}{\partial x} + \frac{\partial F}{\partial j} \frac{\partial j}{\partial x} + \frac{\partial F}{\partial k} \frac{\partial k}{\partial x}$$

$$\frac{\partial F}{\partial i} \approx \frac{F[i + 1, j, k] - F[i - 1, j, k]}{2} \quad (\text{A.3})$$

$$\frac{\partial F}{\partial j} \approx \frac{F[i, j + 1, k] - F[i, j - 1, k]}{2} \quad (\text{A.4})$$

$$\frac{\partial F}{\partial k} \approx \frac{F[i, j, k + 1] - F[i, j, k - 1]}{2} \quad (\text{A.5})$$

and

$$\begin{bmatrix} \frac{\partial x}{\partial i} & \frac{\partial x}{\partial j} & \frac{\partial x}{\partial k} \\ \frac{\partial y}{\partial i} & \frac{\partial y}{\partial j} & \frac{\partial y}{\partial k} \\ \frac{\partial z}{\partial i} & \frac{\partial z}{\partial j} & \frac{\partial z}{\partial k} \end{bmatrix}^{-1} = \begin{bmatrix} \frac{\partial i}{\partial x} & \frac{\partial i}{\partial y} & \frac{\partial i}{\partial z} \\ \frac{\partial j}{\partial x} & \frac{\partial j}{\partial y} & \frac{\partial j}{\partial z} \\ \frac{\partial k}{\partial x} & \frac{\partial k}{\partial y} & \frac{\partial k}{\partial z} \end{bmatrix}$$

$\partial F/\partial y$  and  $\partial F/\partial z$  can be defined similarly.

Using Equation A.2 we can approximate the value of  $\mathcal{F}$  at any point in its domain. Given a point  $(a, b, c)$ , we first find the grid point  $(x, y, z)$  closest to  $(a, b, c)$  and compute  $r = a - x$ ,  $s = b - y$ , and  $t = c - z$ . Assuming  $(x, y, z)$  is located at grid point  $[i, j, k]$ , we compute the necessary partial derivatives using Equations A.3-A.5, and using similar equations for  $\partial x/\partial i$ ,  $\partial y/\partial i$ , etc. Finally, we approximate  $F(a, b, c)$  using Equation A.2.

### A.3 Volumetric Data File Format

Various volumetric data files (e.g. CFD data sets) were generated previous to reconstruction and stored in a local file format. This file format is entitled the Volumetric Data File (VDF) format. It was designed to be a simple format for the task at hand. Several other volumetric data formats exist (e.g. HDF), but these are far too general and complex for our needs. This section describes the VDF format.

A volumetric data set stored in the VDF format consists of two entities; a 128 byte header and the pixel/voxel data values. Both entities are written in binary form. As shown in Figure A.4, the first component of the header is the so-called “magic cookie.” For the file to be recognized as a VDF file the **magic** field must be the four byte hexadecimal number `0xFEEDBEEF`. Following the magic number, is an 80 character string. The **string** field can store any text, but typically stores some description of the data, creator of the data, when it was created, etc.

All data sets stored in the VDF format are assumed to have been generated on a rectilinear grid. The next three fields; **dimX**, **dimY**, and **dimZ**, give the dimensions of the data. **dimX** stores the number of pixels/voxels along the x-axis, and **dimY** stores the number of pixel/voxels along. If the data was generate as some number of planar slices (e.g.



as CT and MRI data sets are), then **dimZ** stores the number of slices. Otherwise, it stores the number of voxels along the z-axis.

The next six fields define the world extents of the rectilinear grid. The point defined by (**minX**, **minY**, **minZ**) is the back bottom left corner of the bounding box, and (**maxX**, **maxY**, **maxZ**) is the front top right corner. This defines a right handed coordinate system. The final two fields are extra padding. Currently, their use is undefined, and free to be used for whatever purpose necessary.

Directly following the header entity are the data values. They are stored with the X coordinate increasing fastest, followed by the Y coordinate. The Z coordinate increases the slowest. Each data value is assumed to only take one byte of storage space. Thus, the legal range of values is 0...255. Data compression techniques are not applied prior to storage.

<b>Magic Number</b> (int) 0xFEEDBEEF
<b>Description</b> (80 bytes)
<b>X Dimension</b> (int)
<b>Y Dimension</b> (int)
<b>Z Dimension</b> (int)
<b>Minimum World X Coordinate</b> (float)
<b>Minimum World Y Coordinate</b> (float)
<b>Minimum World Z Coordinate</b> (float)
<b>Maximum World X Coordinate</b> (float)
<b>Maximum World Y Coordinate</b> (float)
<b>Maximum World Z Coordinate</b> (float)
<b>Undefined</b> (int)
<b>Undefined</b> (int)

```

struct VDFHeader {
    int    magic;
    char  string[80];
    int    dimX;
    int    dimY;
    int    dimZ;
    float  minX;
    float  minY;
    float  minZ;
    float  maxX;
    float  maxY;
    float  maxZ;
    int    extra1;
    int    extra2;
};

```

Figure A.4: VDF Format File Header

## Appendix B

# Definite Integral Approximation

The surface integrals given in this paper cannot be evaluated easily using analytical methods. Therefore, an approximate solution must be obtained using numerical methods. This appendix discusses one such numerical method called Monte Carlo integration. Before this method is described, we first provide some background material [65].

### B.1 Background

A continuous *random variable* can take on a continuum of real values. The behavior of the random variable can be fully described by the distribution of the values it can take on as defined by a *probability density function (pdf)*. Given a random variable  $X$ , the pdf  $f(x)$  is given by the probability that  $X$  will take on some range of values. That is,

$$\int_a^b f(x) dx = \Pr\{a \leq X \leq b\}$$

The pdf has two defining characteristics. If  $x$  is a possible value of  $X$  then  $f(x) > 0$ , and

$$\int_{x \in \mathcal{X}} f(x) dx = 1$$

where  $\mathcal{X}$  is the domain of  $f$ .

**Example: Uniform(a,b)**

A random variable that is uniformly distributed between  $a$  and  $b$  has a pdf given by

$$f(x) = \frac{1}{b-a}$$

As illustrated in Figure B.1. The area under the curve is given by a rectangle with width

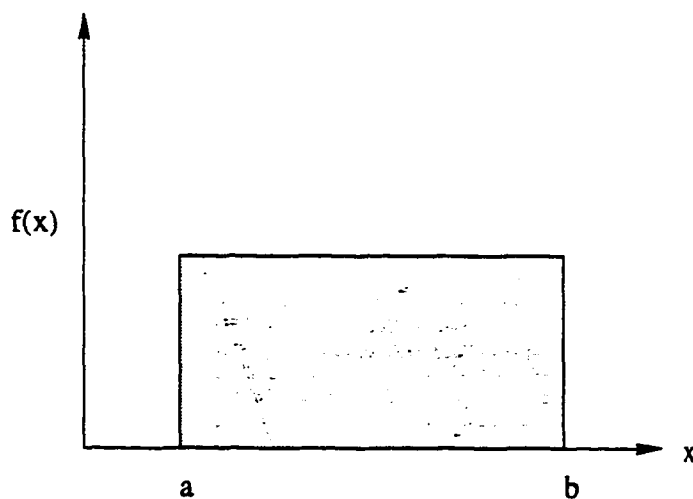


Figure B.1: The Uniform(a,b) pdf

$b-a$  and height  $1/(b-a)$ . This area is equal to  $\int_a^b 1/(b-a) dx$ .

The *average* (or *expected*) value of a continuous random variable is given by

$$\mu = E[X] = \int_{x \in \mathcal{X}} x f(x) dx$$

The expected value has the property that the expected value of a sum of random variables is equal to the sum of the expected values of the random variables ( $E[X+Y] = E[X] + E[Y]$ ). Since the sum of two random variables is itself a random variable this idea generalizes. The function of a random variable is also a random variable, and we can write its expected value as

$$E[g(x)] = \int_{x \in \mathcal{X}} g(x)f(x) dx$$

where  $g(x)$  is a function of the random variable  $X$ . The standard deviation  $\sigma$  of a random variable is given by

$$\sigma = \sqrt{\int_{x \in \mathcal{X}} (x - E[x])^2 f(x) dx},$$

and measures the overall deviation from the mean.

**Example: Uniform(a,b)**

The expected value of a Uniform( $a, b$ ) random variable is

$$\begin{aligned} \mu &= \int_a^b \frac{x}{b-a} dx \\ &= \frac{1}{b-a} \left( \frac{x^2}{2} \Big|_a^b \right) \\ &= \frac{1}{b-a} \left( \frac{b^2 - a^2}{2} \right) \\ &= \frac{a+b}{2} \end{aligned}$$

Likewise, the the standard deviation can be shown to be

$$\sigma = \sqrt{\int_x (x - \frac{a+b}{2})^2 \frac{1}{b-a} dx} = \dots = \frac{b-a}{\sqrt{12}}$$

## B.2 Monte Carlo Integration

With this background we will show how Monte Carlo integration can be used to estimate definite integrals. In the case of the surface integrals given in Section 4.1, we would want to estimate an integral of the form

$$I = \int_c^d \int_a^b h(u, v) \, du \, dv$$

Let  $U$  and  $V$  be independent random variables with possible values  $u$  and  $v$  whose values are uniformly distributed for  $a < u < b$  and  $c < v < d$  respectively, and define a new random variable  $X = h(U, V)$ . The expected value of  $X$  is

$$E[X] = \int_c^d \int_a^b h(u, v) f(u) g(v) \, du \, dv$$

where  $f(u)$  and  $g(v)$  are the probability density functions for the random variables  $U$  and  $V$ . Since  $U$  and  $V$  are uniformly distributed random variables  $f(u) = 1/(b - a)$  and  $g(v) = 1/(d - c)$  and

$$E[X] = \frac{1}{(b - a)(d - c)} \int_c^d \int_a^b h(u, v) \, du \, dv.$$

From this it follows that,  $I = (b - a)(d - c)E[X]$ .

$I$  can be estimated by generating multiple realizations of  $X$  and computing the sample mean. That is, first generate  $N$  samples of  $U$  and  $V$ ;  $u_1, u_2, \dots, u_n, v_1, v_2, \dots, v_n$ . Using these samples, compute  $N$  samples of  $X$  using

$$x_i = h(u_i, v_i) \quad i = 1, 2, \dots, N.$$

The sample mean for this realization of  $X$  is given by

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \approx E[X],$$

The rationalization behind this equation can be seen by generalizing the Law of Large Numbers. That is,

$$\Pr \left\{ \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N x_i = E[X] \right\} = 1$$

Finally the estimate  $\hat{I}$  is given by

$$I \approx \hat{I} = (b-a)(d-c)\bar{x} = \frac{(b-a)(d-c)}{N} \sum_{i=1}^N h(u_i, v_i).$$

### B.3 Interval Estimation

As just shown, the Monte Carlo method can be used as a technique for estimating the value of definite integrals. Since the result is only an estimate, it would be nice to know to what extent we can trust it. We can use sample means, standard deviations, and interval estimation to do so.

Consider the following scenario. Take some random number generator and generate  $M$  samples each of size  $N$ . For each sample  $i$  generate the sample mean  $m_i$  and sample standard deviation  $s_i$ . For each sample, define

$$t_i = \frac{m_i - \mu}{s_i / \sqrt{N-1}}; \quad i = 1, 2, \dots$$

where  $\mu$  is the theoretical (but unknown) mean of the distribution.

An extension of the Central Limit Theorem tells us the histogram of the “ $t$  data” will have the following properties

- the mean is approximately 0
- the standard deviation is approximately  $\sqrt{(N-1)/(N-3)}$
- the density is approximately the shape of the Student( $N-1$ ) pdf

Thus, given a random sample  $x_1, x_2, \dots, x_N$  with mean  $m$  and standard deviation  $s$ , it will be approximately true that

$$t = \frac{m - \mu}{s/\sqrt{N-1}}$$

is a random sample from the Student( $N-1$ ) distribution.

If  $T$  is a Student( $N-1$ ) random variable, then there exists some value  $t^* > 0$  such that  $\Pr\{-t^* \leq T \leq t^*\} = 1 - \alpha$  for some  $0 < \alpha < 1$ . Since  $t$  is a sample from this distribution

$$\Pr\left\{-t^* \leq \frac{m - \mu}{s/\sqrt{N-1}} \leq t^*\right\} = 1 - \alpha$$

By performing some simple algebraic techniques we arrive at

$$\Pr\left\{m - \frac{t^*s}{\sqrt{N-1}} \leq \mu \leq m + \frac{t^*s}{\sqrt{N-1}}\right\} = 1 - \alpha$$

If we choose  $\alpha = 0.05$ , then we are 95% confident that the theoretical mean lies somewhere between

$$m - \frac{t^*s}{\sqrt{N-1}} \quad \text{and} \quad m + \frac{t^*s}{\sqrt{N-1}}$$

$t^*$  is commonly called the critical value, and is given by the inverse distribution function



(idf) of the Student( $n-1$ ) distribution. That is, given  $\alpha$

$$t^* = \text{Student\_idf}(N - 1, 1 - \alpha/2)$$

## B.4 Implementation Issues

In this final section, we present some implementation issues with respect to Monte Carlo integration and interval analysis.

### Computation of the expected value and standard deviation

Computing the sample standard deviation using the expression

$$S = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - m)^2}$$

requires a two pass algorithm. The first pass computes the sample mean, and the second pass computes the sample standard deviation. An equivalent expression for the sample standard deviation which allows it and the sample mean to be computed in one pass is given by

$$S = \sqrt{\left( \frac{1}{N} \sum_{i=1}^N x_i^2 \right) - m^2}$$

The following C code computes the sample mean and standard deviation

```
/* readvalue() returns the next data value */

sum      = 0;
sumsq    = 0;

for (i = 0; i < n; i++) {
    x      = readvalue();
```

```

    sum    += x;
    sumsqr += (x * x);
}

mean    = sum / n;
stddev  = sqrt(sumsqr / n) - (mean * mean);

```

This code, when actually implemented, may have problems due to accumulated round off error. For a solution to this, we direct the reader to Welford's algorithm <sup>1</sup>.

### Critical values in interval estimation

In practice, when Monte Carlo integration is used  $N$  is always large. It is also common practice to choose  $\alpha = 0.05$ . When  $N$  is large the difference between  $N$  and  $N - 1$  is insignificant, and

$$\text{Student\_idf}(N - 1, 0.975) \simeq 2$$

With this in mind, given the sample mean  $m$  and standard deviation  $s$ , we are 95% confident that the actual mean  $\mu$  lies somewhere between

$$m - \frac{2s}{\sqrt{N}} \quad \text{and} \quad m + \frac{2s}{\sqrt{N}}$$

For the integral approximation discussion above, we equivalently define

$$\epsilon = (b - a)(d - c) \frac{2s}{\sqrt{N}}$$

and we are 95% confident that the actual value of the integral lies between  $\hat{I} - \epsilon$  and  $\hat{I} + \epsilon$ .

---

<sup>1</sup>Welford's algorithm can be found in *Technometrics*, Volume 4, Number 3, August 1962.

## Appendix C

# RQS Extension to Tcl/Tk

### C.1 Data Structures

**Data Structure:**

point - A world coordinate location

**Description:**

A **point** is defined as a standard Tcl list containing two to four floating point numbers. If only two numbers are given, the point will be considered to represent a two dimensional location. If three or four numbers are given, it is interpreted to be a three dimensional point (e.g.  $\{x\ y\ z\ [w]\}$ ) in homogeneous space. If  $w$  is not given, it defaults to 1.0.

**Data Structure:**

RGB color - A RGB color triple

**Description:**

A **RGB color** is defined as a standard Tcl list containing three eight bit numbers (e.g.  $\{r\ g\ b\ \}$ ). The expected range is  $0 \dots 255$ . If any value is not within this range, the value will be clamped. For example, if  $r < 0$  then  $r = 0$ .

**Data Structure:**

RGBA color - A RGB color triple with alpha component

**Description:**

A **RGBA color** is defined as a standard Tcl list containing three or four floating point numbers (e.g.  $\{r\ g\ b\ [a]\}$ ). Each number is expected to lie within  $0 \leq r, g, b, a \leq 1.0$ . Any values outside this range will be clamped. If  $a$  is not given, it defaults to 1.0.

**Data Structure:**

colormap - Stores a set of RGB triplets

**Synopsis:**

colormap *filename*

**Description:**

A colormap has been defined as a Tcl list with the following format and interpretation:

$$\{n \{red_0 \ green_0 \ blue_0\} \{red_1 \ green_1 \ blue_1\} \dots \{red_{n-1} \ green_{n-1} \ blue_{n-1}\}\}$$

The list begins with the number of entries in the colormap ( $n$ ), and is followed by  $n$  sublists. These sublists contain three eight bit values representing the red, green, and blue components of the color, respectively.

A colormap is the one new data structure that has a Tcl command. This command allows attempts to read *filename* as a  $n \times 1$  PPM image file. Thus each pixel in this "image" is described by three values. If it successfully reads the file, it returns its contents as a colormap.

Note: a colormap can be created manually by creating a Tcl list with the above format.

## C.2 Objects

Basically every entity in the RQS system is called an *object*. For example, below are the definitions of various entities from functions, to surfaces, to volumes. Each of these entities all belong to the generic class *object*. To reduce the number of new Tcl commands, some commands were created which operate on any of the objects defined below.

### Tcl Command:

`objDelete` - Delete an object from RQS

### Synopsis:

`objDelete` *objectName* [...]

### Description:

`objDelete` removes the specified objects from RQS space. These objects will no longer be accessible. If they were once Tcl commands themselves (e.g. that returned from creating a sphere function), then those commands will also no longer exist.

### Tcl Command:

`objName` - Returns the object's name (type)

### Synopsis:

`objName` *objectName*

### Description:

Returns the given object's name. This will actually be a string representing the type of object. For example, "sphere," "volume," and "mesh."

### Tcl Command:

`objIsFunction` - Decides if the object is a function

### Synopsis:

`objIsFunction` *objectName*

### Description:

`objIsFunction` is a decision command. It will return 1 if the given object is a function, and 0 otherwise. See Section C.2.1 for the available function objects.

**Tcl Command:**

`objIsVolume` - Decides if the object is a volume

**Synopsis:**

`objIsVolume` *objectName*

**Description:**

`objIsVolume` is a decision command. It will return 1 if the given object is a volume, and 0 otherwise. See Section C.2.2 for a description of the volume object.

**Tcl Command:**

`objIsSurface` - Decides if the object is a surface

**Synopsis:**

`objIsSurface` *objectName*

**Description:**

`objIsSurface` is a decision command. It will return 1 if the given object is a surface, and 0 otherwise. See Section C.2.3 for the available surface objects.

### C.2.1 Functions

**Tcl Command:**

`fsphere` - Create a spherical function object

**Synopsis:**

`fsphere` [*options*]

**Description:**

The `fsphere` command is used to define a function object that represents a sphere. The sphere function is defined with respect to an actual spherical object (e.g. its radius and center are specified). However, the function defines a data value for every point in the function's domain.

**Options:****-radius** *value*

Specify the radius of the sphere function. *value* can be any legal floating point number. Legal examples include 0.1, .1, 5, etc. Defaults to 1.0.

**-point** *point*

Specify the reference point for the sphere function. In this case the reference point represents the center of the sphere. Defaults to {0.0 0.0 0.0}

**-world** *point point*

Specify the domain of the sphere function. The two arguments to the `-world` option represent the minimum and maximum world coordinates, respectively. Defaults to {-1.0 -1.0 -1.0} and { 1.01.01.0}, respectively.

**Widget Command:**

The `fsphere` command returns a string of the form `sph#`; where '#' is a number which makes the created object unique with respect to other created sphere function objects. This string represents a new Tcl command with its own options. These commands with the options take the following syntax:

```
objectName option [arg arg ...]
```

**Widget Options:****name**

Returns the object's name. In this case, it should return "sphere."

**radius**

Returns the sphere function's radius.

**point**

Returns the sphere function's reference point (center).

**world**

Returns the sphere function's domain. That is, it returns its minimum and maximum coordinate values.

**min**

Returns the function's minimum value. In this case, the minimum value occurs at the reference point.

**max**

Returns the function's maximum value. In the case of the sphere function, the maximum value occurs at the point in its domain which is furthest from the reference point.

**range**

Return's the minimum and maximum value in a single call.

**area**

Returns the surface area of the object represented by the function (e.g. the sphere defined by the radius and center given when the object was created).

**value *point***

Returns the value of the function object at the given *point*.

**distance *point***

Returns the distance from the given *point* to the point on the surface of the function which is "deemed" closest.

**configure [*options*]**

The **configure** command can be used to redefine an already created object. *options* are the same as those that can be specified when the object was created. If no arguments are given the **configure** command will return a list of the functions defining characteristics in the following form:

{ { *option* value } ... }



**Tcl Command:**

`fcylinder` - Create a cylindrical function object

**Synopsis:**

`fcylinder` [*options*]

**Description:**

The `fcylinder` command is used to define a function object that represents a cylinder. The cylinder function is defined with respect to an actual cylindrical object (e.g. its radius and center are specified). However, the function defines a data value for every point in the function's domain.

**Options:****-radius** *value*

Specify the radius of the cylinder function. *value* can be any legal floating point number. Legal examples include 0.1, .1, 5, etc. Defaults to 1.0.

**-point** *point*

Specify the reference point for the cylinder function. In this case the reference point represents the center of the cylinder. Defaults to {0.0 0.0 0.0}. Note that the *z* value is ignored because there isn't any notion of cylinder height.

**-world** *point point*

Specify the domain of the cylinder function. The two arguments to the `-world` option represent the minimum and maximum world coordinates, respectively. Defaults to {-1.0 -1.0 -1.0} and {1.01.01.0}, respectively.

**Widget Command:**

The `fcylinder` command returns a string of the form `cyl#`; where '#' is a number which makes the created object unique with respect to other created cylinder function objects. This string represents a new Tcl command with its own options. These commands with the options take the following syntax:

```
objectName option [arg arg ...]
```

**Widget Options:****name**

Returns the object's name. In this case, it should return "cylinder."

**radius**

Returns the cylinder function's radius.

**point**

Returns the cylinder function's reference point (center).

**world**

Returns the cylinder function's domain. That is, it returns its minimum and maximum coordinate values.

**min**

Returns the function's minimum value. In this case, the minimum value occurs at any value along the axis defined by the  $x$  and  $y$  coordinates of the reference point that runs parallel to the  $z$  axis.

**max**

Returns the function's maximum value. In the case of the cylinder function, the maximum value occurs at the point in its domain which is furthest from the axis defined by the  $x$  and  $y$  coordinates of the reference point that runs parallel to the  $z$  axis.

**range**

Returns the minimum and maximum value in a single call.

**area**

Returns the surface area of the object represented by the function (e.g. the cylinder defined by the radius and center given when the object was created).

**value *point***

Returns the value of the function object at the given *point*.

**distance *point***

Returns the distance from the given *point* to the point on the surface of the function which is "deemed" closest.

**configure [*options*]**

The **configure** command can be used to redefine an already created object. *options* are the same as those that can be specified when the object was created. If no arguments are given the **configure** command will return a list of the functions defining characteristics in the following form:

{ { *option* value } ... }

**Tcl Command:**

**hyperboloid** - Create a hyperboloid function object

**Synopsis:**

**hyperboloid** [*options*]

**Description:**

The **hyperboloid** command is used to define a function object that represents a hyperboloid. The hyperboloid function is defined with respect to an actual hyperboloid object (e.g. its radius, center, and height are specified). However, the function defines a data value for every point in the function's domain.

**Options:****-radius** *value*

Specify the radius of the hyperboloid function. *value* can be any legal floating point number. Legal examples include 0.1, .1, 5, etc. Defaults to 1.0.

**-point** *point*

Specify the reference point for the hyperboloid function. In this case the reference point represents the center of the hyperboloid. Defaults to {0.0 0.0 0.0}

**-height** *value*

Specify the "height" of the hyperboloid function. The hyperboloid function that is actually created will have its specified radius at its reference point  $\pm$ *height*. *value* can be any legal floating point number. Legal examples include 0.1, .1, 5, etc. Defaults to 1.0.

**-world** *point point*

Specify the domain of the hyperboloid function. The two arguments to the **-world** option represent the minimum and maximum world coordinates, respectively. Defaults to {-1.0 -1.0 -1.0} and {1.01.01.0}, respectively.

**Widget Command:**

The **hyperboloid** command returns a string of the form *hyp#*; where '#' is a number which makes the created object unique with respect to other created hyperboloid function objects. This string represents a new Tcl command with its own options. These commands with the options take the following syntax:

objectName option [arg arg ...] .....

**Widget Options:****name**

Returns the object's name. In this case, it should return "hyperboloid."

**radius**

Returns the hyperboloid function's radius.

**point**

Returns the hyperboloid function's reference point (center).

**height**

Returns the hyperboloid function's height.

**world**

Returns the hyperboloid function's domain. That is, it returns its minimum and maximum coordinate values.

**min**

Returns the function's minimum value.

**max**

Returns the function's maximum value.

**range**

Returns the minimum and maximum value in a single call.

**area**

Returns the surface area of the special case cone represented by the function (e.g. the cone defined by the radius, height, and center given when the object was created).

**value *point***

Returns the value of the function object at the given *point*.

**distance *point***

Returns the distance from the given *point* to the point on the surface of the function which is "deemed" closest.

**configure [*options*]**

The **configure** command can be used to redefine an already created object. *options* are the same as those that can be specified when the object was created. If no arguments are given the **configure** command will return a list of the functions defining characteristics in the following form:

{ { *option* value } ... }

### C.2.2 Volumes

#### Tcl Command:

`volume` - Create a volumetric data set

#### Synopsis:

`volume type width height depth [options]`

#### Description:

The `volume` command is used to create a volumetric data storage structure. A volume is a three dimension array of values, and is defined by a rectilinear grid. The grid is defined by specifying the *type* of data to be stored in the volume, and the number of voxels along the x, y, and z axis. Currently two types of data are supported "byte" and "float." Byte data values are unsigned integers in the range 0 . . . 255. Float values are floating point numbers whose range depends on the particular architecture RQS has been compiled for.

The options below also allow the user to read in previously defined volumetric data from raw and VDF files.

#### Options:

##### `-bad value`

Specify the "bad" (or background) value. Whenever, a request is made for a non-existent voxel's value, this value is returned.

##### `-world point point`

The volumetric data set is defined on a rectilinear grid. This option allows the user to specify the world coordinates for the corners of the grid. The two arguments represent the minimum and maximum coordinates, respectively.

##### `-raw filename`

Read in a volumetric data set from a raw formatted file. In this case, the file is assumed to contain *width* × *height* × *depth* data values.

##### `-vdf filename`

Read in a volumetric data set from a VDF formatted file. The current dimensions of the volume are ignored. If necessary, the volume will be reallocated based on the dimensions given in the file. See Section A.3 for a complete description of the VDF format.

##### `-function function-name`

After allocating the volume, it will generate a data set based on the previously defined RQS function *function-name*.

#### Widget Command:

The **volume** command returns a string of the form *vol#*; where '#' is a number which makes the created object unique with respect to other created volume objects. This string represents a new Tcl command with its own options. These commands with the options take the following syntax:

objectName option [arg arg ...]

**Widget Options:**

**name**

Returns the object's name. In this case, it should return "volume."

**width**

Returns the number of voxels along the X-axis.

**height**

Returns the number of voxels along the Y-axis.

**depth**

Returns the number of voxels along the Z-axis.

**voxelWidth**

Returns the number of width of a voxel. This value will be equal to

$$\frac{world_{max_x} - world_{min_x}}{width - 1}$$

**voxelHeight**

Returns the number of height of a voxel. This value will be equal to

$$\frac{world_{max_y} - world_{min_y}}{height - 1}$$

**voxelDepth**

Returns the number of depth of a voxel. This value will be equal to

$$\frac{world_{max_z} - world_{min_z}}{depth - 1}$$

**bad [value]**

If *value* is specified, this option sets and returns the default value returned when an voxel value request is out of bounds. If *value* is not specified then the its current value is returned.

**world [point point]**

If the two points are given, this command sets the domain of the volume object, and returns it. If they are not given the current domain of the object is returned.

**clear**

Sets every voxel in the volume object to be equal to the "bad" value.

**value *point***

Returns the value of the volume at the given *point*. If *point* does not occur at a voxel corner, trilinear interpolation is used to determine the value.

**min**

Returns the minimum value occurring in the volume.

**max**

Returns the maximum value occurring in the volume.

**range**

Returns the minimum and maximum values occurring in the volume.

**create *width height depth***

(Re)create the volume object with the given dimensions. If the volume object was previously created, that object will be destroyed before the new object is created.

**destroy**

Destroy the volume object. This merely releases the memory consumed by the object. It can later be recreated using the above **create** command.

### C.2.3 Surfaces

#### Tcl Command:

`stype` - Obtain surface type

#### Synopsis:

`stype` *surfaceName*

#### Description:

`stype` is a general surface command which returns the type of surface for the given a surface name. This command will return a string depicting the type of surface. If *surfaceName* is not a surface, `stype` will return the string

*surfaceName* is not a surface

If *surfaceName* is a surface, `stype` will return either "mesh," "spline," or "surface."

#### Tcl Command:

`slow` - Get/Set low threshold value used to define isosurface

#### Synopsis:

`slow` *surfaceName* [*options*]

#### Description:

If *surfaceName* was defined via a reconstruction algorithm (e.g. Marching Cubes), it will have a field holding the low value of the threshold range. `slow` allows the user to obtain this value. `slow` takes an optional floating point number as the second argument. If this argument is given, the low threshold field will be set to the given value.

#### Tcl Command:

`shigh` - Get/Set high threshold value used to define isosurface

#### Synopsis:

`shigh` *surfaceName* [*options*]

#### Description:

If *surfaceName* was defined via a reconstruction algorithm (e.g. Marching Cubes), it will have a field holding the high value of the threshold range. `shigh` allows the user to obtain this value. `shigh` takes an optional floating point number as the second argument. If this argument is given, the high threshold field will be set to the given value.



**Tcl Command:**

`sthreshold` - Get isosurface threshold range

**Synopsis:**

`sthreshold` `surfaceName`

**Description:**

`sthreshold` allows the user to obtain both ends of the threshold range defining a reconstructed isosurface. It returns a list of the form

`{ low high }`

where the two fields are the low threshold and the high threshold respectively.

**Mesh Commands****Tcl Command:**

`mstats` - Gather statistics about a polygonal mesh surface

**Synopsis:**

`mstats` `meshName`

**Description:**

The `mstats` command returns statistics about the polygonal mesh. It returns a list with sublists having the following format

`{ { vertices numVertices } { polygons numPolygons }  
 { extents { minX minY minZ } { maxX maxY maxZ } } }`

Each sublist contains two fields; a string describing the statistic and its corresponding value. Three statistics are provided currently. The first two are the number of vertices and the number of polygons in the mesh. The last statistic represents the world extents ( or bounding box ) of the surface.

**Spline Commands****Tcl Command:**

**spline** - Operate on a spline

**Synopsis:**

**spline** splineName command

**Description:**

**spline** provides the interface to commands which operate on spline surfaces. Currently, the command provided is *npatches*. This command returns the number of control patches defining the spline surface.

**Tcl Command:**

**splmesh** - Generate a polygonal mesh for spline surface

**Synopsis:**

**splmesh** splineName patch# [*options*]

**Description:**

**splmesh** approximates a spline surface patch with a polygonal mesh. The *options* described below allow the user to control the precision with which the polygonal mesh is generated. It will return the name of the new mesh surface.

**Options:**

**-ucurves** *value*

**-ucurves** specifies the number of steps along the parametric *u* axis. This option controls the number of "horizontal" patches defining the resulting mesh.

**-vcurves** *value*

**-vcurves** specifies the number of steps along the parametric *v* axis. This option controls the number of "vertical" patches in the resulting mesh.

**Tcl Command:**

**splcontrol** - Generate a mesh for a spline control patch

**Synopsis:**

**splcontrol** splineName patch#

**Description:**

**splcontrol** creates a polygonal mesh surface object of the requested control polygon patch. **splcontrol** returns the name of the new polygonal mesh surface.

### C.3 Reconstruction Algorithms

**Tcl Command:**

`march` - Performs the Marching Cubes algorithm

**Synopsis:**

`march volumeName [options]`

**Description:**

The `march` command performs the Marching Cubes algorithm as described in Chapter 3 on the previously created volume object given by *volumeName*. By default, the isosurface corresponding to the range  $0 \leq \tau \leq 255$  is reconstructed. This range can be altered using the options described below. This command creates a new mesh surface object, and returns its name.

**Options:**

`-low value`

Specifies the low value of the range for  $\tau$ .

`-high value`

Specifies the high value of the range for  $\tau$ .

`-status`

Adds verbose output showing the changing states of the Marching Cubes algorithm.

**Tcl Command:**

`track` - Performs the surface tracking algorithm

**Synopsis:**

`track volumeName [options]`

**Description:**

The `track` command performs Shu and Krueger's surface tracking algorithm as described in Chapter 3. Again, *volumeName* is the name of a previously created volume object. By default, the isosurface corresponding to the range  $0 \leq \tau \leq 255$  is reconstructed. The range can be altered using the options described below. This command creates a new mesh surface object, and returns its name.

**Options:**

**-low value**

Specifies the low value of the range for  $\tau$ .

**-high value**

Specifies the high value of the range for  $\tau$ .

**-normals**

This option tells the surface tracking algorithm to generate surface normals for the vertices of each surface patch. The normals are generated using the grey level gradients from the volume object.

**Tcl Command:**

`tricon` - Performs the contour triangulation algorithm

**Synopsis:**

`tricon volumeName [options]`

**Description:**

The `tricon` command performs the contour triangulation algorithm as described in Chapter 3. Again, *volumeName* is the name of a previously created volume object. By default, the isosurface corresponding to the range  $0 \leq \tau \leq 255$  is reconstructed. The range can be altered using the options described below. This command creates a new mesh surface object, and returns its name.

**Options:**

`-low value`

Specifies the low value of the range for  $\tau$ .

`-high value`

Specifies the high value of the range for  $\tau$ .

`-npts value`

*value* states the number of points used to describe the contours on each data slice. If *value* is set equal to zero, the number of points in the contours are not altered. This is the default. If *value* > 0, then each contour is altered to consist *value* points uniformly distributed along the original contour.

`-status`

Adds verbose output showing the changing states of the contour triangulation algorithm.

**Tcl Command:**

`splcon` - Performs the contour spline lofting algorithm

**Synopsis:**

`splcon volumeName [options]`

**Description:**

The `splcon` command performs the contour spline lofting algorithm as described in Chapter 3. Again, *volumeName* is the name of a previously created volume object. By default, the isosurface corresponding to the range  $0 \leq \tau \leq 255$  is reconstructed. The range can be altered using the options described below. This command creates a new spline surface object, and returns its name.

**Options:**

`-low value`

Specifies the low value of the range for  $\tau$ .

`-high value`

Specifies the high value of the range for  $\tau$ .

`-npts value`

*value* states the number of points used to describe the contours on each data slice. If *value* is set equal to zero, the number of points in the contours are not altered. If *value* > 0, then each contour is altered to consist *value* points uniformly distributed along the original contour. By default, *value* is equal to twenty-five.

`-status`

Adds verbose output showing the changing states of the contour triangulation algorithm.

**Tcl Command:**

**rawcon** - Generates the contours on each data slice

**Synopsis:**

**rawcon** *volumeName* [*options*]

**Description:**

The **rawcon** command generates a mesh object consisting of the contours present on each data slice. Again, *volumeName* is the name of a previously created volume object. By default, the isosurface corresponding to the range  $0 \leq \tau \leq 255$  is reconstructed. The range can be altered using the options described below. This command does not create a surface. That is, the contours are not connected in any manner.

**Options:**

**-low** *value*

Specifies the low value of the range for  $\tau$ .

**-high** *value*

Specifies the high value of the range for  $\tau$ .

**-npts** *value*

*value* states the number of points used to describe the contours on each data slice. If *value* is set equal to zero, the number of points in the contours are not altered. This is the default. If *value* > 0, then each contour is altered to consist *value* points uniformly distributed along the original contour.

**-status**

Adds verbose output showing the changing states of the contour generation algorithm.



## C.4 Metrics

### Tcl Command:

GSAP - Compute global surface preservation

### Synopsis:

**gsap** *functionName* *surfaceName* [*options*]

### Description:

The **gsap** command computes the global surface area metric defined in Chapter 4. The result of this command is a string of the form

{ **gsap** *value* }

where *value* is the value of **gsap** for the given function (*functionName*) and surface reconstruction (*surfaceName*).

### Options:

**-relative**

States that **gsap** should be computed relative to the ideal surface area (see Equation 4.3 and Equation 4.2).

**Tcl Command:**

PDP - Compute point distance preservation

**Synopsis:**

*pdp functionName surfaceName [options]*

**Description:**

The *pdp* command computes the point distance preservation metric defined in Chapter 4. The result of this command is a string of the form

$$\{ \text{pdp } \{ \textit{nsamples seed} \} \{ \textit{value error} \} \}$$

where

*nsamples* - number of samples used to approximate Equation 4.6.

*seed* - seed presented to the random number generator

*value* - result of *pdp*

*error* - error interval for 95% confidence. Thus, the user can be 95% confident that the actual value of the integral is in the range  $\textit{value} \pm \textit{error}$ .

**Options:**

*-nsamples value*

*value* specifies the number of samples used to approximate the integral given in Equation 4.6. By default, this is equal to 125,000.

*-seed value*

*value* specifies the initial seed for the random number generator.

**Tcl Command:**

PDP Image - Compute point distance preservation

**Synopsis:**

*pdpImage functionName surfaceName*

**Description:**

The *pdpImage* command computes the point distance preservation metric defined in Chapter 4, and returns an eight-bit error image.

**Tcl Command:**

IVP - Compute isovalue preservation

**Synopsis:**

*ivp functionName surfaceName [options]*

**Description:**

The *ivp* command computes the isovalue preservation metric defined in Chapter 4. The result of this command is a string of the form

$$\{ \textit{ivp} \{ \textit{nsamples seed} \} \{ \textit{low high} \} \{ \textit{value error} \} \}$$

where

*nsamples* - number of samples used to approximate Equation 4.7.

*seed* - seed presented to the random number generator

*low, high* - the threshold range on which *value* is based

*value* - result of *ivp*

*error* - error interval for 95% confidence. Thus, the user can be 95% confident that the actual value of the integral is in the range  $\textit{value} \pm \textit{error}$ .

**Options:**

**-nsamples *value***

*value* specifies the number of samples used to approximate the integral given in Equation 4.7. By default, this is equal to 125,000.

**-seed *value***

*value* specifies the initial seed for the random number generator.

**-low *value***

This option alters the value of the low end of the threshold range. This option defaults to the low value used during the reconstruction phase.

**-high *value***

This option alters the value of the high end of the threshold range. This option defaults to the high value used during the reconstruction phase.

**Tcl Command:**

IVP Image - Compute isovalue preservation

**Synopsis:**

*ivpImage functionName surfaceName [options]*

**Description:**

The **ivpImage** command computes the isovalue preservation metric defined in Chapter 4, and returns the result as an eight-bit error image.

**Options:**

**-low** *value*

This option alters the value of the low end of the threshold range. This option defaults to the low value used during the reconstruction phase.

**-high** *value*

This option alters the value of the high end of the threshold range. This option defaults to the high value used during the reconstruction phase.

## C.5 Rendering

The rendering module consists of two parts. The first is a new Tk widget capable of receiving OpenGL rendering commands. The second are the commands which operate on those widgets. The Tk widget does not encompass all abilities supported by the OpenGL graphics programming library, and not all OpenGL functions have been implemented. Below is a description of the Tk OpenGL raster widget, and descriptions of the Tcl implementations of various OpenGL functions.

### C.5.1 OpenGL Raster Widget

**Tcl Command:**

`glRaster` - Tk OpenGL widget

**Synopsis:**

`glRaster` *pathName* [*options*]

**Description:**

`glRaster` is an implementation of a raster widget that supports OpenGL rendering. This widget supports the rgba and eight bit color models, depth buffer, doublebuffering, a single overlay, and a single underlay. It, however, is not a complete implementation of an OpenGL rendering window.

**Options:**

**-height** *value*

This option sets the height (in pixels) of the OpenGL raster window.

**-width** *value*

This option sets the width (in pixels) of the OpenGL raster window.

**-cursor** *cursorName*

This option allows the cursor to be changed for the window. *cursorName* is any legal X11 cursor name. See the Tk options manpage for more information on setting the cursor option.

**-rgbmode**, **-rgb** *value*

This is a boolean option. If *value* is true (e.g. 1), the shell will attempt to create a widget capable of 24 bit color rendering. This option will cause the widget creation to fail if the 24 bit color model is not supported by the host machine. If this option is false, a eight bit rendering window will be created.

**-doublebuffer, -db *value***

These are boolean options which state that the shell should (or should not) attempt to create a widget which supports double buffering. If the host machine does not support double buffering, this option will cause the command to fail.

**-bufferize, -bs *value***

This option specifies the size of the buffer for eight bit rendering widgets. It controls the number of entries available in the widget's colormap. If the host machine does not support a bufferize of *value*, the widget creation will fail.

**-zbufferize, -zbs *value***

This option specifies that a widget which supports a depth buffer should be created, and the size of the depth buffer should provide at least *value* bits of precision. If the host machine does not support a zbuffer size of *value*, the widget creation will fail.

**-redbits, -rb *value*****-greenbits, -gb *value*****-bluebits, -bb *value*****-alphabits, -ab *value***

This collection of options specify the number of bits for the red, green, blue, and alpha channels of a 24 bit color widget. If the host machine does not support a 24 bit buffer defined by the given *value*'s, the widget creation will fail.

**Widget Command:**

The `glRaster` command returns the pathname of the OpenGL raster widget created. This string represents a new Tcl command with its own options. These commands with the options take the following syntax:

```
pathName option [arg arg ...]
```

**Widget Options:****current**

This option returns the id of the active buffer (e.g. overlay). The return value will be 0, 1, or 2. These values represent the normal buffer, overlay, and underlay, respectively. The active buffer receives any rendering commands (see below).

**mapped**

**mapped** returns a boolean value which states whether the widget's window has been mapped to the screen. Rendering should not be performed until the widget has been mapped.

**configure**

The **configure** command can be used to redefine an already created object. *options* are the same as those that can be specified when the object was created. If no arguments are given the **configure** command will return a list of the functions defining characteristics in the following form:

```
{ { option value } ... }
```

Note: If the widget has already been “mapped”, its buffer characteristics (e.g. doublebuffer, redbits, etc.) cannot be changed.

**Buffer Command(s):**

If the widget command has the following format

```
pathName bufferName option [arg arg ...]
```

then *option* should be one of the special buffer commands. In this form, *buffer-Name* can be *normal*, *overlay*, or *underlay*.

**Buffer Options:****exists**

This option returns a boolean value indicating whether the given buffer exists (has been created).

**current**

**current** returns a boolean value indicating whether the given buffer is the active buffer.

**active**

Assuming the given buffer had been created, **active** will make the buffer the active buffer (e.g. the one receiving the graphics commands).

**create [options]**

**create** will attempt to create the given buffer with the specified options. *options* can be any of the options available when creating the `glRaster` widget except **-width** and **-height**. Attempting to create a buffer which already exists is an error.

**configure**

The **configure** command can be used to redefine an already created object. *options* are the same as those that can be specified when the object was created. If no arguments are given the **configure** command will return a list of the functions defining characteristics in the following form:

```
{ { option value } ... }
```

Note: If the widget has already been “mapped”, its buffer characteristics (e.g. doublebuffer, redbits, etc.) cannot be changed.

### C.5.2 OpenGL Rendering Commands

The supported OpenGL functions are described in this section. To make the transition of the Tcl implementations, the Tcl commands have been given the same names as the OpenGL functions.

**Tcl Command:**

`glAddColor`

**Synopsis:**

`glAddColor index red green blue`

**Description:**

`glAddColor` attempts to insert the color defined by the *red*, *green*, and *blue* triple into the *index*<sup>*h*</sup> location of the colormap. Note that this command is only valid when the active buffer is using the eight-bit (colormap) color model.

**Tcl Command:**

`glAddColors`

**Synopsis:**

`glAddColors colormap`

**Description:**

`glAddColors` attempts to insert the colors given by *colormap* into the colormap of the active buffer. *colormap* is the colormap data structure defined earlier in this appendix. Note that this command is only valid when the active buffer is using the eight-bit (colormap) color model.



**Tcl Command:**`glBegin/glEnd`**Synopsis:**`glBegin primitive  
glEnd`**Description:**

This is the Tcl implementation of the OpenGL `glBegin()` function. Its only parameter is the name of the OpenGL primitive. *primitive* can take one of the following values:

<code>points</code>	<code>lines</code>
<code>line_strip</code>	<code>line_loop</code>
<code>triangles</code>	<code>triangle_strip</code>
<code>triangle_fan</code>	<code>quads</code>
<code>quad_strip</code>	<code>polygon</code>

`glBegin...glEnd` is used to delimit a group of like primitives.

**Tcl Command:**`glCallList`**Synopsis:**`glCallList listNumber`**Description:**

`glCallList` executes the display list given by *listNumber*. The display list given by *listNumber* should have been previously defined using `glNewList...glEndList`.

**Tcl Command:**`glClearColor`**Synopsis:**`glClearColor [options]`**Description:**

`glClearColor` clears the current color buffer to the last set color buffer background color.

**Options:**`-color colorSpec`

This option sets the clear (background) color before actually clearing the current color buffer. If the current buffer uses the RGBA colormodel, then *colorSpec* should be a RGBA list as defined at the beginning of this appendix. Likewise, if the current buffer uses the colormap model, then *colorSpec* should be an integral index into the buffer's colormap.

**Tcl Command:**`glClearDepth`**Synopsis:**`glClearDepth [value]`**Description:**

`glClearDepth` clears the current depth buffer to the last set depth buffer background color. If *value* is given, then the background color is first set to *value* before it is cleared.

**Tcl Command:**`glClipPlane`**Synopsis:**`glClipPlane clipPlane plane`**Description:**

`glClipPlane` defines/sets the clip plane specified by *clipPlane* whose value has the form *clip\_plane#*, where *#* = 0, 1, 2, 3, 4, or 5. *plane* is a list containing four items; *A*, *B*, *C*, *D*. These items specify the equation of a plane;  $Ax + By + Cz + D = 0$ . Note, that the clip plane still must be enabled (see `glEnable`).

**Tcl Command:**`glColor`**Synopsis:**`glColor value`**Description:**

`glColor` sets the current drawing color for the active buffer. If the active buffer uses the RGBA color model, then *value* should be a RGBA color list. If it uses the colormap color model, then *value* should be an integral index into the active buffer's colormap.

**Tcl Command:**`glColorMaterial`**Synopsis:**`glColorMaterial face mode`**Description:**

`glColorMaterial` causes a material color to track the current color. *face* specifies whether the *front*, *back*, or *front\_and\_back* face material should track the color. *mode* specifies which material property should track the color. *mode*'s value can be either *ambient*, *diffuse*, *ambient\_and\_diffuse*, *specular*, *shininess*, or *emission*.

**Tcl Command:**`glCullFace`**Synopsis:**`glCullFace mode`**Description:**

`glCullFace` specifies which facets should be culled when facet culling is enabled. *mode* can be either *front*, *back*, or *front\_and\_back*.

**Tcl Command:**`glDeleteLists`**Synopsis:**`glDeleteLists start range`**Description:**

`glDeleteLists` deletes a contiguous group of display lists starting with the display list specified by the integral id *start*. *range* specifies the number of display lists to delete. If *range* is zero, nothing happens.

**Tcl Command:**`glDepthFunc`**Synopsis:**`glDepthFunc function`**Description:**

`glDepthFunc` specifies the depth comparison function used for depth buffer comparisons. The legal values for *function* are

never	less
equal	lequal
greater	notequal
gequal	always

The default value is *less*.

**Tcl Command:**`glDisable/glEnable`**Synopsis:**`glDisable/glEnable cap`**Description:**

`glDisable/glEnable` disables/enables the given capability. *cap* can be one of the following several values;

<code>alpha_test</code>	<code>auto_normal</code>	<code>blend</code>
<code>clip_plane0</code>	<code>clip_plane1</code>	<code>clip_plane2</code>
<code>clip_plane3</code>	<code>clip_plane4</code>	<code>clip_plane5</code>
<code>color_material</code>	<code>cull_face</code>	<code>depth_test</code>
<code>dither</code>	<code>fog</code>	<code>light0</code>
<code>light1</code>	<code>light2</code>	<code>light3</code>
<code>light4</code>	<code>light5</code>	<code>light6</code>
<code>light7</code>	<code>lighting</code>	<code>line_smooth</code>
<code>line_stipple</code>	<code>logic_op</code>	<code>map1_color_4</code>
<code>map1_index</code>	<code>map1_normal</code>	<code>map1_texture_coord_1</code>
<code>map1_texture_coord_2</code>	<code>map1_texture_coord_3</code>	<code>map1_texture_coord_4</code>
<code>map1_vertex_3</code>	<code>map1_vertex_4</code>	<code>map2_color_4</code>
<code>map2_index</code>	<code>map2_normal</code>	<code>map2_texture_coord_1</code>
<code>map2_texture_coord_2</code>	<code>map2_texture_coord_3</code>	<code>map2_texture_coord_4</code>
<code>map2_vertex_3</code>	<code>map2_vertex_4</code>	<code>normalize</code>
<code>point_smooth</code>	<code>polygon_smooth</code>	<code>polygon_stipple</code>
<code>scissor_test</code>	<code>stencil_test</code>	<code>texture_ld</code>
<code>texture_2d</code>	<code>texture_gen_q</code>	<code>texture_gen_r</code>
<code>texture_gen_s</code>	<code>texture_gen_t</code>	

**Tcl Command:**`glFinish`**Synopsis:**`glFinish`**Description:**

`glFinish` causes the rendering pipeline to block until all previously executed OpenGL commands have completed.

**Tcl Command:**`glFlush`**Synopsis:**`glFlush`**Description:**

`glFlush` forces the execution of all previously executed OpenGL commands to complete in finite time.

**Tcl Command:**`glFrontFace`**Synopsis:**`glFrontFace mode`**Description:**

`glFrontFace` defines the vertex orientation of front facing polygons. *mode* can be either *cw* or *ccw*. The first stands for “clockwise” and the second “counter-clockwise.”

**Tcl Command:**`glFrustum`**Synopsis:**`glFrustum left right bottom top near far`**Description:**

`glFrustum` multiplies the current matrix by a perspective projection matrix. *left*, *right* specify the coordinates of the left and right clipping planes. Likewise, *bottom*, *top* specify the coordinates of the bottom and top horizontal clipping planes. Finally, *near*, *far* specify the distances to the near and far clipping planes.

**Tcl Command:**`glGetInteger`**Synopsis:**`glGetInteger parameter`**Description:**

`glGetInteger` is a partial implementation of the OpenGL `glGetInteger()` function. Currently, the only accepted values for *parameter* are *red\_bits*, *green\_bits*, *blue\_bits*, *alpha\_bits*, and *depth\_bits*. This function returns the current value of the requested parameter.

**Tcl Command:**`glIsList`**Synopsis:**`glIsList listID`**Description:**

`glIsList` returns a boolean value indicating whether *listID* is a defined (via `glNewList`) display list.

**Tcl Command:**`glLight`**Synopsis:**`glLight light parameter value`**Description:**

`glLight` is used to define the characteristics of a light source. The *light* parameter has the form “light $i$ ” where  $i = 0 \dots 7$ . *parameter* defines the characteristic which will be set equal to *value*. Legal values for *parameter* are

ambient	spot_exponent
specular	spot_cutoff
diffuse	constant_attenuation
position	quadratic_attenuation
spot_direction	linear_attenuation

The format of *value* depends on *parameter*. The first three parameters expect an RGBA color list, the next two expect a 3d or homogeneous coordinate, and the last five expect a floating point value.

**Tcl Command:**`glLightModel`**Synopsis:**`glLightModel parameter value`**Description:**

`glLightModel` specifies light model characteristics. Legal values for *parameter* are *light\_model\_ambient*, *light\_model\_local\_viewer*, and *light\_model\_two\_side*. *light\_model\_ambient* expects *value* to be a RGBA color list. The other two parameters expect *value* to be a string (either “true” or “false”).

**Tcl Command:**`glLineWidth`**Synopsis:**`glLineWidth value`**Description:**

`glLineWidth` specifies the width of rasterized lines. The default value is 1.0.



**Tcl Command:**`glLoadIdentity`**Synopsis:**`glLoadIdentity`**Description:**

`glLoadIdentity` replaces the current matrix with the identity matrix.

**Tcl Command:**`glLoadMatrix`**Synopsis:**`glLoadMatrix matrix`**Description:**

`glLoadMatrix` replaces the current matrix with *matrix*; *matrix* is a list containing sixteen elements specified in column-major order. That is, the list

$$\{ a_0 a_1 \dots a_{15} \}$$

is interpreted as the following matrix

$$\begin{vmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{vmatrix}$$

**Tcl Command:**`glMaterial`**Synopsis:**`glMaterial face parameter value`**Description:**

`glMaterial` specifies material characteristics for the facets of polygonal models. *face* specifies which side of the facets are affected by the material characteristics. The legal values for this option are *front*, *back*, and *front\_and\_back*. *parameter* specifies the characteristic which will be set equal to *value*. Legal values for *parameter* are *ambient*, *diffuse*, *ambient\_and\_diffuse*, *specular*, *emission*, *shininess*, or *color\_indexes*. The first five parameter values expect *value* to be a RGBA color list. *shininess* expects a floating point value, and *color\_indexes* expects *value* to be a list of indexes into the current colormap.

**Tcl Command:**`glMatrixMode`**Synopsis:**`glMatrixMode mode`**Description:**

`glMatrixMode` specifies the current matrix (e.g. that which will receive subsequent matrix operations). Three values for *mode* are accepted; *modelview*, *projection*, and *texture*.

**Tcl Command:**`glMultMatrix`**Synopsis:**`glMultMatrix matrix`**Description:**

`glMultMatrix` multiplies the current matrix by *matrix*. That is, if the current matrix is *A* and *matrix* is *B*, then the current matrix is replaced by *AB*. *matrix* is a list containing sixteen elements specified in column-major order. That is, the list

$$\{ a_0 a_1 \dots a_{15} \}$$

is interpreted as the following matrix

$$\begin{vmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{vmatrix}$$

**Tcl Command:**

`glNewList/glEndList`

**Synopsis:**

`glNewList listID mode`  
`glEndList`

**Description:**

`glNewList...glEndList` creates or replaces a display list. `glNewList` defines a display list with calling id *listID*. All OpenGL commands which occur after `glNewList` and before `glEndList` become part of the display list, and can be executed at anytime by calling `glCallList` with the proper *listID*. *mode* states whether the display list should just be compiled (*compile*) or compiled and executed (*compile\_and\_execute*).

**Tcl Command:**

`glOrtho`

**Synopsis:**

`glOrtho left right bottom top near far`

**Description:**

`glOrtho` multiplies the current matrix by a orthographic projection matrix. *left*, *right* specify the coordinates of the left and right clipping planes. Likewise, *bottom*, *top* specify the coordinates of the bottom and top horizontal clipping planes. Finally, *near*, *far* specify the distances to the near and far clipping planes.

**Tcl Command:**

`glPixelStore`

**Synopsis:**

`glPixelStore parameter value`

**Description:**

`glPixelStore` sets the pixel storage modes. Legal values for *parameter* are

<code>unpack_swap_bytes</code>	<code>pack_swap_bytes</code>	<code>unpack_lsb_first</code>
<code>pack_lsb_first</code>	<code>unpack_row_length</code>	<code>pack_row_length</code>
<code>unpack_skip_rows</code>	<code>pack_skip_rows</code>	<code>unpack_skip_pixels</code>
<code>pack_skip_pixels</code>	<code>unpack_alignment</code>	<code>pack_alignment</code>

*\*swap\_bytes* and *\*lsb\_first* expect *value* to be either “true” or “false.” All remaining parameters expect an integral value.

**Tcl Command:**

`glPixelZoom`

**Synopsis:**

`glPixelZoom` *xfactor* *yfactor*

**Description:**

`glPixelZoom` sets the pixel zoom factors for the `glDrawPixels` and `glCopyPixels` operations.

**Tcl Command:**

`glPointSize`

**Synopsis:**

`glPointSize` *value*

**Description:**

`glPointSize` sets the diameter of rasterized points. The default value is 1.0.

**Tcl Command:**

`glPolygonMode`

**Synopsis:**

`glPolygonMode` *face* *mode*

**Description:**

`glPolygonMode` specifies how polygons are rendered. *face* specifies which side of a polygon (as defined by `glFrontFace`) is affected by this `glPolygonMode` operation. *face* can be either *front*, *back*, or *front\_and\_back*. *mode* specifies how then given facets will be rendered. Legal values are *point*, *line*, and *fill*.

**Tcl Command:**

`glPushMatrix/glPopMatrix`

**Synopsis:**

`glPushMatrix`  
`glPopMatrix`

**Description:**

These commands operate on the current matrix stack. `glPushMatrix` creates a copy of the current matrix, and places it on top of the current matrix stack. `glPopMatrix` removes the top matrix from the current matrix stack.

**Tcl Command:**

`glRasterPos`

**Synopsis:**

`glRasterPos x y [z] [w]`

**Description:**

`glRasterPos` sets the current raster position to the coordinate given by  $(x, y, z, w)$  where  $z$  and  $w$  are optional.  $z$  defaults to 0.0 and  $w$  defaults to 1.0.

**Tcl Command:**

`glRect`

**Synopsis:**

`glRect x y width height`

**Description:**

`glRect` draws a rectangle whose lower left hand vertex is located at  $(x, y)$  and whose dimensions are *widthXheight*.

**Tcl Command:**`glRotate`**Synopsis:**`glRotate [options]`**Description:**

`glRotate` multiplies the current matrix by the rotation matrix specified through *options*. By default, a rotation of 0 degrees about the *z* axis is performed.

**Options:**`-angle value`

Set the rotation angle to *value*

`-x value``-y value``-z value`

These three options define the axis of rotation. By default, the axis of rotation (and the values default to) is (0.0, 0.0, 1.0)

**Tcl Command:**`glScale`**Synopsis:**`glScale [options]`**Description:**

`glScale` multiplies the current matrix by the scaling matrix specified through *options*. By default, a scale of 1 is performed through each axis. That is, by default no scaling will occur.

**Options:**`-x value``-y value``-z value`

These three options define the amount of scaling in each axis.

**Tcl Command:**`glShadeModel`**Synopsis:**`glShadeModel mode`**Description:**

`glShadeModel` allows the user to select *flat* or *smooth* shading.

**Tcl Command:**`glSwap`**Synopsis:**`glSwap`**Description:**

`glSwap` is the Tcl command to execute the `glXSwapBuffers()` function. It will swap the back and front buffers when double buffering is being used.

**Tcl Command:**`glTranslate`**Synopsis:**`glTranslate [options]`**Description:**

`glTranslate` multiplies the current matrix by the translation matrix specified through *options*. By default, a translation of 0 is performed along each axis. That is, by default no translation will occur.

**Options:**`-x value``-y value``-z value`

These three options define the amount of translation along each axis.

**Tcl Command:**

`glVertex`

**Synopsis:**

`glVertex position`

**Description:**

`glVertex` defines a vertex to be rendered. *position* is a coordinate list as defined in C.1.

**Tcl Command:**

`glViewport`

**Synopsis:**

`glViewport [x y width height]`

**Description:**

`glViewport` sets the viewport. If *x*, *y*, *width*, and *height* are given then the viewport's lower left corner will be located at (*x*, *y*) and have dimensions *width*X*height*. The viewport defaults to the entire window (widget).



### C.5.3 GLU Rendering Commands

**Tcl Command:**

`gluCylinder`

**Synopsis:**

`gluCylinder [options]`

**Description:**

`gluCylinder` draws a cylinder object. The characteristics of the cylinder are specified via the *options* described below. The cylinder is oriented along the z axis with its base at  $z = 0$  and its top at  $z = \text{height}$ . This Tcl command actually combines the following GLU functions; `gluNewQuadric()`, `gluCylinder()`, `gluQuadricOrientation()`, `gluQuadricNormals()`, and `gluQuadricDrawStyle()`. This command returns an id for the new quadric. This id is used with other commands which manipulate quadrics.

**Options:**

**-baseRadius *value***

Specify the length of the radius at the cylinder's base. By default, *value* is set equal to 1.0.

**-topRadius *value***

Specify the length of the radius at the cylinder's top. By default, *value* is set equal to 1.0.

**-slices *value***

Specify the number of subdivisions around the z axis. By default, this characteristic is equal to 5.

**-stacks *value***

Specify the number of subdivisions along the z axis. By default, this characteristic is equal to 5.

**-normals *string***

Specify what type of normals should be generated. *string* can be *none*, *flat*, or *smooth*. By default, it equals *none*.

**-orientation *string***

This option specifies the orientation of generated normals. If *string* is *outside*, then the normals will point away from the z axis. If *string* is *inside*, they will point toward the z axis. By default, the orientation is *outside*.

**-drawstyle *string***

*-drawstyle* specifies how the cylinder should be drawn. Legal values for *string* are *fill*, *line*, *silhouette*, or *point*.

**Tcl Command:**

`gluDeleteQuadric`

**Synopsis:**

`gluDeleteQuadric quadricID`

**Description:**

`gluDeleteQuadric` deletes the quadric defined by *quadricID*. Once *quadricID* is destroyed, it cannot be referenced again.

**Tcl Command:**

`gluDisk`

**Synopsis:**

`gluDisk [options]`

**Description:**

`gluDisk` draws a disk object. The characteristics of the disk are specified via the *options* described below. This Tcl command actually combines the following GLU functions; `gluNewQuadric()`, `gluDisk()`, `gluQuadricOrientation()`, `gluQuadricNormals()`, and `gluQuadricDrawStyle()`. This command returns an id for the new quadric. This id is used with other commands which manipulate quadrics.

**Options:**

**-outerRadius *value***

Specify the length of the outer radius. By default, *value* is set equal to 1.0.

**-innerRadius *value***

Specify the length of the inner radius. By default, *value* is set equal to 0.0. If the inner radius is not equal to 0.0, then a hole will be generated.

**-slices *value***

Specify the number of subdivisions around the z axis. By default, this characteristic is equal to 5.

**-loops *value***

Specify the number of concentric rings about the disk's center. By default, this characteristic is equal to 5.

**-normals *string***

Specify what type of normals should be generated. *string* can be *none*, *flat*, or *smooth*. By default, it equals *none*.

**-orientation *string***

This option specifies the orientation of generated normals. If *string* is *outside*, then the normals will point away from the z axis. If *string* is *inside*, they will point toward the z axis. By default, the orientation is *outside*.

**-drawstyle *string***

*-drawstyle* specifies how the disk should be drawn. Legal values for *string* are *fill*, *line*, *silhouette*, or *point*.

**Tcl Command:**

`gluLookAt`

**Synopsis:**

`gluLookAt [options]`

**Description:**

`gluLookAt` defines a viewing transformation based on an eye point, a reference point, and an up vector. These are specified via the *options*, and are described below.

**Options:****-eye *point***

Specifies the location of the viewer. By default, this is set to  $\{0.0, 0.0, 0.5\}$ .

**-lookat *point***

Specifies the reference point at which the viewer is looking. By default, this is set to  $\{0.0, 0.0, 0.0\}$ .

**-up *point***

Specifies what direction is up. By default, this is set to  $\{0.0, 1.0, 0.0\}$ .

**Tcl Command:****gluNewQuadric****Synopsis:****gluNewQuadric****Description:**

**gluNewQuadric** creates a new quadric object, and returns its id. This id should be used with the other commands that operate on quadric objects.

**Tcl Command:****gluSphere****Synopsis:****gluSphere** [*options*]**Description:**

**gluSphere** draws a sphere object. The characteristics of the sphere are specified via the *options* described below. The sphere is centered at the origin. This Tcl command actually combines the following GLU functions; **gluNewQuadric()**, **gluSphere()**, **gluQuadricOrientation()**, **gluQuadricNormals()**, and **gluQuadricDrawStyle()**. This command returns an id for the new quadric. This id is used with other commands which manipulate quadrics.

**Options:****-radius** *value*

Specify the length of the radius of the sphere. By default, *value* is set equal to 1.0.

**-slices** *value*

Sets the number of subdivisions around the z axis. By default, this is set to 20.

**-stacks** *value*

Sets the number of subdivisions along the z axis. By default, this is set to 20.

**-normals** *string*

Specify what type of normals should be generated. *string* can be *none*, *flat*, or *smooth*. By default, it equals *none*.

**-orientation** *string*

This option specifies the orientation of generated normals. If *string* is *outside*, then the normals will point away from the z axis. If *string* is *inside*, they will point toward the z axis. By default, the orientation is *outside*.

**-drawstyle** *string*

*-drawstyle* specifies how the sphere should be drawn. Legal values for *string* are *fill*, *line*, *silhouette*, or *point*.

### C.5.4 RQS Rendering Commands

This section describes some new Tcl commands that use OpenGL functions to render various RQS objects. These objects include images, polygonal meshes, contours, error images (e.g. via `ivpImage`), and individual slices of volumetric data.

#### Tcl Command:

`glContour`

#### Synopsis:

`glContour volumeName [options]`

#### Description:

`glContour` renders contours present in the volumetric data set define by *volumeName*. *options* control what contours are rendered and how they are rendered.

#### Options:

**-help**

Print out available options

**-low value**

This option sets the low threshold value for segmenting the volumetric data set. By default, low is equal to 0.0.

**-high value**

This option sets the high threshold value for segmenting the volumetric data set. By default, high is equal to 255.0.

**-slice value**

Specify which slice a contour should be generated on. By default, the first slice is selected.

**-npts value**

Specify the number of points defining a generated contour. If *value* is a positive value, the contour will be redefined having *value* uniformly distributed (with respect to arc length) points. By default, *npts* is equal to -1.

**-display mode**

Specify how a generated contour should be rendered. *mode* is one of the following strings; *polygon*, *spline*, or *control*. If *mode* is *polygon* then the contour is displayed as polygon. If *mode* is *spline*, the contour will be fitted first by a uniform B-spline, and then rendered. Finally, if *mode* is *control*, the control polygon of the uniform B-spline is rendered. By default, *mode* is *polygon*.

**Tcl Command:**`glImage`**Synopsis:**`glImage command fileName [options]`**Description:**

`glImage` provides an interface to saving and retrieving images. If *command* is *get*, a subimage is retrieved from the active rendering window and written to *fileName* in a PBM format. If the rendering window uses the RGBA color model, the image will be written using the raw PPM format. If the window uses the colormap color model, the image will be written using the raw PGM format. The retrieved subimage can be defined via *options*.

If *command* is *put*, the image stored in *fileName* is retrieved and rendered in the active rendering window at the current raster position (see `glRasterPos`). The image type must match the widget's color model. For example, if the image is stored in the PPM format, the active rendering widget must use the RGBA color model.

**Get Options:****-x value**

Specify the X coordinate of the lower left hand corner of the subimage to be retrieved. By default, this option is equal to 0.0.

**-y value**

Specify the Y coordinate of the lower left hand corner of the subimage to be retrieved. By default, this option is equal to 0.0.

**-width value**

Specify the width of the subimage to be retrieved. By default, this option is equal to the width of the active rendering widget.

**-height value**

Specify the height of the subimage to be retrieved. By default, this option is equal to the height of the active rendering widget.

**Tcl Command:**

`glMesh`

**Synopsis:**

`glMesh meshName`

**Description:**

`glMesh` will render the polygonal mesh surface given by *meshName*. Typically, *meshName* will be the surface name returned by one of the available surface reconstruction algorithms (e.g. `march`).

**Tcl Command:**

`glMeshErrorImage`

**Synopsis:**

`glMeshErrorImage meshName minVal maxVal`

**Description:**

`glMeshErrorImage` renders the specified RQS mesh surface (*meshName*) in the given mode as an error image. From this, it is assumed the mesh data is a pointer to an index into the current colormap. Thus, the current rendering widget must be in color index mode. The mesh data is usually set by making a prior call to either `ivpImage` or `pdpImage`. *minVal* and *maxVal* give the minimum and maximum values of the vertex data. These are used to scale the vertex data.

**Tcl Command:**

`glVolumeSlice`

**Synopsis:**

`glVolumeSlice volumeName slice [options]`

**Description:**

`glVolumeSlice` gives allows slices of volumetric data to be rendered as images in a active `glRaster` widget. The volumetric data is specified through *volumeName*, and the slice to render is *slice*. The *options* described below allow segmentation to be performed on data slice before it is rendered.

**Options:**

`-low value`

Specify the low end of the segmentation range. By default, the low threshold value is greater than the high threshold value. When this relationship holds, no segmentation is performed.

**-high value**

Specify the high end of the segmentation range. By default, the high threshold value is less than the low threshold value. When this relationship holds, no segmentation is performed.



# Bibliography

- [1] E. Artzy. Display of three-dimensional information in computed tomography. *Computer Graphics and Image Processing*, 9(2):196–198, February 1979.
- [2] L.V. Atkinson and P.J. Harvey. *An Introduction to Numerical Methods with Pascal*. Addison-Wesley Publishing Company, 1983.
- [3] D.H. Ballard and C.M. Brown. Representation of three-dimensional structures. In *Computer Vision*, pages 264–291. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
- [4] T. A. Beauchat. Quantifying the goodness of (iso)surface reconstructions. Technical Report WM-94-3, The College of William and Mary, Williamsburg, VA, November 1994.
- [5] J.D. Boissonnat. Geometric structures for three-dimensional shape representation. *ACM Transactions on Graphics*, 3(4):266–286, October 1985.
- [6] J.D. Boissonnat. Shape reconstruction from planar cross sections. *Computer Vision, Graphics, and Image Processing*, 44(1):1–29, October 1988.
- [7] S. Bright and S. Laffin. Shading of solid voxel models. *Computer Graphics Forum*, 5(2):131–138, March 1986.
- [8] G. Cameron. Modular visualization environments : Past, present, and future. *Computer Graphics*, 29(2):3–4, May 1995.
- [9] L. S. Chen, G. T. Herman, R. A. Reynolds, and J. K. Udupa. Surface shading in the cuberille environment. *IEEE Computer Graphics and Applications*, pages 33–43. December 1985.
- [10] L.S. Chen and M.R. Sontag. Representation, display, and manipulation of 3d digital scenes and their medical applications. *Computer Vision, Graphics, and Image Processing*, 48(2):190–216, November 1989.
- [11] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, 1988.
- [12] J.L. Coatrieux and C. Barillot. A survey of 3d display techniques to render medical data. In K.H. Hoehne, H. Fuchs, and S.M. Pizer, editors, *3D Imaging in Medicine*:

- Algorithms, Systems, Applications*, pages 175–195. Springer-Verlag, Berlin, West Germany, 1990.
- [13] D. Cohen and A. Kaufman. *Scan Conversion Algorithms for Linear and Quadratic Objects*, chapter 5, pages 280–301. IEE Computer Society Press, Los Alamitos, CA, 1991.
- [14] D. Cohen, A. Kaufman, R. Bakalash, and S. Bergman. Real-time discrete shading. *The Visual Computer*, 6(1):16–27, February 1990.
- [15] J.C. Dill. An application of color graphics to the display of surface curvature. *Computer Graphics*, 15:153–161, 1981.
- [16] L.J. Doctor and J.G. Torbrog. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1(3):29–38, July 1981.
- [17] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.
- [18] H. Edelsbrunner and E. P. Mucke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, Jan 1994.
- [19] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1990.
- [20] G. Frieder, D. Gordon, and R. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1), January 1985.
- [21] H. Fuchs, Z.M. Kedem, and S.P. Uzelton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, Oct 1977.
- [22] S. Ganapathy and T. G. Dennehy. A new general triangulation method for planar contours. *Computer Graphics*, 16(3):69–75, July 1982.
- [23] M. P. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40, November 1990.
- [24] A. Globus and E. Raible. Fourteen ways to say nothing with scientific visualization. *IEEE Computer*, 27(7):36–88, July 1994.
- [25] A. Globus and S. Uzelton. Evaluation of visualization software. *Computer Graphics*, 29(2):41–44, May 1995.
- [26] R. Gonzalez and P. Wintz. *Digital Image Processing*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1987.
- [27] D. Gordon and R. A. Reynolds. Image space shading of 3-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 29:361–376, 1985.

- [28] A. Gueziec and R. Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Visualization and Computer Graphics*, 1(4):328–342, December 1990.
- [29] P. Hanrahan. Three-pass affine transforms for volume rendering. *Computer Graphics*, 24(5):71–78, November 1990.
- [30] G.T. Herman and S.K. Gabor. Display of 3d digital images: Computational foundations and medical applications. *IEEE Computer Graphics and Applications*, pages 39–46, August 1983.
- [31] G.T. Herman and H.K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing*, 9(1):1–21, January 1979.
- [32] K.H. Hoehne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, and G. Wiebecke. 3d-visualization of tomographic volume data using the generalized voxel model. *The Visual Computer*, 6(1):28–37, February 1990.
- [33] I. Ihm and B. Naylor. Piecewise linear approximations of digitized space curves with applications. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 545–569. Springer-Verlag, 1991.
- [34] C. Jackins and S. L. Tanimoto. Oct-trees and their use in representing three dimensional objects. *Computer Graphics and Image Processing*, 14(31), November 1980.
- [35] G.J. Jense and D.P. Huijsmans. Interactive voxel-based graphics for 3d reconstruction of biological structures. *Computers and Graphics*, 13(2):145–150, 1989.
- [36] J.T. Kajiya and B.P. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(3):165–174, July 1984.
- [37] A. Kaufman. An algorithm for 3d scan-conversion of polygons. In *Proceedings EUROGRAPHICS '87*, pages 197–208, Amsterdam, The Netherlands, 1987. North-Holland Publishing Co.
- [38] A. Kaufman. Efficient algorithms for 3d scan-conversion of parametric curves. *Proceedings of Computer Graphics '87*, pages 171–179, 1987.
- [39] A. Kaufman. Efficient algorithms for 3d scan-conversion of polygons. *Computer and Graphics*, 12(2):213–219, 1988.
- [40] A. Kaufman. Introduction to volume synthesis. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 27–35. Springer-Verlag, Tokyo, 1991.
- [41] A. Kaufman. *Volume Visualization*, chapter 1, pages 1–18. IEEE Computer Society Press, Los Alamitos, California, 1991.
- [42] A. Kaufman and E. Shimony. 3d scan-conversion algorithms for voxel-based graphics. In *Proceedings ACM Workshop on Interactive 3D Graphics*, pages 45–75, New York, October 1987. ACM Inc.

- [43] A. Kaufman, R. Yagel, R. Bakalash, and I. Spector. Volume visualization in cell biology. In *Proceedings of the First IEEE Conference on Visualization - Visualization '90*, pages 160–167. IEEE, 1990. CG International '91 Course Notes.
- [44] A. Kaufman, R. Yagel, and D. Cohen. Intermixing surface and volume rendering. In K.H. Hohne et al., editor, *3D Imaging in Medicine*, volume F60, pages 217–225, Berlin, 1990. Springer-Verlag. CG International '91 Course Notes.
- [45] A. Kaufman, R. Yagel, and W. Lorensen. Introduction to volume visualization. CG International '91 Course Notes, June 1991.
- [46] R. Lenz, B. Gudmundsson, B. Lindskog, and P.E. Danielsson. Display of density volumes. *IEEE Computer Graphics and Applications*, 6(7):20–29, July 1986.
- [47] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, pages 29–37, August 1988.
- [48] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.
- [49] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, February 1990.
- [50] H. D. Lord. Improving the application development process with modular visualization environments. *Computer Graphics*, 29(2):10–12, May 1995.
- [51] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.
- [52] W. E. Lorensen and H. E. Cline. Volume modeling. In *CG International '91 Course Notes*, June 1991.
- [53] M. Magnusson, R. Lenz, and P.E. Danielsson. Evaluation of methods for shaded display of ct volumes. In *Proceedings 9th International Conference on Pattern Recognition*, volume 2, pages 1287–1294, November 1988.
- [54] S.R. Marschner and R.J. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings Visualization '94*, pages 100–107. IEEE Computer Society, 1994.
- [55] D.F. McAllister, E. Passow, and J. A. Roulier. Algorithms for computing shape preserving spline interpolations to data. *Mathematics of Computation*, 31(139):717–725, July 1977.
- [56] D.F. McAllister and J.A. Roulier. Interpolation by convex quadratic splines. *Mathematics of Computation*, 32(144):1154–1162, October 1978.
- [57] D.F. McAllister and J.A. Roulier. An algorithm for computing a shape-preserving osculatory quadratic spline. *ACM Transactions on Mathematical Software*, 7(3):331–347, September 1981.

- [58] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.
- [59] D. Meyers, S. Skinner, and K. Sloan. Surfaces from contours. *ACM Transactions on Graphics*, 11(3):228–258, July 1992.
- [60] D. Mitchell and A. Netravali. Reconstruction filters in computer graphics. *Computer Graphics*, 22(4):221–228, August 1988.
- [61] H. Neeman. A decomposition algorithm for visualizing irregular grids. *Computer Graphics*, 24(5):49–56, November 1990.
- [62] D.R. Ney, E.K. Fishman, D. Magid, and A. Drebin. Volumetric rendering of computed tomography data: Principles and techniques. *IEEE Computer Graphics and Applications*, 10(3):24–32, March 1990.
- [63] G.M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proceedings of Visualization '91*, pages 83–91, 1991.
- [64] T. Ohashi, T. Uchiki, and M. Tokoro. A three-dimensional shaded display method for voxel-based representation. In *Proceedings EUROGRAPHICS '85*, pages 221–232. Amsterdam, The Netherlands, 1985. EUROGRAPHICS, North-Holland Publishing Co.
- [65] S.K. Park. Lecture notes on simulation. Version 2.0, August 1989.
- [66] S.K. Park. Lecture notes on digital image processing. Version 2.1, August 1991.
- [67] S.K. Park. Lecture notes on fourier methods : The modeling, simulation, and performance analysis of discrete linear systems. Version 4.0, 1993.
- [68] V. Patel, M. Vannier, J. Marsh, and L. Lo. Assessing craniofacial surgical simulation. *IEEE Computer Graphics and Applications*, 16(1):46–54, January 1996.
- [69] A. Pommert, M. Bomans, U. Tiede, and K.H. Hoehne. Simulation studies for quality assurance of 3d-images from computed tomograms. In A. Todd-Pokropek and M.A. Viergever, editors, *The Formation, Handling, and Evaluation of Medical Images*. Springer-Verlag, Berlin, West Germany, 1989.
- [70] A. Pommert, U. Tiede, G. Wiebecke, and K.H. Hoehne. Image quality in voxel-based surface shading. In *Proceedings of the International Symposium on Computer Assisted Radiology*, pages 737–741, Berlin, West Germany, 1989. Springer-Verlag.
- [71] S.P. Raya and J.K. Udupa. Shape-based interpolation of multidimensional objects. *IEEE Transactions on Medical Imaging*, MI-9(1):32–42, March 1990.
- [72] D. F. Rogers and J. A. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, 1990.
- [73] P. Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics*, 22(4):51–57, August 1988.

- [74] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1990.
- [75] H. Senay and E. Ignatius. Compositional analysis and synthesis of scientific data visualization techniques. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 269–280. Springer-Verlag, Tokyo, 1991.
- [76] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, November 1990.
- [77] R. Shu and R. C. Krueger. A 3d surface reconstruction algorithm for volume data. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 251–266. Springer-Verlag, Tokyo, 1991.
- [78] D.A. Southard. Piecewise planar surface models from sampled data. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 667–680. Springer-Verlag, 1991.
- [79] D. Speray and S. Kennon. Volume probes: Interactive data exploration on arbitrary grids. *Computer Graphics*, 24(5):5–12, November 1990.
- [80] A. Sunguroff and D. Greenberg. Computer generated images for medical applications. *Computer Graphics*. 12(3):196–202, August 1978.
- [81] U. Tiede, K.H. Hoehne, M. Bomans, A. Pommert, M. Riemer, and G. Wiebecke. Investigation of medical 3d-rendering algorithms. *IEEE Computer Graphics and Applications*. 10(3):41–53, March 1990.
- [82] T. Totsuka and M. Levoy. Frequency domain volume rendering. *Computer Graphics*, pages 271–278. Aug 1993.
- [83] Y. Trousser and F. Schmitt. Active-ray tracing for 3d medical imaging. In *Proceedings EUROGRAPHICS '87*, pages 139–150, Amsterdam, The Netherlands, 1987. North-Holland Publishing Co.
- [84] H. K. Tuy and L. T. Tuy. Direct 2-d display of 3-d objects. *IEEE Computer Graphics and Applications*, pages 29–33, November 1984.
- [85] C. Upson. The visual simulation of amorphous phenomena. *The Visual Computer*, 2(5):321–326, May 1986.
- [86] C. Upson and M. Keefer. V-buffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, August 1988.
- [87] A. van Gelder and J. Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337–375, oct 1994.
- [88] M.W. Vannier, J.L. Marsh, and J.O. Warren. Three dimensional computer graphics for craniofacial surgical planning and evaluation. *Computer Graphics*, 17(3):263–273, July 1983.

- [89] R.E. Webber. Ray tracing voxel based data via biquadratic local surface interpolation. *The Visual Computer*, 6(1):8–15, February 1990.
- [90] J. Wilhelms. An overview of visualization techniques : Draft. Technical Report UCSC-CRL-89-37, University of California, Santa Cruz, CA, November 1989.
- [91] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *Computer Graphics*, 24(5):57–62, November 1990.
- [92] J. Wilhelms and A. Van Gelder. Topological considerations in isosurface generation. *Computer Graphics*, 24(5):79–86, November 1990.
- [93] K. Wu and L. Hesselink. Computer display of reconstructed 3-d scalar data. *Applied Optics*, 27(2):395–404, January 1988.
- [94] S. Wu, J.F. Abel, and D. P. Greenberg. An interactive computer graphics approach to surface representation. *Communications of the ACM*, 20(10):703–712, Oct 1977.
- [95] R. Yagel, D. Cohen, and A. Kaufman. Context sensitive normal estimation for volume imaging. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*. pages 211–230. Springer-Verlag, Tokyo, 1991.
- [96] K. Yamaguchi, T.L. Kunii, and K. Fujimura. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, pages 53–59, January 1984.

## VITA

### Tracey Allen Beauchat

The author was born in Titusville, Pennsylvania, 15, October 1967. He graduated from Titusville High School June, 1985. After high school, he went on to Allegheny College, a small private college in western Pennsylvania. He received his B.S. in Computer Science from Allegheny College May, 1989, and received honors for his Senior Thesis entitled "Natural Language Processing using Neural Networks."

September 1989, the author entered the College of William and Mary in Virginia. He received his M.S. May, 1991. In September of that same year he was accepted into Candidacy for the Ph.D. During his last three years at The College of William and Mary in Virginia he received a grant from the Virginia Space Grant Consortium.

Tracey is currently employed by E-OIR Measurements, Inc. in Spotsylvania, Virginia.