Robust and Efficient Surface Intersection for Solid Modeling

By

Michael Edward Hohmeyer

B.A. (University of California) 1986

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved:

Chair: Brian A. Barsky .... 3 Feb 92
Date

Robert L. Taylor .... 10 Feb 92

Raimund Seidel 92/02/10

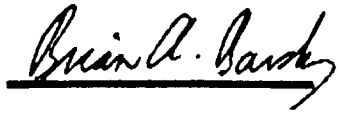
Christoph M. Hoffmann .... 7 Feb 92

* *

Robust and Efficient Surface Intersection for Solid Modeling

by

Michael Edward Hohmeyer



Brian A. Barsky
Thesis Chair

Abstract

Solid Modeling requires robust and efficient surface intersection algorithms that handle a very general class of surfaces, including rational Bézier and B-spline surfaces. Currently, most algorithms that run in an acceptable amount of time lack a theoretical basis and thus are not guaranteed to be reliable. Most that have a solid theoretical basis require enormous computation and are thus impractical.

This thesis presents an algorithm that, given two surfaces not intersecting in any singular points, will find all intersection curves. If there are any singular points the algorithm will find and report at least one. The algorithm does not require tolerances except those associated with machine arithmetic, and can handle any surface representation for which bounds on the position of the surface as well as its Gauss map (the set of normals to the surface) are available. The implemented algorithm is demonstrated on actual manufacturing data, as well as on some pathological examples.

The algorithm is based on a new loop detection criterion. Loop detection criterion-based intersection algorithms recursively subdivide two surfaces until no surface patches intersect in a closed loop. The intersection curves of the original patches can be reliably identified by finding the edge-surface intersections of the resulting sub-patches.

The thesis discusses in detail various problems that are used to implement the intersection algorithm. These include algorithms for: computing bounds on Gauss maps for rational B-splines and Bézier surfaces; intersecting curves with surfaces; solving unbounded linear programming problems; finding tangent directions at singular and non-singular intersection points; handling singularities and degeneracies; and compactly and accurately representing intersection curves.

Robust and Efficient Surface Intersection for Solid Modeling

by

Michael E. Hohmeyer

Contents

List of Figures

Chapter 1

Surface Intersection

1.1 Introduction

The problem of intersecting surfaces has been studied from the time of the ancient Greeks to the present, and yet many important issues remain unresolved. This thesis presents a robust and efficient algorithm for intersecting two surfaces and addresses many related sub-problems.

The work done from the time of ancient Greeks until the 1960's has dealt in a practical way with intersections of low degree surfaces and in a theoretical way with intersections of high degree surfaces. A Computer Aided Design and Manufacturing tool called Solid Modeling now requires a surface intersection algorithm that can deal with these high degree surfaces in a practical way. Such an algorithm must be reliable enough to either produce the correct result or determine that it is unable do so. It must be efficient enough to operate in roughly a minute, on a commonly available computer, on examples typical of those used in manufacturing.

Of the techniques proposed to deal with high degree intersections, those that run in an acceptable amount of time lack a theoretical basis and will fail on examples of sufficient complexity. Those that have a solid theoretical basis require enormous computation and thus remain impractical. This dilemma is being resolved by loop detection based surface intersection algorithms. Such algorithms generally work as follows: The bounds on the surfaces are tested for intersection. If they do not intersect, the algorithm returns. Otherwise, the algorithm applies a test to determine if it is not possible for the surfaces to intersect in a loop. If the surfaces do not pass the criterion they are subdivided and the algorithm is

run on all sub-patch pairs. Such an algorithm will either decompose the surfaces into pairs that do not intersect in a loop, or will discover a singularity.

The test to determine if it is not possible for two surfaces to intersect in a loop is called a loop detection criterion (it could more accurately be called loop exclusion criterion, but the former term is retained for historical reasons). This criterion makes determinations based on the analysis of the bounds on: the position of surfaces; their partial derivatives; and their Gauss maps.

When the algorithm has broken the surfaces into sub-patch pairs that do not intersect in loops, it determines the intersection curves by computing the intersection of the edges of one surfaces with the other surface and vise-versa. By analyzing the number of edge-surface intersections found as well as the orientation of the intersection curve at the edge-surface intersections, the algorithm can delimit the individual curves.

In this thesis a new loop detection criterion is presented. This criterion establishes that if the convex hulls of the Gauss maps of the two surfaces under consideration do not intersect and are not antipodal, then the surfaces cannot intersect in a loop. The loop detection criterion can consequently be implemented with a linear programming algorithm. The use of linear programming allows the algorithm to determine that no loops can exist in situations where other criteria cannot. The resulting surface intersection algorithm is therefore required to perform less subdivision. The algorithm is applicable to a class of surfaces that includes rational B-spline and Bézier surfaces as well as implicit polynomial surfaces. The algorithm's practicality is demonstrated on actual manufacturing data, as well as on some pathological examples.

The surface intersection algorithm requires a number of sub-algorithms in order to operate. To determine if the bounds on two surfaces intersect, it is necessary to find the minimum distance between the convex hulls of two point sets. To determine if the convex hulls of the two Gauss maps intersect or are antipodal, it is necessary to find the great circle on a sphere so that the distance from the convex hull of a point set on the sphere to the great circle is maximized. These two problems are solved using a slightly modified linear programming algorithm which is discussed in detail. Performance data on these techniques are given on problems of parametrizable difficulty. It is shown that linear programming techniques outperform simple bounding box techniques as the difficulty of the problem increases (Section 4.2).

An algorithm to compute the bounds on the Gauss maps of rational B-spline,

Bézier, and implicit surfaces is given. This algorithm computes a scalar multiple of the normal to the surface in the B-spline or Bézier form. The control points of the resulting surface are used as a bound on the Gauss map (Section 4.4).

When two patches have passed the loop detection criteria and their edge-surface intersection points have been found, it is necessary to find the orientation of the intersection curve at these points. This is done by finding the parameter space tangent directions precisely. Algorithms are given for doing this at singular and non-singular intersection points (Section 4.6).

The intersection of two high degree surfaces generally cannot be represented as a collection of parametric polynomial curves. Thus, approximations such as spline curves are often used. An compact and very accurate alternative is discussed. In this representation, the two surfaces that define the curve are recorded along with parameter space approximations. By combining this data with a small number of algorithms one can create an object that behaves much like a spline curve (Chapter 5).

In order to accelerate the detection of singularities an iterative numerical technique is given. It is shown that this iteration will converge to node and isolated singularities. A strategy is described for intersecting surfaces that contain these types of singularities (Chapter 6).

An algorithm is also given for dealing with degeneracies that a arise during the running of the algorithm. Degeneracies that are created by the algorithm itself are resolved using a backtracking method. Degeneracies that are intrinsic to the problem are merely found and reported to the user (Chapter 7).

1.2 A Brief History of Surface Intersection

Computation about complex solids relies upon the computation of surface intersections. The history of surface intersection begins with the history of curves, and that in turn begins with the Greeks. Socrates recognized the lack of knowledge about solids and saw that the study of plane curves would lead to knowledge of solids.

> *Then take a step backward, for we have gone wrong in the natural order of the sciences.*
>
> *What was the mistake? he said.*

After plane geometry, I said, we proceeded at once to solids in revolution, instead of taking solids in themselves; whereas after the second dimension, the third, which is concerned with cubes and dimensions of depth, ought to have followed.

This is true, Socrates, but so little seems to be known about these subjects.

Why, yes, I said, and for two reasons:- in the first place no government patronizes them; this leads to a want of energy in the pursuit of them, and they are difficult; in the second place, students can not learn them unless they have a director. But then a director can hardly be found, and even if he could, as matters now stand, students, who are very conceited, would not attend to him. That, however, would be otherwise if the whole State became the director of these studies and gave honor to them; then disciples would want to come, and there would be continuous and earnest search, and discoveries would be made; since even now, disregarded as they are by the world, and maimed of their fair proportions, and although none of their votaries can tell the use of them, still these studies force their way by their natural charm, and very likely, if they had the help of the State, they would some day emerge into light.

Yes, he said, there is a remarkable charm in them. But, I do not clearly understand the change in the order. First you began with the geometry of plane surfaces?

Yes, I said.

And you have placed astronomy next, and then you made a step backward?

Yes, and I have delayed you by my hurry; the ludicrous state of solid geometry, which in natural order should have followed, made me pass over this branch and go on to astronomy, or the motion of solids.

PLATO *The Republic*

The Greeks distinguished between three kinds of curves: plane curves, which can be constructed with a straightedge and compass; solid curves, the intersections of cones with a plane; and linear curves which were also called mechanical curves. The early Greek philosophers did not like to link their philosophy to the physical and thus regarded the linear curves as a somewhat inferior class.

Conic sections were probably first investigated by Menaechmus, a pupil of Eudoxus and a member of Plato's Academy in the fourth century B.C.. His results are contained in Euclid's *Conics*. Apollonius, in his *Conic Sections*, systematizes the results presented by Euclid and adds new proofs. *Conic Sections* was "so monumental that it practically closed the subject to later thinkers" [57]. Indeed, it wasn't until the seventeenth century that one sees any new contributions to the field.

In the seventeenth century, Desargues employed the techniques of the just emerging field of projective geometry to show how the conic sections (the ellipse, the hyperbola, and the parabola) are related to one another. This resulted in a vast simplification of the theory of conics sections.

A contemporary, John Wallis, was the first to derive the algebraic equations of the conic sections (1655). At the time, algebraic reasoning was not regarded to be as rigorous as geometric reasoning. Wallis' work went a long way to validate the algebraic approach.

In terms of the intersection of surfaces, it was still only known how to intersect a plane and a cone, perhaps partly due to the Greek disinclination to consider other types of curves. Descartes challenged this prejudice. He believed that any curve which could be expressed by an algebraic equation should be accepted as a perfectly valid. Leibniz even protested the requirement that a curve have an algebraic equation.

From this point on, there was a continual effort to transform geometric questions into algebraic questions and an effort to do most of one's thinking in the algebraic domain rather than in the geometric. In the eighteenth century Antoine Parent, John Bernoulli, Alexis-Claude Clairaut, and Jacob Hermann showed how surfaces can be thought of as the solution to an equation in three variables. This resulted in the next step forward in surface intersection, the proof by Gaspard Monge and Jean-Nicolas-Pierre Hachette that a plane intersects a second degree surface in a second degree curve.

In this way, intersection problems were transformed to problems involving plane algebraic curves. The problem then was to understand planar algebraic curves. One simply needed to classify all the curves for every degree. The degree one curve were simply lines. The degree two curves were the conic sections. Newton, in *Enumeratio Linearum Tertii Ordinis*, began by classifying degree three curves. Others worked on forth degree curves. The number of species of curves found was quite large and the process of enumerating them quite laborious. Rather than to proceed in this direction, mathematicians attempted to understand various properties of algebraic curves such as singularities, and the number points in which two curves could intersect.

Noether and Halphen showed that any algebraic space curve could be birationally transformed into a plane algebraic curve. Thus, the intersection of any two algebraic surfaces could always be transformed into a plane algebraic curve. These results would vanish into obscurity for nearly one hundred years before being applied to the problem of intersecting surfaces.

A method to determine the shape of a general algebraic curve, however, did not seem forthcoming. There was some effort in this direction. In [52], C. M. Jessop describes a machine with which one could trace arbitrary algebraic curves. The machine (which was almost certainly never built) consisted of beams, pivots, and sliders, and was quite complex. This work is interesting mostly because it indicates that work on tracing algebraic curves was not going to proceed by paper and pencil alone. The aid of a machine was needed.

1.3 The Computer

With the advent of surface modeling systems on computers came a renewed interest in computing surface intersections. These intersections describe, among other things, the motion of mechanical cutting tools, cross sections of surfaces, and boundaries of surface patches. Since the surfaces used in these modeling systems were much more complex than simply cones, cylinders, or spheres, the problem arose to find the intersection of *arbitrary* surfaces. The initial approaches to surface intersection were strongly influenced by the common representation of curves and surfaces in modeling systems, which was parametric as opposed to implicit. Practitioners found that most mathematics dealing with surfaces assumed that the surfaces were expressed in implicit form. Quite some time had passed since Noether and Halphen so it wasn't until two decades later that it was realized that all surfaces which could be represented in parametric form could also be represented implicit form [93]. For example, in [27] page 258, a handbook of surface modeling techniques, Faux states that

> Great advantage would be had if one surface could be translated into implicit form. Unfortunately, the most commonly used parametric surfaces, the bicubic patches, cannot be expressed in this way.

The designers of these first algorithms, however, proceeded as best they could without much of the knowledge developed in the nineteenth century, [21,30,35,70]. Their early surface intersection algorithms functioned well on surfaces of a fixed complexity, but failed on sufficiently difficult problems. Given the speed of computers at the time, one had to wait for the intersection algorithm and would be inclined to examine the results partly as a reward for one's patience and partly to ensure that the result was correct. If the surface intersection algorithm did not produce the correct result, the user had the option of changing some of the parameters of the algorithm or, in any case, notifying the programmer.

However, as layers of software took the place of the human user and computers became faster, it became less and less likely that the output of an intersection algorithm would be viewed by human eyes before it was allowed to go on and (potentially) interfere with the correct functioning of other algorithms. Thus, there was a need for reliable surface intersection algorithms. This need was also present in the converging area of *solid modeling.*

1.4 Solid Modeling

In the late '60's and early '70's solid modeling was introduced by Voelker and others [2,32,77,87,89,106,107,109] as a more rigorous alternative to surface modeling. A *solid model* is one which contains enough information so that any point (or nearly any, in the presence of rounding arithmetic) can be classified as either inside or outside the solid. Such a model is quite attractive, since it allows one, in principle, to calculate the volumes of solids, their centers of gravity, and to generate volume grids for more sophisticated analysis.

1.4.1 Constructive Solid Geometry

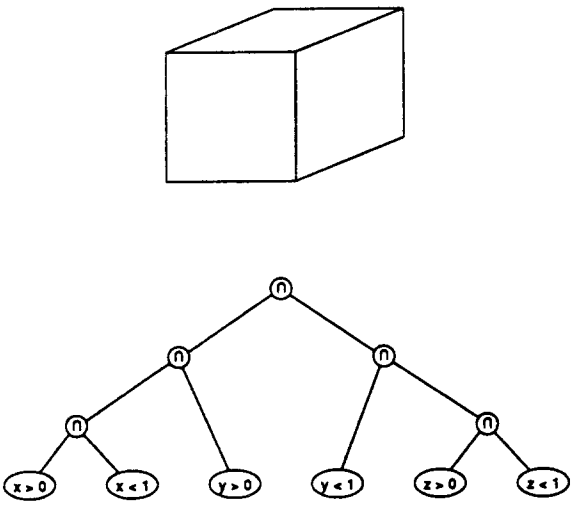


Figure 1.1: CSG representation of a solid.

In the first generation of solid modeling systems, called *Constructive Solid Geometry* (or CSG) modelers, the solid is specified by a Boolean formula whose variables are

Boolean functions on $\mathbf{R}^3$. For example, the box pictured in Figure 1.1 is specified by the conjunction of six halfspaces. For most modeling systems, these functions are not halfspaces but rather the member functions for simple primitives such as boxes, spheres, tori, wedges, and volumes swept out by revolution and translation of planar contours.

A decade later, Voelker, like Plato, lamented that little technical progress had been made in CSG and that CSG systems were not widely used [108]. On the other hand, surface modeling systems such a IBM's CATIA, Volkswagen's SURF, Ford's PDGS, and Mercedes' SIRKO were highly successful. CSG modeling systems failed for two reasons. First, analyses that dealt with the boundary of the object, such as display and surface machining, were not easy to implement. Second, the only surfaces supported were unrealistically limited to planes, quadrics, and tori.

1.4.2 Boundary Representation Solid Modeling

To support analyses that dealt with the boundary and to simultaneously support a larger class of surfaces, an explicit description of the surfaces that bound the volume, was needed. One such representation is the *Boundary Representation* or B-rep [13,14]. A boundary representation is a decomposition of the boundary of a solid according to the dimension of the components of the boundary. There are three underlying geometric structures involved in a B-rep: points, curves and surfaces. The surfaces are usually generated by means external to the solid modeling algorithm. The curves and points are generated as a by product of Boolean operations. The correct computation of these curves is essential to maintaining the B-rep structure through Boolean operations.

In the first B-rep systems, the surfaces allowed where again planes, quadrics and tori. The intersection of these types of surfaces can be obtained by case analysis. Recent B-rep systems, on the other hand, support more general classes of surfaces, specifically piecewise (rational) polynomial surfaces. To perform boolean operations on B-reps containing general surfaces, an algorithm that reliably intersects general surfaces is needed.

Chapter 2

Previous Work

In the following a taxonomy of surface intersection *techniques* is presented. In [85], Pratt gives a taxonomy of surface intersection *algorithms*. It would be desirable to organize all of the research to date which is relevant to surface intersection in a similar way. The limitation to Pratt's organization of this knowledge is that many important parts of it do not constitute complete algorithms, but rather deal with a small part of the whole problem. Also, Pratt's organization gives the false impression that techniques used in one algorithm can't successfully be mixed with techniques present in another. Thus, a taxonomy of surface intersection *techniques* along with some information on which techniques can be combined with others is presented.

First, the surface intersection problem is broken into two sub-problems: *decomposition* and *representation*. Decomposition is the determination of a number of simple curves that make up the intersection of the two surfaces. Representation involves transforming each simple curve into some canonical form from which frequent computations can be performed quickly.

2.1 Decomposition

The decomposition step divides the intersection into smaller surface intersection problems, each of which can be handled by the representation step. At the very least, this step determines the distinct components of the intersection. It may go beyond this and divide components into even simpler curves. It may also generate information regarding the intersection of the components. This will vary from one algorithm to another.

Methods for decomposing the intersection can be divided into two broad categories, *re-approximating* and *direct*. The re-approximating decomposition methods are those which approximate the original surfaces by many simpler surfaces and deal with the intersection of the simpler surfaces. The direct methods take advantage of special knowledge of the surfaces to construct the decomposition of the actual intersection.

2.1.1 Re-approximating Techniques

Re-approximating decomposition is a technique which can be applied easily to any surface whose parametrization is known. In re-approximating decomposition a large number of smaller and simpler surfaces are constructed that approximate the original surfaces. The simpler surfaces are generally triangles but quadric surfaces have been used [3]. (In the following discussion, triangles are discussed but the reader should keep in mind that this may be any simple surface.) Since the approximating sub-surfaces are simpler, direct techniques can be used to intersect them. For instance, triangles can be intersected to form line segments. The resulting line segments will meet end to end and thus can be linked together to for piecewise linear approximations to the intersection curve.

One would like the resulting intersection to be close to the true intersection in the sense that the true intersection curves are a continuous deformation of the intersection of the triangulations. If this is the case, the representation step will have enough information to represent the curves to any precision desired. Unfortunately, the intersection of the approximation by triangles may yield more or fewer curves than are in the true intersection, see Figures 2.1 (adapted from [3]) and 2.3, respectively. If this is the case, the representation step will not be able to represent the curves to the desired accuracy.

To obtain an intersection that is closer (in the sense given above) to that of the true intersection one can use an increasingly larger number of triangles. Algorithms can be further classified according to the manner in which the surfaces are triangulated and according to the algorithm used for deciding which triangles might intersect.

Triangulation Techniques

The most straightforward type of triangulation is a uniform triangulation [36]. This has the disadvantage that the approximation of the surface may be better in some regions of the surface than in others. Specifically, the surface will be better approximated

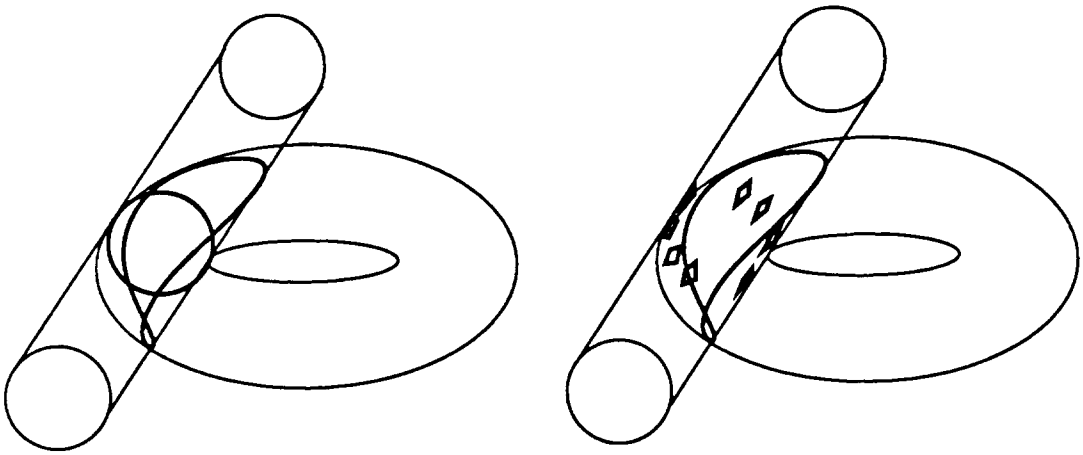


Figure 2.1: Intersection of approximations may yield more curves than actually exist.

in regions where the curvature is small. Thus, techniques have been developed to adapt the triangulation to the surface [10,15,50,82]. In such techniques, triangles are generated in such a way that the original surface is equally well approximated in all regions. This can be carried out as follows: an initial triangulation of the surface is made. Each triangle which does not approximate its portion of the surface to the specified tolerance is subdivided, and the process is repeated. The quality of the approximation is generally measured by measuring the "flatness" of the the approximated surface over the region to be approximated. If the surface is sufficiently flat then the triangle is said to approximate the surface adequately. The process of approximating the surface by smaller triangles in one area and larger triangles in another may create cracks in the approximating triangulation. The cracks may cause the resulting line segments not to meet end to end although they represent a contiguous piece of the intersection curve. Thus, the line segments must be matched using a proximity tolerance.

This flatness criterion is the "Achilles heel" of such methods. Figure 2.2 depicts an intersection problem where the flatness tolerance is very small and considerable work is performed.[1] The intersection of a much coarser triangulation would be close enough (in

[1] In the example pictured, the surface has been subdivided only in the parametric direction in which it is curved. In general, even more subdivision must be done to ensure the sub-surfaces are sufficiently flat.

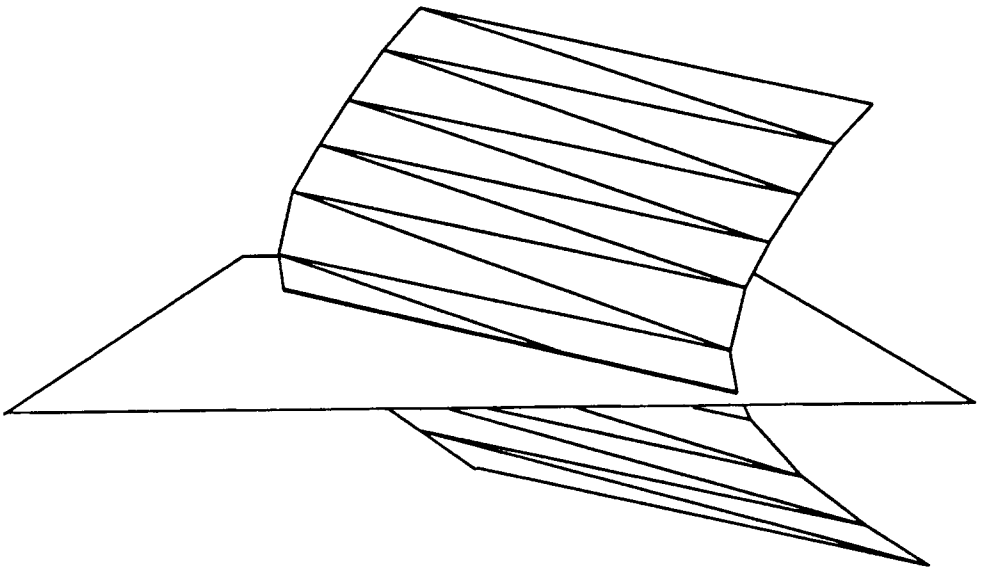


Figure 2.2: Unnecessary subdivision of a surface

the sense given above) to the true intersection. Figure 2.3, on the other hand, depicts a situation where the flatness tolerance is too large and a nearly tangential intersection is missed. The observant reader will notice that the surfaces in Figures 2.2 and 2.3 are simply rotations of one another. Thus, one would expect an flatness based subdivision criterion to subdivide them to the same level of refinement.

Thus, one is forced to make a tradeoff between speed and correctness. To obtain an algorithm that is as correct as possible given the limitations of floating point arithmetic, one must subdivide the surfaces until they are on the order of the size of the machine rounding. To obtain an algorithm that runs in a reasonable amount of time one must risk missing intersection curves.

The fundamental problem with this approach is that the flatness of a pair of surfaces is independent of how they intersect. Figures 2.4 and 2.5 depict two intersection problems each consisting of a bicubic patch intersecting a plane. The intersections curves in 2.4 are identical to those in Figure 2.5. The bicubic patch in Figure 2.4 is highly curved. The bicubic patch in Figure 2.5 is relatively flat. Furthermore, the bicubic patch in Figure 2.5 could be made arbitrarily flat and the intersection would remain unchanged. The fact that two surfaces being intersected are nearly flat provides no information about how they intersect. Consequently, any subdivision algorithm whose termination criterion is based on

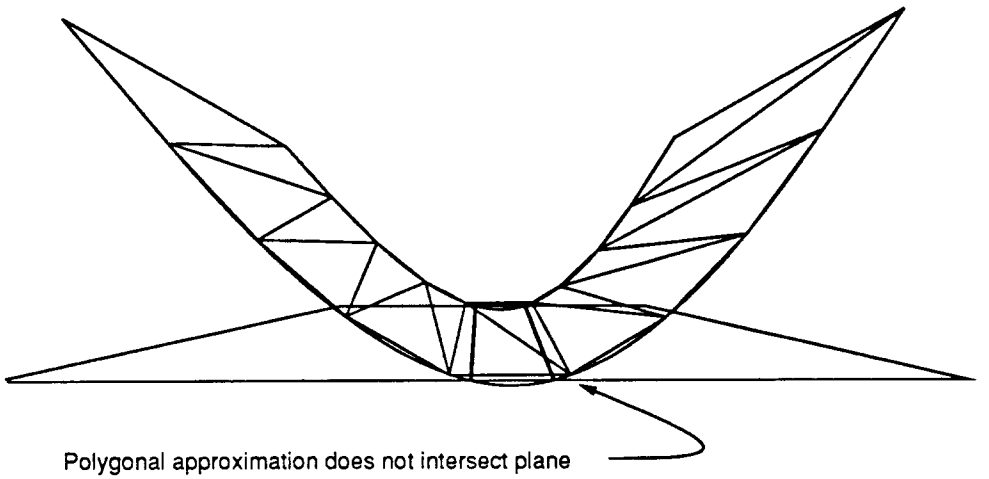


Figure 2.3: Insufficient subdivision of a surface.

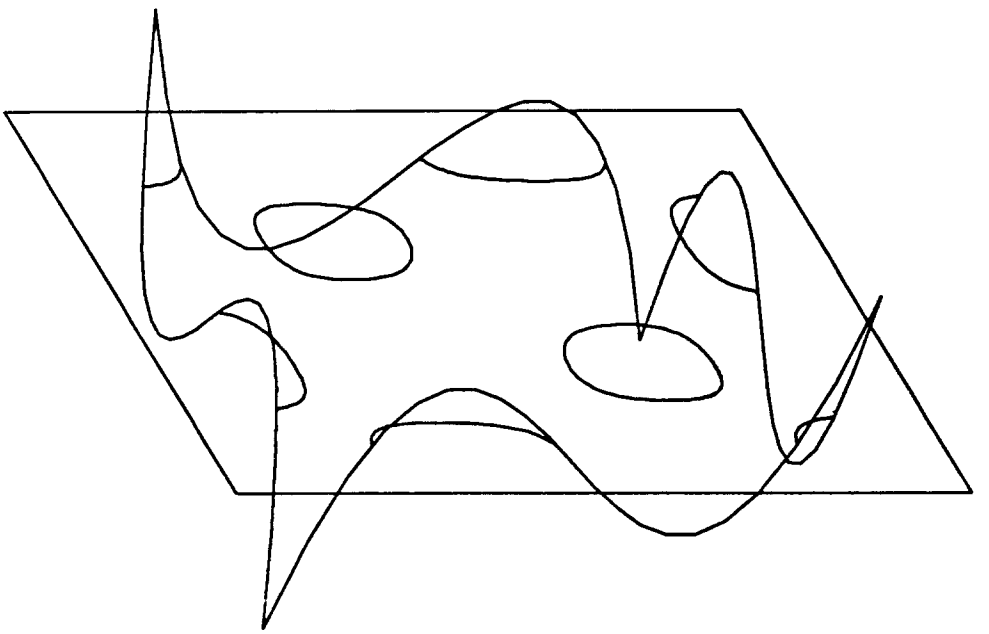


Figure 2.4: An intersection consisting of eight components

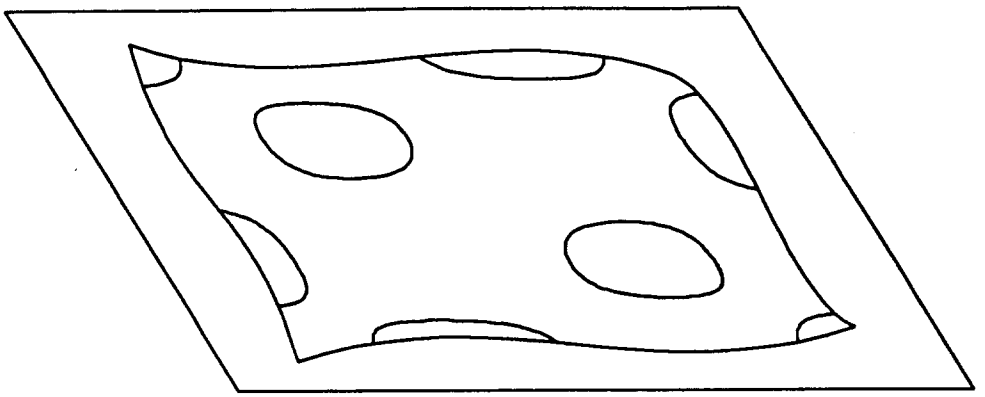


Figure 2.5: An intersection also consisting of eight components.

flatness is doomed to fail on this example. If the bicubic patch is made flatter than the algorithm's tolerance then the algorithm will miss the two internal loops.

Triangle Intersection Techniques

Amongst re-approximating algorithms there remains the problem of finding the sub-surfaces which potentially intersect. In the simplest schemes, each triangle of one surface is compared against each triangle of the other surface. This results in a running time of $O(nm)$ where m and n are the number of triangles in each surface.

This can be sped up considerably by using a sweep plane algorithm or by placing one set of triangles in a spatial subdivision. This will resulting in an $O((m+n)log(m+n)+k)$ algorithm where k is the number of intersecting pairs of triangles.

Recognizing that a great number of triangles do not participate in the intersection, the approximation can be constructed in a lazy or recursive fashion. First a coarse grid is generated on each surface. If two triangles intersect (or, in some methods, if the bounds on the surfaces underlying triangles intersect) then each triangle is replaced by more triangles which approximate the surface more closely. The process is repeated until all triangles which intersect other triangles meet the subdivision criterion [6,11,15,61,78,82]. Natarajan [78] has shown that for "randomized input" this results in a running time of $O(k)$,where k is, again, the number of intersecting pairs of triangles.

2.1.2 Direct Decomposition

Direct decomposition methods remedy the problems of the re-approximating approach by using specific information about the types of surfaces being intersected. These algorithms can be divided into *Quadric Surface*, *Critical Point*, and *Loop Detection* decomposition algorithms. The quadric surface decomposition algorithms use human analysis of the small number of possible intersection types between two quadric surfaces. In the critical point methods, a discrete set of points is determined from which the intersection curves can be recovered. In the loop detection algorithms, the two surfaces are subdivided into pairs that intersect in simple curves.

Quadric Decomposition

For the intersection of quadrics, exhaustive case analysis can be performed [31,62, 72,73,74,83,84,92,101,103]. These methods have the advantage that they can be tailored to be as fast and as stable as is possible. Since these types of intersections arise frequently in geometric calculations, there can be no argument against employing them. Their only disadvantage is that they only cover a limited range of intersection problems.

Critical Point Decomposition

For the critical point methods, a discrete set of points is determined from which the intersection curves can be recovered. For instance, one could obtain a single point on each curve. From this point, the representation step could trace out the intersection curve to the desired accuracy. Of critical point algorithms, Algebraic Decomposition has received the majority of the research attention.

Algebraic Decomposition

If both surfaces are algebraic, the intersection curve can be mapped to a planar algebraic curve. This curve can then be decomposed using a cylindrical algebraic decomposition [4,26]. This decomposition finds a point on each curve and points at the crossing of curve components.

This simplest case in which this can be applied is when one surface is a parametric rational polynomial and the other is an implicit polynomial surface. In that case the para-

metric equations are substituted into the implicit equations resulting in a single equation in two unknowns.

If both surfaces are in parametric form, one surface can be implicitized and the above procedure employed. [7,93].

If both surfaces are in implicit form, one has two equations in three unknowns. Resultants can be used to eliminate one variable again reducing the problem to a single equation in two unknowns.

These techniques may have the following problems:

Stability

The transformation to an implicit equation in two variables via substitution and perhaps implicitization/elimination of variables is not necessarily numerically stable; that is, small changes in the data that define the surface may cause large changes in the implicit form of the algebraic curve. Thus, exact arithmetic, or at least very careful analysis is needed. The amount of analysis necessary to show that implicitization of a *planar cubic curve* is stable is quite large [38]. No such analysis has been carried out with respect to the implicitization of surfaces.

Computation Time and Space

Additionally, the transformation to a planar curve will raise the degree of the equations defining the curve. For example, when intersecting two bicubic patches the curve equation $\mathbf{F}(s,t) - \mathbf{G}(u,v) = 0$ will be of total degree 6, while the corresponding planar algebraic curve $f(s,t) = 0$ will be of degree 108. Because of the high degree the amount of data needed to describe the algebraic curve and hence the amount of computation that needs to be performed on the data are very large.

Extraneous Components and Singularities

Finally, the transformation may introduce new curves and singularities. This is caused by two processes. Firstly, the surface represented by the implicit equation of a parametric surface will be a superset of the original surface. New parts of the surface may contribute new curves and singularities that did not exist in the original problem. Secondly, when variables are eliminated, the mapping may take different points in $\mathbf{R}^3$ to the same point in $\mathbf{R}^2$ resulting in singularities that did not exist in the original curve.

General Critical Point Decomposition

Confronted with these difficulties and with the restriction that the surfaces be algebraic, some researchers have tried to create algorithms that construct a critical point decomposition of the intersection curves without creating the planar algebraic curve. These methods can be used for any surface whose parametrization is known. For instance, Cheng [18] and Kriesis [58,81] propose to consider the function $\phi(u, v)$. Given two surfaces $\mathbf{F}(u, v)$ and $\mathbf{G}(s, t)$, ϕ is defined to be the minimum distance between the point $\mathbf{F}(u, v)$ and the surface $\mathbf{G}$. From a mathematical standpoint one can characterize a set of critical points in terms of ϕ. For instance, Kriezis proposed to find all of the points

$$\phi_u = 0. \tag{2.1}$$

From an algorithmic standpoint, however, there are two problems. The first is to evaluate the function ϕ and the second is to compute bounds on the partial ϕ_u. Both of these must be done to ensure the discovery of all of the roots of equation 2.1. No algorithms have been proposed to accomplish this. Rather ϕ is computed for polyhedral approximations to $\mathbf{F}$ and $\mathbf{G}$. Ensuring that the ϕ computed from the polyhedral approximations to $\mathbf{F}$ and $\mathbf{G}$ adequately resembles the true ϕ is precisely the problem of ensuring that the intersection of the polyhedral approximations adequately resembles the true intersection. Until more is known about bounds on ϕ for specific surface types, these approaches must be classified as re-approximation techniques.

Loop Detection Decomposition

In contrast to the critical point methods, the loop detection methods attempt to find a decomposition of the two surfaces into pairs of subsurfaces so that the intersection of each pair is either empty or isomorphic to the unit interval. Loop detection methods can be used with any surface for which the following are true

1. One can compute bounds on the position, derivative, and normals of the surface.
2. One can subdivide the surface into smaller surfaces.
3. One can reliably intersect the edges of such surfaces with other instances of such surfaces.

B-spline, Bézier and implicit surfaces including quadrics and tori are included in this group.

At present, loop detection techniques comprise a handful of criteria that can be applied to determine if it is possible for two patches to intersect in a loop [22,24,28,47,48,65, 75,94,95,97,102]. There is no guarantee that the subdivision of the surface will eventually produce pairs of intersecting patches all of which pass the criterion. The subject of loop detection will be taken up again in Chapter 3.

2.2 Representation

The output of a surface intersection algorithm will be some representation of the intersection curve. For some motion planning problems it is sufficient to know when there is an intersection. For display, a set of line segments with no connectivity information is all that is required. For numerically control machining, a set of points with a fixed coarse tolerance is needed. For solid modeling, a high precision, compact, and efficient representation is required. One can see that representation techniques are driven by widely differing needs.

2.2.1 Polynomial Representations

While almost all quadric/quadric surface intersections can be decomposed by case analysis, only those that result in conic sections can be represented as rational polynomial curves [91,104]. The intersection of higher degree surfaces will almost always have a genus too high to be parametrized [55]. Thus, other representations must be used.

2.2.2 Approximation

For most applications the technique of generating a large number of points along the intersection curve and then approximating the point set is appropriate. These tracing schemes rely on the decomposition step for enough information to delimit and aid the tracing. For instance, most tracing methods require a point on the intersection curve from which to "march". If there are curve crossings on the component to be traced, the tracing method should be appraised of this.

Curve Tracing

Curve tracing (or marching) generates a sequence of points on the intersection curve. If the two surfaces are quadric special methods are applicable, otherwise general marching techniques must be employed.

Quadric Surface Tracing

If the two surfaces are quadrics the following technique can be employed [62]. Let $f(\mathbf{x}) = 0$ and $g(\mathbf{x}) = 0$ be the two quadric surfaces. For some choice of a and b, $h(\mathbf{x}) = af(\mathbf{x}) + bg(\mathbf{x}) = 0$ is a *ruled* quadric surface. In that case h can be parametrized by $H(u,v) = \mathbf{p}(u) + \mathbf{q}(u)v$ where $\mathbf{p}(u)$ is a rational quadratic polynomial. Points on the intersection can be obtained by evaluating $\mathbf{p}$ and $\mathbf{q}$ at a value u_0 and then finding the solutions to the quadratic equation $f(\mathbf{p}(u_0) + \mathbf{q}(u_0)v) = 0$.

General Surface Tracing

In the case that the surfaces are more general than quadrics, a marching method is employed [90]. These methods can be used with either parametric or implicit surfaces. Marching methods assume that an initial point on the intersection has been found. From the initial point successive points are found. The information from the last point found is used to compute the next point. One has some choice over the dimension in which the tracing is taking place and also over the degree of the curve approximation used to generate successive points.

Dimension

In general one will have a system of equations

$$f_i(x_1, ..., x_d) = 0 \qquad i = 1, ..., d-1 \tag{2.2}$$

where the d is the *dimension* of the space in which the curve is being traced and the f_i are polynomials. (Some papers present algorithms where the f_i are functions such as "the distance to the closest point on the surface" [18]. Such functions are clearly not polynomial. Manocha [66,67], on the other hand, proposes that some of the f_i be the determinants of large matrices of polynomials in the x_i. While not stored in polynomial form, such a function is a polynomial). From a point $\mathbf{x} \in \mathbf{R}^d$ one wishes to iterate to a point on the curve given

by equation 2.2. This is typically accomplished via something resembling Newton iteration [90].

For example, when tracing the intersection of two parametric patches one might trace the intersection in the space of the four parameters of the surfaces (two for each surface). The system of equations would be

$$\mathbf{F}(s,t) - \mathbf{G}(u,v) = 0 \tag{2.3}$$

where $\mathbf{F}(s,t)$ and $\mathbf{G}(u,v)$ are the parametric forms of the two surfaces.

If one were tracing the intersection of two implicit surfaces then the tracing would occur in three dimensions and the system of equations would be

$$f(\mathbf{x}) = 0 \tag{2.4}$$

$$g(\mathbf{x}) = 0 \tag{2.5}$$

where f and g are the implicit forms of the two surfaces.

If one were tracing the intersection of an implicit surface f and a parametric surface $\mathbf{G}$ one would only need to use the two parameters of the parametric surface [8]. The system of equations in that case would be

$$f(\mathbf{G}(u,v)) = 0. \tag{2.6}$$

One often has a choice regarding the dimension of the space in which to trace the curve. For instance the intersection of a pair of parametric surfaces can be traced in $\mathbf{R}^4$ or, via elimination of variables, can also be traced in $\mathbf{R}^2$. In the first case one has more lower degree equations and in the second one has fewer higher degree equations. Researchers have observed that if one has the choice of tracing the same curve in either a higher of lower dimensional space it is faster and numerically more stable to trace curves that are the intersection of many low degree surfaces in a higher dimensional space than to trace the intersection of higher degree surfaces in a lower dimensional space [39].

Degree of Approximation

To generate successive points, a parametric approximation to the intersection curve is constructed. This approximation is then evaluated a small distance from the previous point and the evaluation is used as a starting point for some iterative technique that will

converge to the intersection curve. This iteration step accounts for a large amount of the computation in the tracing step. A starting iterate that is closer to the intersection curve can speed this up considerably. Most commonly, a straight line is used [10,11] but higher degree polynomials and transcendental functions have also been used [5,16,17,41,46]

Three problems must be addressed when tracing a curve: *jumping components*, *singularities* and *backtracking* .

Singularities

If two curve components intersect then the equations that describe the intersection locally become underconstrained. That is, the Jacobian of equation 2.2 is rank deficient by two rather than one. At such points a unique step direction might not exist, and if one does exist, even the smallest step in that direction may place the next point closer to the wrong curve branch than the right one. Techniques which deal with this problem generally use higher order terms in the Taylor expansion of Equation 2.2 [8,9,40,41,69,79].

Jumping components

Jumping components occurs when the tracing algorithm takes a step that is too large and proceeds on a different component of the curve, as depicted in Figure 2.6.

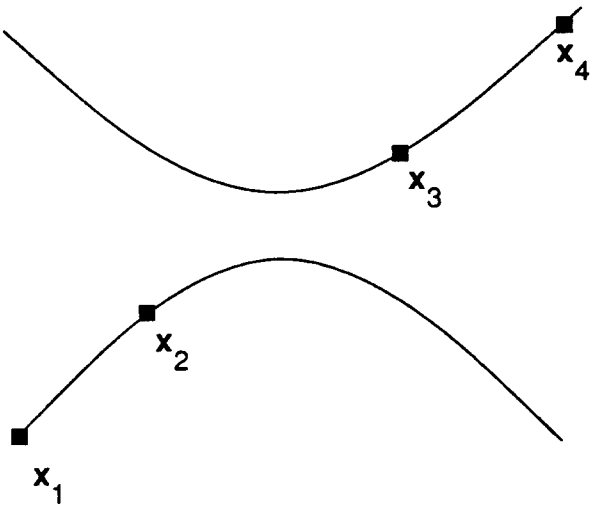


Figure 2.6: Jumping Components.

Backtracking

Backtracking occurs when the tracing algorithm generates points out of order as depicted in Figure 2.7. If the curve is represented as a planar algebraic curve, the points can be sorted [53] in order to determine if backtracking has occurred. If that is not the case, there are numerical techniques that can help to prevent backtracking and jumping components [8,9,17,19,27,46,76] but they are not guarantees. These methods work by estimating a safe step size based on the derivatives of the intersection curve at a single point. Unfortunately, all methods based on a finite number derivatives of the surface at a finite number of points can fail for arbitrary surfaces. Suppose that the algorithm considered the derivatives $\mathbf{c}'(t), \ldots, \mathbf{c}^{(d)}(t)$ of the intersection curve. If $|\mathbf{c}^{(d+1)}(t)|$ were large compared with $|\mathbf{c}'(t)|, \ldots, |\mathbf{c}^{(d)}(t)|$ then the estimated safe step size would be too large and problems would occur.

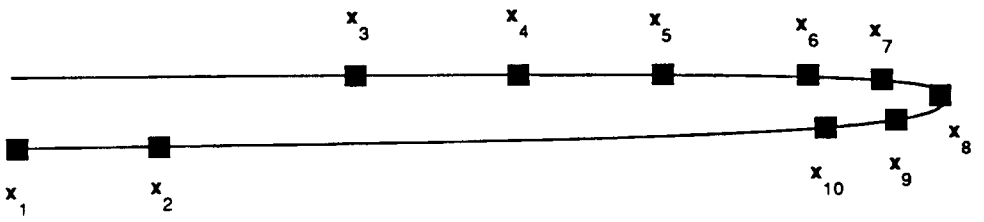


Figure 2.7: Backtracking.

Representation

After a large number of points have been generated on the intersection curve, one can approximate the point set with a parametric function such as a spline, see [20].

In the context of Boundary Representation Solid Modeling the intersection of two surfaces is used to trim the faces bounding a volume. This intersection curve must be very accurate in order to prevent inconsistent models from being created. Balanced against this is the need for speed. In order to get around this problem some modeling systems create a rough approximation to the intersection curve both in $\mathbf{R}^3$ and in the parameter space of the surfaces. This curve is not used for critical calculations. Rather, when a point on the curve is desired, the rough intersection curve is evaluated and the resulting point is "relaxed" onto the true intersection. This method will be discussed in more detail in Chapter 5.

Summary

In this chapter, a taxonomy of surface intersection techniques has been presented. In the next chapter, *loop detection* techniques are explored more fully. Loop detection techniques not only guarantee the discovery of all curve components, but also help in the curve tracing step.

Chapter 3

Loop Detection

3.1 Loop Detection

Having observed some of the problems associated with re-approximation and algebraic decomposition techniques *loop detection* techniques are considered. A *loop*, in this context, is an intersection curve that does not intersect the boundaries of either of the surfaces patches being intersected, as depicted in Figure 3.1. Consider a surface intersection algorithm that subdivides the surfaces until some stopping criterion is satisfied and then intersects the edges of one sub-surface with the other sub-surface and vise-versa. If there are loops in any of the sub-patch pairs as depicted in Figure 3.1, they will go undetected. If there is none, as depicted in Figure 3.2, then all intersection curves will be discovered.

3.1.1 Gauss Maps

All loop detection criteria are based on bounds on the *Gauss maps* of the surfaces being intersected. The Gauss map takes a point $\mathbf{x}$ on the surface in $\mathbf{R}^3$ and maps it to the corresponding surface normal vector $\mathbf{n}(\mathbf{x})$ as depicted in Figure 3.3. The space containing the unit normal vectors is referred to as the *Gaussian sphere.* The *image* of the Gauss map for a surface patch is a subset of the Gaussian sphere. For most loop detection criteria, only the subset of the Gaussian sphere as a point set is important, the function which maps points into the image is not. Thus, the term Gauss map is used to refer to the image and not the function.

Note also that at any point on a surface there are really two normals, each being

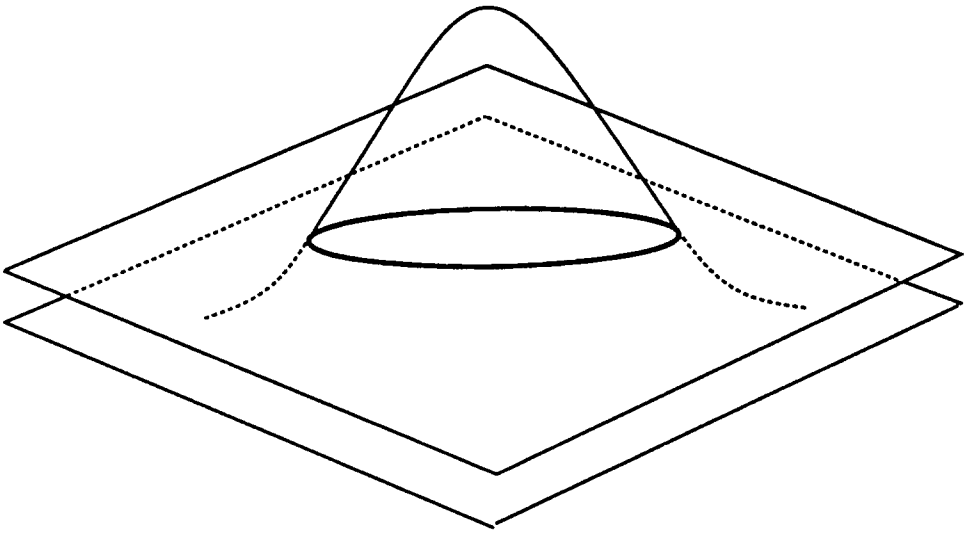


Figure 3.1: If there are loops they may go undetected.

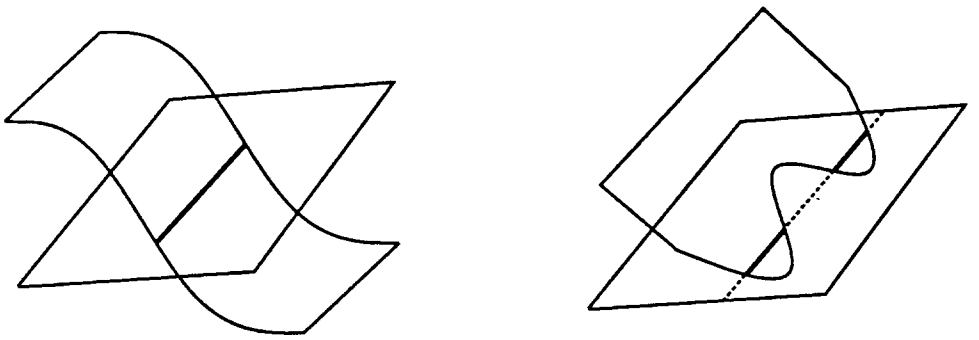


Figure 3.2: If there are no loops then only edge intersections need to be checked.

equal to the other multiplied by -1. By convention, the Gauss map consists only of one of these normals for each point. A pair of points on the Gaussian sphere that are related to one another my the factor -1 are called *antipodal*. Two sets are called antipodal if there is a point in one which is antipodal to a point in the other.

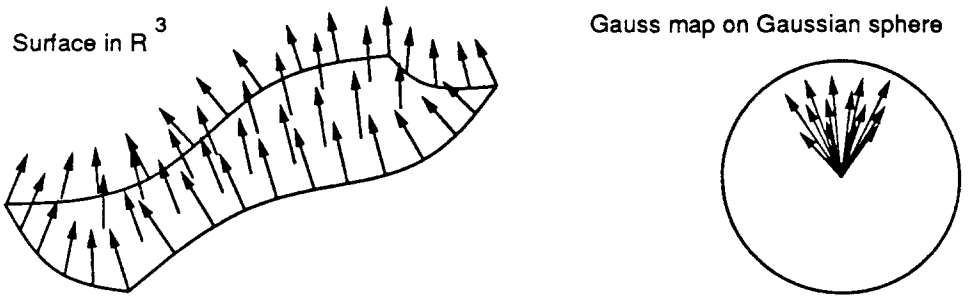


Figure 3.3: The Gauss map of a surface.

3.1.2 Sinha's Criterion

The first published results dealing with loop detection appear in [102]. In that work, Sinha establishes that if the Gauss maps of two surfaces do not intersect and are not antipodal then the surfaces cannot intersect in any loops. Formally:

Theorem 1 (Sinha) *Let S_1 and S_2 be two smooth surface patches in $\mathbf{R}^3$. Let N_1 and N_2 be the Gauss maps of S_1 and S_2 respectively. Let W_1 and W_2 be circular cones such that $N_1 \subset W_1$ and $N_2 \subset W_2$. If W_1 does not intersect W_2 then the surfaces S_1 and S_2 do not intersect in a loop.*

The geometry associated with this theorem is depicted in Figure 3.4. The cones W_1 and W_2 are referred to as *normal cones*. Note that the normal cones are double ended so that the Gauss maps are not allowed to be antipodal.

While this is a valuable tool with which one can construct a robust surface intersection algorithm, it is not quite powerful enough. Consider the example depicted in Figure 3.5. Sections of two cylinders are being intersected. The cylinders' axes form an angle a with one another. The sections are b degrees of each cylinder. The Gauss maps of each surface are parallel lines on the Gaussian sphere of length a separated by a distance b. The cones that bound the Gauss maps will intersect as long as $b \geq a$, preventing them from

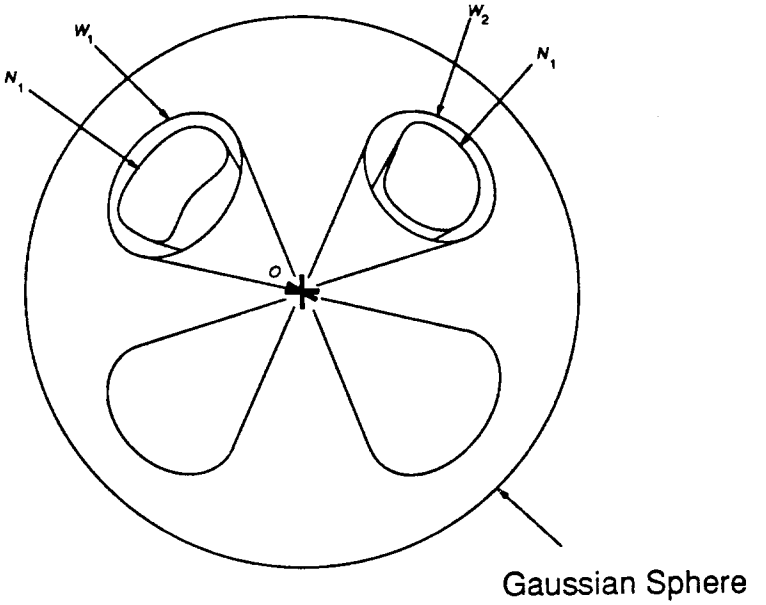


Figure 3.4: Sinha's Theorem.

passing the loop detection criterion. Thus, the cylinders will have to be subdivided until $b < a$. This can result in arbitrarily long running times. Although constructed as a difficult example for this criterion, similar examples arise naturally in many surface intersection problems.

A second theorem in [102] establishes a stronger result but with a stricter hypothesis. A surface is *diffeomorphic* to the unit disc if there is a continuous map that takes the surface into the unit disc. Intuitively, this means that the surface does not have any holes in its domain.

Theorem 2 (Sinha) *Suppose that S_1 and S_2 are two surfaces smoothly embedded in $\mathbf{R}^3$ with $S_1 \cap S_2 = \partial S_1 = \partial S_1$. Suppose S_1 and S_2 are both diffeomorphic to the unit disc, and are transverse to each other along their common boundary. Then there are points $\mathbf{x}_1 \in S_1$ and $\mathbf{x}_2 \in S_2$ so that $\mathbf{n}_1(\mathbf{x}_1)$ is parallel to $\mathbf{n}_2(\mathbf{x}_2)$.*

Sinha then states that if one could show that two surfaces' normals were nowhere parallel, and the surfaces were diffeomorphic to the unit disc, then there could be no loops in the intersection. This is a strong theoretical result but it lacked the details necessary to implement it. For instance, the paper does not explain how one could compute the bounds on the Gauss maps of the surfaces. Nor does it explain if these bounds would be

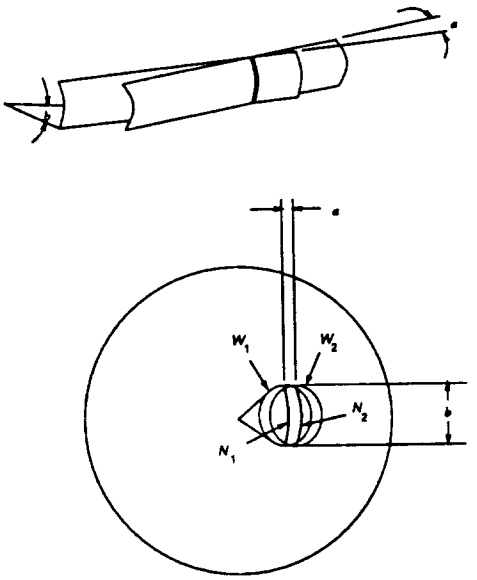


Figure 3.5: An example for which Sinha's Theorem does not work well.

represented as general polygons, convex polygons or by something else. To take advantage of the theorem one would want to represent the bounds at least as general polygons. In that case, one would need a method to determine it if the polygons intersected. Subsequent authors supplied some of the missing details.

3.1.3 Sederberg's Criterion

In [95], Sederberg describes the following method for detecting loops in surface intersections. Let $\mathbf{S}(s,t)$ be a parametric Bézier surface. A bounding cone, called the s-cone is created that bounds the vector function $\mathbf{S}_s(s,t)$ where s and t are allowed to vary over the domain of the surface. An analogous cone, called the t-cone, is created that bounds $\mathbf{S}_t(s,t)$. The normal to the surface at a generic point $\mathbf{S}(s,t)$ is in the direction of the cross product of the partial derivatives, $\mathbf{S}_s(s,t) \times \mathbf{S}_t(s,t)$. Thus, a loose bound on the Gauss map can be obtained by computing the smallest cone that contains every cross product $\mathbf{s} \times \mathbf{t}$ such that $\mathbf{s}$ is contained in the t-cone and $\mathbf{t}$ is contained in the t-cone. This is the cone whose axis is the cross product of axes of the s and t cones and whose half angle is given by

$$d = \sin^{-1}\left(\frac{\sqrt{\sin^2 a + 2\sin a \cos b \sin c + \sin^2 c}}{\sin b}\right) \tag{3.1}$$

where a is the half angle of the t-cone and c is the half angle of the s-cone and b is the angle between the axis of the s and t-cones as shown in Figure 3.6.

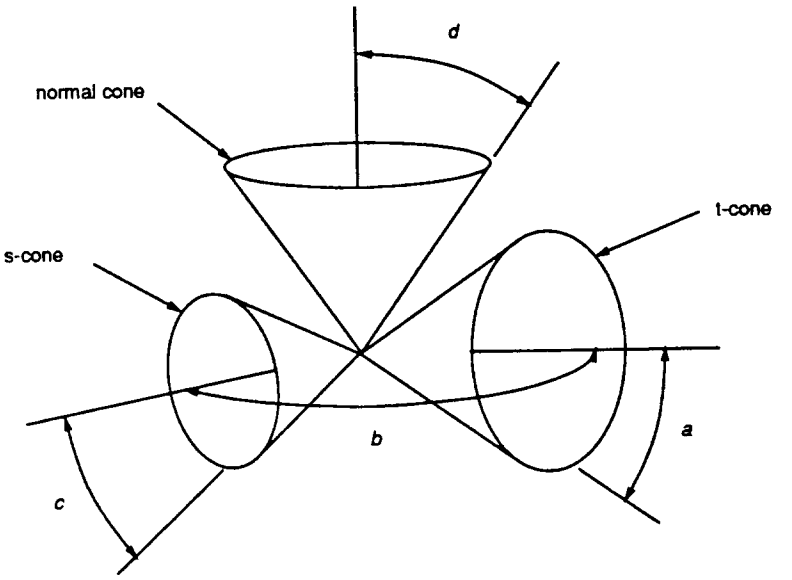


Figure 3.6: Calculation of a bound on the Gauss map.

Sederberg defines a *tangent plane cone* as follows: The tangent plane cone is a cone which if its vertex is translated to any point on the surface, the intersection of the surface and the cone will be just the vertex of the cone. A tangent cone can be constructed from Sederberg's normal cone by using the same axis and a half angle of 90° minus the half angle of the normal cone.

It is important to note that, in general, a normal cone cannot be used in this way to construct a tangent cone. Consider Figure 3.7. The surface depicted is like a parking garage ramp. At every point on the surface, the normal to the surface is nearly in the direction $\hat{z}$, and can be made arbitrarily close to $\hat{z}$. Thus, the axis of the normal cone is $\hat{z}$ and it has a very small half angle. Construct the cone with the same axis as the normal cone and with half angle given by 90° minus the normal cone half angle. For the point **p** at the overlap, this cone contains **q** which is on the same patch.

The particular normal cone constructed using Sederberg's method can be used to construct a tangent cone. Sederberg proposes the following test: If the tangent cone of one surface contains one of either of the s or t cones of the other surface then the intersection cannot contain any loop. While a good tool for constructing a surface intersection algorithm,

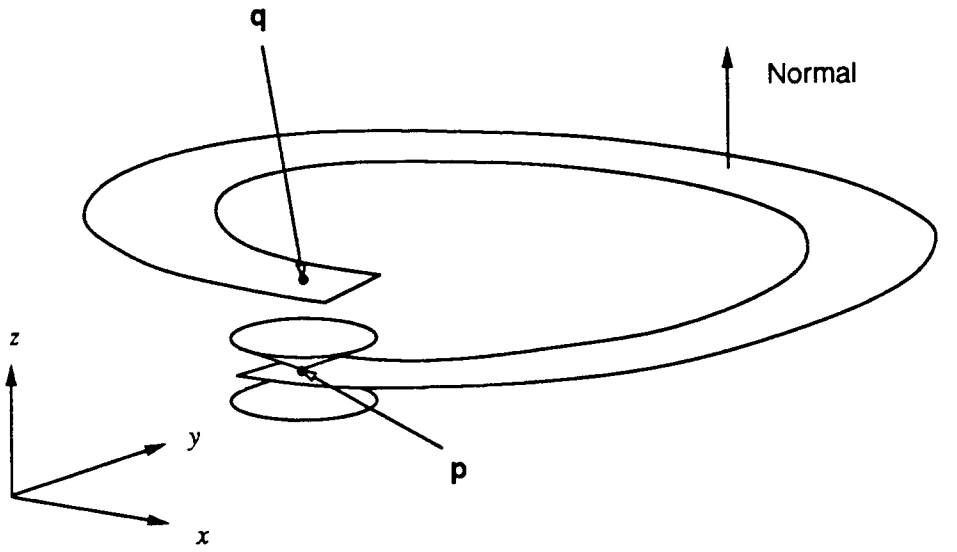


Figure 3.7: An example for which tangent cones do not work.

it would be unable to establish the non-presence of loops for simple examples such as shown in Figure 3.5.

Prognostications

After having read this thesis, Sederberg will publish a paper [97] which significantly improves the ideas presented both here and in [95]. The loop detection criterion presented there will solve the problem of detecting loops in the example shown in Figure 3.5 and will bound functions of significantly lower degree than those proposed in this thesis.

Bounds on the Gauss Map

The first paper [95] is also important because it describes algorithms for computing normal cones for Bézier surfaces. To construct bounds for Bézier surfaces, Sederberg constructs the vector functions $\mathbf{S}_s(s,t)$ and $\mathbf{S}_t(s,t)$ as Bézier surfaces. The control points of these surfaces can be used to bound them. Next, he describes an algorithm to compute the s or t cone given the set of n vectors bounding the functions $\mathbf{S}_s(s,t)$ and $\mathbf{S}_t(s,t)$. In short, the algorithm is as follows:

Bounding Cone($\mathbf{v}_1, \ldots, \mathbf{v}_n$ **)**

Let C be the cone with axis $\mathbf{v}_1$ *and half angle* $0°$.

for $i = 2$ *to* n

Let C be the smallest cone containing C and $\mathbf{v}_i$

endfor

return *(C)*

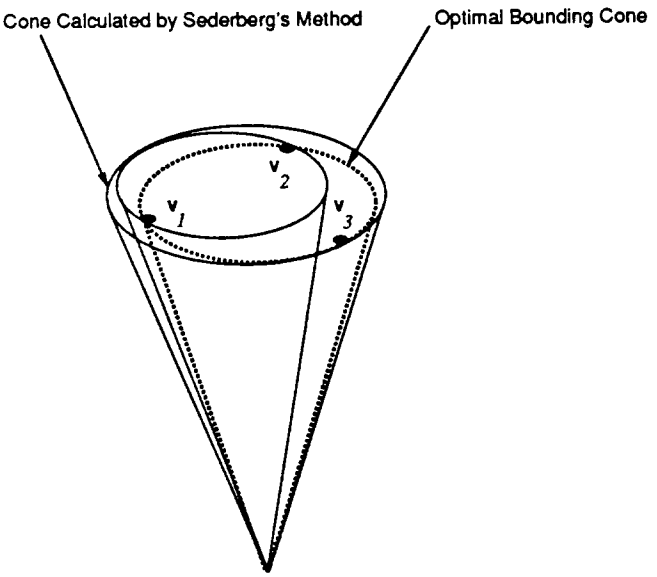


Figure 3.8: Sederberg's enclosing cone algorithm.

This algorithm returns a cone that contains all the vectors but may be larger than the smallest cone containing all the vectors, as depicted in Figure 3.8. As has been shown, the larger cone can cause the loop detection algorithm to be unable to determine that there are no loops.

An algorithm to compute the smallest enclosing cone is a straightforward application of linear programming with a small number of variables and n constraints. Sederberg's satisfaction with a loose bound when the tight bound is available is perhaps due to the wide-spread belief that linear programming is prohibitively slow. As will be seen in Section 4.1, this is not the case.

Collineal Points

Sederberg published a subsequent paper dealing with loop detection [94]. This paper established the following theorem:

Theorem 3 (Sederberg) *If two nonsingular surface patches each isomorphic to the unit disc, S_1 and S_2, intersect in a closed loop, then there exists a line that is perpendicular to both S_1 and S_2 if the following conditions are met:*

1. *The dot product of any two normal vectors (on the same patch or on different patches) is never zero. This means that the total range of normal directions* for both patches considered simultaneously *cannot deviate more than* 90°.
2. *S_1 and S_2 are everywhere tangent continuous.*

A line is said to be perpendicular to a pair of surfaces if the line contains points $\mathbf{p}_1$ on S_1 and $\mathbf{p}_2$ on S_2 so that $\mathbf{n}_1(\mathbf{p}_1)$, $\mathbf{n}_2(\mathbf{p}_2)$ and $\mathbf{p}_1 - \mathbf{p}_2$ are all parallel. The theorem can be rephrased approximately as follows: If there are no lines perpendicular to $\mathbf{S}_1$ and $\mathbf{S}_2$ then $\mathbf{S}_1$ and $\mathbf{S}_2$ do not intersect in a loop.

This theorem presents two problems. The first is to find all lines perpendicular to both surfaces and subdivide the surfaces so that no two intersecting sub-patches contain such a line. Sederberg proposed to guarantee the such lines by using interval arithmetic. The bounds returned by interval arithmetic methods can be quite loose, and Sederberg did not report on the specifics of the interval method.

The second problem is to ensure that the surfaces are isomorphic to the unit disc. This is true for parametric surfaces whose domain is isomorphic to the unit disc, but may not be true for implicit surfaces as shown by the example in Figure 3.9. The cylinder shown has no normal which is collineal to the normal to the plane, and yet the intersection is a loop. Thus, for implicit surfaces, one is faced with the additional task of determining the surface topology before applying the criterion.

3.1.4 de Montaudouin's Criterion

De Montaudouin [75] made the following observation. Let the normals to two surfaces S_1 and S_2 be contained in disjoint cones C_1 and C_2 respectively. Let the axes to C_1 and C_2 be $\mathbf{a}_1$ and $\mathbf{a}_2$, respectively. Then the tangent to the intersection curve always

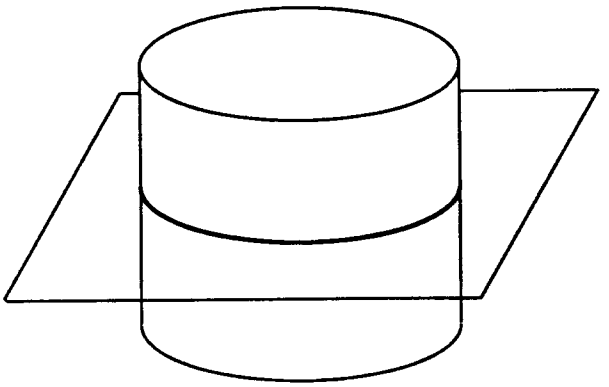


Figure 3.9: A cylindrical surface is not isomorphic to the unit disc.

makes a positive inner product with the vector $\mathbf{a}_1 \times \mathbf{a}_2$. He shows that this precludes the existence of any loops in the intersection.

De Montaudouin's and Sinha's results are identical, but their proofs are different. Notable is the calculation of the vector $\mathbf{t} = \mathbf{a}_1 \times \mathbf{a}_2$. Although de Montaudouin did not point it out, this immensely simplifies the tracing of the curve, as will be seen in Section 3.4. However, the problem of speed still persists. For the example pictured in Figure 3.5, a large amount of subdivision is required before the cylinders pass the loop detection criterion.

3.1.5 Kriezis' Technique

Kriezis [58,59,60] suggests some minor changes to the method of Sederberg. Rather than computing bounding cones, he computes bounding pyramids. He still computes the bounding pyramids about the partial derivatives and then computes the cross product of the two pyramids so obtained. To ensure that they fit tightly about the Gauss maps of the surfaces, Kriezis aligns pyramids with the surfaces. This is perhaps an improvement, but it will be seen that one can compute the tightest possible bounds with only slightly more cost.

3.1.6 Dokken's Observations

Dokken [24,23] devised a surface intersection algorithm based on the following observations:

- If the tangent box of a curve and the normal box of a surface point in the same direction (i.e., all their possible scalar products have the same sign and are different from zero) and the partial derivative cones in both

the u and v directions have an opening of less than $\pi/2$, then only one intersection point is possible.

- If the normal boxes of two surfaces have no overlap, then there can be no closed loops.
- If the normal boxes of two surfaces have no overlap and there exists only two intersection points between the boundaries of the surfaces with the other surface, then only one intersection curve is possible.

In these statements, a normal box and a tangent box are bounds on the normals of a surface and the tangents of a curve. Dokken does not provide any details on the use of these objects. While correct, these statements provide coarse tests. The overlap of the boxes depends on the orientation of the geometry.

3.2 The Loop Detection Criterion

Other loop detection criteria having been reviewed, one which performs uniformly better is presented. The loop detection theorem relies on the following simple geometric observation.

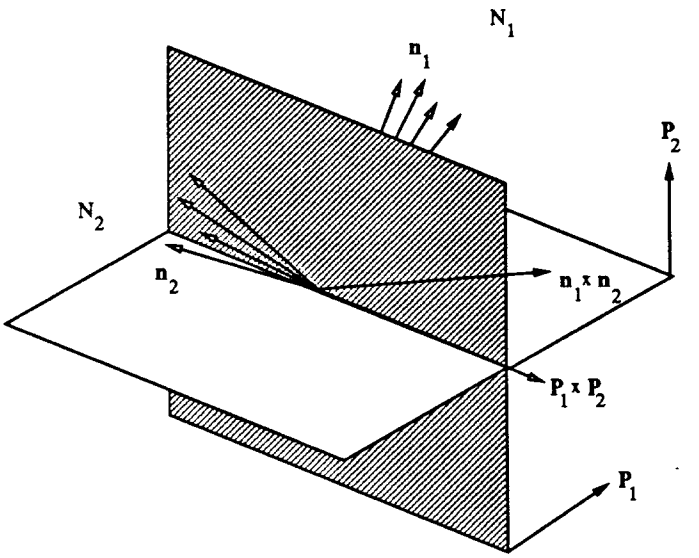


Figure 3.10: Geometry of the loop detection criterion.

Lemma 1 *Let N_1 and N_2 be two sets of vectors and let $\mathbf{P}_1$ and $\mathbf{P}_2$ be two vectors satisfying*

$$\mathbf{P}_1 \cdot \mathbf{n}_1 > 0, \quad \mathbf{P}_1 \cdot \mathbf{n}_2 < 0 \tag{3.2}$$

$$\mathbf{P}_2 \cdot \mathbf{n}_1 > 0, \quad \mathbf{P}_2 \cdot \mathbf{n}_2 > 0 \tag{3.3}$$

for all $\mathbf{n}_1 \in N_1$ *and* $\mathbf{n}_2 \in N_2$, *as in Figure 3.10. Choose any pair of vectors* $\mathbf{n}_1 \in N_1$ *and* $\mathbf{n}_2 \in N_2$. *Then the cross product* $\mathbf{T}$ *given by*

$$\mathbf{T} = \mathbf{n}_1 \times \mathbf{n}_2 \tag{3.4}$$

satisfies

$$\mathbf{T} \cdot (\mathbf{P}_1 \times \mathbf{P}_2) > 0. \tag{3.5}$$

Intuitively this can be interpreted as follows. Let N_1 and N_2 be two sets of vectors and let P_1 and P_2 be planes containing the origin. Let the normal to P_1 be $\mathbf{P}_1$ and the normal to P_2 be $\mathbf{P}_2$. If N_1 and N_2 are on opposite sides of P_1 and on the same side of P_2 then the cross product of any vector in N_1 with any vector in N_2 will make a positive inner product with the vector $\mathbf{P}_1 \times \mathbf{P}_2$.

Proof:

$$\begin{aligned}
(\mathbf{P}_1 \times \mathbf{P}_2) \cdot (\mathbf{n}_1 \times \mathbf{n}_2) &= ((\mathbf{P}_1 \times \mathbf{P}_2) \times \mathbf{n}_1) \cdot n_2 && (3.6)\\
&= -(\mathbf{n}_1 \times (\mathbf{P}_1 \times \mathbf{P}_2)) \cdot \mathbf{n}_2 && (3.7)\\
&= (-(\mathbf{n}_1 \cdot \mathbf{P}_2)\mathbf{P}_1 + (\mathbf{n}_1 \cdot \mathbf{P}_1)\mathbf{P}_2) \cdot \mathbf{n}_2 && (3.8)\\
&= -(\mathbf{n}_1 \cdot \mathbf{P}_2)(\mathbf{P}_1 \cdot \mathbf{n}_2) + (\mathbf{n}_1 \cdot \mathbf{P}_1)(\mathbf{P}_2 \cdot \mathbf{n}_2) && (3.9)
\end{aligned}$$

Thus

$$(\mathbf{P}_1 \times \mathbf{P}_2) \cdot (\mathbf{n}_1 \times \mathbf{n}_2) > 0. \tag{3.10}$$

■

Using this one can easily proceed to a result about surface intersections.

Theorem 4 (Loop Detection Theorem) *Let* S_1 *and* S_2 *be two* C^1 *surfaces whose normals are contained in sets* N_1 *and* N_2, *respectively. If there exist vectors* $\mathbf{P}_1$ *and* $\mathbf{P}_2$ *such that*

$$\mathbf{P}_1 \cdot \mathbf{n}_1 > 0, \quad \mathbf{P}_1 \cdot \mathbf{n}_2 < 0 \tag{3.11}$$

$$\mathbf{P}_2 \cdot \mathbf{n}_1 > 0, \quad \mathbf{P}_2 \cdot \mathbf{n}_2 > 0 \tag{3.12}$$

for all $\mathbf{n}_1 \in N_1$ *and* $\mathbf{n}_2 \in N_2$, *as in Figure 3.10, then the intersection of the two surfaces is a curve, a point, or a set of curves and points. Furthermore, all isolated point intersections are at the boundaries of the surface patches, the curves do not contain singularities, and no intersection curve forms a loop.*

Proof: If the surfaces intersected in a two-dimensional manifold, (i.e., a surface) then at each point of intersection the normals $\mathbf{n}_1(\mathbf{x})$ of S_1 and $\mathbf{n}_2(\mathbf{x})$ of S_2 would satisfy either $\mathbf{n}_1(\mathbf{x}) = \mathbf{n}_2(\mathbf{x})$ or $\mathbf{n}_1(\mathbf{x}) = -\mathbf{n}_2(\mathbf{x})$, in violation of equations (3.11) and (3.12); thus, this cannot happen.

If $\mathbf{p}$ is a point of intersection and is interior to both patches, then the normals to the two surfaces at $\mathbf{p}$ are not collinear by equations (3.11) and (3.12). In a neighborhood of $\mathbf{p}$ the surfaces are very nearly a pair of planes intersecting in a curve that is very nearly a line perpendicular to the normals to both surfaces at $\mathbf{p}$. Thus $\mathbf{p}$ is not an isolated point, but rather lies on a curve.

Let each intersection curve be oriented so that the tangent to the curve at a point $\mathbf{x}$ is:

$$\mathbf{T}(\mathbf{x}) = \frac{\mathbf{n}_1(\mathbf{x}) \times \mathbf{n}_2(\mathbf{x})}{|\mathbf{n}_1(\mathbf{x}) \times \mathbf{n}_2(\mathbf{x})|} \tag{3.13}$$

where $\mathbf{n}_i(\mathbf{x})$ is the normal to surface S_i at the point $\mathbf{x}$. Note that the denominator in equation (3.13) cannot be zero since equations (3.11) and (3.12) preclude the normals from being parallel or anti-parallel. By Lemma 1,

$$\mathbf{T}(x) \cdot \mathbf{P} > 0. \tag{3.14}$$

where $\mathbf{P}$ is given by

$$\mathbf{P} = \mathbf{P}_1 \times \mathbf{P}_2. \tag{3.15}$$

Suppose that there were a closed loop in the intersection of the two surfaces, as depicted in Figure 3.11. The fact that the normals to the surfaces are never collinear precludes any singularities in the intersection curve. Thus, the curve tangent $\mathbf{T}$ is well-defined (up to a choice of orientation). If one were to parametrize the loop by arc length, s, from an arbitrary point on the loop then

$$\int_{s=0}^{s=l} \mathbf{T}(s)ds = \mathbf{0} \tag{3.16}$$

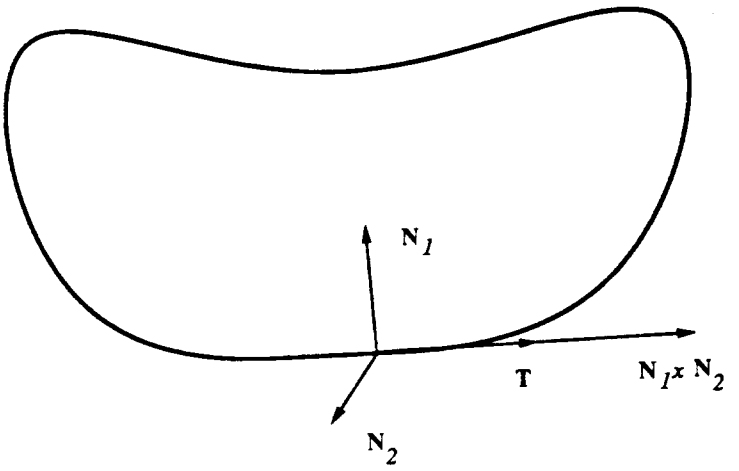


Figure 3.11: An hypothetical intersection loop.

where l is the length of the curve. Taking the inner product with $\mathbf{P}$ yields:

$$\int_{s=0}^{s=l} \mathbf{T}(s) \cdot \mathbf{P} ds = 0. \tag{3.17}$$

The tangent $\mathbf{T}(s)$ is a scalar multiple of the cross product of the normals to the surface. This scalar varies continuously and is never zero. Thus, the sign of this scalar is constant (positive by convention) throughout the loop. Equations (3.17) and (3.14) are clearly inconsistent, thus there can be no loop. ∎

Not only does this allow us to guarantee the absence of loops, it allows us to order points on the intersection curve. Two points $\mathbf{x}_1$ and $\mathbf{x}_2$, can be ordered by ordering the scalar values $\mathbf{x}_1 \cdot \mathbf{P}$ and $\mathbf{x}_2 \cdot \mathbf{P}$. Finally, if there are only two intersections of the boundaries of the patches with the patches then there can be only one intersection curve and the intersection points are its endpoints. In this case, all points are guaranteed to lie on the same component.

3.3 Comparison with Previous Methods

In the case that one has only convex bounds on the Gauss maps of the surfaces in question, this is the strongest possible result. It is reasonable to assume that more general bounds are not available: the known and useful bounds on B-spline and Bézier surfaces and their normals are convex. The theorem states that if the bounds on the Gauss maps are separable by a plane then there can be no loops. Let us investigate the contrapositive. If the bounds on the Gauss maps are not separable by a plane then by Farkas' Lemma [34] they must intersect. If they bounds intersect there exist surfaces with the same bounds which intersect in loops. To see this, consider a vector in the intersection of the bounds. A pair of coincident planes having this vector for a normal intersect in a loop.

Note that the loop detection criterion presented here performs uniformly better than the other loop detection criteria presented. That is: if cones containing the bounds on the Gauss map of the surface do not intersect then clearly there is a separating plane between the bounds on the Gauss maps.

Note also that in the example given in Figure 3.5, the loop detection criterion presented here does arbitrarily better than all the previously presented loop detection criteria. For each of those criteria, the cylinders must be subdivided arbitrarily small to pass the loop detection criteria of Sederberg, Sinha, de Montaudouin or Kriezis, whereas the cylinders will pass the loop detection criterion presented here for any angular separation.

The loop detection criterion does not make any assumption about the form of the surfaces. They may be parametric, implicit or some other unforeseen type [12]. It requires only that the surface is smooth and that bounds on the Gauss map are available. In fact, parametric surfaces are allowed to contain holes in their domains. This is in contrast to Sederberg's criterion [94] which requires that the interior of the loop be diffeomorphic to the unit disc.

3.4 The Tracing Algorithm

If one has subdivided a pair of parametric patches $\mathbf{F}(s,t)$ and $\mathbf{G}(u,v)$ so that they pass the loop detection criterion presented in the last section, obtained the vector $\mathbf{P}$ given by equation (3.15), and calculated the two points $\mathbf{p}_0$ and $\mathbf{p}_1$ where the edges of the surfaces intersect each other, then a simple tracing algorithm is as follows:

Trace

$w_0 = \mathbf{P} \cdot \mathbf{p}_0$

$w_1 = \mathbf{P} \cdot \mathbf{p}_1$

for $w = w_0$ *to* w_1 *stepping by* δw

solve the system of equations

$$\mathbf{F}(s,t) = \mathbf{G}(u,v)$$

and

$$\mathbf{F}(s,t) \cdot \mathbf{P} = w$$

using Newton iteration and the previous solution as a starting iterate

endfor

Since there are only two points of intersection of the curve with the surface edges there can only be one intersection curve. Since the curve tangent always makes a positive inner product with $\mathbf{P}$, it can only intersect a plane perpendicular to $\mathbf{P}$ once. Since the curve must proceed from $\mathbf{p}_0$ to $\mathbf{p}_1$, it must pass through an intermediate plane. Thus, the system of equations has exactly one solution in the region specified by the subdivision of $\mathbf{F}$ and $\mathbf{G}$. No other checks need to be made. Experience has shown that the Newton iteration generally converges. If it does not, one can simply reduce δw until it does. There is no possibility of jumping components since there is only one curve component in the range. There is no possibility of generating points out of order since the series of planes automatically orders them. Thus the value δw does *not* need to be chosen carefully.

3.5 The Intersection Algorithm

3.5.1 Loop Detection

In this section, the use of the loop detection criterion in a surface intersection algorithm is described. The input to the algorithm will be two surfaces. It is assumed that one is able to calculate spatial bounds on the surfaces and bounds on their Gauss maps. It is further assumed that it is possible to subdivide the surfaces and that as one subdivides the surfaces the spatial bounds approach the surface and the bounds on the Gauss map approach the Gauss map.

Intersect *(SurfaceA, SurfaceB)*

if *the bounding volumes of SurfaceA and SurfaceB intersect* **then**

if *the Gauss maps satisfy the loop detection criterion* **then**

Intersect Simple Surfaces *(SurfaceA, SurfaceB)*

else if *the surfaces are within* ϵ *of a point* **p then**

and their Gauss maps contained within ϵ *of a normal* **n**

then

report that a singular point has been found.

else

Subdivide each surface.

Intersect *all pairs.*

endif

endif

Some of the points of this algorithm are explained more thoroughly in Chapter 4: methods for computing bounds on the Gauss maps of common surfaces are presented in Section 4.4; methods for determining if these bounds satisfy the loop detection criterion are presented in Section 4.4.4; methods for determining if the spatial bounds on two surfaces intersect are presented in Section 4.2.

Theorem 5 *Algorithm* **Intersect** *will either decompose the intersection problem into sub-problems containing no loops or will discover a singularity.*

Proof: Clearly the algorithm can only return when it has subdivided the surfaces into sub-surfaces such that each pair passes the loop detection criterion, or when it has found a singularity. The only other option then is that the algorithm recurses indefinitely. This cannot happen for the following reason. The parameter space subdivisions of the surface form a nested set of regions. Consider the limit points (s,t) and (u,v) of these regions. Let $\mathbf{n}_1$ be the unit normal at $\mathbf{F}(s,t)$ and let $\mathbf{n}_2$ be the normal at $\mathbf{G}(u,v)$. If $\mathbf{n}_1 \times \mathbf{n}_2 \neq \mathbf{0}$ then eventually the surfaces must pass the loop detection criterion, terminating the recursion. If $\mathbf{F}(s,t) \neq \mathbf{G}(u,v)$ then eventually the surfaces must pass the $\mathbf{R}^3$ separability test. Otherwise $\mathbf{n}_1 \times \mathbf{n}_2 = 0$ and $\mathbf{F}(s,t) = \mathbf{G}(u,v)$, i.e. the limit is a singular point. The singular point will

eventually be found by subdivision. ■

3.5.2 Identifying Individual Curves

When a pair of surfaces pass the loop detection criterion the structure of the intersection curve can often be inferred from the intersection of the edges of one surface and vise-versa. This involves first intersecting the edges of one surface with the second surface and then intersecting the edges of the second surface with the first. Methods to do this are presented in Section 4.5. One can compute the tangent direction of the curve at the intersection point. One can then decompose this tangent into the parameter space of the surface. Methods for doing this are presented in Section 4.6. With this information one can compute whether the curve is entering or exiting a surface at the point. Based on this one can classify the edge/surface intersection points:

entering A point is *entering* if its classification with respect to the surfaces is one of:

1. entering SurfaceA; interior SurfaceB
2. interior SurfaceA; entering SurfaceB
3. entering SurfaceA; entering SurfaceB

exiting A point is *exiting* if its classification with respect to the surfaces is one of:

1. exiting SurfaceA; interior SurfaceB
2. interior SurfaceA; exiting SurfaceB
3. exiting SurfaceA; exiting SurfaceB

isolated A point is *isolated* if its classification with respect to the surfaces is one of:

1. entering SurfaceA; exiting SurfaceB
2. exiting SurfaceA; entering SurfaceB

Based on the number and type of edge-intersection points the intersection of the surfaces can be determined.

one edge-intersection point

entering: error

exiting: error

isolated: one point

two edge-intersection points

entering and exiting: one curve

entering and isolated: error

exiting and isolated: error

exiting and exiting: error

entering and entering: error

isolated and isolated: two points

more than two edge-intersection points: unknown

Some of these cases are depicted in Figure 3.12.

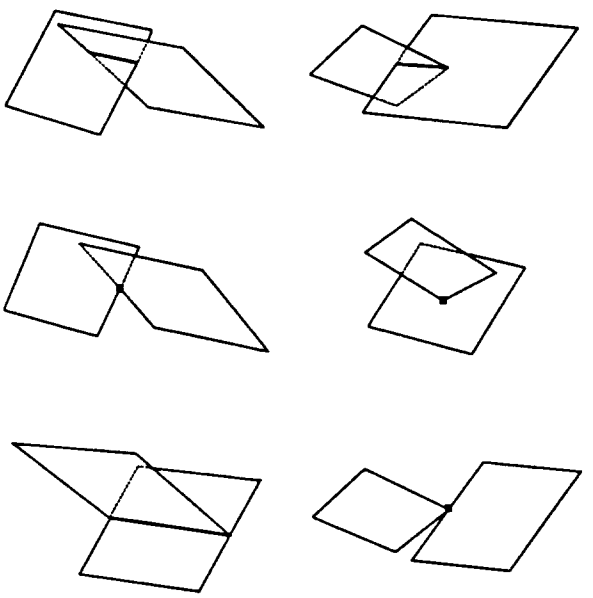


Figure 3.12: Some possible intersection types.

If the number of edge/surface intersection points is greater than two, there are many possibilities. For instance, if there are four points, two entering and two exiting, it

is not known how to find the pairs that form endpoints of curves. In practice this can be handled simply by subdividing the surfaces until there are two or fewer points. This leads us to an algorithm to intersect two simple surfaces:

```
Intersect Simple Surfaces (SurfaceA, SurfaceB)
    Intersect the edges of SurfaceA with SurfaceB
        the edges of SurfaceB with SurfaceA.
    if there are degenerate intersections then
        return report of degeneracies
    else if the intersections indicate an known case then
        Trace the intersection curve
    else
        subdivide SurfaceA and SurfaceB
        for each childA, of SurfaceA
            for each childB, of SurfaceB
                Intersect Simple Surfaces (childA, childB)
            end for
        end for
    end if
```

3.6 Examples

While no number of examples can prove that an algorithm runs quickly or reliably, the absence of examples is a sure indication that it doesn't. In the following, the algorithm is demonstrated on a pathological example and on several manufacturing examples.

3.6.1 A Complex Example

To illustrate the power of this method the two bicubic patches depicted in figures 3.13 and 3.14 have been intersected. The first patch has been constructed by folding it over on itself three times in one direction and then three times in the other. The resulting patches is nine layers deep in most places. The second patch has been constructed by raising and lowering alternate control vertices in a checkerboard manner. This results in a surface with

eight spikes. The two surfaces intersect in seventy-one intersection curves as can be seen most clearly in 3.15. A large proportion of these curves are interior loops. This example takes about four minutes to run on a Silicon Graphics 12 MHz Personal Iris.

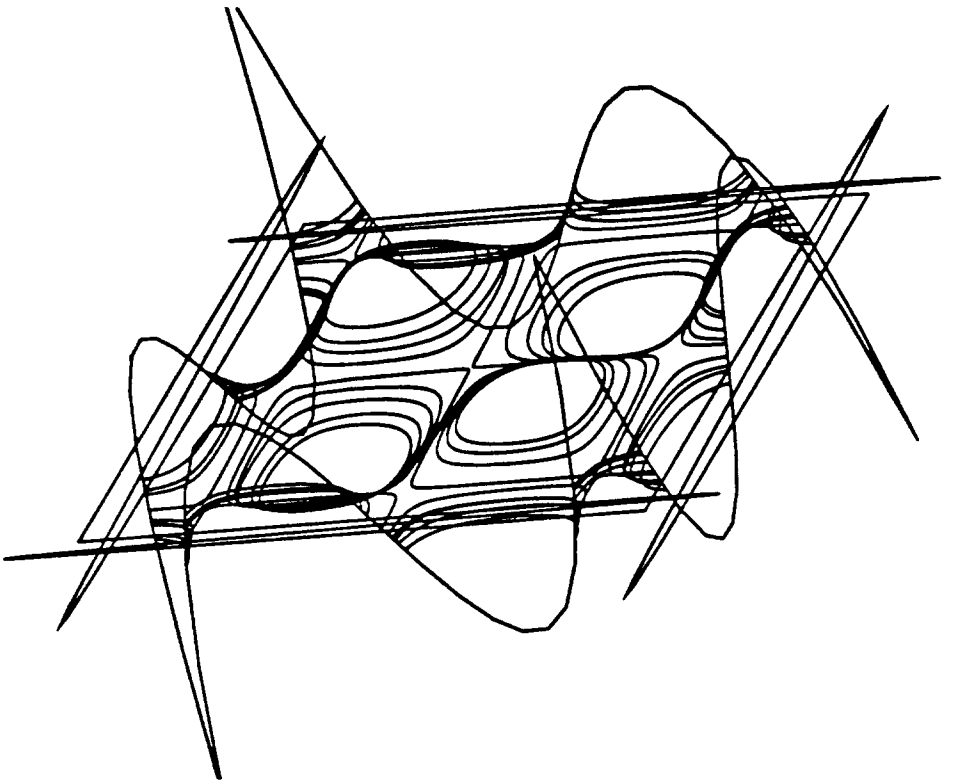


Figure 3.13: Two bicubic patches.

3.6.2 Some Manufacturing Examples

Some surface data describing parts of a various commercial aircraft was used to demonstrate the algorithm's utility on actual manufacturing data.[1] Figure 3.16 depicts the intersection of an engine nascelle with the engine pylon. Figure 3.17 depicts the intersection of the same pylon with the aircraft wing. In Figure 3.18 the intersection of the wing and the fuselage is shown. Finally, Figure 3.19 shows the intersection of the vertical and horizontal stabilizers. In each example, each structure is represented by six to ten surfaces with the surface degrees as high as 16×16.

[1] These data were supplied by a company that wishes to remain anonymous.

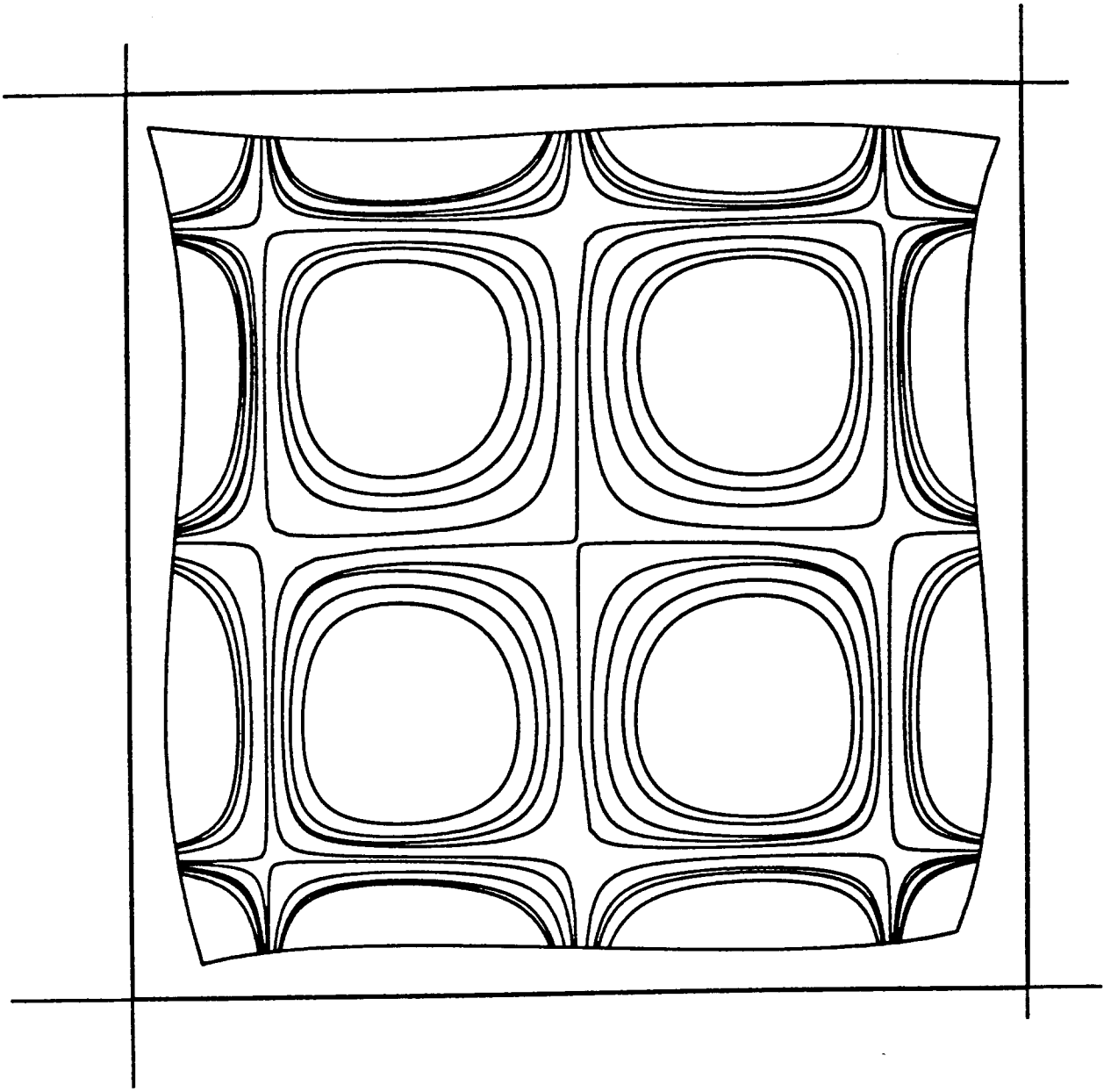


Figure 3.14: Same problem viewed from above.

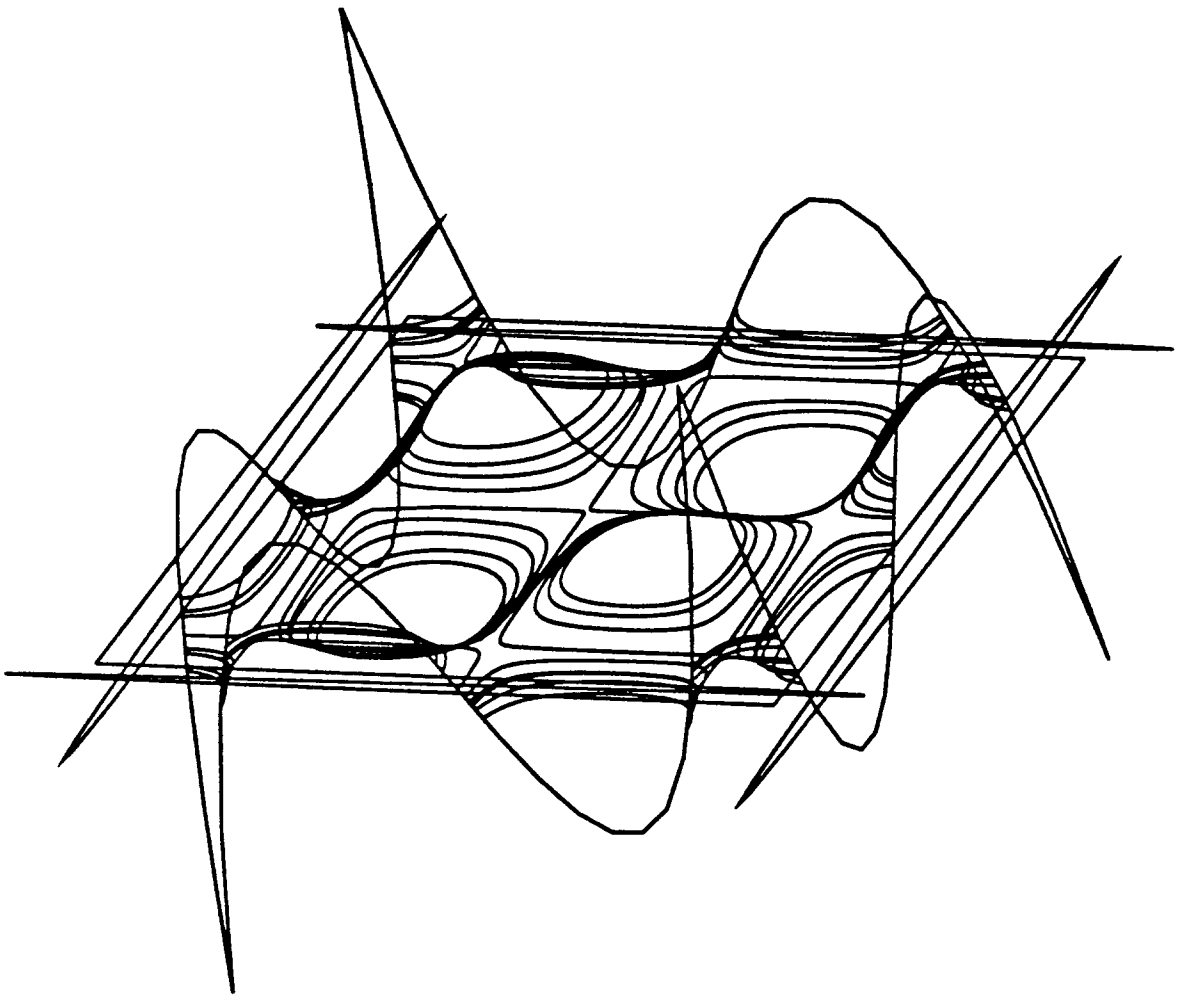


Figure 3.15: Intersection curves of two bicubic patches.

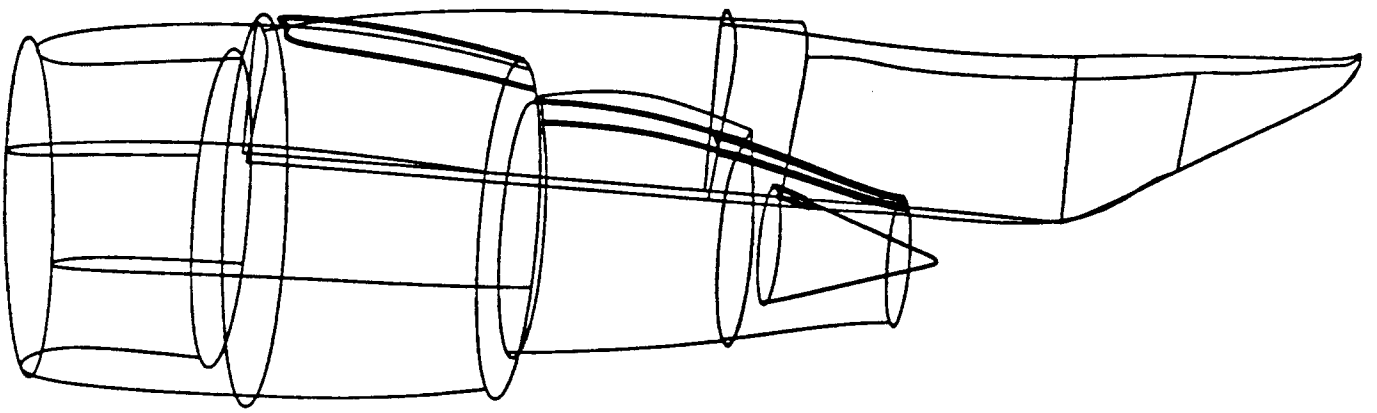


Figure 3.16: The intersection of an engine nacelle and pylon.

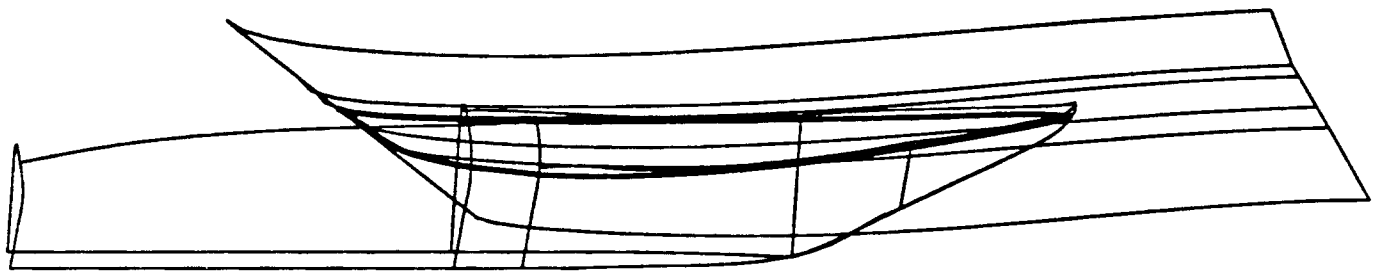


Figure 3.17: The intersection of an engine pylon and aircraft wing.

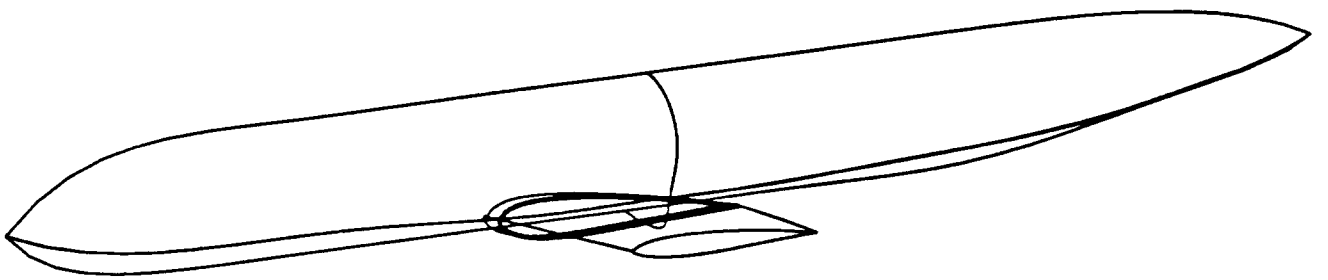


Figure 3.18: The intersection of an aircraft wing and fuselage.

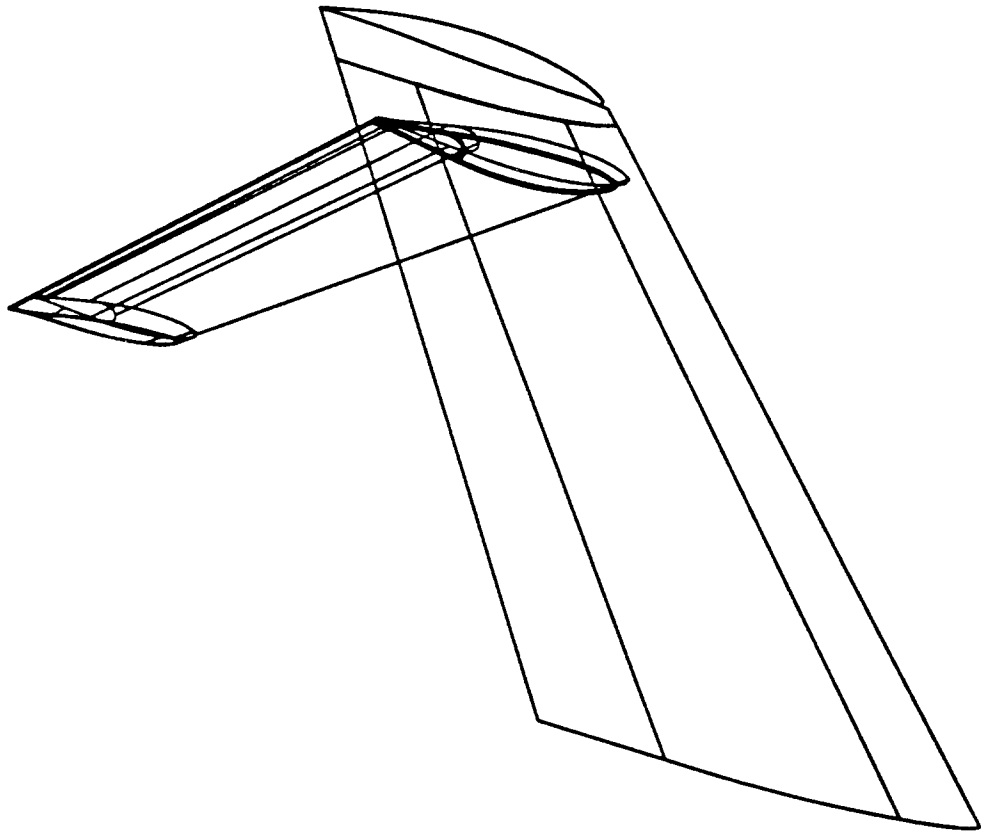


Figure 3.19: The intersection of vertical and horizontal stabilizers.

Chapter 4

Supporting Algorithms

In the previous chapter, a method for intersecting a pair of surfaces has been outlined. In the algorithm described, it is necessary to carry out other computations such as intersecting a curve with a surface, determining if it is *not* possible for two surfaces to intersect, determining bounds on Gauss maps of surfaces, and determining if two Gauss maps intersect. In all three of these problems, linear programming is a very powerful tool. In the first section of this chapter, linear programming is described in general, and a specific linear programming algorithm is described in detail. This is followed by applications to 3-dimensional separability, and two variations of separability on the sphere. Equipped with those tools, the problem of intersecting a curve and surface is addressed. The chapter finishes with discussions of problems associated with the surface intersection problem.

4.1 Linear Programming

An unexpectedly large number of geometric problems can be possed as linear programming problems in low dimension. These include:

1. Determining the smallest circle on a sphere that encloses a set of points.
2. Determining the smallest distance on the sphere between the convex hulls of two point sets.
3. Determining the closest distance between the convex hulls of two point sets in $\mathbf{R}^3$.

A linear programming problem is one that can be stated as follows: given a set of constraints $\mathbf{a}_i \in \mathbf{R}^{d+1}$, $i = 0 \ldots n-1$, and an objective function $\mathbf{c} \in \mathbf{R}^d$, find a solution

$\mathbf{x} \in \mathbf{R}^d$, $\mathbf{x} = (x_0, \ldots, x_{d-1})$, satisfying

$$\sum_{j=0}^{d-1} a_{ij}x_j + a_{id} \geq 0 \tag{4.1}$$

for $i = 0, \ldots, n-1$, that minimizes

$$\sum_{j=0}^{d-1} c_j x_j. \tag{4.2}$$

The set of points satisfying 4.1 is called the *feasible region.* Expression 4.2 is called the *objective* function. A point $\mathbf{x}$ minimizing the objective function in the feasible region is called the *optimum* point. Each row $\mathbf{a}_i$ of a_{ij} is called a *constraint.*

Dot product notation simplifies the expressions considerably. Let

$$\mathbf{a}_i \cdot \mathbf{x} \equiv \sum_{j=0}^{d-1} a_{ij}x_j + a_{id} \tag{4.3}$$

and let

$$\mathbf{c} \cdot \mathbf{x} \equiv \sum_{j=0}^{d-1} c_j x_j \tag{4.4}$$

Notice that the meaning of "$\cdot$" depends on whether it is being used with a constraint or with the objective function (in Equation 4.3, the vector $\mathbf{a}_i$ has $d+1$ components while the vector $\mathbf{x}$ has only d).

4.1.1 Bounded Linear Programming

In [99] a randomized algorithm running in expected time $O(nd!)$ is given for solving linear programming problems when the constraints include d constraints of the form

$$c_i x_i > m_i \tag{4.5}$$

for $i = 0, \ldots, d-1$. The algorithm proceeds by placing the constraints in random order with the provision that the first d constraints are those given in equation (4.5). The algorithm constructs the optimum incrementally. Let the feasible region with respect to the constraints $\mathbf{a}_0, \ldots, \mathbf{a}_{i-1}$ be denoted $\mathbf{F}_i$. At each step, the algorithm maintains a *provisional* optimum $\mathbf{x}_i$. The provisional optimum $\mathbf{x}_i$ is an optimal point in $\mathbf{F}_i$. The first provisional optimum, $\mathbf{x}_d$, is simply $(m_0/c_0, \ldots, m_{d-1}/c_{d-1})$. If any of the $c_i = 0$ then m_i/c_i is replaced by 0. Subsequent provisional optimal points are found incrementally. Suppose one has computed

$\mathbf{x}_i$. If $\mathbf{a}_i \cdot \mathbf{x}_i \geq 0$ then clearly $\mathbf{x}_{i+1} = \mathbf{x}_i$. Otherwise, there is an optimal point $\mathbf{x}_{i+1}$ that satisfies $\mathbf{a}_i \cdot \mathbf{x}_{i+1} = 0$.

One can show this as follows. Consider a straight line connecting $\mathbf{x}_i$ with an optimal point $\mathbf{x}'$ in $\mathbf{F}_{i+1}$. Since $\mathbf{F}_i$ is convex and both $\mathbf{x}_i$ and $\mathbf{x}'$ are in $\mathbf{F}_i$ then the entire line is in $\mathbf{F}_i$. Since $\mathbf{x}_i$ is not in $\mathbf{F}_{i+1}$ and $\mathbf{x}'$ is, there must be a point at which the line enters $\mathbf{F}_{i+1}$. The boundary where the line enters $\mathbf{F}_{i+1}$ must be on the hyperplane $\mathbf{a}_i \cdot \mathbf{x} = 0$ otherwise the line would be entering $\mathbf{F}_i$ as well as $\mathbf{F}_{i+1}$. Call this entry point $\mathbf{x}_{i+1}$. If $\mathbf{x}_{i+1} = \mathbf{x}'$ one is finished. Otherwise, the objective function at $\mathbf{x}_{i+1}$ is less than or equal to the objective function at $\mathbf{x}'$. If it is less then at $\mathbf{x}'$ one has a contradiction. If it is equal then both $\mathbf{x}_{i+1}$ and $\mathbf{x}'$ are provisional optimal points in $\mathbf{F}_{i+1}$.

Since $\mathbf{a}_i \cdot \mathbf{x}_{i+1} = 0$ one can use the equation $\mathbf{a}_i \cdot \mathbf{x} = 0$ to eliminate a variable from the system of inequalities. The objective function is treated just as any of the $\mathbf{a}_i$ except that one ignores the $d+1$st coordinate. Eventually one will be left with a one dimensional problem: finding the smallest or largest of n numbers.

The algorithm runs in expected time $O(d!n)$, where d is the dimension of the space being searched. More importantly, the constant is small. Specifically, the algorithm run in expected time $d!c(n)$ where $c(n)$ is the time it takes to find the smallest of n numbers.

The major shortcoming of this algorithm is that there must be d artificial constraints introduced to keep the solution bounded. If the desired solution is unbounded, one would like to find the unit vector $\mathbf{v}$ such that $\lambda\mathbf{v}$ is feasible for all $\lambda \geq \lambda_0$ for some λ_0 and $\mathbf{v}$ minimizes the inner product $\mathbf{c} \cdot \mathbf{v}$ amongst all such $\mathbf{v}$. This can be thought of as the optimum point on the hyperplane at infinite. This cannot be computed with the method just presented.

In [100] a method is given for computing the asymptotic direction cosines when the artificial constraints are allowed to go to infinite. This can be thought of as the optimum point on the box at infinite. The optimum point on the box at infinite and the optimum point on the hyperplane at infinite are not necessarily the same. For certain calculations the optimum point on the hyperplane at infinite is needed. In the next section an algorithm is presented to find this point.

4.1.2 Unbounded Linear Programming

This can be done by solving the problem in homogeneous coordinates. The solution is now represented by a $d+1$-tuple $\mathbf{x} = (x_0, \ldots, x_d)$ representing the Euclidean point $(x_0/x_d, \ldots, x_{d-1}/x_d)$ with $x_d \geq 0$. If $x_d = 0$ then $\mathbf{x}$ represents the point on the hyperplane at infinity in the direction $(x_0, \ldots, x_{d-1})$. The point $(0, \ldots, 0)$ is disallowed. The constraints are now

$$\sum_{j=0}^{d} a_{i,j} x_j \geq 0 \tag{4.6}$$

for $i = 0, \ldots, n-1$, and the objective function is

$$\frac{\sum_{j=0}^{d-1} c_j x_j}{x_d}. \tag{4.7}$$

Whereas the bounded linear programming algorithm required d extra constraints, the unbounded linear programming algorithm requires the single extra constraint $x_d \geq 0$.

Again, dot product notation simplifies things. Let

$$\mathbf{a}_i \cdot \mathbf{x} \equiv \sum_{j=0}^{d} a_{i,j} x_j \tag{4.8}$$

Then the constraints are

$$\mathbf{a}_i \cdot \mathbf{x} \geq 0 \tag{4.9}$$

and the objective function is

$$\frac{\mathbf{n} \cdot \mathbf{x}}{\mathbf{d} \cdot \mathbf{x}} \tag{4.10}$$

where the numerator functional is $\mathbf{n} = (n_0, \ldots, n_{d-1}, 0)$ and the denominator functional is $\mathbf{d} = (0, \ldots, 0, 1)$. Note that "$\cdot$" now has its conventional meaning for both objective functions and constraints. Since this objective function can take on infinite and undefined values, one orders these values as follows:

1. Let Region 3 be the set of points $\mathbf{x}$ such that $\mathbf{d} \cdot \mathbf{x} = 0$ and $\mathbf{n} \cdot \mathbf{x} \geq 0$.
2. Let Region 2 be the set of points $\mathbf{x}$ such that $\mathbf{d} \cdot \mathbf{x} > 0$.
3. Let Region 1 be the set of points $\mathbf{x}$ such that $\mathbf{d} \cdot \mathbf{x} = 0$ and $\mathbf{n} \cdot \mathbf{x} < 0$.

Region 2 is where the objective function is finite, Region 1 is where it is infinitely negative, and Region 3 is where it if infinitely positive. Let the value of the objective function in

Region i be (by convention) greater than the value of the objective function in Region j if $i > j$. Within Regions 1 and 3 the value of the function is ordered by

$$\mathbf{n} \cdot \frac{\mathbf{x}}{|\mathbf{x}|} \tag{4.11}$$

The objective function is ordered in the usual manner in Region 2.

Before continuing, the following disclaimer needs to be made: if the optimum point is in Region 3, the algorithm may not find it. For this to be the case there can be no finite feasible point and not infinite feasible point where the numerator functional is negative. This case will not be encountered in any of the applications in this thesis.

The homogeneous algorithm works very similarly to the Cartesian algorithm. Recall that the first constraint is required to be $x_d \geq 0$. The optimum with respect to just the first constraint is simply $(-n_0, \ldots, -n_{d-1}, 0)$, the point at infinity in the direction opposite $\mathbf{n}$.

At step i the algorithm will have found a point $\mathbf{x}_i$ that minimizes the objective function within $\mathbf{F}_i$. As in the Cartesian version, if the ith provisional optimum is on the correct side of the $i+1$st constraint it becomes the $i+1$st provisional optimum. Otherwise, if there is any feasible solution, the $i+1$st provisional optimum must satisfy the $i+1$st constraint with equality. This is now stated formally and proven.

Theorem 6 *If* $\mathbf{a}_i \cdot \mathbf{x} < 0$ *then* $\mathbf{x}_{i+1}$ *satisfies* $\mathbf{a}_i \cdot \mathbf{x}_{i+1} = 0$.

Proof: If $\mathbf{x}_{i+1}$ is in Region 3 then there is no finite solution.

The case when $\mathbf{x}_{i+1}$ and $\mathbf{x}_i$ are in Region 2 is entirely finite and has been dealt with in Section 4.1.1.

If $\mathbf{x}_{i+1}$ is in Region 2 and $\mathbf{x}_i$ is in Region 1 then consider the a line joining $\mathbf{x}_i$ with $\mathbf{x}_{i+1}$. This line is entirely contained in $\mathbf{F}_i$ since $\mathbf{F}_i$ is convex. One wishes to show that the value of the objective function increases monotonically on the line from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$. Except at $\mathbf{x}_i$ the line is in Region 2. Thus, one must only show that the value of the objective function on the line is monotonically increasing in Region 2. Parametrize the line by

$$\mathbf{x}_i(1-t) + \mathbf{x}_{i+1}t \tag{4.12}$$

$t \in (0, 1]$. The value of the objective function as a function of t is

$$\frac{\mathbf{n} \cdot (\mathbf{x}_i(1-t) + \mathbf{x}_{i+1}t)}{\mathbf{d} \cdot (\mathbf{x}_i(1-t) + \mathbf{x}_{i+1}t)} \tag{4.13}$$

Noting that $\mathbf{d} \cdot \mathbf{x}_i = 0$ one has

$$(\frac{\mathbf{n} \cdot \mathbf{x}_i}{t} + \mathbf{n} \cdot (\mathbf{x}_{i+1} - \mathbf{x}_i))/(\mathbf{d} \cdot \mathbf{x}_{i+1}) \tag{4.14}$$

Since $\mathbf{n} \cdot \mathbf{x}_i < 0$ and $\mathbf{d} \cdot \mathbf{x}_{i+1} > 0$ one can see that the objective function *does* increase monotonically from $-\infty$ to $(\mathbf{n} \cdot \mathbf{x}_{i+1})/(\mathbf{d} \cdot \mathbf{x}_{i+1})$. Since $\mathbf{x}_i$ is not in $\mathbf{F}_{i+1}$ but $\mathbf{x}_{i+1}$ is, there must be a point $\mathbf{x}'$ at which the line enters $\mathbf{F}_{i+1}$ (crosses the hyperplane $\mathbf{a}_i \cdot \mathbf{x} = 0$) . If $\mathbf{x}' \neq \mathbf{x}_{i+1}$ then the objective function is less at $\mathbf{x}'$ than at $\mathbf{x}_{i+1}$, a contradiction. Thus, $\mathbf{x}' = \mathbf{x}_{i+1}$ and $\mathbf{a} \cdot \mathbf{x}_{i+1} = 0$.

If $\mathbf{x}_{i+1}$ is in Region 1 and $\mathbf{x}_i$ is in Region 1, then consider again the line joining $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$. Now the line will be contained entirely in Region 1. Without loss of generality assume that $|\mathbf{x}_i| = |\mathbf{x}_{i+1}| = 1$. Then parametrize the line as follows

$$\mathbf{x}(t) = \mathbf{x}_i(1-t) + \mathbf{x}_{i+1}t. \tag{4.15}$$

The value of the objective function on the line is

$$\mathbf{n} \cdot \frac{\mathbf{x}(t)}{|\mathbf{x}(t)|} \tag{4.16}$$

Note that

$$|\mathbf{x}(t)|^2 = \mathbf{x}_i \cdot \mathbf{x}_i(1-t)^2 + 2\mathbf{x}_i \cdot \mathbf{x}_{i+1}(1-t)t + \mathbf{x}_{i+1} \cdot \mathbf{x}_{i+1}t^2. \tag{4.17}$$

Since $(1-t)t > 0$ on $(0,1)$ and $\mathbf{x}_i \cdot \mathbf{x}_{i+1} < 1$ one has $|\mathbf{x}(t)| < 1$ when $t \in (0,1)$. One needn't worry about the possibility that $|x(t)| = 0$. This would imply that $\mathbf{x}_{i+1} = -\mathbf{x}_i$. This is not possible since both $\mathbf{x}_{i+1}$ and $\mathbf{x}_i$ are in Region 1.

One would like to show that the value of the objective function on the interior of the line segment from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$ is less than the value of the objective function at $\mathbf{x}_{i+1}$. That is, one would like to show that the following value is negative:

$$\mathbf{n} \cdot \frac{\mathbf{x}(t)}{|\mathbf{x}(t)|} - \mathbf{n} \cdot \mathbf{x}_{i+1} \tag{4.18}$$

for $t \in (0,1)$. Multiplying by $|\mathbf{x}(t)|$ yields:

$$\mathbf{n} \cdot \mathbf{x}(t) - \mathbf{n} \cdot \mathbf{x}_{i+1}|\mathbf{x}(t)| \tag{4.19}$$

Since $\mathbf{n} \cdot \mathbf{x}_i \leq \mathbf{n} \cdot \mathbf{x}_{i+1}$ the next expression is at least as large as the previous:

$$\mathbf{n} \cdot \mathbf{x}_{i+1} - \mathbf{n} \cdot \mathbf{x}_{i+1}|\mathbf{x}(t)| \tag{4.20}$$

Combining terms yields

$$\mathbf{n} \cdot \mathbf{x}_{i+1}(1 - |\mathbf{x}(t)|). \tag{4.21}$$

Since $\mathbf{x}_{i+1}$ is in Region 1 $\mathbf{n} \cdot \mathbf{x}_{i+1} < 0$ and the above is negative. Thus, the value of the objective function is less on the interior of the line than it is at $\mathbf{x}_{i+1}$. Since $\mathbf{x}_{i+1}$ is in $\mathbf{F}_{i+1}$ and $\mathbf{x}_i$ is not, the line must enter $\mathbf{F}_{i+1}$ (i.e cross the hyperplane $\mathbf{a}_i \cdot \mathbf{x} = 0$) at some point $\mathbf{x}'$. If $\mathbf{x}' = \mathbf{x}_i$ then $\mathbf{x}_i$ is feasible and $\mathbf{x}_{i+1} = \mathbf{x}_i$. If $\mathbf{x}' \neq \mathbf{x}_{i+1}$ and $\mathbf{x}' \neq \mathbf{x}_i$ then $\mathbf{x}'$ would be on the interior of the line segment and the value of the objective function at $\mathbf{x}'$ would would less than at $\mathbf{x}_{i+1}$. Thus $\mathbf{x}' = \mathbf{x}_{i+1}$ and $\mathbf{a}_i \cdot \mathbf{x}_{i+1} = 0$.

■

Since the solution $\mathbf{x}_{i+1}$ satisfies $\mathbf{a}_i \cdot \mathbf{x}_{i+1} = 0$, one can recursively solve the problem in the lower dimensional space. The analysis of the running time of this algorithm is identical to that in [99,100].

While the behavior of the algorithm in the infinite region (Region 1) has been treated as a special case, one could run the algorithm with $\mathbf{d} = 0$ from the very beginning. In order to ensure that the solution is found in Region 1, as opposed to Region 3, one must add the additional constraint $\mathbf{n} \cdot \mathbf{x} \leq 0$. By doing this, one can solve minimum distance problems such as the three listed at the beginning of this section, which cannot be posed as finite linear programming problems.

4.2 Spatial Separability

At many times in the surface intersection algorithm it is necessary to perform a test that determines if it is *not* possible for two objects (either curves or surfaces) to intersect. This test, along with the test whether two Gauss maps intersect, constitutes a major portion of the running time of the algorithm, and care must be taken to make it efficient.

In the following, it is assumed that all curves and surfaces have an associated set of *bounding points*. The bounding points must have the property that the curve or surface is contained completely within their convex hull. Bézier curves and surfaces as well as B-spline curves and surface have this property. For implicit surfaces one can specify a finite portion of the surface by a bounding box. To subdivide the surface one merely subdivides the box.

In this way, implicit surfaces also have a set of *bounding points*, namely the corners of the box.

Determining that the convex hulls of two sets of bounding points do not intersect is, by Farkas' Lemma [34], equivalent to find a plane which separates the two sets of points. If the normal to a separating plane where known beforehand, the plane could be computed simply by performing a small number of inner products and comparisons. The cost of finding a separating plane is incurred when one must investigate a large number of candidate separating planes. The candidate planes associated with certain normals are cheap to investigate while other normals are likely to be normals to actual separating planes. The strategy is to investigate the cheap normals first; if that fails to investigate the likely ones; and then if that fails to use linear programming to find a separating plane if one exists.

4.2.1 Bounding Boxes

The least expensive normals to investigate are the axis vectors $\hat{x}$, $\hat{y}$, and $\hat{z}$. Investigating these planes is equivalent to computing the bounding box around a curve or surface. When an object is first queried for intersection testing, the minimum and maximum of the coordinate values of the bounding points are calculated. These values are then stored with the object for later use. If the object is a surface, it will likely be tested against four other surfaces, and the four boundary curves of each surface it intersects. Thus, the cost of computing the bounding box, which is small to begin with, is amortized over all its uses and can be considered virtually free. It is a false conclusion, often reached, that one should therefore use bounding boxes and no other test. The bounding box test can, for a very little amount of work, cause the algorithm to go on and perform a large amount of work, when a more sophisticated test which requires a moderate amount of work can cause the algorithm to finish immediately. This will be shown in Section 4.2.4. Thus, two costlier tests are presented.

4.2.2 Bounding Planes

If the bounding box fails, as it tends to when two nearly parallel but non-intersecting surfaces are presented to the intersection algorithm, the algorithm proceeds to use a more expensive but less crude test. In this test, the candidate normals are roughly the normals to the surfaces.

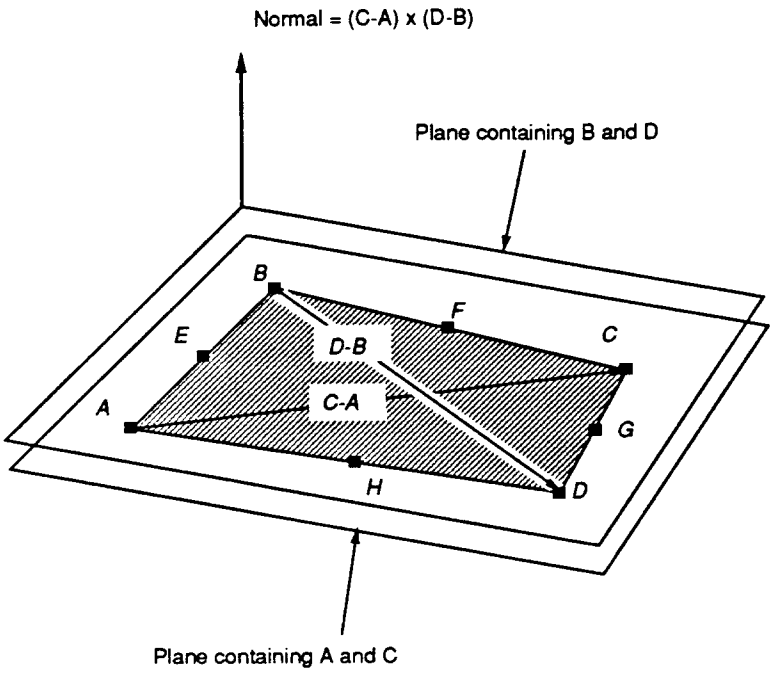


Figure 4.1: Determining bounding planes simply.

For a Bézier surface or B-spline surface, it is reasonable to use the corner control vertices (A, B, C, and D in Figure 4.1) to define a rough tangent plane. Unfortunately, three points define a plane and if the four corner vertices are not co-planar then the normal depends on which are chosen. Selecting any three is asymmetric and aesthetically unappealing. Fortunately, the normal perpendicular to the plane containing the midpoints (E, F, G, and H in Figure 4.1) to the sides of the quadrilateral is symmetrically defined, and if the corner points are nearly planar, minimizes the distance between the resulting parallel planes [110]. The normal to these planes is given by $(C - A) \times (D - B)$.

4.2.3 Separating Planes

If neither the bounding boxes nor the bounding planes establish that the geometry is separable then the most expensive and (hopefully) most fruitful test is employed. This test attempts to find a *separating plane* between the bounding points of one object and the bounding points of the other. If the points in the first set are $\{x_i, y_i, z_i\}$ and the points of the second set are $\{X_j, Y_j, Z_j\}$ then a separating plane is a plane $ax + by + cz + d = 0$ so

that

$$ax_i + by_i + cz_i + d \geq 0 \quad i = 0, \ldots, n-1 \tag{4.22}$$

and

$$aX_j + bY_j + cZ_j + d \leq 0 \quad j = 0, \ldots, N-1 \tag{4.23}$$

([86] page 291). Finding such a plane is a linear programming problem in projective 3-space: the constraints are given in equations 4.22 and 4.23 and the feasible point's homogeneous coordinates are (a, b, c, d). The techniques discussed in Section 4.1 can be applied to solve this efficiently. Notice that there is no objective function to apply, since no separating plane is any better than another.

A slight complication to this scheme is that the curve or surface might lie *on* the separating plane. In this case, the existence of a separating plane would not ensure that the two objects do not intersect. This can be overcome by solving the following linear program instead. Find $(a, b, c, d, \epsilon) \in \mathbf{P}^4$ satisfying

$$ax_i + by_i + cz_i + d + \epsilon \geq 0 \quad i = 0, \ldots, n-1 \tag{4.24}$$

and

$$aX_j + bY_j + cZ_j + d - \epsilon \leq 0 \quad j = 0, \ldots, N-1 \tag{4.25}$$

and minimizing

$$\frac{\epsilon}{\sqrt{a^2 + b^2 + c^2 + d^2 + \epsilon^2}}. \tag{4.26}$$

To have the algorithm return the desired point one simply assigns $\mathbf{n} = (0, 0, 0, 0, 1)$, $\mathbf{d} = (0, 0, 0, 0, 0)$ and adds the constraint that $\epsilon \leq 0$. The algorithm operates entirely in Region 1 and thus returns the point minimizing the expression 4.26. The linear programming algorithm will return the plane $ax + by + cz + d = 0$ such that the distance from the plane to the closest point of either set is maximized. In this way, one can perform the separation test robustly even in the face of roundoff error. If the separating distance is large then one can confidently say that the objects do not intersect. If the distance is small then one needs to subdivide the objects being compared, or investigate the possibility that they intersect.

4.2.4 Performance Analysis

Some performance figures for these methods are presented in Figure 4.2. To obtain these data, the surfaces shown in Figure 4.3 are presented to the surface intersection

algorithm. The separation between the surfaces is varied as well as the method used to determine if the surfaces were separable. For one series of separation distances, the algorithm uses only bounding boxes to determine if the surfaces are separable. At a separation δ, the algorithm must subdivide the surfaces until the sub-surfaces have size approximately δ. This results in roughly $1/\delta^2$ sub-surfaces being created. These sub-surfaces are compared to roughly a constant number of neighbors, resulting in $O(1/\delta^2)$ performance, which can be seen in the graph.

For the next series of separation distances, if the bounding box test fails, the linear programming test is applied. In this case, at a separation of δ the algorithm must subdivide the sub-surfaces until they have dimension approximately $\sqrt{\delta}$. This can be seen by considering a sub-patch of a paraboloid. If the dimensions of a patch are $l \times l$, the patch and its bounding points will deviate from a plane by roughly l^2. When this deviation is roughly equal to δ the patch will pass the linear programming test. This results in roughly $1/\delta$ sub-patches being created, each being compared to an approximately constant number of neighbors. This results in an overall running time of $O(1/\delta)$.

In the next series of runs, the bounding box test is applied, and if that fails, the bounding plane test is applied. The geometry here is optimal for such a test and the graph reveals that this cuts the running time in half compared to the linear programming approach. The asymptotic behavior is identical, as expected.

One cannot expect that the bounding plane approach will always work so well. For instance, if a curve passes to the side of a surface, but through its bounding plane the test will clearly fail. One would like an approach that combines the speed of the bounding plane approach with the flexibility of the linear programming approach. By applying the bounding box test first, followed by the bounding plane test, and finally the linear programming test, this can be achieved. The graph shows that this combination gives nearly the same performance as the bounding box/bounding plane combination.

4.3 Spherical Separability

If the two objects are already known to intersect at a point and one wishes to determine if they intersect in no other points then a variation of 3-D separability test must be employed. Consider, for example the situation depicted in Figure 4.4. A curve and a

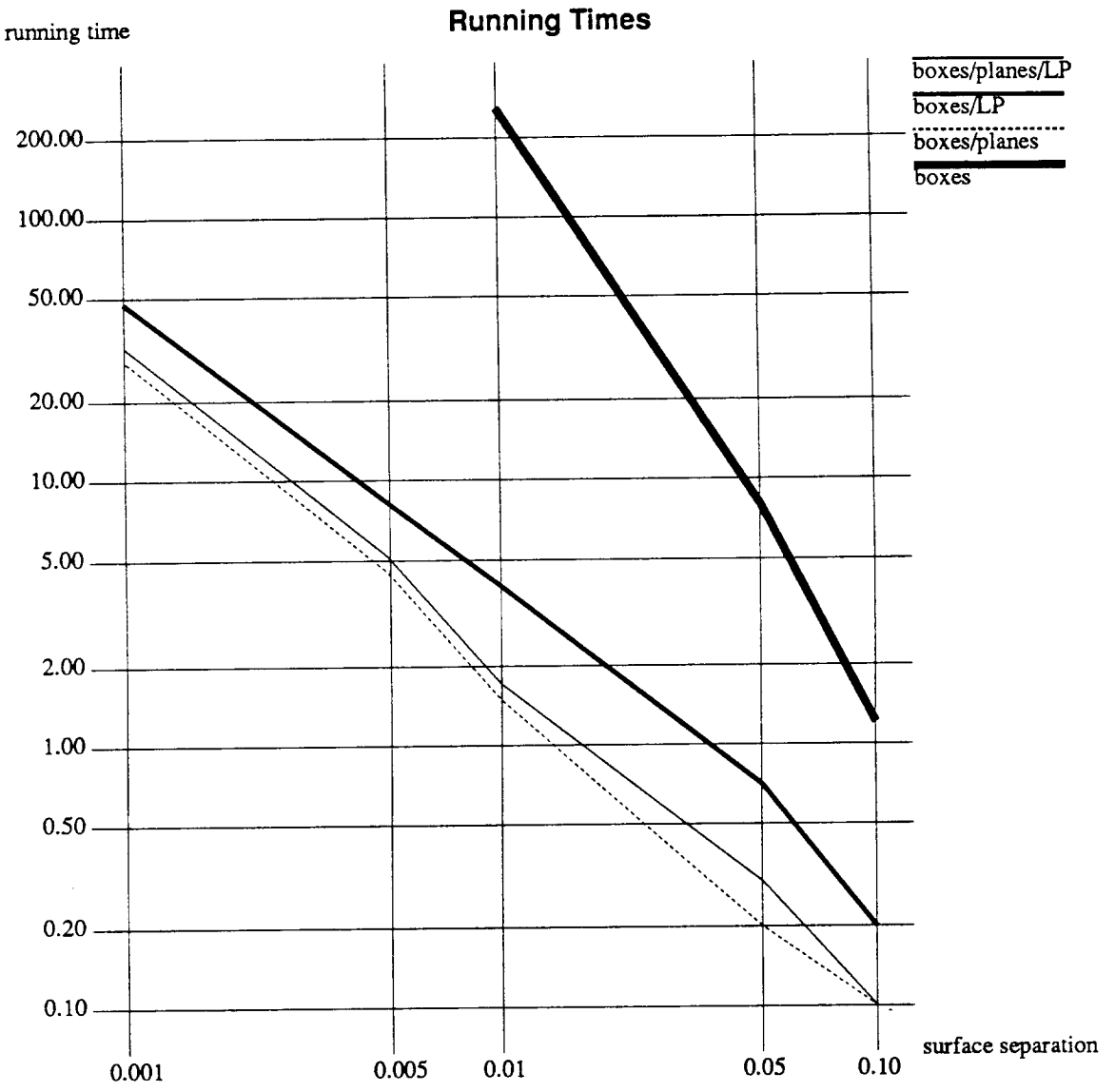


Figure 4.2: Log log plot of the performance of the separation tests.

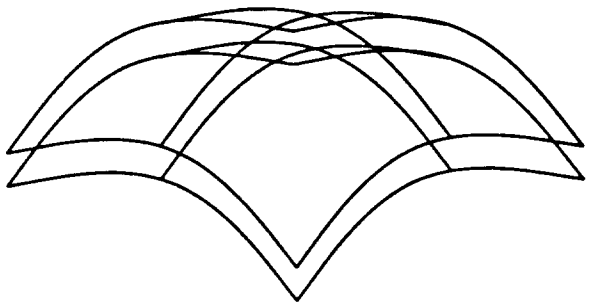


Figure 4.3: Surfaces used to obtain performance for separation tests.

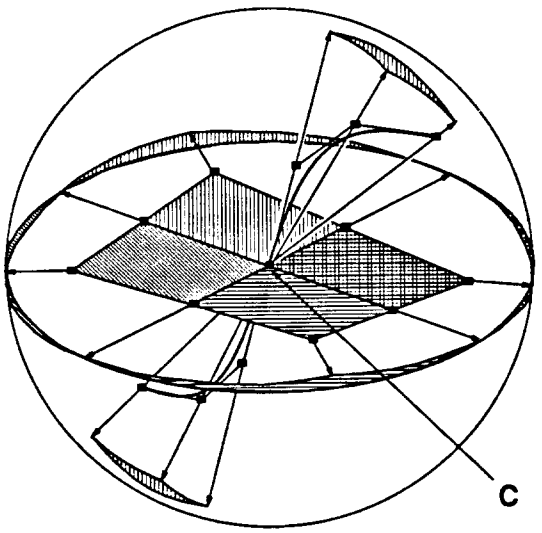


Figure 4.4: Determining if touching objects do not otherwise intersect.

surface are known to intersect at the point **C**. One wishes to know if they intersect at any other point. The surface is split into four sub-patches and the curve into two sub-curves. Every point except **C** is mapped to a point on the unit sphere centered at **C** by a simple central projection. If a sub-patch and a sub-curve intersect then their projections onto the sphere must intersect. If one projects every point of the sub-curve onto the sphere and every point of the sub-patch onto the sphere and these sets do not intersect, then the sub-curve and sub-patch cannot intersect except at **C**.

Practically, this is accomplished as follows. If every control vertex of the Bézier or B-spline surface (curve) except the control vertex corresponding to the surface corner (curve end) at **C** is projected onto the sphere, the convex hull of the resulting point set contains the projection of the surface (curve) onto the sphere. This can be seen as follows: Every point **p** on the surface (curve) is a convex combination of the control points $\mathbf{c}_i$ of the surface (curve)

$$\mathbf{p} = \sum_{i=1}^{n} a_i \mathbf{c}_i \tag{4.27}$$

where $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = 1$. Let $\mathbf{c}_1$ be the control point located at **C**. The vector which projects **p** onto the sphere is

$$\begin{aligned} \mathbf{p} - \mathbf{C} &= \sum_{i=1}^{n} a_i \mathbf{c}_i - \mathbf{c}_1 && (4.28) \\ &= \sum_{i=1}^{n} a_i \mathbf{c}_i - \sum_{i=1}^{n} a_i \mathbf{c}_1 && (4.29) \\ &= \sum_{i=2}^{n} a_i (\mathbf{c}_i - \mathbf{c}_1) && (4.30) \end{aligned}$$

It is a property of B-spline and Bézier surfaces (curves) that the weight of the corner control point of a surface (end control point of a curve), a_1, is strictly less than 1 for all points on the surface (curve) except the corner (end) point. Thus, the a_i for $i = 2, \ldots, n$ are not all zero. Since none of them are negative the projection of **p** is a convex combination of the projections of the other control points. The task then is to determine if the convex hulls on the sphere intersect.

4.3.1 Spherical Bounding Boxes

As in the three-dimensional case, one would like to perform an inexpensive test first and then a more expensive and (hopefully) more fruitful test later. The inexpensive

test is analogous to the bounding box test in three dimensions. The bounding box test in 3-D can be thought of as a limited search for a separating plane. The search is limited in the sense that one is only considering axis-aligned separating planes. This is a sensible thing to do since the plane of the form $\{x = a\}$ with the largest value of a and with all the points of a set S to the right can be found simply by finding the minimum value of the x-coordinates of the points in S, and similarly for y and z. That is, there is a set of separating planes that are markedly less expensive to compute than others.

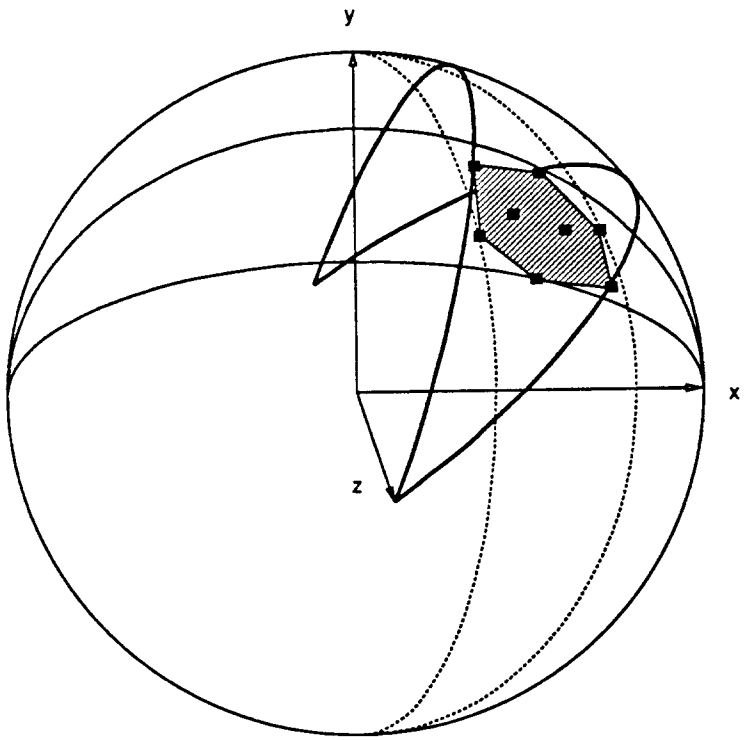


Figure 4.5: "Bounding box" on the sphere.

On the sphere, the separating "lines" are great circles of the unit sphere. The separating lines to consider are those that contain one of the three principle axes, that is, planes of the form $Ax + By = 0$ or $Ax + Bz = 0$ or $Ay + Bz = 0$. For the point set depicted in Figure 4.5, there will be a set of six bounding "lines", two corresponding to each principle axis. If the convex hull of the point set contains an axis point (one of the points $\pm\mathbf{e}_i$) then it will not have any bounding lines corresponding to that axis.

These bounding "lines" can be found as follows. Consider the bounding lines intersecting the x-axis. Construct the set of all pairs $\{y_i, z_i\}$. Define a *wedge* to be the intersection of two half-planes whose boundaries contain the origin. One now has a set of points in the plane and must find the smallest wedge (i.e. the wedge with the smallest angle) containing them. This can clearly be done in $O(n)$ time. When this wedge is extruded into three dimensions and intersected with the unit sphere, one obtains bounding circles depicted in Figure 4.5

4.3.2 Separating Circles

If the spherical bounding box test fails then a more expensive but hopefully more sensitive test is employed. This test casts the problem as a linear programming problem and uses the linear programming algorithm given in 4.1. Let one set of vectors be $\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ and the other be $\{\mathbf{w}_1, \ldots, \mathbf{w}_n\}$. The task is to obtain a vector $\mathbf{P}$ so that $\mathbf{P} \cdot \mathbf{v}_i > 0$ for $i = 1, \ldots, m$ and $\mathbf{P} \cdot \mathbf{w}_j < 0$ for $j = 1, \ldots, n$. This can be cast as a linear programming problem in projective three-space where one tries to find $\mathbf{x} = (\mathbf{P}, \epsilon) \in \mathbf{P}^3$ minimizing

$$\frac{\epsilon}{\sqrt{\mathbf{P} \cdot \mathbf{P} + \epsilon^2}} \tag{4.31}$$

and subject to

$$\mathbf{P} \cdot \mathbf{v}_i + \epsilon \geq 0 \tag{4.32}$$

for all i and

$$\mathbf{P} \cdot \mathbf{w}_i - \epsilon \leq 0 \tag{4.33}$$

for all j. As in the three dimensional case, this is accomplished by setting $\mathbf{n} = (0, 0, 0, 1)$ and $\mathbf{d} = (0, 0, 0, 0)$ and adding the additional constraint $\mathbf{n} \cdot \epsilon \leq 0$.

4.4 Bounding Gauss Maps

Another method demanded by the surface intersection algorithm is the ability to compute bounds on the Gauss maps of surfaces. That is, one must be able to construct the Gauss maps N_1 and N_2 and determine if vectors $\mathbf{P}_1$ and $\mathbf{P}_2$ (see section 3.2) exist. Some work in this direction has been presented in [56,58,94,95,96]. These results provide overly generous bounds on the Gauss maps. This can interfere with the ability of the loop detection

criterion to discern cases in which no loops can exist. Tighter bounds are presented in this section.

If the vectors, $\mathbf{P}_1$ and $\mathbf{P}_2$ do exist then it would be desirable to know them so that points on the intersection curve can be ordered. In the following section, it is discussed how this can be done for certain classes of surfaces. Although methods to do this for all forms of surfaces employed in modeling systems are not presented here, the development of such methods appears straightforward.

4.4.1 Computing the Gauss Map for Quadric Surfaces

For small classes of surfaces, such as quadrics and tori, bounds on the Gauss map can be constructed on a case by case basis. This will undoubtedly realize some performance benefit over a more general approach.

4.4.2 Computing the Gauss Map for Parametric Surfaces

For a parametric surface $\mathbf{F}(u, v)$ it is often possible to explicitly compute the vector function

$$\mathbf{N}(u, v) \equiv \frac{\partial \mathbf{F}}{\partial u} \times \frac{\partial \mathbf{F}}{\partial v}. \tag{4.34}$$

Let us call $\mathbf{N}(u, v)$ the *pseudo-normal function* of $\mathbf{F}$. The Gauss map is obtained from $\mathbf{N}(u, v)$ by projecting $\mathbf{N}(u, v)$ onto the unit sphere. If one can compute bounds on the pseudo-normal function these can be used to bound the Gauss map.

Computing the Gauss Map for Bézier Surfaces

For a polynomial parametric surface $\mathbf{F}(u, v)$ in Bézier form it is possible to compute the pseudo-normal function as a Bézier surface, as depicted in Figure 4.6. All that is necessary for such a computation is the ability to represent each component function separately, and the ability to multiply and add and subtract two such component functions.

Both Sederberg [95] and Kriezis [58] hesitate to compute the tightest possible bounds on the Gauss map because of the computational expense. If one were to compute the pseudo-normal surface via the above method for each sub-surface, the cost would indeed dominate the running time of the surface intersection algorithm. However, one can compute this surface once for the root surface and then compute the pseudo-normal surfaces for any sub-surfaces by subdividing (as a Bézier surface) the pseudo-normal surface of the parent.

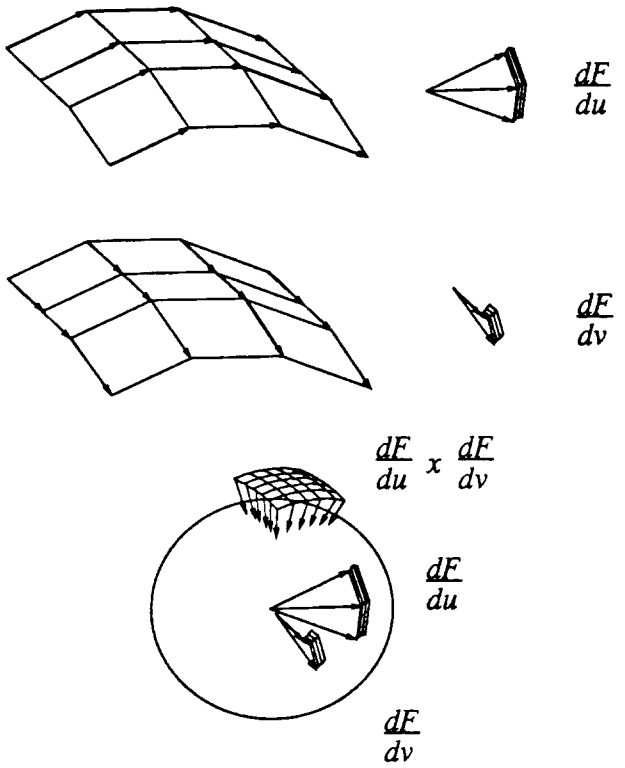


Figure 4.6: Computation of bounds on the Gauss map.

The cost of this will be comparable to the cost of subdividing the Bézier surface itself. Timing results on the implemented algorithm indicate that the cost is not major. Rather the major costs are associated with the tests to determine if the geometry is separable. In light of this, it seems prudent to spend more time computing a tighter bound to avoid the cost associated with recursive operation of the algorithm on the created sub-surfaces.

Computing the Gauss Map for B-spline Surfaces

The pseudo-normal to a B-spline surface is also a B-spline surface. If the original B-spline surface is degree m by degree n, the pseudo-normal surface will be degree $2m-1$ by degree $2n-1$. If the multiplicities of original surface are μ_i and ν_i, then the original surface will be $C^{m-\mu_i}$ at knot i (u-direction) and $C^{n-\nu_i}$ at knot i (v-direction). The continuity of the pseudo-normal surface will be one less in either direction. If the new multiplicities in the u and v directions are denoted μ_i' and ν_i' then $m - \mu_i - 1 = (2m-1) - \mu_i'$ and $n - \nu_i - 1 = (2n-1) - \nu_i'$. Thus $\mu_i' = \mu_i + m$ and $\nu_i' = \nu_i + n$.

To compute the values of the control points of pseudo-normal function, one has two options. One can break the B-spline surfaces into Bézier surfaces and apply the techniques from section 4.4.2. The resulting Bézier surfaces can be combined using knot removal techniques [63,64]. A second approach would be to compute a discrete set of values of the pseudo-normal surface and then solve for the coordinate values of a B-spline with the appropriate degree and multiplicity. A drawback of this approach is that the matrix which takes one from interpolated values to B-spline control points becomes increasingly ill-conditioned as the degree of the B-spline increases. For cubic B-splines with many segments the latter approach seems more appropriate, while for B-splines with few segments and higher order the former methods seems to be appropriate.

Rational Parametric Surfaces

Gauss maps for rational parametric surfaces require some finesse to keep the degree low. The naive approach is as follows. Let $\mathbf{f}$ be the projected surface, that is, the rational function.

$$\mathbf{f}(u,v) = \frac{\mathbf{F}(u,v)}{w(u,v)} \tag{4.35}$$

where $w = F_4$. The first thing to do is to compute the partial derivative of $\mathbf{f}$ with respect to each parameter

$$\mathbf{f}_u = \frac{\mathbf{F}_u w - \mathbf{F} w_u}{w^2} \tag{4.36}$$

and

$$\mathbf{f}_v = \frac{\mathbf{F}_v w - \mathbf{F} w_v}{w^2} \tag{4.37}$$

The pseudo-normal surface N can then be computed as

$$\mathbf{N} = \mathbf{f}_u \times \mathbf{f}_v. \tag{4.38}$$

If the degree of F is m in u and n in v then the degree of $\mathbf{f}_u$ will be $2m-1$ in u and $2n$ in v and the degree of $\mathbf{f}_v$ will be degree $2m$ in u and $2n-1$ in v. Thus the degree of $\mathbf{N}$ will be $4m-1$ in u and $4n-1$ in v. For a rational bicubic Bézier patch the pseudo-normal surface would be degree 11 by degree 11 and would contain 144 control vertices.

One can do somewhat better. Using equations 4.37 and 4.38 one can write the pseudo-normal surface as

$$\mathbf{N} = (\mathbf{F}_u w - \mathbf{F} w_u) \times (\mathbf{F}_v w - \mathbf{F} w_v). \tag{4.39}$$

Expanding yields

$$\mathbf{N} = \mathbf{F}_u \times \mathbf{F}_v w^2 - \mathbf{F}_u \times \mathbf{F} w_v w - \mathbf{F} w_u \times \mathbf{F}_v w + \mathbf{F} \times \mathbf{F} w_v w_v. \tag{4.40}$$

Note that $\mathbf{F} \times \mathbf{F} = 0$ gives

$$\mathbf{N} = \mathbf{F}_u \times \mathbf{F}_v w^2 - \mathbf{F}_u \times \mathbf{F} w_v w - \mathbf{F} w_u \times \mathbf{F}_v w. \tag{4.41}$$

This allows us to divide by w

$$\mathbf{N} = \mathbf{F}_u \times \mathbf{F}_v w - \mathbf{F}_u \times \mathbf{F} w_v - \mathbf{F} w_u \times \mathbf{F}_v \tag{4.42}$$

resulting in a pseudo-normal surface of degree $3m-1$ in u and $3n-1$ in v. Using this method, the pseudo-normal surface of a rational bicubic Bézier patch will be degree 8 in each direction and will have 81 control vertices.

4.4.3 Computing the Gauss Map for Implicit Surfaces

If a surface were given in implicit form, say as the zero set of a tri-variate scalar valued polynomial $f(x, y, z)$, then one could again obtain a pseudo-normal function (in fact the gradient function)

$$\mathbf{N}(x, y, z) \equiv \nabla f. \tag{4.43}$$

The convex hull of $\mathbf{N}$'s three-dimensional lattice of Bernstein control vectors would form a bound on the gradient function and thus the Gauss map.

4.4.4 Gauss Map Separability

Having obtained the discrete sets N_1 and N_2 whose convex hulls are bounds on the Gauss maps of the surfaces $\mathbf{F}(s, t)$ and $\mathbf{G}(u, v)$, respectively, one must find $\mathbf{P}_1$ and $\mathbf{P}_2$ such that

$$\mathbf{P}_1 \cdot \mathbf{n}_1 > 0, \quad \mathbf{P}_1 \cdot \mathbf{n}_2 < 0 \tag{4.44}$$

$$\mathbf{P}_2 \cdot \mathbf{n}_1 > 0, \quad \mathbf{P}_2 \cdot \mathbf{n}_2 > 0 \tag{4.45}$$

for all $\mathbf{n}_1 \in N_1$ and all $\mathbf{n}_2 \in N_2$.

Again, one would like to apply a series of tests, starting with the least expensive and ending with the most expensive. Thus, the first test is the spherical bounding box test described in sub-section 4.3.1. If that fails then a linear programming based test is applied.

The linear programming problem implied by equation 4.44 is the same as that discussed in sub-section 4.3.2. One wishes to find $\mathbf{x} = (\mathbf{P}_1, \epsilon) \in \mathbf{P}^3$ satisfying

$$\mathbf{P}_1 \cdot \mathbf{n}_1 + \epsilon \geq 0 \tag{4.46}$$

for all $\mathbf{n}_1 \in N_1$ and

$$\mathbf{P}_1 \cdot \mathbf{n}_2 - \epsilon \leq 0 \tag{4.47}$$

for all $\mathbf{n}_2 \in N_2$ and $\epsilon \leq 0$ minimizing

$$\frac{\epsilon}{\sqrt{\mathbf{P}_1 \cdot \mathbf{P}_1 + \epsilon^2}}. \tag{4.48}$$

The optimum $\mathbf{P}_1$ returned by the algorithm will be a great circle separating $\{\mathbf{n}_{1i}\}$ and $\{\mathbf{n}_{2i}\}$ that attains the largest possible distance from the two sets.

The linear programming problem associated with 4.45 would be to find $\mathbf{x} = (\mathbf{P}_2, \epsilon) \in \mathbf{P}^3$ satisfying

$$\mathbf{P}_2 \cdot \mathbf{n}_1 + \epsilon \geq 0 \tag{4.49}$$

and

$$\mathbf{P}_2 \cdot \mathbf{n}_2 + \epsilon \geq 0 \tag{4.50}$$

and $\epsilon \leq 0$ minimizing

$$\frac{\epsilon}{\sqrt{\mathbf{P}_2 \cdot \mathbf{P}_2 + \epsilon^2}}. \tag{4.51}$$

This optimum $\mathbf{P}_2$ returned by the algorithm will be a great circle with $\{\mathbf{n}_{1i}\}$ and $\{\mathbf{n}_{2i}\}$ on the same side that attains the largest possible distance from the two sets. The circle formed by the intersection of the unit sphere and the plane $\mathbf{P}_2 \cdot \mathbf{n} + \epsilon = 0$ will be the smallest circle containing the union of $\{\mathbf{n}_{1i}\}$ and $\{\mathbf{n}_{2i}\}$.

Typically, when two Gauss maps fail to pass the loop detection criterion, it is because they are antipodal or intersecting, but not both. Thus, one of the linear programming problems will return a feasible solution while the other will not. If one is using a linear programming algorithm which has a tendency to run in less time when there is no feasible solution as with [99], and if the infeasible problem is run first, less time is spent. In order to reap this reward without relying on chance, the two algorithms can be run in an interleaved fashion. As soon as one problem is found to be infeasible, work on both is halted. Since the linear programming algorithm presented in [99] is incremental, one can implement this quite easily. In this way, a factor of 2 speedup can typically be obtained for criterion failures.

4.4.5 Performance Analysis

The performance of these tests can be observed on the example pictured in Figure 4.7. In this example, two Bézier surfaces, shaped much like quarter cones, are being intersected. The Gauss maps of each surface are non-great circle arcs on the Gaussian sphere. The angle at which the surfaces intersect and consequently the distance between the Gauss maps of the two surfaces in controlled by a parameter d. As d becomes small the Gauss maps cannot be separated unless first subdivided into pieces of size approximately $\sqrt{d}$. This leads to running time proportional to $1/\sqrt{d}$ as can be seen in Figure 4.8.

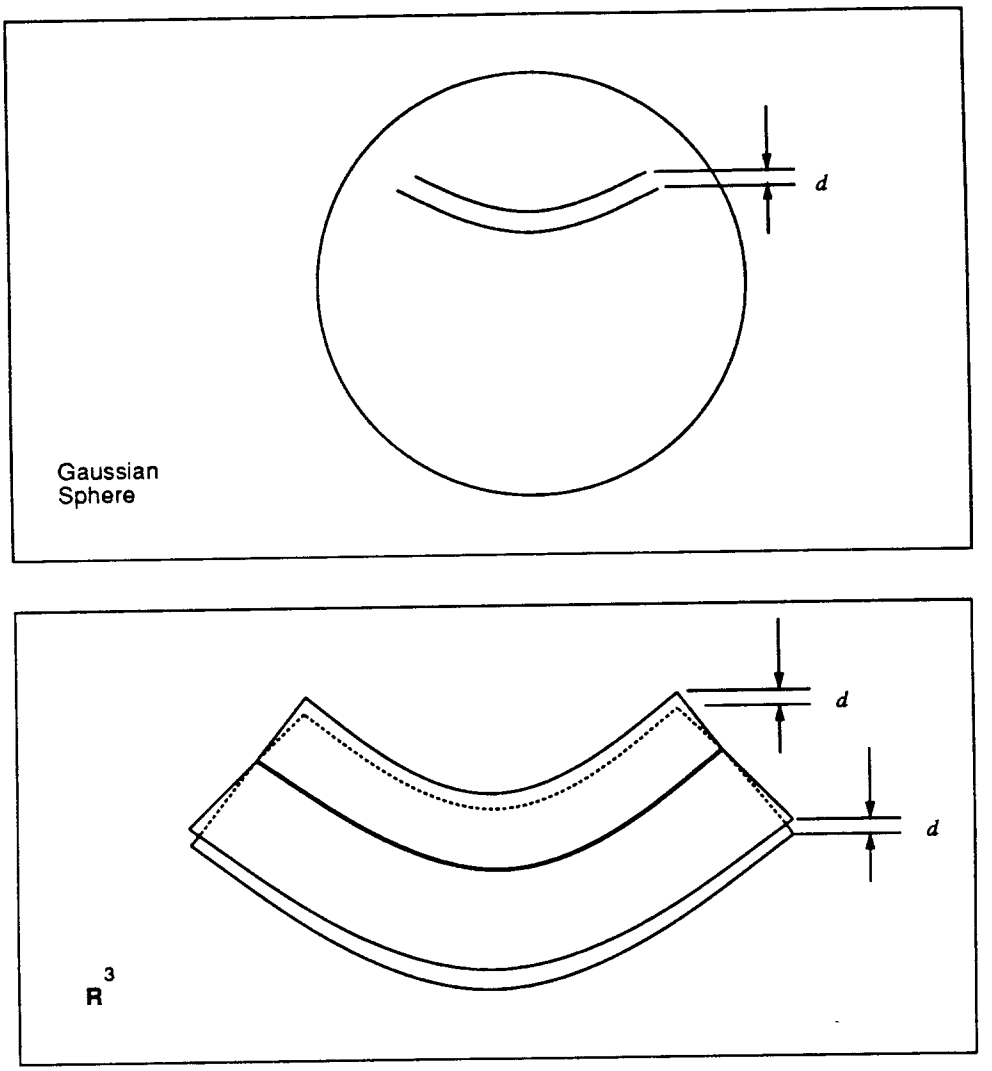


Figure 4.7: Surfaces used to test normal separability test.

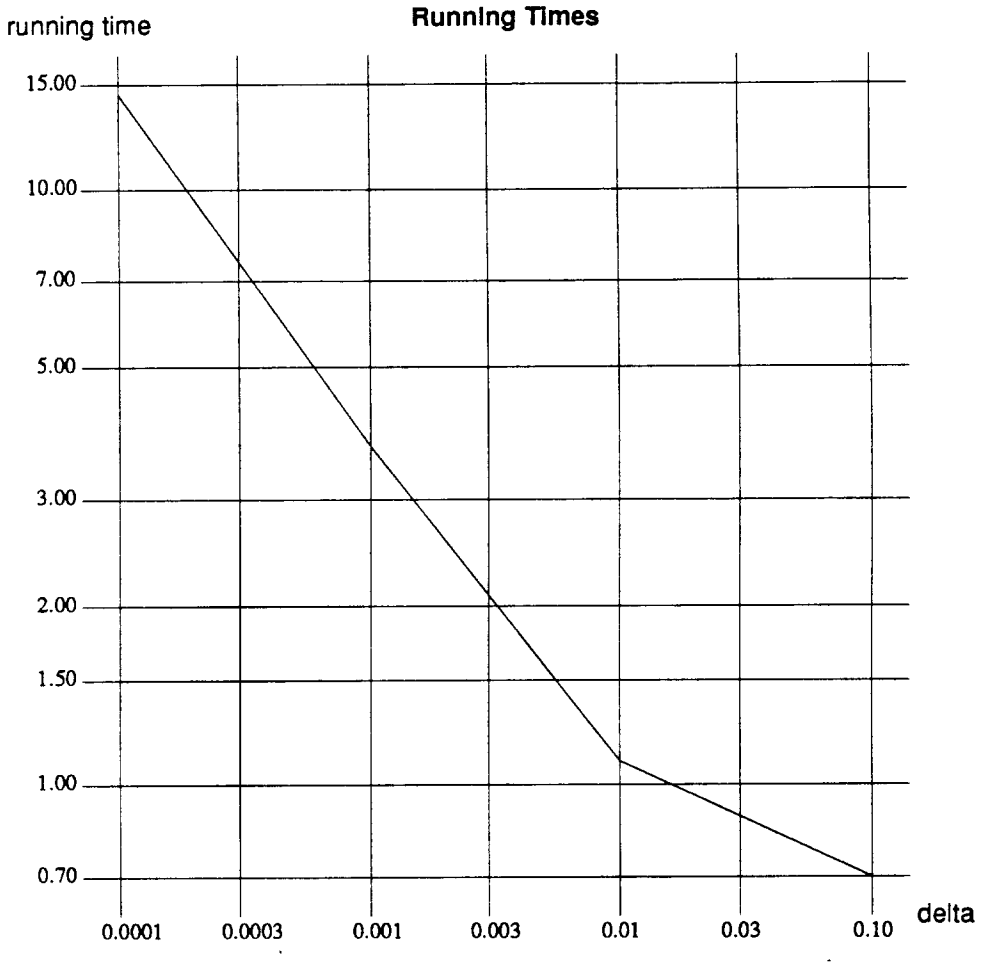


Figure 4.8: Log log plot of the performance of the algorithm for closely spaced Gauss maps.

4.4.6 Faster Gauss Map Separability

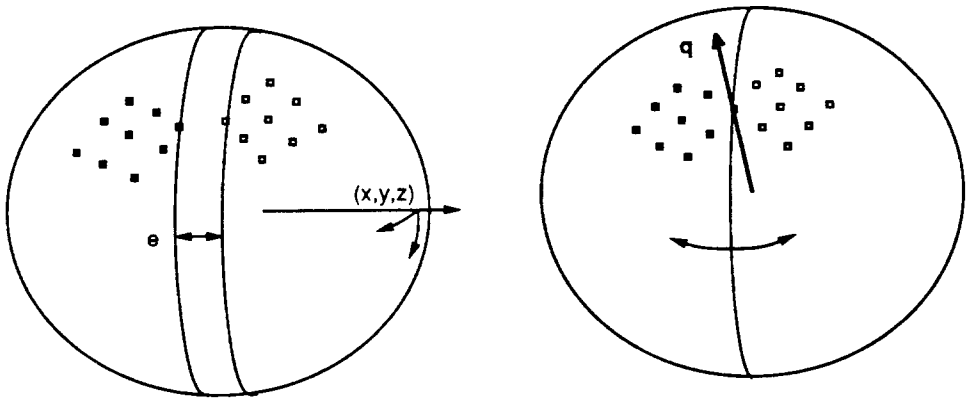


Figure 4.9: Degrees of freedom in separating plane.

If the Gauss maps of two sub-patches have been constructed to intersect at a vector **q** as depicted on the right in Figure 4.9, the intersection test becomes much faster. In that case, the separability test is only a matter of finding a plane containing **q** with the two regions on opposite sides. Such a plane will only have one degree of freedom. Such a vector **q** will be known when the surfaces have been subdivided at a point where they share a common normal. This method of subdivision is explored in section 6.

4.5 Intersecting Curves and Surfaces

Using the tools from the previous sections, an algorithm that reliably computes the points of intersection of a curve and a surface is presented. The scheme for intersecting a curve with a surface is as follows:

Curve Surface Intersect
  if *the curve-surface pair do not intersect except at already discovered intersection points,* **return.**
  if *there are no already discovered points of intersection interior to the curve or surface* **then**
    Perform numerical iteration on the curve-surface pair to find new intersection points.
    if *a degenerate intersection point is found* **then**
      return *a report of the degeneracy.*
    else if *a new intersection point was found* **then**
      Subdivide the curve and the surface at this point
    else
      Subdivide the curve and the surface at their midpoints
    endif
  else
    Subdivide the curve and surface at any points of intersection that are in the interior of the curve or surface.
  endif
  Apply **Curve Surface Intersect** *to each pair.*

A method for determining when the curve and surface do not intersect except at already-discovered intersections is described in Section 4.2. In order to determine if an intersection point has already been discovered, the algorithm must first find all the points common to the curve and surface, and then compare each point to the tentatively "new" point. Determining all the already found intersection points on a curve or surface *efficiently* requires some thought since these points are often discovered by sibling curves and surfaces. A method for doing this is given in 4.9. Since a degenerate (tangential) intersection will

cause **Curve Surface Intersect** to return, the only intersection points that need to be compared are transversal. The precision of the coordinates of such points can be calculated from the numerical conditioning of the intersection and used to test for uniqueness. Thus, the above algorithm, if it terminates, must return all the intersection points each exactly once.

It must now be demonstrated that **Curve Surface Intersect** will terminate. Suppose that it doesn't terminate. In that case, it must recurse indefinitely. At each recursive call, the size of each curve and surface is halved (except if the curve or surface is subdivided at an intersection point). The series of surfaces form a nested sequence of rectangles in surfaces' parameter space. The rectangles will converge to a point in the surfaces' parameter space. Let the image of this point be $\mathbf{p}_s$. Similarly the series of curves form a nested sequence of intervals in the curves' parameter space. Let the image of the limit will be $\mathbf{p}_c$.

If $\mathbf{p}_s \neq \mathbf{p}_c$ then the bounds on the surface must eventually become separable from the bounds on the curve causing the recursion to terminate. If $\mathbf{p}_s = \mathbf{p}_c$ consider the tangent direction of curve $\mathbf{T}_c$ and the normal to the surface $\mathbf{n}_s$.

If $\mathbf{T}_c \cdot \mathbf{n}_s \neq 0$ then the intersection is transversal. Since the numerical iteration is being given initial points arbitrarily close to the solution and the Jacobian is non-singular at the solution, by Kantorovich's Theorem ([51] page 115) the iteration must eventually converge.

If $\mathbf{T}_c \cdot \mathbf{n}_s = 0$ then the curve and surface intersect tangentially. Again, the numerical iteration is being given initial points arbitrarily close to the intersection, but in this case the Jacobian used is singular at the intersection and arbitrarily close to singular for the sequence of initial points. By examining the singular value decomposition of the Jacobian, a singular value can be observed becoming arbitrarily small. Eventually the algorithm will make the determination that the Jacobian is close enough to singular and will report this as a degeneracy terminating the recursion.

4.6 Finding Tangent Directions at a Non-singular point

A non-singular curve of intersection between two patches can be oriented by the cross product of the normals to the two surfaces that define it. To determine if this curve is entering or exiting one of these patches, it is necessary to compute the tangent to the

intersection curve as a linear combination of the partial derivatives of each surface. This is a delicate operation, for if the algorithm calculates that a curve is entering a patch when it is in fact exiting the patch, it will draw the conclusion that a nonsensical intersection has been encountered and will terminate.

The following obvious, but not so accurate method has been presented for calculating the tangent directions [10]. For each surface, obtain the normal by calculating the cross-product of the partial derivatives. To calculate the tangent direction $\mathbf{T}$ in $\mathbf{R}^3$, calculate the cross product of the normals. For the surface $\mathbf{F}(s,t)$ solve the least squares problem

$$s'\mathbf{F}_s + t'\mathbf{F}_t = \mathbf{T} \tag{4.52}$$

and similarly for the surface $\mathbf{G}$. Note that 4.52 is a system of three equations in two unknowns s' and t'.

Alternatively, one can pose the problem as follows. One wishes to find the direction (s',t') in the parameter space of $\mathbf{F}(s,t)$ and the direction (u',v') in the parameter space of $\mathbf{G}(u,v)$ so that

$$s'\mathbf{F}_s + t'\mathbf{F}_t = u'\mathbf{G}_u + v'\mathbf{G}_v \tag{4.53}$$

Note that such tangent direction will automatically be in the direction of the curve tangent. One can pose this as a matrix problem

$$\begin{bmatrix} | & | & | & | \\ \mathbf{F}_s & \mathbf{F}_t & -\mathbf{G}_u & -\mathbf{G}_v \\ | & | & | & | \end{bmatrix} \mathbf{S}^T = 0 \tag{4.54}$$

where $\mathbf{S} = [s', t', u', v']$. The matrix above is 3×4 and one is looking for a fourth row that is orthogonal to the first three. This can be very accurately solved using an SVD algorithm [33,80].

4.7 Finding Tangent Directions at a Singularity

If one is interested in the tangent directions at a singularity, the above matrix will be rank two since the four vectors $\mathbf{F}_s$, $\mathbf{F}_t$, $\mathbf{G}_u$, and $\mathbf{G}_v$ will lie in the tangent plane at the singularity (a two-dimensional subspace). Thus, there will be a two-dimensional subspace of parameter space tangent direction pairs such that moving in corresponding directions in both surfaces, one will remain in both surfaces to the first order. The surfaces being only

two-dimensional manifolds to begin with, this indicates that one may move in *any* direction and remain in both surfaces to the first order.

Moving along an intersection curve one remains in both surfaces to an arbitrary order. Constraining the tangent directions to remain in the surface to the second order will determine the correct tangent directions for a large class of singularities.

Let (s', t') be a tangent direction in the surface $\mathbf{F}(s,t)$ and let (u', v') be a tangent direction in the surface $\mathbf{G}(u,v)$. First, one wants these two tangent directions to agree to the second order. Let w be the parameter of the intersection curve. Its motion is described by

$$(s(w), t(w)) = (s + s'w + s''w/2, t + t'w + t''w/2) \tag{4.55}$$

and similarly for (u, v). The first order constraint is:

$$\begin{bmatrix} | & | \\ \mathbf{G}_u & \mathbf{G}_v \\ | & | \end{bmatrix} \begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} | & | \\ \mathbf{F}_s & \mathbf{F}_t \\ | & | \end{bmatrix} \begin{bmatrix} s' \\ t' \end{bmatrix} \tag{4.56}$$

Let A be the 2 x 2 matrix that maps a (s', t') to a (u', v') so that equation 4.56 holds, i.e. the $\mathbf{R}^3$ directions are the same:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = A \begin{bmatrix} s' \\ t' \end{bmatrix}. \tag{4.57}$$

Combining 4.56 and 4.57:

$$\begin{bmatrix} | & | \\ \mathbf{G}_u & \mathbf{G}_v \\ | & | \end{bmatrix} = \begin{bmatrix} | & | \\ \mathbf{F}_s & \mathbf{F}_t \\ | & | \end{bmatrix} A \tag{4.58}$$

The matrix A can be found using standard least squares methods. Again, it is reasonable to solve this system of equations since $\mathbf{F}_s$, $\mathbf{F}_t$, $\mathbf{G}_u$, and $\mathbf{G}_v$ all lie in a two-dimensional subspace. The second order constraint is

$$\begin{aligned} &\mathbf{F} + (s'\mathbf{F}_s + t'\mathbf{F}_t)w + (s'^2\mathbf{F}_{ss} + t'^2\mathbf{F}_{tt} + 2s't'\mathbf{F}_{st} + s''\mathbf{F}_s + t''\mathbf{F}_t)w^2 \\ &= \mathbf{G} + (u'\mathbf{G}_u + u'\mathbf{G}_u)w + (u'^2\mathbf{G}_{uu} + v'^2\mathbf{G}_{vv} + 2u'v'\mathbf{G}_{uv} + u''\mathbf{G}_u + v''\mathbf{G}_v)w^2 \end{aligned} \tag{4.59}$$

Applying 4.56 one is left with:

$$\begin{aligned} &s'^2\mathbf{F}_{ss} + t'^2\mathbf{F}_{tt} + 2s't'\mathbf{F}_{st} + s''\mathbf{F}_s + t''\mathbf{F}_t \\ &= u'^2\mathbf{G}_{uu} + v'^2\mathbf{G}_{vv} + 2u'v'\mathbf{G}_{uv} + u''\mathbf{G}_u + v''\mathbf{G}_v \end{aligned} \tag{4.60}$$

To make this equality hold in three dimensions, it suffices to make it hold both perpendicular to the tangent plane and in the tangent plane. Let $\mathbf{n}$ be the normal at the singularity

$$\begin{aligned} &\mathbf{n} \cdot (s'^2\mathbf{F}_{ss} + t'^2\mathbf{F}_{tt} + s't'\mathbf{F}_{st} + s''\mathbf{F}_s + t''\mathbf{F}_t) \\ &= \mathbf{n} \cdot (u'^2\mathbf{G}_{uu} + v'^2\mathbf{G}_{vv} + u'v'\mathbf{G}_{uv} + u''\mathbf{G}_u + v''\mathbf{G}_v) \end{aligned} \tag{4.61}$$

Note that $\mathbf{n}$ is perpendicular to each of $\mathbf{F}_s$, $\mathbf{F}_t$, $\mathbf{G}_u$, and $\mathbf{G}_v$ so that

$$\begin{aligned} &\mathbf{n} \cdot (s'^2\mathbf{F}_{ss} + t'^2\mathbf{F}_{tt} + s't'\mathbf{F}_{st}) \\ &= \mathbf{n} \cdot (u'^2\mathbf{G}_{uu} + v'^2\mathbf{G}_{vv} + u'v'\mathbf{G}_{uv}) \end{aligned} \tag{4.62}$$

This equation will determine the tangent directions. In the perpendicular space, equation 4.60 becomes 2 equations in 4 unknowns and thus can be satisfied. Equation 4.62 can be written in matrix form

$$\begin{bmatrix} s' \\ t' \end{bmatrix}^T (Q_{\mathbf{F}} - A^T Q_{\mathbf{G}} A) \begin{bmatrix} s' \\ t' \end{bmatrix} = 0 \tag{4.63}$$

where

$$Q_{\mathbf{F}} = \begin{bmatrix} \mathbf{n} \cdot \mathbf{F}_{ss} & \mathbf{n} \cdot \mathbf{F}_{st} \\ \mathbf{n} \cdot \mathbf{F}_{st} & \mathbf{n} \cdot \mathbf{F}_{tt} \end{bmatrix} \tag{4.64}$$

and similarly for $\mathbf{G}$. The matrix $M = (Q_{\mathbf{F}} - A^T Q_{\mathbf{G}} A)$ gives us a quadratic equation determining the tangent directions. If the entire matrix is zero, then the surfaces agree to the second order or perhaps higher. The determinant of the matrix is the discriminant of the quadratic equation. If it is positive then there are two distinct intersection directions. If it is negative then the two surfaces intersect at an isolated point. If it is zero then the surfaces intersect (at least locally) in a double curve. This could be a cusp, tachnode or a point on a double curve [37] as depicted in 4.10.

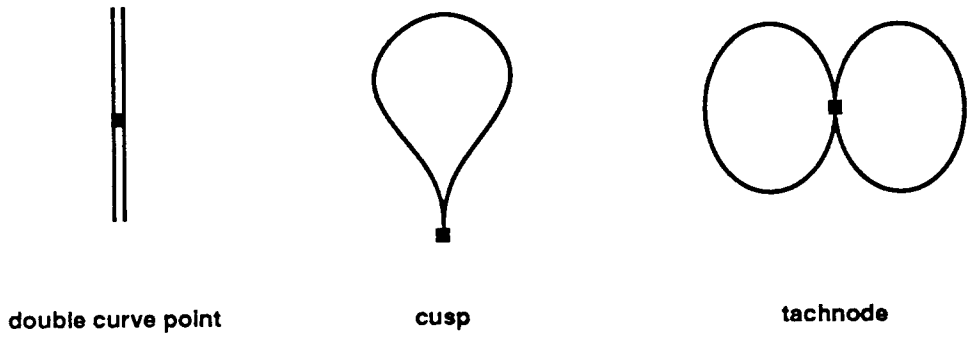


Figure 4.10: Some curve singularities.

4.8 Intersecting Three Surfaces

The scheme for intersecting three surfaces is as follows:

Three Surface Intersect

if *the three surfaces do not intersect except at already discovered intersection points,* **return.**

if *there are no already discovered points of intersection interior to any of the surfaces* **then**

Perform numerical iteration on the surfaces to find new intersection points.

if *a new intersection point was found* **then**

Subdivide the surfaces at this point

else

Subdivide the surfaces in some way

endif

else

Subdivide the surfaces at any points of intersection that are in the interior of some surface.

endif

Perform **Three Surface Intersect** *on each surface triple*

As in Curve Surface Intersect, there is only one way for the above algorithm to return, when the surfaces do not intersect except at intersection points already discovered.

Thus, it must discover all the intersection points. A method for performing this step is described in Section 4.2. Again, the algorithm, if it terminates, must return all the intersection points each exactly once.

4.9 Managing Intersection Points

Throughout the intersection algorithm, it is necessary to determine the set of already found intersection points that lie on a curve or surface. This is mostly necessary to ensure correct connectivity between intersection curve pieces that arise from different sub-surface pair intersections. Since intersection points must be checked for uniqueness, the number of queries is at least as large as the number of updates, and is generally much larger.

For surfaces in general, an intersection point contains information describing how the point is embedded in the surface and the accuracy of the point's coordinates. For example, in the case of a parametric surface, an intersection point contains the parameter values of the point and tolerance associated with the parameter values. The tolerances are based on the conditioning of the curve-surface intersection calculation and vary from point to point.

The problem is: Given a query rectangle (a sub-patch or isoparametric curve) and a large set of small rectangles (the set of already found intersection points and their tolerances) find all the small rectangles that intersect the large rectangle (the set of intersection points on the sub-patch or isoparametric curve). Whatever solution is used must allow for the addition of intersection points between queries.

In the simplest approach, there is a root surface and all surfaces or curves that are obtained from the root through subdivision maintain a pointer to the root. When an intersection is found on a child curve or surface, it is registered with the root. When the intersection points on a child are desired, all the intersection points of the root are examined and those that lie on the child are returned. A single query clearly takes time proportional to the number of intersections found over the entire duration of the surface intersection algorithm, leading to quadratic overall behavior.

A more sophisticated strategy would be to store the intersection points in a 2-D dynamic range search structure [86] according to their parameter values. This would lead to logarithmic entry and query times. Overall one would expect that the amount of time

maintaining the structure would be $O((n+k)\lg n)$ where n is the number of intersection points and k is the sum of the sizes of all the query answers. The implementation has shown that this still takes a considerable fraction of the running time of the surface intersection algorithm. More importantly, one cannot search for points according to their individual tolerances. Rather, a global tolerance must be used to expand the query rectangle. Using some of the ideas presented in [25] it should be possible to create a structure that would allow individual tolerances for each point. The implementation of this appears ponderous and an ad hoc method was developed instead.

Because the algorithm proceeds by walking the subdivision trees of each surface simultaneously, the algorithm tends to make many queries in one region of the surface before proceeding to another region. The idea is to maintain a list of all the intersection points in the area of the surface that is being investigated. In this case the list is the answer to the query and additions to the overall list are simply additions to the small list. Work is only done when the algorithm moves to a different part of the surface the list must be updated. In the following the details are given.

All of the intersection points are kept in a linked list. A surface can be either *active* or *inactive*. In the active state, the surface has a pointer **first_point** to an entry in the linked list. By following the linked list in the forward direction from **first_point** all the intersection points on the surface (and no others) are enumerated.

Clearly, in the active state, the surface can find all the intersection points on itself in time proportional to the number of points. In the subdivision tree for a surface, for any leaf, it is always possible to have every surface on the path from root to leaf active. A leaf can be made inactive in constant time. If a leaf's parent is active, the leaf can be made active in time proportional to the number of intersection points on the parent. If a surface is active and none of its children is active then an entry can be made into the list of intersection points on the surface in constant time.

What then is the overall cost of maintaining this structure? If the surface is active, the cost of a query is proportional to the answer size and the cost of an addition is constant. If the cost of activating any surface is charged to the act of creating an intersection points, then the cost of creating an intersection point is proportional to the total number of surfaces that the point is ever in. For most intersection problems a point is contained in roughly $\lg(n)$ surfaces. Thus, logarithmic entry time and constant query time can be achieved. If n is the number of intersection points, it is expected that $O(n\lg n + k)$ time will be spent

dealing with the structure.

Summary

In this chapter the algorithms that are necessary to implement the surface intersection algorithm have been described in detail. The general problem of linear programming and its application to various spatial separation problems has been discussed. Methods for computing bounds on the Gauss map for rational B-spline and Bézier surfaces have been presented. The problem of finding tangent directions at singular and non-singular points has been dealt with. Also, a method for managing intersection points for the intersection algorithm has been described. In the following chapter curve representation which is well suited to this algorithm and to the problem of solid modeling will be presented.

Chapter 5

Representing the Intersection

5.1 Approximation vs. Representation

A large amount of research has been spent devising methods to approximate general curves, such as intersection curves, with specific well-understood curves such as cubic splines [20]. There are two good reasons to perform such an approximation. First, if all the curves used in a program are represented in the same format, the programmer's task is greatly simplified. Second, if the curve will be interrogated (evaluated, bounded) often, it may be faster to spend time initially to compute the approximate curve to speed up subsequent evaluation.

However, in the case of solid modeling the accuracy requirements outweigh the speed requirements. That is, the curve must be represented with an error which is so small that to approximate the curve with a parametric polynomial curve would require an excessive amount of data.

The alternative is to store a small amount of data from which the intersection curve can be quickly evaluated. This data, together with a small amount of auxiliary data, can be made to look, from a functional standpoint, just like a parametric polynomial curve. Those familiar with object oriented programming will recognize this as a curve object which happens *not* to be implemented with a parametric polynomial curve. In this chapter, it is described how intersection curves can be made into such objects.

5.2 The Curve Object

The following is a list of operations/enquiries that must be supplied by a curve object to a surface intersection algorithm.

Evaluate It should be possible to evaluate the curve at a parameter value.

Subdivide It should be possible to subdivide the curve at a specified point on the curve.

Bound It should be possible to compute a set of points inside of whose convex hull the curve is guaranteed to lie. Additionally, given a point on the curve, it should be possible to construct bounds on the spherical projection of the curve from the point (see Section 4.3).

Intersect with Surface It should be possible to intersect the curve with a surface.

5.3 The Exact Intersection Curve

It is proposed that the intersection curve of the surfaces be implemented by an object that stores the following data.

sub-surface #1 Portion of first intersecting surface.

sub-surface #2 Portion of second intersecting surface.

parameterization vector Vector, $\mathbf{P}$, determining the parametrization of the curve (see Equation 3.15).

parameter curve #1 Curve in parameter space of surface #1.

parameter curve #2 Curve in parameter space of surface #2.

$\mathbf{p}_0$ Start point of the curve.

$\mathbf{p}_1$ End point of the curve.

5.3.1 Evaluation

To evaluate the surface at a parameter, t, one intersects the two defining surfaces with the plane $\mathbf{x} \cdot \mathbf{P} = t$. The start and end parameters are $t_0 = \mathbf{P} \cdot \mathbf{p}_0$ and $t_1 = \mathbf{P} \cdot \mathbf{p}_1$,

respectively. The intersection is found by evaluating the two parameter space curves and using these values as starting points for Newton iteration to solve the intersection problem. Note once an intersection point is found, one does need to verify that there are no others. This is guaranteed by the representation.

5.3.2 Subdivision

To subdivide the intersection curve at a point on it, one merely needs to subdivide one or the other or both of the defining surfaces in such a way that part of the intersection lies on one sub-surface and part on another. After the subdivision all pairs of sub-patches should be checked for intersections and each pair containing an intersection should be added to the list of curves that are returned by this function.

5.3.3 Bounding

A valid but crude bound on the intersection curve is the bound on the two surfaces. This, however can be much larger than the intersection curve, even when the curve is very small. To obviate this, bounds should be estimated on the extent of the intersection curve in the surfaces, and then the surfaces should be trimmed to this bound. If the portions of the surface outside of the bound do not intersect one another then the estimate was valid and the bounds are correct. If the portions do intersect then the estimate was invalid and needs to be repeated.

5.3.4 Intersecting with a surface

This capability may seem somewhat artificial. That is, from the previous functions provided by the exact intersection curve, it is possible to construct a curve/surface intersection algorithm. On the other hand, adding this function allows the intersection algorithm to be constructed in a more symmetric fashion. That is, the intersection that one seeks is the intersection of three surfaces: the two surfaces defining the curve and the surface with which one wishes intersect the curve. In this problem, no surface plays a distinguishable role. An algorithm to accomplish this is given in Section 4.8

Chapter 6

Singularities

Now that it has been shown how to implement the operations needed to support the basic surface intersection algorithm, the problem of singular points of intersection is addressed.

Consider the intersection of a bicubic patch and a plane depicted in Figure 6.1. The intersection consists of two sets of three parallel lines. The lines intersect in singular points, that is, points where the surfaces intersect and the surface normals are parallel. (see [37], page 32), If one were to give this geometry to the intersection algorithm so far described, it would recurse until it has found the singular points to a sufficient accuracy. Unfortunately it would be very slow: it would be similar to finding the solution by bi-section. The difference is that every time size of the problem is halved, not two but rather sixteen sub-problems are generated. Even in cases where the surfaces do not contain any singular points of intersection, but nearly do, speed can be a problem as will be seen in Section 6.3.

Thus, it seems fruitful to investigate whether the singularities can be found via a faster method. *Before embarking on that task it should be pointed out again that any acceleration technique for finding singularities will neither contribute to nor diminish the robustness of the algorithm. Such a technique can only affect its speed.*

To find the singular points quickly, it seems natural to employ a Newton iteration type scheme. To do this, one must first characterize the singular points mathematically. This can be done as follows:

$$\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t) \cdot \mathbf{G}_u(u,v) = 0 \tag{6.1}$$

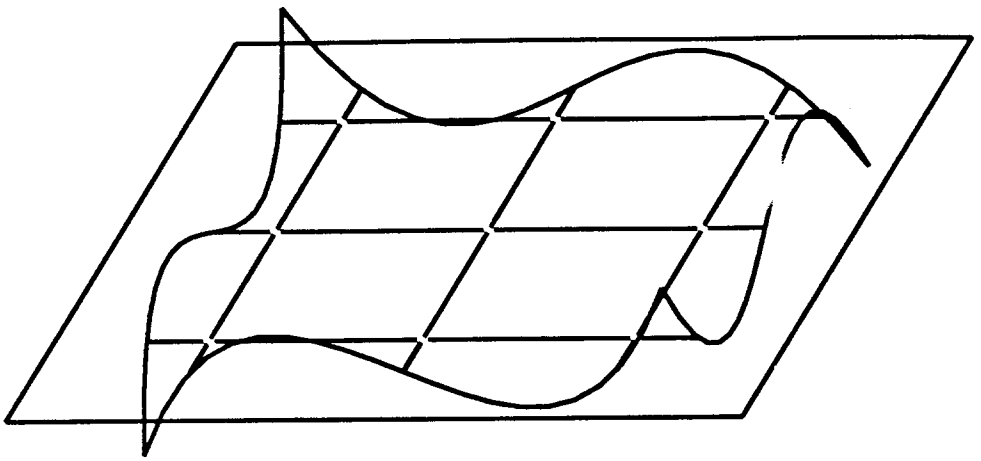


Figure 6.1: Intersection consisting of 6 lines.

$$\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t) \cdot \mathbf{G}_v(u,v) = 0 \tag{6.2}$$

$$\mathbf{F}(s,t) - \mathbf{G}(u,v) = \mathbf{0} \tag{6.3}$$

Equation 6.3 (really three equations) ensures that the surfaces intersect at the point in question, and equations 6.1 and 6.2 ensure that the surfaces have the same normal at the point. Unfortunately, this is a system of five equations in four unknowns. The Jacobian associated with it will be rank deficient simply because it is not square. This is expected because generically surfaces do not intersect in singular points.

6.1 Auxiliary Equations

To overcome this an auxiliary system of equations whose solutions are a superset of the set of singular points is used. The solutions to the auxiliary equations can then be checked to determine if they are also singular points. One set of auxiliary equations that can be used is

$$\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t) \cdot \mathbf{G}_u(u,v) = 0 \tag{6.4}$$

$$\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t) \cdot \mathbf{G}_v(u,v) = 0 \tag{6.5}$$

$$(\mathbf{F}(s,t) - \mathbf{G}(u,v)) \cdot \mathbf{F}_s(s,t) = 0 \tag{6.6}$$

$$(\mathbf{F}(s,t) - \mathbf{G}(u,v)) \cdot \mathbf{F}_t(s,t) = 0. \tag{6.7}$$

Note that this is the same set of equations presented before but with the three equations 6.3 being replaced by the two equations 6.6 and 6.7. The component of equation 6.3 in the direction of the surface normal has been disregarded. Thus, the set of solutions to equations 6.1 to 6.2 will be a subset of the solutions to equations 6.4 to 6.7.

Sederberg [94] also proposed to solve this system of equations, but did not present any analysis of the singularity of the Jacobian. Points on the surface that satisfy this set of equations have been called *magic points* in [69] and also *collinear normal points* [58].

If the Jacobian associated with this system of equations is non-singular then given a starting point close enough, the Newton iteration used to solve this system of equations will converge.

Theorem 7 *The Jacobian associated with the system of equations 6.4 to 6.7 will be non-singular at node singularities and at isolated intersections where the surfaces do not share the same fundamental form.*

Proof:

Let

$$H_1 \equiv \mathbf{F}_s \times \mathbf{F}_t \cdot \mathbf{G}_u \tag{6.8}$$

$$H_2 \equiv \mathbf{F}_s \times \mathbf{F}_t \cdot \mathbf{G}_v \tag{6.9}$$

$$H_3 \equiv (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_s \tag{6.10}$$

$$H_4 \equiv (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_t \tag{6.11}$$

and $w_1 \equiv s$, $w_2 \equiv t$, $w_3 \equiv u$, and $w_4 \equiv v$. Consider the Jacobian of the system of equations 6.4 - 6.7:

$$\mathbf{J} = \left[\frac{\partial H_i}{\partial w_j}\right] \tag{6.12}$$

To make this more manageable one can make some simplifying assumptions. First, one can assume that the solution is at $s = t = u = v = 0$. Thus, $\mathbf{F}(0,0) = \mathbf{G}(0,0)$. By performing an orthogonal change of coordinate system one can arrange that $\mathbf{F}_s \cdot \hat{\mathbf{z}} = \mathbf{F}_t \cdot \hat{\mathbf{z}} = \mathbf{G}_u \cdot \hat{\mathbf{z}} = \mathbf{G}_v \cdot \hat{\mathbf{z}} = 0$ without changing the Jacobian. The next objective is to make $\mathbf{F}_t = \hat{\mathbf{y}}$ and $\mathbf{F}_s = \hat{\mathbf{x}}$. This can be accomplished by reparametrizing $\mathbf{F}$ with the variables s' and t'.

$$s(s', t') = as' + bt' \tag{6.13}$$

and

$$t(s',t') = cs' + dt' \tag{6.14}$$

Firstly, this changes the functions H_i:

$$\begin{aligned} H_1' &= \mathbf{F}_{s'} \times \mathbf{F}_{t'} \cdot \mathbf{G}_u && (6.15)\\ &= (\mathbf{F}_s a + \mathbf{F}_t b) \times (\mathbf{F}_s c + \mathbf{F}_t d) \cdot \mathbf{G}_u && (6.16)\\ &= (\mathbf{F}_s \times \mathbf{F}_t)(ad - bc) \cdot \mathbf{G}_u && (6.17)\\ &= H_1(ad - bc) && (6.18) \end{aligned}$$

and

$$\begin{aligned} H_2' &= \mathbf{F}_{s'} \times \mathbf{F}_{t'} \cdot \mathbf{G}_v && (6.19)\\ &= (\mathbf{F}_s a + \mathbf{F}_t b) \times (\mathbf{F}_s c + \mathbf{F}_t d) \cdot \mathbf{G}_v && (6.20)\\ &= (\mathbf{F}_s \times \mathbf{F}_t)(ad - bc) \cdot \mathbf{G}_v && (6.21)\\ &= H_2(ad - bc) && (6.22) \end{aligned}$$

and

$$\begin{aligned} H_3' &= (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_{s'} && (6.23)\\ &= (\mathbf{F} - \mathbf{G}) \cdot (\mathbf{F}_s a + \mathbf{F}_t c) && (6.24)\\ &= H_3 a + H_4 c && (6.25) \end{aligned}$$

and

$$\begin{aligned} H_4' &= (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_{t'} && (6.26)\\ &= (\mathbf{F} - \mathbf{G}) \cdot (\mathbf{F}_s b + \mathbf{F}_t d) && (6.27)\\ &= H_3 b + H_4 d && (6.28) \end{aligned}$$

So that

$$\left[\frac{\partial H_i'}{\partial w_j}\right] = \begin{bmatrix} ad-bc & 0 & 0 & 0 \\ 0 & ad-bc & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} \left[\frac{\partial H_i}{\partial w_j}\right] \tag{6.29}$$

The Jacobian also changes as a result of the reparametrization of the H_i.

$$\frac{\partial H_i'}{\partial w_1'} = \frac{\partial H_i'}{\partial s'} \tag{6.30}$$

$$= \frac{\partial H_i'}{\partial s}\frac{\partial s}{\partial s'} + \frac{\partial H_i'}{\partial t}\frac{\partial t}{\partial s'} \tag{6.31}$$

$$= \frac{\partial H_i'}{\partial s}a + \frac{\partial H_i'}{\partial t}c \tag{6.32}$$

$$= \frac{\partial H_i'}{\partial w_1}a + \frac{\partial H_i'}{\partial w_2}c \tag{6.33}$$

and

$$\frac{\partial H_i'}{\partial w_2'} = \frac{\partial H_i'}{\partial t'} \tag{6.34}$$

$$= \frac{\partial H_i'}{\partial s}\frac{\partial s}{\partial t'} + \frac{\partial H_i'}{\partial t}\frac{\partial t}{\partial t'} \tag{6.35}$$

$$= \frac{\partial H_i'}{\partial s}b + \frac{\partial H_i'}{\partial t}d \tag{6.36}$$

$$= \frac{\partial H_i'}{\partial w_1}b + \frac{\partial H_i'}{\partial w_2}d \tag{6.37}$$

so that

$$\left[\frac{\partial H_i'}{\partial w_j'}\right] = \begin{bmatrix} ad-bc & 0 & 0 & 0 \\ 0 & ad-bc & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} \left[\frac{\partial H_i}{\partial w_j}\right] \begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.38}$$

Since the matrices that modify $\mathbf{J}$ in equation 6.38 are non-singular, $\mathbf{J}'$ will be singular if and only if $\mathbf{J}$ is.

In a similar way, one can reparametrize $\mathbf{G}$ with variables u' and v' so that $\mathbf{G}_{u'} = \hat{\mathbf{x}}$ and $\mathbf{G}_{v'} = \hat{\mathbf{y}}$. It can again be shown that this will not change the rank of the Jacobian. Let the old coordinates be related to the new according to

$$u(u', v') = au' + bv' \tag{6.39}$$

and

$$v(u', v') = cu' + dv'. \tag{6.40}$$

The partials with respect to the new parameters are

$$\mathbf{G}_{u'} = \mathbf{G}_u a + \mathbf{G}_v c \tag{6.41}$$

$$\mathbf{G}_{v'} = \mathbf{G}_u b + \mathbf{G}_v d \tag{6.42}$$

The Jacobian $\mathbf{J}'$ with respect to the new variables u' and v' is in the following relationship to the Jacobian $\mathbf{J}$ with respect to the old variables u and v:

$$\mathbf{J}' = \begin{bmatrix} a & c & 0 & 0 \\ b & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{J} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix}. \tag{6.43}$$

The proof of this is similar to that for the reparametrization of $\mathbf{F}$ and is thus omitted. It can be seen that the new Jacobian will be singular if and only if the original Jacobian was.

If one expands the Jacobian:

$$\begin{bmatrix} (\mathbf{F}_{ss} \times \mathbf{F}_t + \mathbf{F}_s \times \mathbf{F}_{st}) \cdot \mathbf{G}_u & (\mathbf{F}_{st} \times \mathbf{F}_t + \mathbf{F}_s \times \mathbf{F}_{tt}) \cdot \mathbf{G}_u & \mathbf{F}_s \times \mathbf{F}_t \cdot \mathbf{G}_{uu} & \mathbf{F}_s \times \mathbf{F}_t \cdot \mathbf{G}_{uv} \\ (\mathbf{F}_{ss} \times \mathbf{F}_t + \mathbf{F}_s \times \mathbf{F}_{st}) \cdot \mathbf{G}_v & (\mathbf{F}_{st} \times \mathbf{F}_t + \mathbf{F}_s \times \mathbf{F}_{tt}) \cdot \mathbf{G}_v & \mathbf{F}_s \times \mathbf{F}_t \cdot \mathbf{G}_{uv} & \mathbf{F}_s \times \mathbf{F}_t \cdot \mathbf{G}_{vv} \\ \mathbf{F}_s \cdot \mathbf{F}_s + (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_{ss} & \mathbf{F}_t \cdot \mathbf{F}_s + (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_{st} & -\mathbf{G}_u \cdot \mathbf{F}_s & -\mathbf{G}_v \cdot \mathbf{F}_s \\ \mathbf{F}_s \cdot \mathbf{F}_t + (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_{st} & \mathbf{F}_t \cdot \mathbf{F}_t + (\mathbf{F} - \mathbf{G}) \cdot \mathbf{F}_{tt} & -\mathbf{G}_u \cdot \mathbf{F}_t & -\mathbf{G}_v \cdot \mathbf{F}_t \end{bmatrix} \tag{6.44}$$

it is observed that the second order partials are only involved in inner products with vectors parallel to $\hat{\mathbf{z}}$. Thus, one can assume that the surfaces are in the form:

$$\mathbf{F}(s,t) = (s, t, as^2/2 + bst + ct^2/2) \tag{6.45}$$

$$\mathbf{G}(u,v) = (u, v, du^2/2 + euv + fv^2/2) \tag{6.46}$$

The derivatives are as follows:

$$\mathbf{F}_s = \hat{\mathbf{x}} \quad \mathbf{F}_t = \hat{\mathbf{y}} \quad \mathbf{F}_{ss} = a\hat{\mathbf{z}} \quad \mathbf{F}_{st} = b\hat{\mathbf{z}} \quad \mathbf{F}_{tt} = c\hat{\mathbf{z}} \tag{6.47}$$

$$\mathbf{G}_u = \hat{\mathbf{x}} \quad \mathbf{G}_v = \hat{\mathbf{y}} \quad \mathbf{G}_{uu} = d\hat{\mathbf{z}} \quad \mathbf{G}_{uv} = e\hat{\mathbf{z}} \quad \mathbf{G}_{vv} = f\hat{\mathbf{z}} \tag{6.48}$$

Substituting them into expression 6.44 yields

$$\mathbf{J} = \begin{bmatrix} (a\hat{\mathbf{z}} \times \hat{\mathbf{y}} + \hat{\mathbf{x}} \times \dot{o}\hat{\mathbf{z}}) \cdot \hat{\mathbf{x}} & (b\hat{\mathbf{z}} \times \hat{\mathbf{y}} + \hat{\mathbf{x}} \times c\hat{\mathbf{z}}) \cdot \hat{\mathbf{x}} & \hat{\mathbf{x}} \times \hat{\mathbf{y}} \cdot d\hat{\mathbf{z}} & \hat{\mathbf{x}} \times \hat{\mathbf{y}} \cdot e\hat{\mathbf{z}} \\ (a\hat{\mathbf{z}} \times \hat{\mathbf{y}} + \hat{\mathbf{x}} \times b\hat{\mathbf{z}}) \cdot \hat{\mathbf{y}} & (b\hat{\mathbf{z}} \times \hat{\mathbf{y}} + \hat{\mathbf{x}} \times c\hat{\mathbf{z}}) \cdot \hat{\mathbf{y}} & \hat{\mathbf{x}} \times \hat{\mathbf{y}} \cdot e\hat{\mathbf{z}} & \hat{\mathbf{x}} \times \hat{\mathbf{y}} \cdot f\hat{\mathbf{z}} \\ \hat{\mathbf{x}} \cdot \hat{\mathbf{x}} & \hat{\mathbf{y}} \cdot \hat{\mathbf{x}} & -\hat{\mathbf{x}} \cdot \hat{\mathbf{x}} & -\hat{\mathbf{y}} \cdot \hat{\mathbf{x}} \\ \hat{\mathbf{x}} \cdot \hat{\mathbf{y}} & \hat{\mathbf{y}} \cdot \hat{\mathbf{y}} & -\hat{\mathbf{x}} \cdot \hat{\mathbf{y}} & -\hat{\mathbf{y}} \cdot \hat{\mathbf{y}} \end{bmatrix} \tag{6.49}$$

The determinant of the Jacobian is then

$$\begin{vmatrix} -a & -b & d & e \\ -b & -c & e & f \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{vmatrix} = (d-a)(f-c) - (e-b)^2 \tag{6.50}$$

The determinant 6.50 is zero under two circumstances. The first is when $d = a$, $f = c$ and $e = b$, i.e., when the surfaces have the same second order description. In that case, the surfaces cannot intersect in a simple node singularity. Next, consider the variables $\alpha = a-d$, $\beta = b - e$ and $\gamma = c - f$. The determinant will be zero if $\alpha\gamma - \beta^2 = 0$. Note that this is exactly the condition that the two surfaces intersect in a double curve (see Section 4.6. ■

The Jacobian can still vanish at a singularity, for example if the surfaces intersect in a double curve or in some other higher order singularity. In these cases, one would like to locate this singularity quickly to identify the degeneracy. This can be done by computing the singular value decomposition of the Jacobian. If a very small singular value is discovered, the pseudo inverse of the matrix is used for the Newton iteration. In this way, the Newton iteration will converge to a degenerate point of intersection. The algorithm will then report the point as a degeneracy and terminate.

6.2 Modified Algorithm

The results of the previous section can be used to derive a faster algorithm:

Intersect
  if *the bounding volumes of the surfaces intersect* **then**
    if *the Gauss maps satisfy the loop detection criterion* **then**
      Intersect Simple Surfaces
    else
      Perform numerical iteration to find the solution to equations (6.4 - 6.7)
      if *a solution is found* **then**
        if *the solution is a node or collineal point* **then**
          subdivide at that point.
        else *if the singularity is a double curve point* **then**
          return *a report of the degeneracy*
        else
          subdivide at the collineal point
        endif
      else
        Subdivide the surface(s) with the largest Gauss map(s)
      endif
      Intersect *all pairs.*
    endif
  endif

If a singular point is found, the surfaces are subdivided so that the sub-patches containing the singularity are very small as depicted in Figure 6.2. The subpatches containing the singularity are not intersected one with the other. Rather, the intersection within the region is assumed to be merely a crossing of two intersection curves. The tangent directions at this point can be calculated as described in 4.7.

If a solution is found to equations 6.4 - 6.7 but $\mathbf{F}(s,t) \neq \mathbf{G}(u,v)$ then each surface is simply subdivided at this point. By doing this, it is known beforehand that the Gauss maps of several surfaces intersect in at least a single point. This reduces the dimension of

the linear program that must be employed to determine if the Gauss maps intersect and thus speeds up the computation. In addition, if one of the surfaces is a plane, then (unless there is another collinear point) the surfaces thus subdivided will pass the loop detection criterion.

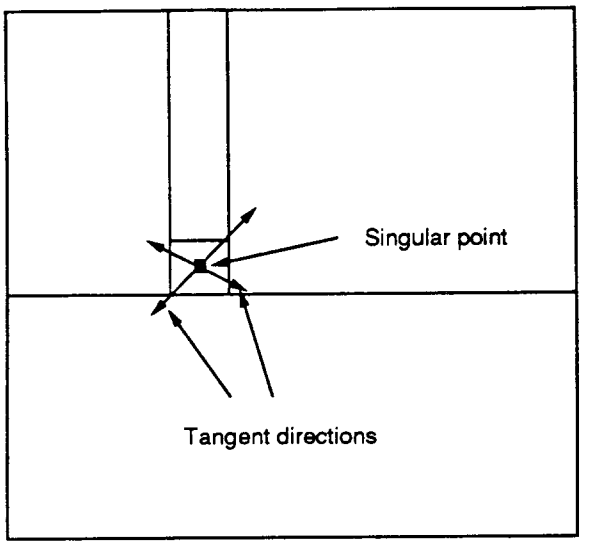


Figure 6.2: Subdivision pattern at a node singularity.

6.3 Application to Example

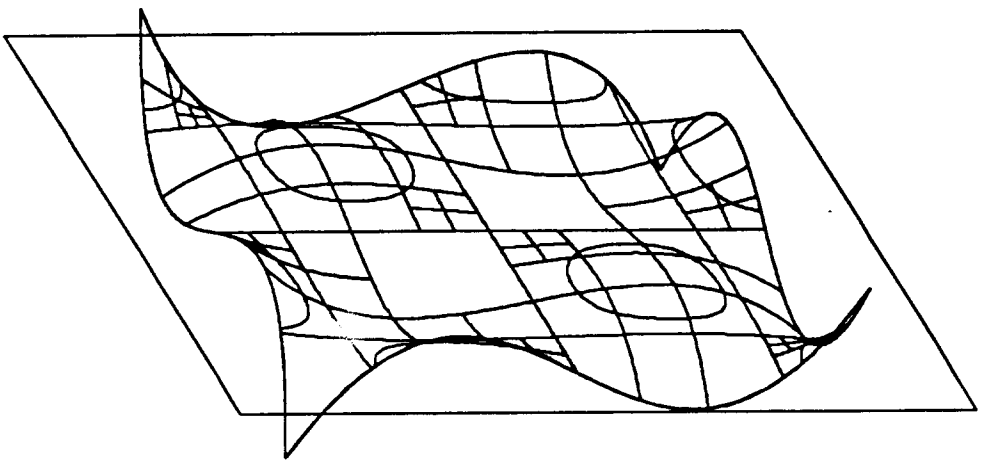


Figure 6.3: Basic algorithm applied to example.

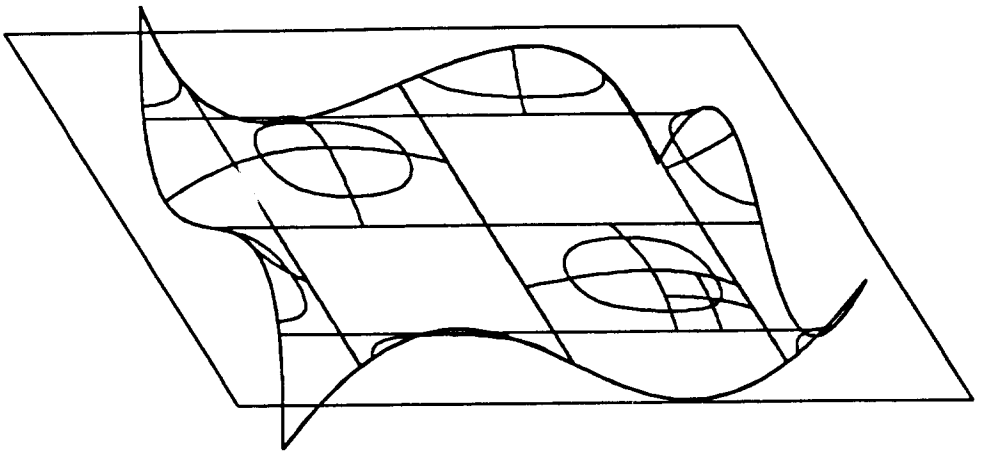


Figure 6.4: Modified algorithm applied to the same example.

In Figure 6.3, a bicubic patch and a plane are intersected using the basic intersection algorithm. It can be seen that a large amount of subdivision takes place before the sub-surfaces pass the loop detection criterion. In Figure 6.4, the intersection problem depicted in Figure 6.3 has been given to the modified algorithm. For this pair of surfaces, there are 13 solutions to equations (6.4) - (6.7). Nine are at the intersection of the two pairs of three lines and the remaining four are at the tops of the two "hills" and the bottoms of the two "valleys". When there was a solution to equations (6.4) - (6.7) in the regions of the surfaces being subdivided, the Newton iteration found it in every case except one. The number of patches in the final subdivision in Figure 6.4 is 31 compared to 88 in Figure 6.3.

As the intersection of the two surfaces becomes closer to singular, the amount of subdivision does not change for the more modified algorithm, but grows for the basic algorithm. In Figures 6.5 and 6.6 the plane has been lowered closer to the singular points. The number of patches remains 31 for Figure 6.6 while it has grown to 110 in Figure 6.5.

In Figure 6.7 the subdivision resulting from the modified surface intersection algorithm, when the intersection actually contains singular points, is shown. In this case, the singularities are discovered and the subdivision pattern depicted in Figure 6.2 is employed.

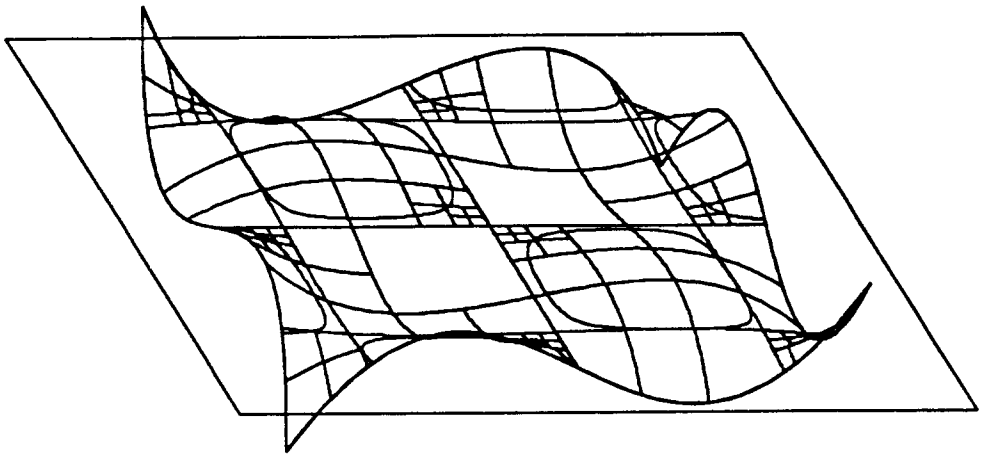


Figure 6.5: The results of basic algorithm when the intersection curve nearly contains a singular point.

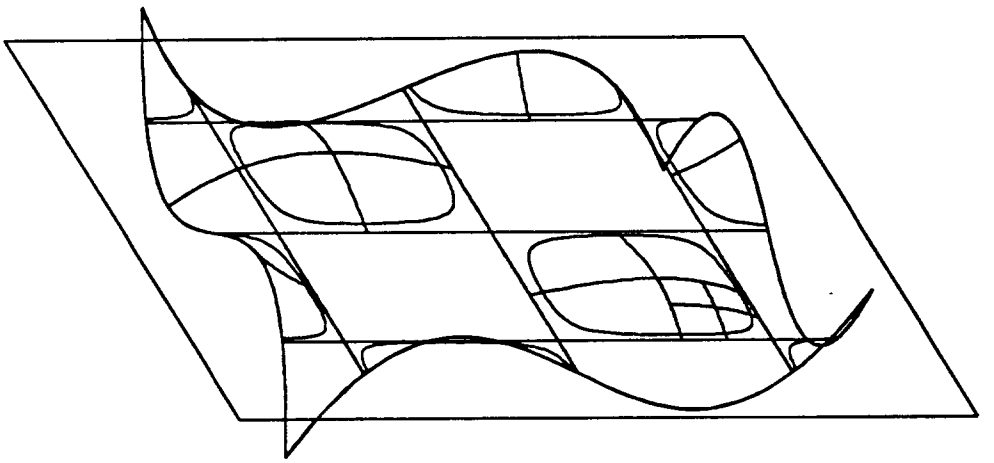


Figure 6.6: Modified algorithm when the surfaces are near the degenerate position.

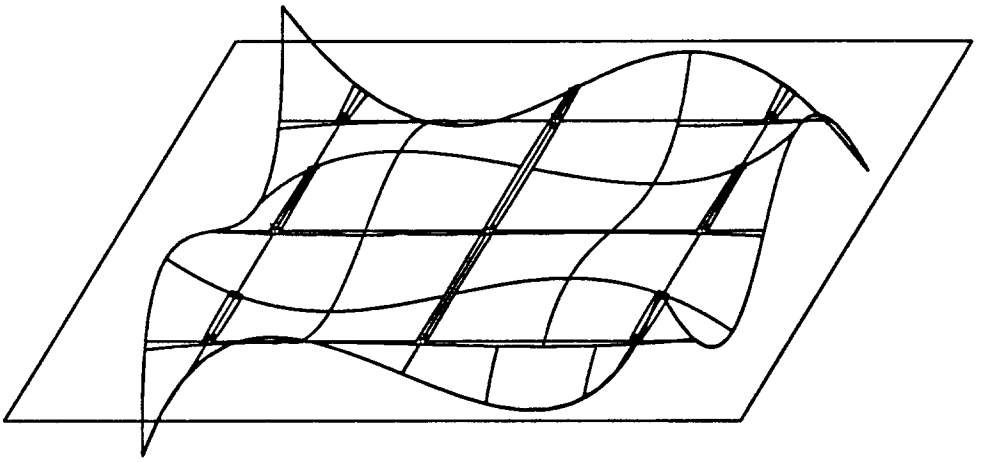


Figure 6.7: The number of patches in the subdivision is moderate even with degenerate intersections.

Chapter 7

Degeneracy

In the description of the algorithm so far, in many cases it has been assumed that the surfaces to be intersected are in general position. Actually, a stronger assumption has been made. It has been assumed that the surfaces are far enough away from any degenerate position that all the floating point comparisons give the same answer as would have been given had the whole algorithm run in infinite precision. If this is the case then the algorithm is guaranteed to return the correct intersection. In practice, this assumption is unworkable: exact degeneracies are often designed into objects used in solid modeling systems. Additionally, near and exact degeneracies can be produced by the surface intersection algorithm itself. Degeneracies designed into the object might include, for example, fillets, rounds and abutting faces. The algorithm's selection of parameter values at which to subdivide the surface can also lead to degeneracies.

It is appropriate when considering degeneracies to consider the context in which the surface intersection algorithm is operating. If the surface intersection problem is degenerate then the solid Boolean operation is also degenerate. If one can determine how the solid operation should proceed in the degenerate case then one can also determine how the surface intersection problem should proceed. In the first section of this chapter, some policies associated with solids in the degenerate position are investigated one that is workable in the context of floating point arithmetic is introduced. In the second section, the implementation in the surface intersection algorithm of a policy consistent with this is described.

7.1 Robustness

Many articles, theses and, books have been written recently that address the problem of designing a robust geometric modeling algorithm and, in particular, a robust solid modeling algorithm [1,29,42,43,45,44,49,54,68,71,88,98,105]. When all of the relevant geometry is in general position then any correct algorithm should function reliably. However, when certain degeneracies occur it is not only unclear how the algorithm should be designed, but it is also unclear how to determine if an algorithm has run correctly. In this section, the problem of correctness in the degenerate position is addressed.

7.1.1 Topological Correctness

Hoffmann, Hopcroft and Karasick consider the correctness of an implementation of an operation on a solid represented by topological data [42]. In their system, a solid is a collection of numerical and topological data. Two solids are identical if there is a one to one correspondence between their numerical and topological data and corresponding numerical data are close in a numerical sense. For such solids correctness is defined as follows:

1. A *model* M is ideal.
2. A rep entation R is real. A representation is a *representation of* a model M if the largest deviation of R from M is less than ϵ, where ϵ is a fixed tolerance.
3. The implementation of a k-ary operation is *correct* if for every input representation R_i, there exists a model M_i such that the following is true:
   (a) The algorithm constructs an output representation R without failing.
   (b) R is a representation of M where $M = op(M_1, \ldots, M_k)$.

Let us refer to this as the *topological* correctness rule. This rule actually tells us nothing about what to do in the degenerate position. Given a model nearly in the degenerate position one can perturb it to one side or the other side of the degeneracy or one can perturb it exactly into the the degenerate position. Any such perturbation will result in valid R_i's. An algorithm is then correct if it mimics the behavior of some ideal algorithm.

The problem with this approach is that for very reasonable input, the algorithm is allowed to produce very unexpected output. Consider the situation depicted in Figure 7.1.

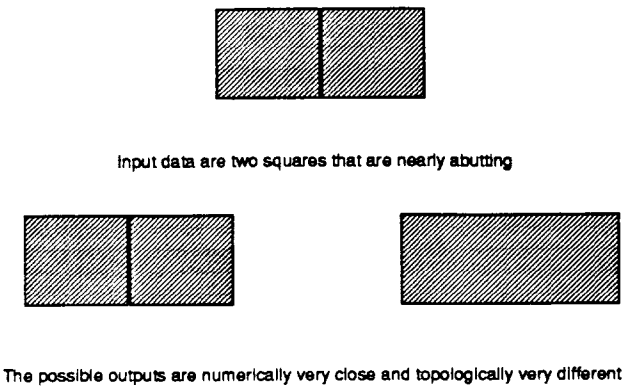


Figure 7.1: Shortcoming of the topological rule.

Two squares are presented to a union algorithm. The numerical data describing the faces that abut one another are identical. However, according to the above criterion the algorithm is allowed to produce either 1) two distinct squares as an output or 2) one rectangle as an output. The "expected" result is a single rectangle, but the expectation is not enforced by the above definition of correctness. Specifically, the following problems can be encountered:

1. An algorithm can produce lower dimensional features: lamina and filaments.

2. An algorithm can produce numerically unstable features: very thin slabs.

The first problem can be skirted by redefining the ideal operation. For most polyhedral modeling algorithms, the ideal operations are not union, intersection and difference, but rather the regularized[1] Boolean operations $\cup^*$, $\cap^*$ and $-^*$.

Even with regularized Boolean operations the second problem remains: the algorithm is allowed to produce output that is arbitrarily close to non-regular. Thus, a second addendum is added to the above correctness rule: The algorithm neither accepts nor outputs *features* that are smaller than a specified tolerance.

More fundamentally, this definition is not so much descriptive as it is prescriptive. That is, one cannot draw conclusions about the output of a correct algorithm short of

[1] A set is regular if it is the closure of its interior.

actually running it. For instance, one cannot determine if a correct algorithm will return a single connected body in Figure 7.1 or two disconnected bodies.

7.1.2 Stereographic Correctness

Hopcroft and Kraft [49] have named the study of "representing physical objects and of manipulating and reasoning about those objects", *stereophenomenology*. The important aspects of solid models are those which correspond to aspects of physical objects. A solid is completely and uniquely defined by its corresponding point set. Having taken this view, the concept of *stereographic correctness* is presented.

What fundamentally does a solid modeling algorithm do? It models certain point sets in $\mathbf{R}^3$ and performs Boolean operations on them. What can be done with point sets? They can be combined via Boolean operations to obtain other point sets and they can be used to discriminate between those points which they contain and those points which they do not contain. Thus, a modeling algorithm should be able to determine whether a point is in a solid or not. Because of numerical rounding, this cannot always be done. Specifically, the boundaries of a solid might be slightly perturbed from the ideal. Thus, the real algorithm only classifies points that are not near the boundary of the solid. For points that are near the boundary of the solid, the algorithm is not obliged to give the correct answer.

A subtle question arises. The boundaries of the solid exempt the modeling algorithm from answering certain questions. To which boundaries is one referring, those of the real model (maintained by the algorithm) or to those of the ideal model? If one choses the the former, one could write a trivially correct modeling algorithm which always returns the answer "inside". In its defense, the algorithm could plead that its internal model had boundaries everywhere. Thus, one is forced to use the latter. Formally then:

1. A *model* M is ideal.

2. A *representation* R is real. A representation R is a *representation of* a model M if every point well within M is in R and every point well outside of M is not in R.

3. The implementation of a k-ary operation is *correct* if, for every input representation R_i, and for every model M_i represented by R_i, the following is true:

   (a) The algorithm constructs an output representation R without failing.

(b) R is a representation of M where $M = op(M_1, \ldots, M_k)$.

First "well within" means inside and not near a boundary and "well outside" means outside and not near a boundary. This will be referred to as the point *stereographic correctness* rule. Here are some important things to observe:

1. Since the algorithm is responsible to produce the correct output for all the inputs that is represents, it is as if the input data has been corrupted even before the algorithm sees it.

2. The algorithm is only exempt from correctly determining the membership of points near the boundary of M. Points near the boundaries of the M_i must be correctly classified if they are a large distance from the boundary of M.

Consider the example depicted in Figure 7.2. Each of the R_i are squares. The coordinates of adjacent sides of the squares are so close that it is not possible to determine if the squares, M_i, intersect, abut, or are disjoint. The top portion of the figure depicts the situation when the squares, M_i, are disjoint and the bottom when they overlap. A modeling algorithm that conforms to topological correctness rule would be allowed to output an R consisting of either one piece or two.

Let us consider the possibilities under the stereographic correctness rule. If the output R consists of two pieces when the M_i overlap one will not only misclassify the test point depicted but will violate the rule since the point is a large distance from any boundary of M. However, if one outputs an R consisting of one piece when the M_i are disjoint, the test pointmay be misclassified but the rule will not be violated since the point is within ϵ of the boundary of M. Thus, the stereographic correctness rule coerces us to take the second choice.

In effect, the rule discourages the algorithm from maintaining *unstable features.* A model is said to have an unstable feature if a point **p** outside (inside) the model becomes inside (outside) the model when the numerical data are perturbed and no point near **p** remains outside (inside).

To illustrate, consider again the squares pictured in 7.1. If the two squares nearly abut and they are modeled as disjoint, then the two boundaries which are very close persist in the representation. A subsequent operation may draw a conclusion that is inconsistent with the squares being disjoint. For instance, one might intersect a horizontal line with the

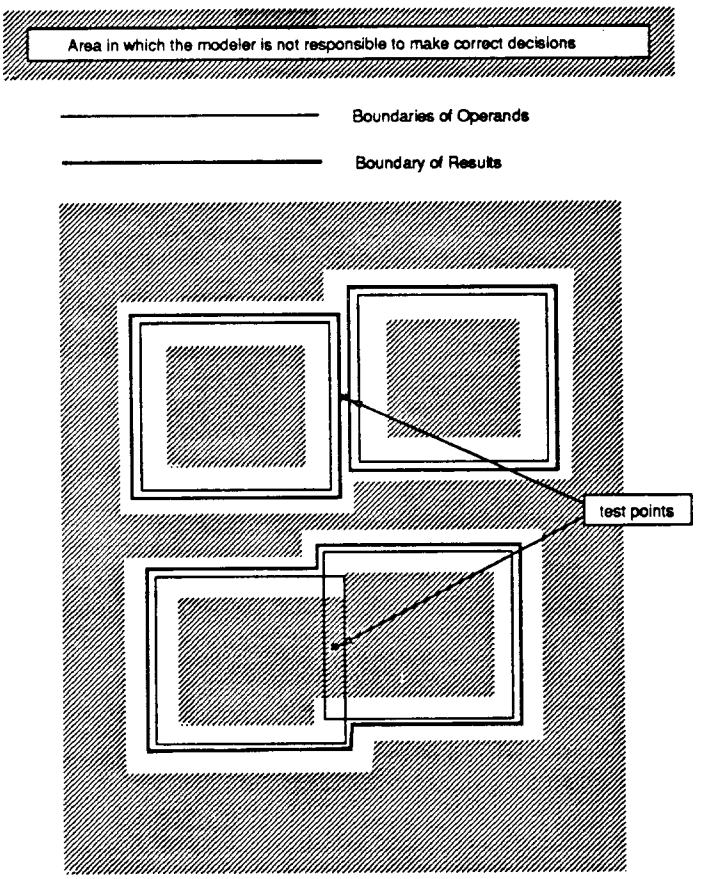


Figure 7.2: Stereographic correctness rule.

boundaries and check the ordering of the points. With the two lines so close together, it is entirely possible that one will record two entries into the object followed by two exits from the solid, which makes no sense at all. If, on the other hand, if one models the squares as overlapping, then the very close lines will be deleted from the boundary of the object, effectively preventing them from causing any trouble in the future.

7.1.3 Comparison of the two Rules

Let us examine the differences between the topological rule and the stereographic rule as the distance between the two squares varies. The situation is depicted in Figure 7.3. By necessity, both definitions have a grey area, where the algorithm is free to make an arbitrary decision. For the topological rule, this is the area near where the squares abut. For the stereographic rule, the point is where the squares are far enough apart that the machine precision can guarantee that the squares really don't touch and are not so far away that the algorithm becomes responsible to correctly classify the points in the gap.

7.2 Application to Surface Intersection

For ease of discussion and depiction, let us consider the two-dimensional analogue of 3-D solid modeling, call it area modeling. In area modeling, one models subsets of the plane by their boundaries. When computing intersections of areas, one will need to compute the intersections of their boundary curves. If one is lucky, all the curve intersections will be well-conditioned. It may be the case that one encounters degenerate intersections as depicted in Figures 7.4 - 7.7. In these figures, the same pair of curves has been depicted eight times. The pair of curves sets within a single figure bound roughly the same area. The numerical uncertainty or perhaps the uncertainty of the original data makes it impossible to distinguish the case on the right from the case on the left. For Figures 7.4 and 7.5, the interpretation on the left is consistent with the stereographic rule while the interpretation on the right is not. For Figures 7.6 and 7.7, both the left and the right interpretations are consistent with the stereographic rule. The difference is that the boundary on the left is simpler than the boundary on the right. The boundary on the left consists of one curve whereas the boundary on the right consists of alternating pieces of both curves. The endpoints of the pieces on the right are numerically ill-conditioned.

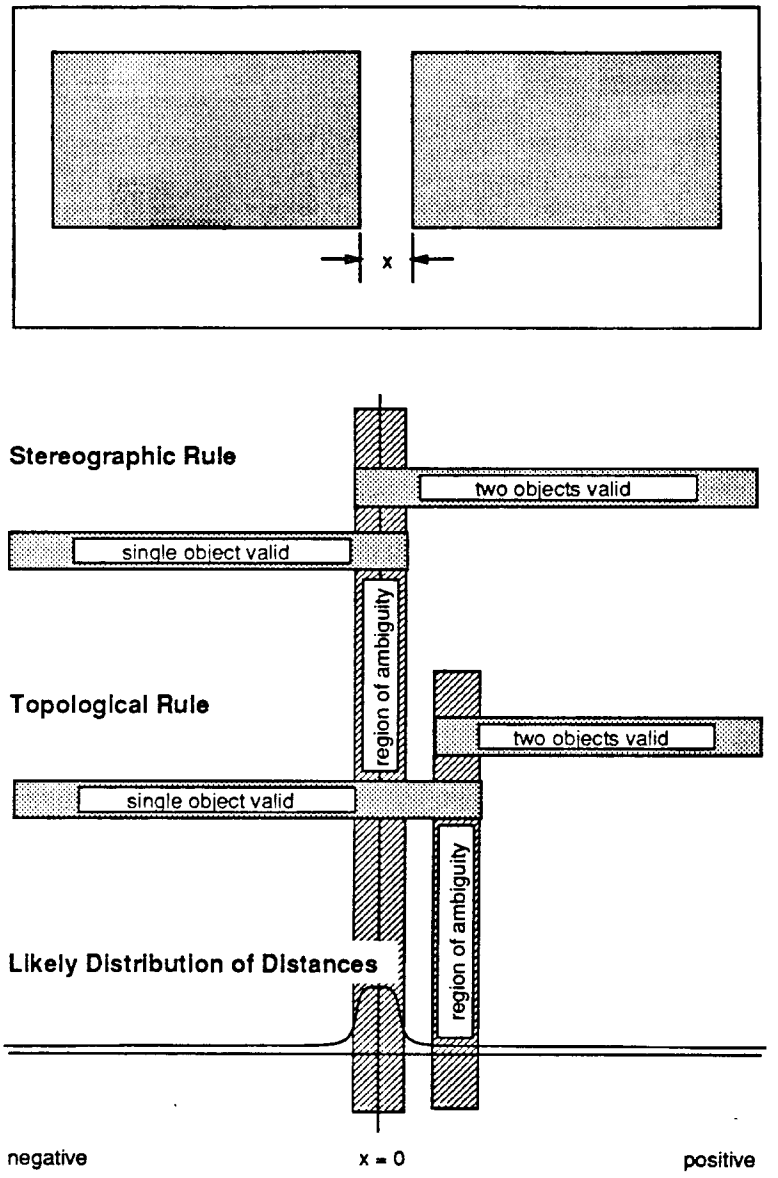


Figure 7.3: Comparison of the rules.

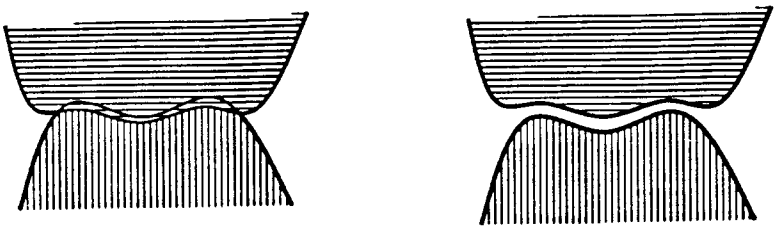


Figure 7.4: Degeneracy of type 1.

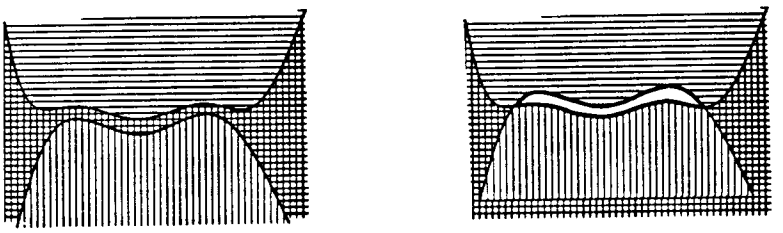


Figure 7.5: Degeneracy of type 2.

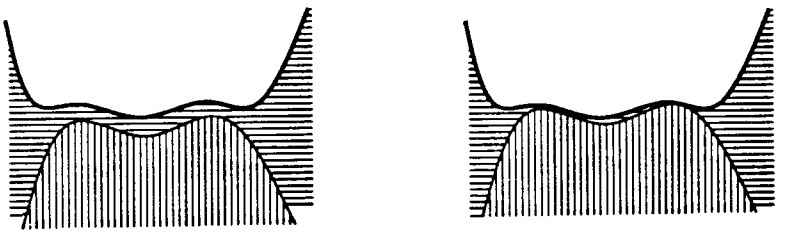


Figure 7.6: Degeneracy of type 3.

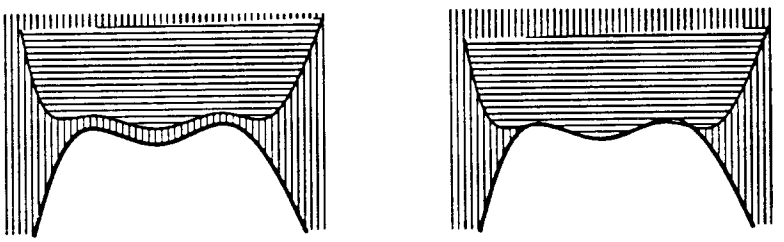


Figure 7.7: Degeneracy of type 4.

If one were two encounter such an intersection problem one would like to offset the boundaries according to Figures 7.4 - 7.7. That is, taking into account the orientation of the boundary, offset the boundary so that the result of the Boolean operation is consistent with the point classification rule and is numerically stable. For a pair of boundaries that intersect in a singular curve, local derivative information can be used to determine the appropriate direction to offset the surfaces. For a node singular point, no direction offset will make the intersection stable. For an area intersection, there may be a direction in which to offset but one cannot determine it. One could rerun the algorithm, first with one offset, then with another and determine which direction is stable.

7.3 Backtracking

Practically speaking, how can one incorporate the above policy into a surface intersection algorithm? All degenerate points (i.e. points that this algorithm cannot handle) must satisfy equations (6.4 - 6.7). Since Newton iteration is used at every subdivision step to try to find a solution to this system of equations, if there is a solution, it will be found.

The surface intersection algorithm is exploring a four dimensional box. The boundaries of the box are determined by an ancestor of the current invocation. There are two kinds of degeneracies. The first is caused by the alignment of the surface data; the second is caused by the alignment of the subdivision boundaries. An example of the latter is an isoparametric curve tangent lying in the tangent plane of the other surface. Viewed in four dimensions, the intersection curve is tangent to one of the boundaries of the box. The aim is to find the recursive invocation that placed the box boundary at that particular point. Suppose a curve tangent aligns with a subdivision boundary. If the tangent does not align with any of the boundaries inherited from the recursive call's parent, then either the recursive call or one of its descendants has placed the subdivision boundary in the degenerate position. Such degeneracies are detected by leaf algorithm calls. Since leaf calls do not place boundaries, the degenerate placement must be due to an ancestor call. The call returns the degeneracy report to the parent. When a node call is notified by a child of a degeneracy, the call determines if the tangent is aligned with any of its inherited boundaries. If not, then that node call must have placed a subdivision boundary in the degenerate position. In that case, the call deletes all results for its hyperbox and places the subdivision boundary in another position and continues. On the other hand, if the tangent does lie on an inherited

boundary, the node call simply returns with the degeneracy report.

This can terminate either by a node call finding that it has placed a subdivision boundary in a degenerate position or by control being returned by the root node call to the application. In this case, the original intersection problem contained the degeneracy. If this degeneracy is boundary placement degeneracy, the application has the option to extend the boundaries of the surface. On the other hand, the algorithm may have discovered a degeneracy of the type outlined in the previous section. In this case the application has the option to deal with the problem as it sees fit.

7.4 Conclusion

In this thesis, an algorithm has been presented for obtaining the curve of intersection between two surfaces. The algorithm has been proven to work under the restriction that the intersection contains no singularities. In the case that the intersection does contain singularities the algorithm will find and report them. A detailed description of all aspects of the algorithm has been given. The algorithms reliability has be corroborated by a number of examples.

7.5 Suggestions for Further Research

The most important question not resolved by this thesis, is the question of singularities. While the algorithm presented here will discover at least one if any exist, it is not guaranteed to discover all of them nor to describe any except the simplest. In a model of computation that allows for uncertainty in the data it is not clear that more is possible

One of the shortcomings of this method is the high degree of the pseudo-normal surfaces. For a polynomial patch, it is roughly twice the degree of the patch, and for a rational patch it is roughly three times the degree of the surface. This is a problem, because the controls points of this surface, if the underlying surface is a B-spline or Bézier surface, have been observed in practice to overestimate the the pseudo-normal surface considerably when the degree is high. In [97] (in preparation), Sederberg proposes a loop detection criterion based on bounds on the partial derivatives of the surface. These bounds will be one degree less that those of the surface (if the surface is polynomial). It is the author's impression that this criterion will be roughly as discriminating as the criterion presented

here. Thus, Sederberg seems to have solved the problem of high degree.

It is a straightforward application of linear programming to determine if there is a plane that separates two point sets, i.e. determines if their convex hulls intersect. To determine if three convex hulls have any point in common it seems to require eight variables making Seidel's algorithm unattractive. Is it possible to solve this problem in a different way?

7.6 Acknowledgements

I would like to thank the my advisor, Brian Barsky, and the members of the committee, Raimund Seidel, Christoph Hoffmann, and Robert Taylor for their help guiding this research and writing this thesis. Thanks to John Canny who served on the quals committee. Thanks to others who have reviewed the thesis: Thomas Sederberg, Thor Dokken, and Jim Miller. Thanks also to Vel Kahan for discussing singular points and telling me I needed to take Jim Demmel's class (Jim also gets thanks). Thanks to John Ousterhout, Terry, and Bob for for lending me the workstation, and to the Sprite group for answering all my questions. Thanks to Kathryn Crabtree without whom neither I nor any other graduate student could survive. Thanks to Carlo Séquin for all the lunches. Thanks to Ron Goldman for telling me I needed to go back to graduate school. Thanks to Bruce van Patten who first introduced me to this problem. Thanks to Ziv for all the help with LaTeX. Thanks to Seth, Paul, Ken, and John, original comrades from '87. Thanks to Russ for being my best friend. Thanks to Nina for having the most glamorous parties. Thanks to all the CS grad students for making Berkeley a great place to be a student. Thanks to my father for getting me into computers, and to my mother for making sure that wasn't all I studied.

Bibliography

[1] George Allen. Testing the accuracy of solid modelers. *Computer Aided Engineering*, pages 50–54, June 1985.

[2] A. Appel. Some techniques for shading machine renderings of solids. In *Proc. AFIPS 1968 Spring Joint Computer Conference*, pages 37–45, 1968.

[3] Paul R. Arner. *Another Look at Surface/Surface Intersection.* PhD thesis, University of Utah, Salt Lake City, 1987.

[4] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical algebraic decomposition i: The basic algorithm. *SIAM Journal of Computation*, 13(4):865–889, November 1984.

[5] C. Asteasu. Intersection of arbitrary surfaces. *Computer-Aided Design*, 20(6):533–538, July 1988.

[6] D. Ayala. Boolean operations between solids and surfaces. *Computer-Aided Design*, 20(8):452–465, October 1988.

[7] Chanderjit Bajaj, Thomas Garrity, and Joe Warren. On the applications of multi-equational resultants. Technical Report TR88-83, Rice University,, Houston, Texas, November 1988.

[8] Chanderjit L. Bajaj, Christoph M. Hoffmann, and John E. Hopcroft. Tracing planar algebraic curves. Technical Report CSD-TR-637, Department of Computer Science, Purdue University, West Lafayette, Indiana 47907, September 1987.

[9] Chanderjit L. Bajaj, Christoph M. Hoffmann, John E. Hopcroft, and Robert E. Lynch. Tracing surface intersections. *Computer Aided Geometric Design*, 5(4):285–307, November 1988.

[10] Robert Barnhill, Gerald Farin, and Bruce Piper. Surface/surface intersection. *Computer Aided Geometric Design*, 4:3–16, 1987.

[11] Robert Barnhill and S. N. Kersey. A marching method for parametric surface / surface intersection. *Computer Aided Geometric Design*, 7(1-4):257–280, 1990.

[12] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[13] Ian C. Braid. The synthesis of solids bounded by many faces. *Comm. ACM*, (18):209–216, 1975.

[14] Ian C. Braid. Notes on a geometric modeler. Technical Report CAD Group Document 101, Computer Labratory, University of Cambridge, June 1979.

[15] Wayne E. Carlson. An algorithm and data structure for 3d object synthesis using surface patch intersections. In *SIGGRAPH '82 Conference Proceedings*, pages 255–259, July 1982.

[16] J. J. Chen and T. M. Ozsoy. An intersection algorithm for C^2 parametric surfaces. In *Knowledge Engineering and Computer Modelling*, pages 69–77. 1986.

[17] J. J. Chen and T. M. Ozsoy. Predictor-corrector type of intersection algorithm for C^2 parametric surfaces. *Computer-Aided Design*, 20(6):347–352, July 1988.

[18] Koun-Ping Cheng. Using plane vector fields to obtain all the intersection curves of two general surfaces. In W. Strasser, editor, *Proc. Theory and Practice of Geometric Modeling*. 1988.

[19] J. H. Chuang and C. M. Hoffman. On local implicit approximation and its applications. Technical Report CSD-TR-812, Computer Sciences Department, Purdue University, September 1988.

[20] Carl de Boor. *A Practical Guide to Splines*. Springer-Verlag, 1978.

[21] I. de Lotto and R. Galimberti. Innovative design with computer graphics. *Alta Frequenza*, (5), 1967.

[22] Tor Dokken. Finding intersections of B-spline representations using recursive subdivision. *Computer Aided Geometric Design*, 2:189–196, 1985.

[23] Tor Dokken. Comments on imprecise formulations in Dokken's 1990 paper. *Personal Communication*, 1991.

[24] Tor Dokken, Vibeke Skytt, and Anne-Marie Ytrehus. *Recursive Subdivision and Iteration in Intersections and Related Probelms*, pages 207–214. Academic Press, 1980.

[25] Herbert Edelsbrunner. *Intersection problems in computational geometry.* PhD thesis, Technische Universität Graz, Austria, 1982.

[26] Rida T. Farouki. Direct surface section evaluation. In Gerald Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 319–334. SIAM, Philadelphia, 1987.

[27] Ivor D. Faux and Michael J.Pratt. *Computational Geometry for Design and Manufacture.* Ellis Horwood, 1979.

[28] Daniel Filip, Robert Markot, and Robert Madgedson. Using bounds on derivatives in computer aided geometric design. Submitted for publication.

[29] Steve Fortune and Victor Milenkovic. Numerical stability of algorithms for line arrangements. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, pages 334–341, June 1991.

[30] G. O. Gellert. Geometric computing - electronic geometry for semi-automated design. *Machine Design*, 1965.

[31] Ronald N. Goldman and James R. Miller. Combining algebraic rigor with geometric robustness for the detection and calculation of conic sections in the intersection of two natural quadrics. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 221–231, June 1991.

[32] R. A. Goldstein and R. Nagel. 3-D visual simulation. *Simulation*, 16(1):25–31, January 1971.

[33] Gene H. Golub. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, Maryland, 1983.

[34] Branko Grunbaum. *Convex Polytopes.* Interscience Publishers, London, New York, Sydney, 1967.

[35] W. L. Hamilton and A. D. Weiss. An approach to computer aided preliminary ship design. 1965.

[36] S. L. Hanna, John F. Abel, and Donald P. Greenberg. Intersection of parametric surfaces by means of look-up tables. *IEEE Computer Graphics and Applications*, 3(3), October 1983.

[37] Robin Hartshorne. *Algebraic Geoemtry.* Springer-Verlag, New York, 1977.

[38] John D. Hobby. Numerically stable implicitization of cubic curves. *ACM Transactions on Graphics*, 10(3):255–296, July 1991.

[39] Christoff M. Hoffmann. A dimensionality paradigm for surface interrogations. *CAGD*, 7:517–532, 1990.

[40] Christoph M. Hoffmann. Algebraic curves. Technical Report Technical Report, Computer Science Department, Purdue University, West Lafayette, Indiana 47907, May 1987.

[41] Christoph M. Hoffmann. Applying algebraic geometry to surface intersection evaluation. Technical Report CSD-TR-772, Computer Science Department, Purdue University, West Lafayette, Indiana 47907, April 1988.

[42] Christoph M. Hoffmann. The problem of accuracy and robustness in geometric computation. Technical Report CSD-TR-771, Computer Science Department, Purdue University, West Lafayette, Indiana 47907, April 1988.

[43] Christoph M. Hoffmann. *Geometric and Solid Modeling.* Morgan-Kaufmann Publishers, Inc., San Mateo, California, 1989.

[44] Christoph M. Hoffmann. Robust set operations on polyhedral solids. *IEEE Computer Graphics and Applications*, pages 50–59, November 1989.

[45] Christoph M. Hoffmann, John E. Hopcroft, and Michael S. Karasick. Robust set operations on polyhedral solids. Technical Report CSD-TR-723, Computer Science Department, Purdue University, West Lafayette, Indiana 47907, November 1987.

[46] Christoph M. Hoffmann and Robert E. Lynch. Following space curves numerically. Technical Report CSD-TR-684, Computer Science Department, Purdue University, West Lafayette, Indiana 47907, May 1987.

[47] Michael E. Hohmeyer. A surface intersection algorithm based on loop detection. *International Journal of Computational Geometry and Applications*, 1(4), 1991.

[48] Michael E. Hohmeyer. A surface intersection algorithm based on loop detection. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 197–207, June 1991.

[49] John E. Hopcroft and Dean B. Krafft. The challenge of robotics for computer science. In Jacob T. Schwartz and Chee-Keng Yap, editors, *Algorithmic and Geometric Aspects of Robotics*, pages 7–41. L. Erlbaum Associates, Hillsdale, New Jersey, 1987.

[50] E. G. Houghton, R. F. Emnet, J. D. Factor, and C. L. Sabharwal. Implementation of divide-and-conquer method for intersection of parametric surfaces. *Computer Aided Geometric Design*, pages 173–183, 1985.

[51] Eugene Isaacson and Herbert Bishop Keller. *Analysis of Numerical Methods.* John Wiley and Sons, 1966. Kantorovich's Theorem page 115.

[52] C. M. Jessop. The mechanical tracing of curves. *Proceedings of the Cambridge Mathematical Society*, 1880.

[53] John K. Johnstone. *The Sorting of Points Along An Algebraic Curve.* PhD thesis, Cornell University, Ithaca, New York, 1987.

[54] Michael Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, Department of Computer Science, McGill University, November 1988.

[55] Sheldon Katz and Thomas W. Sederberg. Genus of the intersection curve of two rational surface patches. 1988.

[56] Deok-Soo Kim. *Cones on Bézier Curves and Surfaces.* PhD thesis, University of Michigan, 1990.

[57] Moris Kline. *Mathematical Thought from Ancient to Modern Times.* Oxford University Press, 1972.

[58] George Anthony Kriezis. *Algorithms for Rational Spline Surface Intersections.* PhD thesis, Massachusetts Institute of Technology, Boston, 1990.

[59] George Anthony Kriezis and Nicholas M. Patrakalakis. Rational polynomial surface intersections. Technical Report 90-9, MIT Ocean Engineering Design Laboratry, June 1990.

[60] George Anthony Kriezis and Nicholas M. Patrakalakis. Rational polynomial surface intersections. In *Proceedings of the 17th ASME Design Automation Conference*, September 1991.

[61] Dieter Lasser. Intersection of parametric surfaces in the Bernstein-Bézier representation. *Computer-Aided Design*, 18(4):186–192, May 1986.

[62] Joshua Z. Levin. Mathematical models for determining the intersections of quadric surfaces. *Computer Graphics and Image Processing*, 11:73–87, September 1979.

[63] Tom Lyche and Knut Mørken. A discrete approach to knot removal and degree reduction algorithms for splines. In J.C. Mason and M.G. Cox, editors, *Algorithms for Approximation*, pages 67–82. Claredon Press, Oxford, 1987.

[64] Tom Lyche and Knut Mørken. A data-reduction strategy for splines with applications to the approximation of functions and data. *IMA Journal of Numerical Analysis*, 8(2):185–208, April 1988.

[65] Jean-Jacques Malosse. Search of intersection loops by solving algebraic equations. Technical report, Intergraph Corporation.

[66] Dinesh Manocha and John F. Canny. A new approach for surface intersection. Technical Report RAMP 90-11/ESRC 90-23, Robotics, Automation, and Manufacturing Program, University of California, Berkeley, December 1990.

[67] Dinesh Manocha and John F. Canny. A new approach for surface intersection. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 209–219, June 1991.

[68] Martti Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, Maryland, 1988.

[69] R. P. Markot and R. L. Magedson. Solutions of tangential surface and curve intersections. *Computer-Aided Design*, 21(7):421–427, September 1989.

[70] K. J. McCallum. The intersection of surfaces and planes. 1966.

[71] Victor Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. pages 197–207, 1989.

[72] James R. Miller. Geometric approaches to nonplanar quadric surface intersection curves. *ACM Transactions on Graphics*, 6(4):274–307, October 1987.

[73] James R. Miller. Sculptured surfaces in solid models: Issues and alternative approaches. *IEEE Computer Graphics and Applications*, 6(12):33–43, 1987.

[74] James R. Miller. Analysis of quadric-surface-based solid models. *IEEE Computer Graphics and Applications*, 8(1):28–42, 1988.

[75] Y. De Mountaudouin. Cross product of cones of revolution. *Computer Aided Design*, 21(6), 1989.

[76] Y. De Mountaudouin, W. Tiller, and H. Vold. Applications of power series in computational geometry. *Computer Aided Design*, pages 514–524, 1986.

[77] Y. Kakazu N. Okino and H. Kubo. *TIPS-1: Technical Information Processeing System for Computer Aided Design and Manufacture*, pages 141–150. North Holland, 1973.

[78] B. K. Natarajan. On computing the intersection of B-splines. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 157–167. ACM, November 1990.

[79] J. Owen and Alan Rockwood. Intersection of general implicit surfaces. In Gerald Farin, editor, *Geometric Modeling: Algorithms and Trends.* SIAM Publications, Philadelphia, 1987.

[80] Beresford N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[81] N. M. Patrakalakis, P. V. Prakash, H. Nebi Gursoy, and George A. Kriezis. *Research Topics in Shape Interrogation.* J. Wiley and Sons, Ltd., 1991.

[82] Qun Sheng Peng. An algorithm for finding the intersection lines between two b-spline surfaces. *Computer-Aided Design*, 16(4), July 1984.

[83] H. U. Pfeifer. Methods for intersecting geometrical entities in the gpm module for volume geometry. *Computer-Aided Design*, 17(7):311–318, September 1985.

[84] Leslie Piegl. Geometric method of intersecting natural quadrics represented in trimmed surface form. *Computer Aided Design*, 21(4):201–212, May 1989.

[85] Michael J. Pratt and A. D. Geisow. Surface/surface intersection problems. In John A. Gregory, editor, *The Mathematics of Surfaces*, pages 117–142. Claredon Press, Oxford, 1986.

[86] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[87] Aristides A. G. Requicha. Mathematical models of rigid solids. Technical Report Tech. Memo. No. 28, Production Automation Project, University of Rochester, 1977.

[88] Aristides A. G. Requicha. Solid modelling - a 1988 update. Technical Report IRIS 242, University of Southern California, August 1988.

[89] Aristides A. G. Requicha and H. B. Voelker. Constructive solids geometry. Technical Report Tech. Memo. No. 25, Production Automation Project, University of Rochester, 1977.

[90] Malcolm A. Sabin. *Interrogation Techniques for Parametric Surfaces*, pages 1095–1118. Plenum Press, London and New York, 1971.

[91] Malcolm A. Sabin. A method for displaying the intersection curve of two quadric surfaces. *The Computer Journal*, 19:336–338, November 1976.

[92] Ramon F. Sarraga. Algebraic methods for intersections of quadric surfaces in gmsolid. *Computer Vision, Graphics and Image Processing*, 22(2):222–238, May 1983.

[93] Thomas W. Sederberg. *Implicit and Parametric Curves and Surfaces for Computer Aided Geometric Design.* PhD thesis, Purdue University, West Lafayette, Indiana, 47907, August 1983.

[94] Thomas W. Sederberg, Hank N. Christiansen, and Sheldon Katz. Improved test for closed loops in surface intersections. *Computer Aided Design*, 21(8):505–508, October 1989.

[95] Thomas W. Sederberg and Ray J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5:161–171, 1988.

[96] Thomas W. Sederberg and X. Wang. Rational hodographs. *Computer Aided Geometric Design*, 4:333–335, 1988.

[97] Thomas W. Sederberg and Alan K. Zundel. Cones that bound rational surface patches. Technical Report ECGL91-2, Brighan Young University, January 1992.

[98] Mark G. Segal and Carlo H. Sequin. Maintaing topology in geometric descriptions with numerical uncertainty. Technical Report UCB/CSD 88/450, Computer Science Division, Electrical Engineering and Computer Science Department, University of California, Berkeley, June 1988.

[99] Raimund Seidel. Linear programming and convex hulls made easy. In *ACM Symposium on Computational Geometry*, pages 211–215. ACM Press, 1990.

[100] Raimund Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry*, 6(5):423–434, 1991.

[101] Ching-Kuang Shene and John K. Johnstone. On the planar intersection of natural quadrics. In *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 233–242, 1991.

[102] Pradeep Sinha, Eric Klassen, and K. K. Wang. Exploiting topological and geometric properties for selective subdivision. In *ACM Symposium on Computational Geometry*, pages 39–45. ACM Press, 1985.

[103] Jack Scott Snoeyink. Intersecting trimmed quadric surface patches: a geometric method using parametric functions, 1990.

[104] D. M. Y. Sommerville. *Analytic Geometry of Three Dimensions*. Cambridge University Press, 1934.

[105] Kokochi Sugihara. On finite-precision representations of geometric objects. *Journal of Computer and System Sciences*, 39:236–247, 1989.

[106] H. B. Voelker. An introduction to PADL: Characteristics, status and rationale. Technical Report Tech. Memo. No. 22, Production Automation Project, University of Rochester, 1974.

[107] H. B. Voelker. The PADL-1.0/2 system for defining and displaying solid objects. *Computer Graphics*, 12(3), July 1978.

[108] H. B. Voelker. Keynote speech, symposium on solid modeling foundations and CAD/CAM applications, June 1991.

[109] H. B. Voelker and A. A. G. Requicha. Geometric modeling of physical parts and processes. *IEEE Computer*, 10(2), 1977.

[110] C. Yao and J. Rokne. An efficient algorithm for subdividing linear Coons surfaces. Submitted for publication.