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.

U·M·I

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

Order Number 9310212

Interactive visualization tools for topological exploration

Heng, Pheng Ann, Ph.D.

Indiana University, 1992

Copyright ©1992 by Heng, Pheng Ann. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

INTERACTIVE VISUALIZATION TOOLS FOR TOPOLOGICAL EXPLORATION

by

Pheng Ann Heng

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

September 1992

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Andrew J. Hanson

Dr. Andrew J. Hanson
(Principal Adviser)

Dr. Dennis Gannon

Chris Haynes

Dr. Christopher T. Haynes

R. Glassey

Dr. Robert Glassey

August 26, 1992.

Dr. Charles Livingston

© Copyright 1992
Pheng Ann Heng
ALL RIGHTS RESERVED

To my wife, Soon-Keo and my parents

Abstract

This thesis concerns using computer graphics methods to visualize mathematical objects. Abstract mathematical concepts are extremely difficult to visualize, particularly when higher dimensions are involved; I therefore concentrate on subject areas such as the topology and geometry of four dimensions which provide a very challenging domain for visualization techniques.

In the first stage of this research, I applied existing three-dimensional computer graphics techniques to visualize projected four-dimensional mathematical objects in an interactive manner. I carried out experiments with direct object manipulation and constraint-based interaction and implemented tools for visualizing mathematical transformations. As an application, I applied these techniques to visualizing the conjecture known as *Fermat's Last Theorem*.

Four-dimensional objects would best be perceived through four-dimensional eyes. Even though we do not have four-dimensional eyes, we can use computer graphics techniques to simulate the effect of a virtual four-dimensional camera viewing a scene where four-dimensional objects are being illuminated by four-dimensional light sources. I extended standard three-dimensional lighting and shading methods to work in the fourth dimension. This involved replacing the standard "z-buffer" algorithm by a "w-buffer" algorithm for handling occlusion, and replacing the standard "scan-line" conversion method by a new "scan-plane" conversion method. Furthermore, I implemented a new "thickening" technique that made it possible to illuminate *surfaces* correctly in four dimensions. Our new techniques generate smoothly shaded, highlighted view-volume images of mathematical objects as they would appear from a

four-dimensional viewpoint. These images reveal fascinating structures of mathematical objects that could not be seen with standard 3D computer graphics techniques. As applications, we generated still images and animation sequences for mathematical objects such as the Steiner surface, the four-dimensional torus, and a knotted 2-sphere. The images of surfaces embedded in 4D that have been generated using our methods are unique in the history of mathematical visualization.

Finally, I adapted these techniques to visualize volumetric data (3D scalar fields) generated by other scientific applications. Compared to other volume visualization techniques, this method provides a new approach that researchers can use to look at and manipulate certain classes of volume data.

Acknowledgements

I would like to thank Andrew Hanson for his continuous teaching, guidance, and support as my thesis advisor. He helped me to launch this research and provided me new ideas, useful suggestions, and important advice that I needed to complete this work. I greatly appreciate the friendship that we developed together during the course of this research.

I would like to thank Dennis Gannon for introducing me to scientific visualization, getting me started working seriously in computer graphics, and making staff and computer resources at the Center for Innovative Computer Applications of Indiana University available to support my research.

I would like to thank the other members of my committee, Robert Glassey, Christopher Haynes, and Charles Livingston, for offering many useful suggestions and helping me improve the quality of this work.

My thanks also go to the staff members of CICA, including Brian Kaplan, Eric Ost, and Donald McMullen for their system and technical support. I would also like to thank Peter Shirley for his advice.

I would like to acknowledge the support that I received from NSF grant IRI-9106389.

Finally, I would like to thank my parents and my wife for their continuous encouragement and love. Their full support has made my graduate student life easier and more enjoyable.

Contents

Abstract v

Acknowledgements vii

1 Introduction 1

1.1 The Problem 1

1.2 Related Work in Mathematical Visualization 3

1.3 Approach 7

1.4 Thesis Overview 10

2 Interactive Mathematical Visualization Techniques 12

2.1 Homogeneous Equations in $CP2$ 13

2.2 Interactive Visualization Tools 17

2.2.1 Projections 17

2.2.2 Varying Model Parameters 17

2.2.3 Surface Evolution and Surface Sweeping 19

2.2.4 Cutaway Surface 20

2.2.5 Transparency 21

2.2.6 Three-dimensional Transformations 23

2.2.7 Four-dimensional Rotations 23

2.2.8 Compactification 25

2.2.9 Color Scheme 25

2.2.10 Phase Transformations 27

2.2.11 Visibility and Invisibility 29

2.3 Topology of the Fermat Surfaces 29
    2.3.1 Euler Characteristic and Genus 29
    2.3.2 Riemann Surface Deformation 31
2.4 User Interface Design 41
    2.4.1 Direct Context-free Manipulation 41
    2.4.2 Constraint-Based Interaction 43
2.5 Summary 46

3 Four-Dimensional Rendering 49
3.1 Related Work 50
3.2 Approach 51
3.3 Mathematical Representations 57
    3.3.1 Parametric Representations 58
    3.3.2 Implicit Representations 60
    3.3.3 Parametric Representations versus Implicit Representations . 61
3.4 Normal Computation for Parametric Three-manifolds 62
3.5 Projections 63
3.6 Shadow Algorithms 64
3.7 Extending Rendering from 3D to 4D 65
    3.7.1 2D Grids versus 3D Grids 69
    3.7.2 Decomposition of a Cubic Cell into Tetrahedra 70
    3.7.3 Scan-line Algorithms versus Scan-plane Algorithms 71
    3.7.4 Z-buffer Algorithms versus W-buffer Algorithms 75
    3.7.5 Number of Triangles in 3D and 4D Rendering 78
3.8 Intensity Shading Methods 79
    3.8.1 Ambient Light 79
    3.8.2 Diffuse Reflection 80

3.8.3 Specular Reflection 82
3.8.4 Displaying the 3D View Volume 83
3.9 W-buffer Algorithms versus 4D Ray-tracing Algorithms 84
3.10 Summary 87

4 Examples of Four-Dimensional Rendering **89**
4.1 Examples of Volumes in 4D 89
4.2 Examples of Surfaces in 4D 92
4.2.1 General Approach for Surfaces 92
4.2.2 Steiner Surface 93
4.2.3 4D Torus 96
4.2.4 Knotted Sphere 100
4.3 Example of Curves in 4D 101
4.4 Example of Points in 4D 103
4.5 Improving Image Interpretability 106
4.6 A Visualization Paradigm 108
4.7 Summary 111

5 Four-Dimensional Rendering in Volume Visualization **113**
5.1 Introduction 114
5.1.1 Elevation Map Techniques 114
5.1.2 Volume Visualization Techniques 115
5.2 Extending 3D Elevation Map Methods to 4D 117
5.2.1 Contour Curves versus Contour Surfaces 117
5.2.2 Grid Lines versus Grid Planes 118
5.2.3 3D Pseudocolor Contours versus 4D Pseudocolor Contours . . 119
5.2.4 3D Oblique Pseudocolor Contour versus 4D Oblique Pseudocolor Contour 121

5.2.5 3D Grid with Hidden Surface Removed versus 4D Mesh with Hidden Volume Removed 125

5.3 Using 4D Illumination on a 3D Scalar Field 125

5.4 Examples of Applications 130

5.5 Summary 131

6 Thesis Summary **134**

6.1 Summary of Results 134

6.2 Future Directions 137

A Computer Graphics Tools **140**

A.1 Available Standard Tools 140

A.1.1 Doré - Dynamic Object Rendering Environment 140

A.1.2 Mathematica 143

A.1.3 Wavefront - Advanced Visualizer 146

A.1.4 AVS - Application Visualization System 149

A.2 Developed Tools: Object-Oriented Graphics Class Library 153

B Basic Four-dimensional Geometry **160**

B.1 Four-dimensional Transformations 160

B.2 Four-dimensional Vector Operations 162

C Mathematics Background **164**

D Examples of Implementation Methods **167**

D.1 Fermat Surface Display 167

D.2 Thickening the Steiner Surface 170

Curriculum Vitae **186**

List of Figures

1 Sphere eversion. 5

2 The $n = 3$ Fermat surface projected from 4D to 3D. 18

3 The $n = 4$ Fermat surface projected from 4D to 3D. 18

4 The $n = 3$ Fermat surface with different parameters. 19

5 The $n = 4$ Fermat surface with different parameters. 20

6 The swept $n = 4$ Fermat surface. 21

7 The cutaway $n = 4$ Fermat surface. 22

8 The transparent $n = 3$ Fermat surface. 22

9 Three-dimensional rotation of $n = 3$ Fermat surface. 23

10 Four-dimensional rotation of Steiner surface. 24

11 The $n = 3$ closed Fermat surface 26

12 The $n = 4$ closed Fermat surface 26

13 Two views of the $n = 3$ Fermat surface rendered with 3 colors. 28

14 The $n = 3$ Fermat surface is a torus. 32

15 The branch points and branch cuts of the $n = 3$ Riemann surface. . . 34

16 The Riemann sheets of the $n = 3$ Fermat surface 35

17 A schematic diagram of the $n = 3$ Fermat surface with the three pairs of fixed points removed. 37

18 A schematic diagram to illustrate the structure of the Riemann surface. 38

19 The deformation of the $n = 3$ Fermat surface to the Riemann surface. 39

20 Two views of the $n = 3$ Fermat surface with the three pairs of fixed points removed. 40

21 Constraint-based interaction techniques 47

22 Rendering in 2D worlds 53
23 Rendering in 3D worlds 54
24 Rendering in 4D worlds 55
25 Comparison of a 3D wire and a 3D tube 56
26 Thickening lower-dimensional objects in 2D worlds 57
27 Thickening lower-dimensional objects in 3D worlds 58
28 Thickening lower-dimensional objects in 4D worlds 59
29 Two 4D views of the Steiner surface with its shadow. 66
30 Two 4D views of the four torus and its shadow. 67
31 2D grid versus 3D mesh 70
32 Decomposition of a cubic cell into five tetrahedra. 72
33 Decomposition of a tetrahedron into five tetrahedra. 73
34 Scan lines versus scan planes 74
35 The transition from a hypersphere to a hypercube 91
36 A 4D rotation of a hypercube 92
37 Two cross-eyed stereo pairs of Steiner surface 97
38 Two cross-eyed stereo pairs of four-torus 99
39 A cross-eyed stereo view of knotted 2-sphere 102
40 Curves in 4D 104
41 Stereographic image of points in 4D and 3D 105
42 Two contour plots of a 3D hemisphere 119
43 Contour surfaces of a 4D hemisphere 120
44 4D Volume Grids of a 4D hemisphere 120
45 Three-dimensional contour plots of a hemisphere 122
46 Pseudocolor 4D hemisphere 123
47 4D pseudocolor grids of a 4D hemisphere 124
48 4D meshes of a 4D hemisphere with hidden volume removal 126

49 Four 3D viewpoints of a shaded hemisphere 127
50 Four 4D viewpoints of a 4D hemisphere 129
51 Volume images of binary star data 130
52 4D pseudocolor representation of the binary star density data 131
53 Volume images of the electron density field of a hydrogen molecule . . 132
54 4D pseudocolor representation of the hydrogen molecule data 133
55 A Mathematica notebook 146
56 User interface of the Wavefront Preview program. 149
57 The AVS Network Editor 153
58 User interface of the object-oriented graphics class library. 159

Chapter 1

Introduction

Visualization has played a very important role in the development of scientific thought. Drawing and physical models have been used as important visual tools for illustrating scientific ideas for a very long time. However, not until the 1970's could scientists begin using computer graphics to generate significant and interesting scientific images. With recent developments in computer graphics hardware and software, visualization has become a very powerful tool enabling scientists to explore and communicate ideas. Complicated simulations and computations can be visualized, guided, and improved interactively. Since scientists from different fields have different needs and expectations for visualization, a substantial amount of research effort is being devoted to developing tools in areas such as medical visualization, engineering visualization, and scientific visualization. Interactive mathematical visualization is the particular area that is the main focus of this research.

We discuss the problem of mathematical visualization in Section 1.1. Some important work in mathematical visualization is described in Section 1.2. We describe our research approach in Section 1.3. Finally, a thesis overview is given in Section 1.4.

1.1 The Problem

Abstract mathematical objects are extremely difficult to visualize, particularly when higher dimensions are involved. The challenge for mathematical visualization is to represent these abstract objects as images that reveal their nature to the viewer and help to foster new insights into their topological structure. Most standard computer graphics techniques can only be applied to visualizing three-dimensional objects. For mathematical objects of higher dimensions, we usually perform certain projections so that they can be represented as three-dimensional objects and rendered on a two-dimensional computer display. This does not consistently provide us with intuitively useful images of high-dimensional mathematical objects.

Human beings are creatures of three dimensions. Most of us cannot imagine what a four-dimensional object looks like. However, through computers, we can create images of these abstract objects. This is accomplished by using precisely defined 4D internal representations and a virtual 4D camera. There are many current mathematical research topics involving two complex or four real dimensions[88], three-manifolds embedded in four dimensions[96, 73], or four-manifolds[69]. In order to make this interesting mathematical world of four dimensions more understandable to many of us, we need better four-dimensional visualization tools.

Only within recent years have we acquired the power of manipulating, animating and interacting with graphical objects on a computer screen. Human-computer interaction plays an important role in the visualization process. Most current interactive visualization systems emphasize standard three-dimensional transformations; special interactive transformations that can be applied to high-dimensional mathematical objects have not been fully explored.

The purpose of this research is to develop new computer graphics methods and interactive visualization techniques that are useful for exploring geometric objects on a computer. We concentrate on subject areas such as the topology and geometry of four dimensions that provide a very challenging domain for developing visualization techniques. These tools are important because they can enhance our ability to understand abstract mathematical concepts, and can help us become more comfortable with four-dimensional intuition.

1.2 Related Work in Mathematical Visualization

Mathematicians and computer scientists have been using computer graphics to visualize mathematical objects for more than twenty years. There has been much successful work done in the area of mathematical visualization[89, 90]. Most of the previous work in this field concentrated on developing computer graphics methods and representations for visualizing mathematical objects. Many of these efforts have inspired new mathematical insights and discoveries. Some of them are excellent examples of using computer graphics to communicate and explore abstract mathematical ideas. The fascinating images, the important and often unexpected mathematical discoveries, and the computer graphics techniques used in this early research have provided motivation and important ideas for our work. Some of the important developments in this field are the following:

- **Computer Graphics in Geometric Research.** In the seventies, mathematician Thomas Banchoff and his collaborators produced computer-generated films related to high-dimensional mathematical objects such as hypercubes and hyperspheres[5, 6, 13]. Banchoff has published numerous articles[14, 12, 7, 8, 9, 11] concerning the use of computer graphics in geometric research. His recent book[10] gives a very good description of how computer graphics and drawings can be used to extend a human's abilities to visualize beyond the third dimension.

- **Sphere Eversion.** Sphere eversion refers to the process of turning a sphere inside out smoothly, without introducing any creases or pinch points in the process. Mathematician Stephen Smale proved that sphere eversion is possible in 1959[94]. French topologist Bernard Morin then succeeded in developing a sequence of three-dimensional construction plans showing how a sphere eversion could be done[43]. This eversion was beautifully rendered by Nelson Max in

1977[81] using computer graphics techniques. In 1988, John Hughes developed polynomial functions for small pieces of surfaces that covered a sphere, and through mathematical "quilting" he also succeeded in carrying out a sphere eversion[90]. Using a rendering program that I developed as part of this thesis research, I adapted Hughes' program to run on the Stardent computer at the Colloquium "Computer Graphics in Pure Mathematics" in Iowa City, May 17 – 19, 1990. Figure 1 shows two sequences of images generated during this eversion process.

- **Romboy Homotopy and the Etruscan Venus.** Apéry and Morin's remarkable discovery of a continuous map between Steiner's Roman surface and Boy's surface solved the long-standing problem of finding an explicit equation for Boy's surface[2]. The transformation from the Roman to the Boy surface is known as the "Romboy Homotopy." Mathematician George Francis worked with artist Donna Cox and computer programmer Ray Idaszak to produce a computer-animated film that shows such a smooth deformation. They also discovered new surfaces by using another distinct but similar homotopy. The video production "The Etruscan Venus" shows these results[34]. George Francis's work[42] has been influential in encouraging mathematicians to illustrate topological concepts, particularly with the help of a computer.

- **Fractals.** Fractals are mathematical descriptions of irregular shapes that occur in nature but nevertheless have sufficient regularity to be programmed into a computer using recursive or iterative functions. Benoit Mandelbrot, who initiated much of the research on fractals, discovered the *Mandelbrot Set*[78, 79, 87] in 1980; computer graphics played a very important role in visualizing this discovery. Research on fractals has depended heavily on computer experimentation, and very often it is impossible to understand the complexity of fractals

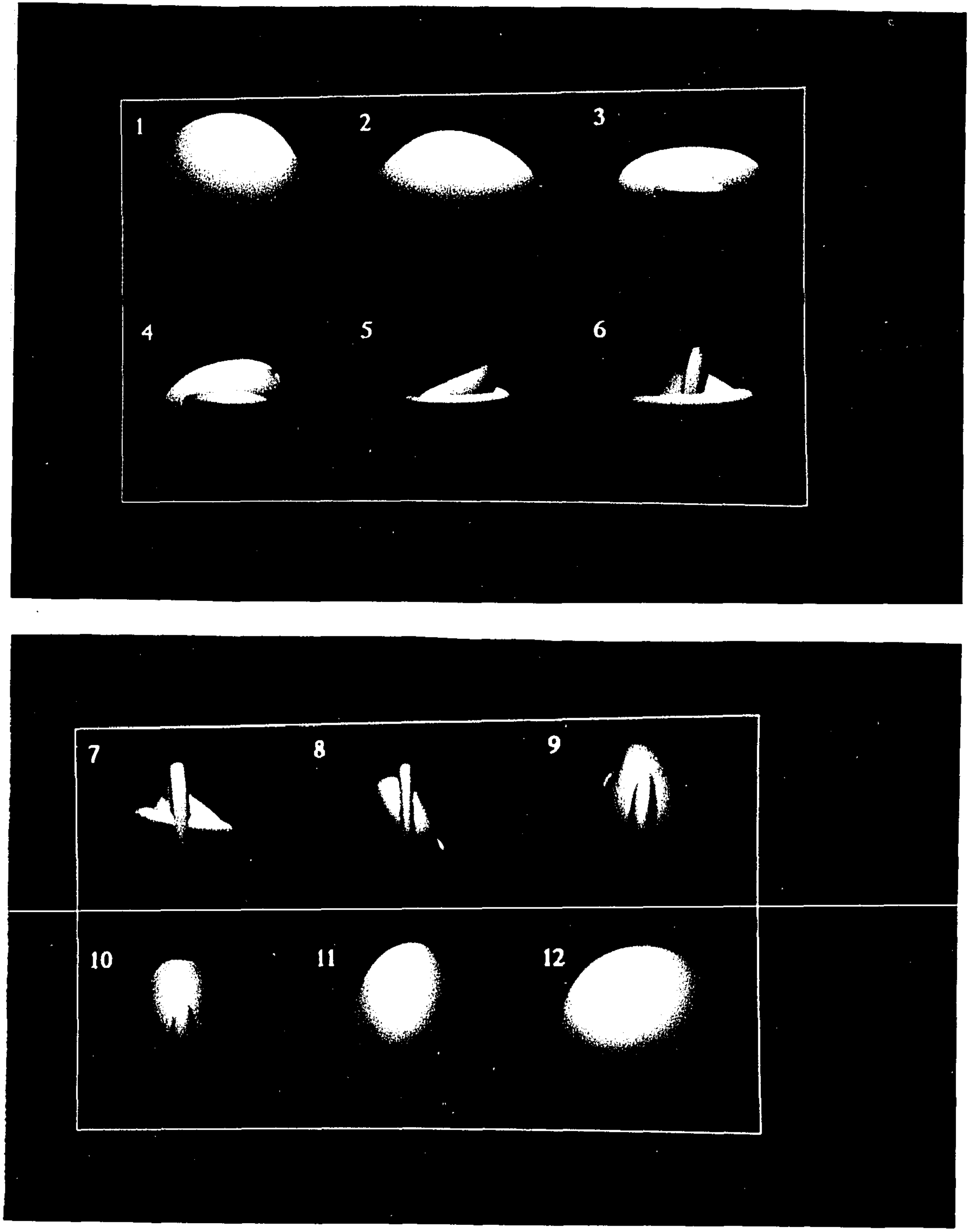


Figure 1: Sphere eversion.

without the help of computer graphics.

- **Discovery of New Classes of Minimal Surfaces.** Minimal surfaces refer to surfaces that take up the least possible area with a certain boundary constraints. David Hoffman and his collaborators used computer graphics techniques to help discover new classes of minimal surfaces[62, 26]. This work was one of the first examples of a new and unexpected mathematical discovery in which computer graphics played an important role.

- **Geometry Supercomputer Project.** Computer graphics methods for investigating knot structure, 3-manifolds, fractals, and minimal surfaces are being explored at the Geometry Supercomputer Project at the University of Minnesota[50]. This group is a major center for mathematical visualization research, and involves dozens of associated mathematicians around the country.

  One of their recent video productions, "Not Knot," provides viewers with a very interesting guided tour into a computer animated hyperbolic space[49].

- **Computer Animated Videos.** James Blinn's videotapes for the *Mechanical Universe* and *Mathematics!* projects[21, 20] are excellent examples of how to communicate mathematical concepts using computer graphics. Some more advanced mathematical and physical concepts are explored in an experimental videotape "Spatial Intuition" by Andrew Hanson[53].

- **Recent Work.** While at Brown University, Scoot Draves developed *fnord*, a system for visualizing mathematical objects using wire-frame techniques. In this system, mathematical equations can be specified in symbolic form, compiled into an executable program, and manipulated interactively[35].

  Steven Hollasch from Arizona State University wrote his master's thesis on four-space visualization of four-dimensional objects. Besides using wire-frame representations for interactive display of mathematical objects such as the hypercube, he also implemented a four-dimensional ray tracer to render hyperspheres, hypertetrahedra, hyperplanes, and hyperparallelepipeds[64].

David Banks from the University of North Carolina at Chapel Hill developed *Fourphront*, an interactive system for exploring mathematical surfaces in four-space. Banks demonstrated the system running on Pixel-Planes 5 machines during the ACM computer graphics conference Siggraph '91[15].

Creation of animated sequences of images has been a primary theme in mathematical visualization. In this research, we emphasized exploring and discovering new computer graphics techniques and algorithms needed to explore the complicated mathematical world in an interactive manner. We have attempted to develop generalizable interactive research tools and to explore prototypical interactive techniques using computer graphics.

1.3 Approach

To achieve our goal of developing new computer graphics methods and interactive techniques for visualizing topological objects on a computer, we have taken the following approach:

- **Exploring current computer graphics techniques and software.** Intelligent use of available computer graphics techniques and software can speed up the progress of research on topological visualization. We have explored and utilized computer graphics software such as *Doré*, *AVS*, *Wavefront*, and *Mathematica* heavily throughout this research. More time can be devoted to higher-level problems when we use such software tools.

Doré provides powerful 3D graphics functions that can be called from either C or Fortran programs. *Mathematica* is excellent for handling complex mathematics and symbolic computations needed in this research. Exciting animations and visual effects can be achieved using the *Wavefront Advanced Visualizer*, while *AVS* provides us some of the most latest visualization techniques in a user-friendly and ready-to-run manner. The utilization of these standard graphics software systems in this research is demonstrated and evaluated in Appendix A.

A substantial amount of effort in software development is unavoidable due to many special requirements of higher-dimensional representation problems. We built an *object-oriented graphics class library* on top of *Doré*, *C++*, and the *X-window system.* Using this graphics class library, we were able to implement an interactive visualization systems for manipulating and exploring abstract mathematical objects graphically. More details of this class library are also given in Appendix A. In Chapter 2, we present the interactive mathematical visualization tools that we developed for this research based on the class library.

- **Development of interactive visualization techniques.** It would have been impossible to carry out this research a few years ago. Powerful graphics machines such as the *KPC Titan* computers and the *Silicon Graphics* computers have put many new opportunities within reach. However, making very fast, hardware-specific implementations is not the primary intent of this research. With the speed enhancements to computer hardware currently being developed, performance will eventually take care of itself. It is more important to design prototypical interactive techniques and representation strategies that will still be useful when the present generation of graphics hardware is obsolete.

  In this research, we attempted to apply existing three-dimensional computer graphics software and techniques to visualize four-dimensional mathematical objects in an interactive manner. Experiments related to user interface design techniques such as direct context-free object manipulation and constraint-based interaction have been carried out. Various interactive techniques such as four-dimensional rotations, surface evolution, and cutaway surface display have been implemented in our visualization system. We also developed tools for performing special mathematical transformations such as phase transformations and deformations of four-dimensional mathematical objects. These techniques helped us to study complicated mathematical concepts such as those found in the conjecture known as *Fermat's Last Theorem*. Animations of computer images generated by these techniques have been recorded on videotape[60].

- **Development of four-dimensional rendering techniques.** Most of the standard rendering techniques available are only useful for rendering objects in a three-dimensional world. Special techniques are required to effectively exploit computer graphics for the visualization of concepts that are important in abstract mathematics. While some of the current three-dimensional computer graphics techniques can be extended to four or even higher dimensions, it is clear that many new techniques need to be explored and developed.

  In this research, we extended standard three-dimensional lighting and shading methods to work in the fourth dimension using virtual four-dimensional lights and a simulated four-dimensional camera. This also involved replacing the standard "Z-buffer" algorithm by a "W-buffer" algorithm for handling occlusion, and replacing the standard "scan-line" conversion method by a new "scan-plane" conversion method. The problem of ill-defined normals in the rendering of four-dimensional points, lines, and surfaces is solved by a thickening approach that generates well-defined three-manifolds with simple 4D normals.

These new techniques generate images of mathematical objects that are smoothly shaded and highlighted based on their four-dimensional geometrical structure and lighting. These images reveal characteristics of mathematical objects that cannot be seen with standard three-dimensional computer graphics techniques. Animating stereo pairs of such images with different four-dimensional rotation angles provides viewers even better insights into their topological properties. We have recorded some fascinating animations of such images on videotape[57, 56]

- **Generalization of four-dimensional computer graphics tools.** The four-dimensional computer graphics tools we developed for topological visualization have been adapted to visualize volumetric data generated from other scientific applications. By systematically generalizing a family of techniques for viewing 2D scalar fields, we developed a new set of techniques for viewing 3D scalar fields as four-dimensional elevation maps. Compared to other volume visualization techniques, our approaches provide a new way for researchers to look at volume data.

1.4 Thesis Overview

In Chapter 2, we describe the interactive visualization techniques we developed and used in this research. The domain used to test these techniques was the problem of visualizing the homogeneous equations in $CP2$. These equations were represented as surfaces in $C2$. We show how to make meaningful computer graphics images of these surfaces using a parametric form for the solutions. Using this parametric equation, we show how standard three-dimensional computer graphics techniques can be used to study, analyze, and discover unique structures of this surface. We present interactive visualization techniques that were developed for four-dimensional rotations, phase-transformations, surface evolution, surface compactification, and surface deformations. Experiments related to user-interface design techniques such as direct context-free manipulation and constraint-based interaction are described.

In Chapter 3, we concentrate on the theoretical ideas of creating interpretable lighting and shading effects for four-dimensional mathematical objects. Generation of shadows of four-dimensional objects is discussed. New methods such as scan-plane conversion, the w-buffer algorithm and systematic approaches for thickening four-dimensional points, lines, and surfaces so that they acquire unique normals in four-space are introduced. Following the introduction of four-dimensional light sources and four-dimensional virtual cameras, standard computer graphics methods such as Gouraud shading and Phong shading are extended and applied to generate smoothly-shaded and highlighted images of four-dimensional objects.

Chapter 4 consists of examples of four-dimensional rendering that we have explored. These include points, curves, surfaces, and volumes embedded in four dimensions. Computer images of the hypercube, the hypersphere, the Steiner surface, the four-dimensional torus, a four-dimensional knotted 2-sphere, and others are shown. Methods that can be used to improve the image interpretability are discussed. We also propose a general visualization paradigm for problems such as the representation of four-dimensional mathematical objects.

In Chapter 5, we present an adaptation of our rendering methods that can be used to visualize volumetric data generated in other scientific applications. The concepts and techniques of three-dimensional plots of 2D scalar fields are introduced and generalized to 4D plots of 3D scalar fields. Examples of applications to real 3D scalar field data are given.

Finally, Chapter 6 presents a summary of our research along with suggestions for future research directions.

Chapter 2

Interactive Mathematical Visualization Techniques

To achieve our goal of exploring nontrivial interactive mathematical visualization techniques, we have focussed on the problem of representing and manipulating four-dimensional mathematical objects. Most of the previous work in visualizing four-dimensional mathematical objects has been done by projecting the four-dimensional objects into three-dimensional space, and using available standard three-dimensional computer graphics techniques to render the projected objects. We started this research with the same approach, and describe in this chapter a family of interactive techniques that we developed for visualizing abstract mathematical objects in this manner. Using our object-oriented graphics class library, described in detail in Appendix A, we implemented a visualization system that runs on KPC Titan computers. These machines are powerful enough for us to achieve almost real-time interaction on the graphics display.

Throughout this chapter, we concentrate on the problems of representing and understanding intuitively a particular family of surfaces embedded in 4 Euclidean or 2 complex dimensions. These surfaces correspond to complex homogeneous equations in $CP2$ and are complex extensions of the equations of the conjecture known as "Fermat's Last Theorem." We introduce the equations in Section 2.1. The visualization techniques developed in Section 2.2 are generalizable and can be used to visualize many other complicated mathematical surfaces. In Section 2.3, we discuss the topological explorations that we have carried out while attempting to understand these surfaces. Techniques related to user-interface design such as direct context-free object manipulation and constraint-based interaction are described in Section 2.4. Section 2.5 provides a summary of this chapter.

2.1 Homogeneous Equations in $CP2$

Our interest in complex surfaces actually began as an effort to make computer graphics images that would help in understanding the mathematical conjecture known as "Fermat's Last Theorem." In 1637, mathematician Pierre de Fermat stated in the margin of his copy of *Arithmetica* by the Greek number theorist Diophantus of Alexandria that

$$x^n + y^n = z^n \tag{1}$$

cannot be solved with positive integers

$$(x, y, z)$$

for any integer power $n > 2$,

He wrote in addition that

> I have discovered a truly marvelous proof that this margin is too small to contain.

This proposition is one of the best-known unsolved problems in mathematics; no one has yet succeeded in proving or disproving it.

A research group consisting of myself, Andrew Hanson and Brian Kaplan began work in the Spring of 1990 on a computer animation attempting to show various fundamental features of Eq. (1), which we will hereafter refer to as "Fermat's equation." In the course of this work, a number of mathematicians suggested to us that the complex extension of Fermat's equation would have many interesting features if we could make pictures of it. This led directly to the visualization applications described in this chapter, as well as a widely-shown videotape "Visualizing Fermat's Last Theorem"[60] and a corresponding paper[59] on our experiences and results.

Fermat's Equation, Homogeneous Equations in $CP2$ and Surfaces in $C2$ The complex extension of Fermat's equation is a homogeneous equation in $CP2$:

$$z_1^n + z_2^n = z_0^n, \tag{2}$$

where z_0, z_1, and z_2 are complex variables. This equation denotes a surface in the complex projective space $CP2$, which contains a complex line at ∞ at $z_0 = 0$. The complement of that line defines a surface in $C2$ (two dimensional complex Euclidean space) representable as

$$z_1^n + z_2^n = 1. \tag{3}$$

We can represent the Eq. (2) alternatively in each of the three regular coordinate $C2$ patches of $CP2 : z_0 \neq 0$, $z_1 \neq 0$, and $z_2 \neq 0$. Eq. (3) is a single complex equation in two complex variables; therefore it is a surface in 4D. The definition of $CP2$ is given in Appendix C.

Explicit parametric solutions of the equation. The following arguments show how to obtain a parametric solution for Eq. (3) in a particular local coordinate system in $C2$. This parameterization is an obvious extension of methods used for superquadric modeling in computer graphics[16], but we have discovered no previous use of the method for representing such mathematical surfaces.

First, we observe that the expressions

$$\begin{aligned} u_1(a,b) &= \frac{1}{2}\left(\exp(a+ib)+\exp(-a-ib)\right) \\ &= \cos b\cosh a + i\sin b\sinh a \qquad (4) \\ u_2(a,b) &= \frac{1}{2i}\left(\exp(a+ib)-\exp(-a-ib)\right) \\ &= \sin b\cosh a - i\cos b\sinh a, \qquad (5) \end{aligned}$$

where $0 \leq b < 2\pi$ and a ranges over all real values, are complex extensions of $\cos(b)$ and $\sin(b)$, that is

$$(u_1)^2 + (u_2)^2 = 1, \qquad (6)$$

for all a,b.

Now, we can define a parameterized complex surface as follows:

$$\begin{aligned} z_1(a,b) &= s_1(k_1,n)(u_1)^{2/n} \qquad (7) \\ z_2(a,b) &= s_2(k_2,n)(u_2)^{2/n}, \qquad (8) \end{aligned}$$

where s_1 and s_2 are nth roots of unity of the form

$$s(k,n) = \exp(2\pi ik/n) \qquad (9)$$

for integers $0 \leq k \leq (n-1)$. By substituting Eq. (7), Eq. (8) for z_1, z_2 directly into

$$(z_1(a,b))^n + (z_2(a,b))^n = 1, \qquad (10)$$

we see that Eq. (10) reduces to the identity of Eq. (6). Equation (10) is a single complex equation in two complex variables; therefore it is a surface in 4D (2 real constraints on 4 real variables). Our parameters a, b explicitly represent points on a coordinate patch of this surface.

Experimentally Finding a Way to Show the Surface. When we first realized that the parameterization (Eq. (7), Eq (8)) solved the complex Fermat equation, it was not at all clear how to draw the complete topological surface. We did not understand the relationship between the parameter ranges and the phases; the first pictures were obviously unreasonable topologically. After a great deal of experimentation with the first version of the interactive exploration system, it slowly became obvious that patches of the surface matched only if b was restricted to the $1st$ quadrant and neighboring patches were related by Eq. (9). While these equations and the patching algorithm appear obvious in retrospect, this process took several weeks and was the first major breakthrough in this project, made possible by my interactive visualization utilities. Explicit representations of this sort are not generally needed by mathematicians, so they are often not aware of them. However, for computer graphics, such explicit forms are absolutely essential.

The Patching of the Fermat Surfaces. The phase factors in z_1 and z_2 have n^2 combinations of values labeled by (k_1, k_2), and therefore Eq. (2) describes n^2 distinct quadrilateral patches in $C2$ given by $(z_1(a, b, k_1), z_2(a, b, k_2))$ for $0 \leq b \leq (\pi/2)$ and $|a| \leq a_{max}$. The complete Fermat surface is represented by piecing together these n^2 patches. The geometry of this surface as a collection of patches was never computed explicitly; from a computer graphics standpoint, we just added each patch to the graphics context as an independent object. The "sewing together" of the patches to represent a complete analytical surface was accomplished automatically by the computer graphics without our having to work it out in detail.

2.2 Interactive Visualization Tools

In the following subsections, we discuss the interactive visualization techniques that we developed and used for exploring the Fermat surfaces and other interesting surfaces.

2.2.1 Projections

Projections transform points in a coordinate system of dimension n into points in a coordinate system of dimension less than n. In our implementation, we combine any two of the four coordinates into one coordinate using a rotation angle. For example, this can be done by transforming the two imaginary components of z_1 and z_2 into a single three-dimensional value z; this $4D \Rightarrow 3D$ projection then becomes

$$\begin{aligned} x &= Re(z_1) \\ y &= Re(z_2) \\ z &= \cos\alpha\, Im(z_1) + \sin\alpha\, Im(z_2). \end{aligned} \tag{11}$$

An interactive slider is used to change the value of α. The resulting surfaces for $n = 3$ and $n = 4$ are shown in Figure 2 and Figure 3. Other rotations in four dimensions can also be applied; details on four-dimensional rotations are given in Appendix B.

2.2.2 Varying Model Parameters

Parametric equations can be used to represent mathematical surfaces. In order to render parametric surfaces on a computer, it is essential for us to know what are the appropriate limits for the variables in the parametric equations. For most classic mathematical surfaces such as the Steiner surface and the four-dimensional torus,

Figure 2: The $n = 3$ Fermat surface projected from 4D to 3D.

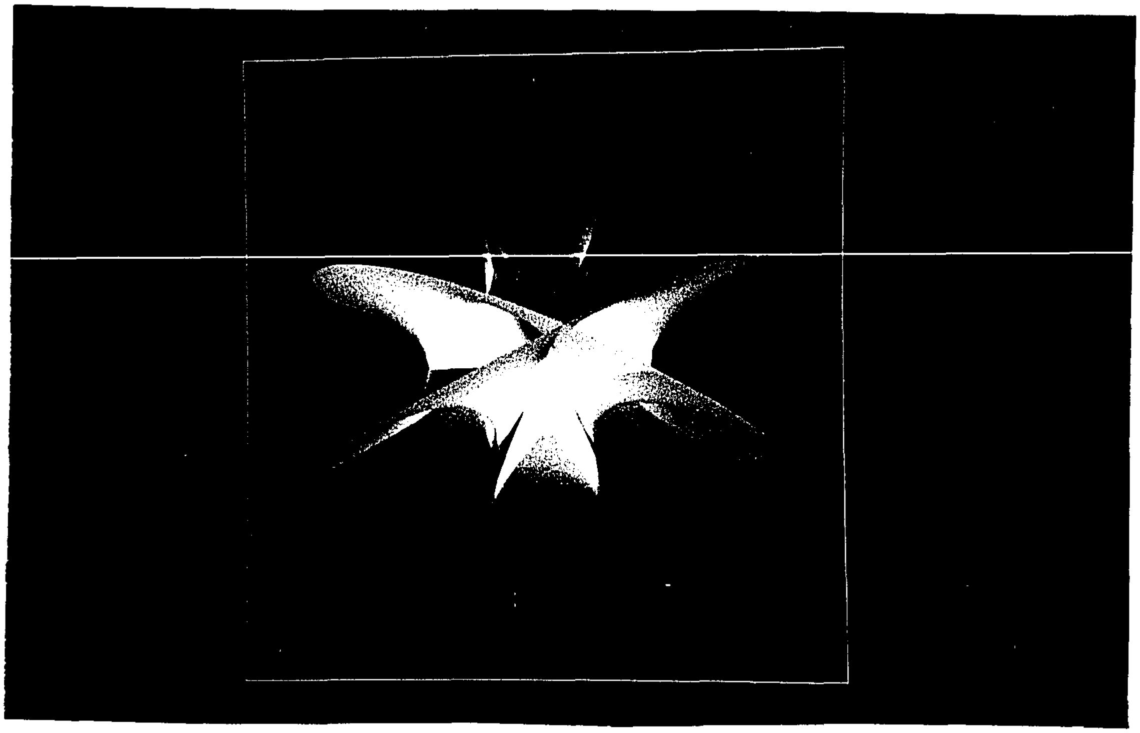


Figure 3: The $n = 4$ Fermat surface projected from 4D to 3D.

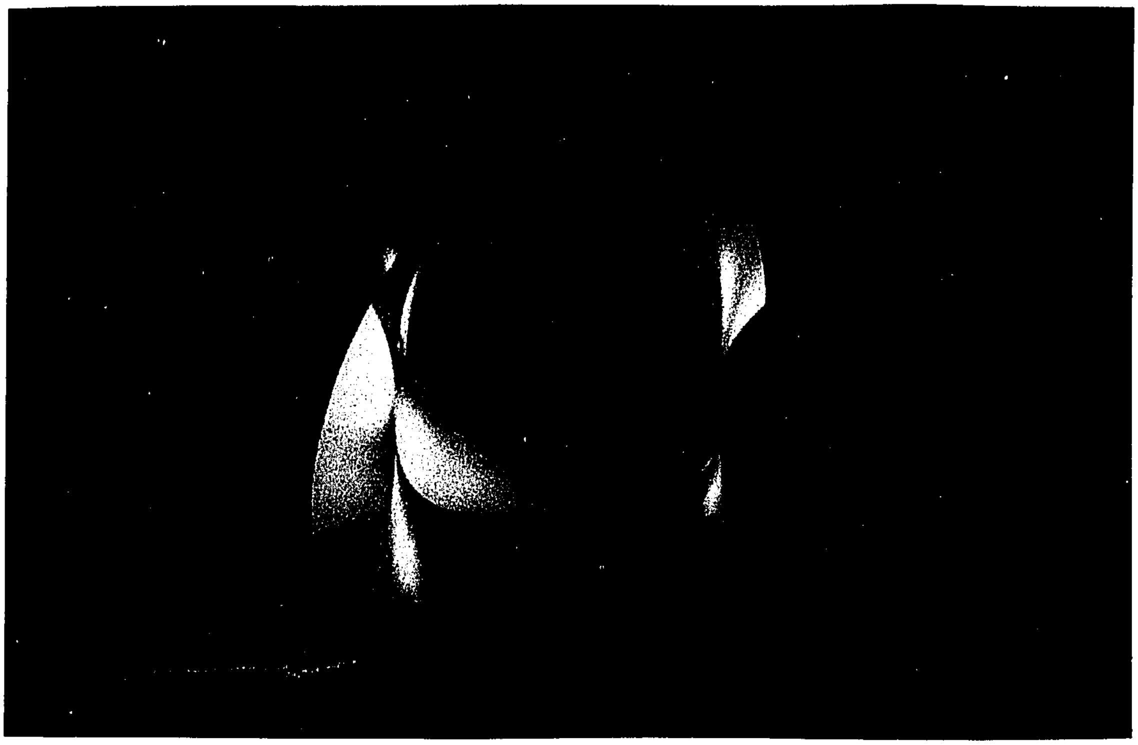


Figure 4: The $n = 3$ Fermat surface with different parameters.

the limits are either known or can be determined analytically. However, for our parameterization of the Fermat surfaces, it was not initially obvious that the range of b in the parametric equation was $[0, \pi/2]$. Interactively changing the limits of the parameters in the equations made it possible for us to discover the right limits. In many cases, when the best set of parameters is not obvious, interactively testing with different sets of parameters may be the only practical way to go about solving the problem.

Using different limits on a, we show the $n = 3$ and $n = 4$ Fermat surfaces in Figure 4 and Figure 5.

2.2.3 Surface Evolution and Surface Sweeping

Once we know the right limits to use in the parametric equations, we can watch the surface evolve by animating the images resulting from varying the bounds of

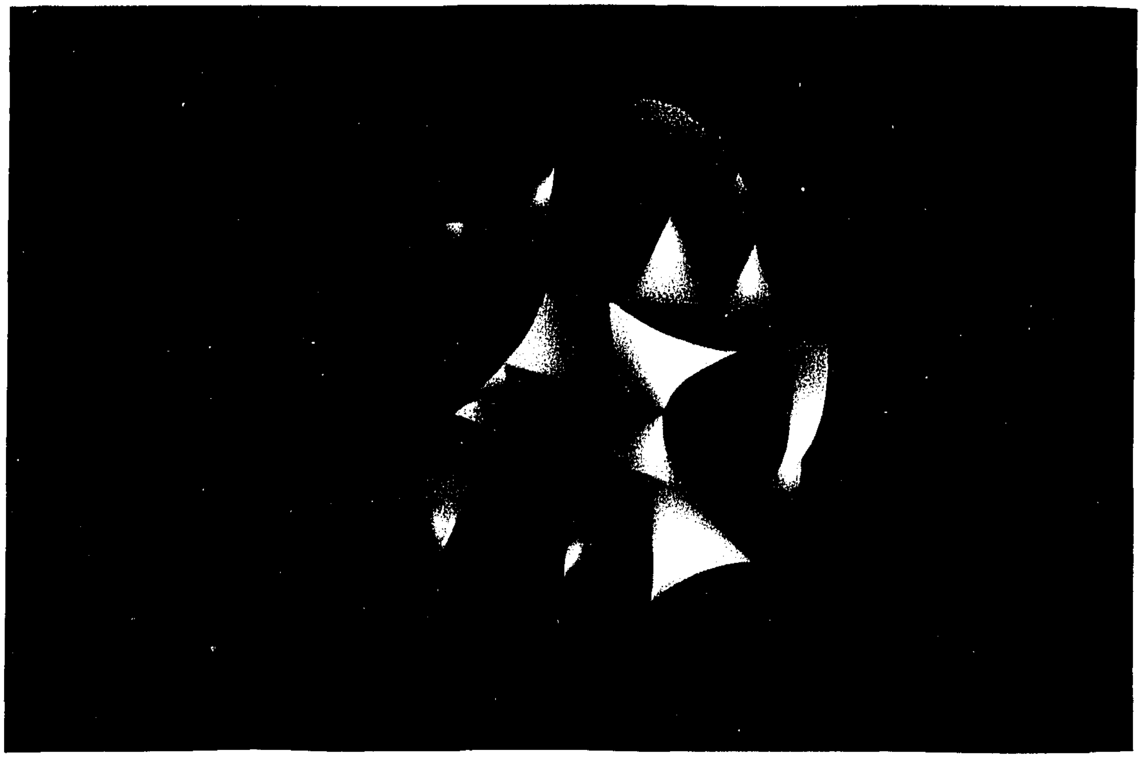


Figure 5: The $n = 4$ Fermat surface with different parameters.

the parameters used in the equations. Different ways of animating the bounds give different effects; for example, we can grow the surface from inside out, from outside in, from middle to both sides or sweep the surface using strips. In Figure 6, we show a few images of the $n = 4$ Fermat surfaces generated by sweeping from a low to a high value of the limits of $\pm a$.

2.2.4 Cutaway Surface

By cutting away the front of the mathematical object, one can expose the centrally detailed structures. This cutting can be done by adjusting the position of the front clipping plane. In our visualization system, the user can move the front clipping plane interactively using a slider. Another better approach is to use arbitrary "negative" cutaway volumes; this would have given even more flexibility. We can also divide the surface into ribbons and cut out alternate ribbons, so that one can see the interior

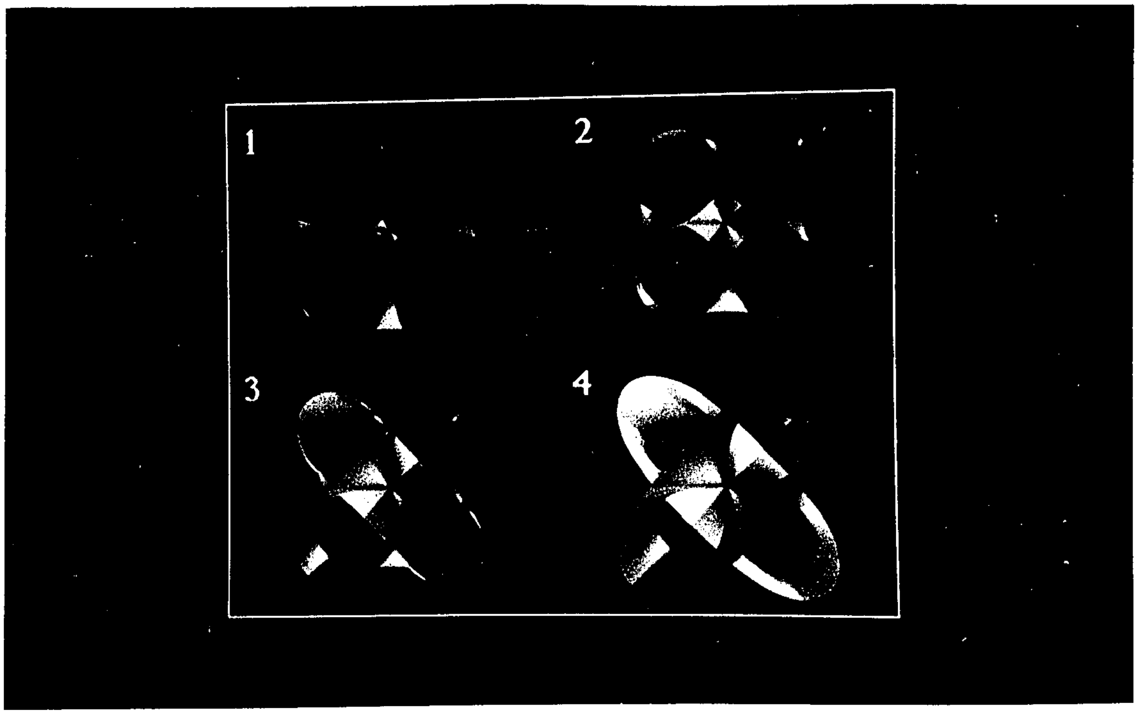


Figure 6: The swept $n = 4$ Fermat surface.

through the missing strips.

Figure 7 shows the process of using the cutaway surface technique to reveal the internal structure of the $n = 4$ Fermat surface.

2.2.5 Transparency

Transparency is another good way of showing the interior properties of an object. It can be more effective when combined with rotation and judicious choice of view angles. In Figure 8, we show an $n = 3$ Fermat surface rendered without using transparency on the left and the same surface rendered with transparency on the right.

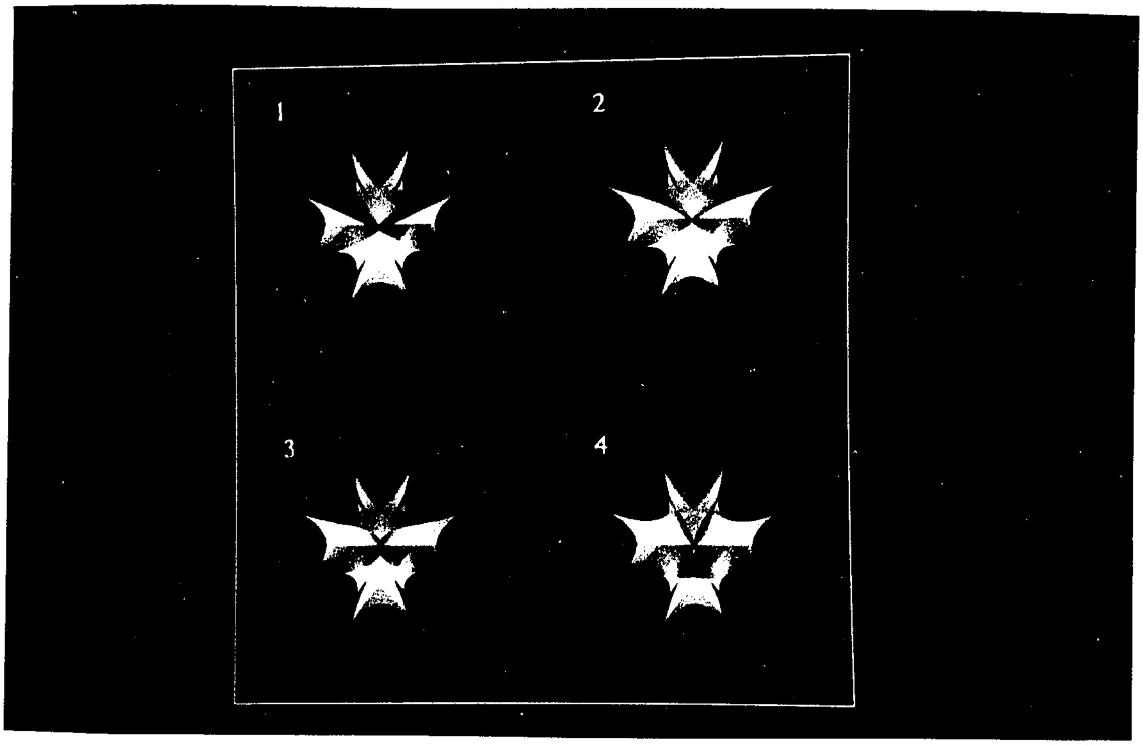


Figure 7: The cutaway $n = 4$ Fermat surface.

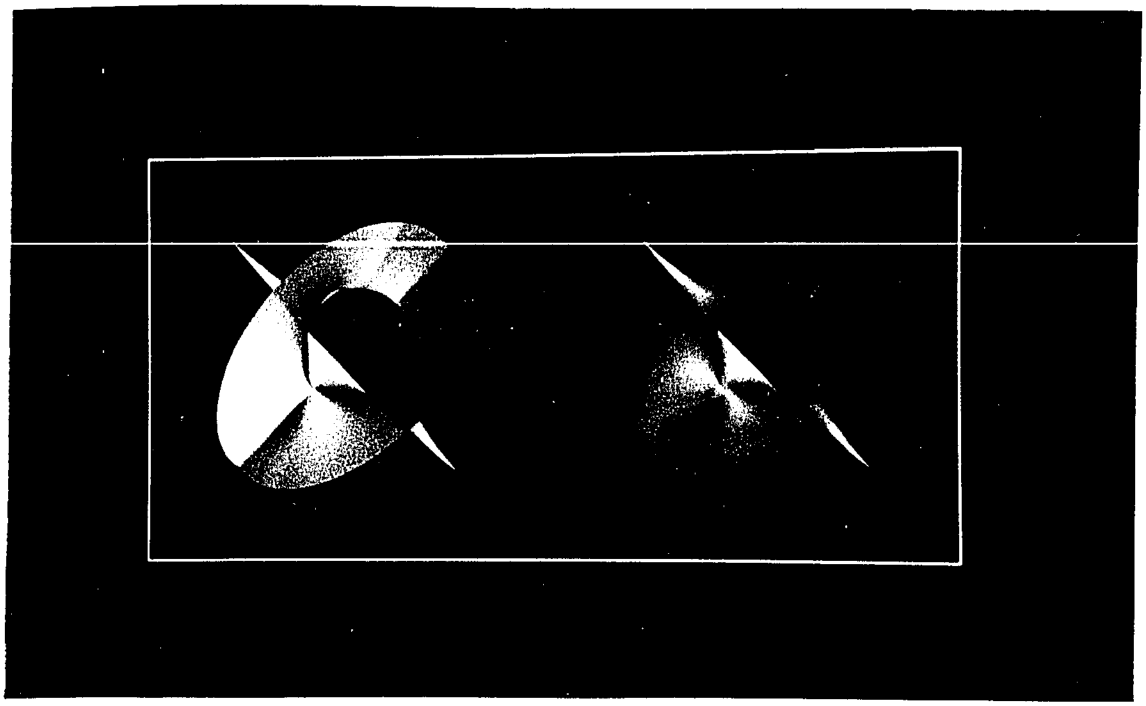


Figure 8: (Left) Rendered without transparency. (Right) Rendered with transparency.

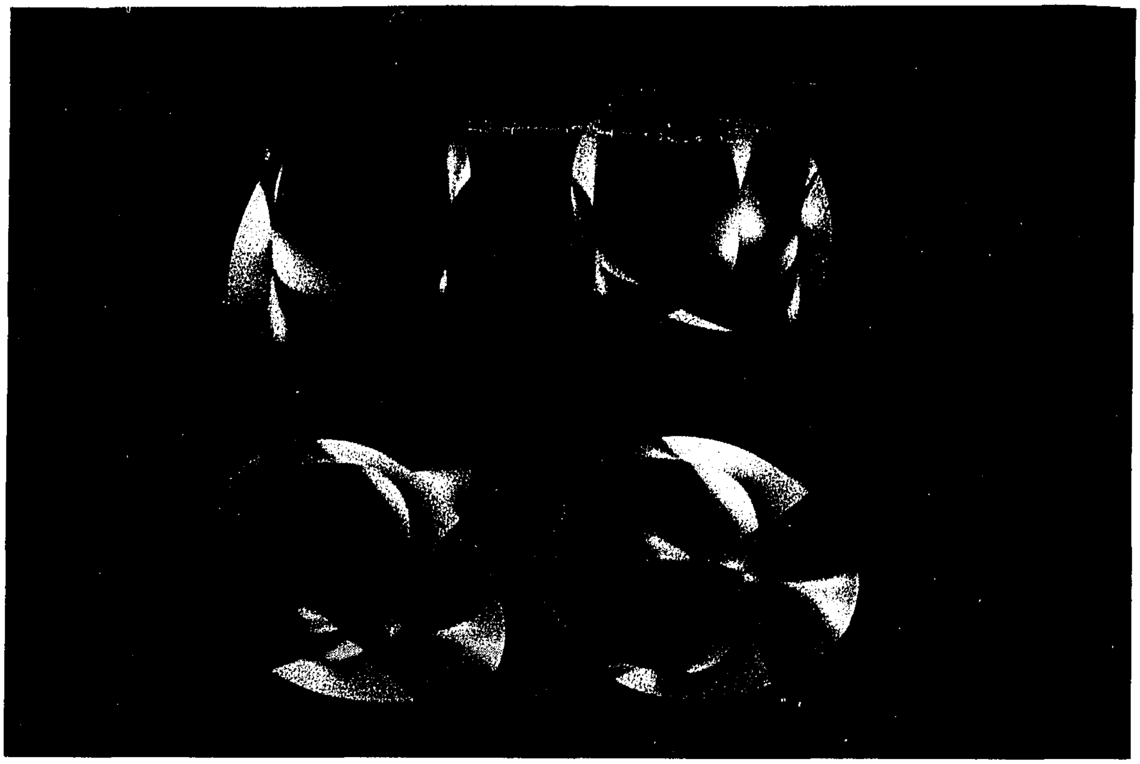


Figure 9: Three-dimensional rotation of $n = 3$ Fermat surface.

2.2.6 Three-dimensional Transformations

Users can rotate, translate, and scale graphical objects interactively using mouse motion or sliders. Through keyboard commands, users can specify the exact transformation that they would like to see. These features provide users more control and a better feeling about the objects being displayed on the computer screen.

Figure 9 shows four different 3D orientations of the same $n = 3$ Fermat surface.

2.2.7 Four-dimensional Rotations

There are three three-dimensional rotations that rotate 3D objects about the X, Y, and Z axes. In 4D, there are six four-dimensional rotations that we can use to manipulate four-dimensional objects. Rotating a mathematical object in four-dimensional space is very effective in generating additional spatial intuition about the object; much information is carried by motion parallax cues. The six basic rotation matrices

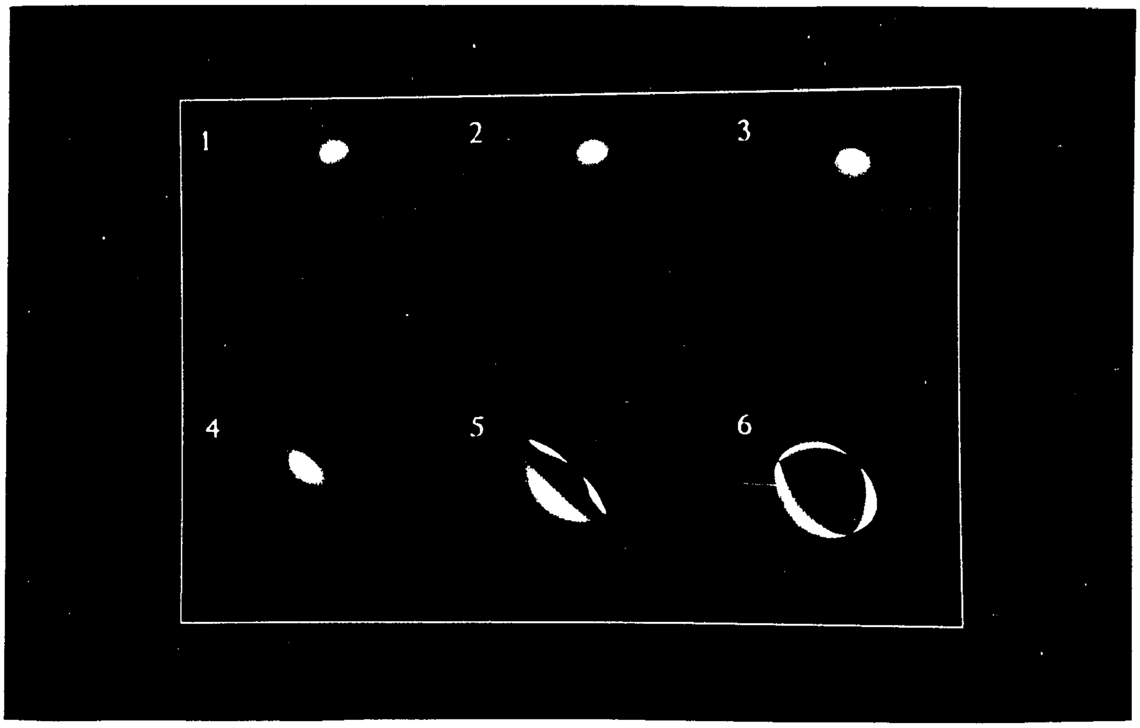


Figure 10: The cross-cap transformed into the Steiner Roman surface via XW rotation.

that are needed to carry out the four-dimensional rotations are given in Appendix B.

In our visualization system, we assigned a slider to each of the above four-dimensional rotations. The user can also start an arbitrary four-dimensional rotation animation sequence by selecting a button from the interface menu. Interesting results can be observed simply from four-dimensional rotations alone; for example, by rotating the 4D equation for the cross-cap about the XW plane, it can be transformed into Steiner Roman surface. Before becoming aware of this well-known fact, we noticed it while rotating the cross-cap using a slider in the visualization system. More details about the Steiner surface are given in Chapter 4. Figure 10 shows the cross-cap being rotated into the Roman surface.

2.2.8 Compactification

For surfaces with points at infinity, a geometric view of the global topology can be achieved by mappings that bring all the points, including those at infinity, within a sphere.

To depict the solutions to Fermat's equation as closed surfaces, we perform the following transformation:

$$\begin{aligned} v_1 = x_1/\phi, \qquad & v_2 = y_1/\phi, \\ v_3 = x_2/\phi, \quad v_4 \quad = \quad & y_2/\phi, \quad v_0 = x_0/\phi, \end{aligned} \tag{12}$$

where

$$x_1 = Re(z_1), \; y_1 = Im(z_1), x_2 = Re(z_2), \; y_2 = Im(z_2),$$
$$x_0 = ((x_1)^2 + (y_1)^2 + (x_2)^2 + (y_2)^2)/2r, \; \phi = 1 + x_0/2r.$$

Here r is a free parameter corresponding to the radius of a generalized "Riemann sphere,"

$$(v_1)^2 + (v_2)^2 + (v_3)^2 + (v_4)^2 + (v_0 - r)^2 = r^2. \tag{13}$$

Eq. (12) automatically satisfies Eq. (13) and parameterizes a 4-sphere embedded in a 5-dimensional space. Examples of the result of the 3D projection of this transformation from 5D are shown in Figure 11 and Figure 12.

2.2.9 Color Scheme

Color is a very powerful tool in visualization when it is used intelligently. We have represented the Fermat surface as a collection of n^2 patches "sewing together". There

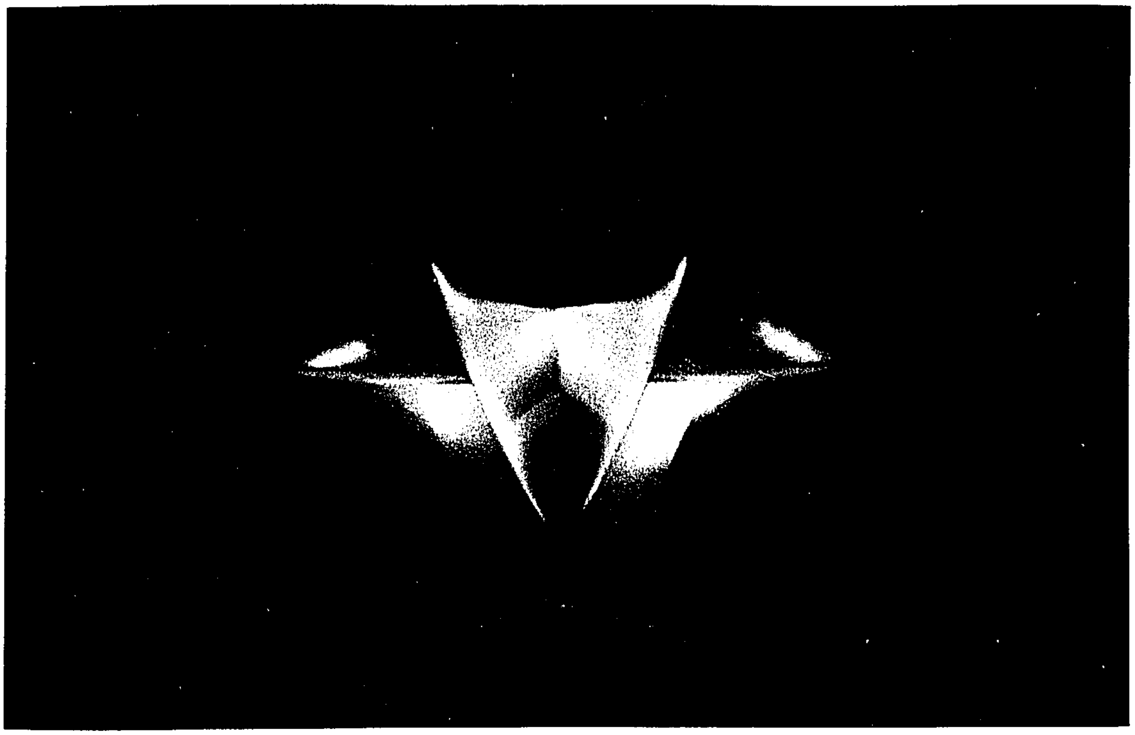


Figure 11: The $n = 3$ close Fermat surface obtained by projecting the entire infinite surface into a compact four-sphere, and then projecting that to 3D.

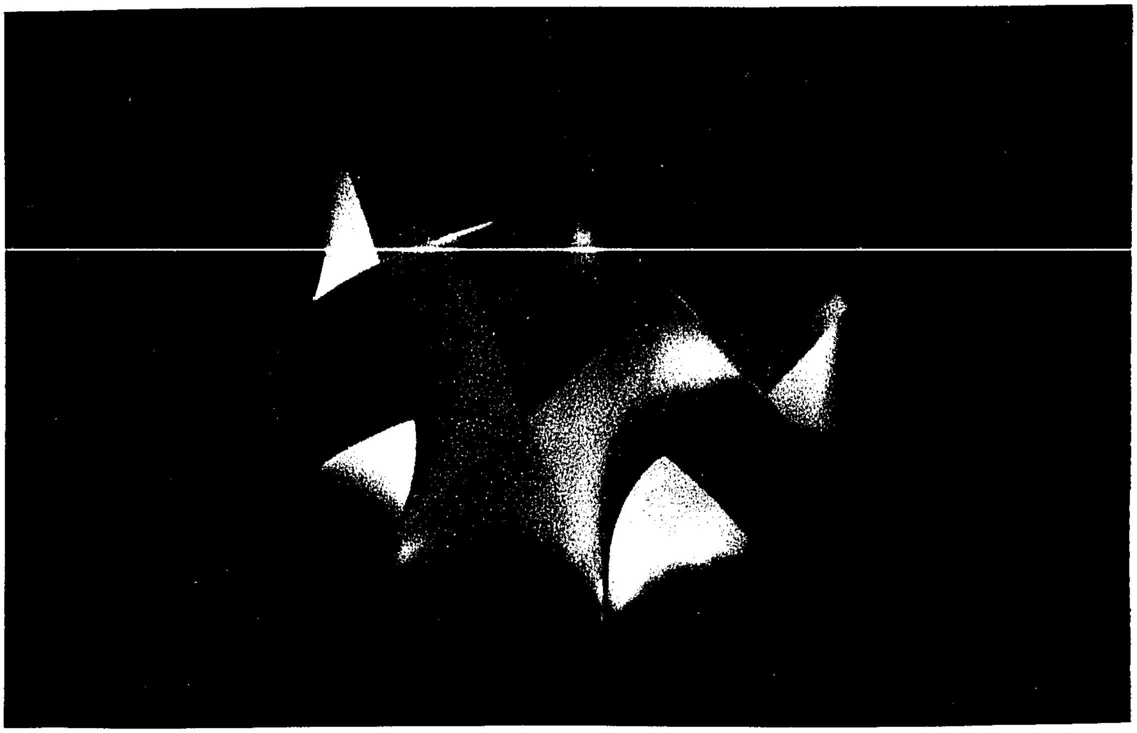


Figure 12: The $n = 4$ closed Fermat surface

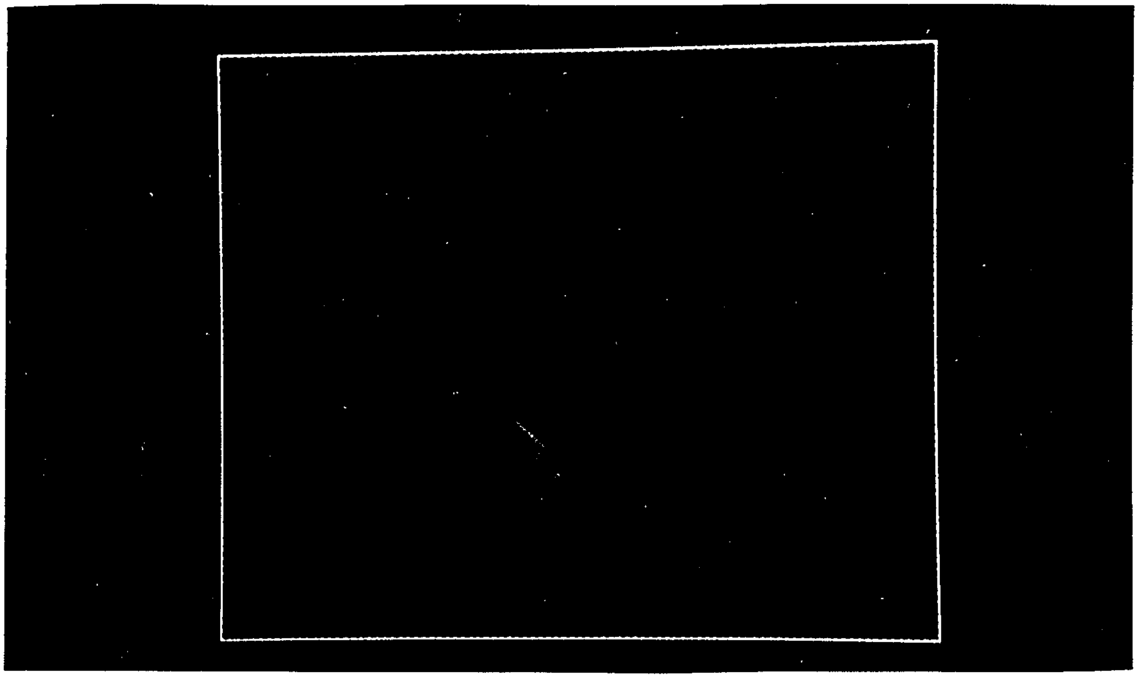
are multiple intersections, and multiple local common points for these complex surfaces. We color-coded the patches by their complex phase s_1, s_2 in Eq. (9) for visual interest, that is we used n^2 different colors for the n^2 patches. The red component and the green component of a patch color is proportional to the values of k_1 and k_2 in Eq. (9) respectively. An accidental side-effect of using this color scheme is that the fixed points of the complex phase transformations (Eq. (9)) are then distinguishable as the centers of "pie charts" with n distinctly colored patches meeting at a single point on the surface. For an example, see Figure 11. When we discovered that these n^2 patches could be divided into n groups containing patches sharing common fixed points, we colored patches within the same group using the same color. Figure 13 shows two views of the 3-color $n = 3$ Fermat surfaces. This n-color color coding was also used in the Riemann surface deformation, which transforms the Fermat surface into a standard representation of a degree n Riemann surfaces with n rings connected by $n(n-1)$ half-twisted ribbons. Details of this deformation are given later in Section 2.3.2.

2.2.10 Phase Transformations

By changing the value of ϕ_1 and ϕ_2 in

$$\begin{aligned} z_1' &= z_1 \times e^{i\phi_1} \\ z_2' &= z_2 \times e^{i\phi_2}. \end{aligned}$$

either through mouse motion or two separate sliders, we can visualize movements of patches of the Fermat surfaces in four-dimensional space. Since each patch of a Fermat surface is assigned a different color, we can see clearly how the n^2 patches of a Fermat surface are connected to each other by the above transformation.



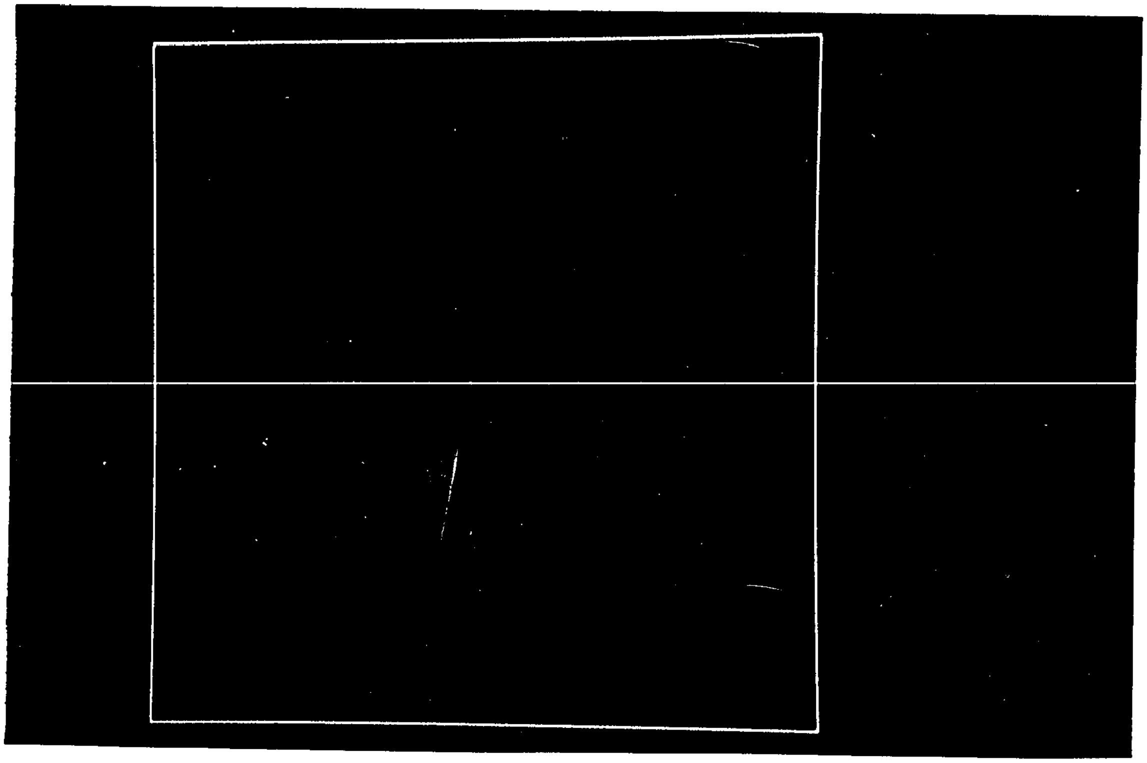


Figure 13: Two views of the $n = 3$ Fermat surface rendered with 3 colors.

2.2.11 Visibility and Invisibility

The three-dimensional projections of the n^2 patches that make up the Fermat surface intersect themselves in a very complex manner. The user must be able to make any patch or group of patches visible and invisible interactively in order to carry out topological exploration of these surfaces. By combining this capability with the other techniques such as color coding and phase transformations, we figured out that Fermat surface can be divided into n groups, and understood how they are connected to each other. This information was needed to carry out the transformation of the Fermat surface that we refer to as the *Riemann surface* deformation.

2.3 Topology of the Fermat Surfaces

Equations (7 – 10) allow us to explicitly show the geometry of the Fermat surfaces in $C2(R4)$. We next discuss a variety of mathematical and graphical tools for exploring and understanding the topology of these surfaces. First, we look at the mathematical properties of the Fermat surfaces and then we present the graphical tools that we used to study these properties.

2.3.1 Euler Characteristic and Genus

Every closed surface in two dimensions can be constructed by adding p handles and q crosscaps to a sphere. A surface is orientable if and only if it has no crosscaps. The alternating sum of the number of faces, edges and vertices of any tessellation of a surface is its *Euler characteristic* χ. For closed surfaces,

$$\chi = V - E + F = 2 - 2p - q.$$

Surfaces with ($q = 0$), such as Riemann surfaces, are specified uniquely by their genus, p, up to topological equivalence[42].

A standard way to compute the Euler characteristic of a Fermat surface is to consider its homogeneous equation Eq.(2) in CP2. When $z_0 \neq 0$, the surface (i.e., in the complement of the line $z_0 = 0$), is locally representable as n copies of $CP1 = S^2$ found by solving

$$z_0 = (z_1^n + z_2^n)^{(1/n)}. \tag{14}$$

However, at $z_0 = 0$, these n solutions degenerate into one homogeneous equation, $z_1^n + z_2^n = 0$, which has n solutions for $z = z_1/z_2$ of the form

$$z = \omega_{n,k} = s(k,n) exp(i\pi/n), \tag{15}$$

where $s(k,n)$ is given by Eq. (9) and $(\omega_{n,k})^n = -1$ are the n roots of -1 for $k = 0, \cdots, n-1$. Thus the total Euler characteristic of the surface is the sum of the Euler characteristics of the n S^2's corrected for the fact that the n vertices $\omega_{n,k}$ in each S^2 were counted n times when they should only have been counted once:

$$\begin{aligned} \chi &= V - E + F \\ &= n(\chi(S^2) - n) + n \\ &= 2n - n^2 + n \\ &= 3n - n^2 . \end{aligned} \tag{16}$$

The corresponding genus p of the surface follows from the definition $\chi = 2 - 2p$ (see, e.g. [48]):

$$p = \frac{1}{2}(2 - \chi)$$

$$
\begin{aligned}
&= \frac{1}{2}(n^2 - 3n + 2) \\
&= \frac{(n-1)(n-2)}{2} \,. \qquad (17)
\end{aligned}
$$

For each integer power n, the complete Fermat surface is a closed manifold with the topology of a 2-sphere having some number of handles p attached.

Thus $n = 2$ is a sphere ($p = 0$), $n = 3$ is a torus ($p = 1$), $n = 4$ is a 3-hole torus ($p = 3$), and so on.

After exploring the $n = 3$ surface with the above visualization tools, we constructed Figure 14 to show how the 9 patches of the $n = 3$ surface are connected to each other. Each hexagon in Figure 14 represents a patch of the $n = 3$ surface and it is labeled with the two integer numbers that are the values of k_1 and k_2 in Eq. (9). Each smaller circular node, which is labeled as $[i]$ in Figure 14, is a point where $z_i = 0$ ($i = 1, 2$). These nodes are the fixed points of phase transformations $s(k, n)$ (Eq. (9)) on z_i. Each bigger circular node, which is labeled as Inf_i in Figure 14, is a point at ∞ where the full surface would intersect the line $z_0 = 0$ in its homogeneous form.

It is clear that the left side of the highlighted parallelogram can be identified with the right side and the top side of the parallelogram can be identified with the bottom side. If we glue the left side to the right side and then glue the top side to the bottom side, the parallelogram becomes a torus. Similar explorations can be carried out to show that $n = 2$ is a sphere, $n = 4$ is a 3-hole torus, and so on.

2.3.2 Riemann Surface Deformation

Another way to study Fermat surfaces is to look at the Riemann surface defined by Eq. (2). A brief introduction to the concept of Riemann surface is given in Appendix C.

The Riemann surface of Eq. (2) is locally describable as a flat disc with n branch

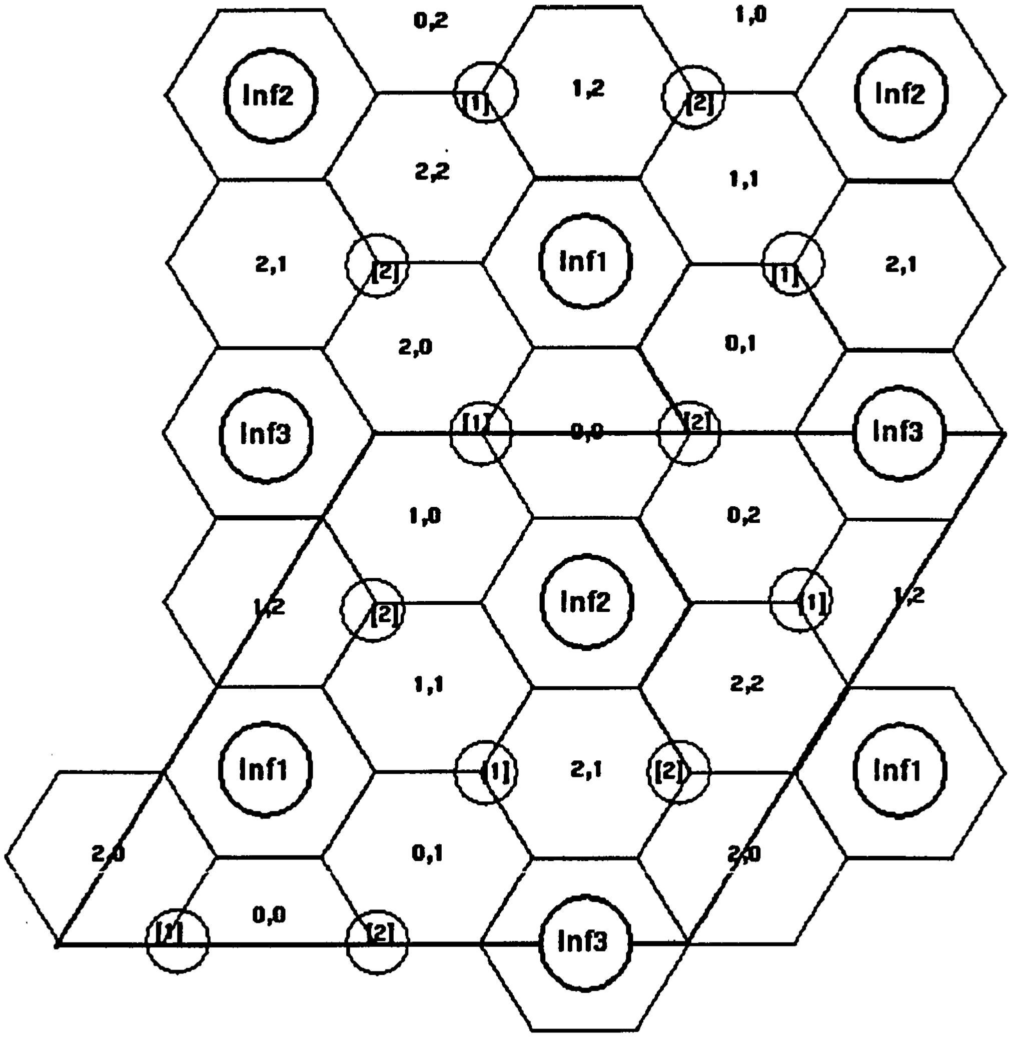
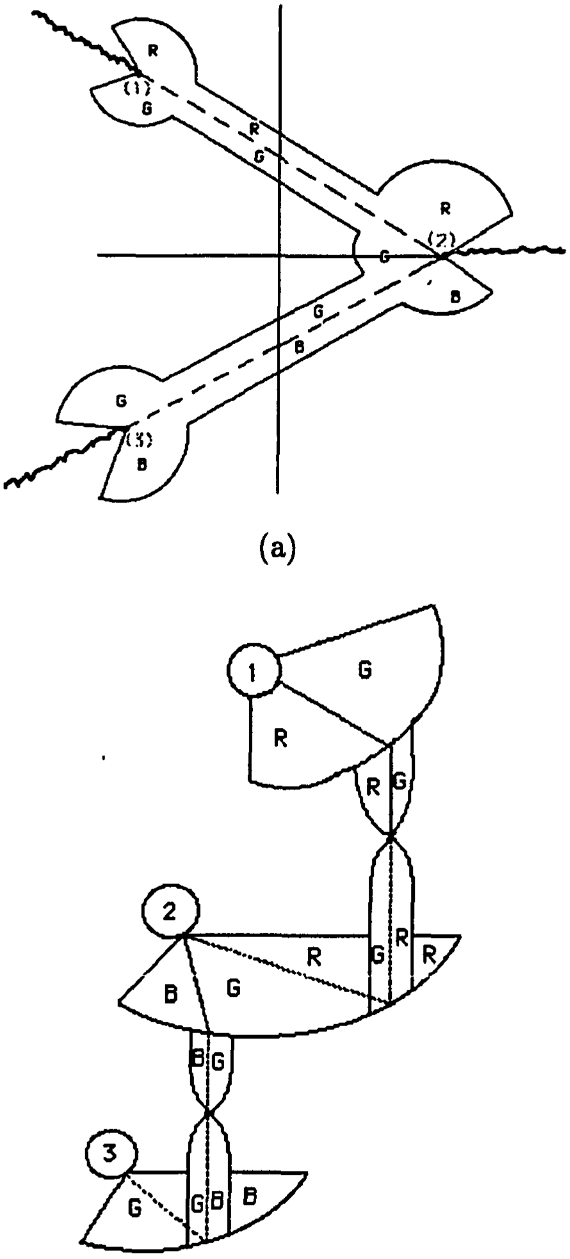


Figure 14: The $n = 3$ Fermat surface is a torus. Each smaller circular node labeled $[i]$ means a point where $z_i = 0$; it is a fixed point of phase transformations of z_i. Each bigger circular node labeled Inf_i is a point at ∞ where the full surface would intersect the line $z_0 = 0$ in its homogeneous form.

points at the roots of $z^n = (z_1/z_2)^n = -1$. These n branch points are the n points on the complex line at infinity that are approached by the n circles that are the boundaries of the surface in $C2$ when the parameter $|a| \to \infty$ in a particular inhomogeneous coordinate system (see Eqs. (4 – 10)). Let us now lay out the topology of the surface in two ways:

- **Riemann sheet approach.** First, as shown in Figure 15(a) for the special case $n = 3$, we separate the branch points from one another by $n-1$ thin ribbons. Next we stretch the ribbons out in 3D space so the branch points lie in a vertical stack; the ribbons necessarily undergo a half twist as shown in Figure 15(b) in this procedure if we keep the orientation of the branch surfaces aligned. Finally, we "close the fan" of the wedge running from either side of the branch cut to the branch point, make n copies of the resulting surface, one for each Riemann sheet, and sew them together; only the branch point itself is common to all n Riemann sheets, so the result is a stack of n disks whose centers are the n branch points, and whose layers are connected by $n-1$ sets of n half-twisted ribbons. This result is shown in Figure 16 for $n = 3$. Filling in the holes in this stack with n disks, we get a complicated, closed Riemann manifold that one can verify has the correct value $\frac{(n-1)(n-2)}{2}$ for its genus.

- **Algebraic approach.** The global properties of the manifold can be understood by considering it in terms of three sets of infinities of the homogeneous equation (Eq. (2)), one set for each of the cases $z_i = 0, i = 0, 1, 2$. Each of these lines intersects the surface at n points, and these special points correspond to the *fixed points of the cyclic group* $e^{2\pi ik/n}$ for $k = 0, 1, \cdots, (n-1)$ for that variable on the finite surface. When the corresponding variable is the denominator of the inhomogeneous equation, these points are the asymptotes at infinity of the n rings at parameter values $a = \pm a_{max}$. Viewed in this symmetric fashion,



(a)

(b)

Figure 15: (a) The three branch points of a single Riemann sheet of the $n = 3$ Fermat surface. (b) Stretching the Riemann sheet up so the branch points are aligned and connected by half-twisted ribbons.

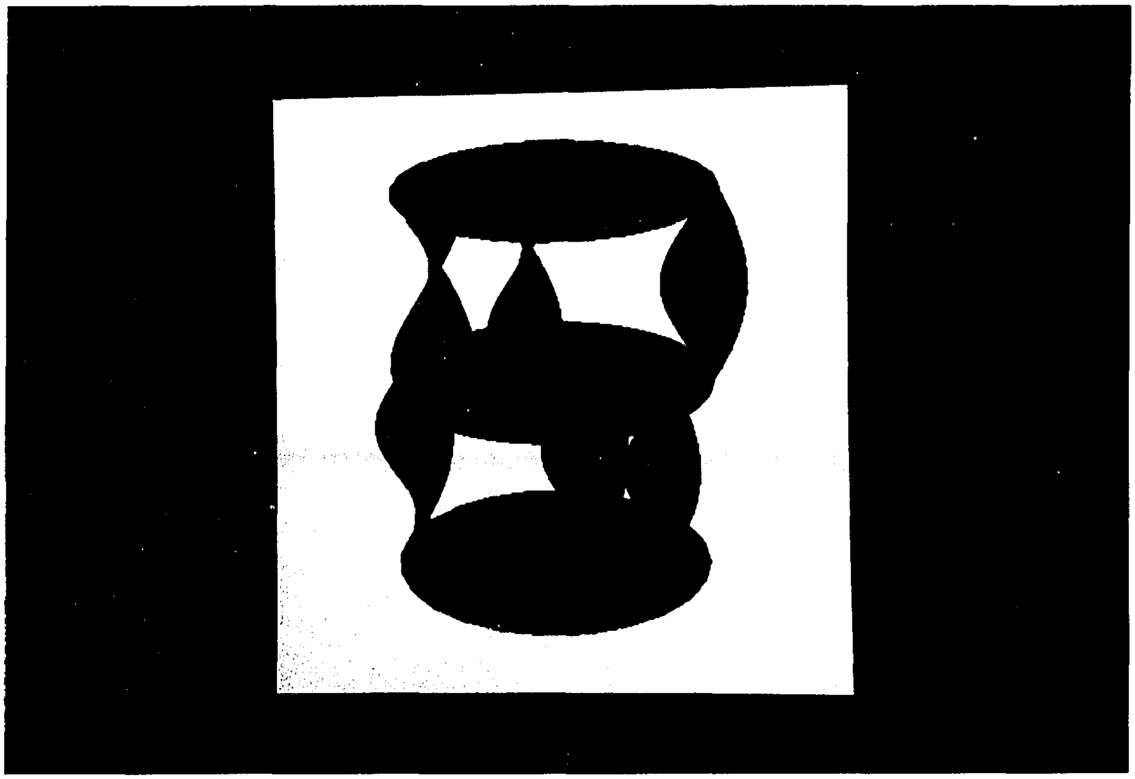


Figure 16: Joining 3 copies of the Riemann sheets together to form the complete $n = 3$ Riemann surface.

the surface is now seen to have a natural set of $3n$ points that can have holes cut out around them as shown in Figure 17. The algebraic surface is seen to correspond to a sort of "inverse" of the Riemann manifold picture, since the n branch points of the Riemann surface in a particular finite coordinate system are the points at infinity of the algebraic manifold, and the remaining $2n$ zeroes of the inhomogeneous variables on the algebraic surface are the "holes" omitted from the Riemann surface picture.

There is an explicit construction that lets us connect these two viewpoints of the manifold: First take the n^2 square patches that make up the algebraic parameterization of the equation. In each square patch, take only the ribbon around $b = \pi/4$ and connect it to the ribbons at $|a| = a_{max}$. These pieces glue together as shown in Figure 18 for $n = 3$ to form the circles approaching infinity, and also glue together to form $2n$ circles around the fixed points of the cyclic group. Comparing Figure 14 to Figure 18, we can see the explicit correspondence between the geometric surface (Figure 13 and Figure 14) and the Riemann surface (Figure 16 and Figure 18). If we leave the empty spaces around the fixed points, we have n^2 ribbons, n connecting each of the circles at infinity to its nearest neighbor. If we cut the n ribbons joining the first and the last circle, we have $n(n-1)$ remaining ribbons; if we fill disks of the circles approaching infinity, we find exactly the Riemann manifold described above.

We have carried out an animation of the deformation from the standard $n = 3$ Fermat surface to its Riemann surface using the correspondence between the geometry and the topology represented by Figure 18. Figure 19 shows several images from this animation.

Another nice way to see the explicit correspondence between Figure 14 and Figure 16 is to generate the manifold by sweeping out n punctured disks, each surrounding a *pair* of fixed points. By using n different colors for the n disks and sweeping

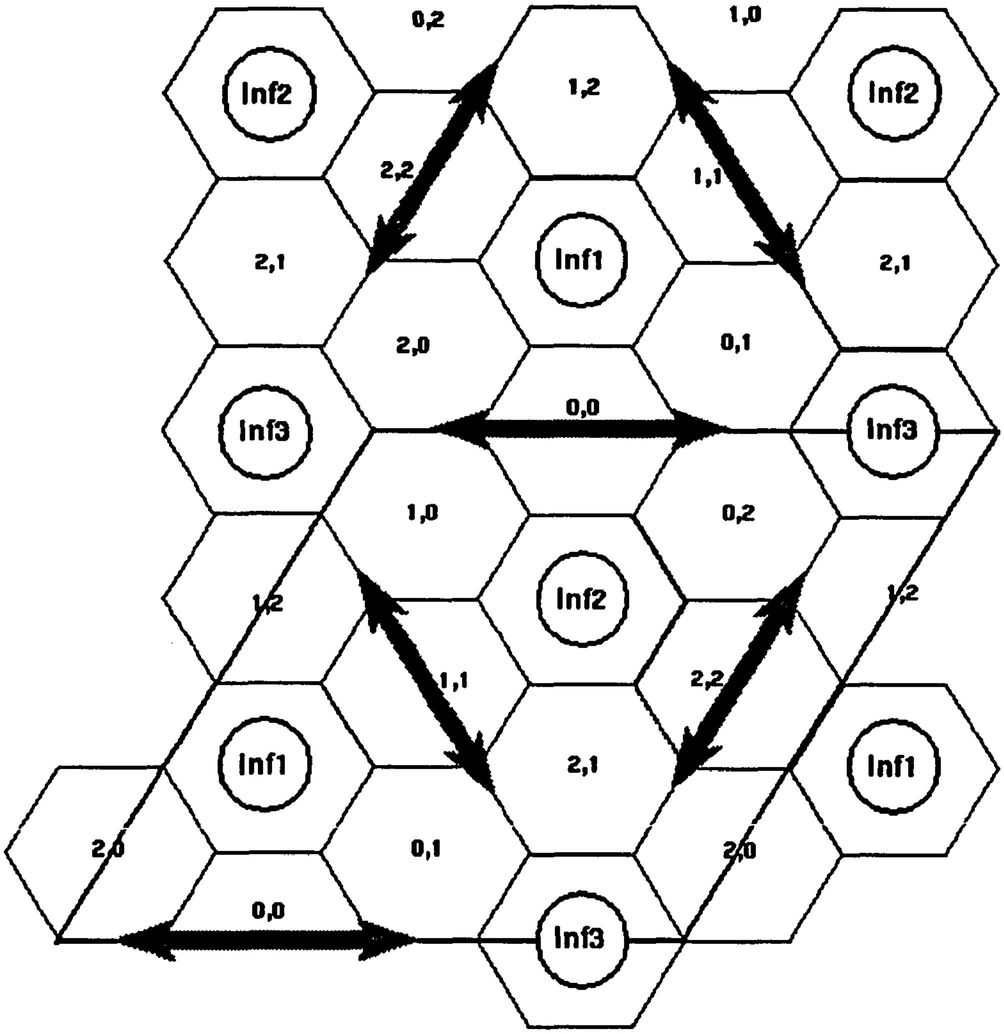


Figure 17: A schematic diagram of the $n = 3$ Fermat surface with the three pairs of fixed points removed. The double headed arrows in the figure indicate the portions of the surface that are removed.

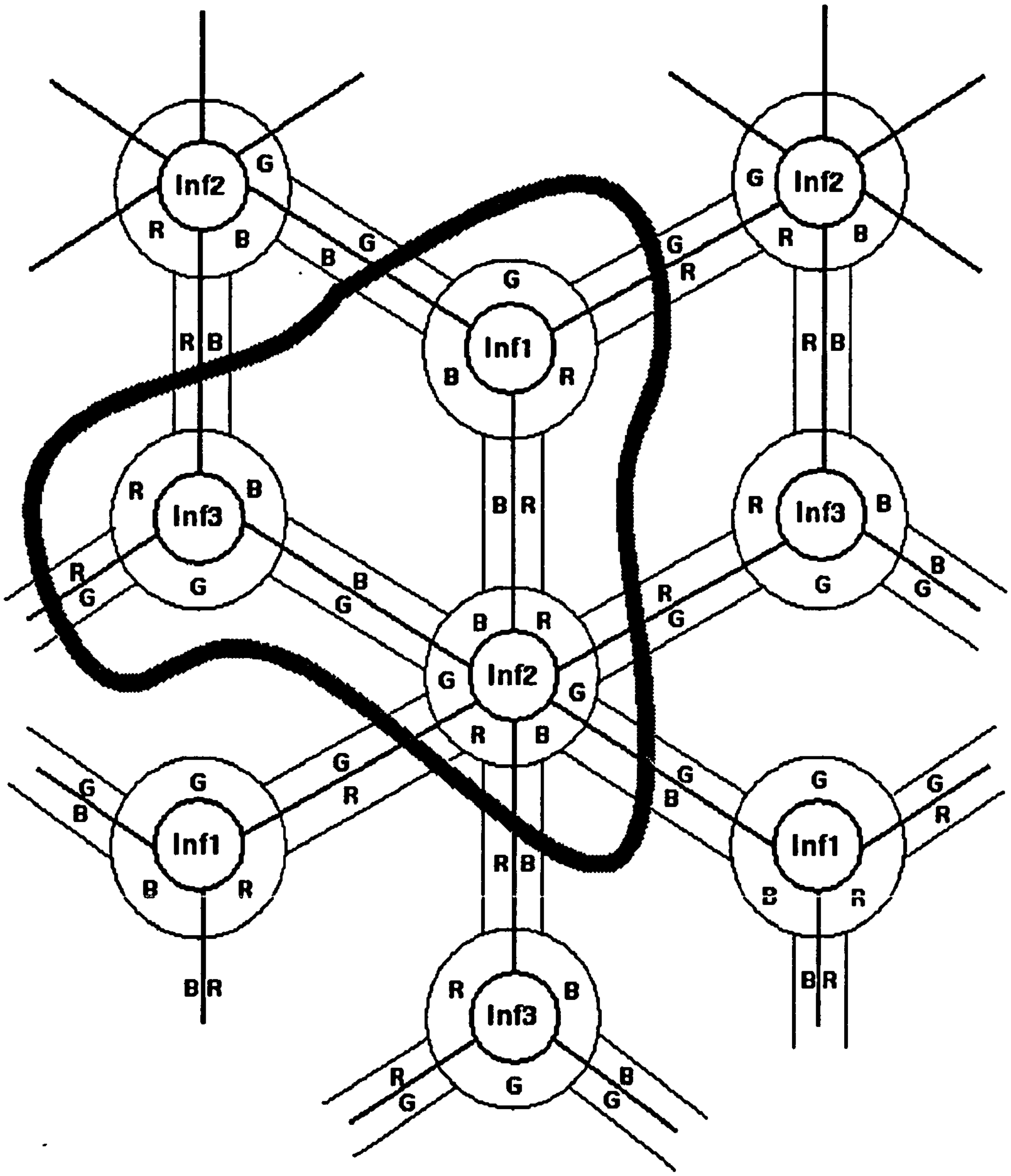


Figure 18: A schematic diagram to illustrate the structure of the Riemann surface. The highlighted path encloses a copy of the surface.

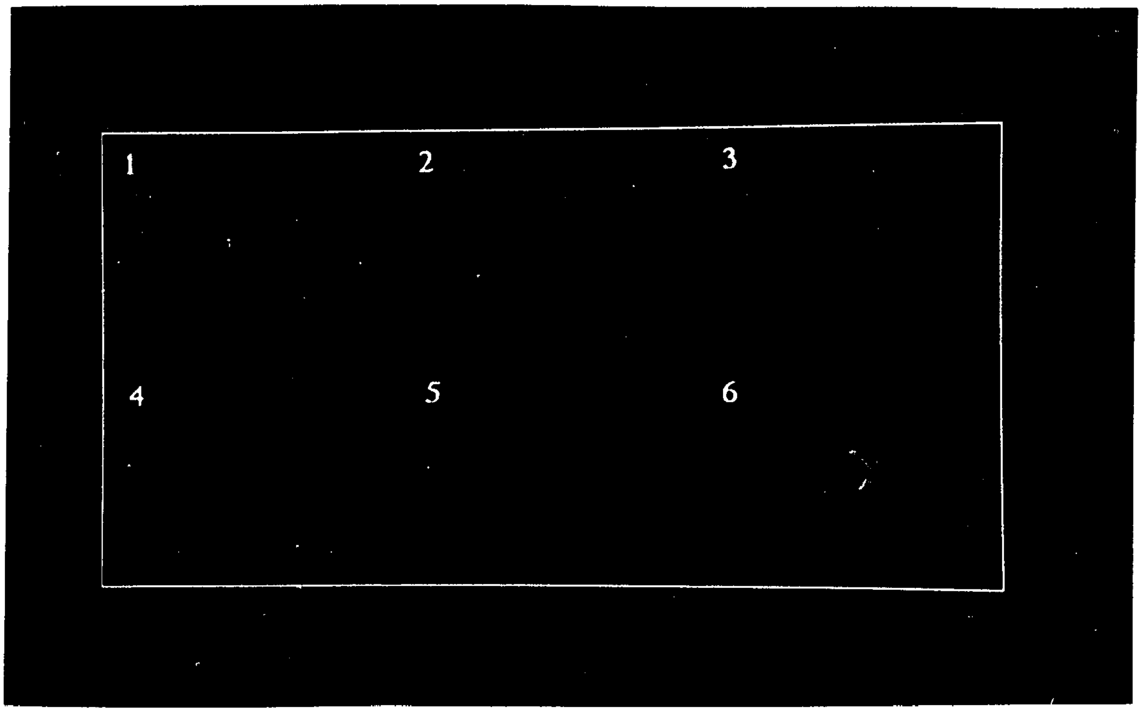


Figure 19: The deformation of the $n = 3$ Fermat surface to the Riemann surface.

out the surface, we can see that each ribbon in the Riemann surface picture (Figure 16) is split down the center, each half belonging to a distinct disk surrounding the fixed points. Figure 17 shows a schematic diagram of the punctured disks and Figure 20 shows two images generated by this sweeping process for the Fermat $n = 3$ surface. The outer boundaries in Figure 20 grow out to the infinity corresponding to the centers of the three disks in Figure 16.

New insight into the topological structure of the Fermat surfaces can be gained by these methods. The same set of tools that we have developed to study the Fermat surfaces can be applied to visualize other mathematical surfaces. In order to better utilize these interactive tools, we need to have a good user interface, which is the subject of next section.

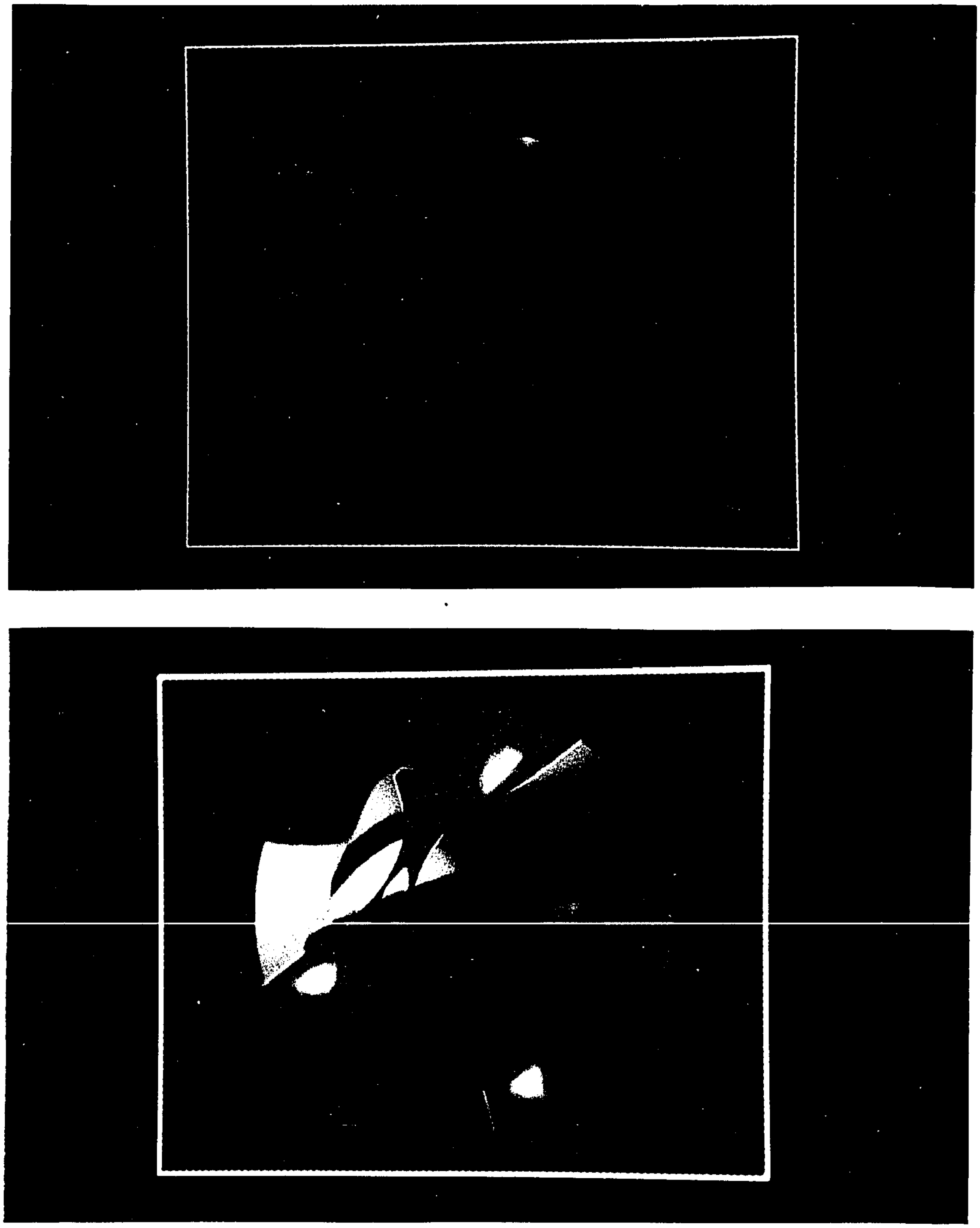


Figure 20: Two views of the $n = 3$ Fermat surface with the three pairs of fixed points removed.

2.4 User Interface Design

Human-computer interaction can play a very important role in visualization. In order to explore abstract and complex four-dimensional topological objects on a two-dimensional computer screen, a well designed user interface is essential. A poorly designed interface can degrade user productivity and increase user frustration[40]. We have used the object-oriented graphics class library described in Appendix A to construct our visualization system; besides sharing all the user interface features of that library, we added many extra slider objects and button objects to handle the interactive techniques described in previous sections.

In this section, we discuss several techniques that we have used in the user interface design.

2.4.1 Direct Context-free Manipulation

A direct context-free user interface allows users to concentrate only on the object that is being manipulated and nothing else. Compared to a mode-dependent interface, it can be more natural and efficient. In order to provide direct context-free manipulation of graphical objects, we added the following features to the existing interface:

- **Picking.** Every graphical object displayed on the screen can be picked by moving the mouse cursor within the object and clicking the left mouse button. A picked object is highlighted by a change of its representation. That is, a fully shaded object will be rendered in wire-frame when it is picked and vice versa. The picked object can be scaled and translated by moving the mouse while pressing a button; the display is updated in real-time. Only the relative mouse movements are needed in the computation of these transformations, so user does not have to pay attention to the absolute position of the cursor.

- **Rolling ball algorithm for rotation.** Instead of implementing popular but context-dependent methods like the "Virtual Sphere"[31] for controlling orientation, we use the context-independent "Rolling Ball" method developed by Andrew Hanson[54]. The basic principle of the *rolling ball* method, is to treat the graphical object to be rotated as a ball lying on a table beneath the horizontal palm of the user's hand. All possible rotations of the graphical object can be carried out by moving the mouse in a plane as you would move your hand in order to rotate the ball to a desired orientation. The position of the cursor is irrelevant for the rolling-ball algorithm, and only the relative mouse movement is needed in the computation. Therefore, this method is truly context-free. There are several advantages of using this technique:
    - **Appearance of Direct Manipulation.** The motion of the object on the screen matches the hand motion. The user gains better control of the object being manipulated.
    - **Freedom from Context.** Motions of objects on the screen are controlled solely by the hand motion. The state, position, or history of the input device can be totally ignored. The user can pay full attention to the object being manipulated.
    - **Four-dimensional Rotations.** The rolling ball method is able to handle three degrees of rotational freedom, which is sufficient for three-dimensional rotations. As mentioned in Appendix B, there are six degrees of rotational freedom in a four-dimensional space. Because the rotating group in 4D ($SO(4)$) decomposes into two independent copies of the 3D rotations ($SO(3) \times SO(3)$), we can use two copies of the rolling ball to handle 4D rotations. A shift key can be used to toggle between the two copies of rolling ball.

The drawbacks of of this method include:

- **Dependence on High-Performance Graphics Updates.** Immediate interactive feedback is critical in this technique. If the object being manipulated is too complex and requires too much time to render, the user can either change its representation from shaded image to wire-frame rendering or invoke the rotating cube option described in Section A.2 of Appendix A, that is interactive rotations are performed on a representative simpler cube instead of on the original object.
- **No Absolute Orientation.** There is no way to return to an exact ideal orientation using this technique. However, this drawback is also shared by other methods such as the "virtual sphere." If an exact orientation is required, the user can either use keyboard commands, button objects or slider objects to perform the desired rotations.

The rolling ball technique works extremely well in our visualization system because it provides direct context-free manipulation of topological objects in both 3D and 4D space with only a simple 2D input device, such as a mouse.

More sophisticated devices such as the data glove and head mounted displays would be ideal to perform advanced direct context-free manipulation in topological visualization. For example, one might "poke," "pinch," or "pull" on pieces of a topological object. Unfortunately, such equipment was not available during the course of this research.

2.4.2 Constraint-Based Interaction

A constraint describes a relationship that must be maintained. Constraint-based tools are useful for building user interfaces, because they can be used to maintain consistency among multiple views, and to specify relations between events and responses for animations and simulations[23]. With only simple 2D input devices such as the mouse, it is difficult to carry out many complicated "moves"[42] that are important in topological visualization. We need to add ***intelligent constraints*** to the interface so that simple input-device motions can be used to indicate complex topological transformations. Rule-based systems for satisfying various constraints would be desirable in such an interface. Although we did not implement any rule-based systems for performing complicated topological operations, we did use constraint tools in various visualization techniques and in the interface design. We needed to use constraints to visualize the nature of the "thickening" algorithm described in Chapter 3 and Chapter 4 for rendering 2-manifolds embedded in 4D. The constraint-based interaction tools we developed for this purpose included the following:

- **Display of tangent vectors at grid points.** The following constraints can be imposed for displaying tangent vectors at each grid point of the mathematical objects being manipulated:
  - If the cursor is within a threshold distance of a grid point, display its tangent vectors.
  - If the cursor is outside of the threshold distance from the previous selected grid point, erase its tangent vectors.
  - If none of the tangent vectors is being displayed and the cursor is not within any threshold, do nothing.

  By using these constraints, tangent vectors of all grid points that are within the threshold distance of the cursor will be drawn. It would be ideal if the size of the threshold distance could be adjusted interactively through a slider object. Similar techniques can be used to display normal vectors at grid points.

- **Maintain consistency between multiple views.** There are several ways to tell the user which portion of the object the cursor is on. One common method is to display the coordinates in text, and update the text while the cursor moves. However, it is difficult to read rapidly changing text, and numbers are also more difficult to interpret than pictures. The method that we used is to show another grid on the display with a simpler mapping of the parameter space. For example, we can represent the Steiner surface with the following equations:

$$\begin{aligned} x &= \cos^2 u \cos^2 v - \sin^2 u \cos^2 v \\ y &= \sin u \cos u \cos^2 v \\ z &= \cos u \sin v \cos v \\ w &= \sin u \sin v \cos v \ , \end{aligned}$$

where $0 \leq u < \pi$, $0 \leq v < \pi$. We can show a 3D projection of this on one part of the display and at the same time display another grid with the following mapping:

$$\begin{aligned} x &= r \cos \theta \\ y &= r \sin \theta \ , \end{aligned}$$

where $r = \pi/2 - v$, $\theta = 3\pi/2 - u$ when $(v < \pi/2)$, and $r = v - \pi/2$, $\theta = \pi/2 - u$ otherwise.

While the cursor moves on the Steiner grid, a flag moves at the same time on the simpler circular grid. The following constraints are used to maintain the consistency between the two views:

- If the cursor is within a threshold distance of a grid point of the Steiner grid, compute r and θ based on the values of u, v. Display a flag at the corresponding grid point of the circular grid.
- If the cursor is out of the threshold distance of a previously selected Steiner grid point, erase its corresponding flag on the circular grid.
- If none of the flags is being displayed and the cursor is not within any threshold distance on the Steiner grid, do nothing.

The moving flag on the simpler grid provides viewers a clearer picture of which portion of the mathematical object they are exploring. We can use this technique together with the display of tangent vectors or normal vectors by combining their constraints. An application of these techniques is shown in Figure 21.

2.5 Summary

In this chapter, we presented all the interactive visualization techniques that we used and developed for exploring topological objects with our visualization system. Interesting surfaces motivated by the study of Fermat's Last Theorem have been used as case studies in developing and applying these techniques. A deformation of the $n = 3$ Fermat surface to its corresponding cubic Riemann surface was discovered after substantial effort was spent in exploring the surface with the visualization tools. Such a deformation provides the viewer better understanding of the topological properties through different representations of the same surface.

User interface design is an important issue in any interactive visualization system. Direct object manipulation and constraint-based interaction are two important principles that we explored and used in constructing user interfaces. Additional button

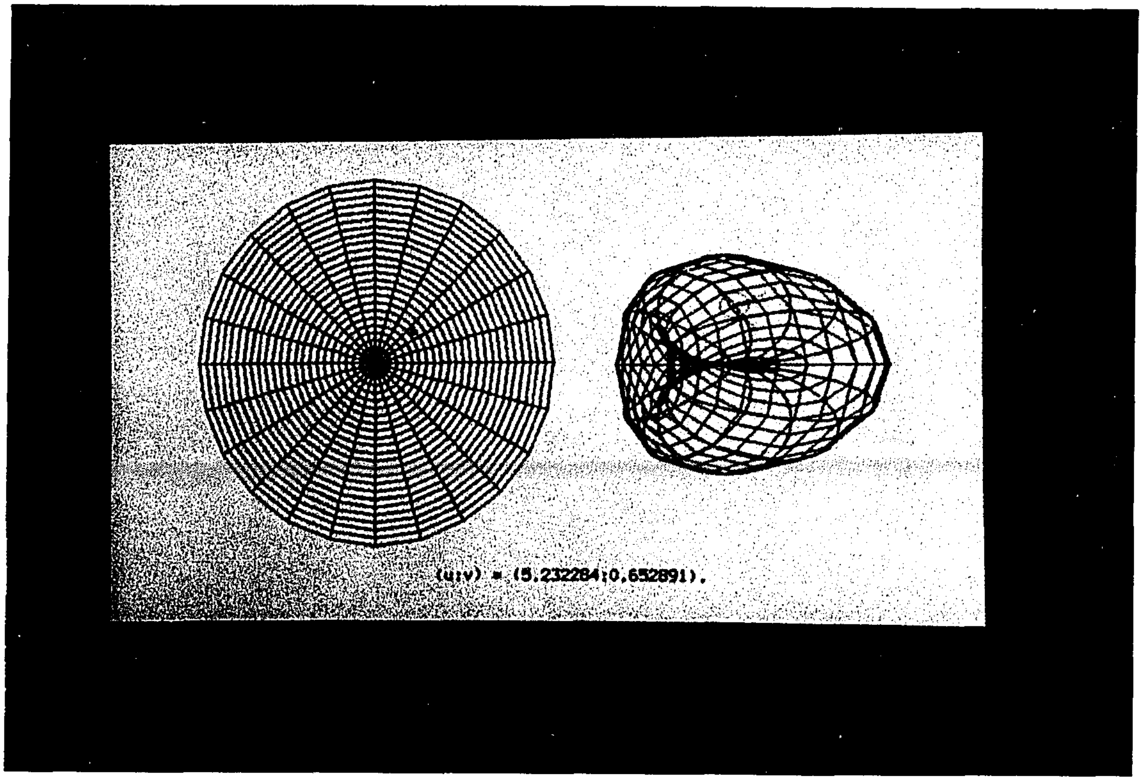


Figure 21: Display of the normal plane and the tangent vectors of the Steiner surface at a selected grid point, with display of a flag at the corresponding simpler circular grid point.

objects and sliders objects for various visualization techniques have been added to the interface using classes available in the object-oriented graphics class library.

We can experience almost real-time interaction in this system because we are only exploring the 3D projected counterpart of the four-dimensional mathematical objects. These projected objects are rendered with standard 3D computer graphics rendering tools; many of these tools are implemented in hardware. In the next chapter, we present four-dimensional rendering tools that generate other types of computer images for mathematical objects by using a 4D virtual camera and 4D lighting. These new rendering methods are much more computing intensive and it is difficult to have real-time interaction on the current hardware. However, when future hardware becomes powerful enough to handle these computations in real-time, all the interactive techniques we described in this chapter will still be applicable; the only difference is that instead of using standard three-dimensional rendering methods, we would use the computing-intensive four-dimensional rendering methods. As we will see in the next two chapters, the new rendering methods expose more 4D geometrical structures than projection to 3D combined with standard three-dimensional rendering methods.

Chapter 3

Four-Dimensional Rendering

In the previous chapter, we used only techniques that allowed users to explore four-dimensional mathematical objects through their three-dimensional projected counterparts. These projected objects were treated as any other objects in the three-dimensional worlds; they were illuminated by usual three-dimensional light sources and viewed from a three-dimensional viewpoint. Standard three-dimensional rendering methods available either in *Doré* or *Wavefront* were used to generate the images shown in the previous chapters.

Naturally, objects embedded in four-dimensional space are better perceived from a four-dimensional viewpoint. Although we cannot physically see these objects in a four-dimensional world, their properties can be represented quite concretely within a computer. In this chapter, we will introduce four-dimensional rendering methods that utilize virtual four-dimensional camera models and lighting to generate shaded images of four-dimensional objects.

Section 3.1 introduces some related work in four-dimensional graphics. The general approach required to render computer images of objects in different dimensions is presented in Section 3.2. Parametric equations and implicit equations can both be used to represent four-dimensional mathematical objects, but it is easier to handle the former rather than the latter in computer graphics. The definitions of these representations and their comparisons are given in Section 3.3. In Section 3.4, we present the computation of 4D normals of parametric 3-manifolds. The 4D normal is needed in determining the shading intensity information during the four-dimensional rendering process. The mathematical principles needed to compute projections and shadows of four-dimensional objects are discussed in Section 3.5 and Section 3.6 respectively. The extension of 3D rendering to 4D rendering is presented in Section 3.7. That includes the tessellation of 3-manifolds into tetrahedra, the implementation of the scan-plane algorithm to scan convert the tetrahedra, and the w-buffer algorithms to handle volume occlusion. We also compare the complexity of 3D rendering and 4D rendering. More detailed implementations of various four-dimensional intensity shading methods are described in Section 3.8. Section 3.9 presents a simple 4D ray-tracer and compares it with the w-buffer algorithm. Section 3.10 provides a summary of this chapter.

3.1 Related Work

Many techniques have been developed in four-dimensional graphics. Banchoff suggested various methods such as slicing, contours, projections, animations, and the use of color keyed to the value of the fourth coordinate in [8, 9, 10]. Bajaj presented algorithms for polygonalizing implicitly defined quadric and cubic hypersurfaces in higher dimensional space and displaying their projections in 3D[3]. Feiner and Beshers implemented *n-Vision*, an interactive system for plotting general $4D$ (or high-dimensional) data[39]. They adopted a $4D$ shading rule by mapping the fourth coordinate of each vertex into a colormap value and used it with Gouraud shading[19]. More computer-generated illustrations that are related to four-dimensional mathematical objects may be found in the works of Apéry[2], Weeks[103], Francis[42], and in the collection edited by Brisson[24]. These efforts have focussed mainly on different ways of displaying the 3D projections of surfaces (manifolds of dimension 2) embedded in 4D.

Methods for rendering 3-manifolds embedded in four dimensions are relatively obvious. We developed our own approaches to the 4D rendering problem as a part of our research. Little has been published on this subject, although many people were certainly aware of the principles. The earliest published work we have found is that of Steiner and Burton[95], and Carey, Burton, and Campbell[27], apparently unrefereed, in the popular magazine "Computer Graphics World". These articles described methods such as hidden-volume removal and solid shading to render 3-manifolds in a four-dimensional world similar to the ones we developed independently as part of this research. Steven Hollasch[64] implemented a four-dimensional ray-tracer using four-dimensional viewpoint and lighting to produce images for 3-manifolds such as hypersphere. These methods share many basic features with our own 4D rendering methods that we present in this chapter. However, none of these other 4D rendering methods are applicable to either points, curves, or surfaces embedded in 4D spaces due to the absence of unique normal vectors for these objects. We solve this problem by adopting a consistent approach to thicken 4D points, lines, and surfaces in a spherically symmetric way, thus producing objects that are uniquely renderable using techniques that have direct analogs in 3D.

3.2 Approach

To understand how to render a 4D mathematical object using computer graphics, it helps to see how rendering works for simpler 2D and 3D objects first.

- **2D.** In a 2D world, the image array is simply a line segment. Since there is a unique normal vector associated with each point of a 2D curve, light rays can be reflected from the curve to form images in the pixels of the "view line." For points, which do not have a single normal vector, light does not reflect coherently to form a useful shaded image. Figure 22 illustrates the 2D rendering process graphically.

- **3D.** We use a two-dimensional pixel array to represent an image of a 3D scene on a computer. Intensities assigned to each pixel element can be computed depending on how light is reflected from three-dimensional objects. However, both bare points and curves reflect light indeterminately and cannot form good shaded images. Only surfaces have unique normal vectors at each point, and reflect light predictably to form interpretable shaded images within the view plane. The 3D rendering process is shown in Figure 23.
- **4D.** By analogy, an image of a four-dimensional scene is a 3D *volume*, which we call the "view volume." In a 4D world, only volumetric objects (3-manifolds) reflect light properly to form shaded images in the view volume. Points, curves, and surfaces in four dimensional space do not have unique normal vectors, thus do not reflect light in an unique manner. Figure 24 illustrates the 4D rendering process graphically.

In 3D space, we solve the problem of rendering points and curves by expanding them into shiny spheres and shiny tubes respectively. The contrast between a wire and a thickened wire is illustrated clearly in Figure 25. By thickening the thin wire with a shiny circle in the normal plane at each point, we make a tube that has a unique normal vector at each point. Standard shading models can be used to generate an interpretable shaded image for such a tube, thus revealing more 3D structure of the thin wire.

Similar approaches can be used to solve the problems of rendering points, curves, and surfaces in 4D space by making these objects "thicker" so that they have unique normal vectors at each point. Basically, the thickening process for low-dimensional objects in 2D, 3D, and 4D can be carried out as follows:

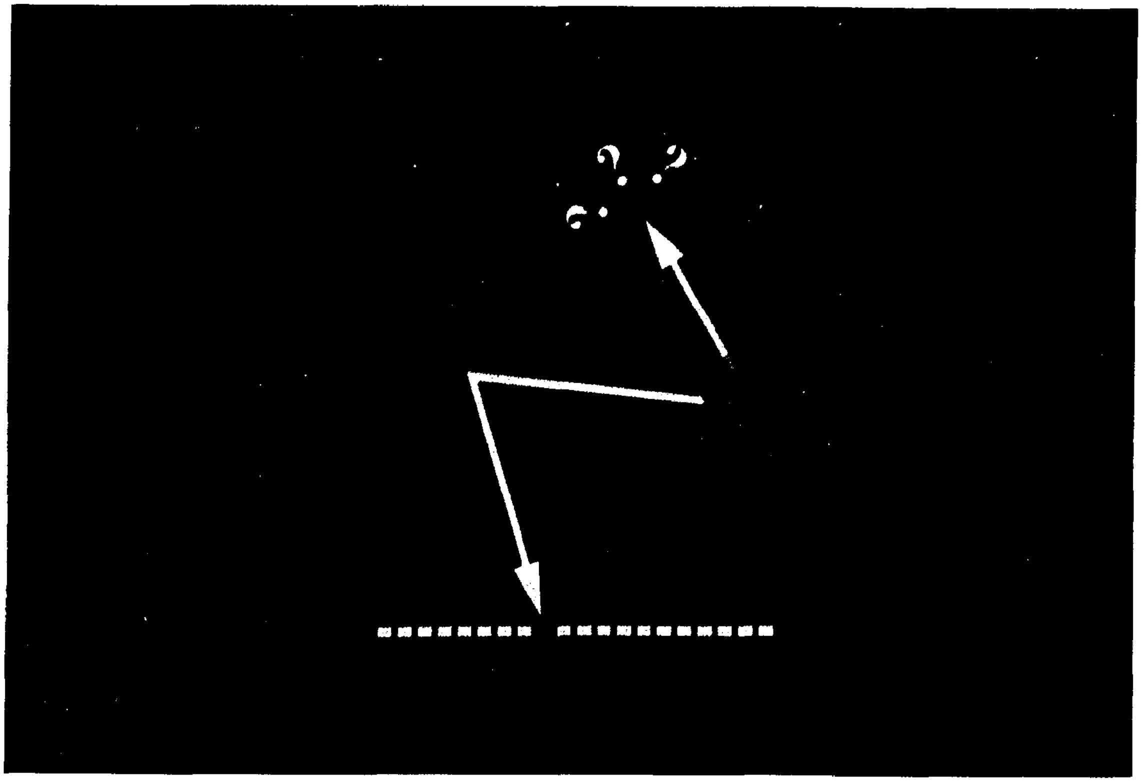


Figure 22: Only curves reflect light in a unique direction in 2D worlds to form an image in the one-dimensional view line.

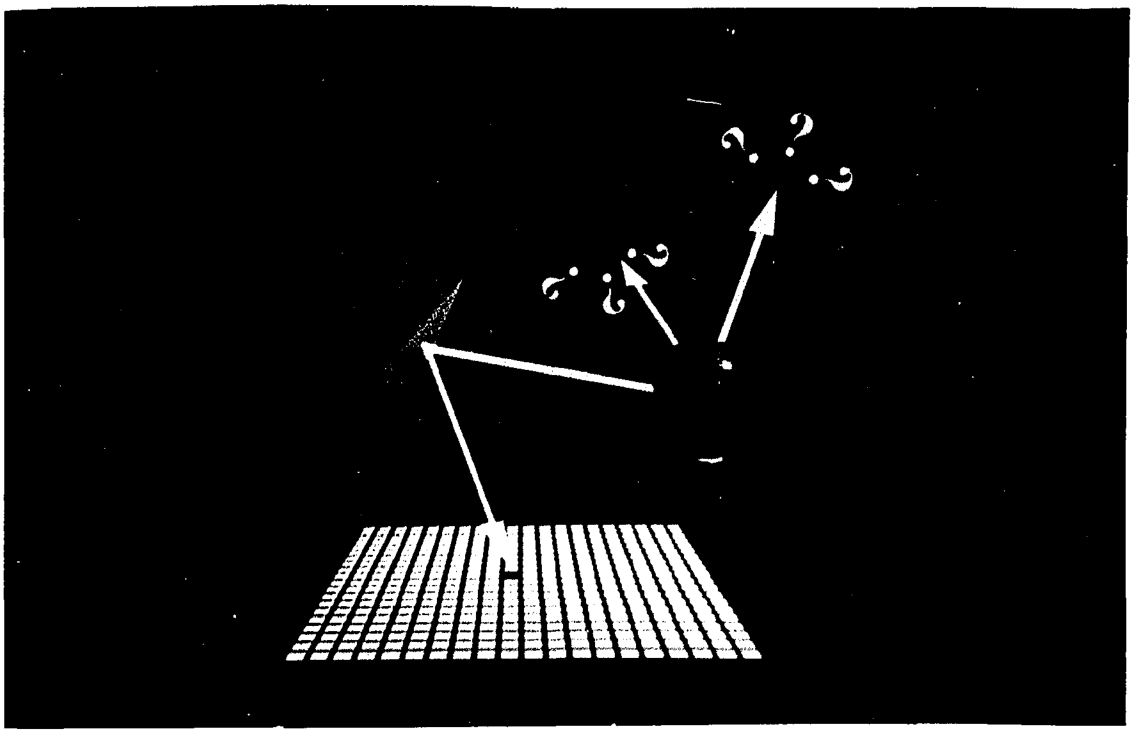


Figure 23: Only surfaces reflect light in a unique direction in 3D worlds to form an image in the two-dimensional view plane.

- **2D.** To make points reflect light uniformly in 2D, we can add a circle to each point and turn them into little curves. The 2D thickening process is shown in Figure 26.
- **3D.** To make points reflect light uniformly in 3D, we expand them into spheres. By sweeping out curves with a circle and transforming them into cylindrical tubes, we can make curves visible in 3D. Figure 27 illustrates the 3D thickening process graphically.
- **4D.** By analogy, we can make points visible in 4D by turning them into 3-spheres. We can make curves visible in 4D by sweeping 2-spheres along them. We can make surfaces visible in 4D by attaching a shiny circle at each point of the surface. The 4D thickening process is illustrated in Figure 28.

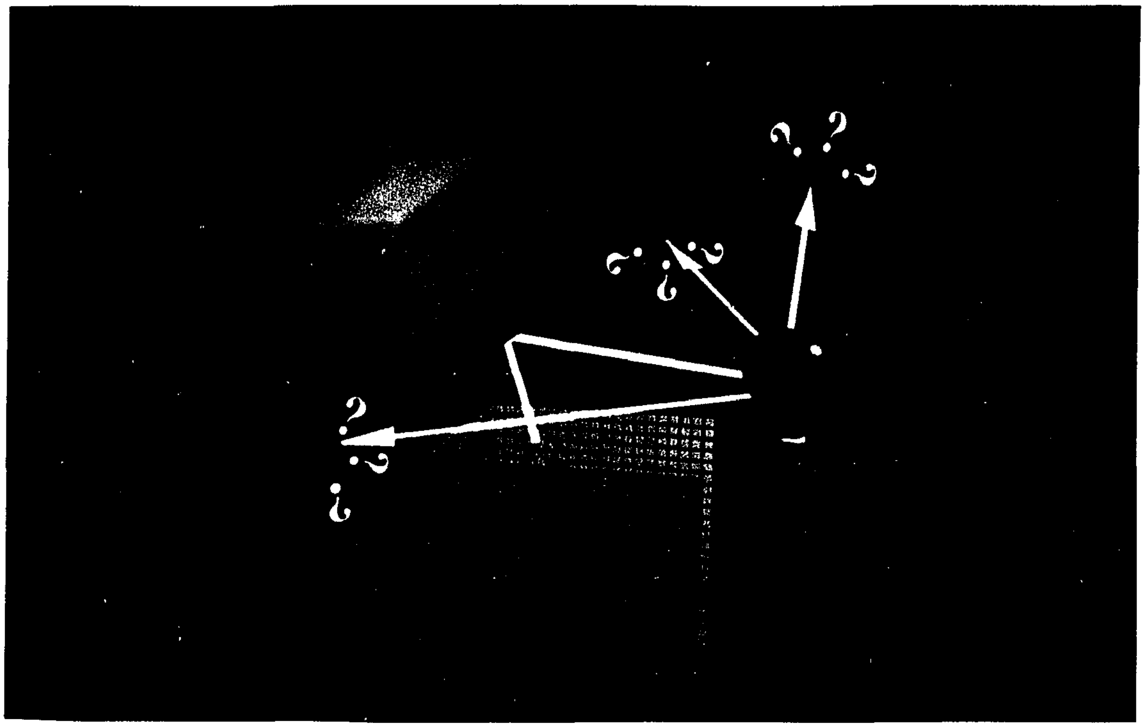


Figure 24: Only volumes reflect light in a unique direction in 4D worlds to form an image in the three-dimensional view volume.

This whole procedure for modifying objects in 2D, 3D, and 4D so that they are renderable by conventional illumination models is summarized in Table 1.

The general rule we use can be summarized as follows:

> *To make an object of dimension d (a d-manifold) interact sensibly with light rays in D dimensions, attach to each point of the object a radially symmetric (spherical) manifold of dimension $(D - d - 1)$ lying in the object's normal space.*

The result is always a manifold of dimension $(D - 1)$, the same dimension as the view array. For example, in two dimensions, $D = 2$, a point (which has dimension $d = 0$) must be expanded into a circle, which is technically a sphere of dimension 1; in four dimensions, $D = 4$, a line ($d = 1$) must have ordinary balloon-like 2-dimensional spheres attached to each point to expand it into a 3-manifold.

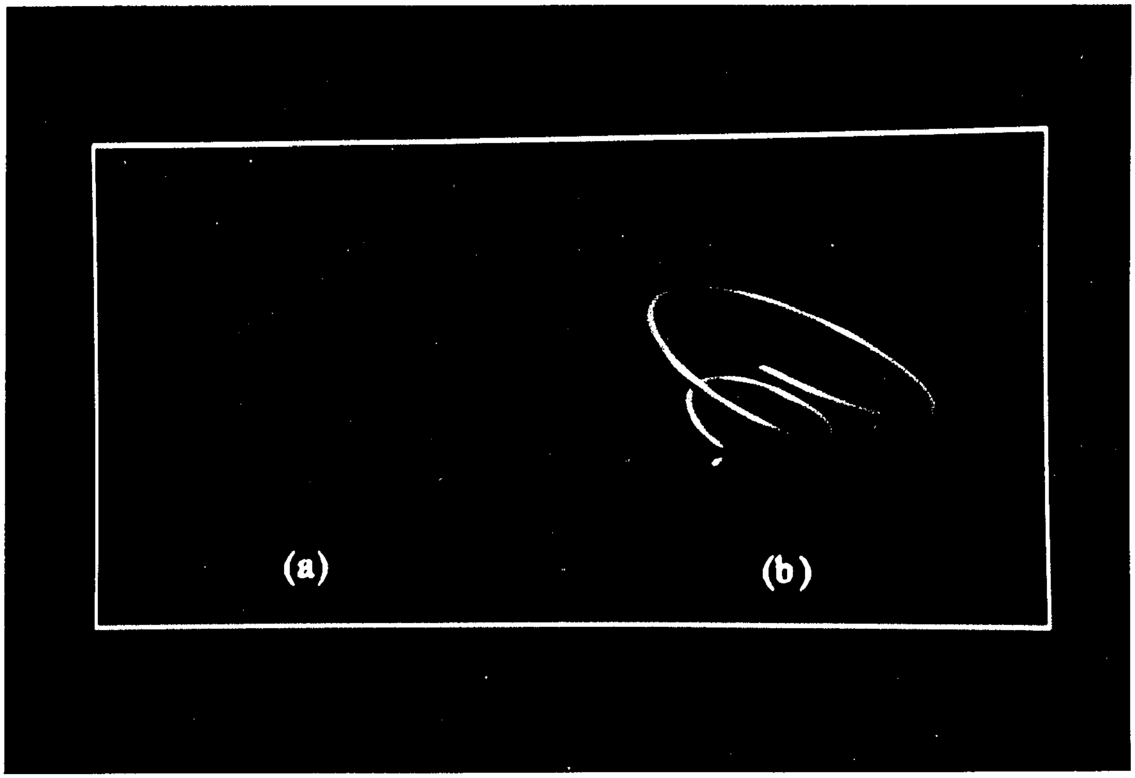


Figure 25: (a) A thin wire in a 3D world shows no 3D structure. (b) A wire thickened to form a shiny tube interacts with light to reveal much of the 3D structure of the underlying thin wire.

	Add Indicated Structure to Make Object Renderable			
World Dimension	Points $d = 0$	Curves $d = 1$	Surfaces $d = 2$	Volumes $d = 3$
$D = 2$	Circle	—		
$D = 3$	Sphere	Circle	—	
$D = 4$	3-Sphere	Sphere	Circle	—

Table 1: To make an object of dimension d in a D-dimensional world renderable, give it dimension $(D - 1)$ by attaching the indicated $(D - d - 1)$-dimensional object to each point.

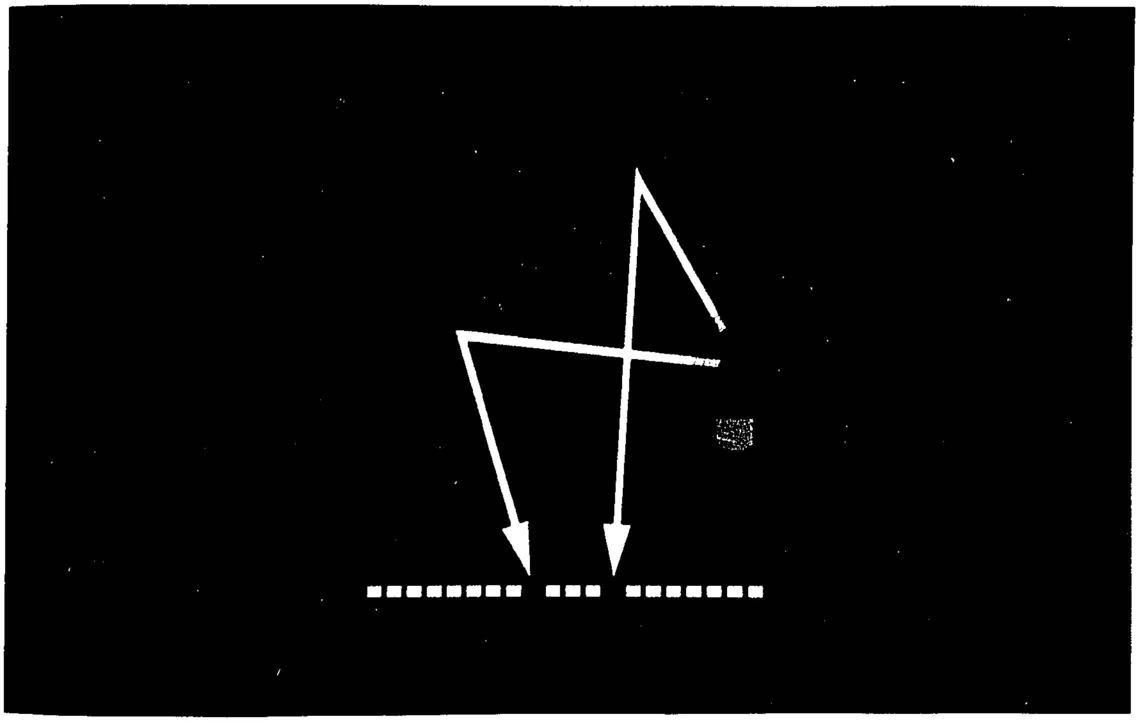


Figure 26: To make points reflect light uniformly in 2D, we can add a circle to each point and turn them into little curves.

With these methods, lower dimensional objects in 4D can be converted into 3-manifolds. The 4D rendering process becomes a process of generating shaded volume images for 3-manifolds, which is very similar to the process of generating shaded view-plane images for 2-manifolds in 3D spaces.

3.3 Mathematical Representations

Mathematical objects can be represented by polynomials or ratios of polynomials. Two common mathematical representations are parametric equations and implicit equations.

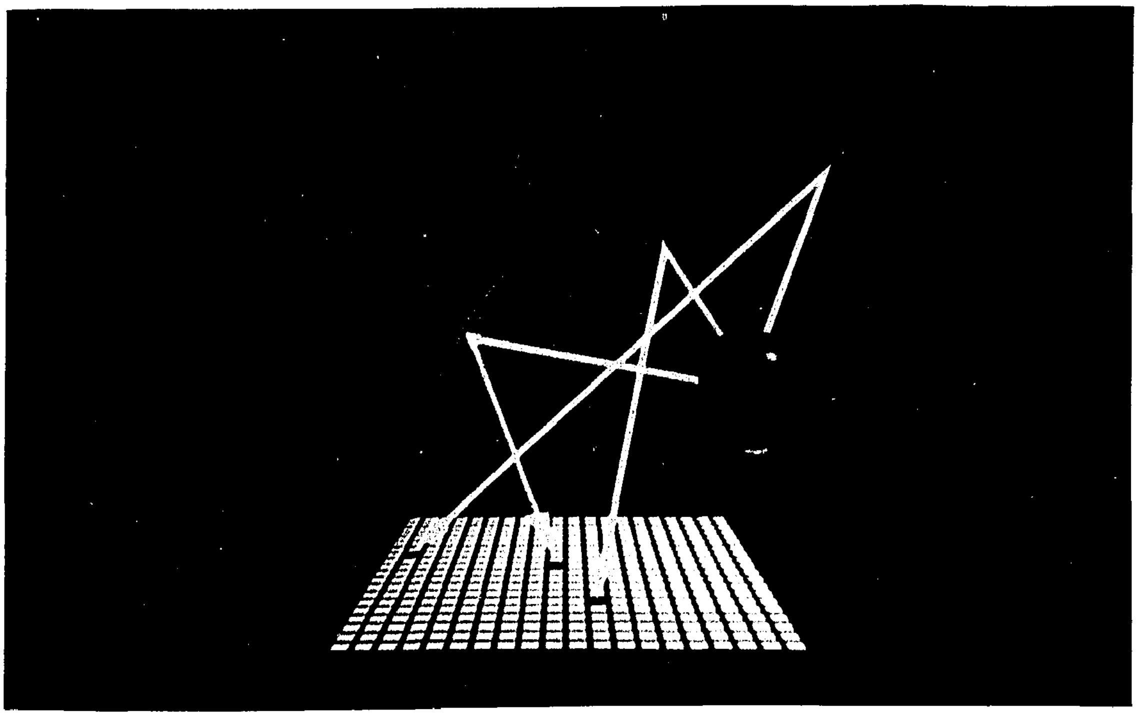


Figure 27: Points in 3D are expanded into spheres, curves in 3D are swept with circles so that they can reflect light uniformly.

3.3.1 Parametric Representations

In 2D, a *parametric curve* can be defined as two functions of one variable:

$$\begin{aligned} x &= h_1(\alpha) \\ y &= h_2(\alpha) \ , \end{aligned} \tag{18}$$

and in 3D, a *parametric surface* can be defined as three functions of two variables:

$$\begin{aligned} x &= h_1(\alpha,\beta) \\ y &= h_2(\alpha,\beta) \\ z &= h_3(\alpha,\beta) \ . \end{aligned} \tag{19}$$

Surfaces (2-manifolds) in 4D can be defined as four functions of two variables, α,

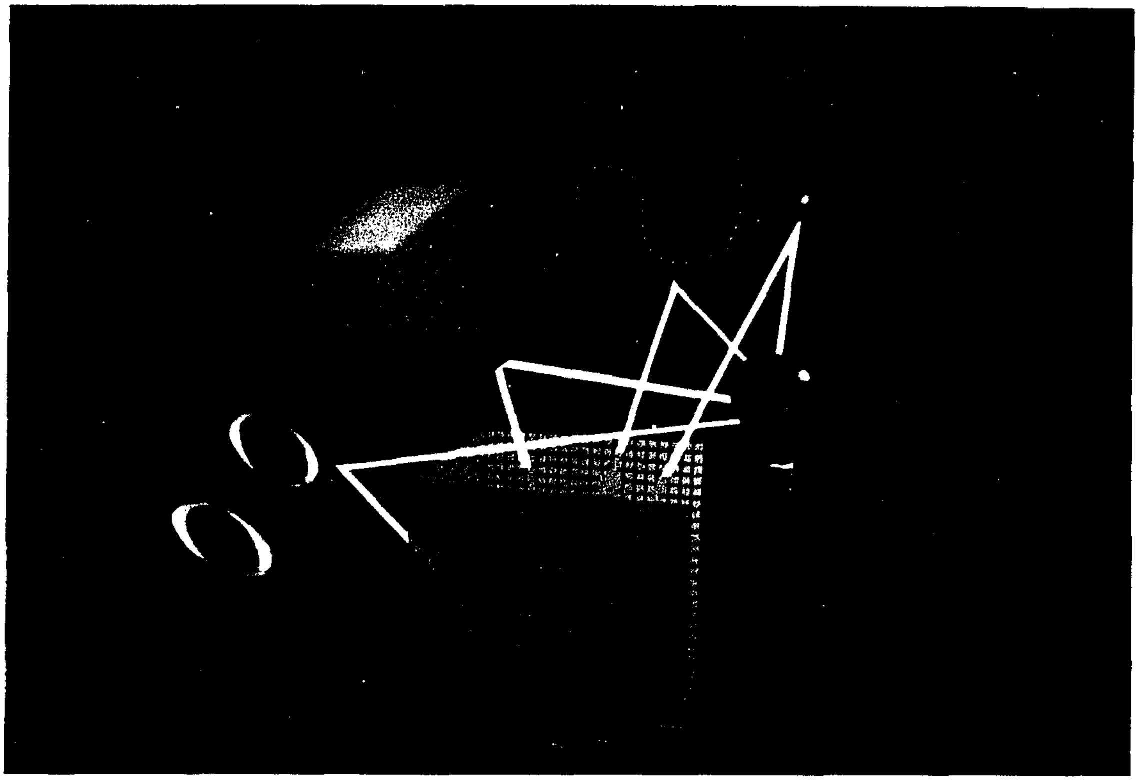


Figure 28: Points in 4D are expanded into 3-spheres, curves in 4D are swept with 2-spheres, and surfaces in 4D are renderable if we attach a shiny circle to each point of the surface.

and β:

$$\begin{aligned} x &= h_1(\alpha,\beta) \\ y &= h_2(\alpha,\beta) \\ z &= h_3(\alpha,\beta) \\ w &= h_4(\alpha,\beta) \ , \end{aligned} \tag{20}$$

and volumes (3-manifolds) in 4D can be defined as four functions of three variables, α, β, and γ:

$$\begin{aligned} x &= h_1(\alpha,\beta,\gamma) \\ y &= h_2(\alpha,\beta,\gamma) \\ z &= h_3(\alpha,\beta,\gamma) \\ w &= h_4(\alpha,\beta,\gamma) \ . \end{aligned} \tag{21}$$

3.3.2 Implicit Representations

Mathematical objects can also be represented by implicit equations. Provided the corresponding functions are not degenerate,

$$g(x,y) = 0$$

describes a curve in 2D,

$$g(x,y,z) = 0$$

is a surface in 3D and

$$g(x,y,z,w) = 0$$

is a 3-manifold in 4D, where g is typically a polynomial in the variables of the space.

3.3.3 Parametric Representations versus Implicit Representations

In computer graphics, it is easier to handle a parametric representation than to deal with an implicit representation. Unfortunately, some curves, surfaces and volumes do not have a parametric form.

Implicit Representations. The drawbacks of using implicit representations include:

- **Multipled valued.** Implicit equations may have more solutions than we need in the rendering. For example, to use $x^2 + y^2 = 1$ for a circle we need to determine which solution is nearest to the "camera".
- **Tangent continuity.** If an object is defined by patching together implicit functions, what should the tangent be at the junctions?
- **Infinite geometric slopes.** Implicit equations may give infinite geometric slopes at certain points.

The advantages of using implicit representations include:

- **Ease of computing the normal.** Once a given point on the mathematical object is known, the normal needed for computing shading intensity can be computed analytically at that point. For example, the surface normal to a function $f(x, y, z)$ is the vector $[df/dx, df/dy, df/dz]$.
- **Point-membership test.** To test whether a point is part of the mathematical object, simply substitute the point into the implicit equation and evaluate. Inside-outside tests are also trivial by checking the sign of the evaluated expression at a point.

Parametric Representations. In the parametric form, each coordinate is represented by its own separate, independent function.

The drawbacks of using the parametric equations are:

- **Addition level of complexity.** For example, derivatives become *vector-valued* instead of *scalar-valued* in the implicit form.
- **Normal computation is more expansive.** To compute the normal vector at a point, we must first compute the tangent vectors at that point and then take the cross product of those tangent vectors, and orthonormalize.

The advantages of using the parametric equations are:

- **Parametric tangent vectors.** Slopes are specified by parametric tangent vectors, which are never infinite.
- **No multiple values.** Parametric representations are single valued.
- **Ease of tessellation.** Tessellations of surfaces into rectangular grids and tessellations of volumes into cubic meshes are straightforward.

Due to their important advantages and ease of implementation, we have sought and used parametric representations in our work. However, 3D scalar fields that are generated by implicit equations can also be rendered using our methods. More details about visualizing general 3D scalar fields are given in Chapter 5.

3.4 Normal Computation for Parametric Three-manifolds

Normals of mathematical objects are needed in computing the shading information during the rendering process. In four-dimensional space, points, curves, and surfaces do not have well-defined normals. Therefore, they are not renderable with standard shading methods unless we "thicken" them into 3-manifolds. In this section, we give the details for computing the 4D normal of a parametric 3-manifold.

The normal of a volume V in four-dimensional space, is defined as a vector associated with each point p on V, that is orthogonal to the tangent vectors at p. Let V be a 3-manifold represented as in Eq. (21), then the three tangent vectors at the point $p = (x, y, z, w)$ are determined by substituting the corresponding values of α, β, and γ into the following equations:

$$\begin{aligned}
\vec{T_\alpha} &= [\frac{\partial h_1(\alpha,\beta,\gamma)}{\partial \alpha}, \frac{\partial h_2(\alpha,\beta,\gamma)}{\partial \alpha}, \frac{\partial h_3(\alpha,\beta,\gamma)}{\partial \alpha}, \frac{\partial h_4(\alpha,\beta,\gamma)}{\partial \alpha}] \\
\vec{T_\beta} &= [\frac{\partial h_1(\alpha,\beta,\gamma)}{\partial \beta}, \frac{\partial h_2(\alpha,\beta,\gamma)}{\partial \beta}, \frac{\partial h_3(\alpha,\beta,\gamma)}{\partial \beta}, \frac{\partial h_4(\alpha,\beta,\gamma)}{\partial \beta}] \\
\vec{T_\gamma} &= [\frac{\partial h_1(\alpha,\beta,\gamma)}{\partial \gamma}, \frac{\partial h_2(\alpha,\beta,\gamma)}{\partial \gamma}, \frac{\partial h_3(\alpha,\beta,\gamma)}{\partial \gamma}, \frac{\partial h_4(\alpha,\beta,\gamma)}{\partial \gamma}] .
\end{aligned} \tag{22}$$

By computing the 4D cross product of $\vec{T_\alpha}, \vec{T_\beta}$, and $\vec{T_\gamma}$, we get the normal vector $\vec{N}$ at p, which is orthogonal to the three tangent vectors. That is

$$\vec{N} = \vec{T_\alpha} \times \vec{T_\beta} \times \vec{T_\gamma} \tag{23}$$

The 4D cross product is defined in the Appendix B.

3.5 Projections

Four-dimensional objects need to be projected into a three-dimensional view volume first before any rendering can take place. To project from D dimensions to $(D-1)$-dimensional view hyperplanes, either orthographic projection or perspective projection can be used. In orthographic projection, no computation is needed; it is done by dropping one of the D coordinates. To perform a perspective projection, we let $\vec{X} = (X_1, X_2, \ldots, X_D)$ be the location of a viewpoint, and let the equation satisfied by points $\vec{x}$ in the image (hyper)plane be

$$\hat{n} \cdot \vec{x} = c \,, \tag{24}$$

where $\hat{n} \cdot \hat{n} = 1$. Given any known point $\vec{H}$ in the (hyper)plane, we can determine the value of c using $c = \hat{n} \cdot \vec{H}$. The image of a scene point $\vec{P} = (P_1, P_2, \ldots, P_D)$ is found by substituting the parametric equation of a line joining the viewpoint $\vec{X}$ to the scene point, $\vec{x}(t) = \vec{X} + t(\vec{P} - \vec{X})$, into Eq. (24), and solving for

$$t_0 = (c - \hat{n} \cdot \vec{X})/(\hat{n} \cdot (\vec{P} - \vec{X})).$$

The image point $\vec{I}$ lying *within* the image (hyper)plane, Eq. (24), is then:

$$\vec{I} = \vec{x}(t_0) = \frac{\vec{X}(\hat{n} \cdot \vec{P} - c) + \vec{P}(c - \hat{n} \cdot \vec{X})}{(\hat{n} \cdot \vec{P} - \hat{n} \cdot \vec{X})}. \tag{25}$$

Perspective projection is used more often because it produces a *perspective foreshortening* visual effect, which is similar to that of the human visual system; objects that are nearer to the viewpoint show up bigger and objects that are farther from the viewpoint appear smaller in the view hyperplanes. More detail about projections can be found in [41, 28].

3.6 Shadow Algorithms

In computer graphics, shadows of three-dimensional objects are cast into a 2D plane; shadow polygons lie in the plane joining the light source to the edges of the three-dimensional objects. By analogy, the shadows of four-dimensional objects are cast into a 3D volume; whenever a ray cast from the 4D light source onto the 3D volume is obstructed by a four dimensional object, its shadow will show up as a darkened, smokelike cloud in that 3D volume. Shadow algorithms determine which parts of objects are visible to the light source. Thus, shadow information can be computed using the same projection equations given in the previous section simply by letting $\vec{X}$ in Eq. (25) be the location of a light source instead of a viewpoint.

A single shadow seen alongside the object in a single view provides addition information that is similar to placing an addition camera at the light source. If we know the location of the light source and of the shadow hyper(plane), even a single shadow can provide some stereographic depth information. Multiple light sources can be used to generate multiple shadows, which provide additional constraints resembling the various views in a typical engineering drawing. Four views are required for such a drawing in four dimensions.

Figures 29 and 30 show examples of (unthickened) 2D surfaces embedded in 4D along with their 4D shadows. These images are made by projecting the object and its shadow into 3D, illuminating the object's opaque projection with 3D lighting, treating the shadow as a gray semi-transparent film, and rendering everything into a conventional 2D image. The shadows here are thin surfaces, artificially thickened for visibility, as one might treat the shadow of a wire in 3D. Rotating one of these objects in 4D interchanges its image outline with that of its shadow, as shown in the figures. The mathematical descriptions of these objects are given in the next chapter.

3.7 Extending Rendering from 3D to 4D

The rendering process in four dimensions is closely analogous to that in three dimensions. The following gives a step by step comparison between 3D rendering and 4D rendering[58].

In 3D, a typical Gouraud or Phong algorithm[46, 91] for rendering the surface of

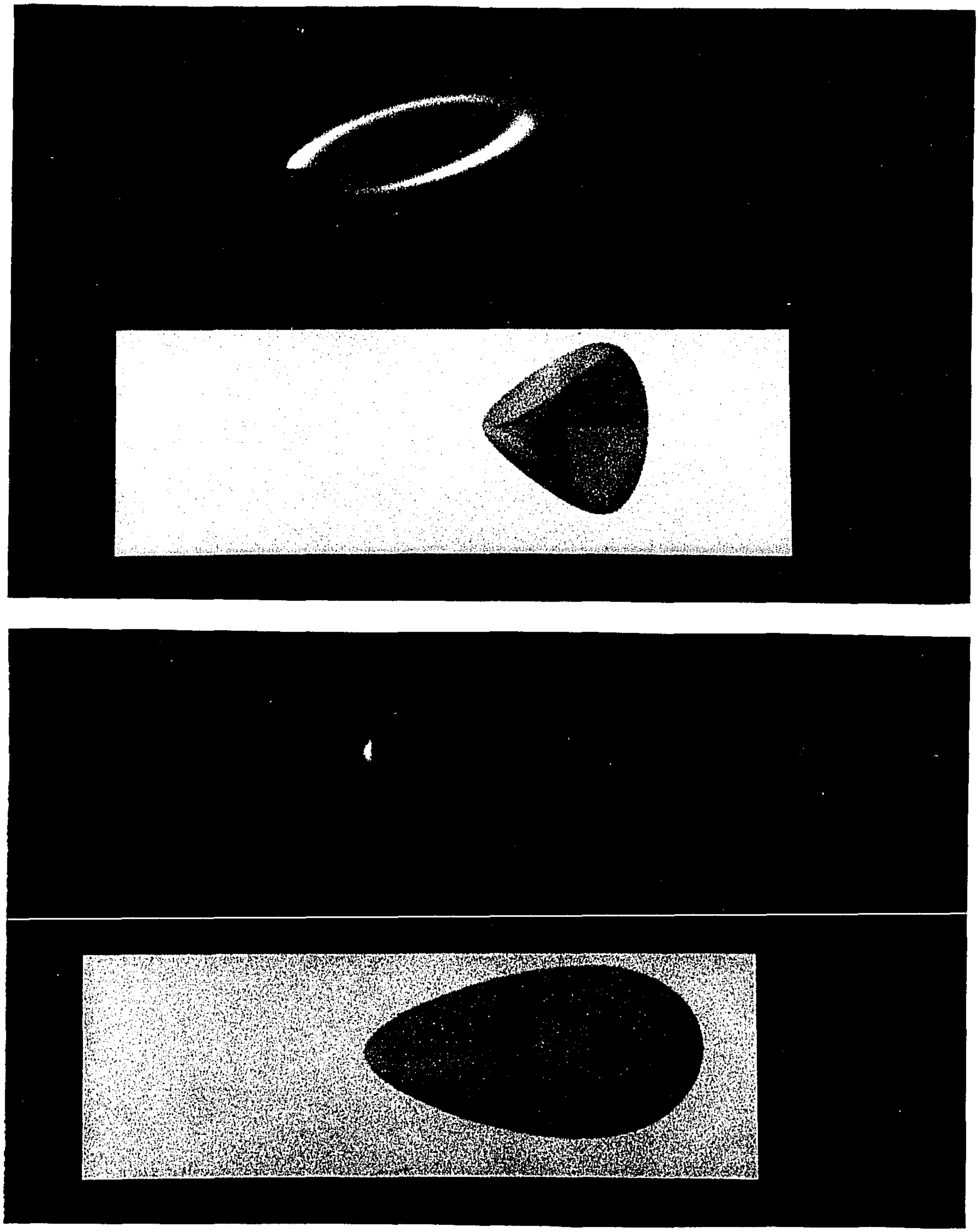


Figure 29: Two 4D views of the Steiner surface with its shadow.

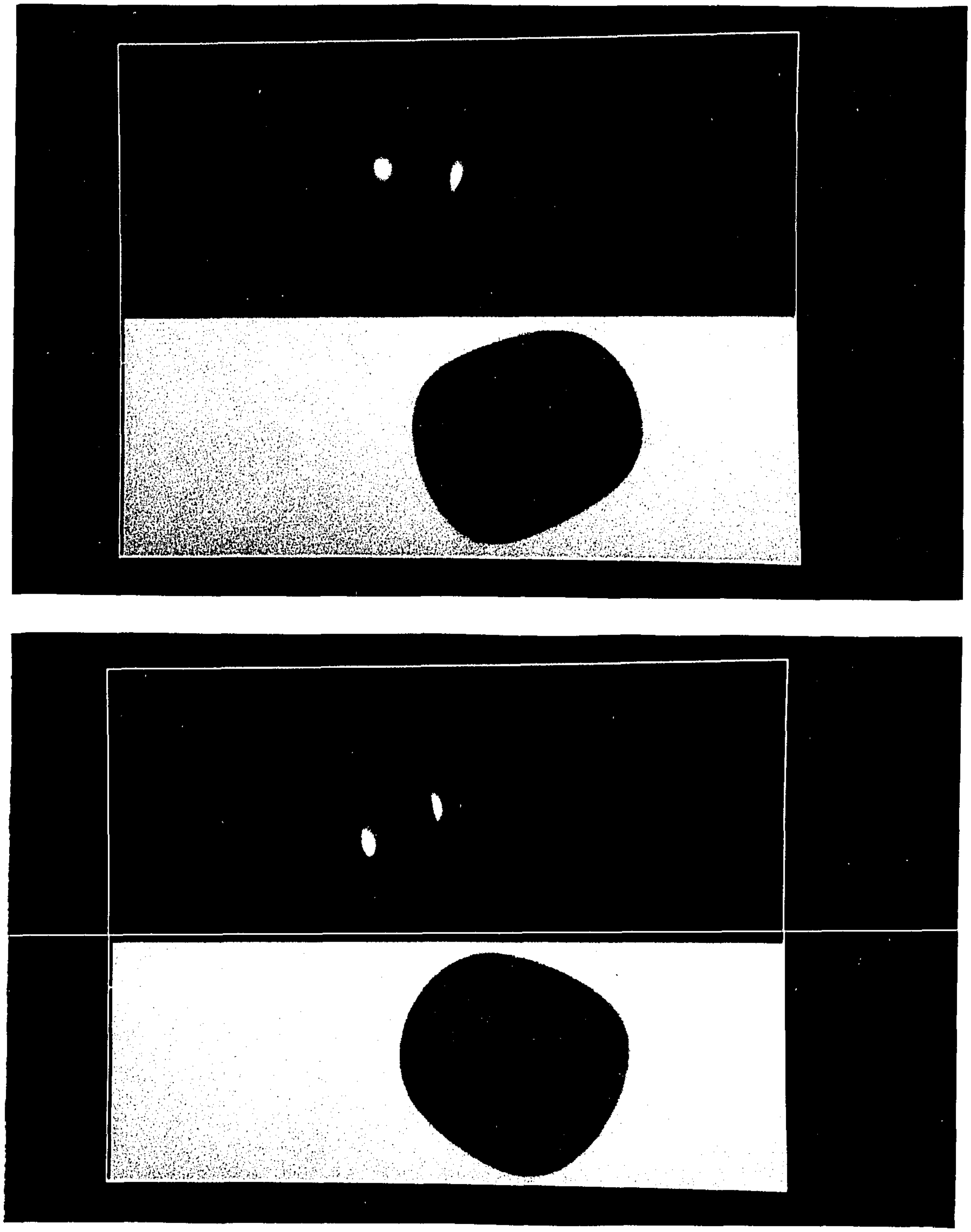


Figure 30: Two 4D views of the four torus and its shadow.

a solid object into the 2D viewplane raster follows these steps:

1. Divide planar faces into triangles.
2. Compute the normals at the face vertices. For example, take the gradient of an implicit surface defined by a single function of the 3D coordinates, compute the cross-product of tangents to a parametric surface, or average the normals of the surrounding faces in the general case.
3. Project each triangle to the 2D view plane.
4. In the view plane, interpolate the intensities (Gouraud shading) or normals (Phong shading) from vertex to vertex to get the values at the beginning and end of each scan line.
5. Interpolate these values along the scan line to get the intensity or normal at an arbitrary pixel in the projected triangle.
6. For Phong shading, compute the intensity at the pixel using the interpolated normal.
7. Use a z-buffer or equivalent method to eliminate hidden surfaces if required.

In four dimensions, we alter these steps to render the volume element as follows:

1. Subdivide the volume into tetrahedra embedded in 4D. Take care to avoid subdivisions that leave empty volume gaps between adjacent nonplanar faces if a rectangular lattice is being used.
2. Compute the intensities or normals at the tetrahedral vertices. For example, find the normals by taking the gradient of a *volume* defined implicitly by a single equation in the four coordinate variables, by taking the cross-product of the tangents, or by averaging the normals of the surrounding *volumes* in the general case.

3. Project each tetrahedron into the view volume.
4. In the view volume, interpolate the intensities (Gouraud shading) or normals (Phong shading) from vertex to vertex to get the values at the *vertices of each polygon* lying in the scan planes slicing up the tetrahedron.
5. In each scan plane, we now have a planar polygon representing a slice of the tetrahedron's voxel. Using the known values at each vertex of this polygon, interpolate to find the values at the polygon's internal voxels using the standard 3D view-plane interpolation.
6. For Phong shading, compute the intensity at each voxel using the interpolated normal.
7. To eliminate occluded volumes from the view volume, use a 4D depth buffer (w-buffer) or equivalent method and store only the intensity of the nearest object lying on a given ray into the view volume voxels.

The following subsections describe the four-dimensional rendering process in greater detail and at the same time compare the complexity of 3D rendering and 4D rendering.

3.7.1 2D Grids versus 3D Grids

In 3D rendering, a parametric surface (2-manifold) defined as in Eq. (19) can be represented as a two-dimensional grid. If there are N_α and N_β grid points along the two directions, we can tessellate the surface into $N_\alpha \times N_\beta$ square patches, as shown in Figure 31(a). In 4D rendering, a parametric volume (3-manifold) defined by Eq. (21) can be represented as a three-dimensional mesh, as in Figure 31(b). If we let $N_\alpha, N_\beta, N_\gamma$ be the number of grid points along the three different directions, the volume can be tessellated into $N_\alpha \times N_\beta \times N_\gamma$ cubic cells. While we can use $N_\alpha \times N_\beta$

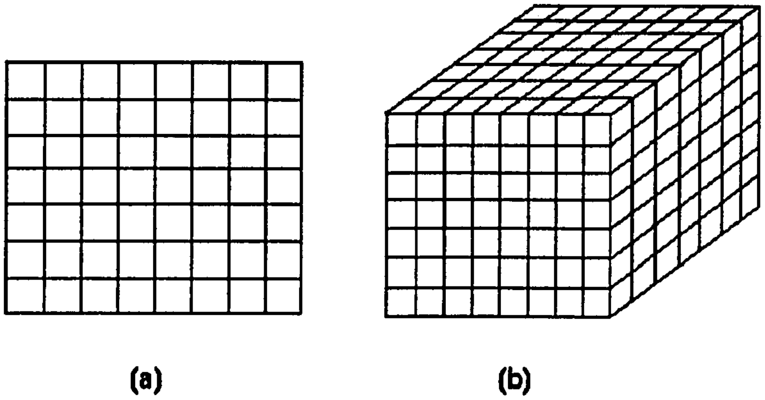


Figure 31: (a) A surface in 3D can be tessellated into a 2D grid. (b) A volume in 4D can be tessellated into a 3D mesh.

3-vectors to represent a parametric surface in 3D, we need to use $N_\alpha \times N_\beta \times N_\gamma$ 4-vectors to represent a 4D parametric volume. Besides using more memory storage, the amount of rendering effort also increases from $N_\alpha \times N_\beta$ square patches to $N_\alpha \times N_\beta \times N_\gamma$ cubic cells.

3.7.2 Decomposition of a Cubic Cell into Tetrahedra

In 3D rendering, one of the square patches in the 2D grid can be subdivided into two triangles. In 4D rendering, a cubic cell of the 3D mesh can be subdivided into 5 tetrahedra, as shown in Figure 32. However, if we apply the same decomposition to all the cells, gaps or overlaps may occur between the adjacent cells. This is known as the *cracking problem* of volume tessellation[93], which is similar to the cracking problem of surface tessellation in 3D rendering. This happens because the vertices that are shared by adjacent cubic cells are divided to form tetrahedra whose faces do not match at the boundary. To avoid this problem, we have to use the opposite rotational state of the decomposition given in Figure 32 to decompose adjacent cubic cells. Based on the index labels assigned to the vertices in Figure 32, the two vertex orderings that can be used for decomposing adjacent cubic cells are:

1. **Regular Order:** (0 1 4 3), (1 5 4 6), (6 7 4 3), (1 6 3 2), and (1 6 4 3).
2. **Opposite Order:** (0 5 2 1), (2 5 7 6), (0 2 7 3), (0 7 5 4), and (0 5 7 2).

3.7.3 Scan-line Algorithms versus Scan-plane Algorithms

In 3D rendering, a projected 2D triangle needs to be subdivided into two triangles, one with a top horizontal edge and the other one with a bottom horizontal edge. Then we can apply the standard triangle scan-line algorithms to scan-convert them into the view plane. Similarly, in 4D rendering a projected 3D tetrahedron needs to be subdivided into one of the following three types of basic tetrahedra:

1. Tetrahedron with flat top face.
2. Tetrahedron with flat bottom face.
3. Tetrahedron with horizontal top edge and horizontal bottom edge.

Next, we can scan-convert each of these tetrahedra into the view volume. Figure 33 shows that a tetrahedron can be sub-divided into five basic tetrahedra.

After checking for degeneracy to find the minimum number of tetrahedra that need to be treated, we convert each tetrahedron into voxels one scan-plane at a time. Figure 34 shows the differences between a scan line algorithm and a scan plane algorithm graphically.

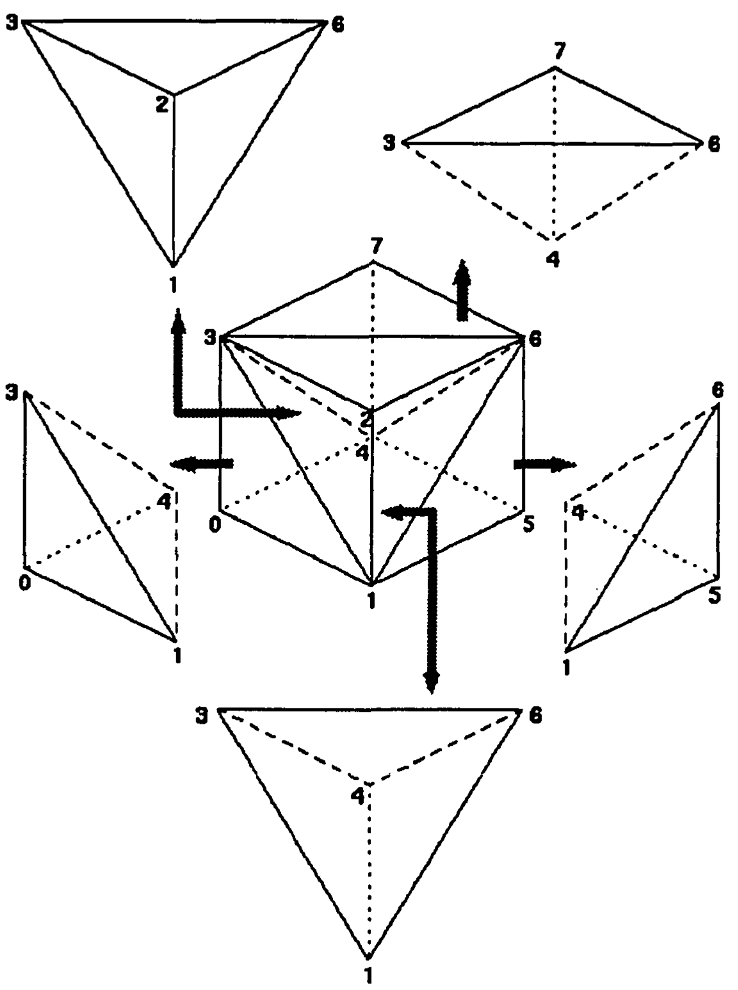


Figure 32: Decomposition of a cubic cell into five tetrahedra.

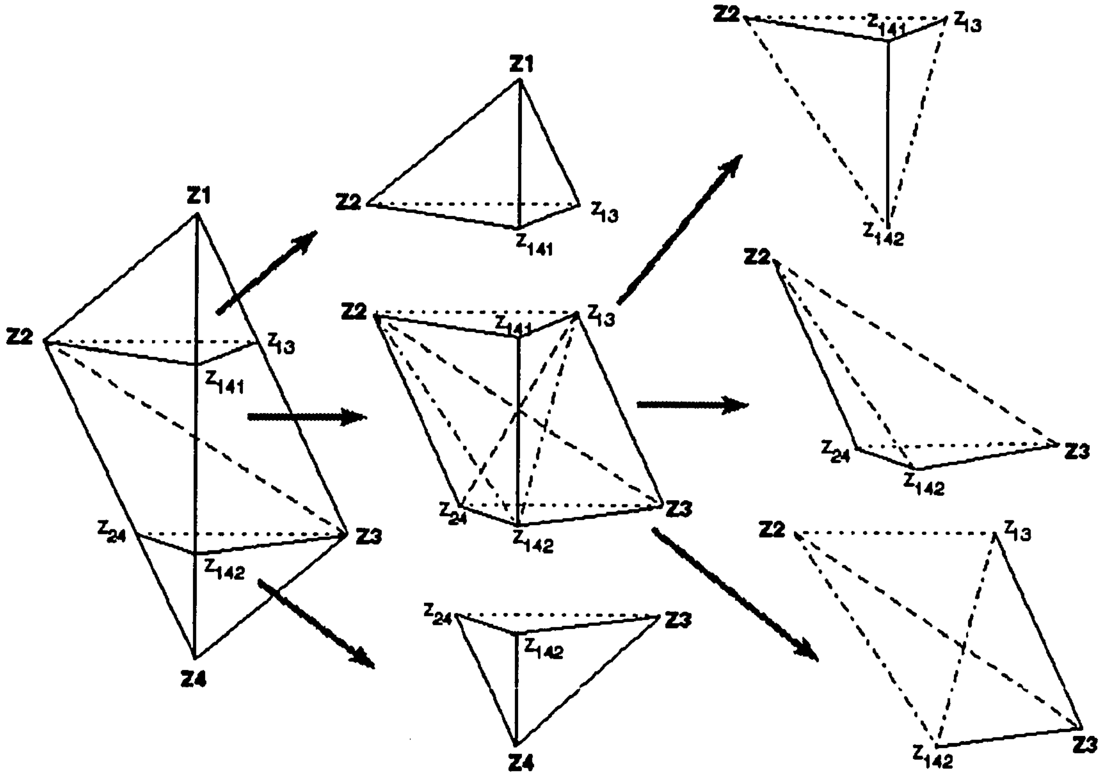


Figure 33: Decomposition of a tetrahedron into five tetrahedra.

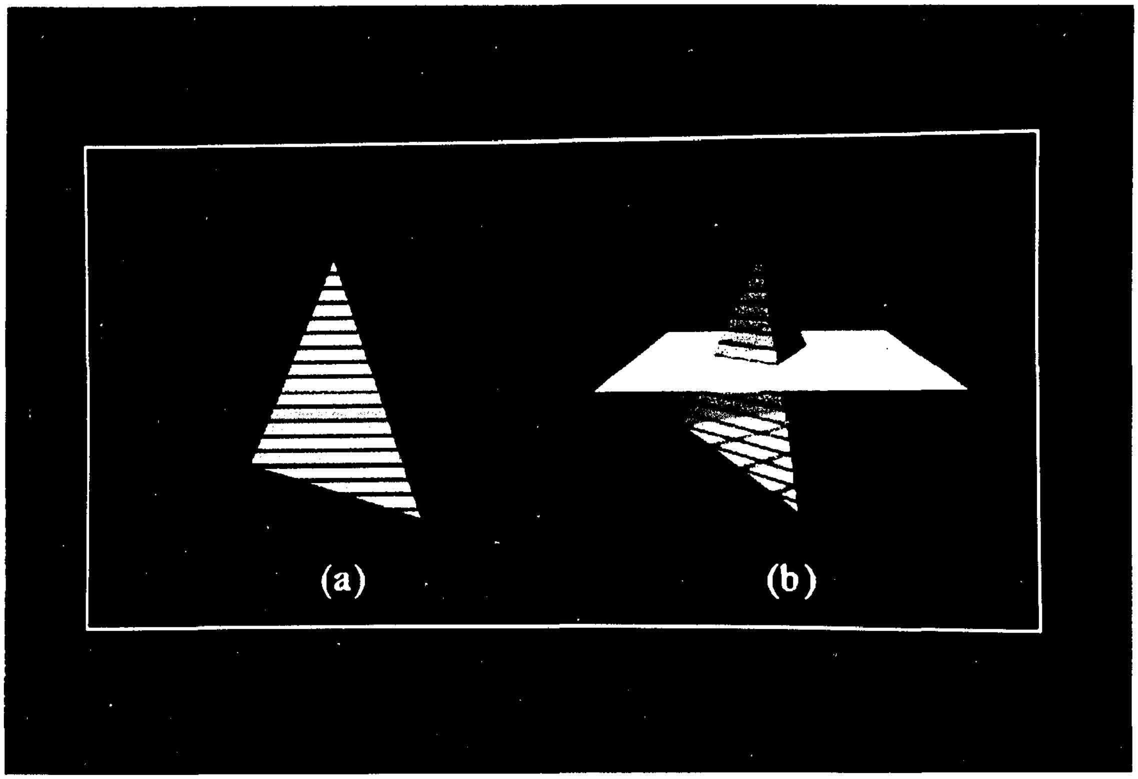


Figure 34: (a) A typical scan conversion of a polygon in 3D involves taking a scan line and projecting it to the 2D film. (b) In 4D imaging, solid objects are converted to tetrahedra and scan converted into a 3D view volume by scan-plane conversion.

3.7.4 Z-buffer Algorithms versus W-buffer Algorithms

Visibility determination is the process of finding out which portions of objects are visible from a given viewpoint. In 3D, the z-buffer algorithm is one of the most commonly used algorithms to solve the visibility determination problem. The following is a simple version of the z-buffer algorithm[67]:

```
Z-buffer Algorithm
  Given
     List of polygons $\{P_1, P_2, \ldots, P_n\}$
     A two-dimensional array z-buffer[$x, y$] initialized to -∞
     A two-dimensional array Intensity[$x, y$]
  begin
     for each polygon $P$ in the polygon list do {
        for each pixel $(x, y)$ that intersects $P$ do {
           calculate z-depth of $P$ at $(x, y)$
           if z-depth > z-buffer[$x, y$] then {
             Intensity[$x, y$] = intensity of $P$ at $(x, y)$
             z-buffer[$x, y$] = z-depth
           }
        }
     }
     Display Intensity array
  end
```

In 4D rendering, multiple volume elements may be on the same 4D line of sights; the resulting occlusion effects may be accounted for using methods such as a "w-buffer" algorithm, the 4D analog of a z-buffer algorithm. The 4D "w-buffer" algorithm that we implemented for rendering 3-manifolds is given as follows:

```
W-buffer Algorithm
    Given
        List of volumes $\{V_1, V_2, \ldots, V_n\}$
        A three-dimensional array w-buffer[$x, y, z$] initialized to -∞
        A three-dimensional array Specular[$x, y, z$] for storing specular intensity
        A three-dimensional array Diffuse[$x, y, z$] for storing diffuse intensity
    begin
        for each volume $V$ in the volume list do {
            Divide the volume into a 3D mesh of cubic cells
            for each cubic cell do {
                Divide the cubic cell into 5 tetrahedra
                Perform scan-plane conversion of each tetrahedron
                for each voxel $(x, y, z)$ that intersects $V$ during the scan-plan
                conversion do {
                    calculate w-depth of $V$ at $(x, y, z)$
                    if w-depth > w-buffer[$x, y, z$] then {
                        Specular[$x, y, z$] = specular intensity of $V$ at $(x, y, z)$
                        Diffuse[$x, y, z$] = ambient intensity + diffuse intensity of $V$
                                             at $(x, y, z)$
                        w-buffer[$x, y, z$] = w-depth
                    }
                }
            }
        }
        Display the 3D intensity array
    end
```

In a z-buffer algorithm, two $N \times N$ arrays are needed to store the intensity values and the z-depth values for an image with $N \times N$ pixels. In order to render a volume image of $N \times N \times N$ voxels, we used three $N \times N \times N$ arrays, one for storing the w-depth values, and the other two for storing the specular intensities and the diffuse intensities. The reason that we separate the diffuse intensities and the specular intensities in the four-dimensional rendering will be given later. Depending on the size of N, the w-buffer algorithm can be very slow due to memory paging. Unnecessary memory paging can be reduced by paying attention to the relation between storage order and array indexing in the loops. For example, in C programming, elements in C array are stored by rows, so the rightmost subscript varies fastest as elements are accessed in storage order; although the following two for-loops will both provide the correct result, the first one will execute much slower due to memory paging, particularly when the number N is large.

Slower For Loop in C

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            W[k][j][i] = 0;
```

Faster For Loop in C

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            W[i][j][k] = 0;
```

3.7.5 Number of Triangles in 3D and 4D Rendering

The following shows a comparison between the numbers of triangles rendered in 3D and 4D rendering.

- **3D.** To render a 3D parametric surface with grid size $N_\alpha \times N_\beta$ into a view plane of $N \times N$ pixels, we need to render $N_\alpha \times N_\beta$ square patches. Since we can subdivide a square patch into two triangles, the number of triangles to be rendered will be $2 \times N_\alpha \times N_\beta$.

- **4D.** To render a 4D parametric volume with grid size $N_\alpha \times N_\beta \times N_\gamma$ into a view volume of $N \times N \times N$ voxels, we need to render $N_\alpha \times N_\beta \times N_\gamma$ cubic cells. The number of tetrahedra involved will be less than or equal to $25 \times N_\alpha \times N_\beta \times N_\gamma$, because a cubic cell can be subdivided into five tetrahedra and a tetrahedron can be further subdivided up to five tetrahedra. Since the dimension of the view volume is $N \times N \times N$, a tetrahedron can be scan converted into at most N triangles using the scan plane algorithm. Thus, the number of triangles to be rendered will be less than or equal to $25 \times N_\alpha \times N_\beta \times N_\gamma \times N$.

In the worst case, the ratio between the number of triangles rendered in the 4D case and the number of triangles rendered in the 3D case is $\frac{25}{2} \times N_\gamma \times N$. If we let N_γ be 40 and N be 200, the number of triangles in 4D rendering will be approximately 100,000 times larger than the number of triangles in 3D rendering. Together with the fact that the w-buffer algorithm is a memory-intensive algorithm, 4D rendering is a very expensive process. The current hardware is just now ready to handle real-time 3D rendering. We need much more powerful computer processors and faster computer memory in order to do real-time 4D rendering.

3.8 Intensity Shading Methods

We are able to understand 2D images of 3D objects mainly due to the shading information that is present in those images. Shading algorithms simulate the effects of light being reflected and absorbed by objects in a scene; this generates a wealth of orientation cues that the human visual system is able to interpret reliably by imposing constraints on the ambiguous data.

Similar shading algorithms, when applied to generate shaded volume images of 4D objects, will produce similar orientation cues. We can better understand the topological structures of 4D objects if we can learn to interpret these orientation cues. In this section, we discuss the issues involved in producing shaded images of objects in 4D.

3.8.1 Ambient Light

In 3D, ambient light is the result of multiple reflections from walls and objects, and is incident on a surface from all directions. If we assume that ambient light impinges equally on all surfaces from all directions, then an ambient intensity can be assigned at each surface point as:

$$I_A = I_a \ , \qquad (26)$$

where I_a is the ambient light intensity. Surfaces that are not directly illuminated by light sources but are still visible to the viewpoint can receive some level of illumination due to this ambient light intensity. In 4D, we can assume such ambient light exists and use exactly the same equation.

3.8.2 Diffuse Reflection

In a typical 3D shading algorithm, the diffuse component of the illumination is the projection of the incoming light onto the surface. The diffuse intensity at each point can be computed using Lambert's Law:

$$I_D = I_0 \hat{n} \cdot \hat{L} \,, \tag{27}$$

where $\hat{n}$ is the normal vector at a surface point (usually a polygon vertex), $\hat{L}$ is the unit vector from the point to the light source, and I_0 is the point light source intensity. I_D is set to zero if the dot product is negative, which happens when the normal is pointing away from the light. If both sides of a polygon might be visible, the absolute value may be used instead or the surface may be doubly covered (e.g., for one-sided surfaces).

Note that $\hat{n}$ can actually be considered to be proportional to the *cross-product* of the tangent vectors $\vec{P}(u,v)$, $\vec{Q}(u,v)$ at a local point (u,v) in the surface, so the diffuse intensity can alternatively be written as

$$I_D = \frac{I_0}{H} \operatorname{Det} \begin{vmatrix} L_1 & P_1 & Q_1 \\ L_2 & P_2 & Q_2 \\ L_3 & P_3 & Q_3 \end{vmatrix} \,, \tag{28}$$

where $\vec{L}$ is the (unnormalized) vector to the light source and the normalization $H = \|\vec{L}\| \times \|\mathrm{Cofactor}\,\vec{L}\|$ is chosen to make the maximum value of the determinant be unity.

In four dimensions, the shading equation for 3-manifolds is obtained either from Eq. (27) with a four-vector normal, or by expanding the determinant in Eq. (28) to four columns and placing the tangent directions $(\vec{P}, \vec{Q}, \vec{R})$ at a local point (u, v, θ) on the 3-manifold into the columns, yielding the form

$$I_D = \frac{I_0}{H} \operatorname{Det} \begin{vmatrix} L_1 & P_1 & Q_1 & R_1 \\ L_2 & P_2 & Q_2 & R_2 \\ L_3 & P_3 & Q_3 & R_3 \\ L_4 & P_4 & Q_4 & R_4 \end{vmatrix} . \tag{29}$$

For points (dimension $d = 0$), curves ($d = 1$), and surfaces ($d = 2$) in 4D, there are zero, one, or two available tangent vectors instead of the three needed in Eq. (29), so we are missing critical information needed to compute a 4D intensity. These objects need to be thickened into 3-manifolds so that there are three available tangent vectors at each point.

Before we tried the "thickening" approaches to create shaded images for surfaces embedded in four-space, we tried using two orthogonal 4D light sources on "unthickened" surfaces. We let P, Q be the two 4-dimensional tangent vectors lying in the surface at each point (u, v), took L, M to be two orthogonal 4D light sources, and formed the determinant analogous to Eq. (29):

$$I_D = \frac{I_0}{H} \operatorname{Det} \begin{vmatrix} L_1 & M_1 & P_1 & Q_1 \\ L_2 & M_2 & P_2 & Q_2 \\ L_3 & M_3 & P_3 & Q_3 \\ L_4 & M_4 & P_4 & Q_4 \end{vmatrix} . \tag{30}$$

Although images generated by this two-light-source method gave the viewer some extra information on the relationship between the surface geometry and the 4D light sources, they did not expose the surface geometry much better than the standard 3D shading methods. We were not satisfied with these images and started looking for better methods; the result was the thickening approach combined with the four-dimensional rendering methods that we described in this chapter.

3.8.3 Specular Reflection

Specular highlights are a dominant source of intuitive shape information in 3D renderings, and so we would expect specularity to be equally important in our perception of four-dimensional geometry. The 4D analog of a specular shading contribution to the image of an object can be constructed in various ways[41]. Here we use a Phong specular shading approach[91] computed by replacing the lighting vector $\hat{L}$ by the normalized sum $\hat{B}$ of the camera direction $\hat{C}$ and the lighting vector $\hat{L}$, that is,

$$\hat{B} = \frac{\hat{L} + \hat{C}}{||\hat{L} + \hat{C}||} ,$$

then raising the appropriate dot product or determinant to a high power k. Thus we simply add to the shading equation a specular term such as

$$I_S = I_1 \left| \hat{B} \cdot \hat{n} \right|^k . \tag{31}$$

where $\hat{n}$ is the appropriate 4D normal and the dot product is set to zero when negative. The final shading equation is

$$I = I_A + I_D + I_S . \tag{32}$$

That is, we render the image into the view volume by summing the ambient, the diffuse and specular components of the intensity at each voxel.

3.8.4 Displaying the 3D View Volume

In order to display the 3D view volume on a 2D computer display, we must choose a volume rendering algorithm to map the 3D volume to a 2D image. We chose the simplest method, orthographic intensity summation, for this work. However, to get display with clear specularity and transparency, we found that we had to store the shading intensities information using two separate 3D arrays during the rendering process as described in the w-buffer algorithm. One array stores the specular component of each voxel, and the other one stores the sum of the ambient and diffuse component of each voxel. When all the cubic cells have been scan-converted and w-buffered into the 3D view volume, we sum the values of the voxels sharing the same (x, y) coordinate into 2D arrays, one for specular and one for diffuse information. That is, an entry with index (x, y) in each 2D array is the sum of all entries along the z-direction of the corresponding 3D array. We then determine the maximum intensities of these two 2D arrays and normalize them separately by dividing each entry of the arrays by their own maximum intensities. The shading information stored in these two 2D arrays can then be combined to create a transparent and specular 2D shaded image. For example, if we want to assign a blue color to the 3-manifolds, we assign the red component and green component of each pixel to have only the specular component, and the blue component of each pixel to have both a specular and diffuse component. The final 2D image will show bluish transparent objects with whitish highlights. Example of such rendering are presented in the next chapter.

The above method for displaying the 3D view volume works very well for individual still images. However, the shading intensities may become discontinuous during a sequence of animated images if we normalize each image separately. This is because we scale the shading intensities differently for each image. To avoid this problem, we can use the same global maximum intensities to normalize the shading intensities of all the images in the same animation sequence. However, if the global maximum intensities are not well chosen, some images may become too dark while some others may become too bright. Therefore, some experimental renderings are necessary before we determine the optimal global maximum intensities.

3.9 W-buffer Algorithms versus 4D Ray-tracing Algorithms

Just as we can extend the 3D z-buffer algorithms to the 4D w-buffer algorithms, the 3D ray-tracing algorithms can also be extended to work in 4D. Steven Hollasch from Arizona State University was implementing a 4D ray-tracer for his master thesis project[64] about the same time that we were implementing the w-buffer algorithms. A basic 4D ray-tracer algorithm, which is a simple extension of a 3D ray-tracer [41] can be described as follows:

```
4D Ray-tracing Algorithm
    select center of projection and view volume
    for each scan plane in view volume do
        for each voxel in scan plane do
        begin
            determine ray from center of projection through voxel
            for each object in scene do {
                if object is intersected with the ray and
                                the intersection is closest to the center of projection
                then
                    record intersection and object name }
            set voxel's color to that at closest object intersection
        end
```

One major advantage of 3D ray-tracers is that they can render realistic 2D images of 3D objects in a straightforward manner. Rendering problems such as hidden surface removal, reflection, refraction, and shadow generation can be handled at the same time by a recursive ray-tracing algorithm. But, do we need "realistic" volume images for 4D objects? While reflection and refraction are important to provide photorealism in 3D rendering, they seem less important in the rendering of four-dimensional mathematical objects. This is because at this stage we are still not familiar with interpreting volume images of four-dimensional objects, so reflection and refraction might cause more confusion than clarification. Geometric accuracy is probably what we should emphasize in four-dimensional rendering at this point. A simpler and non-recursive 4D ray-tracer that ignores the reflection and the refraction components may be more efficient and practical.

Without considering the reflection and the refraction components, the only main difference between the 4D ray-tracing algorithm and the w-buffer algorithm is the nesting order of their main loops. While the w-buffer algorithm handles one volume object at a time and only handles those voxels that the volume object covers, the 4D ray-tracing algorithm handles one voxel at a time and computes the ray-object intersection for all objects in the scene at that particular voxel. Therefore, either method might render a scene faster than the other depending on the number of volume objects in the scene, the actual sizes of these volume objects projected into the view volume, and the dimensions of the view volume.

One advantage of the 4D ray-tracing algorithm over the w-buffer algorithm is that it uses less computer memory storage. Since it processes one voxel at a time, it is not necessary to keep the 3D array for storing the 4D depths; it does, however, need the same number of 3D arrays to store the shading intensities.

Most of the latest graphics hardware supports a built-in z-buffer algorithm, but not a 3D ray-tracer algorithm because the z-buffer is simpler to implement in hardware. The w-buffer algorithm has the same advantages over the 4D ray-tracer. Another advantage of the w-buffer algorithm over the 4D ray-tracer is that its basic drawing unit is a triangle, because the tetrahedra are scan-converted into triangles during the rendering process. This is very important for hardware implementation, since triangle rendering is already available in all graphics hardware. Because of these advantages, we can predict that when the graphics hardware becomes powerful enough to have built-in four-dimensional rendering, it is more likely that a w-buffer algorithm will be available, rather than a 4D ray-tracer.

The other drawbacks of a 4D ray-tracer include:

- **Expensive ray-volume intersection calculation.** Just as the ray-surface intersection calculation is the time bottleneck for 3D ray-tracers, the ray-volume intersection calculation is the bottleneck for 4D ray-tracers. Ray-tracing causes even more problems in 4D because there are many more multiple intersections of 4D objects with 4D rays; furthermore, the number of rays to fire increases from N^2 to N^3.

- **Need different ray-volume intersection calculations for different volume objects.** The 4D ray-tracers give faster results than the w-buffer algorithms in rendering a scene if the number of volume objects in the scene is small. For some 3-manifolds, such as the hypersphere, it is possible to use special ray-volume intersection calculations during rendering. However, for many 3-manifolds, such special calculations are sometime difficult to compute, and more complex objects may have many intersection points with the same ray.

- **Ray-tetrahedron intersection is more complex and expensive.** When there is no special ray-volume intersection calculation method for a volume object, the object must be tessellated into tetrahedra as just in the w-buffer algorithm. The number of objects to be rendered in the scene is then increased by a very large factor. This requires a large number of ray-tetrahedron intersection calculations, which are much more complex and expensive than ray-triangle intersection calculations in a 3D ray-tracer[64].

The common disadvantages shared by both the w-buffer algorithm and the 4D ray-tracing algorithm are:

- Aliasing problems that causes the unwanted visual effect of jagged edges along the object's boundary. This is due to the point-sampling nature of these algorithms.
- Very expensive computational cost due to the complexity of the four-dimensional rendering.

3.10 Summary

In this chapter, we have proposed a family of techniques for creating intuitively informative shaded images of 4D mathematical objects. Standard three-dimensional rendering methods and shading algorithms are extended to work in the fourth dimension. 4D objects are viewed from a four-dimensional view point and illuminated by a four-dimensional point light source. They are projected to a 3D view volume instead of a 2D view plane. A "scan-plane" conversion method is implemented to scan convert a projected 3D tetrahedron to 2D triangles, which can then be further scan converted to the view volume by using standard scan-line algorithms. We replace the 3D "z-buffer" algorithm by a 4D "w-buffer" algorithm to handle occlusion. Shadows of a 4D object provide additional geometrical information and appear as darkened, smoke-like clouds in the view volume when a ray cast from the 4D light source is obstructed by the object.

In 4D, only volumes (3-manifolds) possess a well-defined normal vector at each point, permitting them to be shaded using the 4D shading methods. The problem of ill-defined normals for four-dimensional points, lines, and surfaces is solved by thickening them with 3-spheres, 2-spheres, and circles, respectively. To our knowledge, this is an original approach.

We emphasize generating images that reveal geometric structures of four-dimensional objects. While photorealism is something that we want to look for in 3D computer graphics, it does not quite make sense in 4D computer graphics yet. We are still at the beginning stages of 4D computer graphics.

Comparisons between parametric representations and implicit representations are given. We also compare the complexity of 3D rendering and 4D rendering, and show why it is still very difficult to have real-time 4D rendering on current hardware. Differences and similarities of the w-buffer algorithms and the 4D ray-tracing algorithms are discussed. When computer hardware becomes advanced enough to have built-in four-dimensional rendering algorithms, w-buffer algorithms are more likely to be chosen over the 4D ray-tracer algorithms due to their ease of implementation.

Chapter 4

Examples of Four-Dimensional Rendering

In this chapter, we illustrate our basic concepts presented in Chapter 3 with some classic examples from four-dimensional topology. Examples of volumes, surfaces, curves and points are given in the first four sections. Section 4.5 discusses methods that improve the image interpretability. A visualization paradigm for problems such as the representation of four-dimensional mathematical objects is proposed in Section 4.6. Section 4.7 provides a summary of this chapter.

4.1 Examples of Volumes in 4D

Volumes, or 3-manifolds, embedded in a 4D world have unique normal vectors at each point, therefore they can be rendered by the four-dimensional shading methods in a straightforward manner. We now proceed to give two examples of such renderings.

Hyperspheres and Hypercubes as Superquadrics. The 3-sphere and the hypercube or tesseract are classic subjects of study in 4D. Here we examine them together using the fact that the *superquadric*[16]

$$|x|^{\kappa} + |y|^{\kappa} + |z|^{\kappa} + |w|^{\kappa} = r^{\kappa},$$

reduces to the 3-sphere when $\kappa = 2$, but asymptotically approaches a hypercube or tesseract as $\kappa \rightarrow \infty$.

The following parametric equations are used to represent superquadrics:

$$\begin{aligned}
x &= \mathrm{sign}(\cos\beta\cos\alpha\cos\gamma)\,|\cos\beta\cos\alpha\cos\gamma|^{\frac{2}{\kappa}}\\
y &= \mathrm{sign}(\sin\beta\cos\alpha\cos\gamma)\,|\sin\beta\cos\alpha\cos\gamma|^{\frac{2}{\kappa}}\\
z &= \mathrm{sign}(\sin\alpha\cos\gamma)\,|\sin\alpha\cos\gamma|^{\frac{2}{\kappa}}\\
w &= \mathrm{sign}(\sin\gamma)\,|\sin\gamma|^{\frac{2}{\kappa}}\,. \qquad (33)
\end{aligned}$$

Since superquadrics have a nice implicit equation, the normal $\vec{N}$ can be computed by taking the gradient of the implicit equation:

$$\begin{aligned}
N_x &= \kappa\,\mathrm{sign}(\cos\beta\cos\alpha\cos\gamma)\,|\cos\beta\cos\alpha\cos\gamma|^{(\kappa-1)\times\frac{2}{\kappa}}\\
N_y &= \kappa\,\mathrm{sign}(\sin\beta\cos\alpha\cos\gamma)\,|\sin\beta\cos\alpha\cos\gamma|^{(\kappa-1)\times\frac{2}{\kappa}}\\
N_z &= \kappa\,\mathrm{sign}(\sin\alpha\cos\gamma)\,|\sin\alpha\cos\gamma|^{(\kappa-1)\times\frac{2}{\kappa}}\\
N_w &= \kappa\,\mathrm{sign}(\sin\gamma)\,|\sin\gamma|^{(\kappa-1)\times\frac{2}{\kappa}}\,. \qquad (34)
\end{aligned}$$

In Figure 35, we show how the rendered object evolves in appearance as it makes the transition from a hypersphere to a hypercube. The objects are illuminated by a 4D light and rendered from a particular 4D viewpoint using specular shading, resulting in a volume image whose interior is made visible here by transparently projecting it to the image plane as usual. During the transition, the specular region changes from a roundish volume inside the 3-sphere to a cube in the volume image. Wire frame images of the tesseract are well known[5, 10], but using 4D lighting to produce shaded images showing the evolutionary relationship between the 3-sphere and the tesseract is unique.

Figure 36 shows how the hypercube's specular shading changes as the viewpoint rotates in 4D around the scene center. The hypercube has been slightly smoothed by representing it as a 4D superquadric given by the equation $x^6 + y^6 + z^6 + w^6 = 1$.

Figure 35: This figure shows four stages in the superquadric interpolation from a three-sphere to a hypercube or tesseract.

Observe that a tesseract is specular on entire 3D *cubes*; this is exactly analogous to the fact that an ordinary 3D cube shows specularities on individual *square* faces. We also observe that, just as an edge-on face of a 3D cube reduces to a line when it is aligned with the viewing angle, the cubic volumes forming the bounding "hyperfaces" of the hypercube approach planar polygons embedded in the view volume when aligned with the 4D view direction.

Two animation sequences that animate the above figures have been recorded in a computer animated video, entitled "FourSight"[57]. This video provides an elementary introduction to the production and interpretation of images generated using the 4D rendering methods. It was selected for exhibition in the Electronic Theater Computer Graphics Screening Room of SIGGRAPH '92, the leading computer graphics conference.

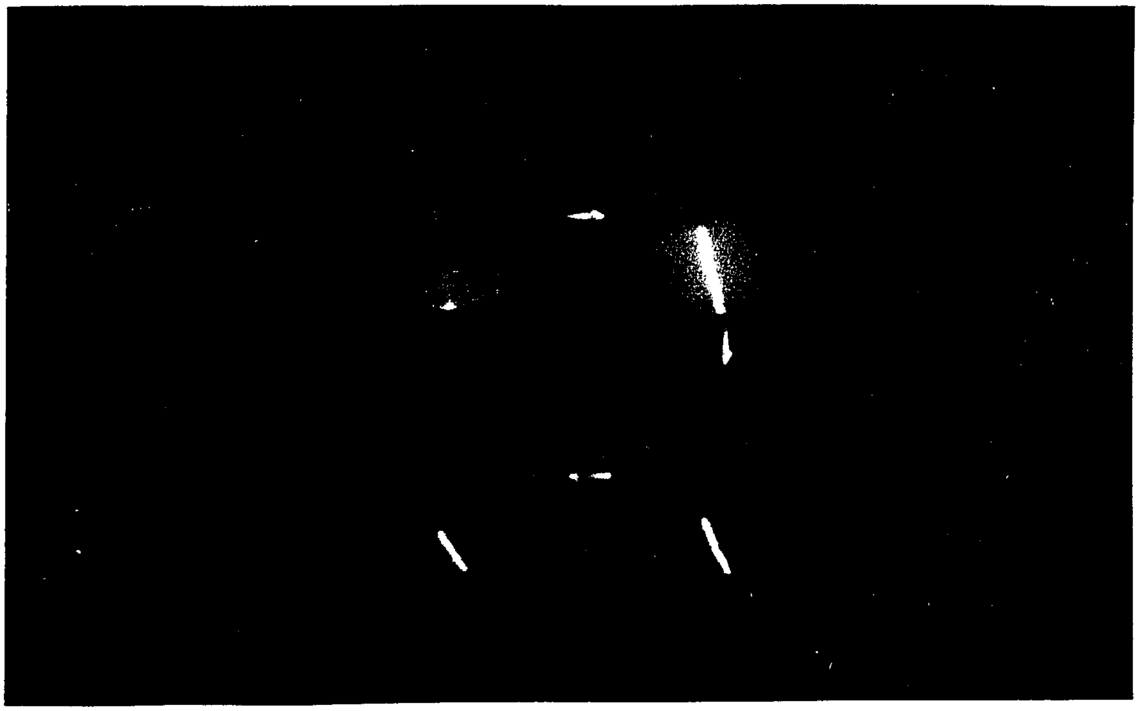


Figure 36: A succession of viewpoints of a four-dimensional hypercube as the 4D viewpoint rotates around the center of the scene.

4.2 Examples of Surfaces in 4D

We next show our results for some classic examples of surfaces from four-dimensional topology.

4.2.1 General Approach for Surfaces

As we have noted, surfaces are intrinsically well-defined for shaded rendering in 3D, but not in 4D. For surfaces in 4D, we must identify the normal plane at each point, and place a circle or ring with a small radius in this plane centered at the point. The radial direction of the ring attached at each point can be understood by taking the local tangent vectors $P^\mu(u,v)$ and $Q^\mu(u,v)$ at a point $x^\mu(u,v)$ of a surface parameterized by the two variables (u,v) and using a Gram-Schmidt procedure to find candidates for an orthogonal coordinate basis $(N_1^\mu,\ N_2^\mu)$ of the plane perpendicular to the surface.

That is, we require $\vec{N}_i \cdot \vec{P} = \vec{N}_i \cdot \vec{Q} = \vec{N}_1 \cdot \vec{N}_2 = 0$. Once $N_1^\mu(u,v)$ and $N_2^\mu(u,v)$ are found, they are normalized and the four-dimensional normal vector

$$\hat{n}^\mu(u,v,\theta) = \hat{N}_1^\mu(u,v)\cos\theta + \hat{N}_2^\mu(u,v)\sin\theta \tag{35}$$

is substituted into the 4D version of Eqs. (27) and Eqs. (31) given in Chapter 3. The three variables parameterize the corresponding 3-manifold

$$X^\mu(u,v,\theta) = x^\mu(u,v) + r\hat{n}^\mu(u,v,\theta) \tag{36}$$

for which $\hat{n}^\mu(u,v,\theta)$ is the current normal direction and r is the radius of the attached circle. One technical point is that it may be difficult to find a smooth transition among coordinate patches for the 3-manifold over the entire surface; in the case of one-sided surfaces, for example, one may need to cover the surface twice.

4.2.2 Steiner Surface

In an appendix to a chapter of their book *Geometry and the Imagination*, Hilbert and Cohn-Vossen[61] give a set of equations for a nonsingular embedding of the projective plane in four dimensions known as Steiner's surface (see also Francis[42]). The projective plane is the space obtained by identifying diametrically opposite points on the 2-sphere. We represent this surface parametrically by the equations

$$\begin{aligned}
x &= \cos^2 u \cos^2 v - \sin^2 u \cos^2 v \\
y &= \sin u \cos u \cos^2 v \\
z &= \cos u \sin v \cos v \\
w &= \sin u \sin v \cos v \ ,
\end{aligned}$$

where $0 \leq u < \pi$, $0 \leq v < \pi$. Since Steiner's surface is one-sided (the projective plane), it is necessary to cover the surface twice in the parametrization in order to generate smooth transitions among sections of the 3-manifold during four-dimensional rendering.

Thickening process and normal computation. The thickening process and the normal computation for Steiner's surface are described as follows:

1. The local tangent vectors $P^{\mu}(u,v)$ and $Q^{\mu}(u,v)$ at a point $x^{\mu}(u,v)$ are computed by taking partial derivatives with respect to u and v, and their normalized values are $T_1(u,v), T_2(u,v)$.

2. In order to determine the normal plane, we need to pick a vector to be orthonormalized with $T_1(u,v), T_2(u,v)$ in the Gram-Schmidt orthonormalization process. We tried using $N_1(u,v) = x^{\mu}(u,v)$ but that resulted in images with glitches. The projective plane is a 2-sphere with opposite points identified; in the chosen parameterization, the origin is at a very singular point of the 3D embedding (x,y,z); this makes $N_1(u,v) = x^{\mu}(u,v)$ a very bad choice for a candidate axis in the normal plane. This problem is overcome by using $N_1(u,v) = x^{\mu}(u,v) - [0.5, 0.0, 0.0, 0.0]$. We then orthonormalize the three vectors $T_1(u,v), T_2(u,v)$, and $N_1(u,v)$ using the Gram-Schmidt algorithm to form a local orthonormal triad.

3. A second normal direction $N_2(u,v)$ is needed to form an orthogonal coordinate basis of the plane perpendicular to the surface. $N_2(u,v)$ can be determined by taking the 4D cross product (see Appendix B) of the three orthonormalized vectors obtained in the previous step.

4. The final normal vector $\hat{n}^{\mu}(u,v,\theta)$ and the coordinate of the point $X^{\mu}(u,v,\theta)$ are computed by using Eq. (35) and Eq. (36).

Methods for checking the normal computation. Although we do not have the mathematical knowledge to prove that the normal bundle we constructed is in fact unique and well-behaved globally over the whole surface, the following methods can be used to detect any irregularities of the defined normal planes:

- **Interactive constraint-based technique for displaying the tangent vectors and normal plane at a grid point.** The relevant interactive techniques for displaying tangent vectors and normal plane at a grid point have been presented in Chapter 2 under Section 2.4. Figure 21 illustrates the application of such techniques to the Steiner surface. By moving the mouse cursor around, the user can observe flippings of the normal circles that might occur if there were irregularities in the constructed normal bundle. None were found.

- **Numerical methods to detect irregularities on the normal bundle.** To numerically check for irregularities on the normal bundle, we first sum over the whole lattice the 4D dot product of $N_1(u,v)$ with the N_1 of its four neighbors, and then divide the sum by 4. This number is +1 if they are all align, 0.75 if one is perpendicular, 0.5 if one is totally opposite, and so on. Perform the same computation for $N_2(u,v)$ at all grid points. Simply by printing out the values of (u,v) where irregularities occur or by plotting these numbers graphically using two 2D histograms, we can easily detect the irregularities on the normal computation. None were found.

4D rendered images of the Steiner's surface. The interesting fact about this equation is that when the corresponding surface is rotated in four dimensions, it changes smoothly from the classic "crosscap" form of the projective plane to Steiner's Roman surface, depending upon the rotation axes chosen.

In Figure 29 of Chapter 3, we show views of this surface from different 4D viewpoints produced by projecting the surface and its shadow to 3D before rendering using conventional 3D lighting methods. We see how the use of shadows can greatly clarify the nature of the projection that is taking place from 4D to 3D; if we animate this rotation, we can simultaneously see the two states of the shape and watch them exchange places as the rotation proceeds[56].

Figure 37 shows stereo pairs of the shaded volume rendering of the Steiner surface when each point is thickened by placing a small specular circle in the normal plane and the resulting 3-manifold is illuminated by a single 4D point light source. To distinguish 4D from 3D effects, we render the full 4D thickened object along with the 3D rendering of its projection. We give images for two separate 4D rotation angles, showing the dramatic changes in the apparent shape and its 4D specular reflections as the orientation evolves.

In the stereo pairs, the images on the left side are for the right eye, and the images on the right side are for the left eye; to view in stereo, place a finger on the page between the two images and focus on the finger while moving it towards your nose until you see three distinct images — the center image should appear in stereo.

4.2.3 4D Torus

Next, we look at the torus, another 4D surface described in Hilbert and Cohn-Vossen[61]. The familiar torus as embedded in 3D is strongly curved; the 4D embedding that we use is flat, like the surface of a cylinder in 3D. The equations in 4D are given simply by taking two separate circles, one in the first two coordinates and the other in the second two coordinates of the 4D space:

$$\begin{aligned} x &= \cos u \\ y &= \sin u \\ z &= \cos v \end{aligned}$$

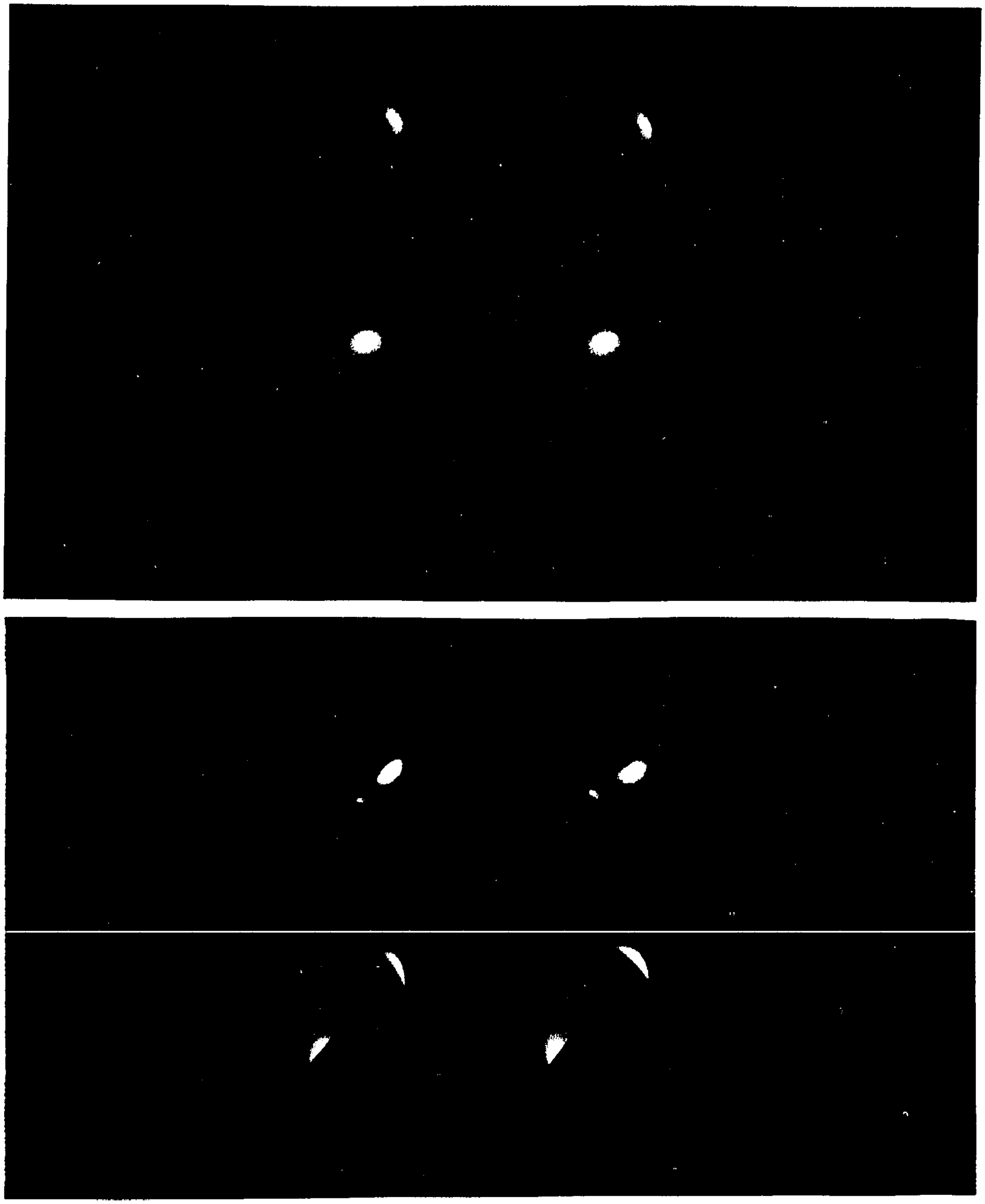


Figure 37: Cross-eyed stereo pairs of two different 4D views of a Steiner surface (a) the crosscap, (b) Steiner's Roman surface. In each view, the upper image pair is a thickened Steiner surface rendered in 4D with 4D lighting. The lower pair in each view is a 3D projection of the unthickened surface rendered using 3D methods available in *Wavefront*.

$$w = \sin v \ .$$

Thickening process and normal computation. The normal computation for this surface is much simpler than for Steiner's surface. Since the first two coordinates and the last two coordinates of the surface are two separate circles, the orthogonal and normalized normals of the two separate circles are the ideal choice for $N_1(u,v)$ and $N_2(u,v)$:

$$\begin{aligned} N_1(u,v) &= [\cos u, \sin u, 0, 0] \\ N_2(u,v) &= [0, 0, \cos v, \sin v] \ . \end{aligned}$$

The final normal vector $\hat{n}^\mu(u,v,\theta)$ and the coordinate of the point $X^\mu(u,v,\theta)$ can be computed using Eq. (35) and Eq. (36).

4D rendered images of the 4D Torus. The 3D projection of the torus object (with 3D rendering) and its shadow are shown in Figure 30 of Chapter 3 for two viewpoints in a 4D rotation sequence. Again, an animation of this rotation shows how the object and its shadow continually interchange places. Using only the 3D projection, it is very difficult to grasp the flat nature of the intrinsic surface in 4D.

Figure 38 shows stereo pairs of the torus rendered in 4D with 4D lighting at two different 4D orientations; the corresponding 3D projection with 3D lighting is shown in the lower portion of each view for comparison. Here the 4D specularities again give a dramatically different picture, and provide the beginnings of some insight into the flatness of the surface in 4D. Additional interesting features can be observed when this surface is rotated in 4D about the yw plane[57].

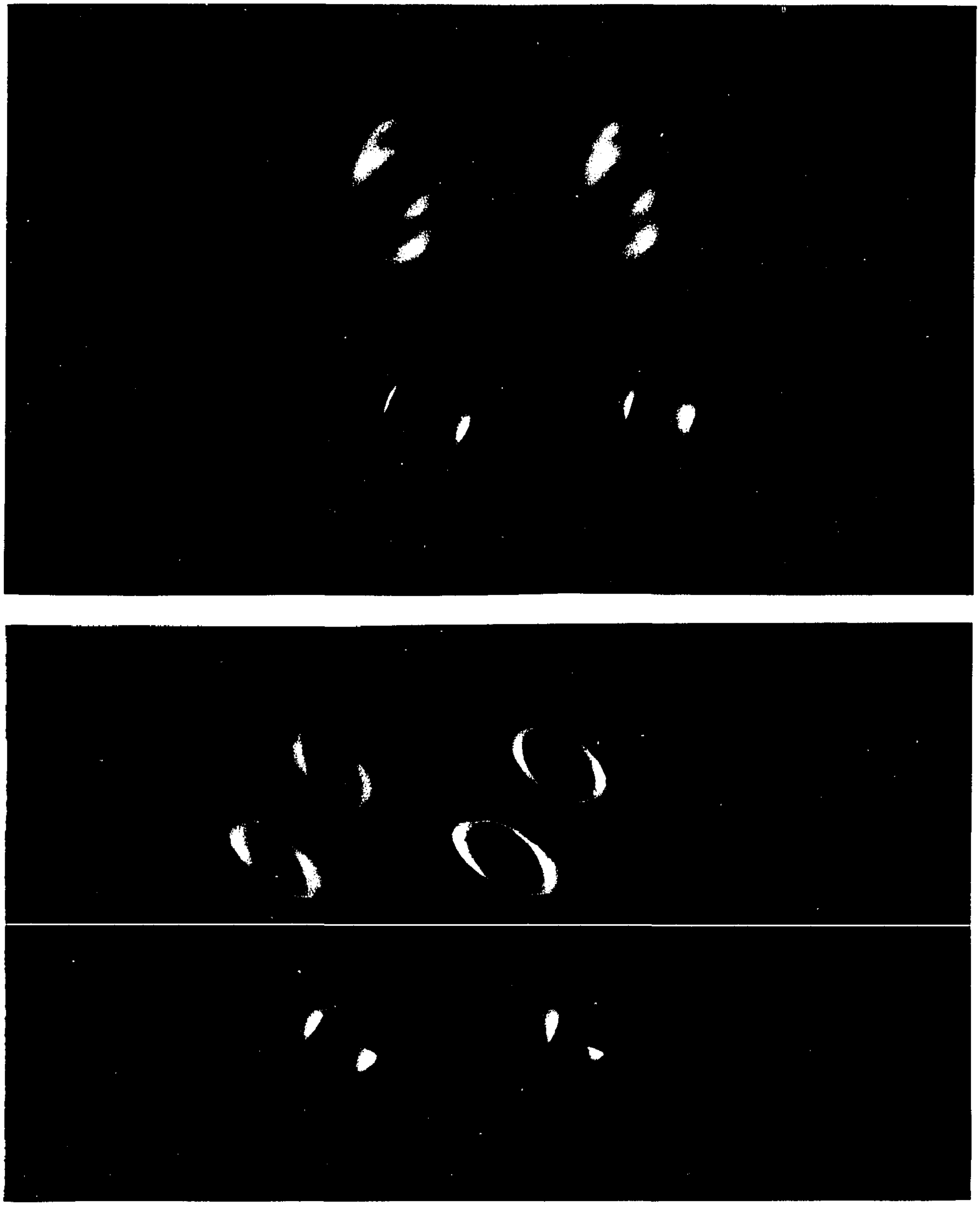


Figure 38: Cross-eyed stereo pairs of two different 4D views of the four-torus. In each view, the upper image pair is a thickened four-torus rendered in 4D with 4D lighting. The lower pair in each view is a 3D projection of the unthickened surface rendered using 3D methods available in *Wavefront*.

4.2.4 Knotted Sphere

Finally, we examine a 4D version of a 3D knot. Knotted curves can only exist in 3D, since in 4D any single knot clashing point can be undone by rotating the displaced clashing point of the knot around the origin by 180 degrees in a plane containing the fourth dimension and the knot displacement. In 4D, however, one can have two *surfaces* that intersect at a point and cannot pass by one another; thus in 4D, knots are formed by closed surfaces such as the sphere, rather than by closed curves (see, for example, Weeks[103]).

We construct a knotted sphere by taking an overhand knot with two free ends and rotating that knot into the fourth dimension about an axis containing both ends. This is known as the "spun trefoil." After one full rotation, the knot rejoins itself and forms a closed surface; a slice through this surface by a hyperplane in 4D gives a knotted curve in 3D.

The equations for the overhand knot that we used (adapted from the cable knot equation given in *fnord*[35]) are given as follows:

$$\begin{aligned} x(u) &= (\sqrt{2}+\cos(3u))\cos(2u) \\ y(u) &= (\sqrt{2}+\cos(3u))\cos(2u) \\ z(u) &= \sin(3u) \\ w(u) &= 0\,, \end{aligned} \tag{37}$$

where $\frac{\pi}{3} \leq u \leq \frac{5\pi}{3}$.

The equations for the knotted sphere are then defined by displacing Eq. (37) along the x-axis, adding a top and bottom link to the "north pole" and "south pole", and rotating the final equations in the x-w plane. In terms of x, y, and z defined in

Eq. (37), the equations of the knotted sphere are given as follows:

$$\begin{aligned} x'(u,v) &= (x(u)+x_0)\cos(v) \\ y'(u,v) &= y(u) \\ z'(u,v) &= (x(u)+x_0)\sin(v) \\ w'(u,v) &= z(u)\,, \end{aligned} \tag{38}$$

where x_0 is the displacement along the x-axis, and $0 \leq v < 2\pi$. We used the same thickening approach described above to attach a circle at each point of the knotted sphere. The result was a 3-manifold that could be rendered by our 4D rendering methods.

In 3D rendering, the structures of knotted curves can be clearly observed if we thicken them into knotted ropes and apply the z-buffer algorithm to handle the occlusions. Similarly, the structures of knotted spheres become clearer if we attach a circle at each point of the surface and render using the w-buffer algorithm.

In Figure 39, we present a stereo rendering of this knotted sphere in 4D, with the 3D projection of the surface below for comparison.

Two sequences of stereo animation that animate Figure 37 and Figure 38 have been collected into a special multimedia issue of IEEE *Computer Graphics and Applications*[58]. We have also recorded three non-stereo animation sequences of these surfaces in [57].

4.3 Example of Curves in 4D

Curves are intrinsically well-defined for shaded rendering in 2D, but not in 3D and 4D. As shown in Figure 25 of Chapter 3, we can sweep out a curve in 3D with a circle to transform it into a cylindrical tube. In order to transform a 4D curve into

Figure 39: A cross-eyed stereo views of the knotted 2-sphere rendered in 4D with 4D lighting (top), and also rendered in its 3D projection with 3D lighting (bottom).

a 3-manifold so that it can be rendered using our techniques, we sweep it out with a 2-sphere.

As an example, a 3-manifold given by the product of an ellipse and a 2-sphere can be represented using the following parametric equations:

$$\begin{aligned} x &= cos(\gamma)(R + rcos(\alpha)sin(\beta)) \\ y &= sin(\gamma)(Rsin(\pi/3) + rcos(\alpha)sin(\beta)) \\ z &= rsin(\alpha)sin(\beta) \\ w &= rcos(\beta) \end{aligned}$$

where R, $Rsin(\pi/3)$ are the lengths of the major and minor axes of the ellipse, r is the radius of the 2-sphere, $\alpha \in [-\pi/2, \pi/2]$, $\beta \in [0, 2\pi]$, and $\gamma \in [0, 2\pi]$.

One test of the effectiveness of shading is to compare shaded images of objects having identical projections from some viewpoint. One example is the comparison of a true elliptical solid with a tilted circular solid. Such objects are 3 manifolds given by the product of a circle and a 2-sphere. We tilt the object back into the fourth dimension w so that its 3D volume rendering is indistinguishable in shape from the an elliptical 3-manifold lying in the $w = 0$ hyperplane. The equations of this tilted circular solid are:

$$\begin{aligned} x &= cos(\gamma)(R + rcos(\alpha)sin(\beta)) \\ y &= (sin(\gamma)(R + rcos(\alpha)sin(\beta)))cos(\pi/3) + rcos(\beta)sin(\pi/3) \\ z &= rsin(\alpha)sin(\beta) \\ w &= rcos(\beta)cos(\pi/3) - sin(\pi/3)(sin(\gamma)(R + rcos(\alpha)sin(\beta))) \ , \end{aligned}$$

where R is the radius of the circle, r is the radius of the 2-sphere, $\alpha \in [-\pi/2, \pi/2]$, $\beta \in [0, 2\pi]$, and $\gamma \in [0, 2\pi]$.

In Figure 40, we show the two 3-manifolds in three different 4D orientations. Four-dimensional shading and specularity are seen to introduce distinguishing features.

4.4 Example of Points in 4D

Finally, points in 4D can only be shaded if we expand them into 3-spheres. As an example, to render a point $P = (P_x, P_y, P_z, P_w)$, we can use the following parametric equations for a 3-sphere whose center is P and whose radius is r :

$$\begin{aligned} x &= P_x + rcos(\beta)sin(\alpha)cos(\gamma) \\ y &= P_y + rsin(\beta)sin(\alpha)cos(\gamma) \\ z &= P_z + rsin(\alpha)cos(\gamma) \end{aligned}$$

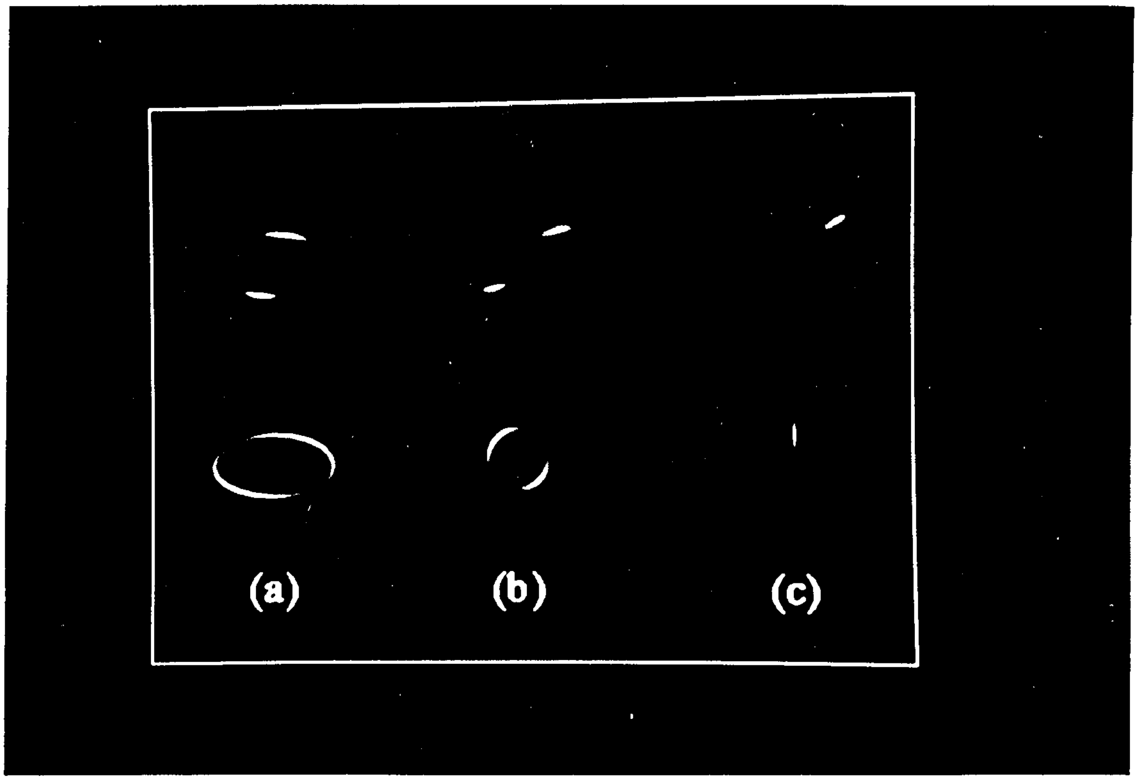


Figure 40: (a) (above) A 3-manifold given by the cross product of a ring and a 2-sphere, tilted back into the fourth dimension w so that its 3D volume rendering is indistinguishable in shape from the elliptical 3-manifold (below) lying in the $w = 0$ hyperplane. Four-dimensional shading and specularity introduce features that distinguish the two 3-manifolds. (b,c) Rotating in the x-w plane reveals the difference between the two manifolds.

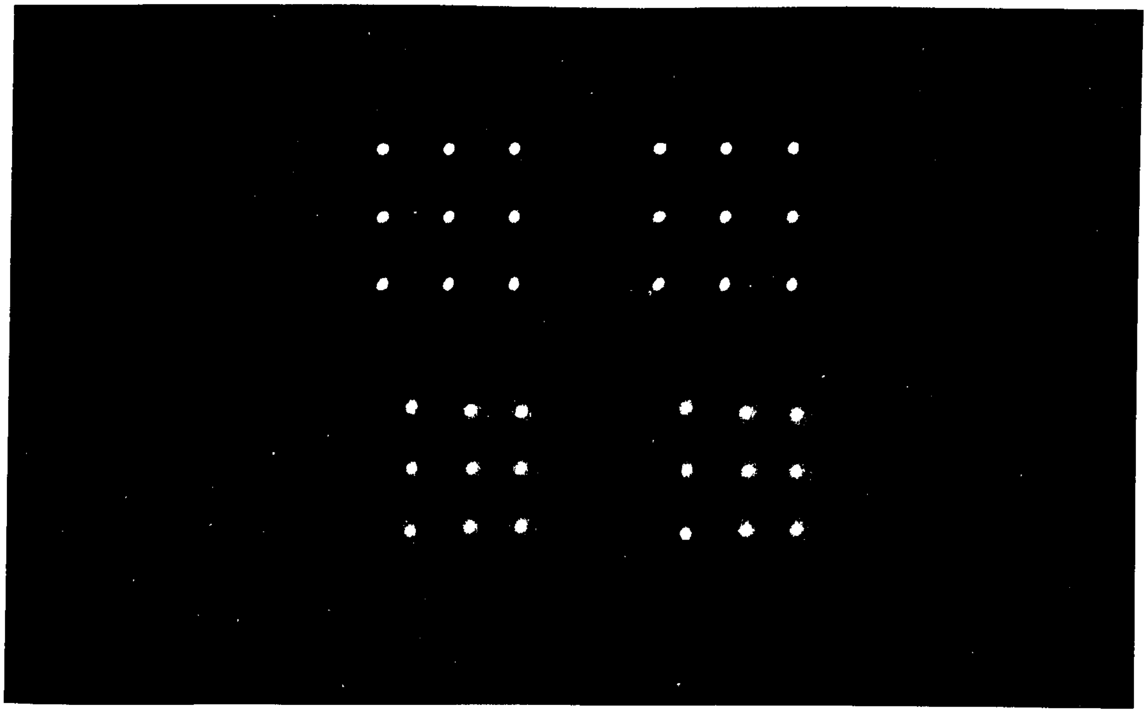


Figure 41: Stereographic image of points volume-rendered as 3-spheres in 4D (top) and depicted as ordinary spheres in 3D (bottom).

$$w = P_w + rsin(\gamma)$$

where $\frac{-\pi}{2} \leq \alpha \leq \frac{\pi}{2}, 0 \leq \beta \leq 2\pi, \frac{-\pi}{2} \leq \gamma \leq \frac{\pi}{2}$. In Figure 41, we show a cross-eyed stereo image of a collection of 4D points that are volume-rendered as shiny 3-spheres; for comparison, we show a collection of 3D points rendered in 3D as shiny ordinary spheres resembling Christmas tree ornaments. The 3-spheres in the volume rendering each have a shiny spot that changes its 3D position within the volume when the light strikes from a different 4D angle; the ordinary spheres have shiny spots that must stay on the surface of the 2-sphere, and do not penetrate the volume.

4.5 Improving Image Interpretability

The quality of the image produced by the 4D rendering techniques can be improved by using the following methods:

- **Antialiasing.** Aliasing refers to the unwanted visual effect of jagged edges along the object's boundary in computer generated images. One major problem with the 3D z-buffer algorithm is aliasing, due to the point-sampling nature of this algorithm[67]. The w-buffer algorithm, which is a straightforward extension of the z-buffer algorithm to the next higher dimension, shares exactly the same problem. Many antialiasing techniques have been developed to correct the aliasing effect in 3D graphics[29, 37, 33]. Just as we can extend the z-buffer algorithm to the w-buffer algorithm, we can also find a suitable 3D antialiasing algorithm and extend it to work in four dimensions. However, a more time-effective alternative is to post-process the final 2D image generated by the w-buffer algorithm. Simply applying some 2D filtering algorithms such as the Gaussian filter[38, 97, 47] can improve the image quality substantially.

- **Gamma correction.** Color computations are based on linear intensity values. Unfortunately, the response of typical video color monitors and of the human visual system is nonlinear. Gamma correction can be used to improve the video image by correcting for the non-linear brightness response of the color phosphors used in video monitors. The monitor correction function is an exponential of the form:

  $$I_\gamma = I^{1.0/\gamma}\ , \tag{39}$$

  where I is the intensity computed using the shading equations, and I_γ is the gamma-corrected intensity. γ represents the nonlinearity of the monitor, which usually is in the range 2.0 to 3.0. Intensities will be brighter if γ is larger and will be less intense if γ is smaller. A correct γ value, which can be found through experimental renderings, is essential for the intensities to appear reasonable. More details about gamma correction are given in [51].

- **Radius of spheres attached to lower-dimensional objects.** To make lower-dimensional objects such as points, curves, and surfaces renderable in 4D, we attach *spheres* of different dimensions at each point of the objects. The radius size must be chosen appropriately in order to get the best result. If the radius is too small, some topological structures of the objects may not show up. However, if the radius is too large, some important topological features of the objects may be obstructed by the expanded spheres. The ideal radius size can also be determined by experiment.

- **Animation with varying view angle and object orientation.** The same mathematical object, when viewed from different viewing angles and with different orientations in 4D, can appear as two different objects (for example see Figure 37). Interesting features of the mathematical objects can be revealed by creating animation sequences with different viewing angles and four-dimensional orientations.

- **Animation with moving 4D light.** Instead of moving the objects and the 4D camera, we can move the 4D light source to generate animation sequences. Such animations will show clearly what the objects look like under different 4D lighting conditions, which in turn provides more insight into their four-dimensional geometrical structure.

- **Stereo Animation.** In any dimension N, a pair of images each consisting of intensity values in an $(N-1)$-dimensional hyperplane is sufficient to determine the N-dimensional position of a known scene point appearing in both images.

If we know the focal centers and film (hyper)planes of both "cameras," the two lines joining the focal centers to the *images* of the same scene point must pass through the *actual* scene point (see, e.g., [65]).

In 4D, projected rays intersect the film volume at an interior point, rather than in a line as one might expect; a 4D line can "go directly" to an interior volume pixel in the same way as a 3D line can pass through a single interior pixel in the 2D film plane. Thus the principles of stereography carry over to all dimensions, and two aligned 3D volume images are sufficient to determine the 4D coordinates of a visible point on a 4D object.

Stereo animation of these images provides intriguing amount of orientation cues and motion parallax cues at the same time. Properties that can be not confirmed in standard animations, such as the rigidity of four-dimensional rotations of mathematical objects, can be clarified through stereo animation.

4.6 A Visualization Paradigm

Two open problems are raised by our work: one concerns the interpretability of the 4D images, and the other concerns the learnability of 4D perception[55]. These problems are described as follows:

- **Interpreting 4D Images.** In our work, we have not proven that the images we produced are *interpretable*. That means we have not proven that the viewer can make a correct mental model of the mathematical objects by looking at their corresponding images created by our rendering methods. A possible solution to this problem is to extend three-dimensional computer vision techniques to work in four dimensions. The shape-from-shading methods such as that of Ikeuchi and Horn[66], and the photometric stereo technique of Woodham[108] seem particularly appropriate. The 4D extension of these methods may be useful for *reinterpreting* 3D volume renderings as images of four-dimensional objects.

Verifying Learnability. What is the cognitive nature of 4D perception? In this work, we have not shown that the necessary interpretation techniques can be taught to human users if they were not already known, and we have not verified that the desired interpretation of the images is *cognitively feasible* within the limitations of the human intellect. If these 4D images are interpretable, what are the most effective techniques for teaching an understanding of such images? From our experience with the 3D world and 3D lighting, we have learned to apply a few ad hoc constraints that let us deduce reliably the 3D content of shaded 2D images of a scene, particularly if shadows, stereo, and motion parallax are available. Can humans acquire facility with four dimensions with enough practice? Just as we can "see" true 3D depth in a monocular sequence of images from motion parallax, is it possible that we can learn to use our binocular vision on volume renderings combined with oscillating motion parallax in the fourth dimension to "see" true four-dimensional depth? Finally, could a single shaded volume rendering in stereo be sufficient for 4D understanding, just as single photographs of a 3D scene are? We need expertise in areas such as cognitive science, computer vision, user interface design, computer-aided instruction, and visualization in order to provide satisfying solutions to the above problems.

Visualization Principle. With all these unanswered questions in mind, we would like to propose a visualization paradigm for problems such as the representation of four-dimensional mathematical objects. This paradigm is based on what we call the *Visualization Principle:*

> ***A useful data depiction must allow the viewer to reconstruct a consistent and relevant model of the original data.***

That is, images produced by a visualization procedure are assumed to be based on some type of underlying experimental or mathematical data. The purpose of such an image is to allow the user to grasp some fact or property of the data that is not intuitively obvious from reading the data values in the form of printed text or formulas. If the viewer cannot produce an accurate mental model for the relevant properties of the data from the image alone, the image is not serving the purposes of the visualization process.

We suggest that a visualization researcher must at least consider the following series of procedures in order to be able to guarantee that the Visualization Principle is satisfied:

- **Generate Images.** In this stage, the researcher studies the available data and the classes of information that are relevant for display, and then proposes various data depiction techniques to elucidate the underlying models for the data. This is as far as we were able to proceed in this research.

- **Interpret Images.** Next, one must find a convincing argument that the *images alone* contain sufficient information to allow the reconstruction of the desired data models. Specific, computer-implementable model reconstruction algorithms should be suggested and, if feasible, tested in order to discover unexpected ambiguities.

- **Teach Interpretation.** The existence of automatable algorithms to reconstruct the data model from the image does not necessarily mean that humans can perform this reconstruction without training. One may need, for example, to build interactive tools that help a human acquire an intuitive understanding of the reconstruction procedure. Typically, one might accomplish this by establishing a simulated environment (sometimes called a "virtual world") that incorporates tasks that cannot be solved successfully without mastery of the reconstruction algorithm.

- **Verify Learnability.** Finally, it is not necessarily true that any given reconstruction method *can* be learned by a human, even with extensive training. One must therefore evaluate both human ability to intuitively employ the proposed reconstruction algorithms and the effectiveness of any proposed teaching methods. This completes the cycle: if one can prove that humans can learn the reconstruction, and that the method can be taught effectively, the visualization procedure is guaranteed to meet its goals.

4.7 Summary

Examples of our 4D rendering methods are given in this chapter. The cross-eyed stereo pairs of both the conventional projected 3D rendering and the new 4D rendering approach provide viewers a clear comparison between the two methods. Image quality of the 4D rendering can be improved through techniques such as antialiasing, gamma correction, and changing radius size of the attaching sphere for lower-dimensional objects. Since real-time animation of the 4D rendering is still not possible on current hardware due to its intensive computing and memory usage, recording of animation sequences on a video tape is still the best alternative to provide viewers a better understanding of these images. Animation sequences can be generated by moving the objects, the 4D camera, and/or the 4D light source. Stereo pair animations are particularly useful for viewing volume images generated by the 4D rendering methods.

Various open questions show up in this work that prevent us from claiming that the work is complete. Problems such as interpretability of these 4D images and the learnability of 4D perception have yet to be looked into. A four-step visualization paradigm based on the *Visualization Principle* has been proposed, and it consists of image generation, image interpretation, teaching of interpretation, and verification of learnability.

In the next chapter, we present extensions and adaptations of these 4D rendering techniques and apply them to visualize volumetric data generated from other scientific applications.

Chapter 5

Four-Dimensional Rendering in Volume Visualization

Volume visualization, which deals with the representation, manipulation, and rendering of volumetric data (three-dimensional scalar fields), is a rapidly growing area within scientific visualization. Volumetric data can be generated from medical applications, molecular modeling, astronomical measurement and simulation, meteorological predictions, industrial computer tomography, finite element analysis, and many other sources. Whether these volumetric data are collected through actual physical measurements, computed from numerical simulations, or generated from scalar functions of three variables $w = f(x, y, z)$, human viewers can better understand the data when they are represented as interpretable computer images instead of just a huge collection of numbers. Different volume visualization techniques produce different visual effects and emphasize different aspects of the data[83, 68, 76].

Although interactive mathematical visualization is the main focus of this work, we also emphasize generalizing the visualization tools so that they can be used in other scientific applications. In this chapter, we present some new techniques for visualizing 3D scalar fields in a variety of ways. Some of these techniques are simple extensions of the 3D elevation map methods and the others are generalizations of the four-dimensional rendering methods described in the previous chapters.

In Section 5.1, we give a brief introduction to 3D elevation map techniques for handling 2D scalar fields and to volume visualization techniques for handling 3D scalar fields. Section 5.2 presents the extensions of 3D elevation map techniques to the corresponding 4D elevation map techniques. These 4D elevation map techniques include grid planes, four-dimensional rotations, 4D pseudocolor volume grids, 4D bare volume grids, and 4D volume grids with hidden volumes removed. In Section 5.3, we present an adapted version of the four-dimensional rendering method that generates shaded volume images of 3D scalar fields. Section 5.4 gives a few examples of applications to real data and Section 5.5 provides a summary of this chapter.

5.1 Introduction

In this section, we would like to give a brief introduction to two classes of visualization techniques. The first class is the elevation map techniques for handling 2D scalar fields. We show how to extend these techniques to handle 3D scalar fields in the next sections. The second class consists of standard volume visualization techniques for handling 3D scalar fields.

5.1.1 Elevation Map Techniques

Elevation maps of 1D and 2D scalar fields have been used in scientific applications for a long time. Scalar functions of one variable, $y = f(x)$, can be viewed as curves in a 2D Cartesian plane, and scalar functions of two variables, $z = f(x, y)$ can be viewed as 3D surfaces seen from oblique angles in 3D space. Analogously, scalar functions of three variables, $w = f(x, y, z)$ are elevation maps in a 4D coordinate system, with the value of the function w identified with the fourth dimension.

Surfaces of the form $z = f(x, y)$ in 3D space may be represented graphically in a variety of ways, including contour plots in a plane, elevation-keyed pseudocolor plots in a plane, oblique views of rectangular grids projected onto the surface, oblique views with elevation-keyed surface pseudocolors, and shaded surface rendering. More detail about these techniques are given and examples are presented in Section 5.2. There are many commercial or public domain graphics software systems, such as NCSA Image, Mathematica, gnuplot, AVS, etc., which can be used to display functions of the form $z = f(x, y)$ and 2D scalar fields in various ways.

Except for some exploratory efforts (see, e.g.,[39, 63]), the analogs of elevation maps for 3D scalar fields have not been pursued to their logical completion. Our goal here is to exploit four-dimensional rendering and shading models to create a more complete family of volumetric analogs of the standard elevation map techniques used for surface representation.

5.1.2 Volume Visualization Techniques

Volume visualization techniques for displaying volumetric data or 3D scalar functions $w = f(x, y, z)$ can be classified into the following two categories according to their intermediate representations:

- **Surface-based techniques.** These techniques are 3D analogs of 2D contour plots and pseudocolor maps. Various methods for extraction of isosurfaces are presented in [77, 4, 32]. Other good examples of this class are techniques for viewing color-coded density data[84, 45, 44], making easily visible pseudocolor maps[85], and exploiting transport theory approaches[70].

  The advantages of the surface-based techniques include smaller computer memory usage, reduced computational requirements, and ability to utilize fast hardware geometric primitives. As a result, it is possible to interactively control the visualization process. However, only partial information is presented in such images due to the extraction of isosurfaces. Moreover, successful automatic fitting of geometric primitives to volume data is sometimes impossible to achieve because of the existence of severe ambiguities, coarse data, or poorly defined features within the volumetric data[76, 68].

- **Direct volume rendering techniques.** These techniques deal directly with volume primitives without any intermediate conversion of the volume data to surface representations. Each voxel of the image volume can be assigned a color value and a partial opacity; a final 2D semi-transparent image can be formed by blending together the voxels. For example, the technique presented by Drebin, Carpenter, and Hanrahan in [36] is useful for rendering images of volumes containing mixtures of materials. Upson and Keeler[99] introduce a ray casting method and a cell-by-cell processing technique to display volumetric data. Other good examples of this class of techniques are presented in [74, 75, 92, 104, 82, 93, 105, 106].

  One major advantage of direct volume rendering is that the entire data set, including those interior parts, can be displayed in a single image because every cell of the data set is contributing to the semi-transparent rendering process. Since these techniques can handle continuous variations easily, small or poorly defined features of the data may be displayed more clearly. However, direct volume rendering is computational expensive due to the amount of information that needs to be processed. Rendering time usually grows linearly with the size of the data set; therefore it is difficult to have real-time interactive control over volume visualization on current hardware[76].

None of the existing volume visualization techniques has explored the concept of treating 3D scalar fields as four-dimensional objects, illuminating them with four-dimensional lighting, and viewing them from a four-dimensional viewpoint. By extending the four-dimensional rendering and shading methods presented in Chapter 3, we have been able to introduce a new class of techniques for visualizing general volumetric data.

5.2 Extending 3D Elevation Map Methods to 4D

In this section, we show how to extend the standard 3D elevation map techniques for handling 2D scalar fields to 4D elevation map techniques for handling 3D scalar fields. The 2D scalar field that we used as an example to illustrate the 3D elevation map techniques are computed from the equation

$$z = \sqrt{(\max(0, 1 - x^2 - y^2))} \tag{40}$$

for a hemisphere plotted in the range $-1 < x < 1$ and $-1 < y < 1$. We can easily define a 4D analog of the above 3D hemisphere by using the equation

$$w = \sqrt{\max(0,\, 1 - x^2 - y^2 - z^2)}\ . \tag{41}$$

for a 4D hemisphere plotted in the range $-1 < x < 1$, $-1 < y < 1$, and $-1 < z < 1$.

The extensions of the 3D elevation map techniques to 4D are described in the following subsections.

5.2.1 Contour Curves versus Contour Surfaces

In Figure 42(a), we show the "straight-down" view of the 3D hemisphere contour curves. The data plots were generated by looking "straight down" at the data and drawing contour lines at the locations of selected level sets. The only cues we have for the depth are the labels or styles of the contour curves. The analog of contour curves for 3D scalar fields is basically a set of nested shells produced by an isosurface construction algorithm[77, 4]. Transparency is needed in order to show the inner contour surfaces. Figure 43 presents contour surfaces of the 4D hemisphere rendered using the "marching cube" techniques described in [77].

5.2.2 Grid Lines versus Grid Planes

Grid lines in 3D plots correspond to the lines (x = constant, y = constant) that divide the surface $z = f(x,y)$ into square patches. Grid lines provides 3D depth cues in standard 3D grid plots when viewed from oblique angles. In Figure 42(b), we show an oblique view of a surface grid for the 3D hemisphere with hidden surfaces removed. This method provides additional clarity to the viewer because only the nearest patches of the tilted checkerboard pattern are visible; lines lying on the grid behind a foreground patch are hidden from view. The distorted projections of individual outlines that are perfect squares in the straight-down view (see Figure 45(c)) also give additional depth information.

The analog of grid lines for 3D scalar fields are grid planes. Grid planes in 4D plots correspond to the planes, (x = constant, y = constant, z = constant) that divide the volume, or 3-manifold, into cubic patches. Similarly, we can observe geometric cues from the distortions of the grid planes in oblique views, that is, through 4D rotations.

Figure 44 show four grid planes plots of the 4D hemisphere, which is being rotated in 4D. The first cubic lattice-work in Figure 44 is an undistorted "3D checkerboard," just as the straight-down view of the square checkerboard grid is undistorted in Figure 45(c). The other three images in Figure 44 show distortion of the cubic lattice that persists regardless of how we change the 3D viewpoint. This is the 4D rotation effect that we emphasize.

The grid plane method is the simplest 4D elevation map technique. It is accomplished by drawing the faces of the alternating cubic cells of 3D scalar fields. The color of each face is white and the color of each edge is black. All the faces and edges are rendered using a hidden-surface algorithm. Hidden volume removal is not considered for this method to permit real-time interaction.

In our implementation, there are six slider objects associate with the six basic 4D rotations. Users can perform 4D rotations on the grid planes interactively by

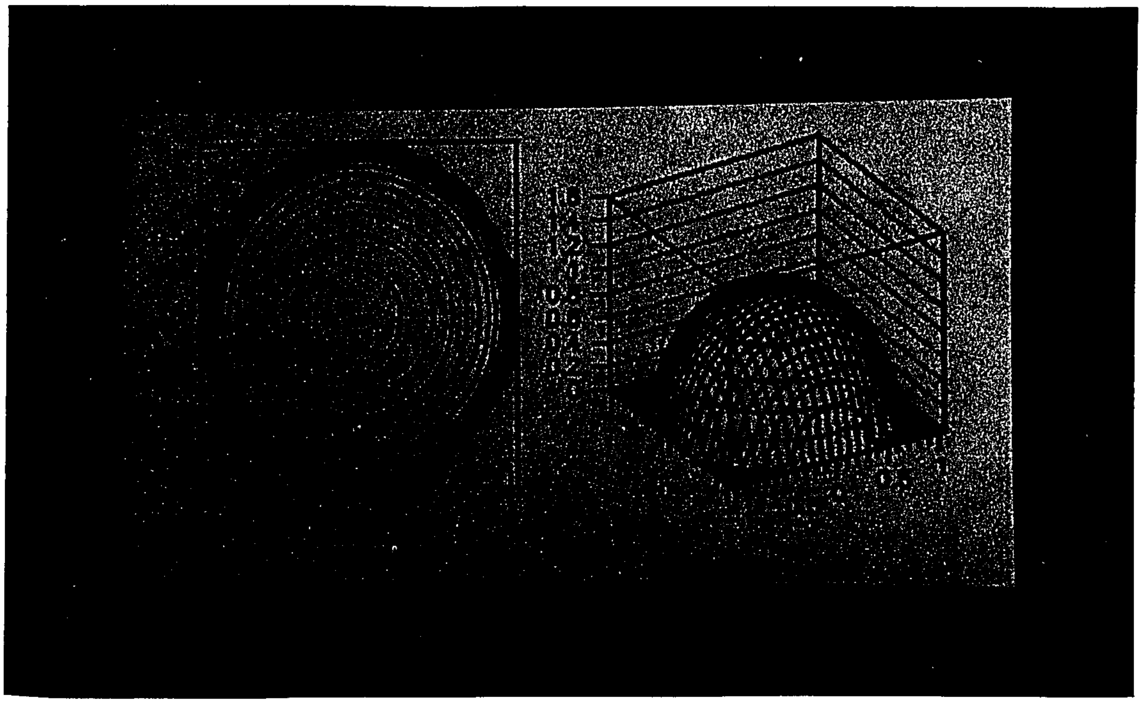


Figure 42: (a) A straight-down contour map of the 3D hemisphere. (b) An oblique grid line elevation map of the 3D hemisphere.

manipulating the slider objects. The amount of open space between the alternating cells can also be adjusted interactively using another slider object.

5.2.3 3D Pseudocolor Contours versus 4D Pseudocolor Contours

Representing the contours of the 3D hemisphere as color areas instead of contour curves, as in Figure 45(a), makes it easier to distinguish limited ranges of elevation if the color map is carefully chosen. No geometric cues are present in such an image.

Similarly, we can add pseudocolor to indicate where the level sets are located within the grid planes. This is similar to the "tiny cubes" algorithm of Nielson et al. [85].

The implementation of the 4D pseudocolor contours is described as follows:

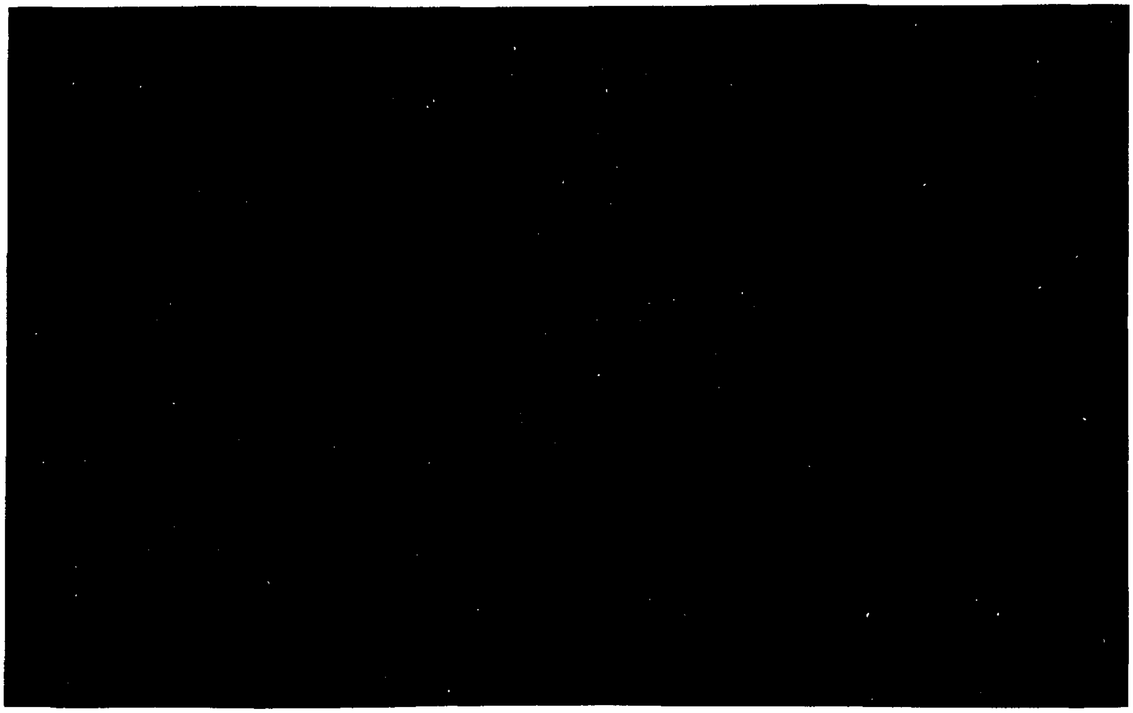


Figure 43: An image of a 4D hemisphere represented as a set of contour surfaces.

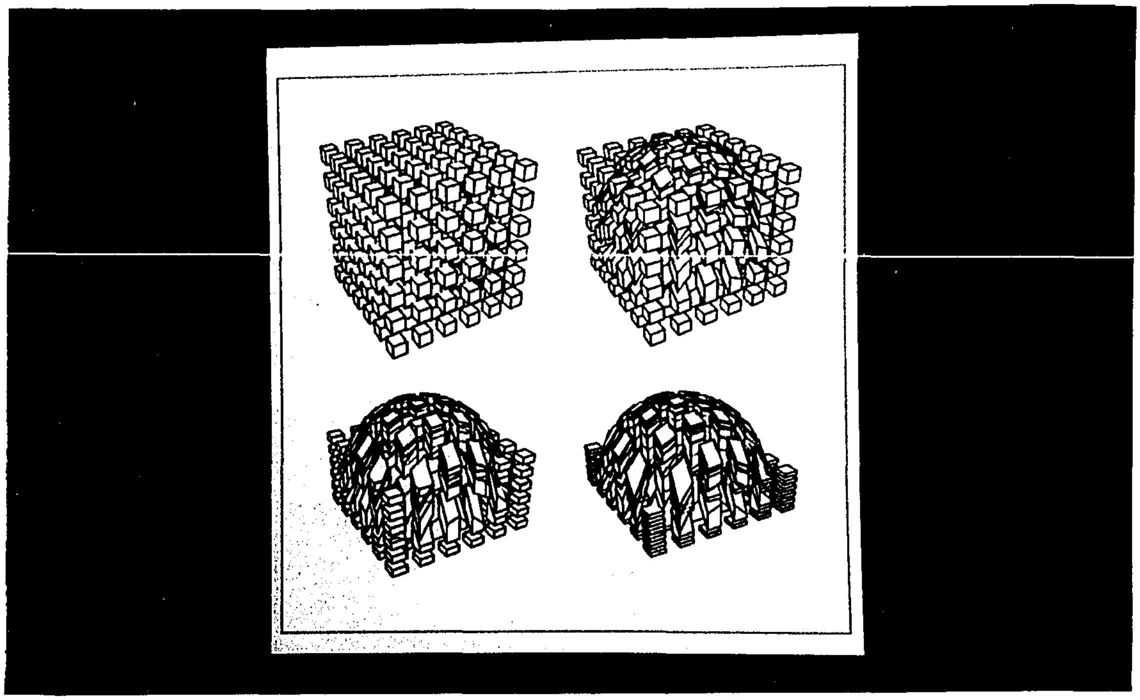


Figure 44: An image of a 4D hemisphere represented as a bare grid without removing hidden volumes. We show four stages in the 4D rotation.

1. Based on the density value of the 3D scalar field and a color map, we assign a color to every grid point in the 3D scalar field.
2. In order to make the interior voxels of the 3D scalar fields visible, we display only the faces of alternate cubic cells. This is similar to one of the cutaway techniques that we describe in Chapter 2. That is, we divide the surface into ribbons and cut out alternate ribbons, so that one can see the interior through the missing strips.
3. Generate Gouraud-shaded polygons with the assigned vertex colors.

The assignment of different colors to each grid point of the cubic mesh is the only difference between the implementation of the 4D pseudocolor gird and the 4D bare grid planes. The pseudocolor provides more information about the 3D scalar field than the bare grid planes do; the extra rendering time required is negligible.

By using this method, we can generate Figure 46 for the 4D hemisphere, which is the analog of Figure 45(a).

5.2.4 3D Oblique Pseudocolor Contour versus 4D Oblique Pseudocolor Contour

In Figure 45(b), we combine the information in the pseudocolor contour plot with the geometric cues of an oblique view of the colored surface without a grid. This provides very useful *redundant* cues for 3D hemisphere, since both the color and the 3D aspect of their image contain similar information about z.

Rotating the pseudocolor contour grid of the 4D hemisphere in 4D causes the colored regions to shift in such a way that a redundant value cue is produced. The image in Figure 47 shows the redundant cues of oblique distortion of the small volume elements and their positions as the rotation passes through four successive angles; the

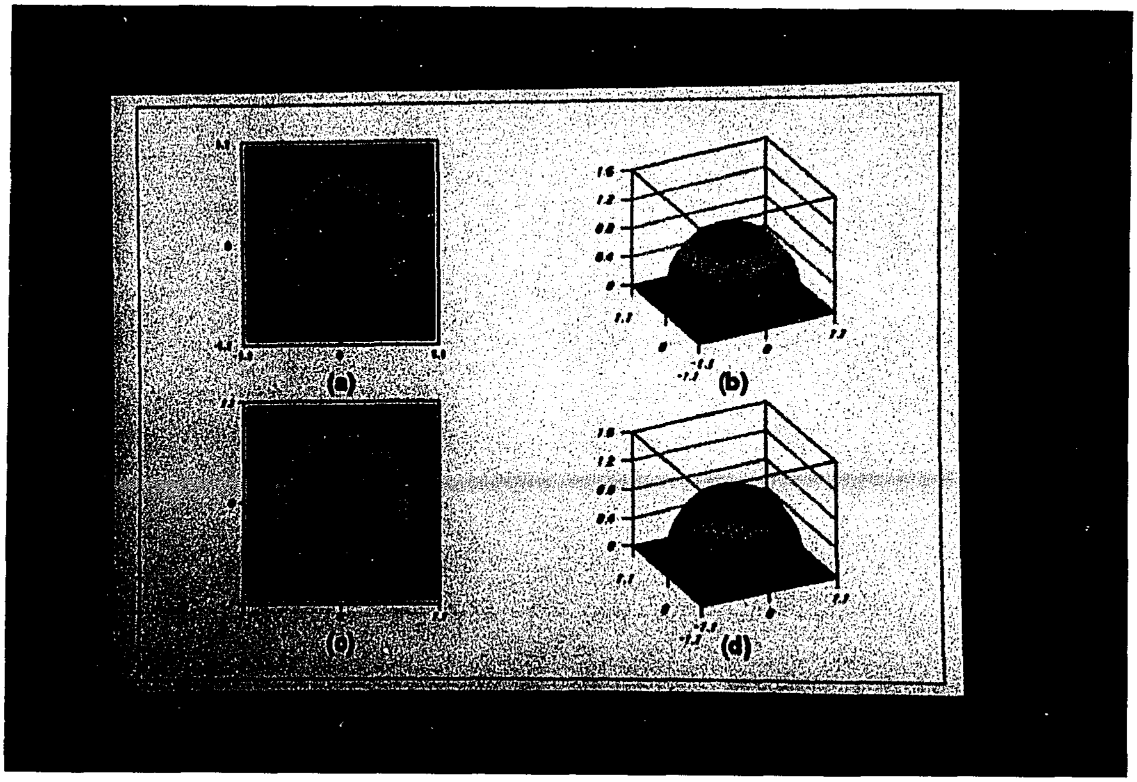


Figure 45: A collection of ways of displaying the 3D structure of a hemisphere, a simple example of a 2D scalar field given by Eq. (40): (a) A straight-down pseudocolor map. (b) An oblique elevation image with pseudocolor only. (c) A straight-down pseudocolor map with the grid lines as seen from this viewpoint. (d) A pseudocolor map of the hemisphere, including the grid lines, as seen from an oblique viewpoint. (Images generated by PAW [25].)

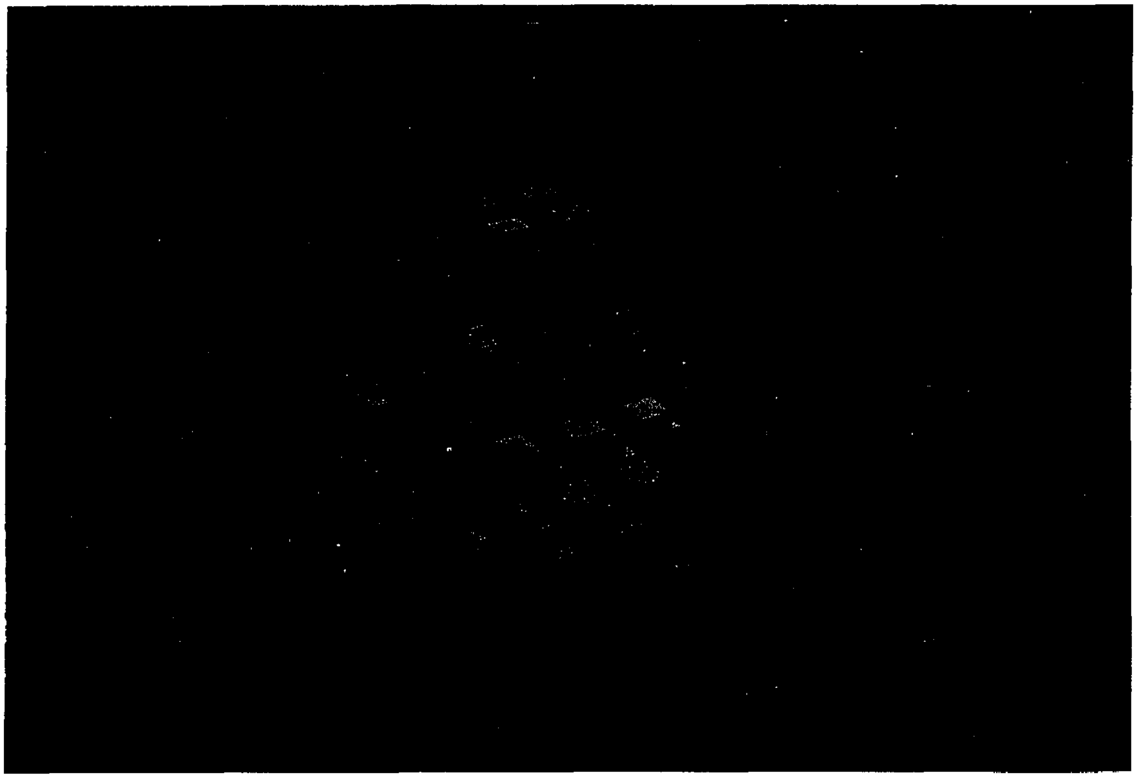


Figure 46: 4D pseudocolor representation of the hemispherical 3D scalar field of Eq. (41). The "3D checkerboard" is not rotated in 4D, and thus is undistorted. The interior colors are made more visible by opening up the gaps between elements using the "tiny cubes" approach.

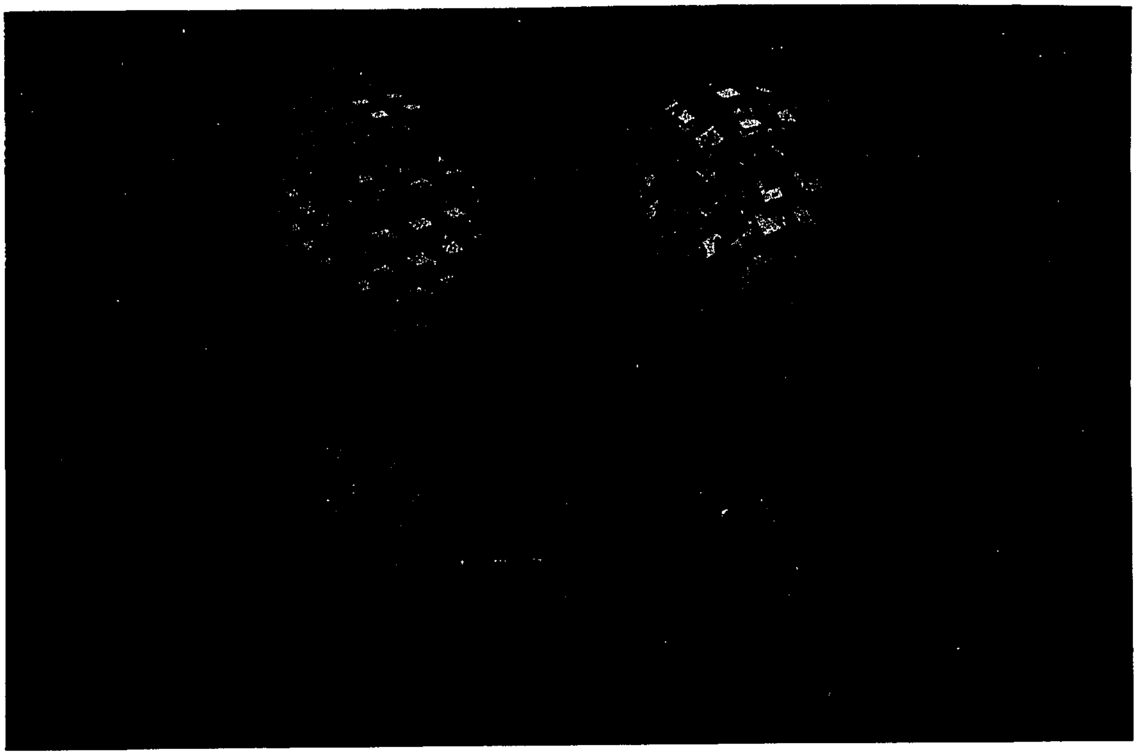


Figure 47: The 4D hemisphere with pseudocolor added to each cube of the volume grid. As we rotate to four different 4D orientations, the geometry changes while the color attached to any particular small cube remains fixed.

pseudocolor attached to any particular volume element remains fixed to a particular element, just as it did when we rotated in 3D to make the transition from Figure 45(a) to Figure 45(b).

Similar to the implementation of the grid plane method, these 4D rotations can be performed interactively through slider objects, and the amount of open space between the grid planes can be adjusted interactively. If the number of cubes to be rendered is not too large, we can have almost real-time interaction while applying the 4D rotation. Compared to the contour surface techniques, this method requires less time to render and also provides more continuous information within each cubic cell because it does not use any surface extraction algorithm.

5.2.5 3D Grid with Hidden Surface Removed versus 4D Mesh with Hidden Volume Removed

If we make an oblique view of the grid line representation of the 3D hemisphere with hidden surfaces removed, as shown in Figure 45(d), we get additional information, now *triply* redundant: the color, the hidden surface effect, and the distortion of the grid lines. This image is the richest we can construct when compared to the other 3D plots because it exhibits the shape information to the viewer in multiple modes with little ambiguity.

For the 4D hemisphere, the grid can now be rendered using 4D depth buffering to eliminate confusion between overlapping volumes. A typical result from a 4D viewpoint is shown in Figure 48. Now we see a very strange phenomenon from the point of view of an observer accustomed to 3D mesh plots: instead of having tilted rectangles hidden by other tilted rectangles in the foreground, we have distorted cubes that are truncated by other opaque distorted cubes that sit in front of them in 4D.

5.3 Using 4D Illumination on a 3D Scalar Field

The most effective way to visualize a 2D scalar field is to consider the surface to be a reflective material, place a 3D light in the scene, and use standard shading techniques[41] to create a shaded scene rendering. The shaded view has a very natural aspect, and contains shape cues about the orientation and shape of the surface that are strong for most viewers. In Figure 49, we show how the shaded 3-D hemisphere would appear from an assortment of viewpoints. As we move from the straight-down viewpoint Figure 49(a), which has no occlusion, to a very oblique viewpoint Figure 49(b), we see the hemisphere "rising out of the background." The most oblique view has the most occlusion, and also shows the profile of the highest elevation points

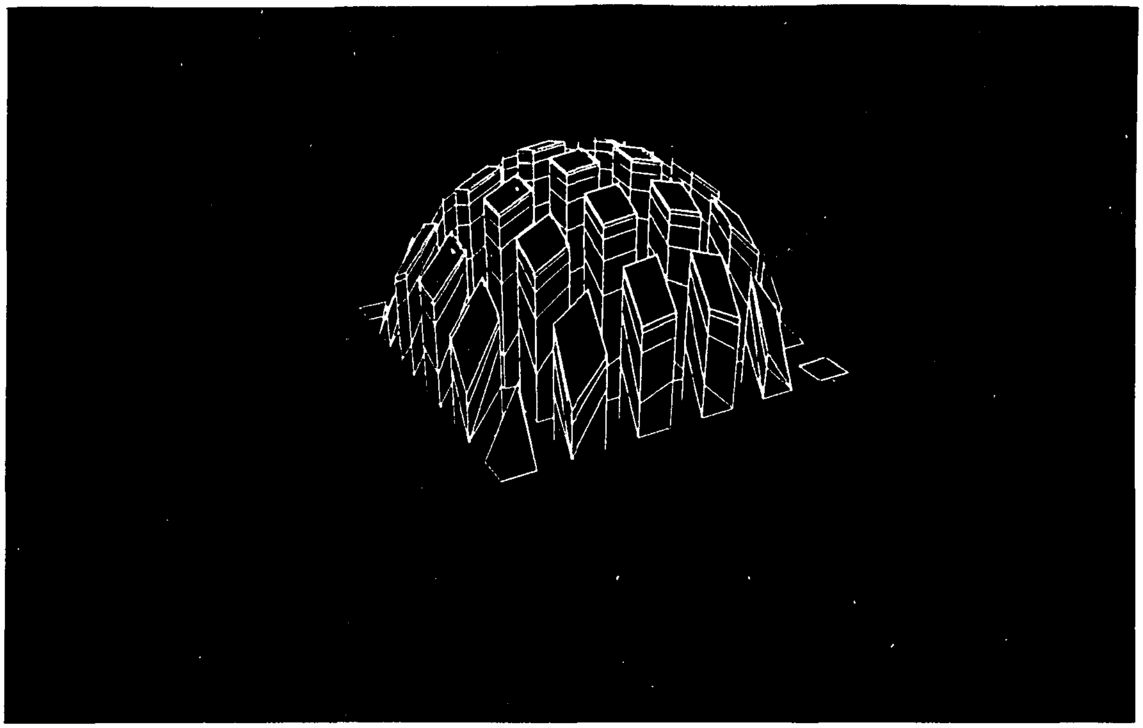


Figure 48: Rendering the 4D hemisphere with hidden volume elimination.

dramatically. The shaded images contains *orientation information* that is more explicit than in the other methods.

The 4D analog of the 3D shading method used in Figure 49 uses lighting and shading in four dimensions to reveal geometric characteristics of the 3D scalar field.

The same rendering methods and shading equations described in Chapter 3 can be used to render the 3D scalar fields. The only difference between the rendering methods for parametric volumes, implicit volumes, and general 3D scalar fields is the computation of normal vectors at the grid points.

Normal computation for parametric volumes is described in Chapter 3 and examples are given in Chapter 4. The normal is found by taking the 4D cross product of the 3 tangent vectors at each grid point.

In Chapter 4, we also showed how to compute the normal vector for superquadrics based on the implicit equations. Computation of the normal is simple for implicit

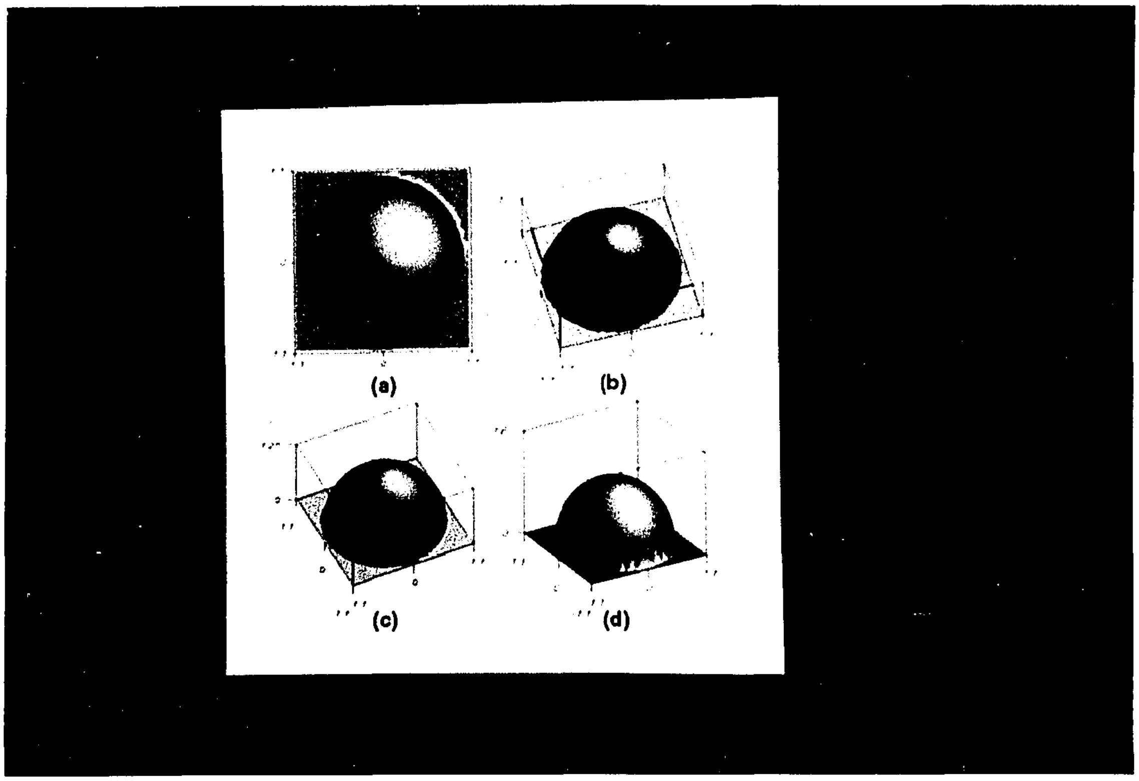


Figure 49: Four 3D viewpoints of the a hemispherical 2D scalar field given by Eq. (40). (a) is straight down, and has no occlusion. (b) and (c) show the hemisphere as it rises from the background as we rotate the 3D viewpoint; we begin to see some occlusion. Finally, in (d) we see a very oblique view with extensive occlusion and a profile explicitly showing the elevation of the hemisphere above the background. (Images generated by PAW [25].)

volumes since all that needs to be done is to take the gradient of the equation defining the 3-manifold.

It is more complicated to compute the normal vector at each grid point for general 3D scalar fields for use in smooth shading.

To compute the normal vector at a grid point of the 3D scalar field, we use an averaging approach that can be described as follows:

1. Each cubic cell of the volume data is decomposed into tetrahedra, as we do for parametric volumes.

2. The normal vector $\vec{n}$ of each tetrahedron is computed by taking the 4D cross product of the three independent vectors on the tetrahedron. If we let V_1, V_2, V_3, and V_4 be the tetrahedron vertices, then

$$\vec{n} = (V_2 - V_1) \times (V_3 - V_1) \times (V_4 - V_1) \,. \tag{42}$$

   The normal vector $\vec{n}$ is then normalized to $\hat{n}$ using the following equation:

$$\hat{n} = \frac{\vec{n}}{\|\vec{n}\|} \,. \tag{43}$$

3. If flat shading is sufficient, then we simply use $\hat{n}$ as the normal vector for the tetrahedron and scan convert the tetrahedron into the view volume by using the w-buffer algorithm and the scan-plane algorithm.

4. If smooth shading is required, then we need to compute the normal vector for each vertex of the tetrahedron, not just the normal of the tetrahedron. In order to compute the normal vector for each vertex, we first need to compute the unit normal vectors of all the tetrahedra surrounding that vertex. Then we sum all the surrounding unit normal vectors and average them. The average normal

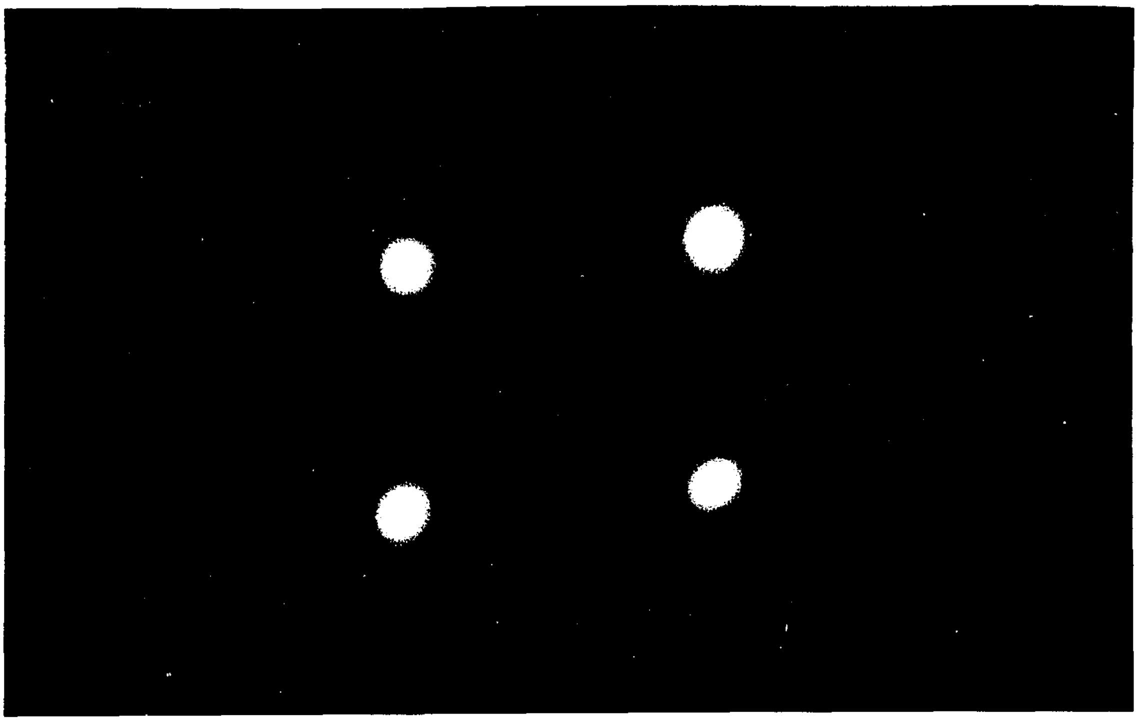


Figure 50: 4D lighting and shading applied to the hemispherical 3D scalar field data of Eq. (41). As we rotate the viewpoint to four different 4D viewing angles, the geometric cues change dramatically, showing the 4D analog of the view changes in the 2D scalar field hemisphere in Figure 49.

vector is normalized, and it becomes the unit normal vector of that particular vertex. Depending on its location within the 3D scalar field, a vertex can be shared by up to 32 tetrahedra if we use the two cubic cell decompositions given in Section 3.7.2 of Chapter 3.

Once the normal vectors of all grid points are computed, the cubic mesh can be scan converted into the view volume using the 4D rendering methods described in Section 3.7 of Chapter 3.

An example of 4D shading applied to the 4D hemisphere is given in Figure 50. The four volume images in Figure 50 are the 4D analogs of Figure 49.

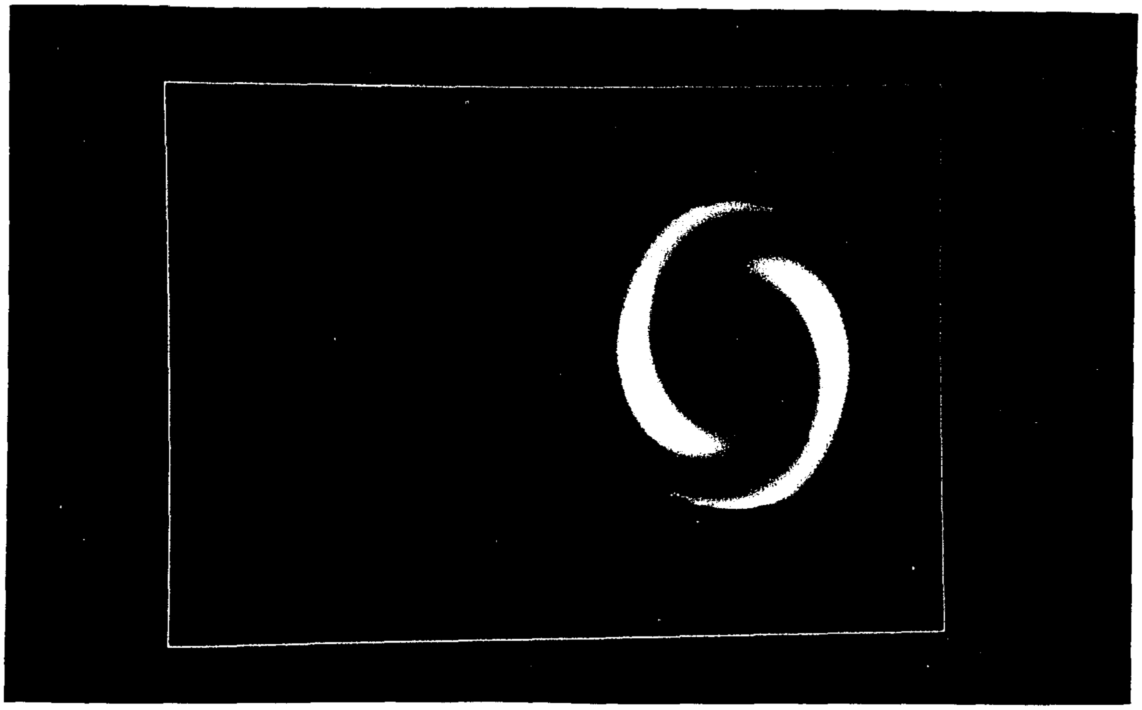


Figure 51: (Left) Volume rendering of binary star density data using the AVS "tracer" module. (Right) The 4D shaded rendering of the data.

5.4 Examples of Applications

To illustrate the application of our method to real data, we first show in Figure 51 a comparison between two ways of rendering a set of astronomical binary star density data. On the left is a standard volume rendering produced by the AVS "tracer" module[98, 1]. On the right, we show a shaded rendering of the same data using our 4D lighting technique. Previously invisible features, basically corresponding to regions with similar 4D gradients, are revealed by our approach. In Figure 52, we show the 4D pseudocolor representation of the same data.

Next we repeat the same comparison for a data set describing the electron density of the hydrogen molecule. The left image in Figure 53 shows a standard AVS volume rendering. On the right, we show for comparison a 4D shaded rendering of the density elevation plot. Again, we see suggestive additional structure, corresponding

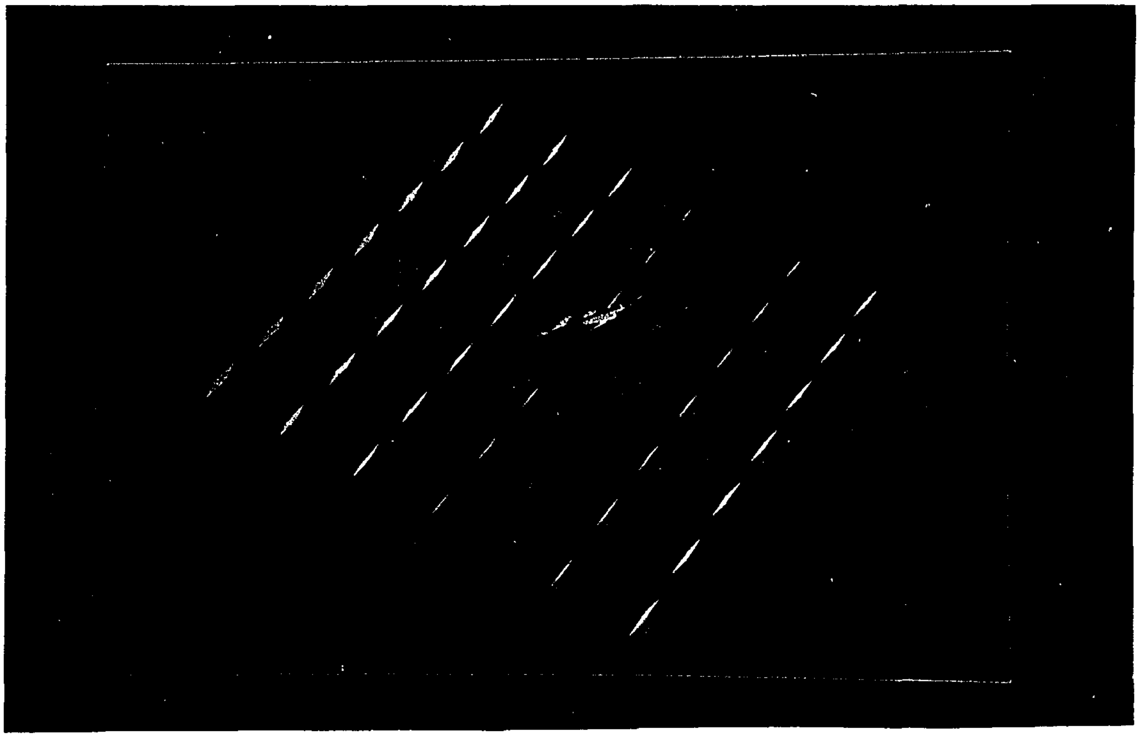


Figure 52: 4D pseudocolor representation of the binary star density data.

to aligned normals, in the 4D rendering. In Figure 54, we show the 4D pseudocolor representation of the same data.

5.5 Summary

Volume visualization techniques are important because volumetric data are common in many scientific applications, and it is easier for human viewers to understand the data when they are represented as interpretable volume images instead of a massive collection of numbers. By treating 3D scalar fields as four-dimensional objects, we were able to extend the four-dimensional rendering and shading methods originally developed for mathematical visualization to create images of volumetric data generated by other scientific applications. The 4D contour plots such as the grid plane method and the pseudocolor volume grid method provide viewers interactive

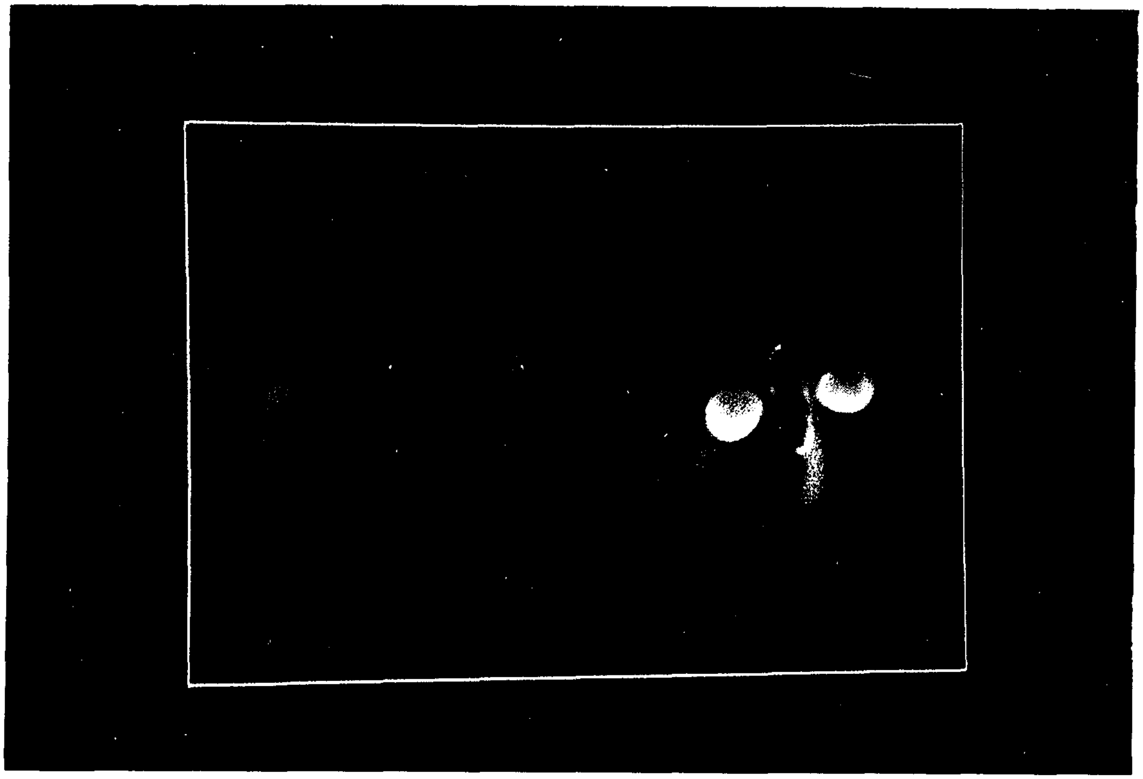


Figure 53: (Left) AVS volume rendering of the electron density field of a hydrogen molecule. (Right) The 4D shaded rendering of the same data.

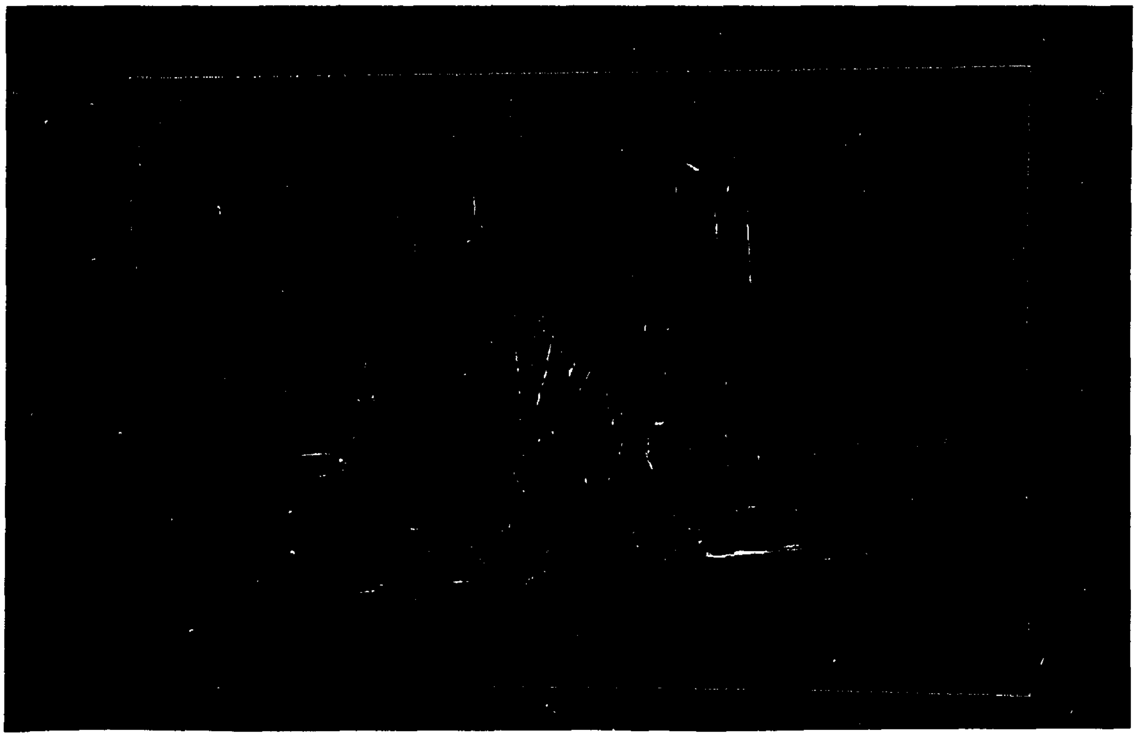


Figure 54: 4D pseudocolor representation of the electron density field of a hydrogen molecule.

control over the visualization of volumetric data. While using the 4D shading and the w-buffer algorithms slows down the visualization process, these enhancements do provide additional details of the geometric structure of the data. Compared to images generated by other standard volume visualization techniques, these new techniques reveal some extra structure that may be useful for scientists to gain new insights into their volumetric data.

Chapter 6

Thesis Summary

Although topology has been one of the main subjects of this work, we did not attempt to do research in topology. We put our focus on developing interactive visualization techniques and innovative computer graphics methods to better solve the problem of visualizing abstract mathematical objects. We also emphasized building visualization tools that are generalizable, so that they can be used in other scientific or industrial applications. We did not want to be restricted by the limitations of the current hardware and had no intent to make very fast, hardware-specific implementations; with the rapid evolution of computer hardware, performance will eventually take care of itself. It is more important to design prototypical interaction and representation strategies that will still be useful when the present generation of graphics hardware is obsolete.

In this chapter, we summarize the major results and propose some future research directions.

6.1 Summary of Results

The following results have been achieved in this work:

- **Exploration of current computer graphics software and techniques.** As presented in Appendix A, we have explored current computer graphics software such as *Doré*, *Mathematica*, *Wavefront* and *AVS* to a significant extent, and have utilized them heavily throughout this research. *Wavefront* and *Doré* have been used to produce visually exciting computer animations that are related to mathematical visualization[56, 60, 57]. Images of some mathematical objects generated by *Wavefront* have been used to show the differences between the standard 3D rendering methods and our 4D rendering methods in Chapter 4. In Chapter 5, the standard volume visualization techniques available in *AVS* are used to generate volume images for comparison with the volume images generated by our adapted four-dimensional rendering methods. We have also used a *AVS* video recording module written by Brian Kaplan and Eric Ost at CICA to control the video recording process of computer animation. *Mathematica* provides excellent help in carrying out certain mathematical computations, particularly in the complex domain; and it is also very useful for simple graphical experimentation.

- **Development of an object-oriented graphics class library.** In this work, an object-oriented graphics class library was constructed on top of *Doré, C++,* and the *X-Window System.* It consisted of approximately 40000 lines of code. The graphics class library has been used to develop interactive mathematical visualization techniques and user interface techniques described in Chapter 2. We have used the library to implement an interactive visualization system for exploring topological objects such as the Fermat surfaces and to implement the four-dimensional contour plot techniques presented in Chapter 5.

- **Development of interactive mathematical visualization techniques.** A long list of interactive mathematical visualization techniques has been implemented in our visualization system; they are presented in Chapter 2.

- **Development of user-interface design techniques.** Issues related to user interface design have been discussed in Chapter 2, and various user-interface design techniques that provide direct context-free manipulation and constraint-based interaction were also implemented into the visualization system.

- **Topological exploration of mathematical surfaces.** By using the interactive visualization techniques together with the user-interface, we were able to carry out interesting topological explorations on some mathematical surfaces, for example Fermat surfaces. More sophisticated topological explorations of these complex surfaces such as the Riemann surface deformation were performed.

- **Development of four-dimensional rendering techniques.** We have extended three-dimensional rendering techniques to work in the four dimensions. Chapter 3 presents all the necessary techniques and methods for generating volume images of four-dimensional mathematical objects. Images of several interesting mathematical objects generated by the four-dimensional rendering methods were given in Chapter 4.

- **Proposition of a complete scientific visualization paradigm.** Our work has raised some open problems such as the interpretability of 4D images and the learnability of 4D perception. Although we did not attempt to solve these open problems, we suggested a *Visualization Principle* that should be used as a guideline for a successful visualization system. A complete scientific visualization paradigm based on this principle was proposed in Chapter 4.

- **Production of educational videos on mathematical visualization.** We produced several computer animated videos[56, 60, 57] that illustrate the important concepts of our interactive techniques and the four-dimensional rendering methods. The applications of these techniques and methods are demonstrated through animation of mathematical objects such as Fermat surfaces, Steiner surfaces, the four torus, a knotted sphere, the hypercube and the hypersphere.

- **Laying theoretical groundwork for interactive methods usable on future machines.** The four-dimensional rendering methods are too computational intensive for real-time interaction on current computer hardware. However, when the hardware becomes powerful enough to run these four-dimensional rendering methods in real-time, the interactive visualization techniques and user interface design methods that we developed will still be applicable. Some of these interactive techniques have been recorded in the form of video animations to simulate interaction.

- **Generalization of four-dimensional rendering techniques.** More importantly, we have generalized the four-dimensional rendering techniques so that they can be used not only in mathematical visualization, but also for visualizing volumetric data generated in other scientific applications. In Chapter 5, we presented the four-dimensional contour plot techniques and the 4D shading methods for viewing general volumetric data. Examples of applications to real data were given.

6.2 Future Directions

Interactive mathematical visualization is still at its early stage, and the potential development for this field is almost unlimited. Although we have achieved some important results in this work, there is still a great deal of work that can be done. Some of the most interesting remaining problems are listed below:

- **Parallelization of four-dimensional rendering algorithms.** The w-buffer algorithm and the scan-plane algorithm developed in this work can be parallelized. Performance of these methods can be improved by a large factor when this is done, moving us closer to real-time interactive control over four-dimensional rendering.

- **More mathematical visualization techniques.** There are many more mathematical transformations that can be applied to high-dimensional mathematical objects: for example, we can apply PGL(3,C) transformations[52], Veronese transformations[42], and other homotopies to the Fermat surfaces. It would be very helpful for topological exploration if these transformations were implemented as interactive visualization techniques in the system.

- **Interpretability of 4D images.** In our work, we have not proven that the 4D images we produced are *interpretable*. A possible solution to this problem is to extend three-dimensional computer vision techniques to work in four dimensions. The 4D extension of the shape-from-shading methods such as that of Ikeuchi and Horn[66] may be useful for *reinterpreting* 3D volume renderings as images of four-dimensional objects.

- **Learnability of 4D perception.** If these 4D images are interpretable, what are the most effective techniques for teaching an understanding of such images? Can humans acquire facility with four dimensions with enough practice? Could a single shaded volume rendering in stereo be sufficient for 4D understanding, just as single photographs of a 3D scene are? There are many unanswered questions in this area.

- **Verification of the Visualization Principle.** In the work of 4D visualization, unless we solve the other two open problems, that is the interpretability of 4D images and the learnability of 4D perception, we are not ready to verify that the *Visualization Principle* is satisfied.

- **Constraint-based interaction.** With only simple 2D input devices such as mouse, it is difficult to carry out straightforwardly many complicated "moves" [42] that are important in topological visualization. We need to add *intelligent constraints* to the user interface so that simple input-device motions can be used to indicate complex topological transformations. Some rule-based systems for satisfying various constraints might be useful in such an interface.

- **Direct context-free manipulation.** If possible, more sophisticated devices such as the data glove and head-mounted displays should be used to perform advanced direct manipulation in topological visualization. For example, one might "poke," "pinch," and "pull" on pieces of a topological object.

- **Optimal Representations.** Mathematical objects can be deformed to many different explicit shapes without disrupting the fundamental topological invariants. What are the criteria that define one form of a given shape as being "better" than others? Are there numerical optimization mechanisms for discovering these optimal representations?

- **Four-Dimensional Virtual World.** We have already utilized four-dimensional virtual cameras and lighting to generate volume images of four-dimensional mathematical objects. However, there is a great deal more work to be done in the extension of 3D interactive techniques to higher dimensions in order to construct a four-dimensional virtual world where users can visualize, interact, and manipulate abstract four-dimensional objects in a more intuitive manner. Special equipment such as that used in virtual reality research is needed to achieve this goal.

Appendix A

Computer Graphics Tools

Developing a visualization system from scratch is extremely time consuming, and it is also not the main purpose of this research. We have taken advantage of available software tools whenever possible to allow more time to be spent on higher-level problems. Unfortunately, many requirements of higher-dimensional mathematical representations are so specialized that the development of a substantial amount of new software is unavoidable.

In this appendix, we describe the computer graphics tools that we have used and developed. In Section A.1, we discuss the available standard computer graphics tools, such as *Dorè*, *Mathematica*, *Wavefront*, and *AVS*. We present an object-oriented graphics class library that has been built on top of *Doré*, *C++*, and the *X-Window System* in Section A.2.

A.1 Available Standard Tools

Listed below are some popular graphics software tools that we used during our research:

A.1.1 Doré - Dynamic Object Rendering Environment

Doré is a product of the Kubota Pacific Computer Inc. It is an advanced three-dimensional graphics library, which provides hundreds of computer graphics functions and procedures that can be called from either C or Fortran programs. Users can create graphics for three-dimensional objects quite easily with the provided primitives such as lines, polygons, polygon meshes, surface patches, etc. There are two built-in renderers in *Doré*, the Dynamic Renderer for interactive rendering and the Production Renderer for producing photo-realistic images through ray-tracing.

Basic steps

The basic steps in programming with the *Doré* library including the following:

- Create geometric models for all objects in a scene using primitives such as points, lines, polygons, and surfaces.
- Assign features such as color, shininess, texture, transparency that determine the appearance of the objects in the scene.
- Scale and transform each object to be at the right place with the right size.
- Choose the light sources, their locations, their colors and intensities.
- Select a camera lens, specified using a parallel projection, a perspective projection, an orthographic projection, or an arbitrary four-by-four matrix. Other details such as the position of the camera, the zooming factor, the front clipping plane, and the back clipping plane can also be specified.
- Choose a renderer, which can either be the fast Dynamic Renderer, the slow but high quality Production Renderer, or even a third party developed renderer. The rendering process may now be carried out[71, 72].

Advantages

Doré was the most powerful graphics tool available to us when this research started; we used it extensively for investigating interactive visualization techniques and for creating interesting video animations[60]. *Doré* has the following advantages:

- **Doré is fast.** For a simple scene, even including features such as Gouraud shading, transparency, and specular highlighting, the rendering process can be done almost instantaneously with the Dynamic Renderer on a Kubota Pacific Titan computer.
- **Doré library is object-based.** Almost every element of the library can be treated as an object. There are methods associated with each type of object that are invoked during the rendering process to create the desired appearance and effect[71]. We have built an object-oriented graphics class library based on *Doré*, *C++*, and the *X-Window system.*
- **Doré is extensible.** The user can use C or Fortran programs to define new primitives to supplement the standard primitives if necessary.
- **Doré supports remote execution.** The Doré generic X-window device lets users display Doré images over a network on any system that supports the X Window System.
- **Doré is portable.** Applications based on Doré can be moved to other graphics platforms such as IBM and SUN workstations quite easily.

Constraints

We encountered a number of difficulties peculiar to the *Doré* graphics library and among these are:

- **Dissolves.** During the process of creating complicated animation sequences, we have no natural utility function available in *Doré* to do dissolves between two different scenes.

- **Transparency.** Transparency works quite well with the dynamic renderer in *Doré*. However, there is no way to change smoothly between an opaque surface and a transparent surface. In *Doré*, a surface assigned with the highest transparent intensity is totally transparent, but a surface assigned with the lowest transparent intensity is not totally opaque as we expected. It already looks rather transparent.

- **Animation.** The default user-interface controls for orientation choices are awkward to use in an exploratory manner, and there is no support for interpolation or splining among orientations or viewpoints.

Figure 1 in Chapter 1 is generated using *Doré*.

A.1.2 Mathematica

Mathematica is a general software system and a programming language developed by Wolfram Research, Inc. for mathematical computation and other applications[107, 80]. It can handle two-dimensional and three-dimensional graphics, numerical computations, and symbolic mathematics. There are more than 800 built-in functions that can be used for problems such as matrix manipulation, solving algebraic solutions, polynomial factorization, symbolic integration and differentiation, power series expansion, etc. We are very impressed by *Mathematica*'s ability to handle prototypes of almost anything we needed.

Advantages

Among the particular strengths of *Mathematica* are:

- **Mathematica notebook.** *Mathematica* provides an innovative front end called the *Mathematica Notebook*. Users are able to mix text, graphics, mathematical equations in a notebook. Together with the parametric plotting packages provided in *Mathematica*, we used *Mathematica notebooks* for testing and drawing mathematical equations, for planning and developing scripts and storyboards to be used in animation sequences, and for recording our research results.

- **Complex Arithmetic.** *Mathematica* handles complex arithmetic in a very natural manner. We needed to deal with mathematical equations in the complex domain quite often, for example in our treatment of Fermat's Last Theorem. It was very useful for us to check our equations in *Mathematica* and prevent us from wasting time programming in C with wrong equations.

- **Graphics capability.** *Mathematica* provides built-in packages for drawing two-dimensional and three-dimensional parametric equations. These packages are quite flexible and easy to use. We can use them to make simple experiments to visualize aspects of the mathematical equations.

- **Numerical computation.** *Mathematica* handles numerical computation extremely well. Thus some of the problems we found when computing surfaces using standard programming languages could be circumvented in *Mathematica*, at the price of reduced speed.

- **Symbolic computation.** We needed to perform many symbolic differentiations for complicated mathematical equations in order to determine their corresponding normal equations. *Mathematica* prevented us from making mistakes in this kind of computation.

Constraints

Among the drawbacks we experienced in using *Mathematica* are the following:

- **Graphics animation.** While graphics animation is supported in *Mathematica*, there is no way to specify a *sequence of times* for a set of graphics frames. This prevented us from being able to test the exact timing of rough draft animation storyboards specified by a set of representative still frames.

- **Memory requirement.** *Mathematica* can crash a system quite easily by using up all the swap space or going into an infinite loop. Memory limitations caused many problems and loss of time; virtual memory support and graceful recovery from memory overflows would have been invaluable. We were prevented from discovering certain basic properties of the equations we explored in *Mathematica* because of memory limitations; the discoveries were made instead after much programming effort in the *Doré* graphics package described above. We originally expected to be able to carry out all such investigations of mathematical properties directly within *Mathematica* .

- **Slow Graphics.** The speed of three-dimensional graphics is slow as well as being very memory intensive even for simple objects. It would have been very useful to have some sort of support for high-speed three-dimensional graphics once the polygon tables had been reduced to pure numbers; in principle, *Mathematica* should be able to know that a polygon table has been evaluated numerically, thus enabling high-speed three-dimensional graphics routines to be used on the numerical results.

- **User interface.** A better user interface for manipulating the graphics is desirable. For example, direct manipulation of graphical objects through mouse movements would be useful.

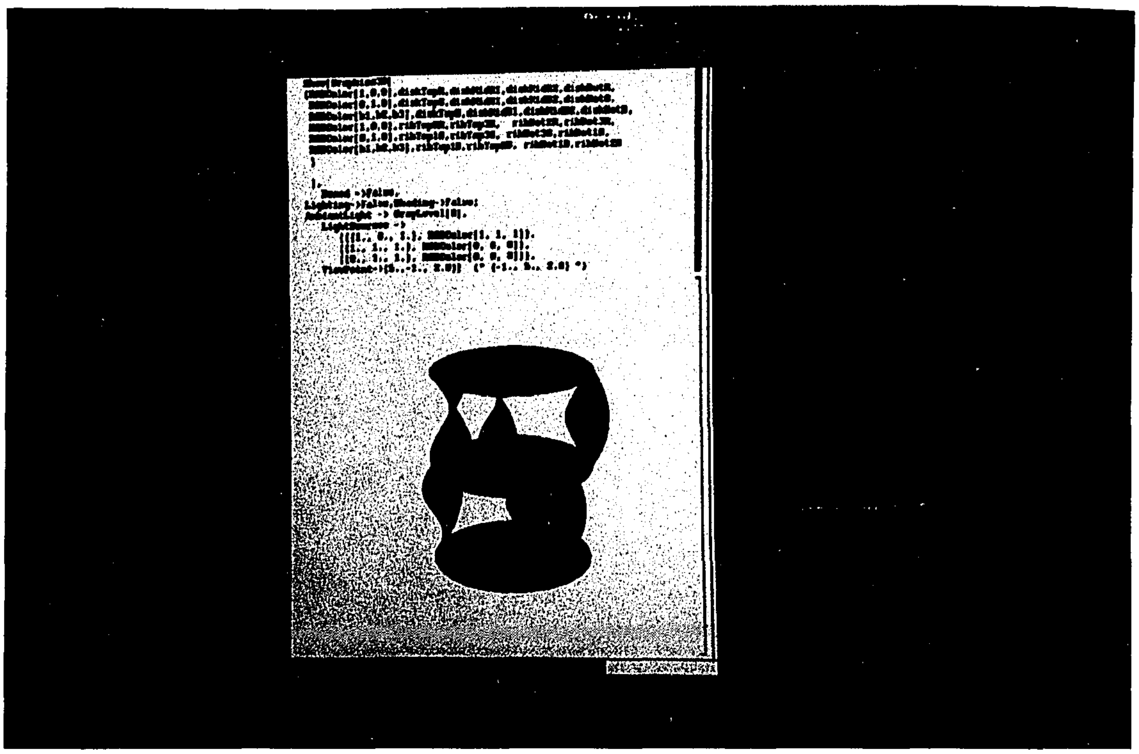


Figure 55: A portion of a *Mathematica* notebook showing a representation of the $n = 3$ Fermat surface.

In Figure 55, we show a *Mathematica notebook* written by Andrew Hanson as part of our research collaboration to explore the Riemann surface corresponding to the $n = 3$ Fermat surface. More details about such surfaces are given in Chapter 2.

A.1.3 Wavefront - Advanced Visualizer

The *Advanced Visualizer* developed by Wavefront Technologies, is a very high quality computer graphics package designed particularly for sophisticated commercial animation[102].

Components

The important components of this package are :

- **Model.** An interactive program for creating geometric objects.

- **Preview.** An interactive program for choreographing the motion of different geometric objects in a scene.
- **Medit.** An interactive program for creating and modifying materials and lights. Materials are assigned to objects in a scene so that they can have color and texture.
- **Image.** This generates images from the models created by Preview.

In this research, we used *Wavefront* to create exciting animation effects for video recording. We also used it to generate images for mathematical objects that we studied. Such images were used to show the differences in the lighting and shading effects between standard three-dimensional computer graphics techniques and the four-dimensional computer graphics techniques that we developed.

Advantages

The following are some advantages of *Wavefront*:

- **Animation effect.** *Wavefront* is designed for commercial animation, not for scientific visualization. *Wavefront* handles those aspects of animation that are used in commercial applications such as camera movements, linear and spline interpolation of motion, special effects, etc., very nicely. It has excellent tools for creating visual excitement in an animation; these tools are not available in either *Doré* or *Mathematica.*
- **Image compositing and dissolve.** There are programs in *Wavefront* that allow the user to dissolve from one image to another image, and composite one sequence of images with another sequence of images.
- **Good image quality.** *Wavefront* uses a combination of traditional scan-line techniques, A-buffer techniques, and reflection mapping to generate realistic

images of three-dimensional scenes. Advanced techniques such as ray tracing, reflection, refraction, transparency, shadow generation, and anti-aliasing are also supported.

- **User interface.** The user interface that *Wavefront* used is excellent. It supports direct object manipulation, picking, interactive manipulation of stereo objects such as lights and cameras, and keyboard input.
- **Batch mode.** The user can generate images using the "image" program in batch mode; thus without any user interaction, hundreds of images can be generated overnight.

Constraints

However, for the particular mathematical domain at hand, the methods available to us in Wavefront presented the following difficulties:

- **Huge object files.** Since *Wavefront* deals with "object files" that are polygon lists, simple mathematical objects, such as a 20 × 20 × 20 grid of spheres, had to be represented as astronomically large (approximately 8 MB) object files.
- **Local execution.** *Wavefront* does not support displaying images on remote machines. Wavefront's images can only be viewed when users are physically sitting in front of the machine that has the software. It would be ideal if the user could run Wavefront remotely over a network on any workstations that support the X Window System.
- **Expensive.** *Wavefront* is very expensive ($60,000) compare to other graphics packages (free or a few thousand dollars) that we have used. Together with the fact that it does not support remote execution, its availability is somewhat limited for users in academic institutions.

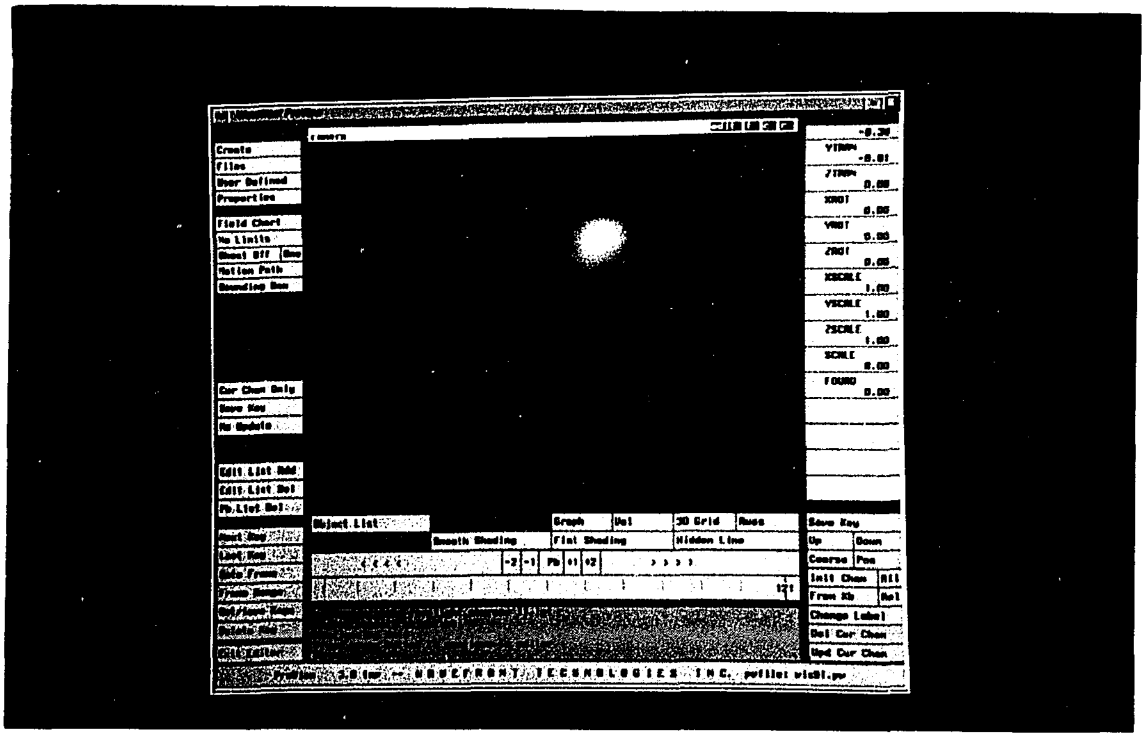


Figure 56: User interface of the Wavefront Preview program.

In Figure 56, we show the user interface of the *Wavefront* preview program. The surface displayed within the graphics window is the Steiner surface. More details about this surface are given in Chapter 4.

A.1.4 AVS - Application Visualization System

AVS is an industry standard environment for visualization applications from Advanced Visual Systems Inc. AVS has been designed in a highly modular manner. It comes with hundreds of useful modules. Each module is a "software building block" with well-defined interfaces, written either in FORTRAN or in C. The user can invoke these modules simply by selecting them from a menu and connecting them into a network through mouse-driven point-and-click operations[100, 98, 1].

Components

There are basically four types of modules:

- **Data input modules.** For reading numerical data in various formats.
- **Filter modules.** For transforming data into data, or geometry into geometry. Typical filters convert the output data of widely-used applications into displayable form.
- **Mapper modules.** For transforming data into geometry. These modules include advanced visualization techniques such as isosurfaces, two-dimensional slices of a three-dimensional data volume, etc.
- **Renderer modules.** For displaying geometry, images and volumes on the screen.

Subsystems

The current version of AVS comes with several useful subsystems such as :

- **Network Editor.** This provides a visual programming interface that allows the user to design a visualization application as a network of modules.
- **Geometry Viewer.** This system allows the user to manipulate graphical objects interactively. The user can have control over light sources and cameras, and can select surface properties such as specularity and transparency, color, and other graphics features like texture mapping and anti-aliasing.
- **Image Viewer.** This provides image-manipulation capabilities such as real-time pan and zoom, rotation and transformation, flip-book animation, etc.
- **Volume Viewer.** This is used for visualizing three-dimensional volume data. A slicer module extracts a two-dimensional slice out of a three-dimensional data

set, a tracer module implements an optimized ray tracing technique for three-dimensional volumes of cell, and a high-performance version of the Marching Cubes algorithm[77] generates isosurfaces.

- **Graph Viewer.** This allows the user to visualize and to output data in the form of two-dimensional graphs, line charts, bar charts, or contour plots.

In this research, we used the volume rendering modules in *AVS* to generate volume images and compare them with images generated by our new techniques. We also used it for controlling the video recording process based on a recording module written by Brian Kaplan and Eric Ost of the Center for Innovative Computer Applications at Indiana University. This module can handle image files of different formats and can record images to either laser disc or videotape.

Advantages

AVS is by far the best commercial visualization system available. The following are some of its advantages:

- **AVS is extensible.** Users can create additional modules either in C or Fortran for AVS; this enables them to solve more complex scientific and engineering visualization problems.

- **Many modules are available.** Besides those modules that come with AVS, there are more than a thousand public domain modules that users can obtain from the International AVS Center. It is not difficult for users to find modules that already meet their visualization needs. That also means that most of the current advanced visualization techniques are available to users with little or no programming effort.

- **Distributed visualization.** AVS allows modules to execute on computational servers in the network as well as to display images on remote X-terminals. This is important for users in a distributed multi-vendor environment.

Constraints

There are only a few constraints that we discovered.

- **AVS cannot be executed in batch mode.** We can generate many images overnight using the WaveFront "image" program in batch mode. We cannot do the same thing in AVS. It would be nice if AVS could accept commands from a script file, and keep on generating thousands of frames for animation without the user's interaction.
- **Programming bugs.** There were many bugs in early versions of AVS. This has been improved in later versions, but AVS releases need to be tested more thoroughly.
- **Debugging tools.** There are not enough debugging tools for writing AVS modules. Sometime it is difficult for programmers to figure out why the modules they write do not perform as expected. AVS should provide more helpful messages or useful tools to help programmers debug their code.
- **Slow labeling.** AVS is very slow when generating text labels within images.

Figure 57 shows the construction of the *tracer* module on the *AVS network editor.* The pseudocolor image in the graphics window is the result of applying the tracer module to ray-traced volumetric data of the hydrogen molecule electron density field.

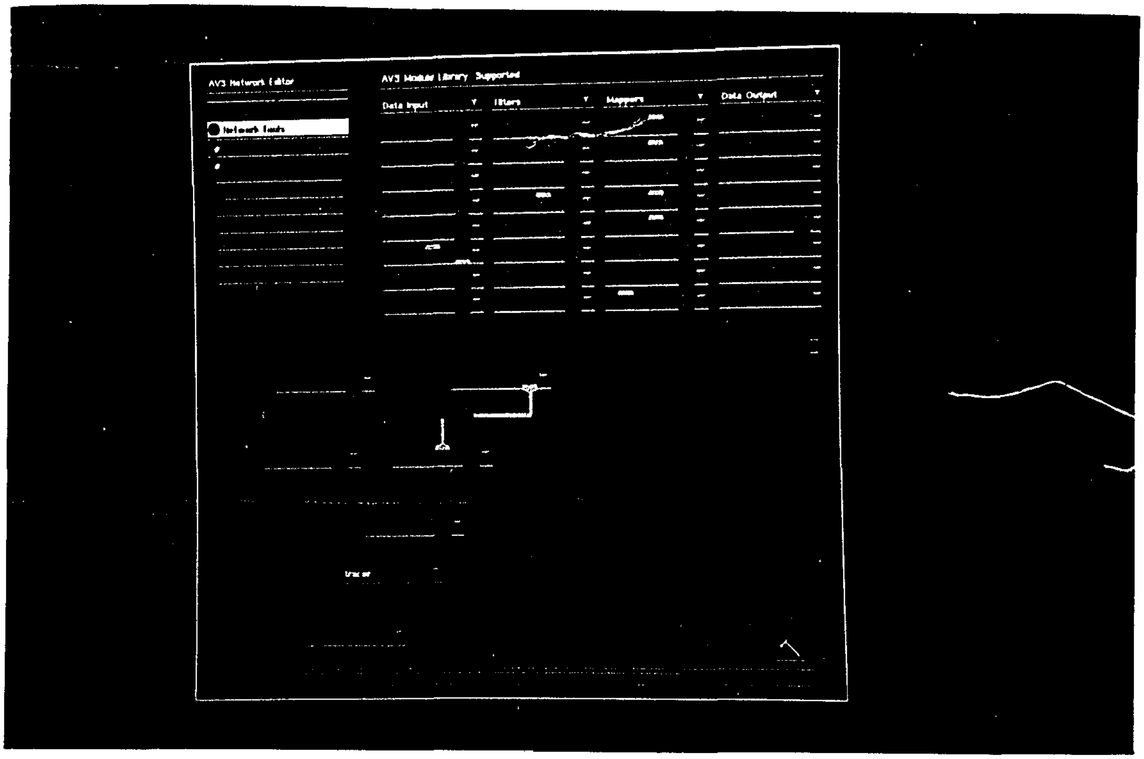


Figure 57: The construction of the *tracer* module on the *AVS network editor.*

A.2 Developed Tools: Object-Oriented Graphics Class Library

An object-oriented graphics class library has been developed by Dennis Gannon and staff members at CICA (the Indiana University Center for Innovative Computer Applications). The objective of this project was to understand the important mathematical abstractions that link various visualization applications and to use the class library to realize these applications[30]. This library has been implemented on the native graphics of many different hardware systems available at CICA in order to serve as a common platform for development. As such, applications built by the users will be extensions of this common class library and can be ported to different systems without any modifications.

We implemented a version of this graphics class library using *C++*, *Doré* and the *X-Window system.* We then developed an interactive system for visualizing and manipulating four-dimensional mathematical objects based on this library.

Important classes

Many classes have been designed and implemented for various kind of graphical objects in the library. Some of the important classes are listed below:

- base_obj. This class is defined for basic graphical object, which in graphics terms, is essentially a segment. One of its purposes is to be a collection of objects from all different classes. All other graphical objects such as polygons, lines, etc., are subclasses of this class. Methods defined in this class are those that are common to all graphical objects. These include methods for performing transformations such as scaling, rotations, and translations, methods for setting an object's color and material properties, and methods to specify an object's graphical representation style.
- text_obj. This class is designed for representing text. It provides users the ability to label and annotate images.
- point_list_obj. This class is designed for representing a point or a list of points.
- line_list_obj. This class is designed for representing a line or a list of lines.
- polygon_obj. This class is designed for representing a polygon or a list of polygons.
- trianglemesh_obj. This class is designed for representing a mesh of triangles.
- analytical_obj. This class is designed for representing analytically defined objects. It contains the following subclasses :

1. analytical_curve. This represents a curve defined via a function of one variable.
2. asurface_of_rotation. This represents a surface defined via a parametric function of one variable.
3. analytical_surface_obj. This represents a surface defined via a parametric function of two variables.

There are methods associated with these classes for setting and changing various parameters on the surface. For example, there are methods for assigning the parametric equations, for changing the bounds in the parametric equations and for changing the dimensions of the grid mesh. We have used analytical_surface_obj to represent projected four-dimensional mathematical objects in our visualization system.

- view_obj. A view_obj is basically a collection of graphical objects, cameras and a frame of reference. Users can add or delete objects to a view, change the camera's parameters and change the lighting parameters.
- slider_obj and button_obj. These objects allow users to add their own interface design. For example, users can assign a function to a slider_obj; when the slider_obj is updated by the user's action, that particular function will be executed. We have used these control objects to implement various kinds of transformations, as well as special animations for investigating new interactive visualization techniques.

User Interface

We used the user interface provided by the *Doré* demonstration software, and added the following extra features:

- **Direct object manipulation.** We added a picking facility to the existing interface so that every graphical object displayed in a view can be picked. Picked object can be scaled, translated and rotated simply by movements of the mouse. The rolling ball technique developed by Andrew Hanson[54], which allows interactive rotation of graphical objects with respect to an arbitrary axis in a context-free manner, has been implemented in the existing interface. These features provide users a better feeling about the graphical objects that are being explored.

- **Interactive manipulation of stereo objects.** Lights and cameras can be added or removed. Intensities, colors, and positions of lights can be modified interactively, as well as the position of the camera and its zoom factor. It is easier for users to play with these parameters without having to recompile the programs each time changes are made (as is needed bare *Doré*).

- **New keyboard commands.** We added many new keyboard commands for manipulating and animating the graphics. As an example, we can have a sequence of commands written on a script file, and then start a long video recording process for several sequences of complicated animations by entering a special command that instructs the program to execute commands given in a script file. Thousands of animation frames can be recorded to videotape or laser disc without any user interaction.

- **Rotating cube.** When the graphical object is too complicated and needs a longer time to render, users can carry out interactive rotation on a simpler cube instead of rotating the original object. When they are satisfied with the rotation angle and stop manipulating the rotating cube, the original object will be rendered with the last selected angle. This is a time-saving feature that allows users to achieve their goal of rotating the object to a desired angle.

- **Sliders.** Sliders in the original interface did not accept keyboard input. We modified the code so that, besides using the mouse to move a slider, users can specify an exact value to be assigned to a slider through keyboard input. Users are thus given better control in manipulating the parameters attached to the sliders.

- **Buttons.** Through the button_obj class in the graphics class library, users can add new buttons in their applications. Special animations and transformations can be activated by users when the corresponding buttons are selected.

- **Remote execution.** We modified the relevant code in the existing user interface so that it is possible for users to execute programs written using this library on any remote machines that support the X-window system, including those with black and white displays.

Advantages

Compared to *Doré*, this library has the following strengths:

- **It is object oriented.** Using *C++*, we transformed *Dorè* into a real object-oriented graphics library. The hierarchical ordering capability of *C++* allows sharing of methods between classes and their subclasses. We used *Virtual functions* to define a set of operations for the most general version of a concept. The interpretation of these operations could be refined for particular special cases in the derived classes.

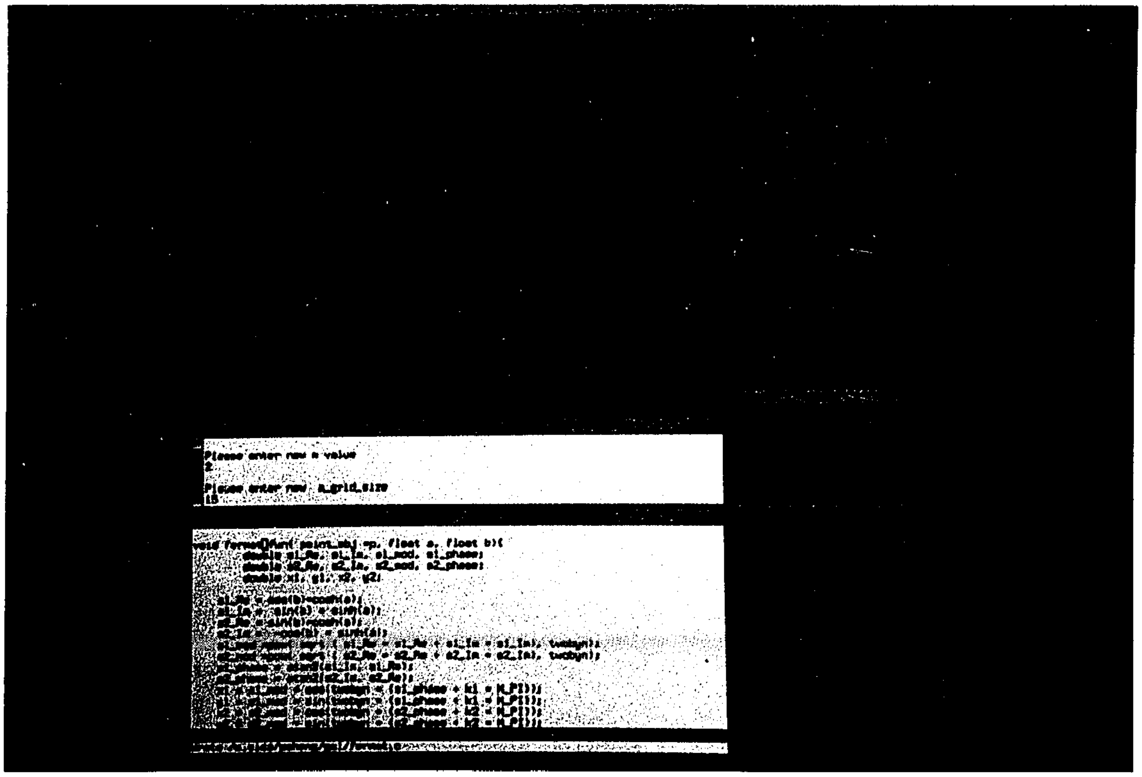
- **It is extensible.** Users can define new classes based on existing classes. New classes can be subclasses of existing classes. They can also be classes that include objects representing various existing classes.

- **Combined advantages of *Dorè*, *C++*, and the *X-window system*.** This library uses Dorè to handle the graphics, the X-window system to handle the user interface, and C++ to handle the programming style. It shares the advantages of all these powerful software systems.

Constraints

There are features that we would like to add to the graphics class library and its user interface design but have not done so due to time constraints. The following are some drawbacks that exist in the current system:

- **Cannot support multiple windows.** By using existing code in the *Doré* demonstration software, we saved substantial time. However, the design of the existing code made it very difficult for us to add support for multiple windows. Too much code needed to be re-written or duplicated in order to achieve that.
- **Too few classes.** A few important classes are still missing in this library. For example, a class for defining 3-manifolds would be extremely useful for mathematical visualization.
- **User interface design inadequacies.** The current user interface does not provide sufficient help to users. Besides buttons, sliders, keyboard input and dial box support, the following useful items should be included:
    1. Pop-up menus to provide users useful information or available options.
    2. Dialogue boxes to provide interactive communications between users and programs.
    3. Highlighting text to show users about the effects or results of an action.
- **Constraints of Doré.** This library is built on top of *Doré*; thus, besides having most of the advantages that *Doré* has, it also shares most of its constraints.



```
Please enter new n value
2
Please enter new n_grid_size
15
```

```
void fermatfun( point_obj *p, float a, float b){
        double s1_Re, s1_Im, s1_mod, s1_phase;
        double s2_Re, s2_Im, s2_mod, s2_phase;
        double x1, y1, x2, y2;
```

Figure 58: User interface of the object-oriented graphics class library.

Figure 58 shows the graphics user interface for the object-oriented graphics class library. The surface being displayed within the graphics window is the $n = 2$ Fermat surface.

Almost all the figures in Chapter 2 are rendered using this graphics class library.

Appendix B

Basic Four-dimensional Geometry

In this work, four-dimensional geometry has been used as the subject of study for developing new visualization techniques and computer graphics methods. Therefore, it is essential for readers to have some basic knowledge of four-dimensional geometry in order to understand the concepts and methods that are presented in this document.

Four-dimensional transformations such as translation, scaling and rotations are presented in Section B.1. In Section B.2, we discuss four-dimensional vector operations including the 4D cross product.

B.1 Four-dimensional Transformations

Most of the four-dimensional transformations are simple extensions of their corresponding three-dimensional operations. The following shows how we can transform an object in four dimensions:

- **Translation.** To translate an object in four dimensions, we simply add the translation distances to each point of the object.

$$x' = x + T_x,\ y' = y + T_y\ z' = z + T_z,\ w' = w + T_w \tag{44}$$

- **Scaling.** To scale an object in four dimensions, we multiply the coordinate value of each point (x, y, z, w) by the scaling factors.

$$x' = xS_x,\ y' = yS_y,\ z' = zS_z,\ w' = wS_w \tag{45}$$

- **Rotations.** To construct the basic rotation matrices for four-dimensional rotations, we first look at rotations in two-dimensional space and in three-dimensional space.

  In two-dimensional space, objects can be rotated about any arbitrary point, and there is only one basic rotation matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} .$$

  In three-dimensional space, objects can be rotated about any arbitrary axes instead of a point, and there are three basic rotation matrices corresponding to rotations about the X, Y, and Z axes:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{pmatrix} \quad \begin{pmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{pmatrix} \quad \begin{pmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

  X Axis Y Axis Z Axis .

  Analogously, objects in four-dimensional space can be rotated about any arbitrary plane instead of an axis. There are now six basic rotation matrices:

$$\begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

XY Plane

$$\begin{pmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

XZ Plane

$$\begin{pmatrix} cos\theta & 0 & 0 & -sin\theta \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ sin\theta & 0 & 0 & cos\theta \end{pmatrix}$$

XW Plane

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

YZ Plane

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & 0 & -sin\theta \\ 0 & 0 & 1 & 0 \\ 0 & sin\theta & 0 & cos\theta \end{pmatrix}$$

YW Plane

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & cos\theta & -sin\theta \\ 0 & 0 & sin\theta & cos\theta \end{pmatrix}$$

ZW Plane .

B.2 Four-dimensional Vector Operations

Almost all of the four-dimensional vector operations are simple extensions of their corresponding three-dimensional methods. If we let $\vec{V_1} = (x_1, y_1, z_1, w_1)$, $\vec{V_2} = (x_2, y_2, z_2, w_2)$ and $\vec{V_3} = (x_3, y_3, z_3, w_3)$ be three 4D vectors, the possible four-dimensional vector operations are:

- **Addition.**

$$\vec{V_1} + \vec{V_2} = (x_1 + x_2, y_1 + y_2, z_1 + z_2, w_1 + w_2) \tag{46}$$

- **Subtraction.**

$$\vec{V_1} - \vec{V_2} = (x_1 - x_2, y_1 - y_2, z_1 - z_2, w_1 - w_2) \tag{47}$$

- **Scalar Product.**
$$\mathbf{k}\vec{V_1} = (kx_1, ky_1, kz_1, kw_1) \tag{48}$$

- **Dot Product.**
$$\vec{V_1} \cdot \vec{V_2} = x_1x_2 + y_1y_2 + z_1z_2 + w_1w_2 \tag{49}$$

- **Cross Product.** To formulate the equation for the four-dimensional cross product, we first look at the three-dimensional cross product. In three-dimensional space, the cross product of two vectors is another vector that is perpendicular to each of the original two vectors. We can use the determinant of the following 3×3 matrix to express the cross product of two vectors $\vec{V_1} = (x_1, y_1, z_1)$ and $\vec{V_2} = (x_2, y_2, z_2)$ in 3D:

$$\vec{V_1} \times \vec{V_2} = \begin{pmatrix} \hat{x} & \hat{y} & \hat{z} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{pmatrix},$$

  where $\hat{x}, \hat{y}$, and $\hat{z}$ are the unit vectors along the X, Y, and Z axes.

  When we take the cross product of three vectors in four-dimensional space, the result is another vector that is perpendicular to each of the three original vectors. We can use the determinant of the following 4×4 matrix to express the cross product of $\vec{V_1}, \vec{V_2}$, and $\vec{V_3}$ in 4D:

$$\vec{V_1} \times \vec{V_2} \times \vec{V_3} = \begin{pmatrix} \hat{x} & \hat{y} & \hat{z} & \hat{w} \\ x_1 & y_1 & z_1 & w_1 \\ x_2 & y_2 & z_2 & w_2 \\ x_3 & y_3 & z_3 & w_3 \end{pmatrix}.$$

Appendix C

Mathematics Background

We introduce some mathematical terms and concepts used in this research. The following references[17, 103, 22, 18, 61, 101, 10] provide more general ideas and definitions of these concepts.

- **Manifold.** *Definition.* A real (complex) n-dimensional *manifold* **M** is a space which looks like a Euclidean space $R^n(C^n)$ around *each point.* More precisely, a manifold is defined by introducing a set of neighborhoods U_i covering **M**, where each U_i is a subspace of $R^n(C^n)$. Thus, a manifold is constructed by pasting together many pieces of $R^n(C^n)$.

- **Geometry versus Topology.** The aspect of a manifold's nature which is unaffected by smooth deformation is called the *topology* of the surface. A manifold's *geometry* consists of those properties which do change when the surface is deformed[103].

- **Local Properties versus Global Properties.** A manifold has both local and global properties. *Local properties* are those observable within a small region of the surface, whereas *global properties* require consideration of the manifold as a whole[103].

- **One-sided versus Two-sided.** Think of a surface as being represented by a membrane and try to determine whether there is a path on it which leads from one side of the surface to the other without crossing the boundary (if any) and without piercing the membrane at a point that is just being traversed by the path. If such a path exists, the surface is called one-sided, otherwise we call it two-sided[61]. For example, a Möbius strip, which can be constructed by joining the two ends of a long, narrow strip of paper after giving the strip a half twist, is one-sided.

- **Orientable versus Non-orientable.** A path in a surface or three-manifold which brings a traveller back to his starting point mirror-reversed is called an orientation-reversing path. Manifolds which do not contain orientation-reversing paths are called orientable; manifolds which do are called nonorientable. For example, a sphere and a torus are orientable surfaces and a Möbius strip is non-orientable[103].

- **Riemann Surface.**

  Riemann surfaces arose in our study of the topology of the Fermat surfaces (homogeneous equations in CP2). The concept of a Riemann surface was introduced by the mathematician Bernhard Riemann in his 1851 dissertation. The main idea of Riemann surface is to modify the domain of the multivalued inverse of an analytic function so that the inverse appears as a single-valued function.

  Let $w = g(z)$ be a multivalued inverse of an analytic function, $z = f(w)$. Although g has several values of w for each z, the number of these values is constant at all points other than branch points. We may make an appropriate choice of branch cut(s) so that all the resulting branches of g have precisely the same domain. Each of these domain copies could be connected to another across their branch-cut boundaries in such a way that the individual function branches would be continuous and analytic across the branch-cut connection. One can then obtain a single, connected domain over which the entire multivalued function can be defined as a single-valued complex function[86].

The following are a few mathematical terms that are used to describe a Riemann Surface:

  - **Branch Points.** A branch point is a point at which the single-valued branches of a multivalued function, $w = g(z)$, are "tied" together. That is, at these branch points, the function $g(z)$ is single-valued. A basic character of a branch point z_0 of a multivalued function $g(z)$ is that if we draw a small simple closed curve about z_0, the values of $g(z)$ along this curve must pass continuously from those of one branch of g to another branch.
  - **Branch Cuts.** A branch cut is a path joining branch points. The values of the branches are continuous across the branch cut.
  - **Riemann sheets.** The domains of those branches are called sheets of the resulting Riemann surface.

- **Projective Spaces.** *Definition.* The projective space P^n is defined as the space obtained by identifying diametrically opposite points on the n-sphere. We call P^1 the *projective line* or the 1-sphere S^1 and P^2 the *projective plane.*

  $CP1 = S^2$ is the set of points (z_0, z_1) equivalent under division. It consists of two coordinate patches, one with coordinate system $z = \frac{z_1}{z_0}$ in the complement of $z_0 = 0$ and the other with coordinate system $z' = \frac{z_0}{z_1}$ in the complement of $z_1 = 0$. These two patches can be viewed as hemispheres which are sewn together to form a 2-sphere.

  $CP2$ is describable by sewing together three different coordinate systems: $(z_1', z_2') = (\frac{z_1}{z_0}, \frac{z_2}{z_0})$ in the complement of the line $z_0 = 0$, $(z_1'', z_2'') = (\frac{z_0}{z_1}, \frac{z_2}{z_1})$ in the complement of the line $z_1 = 0$, and $(z_1''', z_2''') = (\frac{z_0}{z_2}, \frac{z_1}{z_2})$ in the complement of the line $z_2 = 0$.

Appendix D

Examples of Implementation Methods

D.1 Fermat Surface Display

The following segments of code have been used to define the Fermat Surfaces presented in Chapter 2. The code is written using the *object oriented graphics class library.*

```
/* Fermat_function :
Input : The 2 parameters that define a point on the surface : a, b.
Output: A projected 3D point on the Fermat surface : p. */

/* declare some variables */
# define N = 3 /* For Fermat n = 3 surface */
# define TwoByN = 2 / N
double alpha = M_PI_4; /* an angle used in the 4D → 3D projection */
int k1, k2; /* Two integers used in the Fermat parametric equations */

/* The following variables define the mesh grid sizes */
double a_grid_size = 15;
double b_grid_size = 15;

/* The following variables define the range for the two variables, a and b. */
double a_start = -2.0;
double a_stop = 2.0;
double b_start = 0.0;
double b_stop = M_PI_2;

/* The N^2 patches of the Fermat Surface are defined as
   analytical_surface in the graphics library */
analytical_surface *fermat_patch[N][N];

void Fermat_function( point_obj *p, float a, float b)
{
   double x1, y1, x2, y2, s1_Re, s1_Im, s1_mod, s1_phase;
   double s2_Re, s2_Im, s2_mod, s2_phase;

   s1_Re = cos(b)*cosh(a);
   s1_Im = sin(b) * sinh(a);
   s2_Re = sin(b)*cosh(a);
   s2_Im = -cos(b) * sinh(a);

   s1_mod =pow( sqrt ( s1_Re * s1_Re + s1_Im * s1_Im), TwoByN);
   s2_mod =pow( sqrt ( s2_Re * s2_Re + s2_Im * s2_Im), TwoByN);
   s1_phase = atan2(s1_Im, s1_Re);
   s2_phase = atan2(s2_Im, s2_Re);

   x1 = s1_mod * cos(TwoByN * (s1_phase + k1 * M_PI));
   y1 = s1_mod * sin(TwoByN * (s1_phase + k1 * M_PI));
   x2 = s2_mod * cos(TwoByN * (s2_phase + k2 * M_PI));
   y2 = s2_mod * sin(TwoByN * (s2_phase + k2 * M_PI));
```

```
    /* 4D → 3D Projection */
    p→x = x1;
    p→y = x2;
    p→z = (y1 * cos(alpha) + y2 * sin(alpha));
}

    /* The following function is used to generate the Fermat surfaces */
void generate_fermat_surfaces(view_obj *view, window_obj *theWindow)
{

    /* The following for loop initialize the N^2 patches of the Fermat surface */
    for (k1 = 0; k1 < N; k1++)
        for (k2 = 0; k2 < N; k2++) {

            /* Initializing the surface patch */
            fermat_patch[k1][k2] = new(analytical_surface);
            fermat_patch[k1][k2]→assign_name(”fermat_patch”, NULL);

            /* Assign color to the surface patch based on the values of k1
                and k2 */
            fermat_patch[k1][k2]→Set_DiffuseColor(.6 +k1*.2, 0.6 +k2*.2 ,
                0.5, 1.0, 1);

            /* Setting the mesh grid sizes of the surface patch */
                fermat_patch[k1][k2]→set_grid_size(a_grid_size, b_grid_size);
        /* Setting the parameter ranges for the two variables defined
           in the Fermat equations */
        fermat_patch[k1][k2]→set_param_bounds( a_start, a_stop,
           b_start, b_stop);

        /* Assign the Fermat_function with the current values of k1 and k2
           to the particular surface patch */
        fermat_patch[k1][k2]→set_fun(Fermat_function);

        /* Adding the surface patch to the view_obj */
        view→add_object_to_scene(fermat_patch[k1][k2]);
    }

/* Asking the window_obj to render itself */
theWindow→render();
}
```

D.2 Thickening the Steiner Surface

The following program has been used to compute the 4D coordinate and the normal vectors of the Steiner surface that is swept with a circle attached at each point. Images of this thicken Steiner surface are shown in Chapter 4.

```
/* Steiner_function :
   Input : The 3 parameters that define a point : a, b, and β.
   Output: A 4D point on the surface and its 4D normalized normal vector. */
   extern double mat[4][4]; /* The rotation transformation matrix */
   extern double r; /* The radius of the attached circle */
```

```
void Steiner_function(a, b, β, point, normal)
double a, b,β;
point4d_obj *point, *normal;
{
  double u, v, ca, cb, sa, sb, c2a, c2b, s2a, s2b;
  double DotProd, B1, B2, B3, E1, E2, E3;
  point4d_obj pt4d, p4d, t1u, t2u, t1, t2, temp ;
  point4d_obj n1, n2, qu, nn1, nn2, n2u, n1t;

  /* Compute the 4D coordinate of the normal Steiner Surface using the
     parameters a and b only */

  ca = cos(a); cb = cos(b); sa = sin(a); sb = sin(b);
  u = ca * cb;
  v = sa * cb;

  pt4d.x = u * u - v * v ;
  pt4d.y = u * v;
  pt4d.z = u * sb;
  pt4d.w = v * sb;

  /* Compute the two tangent vector, t1 and t2 */
  c2a = ca * ca; c2b = cb * cb;
  s2a = sa * sa; s2b = sb * sb;

  t1.x = -4.0 * ca * c2b * sa;
```

```
t1.y = c2b * (c2a - s2a);
t1.z = -cb * sa * sb;
t1.w = ca * cb * sb ;

t2.x = -2 * c2a * cb * sb + 2 * cb * s2a * sb;
t2.y = -2.0 * ca * cb * sa * sb;
t2.z = ca * c2b - ca * s2b ;
t2.w = c2b * sa - sa * s2b ;
normalize4d(t1, &t1u); /* $t\vec{1}u = \frac{\vec{t1}}{\|\vec{t1}\|}$ */
normalize4d(t2, &t2u);


/* Perform Gram-Schmidt orthonormalization */

copy4d(pt4d, &n2); /* $n\vec{2} = pt\vec{4}d$ */
n2.x = n2.x -0.5;
normalize4d(n2, &n2u);

DotProd = dot4d(t1u,n2u); /* DotProd = $t\vec{1}u \cdot n\vec{2}u$*/
scalar4d(DotProd,t1u,&temp); /* $te\vec{m}p = DotProd * t\vec{1}u$ */
minus4d(n2u, temp, &n1t); /* $n\vec{1}t = n\vec{2}u - te\vec{m}p$ */

DotProd = dot4d(n1t,t2u);
scalar4d(DotProd,t2u,&temp);
minus4d(n1t, temp, &n1);
normalize4d(n1, &nn1);
```

```
/* Convention is $\epsilon_{xyzw}$ = +1, so 1st row is :
   Axx = 0, Axy = +B3, Axz = -B2, Axw = -E1 */

B1 = - t1u.w * t2u.x + t1u.x * t2u.w;
B2 = - t1u.w * t2u.y + t1u.y * t2u.w;
B3 = - t1u.w * t2u.z + t1u.z * t2u.w;

E1 = -t1u.y * t2u.z + t1u.z * t2u.y;
E2 = -t1u.z * t2u.x + t1u.x * t2u.z;
E3 = -t1u.x * t2u.y + t1u.y * t2u.x;


/* This is the 4D cross-product of 3 vectors to give
   a fourth vector that is perpendicular to the other 3 */
n2.x = 0.0 * nn1.x + B3 * nn1.y + (-B2) * nn1.z - E1 * nn1.w;
n2.y = -B3 * nn1.x + 0.0 * nn1.y + B1 * nn1.z - E2 * nn1.w;
n2.z = B2 * nn1.x + (-B1) * nn1.y + 0.0* nn1.z - E3 * nn1.w;
n2.w = E1 * nn1.x + E2 * nn1.y + E3 * nn1.z + 0.0 * nn1.w;
normalize4d(n2, &nn2);

/* qu is the normalized vector to surface of circle lying in plane
   perpendicular to Steiner surface at pt4d. (In local coords.)
   β parameterizes this circle. */
scalar4d(cos(β), nn1, &n1);
scalar4d(sin(β), nn2, &n2);
add4d(n1, n2, &qu);
```

```
    /* p4d is the point on surface with a swept circle of radius r,
       in local coords. */

    scalar4d(r, qu, &temp);
    add4d(pt4d, temp, &p4d); /* $\vec{p4d} = \vec{pt4d} + \vec{temp}$ */

    /* Rotate the normal to world coords, rotate the point
       on swept surface to world coords. */
    apply4d(mat, qu, normal);
    apply4d(mat, p4d, point);
    return;
}
```

Bibliography

[1] ADVANCED VISUAL SYSTEMS INC. *AVS Tutorial Guide.*

[2] APÉRY, F. *Models of the Real Projective Plane.* Vieweg, Brahnschweig, 1987.

[3] BAJAJ, C. J. Rational hypersurface display. *SIGGRAPH Computer Graphics 24*, 2 (March 1990), 117–127.

[4] BAKER, H. Building surfaces of evolution: the weaving wall. *International Journal of Computer Vision 3*, 1 (1989), 51–71.

[5] BANCHOFF, T. The hypercube: Projections and slicing. International Film Bureau, Chicago, Ill.

[6] BANCHOFF, T. The hypersphere: Foliation and projections. Banchoff T. Productions, Providence, R.I.

[7] BANCHOFF, T. Computer graphics in geometric research. In *Recent Trends in Mathematics* (1982), Teubner-Verlag, Leipzig, pp. 317–327.

[8] BANCHOFF, T. Visualizing two-dimensional phenomena in four-dimensional space: A computer graphics approach. In *Statistical Image Processing and Computer Graphics* (1986), Marcel Dekker, Inc., New York, pp. 187–202.

[9] BANCHOFF, T. *Using Computer Graphics to Explore the Generation of Surfaces in Four Dimensional Space.* Prime Computer Inc., 1987.

[10] BANCHOFF, T. *Beyond the Third Dimension:Geometry, Computer Graphics, and Higher Dimensions.* Scientific American Library, New York, 1990.

[11] BANCHOFF, T. Computer graphics tools for rendering algebraic surfaces and for geometry of order. In *Geometric Analysis and Computer Graphics* (1991), Springer-Verlag New York Inc., pp. 31–37.

[12] BANCHOFF, T., FEINER, S., AND SALESIN, D. Dial: A diagrammatic animation language. *IEEE Computer Graphics and Applications 2*, 7 (July 1982), 42–55.

[13] BANCHOFF, T., AND STRAUSS, C. Complex function graphs, dupin cyclides, gauss map, and veronese surface. Computer Geometry Films (Brown University, Providence), 1977.

[14] BANCHOFF, T., AND STRAUSS, C. Real-time computer graphics analysis of figures in four-space. In *Hypergraphics - Viusalizing Complex Relationships in Art, Science and Technolog* (1978), Westview Press, Buolder, Colorado, pp. 159–167.

[15] BANKS, D. A demostration presented at Siggraph '91 in Las Vegas.

[16] BARR, A. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications 1 1* (January 1981), 11–23.

[17] BARSKY, B., BAJAJ, C., DEROSE, T., HOFFMANN, C., AND WARREN, J. Unifying parametric and implicit surface representations for computer graphics. SIGGRPAH 1990, Course Notes.

[18] BERGER, M., AND GOSTIAUX, B. *Differential Geometry: Manifolds, Curves, and Surfaces.* Springer-Verlag, 1988.

[19] BESHERS, C., AND FEINER, S. Real-time 4d animation on a 3d graphics workstation. In *Proceedings of Graphics Interface '88* (1988).

[20] BLINN, J. The mechanical universe. A series of half-hour television programs on elementary college physics and calculus sponsored by the Annenberg/CPB Project (1985-1987). Distributed by FIlms, Incorporated (Chicago, Illinois). Several excerpts from these tapes by James Blinn have appeared in recent issues of the SIGGRAPH video review (1989,1990).

[21] BLINN, J. Project mathematics! A series of videotapes being created at the California Institute of Technology for the teaching of high school level mathematics. Several excerpts from these tapes by James Blinn have appeared in recent issues of the SIGGRAPH video review (1989,1990).

[22] BOOTHBY, W. *An Introduction to DIfferentiable Manifolds and Riemannian Geometry*, second ed. Academic Press, Inc., 1986.

[23] BORNING, A., AND DUISBERG, R. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics 5*, 4 (October 1986), 345–374.

[24] BRISSON, D., Ed. *Hypergraphics: Visualizing Complex Relationships in Art, Science and Technology.* Westview Press, 1978.

[25] BRUN, R., COUET, C. V., AND ZANARINI, P. *PAW - Physics Analysis Workstation, The Complete Reference*, version 1.07 ed. CERN, Geneva, Switzerland, October 1989.

[26] CALLAHAN, M., HOFFMAN, D., AND HOFFMAN, J. Computer graphics tools for the study of minimal surfaces. *Communication of the ACM, Vol. 31, No. 6* (June 1988), 648–661.

[27] CAREY, S., BURTON, R., AND CAMPBELL, D. Shades of a higher dimension. *Computer Graphics World* (October 1987), 93–94.

[28] Carlbom, I., and Paciorek, J. Planar geometric projections and viewing transformations. *Computer Surveys 10*, 4 (December 1978), 465–502.

[29] Carpenter, L. The a-buffer, an antialiased hidden surface method. In *Proceedings of Siggraph '84* (Minneapolis, Minn, July 1984), pp. 103–108.

[30] Center for Innovative Computer Application of Indiana University. *Graphics and Visualization*, June 1990.

[31] Chen, M., Mountford, S. J., and Sellen, A. A study in interactive 3-d rotation using 2-d control devices. In *Proceedings of 1988 SIGGRAPH* (Atlanta, Georgia, August 1988), pp. 121–130.

[32] Cline, H., Lorensen, W., Ludke, S., Crawford, C., and Teeter, B. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics 15*, 3 (May/June 1988), 320–327.

[33] Cook, R. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 1 (January 1986), 51–72.

[34] Cox, D., Francis, C., and Idaszak, R. The etruscan venus. Videotape produced at National Center for Supercomputing Applications, University of Illinois, 1987.

[35] Draves, S. *Fnord examples.* Brown University.

[36] Drebin, R., Carpenter, L., and Hanrahan, P. Volume rendering. *Computer Graphics 22*, 4 (August 1988), 64–75.

[37] Duff, T. Compositing 3-d rendered images. In *Proceedings of Siggraph '85* (San Francisco, Calif., July 1985), pp. 41–44.

[38] FEIBUSH, E., LEVOY, M., AND COOK, R. Synthetic texturing using digital filters. In *Proceedings of Siggraph '80* (Seattle, Wash, July 1980), pp. 294–301.

[39] FEINER, S., AND BESHERS, C. Worlds within worlds : Metaphors for exploring n-dimensional virtual worlds. In *Proceedings of UIST '90* (Snowbird, Utah, October 1990), pp. 76–83.

[40] FOLEY, F., WALLACE, V., AND CHAN, P. The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications 4*, 11 (November 1984), 13–48.

[41] FOLEY, J., VAN DAM A., FEINER, S., AND HUGHES, J. *Computer Graphics: Principles and Practice.* Addison Wesley, 1990.

[42] FRANCIS, G. *Topological Picturebook.* Springer-Verlag, New York, 1987.

[43] FRANCIS, G., AND BERNARD, M. Arnold shapiro's eversion of the sphere. *Math. Intelligencer*, 2 (1979), 200–203.

[44] FUCHS, H., KEDEM, Z., AND USELTON, S. Optimal surface reconstruction from planar contours. *Communications of the ACM 20*, 10 (1977), 693–702.

[45] FUCHS, H., LEVOY, M., AND PIZER, S. Interactive visualization of 3d medical data. *IEEE Computer* (August 1989), 46–51.

[46] GOURAUD, H. Continuous shading of curved surfaces. *Communications of the ACM 18*, 6 (June 1975), 623–629.

[47] GRANT, C. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. In *Proceedings of Siggraph '85* (San Francisco, Calif., July 1985), pp. 79–84.

[48] GRIFFITHS, P., AND HARRIS, J. *Principles of Algebraic Geometry.* Wiley, 1978.

[49] GUNN, C., AND MAXWELL, D. Not knot. Movie and Supplement. Jones and Bartlett Publishers, Inc., 1991.

[50] GUNN, C., AND VAIL, A. Geometry supercomputer project quarterly newsletters vol. 1, numbers 1-4, 1990.

[51] HALL, R. *Illumination and Color in Computer Generated Imagery,* first ed. Springer-Verlag, New York Inc., 1989.

[52] HANSON, A. J. Dual n-point functions in pgl(n-2,c)-invariant formalism. *Physical Review 5,* 8 (April 1972), 1948–1956.

[53] HANSON, A. J. Spatial intuition. A videotape that reports partial results of NSF grant IST-8511751, 1988.

[54] HANSON, A. J. The rolling ball. In *Graphics Gems III* (1992), Academic Press, San Diego, pp. 51–60.

[55] HANSON, A. J., AND HENG, P. A. Visualizing the fourth dimension using geometry and light. In *Proceedings of Visualization 91* (San Diego, California, October 1991), IEEE Computer Society Press, pp. 321–329.

[56] HANSON, A. J., AND HENG, P. A. Visualizing the fourth dimension using geometry and light (videotape). In *Video Proceedings of Visualization 91* (San Diego, California, October 1991), IEEE Computer Society Press.

[57] HANSON, A. J., AND HENG, P. A. Four-sight (videotape). In *SIGGRAPH 92 Computer Graphics Screening Room* (Chicago, Ill., July 1992).

[58] HANSON, A. J., AND HENG, P. A. Illuminating the fourth dimension. *IEEE Computer Graphics and Application 12*, 4 (July 1992), 54–62.

[59] HANSON, A. J., HENG, P. A., AND KAPLAN, B. C. Techniques for visualizing fermat's last theorem: A case study. In *Proceedings of Visualization 90* (San Francisco, CA, October 1990), IEEE Computer Society Press, pp. 97–106.

[60] HANSON, A. J., HENG, P. A., AND KAPLAN, B. C. Visualizing fermat's last theorem (videotape). In *SIGGRAPH 90 Computer Graphics Screening Room* (Dallas, Texas, August 1990).

[61] HILBERT, D., AND COHN-VOSSEN, S. *Geometry and the Imagination.* Chelsea Publishing Company, New York, 1952.

[62] HOFFMAN, D. The computer-aided discover of new embedded minimal surfaces. *Mathematical Intelligence* 9 (1987), 8–21.

[63] HOFFMANN, C., AND ZHOU, J. Displaying manifolds in four-dimensional space. *Purdue University Computer Science Department Report: Computing About Physical Objects* (Fall 1991).

[64] HOLLASCH, S. Four-space visualization of 4d objects. Master's thesis, Arizona State University, August 1991.

[65] HORN, B. *Robert Vision,* first ed. MIT Press, 1986.

[66] IKEUCHI, K., AND HORN, B. Numerical shape from shading and occluding boundaries. *Artificial Intelligence 17* (1981), 141–184.

[67] JOY, K., GRANT, C., MAX, N., AND HATFIELD, L. *Computer Graphics: Image Synthesis.* IEEE Computer Society Press, 1988.

[68] KAUFMAN, A. *Volume Visualization.* IEEE Computer Society Press, 1991.

[69] KIRBY, R. *The Topology of 4-Manifolds, Lecture Notes in Mathematics 1374.* Springer-Verlag, 1989.

[70] KRUEGER, W. The application of transport theory to visualization of 3-d scalar data fields. *Computers in Physics 17* (July-August 1991), 397–406.

[71] KUBOTA PACIFIC COMPUTER INC. *Doré Programer's Guide.*

[72] KUBOTA PACIFIC COMPUTER INC. *Doré Reference Manual.*

[73] LASZLO, M. J. Techniques for visualizing 3-dimensional manifolds. In *Proceedings of Visualization 90* (San Francisco, CA, October 1990), IEEE Computer Society Press, pp. 342–352.

[74] LEVOY, M. Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (March 1988), 29–37.

[75] LEVOY, M. Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261.

[76] LEVOY, M. A taxonomy of volume visualization algorithms. In *Viusalization '90, Volume Visualization Tutorial Notes* (1990), IEEE Computer Society, pp. 6–12.

[77] LORENSEN, W., AND CLINE, H. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Computer Graphics 21*, 4 (July 1987), 163–169.

[78] MABDELBROT, B. *Fractals Form, Chance, and Dimension.* W.H.Freeman, San Francisco, 1977.

[79] MABDELBROT, B. *The Fractal Geometry of Nature.* W.H.Freeman, New York, 1982.

[80] MAEDER, R. *Programming in Mathematica.* Addison-Wesley Publishing Company, Inc., 1990.

[81] MAX, N. Everting the sphere. Distributed by International Film Bureau, Chicago, Illinois.

[82] MAX, N., HANRAHAN, P., AND CRAWFIS, P. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics 24*, 5 (December 1990), 27–33.

[83] MCCORMICK, B., DEFANTI, T., AND BROWN, M. Visualization in scientific computing. *Siggraph Computer Graphics 21*, 6 (November 1987).

[84] NIELSON, G., FOLEY, T., HAMANN, B., AND LANE, D. Visualizing and modeling scattered multivariate data. *Computer Graphics and Applications 11*, 3 (May 1991), 47–55.

[85] NIELSON, G., AND HAMANN, N. Techniques for the interactive visualization of volumetric data. In *Proceedings of Visualization 90* (San Francisco, California, October 1990), IEEE Computer Society Press, pp. 45–50.

[86] PALIOURAS, J., AND MEADOWS, D. *Complex Variables for Scientists and Engineers*, second ed. Macmillan Publishing Company, New York, 1990.

[87] PEITGEN, H.-O., AND RICHTER, P. *The Beauty of Fractals - Images of Complex Dynamical Systems.* Springer-Verlag, 1986.

[88] PETERSON, I. A different dimension. *Science News*, 135 (October 1979), 328–330.

[89] PETERSON, I. *The Mathematical Tourist.* Freeman, New York, 1988.

[90] PETERSON, I. *Islands of Truth.* Freeman, New York, 1990.

[91] PHONG, B.-T. Illumination for computer generated pictures. *Communication of the ACM 18*, 6 (June 1975), 311–317.

[92] SABELLA, P. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics 22*, 4 (July 1988), 51–58.

[93] SHIRLEY, P., AND TUCHMAN, A. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Computer Graphics 24*, 5 (November 1990), 63–70.

[94] SMALE, S. A classification of immersions of the two-sphere. *Trans. Amer. Math. Soc.*, 90 (1959), 879–882.

[95] STEINER, K., AND BURTON, R. Hidden volumes: The 4th dimension. *Computer Graphics World* (February 1987), 71–74.

[96] THURSTON, W., AND WEEKS, J. The mathematics of three-dimensional manifolds. *Scientific American 251*, 1 (1984), 108–121.

[97] TURKOWSKI, K. Anti-aliasing through the use of coordinate transformations. *ACM Transactions on Graphics 1*, 3 (July 1982), 215–234.

[98] UPSON, C., FAULHABER, T., KAMINS, D., LAIDLAW, D., SCHLEGEL, D., VROOM, J., AND VAN DAM, A. The application visualization system : A computational environment for scientific visualization. *IEEE Computer Graphics and Applications* (July 1989), 30–42.

[99] UPSON, C., AND KEELER, M. V-buffer: Visible volume rendering. *Computer Graphics 22*, 4 (August 1988), 59–64.

[100] VANDEWETTERING, M. The application visualization system–avs 2.0. *Pixel* (July/August 1990), 30–33.

[101] WALLACE, A. *Differential Topology: First Steps.* Benjamin, Inc., New York, 1968.

[102] WAVEFRONT TECHNOLOGIES, SANTA BARBARA, CA. *The Advanced Visualizer User's Guide*, 1992.

[103] WEEKS, J. *The Shape of Space.* Marcel Dekker, Inc, 1985.

[104] WESTOVER, L. Footprint evaluation for volume rendering. *Computer Graphics 24*, 4 (August 1990), 367–376.

[105] WILHELMS, J., CHALLINGER, J., ALPER, N., RAMAMOORTHY, S., AND VAZIRI, A. Direct volume rendering of curvilinear volumes. *Computer Graphics 24*, 5 (December 1990).

[106] WILHELMS, J., AND GELDER, A. V. A coherent projectin approach for direct volumer rendering. *Computer Graphics 25*, 4 (July 1991), 275–284.

[107] WOLFRAM, S. *Mathematica-A System for Doing Mathematics by Computer*, second ed. Addison-Wesley Publishing Company, Inc., 1991.

[108] WOODHAM, R. Photometric stereo: A reflectance map technique for determining surface orientation from intensity. In *Proceedings of 22nd International Symp. Society of Photo-Optical Instrumentation Engineers* (San Diego, CA, August 1978), pp. 136–143.

Curriculum Vitae

Pheng Ann Heng was born on 15 November 1961 in Singapore. He graduated with a Bachelor of Science degree in Computer Science from the National University of Singapore in 1985. He began graduate study at Indiana University, Bloomington, in 1985; he received a Master of Science degree in Computer Science in 1987 and a Master of Arts degree in Mathematics in 1988.