Interactive Manipulation and Display of Two-Dimensional Surfaces in Four-Dimensional Space

David Banks
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Surfaces in 4-space generally produce self-intersections when projected to 3-space. The geometry of the projected surface changes as the surface rotates rigidly in 4-space. This paper presents techniques for interacting with such a surface, for recovering the geometry and depth information that the projection destroys, for computing the intersections and the surface when projected to 3-space, and for computing the silhouettes and the surface when projected to the screen. These techniques are part of an interactive system called Fourphront, which uses Pixel-Planes 5 as the graphics engine.

1 Introduction

Versatile high-performance graphics machines let us interactively manipulate surfaces in four dimensions. The projective geometry and linear algebra required for the job are well known [Semple], but surfaces in 4-space present challenges in designing a user interface and a set of visualization cues. This paper presents techniques to address these problems, using Pixel-Planes 5 as the graphics platform. In particular, we present techniques for gathering 3D input to manipulate a surface in 4-space, for providing visualization cues, and for applying 4D depth cues. These techniques are at the heart of an interactive system called "Fourphront."

Why study surfaces in 4-space? One reason is that topologists have yet to classify all the 3-dimensional compact surfaces, but have succeeded with the 2-dimensional surfaces (k-holed donuts and their non-orientable counterparts). Many of the 2-dimensional surfaces require four dimensions in which to imbed, and none of the compact 3-dimensional surfaces can imbed in three dimensions of Euclidean space. It might be enlightening to examine and compare surfaces that are topologically equivalent and that inhabit four dimensions of space. Do they look alike or not?

It is difficult even to illustrate the 3D classification problem with genuine examples; these are volumes without boundaries, residing in up to seven dimensions of space. Even the 2-dimensional surfaces may require four dimensions for their imbedding. Interactive computer graphics can be of service by providing a window on these surfaces in 4-space.

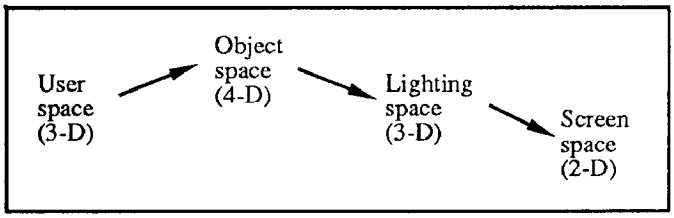


Figure 1. A user in 3-space manipulates a surface in 4-space, which projects to 3-spac and then onto the screen.

The three steps of our task (figure 1) are (§2) mapping input from user space to object space, (§3 and §4) projecting from object space to illumination space, and (§5) projecting from illumination space to the screen.

2 Mapping User Input to World Transformations

The illusion of reality is strongest when the user controls what scene it is that he views. Dynamic control of the transformation matrices requires an input device that offers a natural means for producing the object's motion. There are ten degrees of freedom that we wish to control for manipulating objects in 4-space: four extents of translation in the axial directions (**x**, **y**, **z**, **w**), and six Euler angles of rotation within the axial planes (**xw**, **yw**, **zw**, **xz**, **yz**, **xy**). The 4D rotations look very much like their 3D counterparts, although it becomes more appropriate to think of rotations occurring within a plane rather than occurring about an axis (figure 2). In 3-space, rotations leave a 1-dimensional subspace fixed; that subspace is the rotation axis. In 4D, rotations leave a 2-dimensional subspace fixed, while permuting the points within the 2-dimensional rotation plane and within the bundle of planes parallel to it. In general, the rotation matrix A for the $\mathbf{x}_i\mathbf{x}_j$ plane ($i < j$), contains the elements $a_{ii} = a_{jj} = \cos t$, $a_{ij} = -a_{ji} = (-1)^{(i+j)} \sin t$, and the remaining elements $a_{kl} = \delta_{kl}$. For a more thorough treatment on Euler angles in 4-space, and how to specify orientation, see [Hoffman]. The challenge in assigning the ten degrees of freedom in 4-space to input devices that exist physically in 3-space is to promote kinesthetic sympathy: the similarity of input-motion to object-motion [Gauch].

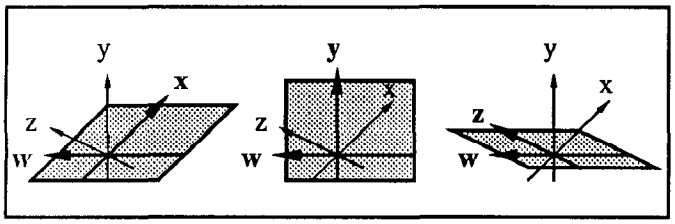


Figure 2. These are three of the six axial planes in xyzw-space, defined by the axis pairs xw, yw, and zw. The other three axial planes (xz, yz, and xy) lie in the 3-dimensional xyz-subspace.

2.1 Mapping 2D Input to 3D Transformations

The fact that an input device is constrained within a physical 3-dimensional world will impair kinesthetic sympathy. 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 a 2-dimensional input space. The popular techniques are to overload the input space, to partition the input space, to discard a dimension of control, or to create a cross-product of the input space by using multiple locators. The following is a highly compressed review of these techniques.

2.1.1 Overloading the Input Mapping

We can overload the input space (x', y', z') by extracting x' and y' components of the locator's velocity, and assigning the magnitude of circular acceleration to the z' component [Evans]. Converting these components into translations in x, y, and z preserves sympathy for x and y, and naturally suggests a screw-translation for z. 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 world space (figure 3).

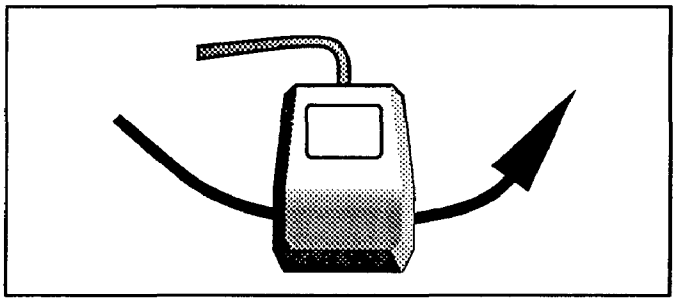


Figure 3. At the bottom point of this circular trajectory, the mouse's velocity is purely horizontal, while its acceleration is purely vertical.

2.1.2 Partitioning the Input Space

We can partition the input space into components, each of which maps the locator motion to the object motion in a different manner. The partition can be explicit, by determining in which of several control areas a cursor lies [Chen]. The partition can be implicit, by comparing the motion of the 2D locator to the orientation of a 3D cursor that is projected to input space [Nielson]. Whichever mapping is employed, the user must be prepared to change his motion when the input space switches context, and must be aware of which mapping is being invoked.

2.1.3 Discarding Input Mappings

There are several ways to discard a degree of control in order to eliminate a dimension from the range of the input mapping. For example, two angles determine a position on the unit 2-sphere. Rather than specify three Euler angles, we can use the locator's velocity vector to determine a rotation of the unit 2-sphere and hence of the 3-space it inhabits. Alternatively, we can map the input space to the tangent space at a point on a surface [Nielson, Bier, Hanrahan, Smith], in order to control the motion of the object by controlling its motion within that tangent plane. Of course the locator's motion becomes less sympathetic as the tangent plane deviates from the image plane. A more abstract problem is that path-planning can become very difficult when it requires a route through successive tangent planes to reach a target orientation. A surface in 3-space that isn't closed or that isn't everywhere differentiable may possess a Gauss map that does not cover the unit sphere. Such a surface is difficult or impossible to orient by controlling it through its tangent or normal space.

2.1.4 Taking a Cross Product of the Input Space

By using k locators, each with n degrees of freedom, we permit n^k degrees of freedom in the input space. These can be realized either as k physical locators, as one logical locator with a k-way selector to map physical-to-logical, or as a hybrid of the two. Thus, a single mouse button can select between two mappings of the mouse position into the world [Chen].

2.2 Mapping Spaceballs and Joysticks to 4D Transformations

What does the experience of mapping 2D input to 3D manipulation suggest for mapping 3D input into 4D manipulation? Consider each of the four approaches outlined above. (1) Overloading the input space can produce transformations in 4-space as side effects of an attempted 3D manipulation – side effects which novice users cannot easily undo. (2) Nielson's method for partitioning a locator's 2-dimensional space extends to 3D for translation, but it does not lend itself to rotations. (3) There are problems with discarding one or more dimensions of manipulation. First, mapping a velocity vector in 3-space into rotations of the unit 3-sphere in 4-space is a promising idea, but it is difficult to restrict the input so as to rotate the projection of the object within its projected 3D subspace. Second, the bigger the dimension of the space, the less of it can be visited by excursions in a 2D tangent plane to a point on a surface, so exploiting local surface properties pays a much smaller dividend than it did in 3-space. (4) Using multiple input devices can be inconvenient, requiring ten sliders or dials, five mice, four 3D joysticks, or two six-degree-of-freedom spaceballs.

What choice is best? There may be no single optimal technique, but multiple input devices at least promise a great deal of kinesthetic sympathy if their input space is 3-dimensional. The relative novelty of interactive manipulation in 4-space is a powerful motivation for designing a sympathetic interface. Not many people have developed a sense of how surfaces look as they rotate in 4-space. Consequently, we do well to approximate 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 then can we use them to create 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 world space. But the projection from 4-space to the screen will annihilate two orthogonal directions **z** and **w**, together with the 2-dimensional plane they define. This plane will apparently go "into" the screen at each point. Translation in the **z** or **w** directions and rotation in the **xw**, **yw**, **xz**, or **yz** planes thus present a problem. If the input device moves toward the screen, we can legitimately map that motion either to **z** or **w**. Either choice preserves kinesthetic sympathy, but the map is not unique. Rotation in the **zw** plane is also problematic. There is no physical rotation of a 3D input device sympathetic to this 4D rotation, since (in our physical 3-space) such a rotation would be confined to the 1-dimensional input space **z'**. The sympathetic maps are tabulated below (figure 4).

x'y'z' Input Space Translation Direction	⟶	xyzw World Space Translation Direction
x'		x
y'		y
z'		z or w

x'y'z' Input Space Rotation Plane	⟶	xyzw World Space Rotation Plane
x'y'		xy
x'z'		xz or xw
y'z'		yz or yw
??		zw

Figure 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 spaceball or a joystick into 4D transformations. A spaceball 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 spaceballs, either physically or logically.

The mapping from input space to object space can be defined as follows. $Spaceball_1$ assigns (**x',y',z'**) to (**x,y,z**) for calculating translations and rotations. $Spaceball_2$ re-interprets the **z'** coordinate, assigning it to **w** instead of to **z**, $Spaceball_2$ also makes the exception that rotations in its **x'y'**-plane map to rotations in the world'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 two physical spaceballs compete to produce **x** and **y** translations under this scheme; it is necessary then to squelch one spaceball's input to these translations. This makes the two-spaceball solution somewhat unattractive.

3D joysticks that use twist (about the joystick axis) as the third degree of freedom can map in a similar way to the spaceballs, using two joysticks to mimic the mappings of a single spaceball. The joystick rotates in each of three planes based at a common origin. Two of the rotations feel 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 5). We exploit this duality to sympathetically map these two motions into either rotation or translation in 4-space. Twist is not kinesthetically sympathetic to translation, but is at least suggestive of forward motion that results from rotating a screw.

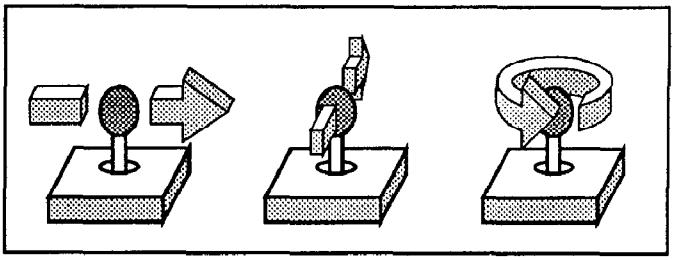


Figure 5. The 3D joystick rotates in the x'z', y'z', and x'y' planes, which can produce a momentary translation in the x and the y directions. In the input space coordinates, x' is rightward, y' is forward, and z' is vertical.

We need four (physical or logical) joysticks in order to supply the ten degrees of freedom necessary in 4-space. We can map pairs of (logical) joysticks the same way we map the spaceballs. Each pair allocates translations to one joystick and rotations to the other. 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.

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

Spaceball Translation Direction	Joystick Rotation Plane	⟶	World Translation Direction
x_1'	$x_1'z_1'$		x
y_1'	$y_1'z_1'$		y
z_1'	$x_1'y_1'$		z
x_2'	$x_2'z_2'$		x
y_2'	$y_2'z_2'$		y
z_2'	$x_2'y_2'$		w

Spaceball Rotation Plane	Joystick Rotation Plane	⟶	World Translation Direction
$x_1'y_1'$	$x_3'y_3'$		xy
$x_1'z_1'$	$x_3'z_3'$		xz
$y_1'z_1'$	$y_3'z_3'$		yz
$x_2'y_2'$	$x_4'y_4'$		zw
$x_2'z_2'$	$x_4'z_4'$		xw
$y_2'z_2'$	$y_4'z_4'$		yw

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

It is inconvenient to re-home the hands from one set of joysticks to another in the midst of manipulating an object. Fourphront therefore uses only two physical joysticks, one for each hand, multiplexed as four logical ones. One physical joystick functions as a logical pair that always maps (**x',y',z'**) to (**x,y,z**). This physical joystick embodies logical joysticks 1 and 3 in the table above. The other physical joystick (corresponding to logical joysticks 2 and 4 in the table) maps

(x',y',z') to (x,y,w), with the same caveat that it nonsympathetically maps rotations from the x'y' input plane to the zw world plane. A binary state variable (governed by a joystick button) determines whether to produce translations or rotations.

It is not uncommon to decouple the positioning and orientation operations in the input domain. Experience shows that that users also decouple 4D manipulations (the ones that involve the w-axis in world space) from 3D manipulations [Hoffman] in order to inspect the change that was made to the 3D projection moving the model in 4-space. So there is some justification in this splitting of the joystick control into four parts. The other natural decomposition would assign logical joysticks 1 and 2 to one device, and joysticks 3 and 4 to the other.

3 Projecting to 3D: Intersections, Transparency, and Silhouettes

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 7).

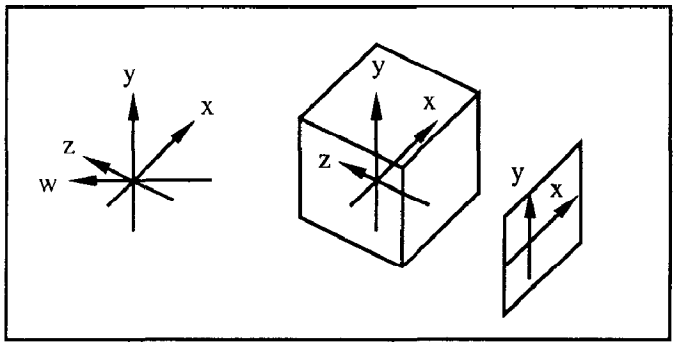


Figure 7. The (x y z w)-axes (left) project in the w-direction to the (x y z)-axes (middle), which project in the z-direction to the (x y)-axes of the image plane.

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 the usual 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 through most of its orientations .

A surface is *imbedded* if it has no self-intersections or singularities. An imbedded surfaces locally looks like a neighborhood in the plane – no creases, no crossings. If a surface imbeds in three dimensions, there's little need (from the standpoint of topology) to study it in four; thus the interesting surfaces are generally the ones that contain self-intersections when projected to 3-space, because they fail to imbed there. None of the one-sided surfaces imbed in 3-space. Happily, all tof he topological surfaces have incarnations that imbed in 4-space.

Typically a surface that we transform and rotate on our graphics machines 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 dutifully performs a crucial topological service: it separates 3-space into an inside and an outside. We 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 we have developed a sufficiently complete mental model of it. We need not cross the surface to the other side.

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 their geometry from us. 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 which we repair in §5.

3.1 Ribboning

To reveal the geometry of a self-intersecting surface, we can slice it into ribbons [Koçak]. 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 close 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. Banchoff has made productive use of this technique to illustrate the foliation of the 3-sphere in 4-space by animating a ribboned torus that follows a trajectory through the 3-sphere.

Surfaces with other topologies do not admit such a simple ribboning. We can slice a surface along level cuts as it sits in 4-space, but the cuts will sometimes produce x-shaped neighborhoods in the ribbons. Morse theory determines whether a surface can be successfully ribboned: the singularities of a Morse function on a surface must all be degenerate with the topology of a circle [Milnor, Morse].

3.2 Clipping

Rather than pre-compute sections of the surface to be sliced away, we can clip them out dynamically. The chief advantages are that (1) many graphics machines implement fast hither-clipping as part of their rendering pipeline; (2) no special treatment is required for the representation of the model; and (3) by clipping the surface as it moves, the user can inspect views of it that a single static segmentation cannot anticipate.

There are drawbacks to clipping. We usually think of clipping a surface against a plane. In fact, 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. Instead, a 4-dimensional halfspace clips the surface, and the boundary of the 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 we 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. We 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.

Recall that the immediate problem is to view the component pieces of a self-intersecting surface. In particular, to see beyond a 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. If this is truly the driving problem, we can sufficiently address it by clipping in that 3-dimensional space, and clipping strictly in the z-direction. This amounts to nothing more than hither clipping. To summarize: clipping in 4-space is mathematically easy but interactively hard. For the purpose of revealing hidden interiors, however, hither clipping suffices.

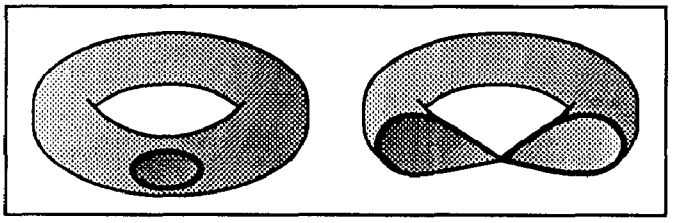


Figure 8. Clipping into a torus produces a figure-eight contour. Clipping reveals internal geometry, but complex contours can confuse the shape.

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 [Francis], but intersections and saddle points on a surface cause the clipped regions to look complex (figure 8). 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. Both of these shortcomings can be remedied by using more exotic, custom-shaped clipping volumes. Thirdly, clipping the frontmost patches of a surface exposes some of the hindmost patches, which may be behind the center of rotation for the objects. The visible part of the surface then seems to rotate in the direction antisympathetic to the input motion. This shortcoming is independent of the shape of the clipping volume.

3.3 Transparency

Ribboning and clipping simulate transparency via a binary classification. Both classify parts of the surface as completely opaque and the other parts as completely transparent. Why not use transparency outright? Ideally a semi-transparent surface presents all of its self-intersecting components on the screen so that the shape of each layer is discernible. 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 we have A atop B (yielding A's color); on the other side B atop A (yielding B's color). As the patches become simultaneously more transparent, their colors blend and the intersection becomes less distinguishable. Intersection curves figure prominently in the study of nonimbedded surfaces, so it seems a shame to apply transparency at their expense.

Disappearing silhouettes. A surface with many self-intersections may require a great deal of transparency to make the deep layers visible, but then the outermost layer becomes nearly invisible. 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.

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. 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 prevents us from ordering the model by a static data structure in 3-space, such as a binary space partition (BSP) tree [Fuchs83]. Does the BSP tree extend to surfaces in 4-space? Alas it does not; a polygon partitions 3-space by the plane in which it lies. But a plane does not separate 4-space.

In short, to render transparent polygons we 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 we want to see, but that opacity serves a positive purpose: to disambiguate 3D depth on a 2D display. Obscuration 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 us 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 is not so apparant in the neighborhoods that are viewed head-on. Phong highlights help exaggerate the curvature, thereby distinguishing the shape of a neighborhood. Where two transluscent surface patches interpenetrate, the Phong highlights can disambiguate which surface is in front, especially when we rock 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, and since any given surface is likely to exhibit self-

intersections when it is projected to 3-space. 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, we can address these demerits in the following ways. (1) Highlight the intersection curves; (2) Highlight the silhouette curves. (3) Order the polygons in sub-linear time. (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 §5 we discuss techniques for computing them 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 algorithms on Pixel-Planes: multipass transparency and deferred shading. In (back-to-front) multipass transparency, the model is sent to the SIMD renderers multiple times. On each pass, a pixel processor retains the geometry of the backmost polygon that it has not previously retained, then blends the shaded result into a temporary frame buffer. This technique requires two z-buffer areas per pixel processor. Deferred shading extracts the shading operation common to all primitives, and posponed applying the operation until after all the primitives have been z-buffered. Thus, only the necessary state information (*e.g.*, color, reflectivity, normal, transparency) is stored per pixel at the time the geometry of the primitive is rendered.

4 Projecting to 3D: Depth Cues

There are several cues that lend a 3D effect to images on a computer screen. Among them are obscuration, shadows, illumination, perspective, parallax, stereopsis, focus, and texture. These are natural cues that we use every day to derive a 3D model of our world from the 2D image of it on our retinas.

But now we confront a serious problem. By projecting the image of a surface in 4-space down to a 2-dimensional screen, not only do we lose depth information in the z-direction, but we lose it in the w-direction as well. What 4-dimensional depth cue does our retina employ that we can now supply when we render the surface? Evidently there is none. Since both the z and the w directions are perpendicular to the screen, we 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 our physical experience. How do the usual z-depth cues extend to four dimensions?

4.1 Obscuration and Shadows

We can drop down a dimension and liken the situation to viewing 1-dimensional curves in 3-space. Space curves rarely obscure or cast shadows on each other: only at isolated points, in general. Similarly, surfaces in 4-space only obscure each other or cast shadows on each other along mere isolated curves (in general). The result is that these cues are not especially helpful for recovering w-depth.

4.2 Illumination

Again we consider the lower-dimensional analog to our problem. Illumination is ill-defined along a curve in 3-space, since a space curve has an entire plane for its normal directions. The usual illumination equation does not apply. Several researchers have observed that any surface with co-dimension 1 submits to ordinary lighting techniques, and have jumped ahead to illuminating 3-dimensional surfaces in 4-space [Burton, Carey]. Burton lets a polygon inherit the normal vector of the 3-dimensional volume whose boundary includes it. This is like illuminating a polygonal surface in 3-space, but only displaying the result on the polygonal mesh. The problem with non-orientable surfaces imbedded in 4-space is that they do not bound any volume at all. Hansen inflates a surface to a small 3-dimensional volume, like wrapping a tube around a space curve, and then illuminates that bounding volume in 4-space and volume-renders it [Hansen]. The images are satisfying, but the technique is fairly slow, since rendering volumes is considerably slower than rendering polygons.

Illuminating surfaces in 4-space is thus an unresolved problem. Fourphront postpones illumination until the surface is projected into 3-space, so that shading looks familiar and realistic on the projected surface, and so that this strong z-depth cue is preserved. 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]. The obvious drawback with this approach is that the shading in 3-space reveals more about the shape of the projected surface than about the shape of the surface as it lies in 4-space.

4.3 Perspective

A perspective projection from 3-space to 2-space behaves like an orthogonal projection where 3-space is pre-warped: planes parallel to the image plane are first shrunk or magnified according to their distance. A perspective projection from 4-space to 3-space has the same general effect. Volumes shrink that are distant from, and parallel to, the volume of projection, but volumes grow that are close to the center of projection eye_4. In particular, translating a neighborhood in the w-direction causes its projection to shrink and approach the origin. This behavior can disambiguate relative w-depth. The nearer neighborhood changes size faster than the farther one.

4.4 Stereopsis and Parallax

Parallax and stereopsis are side-effects of perspective projection, and they offer additional w-depth cueing [Armstrong]. 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)?

Let us again drop down a dimension and examine the situation. Consider a viewpoint eye_3 in 3-space, and the image plane to which the world projects (figure 10). 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", we conclude 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. We have 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.

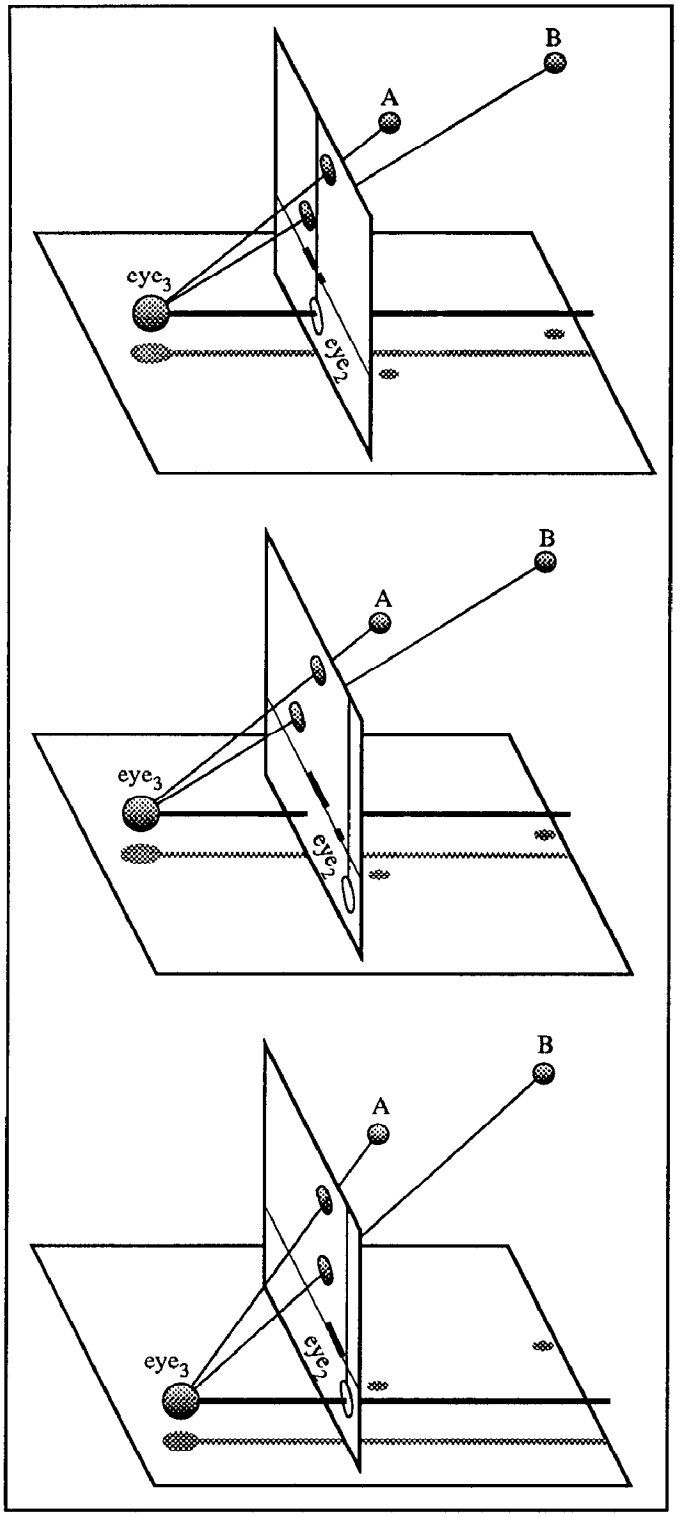


Figure 10. When there are two eye positions involved in projecting an image, either of them can produce parallax. In this figure, spheres A and B project from 3-space onto a 2-dimensional plane as disks. The disks project to a 1-dimensional line as segments. By tilting the page obliquely, you can see what the second eye sees. Moving an eye to the right will make thefarther object seem to move to the right o 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.

4.5 Texture

The texture applied to a surface can be defined dynamically in world space, so that as the surface moves in the w direction, the texture changes. One of the simplest textures is color modulated according to depth. This texture is well-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. The idea is evidently pretty obvious, since there are very old examples of its use [Hinton]. 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 [Freeman, van Wijk].

4.6 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 [Haeberli, Mitchell, Potmesil].

In 4-space we 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 we 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 eye_4 are generally large due to perspective, and often enclose the far-away neighborhoods that have shrunk toward the origin. If the outermost patches of a surface are opaque, they hide the interior geometry. This is the motivation for choosing a focal 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.

5 Finding Silhouettes and Intersections During Projection to 2D

This section describes a screen-oriented technique for locating silhouette curves and intersection curves. In §3 we described the powerful advantage transparency gives for visualizing self-intersecting surfaces, but noted that although transparency lets us see more layers of the surface, it strips those layers of some of their geometric content. In particular, the intersections and silhouettes are less apparent on transparent surfaces.

We can estimate the amount of computation required for calculating the geometry of these curves and for rendering semi-transparent surfaces. The conclusion is that even for a modest-sized polygonal model, the burden on the traditional front end of a graphics system becomes too great. Programmable SIMD renderers let us shift some of the computation away from the math processors on Pixel-Planes 5,

which makes it possible to display silhouettes and intersection curves of a dynamic 3D (projected) surface at interactive rates.

5.1 Calculating in 3-space

Consider the task of manipulating a surface composed of n = 2000 triangles (this is a skimpy polygon budget to spend on self-intersecting surfaces). The cost of transforming and ordering these semi-transparent triangles, along with calculating their silhouettes and intersections, is substantial. Depending on the particulars of the algorithms we employ, we can easily spend $O(n \log n)$ floating-point operations sorting the polygons (as required for transparency) and computing their intersections. Since the geometry is dynamic in 3-space as the surface rotates in 4-space, this cost is charged per frame. The transformations and projections from 4-space to the screen can take another $250n$ floating-point operations. So we easily face over 1.5 million floating-point operations for this meager data set. These estimates disregard all other necessary operations; the front-end system must sustain well over 30 MFLOPS in order to calculate the intersecting geometry at interactive speeds of 20hz. By using multiple CPUs to achieve this speed, we incur substantial communication cost or memory contention. In either case, the time complexity is super-linear in the number of polygons. The conclusion: avoid sorting and avoid analytically computing the intersections in 3-space.

Pixel-Planes 5 offers programmable SIMD logic-enhanced frame-buffers (the renderers) that can offload much of the burden from the geometry processors [Ellsworth, Fuchs89]. In particular, we can use the SIMD renderers to order the polygons, to find the silhouettes, and to find the intersections. For the case of 2000 triangles, the renderers can relieve the geometry processors of over half their floating-point burden and reduce their communication cost.

5.2 Silhouette Curves

Analytic Solution. 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 counts 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 we wish to highlight the silhouettes of the component patches that nest inside a transparent image. A simple way to find a silhouette (whose transverse is non-inflecting) is to locate every edge that is shared by two polygons, one facing forward and the other facing backward from the eye. 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. Note too that this technique only identifies silhouettes along mesh boundaries of a polygonal representation of the model, and not in the polygons' interiors.

We can analytically compute the silhouette for surface patches that are defined parametrically [Schweitzer, Lane], but this does not take advantage of the SIMD renderers of Pixel-Planes.

Screen-based Solution. Consider a screen-oriented approach to finding silhouettes. As a routine step in Phong-shading, the Pixel-Planes renderers hold the information necessary to locate silhouettes, namely, the interpolated surface normals and the eye vector. Each renderer covers a region on the screen and holds hundreds of bits of information per pixel in the region. These pixels are operated on in SIMD fashion. If the normal to a point on a polygon is orthogonal to the eye vector, the point lies on a silhouette curve.

We can use the renderers to perform a dot product between the normal vector and the eye vector at every pixel, which identifies the silhouette if the dot product is zero. (If the eye is sufficiently far away, the projection is nearly orthogonal, and it suffices to test just the z-component of the normal.) This yields, at best, a 1-pixel-thick line on a curved surface; at worst, it misses most pixels on the silhouette because of the imperfect sampling of the normal vector. We 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 11).

But thresholding has problems. As ε gets large, false silhouettes appear wherever the surface is sufficiently edge-on to the eye, and the silhouette becomes much fatter in some places than in others. The false silhouettes are inherent to thresholding since, for example, a planar section of the surface, and containing the eye, may have an inflection whose tangent lies arbitrarily close to the eye vector. The inflection point will appear as a silhouette point, even though there may be no silhouette in its vicinity.

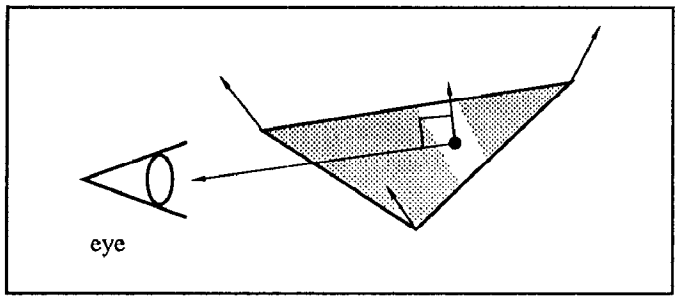


Figure 11. The surface normal is nearly orthogonal to the eye vector in the vicinity of a silhouette curve.

The reason that the thresholded silhouette has varying thickness is that the curvature of the surface may vary from place to place. A silhouette point with a large magnitude of normal curvature in the silhouette's transverse direction will witness its normal vector changing direction quickly along a path toward the eye. A large value of ε may still produce a thin silhouette region. Meanwhile, a silhouette point with a small magnitude of normal curvature in the transverse direction will witness its normal vector changing direction slowly along a path toward the eye. The same value of ε produces a thick silhouette, since there are points over a large area (even as seen from the eye) whose normals are nearly perpendicular to the eye vector.

Note that silhouettes need not be computed when a polygon first enters the pixel's memory. We need only look for silhouettes on visible polygon fragments that ultimately survive z-buffering. We defer shading until after the polygons have been transformed and their z-buffered geometry (including normal) has been stored in the pixel memory. Thus we incur the expense of silhouette computation only once per frame (or, for multipass transparency, only once per pass), rather than once per polygon.

Having found a silhouette, what do we do with it? The question concerns visualization in its abstract sense. How can we effectively map the internal state at a pixel onto the available

dimension 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 stand out. In fact, we can relax the binary classification of silhouttes in favor of a real-valued measure of "silhouetteness." If the intrinsic opacity of the surface at a point is α, let the effective opacity be $1-(1-\alpha)^{1/d}$, 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 transparent laminas. Viewed away from the normal by an angle whose cosine is d, a lamina of width w intercepts a ray through a distance w/d.

5.3 Intersection Curves

If the projected surface in 3-space were static, we could analytically compute the intersection curves [Baraff, Moore] once and for all. Since transformations in 4-space make its 3-space projection change shape dynamically, we recompute it each frame. This can be accomplished easily within the SIMD renderers. The straightforward approach to finding intersections is to modify the usual z-buffer algorithm. We test the z-value of each incoming polygon at each pixel against the contents of the z-buffer, retaining the polygon's state information if the polygon is closer. If the new value matches the z-buffer, we count it as an intersection. If we have flagged an intersection and then a closer polygon comes along, we unset the intersection flag. The result is that all the frontmost intersections will be flagged.

The proof of correctness is easy. Let $\{P_i\}$ be the set of polygons that cover a pixel, indexed by the order in which they arrive, 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. At the end of the pass, we have found an intersection. 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 formally intersect each other along it. Polygons whose edges pass through pixel centers will "intersect" at those pixels. These are spurious intersections, and not the kind of intersection we are trying to show. We could be careful not to scan-convert pixels more than once on 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 intersections we wish to highlight are those of polygons diving through each other, whose normals are different where they interpenetrate. Since the SIMD renderers interpolate vertex normals, that information is available per pixel. We can thus modify the z-comparison, requiring that the dot product of the new normal with the old normal be less than unity in magnitude.

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). We remedy this problem by thresholding. If the incoming pixel is within ε of the z-buffer value, we consider it 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 separation remains small over a large area of the screen, and the curve that satisfies $|z_{new} - z_{old}| < \varepsilon$ 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. We can use the interpolated normals of the polygons at pixels near the intersection in order to approximate a fixed-width intersection curve. But note that the added computation is charged per polygon, and cannot be deferred to end-of-pass unless we retain the geometric state of both polygons. Also note that most implementations of the z-buffer algorithm interpolate reciprocal-z across the polygon. Over small extents or for large original values of z, thresholding produces nearly the same behavior even when using the reciprocal. But for locating intersections across large ranges, it is wise to recover the true depth.

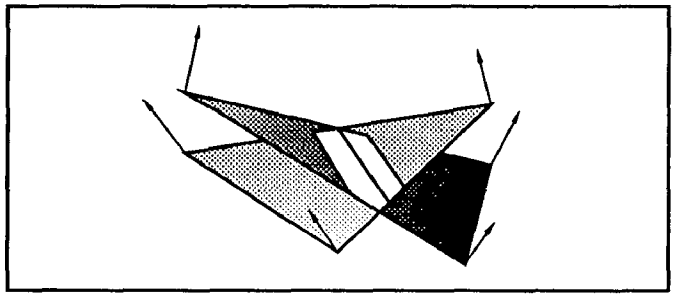


Figure 12. At their common intersection, two polygons share z-values. The z-values are within some threshold of each other along a thickened intersection curve.

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. This artifact is hard to overcome without using pixel-to-pixel communication.

6 Future Work

There are several research areas that this project has identified. A hemi-3-sphere can be mapped to the input space of a spaceball. How effective are the induced rotations in 4-space, and can the user produce the rigid motion within the 3-space to which a surface projects? Surfaces can be clipped in 4-space against volumes with 3-dimensional boundaries. Are there effective ways to shape, to position, and to display the volume or its boundary interactively? Is there an effective algorithm (like the BSP tree) for precomputing the rendering order for polygons projected from 4-space to the screen? Is there a speedy and natural way to illuminate surfaces in 4-space? What is the best interface for producing uncoupled parallax in either 4-space or the 3-space to which it projects? In what ways can texture be used as a w-depth cue? A quadric approximation to a surface contains curvature information, which can improve both the silhouette and intersection calculation for fixed-width curves. What are fast ways to produce this second-degree approximation and fast ways to use it on a per-pixel basis? Our consideration of silhouettes was motivated by the loss of geometric content that transparency produces. Hence we discussed silhouettes as seen by eye_3. What useful information do eye_4 silhouettes add to a surface?

7 Conclusions

The shape of surfaces in 4-space can be difficult to comprehend. Interactive computer graphics provides an excellent tool for making the surfaces seem more real, since we can manipulate them ourselves. The effort is full of trade-offs. In order to control all the degrees of freedom in 4-space, we need multiple input devices in 3-space. We can apply transparency in order to reveal the interior of a self-intersecting projection, but then we lose the intersections and the silhouettes. We can then highlight those special curves, but at the expense of the system's performance or memory. We can steal some of the usual z-depth cues and use them as w-depth cues, but that tends to make z-depth more ambiguous again.

This paper has focused on shortcomings of the various techniques in order to encourage other people to enter the fray and invent solutions. Until the advent of the powerful graphics computers we have today, mathematicians could only imagine interacting in four dimensions. Experience with Fourphront demonstrates that the effort can pay off, that we can open a window on the truly "virtual world" of four dimensions. The collateral spinoffs are algorithms that can be of service to the more pedestrian problems in three dimensions.

8 Acknowledgements

Fourphront is a descendant of interactive applications that were written for Pixel-Planes by Trey Greer (Front), Howard Good (Pphront), and Vicki Interrante (Xphront), all based on a PHIGS-like library developed for Pixel-Planes. Many people had a hand in improving its function and performance. Among them are Greg Turk (debugging), Howard Good (callback functions, one-sided picking), Brice Tebbs (algorithm designs), David Ellsworth (animated cursor, parallel pick), Andrew Bell (multipass transparency), Marc Olano (conditional executes, parallel pick), and Carl Mueller (frame dumps). Greg Turk was instrumental in creating the prototype on Pixel-Planes 4. Jeff Weeks (University of Minnesota) was the first to suggest using the flat imbedding of the torus in 4-space. Nelson Max (Lawrence Livermore National Labs) suggested displaying intersection curves. Robert Bryant (Duke University) and James Stasheff pointed out connections to Morse Theory. Oliver Steele (Apple Computer) helped devise certain parametric models in 4-space. Fred Brooks provided the initial support for this project, and Stephen Pizer currently directs it. Thanks also go to the reviewers, who proposed visualizing silhouetteness by mapping it to opacity.

9 References

William Armstrong and Robert Burton, "Perception Cues for n Dimensions," *Computer Graphics World* (Mar 1985), pp. 11-28.

David Baraff, "Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation," *SIGGRAPH '90 Proc.* (1990), pp. 19-28.

Eric Bier, "Skitters and Jacks: Interactive 3D Positioning Tools," *Proc. 1986 Workshop on Interactive Graphics,* (1986) pp. 183-196.

Robert Burton, "Raster Algorithms for Cartesian Hyperspace Graphics," *Journal of Imaging Technology* (15:2 Apr 1989), pp. 89-95.

Scott Carey, Robert Burton, and Douglas Campbell, "Shades of a Higher Dimension," *Computer Graphics World* (Oct 1987), pp. 93-94.

Michael Chen, Joy Mountford, and Abigail Sellen, "A Study in Interactive 3-D Rotation Using 2-D Control Devices," *SIGGRAPH '88 Proc.* (1988), pp. 121-130.

David Ellsworth, Howard Good, and Brice Tebbs, "Distributing Display Lists on a Multicomputer," *Proc. 1990 Symp Interactive 3D Graphics.* (1990).

Kenneth Evans, Peter Tanner, and Marceli Wein, "Tablet-based Valuators that Provide One, Two, or Three Degrees of Freedom," *SIGGRAPH '81 Proc.* (1981), pp. 91-97.

George Francis, *A Topological Picturebook*, Springer-Verlag (1987).

William Freeman, Edward Adelson, and David Heegar, "Motion Without Movement," *Proc. 1991 Symp Interactive 3D Graphics.* (1991) pp. 27-30.

Henry Fuchs, Greg Abram, and Eric Grant, "Near Real-Time Shaded Display of Rigid Objects," *SIGGRAPH '83 Proc.* (1983) pp. 65-69.

Henry Fuchs *et al.*, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *SIGGRAPH '89 Proc.* (1989), pp. 79-88.

Susan Gauch, Rich Hammer, Dals Krams, Teresa McBennett, and Dabby Saltzman, "An Evaluation of Factors Affecting Rotation Tasks in a Three-Dimensional Graphics System" TR87-002 Dept Comp. Sci, UNC-Chapel Hill (1987).

Paul Haeberli and Kurt Akely, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," *SIGGRAPH '90 Proc.* (1990), pp. 309-318.

Pat Hanrahan and Paul Haeberli, "Direct WYSIWYG Painting and Texturing on 3D Shapes," *SIGGRAPH '90 Proc.* (1990), pp. 215-224.

Andrew Hansen and P. Heng, "Visualizing the Fourth Dimension using Feometry and Light" *Visualization '91.* (1991).

Charles Hinton, *The Fourth Dimension* [Frontispiecs], London and New York, 1904.

Christoff Hoffman and Jianhua Zhou, "Visualizing Surfaces in Four-Dimensional Space," Technical Report CSD-TR-960, Computer Sciences Department, Purdue University (Mar 1990).

Hüseyín Koçak, Frederic Bisshopp, Thomas Banchoff, and David Laidlaw, "Topology and Mechanics with Computer Graphics: Linear Hamiltonian Systems in Four Dimensions," Advances in Applied Mathematics 7 (1986), pp. 282-308.

J. Lane, Loren Carpenter, Turner Whitted, and Jim Blinn, "Scan Line Methods for Displaying Parametrically Defined Surfaces," *Communications of the ACM* 23:1 (Jan 1980), pp. 23-34.

J. Levine, "Imbedding and Immersion of Real Projective Spaces" *Proc. Amer. Math. Soc.* 14 (1963).

Don Mitchell, "Spectrally Optimal Sampling for Distribution Ray Tracing," *SIGGRAPH '91 Proc.* (1991), pp. 157-164.

John Milnor, *Morse Theory*, (*Annals of Mathematical Studies* 51), Princeton University Press (1969).

Matthew Moore and Jane Wilhelms, "Collision Detection and Response for Computer Animation," *SIGGRAPH '88 Proc.* (1988), pp. 289-298.

Marsden Morse, *The Calculus of Variations in the Large*, American Mathematical Society (1934).

Gregory Nielson and Dan Olsen, "Direct Manipulation Techniques for 3D Objects Using 2D locator Devices," *Proc. 1986 Workshop on Interactive Graphics,* (1986) 175-182.

A. Noll, "A Computer Technique for Displaying n-dimensional Hyperobjects" *Comm. ACM* **10** (1967).

Michael Potmesil and Indranil Chakravarty, "Synthetic Image Generation with a Lens and Aperture Camera Model," *ACM Transactions on Graphics* (Apr 1982).

Dino Schweitzer and Elizabeth Cobb, "Scanline Rendering of Parametric Surfaces," *SIGGRAPH '82 PROC.* (1982) pp. 265-271.

J. Semple and G. Kneebone, *Algebraic Projective Geometry*, Clarendon Press, Oxford (1952).

David Smith, "Virtus Walkthrough" [Macintosh application and user manual].

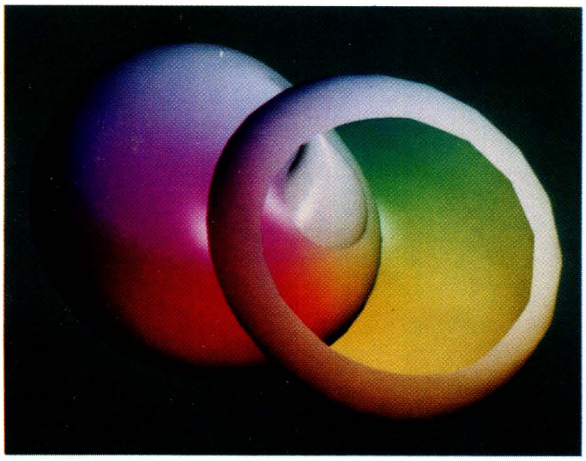
Stringham, "Regular Figures in n-Dimensional Space," *American Journal of Mathematics* (1880).

Jarke van Wijk, "Spot Noise-Texture Synthesis for Data Visualization," *SIGGRAPH '91 PROC.* (1991) pp. 309-318.

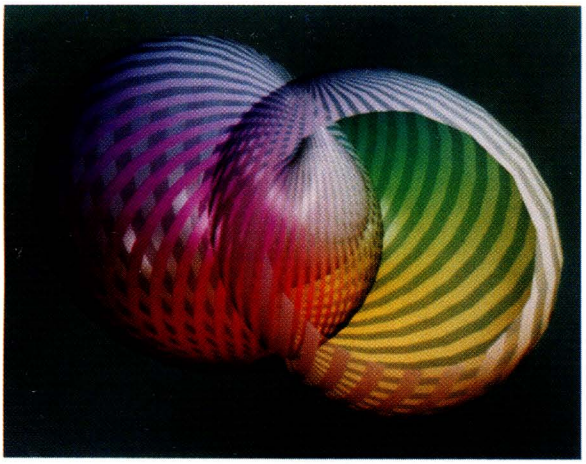
Hassler Whitney, "The Singularities of a Smooth n-manifold in (2n-1)-space" *Ann. of Math.* **45** (1944).

10 Illustrations

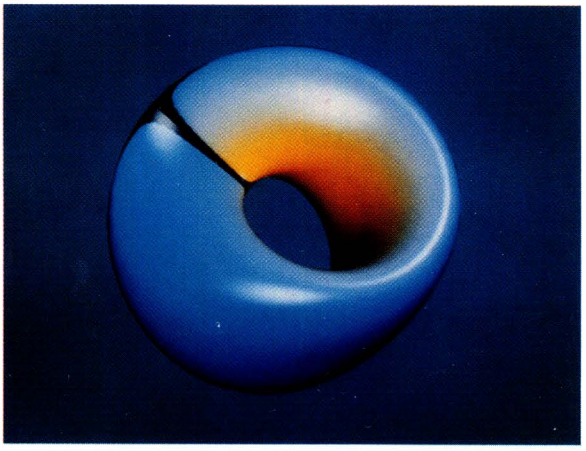
The surfaces in the color plate section were rendered on Pixel-Planes 5. Each surface was transformed, illuminated, and rendered on 5 in 0.2 seconds or less, and each has between 4k and 10k polygons. There are two light sources: one slightly left of the eye, and one above and to the right of the eye.



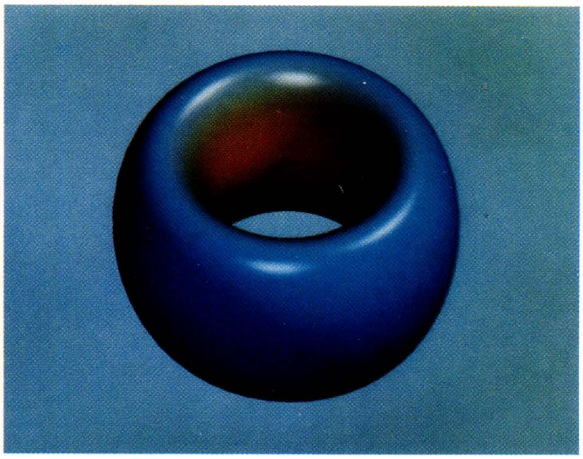
A topological sphere knotted in 4-space. The hither clipping plane in 3-space reveals some of the internal geometric complexity of the surface.



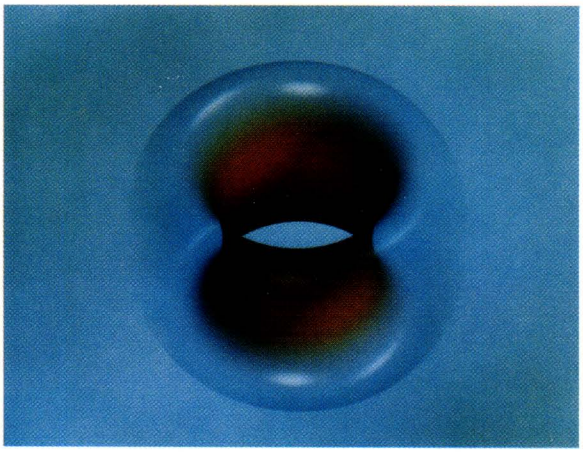
The knotted sphere sliced into ribbons. The inter-ribbon gaps are semi-transparent to suggest the continuity of the geometry. Note the moiré patterns emerging in the middle.



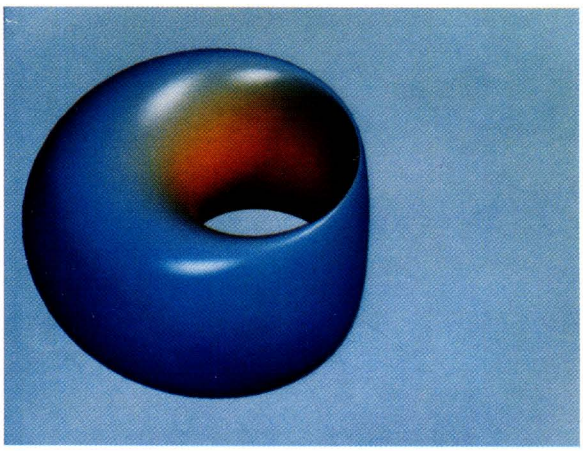
Klein bottle imbedded in 4-space, with color growing more amber in the w-direction. The surface self-intersects in 3-space (black line) but not in 4-space, as revealed by the differing colors on either side of the intersection curve.



Torus imbedded in 4-space as (cos *s*, *sin s*, *cos t*, *sin t*). The inner core of the torus is farther in w than the outer core, hance its color is more amber, and its size is diminished by perspective.

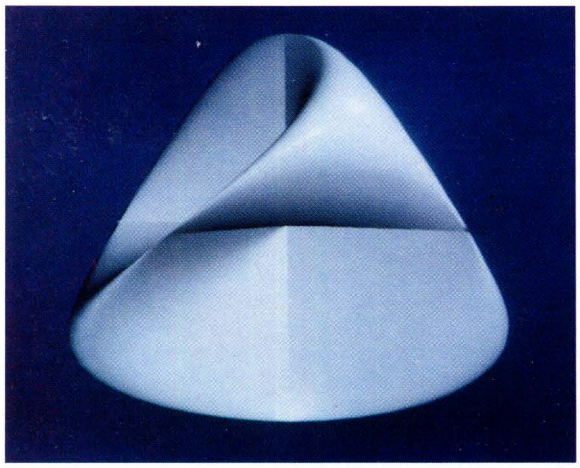


Opacity increasing in the w-direction. The opaque interior reinforces the interpretation that the inner part of the torus is farther away in w than the outer part.

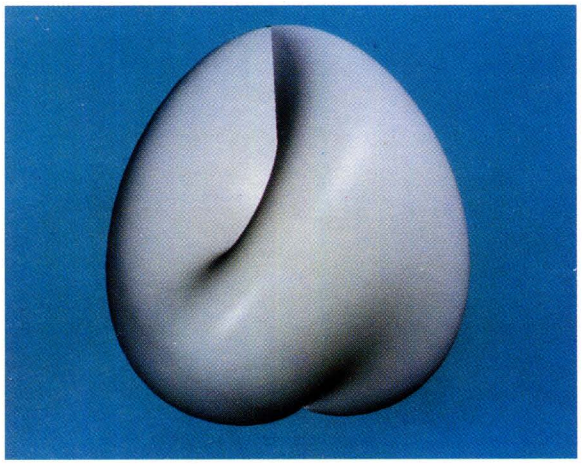


Torus viewed from a different eye point. By shifting the eye to the right in 4-space, we see farther neighborhoods shift right relative to nearer neighborhoods. Thus the farther, inner core of the torus slides right compared to the outer core.

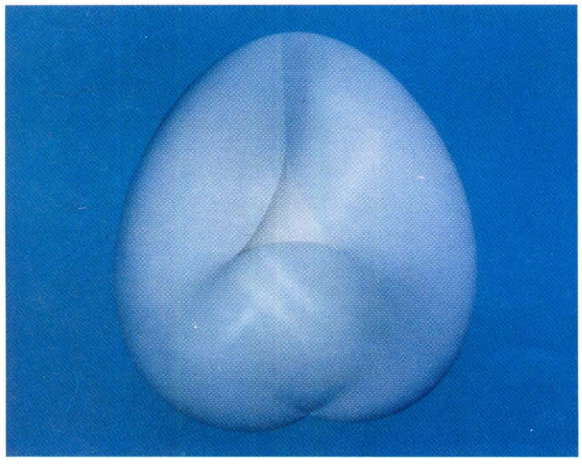
Banks, "Interactive Manipulation and Display of Two-Dimensional Surfaces in Four-Dimensional Space"



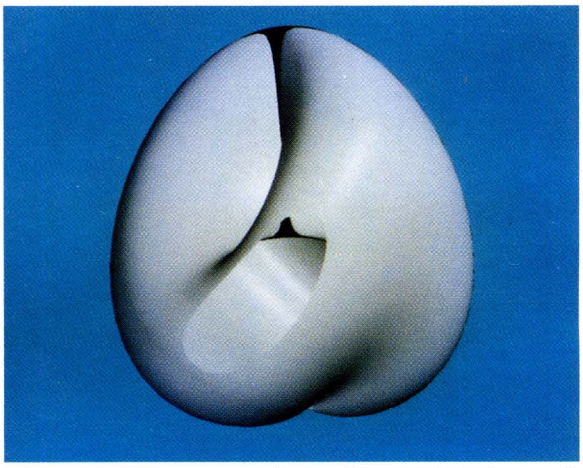
A projective plane that self-intersects in 4-space. The upper part of the vertical self-intersection persists under all rotations in 4-space.



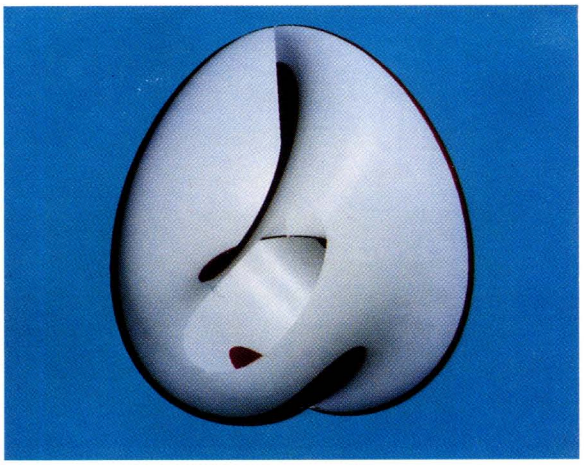
Rotated view of the projective plane. The intersection curve has a terminus at the top of the figure and another terminus midway down the surface, which the frontmost neighborhood hides.



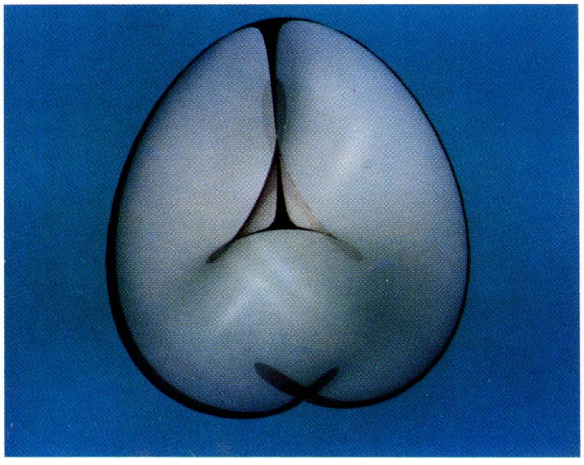
Projective plane rendered with transparency. The interior parts of the surface are visible, showing the lower terminus of the intersection curve. But the intersection curve is less prominent.



Hither clipping the opaque projective plane reveals the bottom of the black intersection curve. The curve is thinner where the intersecting patches dive steeply through each other.



Clipping to reveal internal silhouettes. Their width varies with the surface's curvature. There is a false silhouette where the bottom of the surface inflects in the eye plane across the curve.



Projective plane rendered with transparency, silhouettes, and intersections.

Banks, "Interactive Manipulation and Display of Two-Dimensional Surfaces in Four-Dimensional Space"