

Interacting With Surfaces In Four Dimensions Using Computer Graphics

TR93-011
March, 1993



David Banks

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

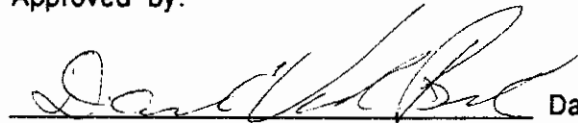
Interacting With Surfaces In Four Dimensions
Using Computer Graphics

A dissertation submitted to the faculty of the University of North Carolina in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

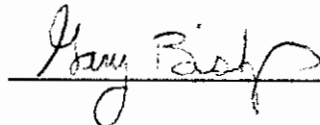
David Banks

Chapel Hill, 1993

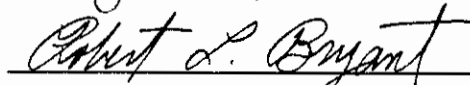
Approved by:



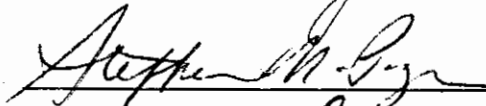
David V. Beard



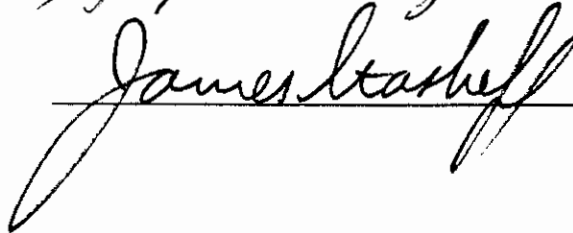
T. Gary Bishop



Robert L. Bryant



Stephen M. Pizer (Advisor)



James D. Stasheff

© 1993

David Cotton Banks

ALL RIGHTS RESERVED

DAVID COTTON BANKS. *Interacting With Surfaces in Four Dimensions Using Computer Graphics* (Under the direction of STEPHEN M. PIZER.)

ABSTRACT

High-speed, high-quality computer graphics enables a user to interactively manipulate surfaces in four dimensions and see them on a computer screen. Surfaces in 4-space exhibit properties that are prohibited in 3-space. For example, non-orientable surfaces may be free of self-intersections in 4-space. Can a user actually make sense of the shapes of surfaces in a larger-dimensional space than the familiar 3D world? Experiment shows he can. A prototype system called Fourphront, running on the graphics engine Pixel-Planes 5 (developed at UNC-Chapel Hill) allows the user to perform interactive algorithms in order to determine some of the properties of a surface in 4-space. This dissertation describes solutions to several problems associated with manipulating surfaces in 4-space. It shows how the user in 3-space can control a surface in 4-space in an intuitive way. It describes how to extend the common illumination models to large numbers of dimensions. And it presents visualization techniques for conveying 4D depth information, for calculating intersections, and for calculating silhouettes.

Acknowledgements

Many minds, many hands, and many pockets contributed to this research. My sincerest thanks go to

Brice Tebbs and Greg Turk for teaching me about computer graphics and showing me how to program Pixel-Planes;

Howard Good (callback functions, one-sided polygon picking), Marc Olano (conditional executes, parallel picking), David Ellsworth (animated cursor, parallel picking), and Andrew Bell (multipass transparency) for contributing code to improve the Fourphront system;

Trey Greer ("Front"), Howard Good ("Pphront"), and Vicki Interrante ("Xfront") for developing the 3D progenitors of Fourphront;

David Beard, Gary Bishop, Robert Bryant, Steve Pizer, and Jim Stasheff for serving on my committee;

Oliver Steele for helping to create 4D datasets;

Nelson Max for suggesting that intersection curves be highlighted;

Trey Greer for teaching me how to write an X-window interface;

Ross Whitaker for suggesting the use of the shape operator in determining fixed-width silhouettes;

David Eberly for solving the differential equations for constant curvature in 4-space;

Robert Bryant for suggesting I use Morse Theory and investigate quaternion rotations;

Russell Taylor for helping to port Fourphront to run with the head-mounted display;

David Harrison for teaching me how to edit videotape;

Howard Lander and Data General Corporation for contract support;

Myron Banks and Warren Herron for their generosity;

Digital Equipment Corporation for scholarship funding;

Terry Yoo and the NSF (Gang of Five) for travel and production support;

Elizabeth Banks for her tremendous love, patience, and encouragement.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS	viii
LIST OF PLATES	xi
1 Overview.....	1
1.1 Goal of the Research	1
1.2 Why Surfaces in 4-space are Interesting.....	3
1.2.1 About Large Dimensions.....	4
1.2.2 About 4-space.....	5
1.2.3 About Surfaces.....	6
1.3 Related Work.....	7
1.3.1 Computer Graphics.....	7
1.3.2 Mathematics	17
1.3.3 Pedagogy	18
2 The Basics of Graphics in 4-space.....	21
2.1 How to Map Input From User Space to Object Space	23
2.1.1 Mapping 2D Input to 3D Transformations	24
2.1.2 Mapping 3D Input to 4D Transformations	32
2.1.3 Simple Approximations of Rotation Matrices.....	39
2.2 Illumination	43
2.2.1 Review of Illuminating Surfaces in 3-space.....	43
2.2.2 Illumination Independent of Dimension	44
2.2.3 Why it Matters	45
2.3 Depth Cues.....	49
2.3.1 Occlusion and Shadows.....	49
2.3.2 Texture and Color.....	49
2.3.3 Focus and Transparency	50
2.3.4 Perspective and Parallax	50
2.4 Projecting From Object Space to Screen Space	55
2.4.1 Stereopsis and Parallax	58
2.4.2 Mixing Rotations, Projections, and Translations.....	69

3 Visualization Tools	64
3.1 Ribboning	64
3.2 Clipping	67
3.3 Transparency	69
3.4 Paint	78
3.5 Silhouettes	85
3.5.1 Varying-width Silhouette Curves	85
3.5.2 Fixed-width Silhouette Curves	88
3.6 Intersections	97
3.6.1 Varying-width Intersection Curves	98
3.6.2 Fixed-width Intersection Curves	99
4 Interactive Algorithms	108
4.1 Intersections in 4-space	108
4.2 Orientability	116
4.3 Genus	124
5 Observations of Interactive Sessions	131
5.1 Intersections	131
5.2 Orientability	132
5.3 Genus	134
5.3.1 Recognizing the Shape of a Surface	134
5.3.2 Applying Morse Theory to a Surface	135
5.3.3 Finding Critical Points at Pinch Points	137
5.4 Links and Knots	138
5.4.1 Knotted Torus	138
5.4.2 Linked Spheres	138
5.4.3 Knotted Sphere	139
5.5 Constant Curvature in 4-space	144
5.6 Quaternion Rotations	147
5.7 Intuition	147
6 Conclusions and Future Work	150
6.1 User Interface	150
6.2 Illumination	151
6.3 Depth Cues	152
6.4 Visualization Tools	152
6.4.1 Seeing Into a Surface	152
6.4.2 Applying Paint	153
6.4.3 Locating Intersections and Silhouettes in Screen-space	153

6.5 Results from Interactive Sessions	154
6.6 Future Work	155
Appendix A Pixel-Planes 5	157
A.1 Hardware Architecture	157
A.2 Renderer Instructions	158
A.3 An Abstract Programming Model of the Renderers.....	160
A.4 PPHIGS.....	162
Appendix B: Fourphront	163
B.1 System Capabilities	163
B.2 User Interface	164
B.3 System Performance.....	169
References	170

LIST OF ILLUSTRATIONS

Figure 1-1.	The six faces on this octagonal figure correspond to the six axial planes of four-dimensional space.....	13
Figure 2-1.	A user in 3-space manipulates a surface in 4-space, which projects to 3-space and then onto the screen.....	22
Figure 2-2.	These are three of the six axial planes in $xyzw$ -space, defined by the axis pairs xw , yw , and zw . The other three axial planes (xz , yz , and xy) lie in the 3-dimensional xyz -subspace.	23
Figure 2-3.	At the bottom point of this circular trajectory, the mouse's velocity is purely horizontal, while its acceleration is purely vertical.	28
Figure 2-4.	The mappings of 3D input space to 4D world space that promote kinesthetic sympathy.....	35
Figure 2-5.	The 3D joystick rotates in the $x'z'$, $y'z'$, and $x'y'$ planes, which can produce a momentary translation in the x and the y directions. In the input space coordinates, x' is rightward, y' is forward, and z' is vertical.	36
Figure 2-6.	The mappings of spaceball and joystick input that promote kinesthetic sympathy in 4D world space.	37
Figure 2-7.	Moving the joystick produces a rotation. The velocity of rotation is proportional to the amount that the joystick is displaced from its centered, "rest" position.	39
Figure 2-8.	Diffuse and specular illumination at a point on a surface are governed by the surface normal N , the light vector L , and the view vector V	43
Figure 2-9.	If the codimension is 2, the normal space at a point on the surface will be a plane N	44
Figure 2-10.	Opacity can be used as a depth cue in the w -direction without significantly interfering with the usual depth cues in the z -direction. For example, points on the surface near the eye are rendered nearly transparent, while points far away in w are nearly opaque. Fourphront uses a linear opacity ramp that is clamped at controllable near and far values.....	50
Figure 2-11.	The $(x\ y\ z\ w)$ -axes (left) project in the w -direction to the $(x\ y\ z)$ -axes (middle), which project in the z -direction to the $(x\ y)$ -axes of the image plane.....	55
Figure 2-12.	When there are two eye positions involved in projecting an image, either of them can produce parallax. In this figure, spheres A and B project from 3-space onto a 2-dimensional plane as disks. The disks project to a 1-dimensional line as segments. By tilting the page obliquely, you can see what the second eye sees. Moving an eye to the right will make the farther object seem to move to the right of the nearer object. Which sphere looks closer? It depends on which eye does the measuring. A is closer to eye3 than B is. But the projection of B is closer to eye2 than the projection of A is.	58

Figure 3-1.	Clipping into a torus produces a figure-eight contour. Clipping reveals internal geometry, but complex contours can confuse the shape.....	67
Figure 3-2.	A polygon can have different colors on its front and back sides.....	79
Figure 3-3.	The surface normal is nearly orthogonal to the eye vector in the vicinity of a silhouette curve.	87
Figure 3-4.	Three points, together with their normals, can be used to estimate the surface's curvature. The dark curve represents the silhouette as seen from the eye position.....	88
Figure 3-5.	The other two vertices from the surface can be used to produce an orthonormal basis for the tangent plane at the point p.....	89
Figure 3-6.	The original surface (white) is approximated by a quadric surface (dark) derived from the shape operator.	90
Figure 3-7.	The approximating surface has its own silhouette curve, as seen from the eye position. The angle between the point p and the silhouette curve measures their distance apart when displayed on the screen.....	91
Figure 3-8.	Now the point p can be compared against a fixed-width threshold based on the quadric approximating surface.	93
Figure 3-9.	At their common intersection, two polygons share z-values. The z-values are within some threshold of each other along a thickened intersection curve.	98
Figure 3-10.	The goal is to determine whether a point on a surface is within d of the intersection. Depending on the slopes of the two surfaces, the point must lie within some ϵ (in the z-direction) of the other surface.....	99
Figure 3-11.	This is the default usage for the 208 bits of per-pixel memory.....	101
Figure 3-12.	The triangle mesh covers the (x,y)-range of the screen when it is parallel to the screen. When it rotates a quarter turn, it covers the range of the z-buffer.	101
Figure 3-13.	Fourfront scavenges bits from the memory-transfer area and from the texture area of the default usage. These bits are used to compute the fixed-width intersection curve.....	102
Figure 4-1.	Two polygons in 4-space (left) intersect a plane which will project to a line in 3-space (right). As a result, the projected polygons will intersect in 3-space.	109
Figure 4-2.	A painted polygon P and an unpainted polygon U share a common edge (left). An intervening polygon I intersects P and U along their shared edge, hiding P (middle). Rotation reveals P, but now I hides U (right).....	117
Figure 5-1.	A Whitney umbrella. The vertical height function on the surface is critical at the pinch point.	137
Figure A-1:	Two stages of the conventional object-order graphics pipeline.....	157
Figure A-2:	Architecture of Pixel-Planes 5. The GPs and renderers at the top embody the transformation and rendering stages of the graphics pipeline of figure A-1. The host workstation connects to the communication network through a special interface. The color display connects to the network via a frame buffer.....	157
Figure A-3:	Logical organization of the Pixel-Planes 5 renderer. Each renderer has an expression evaluator and a grid of pixel processors. A single pixel processor (right) has local memory and a 1-bit ALU. Two bits of the local memory are special: the enable bit and the carry bit.	158

Figure A-4:	Possible operations within the pixel processor. The source of the operation is represented on the top row. The destination is designated on the left column.....	159
Figure B-1.	Performance figures for Fourphront (triangles per second). The left column describes the attributes that triangles were rendered with. The middle and right columns give the performance for high and low resolution images. These measurements were made using a 3-rack configuration with 40 GP's and 20 renderers. The dataset had 5000 triangles, and the time includes 4D transformation and rendering.....	169

LIST OF PLATES

Plate 2.2 A	The torus ($\cos s, \sin s, \cos t, \sin t$). The extreme values of w occur along the outer and inner walls of the torus. With the light shining in the w -direction, the diffuse highlights fall along these extremes of the torus.....	48
Plate 2.2 B	A Klein bottle in 4-space with the light still shining in the w -direction. The highlights again indicate where the extreme values are located on the surface.....	48
Plate 2.3 A	The torus ($\cos s, \sin s, \cos t, \sin t$) with color modulated from black to white according to depth in w . The outer wall of the torus, near to the eye in 4-space, is black. The inner wall, far from the eye in 4-space, is light.....	53
Plate 2.3 B	The torus with opacity modulated from transparent to opaque according to depth in w . In this image, the opacity ramp is centered at $w = -0.25$ (near the eye in 4-space).....	53
Plate 2.3 C	In this image, the opacity ramp is centered at $w = -0.0$	54
Plate 2.3 D	In this image, the opacity ramp is centered at $w = 0.25$ (far from the eye in 4-space).....	54
Plate 2.4 A	The torus, translated in both the x - and y -directions. The effect of the translation is more exaggerated on the outer wall of the torus than on the inner wall. That is because the outer wall is closer to the eye in 4-space where perspective amplifies the motion. As a result, the outer wall of the torus intersects the innerwall in the projection to 3-space.	63
Plate 3.1 A	A torus sectioned into ribbons.....	66
Plate 3.1 B	A torus sectioned into opaque ribbons and semitransparent gaps in between them.	66
Plate 3.3 A	Two opaque intersecting polygons.	73
Plate 3.3 B	Two slightly-transparent intersecting polygons.	73
Plate 3.3 C	Two very transparent intersecting polygons. The polygons have faded from view somewhat, but their intersection has practically disappeared.....	74
Plate 3.3 D	The same two polygons with their intersection calculated at each pixel and highlighted.....	74
Plate 3.3 E	Torus with a full twist, rendered opaque.	75
Plate 3.3 F	Same torus, but semi-transparent.	75
Plate 3.3 G	Same torus, but very transparent. Now the innermost features are visible, but the outermost shell is merging with the background color.....	76
Plate 3.3 H	Same torus as above, but with silhouettes calculated at each pixel and then highlighted.....	76

Plate 3.3 I	Torus with a full twist, rendered opaque and clipped by the hither plane. The twist is visible inside, making a pillow shape. The performance meters show that 31 thousand triangles are being transformed and rendered per second at a frame rate of 12 Hertz. This is the performance on Pixel-Planes 5 using 12 GP's and 7 renderers.	77
Plate 3.3 J	The same view, but rendered semi-transparent. Now the performance is about 25 thousand triangles per second.	77
Plate 3.4 A	Bitmaps for the spraypaint icons. The outline of the paintcan, together with the interior and the spray, form an animated cursor. The animation sequence is 10 frames long.....	82
Plate 3.4 B	A flat, rectangular surface with different paint patterns on the front and on the back. The front has horizontal white and black stripes. The back has vertical white and black stripes. In this image, the surface is opaque.	83
Plate 3.4 C	The surface has become slightly transparent. Clockwise from the upper-right is white-on-white, white-on-black, black-on-black, and black-on-white. Notice that white-on-black is quite distinct from black-on-white.....	83
Plate 3.4 D, E	Same surface, but more transparent. Not only are the different colors blending into the background, but white-on-black and black-on-white are difficult to distinguish from each other.	84
Plate 3.5 A	An oblate spheroid has a fixed-width silhouette curve when the curve is calculated against a threshold, but only because the surface is being viewed from a special vantage point. The curvature is the same all along the silhouette.	96
Plate 3.5 B	As the spheroid rotates, the flatter part of the surface is on the bottom and the rounder part is on the top. As a result, the silhouette curve on the bottom is thicker than it is on the top.....	96
Plate 3.6 A	The surface $(\sin(s), t/5, \sin(2s)(1 + t^2)/10, 0)$. Cross sections of this surface make figure-eights in 3-space. These figure-eights are thin in the middle of the surface and fat at the near and far ends of the surface. The intersection curve they produce forms a straight line.	106
Plate 3.6 B	The intersecting patches have varying slopes along the intersection. The slopes are steep toward the endpoints of the intersection curve, but shallow in the middle. Thresholding based on depth alone produces a varying-width intersection curve (highlighted in black). Notice that the top and bottom silhouettes become intersections curves too, since the polygons along the silhouette "intersect" along their edges.....	107
Plate 3.6 C	Thresholding based on depth and slope compensates for the changing shape of the surface to produce a fixed-width intersection curve.....	107
Plate 4.1 A	The surface $(X^2 - Y^2, XY, XZ, YZ)$ in 4-space, where (X, Y, Z) is a point on the sphere S^2 in 3-space. In this projection, the surface makes a cross-cap in 3-space. A neighborhood (on the left) of the intersection has been marked with paint.....	112
Plate 4.1 B	This is the result of rotating the surface (A) in the xw-plane (in these examples, the x-axis is horizontal, the y-axis is vertical, and both the z- and w-axes point out from the image). The painted patch and the intersection have moved relative to each other in the projected image.....	112
Plate 4.1 C	This is the result of rotating (A) in the yw-plane. Again the paint and the intersection separate in 3-space.	113

Plate 4.1 D	This is the result of rotating (A) in the zw-plane. Each rotation has separated the paint and the intersection, which is the usual case when the intersection is an artifact of projection and does not exist in the actual surface in 4-space.....	113
Plate 4.1 E	The surface $(\cos(s) \sin(t/2), \sin(s) \sin(t/2), \cos(2s) \sin(t/4), -\cos(t/2))$. The intersection curve in the cross-cap is inherited from the surface as it lies in 4-space.	114
Plate 4.1 F	Rotating (E) in the xw-plane preserves the adjacency of the paint and the intersection as they are projected to 3-space.	114
Plate 4.1 G	Rotating (E) in the yw-plane still preserves the adjacency.	115
Plate 4.1 H	Rotating (E) in the zw-plane preserves the adjacency. Since the intersection persists in every orientation, it exists in 4-space as well as 3-space. Notice that this rotation has even changed the topology of the projected surface: there are two new intersection curves near the painted patch.	115
Plate 4.2 A	The surface in plate 4.2 (E) has a true intersection in 4-space. To paint across it, the user clips into the surface toward the intersection.	121
Plate 4.2 B	The clipping plane has nearly reached the intersection at the painted patch.	121
Plate 4.2 C	Clipping now exposes that the patch continues through the intersection.....	122
Plate 4.2 D	Paint is applied to the neighborhood to the right of the patch.	122
Plate 4.2 E	Transparency and ribboning show the painted region as it penetrates the rest of the surface.....	123
Plate 4.3 A	The torus $(\cos s, \sin s, \cos t, \sin t)$ projected to 3-space.....	127
Plate 4.3 B	The clipping plane crosses a critical point of the height function on the torus. The level sets show a point that blooms into an ellipse-like shape.	127
Plate 4.3 C	The clipping plane approaches another critical point.	128
Plate 4.3 D	The clipping plane has just crossed the critical point, and the cross-section curve has separated into two curves.	128
Plate 4.3 E	The clipping plane approaches another critical point.	129
Plate 4.3 F	After the clipping plane crosses the critical point, the two level curves rejoin.....	129
Plate 4.3 G	At the last critical point, the visible part of the surface vanishes from an ellipse-like shape to a single point to empty space.	130
Plate 5.4 A	Trefoil knot in 3-space spun to make a "knotted" torus in 4-space.	141
Plate 5.4 B	The surface is now transparent except for two parametric ribbons, made opaque to reveal the knot that sweeps out the surface.	141
Plate 5.4 C	An open knot in 3-space is spun to produce a knotted sphere in 4-space.....	142
Plate 5.4 D	The north and south poles of the sphere are hidden inside the left and right lobes. Transparency reveals the shape of the regions near the poles.....	142
Plate 5.4 E	The knotted sphere can be rotated in 4-space to make it look more like a surface of revolution.....	143
Plate 5.4 F	The surface is now transparent except for two parametric ribbons, made opaque to reveal the curve that sweeps out the surface.....	143
Plate 5.5 A	A constant-curvature curve in 4-space. The curve is thickened into strips to make its shape more visible.	146

Plate B.2 A	Joystick box with slider bar.....	167
Plate B.2 B	Button controls within the X-window interface.....	168
Plate B.2 C	Slider controls within the X-window interface.....	169

1 Overview

The techniques of interactive 3-dimensional computer graphics can be extended so that users (particularly mathematicians) can examine surfaces that lie in 4-dimensional space. This thesis describes a system for interactive visualization of surfaces in 4-space, the ways it can be used, and the results of letting mathematically knowledgeable users try it out. The first chapter provides an overview of the paper. The first section explains the goal of the research in terms of topology. The second section offers a brief justification for investing one's interest in the topic. The third section surveys the related work that others have done.

The combination of computer graphics and topology in this study presents me with a dilemma in choosing a target audience. Since topology is generally taught late in the college curriculum (compared with calculus), most computer scientists are likely to be unfamiliar with its terminology. Since computer graphics is a small discipline within computer science (compared with operating systems or compilers), most mathematicians are likely to be unfamiliar with its algorithms. There are other textbooks that cover the background material in these subjects, so I will not duplicate that effort here. Instead I am choosing to target the reader who is conversant with topology and differential geometry and is fully fluent in computer graphics. Appendix A contains a description of Pixel-Planes 5 [Fuchs89], the machine that provided the graphics environment for this work. Some of the algorithms are peculiar to its architecture, and that architecture is not necessarily familiar to the computer graphicists outside of this university.

A warning is in order. A document such as this one simply cannot communicate the real-time 3-dimensional experience of interacting with a surface in 4-space. That job is left to "Fourphront," the system I developed to run on Pixel-Planes 5.

1.1 Goal of the Research

An important and current problem in topology is how to classify the different compact manifolds. Meanwhile, today's high-performance graphics machines permit a user to interact with surfaces. Can computer graphics be put to use in the service of topology? Indeed it can.

There exists a progression of curve, surface, and so on, which are known to mathematicians as *manifolds*. At every point of a k -manifold, there is a neighborhood that is topologically equivalent (*homeomorphic*) to a neighborhood in Euclidean k -space. Examples of 1-dimensional and 2-dimensional manifolds can be imbedded in the 3-dimensional world, and so naturally they are relatively easy to picture in one's mind: circles, knots (examples of 1-dimensional manifolds) and spheres (an example of a 2-dimensional manifold). Large-dimensional manifolds are not so easy to imagine, yet they can be defined mathematically.

What sorts of compact k -manifolds are out there? The answer is known for $k < 3$. But topologists have not yet classified all the compact 3-dimensional manifolds. That means that it is not known how to compare two of them and then decide whether they are topologically the same. How remarkable! Imagine that you were presented with a torus and a sphere, but couldn't tell that they were different. Visually, their difference is immediately apparent. Why then is it so hard for topologists to tell which 3-manifolds are which? It is "obvious" that distinguishing between different kinds of manifolds is as easy as taking a good close look at them, isn't it? Granted, this example is absurdly contrived – a sphere and a torus have different genus, which instantly discloses their difference. But the example reveals the pedagogical challenge of presenting this unsolved classification problem in an intuitive way. It would be satisfying to use computer graphics to examine manifolds of bigger dimension and see for oneself why their differences are subtle. That long-range goal is still far off. Even the fastest machines cannot yet generate real-time volume-rendered images of a large data set like a 3-manifold. But a graphics engine like Pixel-Planes 5 is sufficiently powerful to let a user manipulate models of 2-dimensional manifolds.

Now reconsider the "obvious" claim that one can identify a 2-manifold by visually examining it. How does one actually conduct the examination? What does one look for? There is a crucial detail that makes this task especially difficult: the non-orientable 2-manifolds cannot be imbedded in ordinary 3-space. They need four dimensions. So how can a person use interactive computer graphics to examine a surface in 4-space and decide just exactly what that surface is?

I claim that interactive computer graphics lets a user fathom the nature of 4-space by letting him manipulate a surface in 4-space. By using a software system called "Fourphront" I support this thesis in two ways. First, I demonstrate the above "obvious" claim (for compact 2-manifolds) by applying simple interactive tools to a surface in 4-space in order to discover its topology. It is expressly *not* the goal to develop an "expert system" that takes as input some description of a surface and produces as output the classification of the

surface. Instead of using the computer as an off-line number-cruncher devoted to computational topology, I want the machine to support a person as he exercises his own algorithm, so that he can develop a feel for 4-space as he goes along. Secondly, I have let knowledgeable users engage in interactive sessions in order to observe their insights and conjectures. Their discoveries make it clear that the users are in fact gaining familiarity with and intuition for the surfaces they manipulate.

Building such a system first requires solving several collateral problems. Solving them has become an auxiliary goal in this research. These problems can be loosely grouped into four major categories, which are listed below.

- (1) How can a user within 3-space control an object that is in 4-space?
- (2) How can a surface in 4-space be illuminated to give it a realistic, shaded appearance?
- (3) How can a user recover the shape of a surface when its image is displayed on a 2-dimensional screen?
- (4) How can a surface in 4-space be projected to a 2-dimensional screen?

In this paper I will provide answers for each of these basic questions. These answers provide significant and helpful techniques even to the general area of 3D graphics, generating an unexpected benefit of the research.

1.2 Why Surfaces in 4-space are Interesting

In the study of surfaces in 3-space, interactive computer graphics can make a *quantitative* difference. A computer can be used to model and render surfaces faster and cheaper than they could be constructed in the physical world. Examples include (1) architectural walkthroughs for building designers and (2) radiation-treatment planning for physicians who have acquired 3D data from their patients.

In the study of surfaces in 4-space, interactive computer graphics can make a *qualitative* difference. A computer can be used to model and render surfaces that simply cannot be constructed in the physical world. This sort of “virtual world” within the computer makes it possible to interact with surfaces in 4-space in a way that is simply unavailable in 3-dimensional space.

This section is intended to motivate interest in the research. Unlike many areas in computer science, the subject of 4D graphics continually evokes the same question: what is the point

of doing it? After all, the “fourth” dimension doesn't really exist, does it? And even if it makes sense to talk about more than three dimensions, isn't 4-space just an abstraction for mathematicians? The following sections address that question in three parts. First I discuss manifolds and large dimensions in a general way. Then I focus on 4-space in particular. Finally I discuss the importance of surfaces (that is, manifolds of dimension 2).

1.2.1 About Large Dimensions

Does it even make sense to ponder large dimensional spaces, when our own experience is confined to three spatial dimensions? Certainly there have been celebrated thinkers who felt that it was. The study of large-dimensional spaces is ancient, with beginnings as early as Euclid, who considered the progression from point to line to plane and beyond. Philosophers and mathematicians debated whether these “multiply-extended” quantities were meaningful, but the nature of large dimensions remained hidden until after the development of calculus and group theory.

Mathematicians of the nineteenth and twentieth centuries investigated manifolds in two ways. For manifolds of various dimension, they sought (1) to understand the calculus on the manifold, and (2) to understand the topology of the manifold. By providing a set of maps from Euclidean space onto a surface, differential geometers measure properties of the surface in terms of properties of the maps. That is the province of calculus. By computing algebraic quantities on surfaces, algebraic topologists can (sometimes) determine whether two manifolds are homeomorphic or not. That is the province of group theory. These techniques are very abstract, encouraging the mathematician to direct his attention at the mathematical machinery that applies to the manifold. This is just as well: since so many manifolds can't be realized in three dimensions, the mathematician must either imagine larger dimensions or look to the machinery for insight.

A graduate sequence of mathematics courses typically includes studies in group theory, differential geometry, topology, and the calculus of manifolds. This is the abstract machinery that applies to surfaces. But the segment of the population with graduate degrees in mathematics is quite small. If it requires such abstract tools to comprehend large dimensions, very few people can participate in the discussion. It is no surprise then that conversations about n -space, even among practitioners of the “hard” sciences, often degenerate into mystical utterings about higher-dimensional beings and about the nature of time. This is a pity, since mathematics has laid bare so much of the structure of large-dimensional objects and spaces.

1.2.2 About 4-space

Why study surfaces in only 4-dimensional space? Why not tackle surfaces in arbitrary dimensions? For one thing, four dimensions are sufficient to address the important concern: can a person mentally model the shapes of an object situated in a large-dimensional space, when only the 2-dimensional image of it is available? Secondly, every additional dimension requires more degrees of freedom from the physical input devices. It is not my goal within this study to invent general techniques for mapping input into arbitrary dimensions.

There are several properties and problems in 4-space that do not admit a reformulation in three dimensions, and there are “natural” ways that 4-dimensional spaces crop up. These examples come from topology, differential geometry, and physics.

Topology

This is a more detailed discussion of the basic problem described in the section above, “Goal of the Research.” The italicized technical terms are defined in the appendix.

Topology is concerned with sets and the neighborhoods within them. For the student in topology, one of the first examples of a topological set is Euclidean space. Euclidean space has the nice property that it is locally Euclidean: every neighborhood (open ball) “looks like” every other neighborhood. That is, these neighborhoods are homeomorphic with each other. But Euclidean space is infinite, and hence hard to illustrate concretely. So a topologist then wonders: Are there compact (bounded and closed) spaces with that same nice property of being locally like Euclidean space? If so, what are they? These sets are the compact manifolds, and classifying them all is the impetus for a significant portion of topology.

A first course in topology develops the notion of *fundamental group* of a topological space. The classification theorem for surfaces states that a pair of compact surfaces share the same fundamental group exactly when they are homeomorphic, so these groups *classify* the topology of 2-manifolds. Topologists have yet to classify all the 3-dimensional compact manifolds, but have succeeded with the 2-dimensional manifolds (that is, surfaces). The result is that the compact surfaces are completely determined by their genus and their orientability. Intuitively, the genus of a surface is the number of holes it contains, and a surface is orientable if it has two different sides. The Möbius band is a familiar example of a surface (with boundary) that is non-orientable, having only one side. It turns out that non-orientable surfaces (without boundary) require four dimensions in which to imbed. Moreover, none of the compact 3-dimensional surfaces can imbed in three dimensional Euclidean space. It might be enlightening to examine and compare manifolds that are

topologically different and that inhabit four dimensions of space. Do they look alike or not? Computing speed has reached a level of performance that permits one to make the comparison interactively. For surfaces of modest complexity (tessellated by fewer than 10,000 triangles), one can interactively manipulate them (on contemporary high-performance hardware) to reveal their topology and their geometry.

Another reason that surfaces in 4-space are interesting is that they can be knotted. Just as a circle can be knotted in 3-space, a sphere can be knotted in 4-space. Moreover, a sphere can possess the property of knottedness only in 4-space; 3-space is too small, and 5-space is too large. For the uninitiated, it is difficult to picture what a knotted sphere even looks like. Although the subject of knots goes beyond this study, it is a clear candidate for future research using a system like Fourphront, with which one can turn the surface around in its native space and look at it from any direction.

Differential geometry

Topologists have known that Euclidean n -space admits only one *differentiable structure*, the natural one, so long as $n \neq 4$. Michael Freedman proved in 1982 that 4-space is remarkable in the following way. There are infinitely many differentiable structures on Euclidean 4-space that are different from the natural one [Freedman82].

There is a very simple way that surfaces in 4-space arise. When a student first learns calculus, he graphs a function $f(x)$ that forms a curve in the plane, then proceeds to sketch a tangent at a point on the curve. He may never consider that the tangent properly belongs in a separate tangent space (the curve and tangents forming a surface in 4-space). The notion of a *tangent bundle* lies dormant until he takes a graduate-level class, where it immediately presents itself in four (or more) dimensions.

Physics

Physics in the twentieth century has occupied itself with the theories of relativity and quantum mechanics. The former concerns the behavior of 4-dimensional space-time, and the latter concerns the behavior of complex-valued probabilities. The simplest complex-valued functions map one complex number to another, yielding surfaces in four dimensions.

These examples are meant only to indicate that an understanding of surfaces in 4-space has some general value, both theoretically and practically. It is not my aim to solve the classification problem in topology, to exhibit nontrivial differentiable structures on 4-space, or to explain modern physics through a computer graphics system. Rather, I will only

demonstrate that interacting with surfaces in 4-space endows them with a concreteness that gives one a sense of their shape, and especially the topological character of that shape.

1.2.3 About Surfaces

Why study only surfaces? Why not study 1-dimensional and 3-dimensional manifolds as well? In both cases, the answer is tied to computer graphics and to topology. From the standpoint of computer graphics, 3D volumes are too computationally-expensive to render interactively, and 1D curves are too thin. Volume-rendering is still evolving as a new discipline within computer graphics, and hardware technology is still advancing; this may make it attractive within a few years to interact with 3-manifolds. Curves are often rendered as thickened, tubular surfaces in order to make them more visible as well as to exploit algorithms for illuminating 2-dimensional surfaces. The surface is certainly the dominant rendering primitive for computer graphics today.

From the standpoint of topology, compact 1-dimensional curves in 4-space are not as interesting as surfaces are. They only make trivial knots in dimensions greater than three. They only come in one topological type: the circle. On the other hand, volumes (in the form of compact 3-manifolds) are too confined: they need up to 6-dimensional space to be imbedded. The population of 3-manifolds that can be imbedded in 4-space is small.

1.3 Related Work

This section surveys the work that other researchers have done in areas that relate to this thesis: namely, interaction with surfaces in 4-space. Researchers have approached the problem in several different ways according to the audience they target.

I group this work into the three domains 1) computer graphics, 2) mathematics, and 3) pedagogy. Some of the work straddles the divisions, but much of it fits clearly into a single category. I have found no related research that shows how to apply interactive computer graphics to the problem of determining what the topology of a surface is, and this dissertation is intended to fill that gap.

1.3.1 Computer Graphics

The bulk of the related research lies within computer graphics and falls into the following areas: 1) interacting with a larger dimension; 2) illuminating objects in 4-space; 3) viewing objects in 4-space; 4) reducing dimension by projection or intersection; 5) devising depth cues in 4-space; and 6) inventing variant depictions of n -space. Several researchers have

extended conventional 3D-graphics techniques into larger dimensions. A few have invented unconventional solutions. Here is a survey of these works. I shall observe the convention of using the coordinates (x, y, z, w) to refer to points in 4-dimensional space. For screen coordinates, x and y refer to the horizontal and vertical directions respectively, and z corresponds to the depth. Vector quantities appear in boldface, and scalar variables are italicized.

Interacting with a larger dimension

The mouse (or 2-dimensional pointing device) has become fairly ubiquitous as an attachment to personal computers and workstations. As a result, there have been several efforts to use it to control 3D motion within applications that use 3D computer graphics. Michael Chen [Chen88] devised a “virtual sphere” metaphor as a way to link a mouse's motion with rotations in 3-space. In his system, an imaginary sphere is overlaid onto an image of a 3D scene. When the mouse moves over the scene, it makes that sphere rotate to track the mouse; the entire 3D world consequently rotates to match the new orientations of the sphere.

I have found no systematic descriptions of how to map physical input onto motion in spaces of dimension four or greater. This dissertation presents a solution.

Illuminating objects in 4-space

Conventional 3D computer graphics is primarily concerned with how to render (and therefore to illuminate) surfaces. It is secondarily concerned with how to render points, curves, and volumes. There is a good reason for this focus: what a person ordinarily sees from day to day is a collection of surfaces in the world. These surfaces are the boundaries of 3-dimensional solids. A few notable exceptions exist, such as clouds, haze, or smoke, all of which are semitransparent “solids.”

In 3-space, illumination is a description of how light interacts with an object, and that description is based on physics. Without a physical model to appeal to, researchers are left to extrapolate for themselves in order to define an appropriate model for the interaction of light with points, curves, surfaces, and volumes in 4-space. This problem has been fairly ignored, as the examples below indicate.

Points and Curves

Some researchers have used scatter plots of statistical data in 4-space as a way of seeing clusters or trends. Most work in this area is relatively unconcerned with geometry or realism; the goal is generally to find out how data is distributed at a coarse resolution. As a

result there has been no discussion of how to illuminate points in 4-space, with a single exception described below (under “Surfaces”).

There have been several papers that use 1-dimensional curves (whether smooth or piecewise linear) to outline objects that are in 4-space. Burton [Burton82, Burton89], Koçak [Koçak86], and Liu [Liu84] used curves in 4-space, but they rendered these curves using flat shading and only an ambient model of illumination. Christoff Hoffman [Hoffman90] described how to calculate silhouette curves on surfaces in 4-space by analytically computing them for the surface against the eye vector, and he displayed the resulting curves in the same manner: flat shaded with ambient lighting.

Surfaces

Huseyin Kocak [Koçak86], Thomas Banchoff [Banchoff86], and Christoff Hoffman all take the same approach to illuminating surfaces in 4-space. Namely, they postpone illumination until the surface is projected into 3-space. The projected surface then admits the conventional techniques of illumination that are familiar in 3D graphics. This strategy is at least as old as 1880, when it was used to shade polygonal faces as though they were illuminated in 3-space [Stringham]. Postponing illumination is a natural way to apply the techniques of 3D computer graphics to the problem of visualizing surfaces in 4-space. But it does not address the problem of how to visualize the surfaces as they would appear if illuminated in the original 4-dimensional space where they reside.

Andrew Hanson [Hanson91, Hanson92] presented a novel solution for illuminating manifolds (of any dimension) that sit in 4-space. His solution was to promote the dimension of the manifold (if necessary) to make a new 3-manifold. He then volume-rendered the 3-manifold, illuminating it in a nearly-conventional way. The way he promoted a k -manifold (where $k < 3$) was to attach a $(3-k)$ -sphere to it at every point. That is, he mapped the k -manifold M to the 3-manifold $M \times S^{3-k}$. The way he illuminated it was to compute normal-vectors on the 3-manifold, and calculate diffuse and specular illumination against these vectors. Those equations (in vector form) are identical to the usual ones for surfaces in 3-space, as several researchers have pointed out before (see the next section, “Volumes”). (This technique has a familiar analog in 3-space, namely, replacing a curve with a cylindrical tube that surrounds it and then rendering the tube instead.) Equipped with an illumination-value at every point, he finally volume-rendered the resulting 3-manifold.

There are obvious drawbacks: the technique is slow, and the data-size of the promoted manifold is larger than that of the original manifold. But there are two other drawbacks that are somewhat more subtle. First, the promoted manifold must have dimension one-less than

the dimension of the space (they have codimension 1). That means that the storage cost increases exponentially with the dimension of the space. Second, the resulting image depicts the promoted manifold instead of the original manifold. In my view it trades an unsolved problem (how to illuminate a surface in 4-space) for a solved problem (how to illuminate volumes in 4-space). This distinction is not meant to criticize his technique but rather to indicate where the problem area still lies.

In short, illuminating a surface in 4-space remains an unsolved (and largely unaddressed) problem. This dissertation presents a solution.

3-D Volumes

Most of the work on the subject of illumination in 4-space has been concentrated on 3-dimensional objects, since they have codimension one. As long as the object has codimension one, the usual techniques for illumination apply. A point on the manifold has a normal vector (the unit normal is unique up to sign), and a unique reflection vector for a point-light source. The angle between the surface normal and the light vector controls the amount of diffuse reflection. The angle between the reflection vector and the eye controls the amount of specular reflection. Scott Carey [Carey87] and Victor Steiner [Steiner87] illuminated the boundary of a hypercube in this straightforward way, since the hypercube has an available “outward” normal at each point. They simply let the boundary retain the normal vectors it possessed as a portion of the hypercube volume.

Burton [Burton89] described a method for shading volume elements in 4-space by summing the object's color through the entire thickness of the object at each pixel, sometimes called the x-ray method. His illumination model corresponds to the ambient lighting in 3-space. Carey [Carey87] shaded his hypercube two different ways: across surfaces and across volumes. A cube has square faces, with each face bounded by segments; a hypercube has ordinary cubes as faces, with each cube bounded by squares. Cary distinguished between rendering the squares only (“surface shading”) or volume-rendering the illuminated cubes (“solid shading”).

Chandrajit Bajaj [Bajaj90] described how to mesh and display hypersurfaces (3-manifolds). He restricted the domain to include those hypersurfaces implicitly defined by a polynomial of degree 2 or 3. The advantage to using a low-degree polynomial is that an explicit rational parametrization is easily available. The advantage to using an implicit function is that the normals are easily available. After tessellating the 3-manifold, he rendered 3D slices from the 4-dimensional space. Each slice consisted of a 2-manifold in 3-space, so he then applied conventional illumination to the resulting surface.

Mei-chi Liu [Liu84] discussed the problem of calculating shadows in 4-space. Liu showed how a 4-dimensional solid, bounded by 3-dimensional polyhedra, casts a shadow onto an image volume. The reason for the "image volume" instead of an "image plane" is that the technique only applies when the object-boundary and the projection-target both have codimension one. Shadows do not generally occur (that is, their measure is typically zero) when the objects casting them have codimension two or more. Consider curves in 3-space: they only shadow each other at isolated points, except in the singular case where they coincide (from the point of view of the light). If the projection-target has dimension less than k , the shadow cast upon it by a k -manifold does not reveal very much about the manifold's shape. It can't, of course, because it's dimension is simply too small. For example, a sphere in 3-space may cast a shadow onto a string; the shadow is merely a segment. In order to capture the countour of the sphere, the projection-target must be at least two dimensional.

These various efforts raise a single important point. Illumination of a manifold is fairly straightforward if the manifold has co-dimension 1. If the co-dimension is larger, illumination is not well-defined.

Viewing objects in 4-space

Several researchers have explored ways to view objects in 4-space. I mean this in an inclusive sense of either viewing in 4-space or viewing within a 3D object in 4-space. In regard to the first sense, specifying the view of an object dictates the placement and orientation of both the eye and the object. In regard to the second sense, there are 3-manifolds that exist in 4-space but which can be viewed independently of 4-space, for example by constraining the view to lie within the manifold. This work is detailed below.

View specification in 4-space

Christoff Hoffmann [Hoffman90] described how to locate and orient two centers of projection. These two centers correspond to two eye-positions. The first one helps define a view in 4-space that projects onto a 3-dimensional subspace. The second helps define a view in that 3-space that then projects to the 2-dimensional space of the screen. He used Euler angles to specify the orientations of the two eyes, and a pair of radii to establish the eyes' distances from the origin. Specifying the viewing parameters and specifying the object's position/orientation are complementary problems. Hoffman used the same approach (Euler angles) for both, with individual dials controlling each Euler angle. The dials are not meant to be suggestive (in any spatial way) of the angles they govern: the assignment of dials to Euler angles is somewhat arbitrary.

Daniel Asimov [Asimov85] devised a “grand tour” as a way to navigate large-dimensional space. The grand tour is a curve in 4-space that specifies the path for an eye position to follow. A single parameter can then determine where along the path the eye is currently located. If the path is a space-filling curve, then that single parameter can specify any location for the eye to be placed. Asimov used time as the parameter, and produced animations of a flight through 4-space along this virtual camera path.

An unsolved problem remains: how to specify the view naturally. How can a user exercise direct control over both centers of projection or maneuver objects within the 4-dimensional world? The viewpoint and the objects possess more degrees of freedom than the user does, since the space they occupy has a larger dimension. This dissertation presents a solution to this problem.

3-Manifolds in 4-space

There are two different approaches one can take in order to view a 3-manifold. One approach is to realize the manifold in 4-space and project the image onto a screen. This is the method that most researchers have employed. The other approach is to imbed the viewpoint inside the 3-manifold and project the manifold's contents onto the screen. For example, the 3-manifold might be Euclidean 3-space imbedded in 4-space. In that case conventional graphics techniques apply: just restrict the domain to the 3-dimensional subspace of interest, and render whatever is inside it. This technique is often applied by taking “consecutive” 3D slices of 4-space, and rendering their contents. But other 3-manifolds exist besides flat 3D subspaces, and they can be either negatively or positively curved. How does one render their contents?

Charles Gunn and Mark Phillips [Gunn91, Phillips92] described a way of viewing a 3-manifold of constant negative curvature from the inside (as opposed to projecting the manifold into a 4-dimensional space that contains it). They manipulated the transformation matrix in order to preserve the apparent geometry of these hyperbolic spaces. This proved to be very intuitive: a user can virtually fly and turn in any direction through the space, and his motion through the space seems locally to match his motion in the real, physical 3D world.

Although this thesis was motivated by the problem of visualizing 3-manifolds, I have restricted my effort to interacting with 2-manifolds. However I do discuss some of my preliminary investigation of immersing the user within a 3-manifold, using a head-mounted display together with a 6-degree-of-freedom input device. That discussion is below, in section 1.3.3: “Pedagogy.”

Reducing dimension by projection or intersection

One problem with depicting objects in 4-space is their eventual representation on a 2-dimensional display. How does one shed two of the dimensions? There are two basic approaches. One answer is to project the original space and its contents. Under this mapping a compact k -manifold in 4-space still looks like a k -manifold when it is displayed on a computer screen (except in the unusual case where a flat manifold is parallel to the direction of the projection and hence loses a dimension, or except for a generally-small set of singular points). The other answer is to intersect 4-space with a 3D subspace (particularly a 3D Euclidean subspace). Under this mapping the dimension of the space and the manifold are both decreased (except in the unusual case where part of the manifold is flat, and lies entirely within the 3D subspace). Researchers have taken both approaches, as described below.

Projection to 2-space

Koçak and Banchoff [Koçak86] described a technique of projecting multiple views onto an octagon, divided into six rhombus-shaped regions. Each rhombus showed the image that was projected to one of the six (orthogonal) axial planes centered at the origin. In effect, he extended into 4-space the architect's use of multiple views in a blueprint. He used these projections to show the shape of a trajectory that obeys certain differential equations. By rotating the view (projected onto the octagon), he established that the trajectory wrapped around a torus.

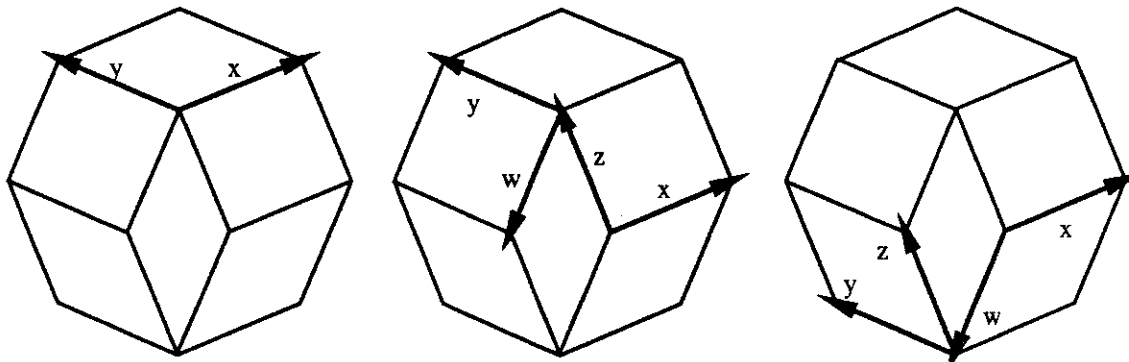


Figure 1-1: The six faces on this octagonal figure correspond to the six axial planes of four-dimensional space.

Banchoff [Banchoff86] used spherical (or stereographic) projection to illustrate properties of the torus in 4-space. (This mapping is not to be confused with image pairs for binocular, stereoscopic vision.) Three-dimensional spherical projection sends points from a sphere into a plane: a line from the north pole connects a point on the 2-sphere (in 3-space) with its image in the plane through the equator. In 4-space, a line from the north pole connects a

point on the 3-sphere with its image in flat Euclidean 3-space through the equator. Banchoff animated a sequence of toruses that foliate the 3-sphere. He first projected them spherically into 3-space, and then projected the result orthographically to the screen. The reason he favored spherical projection for the first step was simple: under orthographic projection the images of the toruses always intersected, even though they do not intersect in 4-space. Under spherical projection, the tori linked without intersecting.

Victor Klymenko [Klymenko90] argued that perspective projection makes rotating objects seem less rigid than does orthographic projection. He employed four different kinds of projections from 4-space to 2-space, based on the choices for the two intervening projections. Each choice could be either ortho(graphic) or persp(ective). Thus there are the following four combinations.

1 (ortho, ortho); 2 (persp, ortho); 3 (ortho, persp); 4 (persp, persp)

Klymenko asked 29 subjects to gauge how rigid a hypercube looked when it was rotated in 4-space. Their collective rankings, which he found to be statistically significant, were ordered as indicated by pairs above. They reported that the object appeared most rigid under doubly-orthographic projection, and least rigid under doubly-perspective projection. When the projections were of mixed type, the subjects gave a slight preference to rigidity when perspective was applied first.

Andrew Glassner [Glassner91] used the surface of a 2-torus as the projection-target, as opposed to the conventional plane of projection. He reasoned that even though the display of an ordinary color monitor is flat, the surface of projection need not be. His idea can be extended to 4-space by choosing an “unusual” subspace for projection (say, a 3-torus) instead of choosing a flat, Euclidean 3D subspace, and projecting along lines of sight that are perpendicular to the subspace.

Intersection with smaller spaces

Christoff Hoffman [Hoffman90] observed that conventional clipping can be applied against the w -coordinates of a surface. He illustrated the technique by clipping a surface against the boundary defined by $w=\text{constant}$. The result is the intersection of the object with the half-space $w \leq \text{constant}$.

A variant of this scheme is so well-known that, for many people, it is the initial suggestion for visualizing objects in 4-space. The suggestion is to use space-time as the four-dimensional system. If the parameter t is treated as the time coordinate, the intersection of the objects with the 3-dimensional subspace defined by $t = \text{constant}$ yields a snapshot of

time-varying data at a given instant. Letting t vary from frame to frame produces a movie of the time-varying data. This approach has a serious drawback if the “time” coordinate is artificially imposed on data without a natural temporal dimension. Without a distinguished “time” axis, any arbitrary direction may be assigned to it in a purely ad hoc manner. Rotation in 4-space creates a further drawback. As an object rotates rigidly in 4-space, its intersection with a 3D subspace may vary in confusing ways. For example, a connected surface in 4-space can intersect with that 3D subspace to produce curves that disconnect and even disappear during the rotation. The surface may exhibit the same behavior by translating through the 3D subspace, but in my experience it is much harder to reconstruct the shape of a surface when it rotates than when it translates through an intersection volume.

The lower-dimension analogy of temporal-slicing is easier to grasp. Consider some time-varying data in two dimensions, like a stack of pages with animated figures drawn on them. The temporal axis runs in and out of the pages, perpendicular to their surface. As the pages are flipped, the cartoon figures jump up and down, perhaps. If the sheaf of pages is turned sideways, the viewing axis no longer aligns with the temporal axis. If the sheaf could somehow be glued together and re-sliced, you could flip through the cartoon from left to right instead of top to bottom. The result, of course, would no longer resemble the original animated figure.

Devising depth cues in 4-space

Armstrong and Burton [Armstrong85] made wire-frame stereo pairs of (x, y, z, w) -space, using stereo in the z and in the w directions. Each pair was rendered from a particular view-specification. The view specification for the left and right image varied in z , in w , or in both coordinates. I was not able to fuse the “four-dimensional stereo” images into meaningful shapes when w varied: the images just looked like two different objects.

In their investigation Armstrong and Burton also tried using a technique they called “isotropic” perspective. They applied perspective from the w -direction onto the x -axis and from the z -direction onto the y -axis individually. They did not base this technique on any physical model; they simply noted that it might serve as a visualization tool. Their objective was to provide enough depth cues to let the viewer disambiguate distances in the z -direction and the w -direction. They also investigated hidden-line removal based on depth in z and in w . Since they rendered the 3-dimensional hypercube only as wire-frame images, the effect was not convincing.

Burton [Burton82] made a more thorough investigation of hidden-line removal and clipping for wire-frame figures in 4-space. Again the technique yields unconvincing results. To

illustrate the technique, he removed hidden lines from a complicated figure after projecting it to (1) 3-space, (2) 2-space, or (3) both. The image that resulted from (1) is too complicated to comprehend, and the images resulting from (2) and (3) are virtually indistinguishable from each other.

Victor Steiner [Steiner87] rendered semitransparent polygons in 4-space to represent the 3-dimensional polyhedra that they tessellate. It is a common practice in 3D graphics to cull back-facing polygons; Steiner applied the analogous procedure to cull backfacing polyhedra in 4-space, where the polyhedra bound some 4-dimensional object. He also ordered the polyhedra to determine their visibility and render them from back to front (the painter's algorithm). By clipping and culling, he produced an image of several solids that appear to interpenetrate when projected to 3-space. The most important drawback to his rendering technique is that it combines two incompatible qualities: occlusion and transparency. Transparent surfaces do not hide each other, so it does not make sense to clip away "hidden" pieces of them.

An unsolved problem remains. What is an appropriate depth cue in four dimensions? This dissertation provides a simple and effective technique.

Inventing variant depictions of n-space

Some researchers have abandoned a geometric approach altogether, instead using novel methods for representing data in four (or more) dimensions. Donald Curtis [Curtis87] used parallel axes to represent coordinates in 4-space. In this scheme, each coordinate of a point in n-space is plotted on one of n parallel lines. These coordinates are then connected consecutively, transverse to those lines. He used this technique to graph several discrete points. Since the resulting graph represents a single point as a 1-dimensional piecewise linear curve, it is not an effective representation for surfaces: they would become 3-dimensional objects in the graph.

Elizabeth Cluff compiled various methods for depicting n-dimensional data. One example of the "unusual" representations in her catalog is the "Chernoff" face. Each parameter is mapped to a feature on a cartoon. The first parameter might turn the lips from a frown to a smile. The second parameter might make the ears grow large or small. The idea is that the overall mood that the face expresses gives the viewer a sense of where the collection of parameters lie in this abstract space. Whether or not this is the case, this scheme has the drawback that an entire face is required to plot a single data-point.

1.3.2 Mathematics

In the previous section, I described other work within the computer-graphics community to create images of objects in 4-space. What work is there within the mathematics community to apply computers (especially computer graphics) to solving topological problems?

There are many mathematical papers that employ illustrations to motivate the problems and their proofs, but these illustrations are usually hand-drawn. Modern commercial packages for modelling and rendering (Mathematica, for example) make it more convenient for mathematicians to use computer graphics to communicate their work. But these packages are generally designed for representing surfaces in 3-space. It is not surprising then that only a small amount of work on the mathematics side uses computer graphics in a substantial way for studying topology in general, or, in particular, surfaces in 4-space.

On the other hand, certain approaches in topology lend themselves naturally to interactive computer graphics. As an example, consider Morse theory, which applies calculus on a surface in order to determine the topology of the surface. The usual textbook introduction to Morse theory begins with a sequence of drawings that illustrate the intersections between a surface (for example, a torus) and a plane that sweeps through it. The moving plane represents level surfaces of a height-function. It is the set of critical values of this function along the surface that disclose the surface's topology. Morse theory provides a natural junction between a computational algorithm (enumerating critical points) and applied topology (classifying a given surface).

Morse theory invites the use of interactive computer graphics to let a user sweep a plane through a surface. There is, however, a small set of work that uses (non-interactive) computational algorithms (in the form of computer programs) in topology. One area of research, which I call computational algebraic topology, is concerned with calculating algebraic structures – homotopies, homologies, and cohomologies. The other area of research applies symbolic algebra-solvers to topology. Neither of these areas makes any explicit use of computer graphics, although both of them invite its use to illustrate individual surfaces under investigation.

Computational Algebraic Topology

The use of computational methods in topology has a long history. In the 1930's, Herbert Seifert [Seifert34] described how to calculate the Betti numbers for a simplicial complex. A 1-simplex is an interval, a 2-simplex is a triangle, a 3-simplex is a tetrahedron, and so forth. A simplicial complex is a collection of simplices. The Betti numbers are topological

invariants. Thus it is necessary (although not generally sufficient) that a pair of simplicial complexes have the same Betti numbers in order for them to be homeomorphic.

Bruce Donald [Donald91] presented a method for calculating the homology group of a triangulated mesh on a surface. A homology group of a triangulation is an abelian group associated with the connectivity of the vertices to each other. Larry Lambe [Lambe] described algorithms for computing cohomologies of certain groups. A cohomology group can be associated with the differential forms defined on a manifold. Both the homology group and the cohomology group are topological invariants of a manifold.

David Anick [Anick] showed that computing homotopy groups can be NP-hard. A homotopy group is an algebraic group defined on a manifold by the behavior of n -spheres as the spheres deform. The homotopy group is another topological invariant, which is a reason one might want to calculate it. Douglas Ravenel [Ravenel] showed how to compute homotopy groups of spheres on a small computer.

Mechanical Theorem-proving in Topology

Working mathematicians have used computational tools to produce real theorems. Donald Davis [Davis] used a software system to help prove a theorem about immersion of the projective plane. The system helped him solve complicated algebraic equations in the intermediate steps.

The two areas I described above indicate that computational algorithms are penetrating even into the abstract area of topology. What distinguishes Fourphront is that it encourages the development of “user algorithms” to study surfaces. This strategy is made possible by modern computer graphics hardware, of which Pixel-Planes 5 is an excellent example, and by the rich development of the mathematical theory of surfaces.

1.3.3 Pedagogy

There is a third active area that involves both computer graphics and mathematics. That area is pedagogy. Several people have undertaken to explain the nature of large-dimensional spaces, or of the objects that inhabit them.

Edwin Abbot's *Flatland* is probably the most famous effort to impart an intuitive sense of geometry in 4-space. Abbott describes a two-dimensional world populated by simple geometric creatures: segments, triangles, squares, circles. One of these creatures is visited by a 2-sphere, then finds himself lifted out of the plane and “into” our three-dimensional world. With this enlightening experience, he suggests to the sphere that there might exist a

fourth dimension, in which direction they could travel. The sphere, as a citizen of 3-space, scoffs at this ridiculous suggestion. The lesson of the allegory is clear. We, the citizens of a three-dimensional realm, have as much trouble visualizing 4-space as the denizens of flatland have of visualizing 3-space. But it might be nonetheless “out there.”

There is quite a collection of “phenomena” that are popular within the folklore of 4-space. Paul Isaacson [Isaacson84] computed a sequence of wire-frame figures in order to illustrate these various phenomena by projecting 4-space objects onto a surface (Note that the captions of his figures 2 and 3 are reversed.). He unfolded a hypercube into its constituent cubes. He intersected it with a 3-dimensional subspace to show how 3D “flatlanders” would perceive its traversal through the 3D universe. He rotated a left-handed object into its right-handed counterpart, illustrating the well-worn cliché that “rotations in 4-space can turn objects inside-out.”

Thomas Banchoff has made animated films of the hypercube and the 2-torus in 4-space. The films relate geometry in four dimensions to the more-familiar geometry of objects in three dimensions. One of the films shows the shapes of cross-sections of the hypercube (like the figures that Isaacson, above, created). Another film shows how the 3-sphere looks as it rotates in 4-space. He foliated the 3-sphere with a collection of 2-tori (the Hopf toruses) and used stereographic projection to map the tori into Euclidean 3-space.

Jeffrey Weeks [Weeks] described how to visualize a torus from the inside of it. He used the metaphor of a room whose ceiling, floor, and four walls have a special property. Each of them acts as a “window” through which the observer can view the room as though he were looking through the opposite surface. An observer looking up at the ceiling sees the bottom of his feet; looking down at the floor he sees the top of his head.

As an exercise I implemented this arrangement on Pixel-Planes 4 [Fuchs85] using a head-mounted display and a 6-degree-of-freedom spatial input device manufactured by Polhemus. I modified code that Warren Robinett had written for an interactive adventure game. This game let the user travel through “portals” from one room to another. A portal was essentially a transformation matrix applied to the geometry of a given room. Weeks described what it would be like to create, within a virtual world, spatial relationships that could not exist in the physical world. For example, the user might cross through a portal from room A to room B, but crossing back through that portal puts him in room C (not back in A again). I modeled a striped, rectangular cage and a ball that can be put into the cage by using the Polhemus tracker to grab and reposition it. The cage and ball were defined within a 6-walled cubical room, for a total of about 30 polygons. I made each wall a

portal that led through the opposite wall; this defined a 3-torus. Displaying a scene consisted of first rendering the contents visible within the room (layer 0). Next, each visible portal of layer 0 was displayed by rendering the geometry associated with that portal (that geometry being the room again). This was layer 1. Likewise each visible portal of layer 1 was displayed, and so on recursively. The number of polygons increases exponentially with the number of layers, by a factor of about $(2n+1)^3$, where n is the number of layers. With only three layers the system had to transform and render about 52000 polygons per frame and was no longer very interactive. Pixel-Planes 4 could draw about 30,000 polygons/second, which required a dataset smaller than 1500 polygons in order to achieve updates of 10 frames/second for binocular images on the head-mounted display.

Even with the slow update rate (about 8 frames/second), the system was responsive enough to let me navigate the 3-torus. Whenever I crossed a portal, I left layer 0 to enter layer 1. Layer 1 then became the new layer 0. This created a distracting visual effect: a brand-new layer 3 suddenly became visible in the distance. When I retreated through the portal, part of the old layer 3 became part of the new layer 4 and suddenly disappeared. This artifact would be less noticeable on a system powerful enough to render more layers, because the appearance/disappearance would occur even farther away. To diminish this artifact, I introduced “fog” into the room. The fog was sufficiently opaque to make the farthest instance of the room become barely discernible.

What I learned was that I could grab an object near me without looking “directly” at it. I looked instead at an instance of the object that was a layer or two away, grabbed it, and positioned it. The other instances of the object all moved in the same manner as the one I was looking at. Perhaps this application of “virtual reality” will help communicate the flavor of 3-manifolds to a wider audience, but the graphics hardware is just not fast enough yet for anything beyond a toy system like the one I built. Suppose the room holds a modest dataset with 1000 polygons, and there are just 4 layers of the room (in the topology of a 3-torus) displayed onto 2 screens (one for each eye) at 20 frames/second. This requires that the system process (transform and render) about 7 million polygons/second. At the time of this writing, no system exists that can process even 3 million polygons/second at a sustained rate.

2 The Basics of Graphics in 4-space

There are three basic tasks (figure 2-1) in conventional interactive 3D graphics. These are 1) mapping control-input from user space to object space, 2) projecting from object space to illumination space, and 3) projecting from illumination space to the screen. The first three spaces are typically Euclidean 3-space, but in the case of surfaces in 4-space the object space is 4-dimensional.

An object in 3-space has fewer degrees of freedom than it would have in 4-space. Is there an intuitive way to promote 3D input so that it controls motion in the 4D object space? In section 2.1 I present a practical solution to the problem of mapping input from user space to object space.

If illumination is deferred until the surface is projected down to 3-space, the conventional calculations (for computing the diffuse and specular illumination) can still be applied. Is there a way to illuminate the surface in 4-space? In section 2.2 I offer a simple technique for illuminating a surface no matter how large the object space is.

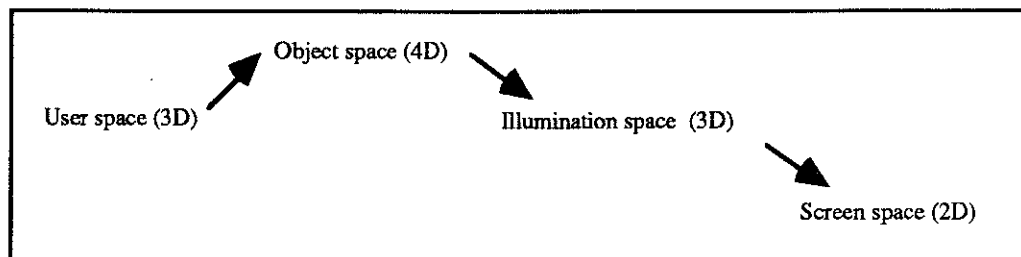


Figure 2-1. A user in 3-space manipulates a surface in 4-space, which projects to 3-space and then onto the screen.

There are several cues that lend a 3D effect to images on a computer screen. Among them are occlusion, shadows, shading, perspective, parallax, stereopsis, focus, and texture. These are natural cues that one uses every day to derive a 3D model of the world from the 2D image of it on the retina. But there is a serious problem. By projecting the image of a surface in 4-space down to a 2-dimensional screen, not only is depth information lost in the *z*-direction, but it is also lost in the *w*-direction as well. What 4-dimensional depth cue does the retina employ that computer graphics can now supply when rendering the surface? Since both the *z*- and the *w*-directions are perpendicular to the screen, one might try applying

some of the usual z -depth cues as w -depth cues. This strategy risks ambiguating the two depths, of course. The alternative is to invent w -depth cues that have no basis in physical experience. So what cues can be applied in four dimensions using computer graphics? Section 2.3 answers this general problem.

Finally, there is a question about how the surface should be projected to screen space. What is the effect of using perspective projection versus orthographic projection? Although perspective can act as a depth cue in its own right, I have collected the mathematical discussion of perspective projection into section 2.4.

2.1 How to Map User Space to Object Space

Dynamic control of the transformation matrices requires an input device that offers a natural means for producing the object's motion. In mapping the state of an input device which exists physically in 3-space to the state of a surface that exists virtually in 4-space, the challenge is to promote kinesthetic sympathy, which simply means that the input motion is similar to the object motion as it appears on the display [Gauch87].

There are ten degrees of freedom that are necessary to move an object in 4-space. Four of the ten degrees of freedom correspond to translations

$$\Delta x, \Delta y, \Delta z, \Delta w$$

in the axial directions. The remaining six degrees correspond to Euler angles of rotation. Christoff Hoffman gave a detailed treatment of Euler angles in 4-space [Hoffman90] and used the following suggestive notation to represent them.

$$R^4_{xw}, R^4_{yw}, R^4_{zw}, R^3_{xz}, R^3_{yz}, R^2_{xy}$$

The subscripts refer to the axes that define the plane of rotation. The superscript is the dimension of the smallest space (chosen from among the 4D $xyzw$ -space, the 3D xyz -space, and the 2D xy -space) that can contain the rotation.

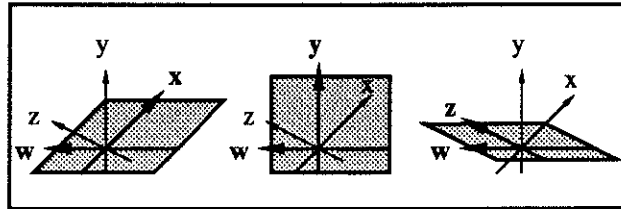


Figure 2-2. These are three of the six axial planes in $xyzw$ -space, defined by the axis pairs xw , yw , and zw . The other three axial planes (xz , yz , and xy) lie in the 3-dimensional xyz -subspace.

The 4D rotations behave very much like their 3D counterparts, although one should think of rotations as occurring within a plane (figure 2) rather than occurring about an axis. In 3-space, rotations leave a 1-dimensional subspace fixed. That subspace is the rotation axis. In 4-space, rotations leave a 2-dimensional subspace fixed while moving the points that are outside of the subspace. In general, the 4×4 rotation matrix A for the $x_i x_j$ plane ($i < j$), contains the elements

$$a_{ii} = a_{jj} = \cos t$$

$$a_{ij} = -a_{ji} = (-1)^{(i+j)} \sin t,$$

with the remaining elements being the same as in the identity matrix.

2.1.1 Mapping 2D Input to 3D Transformations

The fact that an input device is constrained to lie within a physical 3-dimensional world will impair kinesthetic sympathy with respect to motion in 4-space. The question is, how much? This problem is very familiar in a different guise, namely, how to affect direct 3D manipulations with a 2D locator such as a mouse. In this case there are six degrees of freedom (three Euler angles and three orthogonal translations) to associate with the 2-dimensional input space of the mouse. There are several techniques that have been devised to solve the problem. The following classification of them is my own invention – there is considerable published research describing ways to produce the mapping, but a coherent taxonomy to organize the various mapping techniques has not heretofore been devised.

Basis of the Taxonomy

The basis of my classification is as follows. One desires to map n translations and n -choose-2 rotations from n -dimensional input space up to $(n+1)$ -dimensional object space. But the object space has $(n+1)$ translations and $(n+1)$ -choose-2 rotations. The image of a domain, under a continuous map, cannot have a larger dimension than the domain itself does, so there is actually no solution to the stated problem. After all, the range has the larger dimension. But an interactive system does not necessarily require that every possible translation or rotation be available. For example, a user can approximate a diagonal path by using vertical and horizontal motions in alternation. He can reach his desired destination, but he must do so via a somewhat complicated path through the abstract space of axial translations. If the user can produce each of the axial translations or rotations independently, then he can alternate between them to arrive eventually at any particular combination of them. This then is the more modest goal of most of the techniques.

It is easy to state the desired property of an input mapping when the dimensions of the input space and the object space are the same. If the input device has a state vector U

$$U = (u_1 \ u_2 \ u_3 \ \dots \ u_n)$$

and the object has a state vector V

$$V = (v_1 \ v_2 \ v_3 \ \dots \ v_n)$$

then it is desirable to make each variable in U correspond to each variable in V . Thus

$$\frac{\partial v_i}{\partial u_j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

which means that the matrix of partial derivatives is the identity matrix. Actually the requirement that $\frac{\partial v_i}{\partial u_i} = 1$ is too strong. As long as these partials are positive-valued, the mapping will display some kinesthetic sympathy (If the partial derivative is negative, the object's motion will be anti-sympathetic to the user's motion). So the goal is to create a mapping whose matrix of partials is a diagonal matrix. Let the symbol " \sim " denote a positive-valued function. In the case of mapping translations from the 2D input space (u_1, u_2) into translations on the 2D screen (v_1, v_2), one wants to have

$$\begin{bmatrix} \frac{\partial v_1}{\partial u_1} & \frac{\partial v_1}{\partial u_2} \\ \frac{\partial v_2}{\partial u_1} & \frac{\partial v_2}{\partial u_2} \end{bmatrix} = \begin{bmatrix} \sim & 0 \\ 0 & \sim \end{bmatrix}$$

which is a mapping that can easily be accomplished: just let $v_i = u_i$. But consider instead the problem with mapping 2D translations (u_1, u_2) into 3D translations (v_1, v_2, v_3). The shape of the matrix of partial derivatives is 3-by-2, not 3-by-3. The goal then is to make the partial matrix

$$\begin{bmatrix} \frac{\partial v_1}{\partial u_1} & \frac{\partial v_1}{\partial u_2} \\ \frac{\partial v_2}{\partial u_1} & \frac{\partial v_2}{\partial u_2} \\ \frac{\partial v_3}{\partial u_1} & \frac{\partial v_3}{\partial u_2} \end{bmatrix}$$

somehow look like the desired matrix

$$\begin{bmatrix} \frac{\partial v_1}{\partial u_1} & \frac{\partial v_1}{\partial u_2} & \frac{\partial v_1}{\partial u_3} \\ \frac{\partial v_2}{\partial u_1} & \frac{\partial v_2}{\partial u_2} & \frac{\partial v_2}{\partial u_3} \\ \frac{\partial v_3}{\partial u_1} & \frac{\partial v_3}{\partial u_2} & \frac{\partial v_3}{\partial u_3} \end{bmatrix} = \begin{bmatrix} \sim & 0 & 0 \\ 0 & \sim & 0 \\ 0 & 0 & \sim \end{bmatrix}$$

if possible.

The next four sections describe techniques that other researchers have proposed for mapping 2D input to 3D motion. I have collected them together according to the general strategy they employ. These strategies are 1) to discard control of some subset of object space; 2) to overload the input space; 3) to partition the input space; or 4) to create a cross-product of the input space. These examples offer a compressed review of the individual techniques for mapping 2D input to 3D transformations. Their purpose here is to illustrate the taxonomy so that I can then apply it to the problem of mapping 3D input to 4D transformations.

Discarding Variables in Object Space

One can simply discard one or more state variables in object space in order to make the dimensions of the input space and the object space be the same. For example, one can elect to fix the y -coordinates and merely translate an object in the x - and z -directions. Then the input space (x', y') and the object space (x, z) both are 2-dimensional. Admittedly, it is somewhat disingenuous to announce that the object's state variable y has been "discarded". The fact is, the y -values just cannot be changed via the input device.

When a j -dimensional space J of translations and rotations maps to a k -dimensional space K ($j < k$), the rank of the mapping is at most j . So at every point in K there is a local nontrivial normal space. It might be the case that the local normal spaces all share a nontrivial intersection N , spanned by parameters n_i . N is a global normal space and the input mappings from J to N have all been "discarded"; that is, the partial of n_i with respect to any input variable is identically zero. In the simple example above, $\delta y / \delta x'$ and $\delta y / \delta y'$ are both identically zero. If x' maps to x and y' maps to z , the matrix of partials looks like this.

$$\begin{bmatrix} \frac{\partial x}{\partial x'} & \frac{\partial x}{\partial y'} \\ \frac{\partial y}{\partial x'} & \frac{\partial y}{\partial y'} \\ \frac{\partial z}{\partial x'} & \frac{\partial z}{\partial y'} \end{bmatrix} = \begin{bmatrix} \sim & 0 \\ 0 & 0 \\ 0 & \sim \end{bmatrix}$$

As a more interesting example, a cursor's velocity vector can control a rotation of the unit 2-sphere that has been projected to the screen, and hence it can control the rotation of the 3-space that the 2-sphere inhabits. Michael Chen called this the "virtual sphere" controller [Chen88]. Dragging the cursor across the projected image of a sphere causes the sphere to rotate to keep up with the cursor. When the user presses the mouse button, he grabs a point on the sphere. That distinguished point stays virtually attached to the moving cursor, which

induces various rotations in the sphere. The plane of rotation is the plane that contains the sphere's center, the distinguished point, and the cursor's tangent vector. When the mouse button is pressed to initiate rotation, only a restricted set of rotations is available. Locally, the distinguished point moves strictly along great circles. Suppose the user grabs the center of the hemisphere's image on the screen, and that the hemisphere sits in 3D object space so that the x-axis points rightward, the y-axis points upward, and the z-axis points outward through the distinguished point. Again let x' and y' increase in the horizontal and vertical directions of input space. Then the following matrix describes Chen's mapping.

$$\begin{bmatrix} \frac{\partial R_{xy}}{\partial x'} & \frac{\partial R_{xy}}{\partial y'} \\ \frac{\partial R_{yz}}{\partial x'} & \frac{\partial R_{yz}}{\partial y'} \\ \frac{\partial R_{xz}}{\partial x'} & \frac{\partial R_{xz}}{\partial y'} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ \sim & 0 \\ 0 & \sim \end{bmatrix}$$

Another way to discard a dimension of control is to map the input space to the tangent space at a point on a surface [Nielson86, Bier86, Hanrahan90, Smith], thereby manipulating the surface by controlling its motion within that tangent plane. When the user presses the mouse button he selects a point on whatever surface is displayed beneath the cursor.

Translating the mouse consequently translates the surface so that the selected point always remains in the same tangent plane. One problem with this mapping is that it exhibits less kinesthetic sympathy as the tangent plane deviates from the image plane. A more abstract problem is that path-planning can become very difficult when it requires the user to take a route through successive tangent planes in order to reach a target orientation. When a surface in 3-space isn't closed or isn't everywhere differentiable, its Gauss map might not cover the unit sphere. Such a surface can be difficult (or even impossible) to orient by controlling it through its tangent space. The simplest example of this problem is presented by a single rectangular patch. Since it only offers one tangent plane to the cursor, it cannot be rotated out of its plane. When the rectangle is aligned to lie within the xz-plane of object space, the situation matches the first example described above.

Both of these examples share a simple property. The matrix of partials has a row of zeros. That row corresponds to the variable in object space that the input map neglects. (Notice also that both techniques discard a different subset of the object space depending on where the cursor is located when the user's interaction begins. If the user picks a different

distinguished point, he effectively selects a different mapping of his motion. This scheme is described under the heading “Partitioning the Input Space.”)

Overloading the Input Mapping

Ideally, the various components of input (like the horizontal and vertical motion of a mouse) map independently and unambiguously into components of an object’s motion on the screen. For example, a horizontal sweep of the mouse makes a cursor glide horizontally across the display. But the input could also cause additional motion in the graphical object, whether that motion is desired or not. A familiar example from everyday life is a toy car. When the car is shoved forward by the hand pushing in the x' -direction, the wheels translate forward in the x -direction along with the car – but they also rotate in the wheel’s xy -plane because their frictional contact with the floor imparts rotational acceleration to them. The forward motion of the hand is mapped into both translation and rotation in the wheels; this is an overloaded input mapping. Two logical dimensions (translation and rotation) of the object space are both combined from a single dimension (translation) of the physical input space. That single dimension is thus overloaded. In terms of the partial derivatives, the matrix

$$\begin{bmatrix} \frac{\partial x}{\partial x'} \\ \frac{\partial R_{xy}}{\partial x'} \end{bmatrix} = \begin{bmatrix} \sim \\ \sim \end{bmatrix}$$

describes the mapping.

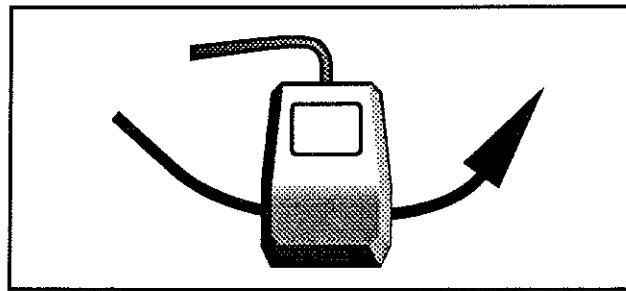


Figure 2-3. At the bottom point of this circular trajectory, the mouse's velocity is purely horizontal, while its acceleration is purely vertical.

One can overload the input space (x', y') of a 2D locator in the following way to produce translations in the (x, y, z) object space. The x' and y' components of the locator’s velocity map to the x and y components of the object’s velocity. Then the magnitude of the locator’s circular acceleration maps onto the z component of the object’s velocity [Evans81]. Converting these components into translations in x , y , and z preserves sympathy for x and y .

The third mapping suggests the “natural” translation for z that an ordinary bolt exhibits: when it twists, it moves forward or backward through a nut. An important drawback to mapping the input space this way is that the locator’s velocity and acceleration are not decoupled. If the user wants to change the direction of the locator’s motion, that change necessarily produces a circular acceleration and hence a z -translation in object space (figure 2-3). In terms of partial derivatives, this mapping looks like

$$\begin{bmatrix} \frac{\partial x}{\partial x'} & \frac{\partial x}{\partial y'} \\ \frac{\partial y}{\partial x'} & \frac{\partial y}{\partial y'} \\ \frac{\partial z}{\partial x'} & \frac{\partial z}{\partial y'} \end{bmatrix} = \begin{bmatrix} \sim & 0 \\ 0 & \sim \\ \sim & \sim \end{bmatrix}$$

which, again, differs from a diagonal matrix.

Note that both of these matrices share a simple property. There exists at least one column containing too many nonzero functions. That means that there are two or more variables in object space that depend nontrivially on a single variable in input space. This is the technical meaning of “overloading” the input space. One can always establish a different coordinate system on the object space so that the mapping is no longer overloaded. The goal, however, is to retain the “canonical” representation of surfaces in object space in terms of individual translations and rotations.

Partitioning the Input Space

One can partition the input space into components, each of which maps the locator motion onto the object motion in a different manner. The partition can be explicit, by determining in which of several control areas a cursor lies. In one area of the screen, the cursor might control a single Euler angle, while in another area the cursor might control a different angle of rotation [Chen88]. A simple version of this mapping is widely used in the form of slider bars, with each slider controlling a single plane of rotation or a single direction of translation. If the slider bar is oriented horizontally, the mouse moves in the x' -direction in order to move the “thumb” of the slider. If the sliders are arranged vertically (slider₁ above slider₂ above slider₃), the mouse moves in the y' -direction in order to select a different slider. Here is how the matrix of partials looks (for translation only), depending on where the mouse is located vertically when it begins its horizontal motion.

$$\begin{bmatrix} \frac{\partial x}{\partial x'} \\ \frac{\partial x}{\partial y'} \\ \frac{\partial x}{\partial z'} \end{bmatrix} = \begin{bmatrix} \sim \\ 0 \\ 0 \end{bmatrix} \text{ (y' in slider1), } \begin{bmatrix} \frac{\partial x}{\partial x'} \\ \frac{\partial x}{\partial y'} \\ \frac{\partial x}{\partial z'} \end{bmatrix} = \begin{bmatrix} 0 \\ \sim \\ 0 \end{bmatrix} \text{ (y' in slider2), } \begin{bmatrix} \frac{\partial x}{\partial x'} \\ \frac{\partial x}{\partial y'} \\ \frac{\partial x}{\partial z'} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sim \end{bmatrix} \text{ (y' in slider3)}$$

The partition can also be implicit, by comparing the motion of the 2D locator to the orientation of a 3D cursor that is projected to input space [Nielson86]. In this scheme, any motion in the direction of a given axis produces a translation in that axial direction. Assume now that the mouse has a local polar coordinate system (r, θ) . When $\theta = 0$, the mouse is moving horizontally with velocity $\frac{dr}{dt}$. When $\theta = 90$ degrees, the mouse is moving vertically. When $\theta = 45$ degrees, the mouse is moving diagonally. The partial matrix then depends on θ , and it looks like the following.

$$\begin{bmatrix} \frac{\partial x}{\partial r} \\ \frac{\partial x}{\partial \theta} \\ \frac{\partial x}{\partial z} \end{bmatrix} = \begin{bmatrix} \sim \\ 0 \\ 0 \end{bmatrix} \text{ (}\theta \text{ near } 0\text{), } \begin{bmatrix} \frac{\partial x}{\partial r} \\ \frac{\partial x}{\partial \theta} \\ \frac{\partial x}{\partial z} \end{bmatrix} = \begin{bmatrix} 0 \\ \sim \\ 0 \end{bmatrix} \text{ (}\theta \text{ near } 90\text{), } \begin{bmatrix} \frac{\partial x}{\partial r} \\ \frac{\partial x}{\partial \theta} \\ \frac{\partial x}{\partial z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sim \end{bmatrix} \text{ (}\theta \text{ near } 45\text{)}$$

Both of these systems share a simple property: each available matrix in the system is a submatrix of the desired diagonal matrix. In other words, there is always some condition that can be met (e.g., “cursor in slider₁”) which insures that a given variable in input space controls only a single variable in object space, and every variable in object space can be controlled by a single input variable.

Whenever a partition is employed, the user must be prepared for the object’s motion to suddenly change when the input space switches context, and he must be aware of which mapping is currently invoked. This is a problem if he inadvertently activates the wrong slider bar, or if he wishes to translate an object diagonally when the z-axis, as it appears on the screen, runs diagonally as well (when using Nielson’s technique).

Taking a Cross Product of the Input Space

If one instance of the input space provides only a few degrees of freedom, an attractive solution to the mapping problem is to use multiple instances of that input space. If these instances are independent of each other, the array of them yields a cross product of the individual spaces. By using k locators, each with n degrees of freedom, one can achieve n^k

degrees of freedom in the input space. This arrangement can be maximally realized by using k different physical locators. The arrangement can be minimally realized by using one physical locator with a k -way selector to map it to each of the n logical locators (like using a switch to change from rotation-mode to translation-mode). On an ordinary mouse, a single button can select between two different mappings from input space to object space [Chen88]. There can also be hybrids of these two extremes, where several input devices each have several modes.

As a simple example, consider the case of using two mice for controlling translations in 3D object space. Each mouse has a 2D coordinate system. Taken together, these systems provide input variables x_1', y_1', x_2', y_2' . Suppose that the first three of these variables map to the variables x, y , and z (respectively) in object space. Then one has the following matrix of partials.

$$\begin{bmatrix} \frac{\partial x}{\partial x_1'} & \frac{\partial x}{\partial y_1'} & \frac{\partial x}{\partial x_2'} & \frac{\partial x}{\partial y_2'} \\ \frac{\partial y}{\partial x_1'} & \frac{\partial y}{\partial y_1'} & \frac{\partial y}{\partial x_2'} & \frac{\partial y}{\partial y_2'} \\ \frac{\partial z}{\partial x_1'} & \frac{\partial z}{\partial y_1'} & \frac{\partial z}{\partial x_2'} & \frac{\partial z}{\partial y_2'} \end{bmatrix} = \begin{bmatrix} \sim & 0 & 0 & 0 \\ 0 & \sim & 0 & 0 \\ 0 & 0 & \sim & 0 \end{bmatrix}$$

This approach is attractive because it satisfies the desirable requirement that each variable in the object space be uniquely controlled by an input-space control. But it too has pragmatic drawbacks: few systems are designed with 4-dimensional space in mind, so they are unlikely to offer three mice or two joysticks for rotational controls. That means the user-interface must multiplex the mapping of a single input device to mimic the effect of several physical devices, which puts the system into different distinct modes of control. The user must then be made aware of which mode the system is in and how to establish a different mode. In the other extreme, even a system that had enough slider bars for every variable would not necessarily be convenient to use. With an abundance of physical controls, the user must become expert at locating the correct control or else he will continually be shifting his gaze from the display to the control area.

2.1.2 Mapping 3D Input to 4D Transformations

One of my goals for the Fourfront system is to let a user gain experience with individual translations and rotations in 4-space. What does the experience of mapping 2D input to 3D manipulation suggest for mapping 3D input into 4D manipulation? I will discuss the approaches outlined above, this time in the context of my actual problem.

(1) Discarding one or more dimensions of manipulation causes a significant problem: it violates one of my goals for the user's interaction. Unless he can manipulate the individual translations and rotations of a surface in 4-space, the user is considerably handicapped. But the techniques I described in the previous section still have merit in 4-space, since they discard a different subspace depending on where the interaction begins. Hence, they deserve more discussion.

Chen's idea was to map 2-dimensional input onto a virtual sphere in 3-space. This idea is extensible: one could map a velocity vector from 3-dimensional input space into rotations of the unit 3-sphere in 4-dimensional object space. As a practical matter, a virtual 3-sphere controller is clumsy to use when one merely wants to achieve ordinary rotations within 3-dimensional xyz -space. To understand why, consider how to manipulate the virtual 2-sphere controller so as to rotate an object precisely in the xy -plane of the display device. Rotating the xy -plane means rotating about the z -axis (which is located in the center of the visible hemisphere). To rotate about an axis that pierces the virtual sphere at a point \mathbf{p} , one must position the mouse to grab a distinguished point on the equator with respect to \mathbf{p} . The equator with respect to the z -axis is the outer edge of the virtual sphere. (Chen made this particular rotation of the xy -plane more accessible by taking all the inputs beyond the visible disk of the virtual sphere and mapping them onto the sphere's outer rim.) The situation is similar for the virtual 3-sphere controller. One hemisphere of the 3-sphere maps from 4-space to 3D input space, filling out a solid spherical volume in input space. To rotate an object precisely within xyz -space requires the user to grab a point on the "outer edge" of the spherical volume and then to keep moving this distinguished point along the outer edge. I have observed that users generally like to apply 4-space rotations (involving the w -axis) briefly and then to re-orient the surface within the more familiar xyz -space. The virtual 3-sphere controller makes the latter task inconvenient. One solution could be to toggle between different modes of rotation, which means using a cross-product of input devices.

Nielson's strategy of exploiting the tangent space in order to define the mapping from input space to object space is generally workable for surfaces in 3-space. In 3-space every compact 2-manifold has a Gauss map that covers the sphere, and hence it possesses every

possible orientation of tangent planes. But in 4-space the tangent planes of a compact 2-manifold cover only a measure-0 set of the possible orientations of all planes. That means the user can only translate a surface in only a scant set of directions, if those directions must lie within the surface's tangent planes.

In summary, these particular techniques can in fact be extended to 4-space. They do, however, exhibit certain drawbacks. Moreover, the general scheme of discarding a parameter in object space causes a system to hinder the user as he attempts to interact naturally with surfaces in that space.

(2) Overloading the input space can produce transformations in 4-space that the user did not intend. This effect is commonplace with an overloaded mapping, since by definition some (or all) of the input dimensions have side-effects. The user may have accidentally translated a surface in the z-direction by looping the 2D mouse in an arc, for example. This can be easy to repair, by looping the mouse along the same arc but in the opposite direction. The 2D mouse is, after all, confined to lie in a plane, so it is fairly easy to make one's hand repeat a brief gesture in reverse. The problem is more serious in 3-space when the user generates side effects during an attempted 3D manipulation. A gesture in 3-space is more difficult to repeat in reverse. The point is that whenever the input space is overloaded, the user is likely to produce unintended motion in object space. When the novice user engages in an interactive session with Fourphront, I want him to be able to recover from errors quickly. Thus I prefer to avoid overloading the input space.

(3) One can partition 3D input space to allow the user to choose from among different input maps. The two examples of 2D partitions both permit the system to give the user some visual feedback on the screen (by displaying the sliders in one case, and by displaying the x, y, and z axes in the other case). The user can easily see how to move the mouse so that he invokes the mapping he desires. In 3-space the problem is harder.

If slider bars are promoted to become slider boxes in a 3D input space, the problem becomes how to access them. Consider first a rectangular 2D space filled with slider bars. The hand can freely move from one slider to another within the 2D input area because the hand and arm lie in 3-space. If the hand were constrained to remain in the plane, moving from one control to another would become a difficult task because the intervening sliders are obstacles. Likewise, if physical input devices fill a region of 3D space, they simply get in the way of each other. The user must be able to reach each of them if he is to use them. They could instead be virtual controls that the user manipulates through a single 3D locator (like a Polhemus tracker, with one user-dimension used to select the active slider box). The

problem then is how to provide feedback so that the user knows which controller he has activated, and how near he is to the controller he wishes to use.

Nielson's method for partitioning a locator's 2D space extends to 3D for mapping translation. The system could display four coordinate axes in 3-space. As a 3-dimensional cursor moves in a direction that closely matches that of an axis, it produces a translation in that axial direction. Choosing rotations in a similar way is problematic. Instead of only four translation axes, there are now six rotation planes to choose among. I experimented with a 4-space gnomon, consisting of the six axial planes, as a visualization tool to help the user track the orientation of a surface. What I found was that the set of a mere six interpenetrating axial rectangles was very confusing and not especially helpful. In an informal survey, most users admitted that they simply stopped looking at it because it was no more informative than the surface itself. There may be other ways to partition the input space to achieve intuitive rotations, but they have not yet been invented.

(4) Using multiple input devices can be inconvenient. In order to collect the ten degrees of freedom that a surface in 4-space possesses, one needs five mice, each having two degrees of freedom. Or instead, one could use four different 3D devices (such as a joystick that can twist). There are input devices that provide six degrees of freedom (such as a Polhemus tracker); one needs two of them to obtain the ten degrees necessary. The problem with using many input devices is that the user can generally use only device one per hand at a time. It is possible, although cumbersome, to provide a device for each hand. The problem is the presence of wires on the input device, which tend to tangle when each hand controls a device.

The proliferation of input devices can be curbed by multiplexing them, that is, by using a k-way switch to treat one physical device as k logical devices. A single 2D-mouse will suffice if there is a 5-way switch, for example. Similarly a single button on each physical 3D-joystick can toggle that joystick between two different modes, thereby producing four logical joysticks from only two physical ones, for a total of twelve degrees of freedom. Likewise a single button can toggle one physical 6D Polhemus tracker between two modes, thereby producing two logical devices and a total of twelve degrees of freedom. A button is preferable to a 5-way switch because a single click unambiguously selects the desired mode.

Mapping the Joystick and the 3-space Tracker

Which of the strategies is the most effective for mapping 3D input to 4D transformations? There may be no single "best" technique, but multiple input devices promise the best coverage of all the parameters in object space. The relative novelty of interactive

manipulation in 4-space is a powerful motivation for designing a sympathetic interface. Few people have developed a sense of how surfaces look as they rotate in 4-space. Consequently, I wish to mirror that motion as closely as possible by the motion of the input device. Of the devices listed above, spaceballs and joysticks provide the most degrees of freedom. How can they produce sympathetic motion in 4-space?

Translations and rotations within an input plane $x'y'$ can sympathetically and uniquely map to motion within an image plane defined by the xy plane in object space. That makes two translations and one rotation which can be mapped trivially from input space to object space. In contrast, translation in the z or w directions and rotations in the wx , wy , zx , or zy planes (the offending direction comes first in each pair) present a problem. The reason is that projection from 4-space to the screen will annihilate two orthogonal directions z and w , together with the 2-dimensional zw -plane they define. This plane will apparently go “into” the screen at each point. As a result, it is natural that the input device move toward (“into”) the screen for one to map that motion to z or to w . Either choice preserves kinesthetic sympathy, but the map is not unique. Rotation in the zw plane (the remaining degree of freedom) is also problematic. There is no physical rotation of a 3D input device sympathetic to this 4D rotation, since (in physical 3-space) such a rotation would be confined to the 1-dimensional input space z' that lies perpendicular to the screen. The sympathetic maps are tabulated below. The goal of each map is to preserve the x' and the y' directions, and to assign z' to either the z or the w direction.

input space	x'	y'	z'	$R^2_{x'y'}$	$R^3_{x'z'}$	$R^3_{y'z'}$???
object space	x	y	z w	R^2_{xy}	R^3_{xz} R^4_{xw}	R^3_{yz} R^4_{yw}	R^4_{zw}

Figure 2-4. The mappings of 3D input space to 4D world space that promote kinesthetic sympathy.

Despite the ambiguities, there are still reasonable ways to convert input from a 3-space tracker or a joystick into 4D transformations. Recall that a 3-space tracker offers six degrees of freedom: three translations (x' , y' , z') and three rotations ($x'y'$, $x'z'$, $y'z'$). To extract ten degrees of freedom requires two trackers, either physically or logically.

The mapping from input space to object space can be defined as follows. Tracker₁ assigns its (x' , y' , z') to (x , y , z) for calculating translations and rotations. Tracker₂ assigns its z' direction to w instead of to z . Tracker₂ also makes the exception that rotations in its $x'y'$ -

plane map to rotations in the object's zw -plane. This rotation is not sympathetic, but, as pointed out above, no rotation in input-space can be sympathetic to a zw rotation. Note that the two physical trackers compete to produce x and y translations under this scheme. It is necessary then to discard one tracker's input or else somehow to average their inputs together. This makes the two-tracker solution somewhat unattractive.

A pair of 3D joysticks, each using twist (about the joystick axis) as the third degree of freedom, can map in a similar way to the 3-space tracker: together the pair of joysticks mimics the mappings of a single tracker. A given joystick rotates in each of three planes based at a common origin. Two of the rotations behave like translations for a short interval: when the joystick is centered, a rotation in its $x'z'$ or $y'z'$ planes is momentarily a linear translation in the x' or y' direction (figure 2-5). The sympathetic map exploits this duality and assigns these two motions into either a rotation or a translation in 4-space. Twist is not kinesthetically sympathetic to translation, but it is at least suggestive of forward motion that results from rotating a screw [Evans81].

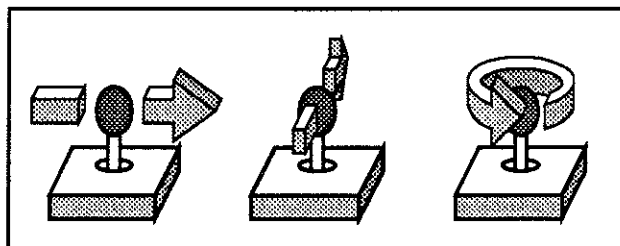


Figure 2-5. The 3D joystick rotates in the $x'z'$, $y'z'$, and $x'y'$ planes, which can produce a momentary translation in the x and the y directions. In the input space coordinates, x' is rightward, y' is forward, and z' is vertical.

One needs four logical joysticks in order to supply ten degrees of freedom. A joystick-pair has six degrees and can map in the same way that a 3-space tracker does. One pair maps its z' direction to z , and the other pair maps its z' direction to w . (There is an additional and important consideration: since joysticks have a small range of motion, it is wise to treat their input as velocity rather than position when gross manipulations are desired.) Rather than use a physical pair of joysticks to emulate a 3-space tracker, I use a single button on the top of the joystick to toggle its state between translations and rotations. Thus there are only two physical devices (joystick₁ and joystick₂) needed to accomplish the mapping. Since each has two modes, the logical devices can be identified as joystick₁₁, joystick₁₂, and joystick₂₁, and joystick₂₂.

The mapping schemes are summarized in the following table (figure 6). The subscripts indicate which device supplies the input.

polhemus ₁	x'	y'	z'		$R^2_{x'y'}$	$R^3_{x'z'}$	$R^3_{y'z'}$			
polhemus ₂	x'	y'		z'				$R^2_{x'y'}$	$R^3_{x'z'}$	$R^3_{y'z'}$
joystick ₁₁	$R^3_{y'z'}$	$R^3_{x'z'}$	$R^2_{x'y'}$							
joystick ₁₂					$R^2_{x'y'}$	$R^3_{x'z'}$	$R^3_{y'z'}$			
joystick ₂₁	$R^3_{y'z'}$	$R^3_{x'z'}$		$R^2_{x'y'}$						
joystick ₂₂								$R^2_{x'y'}$	$R^3_{x'z'}$	$R^3_{y'z'}$
object space	x	y	z	w	R^2_{xy}	R^3_{xz}	R^3_{yz}	R^4_{xw}	R^4_{yw}	R^4_{zw}

Figure 2-6. The mappings of spaceball and joystick input that promote kinesthetic sympathy in 4D world space.

A given joystick thus has two modes: one for translation, and one for rotation. Even though the 3-space tracker permits a user to combine the positioning (translation) and orientating (rotation) operations, I have observed that users often decouple the two tasks. As a result, this assignment of the two modes has some justification. Hoffman observed that users also decouple 4D manipulations (the ones that involve the *w*-axis in world space) from 3D manipulations [Hoffman90] in order to inspect the “damage” that was done to the surface after rotating it in 4-space. Thus an alternative way to multiplex a joystick is to switch its mode from “3D” to “4D.” I chose against this scheme because it reinforces the sensation that the “fourth dimension” is set apart and distinguished in a peculiar way: the user toggles between the familiar 3D rotations and the unfamiliar 4D rotations. It was my goal to permit them to be mixed together naturally. Instead, the 3D mappings are controlled by one hand and the 4D mappings are controlled by the other.

Quaternion Rotations

Although it does not preserve kinesthetic sympathy, there is an exotic input-mapping peculiar to 4-space. This mapping applies quaternion multiplication to rotate objects. Recall that a quaternion is a linear combination of the basis elements 1, *i*, *j*, *k*. These elements obey a special set of multiplication rules: $i^2 = j^2 = k^2 = ijk = -1$; $ij = -ji = k$. It turns out that the set of 4D rotations (called SO(4), the special orthogonal group in 4 dimensions) splits into left- and right-multiplication by unit quaternions. Each joystick’s motion can be decomposed into three Euler angles. These angles can be mapped to the components *i*, *j*, *k* of a unit quaternion, which means that a left joystick and a right joystick can “naturally” produce quaternions **L** and **R**. One then treats the points on a surface like they were points in

quaternion space, mapping \mathbf{p} to \mathbf{LpR}^{-1} and then projecting down the real coordinate into 3-space.

The effect of a pure left (or a pure right) quaternion rotation is that the surface appears to twist while it rotates. When the left and right rotations are equal in direction and magnitude, they combine to produce an Euler rotation in 4-space. In practice it is difficult to make the positions of the joysticks exactly match each other, but the effect is can be produced if one is careful when moving the joysticks in tandem.

2.1.3 Simple Approximations of Rotation Matrices

The purpose of this section is to describe an easy way to collect the inputs from a pair of joysticks and convert them into a single rotation matrix in 4-space. The user can apply force to a joystick at a diagonal angle to produce a rotation that doesn't lie within an axial plane. Given the Euler angles of the individual rotations, one can derive the rotation matrix that describes the plane of rotation passing through the joystick in 3-space. The user can also apply twist to the joystick. That twist produces a rotation in the local coordinate system of the joystick itself (as opposed to a rotation in the original coordinate system of the joystick). In the Fourphront system, I allow the user to produce any combination of the six Euler rotations in 4-space by using two joysticks. Given those angles of separate devices in 3-space, how does one calculate the resulting plane of rotation in 4-space? As it turns out, there's no need to be fussy about how to construct the final transformation. The reason is that there is a very simple approach that suffices. As long as the rotation angle is small, the individual rotation matrices for each Euler angle can merely be multiplied together.

The Fourphront system lets the joysticks control rotational velocity, as opposed to absolute rotation, in object space. Without loss of generality, let x , y , and z denote the coordinate system of a single joystick with the z -direction pointing upward along the axis of the joystick. Let $R_{xy}(r)$, $R_{xz}(r)$, and $R_{yz}(r)$ denote rotations by a combined angle r within the xy -, xz -, and yz -planes. Moving the joystick in the x -direction to a point $(r, 0)$ indicates a relative rotation in the xz -plane. Moving it in the y -direction to a point $(0, r)$ indicates a relative rotation in the yz -plane. Moving it at an angle α off the axes to a point (r_x, r_y) indicates a rotation in the plane defined by the z -axis and the point (r_x, r_y) . This is a rotation by an angle r which is the distance from the origin to the point.

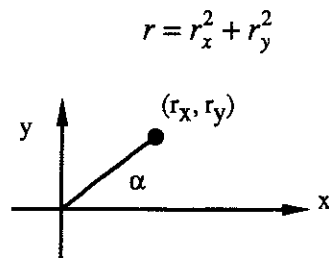


Figure 2-7. Moving the joystick produces a rotation. The velocity of rotation is proportional to the amount that the joystick is displaced from its centered, "rest" position.

The desired rotation is described in the following discussion.

The desired rotation results from spinning the coordinate system by an angle $-\alpha$ (see figure 2-7) in the xy -plane to align the point (r_x, r_y) with the x -axis, applying a rotation by the angle r in the xz -plane, and then unspinning again in the xy -plane by the angle α .

The following relations hold.

$$\cos \alpha = \frac{r_x}{r}; \quad \sin \alpha = \frac{r_y}{r}$$

These three intermediate transformations are thus $R_{xy}(\alpha)$, $R_{xz}(r)$, and $R_{xy}(-\alpha)$. Multiplying the three of them together yields the following result, with the above relations substituted into the first and third matrices.

$$\begin{aligned} R_{xy}(\alpha)R_{xz}(r)R_{xy}(-\alpha) &= \begin{bmatrix} \frac{r_x}{r} & \frac{r_y}{r} & 0 \\ -\frac{r_y}{r} & \frac{r_x}{r} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos r & 0 & -\sin r \\ 0 & 1 & 0 \\ \sin r & 0 & \cos r \end{bmatrix} \begin{bmatrix} \frac{r_x}{r} & -\frac{r_y}{r} & 0 \\ \frac{r_y}{r} & \frac{r_x}{r} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{r_x}{r} & \frac{r_y}{r} & 0 \\ -\frac{r_y}{r} & \frac{r_x}{r} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r_x \cos r}{r} & \frac{r_y \cos r}{r} & -\sin r \\ -\frac{r_y}{r} & \frac{r_x}{r} & 0 \\ \frac{r_x \sin r}{r} & \frac{r_y \sin r}{r} & \cos r \end{bmatrix} \\ &= \begin{bmatrix} \frac{r_x^2 \cos r + r_y^2}{r^2} & \frac{-r_x r_y \cos r + r_x r_y}{r^2} & \frac{-r_x \sin r}{r} \\ \frac{-r_x r_y \cos r + r_x r_y}{r^2} & \frac{r_y^2 \cos r + r_x^2}{r^2} & \frac{r_y \sin r}{r} \\ \frac{r_x \sin r}{r} & \frac{-r_y \sin r}{r} & \cos r \end{bmatrix} \end{aligned}$$

One can approximate this matrix up to $O(r^3)$ by using the Taylor expansion for the trigonometric functions.

$$\begin{bmatrix} \frac{1}{r^2} \left[r_x^2 \left(1 - \frac{r^2}{2} \right) + r_y^2 \right] & \frac{1}{r^2} \left[-r_x r_y \left(1 - \frac{r^2}{2} \right) + r_x r_y \right] & \frac{1}{r} [-r_x r] \\ \frac{1}{r^2} \left[-r_x r_y \left(1 - \frac{r^2}{2} \right) + r_x r_y \right] & \frac{1}{r^2} \left[r_y^2 \left(1 - \frac{r^2}{2} \right) + r_x^2 \right] & \frac{1}{r} [r_y r] \\ \frac{1}{r} [r_x r] & \frac{1}{r} [-r_y r] & \left(1 - \frac{r^2}{2} \right) \end{bmatrix} + O(r^3)$$

$$\begin{aligned}
&= \begin{bmatrix} \frac{1}{r^2} \left(r^2 - \frac{r_x^2}{2} \right) & \frac{1}{r^2} \left(r_x r_y \frac{r^2}{2} \right) & -r_x \\ \frac{1}{r^2} \left(r_x r_y \frac{r^2}{2} \right) & \frac{1}{r^2} \left(r^2 - \frac{r_y^2}{2} \right) & r_y \\ r_x & -r_y & 1 - \frac{1}{r^2} \end{bmatrix} + O(r^3) \\
&= \begin{bmatrix} 1 - \frac{r_x^2}{2} & \frac{r_x r_y}{2} & -r_x \\ \frac{r_x r_y}{2} & 1 - \frac{r_y^2}{2} & r_y \\ r_x & -r_y & 1 - \frac{r^2}{2} \end{bmatrix} + O(r^3)
\end{aligned}
\tag{1}$$

It is simpler to decompose the rotation in to the two "orthogonal" components $R_{xz}(r_x)$ and $R_{yz}(r_y)$ and multiply them together. That yields the following product.

$$\begin{aligned}
R_{yz}(r_y)R_{xz}(r_x) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos r_y & \sin r_y \\ 0 & -\sin r_y & \cos r_y \end{bmatrix} \begin{bmatrix} \cos r_x & 0 & -\sin r_x \\ 0 & 1 & 0 \\ \sin r_x & 0 & \cos r_x \end{bmatrix} \\
&= \begin{bmatrix} \cos r_x & 0 & -\sin r_x \\ \sin r_x \sin r_y & \cos r_y & \cos r_x \sin r_y \\ \sin r_x \cos r_y & -\sin r_y & \cos r_x \cos r_y \end{bmatrix}
\end{aligned}$$

A straightforward calculation will show that this approximation is very nearly the desired one when r is small. One can approximate this matrix up to $O(r^3)$ by using the Taylor expansion for the trigonometric functions.

$$\begin{aligned}
R_{yz}(r_y)R_{xz}(r_x) &= \begin{bmatrix} 1 - \frac{r_x^2}{2} & 0 & -r_x \\ (r_x)(r_y) & 1 - \frac{r_y^2}{2} & (1 - \frac{r_x^2}{2})(r_y) \\ (r_x)(1 - \frac{r_y^2}{2}) & -r_y & (1 - \frac{r_x^2}{2})(1 - \frac{r_y^2}{2}) \end{bmatrix} + O(r^3) \\
&= \begin{bmatrix} 1 - \frac{r_x^2}{2} & 0 & -r_x \\ r_x r_y & 1 - \frac{r_y^2}{2} & r_y - \frac{r_y r_x^2}{2} \\ r_x - \frac{r_x r_y^2}{2} & -r_y & 1 - \frac{r^2}{2} + \frac{r_x^2 r_y^2}{2} \end{bmatrix} + O(r^3)
\end{aligned}$$

$$(2) \quad = \begin{bmatrix} 1 - \frac{r_x^2}{2} & 0 & -r_x \\ r_x r_y & 1 - \frac{r_y^2}{2} & r_y \\ r_x & -r_y & 1 - \frac{r^2}{2} \end{bmatrix} + O(r^3)$$

Since $|r_x| < r$ and $|r_y| < r$, the last step extracts the $O(r^3)$ terms from the matrix and amalgamates them into the final order-notation in the expression. This matrix is a rotation matrix up to $O(r^3)$.

Moreover (2) is a close approximation to (1) for small r . The error is given by the absolute difference between (1) and (2).

$$err = \begin{bmatrix} 0 & \left| \frac{r_x r_y}{2} \right| & 0 \\ \left| \frac{r_x r_y}{2} \right| & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + O(r^3)$$

This error is quite small (quadratic error in two terms, cubic error in the rest of the matrix) for small r . The higher the frame rate, the smaller r needs to be to interactively rotate an object through each frame. In actual practice, users of Fourphront usually do not generate rotation angles larger than 0.1 radians per frame. In other words, they require about 2 seconds (60 frames) to accumulate 2π radians of rotation and spin a surface completely around.

2.2 Illumination

The purpose of this section is to present a simple way to illuminate a surface in 4-space. My aim is to create a general method for illuminating a surface with any positive codimension so that the method will agree with conventional techniques when that co-dimension happens to equal 1 (this being the usual case, namely a surface in 3-space).

2.2.1 Review of Illuminating Surfaces in 3-space

There are well-established methods within computer graphics for calculating the diffuse and specular components of illumination. These calculations are based on the physical interaction of light with a surface in 3-space. Diffuse and specular illumination are the cornerstones of conventional lighting effects.

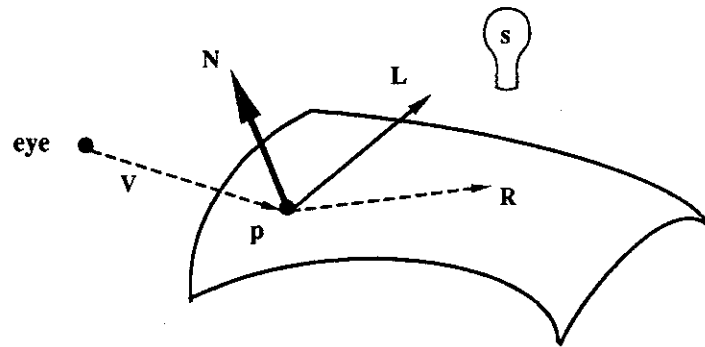


Figure 2-8. Diffuse and specular illumination at a point on a surface are governed by the surface normal N , the light vector L , and the view vector V .

The diffuse illumination of a surface by a point light source s depends on the function $\text{diff}_p(\cdot)$ that operates at each point p of a C^1 surface. Let L be the normalized light vector from p to s , and let N be the unit normal at p . Then $\text{diff}_p(\cdot)$ can be defined as follows. Let p be front-facing, so that $L \cdot N > 0$. In this case,

$$\text{diff}_p(L, N) = L \cdot N \in [0, 1]$$

When the surface is “back-facing” at p , that is, $L \cdot N < 0$, the tangent plane separates the viewer from the light source. The usual convention is to clamp $\text{diff}_p(\cdot)$ to zero in that case. Thus

$$\text{diff}_p(L, N) = 0$$

when $L \cdot N < 0$.

Now consider specular illumination. Let \mathbf{V} be the normalized view vector from the eye point to \mathbf{p} . An incoming ray of light bounces off the surface to produce a reflection vector \mathbf{R} .

$$\mathbf{R} = (2\mathbf{L} \cdot \mathbf{N}) \mathbf{N} - \mathbf{L}$$

Specular illumination at the point \mathbf{p} depends on the function $\text{spec}_{\mathbf{p}}()$:

$$\text{spec}_{\mathbf{p}}(\mathbf{V}, \mathbf{R}) = |\mathbf{V} \cdot \mathbf{R}|^{\text{pow}} \in [0, 1]$$

The exponent pow is some nonnegative real number. The larger it is, the shinier the surface looks.

2.2.2 Illumination Independent of Dimension

How should illumination behave when the codimension is greater than 1? In the case of a surface in 4-space or a curve in 3-space, the formulations for $\text{diff}()$ and $\text{spec}()$ no longer make sense, because there is no longer a 1-dimensional normal vector \mathbf{N} . Instead, there is a 2-dimensional normal vector space N .

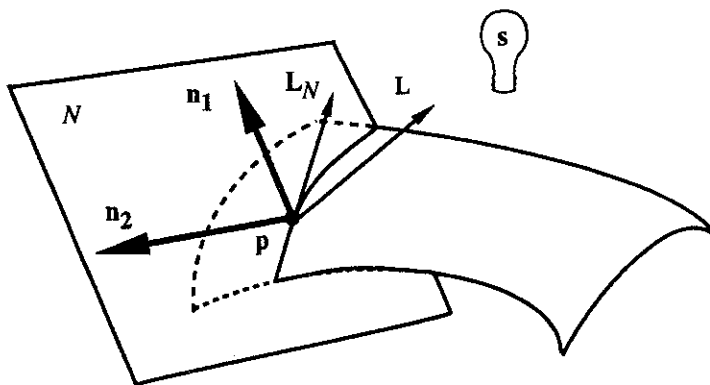


Figure 2-9. If the codimension is 2, the normal space at a point on the surface will be a plane N .

There is a natural remedy. Since N is spanned by k orthogonal basis vectors \mathbf{n}_i , a new dot product can be defined as follows. Just as the scalar quantity

$$\mathbf{L} \cdot \mathbf{N} / (\|\mathbf{L}\| \|\mathbf{N}\|)$$

measures the angle between \mathbf{L} and \mathbf{N} , the quantity

$$\mathbf{L} \cdot \mathbf{L}_N / (\|\mathbf{L}\| \|\mathbf{L}_N\|)$$

measures the angle between the vector \mathbf{L} and its projection \mathbf{L}_N to the vector space N , where

$$\mathbf{L}_N = \sum_{i=1}^k (\mathbf{L} \cdot \mathbf{n}_i) \mathbf{n}_i$$

Let \mathbf{L}'_N denote the normalized vector \mathbf{L}_N . So $\text{diff}_p(\)$ can thus be reformulated independently of the size of the codimension:

$$\text{diff}_p(\mathbf{L}, N) = |\mathbf{L} \cdot \mathbf{L}'_N|$$

When the codimension is greater than 1, the tangent space can no longer separate the light source from the eye (except in the degenerate case where the source, the eye, and \mathbf{p} are co-linear). Consequently, it makes no sense to clamp the dot product to zero (which would indicate the tangent plane is "hiding" the surface normal from the light) in the general case.

The reflection vector \mathbf{R} extends in a similar way. An incoming ray of light can be decomposed into its component \mathbf{L}_T (within the tangent space T) and its component \mathbf{L}_N (within the normal space N) at \mathbf{p} . The reflection preserves the tangent component but reverses the normal component. The usual convention in computer graphics is to reverse the direction that the vector \mathbf{L} points; thus

$$\mathbf{R} = \mathbf{L}_N - \mathbf{L}_T.$$

This calculation can be broken down into its constituent parts as follows.

$$\begin{aligned} \mathbf{R} &= \mathbf{L}_N - \mathbf{L}_T = \mathbf{L}_N - (\mathbf{L} - \mathbf{L}_N) = 2\mathbf{L}_N - \mathbf{L} \\ &= 2 \sum_{i=1}^k (\mathbf{L} \cdot \mathbf{n}_i) \mathbf{n}_i - \mathbf{L} \end{aligned}$$

With this reflection vector in calculated, the rest is easy. The very same specular equation now applies to the view vector and the reflection vector.

$$\text{spec}_p(\mathbf{V}, \mathbf{R}) = |\mathbf{V} \cdot \mathbf{R}|^{pow} \in [0, 1]$$

2.2.3 Why It Matters

One would like for illumination to graphically convey the amount of deviation of the light vector from the normal space (for diffuse) or the deviation of the eye from the reflection (for specular). This is one way to recover shape from shading [Horn89]. It is therefore incorrect to postpone the calculation until after projecting the surface to a lower dimension, because the various angles are not invariant under projection. There is only one circumstance that permits illumination to pass unchanged through the projections. That is when (1) all the

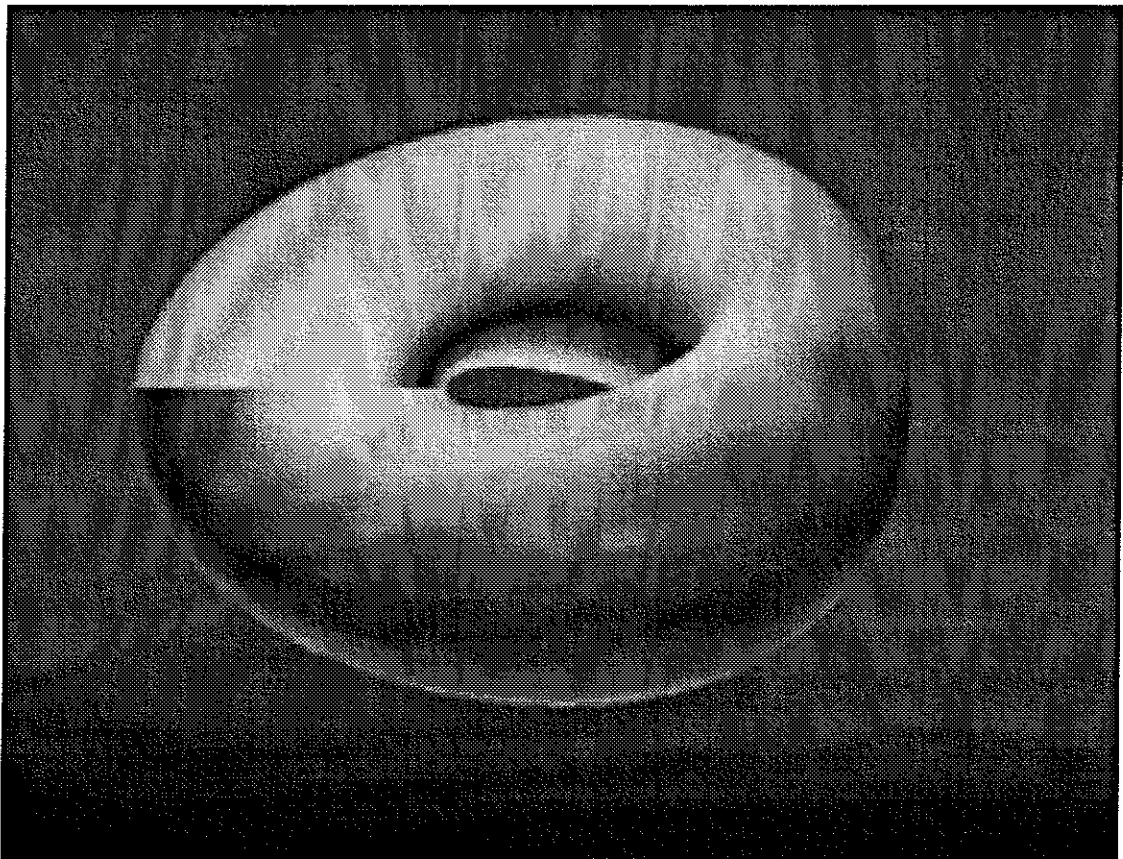
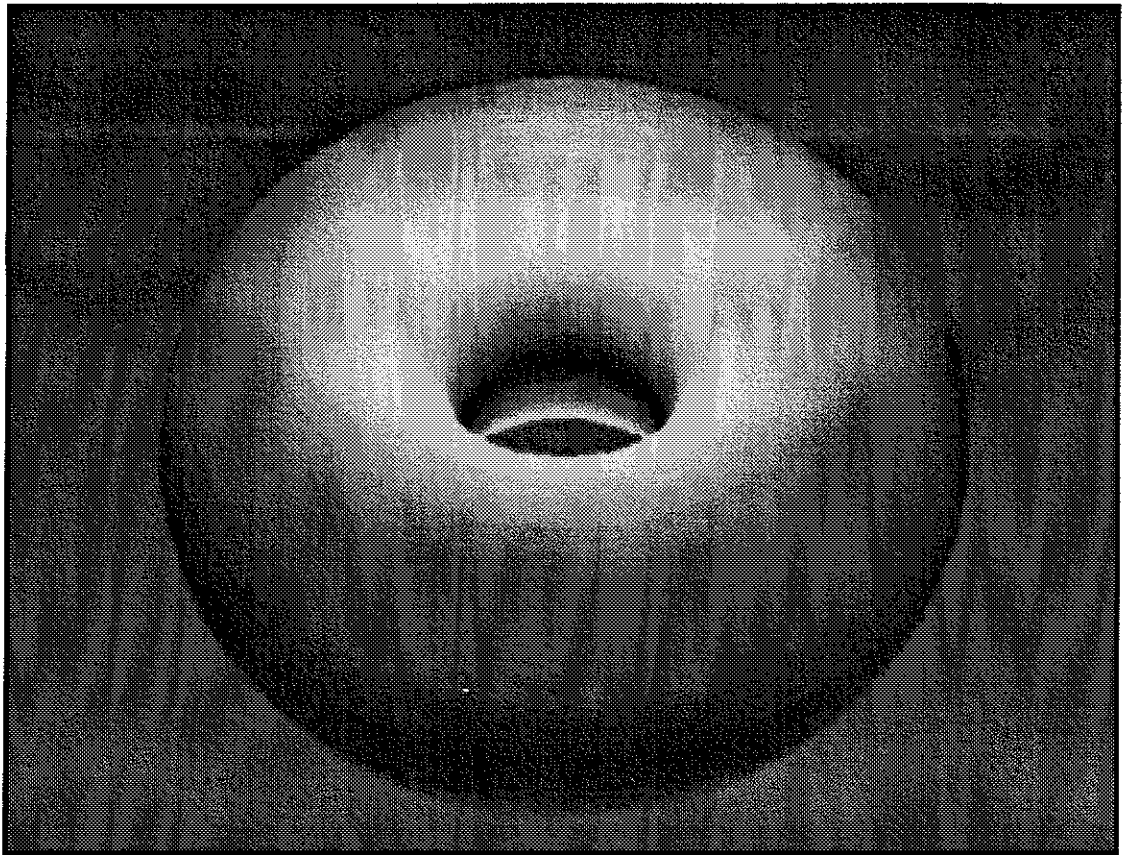
relevant vectors are coplanar and (2) the direction of projection is perpendicular to that plane.

In addition to allowing the recovery of shape from shading, the proposed method grows only linearly in execution time with number of dimensions, as contrasted with the exponential growth exhibited by the method of Kajiya [Kajiya89] and the method of Hansen mentioned in section 1.3.1.

As an example, consider a surface in 4-space with a directional light shining in the w -direction. Wherever the surface has a local maximum or minimum in w , the light vector lies within the normal space at the point and projects, without shrinking, onto that normal space. As a result, the diffuse component of illumination is large at such extreme points. Where the surface is changing rapidly in the w -direction, on the other hand, the diffuse illumination is not as strong. So knowledge of the light direction in 4-space can help the observer comprehend geometric features of a surface. [Plates 2.2 A,B]

Plates 2.2

- A** The torus $(\cos s, \sin s, \cos t, \sin t)$. The extreme values of w occur along the outer and inner walls of the torus. With the light shining in the w -direction, the diffuse highlights fall along these extremes of the torus.
- B** A Klein bottle in 4-space with the light still shining in the w -direction. The highlights again indicate where the extreme values are located on the surface.



2.3 Depth Cues

When a surface is projected from 4-space to a 2-dimensional screen, there are two orthogonal directions that become annihilated. When a user views the 2-dimensional image he must reconstruct those missing dimensions based on the scene he sees. There are a number of well-known depth cues that serve this purpose when surfaces are projected from three dimensions down to two. I will review these cues briefly and discuss how helpful they might be for reconstructing the fourth coordinate.

2.3.1 Occlusion and Shadows

Occlusion and shadows are useful depth cues for recovering z -depth when surfaces are projected from 3-space to 2-space. They are ineffective, however, when the surfaces are projected from 4-space to 2-space. By dropping down a dimension one can liken the situation to viewing 1-dimensional curves in 3-space. In both cases the manifold of interest has co-dimension 2. In general, space curves rarely occlude or cast shadows on each other: only at isolated points. Similarly, surfaces in 4-space only occlude each other or cast shadows on each other along isolated curves (in general). The result is that these cues are not especially helpful for recovering w -depth, because these cues simply are not present.

2.3.2 Texture and Color

Texture helps provide a sense of depth. When a texture is applied to a surface, the texture helps encode the distance that the surface is away from the eye. The farther away the surface is, the more the texture appears to shrink. With computer graphics, a texture can even be applied to a surface such that the texture changes dynamically in world space as the surface moves in the w -direction.

A similar approach involves a coloration that is modulated according to depth. This technique is generally known as intensity depth cueing. In 3-space there is a convenient metaphor for an intensity depth cue – the object looks as though it were obscured by fog, and the fog's color prevails as the object recedes. In practice, the 4D fog-metaphor is considerably less convincing, perhaps because the usual 3D interpretation is so much more natural.

Encoding w -depth by color is nonetheless a useful tool, especially for locating level sets according to the color they share. [Plate 2.3 A] The idea is evidently pretty obvious, since there are very old examples of its use [Hinton04]. A more modern treatment of the strategy might be to apply a dynamic texture to a surface, where the texture continually flows in the w -direction [Freeman91, vanWyjk91]. In the Fourfront system on Pixel-Planes 5, there is

not enough memory in the renderers to create such an animated texture (except at the expense of other basic functions). The next generation of the Pixel-Planes family, called Pixel Flow [Molnar92], should have ample memory to make this idea feasible.

2.3.3 Focus and Transparency

The human eye can focus at various depths. Neighborhoods of a surface that lie within the focal plane in 3-space appear crisp. Neighborhoods that are nearer or farther look increasingly blurry. There are various techniques for producing this effect during rendering [Haeberli90, Mitchell91, Potmesil82].

In 4-space one could define a focal volume at some particular distance in w . Neighborhoods within this volume would appear crisp, while neighborhoods outside would be progressively blurry. In general this is not a fast process, since blurry polygons are effectively semitransparent and hence incur some of the cost of computing transparency. But one can approximate the effect cheaply by simply modulating transparency by w -depth. If the focal volume is at the yon distance, transparency will unambiguously determine w -depth. Recall that neighborhoods near to the viewing position (eye_4 in 4-space) are generally large due to perspective, and they often enclose the faraway neighborhoods that have shrunk toward the origin.

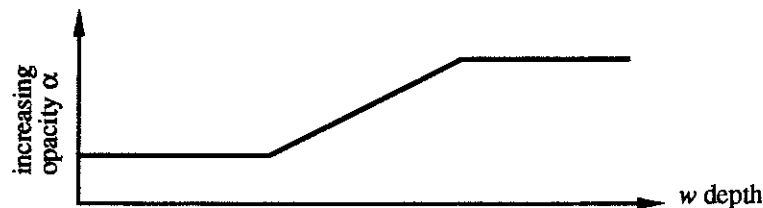


Figure 2-10. Opacity can be used as a depth cue in the w -direction without significantly interfering with the usual depth cues in the z -direction. For example, points on the surface near the eye are rendered nearly transparent, while points far away in w are nearly opaque. Fourfront uses a linear opacity ramp that is clamped at controllable near and far values.

If the nearest patches of a surface are opaque, they hide the interior geometry. This is the motivation for choosing an opaque volume at the yon, rather than the hither, distance: it is more likely to reveal the interior of a self-intersecting surface. Unfortunately, the eye does not resolve transparency with a great deal of resolution, so this technique is best applied for gross classification of relative distances in the w -direction. [Plates 2.3 B,C,D]

2.3.4 Perspective, Steropsis, and Parallax

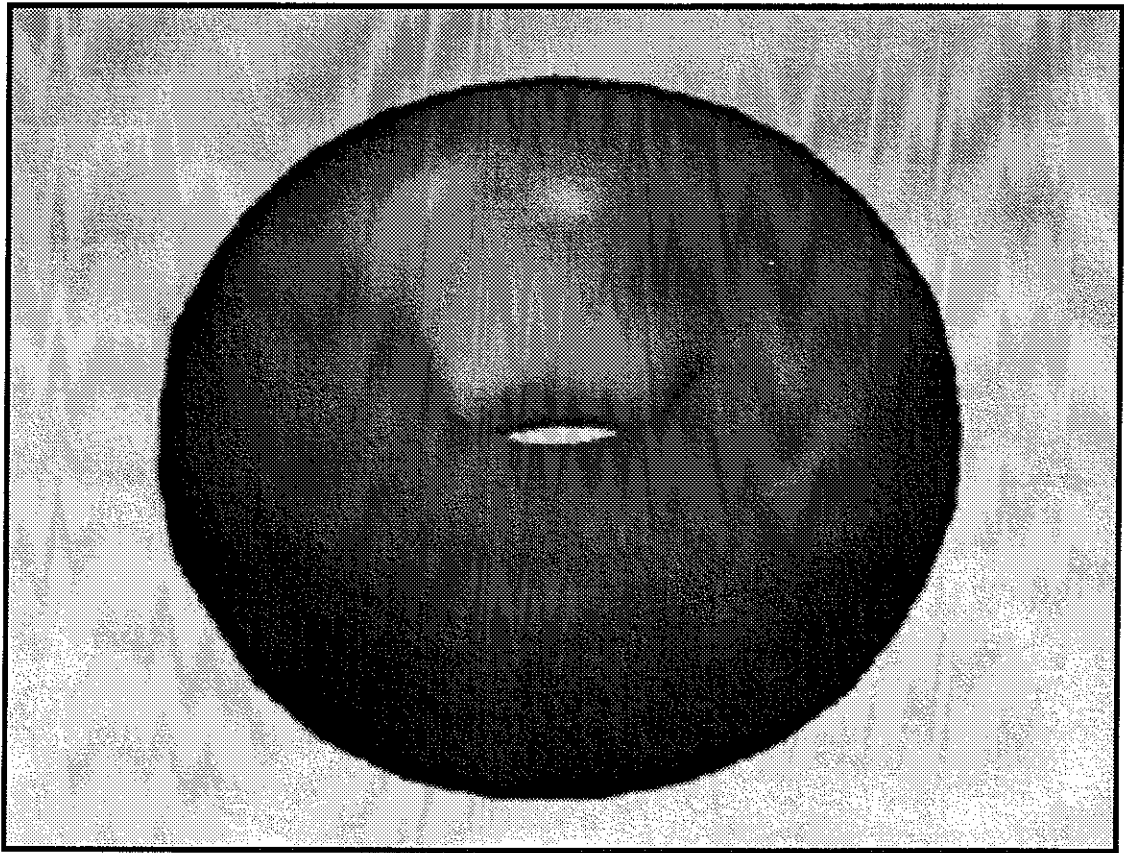
Perspective and parallax are depth cues that take advantage of the finiteness of the distance between the viewing position and the surface being viewed. When a surface patch approaches the viewing position, it subtends an increasingly larger viewing angle. As a

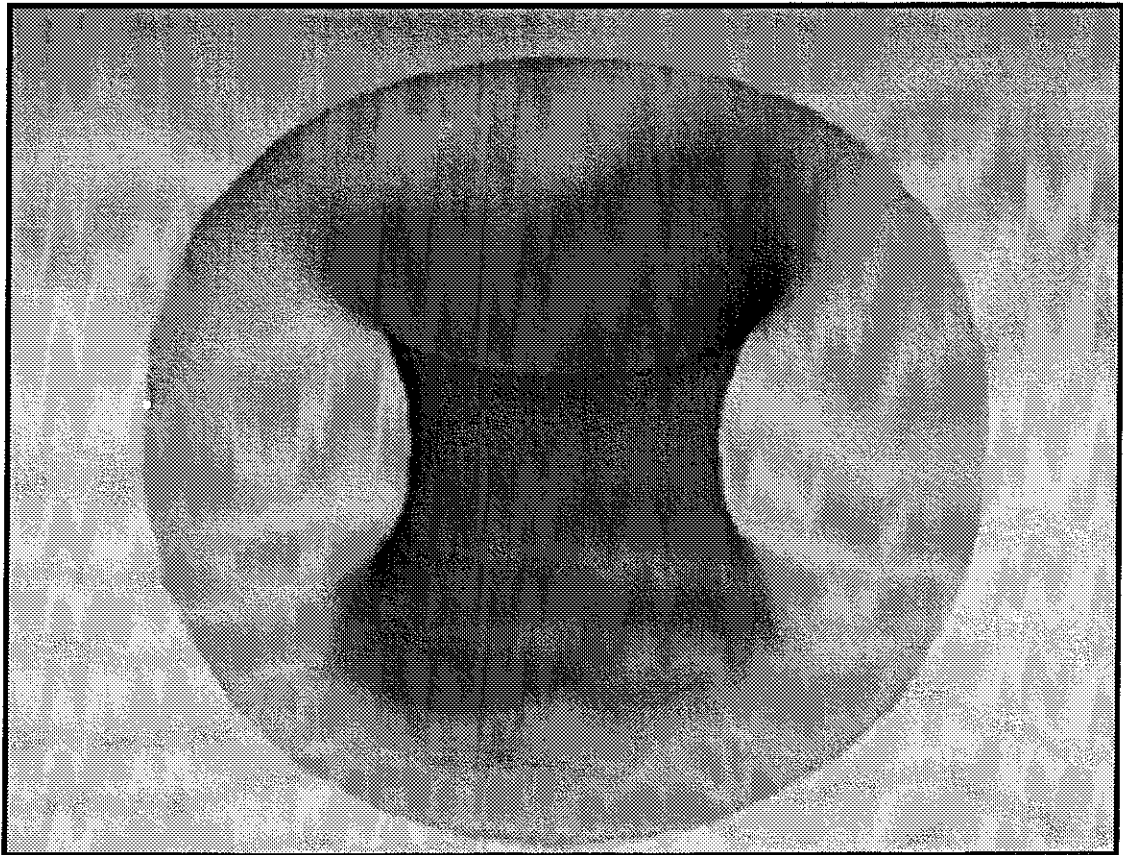
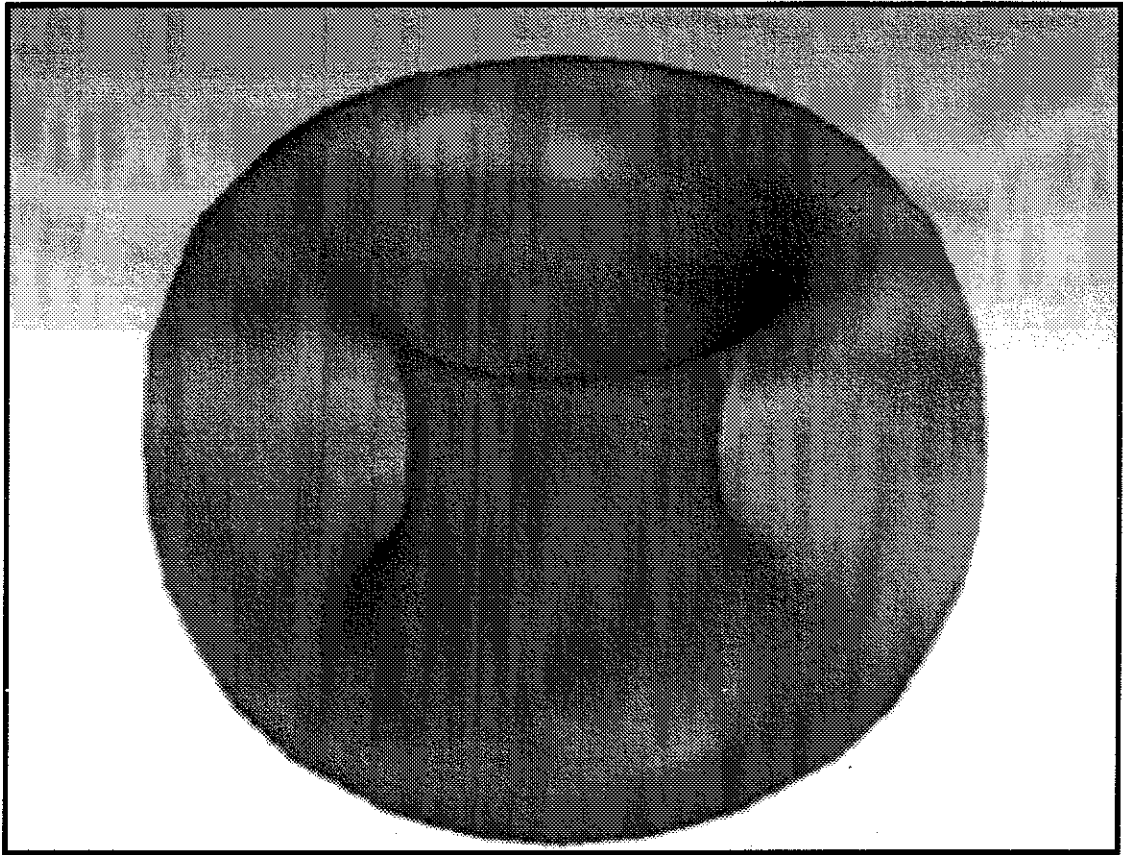
result its projected image becomes larger. When the patch (or the viewing position) moves, that motion produces a greater change in the projected image when the patch is close to the viewing position. Moving the eye in a direction parallel to the plane of projection produces parallax. Stereopsis uses parallax to produce two images from which depth can be reconstructed. These techniques are valuable aids for determining z -depth in a 2-dimensional image of a surface in 3-space. They are also effective for recovering w -depth in a 2-dimensional image of a surface in 4-space.

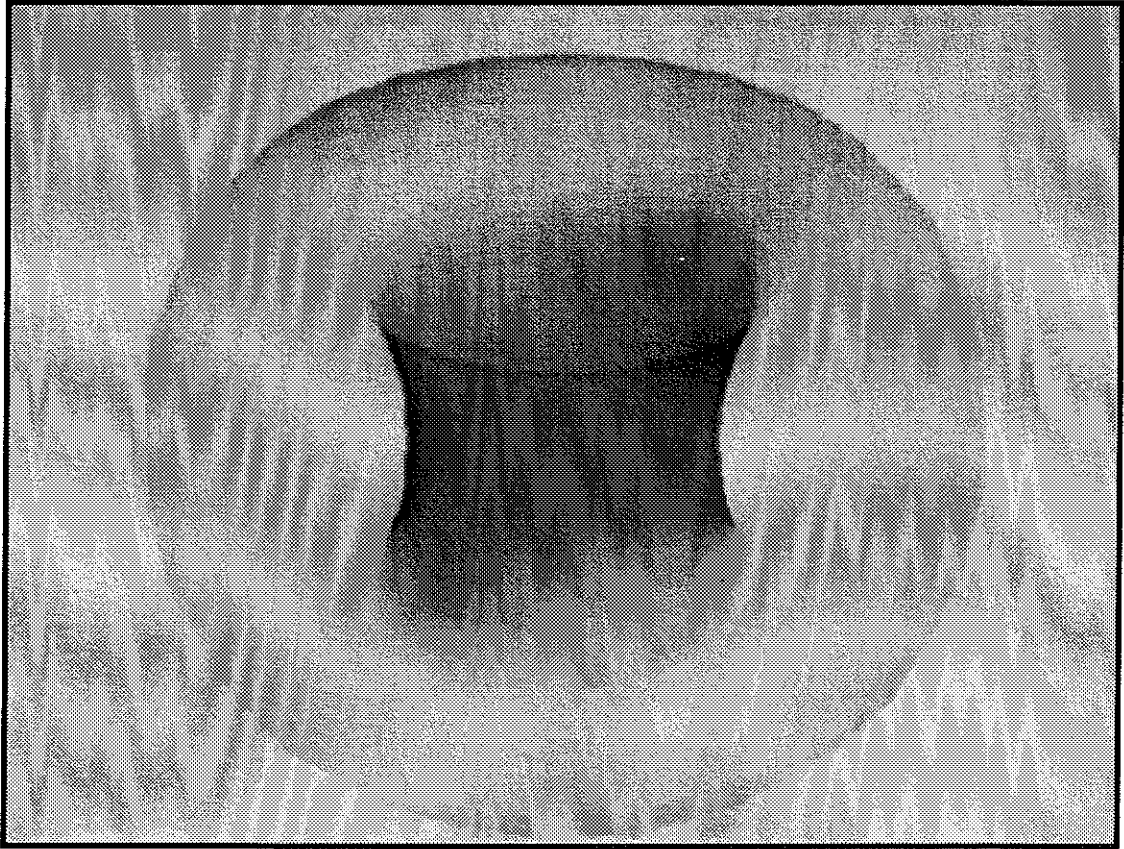
The topic of perspective and parallax is important enough to merit further discussion. That discussion is contained in the next section.

Plates 2.3

- A** The torus $(\cos s, \sin s, \cos t, \sin t)$ with color modulated from black to white according to depth in w . The outer wall of the torus, near to the eye in 4-space, is black. The inner wall, far from the eye in 4-space, is light.
- B** The torus with opacity modulated from transparent to opaque according to depth in w . In this image, the opacity ramp is centered at $w = -0.25$ (near the eye in 4-space).
- C** In this image, the opacity ramp is centered at $w = -0.0$.
- D** In this image, the opacity ramp is centered at $w = 0.25$ (far from the eye in 4-space).







2.4 Projecting from Object Space to Screen Space

The same technique for projecting surfaces from 3-space to 2-space applies to projection from 4-space to 3-space. A perspective projection requires an eye point eye_4 in 4-space. In (non-homogeneous) normalized eye-space coordinates, the point (x, y, z, w) projects to $(x/w, y/w, z/w)$ in the 3-dimensional image volume. A second eye point eye_3 within that volume determines a further projection to the final image plane (figure 2-11). Perspective gives depth cues in each of the directions z and w , so it is helpful to apply perspective in each stage of the projection. As a result, there are two centers of projection, and it is natural to perform the projection in two steps. In order to z -buffer the polygons on the screen (to eliminate hidden surfaces), the z -coordinate in 3-space must be computed at each vertex of a polygon. If instead the surfaces are w -buffered [Hoffman90], the projection can be accomplished in one step for the x and y coordinates without computing z at all. The appearance of the surface becomes non-intuitive when 3D depth no longer corresponds to occlusion. Section 2.4.2 discusses concatenated projections in more detail.

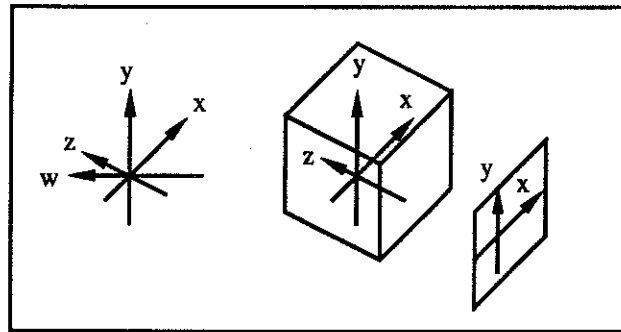


Figure 2-11. The $(x\ y\ z\ w)$ -axes (left) project in the w -direction to the $(x\ y\ z)$ -axes (middle), which project in the z -direction to the $(x\ y)$ -axes of the image plane.

The typical side-effect of projection is that the resulting surface intersects itself in 3-space, even if it has no intersection in 4-space. Why is that? The self-intersections arise when a ray from eye_4 strikes the surface twice, since both of the intersection points must map to a single point in 3-space. This is a typical situation for a closed surface in 4-space, just as it is for a closed curve in 3-space: the shadow of a “curvy” space curve exhibits self-intersections in many of its projections onto the plane.

An imbedded surface locally looks like a neighborhood in the plane – no creases, no crossings. If the surface under consideration imbeds in three dimensions, there’s little need to study it in four. Thus the interesting surfaces in 4-space are generally the ones that contain self-intersections when projected to 3-space, because they fail to imbed there.

None of the non-orientable surfaces imbeds in 3-space. Happily, they can each be imbedded in 4-space.

Typically a surface that is transformed and rotated on a graphics machine is the boundary of a solid object, whether the object be a house or a mountain range. Such a surface may be geometrically complex, but it performs a crucial topological service: it separates 3-space into an inside and an outside. One can tour the surface from the inside (as with a building walkthrough) or from the outside (as with a flight simulation over rugged earth) until a sufficiently complete mental model of it develops. One need not cross the surface to the other side in order to complete the tour of a single side. The tour can remain confined to the inside or the outside.

By contrast, a self-intersecting surface separates 3-space into any number of subsets. If the surface is opaque, some or most of its pieces remain hidden during a tour of a particular volume that it bounds. Rotating the surface in 4-space may reveal a patch of surface that was previously hidden, but only at the expense of another portion of the surface that is now obscured. The fundamental problem of displaying such surfaces is that they continually hide portions of their geometry. Three popular ways to tackle this problem are to use ribboning, clipping, and transparency. Overall, transparency is the most helpful, but it has certain drawbacks that are discussed and repaired in chapter 3.

2.4.1 Stereopsis and Parallax

Parallax and stereopsis are side-effects of perspective projection, and they offer additional w -depth cueing [Armstrong85]. Consider the effect of translating the eye. Objects at various depths in the world change their relative positions when the eye shifts in the x or y directions. But which eye position (eye_4 or eye_3), and which depth (z or w)?

A situation regarding projecting 3-space to 1-space, analagous to the one regarding projecting 4-space to 2-space, is as follows. There is some viewpoint eye_3 in 3-space, and there is some image plane to which the world projects (figure 2-12). Within that plane there is a second viewpoint eye_2 and an image line to which the scene projects further. Two spheres A and B in the 3D world project to two disks A' and B' in the image plane, and then to two segments A'' and B'' in the image line. Suppose A'' and B'' are only slightly separated. If eye_2 shifts to the right and A'' shifts to the right relative to B'' one concludes that A' is farther away than B'. But that does not imply that the source object A is farther from eye_3 than B. It can be the case that shifting eye_3 to the right causes A'' to shift left instead (relative to B''). Translating eye_3 and eye_2 together couple these behaviors. The situation in 4-space is the same. One has a choice of where to apply a translation. Applying it before the projection from 4-space to 3-space produces nonintuitive motion, due to the parallax from the w direction: the projected object is no longer rigid under the expected isometries, although the source object, of course, still is.

The illustration at the end of this section shows the effect of translating a torus in 4-space before projecting it to 3-space. The translation seems not to be rigid, because the result after projection to 3-space is that the inner core of the torus (far away in w) translates less than the outer part of the torus (nearby in w) due to perspective. [Plate 2.4 A]

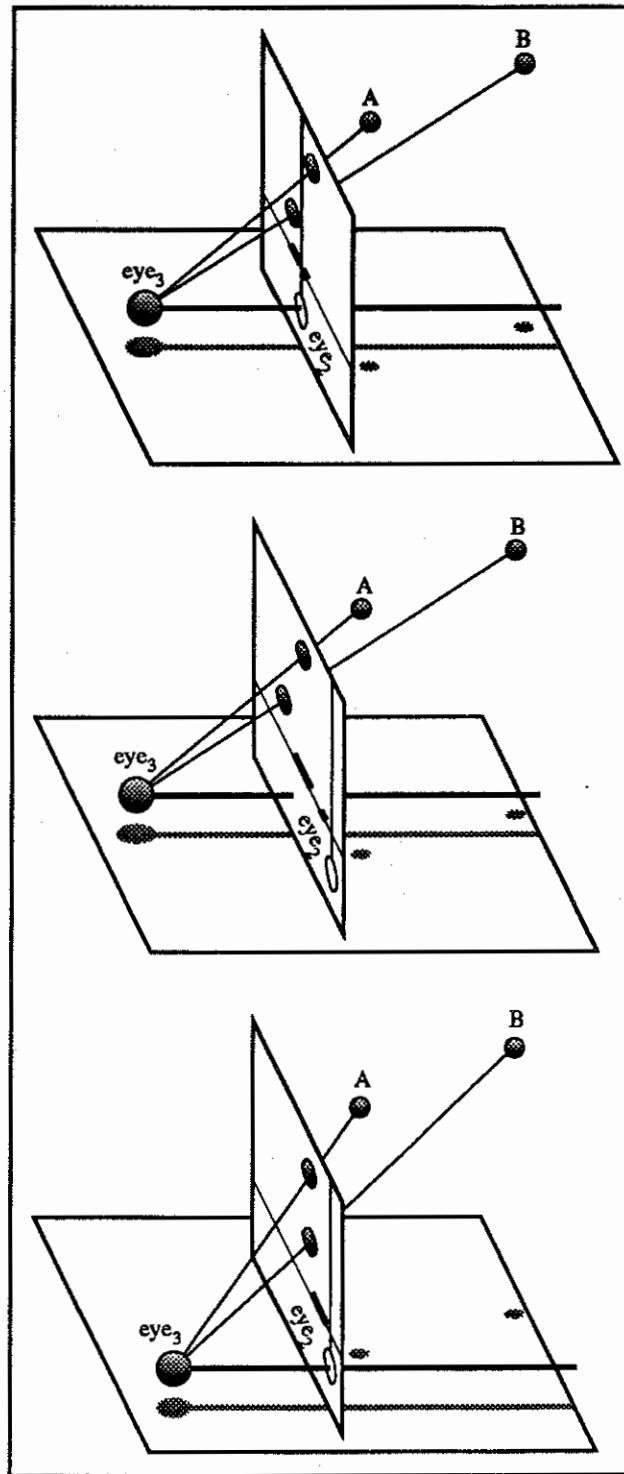


Figure 2-12. When there are two eye positions involved in projecting an image, either of them can produce parallax. In this figure, spheres A and B project from 3-space onto a 2-dimensional plane as disks. The disks project to a 1-dimensional line as segments. By tilting the page obliquely, you can see what the second eye sees. Moving an eye to the right will make the farther object seem to move to the right of the nearer object. Which sphere looks closer? It depends on which eye does the measuring. A is closer to eye₃ than B is. But the projection of B is closer to eye₂ than the projection of A is.

2.4.2 Mixing Rotations, Projections, And Translations

The user of the Foughront system can interactively control the rotations, projections, and translations of a surface in 4-space. This section describes what happens when these transformations are combined together.

Let R denote a rotation within the 3-dimensional subspace spanned by the x , y , and z axes. Let P_4 denote the perspective projection from 4-space to 3-space defined in the canonical way: the origin is the center of projection. The two operations commute, which means that any rotation performed after the projection can be incorporated into the original rotation matrix. This is shown below. In the first case, rotate and then project.

$$\left[\begin{array}{ccc|c} R_{ij} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix} \xrightarrow{P_4} \frac{1}{w} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

In the second case, project and then rotate.

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \xrightarrow{P_4} \frac{1}{w} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \xrightarrow{R_3} \frac{1}{w} \begin{bmatrix} R_{ij} \\ \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \frac{1}{w} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

The results are the same, and so the operators commute.

$$R_3 P_4 \mathbf{v} = P_4 R_3 \mathbf{v}$$

On the other hand, translation does not commute with projection. As an example, consider a translation in the x -direction. In the first case, perform the translation followed by the projection.

$$\begin{pmatrix} x + \Delta x \\ y \\ z \\ w \end{pmatrix} \xrightarrow{P_4} \begin{pmatrix} x/w + \Delta x/w \\ y/w \\ z/w \end{pmatrix}$$

In the second case, project and then translate.

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \xrightarrow{P_4} \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \xrightarrow{+\Delta x} \begin{pmatrix} x/w + \Delta x \\ y/w \\ z/w \end{pmatrix}$$

The results are different whenever Δx is nonzero.

The user might want to perform some rigid motion on the surface before or after each of the two projections: in 4-space, 3-space, and again in 2-space. Since an affine transformation can be written as a rotation and a translation, the user could potentially demand six different transformations in order to completely control the surface and its relation to the two eye points. That demand is unnecessary. As the above calculations show, the rotation components of an affine transformation commute with the projection. Consequently it is sufficient to apply a single rotation matrix in 4-space, followed by translations in 4-space, 3-space, and 2-space.

Next consider the result of two perspective projections. If the origin of each space is the center of projection (the "canonical" case), cancellation occurs.

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \xrightarrow{P_4} \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \xrightarrow{P_3} \begin{pmatrix} \frac{x/w}{z/w} \\ \frac{y/w}{z/w} \end{pmatrix} = \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \end{pmatrix}$$

The result is exactly the same as if there had been no w -coordinate at all! That is, the result is the same as using orthographic projection in the first step. To retain the perspective effect of the first projection, it is enough simply to offset the eye-point from the origin in the second stage of the projection. For example, translate the second center of projection to $(0, 0, z_0)$ and project down the z -axis as before.

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \xrightarrow{P_4} \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \xrightarrow{P_3} \begin{pmatrix} \frac{x/w}{z/w - z_0} \\ \frac{y/w}{z/w - z_0} \end{pmatrix} \neq \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \end{pmatrix}$$

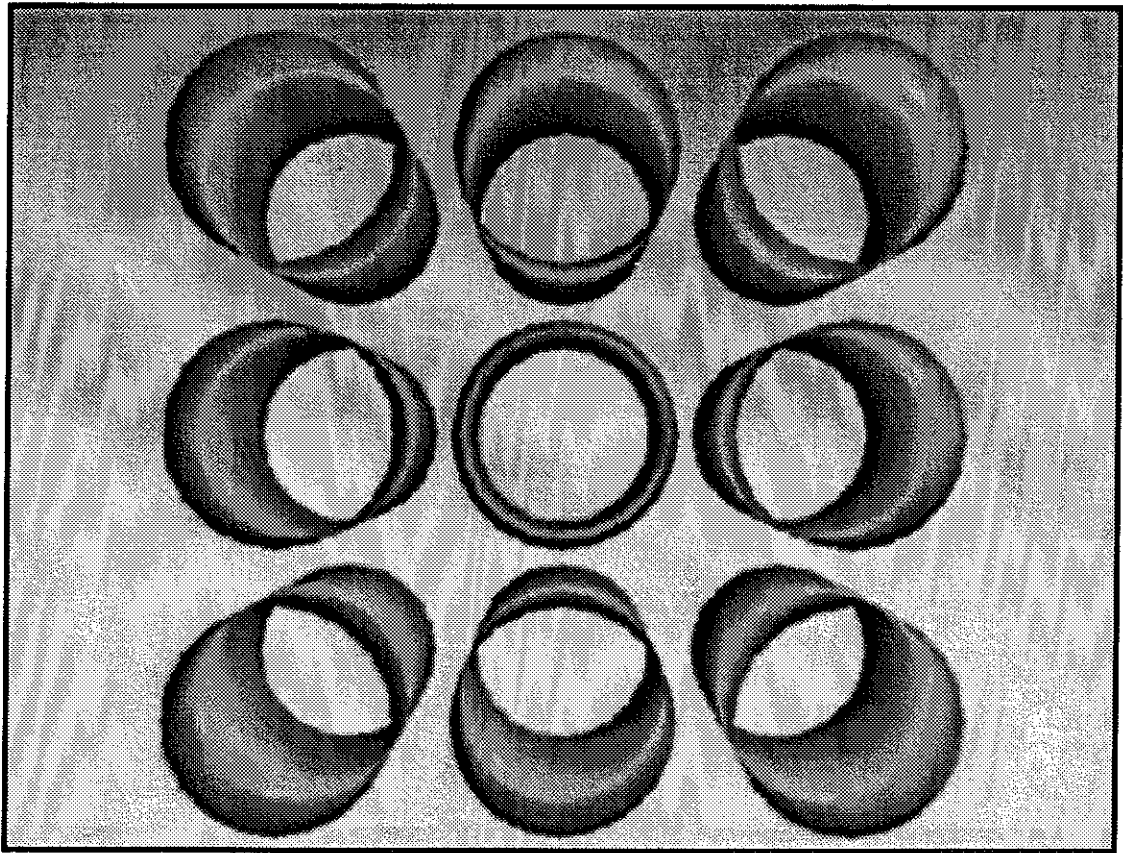
If only the first eye-point is translated, say to $(0, 0, 0, w_0)$, cancellation occurs as before, and the perspective effect disappears.

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \xrightarrow{P_4} \begin{pmatrix} x / (w - w_0) \\ y / (w - w_0) \\ z / (w - w_0) \end{pmatrix} \xrightarrow{P_3} \begin{pmatrix} \frac{x / (w - w_0)}{z / (w - w_0)} \\ \frac{y / (w - w_0)}{z / (w - w_0)} \\ \frac{z / (w - w_0)}{z / (w - w_0)} \end{pmatrix} = \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \\ 1 \end{pmatrix}$$

It can be concluded that in order to achieve a 4-dimensional perspective effect, it is necessary and sufficient to offset the position of the second eye point from the origin of the 3D projection volume in which it lies.

Plate 2.4

- A** The torus, translated in both the **x**- and **y**-directions. The effect of the translation is more exaggerated on the outer wall of the torus than on the inner wall. That is because the outer wall is closer to the eye in 4-space where perspective amplifies the motion. As a result, the outer wall of the torus intersects the innerwall in the projection to 3-space.



3 Visualization Tools for Surfaces in 4-space

There are several ways to enhance the appearance of a surface in 4-space by using computational techniques. This section describes some of the techniques I apply in the Fourphront system in order to help the user better visualize the shape of the surface.

3.1 Ribboning

To reveal the geometry of a self-intersecting surface, one can slice it into ribbons [Banchoff86]. The gaps between ribbons reveal parts of the object that would otherwise be obscured. One advantage of ribboning is that it can be performed once, at model definition time, and then left alone. Some of the drawbacks are that (1) any already-existing non-ribboned datasets must be remeshed and ribboned, (2) the high-frequency edges of thin, closely-spaced ribbons attract the attention of the eye at the expense of the geometric content of the surface, and (3) ribbons can produce distracting moiré patterns when they overlap.

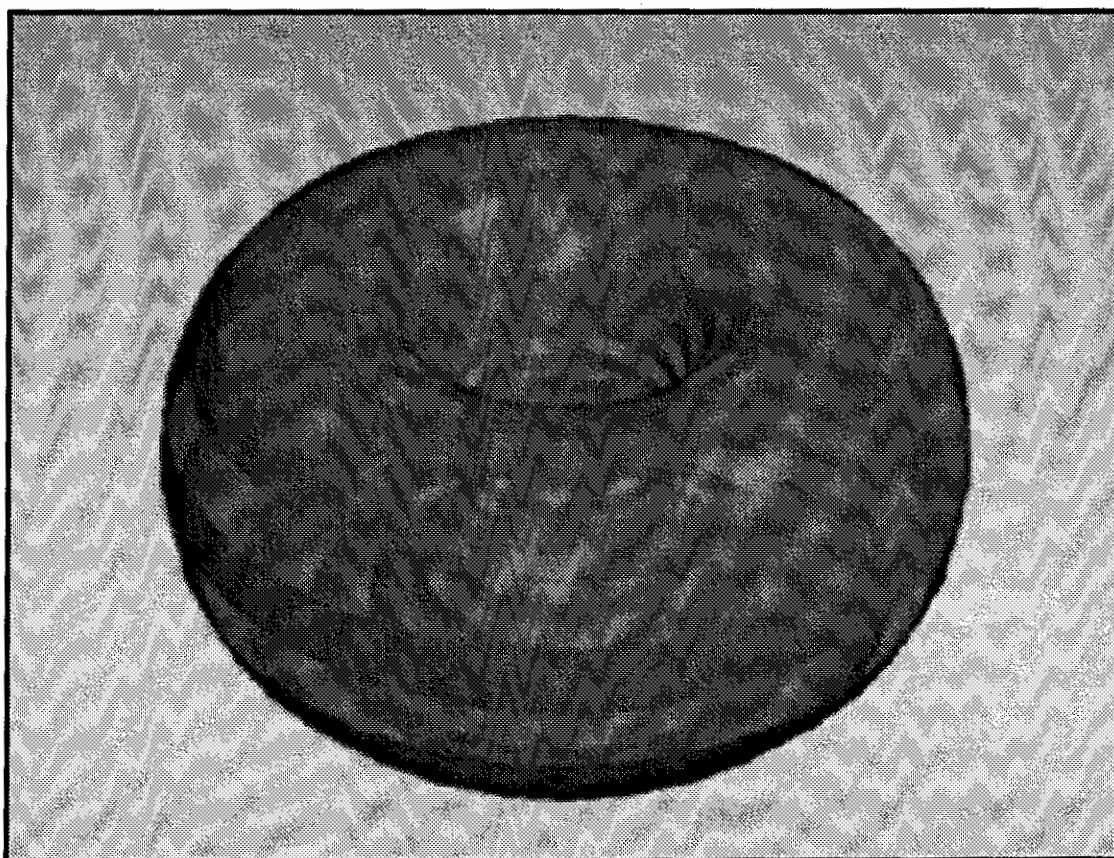
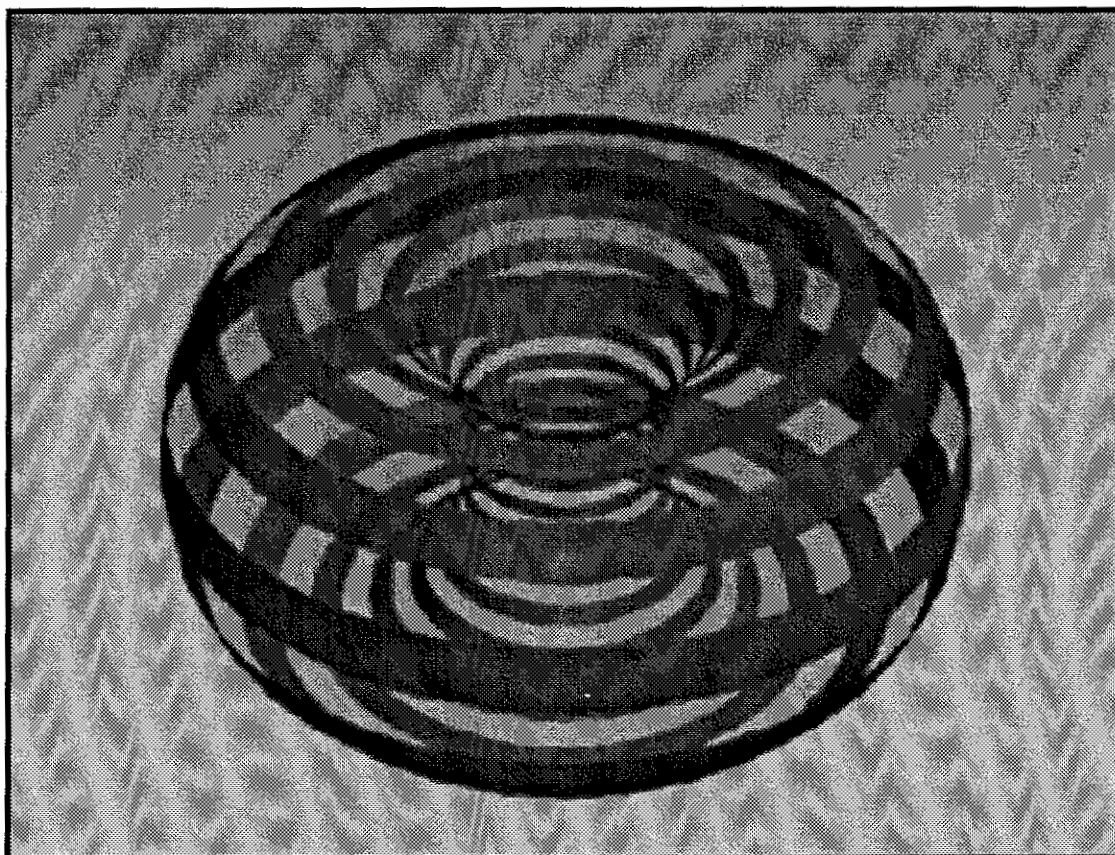
These drawbacks do not mean that ribboning is a clumsy technique. On the contrary, for surfaces that can be foliated by 1-dimensional curves, ribboning is a very elegant means of visualization. The compact surfaces that admit such a foliation are the torus and the Klein bottle. In order to reduce the visual clutter caused by the interference of many ribbons, the intervening parts of the surface can be made semitransparent. This combination preserves the connected appearance of the surface while simultaneously revealing the interior shape of the surface. [Plates 3.1 A, B]

Banchoff has made productive use of ribboning to illustrate the foliation of the 3-sphere in 4-space. The 3-sphere can itself be foliated by tori (together with two degenerate circles). The tori themselves can be foliated by circles, with a single ribbon representing a set of these circles. Banchoff animated a ribboned torus that itself follows a foliating trajectory through the 3-sphere.

Surfaces with other topologies do not admit such a simple ribboning. One can slice a surface along level cuts as it sits in 4-space, but the cuts will sometimes produce x-shaped neighborhoods and degenerate points in the ribbons.

Plates 3.1

- A** A torus sectioned into ribbons.
- B** A torus sectioned into opaque ribbons and semitransparent gaps in between them.



3.2 Clipping

Rather than pre-compute sections of the surface to be sliced away, one can clip them out dynamically. The chief advantages of clipping are that (1) many graphics machines implement fast hither-clipping as part of their rendering pipeline; (2) the surface does not need to be re-modeled in order to be clipped (recall that a surface does need to be re-modeled in order to be represented by ribbons); and (3) by clipping the surface as it moves, the user can inspect views of it that a single static segmentation (like ribboning) cannot anticipate.

There are drawbacks to clipping. One usually thinks of clipping a surface against a plane. However, clipping is properly a geometric intersection of a surface against a 3-dimensional volume whose boundary is the clipping plane. In 4-space a plane does not bound a volume, just as a line does not bound an area in 3-space. So it is a 4-dimensional half-space that clips the surface, and the boundary of that halfspace is a 3-dimensional flat, or hyperplane. It is true that a user could interactively specify the position and orientation of the 4D halfspace that does the clipping, just as he can control the position and orientation of the surface under scrutiny. But consider the problem of providing visual feedback to show where that clipping volume is. The shape of the clipped surface implicitly defines where the boundary of the clipping volume is. In 3-space one can mentally reconstruct the orientation of that volume from the clipped edges it leaves behind. It is much harder to reconstruct the orientation of a clipping volume in 4-space based on the shape of the region it clips away. One might indicate the orientation of the 4D clipping halfspace by volume-rendering its boundary. Unfortunately, that boundary will tend to hide the surface that remains after clipping.

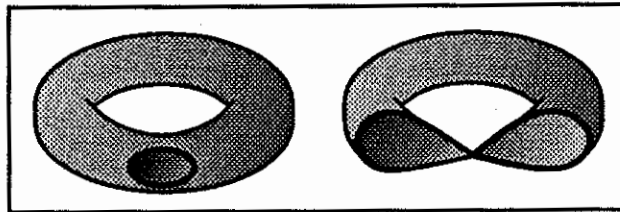


Figure 3-1. Clipping into a torus produces a figure-eight contour. Clipping reveals internal geometry, but complex contours can confuse the shape.

Recall that the incessant problem in viewing a complicated surface in 4-space is that its projection to 3-space exhibits self-intersections. One would like to see the component pieces of a self-intersecting surface as they penetrate each other, to see beyond the patch of surface that hides another patch behind it ("behind" being in the z -direction of the 3-dimensional space to which the surface has been projected). One can address this problem

by clipping in that 3-dimensional space, and clipping strictly in the z-direction. This amounts to hither-clipping. In short, clipping in 4-space is mathematically easy but interactively hard. For the purpose of revealing hidden interiors, however, hither clipping suffices. In the Fourphront system I therefore chose to restrict clipping to the 3-dimensional subspace that the surface is first projected to.

Hither clipping has other problems. The shape of the surface region that gets clipped away can be very complex. A simple shape is one that is topologically equivalent (homeomorphic) to a disk. In general it is easier to make sense of surfaces whose clipped regions have simple shapes rather than complex shapes [Francis87], but intersections and saddle points on a surface cause the clipped regions to look complex (figure 3-1). Secondly, a clipping plane cuts into a concave region of a surface only by cutting into the neighboring regions as well. This is not necessarily the effect a user wants to achieve. Imagine a sphere that has been dented on the front side with the dent bulging inward. As the clipping plane passes into the sphere, it first clips away an annulus around the rim of the bulge. The annulus expands in width until it becomes a disk. Presumably one would like to be able to clip just a small neighborhood of the bulge's center, but that is only possible if the dent bulges outward, not inward. Both of these shortcomings can be remedied by using more exotic, custom-shaped clipping volumes, but defining these volumes interactively then becomes hard. Thirdly, clipping the frontmost patches of a surface exposes some of the hindmost patches, which may be behind the center of rotation for the surface. The visible part of the surface then seems to rotate in the direction antisympathetic to the input motion. This could be considered a depth-cue, but it becomes a nuisance once the user is accustomed to steering the nearest visible part of a surface in the same direction as the input device.

Clipping then can be a useful tool for examining a surface in 4-space, because it reveals portions of the surface that are obscured. Unfortunately, clipping can also be awkward to use interactively.

3.3 Transparency

Ribboning and clipping simulate transparency via a binary classification. Both techniques classify parts of the surface as completely opaque and the other parts as completely transparent. One can use gradations of transparency as well. Ideally a semi-transparent surface presents all of its self-intersecting components on the screen so that the shape of each layer is discernible. The frontmost patch contributes more to the rendered image than the patches behind it do, but the patches behind it are visible as well. In practice the effect is dramatic and helpful for many surfaces. But there are several things that can hinder the usefulness of transparency.

Disappearing intersections

The intersection of two opaque surface patches A and B is readily apparent whenever their colors differ. On one side of the intersection, patch A lies atop B (yielding A's color); on the other side B lies atop A (yielding B's color). As the patches become simultaneously more transparent, their colors blend and the intersection becomes less distinguishable. On either side of the intersection, the semi-transparent surface has a color that interpolates between A's color and B's color. Intersection curves figure prominently in the study of nonimbedded surfaces, so it is unfortunate that transparency makes them less visible.

[Plates 3.3 A-C]

Disappearing silhouettes

A surface with many self-intersections may require a great deal of transparency to make the deep layers visible. In general, the surface exhibits more depth complexity in the interior of the 2D projection than along the silhouette. Along the silhouette, a ray emanating from the eye intersects the surface only once. But such a ray may pierce several layers of the surface as its first point of contact moves away from the silhouette. As a result, the outermost layer becomes nearly invisible because it is so transparent. In particular, it becomes difficult to see the outline, or silhouette, of a very transparent surface because the silhouette includes the rim of the nearly-invisible outermost layer. The outline carries a great deal of shape information [Bruss89], but transparency diminishes it. [plates 3.3 E-G]

Reduced performance

Rotations in 4-space change the geometry of a surface's 3D projection. Polygons that were disjoint one frame ago now interpenetrate. Polygons that were on the outermost side trade places with polygons on the innermost side. Opaque polygons can be rendered in any order, so long as only the nearest polygons (in screen depth) survive the rendering process. On the other hand, transparent polygons can be rendered from back to front or from front to back,

but in any case they must be rendered in sorted order. The dynamic 3D geometry caused by 4-space rotations prohibits ordering the model by a static data structure in 3-space, such as a binary space partition (BSP) tree [Fuchs83]. A polygon partitions 3-space by the plane in which it lies, but a plane does not separate 4-space. It is an open problem to devise a linear-time algorithm that traverses the polygons from front to back after they have been projected to 3-space. [plates 3.3 I, J]

In short, to render transparent polygons one must be prepared to sort them dynamically, perhaps even splitting them to eliminate interpenetrations. But that is computationally expensive and hence slow.

Loss of 3D depth cue

It is true that an opaque self-intersecting surface hides parts of itself that the user wants to see, but that opacity serves a positive purpose: to disambiguate 3D depth on a 2D display. Occlusion is a powerful depth cue. A hidden polygon is obviously farther away than the visible polygon atop it. Transparency reduces or eliminates this depth cue, leaving the viewer to rely on other cues to recover 3D depth. One especially helpful cue is specular reflection.

Specular highlights reveal surface geometry in two ways. The shape of a surface is easy to see along its silhouette, but it is not so apparent in the neighborhoods that are viewed head-on. Phong highlights help exaggerate the curvature, thereby distinguishing the shape of a neighborhood. Where two translucent surface patches interpenetrate, the Phong highlights can disambiguate which surface is in front, especially when the user rocks the surface back and forth. Moreover, the highlights can disambiguate the different layers that transparency reveals. The benefit diminishes, of course, as the number of transparent layers increases, but the effect is appreciable through three or four layers.

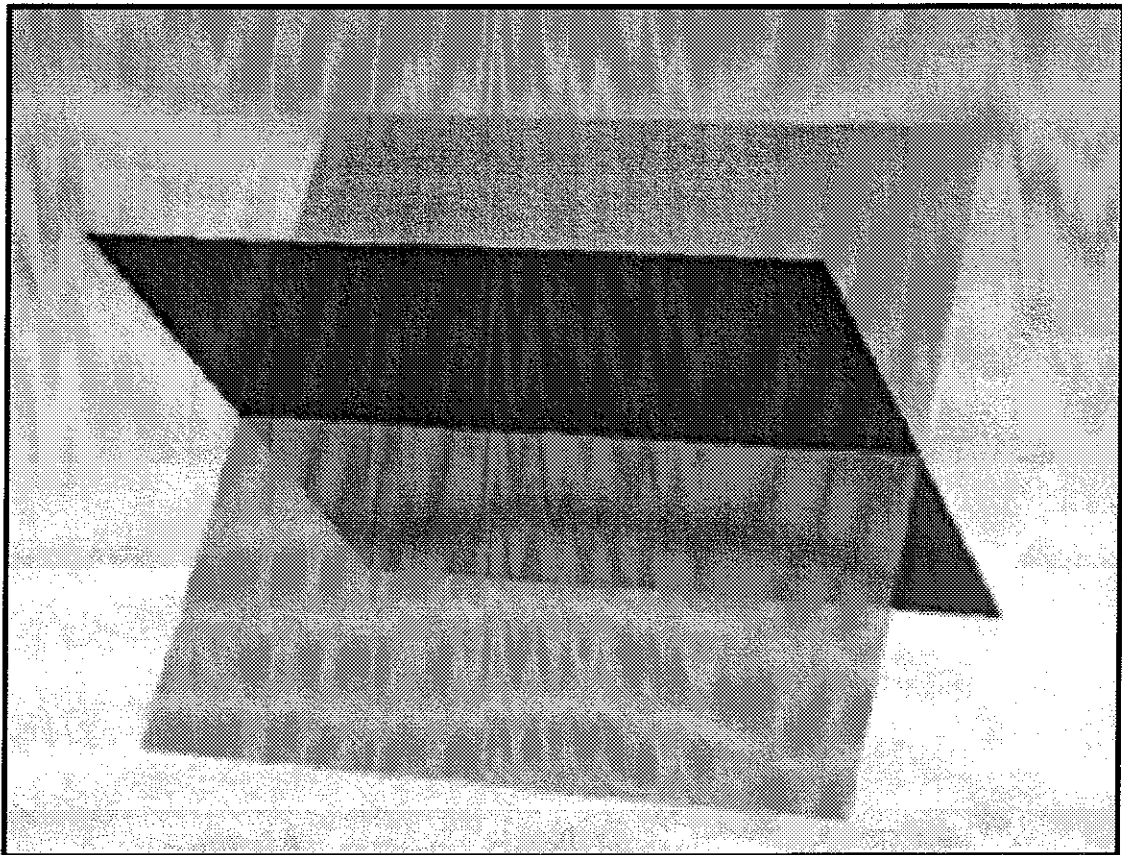
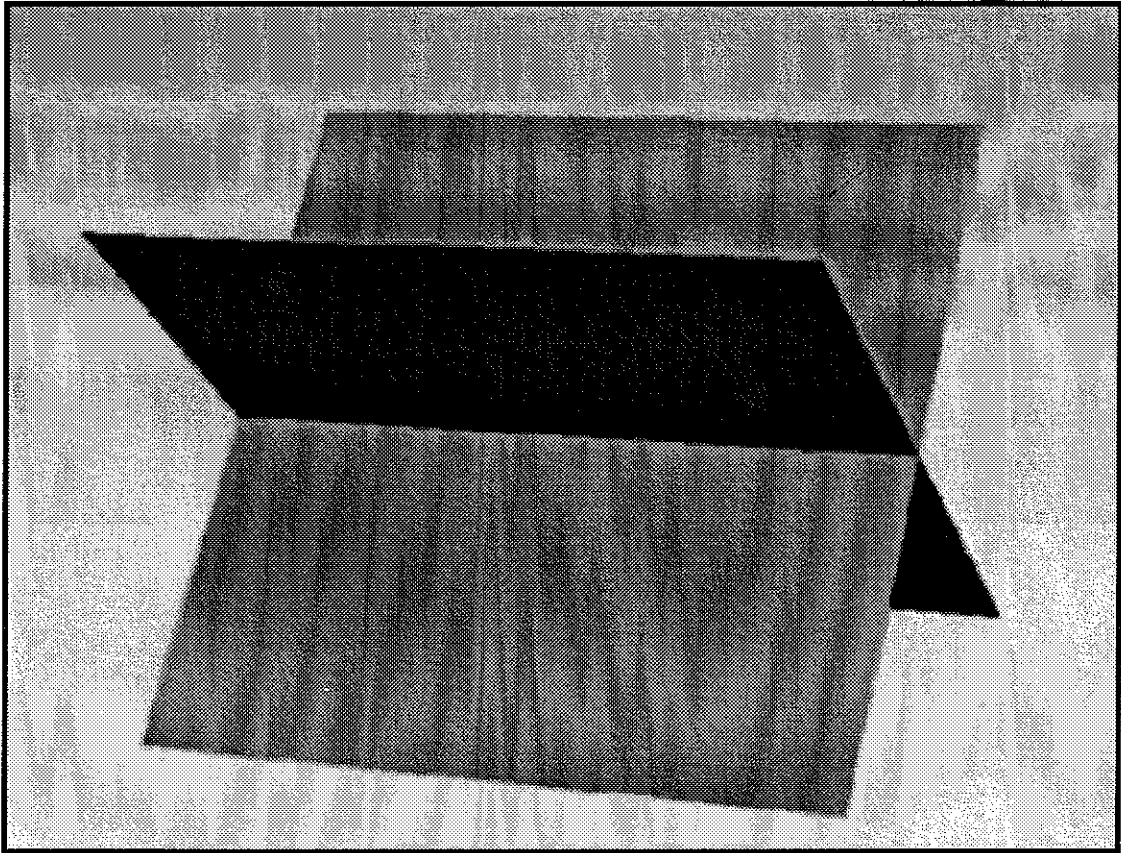
Transparency is an essential tool for studying surfaces in 4-space, since it reveals the behavior of the patches that intersect each other. An "arbitrary" surface in 4-space is likely to exhibit self-intersections when it is projected to 3-space, so interpenetration is a persistent impediment to visualizing surfaces when they are opaque. But transparency comes with a price. It subdues intersections and silhouettes. It makes rendering slower. It makes depth more ambiguous.

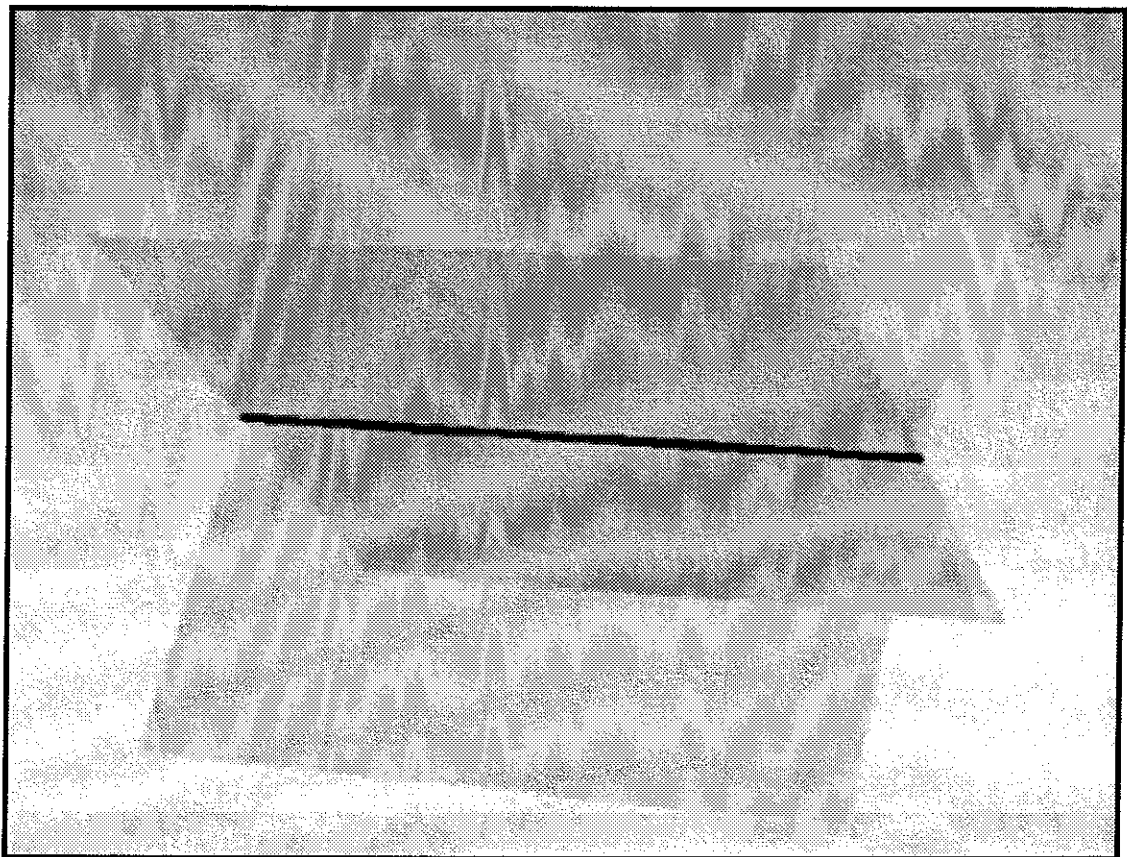
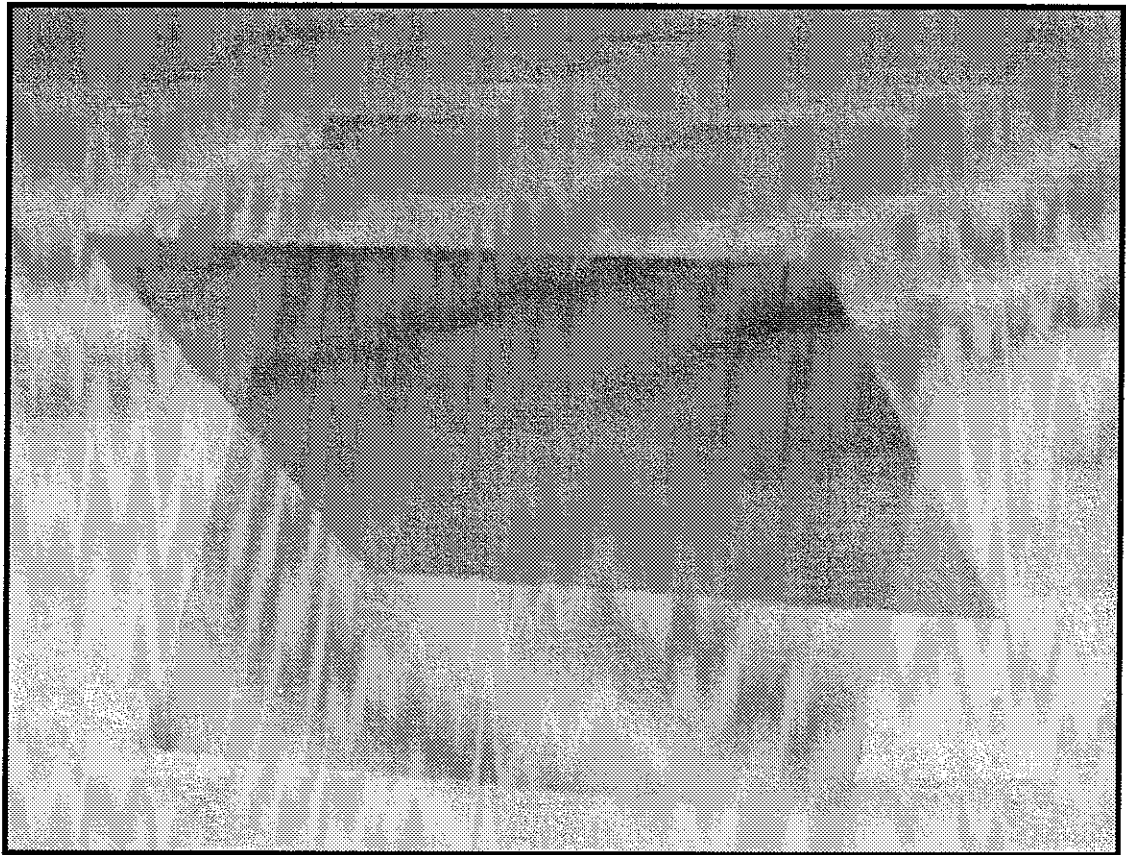
In order to redeem transparency as a tool for rendering surfaces in 4-space, one can address these demerits in the following ways. (1) Highlight the intersection curves [Plate 3.3 D]; (2) Highlight the silhouette curves [plate 3.3 H]; (3) Order the polygons in sub-linear time [Plate 3.3 J]. (4) Apply Phong shading to recover some sense of 3D depth.

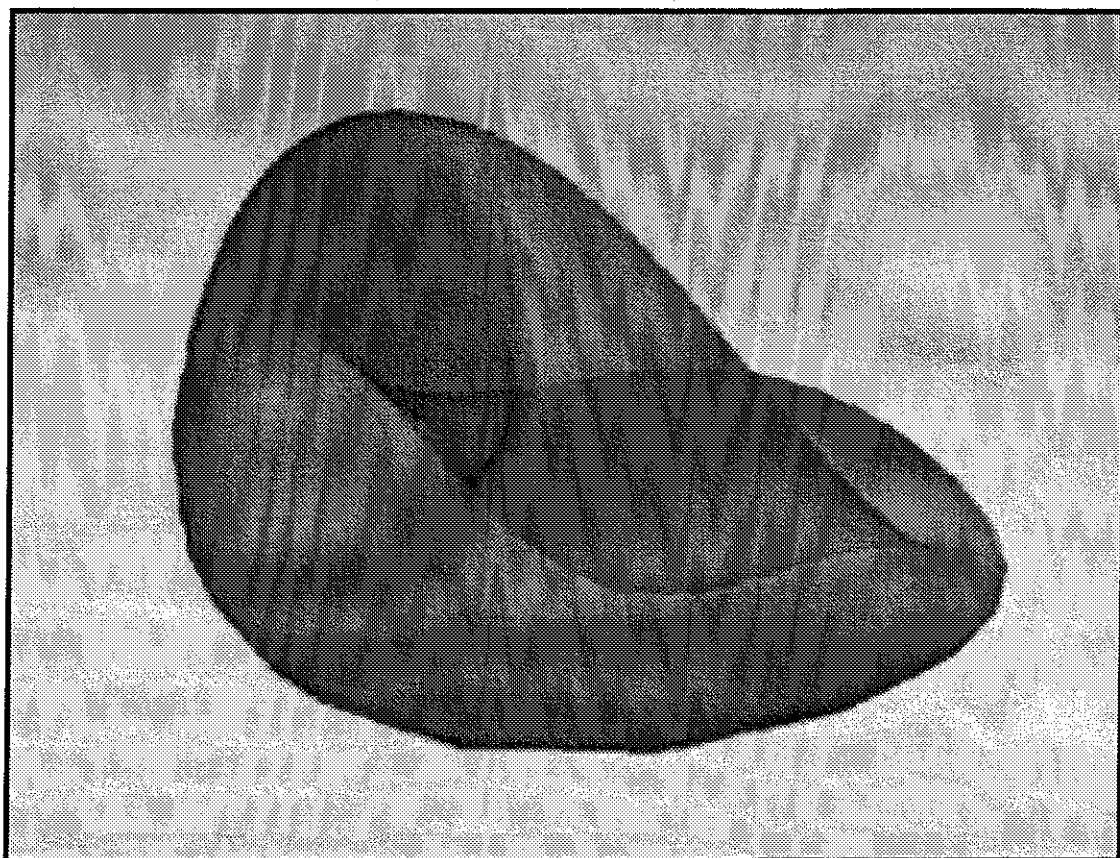
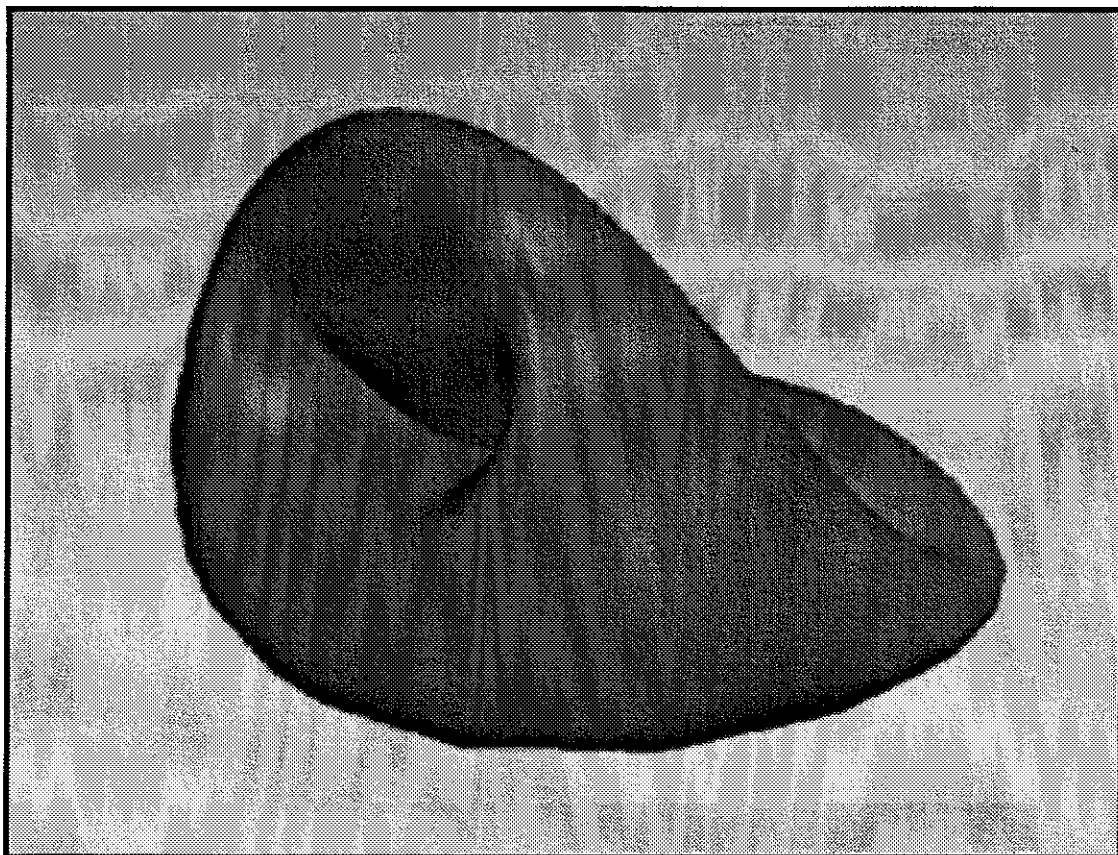
Finding the intersections and silhouettes could be slow, and these curves will often change with every frame. In the next two sections I discuss techniques for computing these special curves after the second projection, from 3-space to the screen. The algorithms exploit the logic-enhanced memory on board Pixel-Planes 5. Fourphront uses these techniques in the presence of transparency and Phong shading by taking advantage of the underlying implementation of multipass transparency on Pixel-Planes. In (back-to-front) multipass transparency the geometric model is sent to the SIMD renderers multiple times. Each pixel has variables `zBack` and `zFar` that are initialized (respectively) to the hither and yon distances at begin-frame time. During a single pass a pixel retains the geometry of the back-most polygon (compared against the depth in `zBack`) that is nearer than `zFar`. At the end of the pass the shaded polygon is blended into a temporary frame buffer, `zFar` assumes the value in `zBack`, and `zBack` is re-initialized to the hither distance. Each pass then extracts the back-most polygon that has not been shaded yet and then shades and blends it. The procedure quits when either (1) a renderer finds no more polygons farther than `zBack`, or (2) the number of passes exceeds a pre-defined maximum. For n polygons and p passes, the algorithm thus uses $O(np)$ time. For surfaces with modest complexity (such as the ones illustrated in this thesis), I have found that 12 passes are an adequate pre-defined maximum.

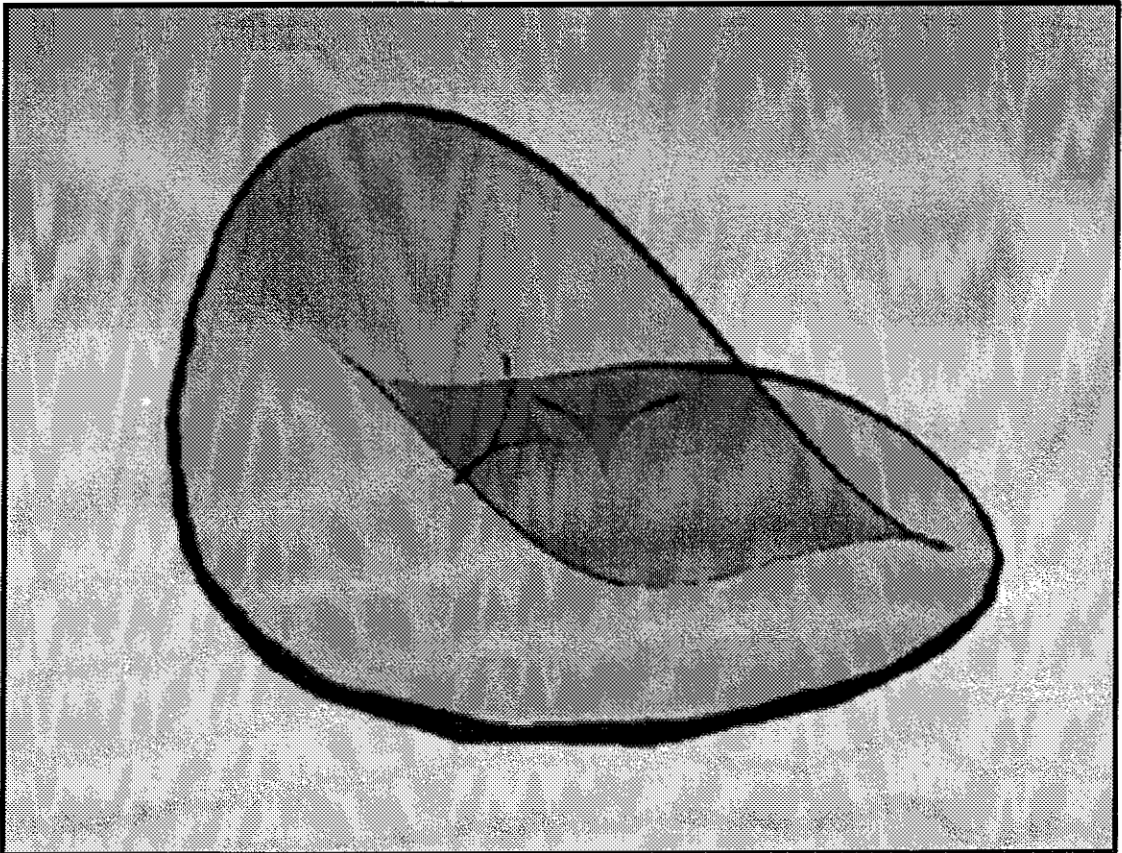
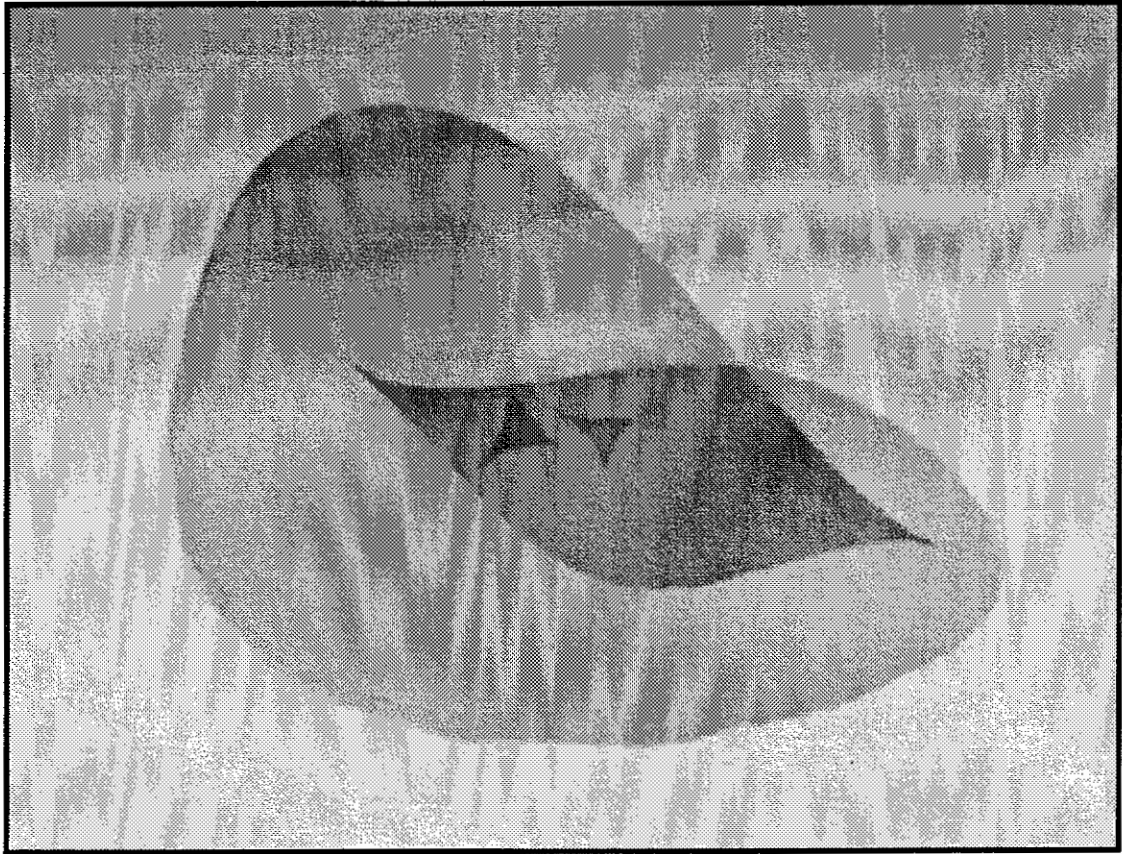
Plates 3.3

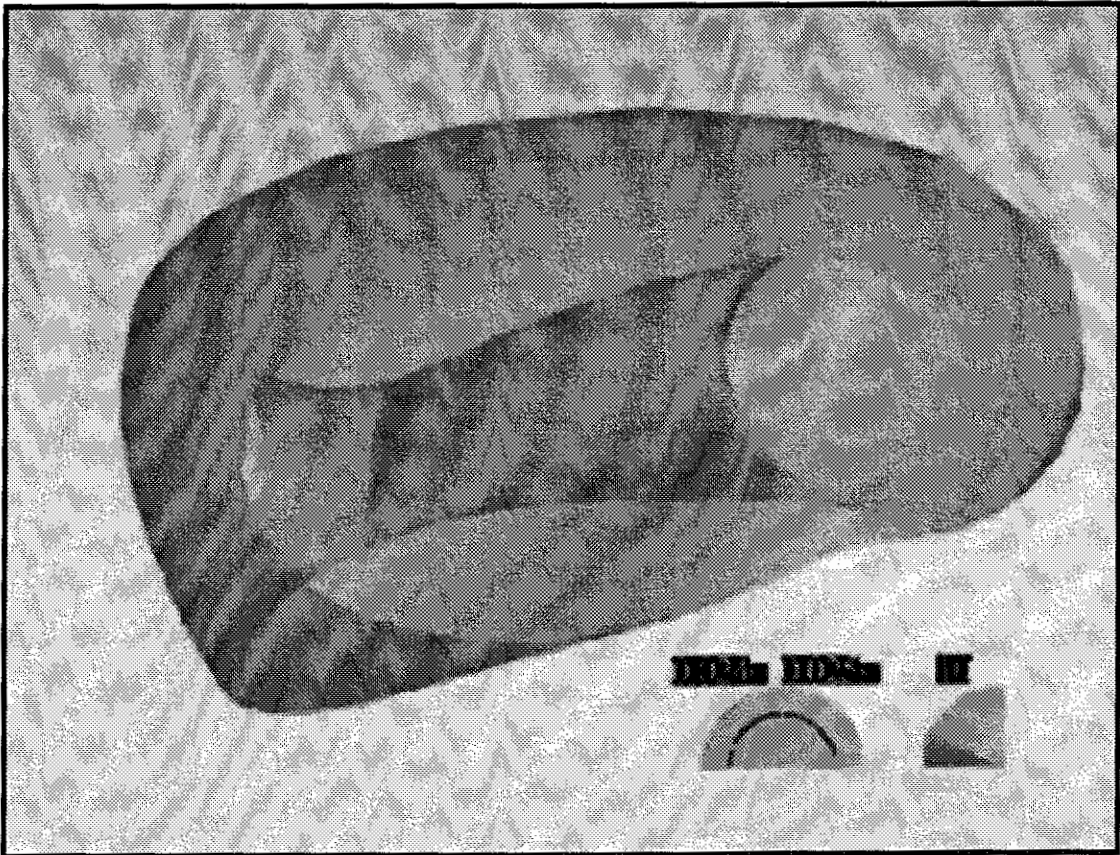
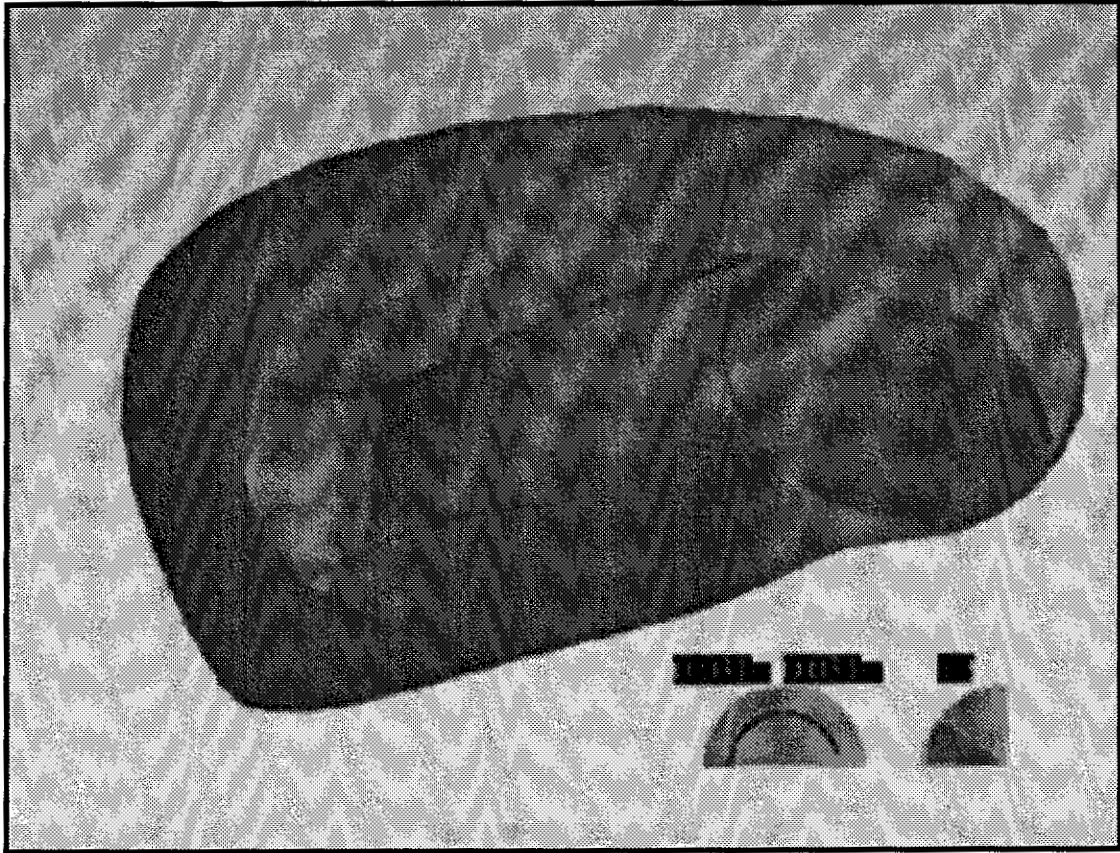
- A** Two opaque intersecting polygons.
- B** Two slightly-transparent intersecting polygons.
- C** Two very transparent intersecting polygons. The polygons have faded from view somewhat, but their intersection has practically disappeared.
- D** The same two polygons with their intersection calculated at each pixel and highlighted.
- E** Torus with a full twist, rendered opaque.
- F** Same torus, but semi-transparent.
- G** Same torus, but very transparent. Now the innermost features are visible, but the outermost shell is merging with the background color.
- H** Same torus as above, but with silhouettes calculated at each pixel and then highlighted.
- I** Torus with a full twist, rendered opaque and clipped by the hither plane. The twist is visible inside, making a pillow shape. The performance meters show that 31 thousand triangles are being transformed and rendered per second at a frame rate of 12 Hertz. This is the performance on Pixel-Planes 5 using 12 GP's and 7 renderers.
- J** The same view, but rendered semi-transparent. Now the performance is about 25 thousand triangles per second.











3.4 Paint

Any interaction that lets the user modify the material properties of a polygon can be considered painting. Thus the user might paint color, opacity, or even shininess onto a surface. By painting color in particular, the user can mark portions of the surface to make them easy to follow with his eye. With 3-space rotations the geometric features on the surface remain rigid. But with 4-space rotations the geometric features in 3-space can change drastically after the surface is projected, making it difficult to track the motion of a given region on the surface.

A simple interface for painting is an iconic cursor whose position on the screen is controlled by some 2D input device. The user positions the icon atop the polygon he wants to paint, and the polygon's color is thereby modified. If the user wants to paint a multi-polygon region of the surface, it is better to leave the painting enabled rather than to require that he activate the paint again for each polygon. The icon should therefore provide visual feedback to indicate whether painting is currently enabled. If the user can select among different colors (or custom-pick a color), the icon should show that color as well. That way he can see what color is about to be painted before he actually marks a region of the surface. Since interactive operations are subject to errors, the paint should be undo-able and the icon should reflect the undo-state visually to the user.

Pixel-Planes 5 has a hardware cursor (or sprite) that is written into the frame buffer at every frame. The sprite's shape is simply a bitmap, and its color can be modified dynamically. I chose to make the sprite look like a can of spraypaint. The interior of the can has the same color as the paint that will be applied to the surface. When painting is enabled, the sprite cycles through an animation of spraypaint droplets issuing from the nozzle. [Plate 3.4 A] During the undo operation, the interior of the can becomes transparent to indicate that the "underlying" color of the surface will be restored.

In a joystick-driven system like Fourphront, painting presents some problems with user interaction. That is because the screen's resolution exceeds that of the joystick. If the joystick position maps directly onto the screen in a one-to-one fashion, then only an interior rectangle of the screen can be reached by the sprite. If the mapping is scaled up to cover the entire screen, the user loses precision in positioning the sprite. That means that a small polygon cannot be painted until it is moved to cover a reachable pixel, or else the paint must be applied over a larger area to reach all the pixels (again causing a loss of precision). Alternatively, the joystick position can govern the velocity of the sprite on the screen. That way the sprite can reach any single pixel with no loss of precision. Unfortunately, this is a

difficult mapping for a user to master: in everyday experience with writing, pointing, or painting, the marking implement moves in direct correspondence to the hand.

I therefore compromised between the two mapping schemes. Over most of the range of the joystick, the sprite moves in direct correspondence to the joystick position. But when the joystick nears the bounds of its range, it imparts an increasing velocity to the sprite. That way the user can relocate the “home” position of the sprite, and then directly paint within a rectangular neighborhood around that position.

Another issue that paint raises is that the user may wish to paint only one side of a polygon at a time, and may even wish to paint the two sides with two different colors. That meant modifying the PPHIGS polygon-picking code so that it would not only supply the polygon’s address, but also indicate whether the polygon is front-facing or back-facing on the screen. Having determined which side to paint which color, Fourphront must also determine how the polygon should behave when the user makes it transparent. What should a thin film of surface look like when its sides have different colors (figure 3-2)?

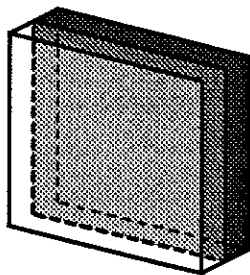


Figure 3-2. A polygon can have different colors on its front and back sides.

I have chosen to treat the two sides as two individual polygons of equal opacity $\hat{\alpha}$ whose cumulative opacity α matches that of the desired polygon (recall that the accumulation of opacity is not additive). Since PPHIGS expects only a single layer of one color, that means I must pre-compute the effective color based on the front and back sides under the effect of transparency. The intensity I_{fb} of a color component (red, green, or blue) on a single-sided polygon combines with that of the background intensity I according to the following formula (for back-to-front transparency).

$$[\alpha I_{fb}] + [(1 - \alpha)I]$$

I wish to solve for the quantity I_{fb} . The intensity of the back face I_b and the front face I_f are known. They can be combined, yielding the following expression for the accumulated intensity:

$$[\hat{\alpha} I_f + (1 - \hat{\alpha}) \hat{\alpha} I_b] + [(1 - \hat{\alpha})(1 - \hat{\alpha})I]$$

By solving for $\hat{\alpha}$ and then I_{fb} , Foughront “tricks” PPHIGS into producing the desired results. To find $\hat{\alpha}$, notice that the two expressions must be equivalent over all values of I_b and I_f . That can only happen if their second terms are the same.

$$(1 - \alpha)I = (1 - \hat{\alpha})(1 - \hat{\alpha})I$$

This gives a quadratic in $\hat{\alpha}$ whose two solutions are given below.

$$\hat{\alpha} = 1 \pm \sqrt{1 - \alpha}$$

Since opacity is constrained to lie in the range $[0, 1]$ the “plus” solution is invalid (except when $\alpha=0$, in which case it is the same as the “minus” solution). This finally yields the effective double-sided intensity for the polygon.

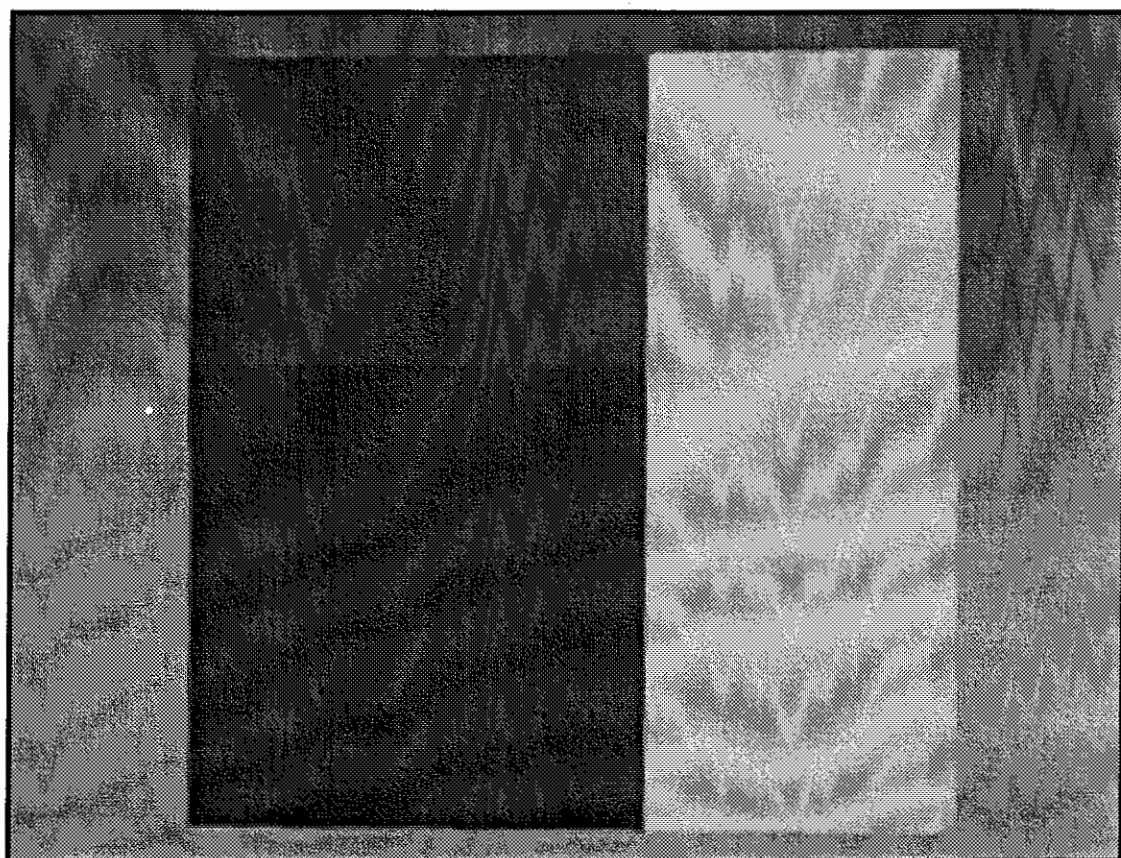
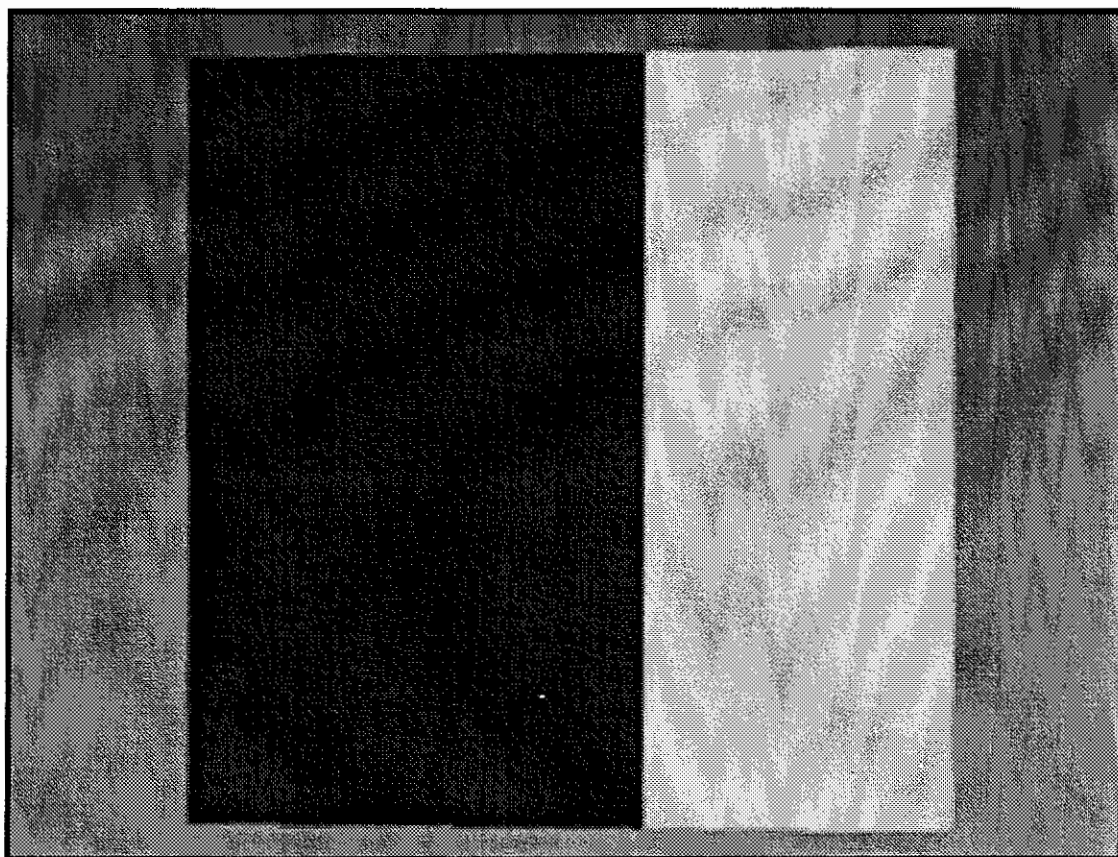
$$I_{fb} = \frac{1}{\alpha} [\hat{\alpha}I_f + (1 - \hat{\alpha})\hat{\alpha}I_b]$$

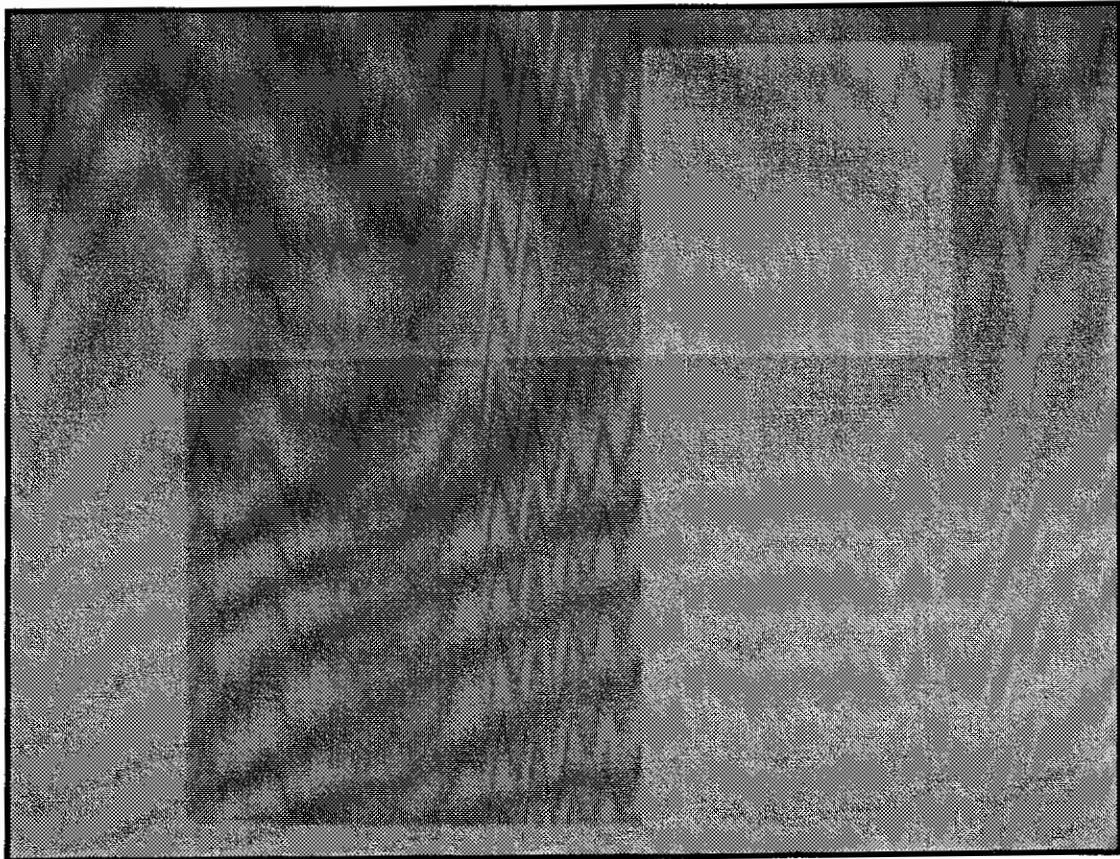
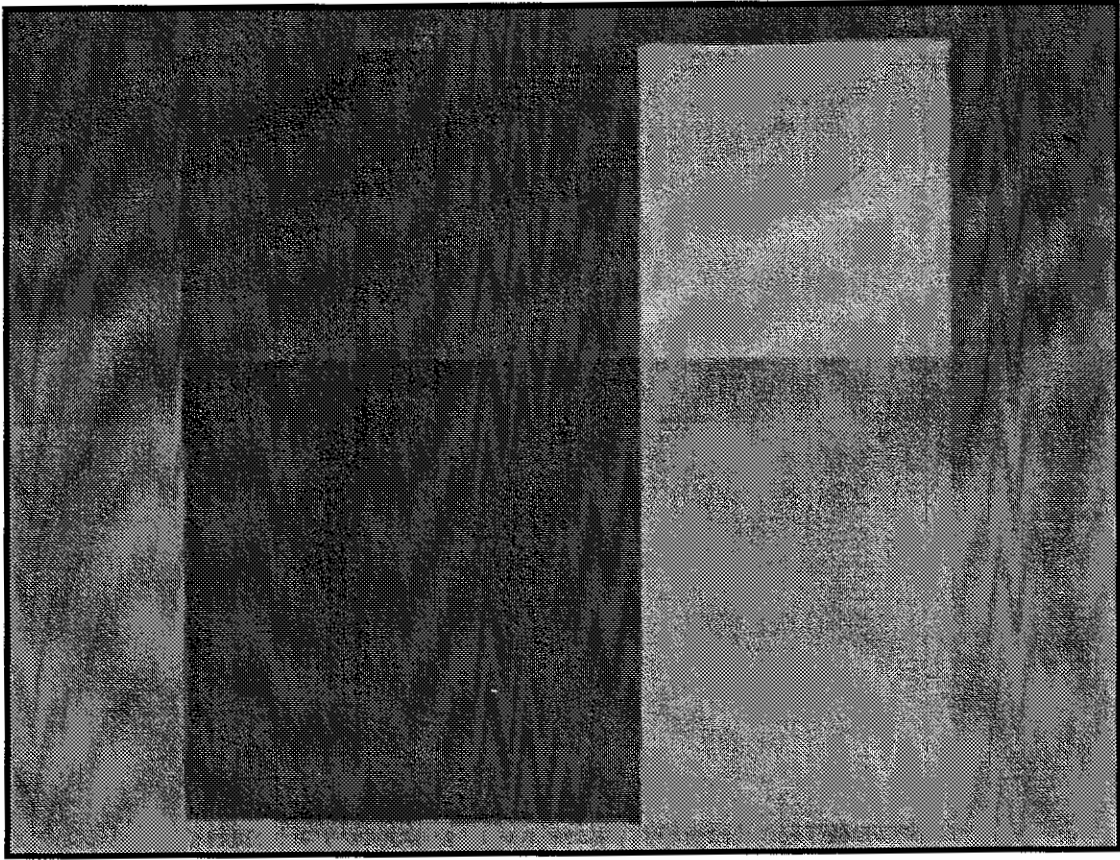
The result is that two different colors on the front and back of a polygon retain their integrity as the polygon fades to transparent. For example, white-atop-black and black-atop-white look like their respective front-most color when opaque, and only gradually become indistinguishable from each other when they are nearly transparent. [plates 3.4 B-D]

Plates 3.4

- A** Bitmaps for the spraypaint icons. The outline of the paintcan, together with the interior and the spray, form an animated cursor. The animation sequence is 10 frames long.
- B** A flat, rectangular surface with different paint patterns on the front and on the back. The front has horizontal white and black stripes. The back has vertical white and black stripes. In this image, the surface is opaque.
- C** The surface has become slightly transparent. Clockwise from the upper-right is white-on-white, white-on-black, black-on-black, and black-on-white. Notice that white-on-black is quite distinct from black-on-white.
- D, E** Same surface, but more transparent. Not only are the different colors blending into the background, but white-on-black and black-on-white are difficult to distinguish from each other.







3.5 Silhouettes

This section describes a screen-oriented technique for locating silhouette curves and intersection curves. Transparency is a powerful technique for visualizing self-intersecting surfaces, but (as I noted in section 3.1) transparency reveals more layers of the surface but substantially diminishes the visibility and the shape of intersection curves and silhouettes. These curves carry important visual information about the shape of the surface, so it is desirable to highlight them to compensate for this negative effect of transparency.

One can estimate the amount of computation required for calculating the geometry of these curves and for rendering semi-transparent surfaces. Even for a modest-sized polygonal model of a few thousand polygons, the burden on the traditional front end of a graphics system becomes prohibitive. Recall that a rigid surface in 4-space projects into 3-space in a non-rigid way, changing shape as it rotates in 4-space. It is possible to calculate transparency, silhouettes, and intersections for a dynamic surface in 3-space, but these tasks are numerically intensive in 3D world-space – enough so that even with tens of high-performance Graphics Processors (GP's) one cannot compute the curves for such a surface at interactive rates. Programmable SIMD renderers shift some of the computation away from the math processors on Pixel-Planes 5 [Ellsworth90, Fuchs89], a setup which makes it possible to display silhouettes and intersection curves of a dynamic 3D surface (projected from 4-space) at interactive rates. This section and section 3.6 describe the details of calculating silhouettes and intersections on the parallel renderers of Pixel-Planes 5.

3.5.1 Varying-width Silhouette Curves

There are several ways to define a silhouette. In common usage, a silhouette is the boundary of the projection of a surface onto the image plane. But a more generous definition regards any point on a differentiable surface as a silhouette (or contour) point if the eye vector lies within the tangent plane to the surface at that point. The second choice is preferable for self-intersecting surfaces since I wish to highlight the silhouettes of the component patches that nest inside a transparent image. A simple way to find a silhouette is to locate every edge that is shared by two polygons, one facing forward and the other facing backward from the eye. (Incidentally, this technique fails to identify contours where the surface inflects as seen from the eye vector.) But if the polygon data is distributed among many processors, the processor that owns a given polygon will not necessarily hold the neighboring ones, even for a mesh that is static in 3-space. Moreover, this technique only identifies silhouettes along mesh boundaries of a polygonal representation of the model, not in the polygons' interiors. PPHIGS uses the Phong shading model, using the normals of an underlying

surface at each vertex of a triangle and then interpolating the sampled normals across the triangle. This technique combines the surface's first-order geometry (for scan conversion) with the surface's second-order geometry (for shading). In essence there is no longer a planar polygon being rendered. Instead there is a polygonal "window" on the screen in which the shaded image of a higher-order surface is displayed. The higher-order surface can exhibit curvature and silhouettes that the planar polygonal approximation itself lacks.

Analytic Solution

One can find the silhouettes on a parametric patch, prior to rendering, by using subdivision [Lane80, Schweitzer82]. There are a couple of reasons why I chose not to require that surfaces be modelled with parametric patches. Modelling with patches (like B-splines or NURBS) is a difficult task even in three dimensions. I want to concentrate on ways to interact with surfaces in 4-space, not on ways to model them. Secondly, I want to use the implementation of PPHIGS on Pixel-Planes 5 wherever I can. This implementation does not handle surface patches.

Screen-based Solution

There is a simple screen-oriented approach to finding silhouettes. As a routine step in Phong-shading, the Pixel-Planes renderers store the primitives' interpolated surface normals at each pixel. Each renderer covers a small region on the screen and holds hundreds of bits of information per pixel. If the normal to a point on a polygon is orthogonal to the eye vector, the point lies on a silhouette curve.

The renderers can calculate a dot product between the normal vector and the eye vector at every pixel, thereby identifying the silhouette when the dot product is zero. This yields, at best, a 1-pixel-thick line on a (curved) surface; at worst, it misses most of the pixels on the silhouette because of the imperfect sampling of the normal vector. One might treat a pixel as a silhouette point if the dot product is within some threshold ϵ of zero, thereby enlarging the silhouette's thickness on the screen (figure 3-3).

Thresholding has two problems. As ϵ gets large, false silhouettes appear wherever the surface is sufficiently edge-on to the eye. Even along true silhouettes, the silhouette curve becomes much fatter in some places than in others. The false silhouettes are inherent to thresholding: a planar cross-section of the surface containing the eye may have an inflection whose normal is nearly orthogonal to the eye vector. The neighborhood of the inflection will fall within a thickened silhouette curve, even though there is no genuine silhouette there.

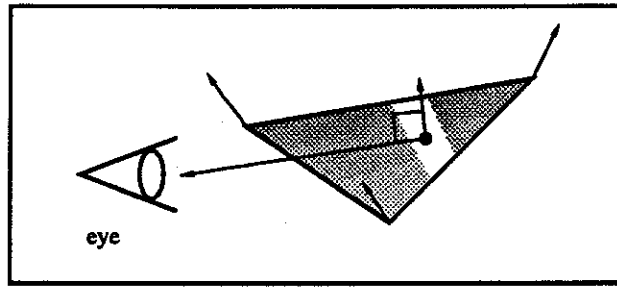


Figure 3-3. The surface normal is nearly orthogonal to the eye vector in the vicinity of a silhouette curve.

Why does the thickness of the silhouette vary when the dot product is allowed to depart from zero? The curvature of the surface may vary from place to place. If a silhouette point has large curvature in the direction of the eye, its normal vector changes quickly along a path toward the eye. Even a large value of ϵ may therefore produce a thin silhouette curve near the point. In contrast, a silhouette point with small curvature in the direction of the eye will witness its normal vector changing slowly along a path toward the eye. The same value of ϵ produces a thicker silhouette on the screen since there are points over a large area (as seen from the eye) whose normals are nearly perpendicular to the eye vector. [plates 3.5 A, B] In the next section I derive formulas for generating better approximations of a fixed-width silhouette curve.

Having found a silhouette, what does one do with it? The question concerns visualization in its abstract sense. How can the internal state at a pixel be effectively mapped onto the available dimensions of output (e.g., red, green, and blue)? A simple solution is to map silhouettes to a particular color that is known to be absent elsewhere in the rendered surface. Such a color may not, of course, exist. But assigning a constant color on the silhouette of a smoothly shaded surface is often, in practice, a sufficient visualization. In the case of a transparent image, it can also be effective to assign complete opacity to a silhouette in order to make it more prominent. One can even relax the binary classification of silhouettes in favor of a real-valued measure of "silhouetteness." If the intrinsic opacity of the surface at a point is α , the effective opacity might be calculated as $1 - (1 - \alpha)^{1/d}$, where d is the dot product of the eye vector and the normal vector. Surfaces then become increasingly opaque near their silhouettes, which mimics the natural behavior of semi-transparent film. Viewed away from the normal by an angle whose cosine is d , a thin film of width w intercepts a ray through a distance w/d . I experimented with this technique on Pixel-Planes 5, calculating the effective opacity per vertex on the GPs and then interpolating the result across the renderers. The contours indeed become more prominent, but the images are confusing if transparency is simultaneously being used as a depth cue.

3.5.2 Fixed-width Silhouette Curves

In order for the renderers to calculate a fixed-width silhouette curve on a surface, each pixel must store sufficient state to estimate its screen-distance from the silhouette curve. A planar approximation to the surface is not enough. A plane is either entirely silhouette in its profile, or else entirely not. A local approximation must therefore be at least quadratic. Six vertex positions in a neighborhood are enough to produce a quadratic approximation, as are three vertices together with their normals. The latter approach has the advantage that a single triangle carries enough information with it to derive a quadratic approximation.

The strategy is to estimate the shape of the surface at \mathbf{p} in two directions within the tangent plane, construct the shape operator S at \mathbf{p} , and find the principal directions and curvatures at \mathbf{p} . This leads to a quadratic approximation of the surface. One can then calculate the distance from the point \mathbf{p} to the silhouette of the quadric surface as it appears on the screen.

The shape operator S , applied to a vector \mathbf{u} in the tangent plane, returns another vector $S(\mathbf{u})$ in the tangent plane which is the (negative of the) the derivative of the normal in the \mathbf{u} -direction; thus, $|S(\mathbf{u}) \cdot \mathbf{u}|$ measures the normal curvature in the direction \mathbf{u} . The directions in which this curvature is a max or a min are called the principal directions.

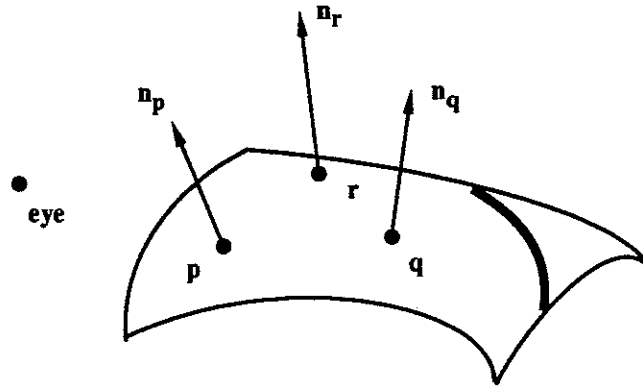


Figure 3-4. Three points, together with their normals, can be used to estimate the surface's curvature. The dark curve represents the silhouette as seen from the eye position.

Three points \mathbf{p} , \mathbf{q} , \mathbf{r} on a surface, together with their unit normals \mathbf{n}_p , \mathbf{n}_q , \mathbf{n}_r , suffice to estimate the surface's curvature. If the three points are sufficiently close to each other, \mathbf{q} and \mathbf{r} nearly lie within the tangent plane at \mathbf{p} . They therefore permit an estimate of the surface's normal curvature in the directions $(\mathbf{q}-\mathbf{p})$ and $(\mathbf{r}-\mathbf{p})$.

Let \mathbf{v}_1 be a unit vector in the $(\mathbf{q}-\mathbf{p})$ -direction and \mathbf{w} be a unit vector in the $(\mathbf{r}-\mathbf{p})$ -direction.

$$\mathbf{v}_1 = \frac{\mathbf{q} - \mathbf{p}}{\|\mathbf{q} - \mathbf{p}\|} \quad \mathbf{w} = \frac{\mathbf{r} - \mathbf{p}}{\|\mathbf{r} - \mathbf{p}\|}$$

The directional derivative of the normal is approximated by $\widehat{\nabla}(\mathbf{n})$ in these two directions and is given by the following:

$$\widehat{\nabla}_{\mathbf{v}_1}(\mathbf{n}) = \frac{\mathbf{n}_q - \mathbf{n}_p}{\|\mathbf{q} - \mathbf{p}\|} \quad \widehat{\nabla}_{\mathbf{w}}(\mathbf{n}) = \frac{\mathbf{n}_r - \mathbf{n}_p}{\|\mathbf{r} - \mathbf{p}\|}$$

The pair of vectors $\{\mathbf{v}_1, \mathbf{w}\}$ are not necessarily an orthonormal basis for the tangent plane since there is no guarantee that they are perpendicular. The Gram-Schmit orthonormalization technique takes the pair $\{\mathbf{v}_1, \mathbf{w}\}$ and produces an orthonormal coordinate system $\{\mathbf{v}_1, \mathbf{v}_2\}$ via an intermediate vector $\widehat{\mathbf{v}}_2$ orthogonal to \mathbf{v}_1 .

$$\widehat{\mathbf{v}}_2 = \mathbf{w} - (\mathbf{w} \cdot \mathbf{v}_1) \mathbf{v}_1$$

$$\mathbf{v}_2 = \frac{\widehat{\mathbf{v}}_2}{\|\widehat{\mathbf{v}}_2\|}$$

In the $\{\mathbf{v}_1, \mathbf{v}_2\}$ -coordinate system, \mathbf{w} has coordinates

$$\mathbf{w} = (\mathbf{w} \cdot \mathbf{v}_1, \mathbf{w} \cdot \mathbf{v}_2) = (w^1, w^2)$$

where the superscripts index the coordinates of \mathbf{w} . The vector \mathbf{v}_1 has coordinates $(1, 0)$.

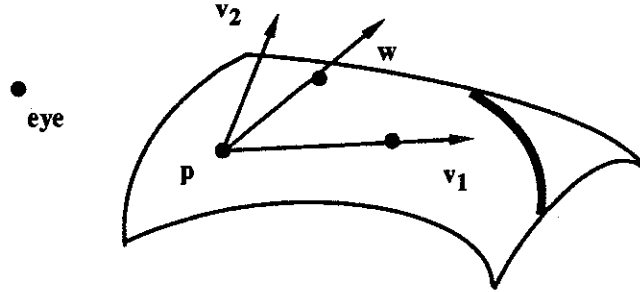


Figure 3-5. The other two vertices from the surface can be used to produce an orthonormal basis for the tangent plane at the point p .

These preliminaries accomplish a simple task. There is now an orthonormal basis for the surface patch. The difference quotient of the normal is available along two legs (courtesy of Phong shading), which provides two estimates of the normal curvature of the underlying surface. If the two vectors \mathbf{v}_1 and \mathbf{w} are not collinear, the shape operator applied in these two vector-directions reveals its matrix elements.

$$S(\mathbf{v}_1) = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} s_{11} \\ s_{21} \end{bmatrix} \approx - \begin{bmatrix} \widehat{\nabla}_{\mathbf{v}_1}^1(\mathbf{n}) \\ \widehat{\nabla}_{\mathbf{v}_1}^2(\mathbf{n}) \end{bmatrix}$$

$$S(\mathbf{w}) = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} w^1 \\ w^2 \end{bmatrix} = \begin{bmatrix} s_{11}w^1 + s_{12}w^2 \\ s_{21}w^1 + s_{22}w^2 \end{bmatrix} \approx - \begin{bmatrix} \hat{\nabla}_{\mathbf{w}}^1(\mathbf{n}) \\ \hat{\nabla}_{\mathbf{w}}^2(\mathbf{n}) \end{bmatrix}$$

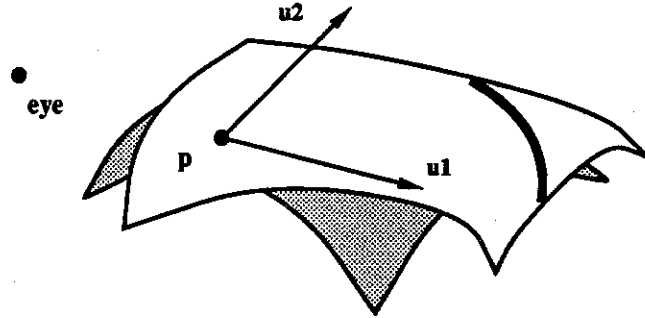


Figure 3-6. The original surface (white) is approximated by a quadric surface (dark) derived from the shape operator.

Having calculated the approximations on the right-hand sides, one can solve for the elements s_{ij} of S . The principal directions $\{\mathbf{u}_1, \mathbf{u}_2\}$ of the surface at \mathbf{p} are the directions in the tangent plane where the normal curvature is extreme. They are also the eigenvectors of the matrix for S . The normal curvature κ_i in these two (orthogonal) directions is given by the corresponding eigenvalues of the matrix. Thus the determinant

is the quadratic equation

$$(s_{11} - \kappa_i)(s_{22} - \kappa_i) - s_{12}s_{21} = \kappa_i^2 + (-s_{11} - s_{22})\kappa_i - s_{12}s_{21} = 0$$

which can be solved for κ_i . Each κ_i has an eigenvector $\hat{\mathbf{u}}_i$, yielding the orthonormal basis $\{\mathbf{u}_1, \mathbf{u}_2\}$ for the tangent plane. One can extend the basis by the cross-product

$$\mathbf{u}_3 = \mathbf{u}_1 \times \mathbf{u}_2$$

in 3-space. The best-fit quadratic approximation to the original surface is then

$$(u_1, u_2, \kappa_1 u_1^2 + \kappa_2 u_2^2)$$

in the $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ coordinate system. The attitude matrix A transforms the local tangent space to the tangent space of the surface patch.

$$A = \begin{bmatrix} u_1^1 & u_2^1 & u_3^1 \\ u_1^2 & u_2^2 & u_3^2 \\ u_1^3 & u_2^3 & u_3^3 \end{bmatrix}$$

Since the silhouette depends on the position of the eye (which is the center of projection), the appropriate local coordinate system is that of eye-space. Thus $A: T_{\text{eye}} \rightarrow T_{\text{patch}}$ defines the transformation between these two coordinate systems. Let \mathbf{q} denote the position of the eye. Then the coordinate representation of \mathbf{q} depends on the coordinate system in use. So

$$\mathbf{q} = (0, 0, 0)_{\text{eye}} = (q_1, q_2, q_3)_{\text{patch}}$$

$$\mathbf{p} = (p_1, p_2, p_3)_{\text{eye}} = (0, 0, 0)_{\text{patch}}$$

are the coordinate representations of \mathbf{q} and \mathbf{p} in eye-space and in patch-space. The translation from one system to the other is just $(\mathbf{p} - \mathbf{q})_{\text{eye}} = -\mathbf{q}_{\text{patch}}$. The notation is cumbersome, but the distinction is important since the silhouette calculations concern the representation of the surface in the different coordinate system.

The coordinates of \mathbf{p}_{eye} are available after the model is transformed to eye-space. But what are the coordinates of $\mathbf{q}_{\text{patch}}$? Since

$$\mathbf{q}_{\text{eye}} = A \mathbf{q}_{\text{patch}} + \mathbf{p}_{\text{eye}}$$

one can solve for $\mathbf{q}_{\text{patch}}$ as

$$\mathbf{q}_{\text{patch}} = A^{-1}(\mathbf{q}_{\text{eye}} - \mathbf{p}_{\text{eye}}) = A^{-1}(-\mathbf{p}_{\text{eye}})$$

since \mathbf{q}_{eye} is the origin.

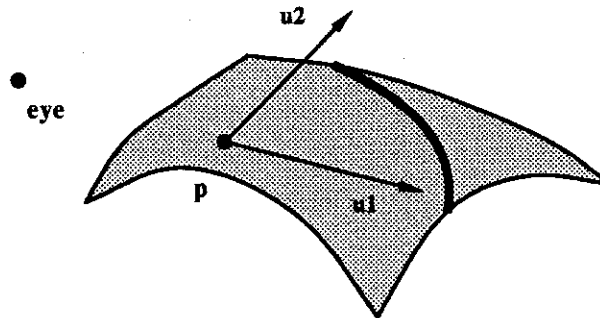


Figure 3-7. The approximating surface has its own silhouette curve, as seen from the eye position. The angle between the point p and the silhouette curve measures their distance apart when displayed on the screen.

Now the silhouette curve can be estimated as the silhouette that lies on the quadratic surface: S is the set of points whose normals are perpendicular to the eye-vector. That is,

$$(s - q) \cdot n_s = 0$$

for any point $s \in S$. The coordinates of s are given by

$$s = s(s_1, s_2)_{\text{param}} = (s_1, s_2, \kappa_1 s_1^2 + \kappa_2 s_2^2)_{\text{patch}}$$

in the parametric coordinates of $\{u_1, u_2\}$ and the patch coordinates of $\{u_1, u_2, u_3\}$. The normal at s is

$$n_s = s_{u_1} \times s_{u_2} = \frac{d}{ds_1} s(u_1, u_2) \times \frac{d}{ds_2} s(u_1, u_2) = (1, 0, 2\kappa_1 s_1) \times (0, 1, 2\kappa_2 s_2)$$

so

$$n_s = (-2\kappa_1 s_1, -2\kappa_2 s_2, 1)_{\text{patch}}$$

in patch coordinates. Substituting the right-hand expression into the dot-product gives

$$(s_1 - q_1, s_2 - q_2, s_3 - q_3) \cdot (-2\kappa_1 s_1, -2\kappa_2 s_2, 1) = 0$$

which can be written as a quadratic in s_1 :

$$-\kappa_1 s_1^2 + 2\kappa_1 q_1 s_1 + (-\kappa_2 s_2^2 + 2\kappa_2 q_2 s_2 - q_3) = 0$$

This one-parameter solution locates the silhouette $S(s_1)$ on the quadratic surface.

The remaining task is to determine the distance on the screen between the point p and the silhouette S . That distance is the minimum of all distances $\text{dist}(p_{\text{screen}}, S(s_1)_{\text{screen}})$ along the silhouette. At such a point s_{screen} , the normal n_s is parallel to $(s-p)$. Hence p , s , and $(s+n_s)$ are collinear. But that makes p , s , $(s+n_s)$, and q (the eye position) coplanar through the origin p in patch-space. So $(s-p)$ and $(s+n_s)-p$ lie in a plane through p , and their difference n_s does as well. In patch coordinates, s , q , and n_s are coplanar if

$$s \cdot (q \times n_s) = 0.$$

The solutions form a discrete, finite set in the general case. The element s with the smallest distance is the closest silhouette point to p . In order for the point p to be a member of the fixed-width silhouette with width δ in the image plane, the distance D between the projections $\pi(p)$ and $\pi(s)$ must not exceed $\delta/2$.

$$D = \| \pi(p) - \pi(s) \| < \frac{\delta}{2}$$

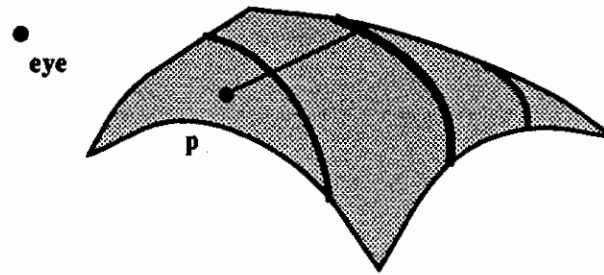


Figure 3-8. Now the point p can be compared against a fixed-width threshold based on the quadric approximating surface.

The degenerate case occurs when the original “triangle” has zero area (the sides are coincident) or when the surface is umbilic at p . The first case has no impact on the image, because such degenerate triangles typically are not rendered onto the screen. In the case of the umbilic point, every direction exhibits the same normal curvature, so there are no principal directions. One can still recover a quadric approximation to the surface in the coordinate system of the tangent plane. One has

$$(u_1, u_2, \kappa(u_1^2 + u_2^2))$$

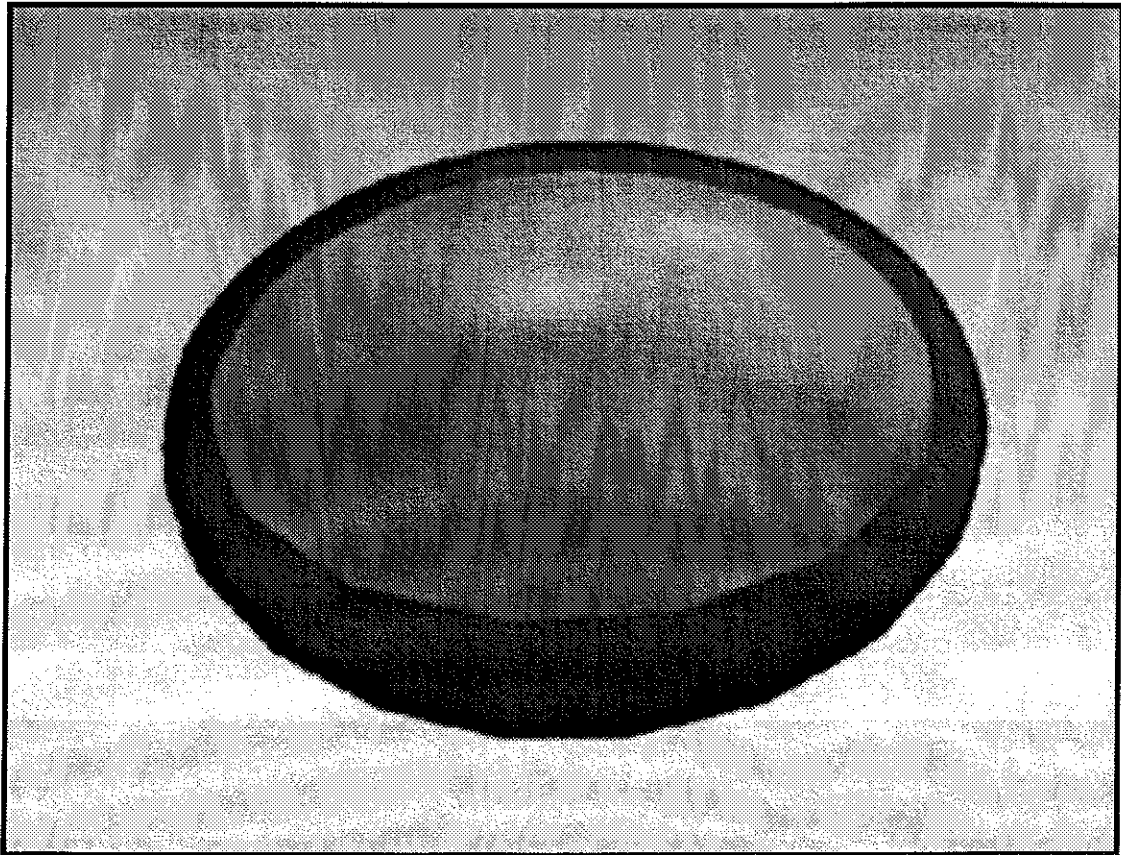
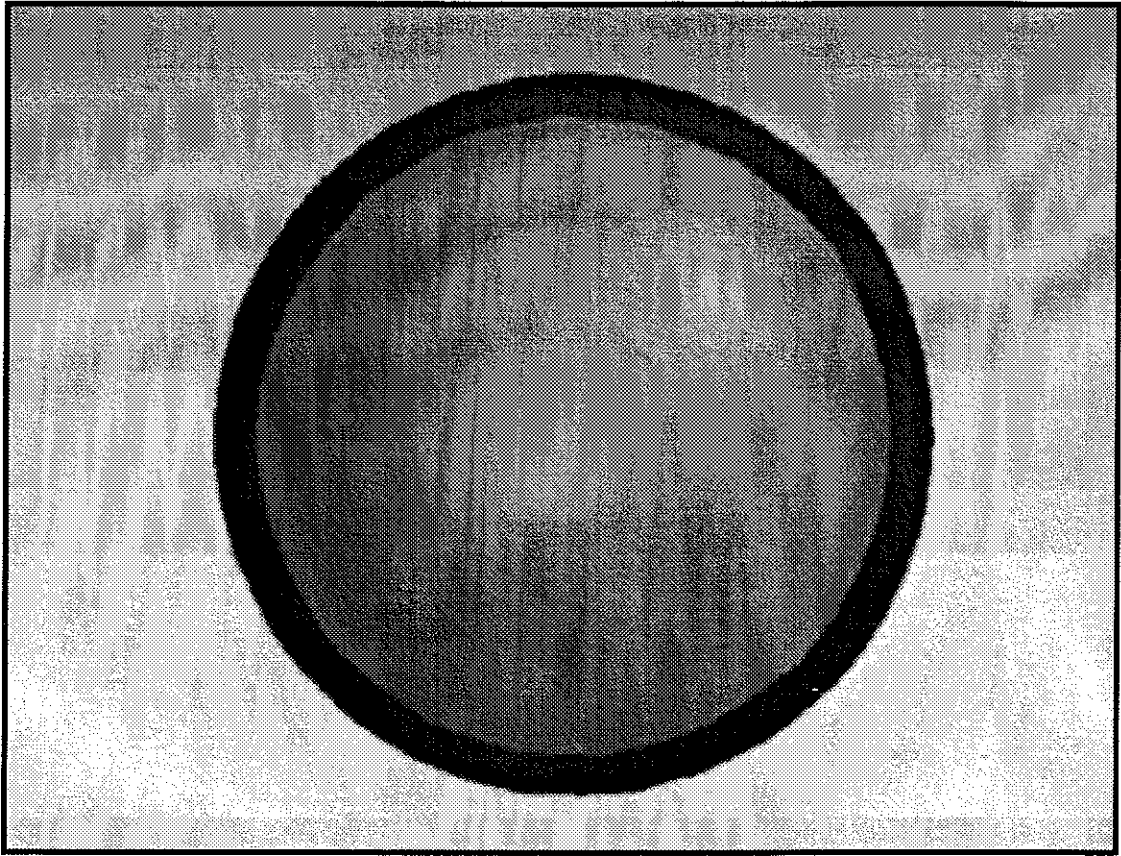
regardless of how u_1 and u_2 are oriented in the tangent plane. The most difficult degeneracy occurs when the umbilic is planar. In that case, the quadric is either nowhere-silhouette or everywhere-silhouette with respect to a given eyepoint. In some sense there is no way to redeem this situation: there really is not enough information at a planar point to provide an estimate of the distance to a genuine silhouette curve. One can arbitrarily choose to include such a triangle in the silhouette or to exclude it.

The calculation of fixed-width silhouettes is, unfortunately, too ambitious for the pixel-processors of Pixel-Planes 5 to perform. These 1-bit processors have only 208 bits of local storage. The computation begins with three points and their normal vectors. In practice, at least 16 bits of accuracy are needed for each coordinate, for a total of nearly 100 bits. Red, green, and blue components of color already use 24 bits. Diffuse and specular coefficients use another 16. The remaining space does not offer enough scratch area for the intermediate calculations. One could instead calculate the distance D at each vertex (using the graphics processors) and interpolate the result across the pixel. That strategy causes two other problems. First, the thickened silhouette curve becomes piecewise linear instead of truly curved. Second, the computation is an order of magnitude longer than just transforming and projecting each vertex to the screen. If the calculation cannot be deferred until end-of-frame

(where the SIMD pixel processors can perform it), it becomes excessively slow. Fixed-width silhouettes must wait on the next generation of machinery before the computation becomes plausible to carry out.

Plates 3.5

- A** An oblate spheroid has a fixed-width silhouette curve when the curve is calculated against a threshold, but only because the surface is being viewed from a special vantage point. The curvature is the same all along the silhouette.
- B** As the spheroid rotates, the flatter part of the surface is on the bottom and the rounder part is on the top. As a result, the silhouette curve on the bottom is thicker than it is on the top.



3.6 Intersection Curves

If the projected surface in 3-space were static, the intersection curves could be analytically computed [Baraff90, Moore88] once and for all. Since transformations in 4-space make the surface's 3-space projection change shape dynamically, those intersections are recomputed each frame. This can be accomplished easily within the SIMD renderers for opaque surfaces. The straightforward approach to finding intersections is to modify the usual z-buffer algorithm. The z-value of each incoming polygon at each pixel is tested against the contents of the z-buffer, acquiring the polygon's state information if the polygon is closer. If the new value matches the z-buffer, it counts as an intersection. If an intersection has been flagged at a pixel and then a closer polygon comes along, the intersection flag becomes unset. The result is that all the frontmost intersections will be flagged.

It is easy to see that order does not matter in processing the polygons. Let $\{P_i\}$ be the set of polygons that cover a pixel, indexed by the order in which they are processed, and let P_j and P_k ($j < k$) be two of them that participate in the front-most intersection at that pixel. The z-buffer must contain z_j after P_j is processed. Since P_j is frontmost at the pixel, the z-buffer still contains z_j when P_k is processed, thereby setting the intersection flag. Since P_k is frontmost at the pixel, the flag will not be unset. By piggy-backing on the multipass algorithm for transparency, we can find all the interior intersections, since they will be frontmost intersections at some particular pass.

Two polygons that share an edge must formally intersect each other along it. Pairs of polygons whose edges pass through pixel centers will "intersect" at those pixels. These intersections are merely artifacts of the surface's representation. To avoid rendering them, one could be careful not to draw pixels more than once along the common boundary of adjacent polygons. This technique presents a problem for a machine like Pixel-Planes, which is suited to rendering entire polygons as primitives without maintaining connectivity information. But in fact the pixel already holds sufficient information to eliminate spurious intersections: surface normals. The genuine intersections are those of polygons that dive through each other, *i.e.*, whose normals are different where they interpenetrate. Since the SIMD renderers interpolate vertex normals, that information is available per pixel. One can thus modify the z-comparison, requiring that the normals be sufficiently different. That is, when a new z-value matches on old one, the new normal vector must deviate from the old as well. Otherwise, the new polygon probably shares an edge with the old polygon at this pixel.

3.6.1 Varying-width Intersection Curves

Exact matching against the z-buffer can identify at best a 1-pixel-wide intersection curve. At worst it misses much of the curve due to imperfect sampling (just as is the case with silhouette curves). One remedy to this problem is to apply a threshold. If the incoming pixel is within ϵ of the z-buffer value, it is considered to be an intersection point. This introduces the same artifact of variable-width curves on the screen. If two polygons intersect each other at a shallow angle, their z-separation remains small over a large area of the screen and the curve that satisfies $|z_{\text{new}} - z_{\text{old}}| < \epsilon$ is many pixels wide. If they intersect each other at a steep angle, a short excursion to neighboring pixels will find them separated far apart in the z-direction. [Plates 3.6 A,B] One can use the interpolated normals of the polygons at pixels near the intersection in order to approximate a fixed-width intersection curve. This added computation is charged per polygon and cannot be deferred to end-of-pass unless the pixel retains the geometric state of both polygons. Most implementations of the z-buffer algorithm interpolate reciprocal-z across the polygon (because z itself does not vary linearly). Over small extents of z, thresholding produces nearly the same images even when using the reciprocal (although the size of the threshold itself must change). But for locating intersections across large ranges, it is necessary to recover the true depth.

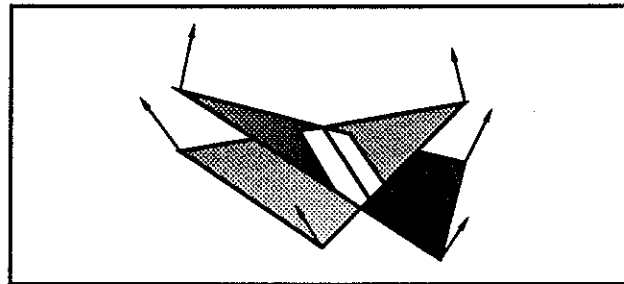


Figure 3-9. At their common intersection, two polygons share z-values. The z-values are within some threshold of each other along a thickened intersection curve.

Another artifact of thresholding is that the thickened intersection curve gets trimmed near silhouettes, since the depth-comparison is strictly within the z-direction rather than the normal directions of the participating polygons. In other words, a pixel can only be considered to be near an intersection if two criteria are met. First, there must be two surface primitives that lie on the pixel. Second, their depths must be nearly equal. The figure above illustrates the situation for two interpenetrating triangles. The white region along the intersection curve can only be calculated within the pixels covered by both triangles. As a result, the thickened intersection curve reveals the boundary of the underlying triangle at its top. This behavior is hard to overcome without using pixel-to-pixel communication.

3.6.2 Fixed-width Intersection Curves

In order to draw a fixed-width intersection curve, one is required to determine a distance at every point on the surface, namely, the distance from the point to the nearest intersection curve. For a meshed surface, the primitives are planar regions. Given two intersecting planes in projection coordinates, and a point $\mathbf{p} = (p_x, p_y)$ on the screen, how far is \mathbf{p} from the intersection?

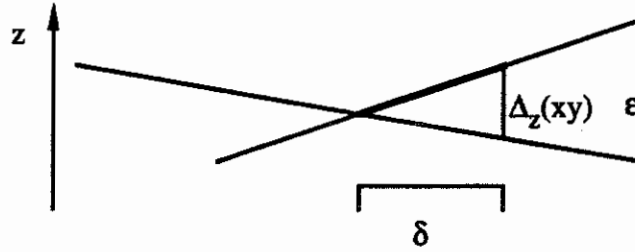


Figure 3-10. The goal is to determine whether a point on a surface is within δ of the intersection. Depending on the slopes of the two surfaces, the point must lie within some ϵ (in the z -direction) of the other surface.

The plane equations suffice to answer the question. If the distance on the screen is less than some fixed δ , \mathbf{p} is included in the fixed-width intersection curve. Let the planes be represented by the following two equations:

$$z_1 = a_1 x + b_1 y + c_1$$

$$z_2 = a_2 x + b_2 y + c_2$$

The z -separation between the two planes is simply their difference $\Delta_z(x,y)$:

$$\Delta_z(x,y) = z_2 - z_1 = (a_2 - a_1)x + (b_2 - b_1)y + (c_2 - c_1)$$

The intersection occurs where $\Delta_z(x,y) = 0$. The vector $(\Delta a, \Delta b)$ specifies the direction of maximum slope of $\Delta_z(x,y)$. In that direction the squared slope is $(a_2 - a_1)^2 - (b_2 - b_1)^2$.

Dividing this plane equation by its gradient-magnitude yields a normalized plane with the same zero-set, but with a slope of 1 in the gradient direction. In the normalized plane, the height $z(x,y)$ is precisely the same as the distance $\Delta_{xy}(p_x, p_y)$ between the point \mathbf{p} and the zero-set in the xy -plane.

$$\Delta_{xy}(x,y) = \frac{\Delta_z(x,y)}{\sqrt{(a_2 - a_1)^2 - (b_2 - b_1)^2}}$$

Therefore, given some $\delta > 0$, choose ϵ so that

$$\varepsilon = \delta \sqrt{(a_2 - a_1)^2 + (b_2 - b_1)^2}$$

which then ensures that

$$|\Delta_z(x,y)| \leq \varepsilon \Rightarrow |\Delta_{xy}(x,y)| \leq \delta$$

That is, the point (p_x, p_y) is within δ of the intersection curve on the screen if $\Delta_{xy}(p_x, p_y)$ is no larger than ε in magnitude. Note that when the two planes are parallel or coincident, ε is exactly zero and the left-hand inequality is either everywhere or nowhere satisfied. The computation is limited in practice by the precision of the numerical representation, so planes separated by small angles are effectively parallel. The one-bit arithmetic unit cannot calculate the square root very speedily, so it is preferable to solve instead for ε^2 .

The Pixel-Planes renderers can compute this distance in two different ways: once per primitive or once at end-of-frame. Deferring it until end-of-frame saves computation over evaluating it for every primitive, but then the pixel memory must retain (or recover) the plane coefficients a_i and b_i for the frontmost two primitives. The pixel already stores the interpolated normal for a polygon in order to Phong-shade it. One could recover the coefficients a and b from this normal vector in order to save bits of pixel-memory. Given a normal vector $\mathbf{n} = (n_x, n_y, n_z)$ with $n_z \neq 0$, let

$$a = -\frac{n_x}{n_z}, \quad b = -\frac{n_y}{n_z}$$

define a plane $z = ax + by$. That the dot product between \mathbf{n} and the gradient vector within the plane is

$$(n_x, n_y, n_z) \cdot \left(-\frac{n_x}{n_z}, -\frac{n_y}{n_z}, \left(\frac{n_x}{n_z}\right)^2 + \left(\frac{n_y}{n_z}\right)^2 \right) = 0$$

shows that this plane has the desired normal. Notice that if $n_z = 0$ the image of the entire plane degenerates to a line and at no point \mathbf{p} away from the line can one reasonably compute a distance to the intersection. In practice, the number of bits devoted to the normal and the plane-coefficients limits how steep the plane can be before it is effectively degenerate due to numerical error. Even with this economical use of memory, there are still not enough bits in the pixels (on Pixel-Planes 5) to make the technique of recovering plane equations from normal vectors feasible on an end-of-frame basis. That memory could be scavenged from other bit-fields, but only at the expense of other algorithms (like transparency).

I will now describe the details of calculating the fixed-width intersection curve on a per-primitive basis, using the pixel memory map that the rest of the underlying graphical system (PPHIGS) shares. There are 208 bits of local pixel memory. The low 32 bits are used for transferring data into and out of a pixel's backing store. The remaining bits are used to collect the various surface attributes during the transformation pass, which are then used for textures and illumination at end-of-frame. PPHIGS uses every single bit of this local memory.

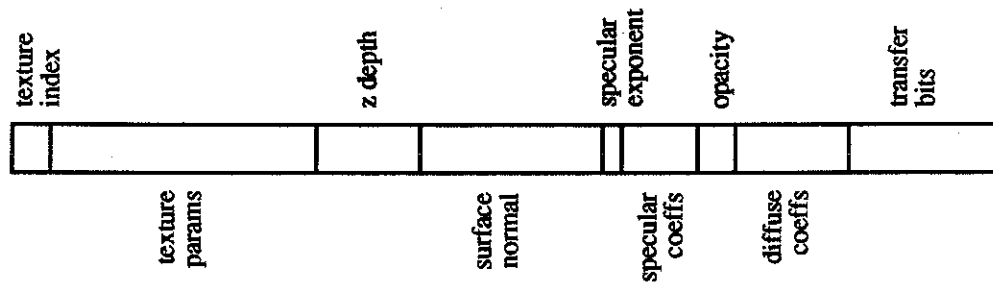


Figure 3-11. This is the default usage for the 208 bits of per-pixel memory.

No additional bits are needed to calculate the variable-width curve, but many bits are needed for scratch computations when calculating fixed-width curves. The z-depth has 22 bits of precision, so the difference $\Delta_z(x,y)$ between the depths of two primitives requires up to 22 bits as well (23 if it is a signed difference). The 32 transfer bits provide enough scratch area for the 22-bit accumulated result of the calculation. How many bits are required for the coefficients of the plane equation? Since a polygon can be edge-on to the eye, the coefficients for its plane equation can be arbitrarily large. In practice, however, about 17 bits are enough.

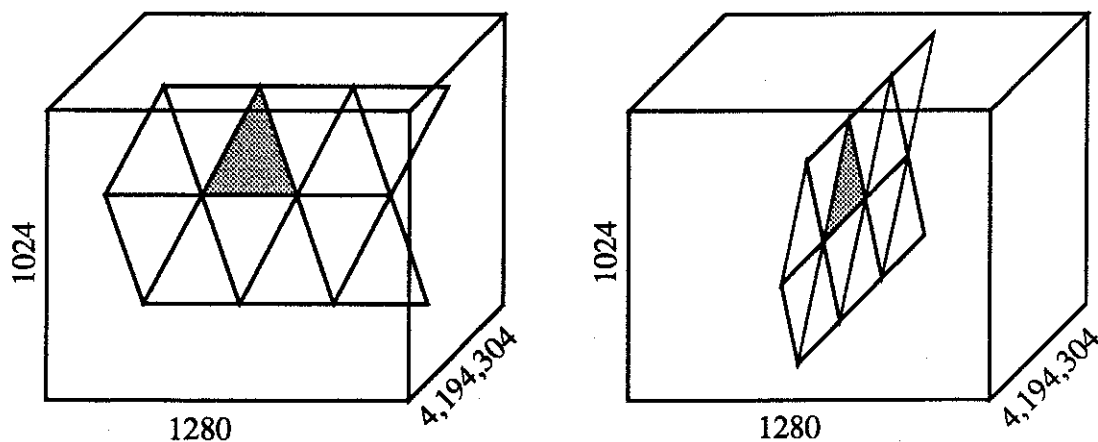


Figure 3-12. The triangle mesh covers the (x,y) -range of the screen when it is parallel to the screen. When it rotates a quarter turn, it covers the range of the z-buffer.

To see why this is true, consider a surface whose image fits within a screen of dimensions 1024 by 1280 pixels. As the surface rotates, its depth should lie within the bounds of the z-buffer range. When Pixel-Planes is configured with about 20 GP's and 20 renderers (typical for a single-rack system), Fourphront can process about 70,000 triangles per second. To maintain an interactive speed of 20 frames per second, Fourphront must therefore have a dataset of 3500 polygons or fewer. If these polygons were of equal area and filled the screen, each would cover about 400 pixels. A triangle with 40-pixel base and 20-pixel height is representative: some triangles are more eccentric and some are less. A 40-pixel base uses about 3% of the available screen width. When the triangle rotates to become edge on, its base occupies about 3% of the available z-depth (about 130,000 z-units). I want to calculate with this triangle's plane coefficients until it is nearly edge-on. When the triangle is nearly edge-on, so that it is only 1 pixel wide (on the screen) but 130,000 deep (in the z-buffer range), its plane equation has an x-coefficient of 130,000. This magnitude is nearly 2^{17} , which is why the scratch calculations need about 17 bits. I use 17 bits for each of the two coefficients (aOld and bOld) of the plane equation and 22 bits for the difference in z-depth, all scavenged from the texture area of pixel memory. That means that I forego using textured polygons in Fourphront.



Figure 3-13. Fourphront scavenges bits from the memory-transfer area and from the texture area of the default usage. These bits are used to compute the fixed-width intersection curve.

Each pixel evaluates an expression and, based on the result, decides if it lies within the fixed-width intersection curve. In high-level pseudo-code this is very simple to do, as shown below.

$$\text{if } (Z_{old} - Z_{new})^2 < \delta^2 ((a_{old} - a_{new})^2 - (b_{old} - b_{new})^2)$$

then the pixel lies within the curve

With the tight space in pixel memory and with only simple macros to control the renderers, the calculation actually takes many steps. The pixel-variable `result` will store the difference between the left- and right-hand expressions in the inequality above. Thus its sign determines whether a pixel lies within the thick curve or not. The variables `aOld` and `bOld` store the plane coefficients for the previous polygons, but pre-multiplied by δ^2 so that the renderer can avoid the extra multiply. The variable `zOld` is the content of the z-buffer area of pixel memory, and `zNew` is evaluated by the Quadratic Expression Evaluator tree.

Individual differences (between coefficients and between z-depths) are stored in `diff`, then squared and accumulated into `result`. The pixel-level pseudo-code is illustrated below.

```

if zOld < zNew then diff := zNew - zOld
else                diff := zOld - zNew
result := diff * diff
result := -result
diff   := aOld -  $\delta$ *aNew
result := result + diff*diff
diff   := bOld -  $\delta$ *bNew
result := result + diff*diff
if result ≤ 0 then
    aOld :=  $\delta$ *aNew
    bOld :=  $\delta$ *bNew
    flag := true

```

A necessary precondition to entering this code is that the variables `aOld` and `bOld` contain the plane-equation coefficients for the front-most (z-buffered) polygon that has been processed, but with those coefficients pre-multiplied by the width δ . At begin-frame, these pixel-variables are initialized to zero.

Most of these instructions map in a straightforward manner onto renderer macros. The exception is the multiply-accumulate operation, for which there is no corresponding renderer instruction. To perform the bitwise signed multiply-accumulate for each primitive, I modified the multiply-accumulate utility that PPHIGS uses at end-of-frame for lighting computations. This bitwise operation generates so many instructions that the instruction buffer must be flushed before beginning the fixed-width calculation, which makes the calculation somewhat slower: polygons with fixed-width intersections are processed 30% slower than polygons with varying-width intersections.

The chief drawback is that the technique extrapolates an intersection based on local linear approximations of the surface. When the neighborhoods of the sampled pixels really do intersect, the technique works very well. But the local approximation can also predict an intersection even when one does not occur, especially if one polygon is nearly edge-on to the eye and the other one is perpendicular to it. That creates spurious curves on the surface. These curves follow the silhouette of the layer of surface behind the frontmost layer. This is not a terrible price to pay. I have argued that enhanced intersection and silhouette curves are

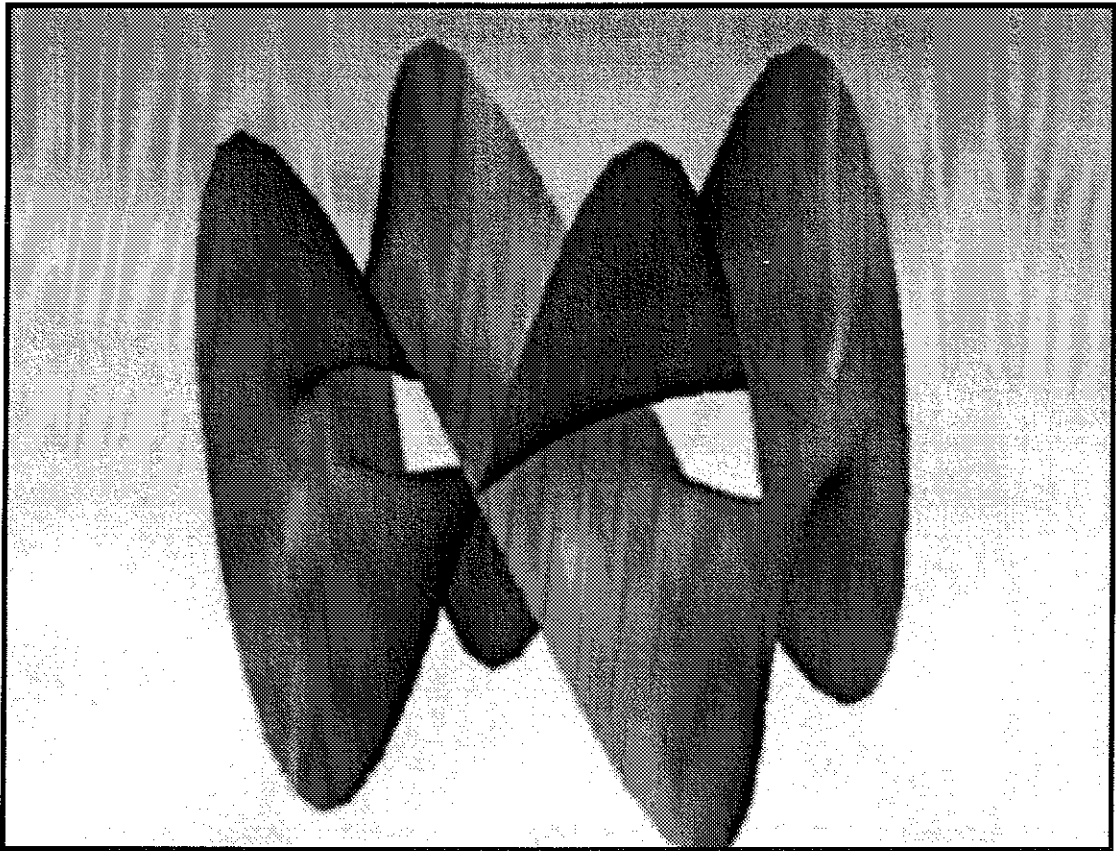
important to compute and to display when one uses transparency. The techniques for computing fixed-width and variable-width intersection curves may inadvertently capture silhouettes as well. But if the silhouettes are also calculated, they can be highlighted anyway.

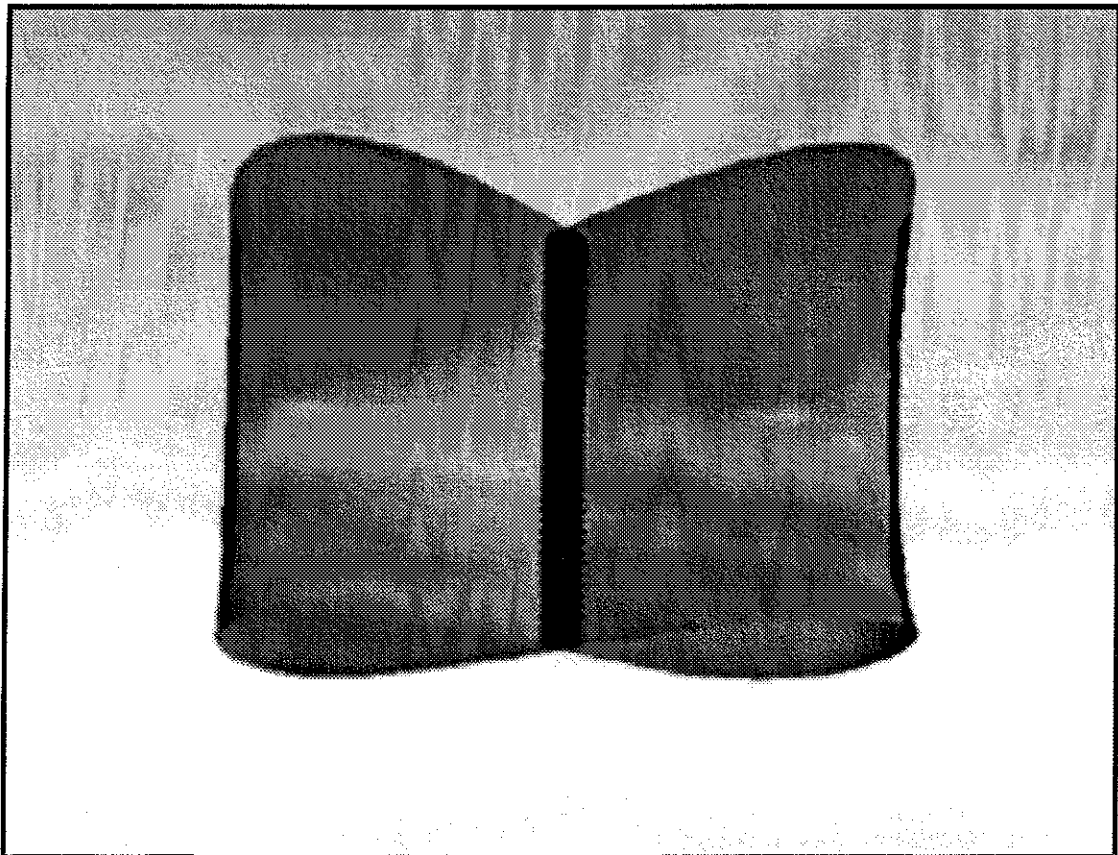
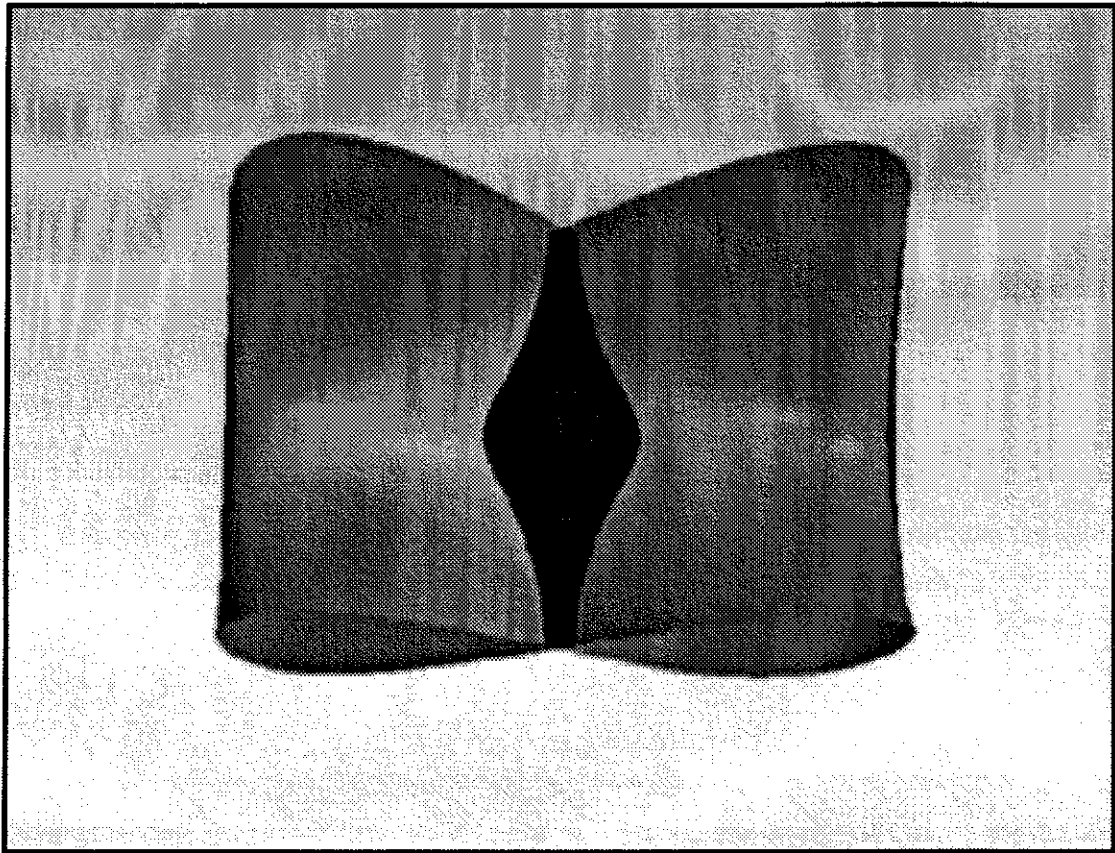
In summary, it is practical to calculate surface-intersections in screen space. This is a linear-time operation, which is important for surfaces in 4-space whose 3-space geometry can be full of intersections that change shape from frame to frame. Detecting intersections makes it possible to enhance their appearance, either by changing their color, their opacity, or their width. Changing their color or opacity compensates for the effect of transparency.

Changing their width makes them quite visible on the screen, but is difficult to fix the width of the thickened curve without pixel-to-pixel communication. The local linear approximations to the patches that intersect makes it possible to estimate the distance to an intersection at each pixel, calculated on a per-polygon basis. [Plates 3.6 A, C]

Plates 3.6

- A** The surface $(\sin(s), t/5, \sin(2s)(1 + t^2)/10, 0)$. Cross sections of this surface make figure-eights in 3-space. These figure-eights are thin in the middle of the surface and fat at the near and far ends of the surface. The intersection curve they produce forms a straight line.
- B** The intersecting patches have varying slopes along the intersection. The slopes are steep toward the endpoints of the intersection curve, but shallow in the middle. Thresholding based on depth alone produces a varying-width intersection curve (highlighted in black). Notice that the top and bottom silhouettes become intersection curves too, since the polygons along the silhouette "intersect" along their edges.
- C** Thresholding based on depth and slope compensates for the changing shape of the surface to produce a fixed-width intersection curve.





4 Interactive Algorithms

In order to classify a given compact surface by using Fourphront, the goal is to determine its genus and its orientability. The interactive tools for this task include rotations and translations, transparency, clipping, intersection highlights, silhouette highlights, and paint. The question is: how can these tools be applied in order to classify the surface? This section describes how a user can discover which surface he is looking at by using the tools that Fourphront provides.

In brief, one tries to paint two colors on a surface's different sides, succeeding exactly when the surface in fact possesses two sides. That determines orientability. Then one examines the level curves that are created by moving the clipping volume through the surface. The number and kinds of critical points on the level curves can be used to determine the Euler characteristic of the surface and hence its genus. These interactions are often easy to carry out, but there are several hazards. The chief hazard is the presence of intersection curves in 4-space. The next sections describe how the user can determine if the surface in 4-space self-intersects, whether it is orientable, and what its genus is.

There is a conflict lurking between the language of mathematics and the common practice of computer graphics that threatens to make these discussions overly detailed to the computer scientist or else overly sloppy to the mathematician. The conflict is about polygons.

Fourphront, like many graphics systems, uses polygons as primitives. These polygons approximate the underlying surfaces of interest. Because my primary audience is the computer graphics community, I will use the vocabulary of triangles and polygons rather than neighborhoods and patches. In general the distinction is not important as long as there are enough polygons in a geometric model to faithfully capture the shape of an underlying surface.

4.1 Intersections in 4-space

How can a Fourphront user determine whether a surface is imbedded in 4-space? The surface is (trivially) imbedded in 4-space if its projection is imbedded in 3-space. But a surface in 4-space typically has self-intersections when it is projected to 3-space. Any 3-

space rotation (in the space spanned by R_{xy} , R_{yz} , and R_{xz}) preserves the intersection curves on the surface in 3-space. When the user rotates the surface in 4-space however, an intersection in 3-space may move with respect to the surface or may even disappear. Consider a polygon P that intersects an intervening polygon I so that that I occludes part of P , in the neighborhood of its intersection with I , as well as part of P 's neighboring polygon U in that neighborhood (figure 4-1). Rotation in 4-space may reveal the previously-hidden parts of both P and its neighbor U in 3-space, because the rotation can move P and I away from any shared plane containing the eye. That situation is impossible in 3-space, but in four dimensions a plane and the eye point define a 3-dimensional subspace of 4-space. There is no guarantee that an arbitrary polygon I will lie in that subspace, and in fact it generally will not do so. Therefore it is possible for the user to decide whether an intersection in 3-space represents a genuine intersection of the surface in 4-space, or whether instead it is an artifact of the projection.

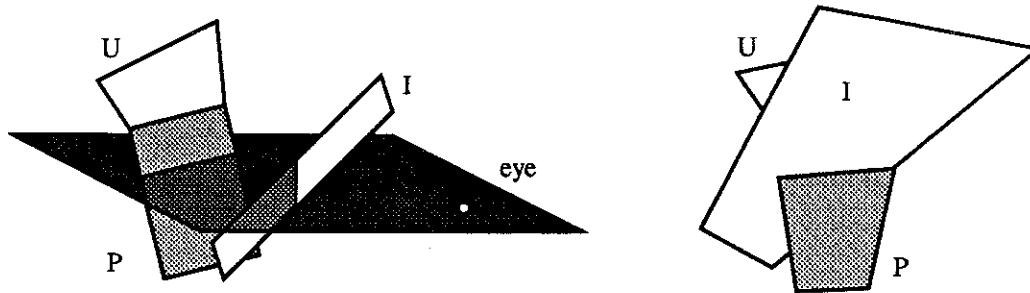


Figure 4-1. Two polygons in 4-space (left) intersect a plane which will project to a line in 3-space (right). As a result, the projected polygons will intersect in 3-space.

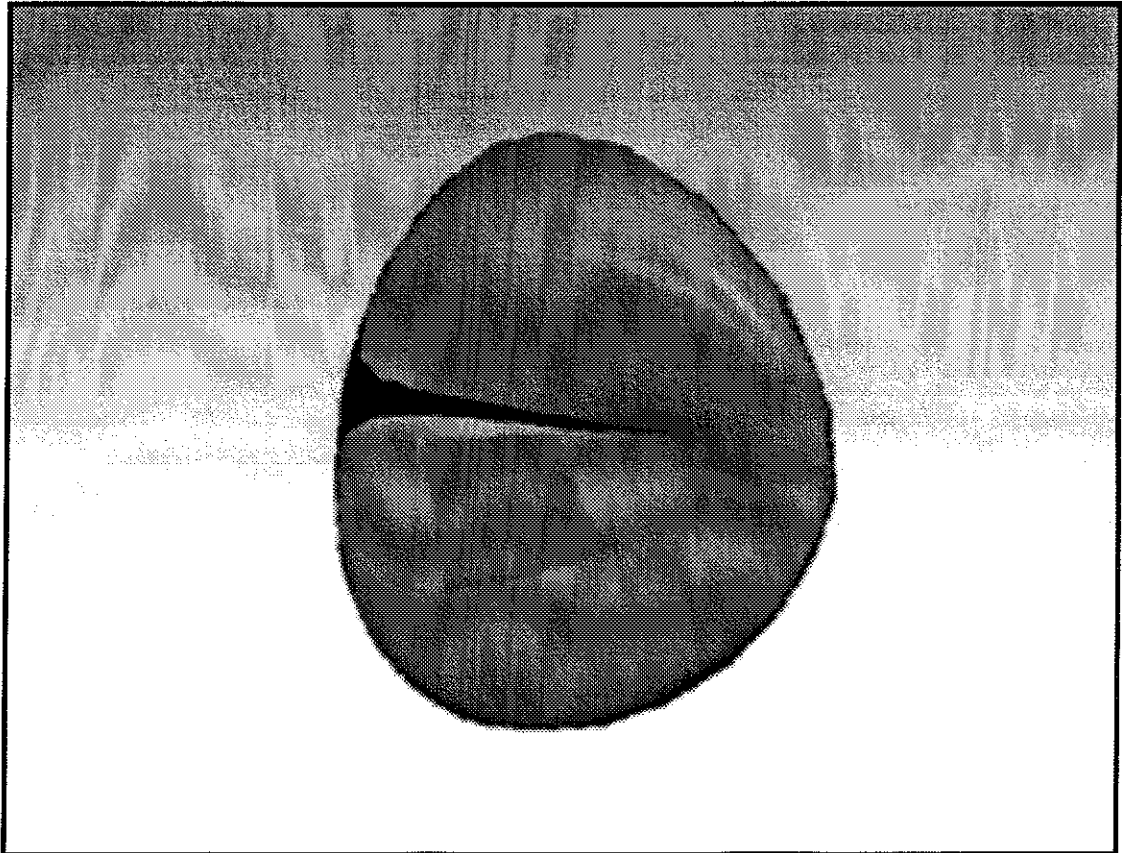
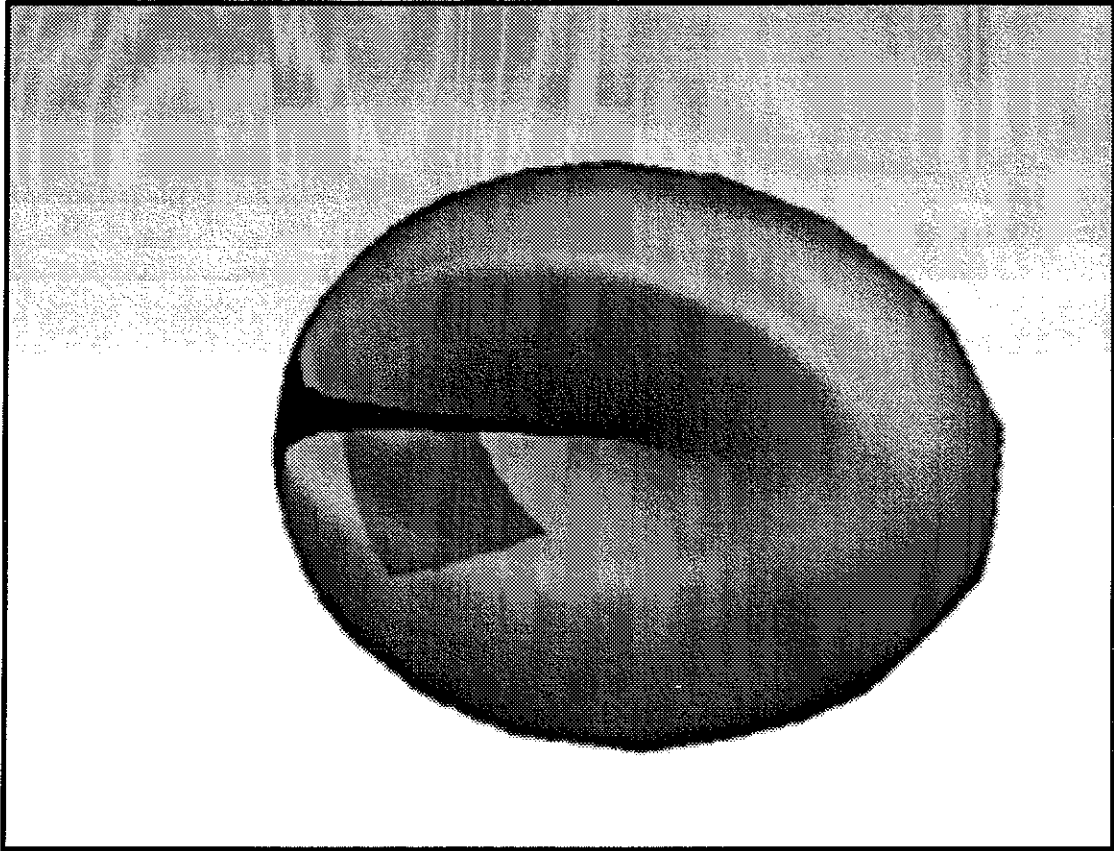
If the polygons actually intersect in 4-space, every rotation preserves the projected intersection. Conversely, if every rotation preserves the projected intersection, the intersection must exist in 4-space. In the 3-dimensional family of rotations that involve the fourth axis (the space spanned by R_{xw} , R_{yw} , and R_{zw}), how large is the space of rotations that preserve the projected intersection between two polygons that do not intersect in 4-space? After all, the larger that space of rotations is the more likely a user will apply a suitable rotation that moves the intersection away from the participating polygons in 3-space.

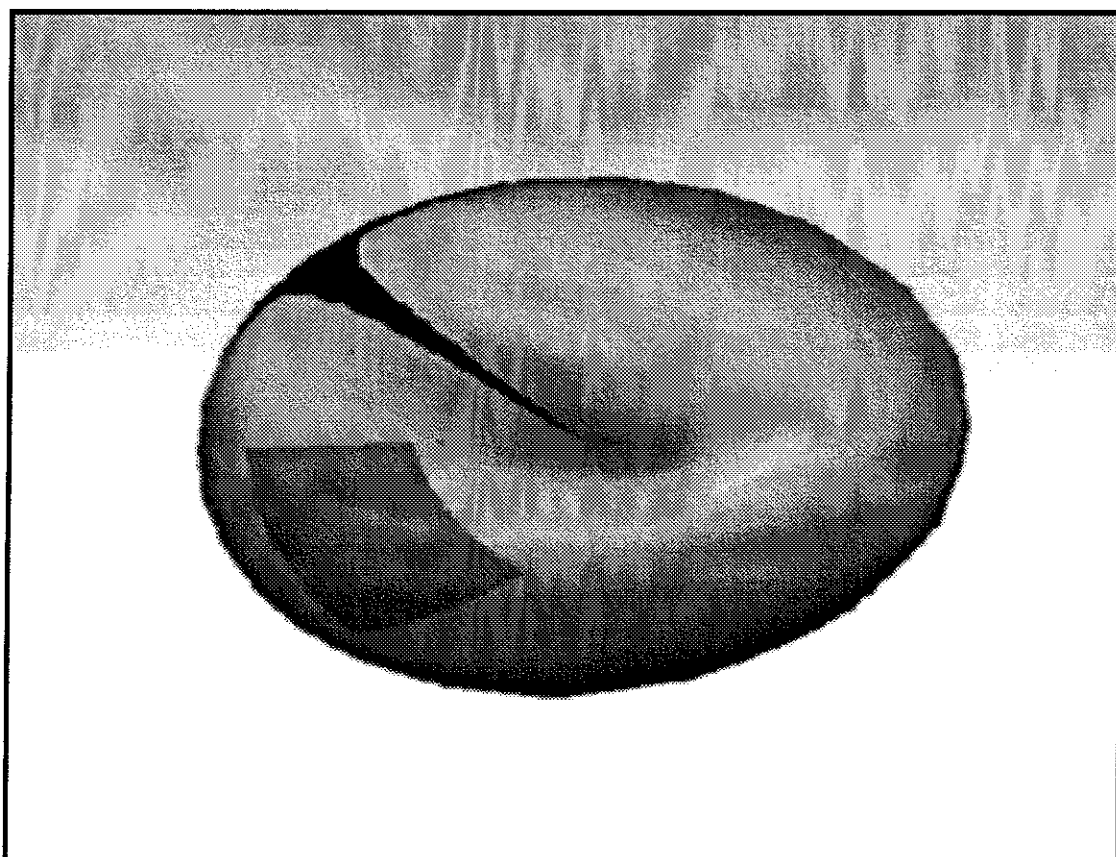
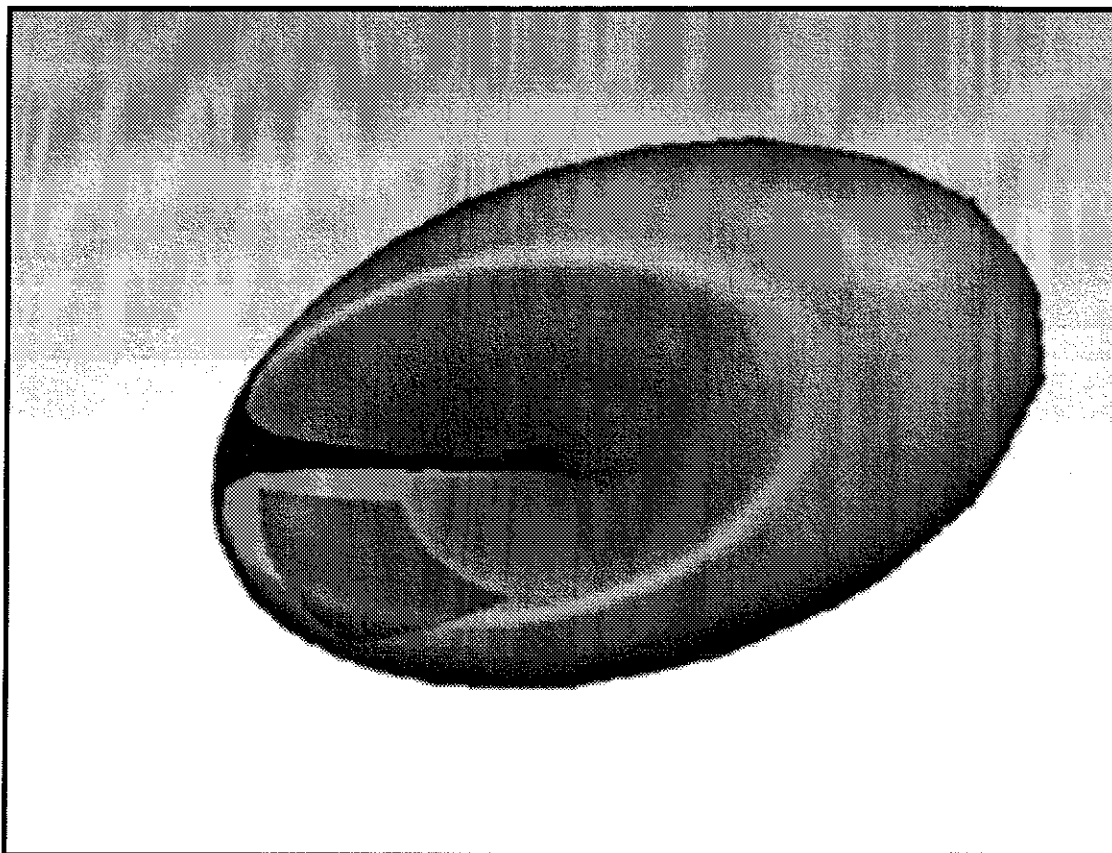
If the polygons do not actually intersect in 4-space, the preimage of the 3-space intersection defines a plane in 4-space containing the eye (located on the w -axis), the preimage of the intersection at a point in polygon P , and the preimage of the intersection at a point in polygon I . In order to preserve their 3-space intersection, the preimages of the intersection must remain on a plane containing the eye, which occurs precisely when that plane is the rotation plane. Any other 4-space rotation will move that plane away from the eye.

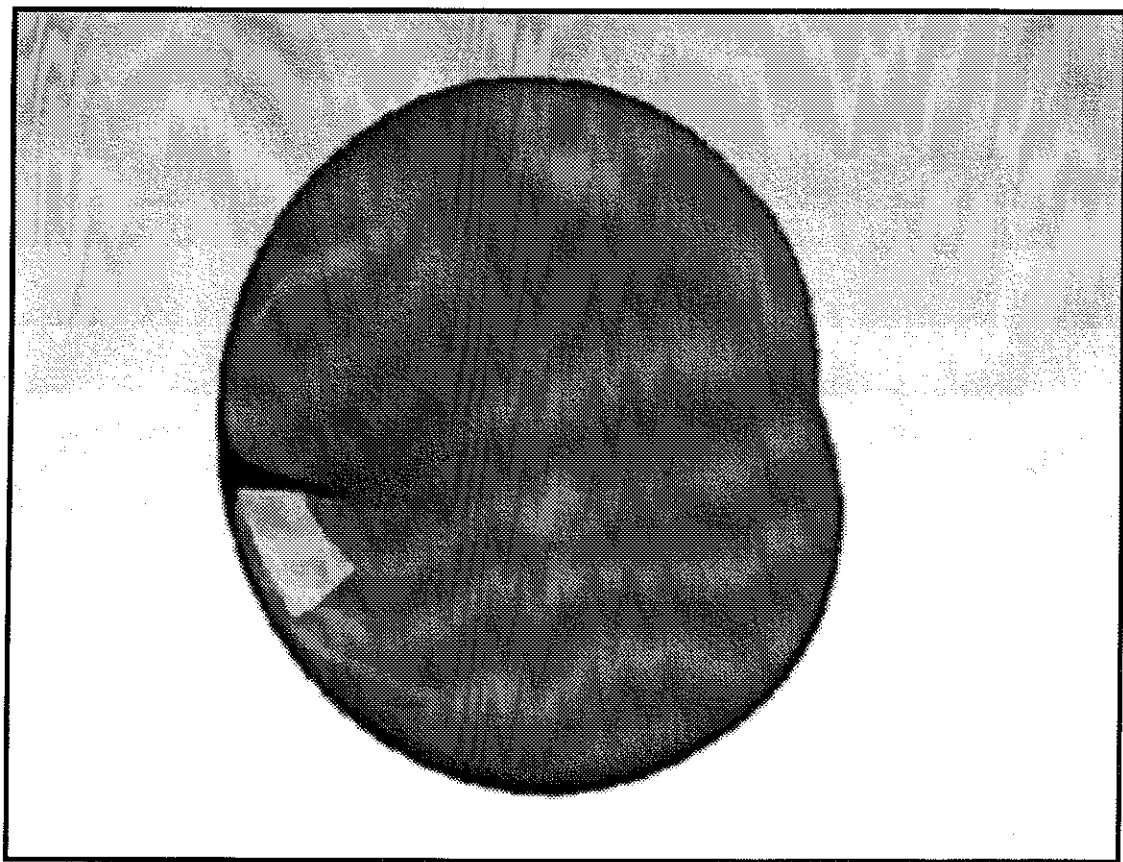
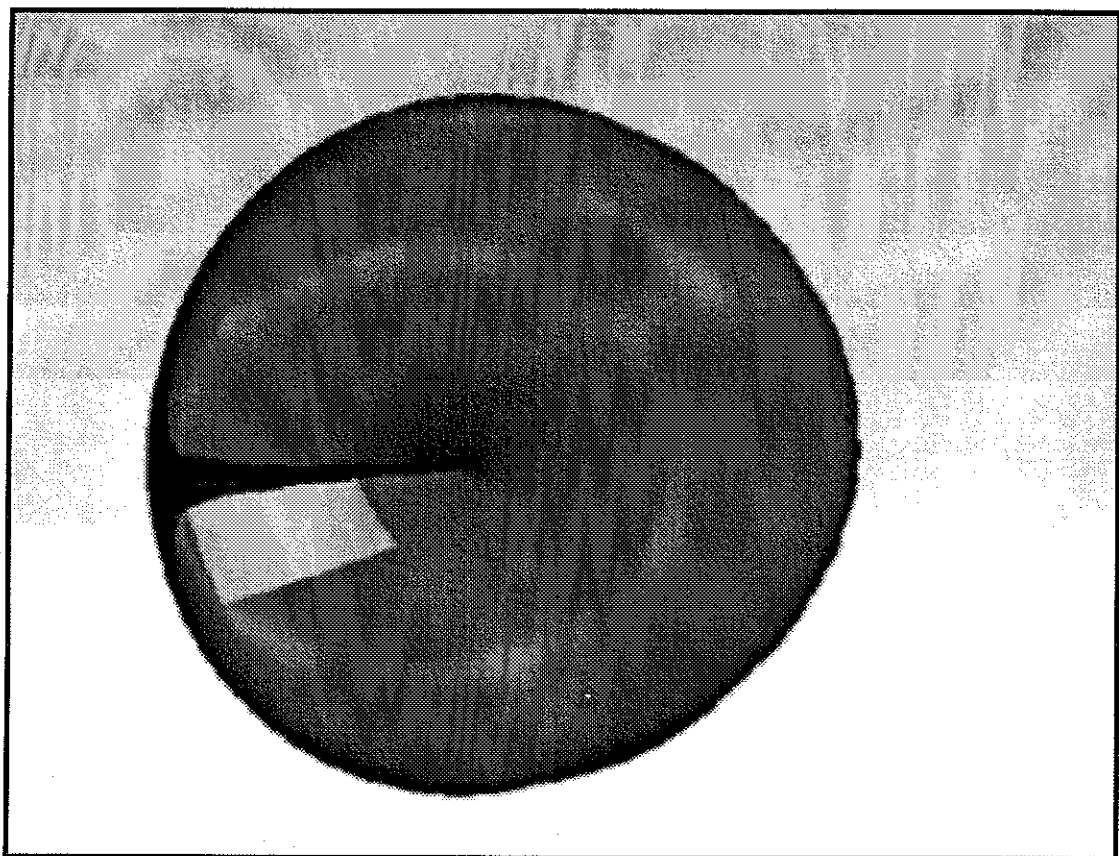
Consequently, almost every rotation that involves the w -axis will move the intersection curve with respect to the participating polygons in 3-space. That means that the user can easily distinguish between intersections in 3-space that are inherited from 4-space, and intersections that are an artifact of projection to 3-space. Genuine 4-space intersections never move from their participating polygons after projection to 3-space. On the other hand, intersections that result from projection to 3-space generally move with respect to their neighboring polygons when the user applies a 4-space rotation. [Plates 4.1 A-H] This means that in practice the user can really expect to determine when a surface self-intersects in 4-space, and hence when a surface is or is not imbedded in 4-space.

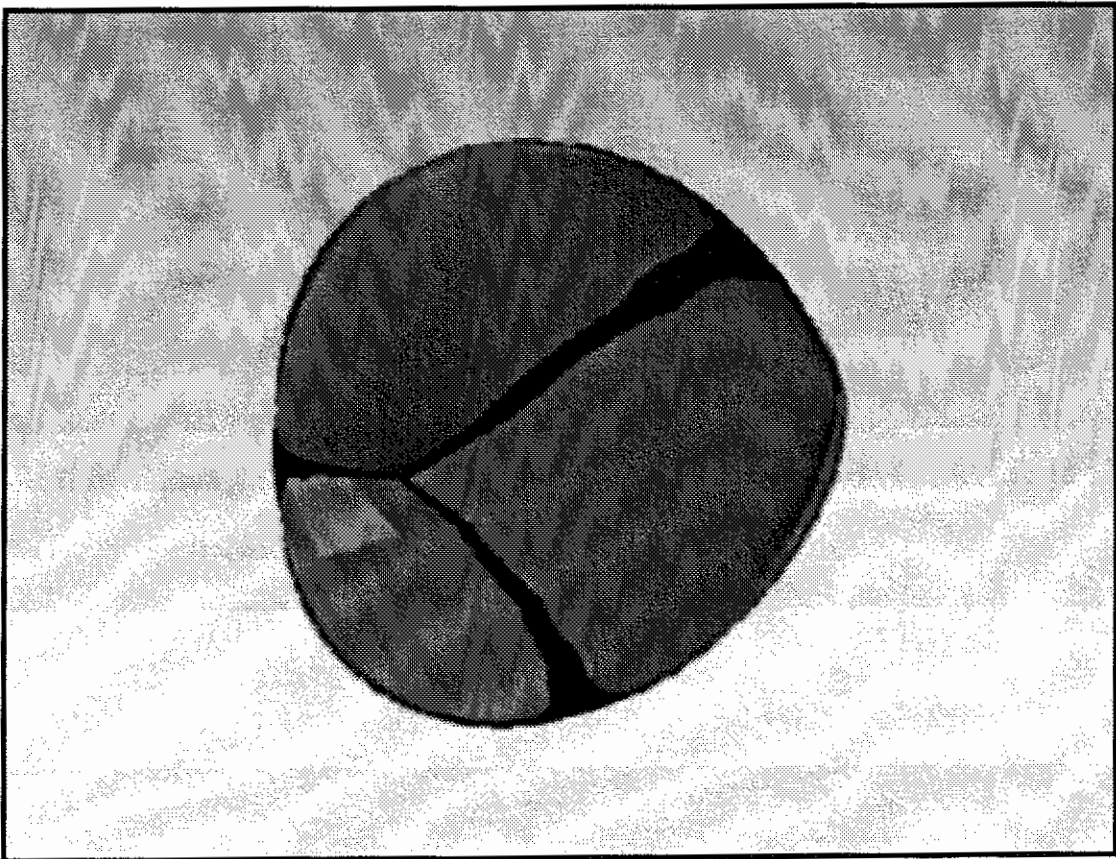
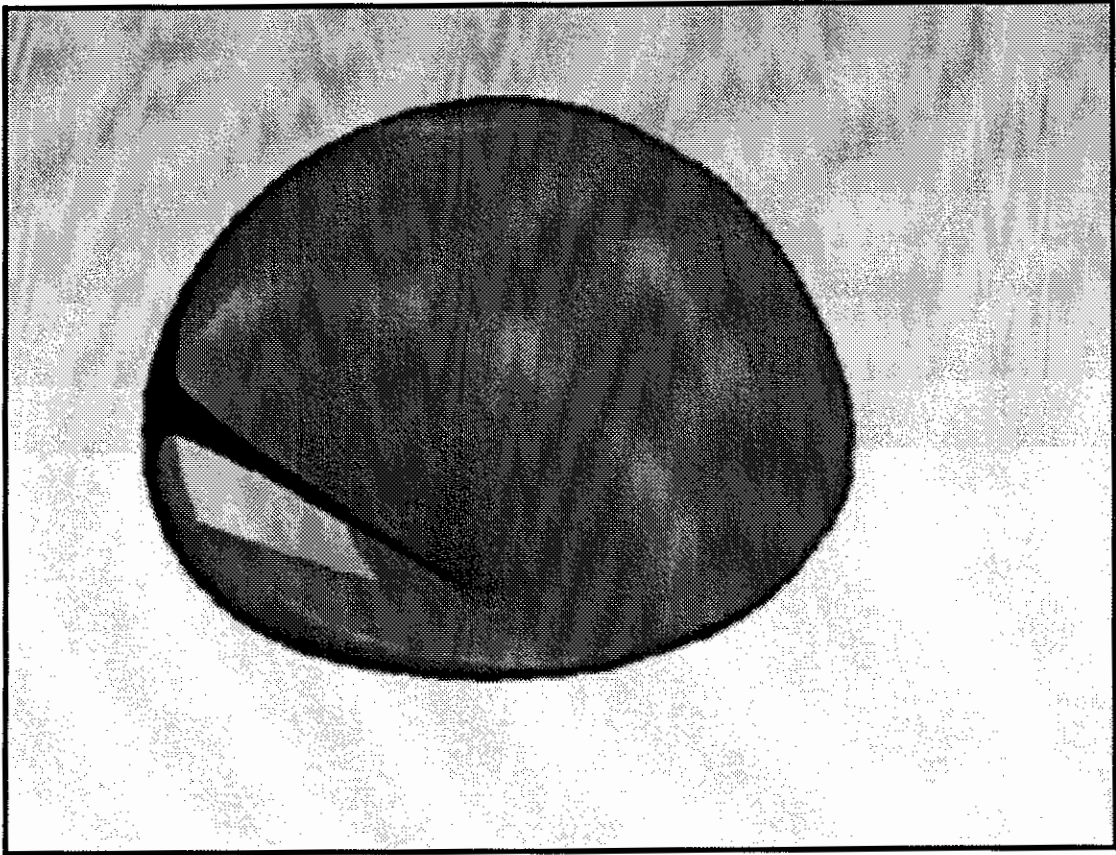
Plates 4.1

- A** The surface (X^2-Y^2, XY, XZ, YZ) in 4-space, where (X, Y, Z) is a point on the sphere S^2 in 3-space. In this projection, the surface makes a cross-cap in 3-space. A neighborhood (on the left) of the intersection has been marked with paint.
- B** This is the result of rotating the surface **(A)** in the xw -plane (in these examples, the x -axis is horizontal, the y -axis is vertical, and both the z - and w -axes point out from the image). The painted patch and the intersection have moved relative to each other in the projected image.
- C** This is the result of rotating **(A)** in the yw -plane. Again the paint and the intersection separate in 3-space.
- D** This is the result of rotating **(A)** in the zw -plane. Each different rotation has separated the paint and the intersection, which is the usual case when the intersection is an artifact of projection and does not exist in the actual surface in 4-space.
- E** The surface $(\cos(s) \sin(t/2), \sin(s) \sin(t/2), \cos(2s) \sin(t/4), -\cos(t/2))$. The intersection curve in the cross-cap is inherited from the surface as it lies in 4-space.
- F** Rotating **(E)** in the xw -plane preserves the adjacency of the paint and the intersection as they are projected to 3-space.
- G** Rotating **(E)** in the yw -plane still preserves the adjacency.
- H** Rotating **(E)** in the zw -plane preserves the adjacency. Since the intersection persists in every orientation, it exists in 4-space as well as 3-space. Notice that this rotation has even changed the topology of the projected surface: there are two new intersection curves near the painted patch.









4.2 Orientability

If a surface in 4-space projects to 3-space with no self-intersections, it is easy to determine how many sides it has simply by painting it (always extending the paint along surface pieces with a continuous tangent plane). One paints the outside with a single color, rotates the object in 4-space in order to expose the other side of the surface, and then paints the second side with a second color. A 4-space rotation by $\pi/2$ radians can expose the other side of the surface. To see why, consider what happens to three unit vectors \mathbf{t} , \mathbf{u} , and \mathbf{v} after applying such a rotation R in the wx -plane.

$$\mathbf{t} = (a^2, 0, 0, b^2), \quad \mathbf{u} = (0, 1, 0, 0), \quad \mathbf{v} = (0, 0, 1, 0)$$

$$R\mathbf{t} = (-b^2, 0, 0, a^2), \quad R\mathbf{u} = (0, 1, 0, 0), \quad R\mathbf{v} = (0, 0, 1, 0)$$

A perspective projection P from the point $(0, 0, 0, 1)$ produces the following vectors in 3-space.

$$P\mathbf{t} = (a^2/(1-b^2), 0, 0), \quad P\mathbf{u} = (0, 1, 0), \quad P\mathbf{v} = (0, 0, 1)$$

$$PR\mathbf{t} = (-b^2/(1-b^2), 0, 0), \quad PR\mathbf{u} = (0, 1, 0), \quad PR\mathbf{v} = (0, 0, 1)$$

Notice that the first triple forms a right-handed coordinate system, while the second forms a left-handed coordinate system. Hence outward normals become inward, and the local patch turns “inside out.” When all the patches of the surface are painted, the surface has been demonstrated to be 2-sided. This example is actually too trivial. If the surface is intersection-free in 3-space, one knows immediately that it must be 2-sided: a non-orientable surface does not imbed in 3-space.

The painting algorithm begins when the user selects a single polygon on the surface to paint (on one side only) with a single color. The user applies more paint to the neighboring polygons, incrementally extending or expanding the painted region. He continues to apply paint as long as he sees any painted polygon that is bordered by an unpainted polygon. This simple algorithm is applied to polygons as they appear on the screen. There are two ways that the user can be prevented from painting an unpainted polygon U that borders a painted polygon P . The first way is for them to be separated from each other by a silhouette. In particular, if the (front-facing) painted polygon P lies on the edge of the visible part of the surface and its unpainted (back-facing) neighbor U lies behind, the user does not have access to the neighbor. U is hidden from view. Once the user rotates the object a little, the back-facing neighbor U crosses the silhouette and comes into view together with P . The

user can then visually determine that it abuts the painted polygon, and he can proceed to paint it too.

The second way that the user can fail to paint the neighboring polygon is for that neighbor to be occluded by some intervening polygon I in the surface. As long as the occlusion is not caused by an intersection in 4-space, the user can rotate the surface in order to expose both the painted polygon P and its unpainted neighbor U (section 4.1). But the surface may truly intersect itself in 4-space: assume polygon I intersects polygons P and U along their shared edge in 4-space (figure 4-2). The resulting intersection in 3-space is preserved regardless of the surface's orientation. No matter how the user rotates the object, part of I hides P near the intersection, or else part of I hides U near the intersection. (The only exceptional case is when polygon I is edge-on to the eye. If there were only three polygons in the scene, the user could finally see the P and U abut. But recall now that there are many polygons in the shape of a closed surface. Polygon I may be edge on, but its neighbors still occlude either P or U .) The user cannot witness that P and U abut along their shared intersection, because I contains that intersection itself. This intersection between I and the pair $\{P, U\}$ effectively changes the topology of the surface from the user's point of view: neighboring patches are no longer demonstrably neighboring, because they cannot be simultaneously exposed on the screen. That changes the fundamental group of the (3-space projection of the) surface, since a loop on the surface that encloses the intersection cannot contract through it. The user may conclude incorrectly that there are two sides on a 1-sided surface, because he can paint with two colors such that the colors only meet along an intersection curve in 3-space. That means, from his point of view, that they do not meet at all.

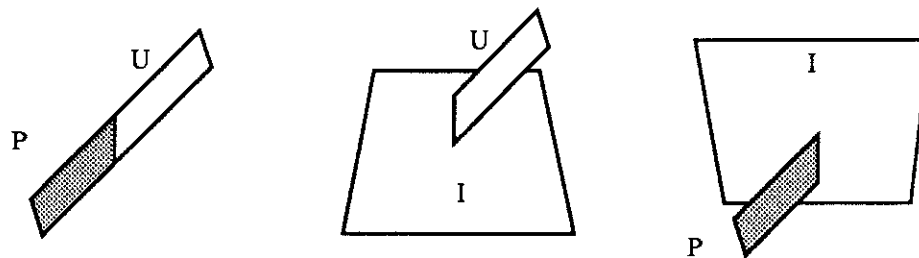


Figure 4-2. A painted polygon P and an unpainted polygon U share a common edge (left). An intervening polygon I intersects P and U along their shared edge, hiding P (middle). Rotation reveals P , but now I hides U (right).

A simple remedy for this problem is to use clipping to expose both the painted visible polygon P and the unpainted neighbor U , which is hidden by some intervening polygon I that intersects P and/or U . When P is parallel to the clipping plane, the user can move the

clipping plane closer and closer to P . When the plane nearly coincides with P , it clips away the part of I that hides U , thereby exposing the neighbor U . [Plates 4.2 A-E]

These observations lead to a general technique for determining whether a surface in 4-space is 1-sided or 2-sided. First, the user paints polygons that border the intersection curves on the surface. Then he rotates the surface in 4-space in order to observe whether the intersection curves move with respect to their neighboring painted polygons. Wherever that happens, the user unpaints those polygons, until all that is left painted are the polygons along intersections in 4-space. These are the intersections where he must take special care when painting. After isolating the 4-space intersections, the user chooses a starting point to begin painting the "first" side. He incrementally extends paint outward from the starting point, rotating the 3-space intersections out of the way and clipping across 4-space intersections, until he can paint no more. He can rotate the surface inside-out (by a half-turn in 4-space) to determine whether there are any unpainted polygons left; if so, they lie on the "second" side. If not, the surface is 1-sided.

There are variations to the painting technique that make the process go faster. Define a 3D-visible component of the surface to be a region in 3-space that is bounded by intersection curves (or else the entire surface if there are no intersections). These components define the regions which might be individually viewed and painted in 3-space. An intersection curve may itself cross other intersection curves, which divide it naturally into arcs. A 3D-visible component is like a curved polygon whose edges are arcs along intersection curves. The user doesn't actually need to paint every polygon within a 3D-visible component; he just needs to paint enough to determine if any component is painted inconsistently (painted on two sides). After he has painted across each arc of the intersection curves, the user has sufficiently painted one side of the surface. Only the 3-space rotations are necessary. If the user rotates the surface in 4-space, he changes the shape of the 3D-visible components in the 3-space projection of the surface, sometimes creating new ones or merging old ones. So in order to find whether the back side of a 3D-visible component is painted, he can make the surface semi-transparent and inspect the components by only using rigid 3-space transformations. The surface is 1-sided exactly when at least one 3D-visible component is painted on its "front" and "back" sides.

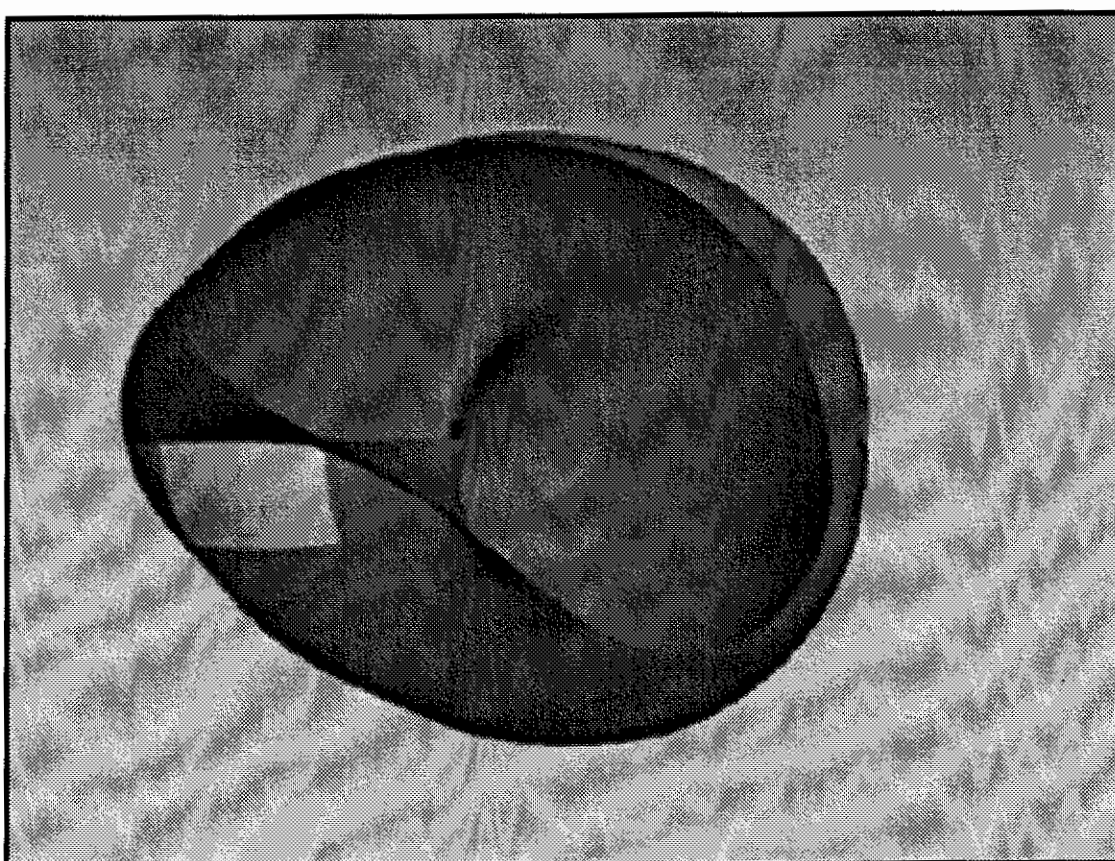
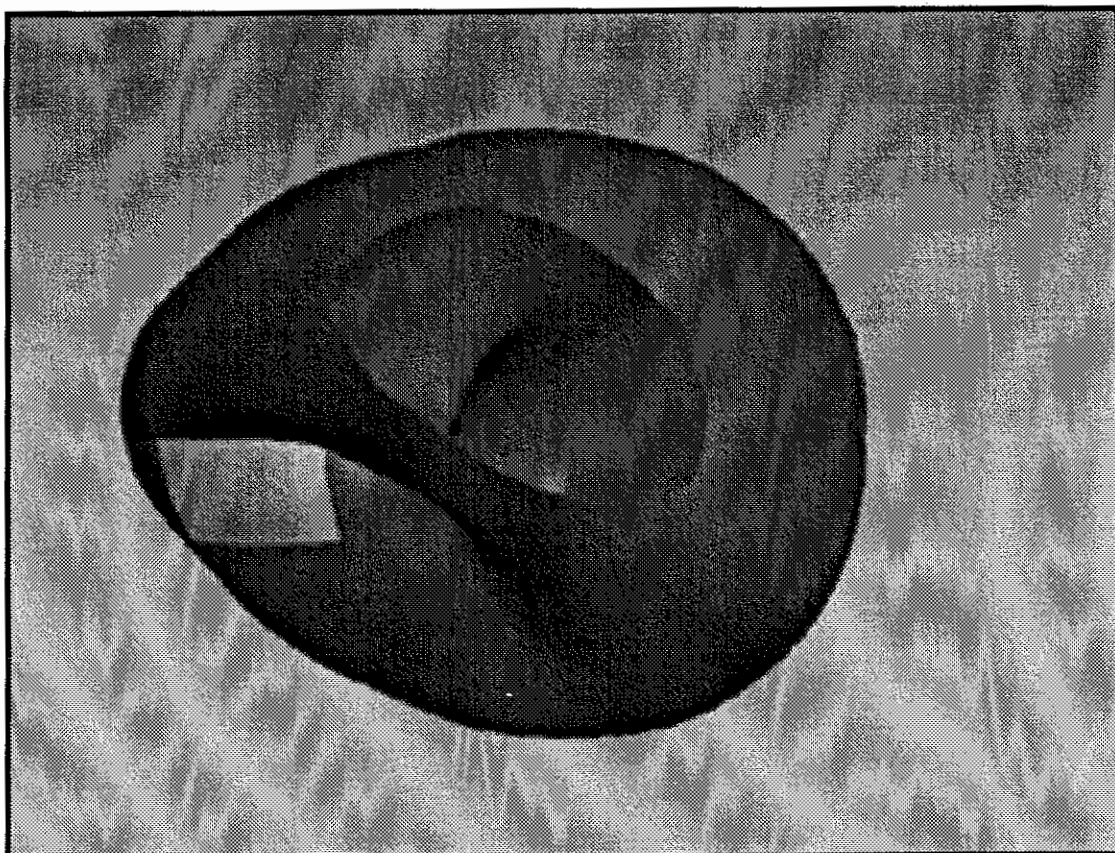
Another method for verifying a 1-sided surface begins with painting a single polygon two different colors on its two sides. This can be done by painting one face of a polygon, then rotating the surface in 3-space so that the surface carries the polygon to the back. The user then turns on transparency so that he can locate the painted polygon, then he clips away the

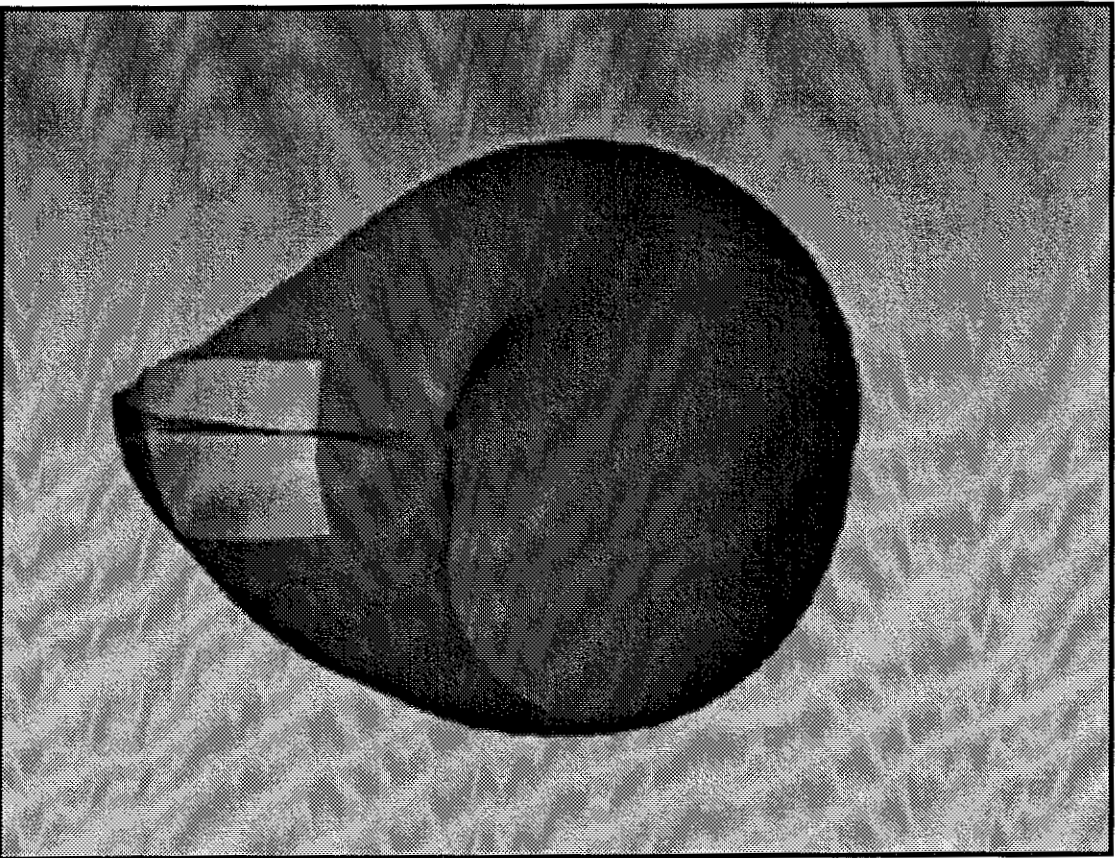
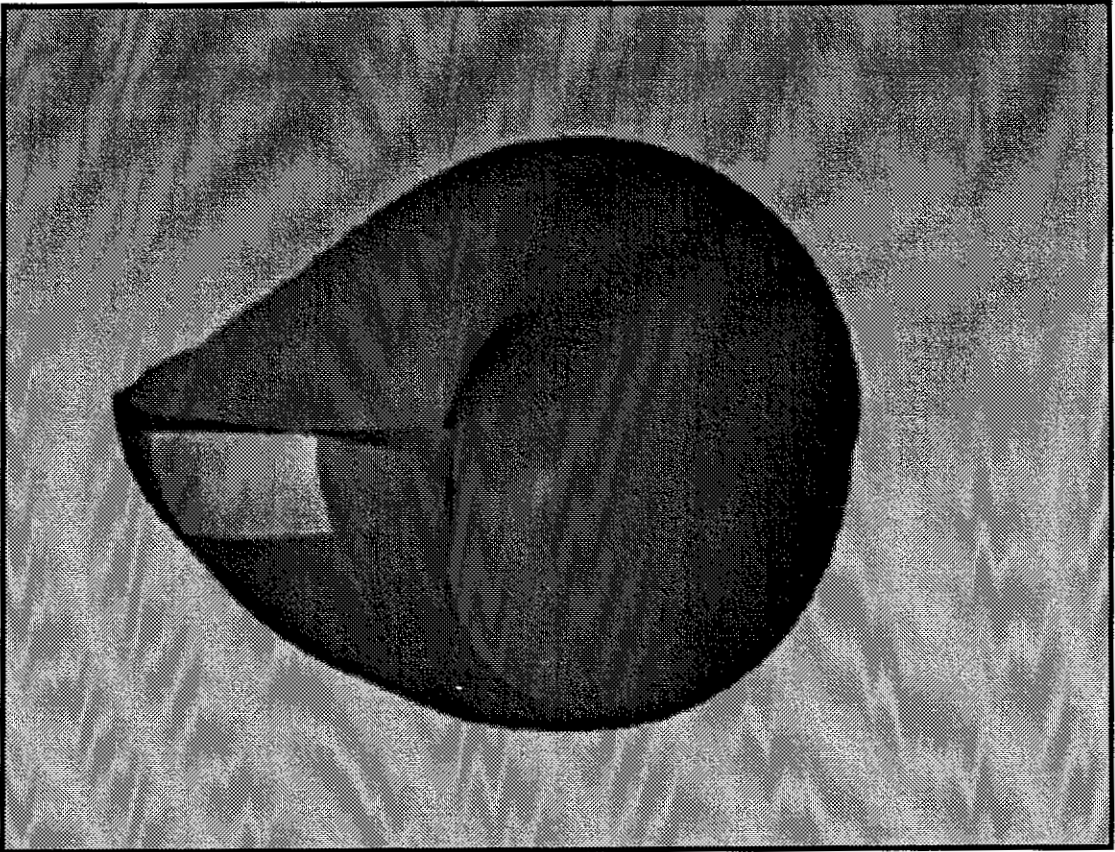
surface until he can paint the back of the polygon with the other color. With the clipping plane retracted and with the surface again opaque, the user can alternately extend either of the two painted regions. When two colors abut (or even if they share a 3D-visible component), he has verified that the surface is 1-sided.

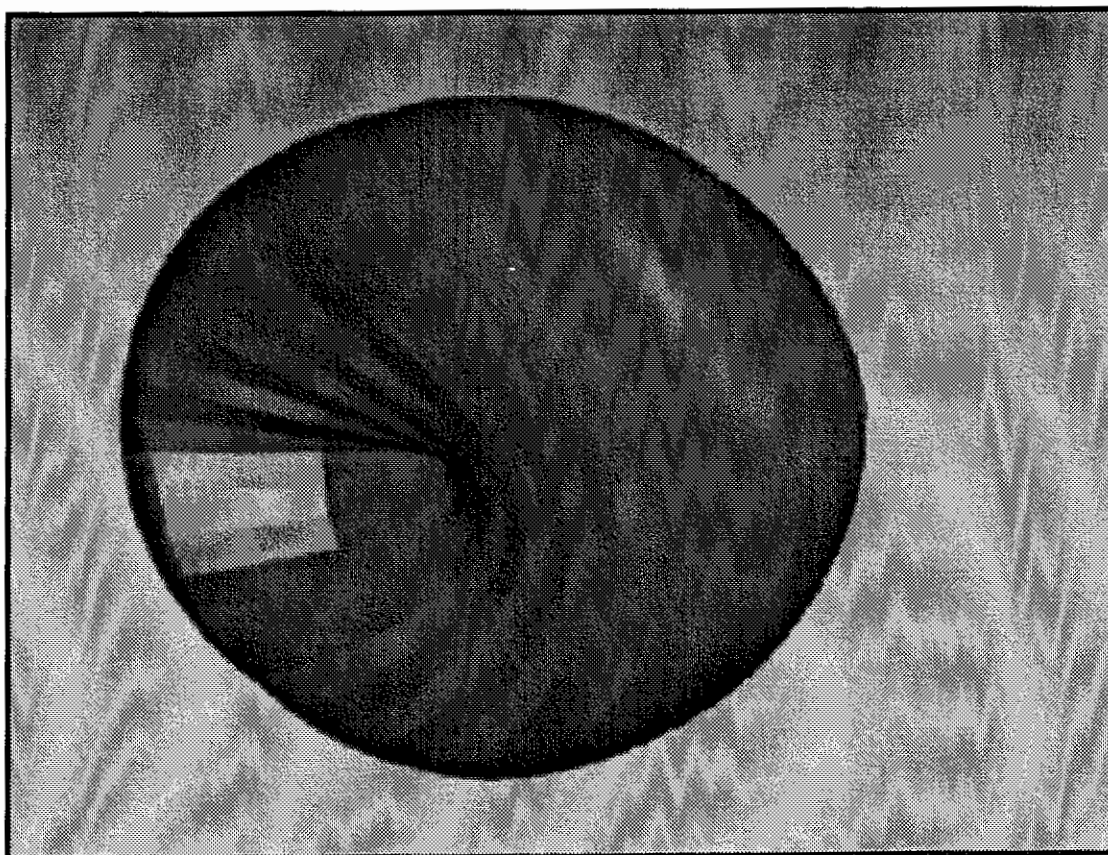
Another variation is useful when the user suspects he has found a Möbius band within the surface. With transparency turned on, the user begins painting polygons along the closed path using a single color. When he finds he is painting a polygon whose other side is already painted (as revealed by transparency), he knows the surface is 1-sided because he has painted a Möbius band within it.

Plates 4.2

- A** The surface in plate 4.2 (E) has a true intersection in 4-space. To paint across it, the user clips into the surface toward the intersection.
- B** The clipping plane has nearly reached the intersection at the painted patch.
- C** Clipping now exposes that the patch continues through the intersection.
- D** Paint is applied to the neighborhood to the right of the patch.
- E** Transparency and ribboning show the painted region as it penetrates the rest of the surface.







4.3 Genus

How can Fourphront be used to determine the genus of a surface in 4-space? Sometimes the number of holes in a surface can be counted easily, just by looking at its image on the screen. But if the surface is complicated – knotted or self-intersecting – the number of holes may not be evident by mere inspection. The genus of a surface can also be found by counting the number of closed curves that can be cut in the surface without separating the surface into two disconnected pieces. Actually cutting and separating the surface becomes a modelling problem, which reaches considerably beyond the scope of this dissertation. Similarly a user could attempt to determine how loops combine and contract in order to visually determine the fundamental group of the surface. Again, this idea goes far beyond my scope.

An easier technique uses Morse Theory. Morse Theory applies calculus locally on the surface in order to determine globally what the topology is of the surface. According to Morse Theory, a differentiable surface can be endowed with a height field (the Morse function) whose critical points are nondegenerate (each has a non-singular Hessian) and are isolated to form a finite set. At each critical point, the surface is either concave upward, saddle-shaped, or concave downward with respect to the function. These types of points have index 0, 1, and 2 respectively. The sum, alternating over index, of the cardinality of points of each index yields the Euler characteristic χ of the surface and hence the genus G (given by the formula $G = 1 - \chi/2$).

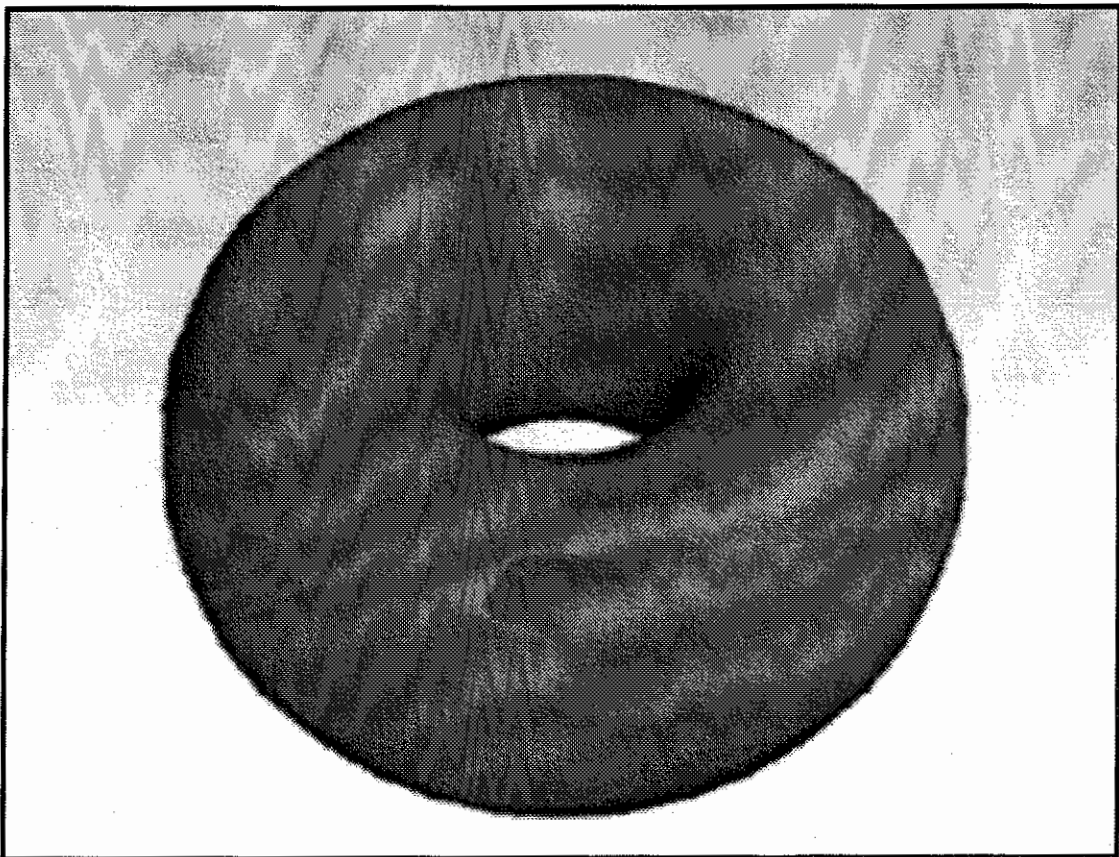
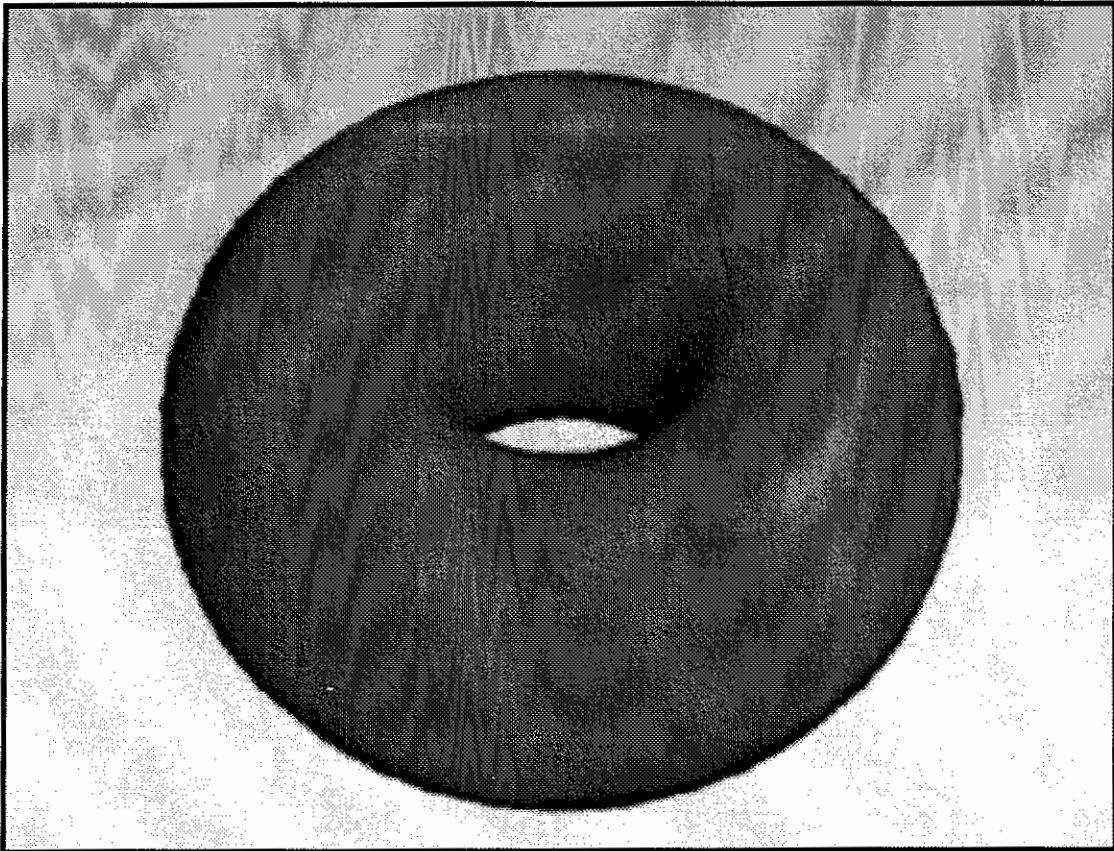
Morse Theory can be applied to interactive manipulations of the surface in the following way. One chooses as a height field the distance in the z -direction from the eye in 3-space. In order for this to be a Morse function, it may be necessary to rotate the surface to eliminate any high-order contact between the surface and the moving plane (but a very small change is sufficient). When the hither clipping plane moves through the surface, the critical points become evident along the level curves at the intersection of the surface and the clipping plane. They are the points where a closed curve is created or destroyed by the moving plane. The plates at the end of this section illustrate Morse Theory in action. The user sweeps the clipping plane through a surface. He locates critical points of index 0, 1, 1, and 2. Thus he finds 1 point of index 0, 2 of index 1, and 1 of index 2. The alternating sum of the cardinalities is $1-2+1=0$, and so the Euler characteristic of the surface is 0 and its genus is 1. [Plates 4.3 A-G] Note that it is highly unlikely that the user will precisely hit a critical point. The plates are truly indicative of that will be seen: it is a pair of before-and-after clips that reveals that the clipping plane has crossed a critical point.

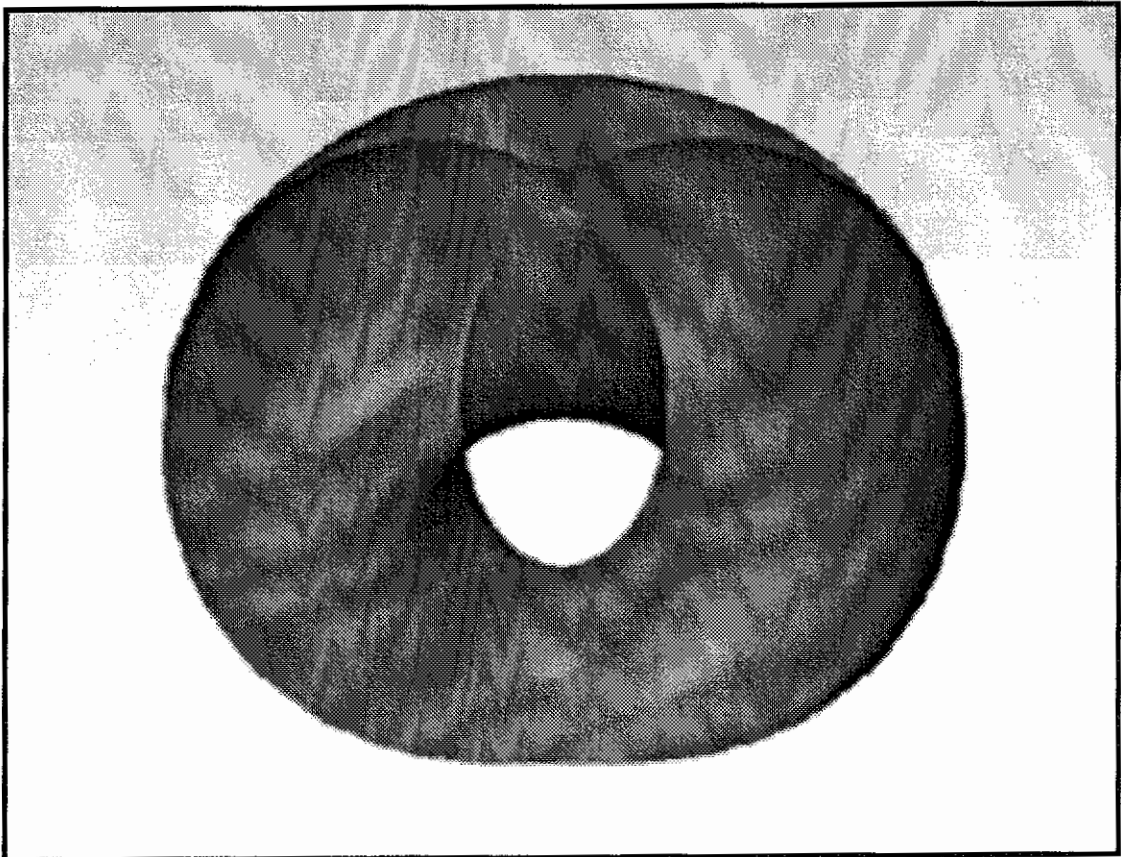
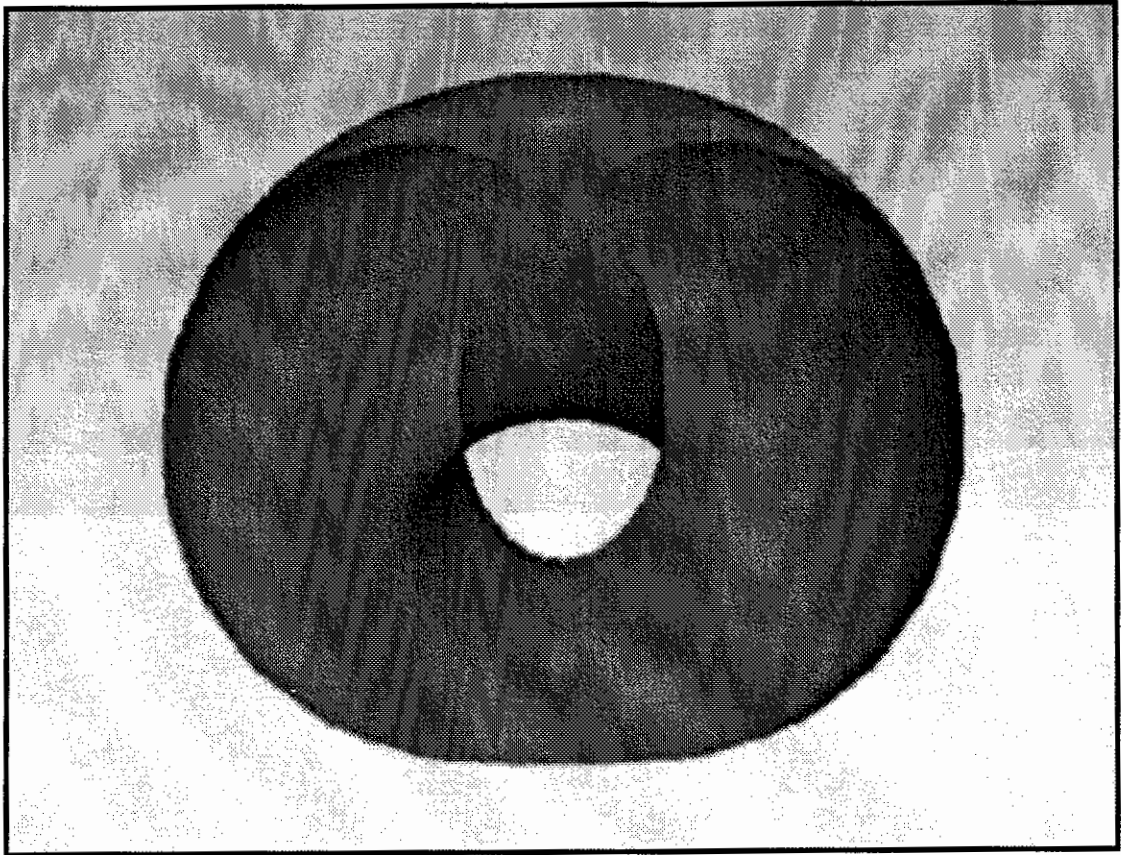
The user can thus select three colors of paint and apply them to the three different kinds of critical points as he moves the clipping plane through the surface. When he is finished, he counts the number of patches of each color and calculates their alternating sum to find the Euler characteristic.

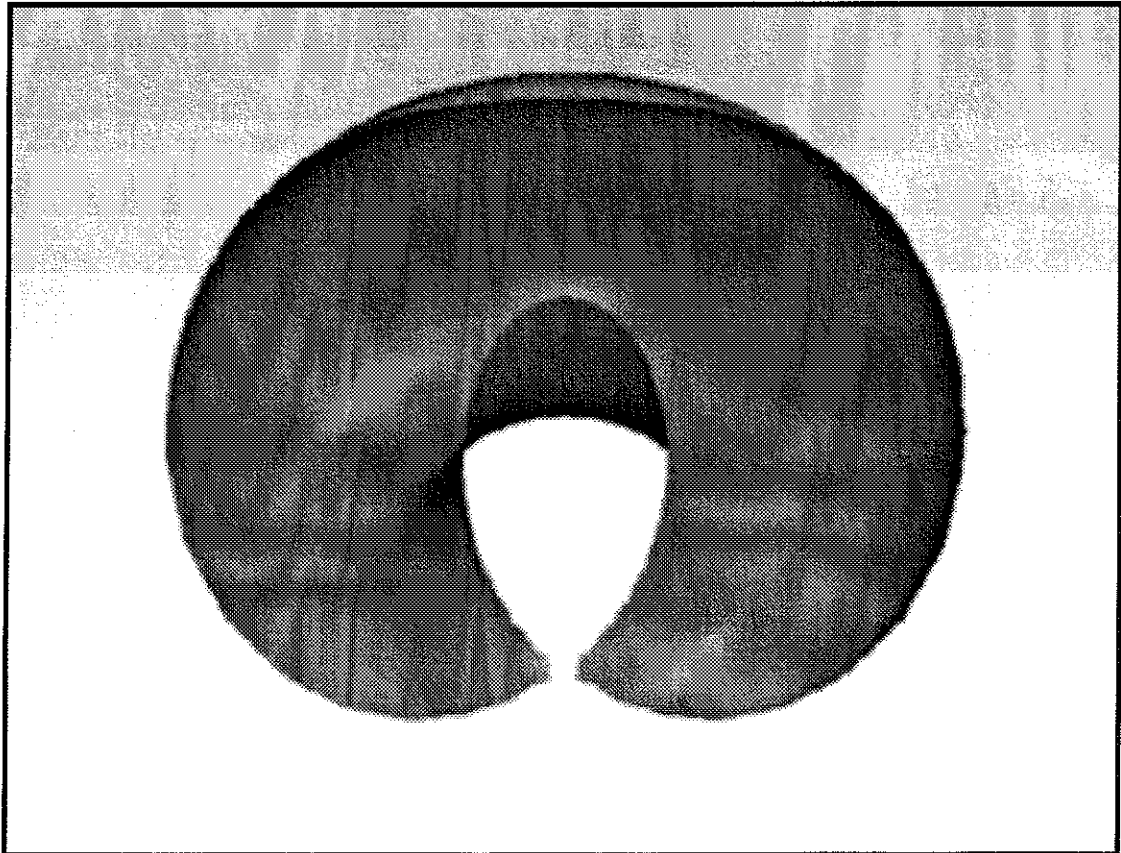
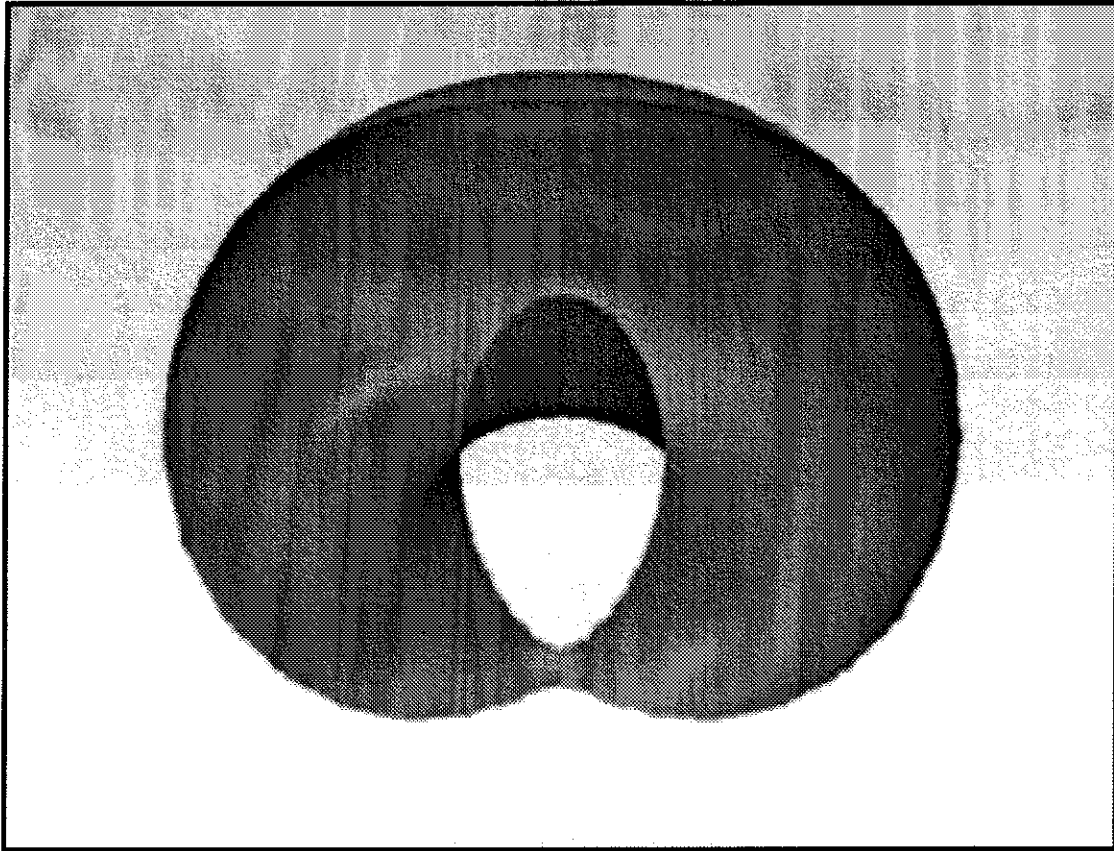
So there are indeed interactive user-directed algorithms that let one determine the orientability and the genus of a surface. These algorithms require rotation, clipping, painting, and transparency from the graphics system, and require that the user be able to see both sides of each polygon (or at least of each visible component) during the interactive session, locate intersection curves, and determine whether an intersection occurs in 4-space.

Plates 4.3

- A** The torus $(\cos s, \sin s, \cos t, \sin t)$ projected to 3-space.
- B** The clipping plane crosses a critical point of the height function on the torus. The level sets show a point that blooms into an ellipse-like shape.
- C** The clipping plane approaches another critical point.
- D** The clipping plane has just crossed the critical point, and the cross-section curve has separated into two curves.
- E** The clipping plane approaches another critical point.
- F** After the clipping plane crosses the critical point, the two level curves rejoin.
- G** At the last critical point, the visible part of the surface vanishes from an ellipse-like shape to a single point to empty space.









5 Observations of Interactive Sessions

In order to learn how well Fourphront communicates the experience of interacting in four dimensions, I scheduled a series of sessions to observe different users as they investigated surfaces using Fourphront. Nine of the users were senior undergraduates in their last month of taking a topology course. One user was a graduate student in computer science. One user was a graduate student in mathematics, finishing his dissertation on knots and links of surfaces in 4-space. Three users held doctorates in mathematics.

The interactive sessions lasted from 45 to 90 minutes each. I began each session by demonstrating how to use the joysticks to manipulate a torus, defined parametrically in 4-space as $(\cos s, \sin s, \cos t, \sin t)$. Then I let the subjects acquaint themselves with the joystick controls for about 5 minutes. Next I demonstrated how to vary the opacity with the slider bar and how to paint the surface with the spraypaint icon. Then I demonstrated the intersection and silhouette highlights. This introduction to Fourphront typically lasted for 15 to 20 minutes.

After the introduction was completed, I answered any questions that the user had and then I replaced the torus with a new surface on the screen. I asked the user to study the surface and tell me about its properties. The sections below describe these sessions and include comments that the users made while exploring the various surfaces. I have arranged these observations according to topic rather than by sequential order of the sessions. In the extended transcripts, I have labeled the speaker (*e.g.*, "Undergrad1") and the observer (myself).

5.1 Intersections

To determine whether an intersection is a part of the surface in 4-space or merely an artifact of projection to 3-space, a user paints a neighborhood of the intersection (section 4.1). Rotation in 4-space then reveals whether the intersection lies in 4-space (by preserving the adjacency of the paint to the intersection) or not (by moving the intersection with respect to the paint). This technique proved successful for all of the subjects I observed.

Undergrad1 rotated a self-intersecting projection of the torus and said, "It's kind of weird the way the intersections go around in the paint. You know, the paint sort of stays still, but the intersections sort of flow around it." Undergrad2 rotated a Klein bottle imbedded in 4-space and saw the same effect. He concluded, "So in four dimensions there's not really an intersection. That's wild. That's so hard to imagine, isn't it? Four dimensions? Beyond my brain."

Interacting with a second surface (a Klein bottle), undergrad1 observed that there was always an intersection curve no matter which way he rotated the surface in 4-space. He fixed a particular projection and painted along an intersection curve with blue paint.

Undergrad1: Well, it looks like it's moving away. Let's see, right there. Well, I don't know. Maybe we can stretch this guy in that way. What do we get? Okay. So it looks like none of the paint is along the intersection line. So it looks one of these - at least one of these intersections is an artifact of projection [to 3-space].

Observer: Okay, what did you learn?

Undergrad1: It looks like that blue is a product of the projection, too. So there appear to be no inherent self-intersections in this!

Even though they could demonstrate that an intersection in the projection to 3-space did not correspond to any such intersection in 4-space, the subjects still seemed surprised that this could be the case.

5.2 Orientability

The undergraduate students generally could not tell by inspection whether a self-intersecting surface was orientable or not. One-sided paint proved to be a useful tool for them to make the decision. For example, undergrad1 had determined that a surface (the Klein bottle) had no self-intersections in 4-space. He seemed to conclude that it must therefore be orientable. I asked him what this surface might be.

Undergrad1: This sphere?

Observer: Could be.

Undergrad1: In this orientable surface?

Observer: Let's decide. Let's see whether it's orientable.

Undergrad1: Okay, so now I have to paint the whole surface? Okay, so now let's paint everything blue. Especially getting rid of this brown here. Well, if it has no self-intersections, how - it can't be non-orientable, can it?

Notice that he began painting with the assumption that every polygon must be painted. Eventually he discovered that only the 3D-visible components required painting (section 4.2). He began by painting two sides of a polygon with two different colors, and

incrementally extended the painted regions. Finally he found a component where the two different colors were present.

Undergrad1: Oh, wow. Well, I don't need to paint. It's obviously one-sided.

Observer: How do you know that?

Undergrad1: Well, these intersection curves that I painted right up to when it was turned the other way around, okay, if the paint crosses the intersection curve after I've turned it inside out, then it must just keep on flowing.

What he saw was that the intersection curve acts as a boundary for the painted region, so the paint within it must all be a single color.

Undergrad1: Since I painted the whole surface that I could see, right, and then when I rotate it around I can see more surface but I still see some of the surface that I used to be able to see. Without intersections.

Observer: What do you mean? Intersections are there.

Undergrad1: No, but they're not along the boundaries of the paint. The intersections should be the boundaries of the paint if there were an inside and an outside.

Observer: So what are you telling me?

Undergrad1: So I'm telling you that this looks non-orientable! So how can it be, if self-intersections are not inherent to the surface, that it's non-orientable? Oh, but it's in 4-space. So, in 4-space they don't have an inherent self-intersection.

He has discovered not only that this surface is non-orientable but imbedded in 4-space. He was surprised to learn this because he knew that compact non-orientable surfaces in 3-space must self-intersect.

In a session with a graduate student in computer science, grad1 also discovered that intersection curves are boundaries of a region being incrementally painted. He described what would happen if a 4-space rotation brought one patch of surface through another patch.

Grad1: Okay, so just call them side 1 and side 2 to begin with. Side 1 is the outside and side 2 is the inside. If I punch this part through until it's exposed, I see a little bit of side 2 here, but mostly I see side 1. I'm not entitled to paint this the same color as that just because I see it.

This subject was examining a projective plane that possessed a self-intersection in 4-space. [Plate 4.1 E] After completely painting the surface, he still could not conclude that it was non-orientable: the color on one "side" and the color on the other "side" only met at the intersection. Hence it was not apparent that adjacent polygons had conflicting colors.

Grad1: Boy, it still looks two-sided to me. I don't know David. Sure it's one-sided?

I suggested that he determine whether the intersection existed in 4-space. He found that it did, and then saw why that created an impediment to determining the surface's orientability.

Grad1: In effect the intersection curves don't move if they're really intersections in 4-space. But you can't paint across that because that's a real intersection!

In a session with two undergrads the subjects took turns applying one-sided paint. This time, they began with a single patch of one color and then incrementally extended the paint. They chose to paint every polygon (on each "side") until the surface was covered with paint. They began to suspect that the surface had only one side because it was taking so long to finish painting. When they were finished, transparency revealed that they had not missed any polygons.

Undergrad2: Maybe this is a Möbius band.

Undergrad3: Well, it's non-orientable, that's for sure.

Observer: And why is that obvious?

Undergrad3: Because it's only one color.

Even though they did not recognize the surface, they correctly determined that it was non-orientable.

5.3 Genus

In general the subjects were successful at determining whether a surface was imbedded in 4-space and, if it was, whether it was orientable or not. According to the classification theorem for compact surfaces, all that remained was to determine the genus of the surface. That proved to be difficult for the undergraduates. Unless the surface "looked like" an image they had seen before, the students could only guess what its identity was.

5.3.1 Recognizing the Shape of a Surface

During an examination of a projective plane, two undergraduates found that the surface was non-orientable. But they found it difficult to decide which non-orientable surface it might be. Eventually they brought it into an orientation where it resembled a picture of a cross-cap they had seen before.

Undergrad4: It's a Möbius band. Is it? It sort of looks like one. But, I don't know. [Rotates the surface in 4-space.] It doesn't look like it anymore. It comes around like that, and then it twists like right here. Maybe it's a Klein bottle. [Rotates it some more.] That looks like a projective plane.

Undergrad5: That little crease that was at the top up there.

Undergrad4: Yeah, so that's where it's a bunch of Möbius bands.

The two students who painted all of a Klein bottle with a single color were not sure what the surface actually was. They tentatively guessed that it might be a Klein bottle, but then ruled it out based on previous representations of a Klein bottle that they had seen before.

Undergrad2: It's not a sphere.

Observer: How do you know?

Undergrad2: It's got that gaping hole in it.

Observer: Where?

Undergrad3: Like right in here. But it might just go back in there. I don't know.

Undergrad2: No.

Undergrad3: No, it goes all the way.

Undergrad2: It looks like it self-intersects twice. It's probably a Klein bottle. No. Of course our in-class demonstrations of what a Klein bottle looks like looks nothing like this. Ours just looks like a bottle. I didn't expect that. Or a big square with arrows drawn this way.

They observed that the surface on the screen looked like “a sphere with a cross-cap on the top and on the bottom.” I prompted them to recall some of the results that they had learned in class about compact surfaces. In particular, I asked them how to form connected sums of surfaces.

Undergrad3: You take out a disk in each surface and glue 'em.

Observer: Great. So like if I had two projective planes, and I took their connected sum...

Undergrad2: Still non-orientable.

Observer: ... what would it look like?

Undergrad2: Maybe like that! (laughter) As we're steadily going along ... [thinking it may be a projective plane] So it is the Klein bottle! That's two projective planes.

The first undergraduate had also recognized the surface as a Klein bottle in the very same manner.

Undergrad1: Looks like two projective planes to me.

Observer: Which means what?

Undergrad1: It's a Klein bottle.

Observer: How did you decide it was two projective planes?

Undergrad1: Because it's got two of these little cross caps on it. And I couldn't make them go away.

5.3.2 Applying Morse Theory to a Surface

Dr. Sue Goodman, a member of the mathematics faculty at UNC, engaged in two sessions using Fourphront. Since she was familiar with Morse theory, I invited her to use the clipping plane to locate critical points on a surface. The surface was a spun open-knot, which forms a knotted sphere in 4-space. This surface presents many self-intersections when projected to 3-space and so is not obviously a sphere.

Goodman: Well it certainly looks like a surface of high genus.

Observer: Let's find its Euler characteristic. I'll keep a running total of the critical points of each degree as you find them.

She controlled the depth of the clipping plane by operating the slider in the X-window interface. Whenever she located a critical point and established its degree, I updated the corresponding total. She spent several minutes maneuvering the surface into an orientation that isolated the critical points. She found there were 2 critical point of degree 0, 2 of degree 1, and 2 of degree 2.

Observer: So 2 plus 2 minus 2 gives ...

Goodman: The Euler characteristic must be 2. So the genus is zero. It didn't have high genus after all. It's just a sphere!

I asked her if she recognized what kind of sphere it was in 4-space. She said that if it were knotted she probably would not be able to tell. I had constructed the sphere so that it was intrinsically transparent everywhere except for a single opaque ribbon that made a trefoil knot (in certain projections to 3-space). I moved the slider bar to display the intrinsic opacity rather than displaying the surface as completely opaque. This made the knotted ribbon more prominent.

Observer: Can you tell what this parametric curve does?

Goodman: Let me trace what happens with my finger. This part of the ribbon goes under, over, under... Could we paint these parts with different colors to help see which parts of the surface are in front when it becomes transparent?

We painted different parts of the ribbon with different colors to disambiguate "over" and "under" crossings in the transparent rendering.

Goodman: Yes, it seems to be a knot. I was concerned at first that the continuation of the ribbon might reverse the sense of the knot and unknot it. But that doesn't seem to be the case.

Next I showed her a Klein bottle and asked her to identify it, which she did immediately.

Observer: How can you prove that this is a Klein bottle? How can you demonstrate, for example, that it's not orientable?

Goodman: Well, I bet we could find a Möbius band in it.

Observer: Okay, let's paint one on.

Painting a Möbius band (consisting of roughly 100 polygons) took about 5 minutes to complete. When we had finished painting, Goodman used transparency to make the entire band visible within the surface and verified visually that this was indeed a Möbius band.

Observer: So let's find the genus of the surface.

Goodman: Okay. You want to keep track of the critical points again?

This time there was 1 critical point of degree 0, 2 of degree 1, and 1 of degree 2.

Goodman: So the Euler characteristic is 0. Good. It really is a Klein bottle then.

After she had completed the session, I asked her whether the system could be useful to her for research. She suggested first that this would be an excellent way to give exams to

students taking a first or second semester topology course. She also suggested that it would be illuminating to see loops contract on the surface as a way of visually determining the fundamental group of a surface. In her research she has found that the flow on a 3-manifold can be characterized by branched surfaces, and that there are interactive algorithms for determining the fundamental group of such a branched surface that might be carried out with Foughront.

5.3.3 Finding Critical Points at Pinch Points

Dr. Robert Bryant, from the mathematics department at Duke University, spent about an hour using Fourphront. He rotated a torus in 4-space so that it exhibited two intersection curves on the screen. Then he clipped into the surface to find the critical points of the height function on the surface. I asked him if the pinch points could be critical points.

Observer: What happens when the critical point occurs at a pinch point?

Bryant: Yes, that can happen.

Observer: Does that spoil the alternating sum?

Bryant: Not necessarily. Although most texts don't talk about the case of degenerate critical points. Look, we can clip into one right here. This one has index 0.

Observer: Oh, right. So if the pinch point were on the other side, it could have index 2.

Bryant: Yes, that's right.

Observer: Can you make a pinch point be a critical point of any index? Can we make an index-1 degenerate critical point?

Bryant: I'll have to think about that.

Observer: Let's try to do it. If I rotate the surface to bring a saddle over to a pinch point... Well, this may be hard to do. I think I can get it close...

Bryant: Think of a Whitney umbrella. The cross-sections look like parabolas that squeeze down to a line and then open up the other way. If you make the bottom of these parabolas lie on another parabolic curve, the saddle occurs at the pinch point. Here's a sketch of what I mean (figure 5-1).

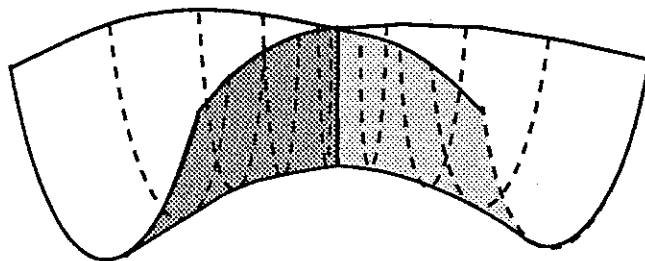


Figure 5-1. A Whitney umbrella. The vertical height function on the surface is critical at the pinch point.

Evidently our session together with Foughront gave Bryant the impetus to think about a particular combination of conditions that can exist on a surface in 4-space.

5.4 Links and Knots

Unlike spheres that are in 3-space, spheres in 4-space can link with each other and can knot. A torus can also “knot” in an unusual way in 4-space (although technically only a k -sphere can properly be called a knot). I asked a doctoral student to explore links and knots with Fourphront. He was completing his dissertation that semester on the subject of links and knots in 4-space.

5.4.1 Knotted Torus

In a session with a doctoral student (Colin Day) in mathematics, I displayed a torus which was modelled as a trefoil’s surface of revolution in 4-space. I asked the student if he could tell what the surface was. “I’m not sure. It looks like a torus though. Is it knotted?” When I highlighted the intersection curves, he said “Oh, it must be knotted, because of all those self-intersections. Is it?”

I had constructed the geometric model so that it was intrinsically transparent except for a parametric curve that was intrinsically opaque. I moved the slider bar so that the surface would exhibit its intrinsic opacity instead of being everywhere opaque. Now by rotating the surface on the screen, he could see that the parametric ribbon looked like a trefoil. “Well this is clearly a surface of revolution generated by a trefoil knot,” he observed. He might have found a knotted ribbon on the surface by applying paint to distinguish it, but the unmarked opaque surface was difficult for him to identify with certainty.

Day’s research involves finding ways that a circle can be deformed from a knot to an unknot and back again. This deformation can be modelled by sweeping a 3-space curve (which changes its knottedness) through 4-space to produce a surface. [Plates 5.4 A-B] The “knotted” torus illustrates how the constant map (preserving the knot at all times) might look geometrically. After examining variations of the surface during the two sessions, Day remarked that interactive visualization had helped him think about the problem. “I hadn’t really thought about the geometric aspect of these mappings. This is a very nice way to see the homotopy all at once. It’s changed the way I see the problem.”

5.4.2 Linked Spheres

Day also interactively examined a pair of spheres in 4-space to decide whether they were linked. I modelled two sets of spheres. The first set had a unit sphere centered at the origin and imbedded within the xyz -subspace of 4-space, and a second unit sphere centered at the point $(0, 0, 1, 0)$ and imbedded within the yzw -subspace. In the second set, the second sphere was centered at the point $(0, \frac{1}{2}, 0, 0)$ so that it intersected the first sphere at the

points $(0, \frac{1}{4}, \frac{\sqrt{15}}{4}, 0)$ and $(0, \frac{1}{4}, -\frac{\sqrt{15}}{4}, 0)$. Day did not notice any topological difference between the two sets of spheres until he painted the neighborhood of the intersections in the second set. The paint always remained at the intersection as the spheres were rotated in 4-space, which revealed that the second set was not linked. In 3-space it is not so difficult to decide if two circles are linked. This exercise indicated that it is much harder to perceive linking in 4-space.

5.4.3 Knotted Sphere

I showed Day the knotted sphere and asked him to identify it. In its initial orientation the surface looked like it might have high genus (as Dr. Goodman had observed): it looked like two lobes with portions of each lobe penetrating into the other. [Plates 5.4 C-D]

Day: If these parts terminate inside the lobes it might just be a sphere.

Observer: Let's turn on transparency and find out.

Day: They do. I think it's a sphere.

After rotating the surface for awhile, Day asked about the ribbons that were intrinsically opaque. I explained that they followed parametric curves in the model. [Plates 5.4 E-F]

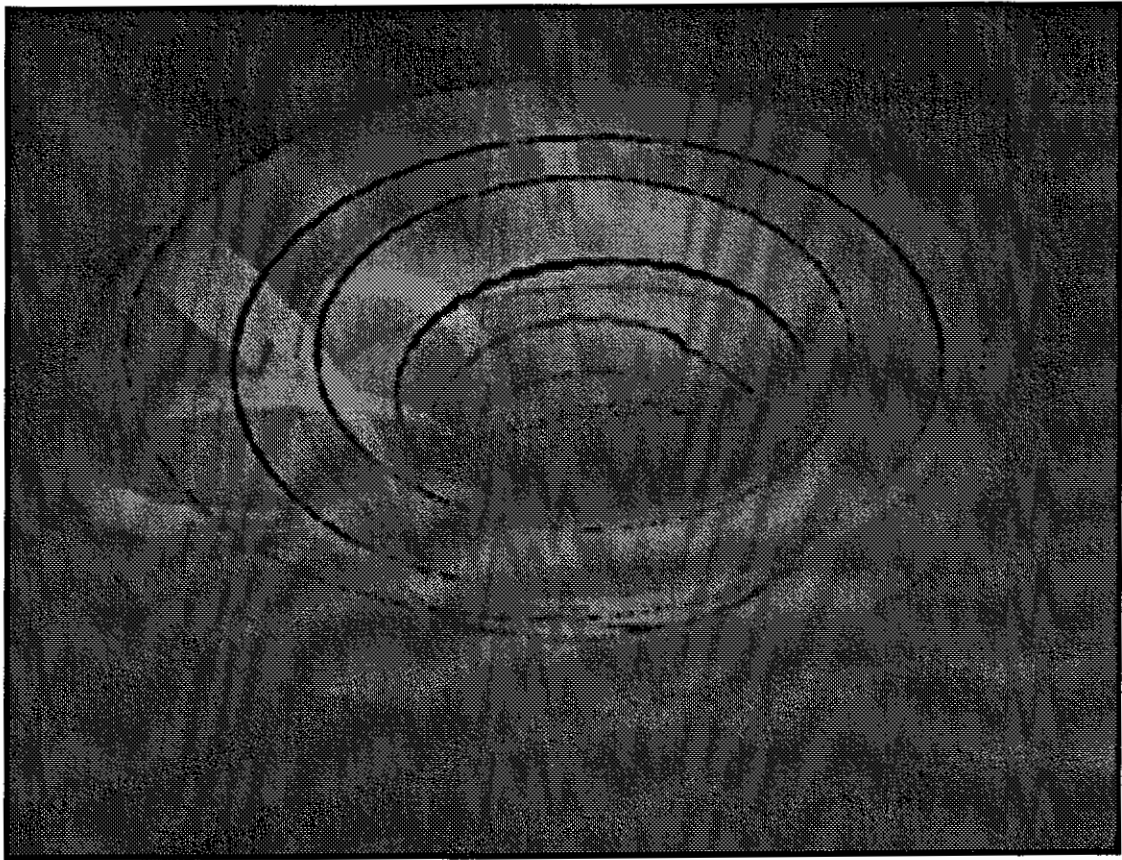
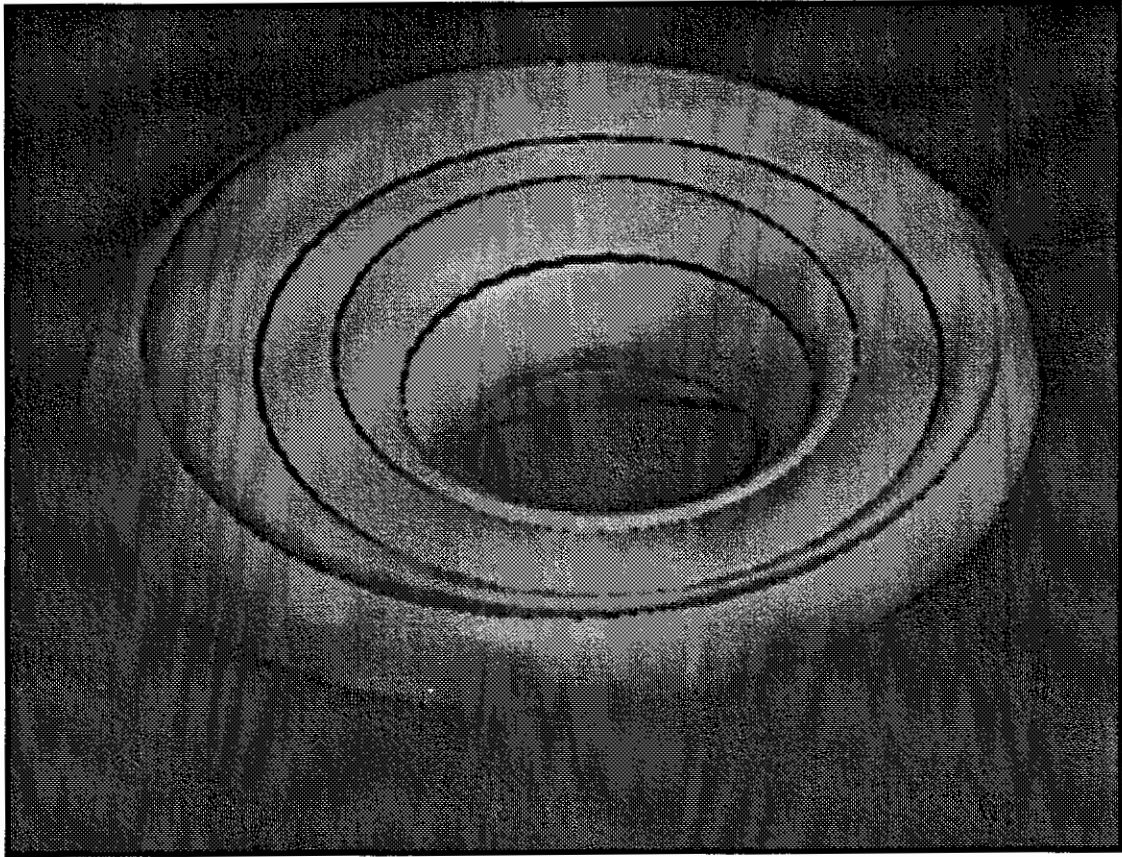
Day: This looks like it's trying to be a trefoil. It's a surface of revolution, isn't it? This is the open knot we talked about before.

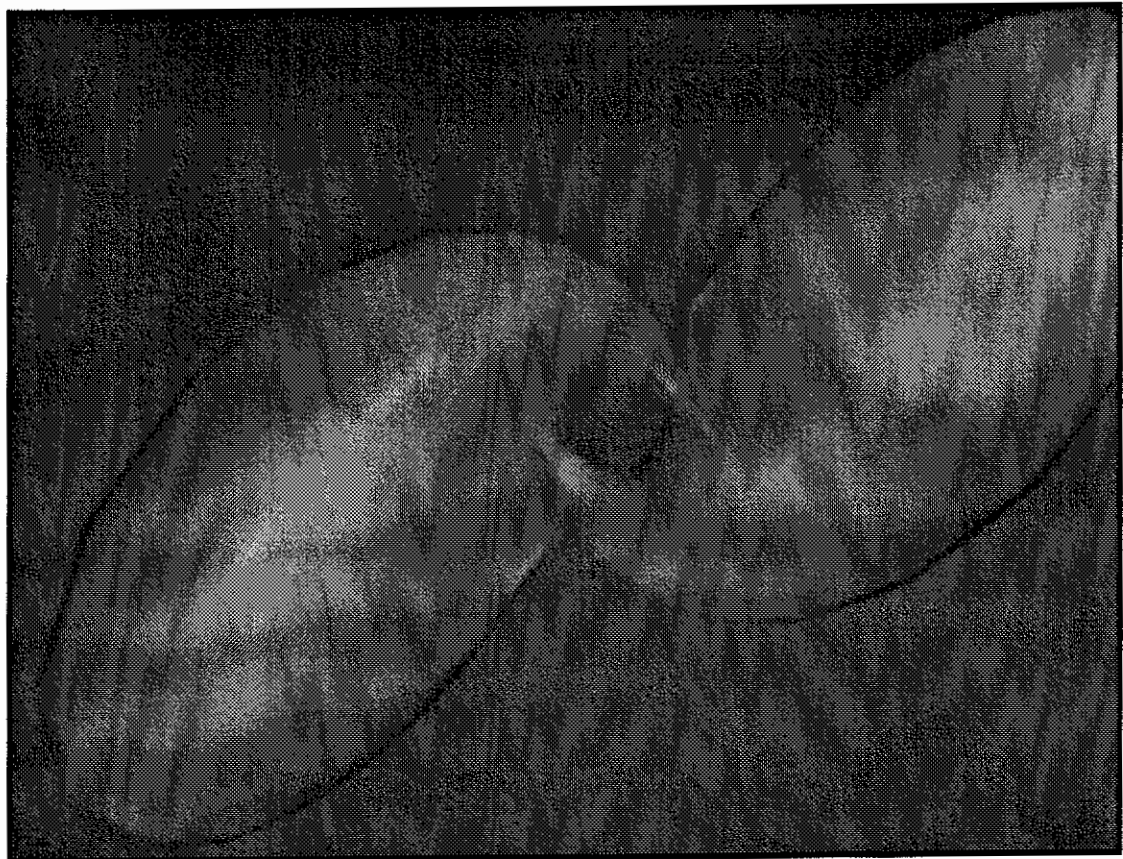
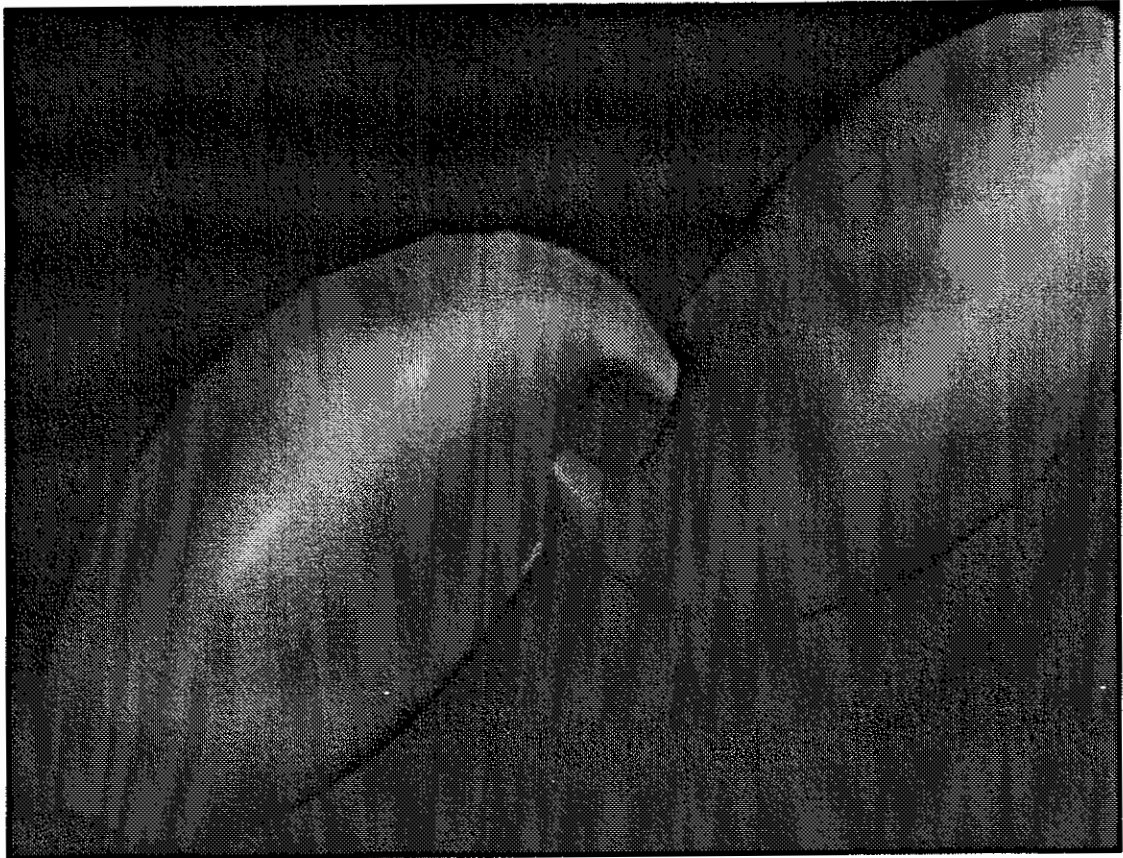
Observer: So what is the surface?

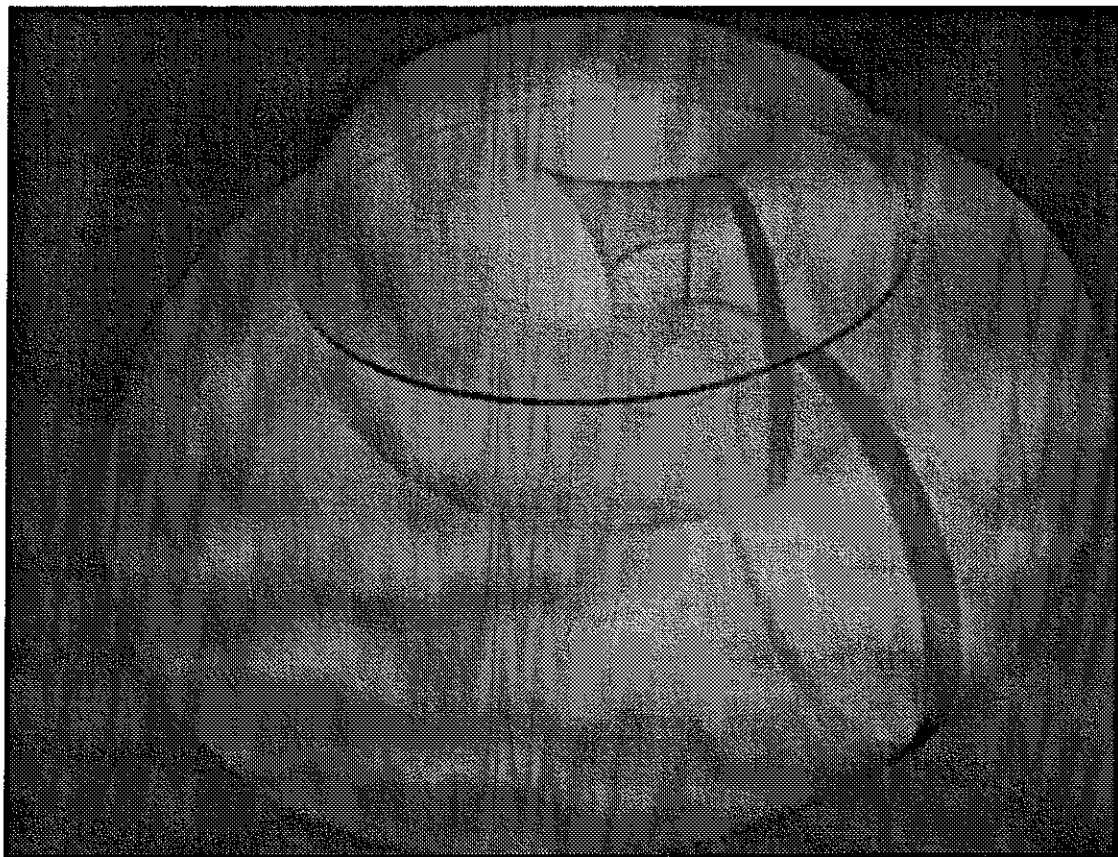
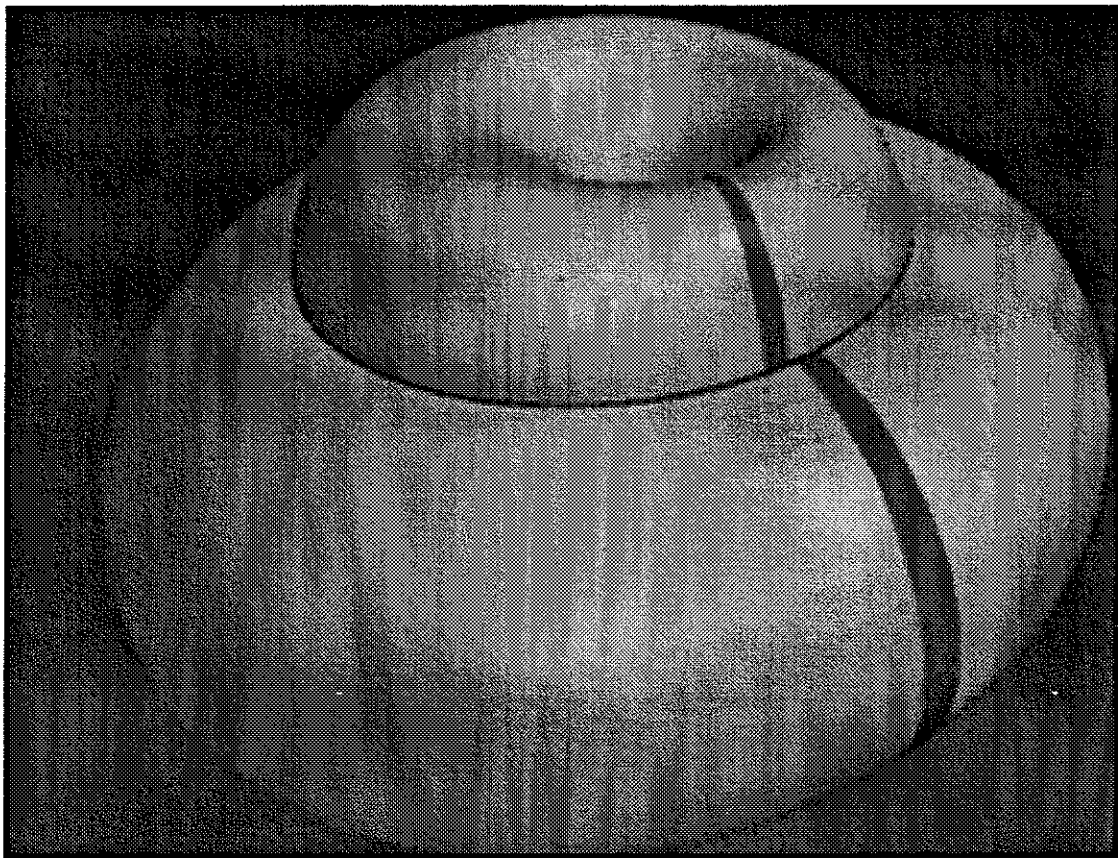
Day: A knotted sphere. Very nice. It has a lot of intersections.

Plates 5.4

- A** Trefoil knot in 3-space spun to make a “knotted” torus in 4-space.
- B** The surface is now transparent except for two parametric ribbons, made opaque to reveal the knot that sweeps out the surface.
- C** An open knot in 3-space is spun to produce a knotted sphere in 4-space.
- D** The north and south poles of the sphere are hidden inside the left and right lobes. Transparency reveals the shape of the regions near the poles.
- E** The knotted sphere can be rotated in 4-space to make it look more like a surface of revolution.
- F** The surface is now transparent except for two parametric ribbons, made opaque to reveal the curve that sweeps out the surface.







5.5 Constant Curvature in 4-space

In the plane, a curve can be defined with constant curvature κ . That curve must necessarily be a (portion of a) circle. Likewise in 3-space, a curve can be defined that has constant curvature κ and constant torsion τ . That curve must be a (portion of a) helix. One can similarly define a constant-curvature curve in 4-space as follows. Let

$$\gamma(t): [0,1] \rightarrow \mathbb{R}^4$$

be a curve fitted with (differentiable) local frame U_1, U_2, U_3, U_4 , where U_1 is tangent to the curve. One can require that

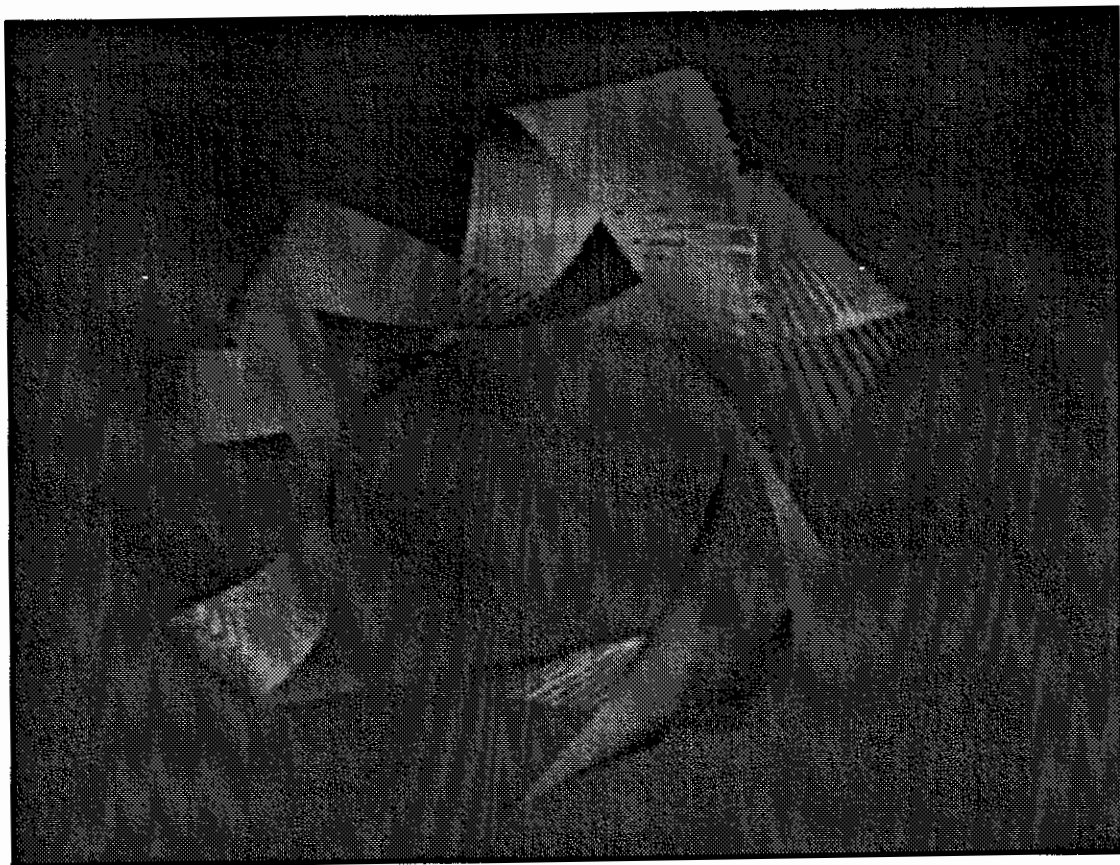
$$\begin{bmatrix} U_1' \\ U_2' \\ U_3' \\ U_4' \end{bmatrix} = \begin{bmatrix} 0 & k_1 & 0 & 0 \\ -k_1 & 0 & k_2 & 0 \\ 0 & -k_2 & 0 & k_3 \\ 0 & 0 & -k_3 & 0 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

which yields a constant-curvature curve in 4-space. What does such a curve look like?

I modeled this curve using a polygonal ribbon [Plate 5.5 A] and showed it Dr. David Eberly, whose dissertation area was differential equations. He believed the curve looked like it lay on a torus in 4-space, and then proved that this was so by solving the differential equation for the curve. He commented that "I never would have guessed that this is what the curve would do. But moving it around on the screen really made it clear that it had to lie on a torus."

Plate 5.5

- A** A constant-curvature curve in 4-space. The curve is thickened into strips to make its shape more visible.



5.6 Quaternion Rotations

Dr. Robert Bryant (of the mathematics department at Duke University) asked to see quaternion rotations in Fourphront (section 2.1.2). Recall that ordinary Euler rotations can be concatenated on the left or on the right (post-multiplication or pre-multiplication) before transforming points in a surface. The effect is either to rotate the surface in eye coordinates or in model coordinates. In a similar way, the left and right quaternions can be incrementally pre-multiplied or post-multiplied before transforming points on a surface. Bryant tried manipulating the surface (a torus) with both mappings, first using pre-multiplied rotations. He spent about 20 minutes examining the torus. During this time he located a family of rotations (in the k -direction) that rotated points on the torus but left the entire set of point fixed. These rotations made the torus look like it was a motionless object with a rotating "skin" on it. He also found a family of rotations (left rotation in the j -direction and right rotation in the $-j$ -direction) that produced Euler rotations when the surface was in a particular orientation.

Bryant: Look. See it wobble? That's because I don't have it in exactly the right orientation for this rotation. If it was just right you'd see it rotate on the screen without any wobble.

Observer: How did you find this combination of orientation and rotations?

Bryant: Oh, I already knew it was out there. It just took me a little while to find it.

Bryant then spent about 10 minutes using post-multiplied quaternion rotation. He found this scheme to be less effective than pre-multiplication, suggesting that he probably would have preferred whichever rotation-mode he used first. I think pre-multiplication also happened to be particularly well-suited to the torus, since there was a rotation direction that fixed the shape of the torus regardless of the surface's orientation on the screen. With post-multiplication, the surface's orientation on the screen mattered.

Bryant: I think that in order to really find out whether these rotations can become intuitive, you would have to spend many many hours with this system. It would be nice to be able to use it during idle moments during the day. It takes a long time for you to build a sense of how rotation works in the world, and you do it by holding things and turning them from the time you are a child.

5.7 Intuition

During their sessions with Fourphront, several subject made comments that revealed how much intuition they had, or were developing, concerning the shape of a surface in 4-space. Often these comments took the form of guesses about what the 3-space projection of the surface would do next in the course of a 4-space rotation.

Several subjects seemed to be frustrated that they did not have any empirical experience with 4-space to compare with the images they were seeing on the screen.

Undergrad6: So, I mean, what is this? What is this really like? Is this saying, like, if you were in, if we were in four dimensions, that's what we could make - I mean if we were holding that torus in our hand and were twisting it around, that's what it would look like?

This particular comment shows that the student had trouble believing that the image on the screen corresponded to the image he would actually see in four dimensions. This disbelief indicates that even though he could answer questions correctly about the surface's shape and topology in 4-space, he had not developed a mental model of the surface in the way that he could for a surface in 3-space.

The subjects quickly learned how to anticipate the effect that 4-space rotations would produce on the shape of the surface's projection to 3-space.

Undergrad3: It seems like it's going to intersect.

D: How could you tell?

Undergrad3: It's getting bigger.

D: But you were right. Just before it self-intersected you said, "It looks like it's going to intersect." And I wondered how you knew that. You've done this before?

Undergrad3: No. Ha.

Part of that anticipation may be due to the regular, periodic way the projected surface changes shape when a constant rotation in 4-space is applied. Part of the anticipation may be due to mental extrapolation of the changing shape of the surface in 3-space.

Even though they could maneuver the surface into a desired orientation, some of the subjects were still puzzled by the effect of rotation in 4-space. One student tried to justify the behavior he saw on the screen by thinking about linear algebra.

Undergrad4: I'm trying to think of, like, some linear algebraic way that, I mean, what, you, know, how this fits in the 4-space or whatever, you know? See, I don't understand. Basically, adding the fourth dimension let's you turn it inside out. Yeah, well is that really a direct correspondence, though? I mean, should it? I mean, does that fourth dimension have anything to do with being able to turn it, in reality, or is that just the way it looks?

He was struggling to see how an additional dimension permits a new type of rotation: turning a surface inside-out. Even though he could not explain why the surface should behave the way it did, the subject was adept at orienting it in 4-space to allow him to paint all of its polygons.

Only one student offered any remark to indicate that the 4-space rotations actually seemed to behave in a way consistent with his experience with 3-space rotations. This occurred when the student had made the surface semi-transparent before rotating it. Andrew Hansen

(of the University of Indiana) had told me of a similar experience he had when he watched an animation of a 3-manifold rotating in 4-space. He too had used transparency (in his case, to volume-render the 3-manifold).

Undergrad1: It looks like I'm just rotating it in 3-space.

Observer: Yes, it does. Transparency, I think, helps that effect.

Undergrad1: Helps it to look like it's not really rotating in 4-space?

Observer: That's right, because all the self-intersections, as they change, make it clear that the surface changes shape in 3-space. And with transparency you don't really see those self-intersections as much.

I believe that this effect, when it occurs, is due to the inability of the visual system to correctly reconstruct the shape in 3-space of the transparent object. The object's projection to 3-space really does change shape as the object rotates in 4-space, but the rotating projection can be conveniently misconstrued as a rigid object rotating in 3-space. This illusion of rigidity is undermined by opacity, which makes the object's projected shape unambiguous as it rotates.

6 Conclusions and Future Work

This dissertation presents techniques for interacting with surfaces in four-dimensional space for the purposes of determining their topology and gaining an intuitive sense of their shape in 4-space. I developed an interactive system called Fourphront (see Appendix B) that runs on the graphics supercomputer Pixel-Planes 5 (see Appendix A). This system incorporates new techniques devised for the user interface, illumination, depth cues, and visualization. It allows the user to apply well-known results from topology in an interactive way to a surface projected onto a computer screen.

6.1 User Interface

In designing an interface to map user-control from 3-space into the motion of an object in 4-space, the chief concern is to preserve kinesthetic sympathy. That means the object on the screen should move in a way that mimics the motion of the physical input device; thus the user can experience a natural sense of manipulating the object via the input device. The xy -plane of the screen can map naturally to both the 4D object-space and the 3D space of the input device. The remaining z - and w -axes in 4-space are each perpendicular to the xy -plane in object-space. On the screen, they are each “perpendicular” to the xy -plane in the sense that they are projected down to zero-length vectors. A 3D input device offers a single direction perpendicular to its xy -plane. That direction can be reasonably mapped to either the z - or the w -direction in the 4D object-space. A pair of 3D input devices can therefore control the object’s motion while preserving kinesthetic sympathy in the xy -plane: one maps the input-space to the (x, y, z) -space of the object, while the other maps to the (x, y, w) -space of the object.

There is a drawback. Since this scheme does not map any plane of input-space onto the zw -plane in object-space, there is not a sympathetic way to rotate the zw -plane from 3-space. But note that both mappings (to (x, y, z) and to (x, y, w)) provide an Euler rotation in the xy -plane. One can exchange the redundant rotation for the missing one: in the (x, y, w) -mapping of the input device, xy -rotation can be re-assigned so that it produces zw -rotation in the object. In that way, each of the six Euler rotations in 4-space corresponds to a single Euler rotation in one of the two input devices. This is the mapping that Fourphront uses.

In actual sessions with Fourphront, users were indeed able to select the appropriate rotations to apply to surfaces that they viewed on the screen. I did not perform timed experiments with the users, but I did observe how they manipulated the surfaces in 4-space. Not surprisingly, they found the (x, y, z) -mapping most intuitive. Users gradually became accustomed to the (x, y, w) -mapping as well, and I could not detect that they applied xw - or yw -rotations with greater facility than zw -rotations. For the purpose of examining a surface, painting it, finding intersections, or finding critical points, this mapping of the input from 3D to 4D proved to be adequate.

6.2 Illumination

There are two basic strategies for illuminating a surface that is projected from 4-space onto a screen. The first is to illuminate it in 4-space and then project the result. The second is to postpone illumination until the surface is projected to 3-space, where it is illuminated and then projected onto the screen.

The problem with the usual models of illumination used in 3D computer graphics is that they require the illuminated object to have co-dimension 1 (hence, a 1-dimensional normal vector). Surfaces have co-dimension 1 in 3-space, but co-dimension 2 (hence a 2-dimensional normal plane) in 4-space. This dissertation presents a general method for illuminating differentiable k -manifolds in n -space, where $n > k$. This general method is consistent with the usual one in the case where $k=2$ and $n=3$. In particular, I describe how to calculate diffuse and specular components of illumination for a surface in 4-space.

The heart of the technique is to consider the unit-length light vector \mathbf{L} and its components \mathbf{L}_T and \mathbf{L}_N that lie in the tangent space and the normal space at a point on the surface. Diffuse illumination measures how nearly \mathbf{L} coincides with the normal space, namely the magnitude of \mathbf{L}_N . Specular illumination measures how nearly the reflected light vector \mathbf{R} coincides with the eye vector \mathbf{E} , calculated as the magnitude of their dot-product. The reflection preserves \mathbf{L}_T but reverses \mathbf{L}_N . This interpretation of diffuse and specular illumination allows one to illuminate a surface in 4-space or even a curve in 3-space.

Fourphront can illuminate a surface in 4-space or its projection in 3-space. The system only implements the diffuse model of 4-space illumination; interpolating the tangent plane and the normal plane in 4-space (for specular illumination) requires more memory than is available when using the native graphics library on Pixel-Planes 5 (see Appendix A). I did not test how well users interpret 4D illumination compared to 3D illumination, but my own experience with Fourphront suggests that 4D illumination enhances one's sense of surface

shape in 4-space, at least indirectly. As I view a surface illuminated in 4-space, I mentally proceed through a chain of implications: knowing that the light shines down *this* axis and the surface is bright in *this* region, I conclude that the surface must therefore possess a maximum in *that* direction and thus be slightly rounded in *this* vicinity. Knowing the shape of the surface helps me comprehend the way it is illuminated about as much as knowing the way it is illuminated helps me comprehend the surface's shape.

6.3 Depth Cues

Two axial directions are annihilated during the projection of a surface from 4-space to a 2D screen. There are many techniques in computer graphics that help one recover a sense of 3D depth in a single annihilated direction. How can one recover an additional dimension of depth? In general, one surface patch in 4-space may occlude or shadow another patch along only a 1-dimensional curve, which offers little help for recovering depth at points in the remainder of the patches. Texture and color, by varying exclusively in the *w*-direction, can enhance the perception of depth in that direction. Perspective (associated with the projection from 4-space to 3-space) can provide a relative sense of how near or far a patch is in the *w*-direction as a consequence of how large or small its projected image is. This dissertation presents a new technique for depth-cueing the *w*-direction: modulating opacity according to *w*-depth. By making points on a surface become more opaque as they recede in the *w*-direction, one can often clearly distinguish the near regions of a surface in 4-space from the far regions.

6.4 Visualization Tools

There are several graphical techniques that help to enhance the shape of a surface. These techniques apply in three dimensions, but are especially useful for interacting with a surface in 4-space.

6.4.1 Seeing Into a Surface

A surface often intersects itself after being projected from 4-space to 3-space. In order to see past the intersections and examine hidden patches of the surface, one can slice the geometric model of the surface into ribbons, one can clip into the surface, and one can apply transparency to the surface. Each technique is straightforward to implement, but each has drawbacks. The ribbons can produce distracting moiré patterns where they overlap, although that effect is diminished by making the gaps between them more opaque. Clipping a surface in an arbitrary direction in 4-space presents a challenge in controlling the 10 degrees of freedom for translation and orientation. Fourfront constrains clipping to occur in only the

z-direction after the surface is projected to 3-space in order to simplify the operation. Transparency is an especially valuable visualization tool, but rendering transparent surfaces (first sorting the surface primitives in order of depth) is slower than rendering opaque ones.

6.4.2 Applying Paint

Applying paint to some or all of a surface is another tool that helps one visualize the shape of a surface in 4-space. By painting the neighborhood of a self-intersection and then rotating the surface in 4-space, one can visually determine whether the intersection is merely a result of projection to 3-space or if the intersection actually lies on the surface in 4-space. Interactive painting (using one-sided paint) also allows one to decide whether a surface is non-orientable, either by painting a Möbius band within the surface, by painting two colors to the two “sides” of the surface until different colors collide, or by painting a single side until the entire surface is covered. I describe a method for applying transparency to a surface primitive that has different colors on its front and back sides.

In interactive sessions, users were able to effectively apply incremental painting techniques to determine whether a surface imbedded in 4-space was orientable. If a non-orientable surface was not imbedded, most users failed to paint the surface correctly. Some users saw where a Möbius band lay in a non-orientable surface; they then proceeded to paint it correctly. Users who groped blindly to find a Möbius band had less success.

6.4.3 Locating Intersections and Silhouettes in Screen-space

As a surface becomes more transparent, its intersection curves and its silhouette become less prominent. One can estimate, at each pixel, whether an intersection or a silhouette occurs nearby. To locate intersections between opaque surface primitives in 3-space, one compares the z-depth of the frontmost two primitives at a pixel. Wherever they are identical, the primitives intersect. (One must take care to scan-convert adjacent primitives so that they do not redundantly raster the pixels along their shared edge.) Instead of requiring an exact match of z-values one can require only a near-match, thereby identifying (and highlighting) larger neighborhoods of the intersections. The width of the highlighted intersection varies according to the gradients of the two primitives at a pixel. One can compensate for this variation to produce fixed-width intersection curves on the screen. This dissertation describes how to calculate intersection curves in parallel using the SIMD renderers on Pixel-Planes 5. The Fourfront system implements variable-width and fixed-width intersection curves.

One can also locate (and thus highlight) silhouettes by finding pixels where the surface normal is perpendicular to the eye-vector. These curves can be thickened by relaxing the requirement that the normal and eye-vector be exactly perpendicular. In order to make the thickened curve be fixed-width, a local quadratic approximation to the surface is required. This dissertation described how to calculate variable-width and fixed-width silhouette curves. The Fourphront system can highlight variable-width silhouettes, but calculating fixed-width silhouettes at each pixel requires more memory than the Pixel-Planes 5 renderers possess.

Since intersection curves are important in determining whether a surface is imbedded in 4-space (which in turn determines how simple it may be to paint the surface to decide whether it is orientable), highlighting them is very useful. Transparency makes intersections more difficult to detect visually. Highlighting intersections on transparent surfaces is therefore an effective visualization technique. Highlighting the silhouette curves is likewise beneficial in the presence of transparency. They reinforce the shape of the surface as the surface's color blends into the background color, and they reveal the shape of the interior portions of a surface that self-intersects in 3-space (as surfaces often do after projection from 4-space).

6.5 Results from Interactive Sessions

Most of the undergraduates who used Fourphront expressed concerns that they did not understand four dimensional space and might not do very well using the system. But I discovered that by applying paint to a surface in 4-space, these students were very successful in answering questions about the topology of the surface. They had the most success deciding whether a surface self-intersected in 4-space or not, probably because the painting algorithm was so simple to carry out and to interpret. They generally had success deciding whether a surface was orientable, again by applying paint to it. The one exception occurred when the surface self-intersected in 4-space and the subject had not already identified that self-intersection.

Subjects who were familiar with topology could often classify the surface, but seemed to rely heavily on mental comparisons with pictures that they had seen before. Subjects who were familiar with Morse theory could classify a surface with confidence. For them the only problems were how to orient the surface to isolate critical points and how to treat pinch points as the clipping plane crossed them.

Knottedness and linking appear to be properties of surfaces in 4-space that are difficult to resolve with interactive computer graphics. Colin Day suggested that one might try to

construct non-contractible loops around a knotted sphere in 4-space in order to show that its complement had a nontrivial fundamental group. He also suggested that watching surfaces deform in 4-space (animating them) could reveal their topological properties.

Using Fourphront led one mathematician to prove how a particular curve in 4-space must behave – a result he did not already know and that he might not have thought about without using the system. It led one graduate student to conjecture (correctly) about the set of points that can coincide with a given point after projection to 3-space. One undergraduate student discovered for himself that a compact surface can imbed in 4-space even though it is non-orientable. Many students discovered for the first time what it is like to manipulate an object in 4-space.

Dr. Goodman contended that a system like Fourphront provides an excellent environment for demonstrating topological properties of surfaces and also for examining the knowledge of students taking topology courses. Is such a system useful for research as well?

Goodman suggested that it would be valuable for exploring branched surfaces, because even elementary calculations are tedious when they involve the quotient spaces that give rise to these surfaces. Some of these calculations might be carried out by interacting with the branched surfaces. Dr. Bryant concluded that the system would be useful for research only after logging many hours with it.

Bryant: I'm probably not the best person to ask about this. You see, I've spent many years thinking about how surfaces behave in 4-space. There might be holes in my understanding of the way things act in 4-space. But I would have to first spend a lot of time registering what I see on the screen with what I think I already know. After that, maybe I could begin finding holes my understanding.

On the other hand, this kind of system is ideal for a second-year graduate student who is just beginning to think about these things.

So Fourphront seems to be very effective in helping the "junior mathematician" think about surface in 4-space and to develop conjectures about their topological and geometric properties. Whether such a system is valuable for advanced research has yet to be proved.

6.6 Future Work

There are several directions in which to extend Fourphront and the techniques that it uses. I plan to port the system onto other hardware platforms that are available outside the department of computer science at UNC. I would like to interact with 3-manifolds in 4-space by computing and displaying their silhouette surfaces and intersection surfaces. I plan to implement specular illumination for surfaces in 4-space and calculate fixed-width

silhouette curves in screen-space. I would also like to use a headmounted-display and 3D trackers for navigating in four dimensions.

Appendix A: Pixel-Planes 5

A.1 Hardware Architecture

Pixel-Planes 5 is a heterogeneous multicomputer. [Plate A.1] It is a multicomputer because it has many different (programmable) execution nodes connected in a message-passing network. It is heterogeneous because those nodes are not all identical. Recall that the canonical graphics pipeline for object-order surface-based rendering has two important stages. During each frame the primitives are (1) transformed into screen coordinates, then (2) drawn into a frame buffer.

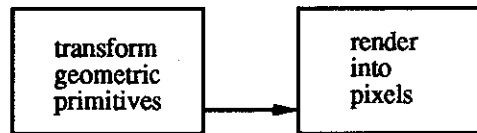


Figure A-1: Two stages of the conventional object-order graphics pipeline.

Pixel-Planes 5 parallelizes each stage by devoting multiple processors to the task. The Graphics Processors (GPs) are high-performance Intel i860's that transform the primitives and route the results to the correct renderers. The renderers are custom-designed processors that light and shade the primitives.

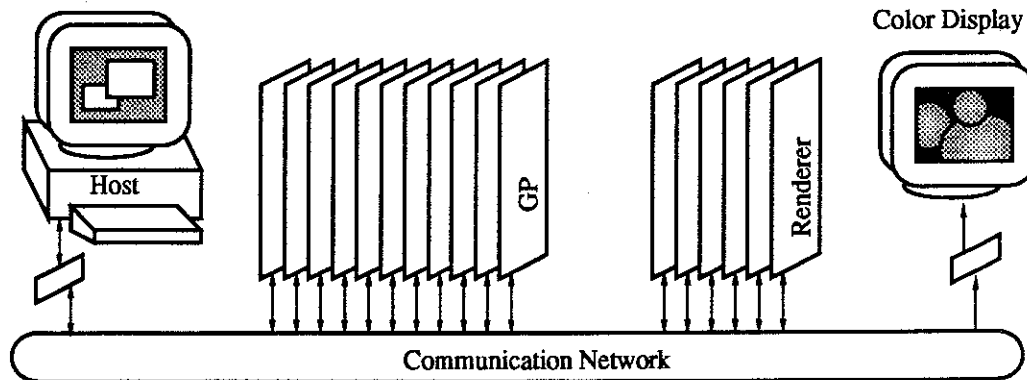


Figure A-2: Architecture of Pixel-Planes 5. The GPs and renderers embody the transformation and rendering stages of the graphics pipeline of figure A-1. The host workstation connects to the communication network through a special interface. The color display connects to the network via a frame buffer.

During each frame there is a logical renderer assigned to each region of the screen. The pool of physical renderers is dynamically assigned to act as logical renderers, so each pixel

has a logical processor dedicated to it. The GPs transform the geometric primitives and issue instructions for the renderers to carry out. These instructions cause the primitives to be scan-converted, illuminated, shaded, and z-buffered by the renderers.

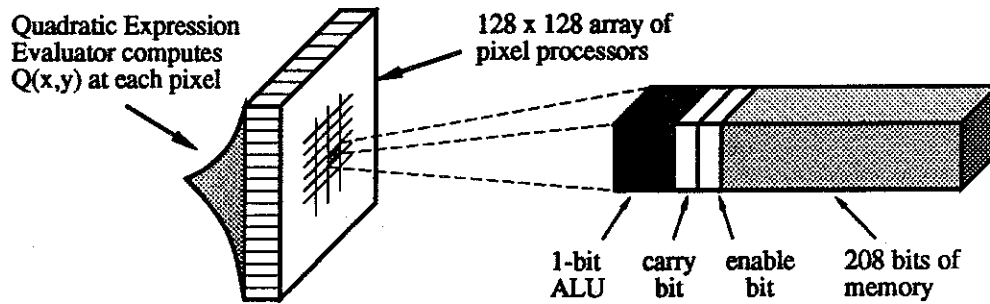


Figure A-3: Logical organization of the Pixel-Planes 5 renderer. Each renderer has an expression evaluator and a grid of pixel processors. A single pixel processor (right) has local memory and a 1-bit ALU. Two bits of the local memory are special: the enable bit and the carry bit.

A renderer itself has two parts: a Quadratic Expression Evaluator (QEE) and a Single-Instruction-Multiple-Data (SIMD) array of simple pixel processors. The QEE evaluates a quadratic function $Q(x, y)$ at each pixel and makes the result available to the pixel processor there. Each pixel processor can read the value $Q(x, y)$ provided to it by the QEE and can read and modify its pixel memory using its own 1-bit Arithmetic and Logic Unit (ALU) that operates in a bit-serial fashion. A pixel processor also has a carry bit (for performing 1-bit addition) and an enable bit that determines whether the processor will participate in a SIMD operation. When a renderer receives an instruction, only its enabled pixels will execute the instruction.

A.2 Renderer Instructions

A set of rendering instructions, written as macros, are available to the programmer. These instructions operate on the components of the renderer represented in figure A-3.

Programming the renderers using these macros is generally a tedious process. The programmer can modify the quadratic expression that the QEE evaluates per pixel, and he can perform operations on the various parts of the pixel memory. Some of these operations are guarded by the enable bit so that they are executed by a pixel if and only if the enable bit is set (equal to 1). Writing renderer code means sending the renderers a stream of instructions that operate at this low level of bit operations. In order to display an image, the 24 bits corresponding to red, green, and blue intensities are sent to a frame buffer via the communication network.

The ALU can perform several arithmetic and boolean operations such as addition, subtraction, bitwise copying, logical and, logical or, and negation. The source of these operations can be the QEE, the ALU, the carry bit, the enable bit, or a contiguous section of pixel memory (as in figure A-3). The destination of an operation can be the carry bit, the enable bit, or a contiguous section of pixel memory. Not every combination of source and destination is possible, however. Figure A-4 summarizes these functions and gives examples of each.

Src Dst	QEE	ALU	Carry	Enable	Memory
Carry	_____	if (enab) carry = 0	_____	carry = enab	_____
Enable	enab = (QEE < 0)	enab = 1	enab = carry	enab = ~enab	enab = $m_1 + m_2$
Memory	if (enab) mem = QEE	if (enab) mem = 0	_____	mem = enab	if (enab) $m_1 = m_2 + m_3$

Figure A-4: Possible operations within the pixel processor. The source of the operation is represented on the top row. The destination is designated on the left column.

Both the QEE and the memory can be used as a pair of operands, for example, comparing a memory value to the value in the QEE at a pixel. In addition to generating these pixel instructions, a program can load coefficients into the QEE, yielding a result that the pixel's ALU may subsequently use as a source operand.

A.3 An Abstract Programming Model of the Renderers

The programming model for the renderers can be expressed in C-language code (with a few extensions to make the discussion simpler). I will present a simple implementation of the rendering system and then show how a program might look using this model. This implementation looks very different from the set of renderer-macros that a programmer actually uses, but those macros (with names like IGC_MEMleTREE) are more complicated to explain and are very difficult to read. The abstract model I have devised is functionally equivalent to the set of renderer macros and is expressed in a high-level structured language.

I first define the global state of the renderers. That state includes the carry bits, the enable bits, and the QEE. In this model, each variable is a two-dimensional array. In reality, pixel variables contend for the 208 bits of pixel memory, so the programmer must carefully manage the location and length of each pixel variable. But in the abstract programming model I simply define a pixel variable as an array of bits. The length of the pixel variable will be supplied to an initialization routine which allocates actual bits.

```
bit          Carry [1280][1024][1];    /* Carry bit */
bit          Enable[1280][1024][1];    /* Enable bit */
float        QVal  [1280][1024];        /* Local evaluation of quadratic */
typedef bit *PixVarType[1280][1024];    /* Pixel variable of unknown length */
```

I have introduced a non-standard data type: a “bit” type. It is convenient to assume that arithmetic and logical operations can be applied to bit arrays so that numbers are operated on regardless of their length. (This is not a feature of the C language, although languages like C++ can “overload” operators to apply to new data types.)

Each pixel must be able to read its local evaluation of the quadratic expression, and the pixel variables must behave like plural instantiations of a single variable. A function `LoadQVal()` satisfies the requirements of the QEE, and the function `InitPixVar()` allocates bits for a pixel variable (bit allocation is another non-standard function). Rather than explicitly looping over all the pixels with every pixel-variable operation, I define a simple macro that suggests the SIMD nature of the pixel processors. This macro loops over the entire screen using index variables `X` and `Y` to represent pixel locations.

```
#define ForAllPixels for(X=0; X<1280; X++) for(Y=0; Y<1024; Y++)
```

A renderer actually executes an instruction in parallel at all of its processing nodes, but the nested loops describe the function correctly, if not the timing. The routines `LoadQVal()` and `InitPixVar()` are shown below.

```

LoadQVal(float a, b, c, d, e, f) {
    ForAllPixels {
        QVal[X][Y] = a*X*X + b*X*Y + c*Y + d*X + e*Y + f;
    }
}

InitPixVar( PixVarType v, int len) {
    ForAllPixels {
        v[X][Y] = bitAlloc(len);          /* Install the variable at each pixel */
    }
}

```

The ForAllPixels macro makes it simpler to write a simple program for the renderers. The example below shows how to create a white rectangle on the right side of the screen. The example uses the quadratic expression $0x^2 + 0xy + 0y^2 + 0.25x + 0y - 250$ to enable only those pixels that are within the rectangle. Then QVal is given the constant quadratic $Q(x,y) = 255$ so that the red, green, and blue pixel variables can all be saturated to the value of 255, creating white.

```

bit  Red, Green, Blue;          /* Color components of the pixels */
int  X,   Y;                   /* Satisfy the ForAllPixels macro */

InitPixVar(Red,   8);           /* Allocate 8 bits of red */
InitPixVar(Green, 8);           /* Allocate 8 bits of green */
InitPixVar(Blue,  8);           /* Allocate 8 bits of blue */

ForAllPixels {
    Enable[X][Y] = 1;           /* Enable all pixels */
}

LoadQVal(0, 0, 0, 0.25, 0, -250); /* Define a plane having positive */
                                     /* values in the screen's right half */

ForAllPixels {
    Enable[X][Y] &&= (QVal[X][Y]>=0); /* Enable the right half-screen */
}

LoadQVal(0, 0, 0, 0, 0, 255);     /* Define a plane with a constant */
                                     /* value of 255 at each pixel */

ForAllPixels {
    if (Enable[X][Y])
        Red  [X][Y] = QVal[X][Y];    /* Load 255 into the red bits */
}

ForAllPixels {
    if (Enable[X][Y])
        Green[X][Y] = QVal[X][Y];    /* Load 255 into the green bits */
}

ForAllPixels {
    if (P[X][Y].Enable)
        Blue [X][Y] = QVal[X][Y];    /* Load 255 into the blue bits */
}

```

A.4 PPHIGS

PPHIGS is the Pixel-Planes Hierarchical Interactive Graphics System. It is a collection of about 100 different routines designed to be the standard 3D graphics library for applications that use Pixel-Planes 5. PPHIGS is patterned after the PHIGS (Programmer's Hierarchical Interactive Graphics System) library (ANSI Standard X3.144-198x). PHIGS has become an industry standard, (available on many workstations as an extension to MIT's X Windowing System). PHIGS has many features that PPHIGS does not (like support for 2D graphics and text). PPHIGS has a few features that PHIGS does not (like sphere primitives, callback functions, pointers to display list elements, and antialiasing). But the two libraries overlap significantly in their conceptual design.

The primary purpose of PPHIGS[] is to manage display lists and convert them into a picture once per frame. A display list is a data structure in the form of a directed acyclic graph (like a tree). The nodes of the tree can be geometric primitives, transformation matrices, callback functions, and pointers to other trees. PPHIGS allows an application to create a tree and dynamically edit its contents. In order to draw a picture, PPHIGS traverses the tree, applies the (concatenated) transformations to the primitives and then renders them into the frame buffer. Rendering requires a certain amount of global state, so PPHIGS supplies functions that establish this state. The global state includes lights, viewing parameters, fog, a hither plane, background color, and an antialiasing kernel. Finally, PPHIGS provides miscellaneous control functions that open and close PPHIGS, read and write archive files, display a frame, and traverse a display list.

A special feature of PPHIGS is that an element of the display list can be a callback function. A callback function is a user-written routine that is executed on the GP's. I placed a single callback at the top of my display list of triangle data for a surface in 4-space. The callback function was then executed before the rest of the display list was traversed. That callback itself traversed all the 4D data, transformed it, and projected it to 3-space. After the callback finished executing, PPHIGS resumed control to render each of the triangles it found in the display list.

Appendix B: Fourphront

B.1 System Capabilities

Fourphront lets a user interact with a surface in 4-space by providing a small set of basic tools. Those tools can be organized into the following categories.

Rotation in 4-space	A surface can be rotated with any combination of 6 Euler rotations in 4-space. Alternatively, it can be rotated by left- and right-multiplying each vertex by quaternions.
Translation	A surface can be translated in 4-space before being projected, or in 3-space after being projected.
Scaling	A surface can be uniformly scaled larger or smaller.
Clipping	A surface can be clipped against the hither plane after projection to 3-space.
Spraypainting	The polygons comprising a surface can be painted any color from a palette of 4 colors. To paint a polygon, the user positions an animated icon of a can of spraypaint over the polygon to be painted. Polygons can also be unpainted, restoring them to their original color.
Transparency	Each vertex v has an intrinsic opacity $I(v)$ and a displayed opacity $D(v)$. The displayed opacities are interpolated across a polygon when it is rendered. The user can globally change the values $D(v)$ so that they interpolate between total opacity and $I(v)$. That means that transparent parametric ribbons can be gradually faded in and out.
Intersection highlights	Intersection highlights can be toggled on and off, and the user can choose between variable-width and fixed-width calculations. The width of the curve can be controlled dynamically.
Silhouette highlights	Silhouette highlights can be toggled on and off. The width of the silhouette curve can be controlled dynamically.
Depth-cueing	The user can control the ramp that modulates opacity according to w-depth.
Color selection	The background color can be dynamically changed, and the four colors in the palette of spraypaint can be individually changed.

B.2 User Interface

Joystick Box

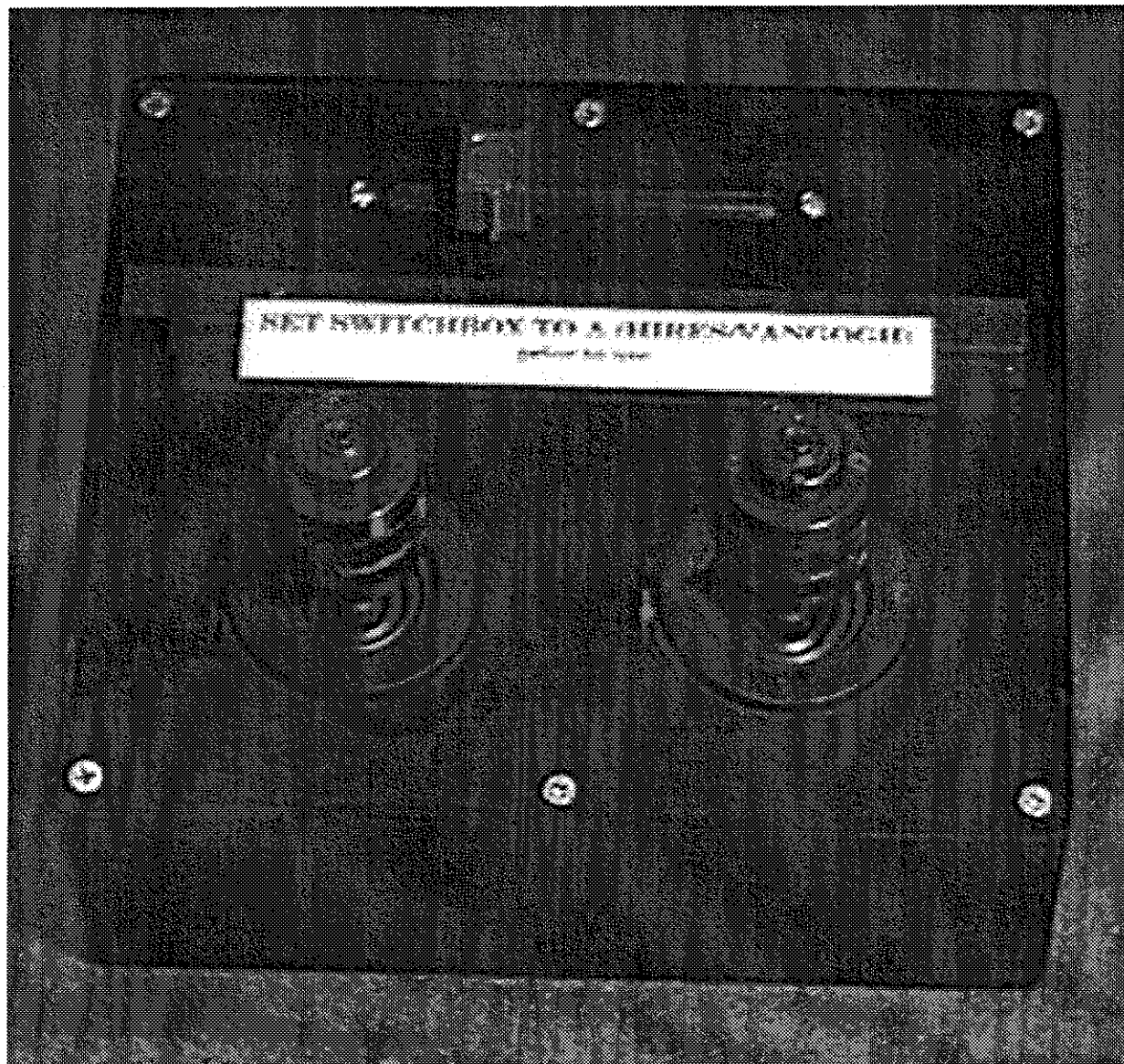
Fourphront uses an interface designed to work with a joystick box (pictured in Plate B.2 A). The joysticks operate in several different modes: rotation, translation, scale, quaternion, and spraypaint. In rotation mode, the left joystick controls rotations that involve the w -axis and the right joystick controls rotations that involve the xyz -subspace of 4-space (chapter 2.1.2). In translation mode, the left joystick controls z - and w -translations while the right joystick controls x - and y -translation. In scale mode, either joystick can uniformly scale the surface larger or smaller. In quaternion mode, the left joystick produces quaternions for left-multiplication, and the right joystick produces quaternions for right-multiplication.

The slider bar has two purposes. As it moves from left to right it produces an interpolation variable t in the interval $[0, 1]$ that governs transparency. Each vertex v in a surface has an intrinsic opacity $\alpha(v)$, also in the range $[0, 1]$. The displayed opacity is computed by the expression $t \alpha(v) + (1-t)$. Whenever the slider reaches the far right, it acts as a toggle switch to enable or disable painting. When painting is enabled, the right joystick controls the motion of the spraypaint icon and the left joystick controls the color of the paint. There are 4 colors available in the palette at any time. To select one, the user moves the left joystick forward, back, left, or right. To restore the original color to a polygon (unpaint), the user spins the joystick clockwise. To stop applying paint, the user spins the joystick counterclockwise.

X Window Interface

The user can also control the state of the Fourphront system by using buttons and sliders in an X Window interface. These controls are shown in Plates B.2 B,C. The set of button controls works as follows. The "Opacity Mode" button determines whether transparency is applied according to the intrinsic value at each vertex or whether it is used as a depth cue in the w -direction (chapter 2.3). The "Meter" button determines whether the performance meter is displayed on the screen. This meter graphically indicates the rate that primitives are transformed and rendered. The "Transform" button indicates which of the four modes the joysticks are operating in (see above). The "Resolution" button selects between low and high raster-resolution displays. The "Intersection" button determines whether Fourphront computes variable-width, fixed-width, or no intersection curves. The "Silhouettes" button determines whether Fourphront computes silhouettes curves.

The slider bars control state-variables with large or continuous ranges. Each slider has two special buttons on either end that display "<" and ">" icons. When the user presses one of these controls, a pop-up window appears that lets him establish the minimum or the maximum value of the slider's range. The "Hither Plane" slider controls the distance of the hither plane from the eye. The "Transparent" slider is a duplicate control for the physical slider on the joystick box. The "Scale" and "Offset" sliders control the shape of the ramp that governs depth-cued transparency in the w-direction (chapter 2.3). The "Intersection" and "Silhouette" sliders control the width of the respective curves. There is a "Color" button to select either the screen's background color or one of the 4 spraypaint colors. The "Red," "Green," and "Blue" sliders then control the components of the selected color.



☒ Button Controls

popdown

Button Controls

Opacity Mode	intrinsic
Meter	Off
Transform	Rotate
Resolution	Lo-Res
Intersections	Off
Silhouettes	Off

Slider Controls

pop-down

Slider Controls

Render Plane

3200

<

>

Transparent

1.778

<

>

Scale

0.05

<

>

Offset

-80

<

>

Intersection

1000

<

>

Silhouettes

2000

<

>

Color

Background

<

>

Red

30

<

>

Green

40

<

>

Blue

45

<

>

B.3 System Performance

The graphics performance of Fourphront is considerably slower than Pixel-Planes 5 can sustain for 3D graphics. A fully loaded Pixel-Planes system can transform and render up to 2 million triangles per second. In order to reach such speeds, a graphics application must take advantage of special microcoded routines that fit into the instruction cache on the GP's. Fourphront's transformation and rendering routines are too large to cache, and since I needed to modify them from time to time I did not microcode them. System performance for actual datasets is shown in the table below (figure B-1).

Render attribute \ Frame buffer type	High Resolution (1280 x 1024)	Low Resolution (640 x 512)
Plain	99 000	110 000
Intersection Curves (variable width)	90 000	108 000
Intersection Curves (fixed width)	66 000	69 000
Silhouette Curves	91 000	104 000

Figure B-1. Performance figures for Fourphront (triangles per second). The left column describes the attributes that triangles were rendered with. The middle and right columns give the performance for high and low resolution images. These measurements were made using a 3-rack configuration with 40 GP's and 20 renderers. The dataset had 5000 triangles, and the time includes 4D transformation and rendering.

REFERENCES

- [Anick89] Anick, David J. "The Computation of Rational Homotopy Groups is NP-Hard," *Computers in Geometry and Topology* (Edited by Earl Taft *et al.*), Marcel Dekker, Inc. New York, 1989, 1-56.
- [Armstrong85] Armstrong, William and Robert Burton. "Perception Cues for n Dimensions," *Computer Graphics World*, March 1985, 11-28.
- [Asimov85] Asimov, Daniel. "The Grand Tour: A Tool for Viewing Multidimensional Data," *SIAM Journal of Scientific and Statistical Computing*, Vol. 6, No. 1, January 1985, 128-143.
- [Bajaj90] Bajaj, Chanderjit L. "Rational Hypersurface Display," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, 117-123.
- [Banchoff86] Banchoff, Thomas F. "Visualizing Two-Dimensional Phenomena in Four-Dimensional Space: A Computer Graphics Approach," *Statistical Image Processing and Graphics* (Edited by Edward Wegman and Douglas DePriest), Marcel Dekker, Inc. New York, 1986, 187-202.
- [Baraff90] Baraff, David. "Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation," *SIGGRAPH '90 Proceedings*, 19-28.
- [Bier86] Bier, Eric. 1986. "Skitters and Jacks: Interactive 3D Positioning Tools," *Proceedings of the 1986 Workshop on Interactive Graphics*, 183-196.
- [Bruss89] Bruss, Anna R. "The Eikonal Equation: Some Results Applicable to Computer Vision," *Shape From Shading* (Edited by Berthold Horn and Michael Brooks). MIT Press, 1989, 69-87.
- [Burton89] Burton Robert. "Raster Algorithms for Cartesian Hyperspace Graphics," *Journal of Imaging Technology* Vol. 15, No. 2, April 1989, 89-95.
- [Burton82] Burton, Robert P. and David R. Smith. "A Hidden-line Algorithm for Hyperspace," *SIAM Journal of Computing*, Vol. 11, No. 1, February 1982, 71-80.
- [Carey87] Carey, Scott, Robert Burton, and Douglas Campbell. "Shades of a Higher Dimension," *Computer Graphics World*, October 1987, 93-94.
- [Chen88] Chen, Michael, Joy Mountford, and Abigail Sellen. "A Study in Interactive 3-D Rotation Using 2-D Control Devices," *SIGGRAPH '88 Proceedings*, 121-130.
- [Curtis87] Curtis, Donald B., Robert P. Burton, and Douglas M. Campbell. "An Alternative to Cartesian Graphics," *Computer Graphics World*, June 1987, 95-98.

- [Davis89] Davis, Donald M. "Use of a Computer to Suggest Key Steps in the Proof of a Theorem in Topology," *Computers in Geometry and Topology* (Edited by Earl Taft *et. al.*), Marcel Dekker, Inc. New York, 1989, 121-130.
- [Donald91] Donald, Bruce R. and David R. Chang. "On the Complexity of Computing the Homology Type of a Triangulation," *IEEE*, 1991, 650-661.
- [Ellsworth90] Ellsworth, David, Howard Good, and Brice Tebbs. "Distributing Display Lists on a Multicomputer," *1990 Workshop on Interactive 3D Graphics*.
- [Evans81] Evans Kenneth, Peter Tanner, and Marcell Wein. . "Tablet-based Valuator that Provide One, Two, or Three Degrees of Freedom," *SIGGRAPH '81 Proceedings*, 1981, 91-97.
- [Francis87] Francis George. *A Topological Picturebook*, Springer-Verlag, 1987.
- [Freedman82] Freedman, Michael and Frank Quinn. *Topology of 4-Manifolds*, Princeton University Press, 1990.
- [Freeman91] Freeman, William, Edward Adelson, and David Heegar. "Motion Without Movement," *1991 Symposium on Interactive 3D Graphics*, 27-30.
- [Fuchs83] Fuchs, Henry, Greg Abram, and Eric Grant. 1983. "Near Real-Time Shaded Display of Rigid Objects," *SIGGRAPH '83 Proceedings*, 65-69.
- [Fuchs85] Fuchs, Henry, *et al.* "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *SIGGRAPH '85 Proceedings*, 111-120.
- [Fuchs89] Fuchs, Henry *et al.* "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *SIGGRAPH '89 Proceedings*. 79-88.
- [Gauch87] Gauch, Susan, Rich Hammer, Dals Krams, Teresa McBennett, and Dabby Saltzman. "An Evaluation of Factors Affecting Rotation Tasks in a Three-Dimensional Graphics System" TR87-002 Department of Computer Science, UNC-Chapel Hill, 1987.
- [Glassner91] Glassner, Andrew. "Spectrum: A Proposed Image Synthesis Architecture" (Lecture), *Frontiers in Rendering*, SIGGRAPH 1991.
- [Gunn91] Gunn, Charlie. "Visualizing Hyperbolic Space" (Research Report GCG43), The Geometry Center, University of Minnesota, 1991.
- [Haeberli90] Haeberli, Paul and Kurt Akely. "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *SIGGRAPH '90 Proceedings*, 309-318.
- [Hanrahan90] Hanrahan, Pat and Paul Haeberli. "Direct WYSIWYG Painting and Texturing on 3D Shapes," *SIGGRAPH '90 Proceedings*, 215-224.
- [Hanson91] Hanson, Andrew and P. Heng. "Visualizing the Fourth Dimension using Geometry and Light" *Visualization '91 Proceedings*.
- [Hanson92] Hanson, Andrew, and Pheng A. Heng. "Illuminating the Fourth Dimension," *IEEE Computer Graphics & Applications*, July 1992, 54-62.
- [Hinton04] Hinton, Charles. *The Fourth Dimension* (frontispiece), London and New York, 1904.
- [Hoffman90] Hoffman, Christoff and Jianhua Zhou. "Visualizing Surfaces in Four-Dimensional Space," Technical Report CSD-TR-960, Department of Computer Science, Purdue University, 1990.

- [Horn89] Horn, Berthold K. P. and Michael J. Brooks. "Obtaining Shape from Shading Information," *Shape From Shading* (Edited by Berthold Horn and Michael Brooks). MIT Press, 1989, 123-172.
- [Isaacson84] Isaacson, Paul L., Robert P. Burton, and Douglas M. Campbell. "Presentation of Hypothesized 4-D Phenomena," *Computer Graphics World*, August 1984, 48-63.
- [Klymenko90] Klymenko, Victor. "Hyperspace Perception: The Final Frontier" (Technical Report). Radiology Department, University of North Carolina at Chapel Hill, 1990.
- [Kajiya89] Kajiya, James, and Timothy Kay. "Rendering Fur With Three Dimensional Textures," *SIGGRAPH '89 Proceedings*, 271-280.
- [Kocak86] Kocak, Hüseyin, Frederic Bishopp, Thomas Banchoff, and David Laidlaw. "Topology and Mechanics with Computer Graphics: Linear Hamiltonian Systems in Four Dimensions," *Advances in Applied Mathematics*, Vol. 7, 1986, 282-308.
- [Lambe89] Lambe, Larry. "Algorithm for Computing the Cohomology of Nilpotent Groups," *Computers in Geometry and Topology* (Edited by Earl Taft et. al.), Marcel Dekker, Inc. New York, 1989, 189-210.
- [Lane80] Lane, J., Loren Carpenter, Turner Whitted, and Jim Blinn, "Scan Line Methods for Displaying Parametrically Defined Surfaces," *Communications of the ACM* Vol. 23, No. 1, 1980, 23-34.
- [Liu84] Liu, Mei-chi, Robert P. Burton, and Douglas M. Campbell. 1984. "A Shadow Algorithm for Hyperspace," *Computer Graphics World*, July 1984, 51-59.
- [Mitchell91] Don Mitchell, "Spectrally Optimal Sampling for Distribution Ray Tracing," *SIGGRAPH '91 Proceedings*, 157-164.
- [Milnor69] Milnor, John. *Morse Theory*, (Annals of Mathematical Studies, Vol. 51), Princeton University Press, 1969.
- [Molnar92] Molnar, Steven, John Eyles, and John Poulton. "PixelFlow: High-Speed Rendering Using Image Composition," *SIGGRAPH '88 Proceedings*, 231-240.
- [Moore88] Moore, Matthew and Jane Wilhelms, "Collision Detection and Response for Computer Animation," *SIGGRAPH '88 Proceedings*, 289-298.
- [Morse34] Morse, Marsden. *The Calculus of Variations in the Large*, American Mathematical Society, 1934.
- [Nielson86] Gregory Nielson and Dan Olsen, "Direct Manipulation Techniques for 3D Objects Using 2D locator Devices," *Proc. 1986 Workshop on Interactive Graphics*, 175-182.
- [Noll67] A. Noll, "A Computer Technique for Displaying n-dimensional Hyperobjects" *Communications of the ACM*, Vol. 10, 1967.
- [Phillips92] Phillips, Mark and Charlie Gunn. "Visualizing Hyperbolic Space: Unusual Uses of 4x4 Matrices," *1992 Symposium on Interactive 3D Graphics*, 209-214.
- [Potmesil82] Potmesil, Michael and Indranil Chakravarty, "Synthetic Image Generation with a Lens and Aperture Camera Model," *ACM Transactions on Graphics*, April 1982.

- [Ravenel89] Ravenel, Douglas C.. "Homotopy Groups of Spheres on a Small Computer," *Computers in Geometry and Topology* (Edited by Earl Taft *et al.*), Marcel Dekker, Inc. New York, 1989, 285-296.
- [Schweitzer82] Schweitzer, Dino and Elizabeth Cobb, "Scanline Rendering of Parametric Surfaces," *SIGGRAPH '82 Proceedings*, 265-271.
- [Seifert34] Seifert, H. and W. Threlfall. *A Textbook of Topology* (translation by Michael Goldman of *Lehrbuch der Topologie*, 1934). Academic Press, 1980.
- [Smith90] Smith, David, "Virtus Walkthrough" (Macintosh application and user manual), 1990.
- [Steiner87] Steiner, K. Victor and Robert P. Burton. "Hidden Volumes: The 4th Dimension," *Computer Graphics World*, February 1987, 71-74.
- [Stringham] Stringham, E. "Regular Figures in n-Dimensional Space," *American Journal of Mathematics*, 1880.
- [vanWyjk91] van Wijk, Jarke. "Spot Noise-Texture Synthesis for Data Visualization," *SIGGRAPH '91 Proceedings*, 309-318.
- [Weeks85] Weeks, Jeffrey. *The Shape of Space*, Marcel Dekker, Inc. New York, 1985.
- [Whitney44] Whitney, Hassler, "The Singularities of a Smooth n-manifold in $(2n-1)$ -space" *Annals of Mathematics*, Vol. 45, 1944.