Efficient and Accurate Boundary Evaluation Algorithms for Boolean Combinations of Sculptured Solids

by

Shankar Krishnan

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

Chapel Hill

1997

Approved by:

Dr. Dinesh Manocha, Advisor

Dr. Gregory Turk, Reader

Dr. David Eberly, Reader

To my parents, Saroja Krishnan and C. N. Krishnan

SHANKAR KRISHNAN. Efficient and Accurate Boundary Evaluation Algorithms for Boolean Combinations of Sculptured Solids
(Under the direction of Dr. Dinesh Manocha.)

ABSTRACT

This dissertation presents techniques to effectively compute Boolean combinations of solids whose boundary is described using a collection of parametric spline surfaces. It also describes a surface intersection algorithm based on a lower dimensional formulation of the intersection curve that evaluates all its components and guarantees its correct topology. It presents algorithms and data structures to support the thesis that the lower dimensional formulation of the intersection curve is an effective representation to perform Boolean operations on sculptured solids. The thesis also analyzes different sources of problems associated with computing the intersection curve of two high degree parametric surfaces, and presents techniques to solve them.

More specifically, the intersection algorithm utilizes a combination of algebraic and numeric techniques to evaluate the curve, thereby providing a solution with greater accuracy and efficiency. Given two parametric surfaces, the intersection curve can be formulated as the simultaneous solution of three equations in four unknowns. This is an algebraic curve in $\mathcal{R}^4$. This curve is then *projected* into the domain of one of the surfaces using *resultants*. The intersection curve in the plane is represented as the *singular set* of a bivariate matrix polynomial. We present algorithms to perform *loop* and *singularity detection*, use curve tracing methods to find all the components of the intersection curve and guarantee its correct topology. The matrix representation, combined with numerical matrix computations like singular value decomposition, eigenvalue methods, and inverse iteration, is used to evaluate the intersection curve.

We will describe a system BOOLE, a portable implementation of our algorithms, that generates the B-reps of solids given as a CSG expression in the form of *trimmed* Bézier patches. Given two solids, the system first computes the intersection curve between the two solids using our surface intersection algorithm. Using the topological information of each solid, we compute various components within each solid generated by the intersection curve and their connectivity. The *component classification* step is performed by using ray-

shooting. Depending on the Boolean operation performed, appropriate components are put together to obtain the final solid. The system has been successfully used to generate B-reps for a number of large industrial models including a notional submarine storage and handling room (courtesy - Electric Boat Inc.) and Bradley fighting vehicle (courtesy - Army Research Labs). Each of these models is composed of over 8000 Boolean operations and is represented using over 50,000 trimmed Bézier patches. Our exact representation of the intersection curve and use of stable numerical algorithms facilitate an accurate boundary evaluation at every Boolean set operation and generation of topologically consistent solids.

Acknowledgements

I wish to thank my advisor, Dr. Dinesh Manocha, for his excellent guidance and continued support throughout my dissertation period. I am very grateful for his willingness to find time for me in spite of his busy schedule and keeping my spirits up during some tough times. I would like to express my deepest gratitude to the other members of my committee: Dr. Stephen Pizer for his expectation of the highest standards, Dr. Pankaj Agarwal for providing me with a wealth of knowledge in computational geometry, Dr. David Eberly and Dr. Gregory Turk for agreeing to read my thesis and providing insightful criticisms and Dr. Ilse Ipsen for improving my understanding of numerical analysis. I would like to thank Dr. Frederick Brooks for taking interest in my work.

I would like to thank Dr. Thomas Sederberg, Mike Hohmeyer and Atul Narkhede for providing their implementations of Bézier Clipping, linear programming and polygon triangulation. Thanks also to Ken Fast, Greg Angelini and Jim Boudreaux at Electric Boat and Mike Muuss at Army Research Labs for providing their CSG models for me to use. A special thanks to the members of the Modeling and Walkthrough groups at UNC, especially Subodh Kumar, Gopi Meenakshisundaram, John Keyser and Kenny Hoff for providing display routines to test the results of my algorithms. I would like to thank Gopi for implementing the parallel version of our algorithm and Sumedh Barde for providing the GLUT graphical user interface.

Last but certainly not the least, I would like to thank my mother, Saroja Krishnan, whose immense hard work and sacrifice made it possible for me to get to this point. I would also like to thank my sisters, Lakshmy and Meenakshi, for their love and care.

This work was supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9319957, NSF Career Award CCR-9625217, ONR Young Investigator Award (N00014-97-1-0631), Intel Corp., DARPA Contract DABT63-93-C-0048 and NSF/ARPA Center for Computer Graphics and Scientific Visualization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of solid modeling deals with design and representation of physical objects. One of its main emphases has been on the consistency of models generated. Boolean operations, such as regularized unions, intersections and differences, on solids play a fundamental role in solid modeling. They are used in various applications in mechanical engineering, computer graphics, robotics and computer vision. The two major representation schemata used in solid modeling are constructive solid geometry (CSG) and boundary representations (B-rep). B-reps describe solids as a set of vertices, edges, and faces with topological relations among them. In contrast, CSG considers solids as expressions of Boolean operations and rigid motion transformations of primitive solids which typically include polyhedra, spheres, cylinders, cones, tori and surfaces of revolution. Both these representations have different inherent strengths and weaknesses, and for most applications both are desired. For instance, a CSG object is always valid in the sense that its surface is closed, orientable and encloses a volume, provided the primitives are valid in this sense. A B-rep object, on the other hand, is easily rendered on a graphic display system and is useful for visual feedback in solid design. Figure 1.1 shows the model of a notional submarine storage and handling room that we obtained from Electric Boat, a division of General Dynamics. This model consists of more than 5000 solids, each designed using Boolean operations. The primitives used to generate these models vary from simple polyhedral objects, spheres and cylinders to fairly complex ones like generalized prisms, surfaces of revolution and offset surfaces. Figure 6.1

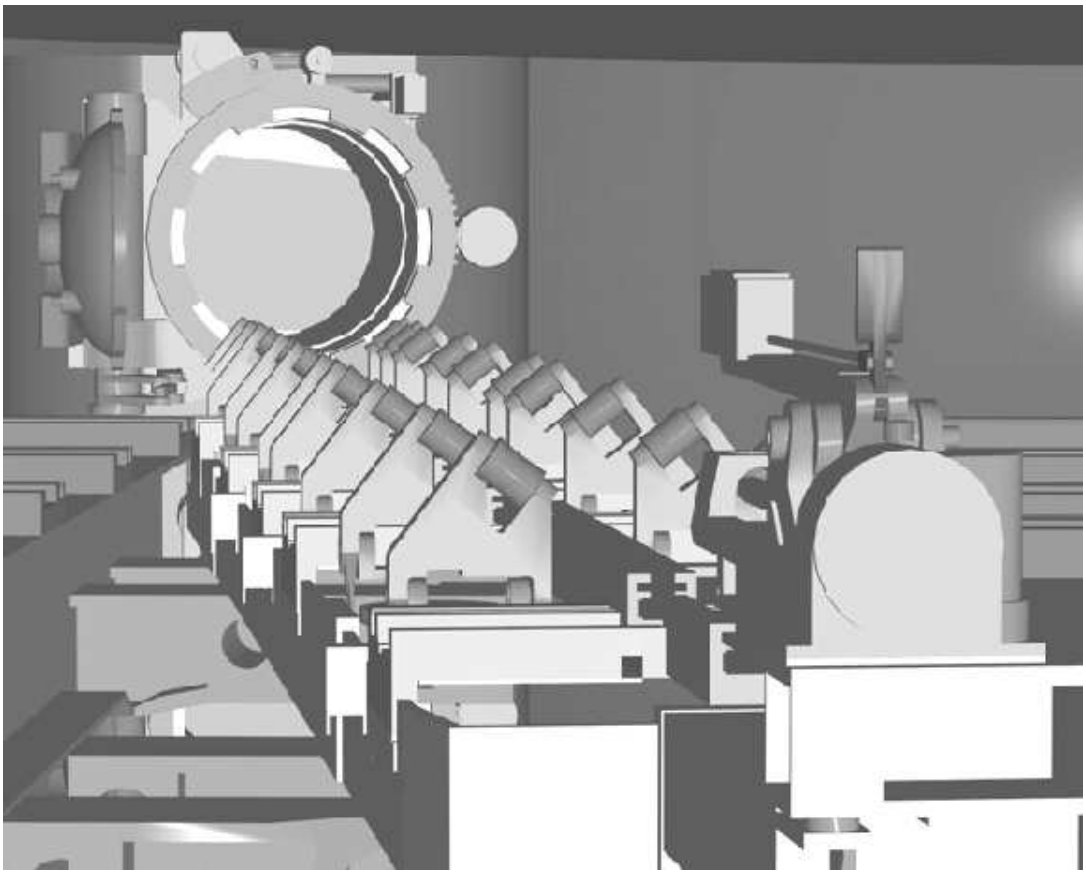

Figure 1.1: Submarine storage and handling room

is a model of a real Bradley fighting vehicle from Army Research Labs. This model has over 8500 solids generated entirely using Boolean operations as well. Generating the B-reps of such large CAD models is necessary for applications like interactive visualization and model verification. Another application where B-reps are required is in collision detection for dynamic simulation of machine parts. For example, consider the track of the Bradley shown in Figure 7.11. The toothed circular structure shown in the left hand side of the image is the drivewheel. It is placed in the track in such a way that when it rotates without slippage, the Bradley vehicle moves forward. Placement of the drivewheel is very critical to obtain this effect. Dynamic simulations are performed to study the model placement. To simulate these realistically, we require algorithms that can perform interference detection. B-reps are necessary for this purpose.

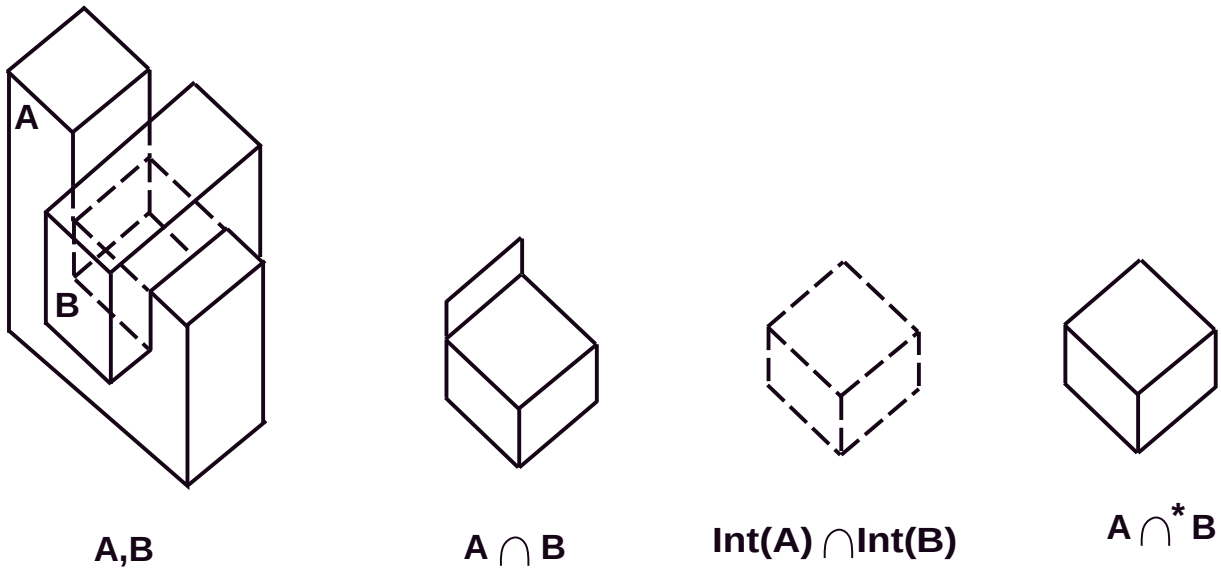

Figure 1.2: An example of regularized intersection operation [Hof89]

Algorithms for determining the *regularized* union, intersection, or set difference of two solids is a useful component of most B-rep modelers. The *regularized* operations differ from their corresponding set-theoretic counterparts in that the result is the *closure* of the operation on the interior (mathematically speaking, this refers to all of the solid except its boundary) of the two solids, and are used for eliminating "dangling" lower-dimensional structures (see Figure 1.2). If $Int(A)$ represents the interior of solid A and op corresponds to one of the set operations, we define op^*, the regularized version of the Boolean operation as

$$A \; op^* \; B \; = \; cl(Int(A) \; op \; Int(B))$$

where $cl(A)$ denotes the closure (generates boundary) of A. These operations can be used to convert solids represented by CSG trees (see Figure 1.3) to an equivalent B-rep. These processes for performing Boolean operations on B-reps are called *boundary evaluation* algorithms. These algorithms are not difficult conceptually, but their implementation requires substantial work for several reasons. Implementation of layers of primitive geometric and topological operations have to be designed. Examples of primitive geometric and topological layers include polygon triangulation, point location in a planar arrangement, linear

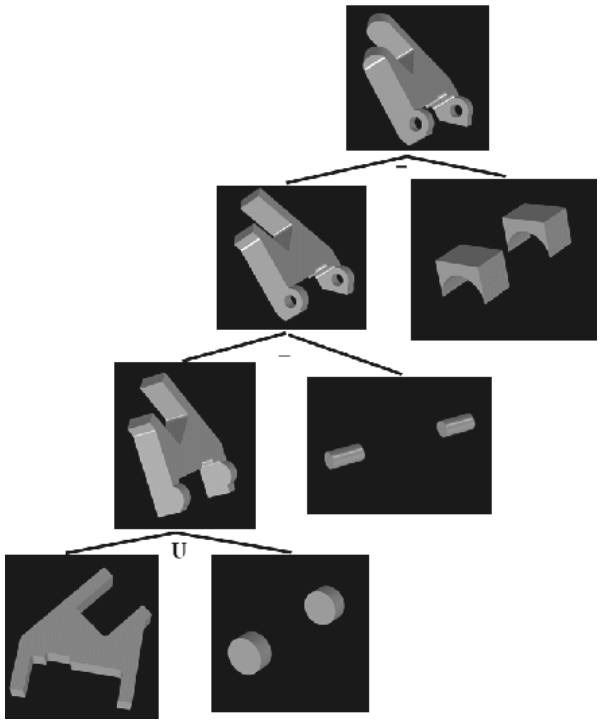

Figure 1.3: Part of a CSG tree of the roller from the submarine model

programming and graph algorithms. Finding a good structure for these layers is nontrivial, and accounting for all the special positions of incident structures can be quite difficult. Further, the presense of curved surfaces introduces a number of difficult mathematical problems. The use of numerical techniques to solve these problems inevitably leads to the problem of numerical precision and stability of computation.

The use of free-form surfaces in model design has been the primary contribution of the field of computer-aided geometric design (or geometric modeling) to solid modeling. This field deals with the design of curved surfaces using parametric surface patches and, in particular, various types of spline surfaces subject to aesthetic and functional constraints. Surface design with splines originated in the automobile industry, principally for car-body design. It was also used in shipbuilding and design of aircraft wings. Other researchers have been using free-form surfaces in design of submarines (Figure 1.1), fighting vehicles (Figure 6.1) and other applications.

Earlier, most B-rep modelers were able to support solids composed of polyhedral models and quadric surfaces (like spheres, cylinders etc.) and their Boolean combinations only. Over the last few years, modeling using free-form surfaces (*sculptured models*) has become very useful throughout the commercial CAD/CAM/CAE industry. On the research front, there has been considerable effort in integrating geometric and solid modeling [Kal82, Jar84, CK83, VP84, KGI84, FH85, Far86]. In particular, there is a lot of interest in building complete solid representations from spline surfaces and their Boolean combinations [Hof89, RR92, CS85, Cas87, Wei85, RV82, Cha87, Men92]. However, the major bottleneck is in performing robust, efficient and accurate Boolean operations on the sculptured models. According to Hoffmann [Hof89]: "*The difficulty of evaluating and representing the intersection of parametric surface patches has hindered the development of solid modelers that incorporate parametric surface patches*". The topology of a surface patch becomes quite complicated when a number of Boolean operations are performed and finding a convenient representation for these topologies has been a major challenge.

In many applications involving CAD/CAM, solids are designed in terms of tensor product trimmed Non-Uniform Rational B-Spline (NURBS) surfaces. This class includes a number of rational parametric surfaces like tensor-product and triangular Bézier patches. A detailed description of NURBS and tensor-product surfaces is given in chapter 2. The representation capability of these surfaces is quite large, and is sufficient to represent all primitive solids encountered in boundary evaluation systems. Due to the difficulty in performing free-form surface intersection, many B-rep modelers use high-resolution polyhedral approximations to these surfaces and apply existing algorithms to design and manipulate these polyhedral objects. Apart from the fact that the resulting solids are inaccurate, there is an additional cost in terms of increased memory usage due to data proliferation. This dissertation seeks to change that by providing effective strategies to perform Boolean operations on sculptured solids without resorting to polyhedralization.

1.1 Summary of Results

In order to compute boundary representations of Boolean combinations of sculptured solids directly, we have developed a number of techniques. The main theme of our approach is to use a combination of symbolic and numeric algorithms for efficient and accurate computation.

The main contribution of this work is to show the effectiveness of a surface intersection algorithm which computes the intersection curve in a plane as opposed to higher dimensions. Formulating the intersection curve as a planar algebraic curve has been known for quite some time. However, it was believed that this approach would suffer from problems of inaccuracy and numerical instability (for example, symbolic determinant evaluation). By making use of a new representation for the intersection curve coupled with stable numerical methods, we have alleviated most of these problems. The surface intersection algorithm also uses newly developed algorithms for performing curve-surface intersections, loop detection and curve tracing. To the best of our knowledge, the loop detection algorithm presented in this dissertation is the first comprehensive algorithm to detect closed loops of an algebraic plane curve inside a finite domain of interest. Previous loop detection algorithms have been restricted to intersection curves and provide only necessary criteria for loops. Further, the efficiency of these methods suffer in the presence of singular points in the intersection curve. Current surface intersection algorithms that use numerical marching methods to evaluate the intersection curve suffer from reliability problems when different curve components come close to each other. Our curve tracing algorithm guarantees the correct topology of the curve under these situations.

Our boundary evaluation algorithm generates B-reps of solids in terms of trimmed parametric surfaces. Refer to chapter 2 for more details on trimmed patches. The trimming boundaries in these surfaces are high-degree algebraic curves. By maintaining an accurate representation of the trimming curves, we are able to perform accurate trimmed surface intersection. Earlier methods resort to approximations of the algebraic curves which result in solids that have inconsistent topology. The component classification step is used to determine which parts of the original solids are to be retained in the final solid. This step

consists of a number of fairly expensive operations especially when dealing with sculptured solids. By using the topological information of each solid, we perform an optimal number of such operations to improve the efficiency of this method.

1.2 Previous Work

The need to generate accurate boundary representations of solid objects in many applications involving design and manufacturing has generated significant interest in the research community. Over the years, the body of literature addressing these problems has grown to be quite extensive. Some of the earliest work in generating B-reps was done on polyhedral solids. The need to use free-form surfaces to represent solids has led to research in the problems of curve-surface and surface-surface intersection and loop detection which are important for B-rep generation.

Currently, most boundary evaluation algorithms follow a general framework. This framework was first introduced by Requicha and Voelcker [RV85] to perform Boolean operations on polyhedra. However, this can be extended easily to accommodate curve surface domains as well. Given two polyhedra, A and B, the conceptual structure of the algorithm [Hof89, HHK89] is

1. Determine which pairs of faces $f \in A$ and $g \in B$ intersect. If there are none, test for containment only and skip all other steps.

2. For each face $f \in A$ that intersects faces $g_i \in B$, compute the intersecting line segments between f and g_i's. The set of all intersecting line segments partitions the surface of face f. Determine the partitions of f that contribute to some of the surface area of the resulting solid.

3. Perform the same for all faces of B.

4. Assemble all the faces into the new solid.

1.2.1 Boundary Evaluation Techniques

Polyhedral solids: Algorithms for performing Boolean operations on polyhedra

in B-rep have been proposed by a number of researchers [Bra75, Hil82, Man86, OKK73, Voe74, Wes80]. Most of these techniques rely heavily on the algebraic formulation of the problem. Cameron [Cam85] considers several strategies and redundancy tests to propagate approximations of CSG primitives from the root of the CSG tree down to the leaves, and possibly refining them on the way. Rossignac and Voelcker [RV89] consider redundancy determination without approximating the primitives. They define certain *active zones* on solids and show how knowledge of active zones can be used to improve conversion from CSG to B-rep, detection of redundancy and other operations on CSG trees.

The use of topological structures of solids has been very popular in B-rep solid modeling. The winged-edge style of boundary representation is due to Baumgart [Bau75]. Many variants of the method, and other alternatives, have been proposed and used in B-rep modeling systems since then. A complete survey of topological structures in solid modeling is given in [Wei86]. The use of non-manifold boundary representations was first proposed by Wesley [Wes80]. Weiler [Wei85, Wei86] observed that a number of geometric operations on polyhedra simplify when non-manifold structures are permitted. Paoluzzi et. al. [PRS86] implement Boolean operations on B-rep solids by using only triangular faces for their polyhedra. Laidlaw et. al. [LTH86] describes another method in which all faces must be convex polygons, and suggest random perturbations to eliminate complex vertex intersection cases.

A number of approaches have been proposed for robust and accurate B-rep computation in polyhedral modelers. One of the most common approaches is based on using *tolerances* with floating-point arithmetic [Jac95]. However, it is hard to decide a global tolerance value for all computations. To circumvent these problems, combinations of symbolic reasoning [HHK89] and adaptive tolerances [Seg90] have been proposed. Other algorithms include those based on redundancy elimination [FBZ93]. Many algorithms based on *exact arithmetic* have been proposed for reliable numeric computation for polyhedra [SI89, For95, BMP94, Hof89].

Sculptured Solids: The idea of using free-form surfaces in solid modeling was introduced by Chiyokura et. al [CK83]. It describes the implementation of a system called *Designbase* with some curved-surface capabilities. In this system, curved solids are designed

and modified by local operations such as altering the shape of certain edges and faces. However, Boolean operations require that one of the intersecting objects be polyhedral. Geisow [Gei83] maps surface intersection curves to the plane and uses subdivision methods to solve surface interrogation problems. Requicha and Voelcker [RV85] describes the PADL system developed at University of Rochester. This system supports Boolean operations on polyhedral solids and a few curved primitives. Casale et. al. [CS85, Cas87, CB89] use trimmed parametric surfaces to generate B-reps of sculptured solids. The algorithm uses subdivision methods to evaluate surface intersections, and represents the trimming boundary with piecewise linear segments. Chan [Cha87] uses special properties of quadric surfaces and other free-form surface to design industrial parts. A number of techniques like interval arithmetic and shell representations [VP84, KGI84, Taw91, Sat91, Men92, Duf92] have been developed to perform solid design with free-form geometries. Sorting points along intersection curves [Joh87] was used to classify components with respect to solids.

The Alpha_1 CAD system developed at the University of Utah has many features to combine solids composed of sculptured surfaces. A systematic approach for design, analysis and illustration of assemblies has been presented in [DC95, RMS92]. Ray representations along with specialized parallel architectures [Mea84, EKL$^+$91, Men92] like the RayCasting engine and 'Solids engine' were used to achieve interactive solid modeling on low-degree primitives like quadrics. Mantyla and Ranta [MR86] describe methods to perform solid modeling using HutDesign. Rossignac et. al. [RMS92] present algorithms for inspection of cross-sections and interference between solids with bounded degree and limited height of CSG trees.

Most of the recent work in the literature on Boolean combinations of curved models has focussed on computing the surface intersection between a pair of B-spline surfaces [KS88, SN91, Nat90, Hoh92, MC91, KPW90, BHHL88, BK90, KM97]. We shall now look at some of these methods.

1.2.2 Surface Intersection Techniques

There is a significant body of literature addressing the surface intersection problem. Some recent surveys include [Pat93, Pra86, Hof89]. One of the main issues that has

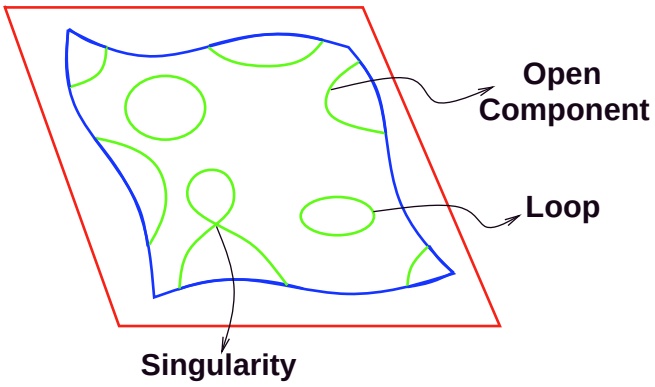

Figure 1.4: Various components of the intersection curve

to be considered while designing surface intersection algorithms is that two surfaces can intersect in a number of components including small loops and singularities (see Figure 1.4). Evaluation of all the components along with their correct connectivity structure is very important. In general, surface intersection techniques can be broadly classified into four major categories: *subdivision, lattice evaluation, analytic methods,* and *marching methods.* More recently, techniques have been designed that combine features of different categories. These are generally referred to as *hybrid methods.* Our approach uses a combination of analytic and marching methods to compute the intersection curve between surfaces.

Subdivision methods: The basic idea of these methods is to decompose the problem recursively into similar problems which are much simpler. Decomposition continues until a desired level of simplicity is achieved and then the corresponding intersection is obtained directly. The last step is to merge all the individual curves together to get the final solution. This approach has the flavor of the *divide and conquer* paradigm used extensively in algorithmic design. Subdivisions are based on the geometric properties of the control polytopes [LR80, Gei83, Las86]. These methods are convergent in the limit, but if used for high-precision results, lead to data proliferation and are consequently slow. In case subdivision is stopped after some finite number of steps, it may miss small loops or lead to incorrect connectivity in the presence of singularities. The robustness of this approach can be improved by posing the problem algebraically and using *interval arithmetic* [Sny92].

Lattice evaluation: These techniques decompose surface intersection into a series of lower geometric complexity problems like curve-surface intersections [RR87]. This is followed by connecting the discrete points into curves. Determination of the discrete step size to guarantee robust solutions is hard. Further, these techniques can be slow and suffer from robustness problems in terms of finding all the small loops and singularities.

Analytic methods: Analytic methods are based on explicit representation of the intersection curve and have been restricted to low degree intersections [Sed83, Sar83]. Another alternative to the analytic methods is the use of geometric methods developed by [Pie89]. In this paper, Piegl uses geometric principles to compute the intersection of quadric surfaces very accurately. However, the algorithm cannot be easily extended to the general intersection problem.

Marching methods: These are by far the most widely used [Far86, BHHL88, BK90, KPW90] and are easy to implement. The main advantage of this technique is its generality, allowing intersection of arbitrary parametric surfaces as well as their offsets and blends. The idea behind marching methods involves analytic formulation of the intersection curve, determination of a start point on each component, and the use of local geometry to trace out the curve. The intersection curve is defined implicitly as an algebraic set based on the surface equations, as a curve of zero distance between the two surfaces, or as a vector field [Hof90, KPW90, Che89]. Tracing is done on the intersection curve in higher dimensions or on its projection in the plane. Most algorithms use the local geometry of the curve coupled with quasi-Newton's methods [BHHL88, BK90] for tracing. These methods do not converge well sometimes [FF92] and many issues related to choice of step size to prevent *component jumping* are still open. Therefore, most implementations use very conservative step sizes for tracing and this slows down the algorithm. Overall, current tracing algorithms are not considered robust [Sny92].

The components of an intersection curve consist of *open components* and *closed loops* (see Figure 1.4). Start points on the open components are obtained by curve-surface intersections. Developing competitive algorithms for curve-surface intersection problems and detection of closed loops has itself been an active area of research.

Curve-Surface Intersection

The three major approaches for computing curve-surface intersections are based on subdivision, interval arithmetic and algebraic methods. Subdivision based approaches use the geometric properties of curve and surface representations [LR80]. Given two spline curves, the intersection algorithm proceeds by comparing the convex hulls of their control polytopes. Control polytopes and their relation to splines are described in chapter 2. If they do not overlap, the curves or surfaces do not intersect. Otherwise the curves are subdivided and the resulting convex hulls are checked for intersection. At each iteration the algorithm rejects regions of the curve that do not contain any intersection point. Eventually, the curve segments are approximated by straight lines up to a certain tolerance, and their intersection point is accepted as the intersection of two curves. A simple subdivision algorithm has linear convergence in the domain. Its convergence is improved using Bézier clipping [SWZ89, Sed89, NSK90]. Bézier clipping makes use of the convex hull property in a powerful way, by determining parameter ranges which are guaranteed not to include points of intersection.

The interval arithmetic approach is similar to subdivision [KM83]. The curves are divided into intervals using vertical and horizontal tangents which define rectangular bounding boxes. The subdivision amounts to evaluating the coordinate of the midpoint of the interval and defining the resulting rectangles.

Algebraic methods formulate the intersection problem in terms of solutions of a system of algebraic equations. Given the equations, the variables are eliminated using techniques from elimination theory [Sal85] and the problem is reduced to finding roots of a univariate polynomial. This approach was applied to ray-tracing by Kajiya [Kaj82] and to curve intersections by Sederberg [Sed83]. For lower degree curve intersection (up to degree three or four), the implicitization approach results in the fastest algorithms. However, the problem of finding roots of higher degree polynomials can be numerically unstable [Wil59]. Therefore, the overall algorithm for intersection may not be accurate. Moreover, the symbolic expansion of determinants to compute resultants can be computationally expensive [Hof90]. To circumvent these problems, Manocha et. al. [Man92, MD94] have proposed methods combining elimination theory with matrix computations. The resulting problem

is reduced to computing the eigenvalues of a matrix as opposed to roots of a polynomial. Eigenvalue algorithms like the QR algorithm [GL89] are backward stable and as a result the intersections can be computed accurately and efficiently for high degree curves and surfaces.

Loop Detection

As shown in Figure 1.4, the intersection curve of two surfaces can result in a number of different components like closed loops and singularities. While evaluating these curves, it is essential to determine all the components. Loop detection deals with the problem of determining whether an algebraic curve contains closed loops, and if so, where they occur.

There is a considerable amount of work in classic and modern literature related to complete evaluation of algebraic curves. Every algebraic space curve is birationally equivalent to an algebraic plane curve and the latter can be computed using Gröbner bases [Buc89] and resultants. Given an algebraic plane curve, techniques for desingularization based on quadratic transformations are given in [Wal50, Abh90, AB88b]. An excellent introduction to desingularization techniques is provided in [Abh90, Hof89]. However, the resulting algorithm can be exponential in the degree of the curve. Algorithms based on Collins' *cylindrical algebraic decomposition* (CAD), [Col75, ACM84], have been used for evaluating all components of algebraic curves [Arn83, SS83]. However, its worst case complexity is doubly exponential in the number of variables. For plane curves, improved polynomial time algorithms based on CAD have been presented in [AF88, AM88].

The problem of evaluating all the loops of an algebraic curve numerically has been studied in the modeling literature and a number of techniques based on subdivision methods, marching methods and lattice evaluations [Hof89, RR92] have been developed. The subdivision based algorithms subdivide the domain up to a user-specified tolerance and evaluate the curves accordingly [Gei83, LR80, MP93]. No good methods are known for computing a good tolerance value during curve tracing. Thus, most implementations use a conservative value for the tolerance. In general, the two components of a curve can be very close, and as a result, there is a potential danger of merging two isolated components into a single one. Some of the other approaches are based on *lattice evaluation* where the surface-surface intersection problem is simplified to a set of curve-surface intersection problems.

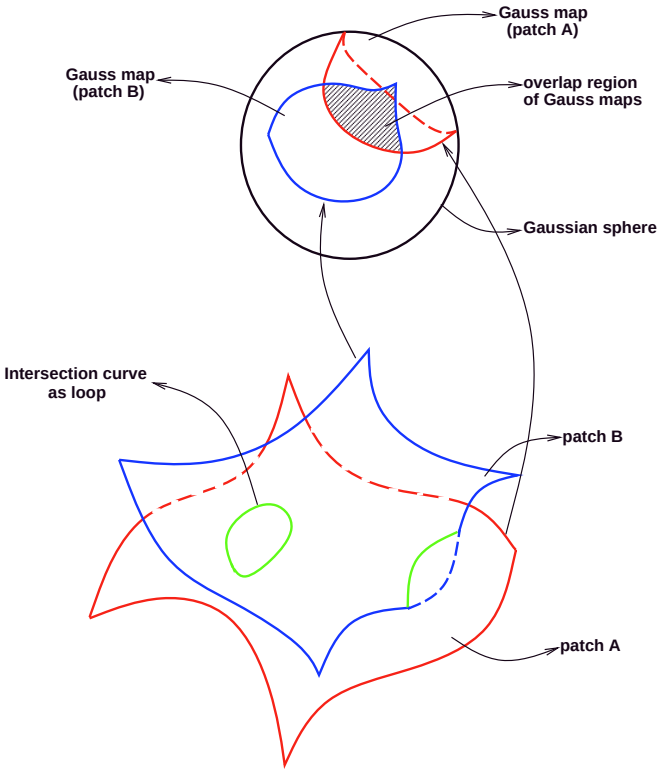

Figure 1.5: Loop detection based on Gauss maps

The curves are obtained by evaluating the surface patch at a number of constant parameter values. The biggest drawback in this approach is the lack of robustness. Small loops could easily be missed depending on the frequency with which the curves are evaluated.

In the last decade, techniques based on curve tracing have been widely used to evaluate high degree curves [BFJP87, BHHL88, KPP90, MC91]. The main idea is to compute at least one point on every component of the curve and use the local geometry of the curve to evaluate successive points. In this class of methods, identifying a point on every loop is significantly harder than identifying a point on open components. As a result, simultaneously with the development of new ideas for evaluating such curves, a number of techniques for loop detection have been proposed [SKW85, SM88, THS89, Che89, Hoh91, Kim90, KPP90, KPW90]. Most of the loop detection criteria are based on bounds on the *Gauss map* of the surfaces being intersected. Sinha et. al. [SKW85] had shown that if two (at least C^1) surfaces intersect in a closed loop, there exists a normal vector on one surface

that is parallel to a normal vector of the other surface. The use of overlaps of Gauss maps in loop detection is illustrated in Figure 1.5. Sederberg et. al. [THS89, SM88] strengthened the above work by proving that if two (at least C^1) surfaces intersect in a closed loop, there exists a line which is perpendicular to both surfaces (collinear normal vectors), provided the inner product between any normal on one surface and any other normal on the other surface is never zero. Patriakalakis et. al. [KPP90] precomputed the most significant points of the intersection curve between an algebraic surface and a parametric patch to identify the main features of the curve. Sederberg et. al. [SM88, ZS93] developed an efficient way to bound the normals and tangents of a surface using *bounding cones* and *pyramidal surfaces*, thereby giving a faster way to achieve the no loop condition. Hohmeyer [Hoh91] bounded the Gauss maps using pseudo-normal patches and used an efficient algorithm for linear programming [Sei90a] to test the separability criterion. In all these algorithms, if the loop detection criterion is satisfied, each surface is divided into a pair of sub-patches and the criterion is recursively tested on each pair combination. This is continued until all patch pairs fail the test. The number of levels of subdivision depends on the tightness of Gauss map bounds [EC94] and curvature variations of the two surfaces. Furthermore, these algorithms may not work well if the intersection curve is self-intersecting.

Techniques based on finding critical points of plane vector fields inside the domain of the surfaces have been proposed by [Che89, KPP90, KPW90, ML95]. Cheng [Che89] defined a plane vector field as the gradient of an oriented distance function of one surface from the other. The critical points are found by following special integral curves that connect all the critical points. [KPP90, KPW90, ML95] use rotational indices of (planar and three-dimensional) vector fields to determine presence of critical points. However, these methods rely on some form of subdivision in the domain, and hence, cannot robustly guarantee detection of all the critical points.

1.3 Thesis Statement

My thesis is

> The *lower dimensional* surface intersection formulation provides an effective representation to perform Boolean operations on sculptured models.

The algorithm's performance on *large real world models* proves this claim. Depending on the representation of the surfaces (implicit or parametric), their intersection curve can lie in three or four dimensions. However, from classic algebraic geometry we know that it is always possible to determine an equivalent curve that lies in a plane. The lower dimensional formulation refers to evaluation of the equivalent plane curve. We shall describe all the algorithms with the use of splines like NURBS which are widely used in most modeling applications. By "large real world models" we mean industrially designed models which have thousands of parts built using Boolean operations.

1.4 Main Contributions

This dissertation presents a number of techniques to effectively compute boundary representations of Boolean combinations of sculptured primitives and perform associated surface interrogations. It employs a combination of symbolic and numeric methods to compute the B-reps accurately and efficiently. The input to our algorithm is a CSG tree that describes the solid as a Boolean expression of primitive solids. In this thesis, we assume that the surface boundaries of all the primitives can be represented as a piecewise collection of parametric surface patches. However, our algorithms apply equally well on solids composed of algebraic surfaces. We use trimmed tensor-product rational Bézier patches (see chapter 2), a special type of NURBS, to represent the surfaces. In order to compute the B-rep of the final solid, our algorithm computes the Boolean combination of the solids at the leaves of the CSG tree and propagates the results up the tree.

Given two such solids, our algorithm identifies pairs of surface patches from the two solids that intersect. The intersection curve between each such pair is computed using a new *surface intersection* algorithm. The surface intersection algorithm ensures accurate evaluation of the intersection curve using algorithms for *curve-surface intersection*, *loop detection* and *curve tracing*. It makes use of a *matrix representation* of the intersection

curve to accurately compute intersections between *trimmed surfaces* and to *classify* the various topological features generated by the intersection curve. The main contributions of this dissertation are briefly described next.

1.4.1 Surface Intersection

Computing intersections of surfaces forms a critical part of any boundary evaluation algorithm. Modelers that perform Boolean operations on polyhedral solids have to deal only with plane-plane intersections. The essential difference between intersecting two planes and two free-form surfaces is that while the former generates a single line, the latter results in a high degree algebraic space curve with a number of components including open components, closed loops and singularities (see Figure 1.4).

The main theme of our approach is to combine well known symbolic and numeric techniques for accurate and efficient computation. Our algorithm borrows a basic theorem of space curves from algebraic geometry. The crux of the theorem is that any algebraic space curve can be projected into an equivalent plane curve after a suitable linear transformation of the coordinates. Using this idea, we obtain a new representation of the intersection curve in a plane in the form of a matrix polynomial. We then evaluate the curve using numeric matrix computations and tracing algorithms. The algorithm guarantees determination of all components of the intersection curve for well-conditioned input cases by employing newly developed algorithms for curve-surface intersection and loop detection. Since all the computation is performed in floating point arithmetic, we evaluate the intersection curve to a user-specified tolerance[1]. The main steps of the algorithm are

- Given the two parametric surfaces, eliminate two of the variables using Dixon's resultant (see chapter 4) and obtain the intersection curve as a matrix polynomial.
- Compute a starting point on each component of the intersection curve using curve-surface intersection and loop detection algorithms.
- Subdivide the domain of the surface into regions such that each sub-region has at most one curve component.

[1] we use 10^{-5} in our implementation

- If the separability condition is not satisfied due to singularities in the intersection curve, use local optimization techniques to isolate singular points within small portions of the domain.

- For each starting point, follow that component of the intersection curve using tracing methods.

Of all these steps, the elimination step dominates the computational cost. However, most of the computation involved in this stage can be performed off-line, and its cost amortized over a large number of surface intersection operations. This is particularly advantageous in boundary evaluation algorithms where the surface intersection routine is called hundreds of times for each solid. We have used our algorithm to generate surface boundaries of models like the submarine storage and handling room (Figure 1.1) and the Bradley fighting vehicle (Figure 6.1). On an average, our algorithm takes a fraction of a second (0.2–0.5 seconds) to perform one surface intersection.

1.4.2 Curve-Surface Intersection

We use curve-surface intersections to evaluate starting points on intersection curves of two surfaces and to perform ray-shooting tests (see chapter 6) to classify surface features with respect to solids. Other applications for this algorithm include ray-tracing and visible surface determination in computer graphics. In all these applications, we are interested in finding intersections only in a small subset of the real domain.

In this dissertation, we introduce a new technique called *algebraic pruning* which uses matrix computations effectively to prune out regions of the domain with no intersections quickly. The basic idea of the algorithm is: Assume that we have an algorithm A which given a guess ρ to an intersection point generates the closest intersection point α. Let the separation between ρ and α be $\delta = \mid \rho - \alpha \mid$. Then, we know that there is no intersection point in the region $(\rho - \delta) < t < (\rho + \delta)$. We can safely prune out this region.

We use inverse power iterations (an iterative matrix computation algorithm) to converge to the closest intersection point. To the best of our knowledge, our algorithm performs faster than previously known curve-surface intersection algorithms when the number

of intersections is fairly sparse. It performs competitively even when the intersection set is not sparse. This algorithm can be used without significant modification for finding zero-dimensional intersection sets like planar curve-curve intersection as well.

1.4.3 Loop Detection

Loop detection in algebraic curves is an important part of any curve evaluation algorithm, and is traditionally considered hard. The reason for this is because searching for such curve features in higher dimensions is difficult. Any discretized search strategy suffers from the possibility of missing small loops. We have devised an efficient algorithm for loop detection based on a simple algebraic characterization. We use the fact that any real algebraic plane curve is continuous in the complex projective plane. Put simply, it means that while curve components appear disjoint when restricted to the real plane, they are actually connected into one single component in the complex plane. Therefore, by following the curve in complex space, we can reach at least one point on every loop component. The overview of the algorithm is described below.

- Evaluate all the starting points of the curve (in complex space) at the boundary of the domain.
- Follow each starting point by tracing out the curve in complex space.
- Few of these paths meet the real plane. These form candidates for loop components of the curve.

Compared to some of the traditional algebraic approaches which exhibit quadratic complexity in terms of the degree of the curve, our method traces out only a linear number of paths (our algorithm takes about 10-20 milliseconds, depending on the length, to trace out a single complex path completely). However, the number of complex paths to be traced could be high depending on the degree of the algebraic curve. This method offers the flexibility of being combined with other heuristics that would limit the number of complex paths traced. Another advantage of our method is that it is general enough to be used with any algebraic curve. We show the working of our algorithm on another algebraic curve - a

silhouette curve of parametric surfaces from a given viewpoint. A silhouette curve locally separates front and back facing portions of a surface from a given viewpoint. Some of the most effective numerical methods previously developed for loop detection are only restricted to curves resulting from the intersection of two parametric surfaces.

In this dissertation, we present another loop detection algorithm that is restricted only to planar surface sections (intersection of a surface by a plane). One important application for obtaining cross sections of solids is in the process of stereolithography for rapid prototyping. The algorithm we present can be implemented in exact arithmetic or finite precision depending on the accuracy demands of the application. The algorithm uses the idea of Sturm sequences (described in chapter 5) to evaluate certain critical points that are always present inside loops. By determining these points and subdividing the surface, we can ensure that no loops will be missed. Our implementation of this algorithm is done only in exact arithmetic, and is thus restricted to low degree surfaces for reasons of efficiency and memory utilization. We believe, however, that this algorithm can be applied extensively if it is implemented in double precision arithmetic.

1.4.4 Trimmed Surface Intersection

Our algorithm for boundary evaluation generates surface boundaries in the form of trimmed NURBS patches. A detailed definition of trimmed NURBS is given in chapter 2. For the moment, it suffices to say that along with the definition of the parametric surface, we also have an *oriented* closed curve called the *trimming curve* in the domain. This trimming curve determines the portion of the patch that is valid. For example, in Figure 1.6, the trimming curve is generated in a counterclockwise sense and the portion of the patch that is on the left of the curve is valid.

Let us look at why we need to have trimming curves for our surfaces. When we perform a Boolean operation (union, intersection or difference) between two solids, their intersection curve determines which part of the original surface belongs to the final solid. If we look in the domain of one of these surfaces, the intersection curve partitions it. Only a few of the partitions are retained in the final solid. For the kind of operations we perform on solids, it is therefore, natural to represent their surface boundaries using trimmed parametric

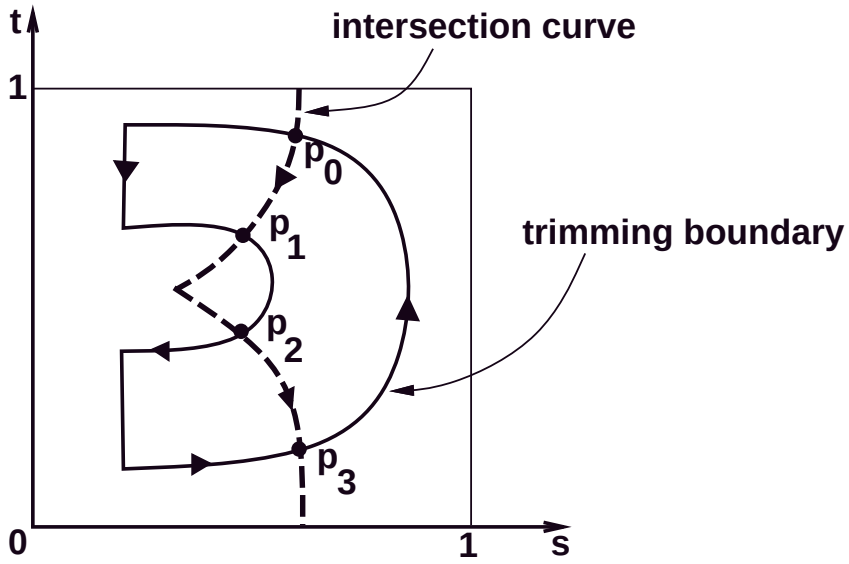

Figure 1.6: Obtaining intersections between trimmed surfaces

patches. Moreover, the trimming curves are portions of intersection curves themselves.

The surface intersection algorithm that we describe in chapter 3 deals with untrimmed parametric surfaces only. Applying this algorithm, only some parts of the intersection curve generated are valid for trimmed surfaces. For example, in Figure 1.6, the valid intersection curve is only between (p_0, p_1) and (p_2, p_3). Generating the p_i's accurately is not an easy problem because it involves intersections of two fairly high degree algebraic curves. The accuracy of these points is crucial because they determine important surface features of the new solid.

We present an efficient and accurate algorithm to generate these intersection points. The algorithm uses the piecewise linear representation (generated by curve tracing) of the intersection and trimming curves to compute approximations for these points. We then use the patch parameterizations of the surfaces involved and the analytic representation of the intersection curve to refine the approximations using iterative minimization techniques. Details of this algorithm are given in chapter 6.

1.4.5 Component Classification

When two solids enter into a Boolean operation, only portions of the surfaces of each solid remain in the final solid. The portions to be retained are determined by the intersection curve between the two solids. For example, consider a union operation between two solids A and B. After computing the intersection curve, only portions of A that lie outside B and those of B that lie outside A are retained in the solid $A \cup B$. Similar characterizations exist for other operations as well. Component classification refers to algorithms that generate maximally connected portions of the boundary π of a solid that have the property that π either lies completely inside or outside (*orientation-invariant component*) the other solid. Furthermore, it also deals with the resolution of the inside/outside nature of each orientation-invariant component.

We use the topological information (connectivity between the various features) of each solid and the intersection curve between them to generate the various orientation-invariant components. Our algorithm creates an associated undirected graph and computes its connected components for this purpose. It also generates another graph whose vertices are the various orientation-invariant components. An edge exists between two such vertices if and only if orientations are opposite with respect to the other solid. This connectivity information turns out to be very useful in classifying the various components efficiently.

When two polyhedral solids intersect, it is fairly easy to classify the inside/outside nature of the various components by performing simple local tests based on the orientation of the intersection curve [Hof89]. However, for solid boundaries composed of curved surfaces, local tests cannot be performed. The main reason for this is the complicated nature of the intersection curve. We use an algorithm based on ray-shooting to perform the classification tests. Ray-shooting is based on the following simple fact: A point is inside a closed solid if any semi-infinite ray originating from that point intersects the boundary of the solid odd number of times; otherwise, it is outside. We use our curve-surface intersection algorithm to perform ray-shooting. Since curve-surface intersection is a fairly expensive operation, it behooves us to reduce the number of such invocations. Our algorithm uses the connectivity information between the various components and performs just one ray-shooting test per

solid per operation. This significantly speeds up our computation.

The accuracy of the ray-shooting test is very important in determining the final solid. Double precision arithmetic or degenerate ray-surface intersections could possibly change a result from inside to outside or vice-versa. We use an analytic representation of the intersection curve and stable matrix computations to prevent such catastrophic errors. More details are presented in chapter 6.

1.5 A Guide to the Chapters

To understand some of our algorithms better, some mathematical background is required. Chapter 2 lays these mathematical foundations and introduces our terminology. Chapter 3 presents methods to compute intersection points when a curve meets another curve or surface. This algorithm is used by our surface-surface intersection algorithm and component classification algorithm. Chapter 4 describes our surface-surface intersection algorithm. It describes our formulation of the intersection curve, and the curve tracing methods used to evaluate it. The problem of identifying closed loops in algebraic curves is discussed in Chapter 5. It also presents our algorithm to identify loops when surfaces are sectioned by a plane. Two additional applications of our loop detection algorithm are presented in Chapter 5. Chapter 6 discusses our algorithm to evaluate curved surface boundaries of Boolean combinations of solids. Our implementation of the various algorithms presented in this dissertation and techniques to parallelize the computation are presented in Chapter 7. Chapter 8 suggests some extensions to this work and concludes this dissertation.

Chapter 2

Mathematical Background

In this chapter, we will briefly introduce some mathematical preliminaries and representation issues that are required to get a better understanding of this thesis.

2.1 Affine and Projective Spaces

In our discussion, we will use both affine and projective spaces.

Definition 1 *Affine n-dimensional space is a space where all points p have coordinates* $p = (x_1, x_2, \ldots, x_n) \in \mathcal{R}^n$, *where all the* x_i*s are always finite.*

Euclidean space is the most familiar form of affine space with the Euclidean distance metric associated with it.

Definition 2 *Projective n-dimensional space* ($\mathcal{P}^n$) *consists of all points with* $(n+1)$ *coordinates* $(x_1, x_2, \ldots, x_n, x_{n+1})$, *where not all* x_i *are zero and all* x_i *are finite. Further, for all* $\alpha \neq 0$, *both* $(x_1, x_2, \ldots, x_n, x_{n+1})$ *and* $(\alpha x_1, \alpha x_2, \ldots, \alpha x_n, \alpha x_{n+1})$ *represent the same point.*

The variable x_{n+1} is called the homogenizing variable. Therefore, these coordinates are also called *homogeneous coordinates*. In all our usage throughout this dissertation, the homogenizing variable is written as the last coordinate. An interesting connection between the two spaces is that projective n-space is the space of all lines in affine $(n+1)$-space that contain the origin. In Figure 2.1, point p is the representative point of line $L = (a_1 t, a_2 t, a_3 t)$. The plane $x_3 = 1$ represents $\mathcal{P}^2$.

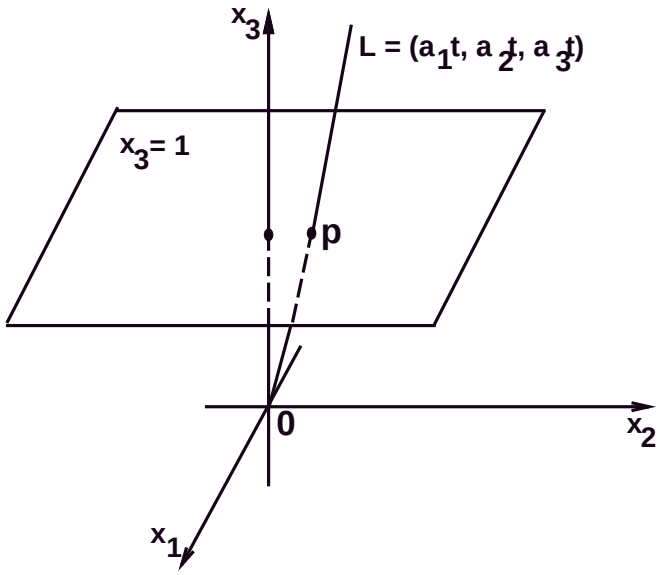

Figure 2.1: Embedding projective space into affine space

2.2 Curve and Surface Representation

Most algebraic curves and surfaces in 3D space can be represented using their implicit form, $f(x, y, z) = 0$. Geometric modeling applications frequently involve computing a set of points on a given curve or surface. But the process of computing points on surfaces with implicit representation is computationally intensive. An alternative representation is the parametric form. For example, a parametric space curve is a mapping from the real line to $\mathcal{R}^3$. The domain of these functions is also called the *parameter* of the curve. By substituting different values for the parameter, we obtain different points on the curve.

A NURBS curve [Far93] is a special kind of parametric curve. This curve is completely specified by a set of points in space and a few smooth functions. These points are called the *control points* of the NURBS curve. The pre-specified functions are called the *basis or blending functions*. The exact forms of these basis functions are given later in this section. The control points and the blending functions are combined mathematically to give rise to a single curve.

The NURBS curve is composed of a number of segments or *spans*. In the parametric domain, these spans are described by a *knot vector*, which is basically a non-decreasing

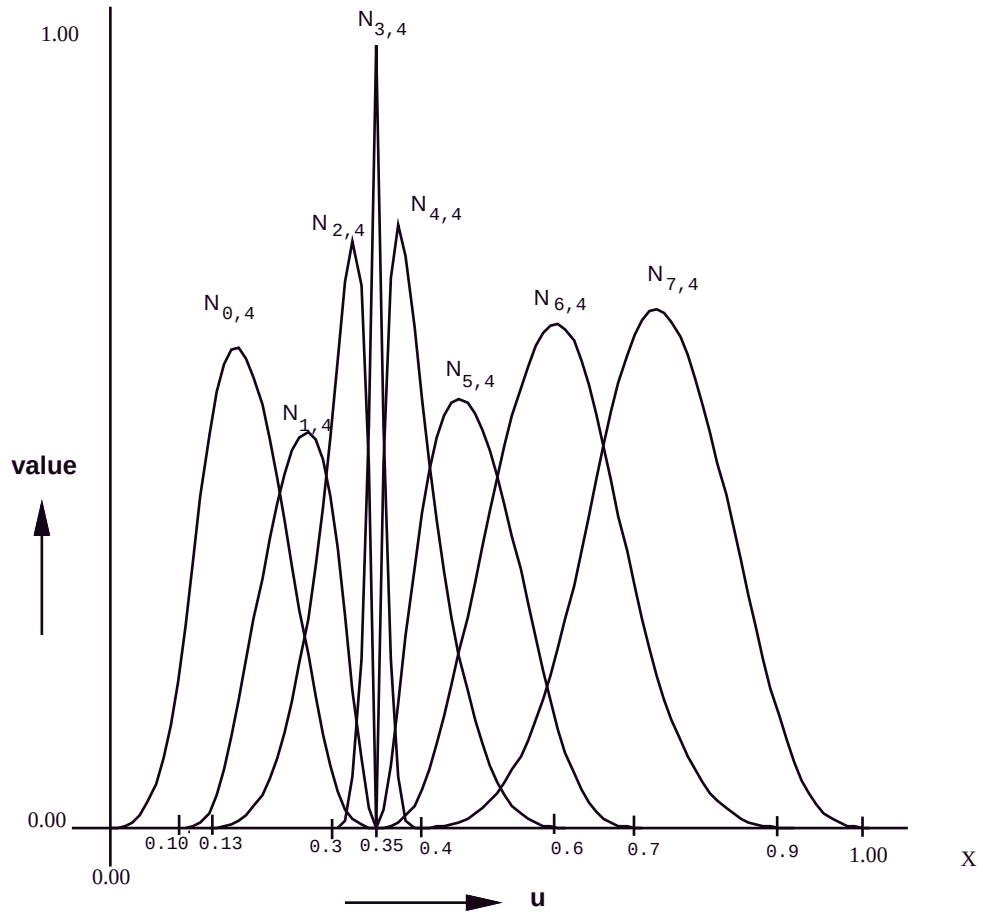

Figure 2.2: B-Spline blending functions for a cubic curve

sequence of parameter values. The knot vector determines the region of influence a particular basis function has on the curve. A NURBS polynomial is defined as a linear combination of basis functions. When the coefficients of the linear combination expression are 4-tuples, the set of four implied polynomials form a curve. Each 4-tuple is a homogeneous representation of a control point in projective 3-space, and the homogenizing variable (4^{th} coordinate) is called a *weight*. In the rest of this dissertation, we assume that the weights are non-negative. Essentially, this assumption ensures that the curve or surface is completely contained within the convex hull of the control points. This is not a major restriction because most curves and surfaces occurring in CAD applications can be represented using non-negative weights.

We shall represent control points in homogeneous coordinates $(\mathbf{v_i}, w_i)$, where $\mathbf{v_i} = (w_i x_i, w_i y_i, w_i z_i)$ and w_i is the weight. Therefore the parametric curve $f(t)$ of degree $k-1$ with n control points and the standard basis functions $\mathcal{N}_{i,k}$ is given by

$$f(t) = \frac{\sum_{i=0}^{n} \mathbf{v_i} \mathcal{N}_{i,k}(t)}{\sum_{i=0}^{n} w_i \mathcal{N}_{i,k}(t)}$$

$\mathcal{N}_{i,k}(t)$ is defined recursively over the knot interval $[t_i, t_{i+k}]$ as

$$\mathcal{N}_{i,1}(t) = \begin{cases} 1, \ if \ t_i \leq t \leq t_{i+1} \\ 0, \ otherwise \end{cases}$$

$$\mathcal{N}_{i,k}(t) = \frac{(t_{i+k} - t)\mathcal{N}_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}} + \frac{(t - t_i)\mathcal{N}_{i,k-1}(t)}{t_{i+k-1} - t_i}$$

Figure 2.2 shows the various blending functions for a cubic NURBS curve.

Based on the above formulation of the parametric curve, it is clear that the control points determine the shape of the curve. Further, since each control point has only a limited range of influence, it is very easy to shape the curve (or surface) by local modification of the control points. In most parametric specifications, the domain is normalized to lie in the unit interval, $t \in [0, 1]$, without loss of generality. We shall stick to this convention throughout this dissertation.

A *tensor product* NURBS surface is defined over a two dimensional parametric domain over the parameters $0 \leq s, t \leq 1$. The shape of the surface is determined by two array of knot vectors (one for each parameter) and a two dimensional array of control points [Far93]. Figure 2.3 shows the relationship between a surface patch and its parametric domain. The weighted sum formulation of a NURBS surface is

$$\mathbf{F}(s,t) = \frac{\sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{v_{ij}} \mathcal{N}_{i,k}(s) \mathcal{N}_{j,l}(t)}{\sum_{i=0}^{m} \sum_{j=0}^{n} w_{ij} \mathcal{N}_{i,k}(s) \mathcal{N}_{j,l}(t)}$$

In this equation, the surface is of degree $k-1$ in s and $l-1$ in t (degree $(k-1)\times(l-1)$, for short). A *trimmed* NURBS surface, $\mathbf{F}'(s,t)$, is a subset of $\mathbf{F}(s,t)$ defined by a set of trimming curves. A trimming curve is a simple, closed, piecewise sequence of curves (linear, NURBS or algebraic) defined in the domain, $\mathcal{D} = [0,1] \times [0,1]$, of $\mathbf{F}(s,t)$. The subset of the domain that is part of the trimmed surface is usually given by a unambiguous rule. For consistency, we shall define a rule that we follow for algorithmic description and implementation purposes.

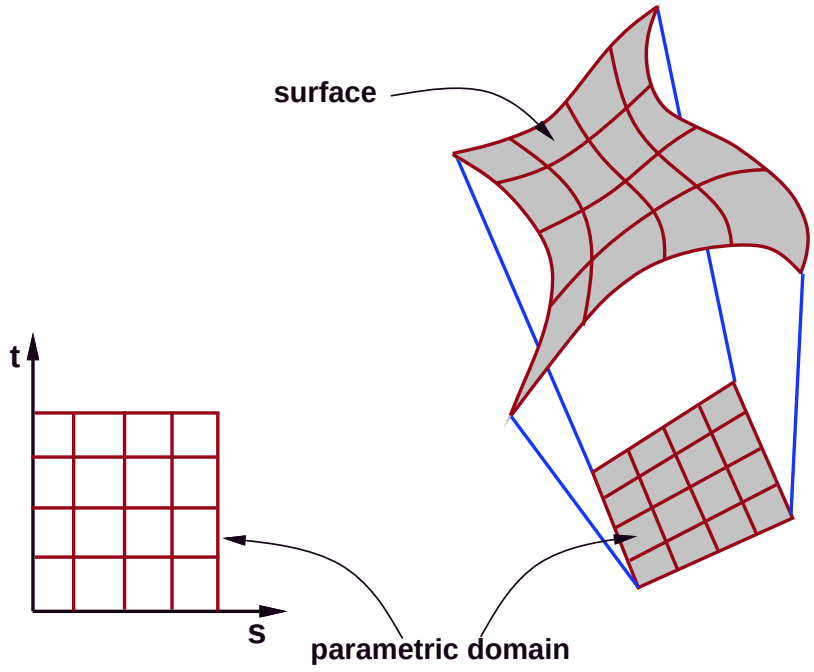

Figure 2.3: A surface patch and its parametric domain

- The trimming curve is oriented counterclockwise when looking into the plane of the paper from above (see Figure 2.4). More precisely, the simply closed trimming curve is homeomorphic to a circle which is oriented counterclockwise.

- The curve retains the part of the surface domain immediately to the left of it. Consider a point q on the curve and a domain point $q^{'}$ arbitrarily close to q (see Figure 2.4). Let the tangent at q be $\vec{\mathbf{t}}$. Then $q^{'} \in \mathcal{D}_{F'}$ is a part of the trimmed surface if the counterclockwise angle between $\vec{\mathbf{t}}$ and $\vec{qq'}$ is less than π.

- Two points q_1 and q_2 belong to the same trimmed region ($\mathcal{D}_{F'}$ or $\mathcal{D} - \mathcal{D}_{F'}$) if and only if the line segment q_1q_2 intersects the trimming curve even number of times (counting multiplicities).

Therefore,

$$\mathbf{F}^{'}(s,t) = \{\mathbf{F}(s,t) \mid (s,t) \in \mathcal{D}_{F'}\}$$

Bézier surfaces are special types of NURBS surfaces, that do not have any knots

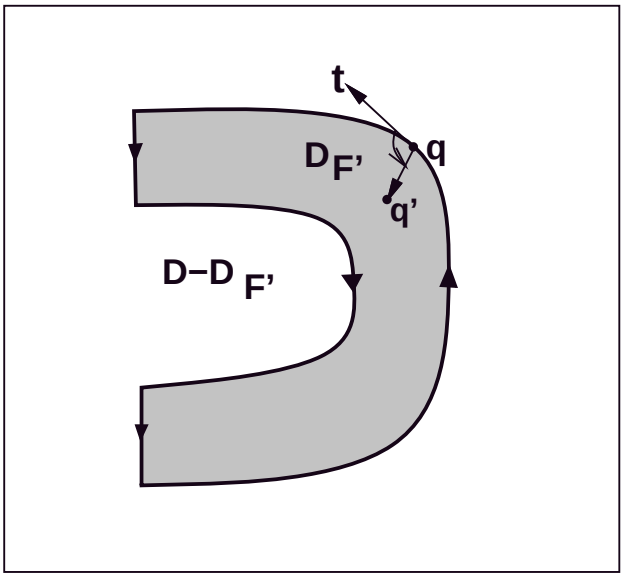

Figure 2.4: Trimming rule

except at the corner points (i.e., $(s,t) = (0,0),(0,1),(1,0),(1,1)$). The multiplicity of s and t knots is one more than s and t degrees, respectively, of the surface. The main advantages of the Bézier representation is that they are more easy to evaluate than general NURBS. Using *knot insertion* algorithms [Far93], it is possible to decompose each NURBS surface into a series of rational Bézier patches. We use Bézier patches to represent boundaries of the solid primitives in our algorithms.

A rational Bézier patch, $\mathbf{F}(s,t)$, of degree $m \times n$, defined in the domain $(s,t) \in [0,1] \times [0,1]$ and specified by a two dimensional array of control points $(\mathbf{v_{ij}}, w_{ij})$ (see Figure 2.5) is given by

$$\mathbf{F}(s,t) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{v_{ij}} \mathcal{B}_i^m(s)\mathcal{B}_j^n(t)}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij} \mathcal{B}_i^m(s)\mathcal{B}_j^n(t)}. \tag{2.1}$$

$\mathcal{B}$ is the Bernstein basis function defined as

$$\mathcal{B}_i^m(s) = \begin{pmatrix} m \\ i \end{pmatrix} s^i (1-s)^{m-i}$$

A trimmed Bézier patch, as shown in Figure 2.6, has trimming curves in the domain of the patch and trims out the domain similar to its NURBS counterpart.

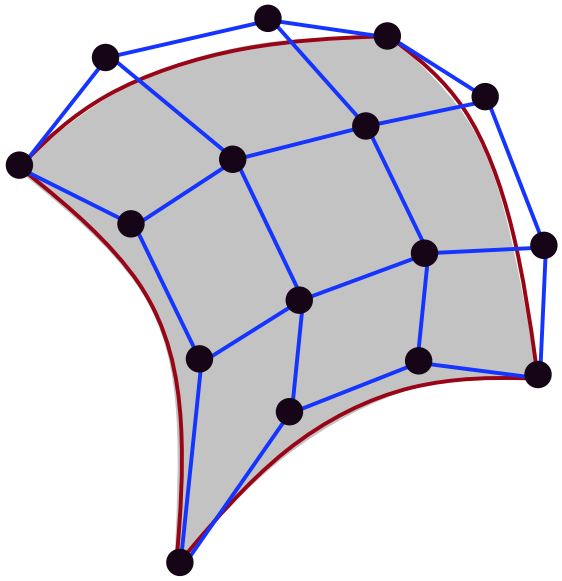

Figure 2.5: Sixteen control points of a bicubic Bezier patch

The partial derivatives of a Bézier patch, $\mathbf{F}(s,t)$, with respect to s and t and denoted by $\mathbf{F_s}$ and $\mathbf{F_t}$, lie in the tangent plane of $\mathbf{F}(s,t)$. If $\mathbf{F_s}$ and $\mathbf{F_t}$ are linearly independent, then the normal direction at (s,t) is given by

$$\mathbf{N}(s,t) = \mathbf{F_s}(s,t) \times \mathbf{F_t}(s,t) \tag{2.2}$$

In the above definition, $\times$ refers to the cross product between two vectors in 3-space. The partial derivatives of a rational Bézier patch are computed as follows. Let the rational form of the patch $\mathbf{F}(s,t)$ be $\frac{\mathbf{V}(s,t)}{W(s,t)}$. Then,

$$\begin{aligned}
\mathbf{V_s}(s,t) &= \textstyle\sum_{i=0}^{m-1}\sum_{j=0}^{n}(\mathbf{v}_{i+1,j} - \mathbf{v}_{i,j})\mathcal{B}_i^{m-1}(s)\mathcal{B}_j^n(t) \\
W_s(s,t) &= \textstyle\sum_{i=0}^{m-1}\sum_{j=0}^{n}(W_{i+1,j} - W_{i,j})\mathcal{B}_i^{m-1}(s)\mathcal{B}_j^n(t) \\
\mathbf{V_t}(s,t) &= \textstyle\sum_{i=0}^{m}\sum_{j=0}^{n-1}(\mathbf{v}_{i,j+1} - \mathbf{v}_{i,j})\mathcal{B}_i^m(s)\mathcal{B}_j^{n-1}(t) \\
W_t(s,t) &= \textstyle\sum_{i=0}^{m}\sum_{j=0}^{n-1}(W_{i,j+1} - W_{i,j})\mathcal{B}_i^m(s)\mathcal{B}_j^{n-1}(t)
\end{aligned}$$

It is easy to note that the above four functions are also in Bézier form. We compute the partial derivatives of the patch using these functions and the quotient rule.

$$\left(\frac{\mathbf{V}}{W}\right)_s = \frac{W\mathbf{V_s} - \mathbf{V}W_s}{W^2}, \quad \left(\frac{\mathbf{V}}{W}\right)_t = \frac{W\mathbf{V_t} - \mathbf{V}W_t}{W^2}$$

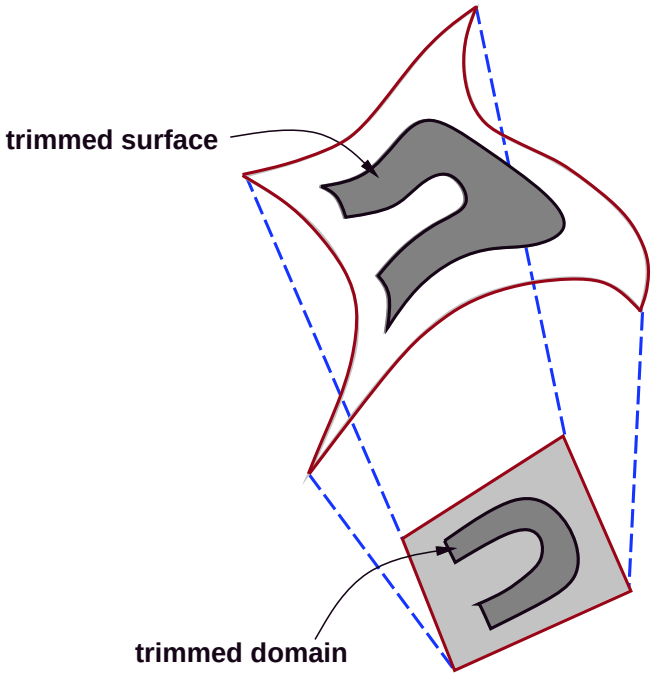

Figure 2.6: A trimmed surface patch

2.2.1 Gauss Maps

Gauss maps provide a convenient way to describe the normals of a surface. We define the Gauss map of a surface with a continuous unit normal vector field as [O'N66]

Definition 3 *The Gauss map* $\mathbf{G_f}$ *of a surface* $\mathbf{F}$*, is a map* $\mathbf{G_f} : \mathbf{F} \rightarrow S^2$*, a sphere embedded in* $\mathcal{R}^3$*, which maps point* $\mathbf{F}(s,t)$ *to the vector* $\mathbf{U}(s,t)$*, translated to the origin, where* $\mathbf{U}(s,t) = \frac{\mathbf{N}(s,t)}{|\mathbf{N}(s,t)|}$.

Therefore, at some point $p = (s,t)$ on the surface, $\mathbf{G_f}(s,t) = \mathbf{U_p}$, where $\mathbf{U_p}$ is a point on the unit sphere centered at the origin $\mathbf{O}$ such that the vector $\mathbf{O}\vec{\mathbf{U}}_\mathbf{p}$ is along the same direction as $\mathbf{N}(s,t)$. The function $\mathbf{G_f}$ can be used to compute the unit normal direction at a given point on the surface. But computing an exact representation of $\mathbf{G_f}$ is quite complicated. Further, the continuous unit normal vector field assumption is not valid for some cases of degenerate parameterizations. Fortunately, in our applications, we are only interested in the direction of the normals. The pseudo-Gauss map, $\mathbf{G} = \mathbf{F_s} \times \mathbf{F_t}$ gives this information. It is called a "pseudo"-Gauss map because the vector field is not normalized

and for certain parameterizations of surfaces, it may be incorrect at finite number of points. The computation of the pseudo-Gauss map is quite efficient and is based on the partial derivative computation described above. Another benefit of using the pseudo-Gauss map is that **G** is itself a Bézier surface, and can thus be described in terms of its control points. Hohmeyer [Hoh91] uses pseudo-Gauss maps to perform efficient loop detection in surface intersection algorithms.

If $\mathbf{F}(s,t)$ has a polynomial parameterization of degree $m \times n$, then the pseudo-normal surface is a degree $(2m-1) \times (2n-1)$ Bézier patch. For rational surfaces, the degree of the cross product is $3m \times 3n$.

2.2.2 Multipolynomial Resultants

Elimination theory investigates the conditions under which sets of polynomials have common roots. Usually, it concerns itself with sets of n homogeneous polynomials in n unknowns, and finds the relationship between the coefficients of the polynomials which can be used to determine whether the polynomials have a non-trivial common solution.

Definition 4 *[Sal85] A* **resultant** *of a set of polynomials is an expression involving the coefficients of the polynomials such that the vanishing of the resultant (evaluating to zero) is a necessary and sufficient condition for the set of polynomials to have a common non-trivial root.*

In this dissertation, we use resultants to compute implicit forms of surfaces from their parametric representation and to detect the presence of loops in our surface intersection algorithm. [Mac02] provided a general method for eliminating n variables from n homogeneous polynomials. The resultant is expressed as a ratio of two determinants. However, a single determinant formulation exists for $n = 2$ and 3 [Sal85, Dix08]. For $n = 3$, however, [Dix08] gives the resultant only if the three equations have the same degree. In our applications, it is sufficient to compute resultants for the cases when $n = 2$ and 3.

Sylvester's method [Sal85] can be used to express the resultant of two polynomials of degree m and n respectively as a determinant of a matrix with $(m+n)$ rows and columns. Given two polynomials,

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0, \tag{2.3}$$

and

$$g(x) = b_m x^m + b_{m-1} x^{m-1} + \ldots + b_1 x + b_0, \tag{2.4}$$

generate m polynomials by multiplying $f(x)$ with $x^i, i = m-1, m-2, \ldots, 1, 0$ and n polynomials by multiplying $g(x)$ with $x^i, i = n-1, n-2, \ldots, 1, 0$. This results in a total of $(m+n)$ polynomials. By treating the $(m+n)$ monomials, $x^i, i = 0, 1, \ldots, m+n-1$ as independent variables, the Sylvester's resultant for the above two polynomials is

$$R_x(f,g) = \begin{vmatrix} a_n & a_{n-1} & \ldots & & a_0 & 0 & \ldots & 0 \\ 0 & a_n & a_{n-1} & \ldots & & a_0 & 0 & \ldots \\ 0 & \ldots & 0 & a_n & a_{n-1} & & \ldots & a_0 \\ 0 & \ldots & & 0 & b_m & b_{m-1} & \ldots & b_0 \\ 0 & \ldots & 0 & b_m & b_{m-1} & \ldots & b_0 & 0 \\ b_m & b_{m-1} & \ldots & b_0 & 0 & \ldots & & 0 \end{vmatrix} \tag{2.5}$$

The problem of computing the implicit representation of a parametric surface $\mathbf{F}(s,t) = (X(s,t), Y(s,t), Z(s,t), W(s,t))$ involves eliminating s and t from the three polynomials

$$X(s,t) - xW(s,t) = 0$$
$$Y(s,t) - yW(s,t) = 0$$
$$Z(s,t) - zW(s,t) = 0$$

We use Dixon's resultant [Dix08] to compute the implicit form. We discuss this in more detail in chapter 3.

The results and algorithms developed in elimination theory assume that the polynomials are described in their monomial basis (like polynomials given in equations (2.3) and (2.4). This assumption is not true in cases like NURBS and Bézier patches where the polynomials are given using other basis functions. For example, tensor product Bézier

patches are given in the Bernstein basis. In order to apply resultant algorithms on these polynomials, we have to convert them into the monomial (or power) basis.

Bernstein to Power Basis: In order to convert from Bernstein to power basis, we perform a reparametrization of the form

$$\overline{s} = g(s) = \frac{s}{(1-s)}, \quad \overline{t} = g(t) = \frac{t}{(1-t)},$$

for tensor product surfaces. In the resulting formulation we substitute $\overline{s} = \frac{s}{1-s}, \quad \overline{t} = \frac{t}{1-t}$ and the resulting parametrizations are in power basis in terms of $\overline{s}$ and $\overline{t}$. For tensor product Bézier surfaces, like the one defined in equation (2.1), the new formulation becomes

$$\mathbf{F}(s,t) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n}\begin{pmatrix} m \\ i \end{pmatrix}\begin{pmatrix} n \\ j \end{pmatrix}\mathbf{v_{ij}}\overline{s}^i\overline{t}^j}{\sum_{i=0}^{m}\sum_{j=0}^{n}\begin{pmatrix} m \\ i \end{pmatrix}\begin{pmatrix} n \\ j \end{pmatrix}w_{ij}\overline{s}^i\overline{t}^j}.$$

The domain of the surfaces are suitably transformed. The reparameterization strategy can be applied to transform other basis functions to the power basis as well. Two observations are made regarding these transformations.

- Bounded domains get transformed to unbounded domains. For example, the unit square domain is mapped to the positive quadrant of the real plane.
- Computations may become numerically unstable near the boundary values of the parameters. This is the case in tensor product Bézier patches when s and t are close to 1.0.

To avoid these problems, we apply this transformation only during phases of the algorithm where resultant algorithms are used. Once that is done, we apply the inverse transformation to restore the stability of the Bernstein basis [FR87].

2.3 Sturm Sequences

Let $f(x)$ be a polynomial of degree n,

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0, \quad a_n \neq 0 \tag{2.6}$$

We construct a sequence of polynomials $f_i(x), i = 0, 1, \ldots, m$, of descending degrees. By examining the number of *sign changes*, $w(a)$, for certain points $x = a$, we can determine the number of real roots of $f(x)$ in a specified region. Such a sign change happens whenever the sign of a polynomial value differs from that of its successor. Furthermore, if $f_i(a) = 0$, then this entry is removed from the sequence before counting the sign changes. Suitable sequences of polynomials are called *Sturm sequences*.

Definition 5 *[SB93] The sequence*

$$f_0(x) = f(x), f_1(x), \ldots, f_m(x) \tag{2.7}$$

of real polynomials is a Sturm sequence for the polynomial $f(x)$ if:

- *All real roots of $f_0(x)$ are simple.*
- *sign $f_1(\xi)$ = - sign $f_0'(\xi)$ if ξ is a real root of $f_0(x)$.*
- *For $i = 1, 2, \ldots, m-1$,*

  $$f_{i+1}(\xi) f_{i-1}(\xi) < 0$$

  if ξ is a real root of $f_i(x)$.
- *The last polynomial $f_m(x)$ has no real roots.*

For such Sturm sequences we have the following theorem that we state without proof.

Theorem 1 *[SB93] The number of real roots of $f(x) \equiv f_0(x)$ in the interval $[a, b)$ equals $w(b) - w(a)$, where $w(x)$ is the number of sign changes of a Sturm sequence $f_0(x), f_1(x), \ldots, f_m(x)$ at location x.*

There is a simple recursive algorithm to construct one such Sturm sequence for the polynomial $f(x)$, provided it has only simple real roots. We define the first two polynomials in the sequence as

$$f_0(x) \ := \ f(x), \quad f_1(x) \ := \ -f_0^{'}(x) \ = \ -f^{'}(x)$$

The remaining polynomials $f_{i+1}(x)$ are defined recursively as the remainder when $f_{i-1}(x)$ is divided by $f_i(x)$.

$$f_{i-1}(x) \ = \ g_i(x)f_i(x) \ - \ c_i f_{i+1}(x), i = 1, 2, \ldots,$$

where c_i is a positive constant. Further, the degrees of the polynomials obtained should form a strictly decreasing sequence. This algorithm is the well-known *Euclid's algorithm* for obtaining the greatest common divisor of two polynomials. Because of the decreasing degree condition, the sequence must terminate after at most $m \leq n$ steps.

The main use of Sturm sequence in combination with bisection methods is to isolate roots of polynomials in a given domain. They are also used to find eigenvalues of *Hermitian tridiagonal symmetric* matrices. For a system of multivariate polynomials with a discrete set of roots, there are extensions of Sturm sequences. Chapter 5 describes one such technique. We will use multivariate Sturm sequences in evaluating certain critical points for loop detection.

2.4 Algebraic Curves

In this section, we describe properties of algebraic curves in the context of curve evaluation that we use in the remainder of this dissertation. An algebraic curve in $\mathcal{R}^{n+1}$ can be expressed as a solution of n affine polynomial equations in $(n+1)$ unknowns.

$$\begin{aligned} F_1(u_1, u_2, \ldots, u_{n-1}, u, v) &= 0 \\ F_2(u_1, u_2, \ldots, u_{n-1}, u, v) &= 0 \\ &\vdots \\ F_n(u_1, u_2, , \ldots, u_{n-1}, u, v) &= 0. \end{aligned} \tag{2.8}$$

The functions F_i, $i = 1, 2, \ldots, n$, are the components of a vector field $F : D \rightarrow \mathcal{R}^n$, $D \subset \mathcal{R}^{n+1}$. In this context, we are only interested in evaluating all the components of the curve inside the region $D = [U_{11}, U_{12}] \times [U_{21}, U_{22}] \times \ldots \times [U_{n-1,1}, U_{n-1,2}] \times [U_1, U_2] \times [V_1, V_2] \in \mathcal{R}^{n+1}$. The solution to the problem are elements of D that map to the *zero* vector under F. This can be illustrated by taking the example of parametric surface intersection. Given two Bézier surfaces,

$$\begin{aligned} \mathbf{F}(s,t) &= (X(s,t), Y(s,t), Z(s,t), W(s,t)) \\ \mathbf{G}(u,v) &= (\overline{X}(u,v), \overline{Y}(u,v), \overline{Z}(u,v), \overline{W}(u,v)) \end{aligned}$$

represented in homogeneous coordinates, their intersection curve is defined as the set of common points in 3-space and is given by the vector equation $\mathbf{F}(s,t) = \mathbf{G}(u,v)$. This results in the following set of three equations in four unknowns.

$$\begin{aligned} F_1(s,t,u,v) = X(s,t)\overline{W}(u,v) - \overline{X}(u,v)W(s,t) &= 0 \\ F_2(s,t,u,v) = Y(s,t)\overline{W}(u,v) - \overline{Y}(u,v)W(s,t) &= 0 \\ F_3(s,t,u,v) = Z(s,t)\overline{W}(u,v) - \overline{Z}(u,v)W(s,t) &= 0, \end{aligned} \tag{2.9}$$

and the domain of the intersection curve is $(s,t,u,v) \in [0,1] \times [0,1] \times [0,1] \times [0,1]$.

The degree of an algebraic curve is a fairly accurate measure of its complexity. Geometrically, the degree of a space curve is the maximum number of intersections it has with any plane. Algebraically, the degree of a space curve is the degree of the implicit equation of its projection onto a plane. Bezout's theorem is an important theorem that relates the degree of an algebraic curve with the degrees of the surfaces that intersect to produce the curve.

Theorem 2 Bezout's Theorem: *Two algebraic surfaces of degree d_1 and d_2, respectively, intersect in an algebraic curve of degree $d_1 d_2$ unless they have a common component.*

Sederberg [Sed83] developed a computational method based on resultants to compute the implicit form of any parametric surface. For the special case of a degree $m \times n$ tensor product Bézier surface, Sederberg showed that its implicit form is of degree $2mn$.

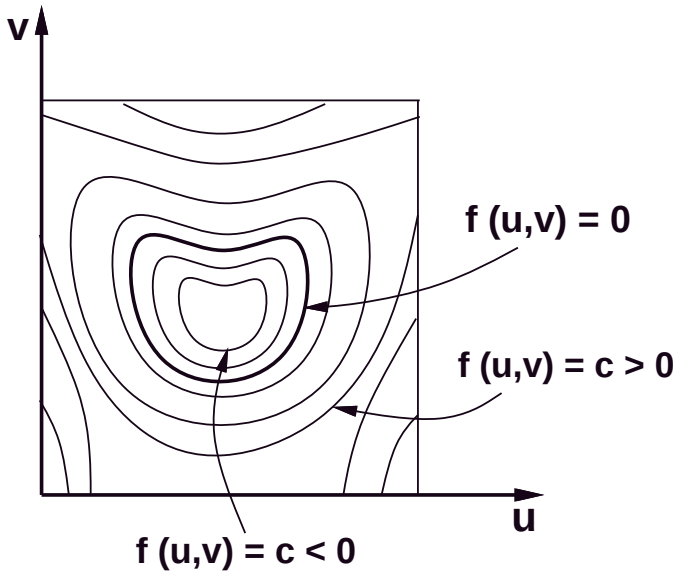

Figure 2.7: Isocontours of bivariate polynomial

Consider two rational Bézier surfaces to degree $m_1 \times n_1$ and $m_2 \times n_2$ respectively. Using the previous fact, we know that the algebraic degree of these surfaces are $2m_1n_1$ and $2m_2n_2$. Bezout's theorem states that the intersection of these surfaces results in a curve of degree $4m_1n_1m_2n_2$. In most modeling applications, it is typical to design bicubic (degree 3×3) Bézier surfaces to preserve smoothness constraints. Intersection of two bicubic Bézier surfaces results in an algebraic space curve of degree 324.

Equation (2.9) gives an intersection curve formulation in parametric space. Using resultant techniques, it is possible to project this curve onto a plane (domain of one of the patches actually - see chapter 3). In this case, we obtain an implicit equation of the plane curve of the form $f(u, v) = 0$. The degree of this curve is $2m_1n_1(m_2 + n_2)$. For the bicubic case, this expression evaluates to 108.

Tangents to algebraic curves: Numerical curve tracing methods are one of the more popular techniques to evaluate algebraic curves. These methods usually evaluate the tangent at some point on the curve and step along the tangent using a step size. This approximant is refined to estimate the new curve point.

Given a plane curve, $f(u, v) = 0$, we want to find the tangent at $p = (u', v')$ on

it. Consider the surface $w = f(u, v)$. The curve produced by cutting the surface with a plane $w = constant$ is called an isocurve. The curve $f(u, v) = 0$ is the special case when the constant is zero (see Figure 2.7). It is a well known fact that the direction of greatest change of a function is along the *gradient* of the function. Further, the gradient is along the normal to the isocurves at all times. Therefore, the normal vector $\vec{\mathbf{n}}(u', v')$ is given by

$$\vec{\mathbf{n}}(u', v') = \vec{\nabla} f = (f_u, f_v)(u', v')$$

For plane curves, the tangent is obtained easily from the normal vector. For the above normal vector, the tangent is $(-f_v, f_u)(u', v')$. We use this definition to evaluate tangent vectors of intersection curves in chapter 3.

2.5 Matrix Computations

Given an $n \times n$ matrix $\mathbf{A}$, its eigenvalues and eigenvectors satisfy the equation

$$\mathbf{A}\mathbf{x} = s\mathbf{x},$$

where s is an eigenvalue and $\mathbf{x} \neq \mathbf{0}$ is the corresponding eigenvector. The eigenvalues of a matrix are also the roots of its characteristic polynomial, Determinant$(\mathbf{A} - s\mathbf{I})$. Given $n \times n$ matrices, $\mathbf{A}$ and $\mathbf{B}$, the generalized eigenvalue problem is

$$\mathbf{A}\mathbf{x} = s\mathbf{B}\mathbf{x}.$$

where s is an eigenvalue and $\mathbf{x} \neq \mathbf{0}$ is the corresponding eigenvector. The eigenvalues of the matrix pencil $(\mathbf{A}, \mathbf{B})$ are all elements s such that Determinant$(\mathbf{A} - s\mathbf{B}) = 0$. If $\mathbf{B}$ is non–singular, the problem can be reduced to a standard eigenvalue problem by multiplying both sides of the equation by $\mathbf{B}^{-1}$ and thereby obtaining

$$\mathbf{B}^{-1}\mathbf{A}\mathbf{x} = s\mathbf{x}.$$

When $\mathbf{B}$ has a high condition number, such a reduction may be numerically unstable. Standard algorithms for computing eigenvalues, like the QR algorithm for the standard eigenvalue problem and QZ algorithm for the generalized eigenvalue problem, are based on orthogonal similarity transformations [GL89].

We reduce our problems of intersection computation to computing certain eigenvalues of a matrix pencil (a parameterized matrix form). Since the QR or QZ algorithm compute all the eigenvalues, they can be inefficient. Iterative techniques based on the power method are more efficient because they find only certain eigenvalues.

2.5.1 Power Method

The Power method computes certain eigenvalues and eigenvectors of a matrix. Let $\mathbf{A}$ be a diagonalizable matrix such that $\mathbf{X^{-1}AX} = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ where $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$ and $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$. $diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ is a diagonal matrix with λs as its diagonal entries. Given a unit vector $\mathbf{q}_0$, the *power method* produces a sequence of vectors $\mathbf{q}_k$.

for $k = 1, 2, \ldots$

$\mathbf{z}_k = \mathbf{A}\mathbf{q}_{k-1}$

$\mathbf{q}_k = \mathbf{z}_k / \parallel \mathbf{z}_k \parallel_\infty$

$s_k = \mathbf{q}_k^T A \mathbf{q}_k$

end

where $\parallel \mathbf{z}_k \parallel_\infty$ is the element of maximum magnitude in the vector $\mathbf{z}_k$. Using exact arithmetic power method, it is known that s_k converges to λ_1 and $\mathbf{q}_k$ converges to $\mathbf{x}_1$, the eigenvector associated with λ_1 provided $\mathbf{q}_0$ is not orthogonal to $\mathbf{x}_1$. Moreover, the asymptotic convergence rate is $|\lambda_1|/|\lambda_2|$. λ_1 is the dominant eigenvalue of $\mathbf{A}$. The power method is described in detail in [GL89, Wil65].

In our applications, we use power iterations to compute the smallest eigenvalues (in magnitude) of matrix pencils of the form, $\mathbf{A}s' + \mathbf{B}$. The smallest eigenvalue of $\mathbf{A}s' + \mathbf{B}$ corresponds to the largest eigenvalue of $(\mathbf{A}s' + \mathbf{B})^{-1}$. Instead of computing the inverse explicitly, given $\mathbf{q}_0$, inverse power iteration solves a linear system of equations,

for $k = 1, 2, \ldots$

Solve $(\mathbf{A}s' + \mathbf{B})\mathbf{z}_k = \mathbf{A}\mathbf{q}_{k-1}$

$\mathbf{q}_k = \mathbf{z}_k / \parallel \mathbf{z}_k \parallel_\infty$

$$s_k = -(\mathbf{q}_k^T \mathbf{B}\mathbf{q}_k)/(\mathbf{q}_k^T \mathbf{A}\mathbf{q}_k)$$

end

where $\| \mathbf{z}_k \|_\infty$ is the element of maximum magnitude of $\mathbf{z}_k$. We compute an LU decomposition of $\mathbf{A}s' + \mathbf{B}$ using Gaussian elimination. The vector $\mathbf{q}_0$ is chosen randomly. Given $\mathbf{A}\mathbf{q}_{k-1}$, the resulting triangular linear system can be solved in $O(n^2)$ steps. Assume that the eigenvalues $(\lambda_1, \lambda_2, \ldots, \lambda_n)$ of $\mathbf{A}s' + \mathbf{B}$ can be ordered such that

$$|s' - \lambda_1| < |s' - \lambda_2| \leq, \ldots, \leq |s' - \lambda_n|.$$

The asymptotic convergence rate is $|s' - \lambda_1|/|s' - \lambda_2|$. If two eigenvalues, λ_1 and λ_2, are almost at the same distance from s', the convergence can be slow. The convergence can be further improved using the following procedure (given $\mathbf{q_0}$ and $\mathbf{u}_0$).

for $k = 1, 2, \ldots$

$$\textit{Solve } (\mathbf{A}s' + \mathbf{B})\mathbf{z}_k = \mathbf{A}\mathbf{q}_{k-1}$$

$$\textit{Solve } (\mathbf{A}s' + \mathbf{B})^T\mathbf{v}_k = \mathbf{A}\mathbf{u}_{k-1}$$

$$\mathbf{q}_k = \mathbf{z}_k / \| \mathbf{z}_k \|_\infty$$

$$\mathbf{u}_k = \mathbf{v}_k / \| \mathbf{v}_k \|_\infty$$

$$s_k = -(\mathbf{u}_k^T \mathbf{B}\mathbf{q}_k)/(\mathbf{u}_k^T \mathbf{A}\mathbf{q}_k)$$

end

In exact arithmetic, this process is locally cubically convergent [Wil65]. Many other techniques for improving the accuracy and convergence of the algorithm in the presence of higher multiplicity eigenvalues or closely spaced eigenvalues are presented in [Wil65].

2.5.2 QR Algorithm

The QR algorithm computes eigenvalues of a matrix. The basic QR algorithm makes use of the *Schur Normal Form*. Schur's theorem states that

Theorem 3 *[SB93] For every $n \times n$ matrix* $\mathbf{A}$ *there is a unitary $n \times n$ matrix* $\mathbf{U}$ *such that*

$$\mathbf{U^H A U} = \begin{pmatrix} \lambda_1 & * & \dots & * \\ 0 & \lambda_2 & \dots & * \\ & & \vdots & \\ 0 & \dots & 0 & \lambda_n \end{pmatrix}$$

The diagonal elements are the eigenvalues of $\mathbf{A}$.

$\mathbf{U^H}$ is the conjugate transpose of $\mathbf{U}$, *i.e.,* for real matrices, $\mathbf{U^H} = \mathbf{U^T}$. A matrix $\mathbf{U}$ is *unitary* if $\mathbf{U^H U} = \mathbf{I}$, the identity matrix. Two matrices $\mathbf{A}$ and $\mathbf{B}$ are *similar* if $\mathbf{B} = \mathbf{T^{-1} A T}$, for some nonsingular matrix $\mathbf{T}$. Similar matrices have the same eigenvalues. The transformation $\mathbf{T}$ is called a *similarity transformation.*

The basic QR algorithm can be written as

Given $\mathbf{A} \in \mathcal{R}^{n \times n}$, define $\mathbf{A_1} = \mathbf{A}$.

For $k = 1, 2, \dots,$ do

Calculate the QR decomposition $\mathbf{A_k} = \mathbf{Q_k R_k}$,

Define $\mathbf{A_{k+1}} = \mathbf{R_k Q_k}$.

In the above algorithm, $\mathbf{Q_k}$ is an orthonormal matrix ($\mathbf{Q_k^T Q_k} = \mathbf{I}$) and $\mathbf{R_k}$ is an upper triangular matrix. The iterates $\mathbf{A_k}$ are *similar* to each other because

$$\mathbf{A_{k+1}} \;=\; \mathbf{R_k Q_k} \;=\; \mathbf{Q_k^T A_k Q_k}$$

The basic idea of QR iteration is that if these transformations are carried out enough times, the upper triangular matrix will eventually have the eigenvalues in its diagonal elements. Computing a QR decomposition of a general matrix requires ($O(n^3)$ operations) per iteration. To reduce the operation count, we use similarity transformations to convert $\mathbf{A}$ to an upper Hessenberg matrix.

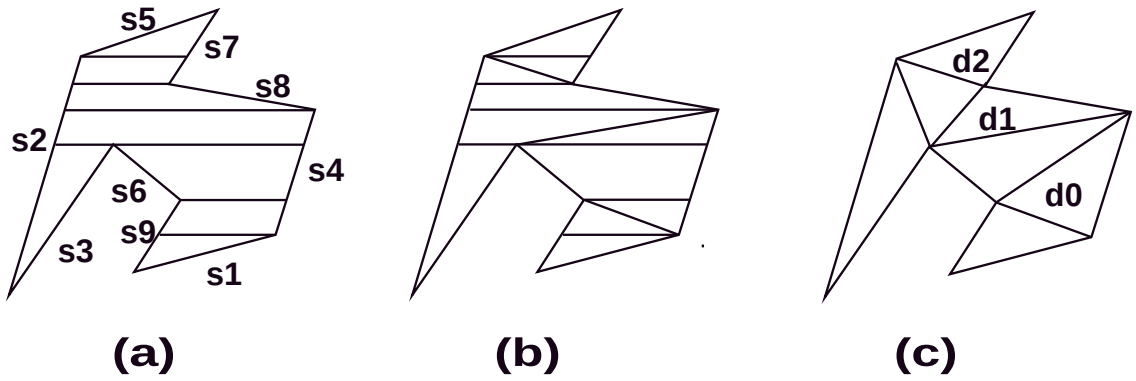

Figure 2.8: Seidel's algorithm for polygon triangulation

A matrix $\mathbf{H}$ is in *upper Hessenberg* form if its elements $h_{ij} = 0$ for all $j \leq i - 2$, i.e.,

$$\mathbf{H} = \begin{pmatrix} * & * & \dots & * & * \\ * & * & \dots & * & * \\ 0 & * & \dots & * & * \\ 0 & 0 & * & \dots & * \\ & & \vdots & & \\ 0 & \dots & 0 & * & * \end{pmatrix}$$

Computing a QR decomposition of an upper Hessenberg matrix (using Given's rotations) requires only $O(n^2)$ operation [SB93]. The QR decomposition of an upper Hessenberg matrix yields an orthogonal component $\mathbf{Q}$ which is also upper Hessenberg. Therefore, the basic QR algorithm preserves upper Hessenberg form.

2.6 Seidel's algorithm for polygon triangulation

Seidel's algorithm [Sei91] is an incremental randomized algorithm to compute the trapezoidal decomposition induced by a set of n lines segments in 2D. The expected time complexity of this algorithm in $O(n \log^* n)$. This algorithm can be used for fast polygon triangulation and, as a by-product, produces a query structure which can be used to answer point-location queries in $O(\log n)$ time.

The algorithm proceeds in three steps as described below (shown in fig. 2.8).

- **Trapezoidation of the polygon:** Let S be a set of non-horizontal, non-intersecting line segments of the polygon . A randomized algorithm is used to create the trapezoidal decomposition of the $X-Y$ plane arising due the segments of set S. This is done by taking a random ordering $s_1, \ldots, s_n$ of the segments in S and adding one segment at a time to incrementally construct the trapezoids. This divides the polygon into trapezoids (which can degenerate into a triangle if any of the horizontal segments of the trapezoid is of zero length). The restriction that the segments be non-horizontal is necessary to limit the number of neighbors of any trapezoid. However, no generality is lost due to this assumption as it can be simulated using lexicographic ordering. That is, if two points have the same Y-coordinate then the one with larger X-coordinate is considered *higher*. The number of trapezoids is linear in the number of segments. Seidel proves that if each permutation of $s_1, \ldots, s_n$ is equally likely then trapezoid formation takes $O(n \log^* n)$ expected time.

- **Decomposition of the trapezoids into monotone polygons:** A monotone polygon is a polygon whose boundary consists of two Y-monotone chains. These polygons are computed from the trapezoidal decomposition by checking whether two vertices of the original polygon lie on the same side of the horizontal line. This is a linear time operation.

- **Triangulation of monotone polygons:** A monotone polygon can be triangulated in linear time by using a simple greedy algorithm which repeatedly cuts off the convex corners of the polygon [FM84]. Hence, all the monotone polygons can be triangulated in $O(n)$ time.

In our algorithm for boundary evaluation, we represent the trimming boundary of a surface patch as a simple polygon. During ray shooting and trimmed surface intersection operations, we have to perform point location queries in the trimmed domain. We use a fast implementation [NM95] of Seidel's algorithm for this purpose.

Chapter 3

Curve Surface Intersection

The problems of computing the intersection of curves and surfaces are fundamental in computer graphics and geometric modeling. Common applications include surface-surface intersection, ray-tracing, hidden-curve removal and visibility algorithms [Hof89, EC90, NSK90, SP86]. Our surface-surface intersection algorithm (chapter 4) needs starting points on each component of the intersection curve. We use curve-surface intersection to evaluate these starting points. Our algorithm for boundary evaluation relies on a ray-shooting approach for the classification of certain solid features. Ray-shooting can be reduced to a series of ray-surface intersection tests. In this chapter, we provide an efficient, accurate and general algorithm to solve curve-surface intersection problems. We assume that problems like curve-curve intersection, curve-surface intersection and ray-surface intersection result only in a zero dimensional intersection set. Chapter 4 deals with intersection problems that result in algebraic curves.

3.1 Intersection Problems and Algebraic Formulation

We only consider the intersections of rational parametric and algebraic curves and surfaces. These include Bézier curves and surfaces, NURBS, quadric patches etc. A rational Bézier plane curve is represented as:

$$\mathbf{P}(t) = (X(t), Y(t)) = \frac{\Sigma_{i=0}^{n} \mathbf{P}_i B_i^n(t)}{\Sigma_{i=0}^{n} w_i B_i^n(t)}, \qquad 0 \leq t \leq 1$$

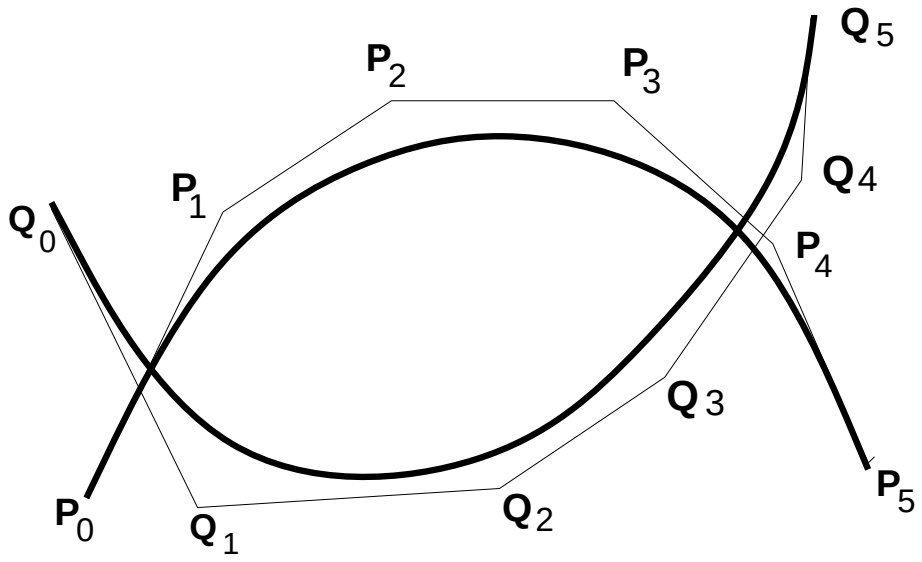

Figure 3.1: Intersection of Bézier curves

where $\mathbf{P}_i = (w_i X_i, w_i Y_i)$ are the coordinates of a control point, w_i is the weight of the control point and $B_i^n(t)$ corresponds to the Bernstein polynomial. Other rational formulations like B-splines can be converted into a series of rational Bézier curves by knot insertion algorithms. Algebraic plane curves (of degree n) are generally expressed in standard power basis:

$$F(x, y) = \Sigma_{i+j \leq n} c_{ij} x^i y^j = 0.$$

They can also be represented in Bernstein basis. The problem of intersection corresponds to computing the common points on such curves in a particular domain.

The parametric surfaces may correspond to tensor product Bézier patches or NURBS patches. Their representations were discussed in chapter 2.

The problem of intersection can always be reduced to solving a system of algebraic equations. For example, given the homogeneous representation of two rational Bézier curves, $\mathbf{P}(s) = (X(s), Y(s), W(s)))$ and $\mathbf{Q}(t) = (\overline{X}(t), \overline{Y}(t), \overline{W}(t)))$, the problem of intersection corresponds to computing all the common solutions of

$$\begin{aligned} X(s)\overline{W}(t) - \overline{X}(t)W(s) &= 0 \\ Y(s)\overline{W}(t) - \overline{Y}(t)W(s) &= 0 \end{aligned} \tag{3.1}$$

in the domain $(s,t) \in [0,1] \times [0,1]$.

Given a Bézier surface $\mathbf{F}(s,t) = (X(s,t), Y(s,t), Z(s,t), W(s,t))$, its intersections with a ray represented as the intersection of two planes

$$a_1 X + b_1 Y + c_1 Z + d_1 = 0$$

and

$$a_2 X + b_2 Y + c_2 Z + d_2 = 0$$

can be reduced to the solutions of

$$\begin{aligned} a_1 X(s,t) + b_1 Y(s,t) + c_1 Z(s,t) + d_1 W(s,t) &= 0 \\ a_2 X(s,t) + b_2 Y(s,t) + c_2 Z(s,t) + d_2 W(s,t) &= 0 \end{aligned} \tag{3.2}$$

in the domain $(s,t) \in [0,1] \times [0,1]$.

Given a Bézier space curve, $\mathbf{P}(u) = (\overline{X}(u), \overline{Y}(u), \overline{Z}(u), \overline{W}(u))$, its intersections with the Bézier surface $\mathbf{F}(s,t)$ can be formulated as all solutions of

$$\begin{aligned} X(s,t)\overline{W}(u) - \overline{X}(u)W(s,t) &= 0 \\ Y(s,t)\overline{W}(u) - \overline{Y}(u)W(s,t) &= 0 \\ Z(s,t)\overline{W}(u) - \overline{Z}(u)W(s,t) &= 0 \end{aligned} \tag{3.3}$$

in the domain $(s,t,u) \in [0,1] \times [0,1] \times [0,1]$.

Accuracy: It is clear that in all these formulations, we are trying to solve a system of equations. Consider the system of equations described by equation (3.3). Let the three equations be denoted by $f_1(s,t,u), f_2(s,t,u)$ and $f_3(s,t,u)$. Another way of looking at these equations is a vector field $F : \mathcal{R}^3 \rightarrow \mathcal{R}^3$ such that

$$F : (s,t,u) \rightarrow (f_1(s,t,u), f_2(s,t,u), f_3(s,t,u))$$

The ideal (and accurate) solutions are those domain points which map to the zero vector under F. Since we are working in finite precision arithmetic, it is not possible to recover these values exactly. Instead, we settle for domain points that map under F to a vector whose norm is smaller than a user-specified value. In all the algorithms presented in this

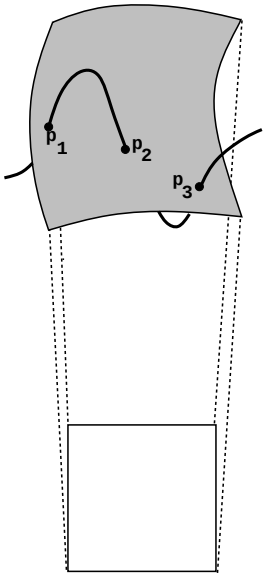

Figure 3.2: Intersection of a Bézier curve and surface

dissertation, we use the Euclidean norm for this purpose. A consequence of this condition is that while it includes all the roots of the system, it may also include spurious intersection points which satisfy the criterion. Unfortunately, as long as we are using finite-precision arithmetic we will never be able to tell the difference. By using exact rational arithmetic and algebraic methods like Sturm sequences (see chapter 2), we may be able to eliminate them.

For the rest of this dissertation, we will use this definition of accuracy for all our algorithms.

3.1.1 Reduction to Eigenvalue Formulation

Given a system of equations, we eliminate variables using resultants. In intersection problems, we obtain systems consisting of two or three algebraic equations. For two equations corresponding to curve-curve intersection and ray-tracing we use Sylvester resultant [Sal85], and for curve-surface intersections we use Dixon's formulation [Dix08]. In either case the resultant can be expressed as a matrix determinant and the entries of the matrix are univariate polynomials. Such matrices are called *matrix polynomials*. Instead

of symbolically expanding the determinant, we reduce the problem to an eigenvalue formulation [Man92]. In particular, the resultant corresponds to a matrix polynomial of the form:

$$\mathbf{M}(s) = \mathbf{M}_n s^n + \mathbf{M}_{n-1} s^{n-1}(1-s) + \mathbf{M}_{n-2} s^{n-2}(1-s)^2 + \ldots + \mathbf{M}_0 (1-s)^n,$$

where $\mathbf{M}_i$ is an $m \times m$ matrix with numeric entries. m and n are function of the degree of the curves and surfaces. Dividing the matrix polynomial by $(1-s)^n$ and substituting $u = \frac{s}{1-s}$ (substitution to change from Bernstein basis to power basis - see section 2.2.2) yields a matrix polynomial

$$\mathbf{L}(u) = \mathbf{M}_n u^n + \mathbf{M}_{n-1} u^{n-1} + \ldots + \mathbf{M}_0. \tag{3.4}$$

The intersection algorithm computes the roots of $Determinant(\mathbf{L}(u)) = 0$ by solving an eigenvalue problem in the following manner [Man92]:

Theorem 4 *Given the matrix polynomial, $\mathbf{L}(u)$ the roots of the polynomial corresponding to its determinant are the eigenvalues of the generalized system $\mathbf{C}_1 u + \mathbf{C}_2$, where*

$$\mathbf{C}_1 = \begin{bmatrix} \mathbf{I}_m & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_m & \mathbf{0} & \ldots & \mathbf{0} \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \ldots & \mathbf{I}_m & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{M}_n \end{bmatrix} \quad \mathbf{C}_2 = \begin{bmatrix} \mathbf{0} & -\mathbf{I}_m & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & -\mathbf{I}_m & \ldots & \mathbf{0} \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & -\mathbf{I}_m \\ \mathbf{M}_0 & \mathbf{M}_1 & \mathbf{M}_2 & \ldots & \mathbf{M}_{n-1} \end{bmatrix}, \tag{3.5}$$

where $\mathbf{0}$ and $\mathbf{I}_m$ are $m \times m$ null and identity matrices, respectively. If $\mathbf{M}_n$ is well-conditioned, the matrix equation can be reduced to the eigenvalues of the following associated companion matrix.

$$C = \begin{bmatrix} 0 & I_m & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & I_m \\ -\overline{\mathbf{M}}_0 & -\overline{\mathbf{M}}_1 & -\overline{\mathbf{M}}_2 & \ldots & -\overline{\mathbf{M}}_{n-1} \end{bmatrix} \tag{3.6}$$

where $\overline{\mathbf{M}}_i = \mathbf{M}_n^{-1} \mathbf{M}_i$.

Based on the problem formulation and properties of resultants, it follows that the eigenvalues of $C_1u + C_2$ correspond to one of the unknowns in (3.1), (3.2) or (3.3). The other variables can be recovered from the corresponding eigenvectors. For most intersection applications, we are interested in computing the eigenvalues in a finite interval of the real domain. For example, for Bézier curves and surfaces, the domain is $s \in [0, 1]$. However, the variable u in $\mathbf{L}(u)$ takes values in the interval $[0, \infty]$. To avoid this problem, we back-substitute $u = \frac{s}{1-s}$ and transform the matrix pencil $\mathbf{C}_1u + \mathbf{C}_2$ to $(\mathbf{C}_1 - \mathbf{C}_2)s + \mathbf{C}_2$. For the rest of the chapter, we assume that this transformation has been performed and shall concentrate in computing all the eigenvalues of a matrix pencil in a finite interval.

3.2 Algebraic Pruning

The intersection problem is now reduced to finding eigenvalues of the matrix pencil, $\mathbf{C}_1s+\mathbf{C}_2$. The QR algorithm for the standard eigenvalue problem and QZ algorithm for the generalized eigenvalue problem [GL89] compute all eigenvalues and it is difficult to restrict them to eigenvalues in the given domain. But in our applications we are only interested in finding intersections that lie inside the given domain. In this section, we describe a new algorithm called algebraic pruning to compute intersections restricted to a domain.

Initially, we use linear programming [Sei90b] to check if the control polytopes of the pairs of curves (or a curve and a surface) have a separating line (or plane) between them. If they do, then the given pair does not intersect. Otherwise, there is probably an eigenvalue of the pencil $\mathbf{C}_1s + \mathbf{C}_2$ close to the domain $[0, 1]$. This also includes complex eigenvalues.

The main idea behind our algorithm is to use inverse iteration to find some eigenvalues in the domain, and at the same time prune out portions of the domain not containing any solution. In particular, we start with a guess $s' \approx 0.5$, which is the midpoint of the domain. Using inverse iteration, we find an eigenvalue closest to s'. Let that eigenvalue be t. If t is a complex number we compute a complex conjugate pair of eigenvalues. Assuming that we chose random start vectors, $\mathbf{q}_0$ and $\mathbf{u}_0$ (as used in section on power iterations in section 2.5.1), t is an eigenvalue of $\mathbf{C}_1s + \mathbf{C}_2$ which is closest to s'. As a result, there are

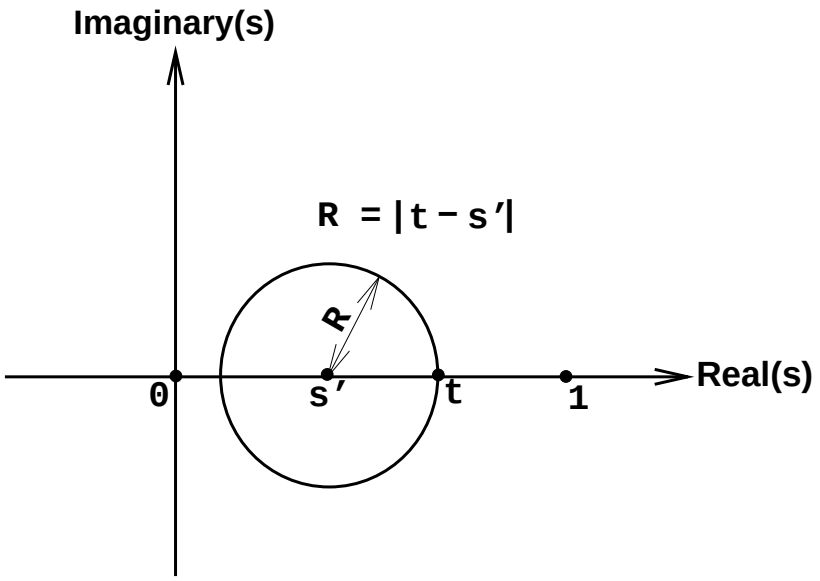

Figure 3.3: Domain Pruning based on Inverse Iteration

no other eigenvalues of the pencil in the circle centered at $s = s^{'}$ with radius $R = |t - s^{'}|$ [Wil65] (as shown in Figure 3.3).

We draw the following conclusions:

- If $t \in [0, 1]$, t corresponds to an intersection point. The rest of the unknowns are computed from the corresponding eigenvector.
- There are no other intersections in the real domain, $(s^{'} - R, s^{'} + R)$.
- The technique is recursively applied to find all the intersections in the following domains:
  - $[0, s^{'} - R]$, if $(s^{'} - R) \geq 0$.
  - $[s^{'} + R, 1]$, if $(s^{'} + R) \leq 1$.

Therefore, we are able to compute an intersection and prune the domain with inverse iteration. The algorithm is applied recursively to each domain after pruning. Our test examples contain only a few intersections in the domain. In those cases, the algorithm converges to the intersection fast and we need to apply this technique only a few times.

3.2.1 Computation of Multiple Solutions

In the previous section, we described the technique of algebraic pruning based on inverse iteration. Often the inverse iteration does not converge to a real solution t or the convergence can be slow because the closest eigenvalue corresponds to a pair of complex conjugate eigenvalues. The latter is due to the fact that there are two or more real eigenvalues which have roughly the same distance from $s^{'}$. In this section, we modify the inverse iteration to compute more than one intersection point at the same time. This includes complex conjugate pairs as well. We describe the technique to compute two solutions at the same time and it can be easily extended to find more than two solutions.

Given the approximation, $s = s^{'}$, let the two closest eigenvalues of the matrix $(\mathbf{C_1}s + \mathbf{C_2})$ be t_1 and t_2. Let the corresponding eigenvectors be $\mathbf{x}_1$ and $\mathbf{x}_2$ and $\mathbf{A} = (\mathbf{C}_1 s^{'} + \mathbf{C}_2)^{-1}$. Then $\frac{1}{(s^{'}-t_1)}$ and $\frac{1}{(s^{'}-t_2)}$ are the largest eigenvalues of $\mathbf{A}$ in magnitude. If we start with a random unit vector $\mathbf{u}_0$ and solve for p and q such that

$$\lim_{i\to\infty} (\mathbf{A}^{i+2} - p\mathbf{A}^{i+1} - q\mathbf{A}^{i})\mathbf{u}_0 = 0,$$

then $\frac{1}{t_1}$ and $\frac{1}{t_2}$ are the solutions to the equation $x^2 - px - q = 0$. Whether the solutions are real or complex depends on the sign of $(p^2 + 4q)$. We compute these multiple eigenvalues in the following manner. Let $\mathbf{u}_0$ be a random start vector.

for $k = 1, 2, \ldots$

Solve $(\mathbf{C}_1 s^{'} + \mathbf{C}_2)\mathbf{v}_k = \mathbf{C}_1\mathbf{u}_{k-1}$

$s_k = \| \mathbf{v}_k \|_\infty$

$\mathbf{u}_k = \mathbf{v}_k / s_k$

Solve for p_k *and* q_k *from*

$$s_k s_{k-1} \begin{pmatrix} \mathbf{u}_k^T \mathbf{u}_{k-1} \\ \mathbf{u}_{k-2}^T \mathbf{u}_k \end{pmatrix} = \begin{pmatrix} \mathbf{u}_{k-1}^T \mathbf{u}_{k-1} & \mathbf{u}_{k-1}^T \mathbf{u}_{k-2} \\ \mathbf{u}_{k-2}^T \mathbf{u}_{k-1} & \mathbf{u}_{k-2}^T \mathbf{u}_{k-2} \end{pmatrix} \begin{pmatrix} p_k s_{k-1} \\ q_k \end{pmatrix}$$

if $\| p_k - p_{k-1} \|_2 < \epsilon$ *and* $\| q_k - q_{k-1} \|_2 < \epsilon$, **quit**

end

After p_k and q_k converge, we compute the closest eigenvalues in the following

manner. Let $D = p^2 + 4q$. If $D > 0.0$ the two closest eigenvalues are

$$t_1 = s^{'} - 2.0/(p + \sqrt{D}),$$

$$t_2 = s^{'} - 2.0/(p - \sqrt{D}).$$

Furthermore, the radius R for pruning corresponds to the minimum of $|t_1 - s^{'}|$ and $|t_2 - s^{'}|$.

In case the closest pair of eigenvalues is a complex conjugate pair, it is computed as:

$$Real(t) = s^{'} - \frac{2p}{p^2 - D},$$

$$Imag(t) = \frac{2\sqrt{-D}}{p^2 - D}.$$

and the radius R for pruning is

$$R = \sqrt{(Real(t) - s^{'})^2 + Imag(t)^2)}.$$

3.2.2 Use of Matrix Structure

In inverse iteration, the two main operations are the LU decomposition of the matrix $\mathbf{C}_1 s^{'} + \mathbf{C}_2$ and solution of the resulting upper triangular systems. The matrix pencil defined in Theorem 4 has order $N = m * n$. In particular, it has a block companion structure being linearized from a $m \times m$ matrix polynomial of degree n. The LU decomposition is computed using Gaussian elimination [GL89] which takes about $\frac{1}{3}N^3$ operations (without pivoting). Solving each triangular system costs about $\frac{1}{2}N^2$ operations. As a result, inverse iteration requires $\frac{1}{3}N^3 + kN^2$ operations, where k is the number of iterations.

The structure of the matrices can be used to reduce the number of operations for LU decomposition as well as solve the triangular systems. Given $s^{'}$, let $\mathbf{A} = \mathbf{C}_1 s^{'} + \mathbf{C}_2$. Using the structure of $\mathbf{C}_1$ and $\mathbf{C}_2$, it can be shown that $\mathbf{A}$ is a matrix of the form:

$$\mathbf{A} = \begin{pmatrix} \alpha_1 \mathbf{I}_m & \alpha_2 \mathbf{I}_m & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \alpha_1 \mathbf{I}_m & \alpha_2 \mathbf{I}_m & \dots & \mathbf{0} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \alpha_1 \mathbf{I}_m & \alpha_2 \mathbf{I}_m \\ \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 & \dots & \mathbf{P}_n \end{pmatrix}$$

where α_1 and α_2 are functions of $s^{'}$. $\mathbf{P}_i$'s are $m \times m$ matrices, which are functions of $\mathbf{M}_i$'s and $s^{'}$. A LU decomposition of $\mathbf{A}$ can be written as

$$\mathbf{A} = \begin{pmatrix} \alpha_1\mathbf{I}_m & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \alpha_1\mathbf{I}_m & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \alpha_1\mathbf{I}_m & \mathbf{0} \\ \mathbf{R}_1 & \mathbf{R}_2 & \dots & \mathbf{R}_{n-1} & \mathbf{L}_n \end{pmatrix} \begin{pmatrix} \mathbf{I}_m & \frac{\alpha_2}{\alpha_1}\mathbf{I}_m & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_m & \frac{\alpha_2}{\alpha_1}\mathbf{I}_m & \dots & \mathbf{0} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{I}_m & \frac{\alpha_2}{\alpha_1}\mathbf{I}_m \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{U}_n \end{pmatrix},$$

where

$$\begin{aligned} \mathbf{R}_1 &= \mathbf{P}_1 \\ \mathbf{R}_2 &= \mathbf{P}_2 - \tfrac{\alpha_2}{\alpha_1}\mathbf{R}_1 \\ \mathbf{R}_3 &= \mathbf{P}_3 - \tfrac{\alpha_2}{\alpha_1}\mathbf{R}_2 \\ &\vdots \\ \mathbf{R}_{n-1} &= \mathbf{P}_{n-1} - \tfrac{\alpha_2}{\alpha_1}\mathbf{R}_{n-2} \\ \mathbf{R}_n &= \mathbf{P}_n - \tfrac{\alpha_2}{\alpha_1}\mathbf{R}_{n-1} \end{aligned}$$

Moreover, $\mathbf{L}_n$ and $\mathbf{U}_n$ correspond to the LU decomposition of $\mathbf{R_n}$.

This formulation is constructive and based on it, LU decomposition of $\mathbf{A}$ requires $\frac{1}{3}m^3 + (n-1)m^2$ operations. Furthermore, given the LU decomposition, solving the lower triangular system takes $(n-\frac{1}{2})m^2$ operations and solving the upper triangular system takes $\frac{1}{2}m^2 + (n-1)m$ operations. As a result, the total number of operations for solving the linear system is $nm^2 + (n-1)m$.

Sometimes numerical difficulties can arise with the LU decomposition of $\mathbf{A}$ when it is ill-conditioned. Even though *pivoting* can be used to improve numerical reliability, it destroys the structure of the matrix. In such cases, we use LQ factorization (similar to QR decomposition in eigenvalue computation), where L is a lower triangular matrix and Q is an orthogonal matrix. This factorization can be performed either using Householder matrices or Given's rotation. We have used Householder transformations to carry out the factorization.

3.2.3 Algorithm for Intersection

The algorithm for intersection computation combines algebraic pruning with properties of curves and surfaces. We assume that each curve and surface is associated with a corresponding control polytope. For Bézier curves and surfaces such control polytopes are formed by their control points. Similar geometric representations are known for algebraic curves and surfaces as well [Sed89]. The simplest algorithm for intersection is based on the version of algebraic pruning from Section 3.2. We start with the middle point of the domain, find a closest eigenvalue using inverse iteration and prune the domain. The algorithm can be recursively applied to each domain obtained after pruning.

Although this algorithm works well, its performance can be improved tremendously using properties of curves and surfaces and behavior of inverse iteration. The convergence of inverse iteration is a function of the distance of eigenvalues from the guess $s^{'}$. If there are a number of intersections in the domain, the ratio $|s^{'} - \lambda_1|/|s^{'} - \lambda_2|$ may not be small and the overall convergence may therefore be slow. However, the convergence is faster if there are very few intersections in the domain. In the applications involving curve and surface intersections, we make use of the geometric properties of the control polytopes in the following manner:

- Compute the number of intersections between the control polytopes of the curves or curve-surface pair. This number is used as a good guess to the actual number of intersections.

- Subdivide the curves and surfaces such that each pair obtained after subdivision consists of at most one or two intersections between the control polytopes.

In case the actual number of intersections between the original pair of control polytopes is high, we use the QR algorithm [MD94]. The technique of algebraic pruning works best when there are relatively few intersections. Let k be the number of intersections between the control polytopes. We use the algorithm based on algebraic pruning if $k < 2\sqrt{N}$, where N refers to the order of matrix pencil in Theorem 4. This is a heuristic derived based on our experience. Eventually, we apply algebraic pruning to find eigenvalues in each

domain, such that the associated control polytopes have only one or two intersections. Although we subdivide the curve and obtain new control points, the implicit representation and the matrix pencil $\mathbf{C}_1 s + \mathbf{C}_2$ remain the same and we associate each subdivided curve with a different domain. It is still possible that after subdivision the algorithm does not converge to a solution fast enough. In that case, it could be because multiple eigenvalues are close to the chosen approximation $(s^{'})$.

Inverse iteration terminates once successive values of s_i differ by less than a tolerance, TOL. In applications involving computation of multiple solutions, we terminate the iteration once p_k and q_k differ between successive iterations by at most TOL. The accuracy and performance of the overall method is a function of TOL. Depending on the number of iterations it takes to converge, we use the following values of TOL and use a one-pass or two-pass approach. Let $[a, b]$ be the interval in which all eigenvalues need to be computed. $\mathbf{u}_k$ refers to the eigenvector computed at each iteration. The eigenvector is used to extract the other variable in the system of equations.

- Use inverse iteration to compute a closest eigenvalue of $\mathbf{A} = \mathbf{C}_1 s^{'} + \mathbf{C}_2$. Initially use $TOL = 0.01/(b-a)$. After three iterations if

$$|s_3 - s_2| < TOL, \qquad |\mathbf{u}_3[1]/\mathbf{u}_3[0] - \mathbf{u}_2[1]/\mathbf{u}_2[0]| < TOL$$

  we continue using inverse iteration and modify $TOL = (1.0e^{-6})/(b-a)$.

- – In case inverse iteration does not converge to two digits of accuracy in the first three iterations, TOL remains the same and we compute p_k and q_k using the algorithm for multiple solutions. Let the inverse iteration converge to a real solution s_k or a pair of real solutions, t_{1k} and t_{2k}. Each of them is accurate up to two digits.

  – For each eigenvalue t computed with two digits of accuracy, we set $s^{'} = t$ and perform some inverse iteration on $\mathbf{A} = \mathbf{C}_1 t + \mathbf{C}_2$ using $TOL = (1.0e^{-6})/(b-a)$.

The tolerances are user driven and the number of iterations used to switch the tolerance can be varied by the user as well. The entries $\mathbf{u}_k[1]$ and $\mathbf{u}_k[0]$ of the eigenvector are used to compute the other variable in the curve-curve intersection problem.

3.2.4 Illustration

In this section, we demonstrate our algorithm on two examples:

- intersection of two planar Bézier curves each of degree four, and
- intersection of a cubic Bézier curve with a bicubic tensor-product Bézier patch.

Intersection of two planar curves

The control points of the curves in homogeneous coordinates are defined as:

$$\mathbf{P}(s) = ((-2.5, 0.3, 1.0), (4.0, 6.1, 1.0), (-1.0, -3.3, 1.0), (5.0, 1.4, 1.0), (7.2, 3.3, 1.0))$$

and

$$\mathbf{Q}(t) = ((-2.0, 2.3, 1.0), (3.0, 2.1, 1.0), (-4.0, -2.3, 1.0), (2.0, 1.3, 1.0), (3.0, 3.3, 1.0)).$$

The curves are shown in Figure 3.4. We implicitize the first curve and obtain its implicit representation as a 4×4 matrix $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} 23.2x - 26.0y + 65.8w & -21.6x - 9.0y - 51.3w & 4.4x - 30.0y + 20.0w & 3.0x - 9.7y + 10.41w \\ -21.6x - 9.0y - 51.3w & -221.2x + 90.0y + 190.4z & -72.2x - 25.7y + 408.81z & -11.2x - 12.8y + 122.88w \\ 4.4x - 30.0y + 20.0w & -72.2x - 25.7y + 408.81w & 101.6x - 156.8y - 239.52w & 39.6x - 49.2y - 122.76w \\ 3.0x - 9.7y + 10.41w & -11.2x - 12.8y + 122.88w & 39.6x - 49.2y - 122.76w & 7.6x - 8.8y - 25.68w \end{pmatrix}.$$

We substitute the second parameterization and obtain a matrix polynomial. Using Theorem 2.1 it can be reduced to finding eigenvalues of the matrix pencil $\mathbf{C}_1 t + \mathbf{C}_2$, where $\mathbf{C}_1$ and $\mathbf{C}_2$ are listed in Appendix A.

We apply the pruning algorithm on this example. The performance of the algorithm is shown in Table 3.1. In the table, t refers to the eigenvalue computed by inverse iteration and $s = \mathbf{u}[1]/\mathbf{u}[0]$ is the corresponding point on the other curve, $\mathbf{P}(s)$, based on the eigenvector. At each instance, we first compare the control polytopes of the curves for intersection. The simplest algorithm involves use of bounding box tests followed by testing the convex hulls of their control polytopes for overlap. The convex hull test is reduced to a linear programming problem, whose complexity is linear in the number of constraints. In this case, each control points contributes one constraint. Good randomized algorithms for linear programming are described in [Sei90b] and they work very well in practice. We use an implementation of Seidel's algorithm [Sei90b] given to us by Mike Hohmeyer [Hoh91].

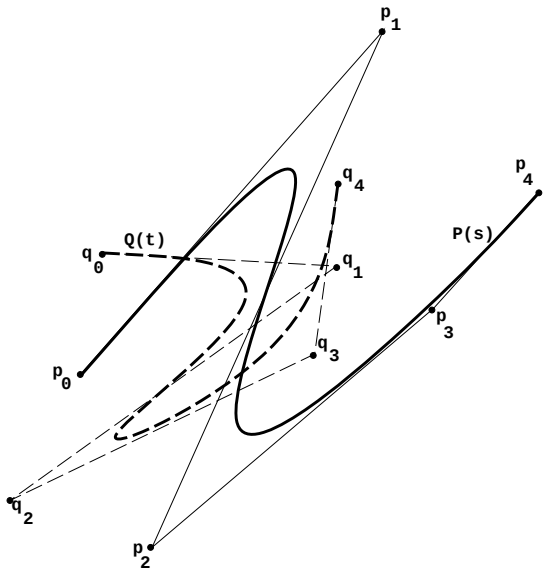

Figure 3.4: Intersection of Fourth Order Bézier Curves

Curve-Surface Intersection

The control points of the curve in homogeneous coordinates are given by

$$\mathbf{P}(s) = ((85.0, 0.0, -150.0, 1.6), (104.0, -50.0, -50.0, 1.6), (90.0, 50.0, 50.0, 1.4), (120.0, 0.0, 150.0, 1.3)).$$

The control points of the surface are given by the matrix:

$$\mathbf{Q}(u,v) = \begin{pmatrix} (160.0,-10.0,-140.0,1.8) & (140.0,-45.0,-70.0,1.4) & (135.0,64.0,40.0,1.7) & (148.0,10.0,120.0,1.2) \\ (104.0,20.0,-120.0,1.3) & (95.0,55.0,-40.0,1.2) & (109.0,-54.0,80.0,1.1) & (110.0,6.0,180.0,1.9) \\ (42.0,-159.0,-120.0,1.0) & (65.0,45.0,-60.0,1.8) & (50.0,-28.0,70.0,1.7) & (55.0,160.0,130.0,1.5) \\ (-3.0,154.0,-110.0,1.3) & (10.0,-35.0,-50.0,1.4) & (-9.0,40.0,55.0,1.6) & (9.0,-139.0,170.0,1.6) \end{pmatrix}$$

The implicit representation of the surface $\mathbf{Q}(u,v)$ is an 18×18 matrix and is denoted by $\mathbf{M}(x,y,z)$. $\mathbf{M}(x,y,z)$ is shown in Appendix B. After substituting the parameterization of $\mathbf{P}(s)$ (a cubic curve), we obtain a matrix pencil of order 54. The results of the pruning method are shown in Table. 3.2. s refers to the converged eigenvalue and u and v are obtained from the corresponding eigenvector. $\mathbf{Q}(u,v)$ thus obtained is the intersection point on the patch.

Interval	$t^{'}$	t	$s = \mathbf{u}[1]/\mathbf{u}[0]$	No. of Iterations
[0.0,1.0]	0.4	0.1297	0.1070	7
[0.0,0.5]	0.2	0.1298	0.1074	4
[.27,0.5]	0.362	0.1298	0.1066	5
[0.0,0.1298]	0.0528	0.1298	0.1073	3
[0.5,1.0]	0.7	0.7954	0.354	3
[0.7954,1.0]	0.8784	0.7951	0.351	3
[0.5,0.604]	0.541	0.7951	0.353	3

Table 3.1: Algebraic pruning on curves shown in Figure 3.4

Interval	$s^{'}$	s	$u = \mathbf{u}[1]/\mathbf{u}[0]$	$v = \mathbf{u}[3]/\mathbf{u}[0]$	Iterations
[0.0,1.0]	0.5	0.5976	0.3367	0.5738	9
[0.0,0.3923]	0.1961	0.2021	-0.7150	0.4619	5
[0.0,0.1802]	0.0901	0.1210	2.3133	0.4864	7
[0.1310,0.1802]	0.1556	0.1663	0.4871	0.1582	3
[0.2121,0.3923]	0.3022	0.2900	0.1880	-1.0303	4
[0.2121,0.2800]	0.2461	0.2394	-0.5077	-0.3593	3
[0.6076,1.0]	0.8038	0.7973	0.2332	0.8176	4
[0.8204,1.0]	0.9102	$0.9177 \pm i0.0317$	–	–	5

Table 3.2: Algebraic pruning on the curve and surface shown in Figure 3.5

3.3 Performance and Comparison

We have implemented the algorithm using LAPACK routines [ABB+92]. The overall algorithm has been applied to many cases of parametric curve intersection, algebraic curve intersection, intersection of a ray with a parametric surface and curve-surface intersections. In each case, the problem is reduced to an eigenvalue problem and we compute the eigenvalues in a domain. We have performed comparisons with the QR algorithm in [MD94] and an implementation of implicitization based algorithm described in [SP86] and Bézier clipping described in [SN90].

Bézier Clipping: Bézier Clipping is an iterative method which takes advantage of the convex hull property of Bézier curves, and iteratively clips away regions of the curve that does not intersect with the surface. Bézier clipping converges more robustly with the

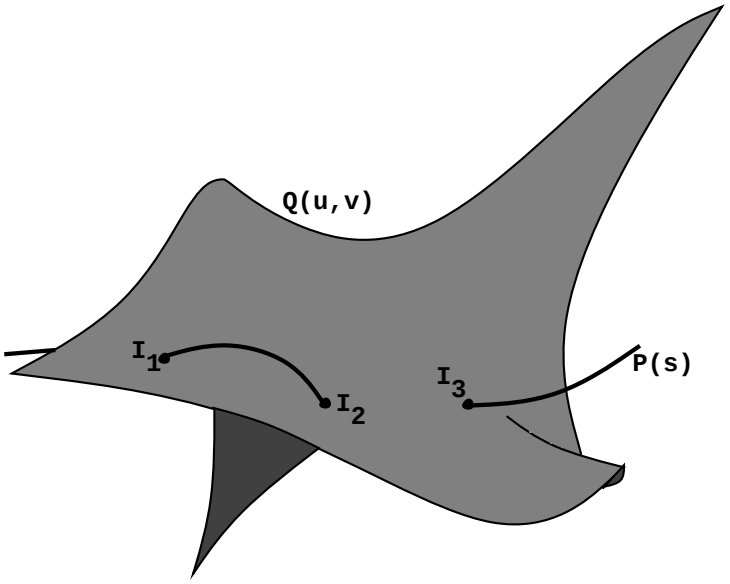

Figure 3.5: Intersection of a cubic Bézier Curve and a bicubic patch

polynomial's solution than does Newton's method. This method was first developed for ray-tracing Bézier patches [NSK90]. The main advantages of this method include applicability to high-degree polynomials, robustness and faster convergence. It does not require initial guesses unlike Newton's method and can provide all solutions within a specified range.

Implicitization algorithm: It uses the fact that any rational curve can be expressed as an implicit equation, $f(x, y, w) = 0$ [Sed83]. The problem of computing the intersection of two parametric curves is solved by implicitizing one of the curves. The parameterization of the second rational curve is substituted into the implicit form to obtain an equation of the form $f(x(s), y(s), w(s)) = 0$. The roots of this equation in the range $s \in [0, 1]$ is solved using standard numerical polynomial root finding methods like the Jenkins-Traub method [SB93].

The actual timings obtained from our implementation can be greatly improved by careful programming and using the structure of the problem. For example, the technique applied to intersection of parametric curves results in a matrix polynomial with symmetric matrices. This structure can be exploited in solving linear equations arising in inverse iteration and giving us almost a speed up of two over the implementation not making

use of symmetric structure of the matrices. As a result, if we implement algorithms as part of a generic package or specialize to particular cases (like intersection of cubic Bézier curves), we can see a considerable difference in their running time. A similar analysis holds for the implementation of implicitization based algorithm described in [SP86]. Thus, it is rather difficult to perform an exact comparison between two algorithms and in this section we analyze them in terms of convergence per iteration and the total number of iterations required on various examples.

One of the main advantages of using inverse iteration is that whenever it converges the closest eigenvalue is obtained. This is the basis of algebraic pruning. However, it is possible that inverse iteration may stagnate at a spurious eigenvalue. But these can be detected by resubstitution into the original curve and surface equations and retried with a different starting vector. It is also possible that inverse iteration does not converge for particular choices of initial eigenvalue and eigenvector. As discussed earlier, this can occur when the initial guess is close to multiple eigenvalues. Another possibility for non-convergence is the initial choice of the eigenvector u_0. Let λ_i be a closest eigenvalue to s' with corresponding eigenvector x_i. If u_0 has a very small component of x_i then rounding errors may prevent these components from being enriched and the algorithm may not converge. However, according to [Wil65], this possibility is extremely rare if u_0 is chosen at random.

If a given pair of curves or a curve and a surface have very few intersections, the technique based on algebraic pruning gives almost an order of magnitude improvement over the algorithm presented in [MD94]. This is mainly due to the fact that we are only computing the relevant solutions in the domain of interest as opposed to computing all the solutions. For example, on a DEC 5000/25, it takes about 8.5 milliseconds to compute all the intersections of the example in Section 3.2.4 using algebraic pruning. On the other hand, application of QR algorithm takes about 78.2 milliseconds on the same matrix to compute all the eigenvalues and the eigenvectors corresponding to eigenvalues in the domain. In the case of a cubic curve and bicubic surface intersections, we obtain a 54×54 matrix. For cases, involving two or three intersections the algebraic pruning performed better by more than an order of magnitude (almost 20-fold speedup) as compared to the QR algorithm. The QR algorithm has a comparable performance if the number of intersections in the domain

is at least equal to $2\sqrt{N}$.

Compared to the implicitization based approach in [SP86], algebraic pruning almost shows the same kind of performance (on low degree curves of degree three or four). The implicitization based approach involves expansion of the symbolic determinant and computing all the roots in the domain of interest of the resulting polynomial. The latter can be computed efficiently using the Bernstein representation of the resulting polynomial. On the other hand algebraic pruning reduces it to an eigenvalue problem and does not involve symbolic expansion. For most applications on degree three and degree four curves, both algorithms take about 5 to 7 milliseconds on the DEC 5000/25. However, on degree five curves consisting of at most two or three intersections, algebraic pruning performed better by a couple of milliseconds. We would again like to emphasize the fact that these are the performance figures corresponding to our implementation and an earlier implementation of implicitization based intersection algorithm [SP86]. Other implementations may result in a different set of timings.

Finally, we compared our algorithm to Bézier clipping [SN90]. This comparison has been performed only for curve intersections. The actual performance of the algorithm is actually a function of the geometry of the curves and the number of intersections in the given domain. On low degree curves of degree less than five or six, both algorithms take about 8 to 16 milliseconds on the DEC 5000/25. The algebraic pruning is typically faster in cases consisting of one or two intersections, whereas Bézier clipping is faster by a few milliseconds in the other cases. This is consistent with the fact that the convergence of algebraic pruning is a function of the proximity to other intersections. At a conceptual level, it appears that the convergence of algebraic pruning is slightly better than that of Bézier clipping for cases consisting of a few intersections. On the other hand, the number of operations at each iteration of Bézier clipping is less than that of algebraic pruning. Subdividing a Bézier curve takes $O(n^2)$ operations whereas each each iteration of algebraic pruning takes $O(nm^2)$ operations. However, in our examples we are dealing with low values of m and n and one needs to take care of the constants in front of these asymptotic bounds.

We used an implementation of Bézier clipping by John Keyser at UNC-Chapel Hill for performance comparisons. In this implementation, we found that when the toler-

Curve Degree	Patch Degree	No. of intersections	Timing (in milli sec.)		
			Bezier Clipping	QR method	Algebraic Pruning
1	1 X 1	0	3.133	0.998	0.355
1	1 X 1	1	5.140	1.102	0.561
2	1 X 1	1	4.556	1.254	0.779
1	2 X 2	1	13.67	10.50	3.91
1	2 X 2	2	34.59	10.09	6.11
2	2 X 2	1	10.34	28.80	11.65
2	2 X 2	2	18.83	26.98	17.22
3	2 X 2	1	15.53	70.73	13.28
1	3 X 3	1	9.92	65.38	11.23
2	3 X 3	1	10.40	67.96	9.21
2	3 X 3	2	18.64	91.80	22.17
3	3 X 3	2	25.01	99.55	31.67
3	3 X 3	3	52.94	650.13	89.12

Table 3.3: Comparison between three curve-surface intersection algorithms

ances were very small, the Bézier clipping algorithm may suffer from robustness problems due to sign evaluation. We performed comparisons between the QR algorithm, algebraic pruning and Bézier clipping. Table 3.3 shows the comparison among the various methods on randomly generated curves and surfaces of varying degrees. When the number of intersections is small, algebraic pruning performs best of all. But as the degrees and number of intersections rise, Bézier clipping performs best.

In applications like ray-tracing, algebraic pruning can be very well combined with ray to ray coherence. For example, the intersection of the previous ray with the surface can be used as a starting guess for the next ray-surface intersection. Given a good starting guess, inverse iteration converges in a very few iterations. As a result, inverse iteration combines very well with spatial and temporal coherence.

3.4 Robustness of Algebraic Pruning

Our method of algebraic pruning relies heavily on the performance and robustness of inverse power iterations. In this section, we shall discuss some of the issues concerning

inverse iterations.

The i^{th} step of inverse iterations of a matrix $\mathbf{A}$ with shift p and initial vector $\mathbf{u}_0$ proceeds as follows:

$$(\mathbf{A} - p\mathbf{I})\ \mathbf{v}_{i+1} = \mathbf{u}_i, \qquad \mathbf{u}_{i+1} = \frac{\mathbf{v}_{i+1}}{\| \mathbf{v}_{i+1} \|_\infty} \tag{3.7}$$

Let the initial vector $\mathbf{u}_0$ be written as a linear combination of the eigenvectors $\mathbf{x}_j$ of the matrix $\mathbf{A}$. That is,

$$\mathbf{u}_0 = \alpha_1\ \mathbf{x}_1 + \alpha_2\ \mathbf{x}_2 + \ldots + \alpha_n\ \mathbf{x}_n$$

We make a few assumptions while making this claim. Firstly, we assume that the set of eigenvectors of $\mathbf{A}$ span the entire space. This is true only if the matrix is *diagonalizable*, *i.e.,* if the Jordan canonical form [SB93] is a diagonal matrix, or it has distinct eigenvalues. Secondly, the assumption that a random initial vector has a non-zero component along each eigenvector may not be true. However, this seems very unlikely for purely random initial choices. In our experience, if p is very close to λ_k then initial vectors which are deficient in the $\mathbf{x}_k$ component usually have a substantial component after one iteration.

Given these assumptions, choice of $\mathbf{u}_0$, and the fact that $\mathbf{A}\ \mathbf{x}_k = \lambda_k\ \mathbf{x}_k$, after i steps of the inverse iteration [JI92]

$$\mathbf{u}_i = \Sigma_{j=1}^{n} \alpha_j\ (\lambda_j - p)^{-i}\ \mathbf{x}_j$$

It is easy to see that as $i \to \infty$ the term that dominates corresponds to an eigenvalue that is closest to the point p in the complex plane (if there is such a unique eigenvalue).

As seen in equation (3.7), a linear system has to be solved at each step of the iterative algorithm. If p is very close to the eigenvalue λ_k, the matrix $(\mathbf{A} - p\ \mathbf{I})$ is close to singular. Thus the linear system may be ill-conditioned. However, we will show that inverse iteration gives good results even when p is very close to λ_k unless the corresponding eigenvector $\mathbf{x}_k$ is ill-conditioned. Since our algorithm runs in finite precision, let us assume that $\mathbf{v}_{i+1}$ is the exact solution of a perturbed matrix $(\mathbf{A} + \mathbf{\Delta})$, $\| \mathbf{\Delta} \|_\infty = \delta$. Let $p = \lambda_k + \epsilon$. Thus,

$$(\mathbf{A} - p\ \mathbf{I} + \mathbf{\Delta})\ \mathbf{v}_{i+1} = \mathbf{u}_i$$

$$
\begin{aligned}
(\mathbf{A} - \lambda_k \mathbf{I} - \epsilon \mathbf{I} + \mathbf{\Delta}) \frac{\mathbf{v}_{i+1}}{\|\mathbf{v}_{i+1}\|_\infty} &= \frac{\mathbf{u}_i}{\|\mathbf{v}_{i+1}\|_\infty} \\
(\mathbf{A} - \lambda_k \mathbf{I}) \mathbf{u}_{i+1} &= (\epsilon \mathbf{I} - \mathbf{\Delta}) \mathbf{u}_{i+1} + \frac{\mathbf{u}_i}{\|\mathbf{v}_{i+1}\|_\infty} = \rho ,
\end{aligned}
$$

where $\| \rho \|_\infty < |\epsilon| + \delta + \frac{1}{\|\mathbf{v}_{i+1}\|_\infty}$ [JI92].

This shows that if $\| \mathbf{v}_{i+1} \|_\infty$ is large and ϵ and δ are small, $\mathbf{u}_{i+1}$ gives a small residue. To derive the lower bound for $\| \mathbf{v}_{i+1} \|_\infty$, let us assume that

$$
\mathbf{u}_i = \Sigma_{j=1}^{n} \alpha_j \mathbf{x}_j, \qquad \mathbf{v}_{i+1} = \Sigma_{j=1}^{n} \beta_j \mathbf{x}_j,
$$

and let $\| \mathbf{x}_j \|_\infty = 1$. For each $\mathbf{x}_j$, let $\mathbf{y}_j$ be a unit vector such that for $l \neq j$, $\mathbf{y}_j^T \mathbf{x}_l = 0$ and $\mathbf{y}_j^T \mathbf{x}_j = \cos\theta_j$, where θ_j is the angle between $\mathbf{x}_j$ and $\mathbf{y}_j$. From this, it is obvious that $\mathbf{y}_k^T \mathbf{u}_i = \cos\theta_k \alpha_k$, $\mathbf{y}_k^T \mathbf{v}_{i+1} = \cos\theta_k \beta_k$, and $\mathbf{y}_k^T (\mathbf{A} - \lambda_k \mathbf{I}) \mathbf{u}_{i+1} = 0$. Earlier, we had

$$
(\mathbf{A} - p \mathbf{I} + \mathbf{\Delta}) \mathbf{v}_{i+1} = \mathbf{u}_i
$$

Premultiplying both sides by $\mathbf{y}_k^T$ and simplifying, we get

$$
-\epsilon \cos\theta_k \beta_k + \mathbf{y}_k^T \Delta \mathbf{v}_{i+1} = \cos\theta_k \alpha_k
$$

Hence,

$$
|\alpha_k| \leq \frac{(|\epsilon| + \delta) \|\mathbf{v}_{i+1}\|_\infty}{|\cos\theta_k|}
$$

giving

$$
\| \mathbf{v}_{i+1} \|_\infty \geq \frac{|\alpha_k \cos\theta_k|}{(|\epsilon| + \delta)}
$$

The above analysis shows that inverse iteration has small backward error. Unfortunately, however, a small backward error does not imply a small forward error [Ips97]. Their relationship also depends on the distribution of eigenvalues. In the absence of this information, we have to resort to backward analysis.

Chapter 4

Surface Intersection Algorithm

4.1 Overview

The problem of surface intersection corresponds to computing an accurate representation of the intersection curve. The difficulty of the problem lies both in the *algebraic* and the *geometric complexity* of the intersection curve. The degree of the curve resulting from the intersection of rational parametric surfaces can be very high, and computing an accurate representation is nontrivial. For example, the degree of the intersection curve of two tensor-product bicubic Bézier surfaces can be as high as 324. In terms of geometric complexities, the curve may have multiple components, small loops, singularities, and multiple branches at the singularities. Our approach is based on an accurate representation of the intersection curve. It is a well known result in algebraic geometry that the intersection space curve has a one-to-one correspondence with an algebraic plane curve after suitable linear transformations(except for a finite number of points). The plane curves with one-to-one correspondence with the intersection curve in space are shown in Figure 4.1. We represent the plane curve as an unevaluated determinant [MC91].

Given two Bézier surfaces,

$$\mathbf{F}(s,t) = (X(s,t), Y(s,t), Z(s,t), W(s,t))$$
$$\mathbf{G}(u,v) = (\overline{X}(u,v), \overline{Y}(u,v), \overline{Z}(u,v), \overline{W}(u,v))$$

in homogeneous coordinates, implicitize $\mathbf{F}(s,t)$ to the form $f(x,y,z,w) = 0$ [Sed83, Hof89]

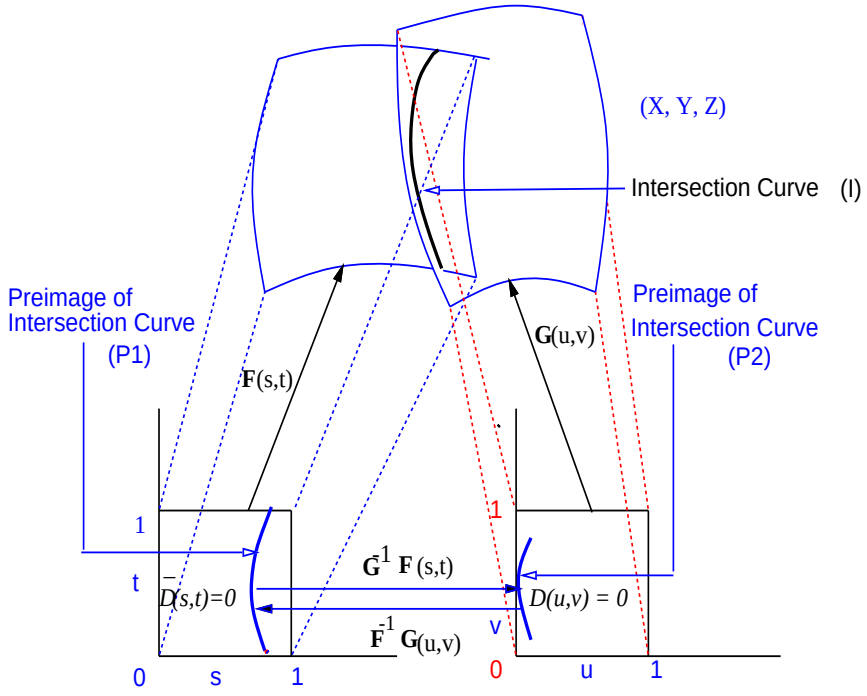

Figure 4.1: Intersection curve and its planar preimages [MC91]

and substitute the parametrization of $\mathbf{G}(u, v)$ into f to get an algebraic plane curve of the form

$$f(\overline{X}(u, v), \overline{Y}(u, v), \overline{Z}(u, v), \overline{W}(u, v)) = 0. \tag{4.1}$$

This corresponds to an algebraic plane curve birationally equivalent to the original intersection curve. However, its degree is rather high and leads to efficiency and accuracy problems [Hof89]. Instead of explicitly computing the plane curve, we use numerically stable algorithms like eigenvalues and Singular Value Decomposition (SVD) on a matrix representation.

4.1.1 Matrix formulation of intersection curve

The first step in generating the algebraic plane curve is to compute the implicit form of one of the parametric surfaces. To perform this computation, we use an algorithm by [Sed83]. Let us assume that the surface we are implicitizing is denoted by $\mathbf{F}(s, t)$ with

coordinate functions as above.

The implicit representation of $\mathbf{F}(s,t)$ is obtained by eliminating s and t from the following set of equations.

$$\begin{aligned} xW(s,t) - X(s,t) &= 0 \\ yW(s,t) - Y(s,t) &= 0 \\ zW(s,t) - Z(s,t) &= 0 \end{aligned} \tag{4.2}$$

Here x, y and z are the individual components of $\mathcal{R}^3$ and are treated as constants here. The implicit form of $\mathbf{F}(s,t)$ is basically an expression in terms of x, y, z and other numeric coefficients of the patch equations that simultaneously satisfy (4.2). Dixon's resultant [Dix08] provides an elegant way to compute this expression.

Dixon's resultant: Let us denote the three equations in (4.2) as $p_1(s,t)$, $p_2(s,t)$ and $p_3(s,t)$. If there exists an $(\tilde{s},\tilde{t})$ which simultaneously satisfies the three equations, the following determinant will vanish for that value of s and t regardless of the values of α and β.

$$Det(s,t,\alpha,\beta) = \begin{vmatrix} p_1(s,t) & p_2(s,t) & p_3(s,t) \\ p_1(\alpha,t) & p_2(\alpha,t) & p_3(\alpha,t) \\ p_1(\alpha,\beta) & p_2(\alpha,\beta) & p_3(\alpha,\beta) \end{vmatrix}$$

The determinant vanishes for any $(\tilde{s},\tilde{t})$ which satisfy $p_1 = p_2 = p_3 = 0$ because the top row vanishes. The determinant also vanishes if either $s = \alpha$ or $t = \beta$ because two rows would then be identical. Hence, $(s-\alpha)$ and $(t-\beta)$ are factors of the determinant. Define

$$\delta(s,t,\alpha,\beta) = \frac{Det(s,t,\alpha,\beta)}{(s-\alpha)(t-\beta)}$$

δ vanishes for arbitrary values of α and β if and only if $s = \tilde{s}$ and $t = \tilde{t}$. If the parametric surface is of degree $m \times n$, then the maximum degree monomial of δ is $s^{m-1}t^{2n-1}\alpha^{2m-1}\beta^{n-1}$. Considering δ as a polynomial in α and β whose coefficients are polynomials in s and t, we can write

$$\delta(s,t,\alpha,\beta) = \Sigma_{i=0}^{2m-1}\Sigma_{j=0}^{n-1} f_{ij}(s,t)\alpha^i\beta^j$$

This polynomial has $2mn$ terms. Because δ must vanish for any value of α and β if $s = \tilde{s}$ and $t = \tilde{t}$, all of $f_{ij}(\tilde{s}, \tilde{t})$ must also vanish. Therefore, $2mn$ polynomials have been generated each with $2mn$ terms in s and t. The determinant of these coefficients will serve as the resultant. Let $C_{i,j,k,l}$ denote the coefficient of $s^k t^l$ in $f_{ij}(s,t)$.

$$\begin{matrix} \alpha^0\beta^0 \\ \vdots \\ \alpha^i\beta^j \\ \vdots \\ \alpha^{2m-1}\beta^{n-1} \end{matrix} \begin{pmatrix} C_{0,0,0,0} & \cdots & C_{0,0,k,l} & \cdots & C_{0,0,m-1,2n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ C_{i,j,0,0} & \cdots & C_{i,j,k,l} & \cdots & C_{i,j,m-1,2n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ C_{2m-1,n-1,0,0} & \cdots & C_{2m-1,n-1,k,l} & \cdots & C_{2m-1,n-1,m-1,2n-1} \end{pmatrix} \begin{pmatrix} s^0t^0 \\ \vdots \\ s^k t^l \\ \vdots \\ s^{m-1}t^{2n-1} \end{pmatrix} = 0$$

It turns out that for the set of equations in (4.2) each of the $C_{i,j,k,l}$ is a linear expression of the form $a_{ij}x + b_{ij}y + c_{ij}z + d_{ij}w$. Let us denote this matrix as $\mathbf{M}(x,y,z,w)$. Further, the *kernel* of $\mathbf{M}(x,y,z,w)$, represented by the rightmost vector in the left hand side of the above expression, have entries that are simple functions of s and t. So for a given value of x, y and z that lie on the surface, the corresponding parameters s and t can be obtained from the kernel vector. A vector $\mathbf{x}$ lies in the *kernel* of a matrix $\mathbf{A}$ if $\mathbf{Ax} = \mathbf{0}$.

We substitute the parametrization of $\mathbf{G}(u,v)$ into this matrix and obtain a representation of the form $\mathbf{M}(u,v)$, where each entry is a polynomial in u and v. This substitution is very simple because every entry of the matrix is just a linear term. The degree of each polynomial in $\mathbf{M}(u,v)$ corresponds to the degree of the patch $\mathbf{G}(u,v)$.

4.1.2 Parameterizations with base points

The base points of a parameterization are the common solutions of the equations

$$\begin{aligned} X(s,t) = 0, \quad Y(s,t) = 0, \\ Z(s,t) = 0, \quad W(s,t) = 0 \end{aligned} \tag{4.3}$$

The base points also include common solutions at infinity. In general, any faithful parameterization of a rational surface whose algebraic degree is not a perfect square has base points. Consider a base point $\mathbf{p} = (s', t')$. By definition,

$$X(s',t') \;=\; Y(s',t') \;=\; Z(s',t') \;=\; W(s',t') \;=\; 0$$

It is therefore obvious that (s', t') is a non-trivial solution to (4.2). Further, this is a solution irrespective of the values of x, y or z. Therefore, the resultant of this set of equations is identically zero.

Checking for base points: The most obvious way to identify whether the given parameterization contains base points is to compute all the common solutions of (4.3). The basic idea is to find all the solutions to two of the equations (say, $X(s,t) = Y(s,t) = 0$). Each element of the solution set (assuming a finite set) is tested for satisfiability by substituting in the other two equations to recover the base points. However, this method cannot detect parameterizations which are very close to having base points (we call it the *near base point case*). The implicit form of the surface is highly error-prone if the parameterization contains *near base points*. Further, recovering the implicit form of the surface is not possible from this method.

It was mentioned earlier that in the presence of base points, the resultant of (4.2) is identically zero (independent of the values of x, y or z). Therefore, the matrix $\overline{\mathbf{M}}(x, y, z, w)$ (whose determinant gives the resultant) is always singular (rank deficient). SVD is a popular method to find the rank of a matrix. We substitute random values for x, y and z (making sure they do not lie on the original surface) in the matrix and perform SVD. If it contains zero singular values, it implies that the given parameterization contains base points. This method can also identify parameterizations with *near base points*. In such cases, some singular values of the matrix $\overline{\mathbf{M}}(x, y, z, w)$ are very close to zero. We treat near base point cases as if they contain base points (by zeroing the corresponding singular values).

Computing the implicit form: The resultant (determinant of $\overline{\mathbf{M}}(x, y, z, w)$) provides the implicit representation of the surface if its parameterization does not contain base points. However, in their presence, the resultant method will not work. It was shown in [Man92] that the maximum rank submatrix (largest non-vanishing minor) contains the implicit form as a factor in such cases. In order to obtain the implicit representation of the surface, we have to find the maximum rank submatrix. This can be achieved by performing Gaussian elimination on the original matrix. Substitution of the parameterization of $\mathbf{G}(u, v)$ into this minor gives us the planar projection of the intersection curve. It must be observed that the rank submatrix could contain extraneous factors (other than the implicit form) which must

be eliminated by testing the solutions obtained with the original set of surface equations.

4.1.3 Computing partial derivatives of intersection curve

Let us denote the determinant of the matrix $\mathbf{M}(u,v)$ as $D(u,v)$. $D^u(u,v)$ and $D^v(u,v)$ represent the first order partial derivatives with respect to u and v. To be able to trace through the intersection curve we need to evaluate $D(u_1,v_1)$, $D^u(u_1,v_1)$ and $D^v(u_1,v_1)$ (used for tangent computation - see Chapter 2) for a given point (u_1,v_1) accurately and efficiently. To compute the first and higher order partials, we use a simple variation of Gaussian elimination [MC91]. The basic idea is to compute the partial derivative of each matrix entry at the beginning of computation and update the derivative information with each step of Gaussian elimination. In this case, we modify the matrix structure such that each entry consists of a tuple $\mathbf{G}_{ij}(u_1,v_1) = (g_{ij}(u_1,v_1), g^u_{ij}(u_1,v_1), g^v_{ij}(u_1,v_1))$, where $g^u_{ij}(u_1,v_1)$ and $g^v_{ij}(u_1,v_1)$ represent the partial derivatives of $g_{ij}(u,v)$ with respect to u and v respectively at (u_1,v_1). The resulting matrix is of the form

$$M^*(u_1,v_1) = \begin{pmatrix} \mathbf{G}_{11}(u_1,v_1) & \dots & \mathbf{G}_{1n}(u_1,v_1) \\ \vdots & \dots & \vdots \\ \mathbf{G}_{n1}(u_1,v_1) & \dots & \mathbf{G}_{nn}(u_1,v_1) \end{pmatrix}$$

To compute $D(u_1,v_1), D^u(u_1,v_1)$ and $D^v(u_1,v_1)$, we perform Gaussian elimination. We consider the matrix formed by first entry of each tuple (equal to $M(u_1,v_1)$) and proceed to compute its determinant using Gaussian elimination. As a side effect we change the entry in the other tuples. Assume we are operating on the ith and kth rows of the matrix. A typical step of Gaussian elimination is of the form

$$\begin{aligned} \overline{g}_{kj} &= g_{kj} - \tfrac{g_{ki}}{g_{ii}} g_{ij} \\ g_{kj} &= \overline{g}_{kj} \end{aligned}$$

where g_{kj} represents the element in the kth row and jth column of the matrix. In the new formulation this step is replaced by

$$\begin{aligned} \overline{g}_{kj} &= g_{kj} - \tfrac{g_{ki}}{g_{ii}} g_{ij}, \\ \overline{g}^u_{kj} &= g^u_{kj} - \tfrac{(g^u_{ki} g_{ij} + g_{ki} g^u_{ij}) g_{ii} - (g_{ki} g_{ij}) g^u_{ii}}{(g_{ii})^2}, \end{aligned}$$

$$
\begin{aligned}
\overline{g}^v_{kj} &= g^v_{kj} - \tfrac{(g^v_{ki}g_{ij}+g_{ki}g^v_{ij})g_{ii}-(g_{ki}g_{ij})g^v_{ii}}{(g_{ii})^2}, \\
g_{kj} &= \overline{g}_{kj}, \\
g^u_{kj} &= \overline{g}^u_{kj}, \\
g^v_{kj} &= \overline{g}^v_{kj}
\end{aligned}
$$

We make a choice for the pivot element based on the first tuple (i.e. g_{ij} entry). After Gaussian elimination is complete, we compute $D(u_1, v_1), D^u(u_1, v_1)$ and $D^v(u_1, v_1)$ as

$$
\begin{aligned}
D(u_1, v_1) &= \textstyle\prod_{i=1}^{n} g_{ii} \\
D^u(u_1, v_1) &= D(u_1, v_1) \textstyle\sum_{i=1}^{n} \frac{g^u_{ii}}{g_{ii}} \\
D^v(u_1, v_1) &= D(u_1, v_1) \textstyle\sum_{i=1}^{n} \frac{g^v_{ii}}{g_{ii}}
\end{aligned}
$$

This procedure can be easily extended to compute the higher order partial derivatives as well. Furthermore, the analysis of Gaussian elimination may be used to analyze the numerical accuracy of partial derivatives computation.

4.2 Intersection Computation

The intersection curve in the domain of $\mathbf{G}(u, v)$ is defined as the *singular set* of the matrix polynomial $\mathbf{M}(u, v)$. In other words, it consists of all points $(u_1, v_1) \in [0, 1] \times [0, 1]$ such that $\mathbf{M}(u_1, v_1)$ is singular. Corresponding to each point (u_1, v_1) there exists a point $(s_1, t_1) \in [0, 1] \times [0, 1]$ in the domain of $\mathbf{F}(s, t)$. Given a point (u_1, v_1) in the domain of $\mathbf{G}(u, v)$, (s_1, t_1) can be computed from a vector in the *kernel* of $\mathbf{M}(u_1, v_1)$ [Dix08]. The main advantages of this matrix representation are its efficiency and accuracy. Although the singular set is defined in terms of a determinant, we use algorithms based on *eigenvalues* and *singular values* for numerical stability. Efficient and accurate algorithms for computing the eigendecomposition and SVD (**S**ingular **V**alue **D**ecomposition) are well known [GL89], and good implementations are available as part of numerical libraries like EISPACK and LAPACK.

4.2.1 Start Points

Given the matrix representation of the intersection curve, we use numerical marching methods to evaluate points on the intersection curve. Figure 1.4 shows a wireframe of a planar surface intersecting a bicubic patch. As shown in the figure, quite a few curve components are generated as a result of this intersection. The marching algorithm needs at least one start point on each such component. These components can be classified into *open* and *closed* components. Open components have an intersection with one of the boundary curves of the surface as shown in Figure 1.4. These are points on the plane curve where one of the parameter values is 0 or 1. The other components are closed loops. The start points on the open components are computed using curve-surface intersections (using algorithms from previous chapter). In particular, we substitute $u = 0, u = 1, v = 0$ and $v = 1$ into the representation of the intersection curve, $\mathbf{M}(u, v)$, and compute the intersection points.

The algorithms for computing start points on the open components of the intersection curve are based on Bézier curve-surface intersections. Our curve-surface intersection algorithm based on eigenvalue computation and inverse power iterations was described in chapter 3.

The difficulty in identifying start points on closed components lies in the fact that loops have no simple characterization such as the one for open components. However, we show that we can use a simple algebraic property which will guide us to some point on every loop. The loop detection algorithm is elaborated in chapter 5. We will now assume that we have obtained at least one starting point on every component of the intersection curve, and proceed to describe the tracing algorithm.

4.3 Tracing

Given the start points, we evaluate the curve using our tracing algorithm. There are a number of algorithms proposed for tracing [BHHL88, BK90, Che89, KPW90]. Given a point on the curve, an approximate value of the next point is obtained by taking a small step size in a direction determined by the local geometry of the curve. A single tracing step is shown in Figure 4.2. Given the approximate value, these algorithms use iterative

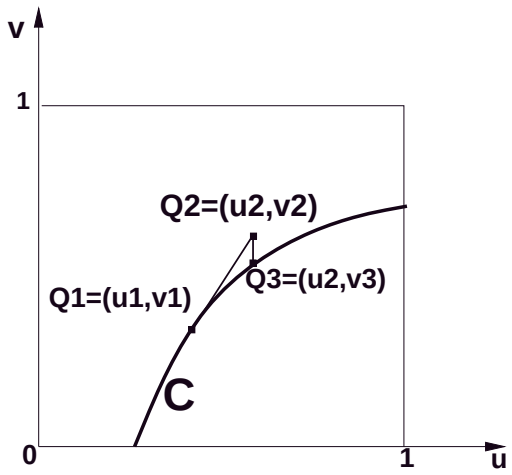

Figure 4.2: A single tracing step

methods like Newton's method to trace back on to the curve.

The three main problems with tracing algorithms are [FF92, Sny92]

1. Converging back on to the curve.

2. Component jumping.

3. Inability to handle singularities and multiple branches.

The convergence problems arising from the behavior of Newton's method are described in [FF92]. It is rather difficult to predict the convergence of Newton's method on high degree equations corresponding to the intersection (for bicubic patches). Component jumping can occur when two components of the curve are close to each other as shown in Figure 4.3. In this case, the tracing algorithm can jump from point A on component C1 to point B on component C2. Most implementations circumvent this problem by choosing *very small* and *conservative* step sizes. But this still cannot guarantee correctness and, moreover, slows down the algorithm. No efficient algorithms are known for handling singularities on the intersection curve of high degree surfaces (such as bicubic patches).

We present an efficient tracing algorithm that can resolve all these issues most of the time. In particular, we introduce a technique called *domain decomposition* and tracing

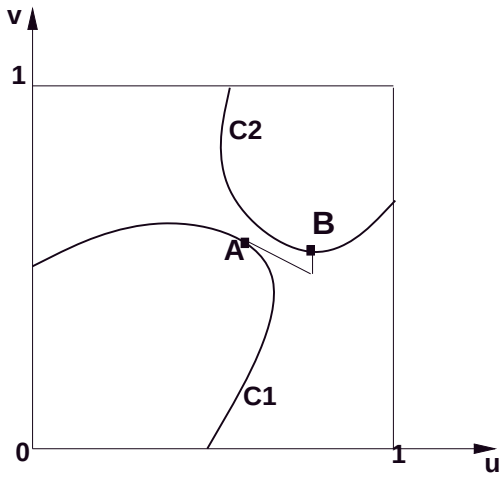

Figure 4.3: Component jumping

based on *inverse power iterations*. Singularities are also handled efficiently under well-placed assumptions.

4.3.1 Domain Decomposition

After performing curve-surface intersection and loop detection, a sequence of points is obtained on the curve ($(u, v) \in [0, 1] \times [0, 1]$) which either corresponds to starting points on open components or some points on loops. Using these points, the intersection curve is traced completely without missing any important curve features. The idea behind *domain decomposition* is that if there are only two boundary points inside a domain with no loops, these points belong to the same component of the intersection curve. Further, there exists exactly one component of the intersection curve inside this domain. We are guaranteed that each subdomain does not have any new loops because the loop detection algorithm has already been applied. Therefore, the purpose of the algorithm is to subdivide the original domain into smaller subdomains such that each subdomain contains exactly one curve component.

We now describe the working of domain decomposition. The input into the domain decomposition routine is a rectangular domain, specified as $[L_u, H_u] \times [L_v, H_v]$, and a set of points, S, on the intersection curve inside this domain. S covers all the components of the

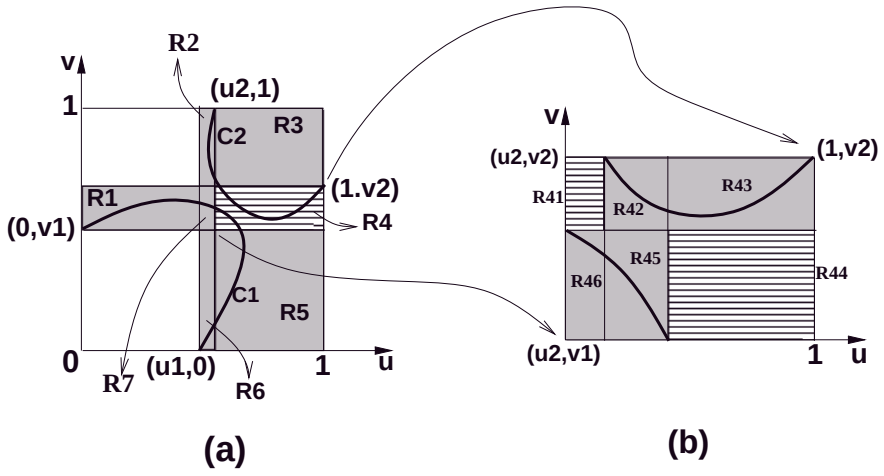

Figure 4.4: (a) First level domain decomposition (b) Second level decomposition

intersection curve inside the domain. If the cardinality of S is two, then we are assured of a single curve component inside the domain and the decomposition terminates.

If the cardinality of S is greater than two the algorithm subdivides the domain along isoparametric lines determined by the parametric values of points of S. The isoparametric lines chosen at every point (u_1, v_1) could either be a *u-isoline* $(u = u_1)$ or a *v-isoline* $(v = v_1)$. The algorithm arbitrarily chooses the v-isoline to subdivide the domain. If subdivision is not possible (all the points in S have v coordinates as L_v or H_v), then u-isoline is chosen for subdivision. In the process, new points corresponding to the intersections of the isoparametric lines with the intersection curve are generated and inserted into the appropriate subdomains. Domain decomposition is then applied recursively to each subdomain.

In most cases of surface-surface intersection, this process quickly separates out the various curve components. But in the presence of singularities, no amount of subdivision helps. In such cases, subdivision of domains is not carried out indefinitely. If the dimensions of a domain become smaller than a specified tolerance[1], the subdivision is stopped and checked for singularities. Informally, singularities are points on the intersection curve where the curve self-intersects. A more formal treatment of singularities is given in the next section. In the presence of singularities (except for cusps), no level of decomposition can

[1] we use 10^{-5} in our implementation

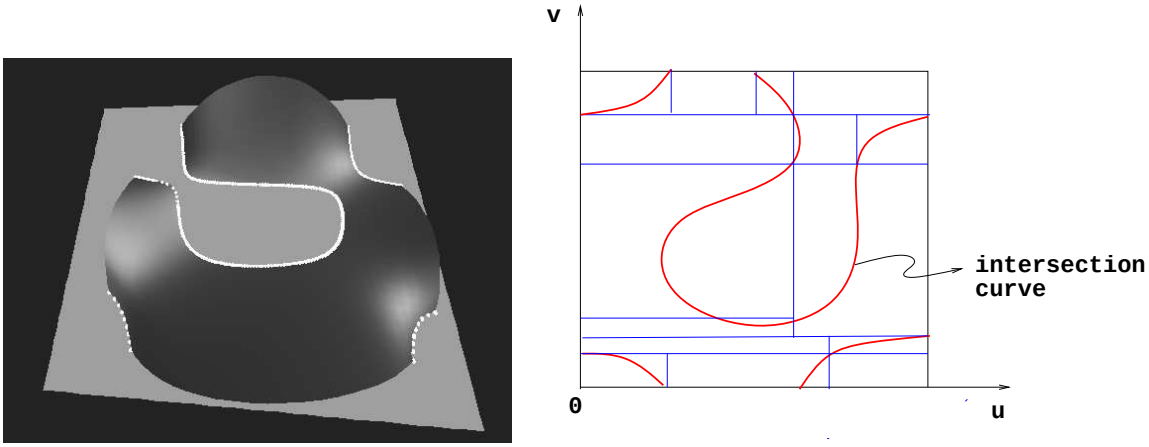

Figure 4.5: Application of domain decomposition

produce subdomains with one simple curve component unless the singular point is determined accurately. If domain decomposition is unable to isolate single curves in a domain after repeated levels of subdivision, then one of two cases can occur.

- Curve has a singularity, or
- Two components of the intersection curve are very close.

At this point, minimization of an energy function $E(u,v,s,t)$ distinguishes the two cases.

$$E(u,v,s,t) = (D(u,v,s,t)^2 + N(u,v,s,t)^2) \tag{4.4}$$

where,

$$D(u,v,s,t) = \mid \mathbf{F}(s,t) - \mathbf{G}(u,v) \mid \tag{4.5}$$

and,

$$N(u,v,s,t) = \mid (\mathbf{F_s}(s,t) \times \mathbf{F_t}(s,t)) \times (\mathbf{G_u}(u,v) \times \mathbf{G_v}(u,v)) \mid \tag{4.6}$$

$\mid . \mid$ refers to the length operator and $\times$, the cross product of two vectors. The minimization is applied with the midpoint of the region given as the initial point. A minimum value of zero corresponds to a singularity. A non-zero minimum value means that the curve has two very close components. If there is a singularity, then subdivision is done at the singular point and domain decomposition is performed in each subdomain. Singularities

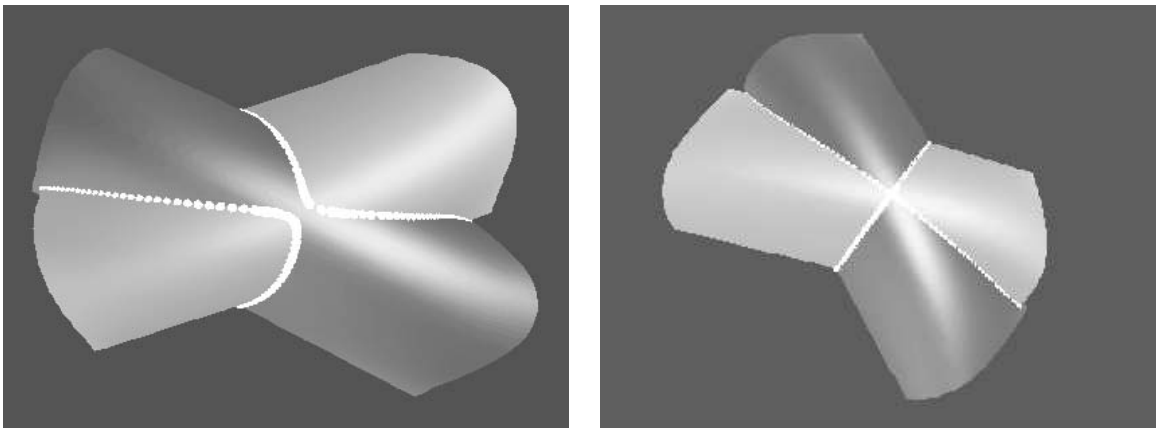

Figure 4.6: (a) Intersection curve components lying close to each other (b) Two patches intersecting in a singularity

are unstable points and are very sensitive to small input perturbations and floating point errors. Therefore, the algorithm reports a singularity if the minimum value obtained is smaller than a user-specified value[2].

The pseudocode for the domain decomposition algorithm is described below.

- **DomainDecomposition**(*domain, Xsection_points, tolerance*)

  1. If (there are only two *Xsection_points*) trace the curve inside the region and return.
  2. If (region size is smaller than *tolerance*)
     - Apply singularity criterion.
     - If there is a singularity
       * Subdivide the domain at the singular point along both axes.
       * Find all intersection points along the subdivided curves.
       * for each subregion, do DomainDecomposition(subregion, new_points, *tolerance*).
       * Return.
  3. If (domain convergence is slow)

[2] 10^{-8} in our system

- Divide the domain at midpoint of one of the parameters.
- Compute the intersections of the curve with the dividing line.
- For each of the two subregions, do DomainDecomposition(subregion, new_points, *tolerance*).
- Return.

else

- Divide the domain along isoparametric lines from every *Xsection_points*. For the point (L_u, v_1), the corresponding line is $v = v_1$.
- Compute the intersection of the curve with each such line.
- for each subregion, do DomainDecomposition(subregion, new_points, *tolerance*).
- Return.

Each time a subdivision of the domain is performed, we also maintain the connectivity of the curves between various cells. At the end of the DomainDecomposition algorithm, a set of curves traced out inside each region is obtained. Some of these are parts of the same curve component. By using the cell connectivity structure and matching their endpoints, they are connected appropriately to obtain the original intersection curve in the $[0,1] \times [0,1]$ domain. This algorithm guarantees

- No component jumping - tracing is performed only inside a region that is guaranteed to contain just one curve.
- Singularity detection - During all stages of the algorithm, singular points are always bracketed. It is possible to miss some singular points, however, if they are not well-separated.

The decomposition algorithm is illustrated on tensor product surfaces in Figures 4.4(a), 4.4(b) and 4.5. In our test examples, the algorithm uses one or two levels of decomposition. However, the number of levels may be more in the presence of singularities or close components. In that case, the geometry of the curve is not simple and increases

the complexity of any robust algorithm. The decomposition step is similar in nature to that of subdivision or interval arithmetic based algorithms. However, subdivision in our case is done only for separating the components and not for evaluating them to a certain accuracy. For almost all cases we have tested, domain decomposition performs fewer decompositions. The number of decompositions in subdivision and interval arithmetic based algorithms depends on an accuracy parameter. The algorithm has been implemented and tested on a wide variety of intersections and it is an order of magnitude faster than previously known robust algorithms (like interval arithmetic).

Figures 4.5 and 4.6 show the power of domain decomposition. It can take care of arbitrary intersections. Figure 4.6(a) shows the intersection of two patches where the intersection curves come very close to each other. Figure 4.6(b) shows the same two patches intersecting in a singularity. Domain decomposition was able to detect the presence of singularity in the second case, and traced all the branches correctly.

The algorithm given above is used to partition the domain of the curve into regions with a single curve component. Its complexity is a function of the number of components and the separation of the components into various regions. For most practical cases, there are a few well-separated components in the real domain and the algorithm performs well for such cases. In many ways the underlying philosophy is rather similar to *cylindrical algebraic decomposition* [Col75] based algorithms for partitioning the domain into regions. Our algorithm uses an efficient and accurate zero-dimensional solver (described in chapter 3) and works well using finite precision arithmetic. On the other hand, the algorithms based on algebraic decomposition [Arn83] compute all the extremal point and turning points using purely symbolic methods and exact arithmetic. Even though this method guarantees that the solution is always topologically reliable, they are impractical because of their large memory requirements and poor efficiency.

Analysis of domain decomposition

Figure 4.7 provides a comparison between ordinary bisection and domain decomposition. It can be seen that in the case depicted by the figure, our method performs much better than bisection. In fact, on an average, domain decomposition achieves the desired

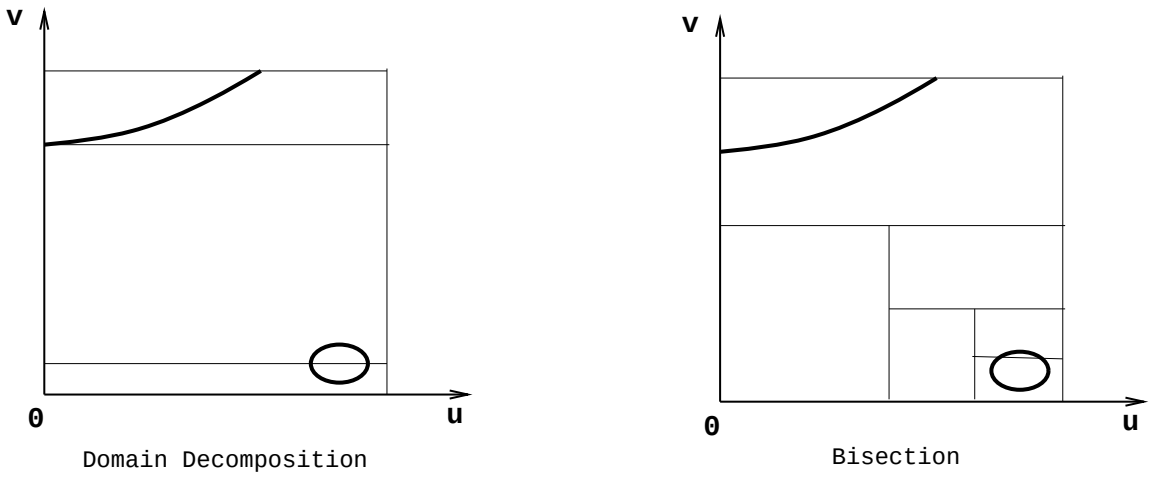

Figure 4.7: Comparing Domain Decomposition and Bisection

level of subdivision much faster than bisection. This is because our method is a form of guided subdivision as opposed to blind partitioning adopted by bisection.

However, there are instances when the algorithm does not reduce the region size appreciably. This usually happens when the intersections are very close to the corners of the region. One such example is described in Figure 4.8. These cases can be detected easily because the area of one subdomain is almost as large as that of the domain before subdivision. When such instances are encountered, bisection is performed once on the domain to break the symmetry (of points in set S). Domain decomposition is then performed on each half. The step size of tracing is determined by the size of each region.

In the worst case, for n components of the intersection curve, domain decomposition can perform $O(n^2)$ subdivisions. This is asymptotically as bad as bisection based algorithms. Computational geometry literature provides a number of techniques like binary space partitioning trees and horizontal cell decomposition for separating line segments and curves of bounded degrees where the number of cuts performed is $O(n)$. But all these algorithms have the restriction that the boundary of individual cells contain portions of algebraic curves themselves. Essentially, portions of curves share boundaries of two adjacent cells. They do not accomplish the separability condition that we require. Other techniques

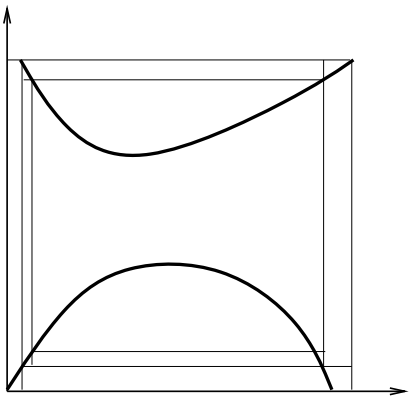

Figure 4.8: Case of slow convergence of domain decomposition

to perform cuts non-orthogonally (oblique line cuts) or low degree curves also reduce the number of subdivisions. However, the resulting domains are much more complex, and the algorithms and data structures required to perform and maintain such subdivisions are quite difficult. In our experience, for most practical cases, the complexity of these algorithms clearly outweigh their advantages.

4.3.2 Tracing in lower dimension

After domain decomposition, the entire domain ($[0,1] \times [0,1]$) is subdivided into smaller regions each with at most one curve segment. Further, domain decomposition returns the two endpoints of the curve inside the region. Starting from one of the endpoints, the *tracing* algorithm computes successive curve points using the local geometry of the curve until the other endpoint is reached. Let the component be C. Given a point $\mathbf{Q}_1 = (u_1, v_1)$ the skeleton of the tracing algorithm is given below.

- Compute $D^u(u_1, v_1)$ and $D^v(u_1, v_1)$, the partial derivatives of the curve with respect to u and v, respectively. These are the components of the vector normal to the plane curve. Methods to compute the partial derivatives were described earlier.

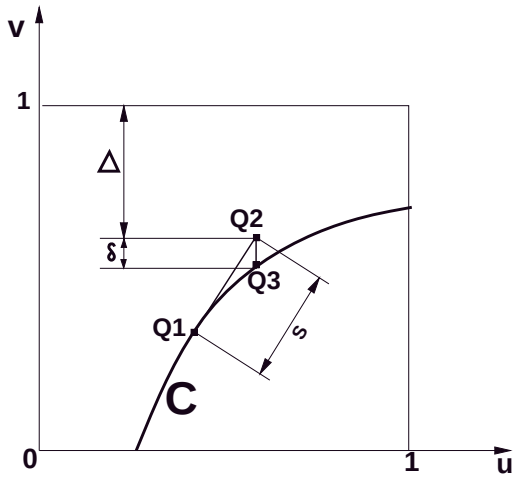

Figure 4.9: Step size computation

- Given the normal vector, find the unit vector corresponding to the tangent. Let this vector be (t_u, t_v).
- Find an approximate point $\mathbf{Q}_2 = (u_2, v_2)$, where $u_2 = u_1 + t_u S$, $v_2 = v_1 + t_v S$, and S is the step size.
- Using (u_2, v_2), converge back to the curve at $\mathbf{Q}_3 = (u_2, v_3)$, if $\mid t_u \mid > \mid t_v \mid$, or to $\mathbf{Q}_3 = (u_3, v_2)$, if $\mid t_v \mid > \mid t_u \mid$ using *inverse power iterations.*

A single tracing step is shown in Figure 4.2. The two main components of the tracing algorithm are the choosing the step size and tracing back to the curve component using inverse power iterations. We explain each of them in detail. For the rest of the analysis we will assume that $\mathbf{Q}_3 = (u_2, v_3)$.

In the tracing algorithm, we compute the eigenvalue of $\mathbf{M}(u_2, v)$ which is closest to v_2. As a result, we compute the companion matrix $\mathbf{C}$ from $\mathbf{M}(u_2, v)$ (see eq. (3.6) in chapter 3) and set $s = v_2$. In our application, we need to compute a smallest (in magnitude) eigenvalue of the matrix $\mathbf{C} - s\mathbf{I}$, which is equal to a largest eigenvalue of $(\mathbf{C} - s\mathbf{I})^{-1}$. Instead of computing the inverse explicitly (which is numerically unstable), we use inverse power iterations. Inverse iteration was described in detail in the context of curve-surface intersection in the previous chapter. Our use of the companion matrix structure to perform

LU decomposition was also discussed.

The inverse iteration terminates when an eigenvalue and its corresponding eigenvector satisfy the convergence criterion. We refine the solution obtained from inverse power iterations by minimizing the distance function $D(u, v, s, t)$ defined in eq. (4.5). Inverse power iterations followed by a few minimization steps give very good accuracy in practice.

Step Size Computation: The step size S is chosen to prevent component jumping. To avoid component jumping the following constraints are imposed on $\mathbf{Q}_2$. Let the closest distance of $\mathbf{Q}_2$ to the domain boundary be Δ as shown in Figure 4.9. As a result, any point on any other component of the curve is at least Δ away. Furthermore, the distance δ from $\mathbf{Q}_2$ to $\mathbf{C}$ is at most S. (This statement is not true in regions of very high curvature or cusps. These cases are handled separately.) If $\delta < \Delta$, inverse iteration produces a point on $\mathbf{C}$. Therefore, an upper bound on the choice of stepsize is given by the condition $\delta < S < \Delta$. We initially choose a value of S and check whether $S < \Delta$. If this constraint is not satisfied we refine the value of S using a binary search over the range $[0, S]$. Thus making use of domain decomposition and inverse power iterations, we ensure that there is no component jumping during tracing. It is possible to compute less conservative step sizes by using higher order derivatives of the intersection curve [Dok85, DSY89]. However, we feel that the complexity of computing higher order derivatives is much more than tracing with a smaller step size.

4.4 Singularities

In this section, we describe algorithms to detect singular points. Different types of singularities that can occur in intersection curves are shown Figure 4.10. Algebraically the singularities are classified by the number of branches or *places* the curve has at that point [Abh90]. The main difference between cusps and nodes (also tacnodes) is that the former has only one branch while nodes have more than one branch.

The tracing algorithm evaluates an algebraic plane curve ($D(u, v) = 0$). Singularities on the plane curve $D(u, v) = 0$ are characterized by the common solutions of $D(u, v) = D^u(u, v) = D^v(u, v) = 0$. Singularities on the intersection curve correspond to

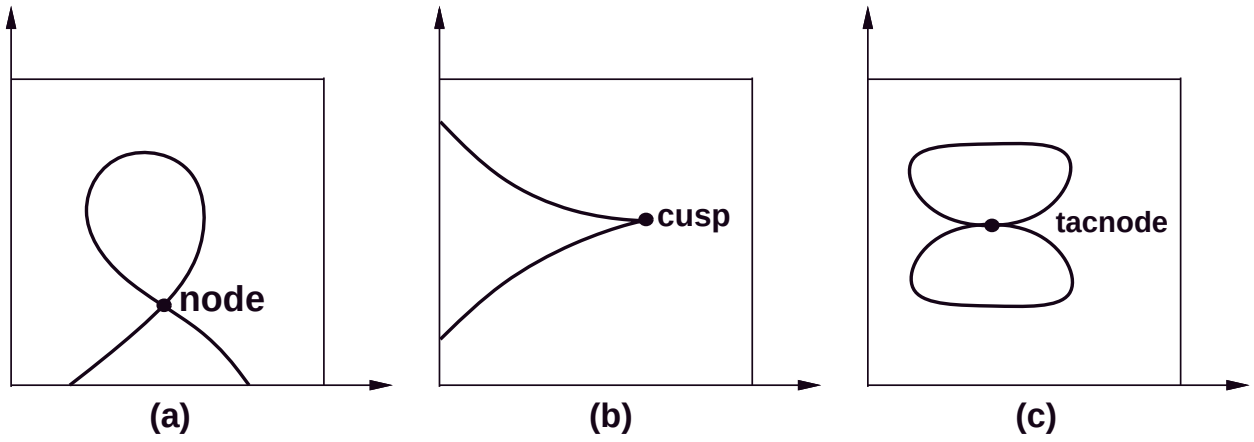

Figure 4.10: Types of singularity (a) Noop (b) Cusp (c)Tacnode

points where the tangent vector is undefined. The tangent to the intersection curve is obtained by taking the cross-product of the surface normals at that point. As a result, the preimages of singular points on the intersection curve $\mathbf{I}$ are the common solutions of

$$\begin{aligned} \mathbf{F}(s,t) &= \mathbf{G}(u,v) \\ (\mathbf{F}^s(s,t) \times \mathbf{F}^t(s,t)) \times (\mathbf{G}^u(u,v) \times \mathbf{G}^v(u,v)) &= (0 \quad 0 \quad 0)^T \end{aligned} \tag{4.7}$$

The curve $\mathbf{I}$ may have more than one branch at the singularity. The *nodes* on the intersection curve and the plane curve are related by the following lemma.

Lemma 1 *If the surface has no self-intersections, a* node *on the plane curve* $D(u,v) = 0$ *corresponds to a* node *on the intersection curve* $\mathbf{I}$.

Proof: A patch is said to be *faithfully* parametrized if the mapping from parametric space to the surface is bijective. In other words, there are no self-intersections on the surface.

Let $\mathbf{Q}$ be a singular point of the curve in the domain and $\mathbf{P}$ be its image on $\mathbf{I}$. Corresponding to each branch of $\mathbf{P}$, there is a sequence of points on the curve $D(u,v) = 0$ converging to $\mathbf{Q}$. The images of these points on the intersection curve converge onto $\mathbf{P}$. The one-to-one mapping would imply that there are different branches in the neighborhood of $\mathbf{P}$ as well. Therefore, $\mathbf{P}$ corresponds to a *node*.

□

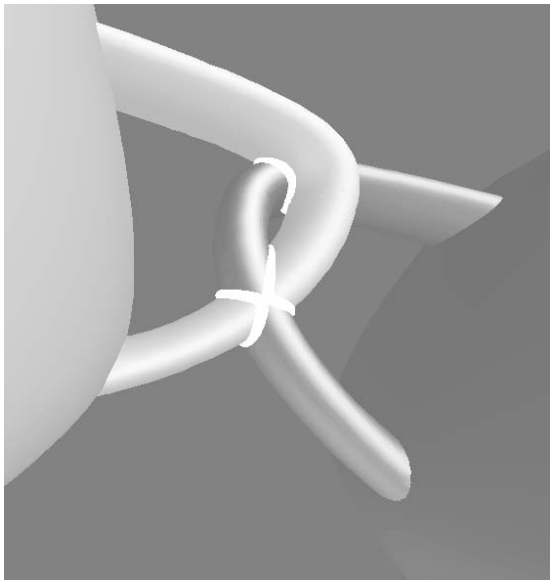

Figure 4.11: Teapot handles intersecting at a tacnode

However, there is no direct relationship between cusps on the plane curve and those on the intersection curve. In particular, $D(u,v) = 0$ may have cusps but $\mathbf{I}$ need not have cusps and vice-versa. Since we are evaluating the plane curve, we compute all the singularities and the branches.

In general the problem of computing the singularities in the intersection curve of high degree surfaces in floating point arithmetic can be numerically unstable [FR87]. Algorithms based on exact arithmetic and birational transformations have been proposed in [AB88a]. However they are computationally very slow. Our algorithms are based on the local geometry of the curve and the properties of the representation $\mathbf{M}(u,v)$. The algorithm proposed here assumes that all the singular points on the curve are geometrically isolated and well apart. In a design process, the designer often tends to use operations that result in singular intersections. Our algorithm can handle these situations well as long as the designer generates singular points that are at least separated by a specified tolerance.

4.4.1 Detection of Cusps

The previous section described a method of detecting nodal singularities during domain decomposition. Domain decomposition made use of the fact that any region with just two boundary intersections consists of a single curve component. However this curve could contain cusps which might introduce problems during tracing. Cusps on the plane curve are computed based on the following lemma.

Lemma 2 *Given a singular point (u_1, v_1) on the curve, one of the following must be true,*

- $\mathbf{M}(u_1, v_1)$ *has more than one zero singular value, or*
- *the entries $g^u_{nn}(u_1, v_1)$ and $g^v_{nn}(u_1, v_1)$ obtained after performing Gaussian elimination of $\mathbf{M}(u_1, v_1)$ are both zero.*

Proof: In section 4.1.3 it was shown that using a slight variant of Gaussian elimination, $D(u_1, v_1) = \Pi^n_{i=0} g_{ii}$ and $D^u(u_1, v_1) = D(u_1, v_1) \sum^n_{i=0} \frac{g^u_{ii}}{g_{ii}}$. It is a well known fact that if complete pivoting is used during Gaussian elimination then for $i < j$, $g_{ii} \geq g_{jj}$.

Since (u_1, v_1) is a point on the curve, $D(u_1, v_1) = 0$. Therefore, at least $g_{nn}(u_1, v_1) = 0$. In addition, since (u_1, v_1) is also a singular point, $D^u(u_1, v_1) = D^v(u_1, v_1) = 0$. Thus,

$$\Pi^n_{i=0} g_{ii}(u_1, v_1) \quad = \quad \Pi^n_{i=0} g_{ii}(u_1, v_1) \sum_{i=0}^{n} \frac{g^u_{ii}(u_1, v_1)}{g_{ii}(u_1, v_1)} \quad = \quad 0.$$

This implies one of two cases -

- at least another $g_{ii}(u_1, v_1) = 0, i \neq n$, or
- $g^u_{nn}(u_1, v_1) = g^v_{nn}(u_1, v_1) = 0$.

The latter case satisfies the second part of the lemma. If the former is true, then $\mathbf{M}(u_1, v_1)$ must be rank deficient by at least two. This is equivalent to two or more singular values being zero.

□

The tracing algorithm computes the partial derivatives at each stage and checks for the conditions in Lemma 2. If one of them is *approximately* (if a second singular value is less than 10^{-5}, or the partial derivatives are each less than 10^{-3}) satisfied, we conjecture that we are near a cusp. Tracing is then abandoned on this path and started from the other endpoint. If the curve has a cusp in this region, the paths from the two endpoints meet at this cusp. Once the two paths come close (as demanded by the application) to each other the tracing stepsize is progressively reduced. Once the two paths are close enough (smaller than specified tolerance), the cusp point is obtained by minimizing an energy function. The energy function is determined by the condition satisfied in Lemma 2. For example, if the second condition is satisfied, the energy function is $E(u,v) = (g^u_{nn}(u,v))^2 + (g^v_{nn}(u,v))^2$.

4.5 Robustness and Efficiency

The two most important considerations in the design of any surface intersection algorithm are *robustness* and *efficiency*. There is a clear trade-off between these two, since the more care we take to improve the robustness of our algorithm the slower the execution time of the algorithm. While it is almost impossible to provide an algorithm that can satisfy both completely together, one must at least try to design an algorithm that can provide a good fraction of both in most cases and can be fine-tuned according to the requirements of the application.

Our algorithm has been tested on a number of models, and we have obtained encouraging results. There are no benchmarks available to test its efficiency, but our algorithm compares favorably to many of the published timings. For example, it performs an order of magnitude faster than techniques like *interval arithmetic*. [Sny92] reports that a *difference* operation (Boolean operation) between a bumpy sphere and a cylinder using trimmed parametric surfaces takes order of a few minutes on a HP workstation. We can perform such operations on similar solids (like generalized prisms, cylinders, spheres etc.) in a few seconds on the same machine. By performing extra work like tracing all complex paths during loop detection (see chapter 5) and perform domain decomposition to much smaller domain extents before isolating singularities, we can increase the robustness guarantees of

our algorithm.

In order to achieve robustness, a general intersection algorithm must be able to determine the *conditioning* of the problem. The conditioning becomes more significant because of errors introduced by numerical computations. If the input data changes by ϵ, the output results will change by a function $\delta(\epsilon)$. For very small values of ϵ, there may exist a constant κ such that $\delta(\epsilon) \approx \kappa\epsilon$ [Hof89]. If κ is small the problem is said to be *well-conditioned*. A large value of κ signifies an *ill-conditioned* problem. The value κ is called the *condition number*. However, it is nontrivial to calculate κ for surface intersection problems. Because of such difficulties, we restrict ourselves to robustness issues for well-conditioned problems only.

We identify four main areas in our algorithm where robustness enhancing modifications can be made. They are

- **tracing -** We perform tracing after domain decomposition. We had shown that when using inverse power iterations, convergence back to the curve is guaranteed except in places of very high curvature (or cusps). In such situations, we reduce the step size by half and repeat the process. Robustness can be enhanced by reducing the *minimum step size*.

- **component jumping -** Domain decomposition is adopted to prevent component jumping. Robustness is determined by the extent to which subdivision is carried out. It is possible to control this effectively based on the requirement of the application.

- **singularities -** The accurate detection of all the singular points is contingent on the assumption that the singular points are separated and well apart. This assumption is not unreasonable for most of the practical applications, but a pathological case violating this condition can be created.

- **loop identification -** We identify loops by performing curve-surface intersection and complex tracing. The number of paths to be traced grows as a cubic function of the degree of a patch. In most practical examples it is enough to trace a small fraction of the paths, but to ensure absolute robustness, all the paths have to be traced. More details are given in the next chapter.

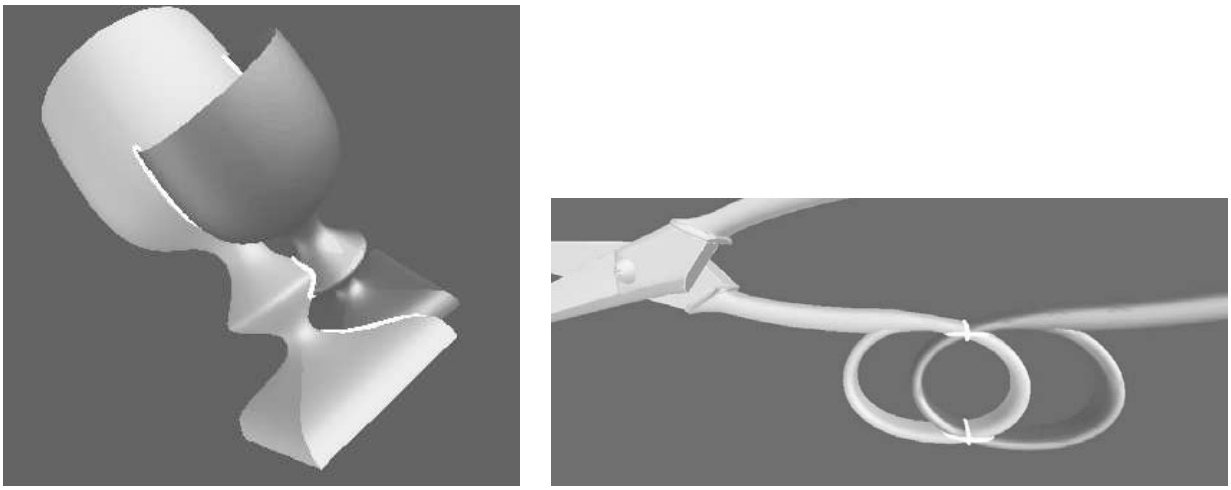

Figure 4.12: (a) Intersecting Goblets (b) Intersecting Scissors

4.6 Models Composed of Piecewise Surfaces

The algorithms in the previous sections compute the intersection of a pair of Bézier or algebraic surfaces only. Most models consist of tens or hundreds of such surface patches. To compute the intersections of these models we compute the intersection of each overlapping pair.

Typically only a small percentage of the $O(N^2)$ possible surface pairs intersect. Our algorithm prunes out most of non-intersecting combinations using spatial techniques based on bounding boxes and linear programming.

- Initially the axis-aligned rectangular bounding boxes are computed for each Bézier surface for both models. Each bounding box is then projected on each of the three axes to obtain three sets of intervals. We denote them by *x-lists, y-lists and z-lists*. The *x-lists* is sorted first and all the non-intersecting pairs are discarded. The *y-lists* of the remaining pairs are again sorted to check for overlaps. This is repeated for the *z-lists* as well. The pairs that remain at the end of this operation have intersecting bounding boxes.

- The Bézier patches are contained in the convex hull of their control points. Given all the pairs of surfaces obtained after bounding box tests, a test for separating plane

Model	No. of patches	No. of intersections	Timing (in sec.)		
			Curve-Surf. intersection	Tracing	Total
Teapot	32	8	0.4	2.6	3.5
Goblet	72	57	2.4	4.9	7.8
Scissor	505	82	3.1	7.8	11.6

Table 4.1: Performance Statistics of Intersection Algorithm

between their control polytopes is performed using linear programming. The existence of a separating plane implies that the surface pairs have no intersection.

After execution of these two steps, the intersection algorithm is applied to each existing pair. The intersection of each patch pair results in a (possibly empty) set of open components and loops. The open components could be part of a larger curve in the model. Two open components are spliced together if an endpoint of one is coincident with an endpoint of the other (actually the test is made over a small disk of influence). Finally, a piecewise representation of each component of the intersection curve is obtained for the original models. The results of the intersection algorithm on large models like goblets and scissors are shown in Figure 4.12.

4.7 Implementation and Performance

The algorithm has been implemented and its performance was measured on a number of models. The algorithm uses existing EISPACK and LAPACK routines for the matrix computations. The algorithm was implemented on a high-end SGI Onyx workstation with an R4400 CPU and a clock frequency of 250MHz. In Table 4.1, we illustrate the algorithm's performance on different models. The second column represents the number of patches in each model, and the third column represents to the number of patch pair intersections after linear programming. We have not shown the time taken to perform bounding-box tests and linear programming. We believe that a number of optimization techniques can be incorporated in our implementation to give better results. Unfortunately, there are no existing benchmarks available to test our algorithm and there are very few

published performance results on surface intersection algorithms. Our algorithm performs almost 10-15 times faster than algorithms based on interval arithmetic [Sny92].

Chapter 5

Loop Detection Algorithm

In our discussion of the surface intersection problem in chapter 2, the intersection curve was formulated as an algebraic curve in higher dimensions ($\mathcal{R}^4$, to be precise). Further, these algebraic curves typically are of high degree with multiple components (as shown in Figure 1.4). In most cases we are interested in evaluating all the components in the subset of the real domain. Components that intersect with the boundaries of the real domain are called *open* components. *Loops* are components where the curve folds back into itself and is completely contained inside the domain of interest. In this chapter, we describe two techniques for efficient and accurate evaluation of loops using a combination of symbolic and numeric methods. The first method is applicable to arbitrary algebraic curves, while the second is suited only for planar sections of surfaces. The loop detection and evaluation algorithms are used in our surface-surface intersection algorithm to compute the intersection curve.

Evaluation of high degree algebraic curves is fundamental in a number of areas of scientific computing. Algebraic sets are widely used for representing objects and constraints in computer graphics, geometric modeling, robotics, computer vision and molecular modeling. Many problems like surface-surface intersection, offsets of curves and surfaces, slicing operations on surface models, Voronoi sets generated by curves and surfaces in geometric modeling [Hof89], kinematic analysis of a redundant robot [Cra89], robot motion planning [Can88], object recognition in computer vision [PK92] and conformation space of molecular

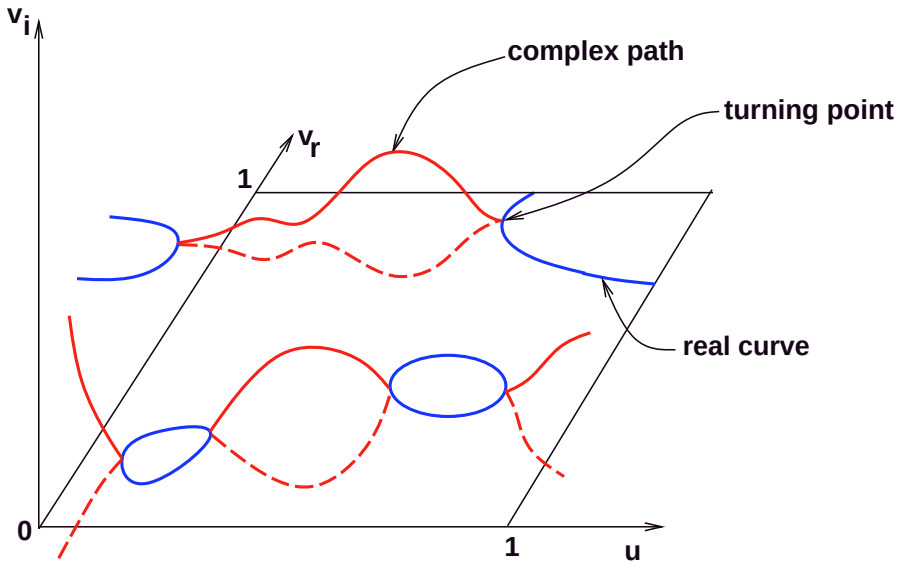

Figure 5.1: Algebraic curve continuous in complex projective plane

chains [CH88] correspond to evaluating algebraic curves.

5.1 Loop Detection I: Algebraic Curves

We apply our loop detection algorithm to find all the loops of an algebraic plane curve. We use a matrix determinant representation (like the one described in chapter 4) to deal with high degree curves, but any general form (like power or Bernstein basis) is sufficient for our algorithm. In this section, we shall describe our loop detection algorithm assuming that we have a matrix representation of the plane curve (denoted by $\mathbf{M}(u, v)$).

The curve we are evaluating is an algebraic plane curve in the complex projective plane defined by u and v. We are, however, interested only in finding the part that lies in the portion of the real plane defined by $(u, v) \in [0, 1] \times [0, 1]$. If we relax this restriction so that one of the variables, say v, can take complex values, this curve is defined as a continuous set consisting of real and complex components (see Figure 5.1). Before we give our algorithm, we introduce some definitions.

Definition 6 *Turning points are points on the curve where the tangent vector, as projected*

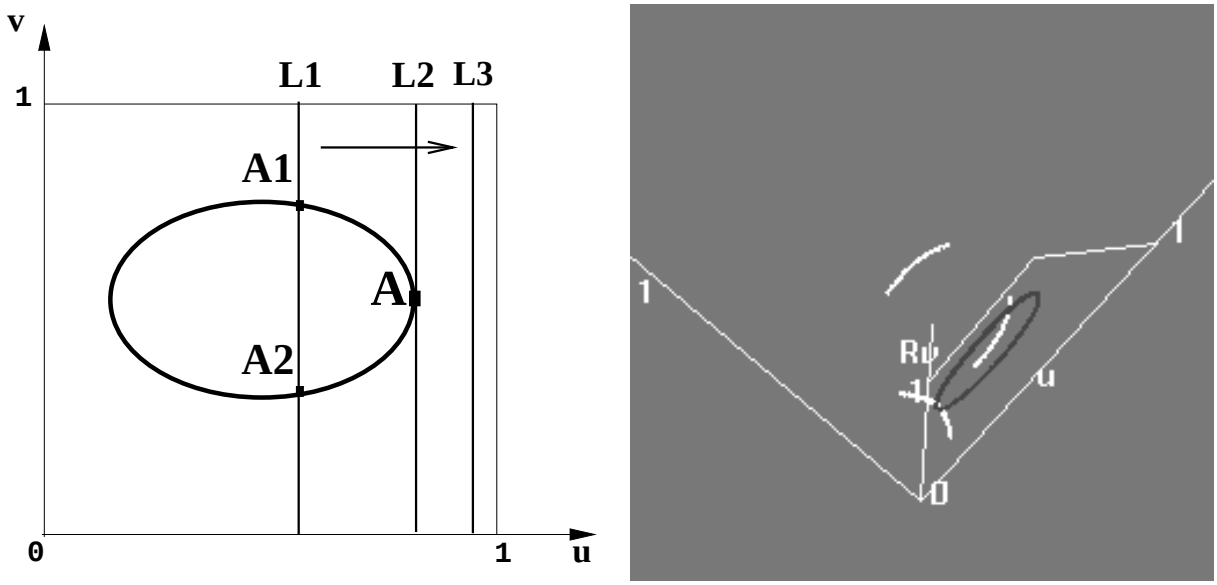

Figure 5.2: Characterization of loops based on complex tracing

in the (u, v) space, is parallel to the u or v parameter axes. In other words, one of the partial derivatives (with respect to u or v) of the intersection curve is 0.

A turning point where the tangent is parallel to v-axis is called a *u-turning point*. We classify u-turning points into *left u-turning points* and *right u-turning points*. A point (u_1, v_1) is a *left u-turning point* if the curve goes into the complex domain in the left neighborhood of u_1 ($u = u_1 - \delta$, where δ is a small positive value). A point (u_1, v_1) is a *right u-turning point* if the curve goes into the complex domain in the right neighborhood of u_1 ($u = u_1 + \delta$).

Definition 7 *Isoparametric curves are curves lying on a parametric patch (surface) where one of the parameters of the patch (u or v) remains constant.*

The main idea behind our loop detection algorithm is based on the following lemma.

Lemma 3 *If the curve in the real domain $[0, 1] \times [0, 1]$ consists of a closed component, then two arbitrary complex conjugate paths meet at one of the real points (corresponding to a turning point) on the loop.*

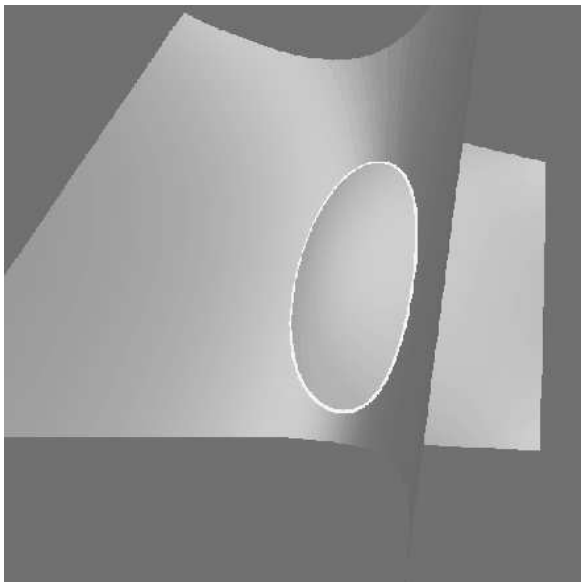

Figure 5.3: Two surfaces intersecting in a loop

Proof: The proof is based on *Bezout's theorem* which states that *if f and g are two algebraic curves of degree m and n respectively, then f and g intersect in exactly mn points in the complex domain counted properly, or they have a common component.* We use Bezout's theorem and the fact that the curve forms a continuous set in the complex domain to prove the result.

Let us consider an algebraic curve that forms a loop in the real domain, like the one shown in Figure 5.3. All isoparametric curves of one variable on a surface have the same degree, namely the degree of the other parameter defining the surface. Therefore, the number of intersections of the algebraic curve with any isoparametric curve equals the Bezout bound in complex space. Figure 5.2 (left) illustrates the argument. The line $L1$ intersects the curve at two different real points. As we move the line continuously from $L1$ to $L2$, the two intersection points come closer, and at line $L2$, both of them coincide to form a double root maintaining the intersection count constant. This double root also corresponds to a u-turning point. As $L2$ approaches $L3$, all the real intersections vanish. Since the algebraic curve is continuous in complex domain, the double root bifurcates into complex values and occur in conjugate pairs (because all curve coefficients are real).

Now if the sweep is started from $L3$ towards $L2$, the complex conjugate components come closer together, and at line $L2$, their imaginary part vanishes to yield a double root. This argument shows that the complex conjugate pairs meet the real plane at some turning point of every component. Observing that every loop component must have at least two turning points completes the proof.

□

The domain of the intersection curve in the complex space is shown in Figure 5.2 (right). The third axis corresponds to the imaginary components of v. It represents a continuous component of the intersection curve. The white curve is the intersection curve in the complex space and the dark curve is the part of the curve that lies in the real plane.

We need only one start point on each loop to trace it completely. So we restrict ourselves to u-turning points. Henceforth, we shall use *turning points* to denote u-turning points. Our domain has changed from the real plane to a three dimensional space formed by u, v_r and v_i, where v_r and v_i are the real and imaginary values of v. To compute the turning points on the curve, we combine boundary computations with complex tracing.

Boundary intersections: Boundary intersections refer to the portions of the curve that lie along the boundary of the surface (in our case, when $u = 0$, $u = 1$, $v = 0$ or $v = 1$). This corresponds to a curve-surface intersection computation which we reduced to an eigenvalue problem in chapter 3. However, in this case, we have to evaluate all the complex eigenvalues. Since *algebraic pruning* converges to only few of the complex eigenvalues in the domain, we do not use this approach. We compute all the eigenvalues using the QR algorithm.

Tracing: Given the complex start points on the boundary of the surface, we use tracing in the complex domain to reach the turning points on every loop. The general tracing step proceeds as follows. Given a point on the curve, an approximate value of the next point is obtained by taking a small step size in a direction determined by the local geometry of the curve (tangent or curvature information). This approximate value is then refined using iterative techniques. We use *inverse power iterations* to trace the curve in complex space. This method was described in detail in chapters 2 and 3.

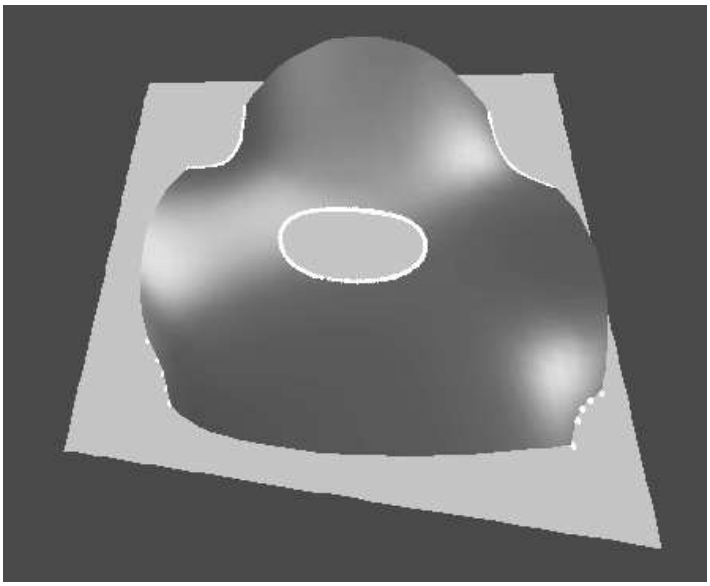

Figure 5.4: A pair of intersecting surfaces

The basic technique of obtaining all the turning points is to evaluate the starting complex points on one of the boundaries and follow all these paths until they either leave the domain or meet the real plane. Unfortunately, these are not the only complex paths that could lead to a turning point. There could be complex paths starting from *right turning points* of some other component of the intersection curve. This can be illustrated by considering the intersection between a bicubic patch and a plane (see Figure5.4). The curve $\mathbf{M}(0, v) = 0$ is a cubic curve with all real solutions. This implies that there cannot be any complex solution to this equation. Therefore, the left turning point on the loop is connected in complex space to the right turning point of another component. So we use the following strategy to complete a sweep of the complex paths from $u = 0$ to $u = 1$.

Since complex solutions occur in conjugate pairs for real algebraic equations, we restrict ourselves to complex paths whose imaginary parts are strictly positive. When a complex path touches the real plane the imaginary part (of v) must reach some small constant value $\epsilon > 0$ before reducing to zero. These are precisely the common points of the curve with the plane $v_i = \epsilon$. In other words, we are trying to find all the real solutions to the equation $\det \mathbf{M}(u, v_r + i\epsilon) = 0$ $(i = \sqrt{-1})$. Expanding out the expression and collecting

the real and imaginary terms we can write

$$\det(\mathbf{M_r}(u, v_r) + i\mathbf{M_i}(u, v_r)) = 0 \tag{5.1}$$

Lemma 4 *The solutions* $(u, v_r) \in \mathcal{R}^2$ *satisfying equation (5.1) also satisfy the solution of* $\det(\mathbf{P}(u, v_r)) = 0$, *where*

$$\mathbf{P}(u, v_r) = \begin{pmatrix} \mathbf{M_r}(u, v_r) & -\mathbf{M_i}(u, v_r) \\ \mathbf{M_i}(u, v_r) & \mathbf{M_r}(u, v_r) \end{pmatrix} \tag{5.2}$$

Proof: If (5.1) is satisfied, the matrix $\mathbf{M_r}(u, v_r) + i\mathbf{M_i}(u, v_r)$ is singular. This implies that there is at least one non-trivial vector in its kernel of the form $\mathbf{a} + i\mathbf{b}$. Therefore,

$$(\mathbf{M_r}(u, v_r) + i\mathbf{M_i}(u, v_r))(\mathbf{a} + i\mathbf{b}) = \mathbf{0}$$

Equating the real and imaginary components to zero separately, we get

$$\begin{aligned} \mathbf{M_r}(u, v_r)\mathbf{a} \; - \; \mathbf{M_i}(u, v_r)\mathbf{b} \; &= \; 0 \\ \mathbf{M_i}(u, v_r)\mathbf{a} \; + \; \mathbf{M_r}(u, v_r)\mathbf{b} \; &= \; 0 \end{aligned}$$

These two equations can be combined in a matrix form as

$$\begin{pmatrix} \mathbf{M_r}(u, v_r) & -\mathbf{M_i}(u, v_r) \\ \mathbf{M_i}(u, v_r) & \mathbf{M_r}(u, v_r) \end{pmatrix} \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix} = \mathbf{0}$$

Therefore, $\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix}$ is a non-zero vector in the kernel of $\mathbf{P}(u, v_r)$. Hence the proof.

□

As before, the solutions to (5.2) can be posed as the singular set of matrix $\mathbf{P}(u, v_r)$. The singular set of $\mathbf{P}(u, v_r)$ is a discrete point set. The order of the matrix $\mathbf{P}(u, v_r)$ is twice that of $\mathbf{M}(u, v)$. Therefore, there are twice as many paths to trace in general. For an intersection curve, if the patches are of degree $m \times n$ and $p \times q$, then at most $2mn \min(p, q)$ paths have to be traced.

Initially we form the companion matrix of $\mathbf{P}(u, v_r)$, C_p, similar to the one in Eq.(3.6). We compute all the eigenvalues of C_p at $u = 0$ (we expect all of them to be

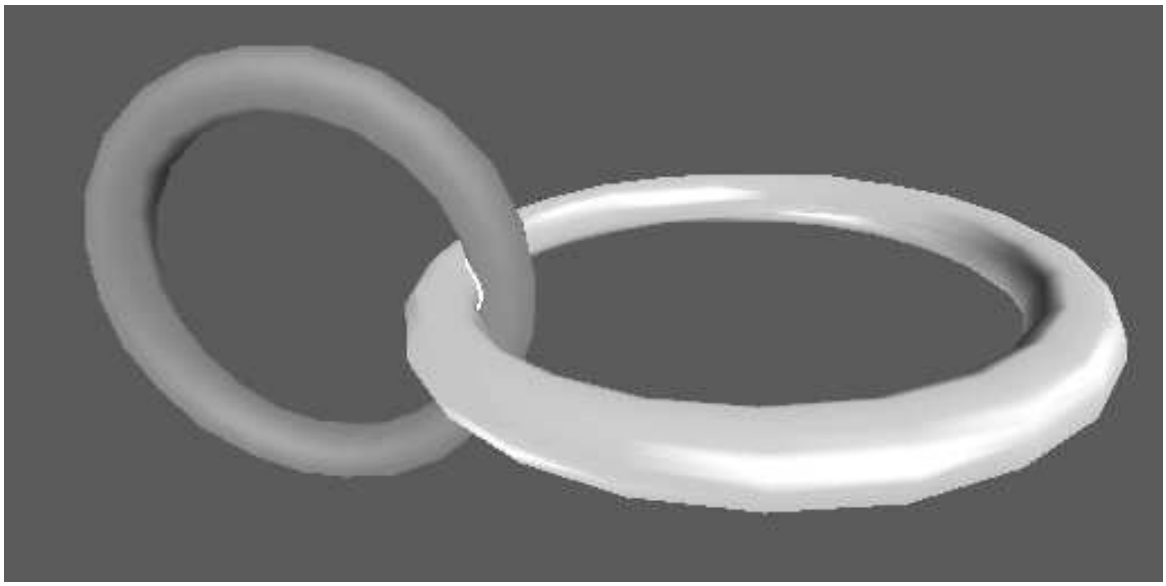

Figure 5.5: Two tori intersecting in a small loop

complex). We use them as starting points and trace all the paths in increasing u direction until it either crosses the $u = 1$ plane or become real. All the real values of v_r are points lying very close to the turning points of the intersection curve. The corresponding point on the real plane is (u_r, v_r). This is used as an initial guess to approach the actual turning point using inverse power iterations.

5.2 Implementation, Performance and Applications

The loop detection algorithm has been implemented and its performance was measured on a number of models. The algorithm uses existing EISPACK [GBDM77] and LAPACK [ABB+92] routines for some of the matrix computations. We report the results of our implementation on an SGI Onyx workstation with 128MB of main memory and a specFP rating of 98.1.

Tracing in the complex space is a guided form of search for all the turning points of the loops. In a purely algebraic form, all the turning points of a curve $f(u, v) = 0$ can be posed as the common solutions of $f(u, v) = f_u(u, v) = 0$. Using the Bezout bound, the number of possible turning points is *quadratic* in the degree of the curve. However, the

maximum number of complex paths that need to be traced in our loop detection algorithm is *linearly* related to the degree of the curve. The performance of the tracing algorithm is directly dependent on the efficiency of linear system solvers. While methods like LU and LQ decomposition take $O(n^3)$ operations, our use of the special structure of the matrix has almost quadratic complexity. Our implementation of the algorithm consists of two major modules - the boundary computation part and the complex tracing part. The boundary computation module computes starting points on all the complex paths using eigensolvers. For our implementation, we used an ϵ (see previous section) value of 0.01. The complex tracing step is done using inverse power iterations. The total time to trace a path across the domain is about 20-50 milliseconds.

5.2.1 Application to surface intersection

Our loop detection algorithm is part of the surface intersection algorithm, which was described in chapter 4. This, in turn, has been applied to a number of intersecting surfaces and has worked well consistently. Our algorithm evaluated the intersection curve of the surfaces in Figure 5.3 in about 4 seconds. A total of 54 complex paths were traced which consumed about 70% of the time.

For efficiency considerations, it may not be necessary to trace all the complex paths. In our test examples, very few complex paths meet the real plane inside the domain of the patch. It is very difficult to give exact algorithms to prune out paths that cannot touch the real plane because of the high degree nature of the curve. However, through repeated application of our algorithm we found that paths that start very high in the complex axis rarely hit the real plane. This observation could be used to speed up the tracing step depending on the robustness requirements of the application.

In order to compare our algebraic method with the Gauss map based approaches to loop detection, we implemented Hohmeyer's algorithm (in the context of surface intersection) using pseudo-normal patches [Hoh91]. Hohmeyer's algorithm performed slightly slower than our algorithm on the example in Figure 5.3. Eight levels of subdivision were performed, and most of the time was consumed in the repeated computation of the Gauss map and application of linear programming. We observed that his algorithm works very well

when the patches are relatively flat and do not intersect in loops. However, these methods perform a number of subdivisions (to achieve the no loop criterion) when the patches have high curvature and intersect in small loops or singularities.

Hybrid approach: We suggest the following hybrid approach when dealing with intersection curves. Initially, we test for the possible absence of loops using the Gauss map approach. We perform subdivisions based on this approach for about 2-3 levels. In the event that Gauss maps are still not separated, we apply our algorithm to identify turning points on loops in the smaller domains. This method has been applied to compute intersections of high degree surfaces. On an average, our algorithm takes less than one second to compute one patch-pair intersection. For the intersecting surfaces in Figure 5.3 and Figure 5.5, our method performs better than Hohmeyer's algorithm. His method, however, performed better when applied to the surfaces in Figure 5.4.

5.2.2 Silhouette Computation

To show the generality of our algorithm, we have also applied it to compute the silhouettes of parametric surfaces. The property of the silhouette curve is that it subdivides the surface into front and back facing regions. We shall restrict our discussion to surfaces whose silhouette (from a given viewpoint) is a curve on the surface. We now describe our formulation of the silhouette curve on a parametric (represented as a tensor product Bézier [Far93]) patch from a given viewpoint.

We assume for the sake of simplicity that the viewpoint is located at $(0, 0, -\infty)$. It is easy to see that even if this is not the case, one can always achieve it by applying an appropriate perspective transformation to the parametric surface $\mathbf{F}(u, v)$. We also require that all the surfaces are at least C^1 everywhere. We formulate the silhouette curve as an algebraic plane curve in the domain of $\mathbf{F}(u, v)$. Figure 5.6 shows a patch that has a loop as part of its silhouette.

Let $\mathbf{F}(u, v)$ denote the parametric (differentiable) surface and let $\phi_1(u, v), \phi_2(u, v)$ and $\phi_3(u, v)$ denote the mappings from the parametric space to (x, y, z) space.

$$\mathbf{F}(u, v) = \langle X(u, v), Y(u, v), Z(u, v), W(u, v) \rangle$$

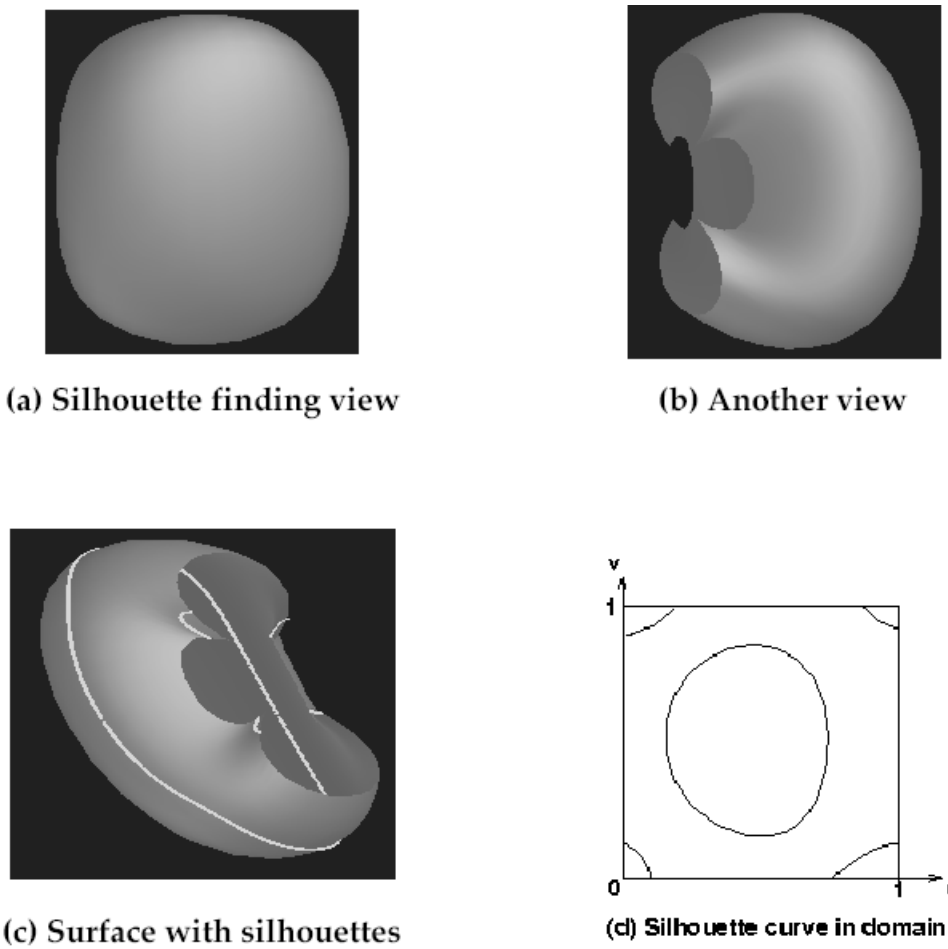

Figure 5.6: Loop as part of a silhouette curve

$$\phi_1(u,v) = \frac{X(u,v)}{W(u,v)}, \quad \phi_2(u,v) = \frac{Y(u,v)}{W(u,v)}, \quad \phi_3(u,v) = \frac{Z(u,v)}{W(u,v)}$$

In the rest of this section, we shall drop the (u,v) suffixes from all the functions for more concise notation. The $z-$component of the normal at an arbitrary point on the surface is given by the determinant

$$N_z = \begin{vmatrix} \phi_{1_u} & \phi_{1_v} \\ \phi_{2_u} & \phi_{2_v} \end{vmatrix} \tag{5.3}$$

where ϕ_{i_u} and ϕ_{i_v} denote the partial derivatives of the appropriate function ϕ_i with respect

to u and v.

$$\phi_{1_u} = \frac{(WX_u - W_uX)}{W^2} \qquad \phi_{1_v} = \frac{(WX_v - W_vX)}{W^2}$$
$$\phi_{2_u} = \frac{(WY_u - W_uY)}{W^2} \qquad \phi_{2_v} = \frac{(WY_v - W_vY)}{W^2}$$

On the silhouette curve, $N_z = 0$. Since $W(u,v) \neq 0$, we can express the plane curve representing the silhouette as the determinant

$$N_z = \begin{vmatrix} (WX_u - W_uX) & (WX_v - W_vX) \\ (WY_u - W_uY) & (WY_v - W_vY) \end{vmatrix} = 0 \tag{5.4}$$

Expanding the determinant and rearranging the terms, we can express it as the singular set of the matrix $\mathbf{M}(u,v)$

$$\mathbf{M}(u,v) = \begin{pmatrix} X(u,v) & Y(u,v) & W(u,v) \\ X_u(u,v) & Y_u(u,v) & W_u(u,v) \\ X_v(u,v) & Y_v(u,v) & W_v(u,v) \end{pmatrix} = 0 \tag{5.5}$$

For curves like silhouettes, the Gauss map approach is not very practical. In order to apply their loop detection criteria on a bicubic patch (like that in Figure 5.6(b)), one would have to perform repeated subdivisions on rational patches of degree 27×27. This makes the algorithm very slow because each subdivision step takes $O(n^3)$ operations (in terms of the degree). We were able to determine all the components of the silhouette for the same surface using our algorithm in about 2 seconds. Performing boundary computations to determine all the starting points roughly takes 40% of this time. The rest of the time is spent in curve tracing. For this particular example, a total of *two complex paths* and *five real components* were traced along the entire domain. The real components of the silhouette curve in the domain are shown in Figure 5.6(d).

5.3 Loop Detection II: Surface Sectioning

In this section, we describe an algorithm to perform loop detection on intersection curves obtained by taking planar sections of surface models. This operation is widely used in rapid prototyping to obtain cross-sectional information of such models. A major concern

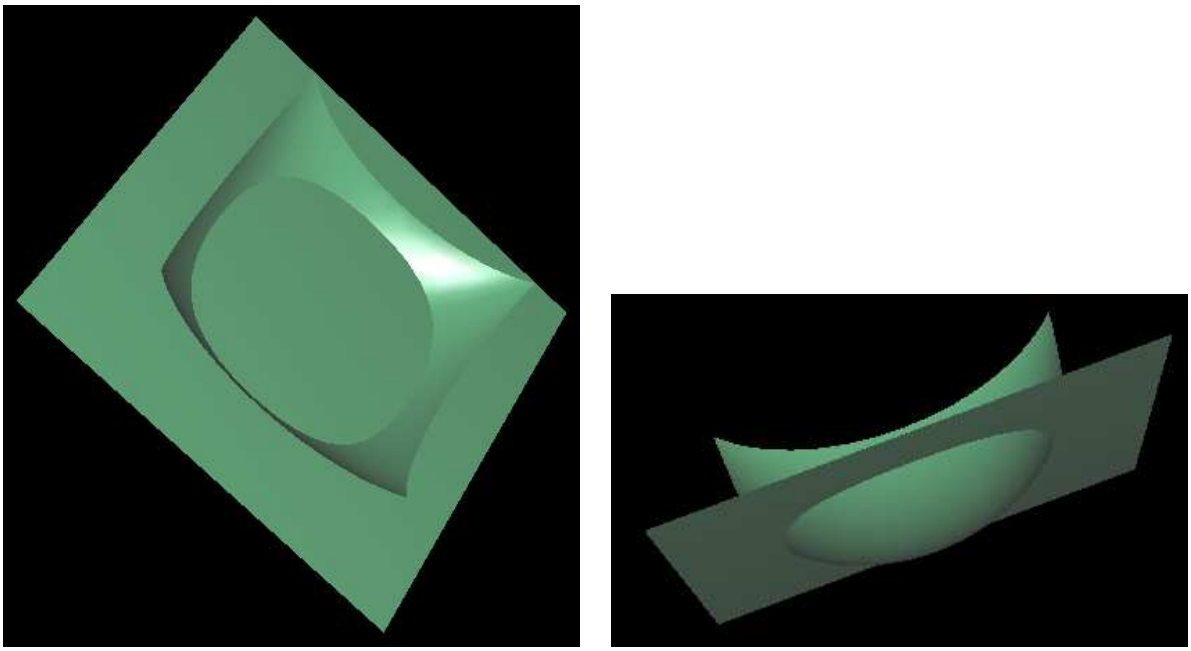

Figure 5.7: Intersection of a plane with a biquadric surface

in such applications is the correctness of the resulting curve *i.e.*, its topological type and detection of all components.

Figure 5.7 shows a simple example of a biquadratic surface intersected by a plane. In this case, the intersection curve has a single loop component. We shall now formulate the loop detection problem as critical points of a plane vector field. The vector field is obtained as the gradient of a distance function introduced by [Che89].

5.3.1 Intersection formulation using distance function

The intersection set between a pair of parametric surfaces can be formulated as a minimization problem in which the distance between two variable points on the two surfaces becomes zero. Basically, the intersection set can be expressed as the sequence of points in the two surfaces with zero distance between them.

The oriented distance function ϕ between a surface $\mathbf{Q}(s,t)$ and a point moving on

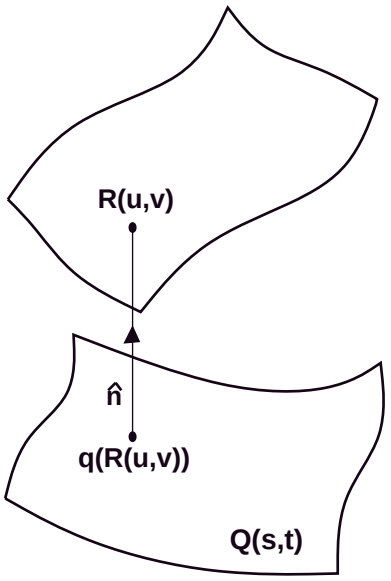

Figure 5.8: Distance function between two surfaces

another surface $\mathbf{R}(u,v)$ is defined on the (u,v) parameter space as

$$\phi(u,v) = \hat{\mathbf{n}}[\mathbf{q}(\mathbf{R}(u,v))] \bullet (\mathbf{R}(u,v) - \mathbf{q}(\mathbf{R}(u,v))) \tag{5.6}$$

where $\mathbf{q}(\mathbf{R}(u,v))$ is a point on the surface $\mathbf{Q}(s,t)$ which is nearest to the point $\mathbf{R}(u,v)$, and $\hat{\mathbf{n}}$ is the unit normal vector on $\mathbf{Q}(s,t)$ at the point $\mathbf{q}(\mathbf{R}(u,v))$ (see Figure 5.8). $\bullet$ is the dot product operator for vectors. In our case, however, one of the surfaces is a plane, and the normal is constant at all points (say $\hat{\mathbf{n}}$). Therefore the distance function becomes

$$\phi(u,v) = \hat{\mathbf{n}} \bullet [\mathbf{R}(u,v) - \mathbf{q}(\mathbf{R}(u,v))] \tag{5.7}$$

Assuming that ϕ is a well-defined distance function, the intersection set is the zero set of ϕ. There are cases when ϕ is not well-defined (when there are more than one closest point to $\mathbf{R}(u,v)$, or when the line joining the two points $\mathbf{R}(u,v)$ and $\mathbf{q}(\mathbf{R}(u,v))$ is not collinear to $\hat{\mathbf{n}}$ because of patch boundaries). However, these special cases do not occur while looking for loops in the intersection curve. As we will see in the next section, inside every loop of the intersection curve, there are critical points of ϕ. But points where ϕ is not well-defined (boundary points of surfaces) cannot be enclosed by a loop. Therefore, we can assume for the purposes of this paper that ϕ is well-defined.

5.3.2 Collinear normal points and Distance function

Sederberg [SM88, THS89] was the first to recognize the importance of collinear normals in detecting existence of closed loops in intersection problems. It is easy to see that collinear normal points between two surfaces are critical points of the distance function ϕ. This is because collinear normal points are extremal distance point pairs and the gradient vector of the distance function is zero. Therefore, if we have a method to find the number of collinear normal points within a particular domain of interest, we can use a simple subdivision scheme to compute these points to arbitrary precision. Most existing methods use the idea of rotational index of the vector field inside a closed curve [KPP90, KPW90]. However, this test is inconclusive because if a particular region contains two critical points of opposite rotational index, then we obtain net rotational index of zero. Recently [ML95] extended this to a three-dimensional vector field such that rotational index of this field decides conclusively the number of critical points (provided they are *non-degenerate*). However, their method is susceptible to failure if the sampling grid in the domain contains contours of zero Jacobians completely inside them. Further, the use of local minimization methods and Newton type marching methods to locate all the critical points are error-prone.

The following theorem formulates the gradient of the distance function. The critical points of this vector field provides the set of collinear normal points. We denote partial derivatives with respect to u (v) by subscripted u (v, respectively).

Theorem 5 *Given the oriented distance function ϕ as in eq. (5.7), the gradient is given by*

$$\begin{aligned} \phi_u(u,v) &= \hat{\mathbf{n}} \bullet \mathbf{R_u}(u,v) \\ \phi_v(u,v) &= \hat{\mathbf{n}} \bullet \mathbf{R_v}(u,v) \end{aligned} \tag{5.8}$$

Proof: The distance function $\phi(u,v)$ is given by

$$\phi(u,v) = \hat{\mathbf{n}} \bullet [\mathbf{R}(u,v) - \mathbf{q}(\mathbf{R}(u,v))]$$

Taking partial derivative with respect to u, we get

$$\begin{aligned} \phi_u(u,v) &= \hat{\mathbf{n}} \bullet [\mathbf{R_u}(u,v) - \mathbf{q_u}(\mathbf{R}(u,v))] + \hat{\mathbf{n}}_\mathbf{u} \bullet [\mathbf{R}(u,v) - \mathbf{q}(\mathbf{R}(u,v))] \\ &= \hat{\mathbf{n}} \bullet \mathbf{R_u}(u,v) \end{aligned}$$

This is because $\hat{\mathbf{n}}_\mathbf{u}$ is zero because normal does not change for a planar patch, and so is $\hat{\mathbf{n}} \bullet \mathbf{q}_\mathbf{u}(\mathbf{R}(u,v))$ since $\mathbf{q}_\mathbf{u}$ lies in the tangent plane of $\mathbf{Q}(s,t)$ and $\hat{\mathbf{n}}$ is the normal to it.

The result for partial w.r.t v can be proved similarly.

□

5.4 Loop Detection Algorithm

In this section, we describe our loop detection algorithm based on finding all the critical points of a two-dimensional vector field. Sturm sequences were introduced by Hermite (1853) in order to count the number of real roots of a univariate polynomial inside a given interval. Sturm sequences are generated by performing g.c.d. (greatest common divisor) computation using Euclid's algorithm on the given polynomial and its negative derivative. An introduction to univariate Sturm sequences was given in chapter 2. The number of real roots is computed by counting the number of sign changes of this sequence at the endpoints of the interval. Extending this idea to multivariate polynomial systems (that yield zero-dimensional solution sets) has been the focus of research for quite some time. Milne [Mil92] introduced the *volume function* which essentially achieved the extension to multivariate polynomial systems.

5.4.1 Multivariate Sturm sequences

Here, we describe briefly the algorithm proposed by Milne [Mil92] to compute the number of common real solutions of n polynomials in n variables inside an n-dimensional rectangle. This algorithm is an extension of the univariate case which constructs a polynomial sequence, and measures sign variations of this sequence at the endpoints of the interval. We restrict ourselves to the case when $n = 2$.

Given two polynomials, $f_1(s,t)$ and $f_2(s,t)$, we construct the *volume function*, $V(u,s,t)$, as

$$V(u,s,t) = \frac{Res_{a_2}(Res_{a_1}(f_1(a_1,a_2),f_3), Res_{a_1}(f_2(a_1,a_2),f_3))}{u^{deg(f_1(s,0))deg(f_2(s,0))}},$$

where $f_3(u,s,t,a_1,a_2) = u+(s-a_1)(t-a_2)$, Res_x refers to the resultant of two polynomials after eliminating x, and deg refers to the degree of the polynomial. a_1 and a_2 are two new

symbolic variables. Because of the special bilinear form of f_3, the two resultants involving elimination of variable a_1 inside the expression for volume function is easily performed by hand. We use the Sylvester resultant [Sal85] (see chapter 2) to eliminate variable a_2.

We use an algorithm based on multivariate interpolation [MC93] to compute the resultant of a set of polynomials efficiently. The main bottleneck in most resultant algorithms is the symbolic expansion of determinants. Most of the computer algebra systems use symbolic algorithms like polynomial manipulations for resultants, which are very expensive. Further, the magnitude of intermediate expressions grows quickly, and the memory requirements are high. The algorithm in [MC93] performs all computations over finite fields (all numbers are computed modulo some prime number), and uses a probabilistic incremental algorithm based on the Chinese Remainder Theorem to recover actual coefficients.

A practical implementation of the Sylvester resultant introduces extraneous factors in the resultant that must be removed. For the special form of f_3, the extraneous factor introduced by Sylvester resultant is u^{pq}, where p and q are the maximum degrees of s in f_1 and f_2 respectively, and we factor it out immediately.

Given a square-free polynomial $p(x)$ we can construct a Sturm sequence of polynomials $S_i = -remainder(S_{i-2}(x), S_{i-1}(x))$, where $S_1(x) = p(x)$ and $S_2(x) = p^{'}(x)$. Treating the volume function V as a univariate polynomial in u, we construct its Sturm sequence $S_i(u, s, t)$. The Sturm sequence is specialized at $u = 0$ to give a sequence of bivariate polynomials $M(s, t)$.

Definition 8 *Given a sequence of polynomials $M(s,t)$ of length n, the* $\mathbf{V}$ *operator at (a_1, a_2) ($\mathbf{V}(M(a_1, a_2))$) gives the number of sign changes between consecutive terms of the sequence evaluated at (a_1, a_2). Correspondingly, the* $\mathbf{P}$ *operator is defined as* $\mathbf{P}(M(a_1, a_2)) = n - 1 - \mathbf{V}(M(a_1, a_2))$.

Given the bivariate sequence $M(s, t)$ and a rational axis aligned rectangle $\Gamma = [a_1, b_1] \times [a_2, b_2]$, the number of real roots of f_1 and f_2 inside Γ is given by

$$\frac{\mathbf{P}(M(b_1, b_2)) + \mathbf{P}(M(a_1, a_2)) - \mathbf{P}(M(b_1, a_2)) - \mathbf{P}(M(a_1, b_2))}{2}.$$

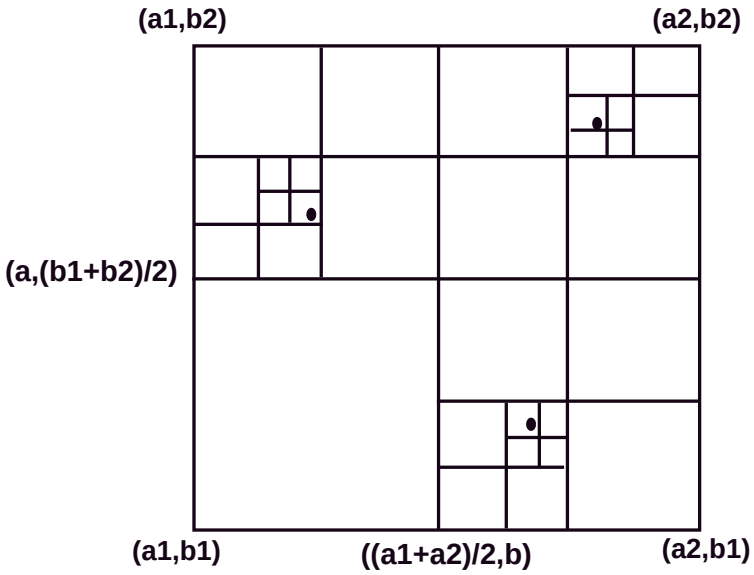

Figure 5.9: Linear convergence of roots

The justification for various steps and extension to arbitrary dimensions can be found in [Mil92].

5.4.2 Converging to the critical points

In order to find all the collinear normal points between the two surfaces, we have to converge to each critical point within a given tolerance. Once that is done, subdividing the domain of the surface at these points ensures that there are no loops within each subdomain. We could then use any marching method to trace the intersection curves.

The algorithm to converge to each critical point within a tolerance is fairly simple. Given an initial domain, we compute the number of common roots within it. If it is zero, we stop. Otherwise, the domain is divided into four parts (by simple bisection), and the computation for number of solutions is performed again. It should be noted that the most expensive step of computing the Sturm sequence is performed only once. Substitution at the various endpoints of the interval is done at each step of the recursion. Once the interval size is within the tolerance, we stop and declare that as a root. It is easy to see that the convergence of this method is linear. Figure 5.9 shows the sequence of subdivisions for a

particular case with three real roots.

5.5 Implementation and Demonstration on Examples

In this section, we show some important steps of our algorithm on a few examples. The implementation of our algorithm was carried out in exact rational arithmetic using LiDIA [BBP95] (a rational number library). By using exact arithmetic, we can assure that the algorithm gives accurate results, however, for the sake of efficiency it is better to implement the algorithm in finite precision.

Example 1: The first example is that of a biquadratic (degree 2×2) Bézier patch sectioned by a plane parallel to the xy-plane (see Figure 5.7). The parametric form of the biquadratic patch is given below.

$$
\begin{aligned}
X(s,t) &= -2 + 4\,s + 2\,t - 4\,s\,t - 2\,t^2 + 4\,s\,t^2 \\
Y(s,t) &= -2 + 2\,s - 2\,s^2 + 4\,t - 4\,s\,t + 4\,s^2\,t \\
Z(s,t) &= 3 - 2\,s + 2\,s^2 - 4\,t - 16\,s\,t + 16\,s^2\,t + 4\,t^2 + 14\,s\,t^2 - 14\,s^2\,t^2
\end{aligned}
$$

Using the above patch equations and computing their partial derivatives, we obtain the following planar vector field.

$$
\begin{aligned}
f_1(s,t) &= -2 + 4\,s - 16\,t + 32\,s\,t + 14\,t^2 - 28\,s\,t^2 \\
f_2(s,t) &= -4 - 16\,s + 16\,s^2 + 8\,t + 28\,s\,t - 28\,s^2\,t
\end{aligned}
$$

After adding the third polynomial, $f_3(s,t) = s\,t + u - t\,x_1 - s\,x_2 + x_1\,x_2$ into the given system of equations and computing successive Sylvester resultants, we obtain the volume function for this bivariate case.

$$
\begin{aligned}
V(u,x_1,x_2) = {} & -104 - 3702\,u + 28244\,u^2 + 80362\,u^3 - 333592\,u^4 - 236670\,u^5 - 528\,x_1 + 184\,u\,x_1 - \\
& 30728\,u^2\,x_1 + 530012\,u^3\,x_1 + 667184\,u^4\,x_1 + 920\,x_1^2 + 36202\,u\,x_1^2 - 77280\,u^2\,x_1^2 - \\
& 530012\,u^3\,x_1^2 + 3680\,x_1^3 - 72772\,u\,x_1^3 + 51520\,u^2\,x_1^3 - 6440\,x_1^4 + 36386\,u\,x_1^4 + \\
& 2576\,x_1^5 - 1469\,x_2 - 21374\,u\,x_2 - 130525\,u^2\,x_2 + 358064\,u^3\,x_2 + 591675\,u^4\,x_2 - \\
& 7458\,x_1\,x_2 - 28704\,u\,x_1\,x_2 - 533968\,u^2\,x_1\,x_2 - 2668736\,u^3\,x_1\,x_2 - 1183350\,u^4\,x_1\,x_2 + \\
& 12995\,x_1^2\,x_2 + 131744\,u\,x_1^2\,x_2 + 2385054\,u^2\,x_1^2\,x_2 + 2668736\,u^3\,x_1^2\,x_2 + 51980\,x_1^3\,x_2 - \\
& 206080\,u\,x_1^3\,x_2 - 1590036\,u^2\,x_1^3\,x_2 - 90965\,x_1^4\,x_2 + 103040\,u\,x_1^4\,x_2 + 36386\,x_1^5\,x_2 -
\end{aligned}
$$

$$2080\ x_2^2 + 84920\ u\ x_2^2 + 166152\ u^2\ x_2^2 - 338100\ u^3\ x_2^2 - 10560\ x_1\ x_2^2 - 218546\ u\ x_1\ x_2^2 +$$
$$1669248\ u^2\ x_1\ x_2^2 + 2366700\ u^3\ x_1\ x_2^2 + 18400\ x_1^2\ x_2^2 - 1371490\ u\ x_1^2\ x_2^2 - 6004656\ u^2\ x_1^2\ x_2^2 -$$
$$2366700\ u^3\ x_1^2\ x_2^2 + 73600\ x_1^3\ x_2^2 + 3180072\ u\ x_1^3\ x_2^2 + 4003104\ u^2\ x_1^3\ x_2^2 - 128800\ x_1^4\ x_2^2 -$$
$$1590036\ u\ x_1^4\ x_2^2 + 51520\ x_1^5\ x_2^2 + 21398\ x_2^3 - 113134\ u\ x_2^3 - 84525\ u^2\ x_2^3 +$$
$$108636\ x_1\ x_2^3 + 404432\ u\ x_1\ x_2^3 - 1014300\ u^2\ x_1\ x_2^3 - 189290\ x_1^2\ x_2^3 + 2264304\ u\ x_1^2\ x_2^3 +$$
$$3550050\ u^2\ x_1^2\ x_2^3 - 757160\ x_1^3\ x_2^3 - 5337472\ u\ x_1^3\ x_2^3 - 2366700\ u^2\ x_1^3\ x_2^3 + 1325030\ x_1^4\ x_2^3 +$$
$$2668736\ u\ x_1^4\ x_2^3 - 530012\ x_1^5\ x_2^3 - 26936\ x_2^4 + 48510\ u\ x_2^4 - 136752\ x_1\ x_2^4 -$$
$$169050\ u\ x_1\ x_2^4 + 238280\ x_1^2\ x_2^4 - 1014300\ u\ x_1^2\ x_2^4 + 953120\ x_1^3\ x_2^4 + 2366700\ u\ x_1^3\ x_2^4 -$$
$$1667960\ x_1^4\ x_2^4 - 1183350\ u\ x_1^4\ x_2^4 + 667184\ x_1^5\ x_2^4 + 9555\ x_2^5 + 48510\ x_1\ x_2^5 -$$
$$84525\ x_1^2\ x_2^5 - 338100\ x_1^3\ x_2^5 + 591675\ x_1^4\ x_2^5 - 236670\ x_1^5\ x_2^5$$

We computed the Sturm sequence of this volume function, and isolated the roots of the original equation in the interval $[0,1] \times [0,1]$ to within a precision of $\frac{1}{100}$. There was a single root as expected. The domain value of the collinear normal was

$$(s,t) = \left(\left[\frac{39038}{78125}, \frac{7894}{15625} \right], \left[\frac{8326}{15625}, \frac{42278}{78125} \right] \right)$$

If we subdivide the original patch at this point, we are guaranteed to have no loop in the resulting intersection.

Example 2: This is a slightly complicated example with multiple collinear normals (see Figure 5.10). Here a bicubic (degree 3×3) parametric surface is cut by a plane parallel to the xy-plane. The coordinate equations of the surface are

$$X(s,t) = -2 + 3\ s + s^3$$
$$Y(s,t) = -2 + 3\ t + t^3$$
$$Z(s,t) = 2 + 3\ s - 6\ s^2 + 3\ s^3 - 72\ s\ t + 189\ s^2\ t - 117\ s^3\ t + 171\ s\ t^2 -$$
$$486\ s^2\ t^2 + 315\ s^3\ t^2 - 99\ s\ t^3 + 294\ s^2\ t^3 - 195\ s^3\ t^3$$

Corresponding to these equations, the vector field is

$$f_1(s,t) = 1 - 4\ s + 3\ s^2 - 24\ t + 126\ s\ t - 117\ s^2\ t + 57\ t^2 - 324\ s\ t^2 + 315\ s^2\ t^2 - 33\ t^3 + 196\ s\ t^3 - 195\ s^2\ t^3$$
$$f_2(s,t) = -8\ s + 21\ s^2 - 13\ s^3 + 38\ s\ t - 108\ s^2\ t + 70\ s^3\ t - 33\ s\ t^2 + 98\ s^2\ t^2 - 65\ s^3\ t^2$$

The volume function for this vector field corresponds to a polynomial which is of degree 13 in u, x_1 and x_2 (the number of terms in this polynomial is too large to list it here). Computation the Sturm sequence and isolation within $\frac{1}{100}$ of precision yielded four roots. They are

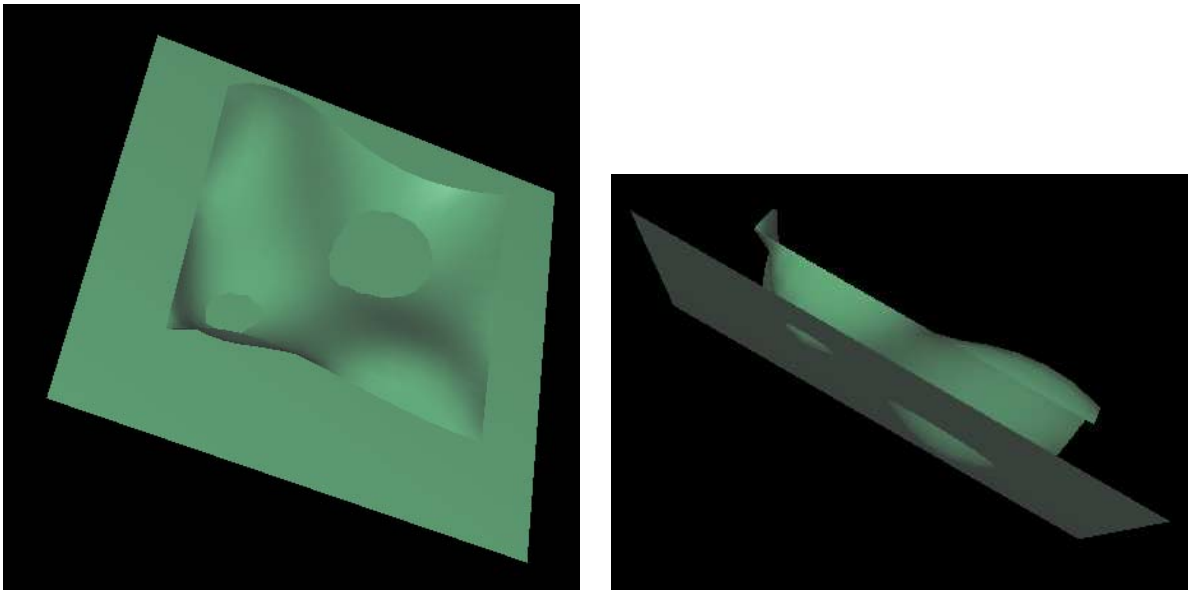

Figure 5.10: Planar section of a bicubic surface

$$(s,t) = \left(\left[\frac{2499}{15625}, \frac{528}{3125}\right], \left[\frac{71874}{78125}, \frac{14484}{15625}\right]\right), \left(\left[\frac{3187}{15625}, \frac{663}{3125}\right], \left[\frac{2453}{15625}, \frac{2547}{15625}\right]\right),$$
$$\left(\left[\frac{1962}{3125}, \frac{9861}{15625}\right], \left[\frac{55937}{78125}, \frac{11281}{15625}\right]\right), \left(\left[\frac{499}{625}, \frac{2518}{3125}\right], \left[\frac{2078}{15625}, \frac{2204}{15625}\right]\right)$$

Even though there are only two loops in the cross section, it is clear that there are other points where the normals of the two surfaces actually match.

Once the critical points have been computed, we subdivide the surfaces along these points and compute the intersection between each of the subdivided surfaces. This is simple because we know that the subdivided surface pairs cannot intersect in a loop. Putting the individual curves together gives us the complete intersection curve.

The algorithms presented in this chapter provide an effective way to detect loops in algebraic curves. The surface intersection algorithm described in the previous chapter employs these techniques with encouraging results. We now proceed to describe the main problem of computing B-reps of Boolean combinations of solids.

Chapter 6

Boundary Computation of Sculptured CSG Solids

In this chapter, we present an efficient algorithm for representation and computation of surface boundaries of CSG solids. Every CSG object is built from a set of primitive objects which are of a simpler structure. The set of primitives include polyhedra, quadrics, generalized prisms and pyramids, tori, surfaces of revolution and sculptured solids (whose boundaries can be represented as NURBS surfaces). The techniques presented can also be generalized to all algebraic surfaces. An example of a CSG tree is shown in Figure 1.3. Boolean combinations of such solids are used in most CAD and modeling systems. For example, the Bradley fighting vehicle (shown in Figure 6.1) has been modeled using boolean operations. The model consists of more than 8500 solids, each designed using 5 to 8 boolean operations.

6.1 Representation of Solids

In this section, we describe our representation for a solid. Our algorithms assume that all B-rep solids are specified in this format. Every solid is represented as a set of *trimmed* parametric surface (tensor-product Bézier) patches (for definition, see chapter 2) which define the solid boundary.

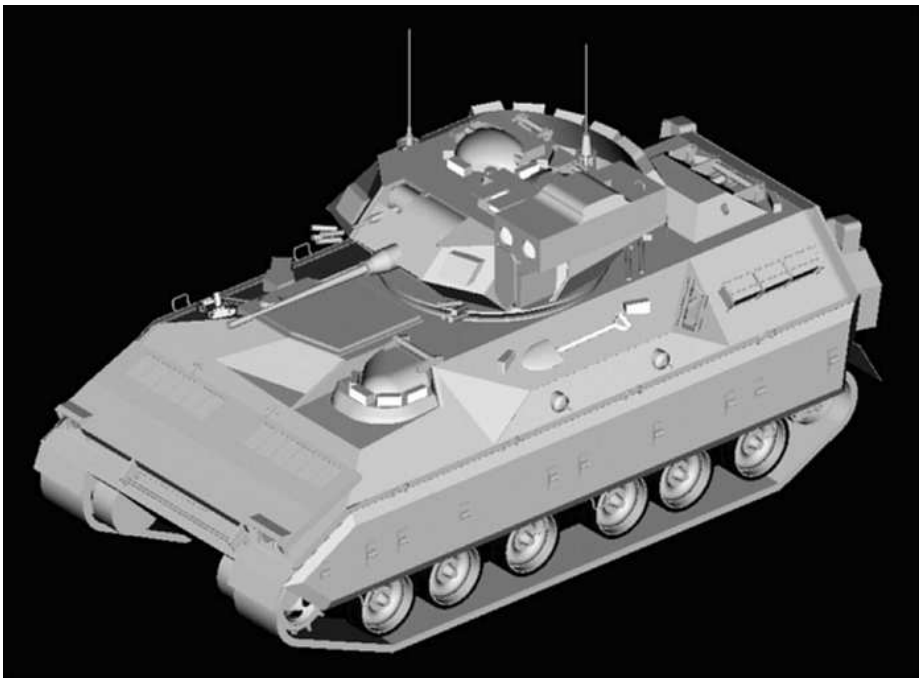

Figure 6.1: Exterior of a Bradley fighting vehicle

Topological information of the solid is maintained in terms of an adjacency graph. It is similar to the winged-edge data structure [Hof89, MT83]. To start with, we assume that each of the input objects has *manifold* boundaries, and the Boolean operation is *regularized* [Man88]. While it is possible to generate non-manifold objects from regularized Booleans on manifold solids, we assume for the sake of simplicity that such cases do not occur. Given this assumption, it has been shown that an unambiguous topological representation is possible for a solid [Hof89].

A trimmed patch consists of a sequence of curves defined in the domain of the patch such that they form a closed curve ($\mathbf{c_i}$'s in Figure 6.2). In the figure, the $\mathbf{c_i}$ refer to the algebraic curve segments forming the trimming boundary. The portion of the patch that lies in the interior of this closed curve is retained (the trimming rule is described in chapter 2). Most of these trimming curves correspond to intersection curves between two surfaces. Therefore, these curves are typically algebraic curves that do not admit a rational parametrization [AB88a]. We represent these curve segments ($\mathbf{c_i}$) by their algebraic equation (for accuracy), and a piecewise linear approximation (for efficient computation) and the two

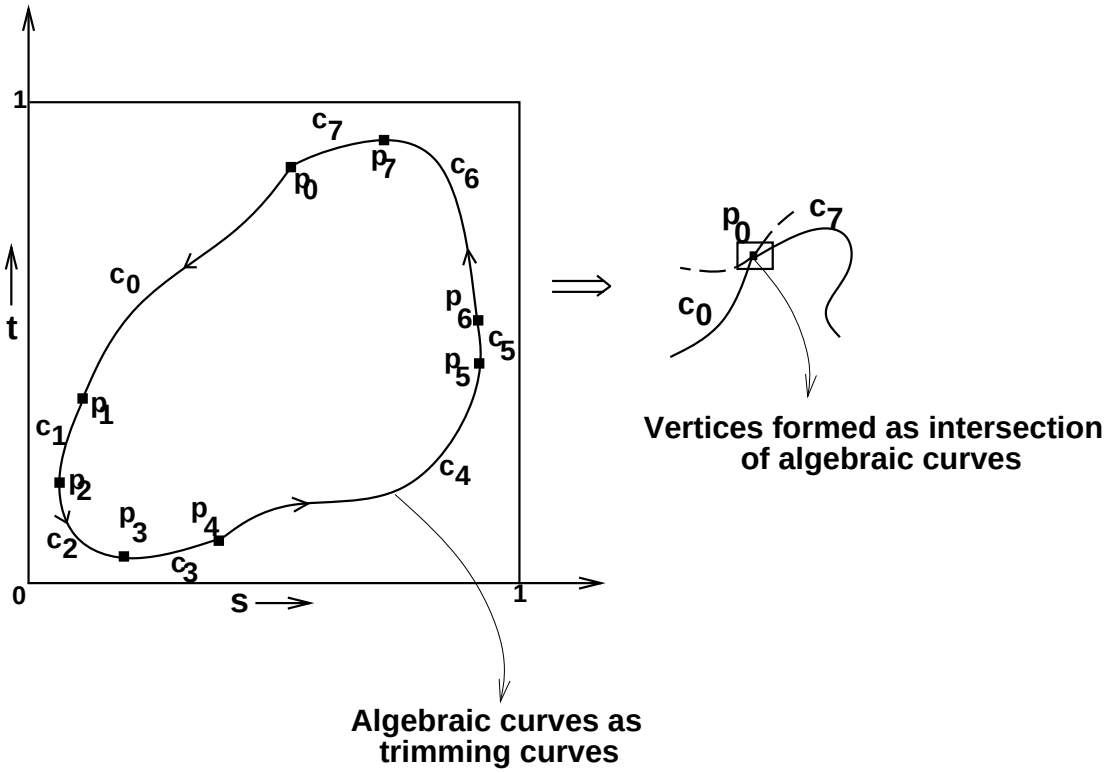

Figure 6.2: Representation of a trimmed patch as algebraic curve segments

endpoints ($\mathbf{p_i}$ and $\mathbf{p_{i+1}}$).

This representation of a solid lends itself to a description in terms of *faces, edges, and vertices* analogous to the polyhedral case. Each *face* is a trimmed patch. Each of the trimming curves form an *edge*, and are formed as an intersection of two surfaces (faces). Finally, endpoints of edges form the *vertices*. They can be represented as an intersection of three surfaces. Figure 6.3 shows an example solid and the face connectivity structure that we maintain. We also maintain the two faces that are adjacent to each edge, and an anticlockwise order of faces around each vertex.

6.2 Set Operations between Solids

In this section, we shall describe the algorithm to compute the solid which is the result of some set operation between two given solids, $solid_1$ and $solid_2$. We shall denote

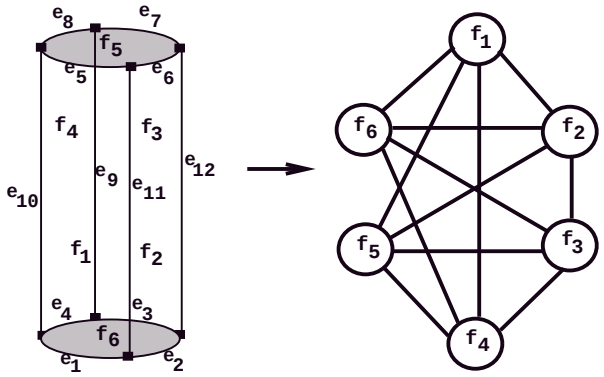

Figure 6.3: A cylinder and its face connectivity structure

the resulting solid as $solid_{12}$. Let the number of patches in $solid_1$ be m and those in $solid_2$ be n and let the maximum degree of each patch be $d_s \times d_t$ (maximal degree monomial is of the form $s^{d_s} t^{d_t}$, where s and t are the parameters defining the surface).

The first step in computing $solid_{12}$ is to find the curve of intersection between the two solids. Since each solid is composed of a set of trimmed patches, we have to compute the component of the curve inside each patch. However, not all the mn pairs would intersect typically. We prune out most of the non-intersecting pairs based on a two-step process. Initially, we compute a $3D$ axis-aligned bounding box for each patch. Since tensor-product Bézier patches have the *convex hull property* [Far93], the bounding box and convex hull of the control points encloses the entire surface. Therefore, if a pair of bounding boxes do not intersect, the corresponding patches are also non-intersecting. Further, determining the bounding box of the surface just requires computation of the minimum and maximum extents of all the control points along the three coordinate axes. This step can be done in time $O(d_s\ d_t)$. Since each of the surface patches comprising the solids are trimmed, we also compute the tightest fitting axis-aligned rectangle in the domain that encloses the trimming region. This ensures a tighter bounding box for the actual surface.

Bounding box tests: Two bounding boxes overlap if and only if their interval extents projected along each of the coordinate axes overlap. This condition gives us a naive method to perform bounding box overlap tests. For each of the $O(mn)$ pairs performing the above mentioned test gives us the answer. While this algorithm is time optimal if

there are $O(mn)$ overlapping bounding boxes, this algorithm is quite inefficient if there are very few actually intersecting boxes. Obtaining an output sensitive algorithm that performs asymptotically better on average has been a well studied problem in computational geometry. Edelsbrunner [Ede83] obtained an $O(n \log n + k)$ for the three dimensional rectangle intersection problem. We, however, use an algorithm which runs in $O(n \log^2 n + k)$ time using nested segment trees.

Initially, we sort the Z-extents of all the bounding boxes for a *plane sweep* algorithm. The data structures we maintain at each step of the plane sweep are two nested segment trees (one for each solid) - the first level is a segment tree where the X-intervals are stored. At each node of this tree, we maintain the Y-intervals of all the bounding boxes active in the form of a secondary segment tree. Every insertion, deletion and search step requires $O(\log^2 n)$ time. Each step of the algorithm corresponds to the start or end of a bounding box belonging to one of the patches in $solid_1$ or $solid_2$. If it is a starting case, we simply augment the appropriate data structure by inserting the corresponding X and Y intervals. However, if the sweeping plane is at the end of a Z-interval from a patch in $solid_1$ ($solid_2$), we delete the corresponding X and Y intervals from the segment tree of the first (second) solid. In addition, we report all the bounding boxes intersecting this box by performing a query operation on the segment tree of the second (first) solid. The query operation takes $O(\log^2 n)$ time because the Y-interval overlap query in the secondary structures can be performed from as many as $O(\log n)$ nodes. At the end of the plane sweep, all the bounding box overlaps are reported (there may be repetitions). This algorithm is space inefficient (takes $O(n^2 \log^2 n)$ space). We chose this algorithm, however, for ease of implementation and the nature of our application.

Convex hull tests: The next stage of pruning uses convex hulls of the control polytope to eliminate non-intersecting patches. This test is performed only for those pairs of patches whose bounding boxes overlap. The test can be formulated as a linear programming problem as follows. Two patches do not intersect if there exists a separating plane between them. Therefore, if there exists a separating plane between the two sets of control points, then the patches are non-intersecting.

The problem is set up as follows. Let the control points for the first patch be

$(X_{1i}, Y_{1i}, Z_{1i}, W_{1i})$ and those for the second patch be $(X_{2i}, Y_{2i}, Z_{2i}, W_{2i})$ for $i = 1, 2, \ldots, (d_s+1)(d_t+1)$. The equation of a plane in projective coordinates is $ax + by + cz + dw = 0$. If this plane is a separating plane for the two sets of control points, then the substitution of the control points into the plane equation gives different signs for each patch. Thus,

$$\begin{aligned}
aX_{11} + bY_{11} + cZ_{11} + dW_{11} &< 0 \\
aX_{12} + bY_{12} + cZ_{12} + dW_{12} &< 0 \\
&\vdots \\
aX_{21} + bY_{21} + cZ_{21} + dW_{21} &> 0 \\
aX_{22} + bY_{22} + cZ_{22} + dW_{22} &> 0 \\
&\vdots
\end{aligned}$$

We are looking for at least one feasible set of solutions for a, b, c and d that makes these constraints true. The dimension of this problem is four. The number of control points for each patch is $(d_s+1)(d_t+1)$. Therefore, the number of constraints in the linear programming problem is $2(d_s+1)(d_t+1)$. Seidel's algorithm for linear programming runs in time linearly proportional to the number of constraints for fixed dimensions. The running time for the entire process, thus, is $O(k\ d_s\ d_t)$. Typically, d_s and d_t are much smaller than k (number of pairs with overlapping bounding boxes). Therefore, the running time is $O(k)$. By applying these two methods on the two solids, we are left with few pairs of patches that are most likely to intersect. We use Mike Hohmeyer's implementation of the linear programming algorithm developed by Seidel [Sei90b].

6.2.1 Intersection Curve between Trimmed Patches

In order to compute the intersection curve between the two solids, we compute a series of intersections between pairs of trimmed patches. Since a trimmed patch is a strict subset of the original patch, then so is any intersection curve that lies inside this patch. We use the algorithm described in chapter 4 to compute the complete intersection curve of the two patches (ignoring the trimming curves). The intersection curve so obtained from the

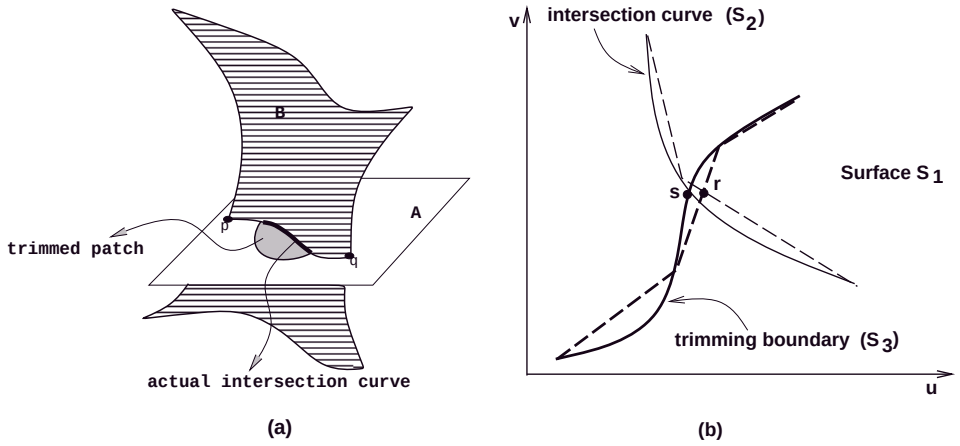

Figure 6.4: (a) Intersection of trimmed surfaces (b) Computing curve intersections with trimming boundary

algorithm is represented as a piecewise linear chain in parameter space. We also maintain an accurate representation of the intersection curve in the domain of each patch as a bivariate matrix polynomial Figure 6.4(a) shows the surface B intersecting with a trimmed patch A. A planar surface is trimmed so that only the portion inside the circular region (not shown completely) belongs to A. The actual intersection curve is highlighted in Figure 6.4(a). However, the curve from p to q is the intersection curve of B with the complete planar patch.

Given the complete intersection curve pq (represented as a different polygonal chain in each of the two patches), we have to compute the intersection curve of the trimmed patches. This curve is determined by finding portions of pq that lie inside the trimmed region of both the patches. This problem can be solved by accurately finding the intersection points between the intersection curve and the trimming boundary. The trimmed region is represented as a simple polygon in parameter space as mentioned earlier. This reduces the problem to finding the portions of a polygonal chain (represented differently in the other surface domain, but corresponds to the same space curve) that lies inside two simple polygons simultaneously. If the length of the chain is m, and the sizes of the two polygons are n_1 and n_2, then this problem can be solved in time $O(m(\log n_1 + \log n_2))$ using point location queries each of which take $O(\log n)$ time [Sei91]. The intersection points obtained

by this process is only an approximation to the true intersection point. Figure 6.4(b) illustrates this point on a sample surface S_1 whose trimming boundary is the result of a previous intersection computation with surface S_3. The intersection curve is generated with surface S_2. The intersection point computed using the piecewise linear curves is r, while the actual intersection point between the intersection curve and trimming boundary is s. The image of point s in $\mathcal{R}^3$ is the point of intersection between the three surfaces S_1, S_2 and S_3. In order to refine r, we use the patch equations of the three surfaces involved. r is a point in the domain of S_1. Since the piecewise linear curves have corresponding representations in the domains of S_2 and S_3 and r lies on both the curves, it is easy to find the preimage of r in the domains of S_2 and S_3. Let us call them $\dot{r}$ and $\ddot{r}$ respectively. Using these points as the initial guess, we perform few iterative steps of local minimization on an energy function $E(r, \dot{r}, \ddot{r})$ that is defined as

$$E(r, \dot{r}, \ddot{r}) = \| S_1(r) - S_2(\dot{r}) \|^2 + \| S_1(r) - S_3(\ddot{r}) \|^2$$

The number of iterations taken by the minimization routine is directly dependent on the accuracy of the piecewise linear curves. We use Powell's method [PFTV90] as our local minimization algorithm. In our experience, the minimization step converges within 3-5 iterations for a tolerance of 10^{-8}.

By applying this algorithm on all pairs of patches, we obtain a set of curves in the domain of every patch. Since each solid is closed (and compact), the intersection curve between two such solids must form a collection of closed curves in space. This implies that locally in every patch, the set of curves must partition the domain of the patch. Therefore, we merge two curves that share an endpoint in the interior of the patch. Since we are working in double-precision arithmetic, the endpoints of each curve has to be compared with a tolerance. Determining a consistent tolerance is extremely difficult because propagation of numerical errors in such complicated algorithms is not clearly understood. However, we avoid floating point comparisons if we observe that two curves can be merged only if the two surfaces that generated these curves are adjacent in the other solid.

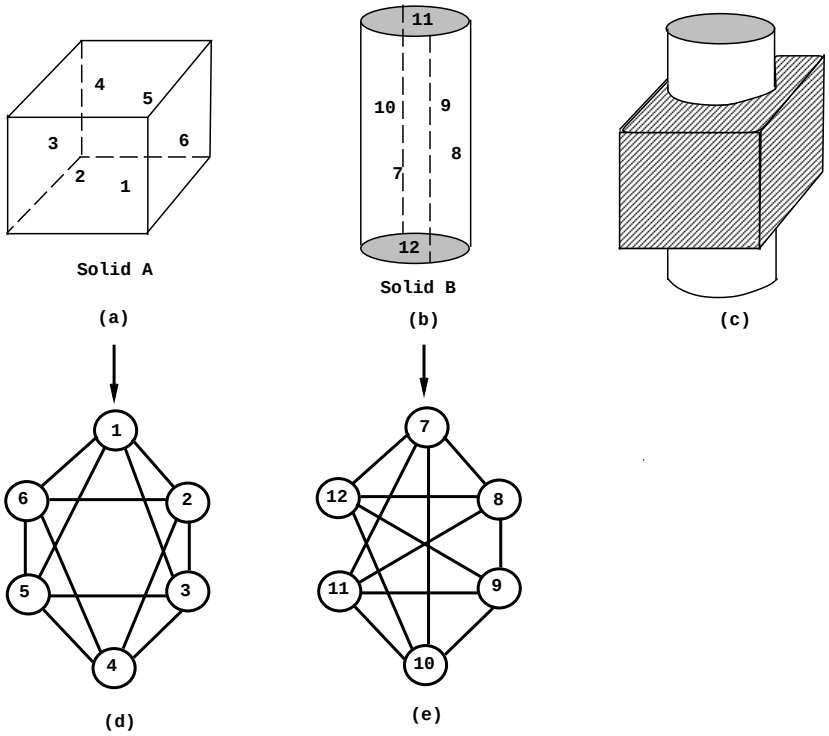

Figure 6.5: (a) A cube (b) A cylinder (c) Cylinder used to drill a hole right through the cube (d),(e) Connectivity graphs of the cube and cylinder

6.2.2 Partitioning Trimming Boundaries

Once all the intersection curves are merged within each patch, they will partition the trimmed domain (if the assumptions that the individual solid boundaries are closed and compact are maintained). Figure 6.6(a) shows intersection curves inside a trimmed domain. $\mathbf{c_i}$'s (with endpoints $\mathbf{p_i}$ and $\mathbf{p_{i+1}}$) form the trimmed boundary of the patch. $\mathbf{i_0}$, $\mathbf{i_1}$, and $\mathbf{i_2}$ are the intersection curves computed with various patches of the other solid. $\mathbf{t_0}$ is a turning point on the curve $\mathbf{i_2}$. $\mathbf{q_i}$s are points on the intersection curve where the curve intersects the trimmed boundary. Given this information, Figure 6.6(b) shows the actual partitions ($\mathbf{R_i}$s). To compute the explicit B-rep of the resulting solid, each of these partitions are generated. We now present an algorithm that computes these partitions provided the intersection curves have no singularity in the trimmed domain.

The main idea in this algorithm is the fact that since the intersection curve segments (I_0 and I_1 in Figure 6.6(c)) do not cross each other, each resulting partition starts at

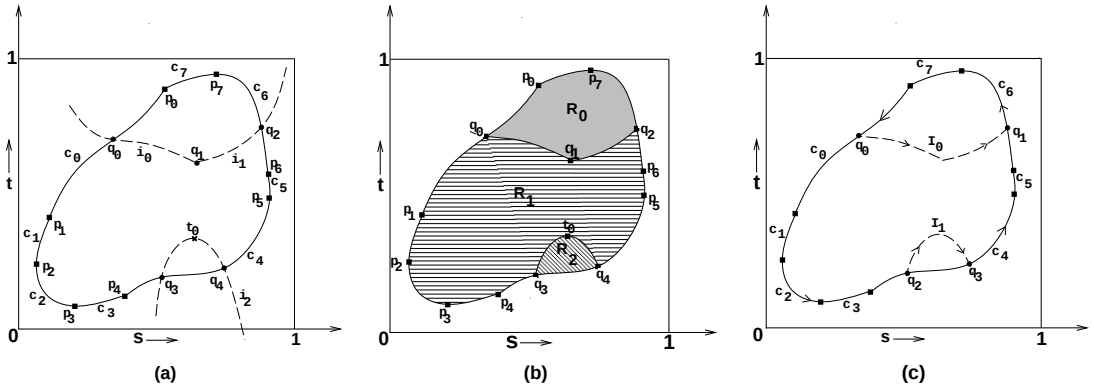

Figure 6.6: (a) Intersection curves inside trimmed domain (b) Partitions introduced by intersection curves (c) Partitioning a trimmed patch with chains of intersection curves

one endpoint of a curve segment, and ends at the other endpoint of the same curve segment. We shall assume that the trimming curves and the intersection curves are given in a specific order. We number the endpoints of the intersection curve segments such that $\mathbf{q_{2j}}$ and $\mathbf{q_{2j+1}}$ belong to $\mathbf{I_j}$. The algorithm works in three steps.

- Each endpoint of a curve segment (for example, $\mathbf{q_0}$ of $\mathbf{I_0}$) lies on a unique curve (except when it coincides with one of the curve endpoints of the boundary) of the trimming boundary. In fact, points like $\mathbf{q_0}$ are determined as the intersection of $\mathbf{I_0}$ with $\mathbf{c_0}$. Each boundary curve $\mathbf{c_i}$ is then partitioned into multiple segments depending on the number $\mathbf{q_j}$s lying on it.

- This is followed by a traversal of the trimming boundary in a consistent order by maintaining a stack. Two types of elements are pushed in the stack - curve segments, and curve endpoints. Initially, we keep pushing in the boundary curve segments until we reach a vertex like $\mathbf{q_0}$. Let the vertex number be k. If the topmost curve endpoint type of the stack (say, $\mathbf{l}$) has a number $(k+1)$ or $(k-1)$, then a partition has to be read out. Otherwise, vertex k is pushed into the stack followed by all the curves that comprise $I_{\lfloor k/2 \rfloor}$. If a decision to read out a region has been reached, all the curve segments until vertex $\mathbf{l}$ are popped. Curves comprising $I_{\lfloor k/2 \rfloor}$ are pushed again because they are required by the next region too. The order in which these curve

segments are pushed into the stack has to be monitored carefully so that a region which is read out is oriented consistently.

- Till now, we have considered only intersection curve segments whose endpoints lie on the trimming boundary. However, there may be loops that lie completely inside the boundary. Any loop is present (if at all) inside one of the obtained partitions. Each of the loops (starting from the innermost if the loops are nested) themselves form a partition. The remaining part of the region (it has boundaries with multiple components) is broken into simple regions by introducing a simple cut from the loop to the boundary of the partition or the next loop.

This completes the algorithm to compute the partitions introduced by intersection curves. A feature of this algorithm is that the adjacency structure between the various partitions (which is necessary to avoid redundant, expensive ray-shooting queries during component classification) are obtained by the order in which they are read out.

6.2.3 Updating Topological Information

It is clear that intersection computation introduces new vertices, edges, and faces in the solid. This change needs to be incorporated in our topological structure. Further, information about the adjacency between the various faces significantly reduces the component classification time. At this time, we just concentrate on the face adjacency. Vertex and edge adjacency are updated during final solid generation.

The new graph is a refinement of the original adjacency graph. Since a vertex of the graph corresponds to a face of the solid, each vertex in the original graph is split into a few vertices depending on the partitions obtained due to the intersection curves. We have to determine the adjacency relationship between the newly created vertices. Consider, for example, that vertices $\mathbf{u}$ and $\mathbf{v}$ were adjacent in the original graph. We create a new graph to extract adjacencies between various *orientation invariant* components. Due to the intersection curves, let the vertex $\mathbf{u}$ be split into $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_m}$, and let the vertex $\mathbf{v}$ be split into $\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_n}$. The adjacency between the various $\mathbf{u_i}$'s (similarly $\mathbf{v_i}$'s) has already been determined (during partitioning). These adjacencies (let the corresponding set of edges

be denoted by S) are purposely left out in the new graph. Let e be the edge along which $\mathbf{u}$ and $\mathbf{v}$ were adjacent in the original graph, and let it be divided into k portions during partitioning. Then all the adjacencies between $\mathbf{u_i}$'s and $\mathbf{v_j}$'s can be obtained in $O(k)$ time. The number of connected components in this graph gives the number of solid components introduced by the intersection curves. Let the solid components be named $CC_0, CC_1, \ldots$. Note that each CC_i has a collection of faces. We observe that each of the CC_is satisfies the orientation invariance property that all the patches corresponding to them lie either completely inside or completely outside the other solid (because there is no intersection curve passing through the interior of these components).

To obtain the connectivity between the various CC_is (in a graph Γ), we introduce some notation. Let R be a mapping which takes a vertex in the new graph to the corresponding vertex in the original graph. For example, if $\mathbf{u}$ was split into $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_m}$, then $R(\mathbf{u_i}) = \mathbf{u}$. Two components CC_i and CC_j are connected in the graph Γ if

$$\{\exists \mathbf{u_1} \in CC_i, \exists \mathbf{u_2} \in CC_j | R(\mathbf{u_1}) = R(\mathbf{u_2}) \; and \; (\mathbf{u_1}, \mathbf{u_2}) \in S\}$$

Using this, we obtain the various components and their connectivity structure. Next we resolve each of these components (inside/outside) with respect to the other solid.

Figure 6.5 shows two solids and their connectivity graphs that enter into a set operation (difference). The cylinder is represented by four Bézier patches along the side and two planar trimmed surfaces for the top and bottom. Given these two solids, their connectivity graphs and all the intersection curves, we obtain the topological information of the final solid.

6.2.4 Component Classification

Component classification involves determining whether a given component (obtained by the graph algorithm) of one solid lies inside or outside the other solid. In most polyhedral modelers, component classification is carried out locally [Hof89] by looking at the relative orientation (left/right) of the intersection curve. When dealing with sculptured surfaces, though, the same technique cannot be used, primarily due to the complexity of

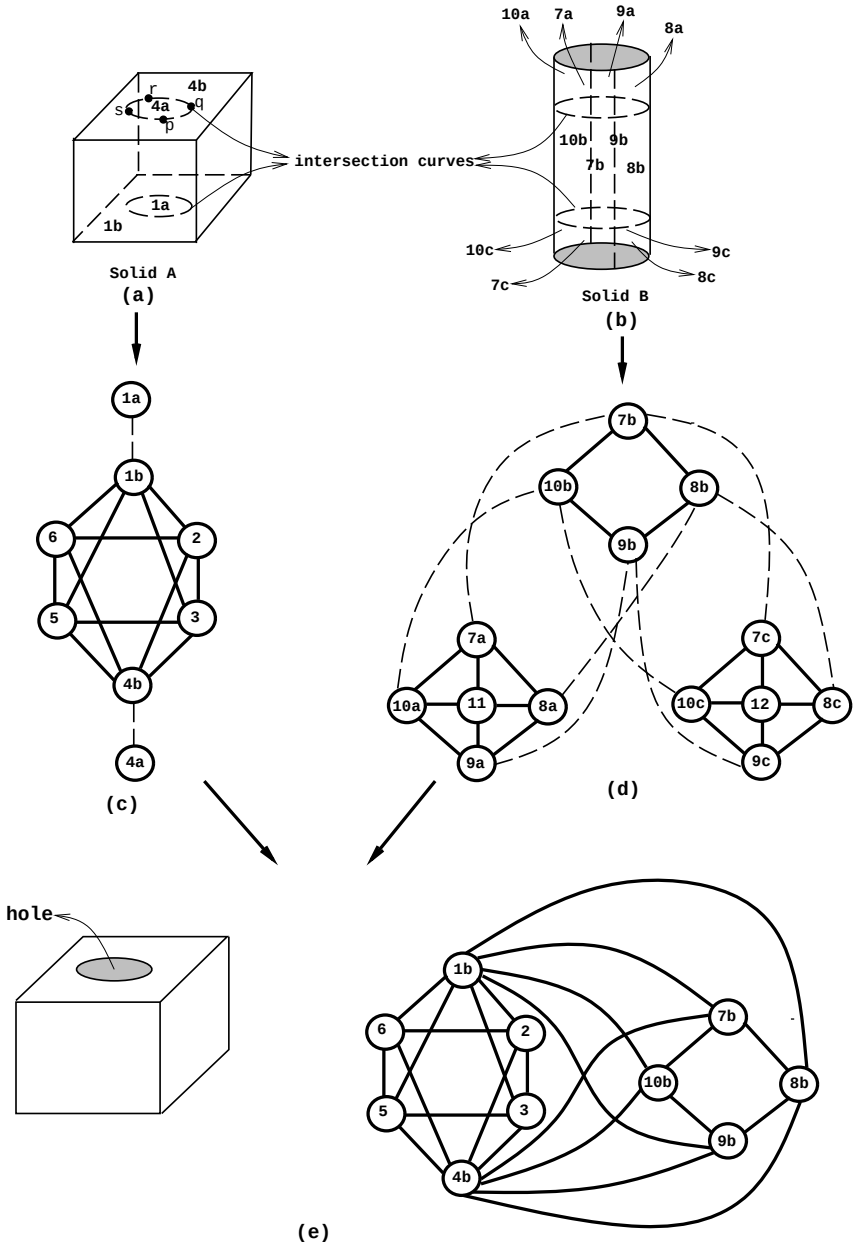

Figure 6.7: (a),(b) Intersection curves on cubes and cylinders (c),(d) Updated connecivity graphs based on partitions (e) Connectivity graph of final solid

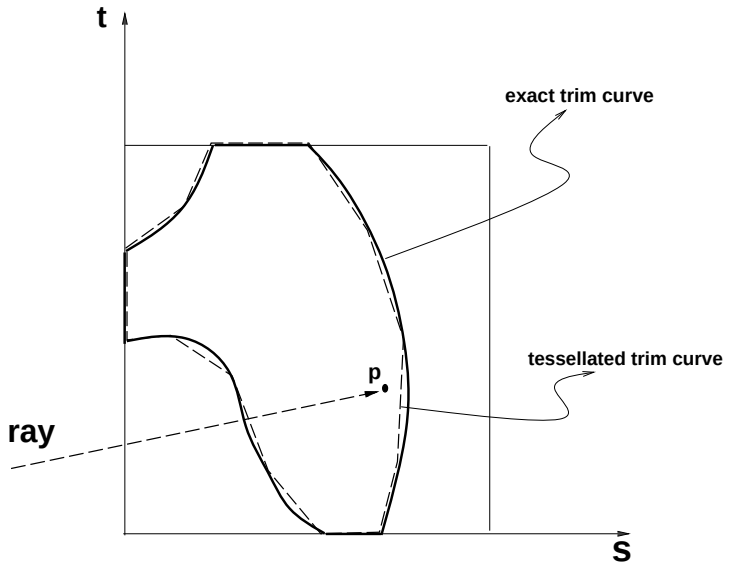

Figure 6.8: 2D Classification

the intersection curve topology. The most general method used instead is based on ray-shooting. Ray-shooting is done by firing a semi-infinite ray in an arbitrary direction from a representative point of the component and checking for intersections with the other solid. For closed solids, if the number of intersections is even, the point (and hence, the entire component) lies outside the solid; if it is odd, it lies inside.

There are two steps involved in our algorithm to perform component classification. The first step involves getting a point that is part of the component. This is accomplished by 2D ray-shooting. We initially choose some point $p = (s, t)$ on the trimming boundary such that t lies between the lower and upper extents of the trimming boundary (any appropriate vertex of the polygon would suffice). A horizontal ray through p (in both directions) is intersected with the boundary. Computing all the intersections is very easy because only those segments whose endpoints (a, b) and (c, d) satisfy $(t - b)\ (d - t) > 0$ will intersect the ray. Further, the intersections must be even in number and are of the form $(s_1, t), (s_2, t), \ldots, (s_{2n}, t)$. Choosing the midpoint of s_{2i-1} and s_{2i} for $i = 1, 2, \ldots, n$ gives one point inside the trimming boundary. Let this point be called $\mathbf{q}$. Another method is to maintain the triangulation of the trimming polygon and choosing the centroid of one

of the triangles. When points have to be repeatedly generated, it is beneficial to preserve the triangulation. We perform triangulation using an implementation of Seidel's algorithm [Sei90b].

The second step involves actual ray-shooting in 3-space. The algebraic pruning algorithm described in chapter 3 gives the number of intersections of a ray with an untrimmed patch. Since each solid is made up of trimmed patches, it is necessary to test if the intersected point lies inside the trimmed region or not. This step is called *2D classification*. Essentially 2D classification can be done by shooting a ray from the intersected point in the plane of the trimming polygon. However, we use trapezoidation of the trimming polygon (using Seidel's algorithm) to perform logarithmic time point location queries.

The accuracy of this result depends on the magnitude of errors introduced by approximating the high degree trimming curve. For example, consider the point p near the boundary of the trimming polygon in Figure 6.8. It is unclear if the result of the point location query for a point very close to the boundary of the polygon is, in fact, correct. We improve the accuracy of the classification test by using the analytic representation of the trimming curve (bivariate matrix polynomial). Since the algebraic curve is a zero set of a polynomial, there is a sign change on either side of the curve in the local neighborhood of the boundary. The sign of the polynomial with the point p substituted for the variables gives the classification of the point. Since the curve is represented as the determinant of a matrix polynomial, we have to evaluate the sign of this determinant. We use *singular value decomposition* (SVD) to accomplish this task.

Given a numerical square matrix $\mathbf{A}$, SVD decomposes it into the form

$$\mathbf{A} = \mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V^T},$$

where $\mathbf{U}$ and $\mathbf{V^T}$ are orthonormal matrices, and $\mathbf{\Sigma}$ is a diagonal matrix whose entries are all positive. This implies that the sign of the determinant of $\mathbf{A}$ is the same as the product of the signs of the two orthonormal matrices (determinant is +1 or -1). We can safely perform Gaussian elimination to determine the sign of these determinants. SVD is a very stable numerical algorithm, and hence its results are usually reliable.

From the connectivity information among various components and the classifica-

tion of one of them, we can classify all the other components. This is because two adjacent components must have opposite classifications. Ray shooting is a fairly expensive operation, and its complexity depends on the degree of the surface patch and the trimming curve. The method described above requires only two (one for each solid) ray shooting operations per CSG operation.

6.2.5 Final B-rep Generation

The trimmed patches that make up the final solid are determined by the Boolean operation performed. Given two solids $solid_1$ and $solid_2$, we decide on the final B-rep depending on the Boolean operation.

- *Union:* All components of $solid_1$ that lie **outside** $solid_2$, and vice-versa are retained.
- *Intersection:* All components of $solid_1$ that lie **inside** $solid_2$, and vice-versa are retained.
- *Difference:* All components of $solid_1$ that lie **outside** $solid_2$, and all components of $solid_2$ that lie **inside** $solid_1$ are retained.

We also update the topology information. Each connected component that is retained in the final solid has some graph vertices (faces of the solid) whose complete adjacency is not determined. These missing adjacencies correspond to edges which are formed by intersection curves. This edge connects two vertices from different solids. Since an intersection curve is determined by a unique pair of surfaces, the two endpoints of this edge is also unique. For every intersection curve in a solid, we maintain the corresponding patch number of the other solid, and use it to complete the adjacency information. From this graph, the entire topological information is easily computable.

6.3 Degeneracies

A number of degenerate cases can arise when dealing with curved surfaces. Some of these degeneracies are of the same general type as is found in a polyhedral modeler, while some others arise only with curved surface modelers. These include

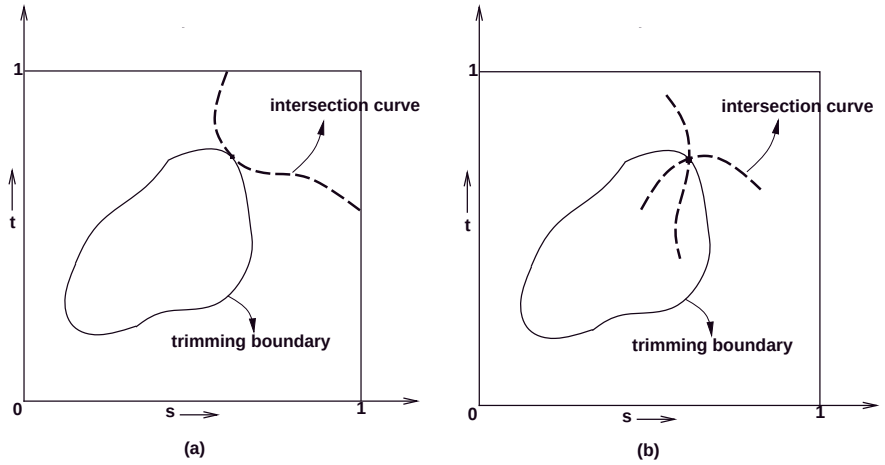

Figure 6.9: (a) Surface-edge contact degeneracy (b) Four surfaces meeting at a point

- **Two surfaces meeting at a point:** This case is particular only to curved surfaces. Since the surfaces meet at a point which lies in the interior of their respective domains, their normals are coincident. This corresponds to a singularity. We determine this by minimizing the energy function (equation (4.4) in chapter 4) used to determine singularities.

- **Two surfaces tangentially intersecting at a curve:** This is a degenerate case when the surfaces are tangent to each other along that curve. This case also occurs only with curved surfaces. We will be able to detect this when we generate the adjacency graph by finding that two adjacent components actually have the same orientation with respect to the other solid. Another scenario when this case occurs is if the intersection curves do not form a closed loop in space.

- **Two surfaces overlapping:** This corresponds to a face-face overlap in the polyhedral domain. If two surfaces are overlapping, their intersection set is two-dimensional. Essentially, our bivariate matrix polynomial representing the intersection curve is singular for all values in the domain. We perform this test by sampling the domain and determining the ranks of the resulting numeric matrices using SVD.

- **A surface just touching an edge:** This is an edge-face contact in the polyhedral domain, and can happen when three surfaces meet in a curve. In our representation,

this will appear as an intersection curve which is tangent to a trimming curve (see Fig. 6.9(a)). Such a case can be automatically eliminated if we check *each* component of the intersection curve to see whether it is in the trimmed region. This does not allow us to use the speed-up of propagating the information about one component of the intersection curve to all other components of that curve.

- **Four surfaces meeting at a point:** This, is the foundation for several types of degeneracies and will be discussed next.

Examples of four surfaces meeting at a point include when a vertex of one solid lies on the surface of another solid, or when the edges of two solids meet. Obviously, the vertex can be thought of as the intersection of three surfaces, and the edges can be thought of as the intersection of two surfaces, thus the cases mentioned would involve the intersection of four surfaces.

Even more degenerate cases, such as two vertices meeting, or a vertex lying on an edge, are possible, but these can be viewed as 5 or 6 surfaces meeting at a point - i.e. at least four surfaces are still meeting at a point.

These cases will manifest themselves in our modeler as three (or more) curves meeting at a common point in the domain of some patch (see Fig. 6.9(b)). Assume these three curves are $f1$, $f2$, and $f3$. We can find out whether this case has occurred by checking equality of the intersection of $f1$ and $f2$ with the intersection of $f1$ and $f3$ (or $f2$ and $f3$).

Degeneracies in the polyhedral case can generally be classified into the category of four planes meeting at a point. It has been shown [For95] that a simple perturbation scheme applied to a single basic geometric predicate can eliminate these degeneracies. No obvious extension of this method exists in the curved surface domain, though there is hope that some perturbation method can be developed using exact rational arithmetic which would work similarly.

Chapter 7

Implementation and Performance

One of the main contributions of this dissertation includes a complete implementation of all the algorithms presented. The implementation of algebraic pruning, loop detection, surface-surface intersection and boundary evaluation algorithms are parts of the BOOLE solid modeling system. Given a CSG tree whose leaves are chosen from a predefined set of primitive solids, BOOLE generates the surface representation of the boundary of the final solid as a collection of trimmed Bézier patches as well as the topological information in a graph structure. The various modules in our system and their dependency relations are shown in Figure 7.1.

We have implemented our system on single processor architectures like SGI Maximum Impact (with one 250MHz R4400 CPU) and Sun-Solaris, as well as a parallel version of the algorithms on shared memory multiprocessor architectures like SGI Onyx (with up to 6 194MHz R10000 CPUs, 1MByte main memory). Our current sequential implementation can perform one Boolean operation on solids like conicoids (spheres, ellipsoids, tori, cylinders and cones) in about 3-4 seconds, while the parallel version can do the same in one second or less.

Given a CSG tree, our system generates the boundary representation of all the primitives involved in the form of trimmed Bézier patches along with their topology information. For each Boolean operation, the B-reps of the two solids are passed to the solid intersection module. This module is responsible for generating the intersection curve be-

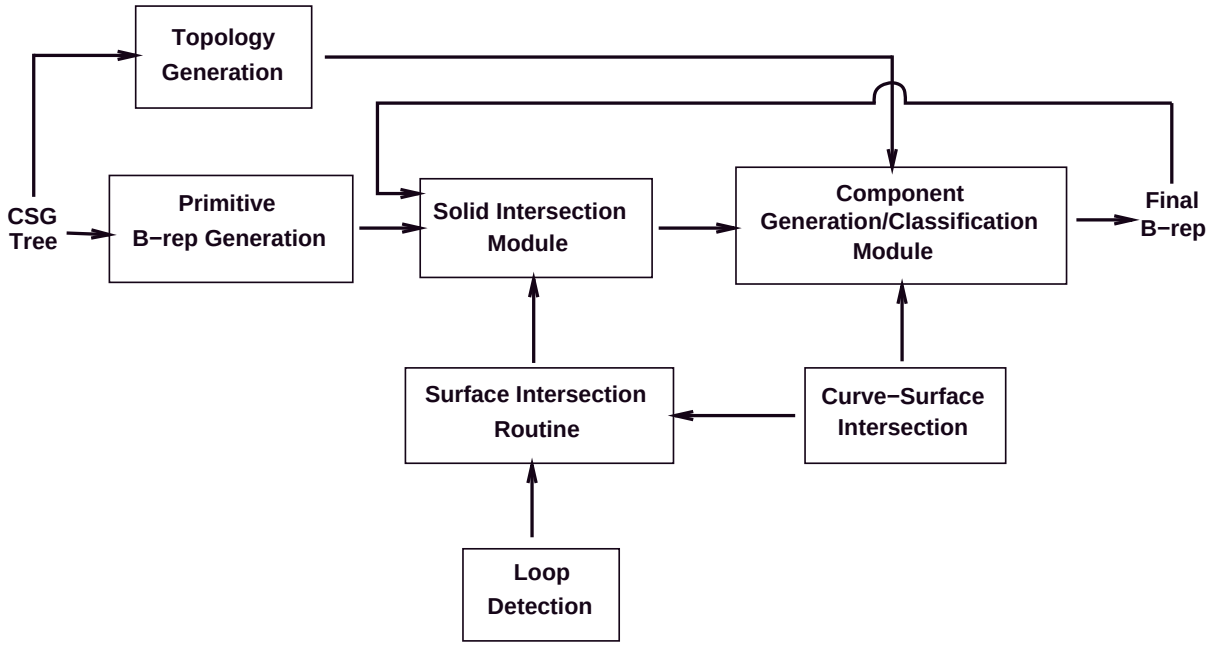

Figure 7.1: Functional modules in the BOOLE system

tween the two solids. The curves are generated in the domain of each patch as well as in 3-space. We maintain the curve in 3-space (space curve) so that we can verify if the intersection curves form a closed loop. This is a checkpointing operation, and if the curve is not closed, we declare an error and try to recompute the curve. The space curve is also used during model visualization. The solid intersection module relies on the surface-surface and curve-surface intersection algorithms to generate the curves. These algorithms are implemented in C and makes use of a number of matrix operations like SVD, matrix eigendecomposition and inverse iterations. These routines are available in public domain in the form of Fortran libraries like EISPACK [GBDM77] and LAPACK [ABB+92]. The main advantage of using these libraries is that they are carefully and efficiently implemented by numerical analysts and well tested on a number of benchmarks. Further, most of the matrix routines also return the condition number of the problem. We use this information to predict the conditioning of our original problem or to detect inaccuracies in our computation.

The intersection curves are fed into the component generation/classification mod-

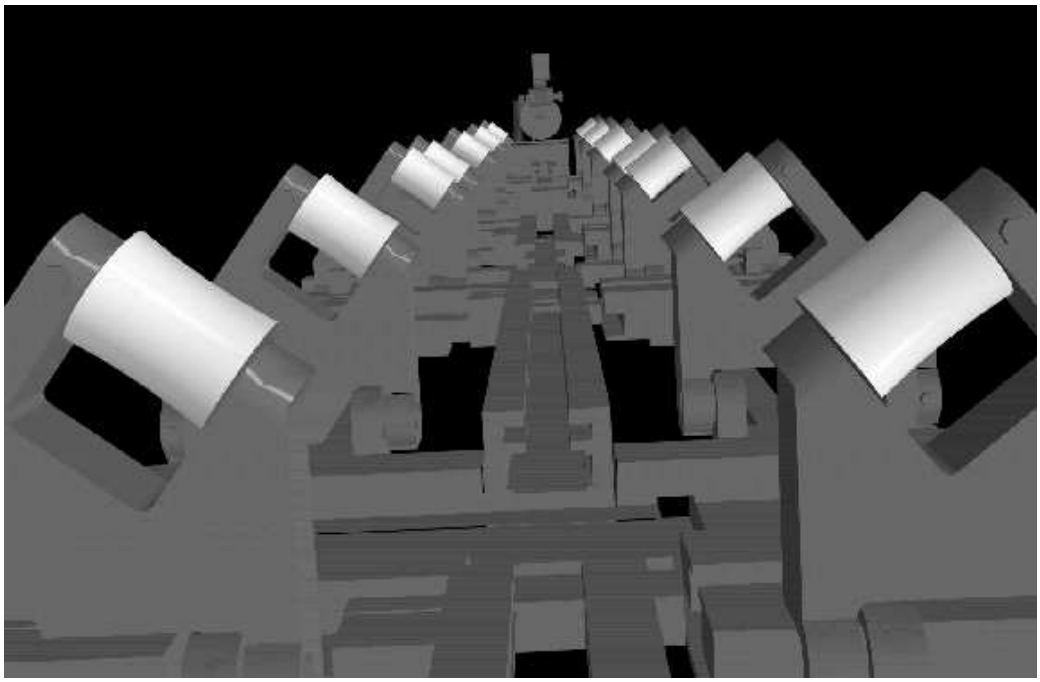

Figure 7.2: B-rep of Pivot from Submarine model (4100 Bézier patches) [Courtesy: Electric Boat]

ule. Initially, we partition the domain of each patch as determined by the intersection curve. The partitioning algorithm described in section 6.2.2 also generates the connectivity structure within each patch. Using this information and the original topology of the two solids, we create the graph whose connected components generate orientation invariant surface partitions. Construction of the graph Γ (connectivity information between various orientation invariant surface partitions) was described in section 6.2.3. Classification is done by ray-shooting. The ray-shooting test can be reduced to a collection of ray-surface intersections. In our implementation, we use *algebraic pruning* (chapter 3) to perform this operation.

The algorithm for component classification proceeds by computing all the intersections of a randomly directed ray with all the trimmed patches of the other solid. The parity (odd/even) of number of intersections decides the orientation (inside/outside) of the component. Guaranteeing the correctness of this operation is very crucial for the correctness of the final B-rep. In our system, we perform a number of redundant computations

to ensure this. The ray-surface intersection algorithm generates intersection points in the domain of each surface. If the chosen ray passes through the boundary of two adjacent patches, this point may be counted twice (once for intersection with each patch). To avoid this, we compare the corresponding intersections in 3-space and eliminate duplications. We also shoot multiple random rays to ensure correct parity. The result of the classification of one component is propagated throughout the adjacency graph Γ to resolve the other components. The propagation prevents us from having to do ray-shooting for each component, which is quite expensive.

The B-rep of the resulting solid and its topological structure are generated based on the Boolean operation begin performed. This data is fed back to the solid intersection module if the new solid enters into another Boolean operation.

7.1 Architecture of the BOOLE system

Figure 7.3 shows the basic architecture of the BOOLE system. The bottommost layer (Layer I) is composed of five major modules - the set of numeric libraries, symbolic module, geometric module, routines to manipulate parametric curves and surfaces, and graph algorithms. Here is a brief description about each.

- **Numeric libraries:** We make use of the public domain Fortran libraries EISPACK [GBDM77] and LAPACK [ABB+92]. These libraries provide most of the routines required by our algorithms like QR decomposition for computing eigenvalues and eigenvectors, LU decomposition for solution of linear systems and Singular Value Decomposition. Various parts of our surface-surface intersection algorithm use these numerical algorithms. We have also implemented the algorithm for local minimization given in Press et. al [PFTV90]. The minimization routine is used in conjunction with the tracing algorithm to improve the accuracy of the intersection curve.

- **Symbolic module:** This module comprises basically of routines for computing various resultants. We require only two kinds of resultant routines - Sylvester (eliminating one variable from system of two equations) and Dixon (eliminating two variables from

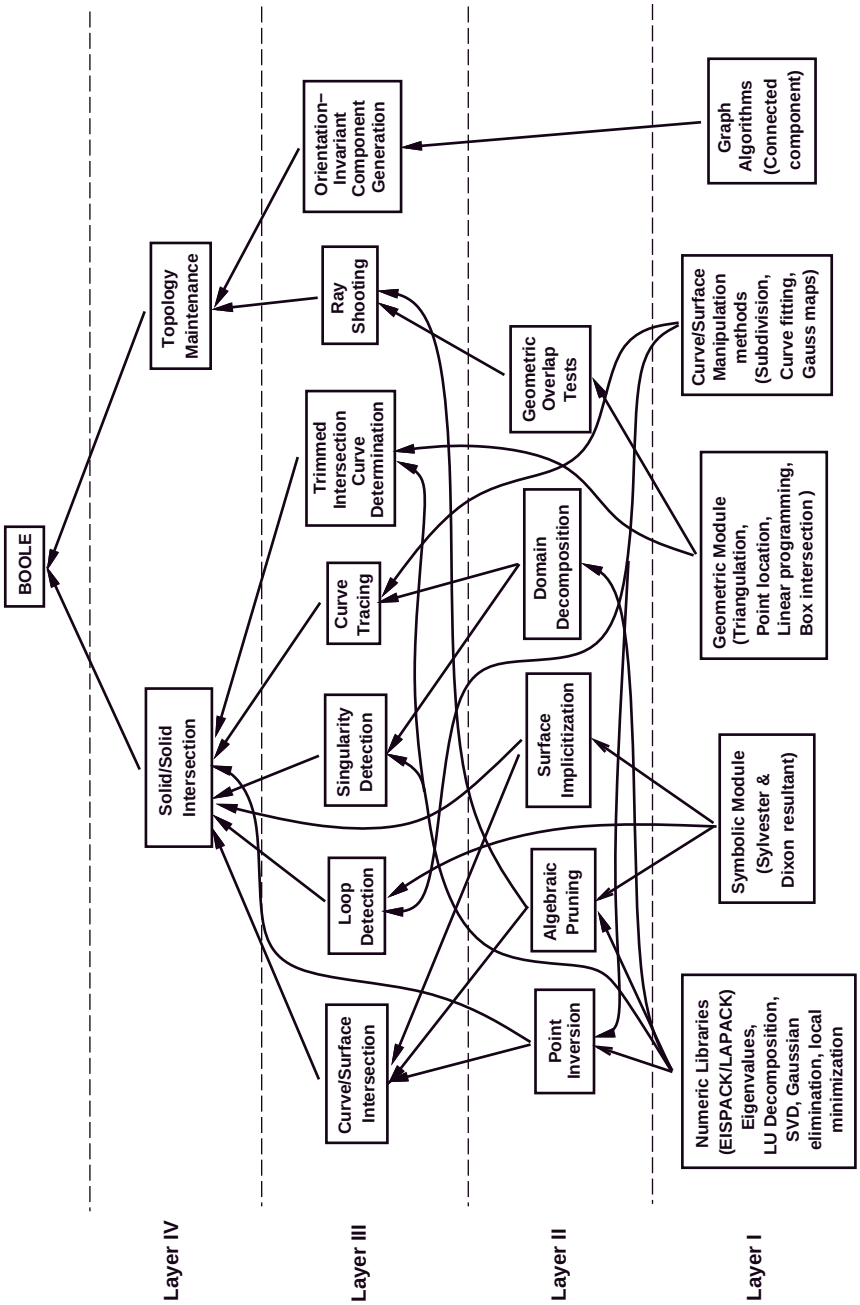

Figure 7.3: Various implementation layers in BOOLE

system of three equations). We use Sylvester resultant during curve-curve intersection as part of the algebraic pruning algorithm (chapter 3) and in the loop detection algorithm for planar sections of surfaces (chapter 5). Dixon's resultant is mainly used to compute implicit forms of surfaces (see chapter 4). These routines are implemented both in double precision arithmetic and in exact rational arithmetic.

- **Geometric module:** The geometric module contains algorithms for triangulation of simple polygons, point location in planar arrangements, linear programming and bounding box overlap tests. We use a very fast implementation of Seidel's triangulation algorithm [Sei91] provided by Atul Narkhede et al [NM95]. The point location algorithm based on the triangulation algorithm was also implemented by Atul Narkhede. We use Mike Hohmeyer's [Hoh91] implementation of Seidel's randomized linear programming algorithm [Sei90b]. We implemented the segment tree version of the bounding box intersection test described in chapter 6.

- **Curve/Surface manipulation module:** This module primarily handles all the low-level routines for manipulating parametric curves and surfaces. Typical algorithms are curve and surface subdivision (at certain parameter values), point evaluation on surfaces, pseudo-Gauss map evaluation and curve fitting. Curve fitting is a part of the BOOLE system that fits a parametric curve to an ordered set of points obtained after curve tracing. This routine is not used by the BOOLE system directly for B-rep computation. Rather, it is used as a means of data compaction by a display system (developed at UNC) that renders large NURBS models. Details of the curve fitting method can be found in [KKMN95].

- **Graph Algorithms:** This final module is used in maintaining topology information for each solid in our system. Apart from the simple tools to manipulate graph structures, it contains an algorithm to generate connected components in graphs. The algorithm uses repeated calls to a depth-first traversal routine in graphs. The running time of this algorithm is linearly proportional to the number of edges in the graph.

Layer II of our system contains routines that are directly called by our algorithms for curve-surface intersection, loop and singularity detection, curve tracing etc. These routines are listed in Figure 7.3. Given a point on a curve or surface, the problem of *point inversion* deals with the determination of parameter values which results in that point. Mathematically speaking, given a rational parameterization of a surface, $\mathbf{F}(s,t) = (X(s,t), Y(s,t), Z(s,t), W(s,t))$ and a point $(x,y,z) \in \mathcal{R}^3$, find the parameters (s_1, t_1) such that

$$X(s_1,t_1) = xW(s_1,t_1)$$
$$Y(s_1,t_1) = yW(s_1,t_1)$$
$$Z(s_1,t_1) = zW(s_1,t_1)$$

This operation is performed very often during curve tracing. *Algebraic pruning* is our method of solving zero-dimensional systems based on inverse power iterations. This algorithm relies heavily on the numeric libraries. We use algebraic pruning for curve-surface intersection queries and ray-shooting. The role of *surface implicitization* and *domain decomposition* in the surface-surface intersection algorithm were highlighted in chapter 4. *Geometric overlap* tests are performed to quickly prune out non-intersecting curves and surfaces. We use the implementation of linear programming and bounding box overlaps from layer I for this purpose.

The modules in Layer III include curve/surface intersection, loop and singularity detection, curve tracing, trimmed intersection curve determination, ray-shooting and orientation-invariant component generation. Each of these modules call a number of routines from layers I and II. The dependency structure of the various modules is shown in the figure. The modules in Layer III are in turn called by the topmost layer which includes solid-solid intersection and topology maintenance modules.

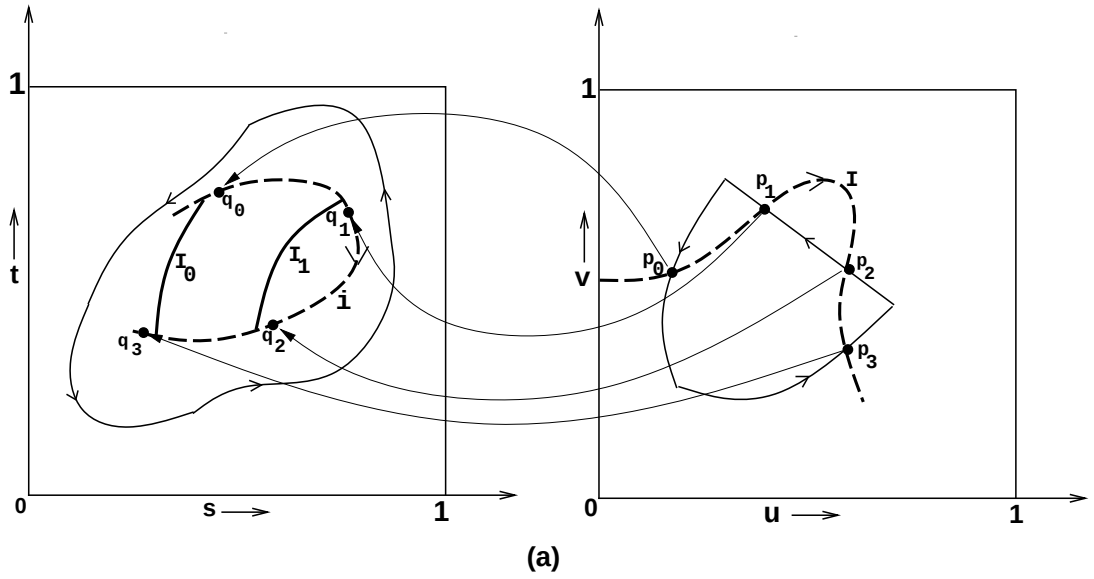

Figure 7.4: Inaccurate point inversion for curve merging

7.2 Robustness and Accuracy

One of the main problems in B-rep generation is robustness. An algorithm is said to be robust if for every valid input instance of the problem, it generates the corresponding valid output member. Consider the algorithm as a function $\mathcal{F}$ from the input set $\mathcal{I}$ to the output set $\mathcal{O}$.

$$\mathcal{F} : \mathcal{I} \rightarrow \mathcal{O}$$

In this definition, it is important for the algorithm to identify the type of input instance $i \in \mathcal{I}$ because the sequence of steps executed by the algorithm depends directly on i.

Most geometric algorithms are developed assuming that the input data are in general position, and that exact arithmetic provides reliable geometric primitives. However, for reasons of efficiency and feasibility, most implementations use floating point instead of exact arithmetic. Thus, the correctness of the mathematical algorithm does not extend directly to the implementation, and the system fails for seemingly innocuous input data (failure to classify the input instance correctly). This is the problem of "robustness" in

geometric computing.

However, if a particular instance is degenerate, the value of the corresponding expression is smaller than the errors accumulated due to fixed precision. There are two ways of dealing with this problem - tolerances and error estimates [For95]. Estimating tolerances when evaluating a complex sequence of predicates is non trivial, and error estimates are too pessimistic to be useful.

We shall now identify two areas where our algorithm is susceptible to failure when using floating point arithmetic. Most of these errors finally boil down to either point orientation tests or comparison between two floating point numbers.

Inaccurate point inversion for curve merging: It is a well-known fact that the intersection curve of two parametric surfaces is not rationally parametrizable in general. As a result, these curves are approximated as piecewise linear curves or splines to within a fixed tolerance (which is either too conservative or arbitrarily chosen). Since most of the surface patches we are dealing with are trimmed, we need to compute portions of the intersection curve that lie inside the trimmed boundaries of both the patches. Figure 7.4 shows one such example. The curve $\mathbf{I}$ shown in dotted lines is the intersection curve in both the domains. $\mathbf{I_0}$ and $\mathbf{I_1}$ are the intersection curves on the left patch obtained from other surfaces. To compute the actual intersection curve for trimmed patches, we need to compute the intersection points of the curve with the trimming boundary. $\mathbf{p_0}$, $\mathbf{p_1}$, $\mathbf{p_2}$ and $\mathbf{p_3}$ are four such points on the right patch. If the boundary curves or the intersection curve are not accurate, neither are the $\mathbf{p_i}$'s. They may not even lie on the actual intersection curve. Corresponding to the $\mathbf{p_i}$'s, we need to compute $\mathbf{q_i}$'s on the other patch to determine which portions of the intersection curve to retain. This process is *point inversion* which was described earlier in section 7.1. Two problems can arise in inversion: (a) there may not be any corresponding point on the other patch (because $\mathbf{p_i}$'s do not lie exactly on the intersection curve), or (b) the $\mathbf{q_i}$'s could be positioned such that the curve segments $\mathbf{q_0q_1}$ and $\mathbf{q_2q_3}$ do not match up with $\mathbf{I_0}$ and $\mathbf{I_1}$ for curve merging.

Using our representation of the intersection curve as the singular set of a bivariate matrix polynomial, we ensure accurate computation of $\mathbf{p_i}$'s (see section 5.1). Further, using our intersection curve formulation, the inverted point in the other domain can be obtained

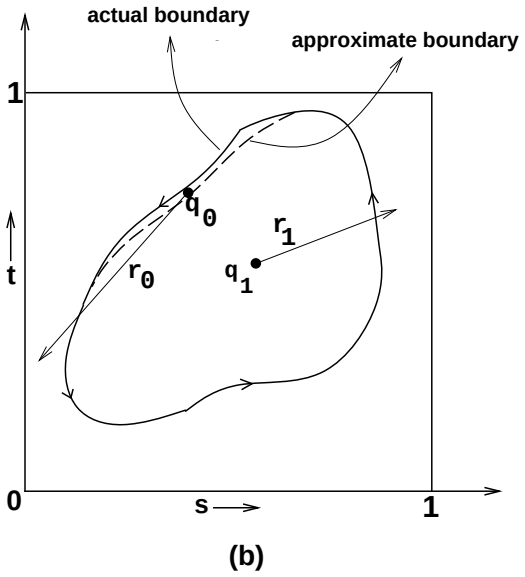

Figure 7.5: Inaccurate point classification

from the kernel of the singular matrix (chapter 4).

Inaccurate point classification: Another area where floating point errors result in failure of the algorithm is during component classification. As described earlier, we use ray shooting for this purpose. The entire computation boils down to classifying whether a point lies inside or outside the trimming region. Figure 7.5 shows an example. In most cases, classifying points like $\mathbf{q_1}$ is not a problem. One ray-shooting query will determine it. However, consider a point like $\mathbf{q_0}$ which lies very close to the boundary. Approximate representations of the trimming boundary makes classifying $\mathbf{q_0}$ a major problem. Depending on the choice of ray directions and the tolerances used we may get different classifications. This error could result in topologically inconsistent answers. As described in section 5.4, we use singular value decomposition to resolve this problem.

We also perform a number of checkpointing operations in our implementation that control the accumulation of floating point error. We also handle degenerate cases like face-face and edge-edge overlaps while performing regularized Boolean operations. These are handled as special cases in our system. In practice we have observed that our system generates accurate B-reps on most input cases. Since the implementation was done using

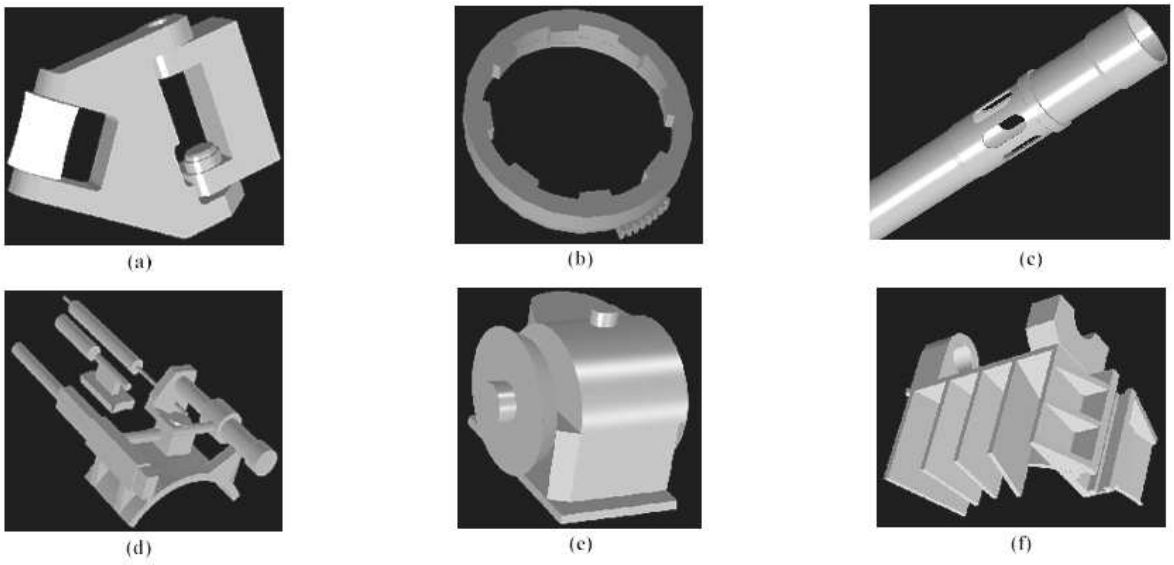

Figure 7.6: B-reps of some solids from the submarine storage and handling room

floating point arithmetic, we also use tolerances to compare such values. Finding a tolerance that works for all models is very difficult. In some cases, we had to change tolerances to make our system work. Currently, we are incorporating B-rep computation using exact rational arithmetic [KKM97] to prevent most robustness and accuracy problems. The use of exact arithmetic can slow down the computation time, however exploiting parallelism helps significantly in the overall speed.

The accuracy of the B-rep generated is determined by the accuracy of the intersection curves between solids. In our system, the accuracy of these curves can be controlled by the user. Depending on the application, our system can generate very accurate B-reps at the expense of computation time.

7.3 Parallel Implementation

The various stages of our algorithm is explained using an example in Figure 7.7 and Figure 7.8. Since we are dealing with sculptured solids with trimmed Bézier patches, as opposed to polyhedral solids, the complexity of the whole system is increased significantly.

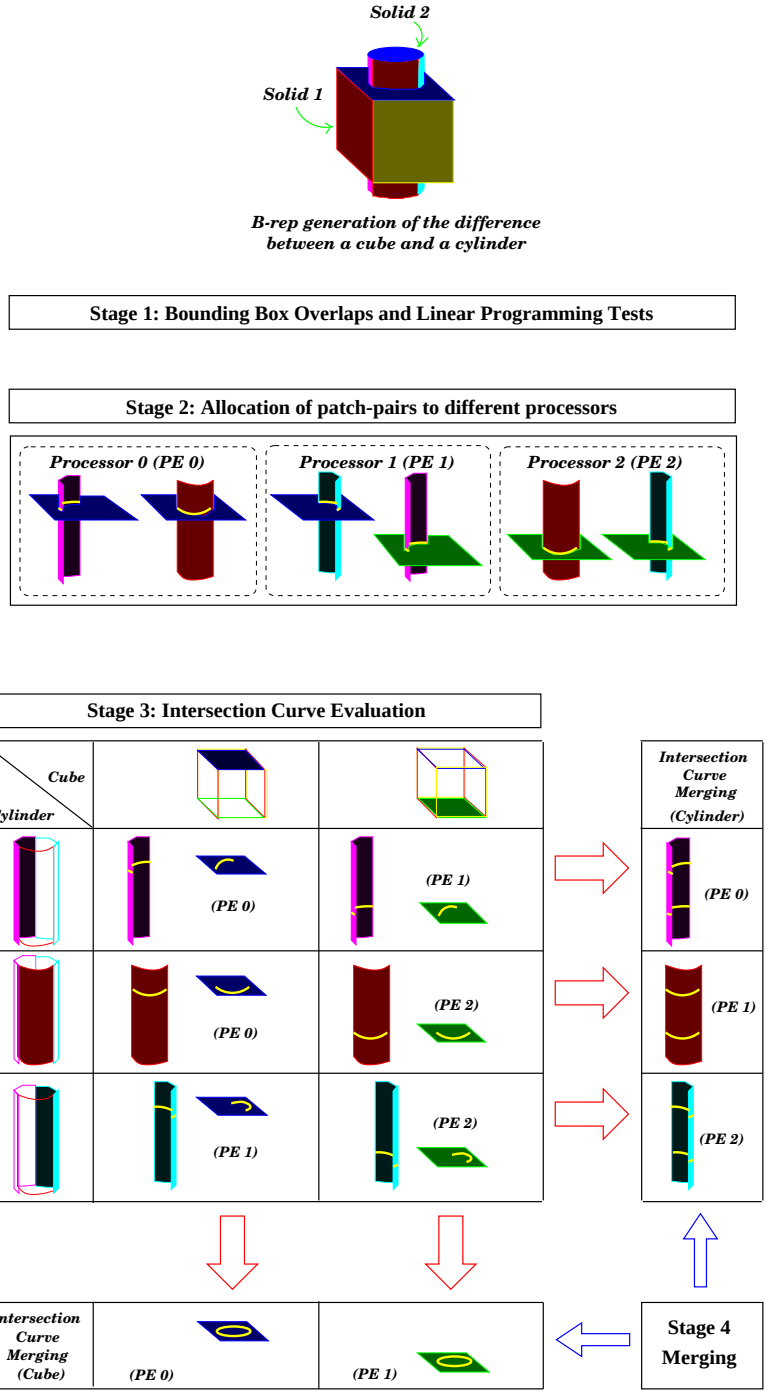

Figure 7.7: Intersection curve computation and curve merging

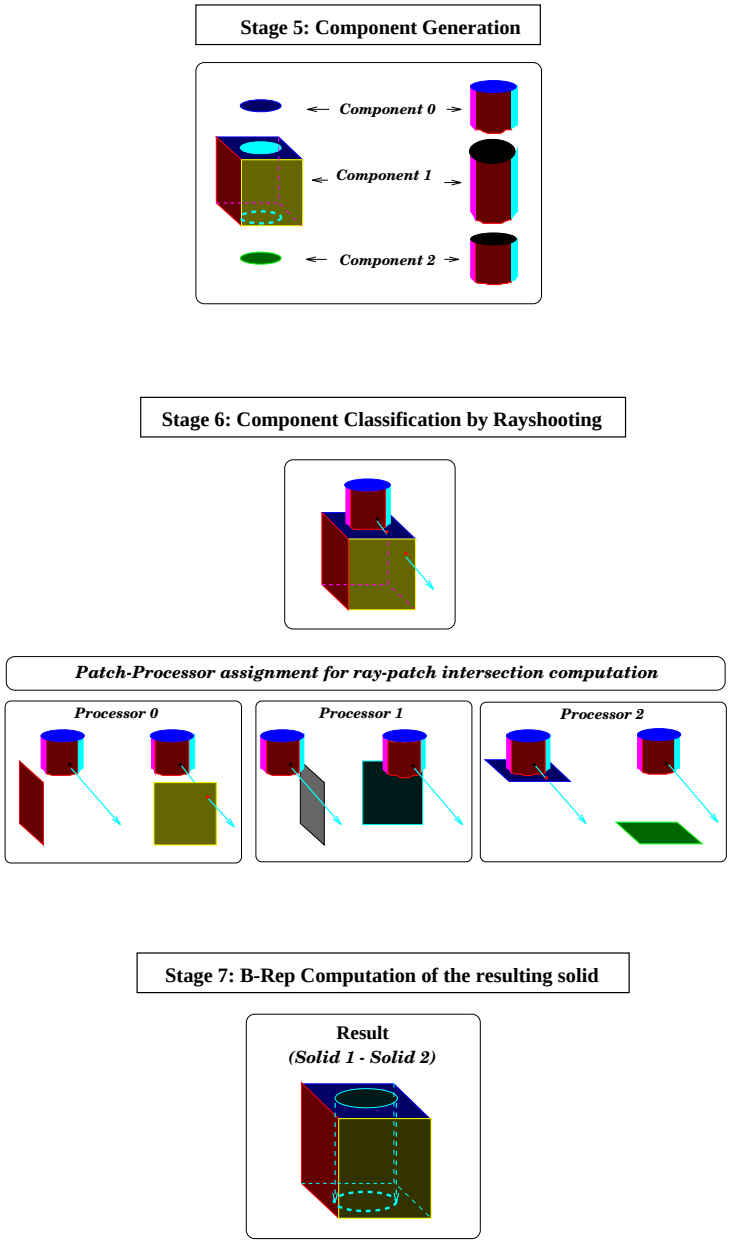

Figure 7.8: Component generation, classification and B-rep computation

The time taken for the surface-surface intersection algorithm described in chapter 4 is a cubic function of the degree of the patch in the worst case. Further, the complexity of ray-patch intersection evaluation is again dependent on the degree of the patch. These parts are computationally most intensive and form the main bottleneck in terms of system performance. To improve the computation time, we have implemented a parallel version of the algorithm on existing shared memory multiprocessor architectures like SGI-Onyx. One of the main issues that arise while parallelizing an algorithm over many processors is to ensure that each processor performs roughly equal amount of work. This issue of load balancing is discussed next.

7.3.1 Load Balancing Algorithm

The problem of load balancing arises when an algorithm has to be parallelized among a number of processors. The running time of the parallel algorithm is directly related to the maximum execution time of the task at a single processor. It is clear that the most effective parallel algorithm is one where the tasks are equally distributed among all the processors. The problem of load balancing has received considerable attention for a long time due to the fact that a single scheme is not applicable for parallelizing all algorithms [Lam87, YA93, Whi94, Gea95, HL95]. The effectiveness of different techniques varies with the nature of the problem it is used for. Hence there arises a need for newer problem specific analysis methods which help in choosing the most effective load balancing technique. We shall now describe three such techniques that we use to shared memory multiprocessor architectures for boundary computation.

- **Static load balancing:** Static load balancing is done by dividing the given problem consisting of n tasks into p (number of processors) parts and submitting each part to a single processor. The size of each problem piece is precomputed and is not changed during execution. This technique works best when the processing time of each of the tasks is known, and the number of tasks does not change during execution. Extracting parallelism in our B-rep converter starts from computing the bounding boxes for all the patches (Stage 1 in Figure 7.7). As the bounding box computation for each

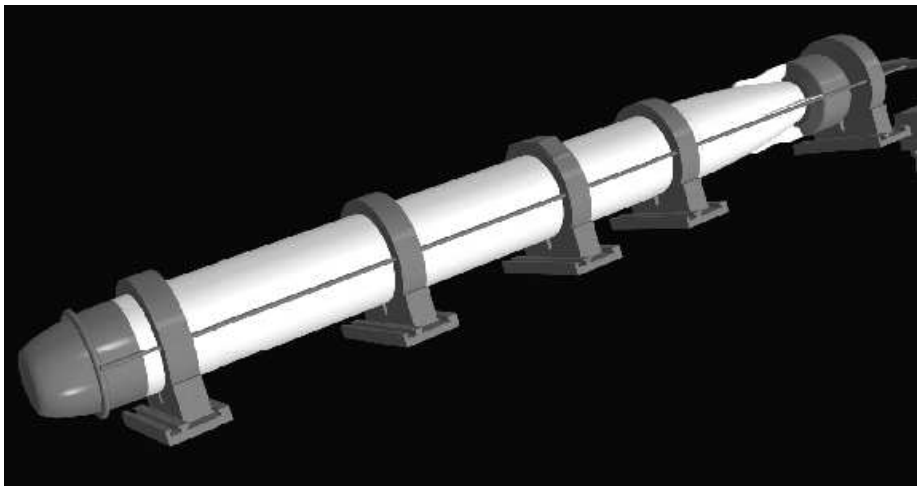

Figure 7.9: B-rep of Shipping line from Submarine model (3400 Bézier patches) [Courtesy: Electric Boat]

patch is independent of the other, this can be easily parallelized. Further as the amount of work that is to be done for the bounding box computation for each patch is approximately the same, load balancing is achieved statically. Once the bounding boxes for all the patches have been computed, the overlap tests is also performed in parallel.

- **Global queue:** In many algorithms, it is not possible to estimate the execution time of each task. For example, execution time for computing the intersection curve between two surfaces can vary depending on the number of curve components, and length (in terms of number of points traced) of each component. In this technique, when one processor is accessing the task queue, the queue should be locked to ensure exclusive access (mutual exclusion). This technique achieves the best load balancing, though the extra work done for balancing the load in the form of locks might offset its advantage. In our system, using global queues with locks to perform load balancing was not as efficient as dynamic load balancing (described below). We believe it is because of the reasons cited above.

- **Dynamic load balancing:** In this technique, a local job queue is maintained for each processor. Initially, tasks are assigned to every processor similar to the static load

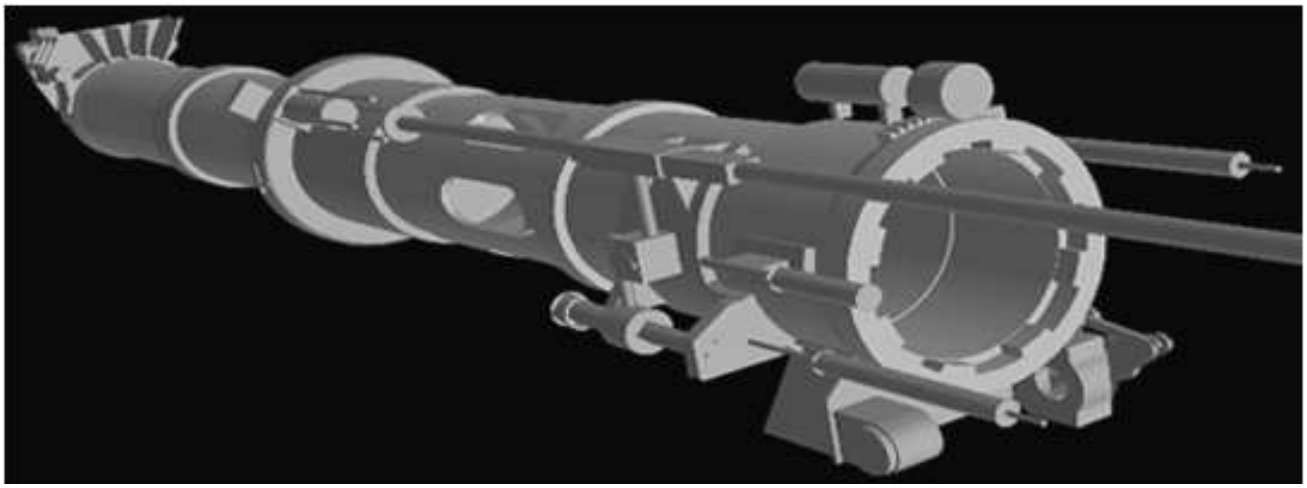

Figure 7.10: B-rep of Torpedo tube from Submarine model (1200 Bézier patches) [Courtesy: Electric Boat]

balancing scheme. However, due to suboptimal task division, some processors might complete their tasks before others. In this scenario, the idle processors share the load with the busy processors, thereby balancing the load dynamically. If we can ensure that each busy processor is accessed by only one idle processor at any time, then a lock-free implementation of this scheme is possible. We can also ensure that each task is processed only once, and no task is left out. In our application, load balancing is efficiently achieved by minimal use of locks. Therefore, we use this approach for our most computationally intensive tasks like surface-surface intersection (Stage 3 in Figure 7.7) and ray-shooting computation (Stage 6 in Figure 7.8).

If we ensure that only one idle processor will access a particular busy processor, then a lock free implementation of dynamic load balancing is possible. We enforce a unique one to one correspondence between an idle and busy processor using the following algorithm. A shared global variable *WhichIdleProc* stores the id of the idle processor, which now has the chance to choose its busy processor. This serializes the operation of finding an idle-busy processor pair. In our implementation, we choose a single lock to guard this critical section because the computation time for surface-surface intersection and ray-shooting dominates one locking operation. With each busy processor, we associate a shared variable *MyIdleProc*, which stores the idle processor id that has been paired up with that particular

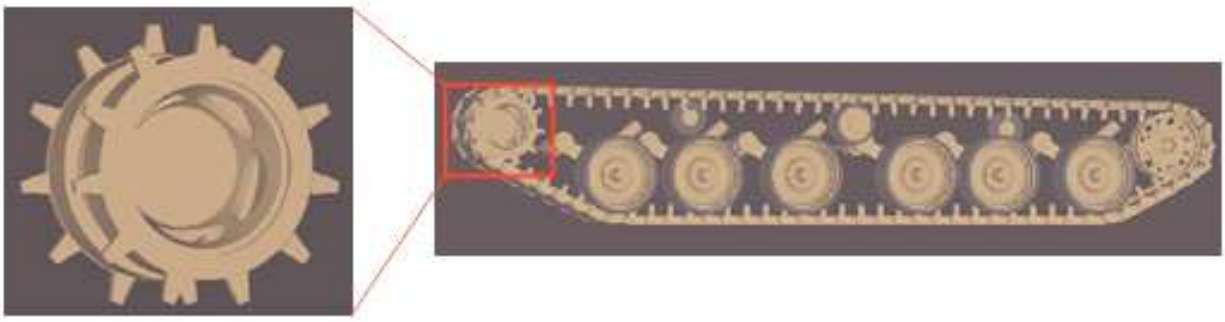

Figure 7.11: Track from the Bradley model showing placement of drivewheel model (15000 Bézier patches) [Courtesy: Army Research Labs]

busy processor. These variables are initialized to **NIL**, referring to none of the processors. Each processor also maintains its processor number in a local variable **myid**. Whenever a processor becomes idle, it executes the following code.

```
{
 If (WhichIdleProc == NIL) then {
   GetLock( GetMeAccess);
     if(GetMeAccess == NIL) {
       GetMeAccess = myid;
       WhichIdleProc = myid;
     }
   ReleaseLock(GetMeAccess);
 }
 /* Waiting for my chance */
 while (WhichIdleProc ≠ myid);

 /* All tasks completed */
 If (NoMoreBusyProc()) then exit;

 /* All Busy processors are being load
 balanced by some idle processor */
```

```
while (GetBusyProc() == NIL);

/* Got a Busy Processor to pair up with */
MyBusyProc = GetBusyProc();

/* Make sure no one else captures this busy processor */
MyIdleProc[MyBusyProc] = myid;

/* Give chance to next idle proc to find its partner */
If (NextIdleProc()) then  WhichIdleProc = NextIdleProc();

/* No one to grab the chance */
                    else {
                     GetLock(GetMeAccess);
                        WhichIdleProc = NIL;
                        GetMeAccess = NIL;
                     ReleaseLock(GetMeAccess);
                    }

/* Balancing the load with the partner */
LoadBalance(MyBusyProc);

/* Finished load sharing; Freeing my partner */
MyIdleProc[MyBusyProc] = NIL;

/* Register myself as busy */
If (IHaveLoad()) BUSY[myid] = TRUE;

/* Work on new list of tasks */
```

Figure 7.12: B-reps of some solids from the Bradley fighting vehicle

```
 PerformSurfaceIntersection(); or PerformRayShooting();

 /* Register myself as idle */
 BUSY[myid] = FALSE;
}
```

Initially, the variable *WhichIdleProc* has to be set by the idle processor to gain access to the list of busy processors. Race condition occurs only when the variable *WhichIdleProc* is **NIL** and more than one idle processor try to access it. By making *WhichIdleProc* a critical resource, we can ensure mutual exclusion while setting this variable. This can be achieved by using locks. The number of locking operations can be reduced by allowing free access to *WhichIdleProc* and introducing a new shared variable *GetMeAccess*, which is locked only when a race condition occurs. Locks can be totally avoided by maintaining a random permutation of the busy processor list locally in every processor. This does not guarantee that a single idle processor captures a busy processor, however, the probability of a race condition is very small.

7.4 Performance

In this section, we highlight the performance of both the sequential and parallel algorithm on some real-world models. We obtained a model of a submarine storage and

Model	# of CSG opns.	Running time (in secs.)				# of patches (in B-Rep)
		BB & LP test	SSI	Ray-shooting	Total	
Fig. (a)	20	1.7	41.3	13.6	77.0	137
Fig. (b)	5	0.3	9.7	3.6	16.3	89
Fig. (c)	5	0.8	11.6	5.4	18.5	116
Fig. (d)	27	2.3	58.9	17.8	98.7	169
Fig. (e)	10	1.8	28.5	6.7	41.1	69
Fig. (f)	21	2.0	35.1	13.8	64.2	146

Table 7.1: Performance of our system on parts of the submarine model

Model	# of CSG opns.	Running time (in secs.)				# of patches (in B-Rep)
		BB & LP test	SSI	Ray-shooting	Total	
Link	16	1.3	26.3	9.6	47.81	76
Drivewheel	44	5.8	54.3	27.1	97.23	289
Idlerwheel	48	5.1	59.8	28.9	106.93	235

Table 7.2: Performance of our sequential algorithm on parts of the Bradley model

handling room through the courtesy of Electric Boat Inc., a division of General Dynamics. This model consists of about 2000 solids. Many of the primitives are composed of polyhedra and conicoids like spheres or cylinders. Additional primitives include generalized prisms and surfaces of revolution of degrees 6 or more. A few of the primitives are composed of Bézier surfaces of degree as high as 12. Most of the CSG trees have heights ranging between 6 and 12 and some of them are as high as 30. Table 7.1 shows the performance of the sequential algorithm on some solids from this model (see Figure 7.6). The column with running time is broken into four parts: the bounding box and linear programming, surface-surface intersection, ray-shooting and total. The final column indicates the number of trimmed patches that the final model has.

The model of the Bradley fighting vehicle was obtained from Army Research Laboratories. It is composed of more than 8500 solids each consisting of about 5-8 Boolean operations. The primitives in the Bradley are solids like conicoids (spheres, cylinders, ellipsoids etc.) and tori whose B-reps can be represented by biquadric (degree 2 *times* 2) Bézier patches. We present the performance of our sequential and parallel algorithms on three of

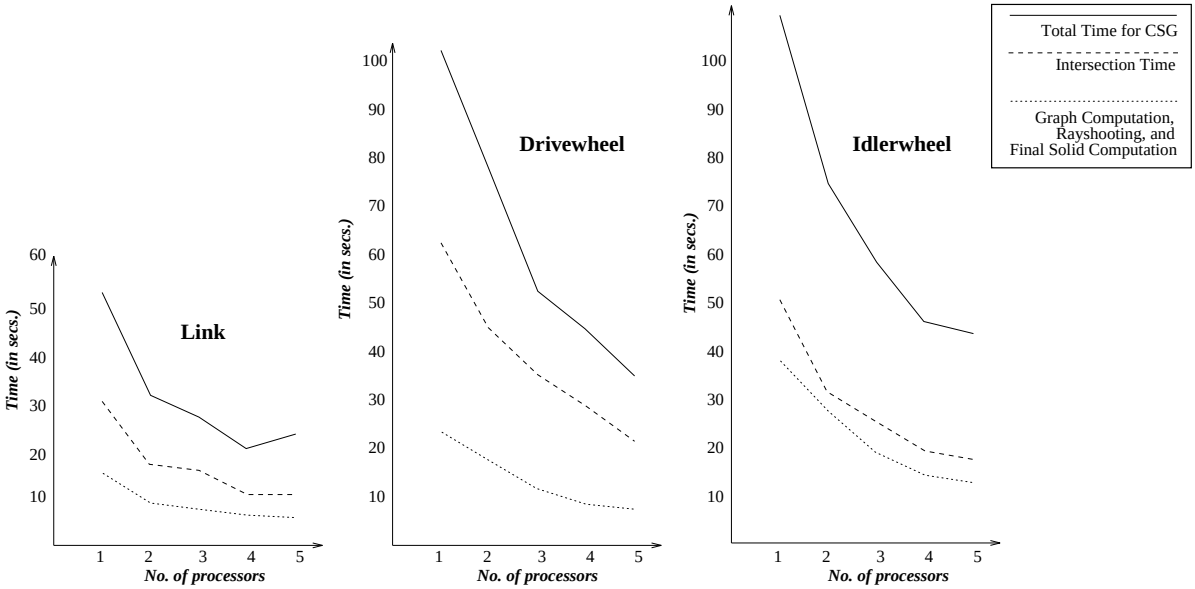

Figure 7.13: Performance of our parallel algorithm as a function of processor count

Model	Total running time (in secs.)				
	1 proc.	2 proc.	3 proc.	4 proc.	5 proc.
Link	51.95	30.88	26.93	20.55	23.44
Drivewheel	102.32	77.39	53.39	49.02	35.67
Idlerwheel	112.40	74.51	58.96	46.10	44.23

Table 7.3: Performance of our parallel algorithm on parts of the Bradley model

the solids in the Bradley fighting vehicle.

- **Link model:** It consists of 16 Boolean operations and the B-rep contains 76 trimmed Bézier patches. Figure 7.12(a) shows the model. The graph in Figure 7.13 shows the performance of our system on varying number of processors. It can be seen that the performance becomes worse when we go from four to five processors. Since this is not a very complex model, the setup costs of using five processors outweigh the benefit of parallelism.

- **Drivewheel model:** This model is constructed using 44 Boolean operations. The B-rep is shown in Figure 7.12(b) and consists of 289 trimmed Bézier patches.

- **Idlerwheel model:** The B-rep of the idlerwheel (composed of 235 trimmed Bézier patches) is shown in Figure 7.12(c) and took 48 Boolean operations to generate. Again increasing the processor count reduces the running time because of complexity of the model.

Table 7.2 and Table 7.3 shows the performance of our sequential and parallel algorithm on the parts of the Bradley model shown in Figure 7.12 respectively.

Chapter 8

Conclusion and Future Work

Evaluating Boolean set operations of sculptured solid objects is one of the most powerful facilities available in a solid modeler. In modelers based on boundary representations, the Boolean set operation algorithm is also technically one of the most demanding component. A significant portion of the complexity is due to the computation and representation of intersection curves between free-form surfaces. Apart from the algebraic and geometric difficulties, a convenient representation of the intersection curve is essential to effectively compute the boundary. Another important issue in this context is that of robustness on models of large scale. Our experience with the Bradley fighting vehicle and submarine model shows that extremely large CAD models are designed using Boolean set operations for physical analysis and model verification. Individual solids are generated using a large number of successive Boolean operations. In such cases, systematically dealing with the growth of errors due to finite precision arithmetic is very difficult and impractical, especially for solids with curved primitives. The best way to deal with such problems is to combine numerically stable algorithms with the use of easy-to-implement heuristics that control the growth of error. In this dissertation, we have explored each of these issues and proved the thesis,

> The *lower dimensional* surface intersection formulation provides an effective representation to perform Boolean operations on sculptured models.

This dissertation presents a number of techniques to effectively compute boundary representations of Boolean combinations of sculptured primitives and perform associated surface interrogations like surface-surface and curve-surface intersections. It employs a combination of symbolic and numeric methods to compute the B-reps accurately and efficiently. The input to our algorithm is a CSG tree that describes the solid as a Boolean expression of primitive solids. The choice of the set of primitive solids is arbitrary as long as they can be represented as a piecewise collection of parametric surface patches. Our portable implementation of the algorithms presented in this dissertation, BOOLE, has been successfully applied to generate the boundary representations of industrial models composed of thousands of Boolean set operations. BOOLE is currently available for download at http://www.cs.unc.edu/~geom/CSG/boole.html.

One of the main contributions of this dissertation is a new algebraic representation for the intersection curve of two parametric surfaces. The intersection curve is evaluated by adopting numerical curve tracing methods on this representation. The performance of this algorithm is output sensitive (in terms of separability of curve components), and typically performs an order of magnitude faster than previously known robust algorithms. This method is an advancement over existing finite-precision surface intersection algorithms, in that it guarantees the correct topology of the intersection curve. It does not suffer from robustness problems like *loop* and *singularity* detection, and *component jumping* present in numerical approaches, or the efficiency problems of purely algebraic methods. Further, the accurate representation of the intersection curve in the parametric space of each surface is appropriate for use in boundary evaluation algorithms.

The boundary evaluation algorithm generates B-reps in the form of trimmed parametric surfaces. The trimming curves on each patch are the result of intersections with surfaces of other solids. The accurate representation of the intersection curves guarantee accurate B-reps as well. Furthermore, the algorithm is implemented on general purpose processors, and has been parallelized on existing shared memory architectures. In our parallel implementation on an SGI-Onyx with four R10000 processors, we are able to perform Boolean operations on sculptured solids at interactive rates.

The research work conducted in this dissertation shows that it is possible to gen-

erate accurate B-reps of sculptured solid objects in an efficient manner. However, this work is merely a first step towards obtaining a B-rep modeler that provides accurate results at all times. There are still quite a few important questions that are yet to be addressed.

8.1 Ongoing and Future Work

In this section, we discuss a few of the many extensions that are possible from this work.

- **Robustness:** Our current work (in collaboration John Keyser at UNC) is focused on issues of robustness, which arise in solids model designs with curved surfaces. By robustness in this context, we mean that the boundary evaluation algorithm must be able to handle all geometric situations (including degeneracies) and produce correct results. The use of exact arithmetic coupled with perturbation schemes has been shown to be successful in polyhedral modeling [For95]. We are trying to extend this work to curved geometries. One class of robustness problems is degeneracies like the ones listed in section 6.3. The other class of robustness problems involves cases in which the use of fixed precision (double-precision floating-point arithmetic) is not accurate enough to determine B-reps correctly. Previous work on robustness issues has dealt primarily with linear (polyhedral) cases. Robustness problems in curved surface cases are both more numerous and more difficult to handle. Experience with the BOOLE system has shown us that these problems can arise in a significant number of real-world cases.

  We are currently exploring approaches to address the robustness issue by making use of exact arithmetic and exact representations [KKM97]. This eliminates the problems related to numerical precision. In addition, the use of exact arithmetic allows us to use perturbations methods to eliminate degeneracies. Perturbation methods have proven to be useful at eliminating degeneracies in the linear case and may be similarly useful in the curved-surface domain. The use of exact arithmetic can lead to highly inefficient implementations. In order to increase the efficiency of our approach, we have isolated a few key kernel routines which govern the efficiency of the overall program.

We use improved symbolic techniques and a combination of exact and floating-point arithmetic to speed up our kernel functions, and thus the entire algorithm.

- **Nonmanifold solids:** This dissertation does not discuss set operations on nonmanifold geometries. The number of cases of classification tests to be performed is significantly higher, and thus makes it much more complicated.

- **Use of implicit surfaces:** Our algorithms generate boundary representations in terms of parametric surfaces. The ability to generate boundaries of solids using implicit surfaces greatly enhances the power of the solid modeling system because of the greater representation power of these surfaces. However, some of the questions we have to answer deal with the representation of trimmed implicit surfaces and numerical stability of algorithms.

- **Extension to other problem domains:** The surface-surface intersection algorithm discussed in this dissertation is applicable in a wider range of problems. It can be applied to solve any algebraic system that has a one-dimensional solution set. For example, we have use the algorithm to find the silhouette of a parametric surface from a given viewpoint. Another application in the field of geometric modeling pertains to the generation of offsets and blends of surfaces. Applications to vision and path planning related problems seem possible.

Bibliography

[AB88a] S.S. Abhyankar and C. Bajaj. Automatic parametrizations of rational curves and surfaces iii: Algebraic plane curves. *Computer Aided Geometric Design*, 5:309–321, 1988.

[AB88b] S.S. Abhyankar and C. Bajaj. Computations with algebraic curves. In *Lecture Notes in Computer Science*, volume 358, pages 279–284. Springer Verlag, 1988.

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. *LAPACK User's Guide, Release 1.0.* SIAM, Philadelphia, 1992.

[Abh90] S. S. Abhyankar. *Algebraic Geometry for Scientists and Engineers.* American Mathematical Society, Providence, R. I., 1990.

[ACM84] D. S. Arnon, G.E. Collins, and S. McCallum. Cylindrical algebraic decomposition. *SIAM J. on Computing*, 13:878–889, 1984.

[AF88] S. Arnborg and H. Feng. Algebraic decomposition of regular curves. *Journal of Symbolic Computation*, 5:131–140, 1988.

[AM88] D. Arnon and S. McCallum. A polynomial time algorithm for the topological type of a real algebraic curve. *Journal of Symbolic Computation*, 5:213–236, 1988.

[Arn83] D. S. Arnon. Topologically reliable display of algebraic curves. *Computer Graphics*, 17:219–227, 1983.

[Bau75] B. Baumgart. A polyhedron representation for computer vision. In *National Computer Conference, AFIPS Conf. Proc.*, pages 589–596, 1975.

[BBP95] I. Biehl, J. Buchmann, and T. Papanikolaou. Lidia: A library for computational number theory. Technical Report SFB 124-C1, Fachbereich Informatik, Universitt des Saarlandes, 1995.

[BFJP87] R. Barnhill, G. Farin, M. Jordan, and B. Piper. Surface/surface intersection. *Computer Aided Geometric Design*, 4(3):3–16, 1987.

[BHHL88] C.L. Bajaj, C.M. Hoffmann, J.E.H. Hopcroft, and R.E. Lynch. Tracing surface intersections. *Computer Aided Geometric Design*, 5:285–307, 1988.

[BK90] R.E. Barnhill and S.N. Kersey. A marching method for parametric surface/surface intersection. *Computer Aided Geometric Design*, 7:257–280, 1990.

[BMP94] M. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation. *Computer-Aided Design*, 26(6), 1994.

[Bra75] I. Braid. The synthesis of solid bounded by many faces. *Comm. ACM*, 18:209–216, 1975.

[Buc89] B. Buchberger. Applications of groebner bases in non–linear computational geometry. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 415–447. MIT Press, 1989.

[Cam85] S. A. Cameron. A study of the clash detection problem in robotics. *IEEE Conference on Robotics and Automation*, pages 488–493, 1985.

[Can88] J.F. Canny. *The Complexity of Robot Motion Planning.* ACM Doctoral Dissertation Award. MIT Press, 1988.

[Cas87] M. S. Casale. Free-form solid modeling with trimmed surface patches. *IEEE Computer Graphics and Applications*, pages 33–43, January 1987.

[CB89] M. S. Casale and J. E. Bobrow. A set operation algorithm for sculptured solids modeled with trimmed patches. *Computer Aided Geometric Design*, 6:235–247, 1989.

[CH88] G.W. Crippen and T.F. Havel. Distance geometry and molecular conformation. Research Studies Press, New York, 1988.

[Cha87] K. Chan. *Solid Modelling of Parts with Quadric and Free-form Surfaces.* PhD thesis, University of Hong Kong, 1987.

[Che89] K.P Cheng. Using plane vector fields to obtain all the intersection curves of two general surfaces. In *Theory and Practice of Geometric Modeling*, pages 187–204, 1989.

[CK83] H. Chiyokura and F. Kimura. Design of solids with free-form surfaces. *Computer Graphics*, 17:289–298, 1983.

[Col75] G.E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Lecture Notes in Computer Science*, number 33, Springer-Verlag, 1975.

[Cra89] J.J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison–Wesley Publishing Company, 1989.

[CS85] M.S. Casale and E.L. Stanton. An overview of analytic solid modeling. *IEEE Computer Graphics and Applications*, 5:45–56, February 1985.

[DC95] E. Driskill and E. Cohen. Interactive desigb, analysis, and illustration of assemblies. In *Proc. of 1995 Symposium on Int. 3D Graphics*, pages 27–34, 1995.

[Dix08] A.L. Dixon. The eliminant of three quantics in two independent variables. *Proceedings of London Mathematical Society*, 6:49–69, 209–236, 1908.

[Dok85] T. Dokken. Finding intersections of b-spline represented geometries using recursive subdivision techniques. *Computer Aided Geometric Design*, 2:189–195, 1985.

[DSY89] T. Dokken, V. Skytt, and A.M. Ytrehus. Recursive subdivision and iteration in intersections and related problems. In *Mathematical Methods in Computer-Aided Geometric Design*, pages 207–214. Academic Press, 1989.

[Duf92] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics*, 26(2):131–139, 1992.

[EC90] G. Elber and E. Cohen. Hidden curve removal for free form surfaces. *Computer Graphics*, 24(4):95–104, 1990.

[EC94] G. Elber and E. Cohen. Exact computation of gauss maps and visibility sets for freefrom surfaces. Technical report CIS #9414, Computer Science Department, Technion, 1994.

[Ede83] H. Edelsbrunner. A new approach to rectangle intersections, part i. *Int. J. of Comput. Math*, 13:209–219, 1983.

[EKL+91] J. L. Ellis, G. Kedem, T. C. Lyerly, D. G. Thielman, R. J. Marisa, J. P. Menon, and H. B. Voelcker. The raycasting engine and ray representations. In *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 255–267, 1991.

[Far86] R.T. Farouki. The characterization of parametric surface sections. *Computer Vision, Graphics and Image Processing*, 33:209–236, 1986.

[Far93] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1993.

[FBZ93] S. Fang, B. Bruderlin, and X. Zhu. Robustness in solid modeling: a tolerance-based intuitionistic approach. *Computer-Aided Design*, 25(9):567–576, 1993.

[FF92] D.A. Field and R. Field. A new family of curves for industrial applications. Technical report GMR-7571, General Motors Research Laboratories, 1992.

[FH85] R.T. Farouki and J.K. Hinds. A hierarchy of geometric forms. *IEEE Computer Graphics and Applications*, 5:51–78, May 1985.

[FM84] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. Graph.*, 3:153–174, 1984.

[For95] S. Fortune. Polyhedral modeling with exact arithmetic. *Proceedings of ACM Solid Modeling*, pages 225–234, 1995.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.

[GBDM77] B.S. Garbow, J.M. Boyle, J. Dongarra, and C.B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51. Springer-Verlag, Berlin, 1977.

[Gea95] G. Georgiannakis and C. Houstis et. al. Description of the adaptive resource management problem, cost functions and performance objectives. Technical Report TR130, The Institute of Computer Science, Foundation for Research and Technology, 1995.

[Gei83] A. Geisow. *Surface Interrogations*. PhD thesis, School of Computing Studies and Accountancy, University of East Anglia, 1983.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins Press, Baltimore, 1989.

[HHK89] C. Hoffmann, J. Hopcroft, and M. Karasick. Robust set operations on polyhedral solids. *IEEE Computer Graphics and Applications*, 9(6):50–59, 1989.

[Hil82] R. C. Hillyard. The build group of solid modellers. *IEEE Computer Graphics and Applications*, 2:43–52, 1982.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Proc. Supercomputing '95*, 1995.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[Hof90] C.M. Hoffmann. A dimensionality paradigm for surface interrogations. *Computer Aided Geometric Design*, 7:517–532, 1990.

[Hoh91] M.E. Hohmeyer. A surface intersection algorithm based on loop detection. *International Journal of Computational Geometry and Applications*, 1(4):473–490, 1991. Special issue on Solid Modeling.

[Hoh92] M.E. Hohmeyer. *Robust and Efficient Intersection for Solid Modeling.* PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[Ips97] I. C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Reviews*, 39(2):254–291, 1997.

[Jac95] D. Jackson. Boundary representation modeling with local tolerances. *Proceedings of ACM Solid Modeling*, pages 247–253, 1995.

[Jar84] G.E.M Jared. Synthesis of volume modeling and sculptured surfaces in build. In *CAD84, Computers in Design Engineering Conference Proceedings*, pages 481–495, 1984.

[JI92] E. R. Jessup and I. C. F. Ipsen. Improving the accuracy of inverse iteration. *SIAM Journal on Scientific and Statistical Computation*, 13(2):550–572, 1992.

[Joh87] J.K. Johnstone. *The Sorting of points along an algebraic curve.* PhD thesis, Cornell University, Department of Computer Science, 1987.

[Kaj82] J. Kajiya. Ray tracing parametric patches. *Computer Graphics*, 16(3):245–254, 1982.

[Kal82] Y.E. Kalay. Modeling polyhedral solids bounded by multi-curved parametric surfaces. *ACM IEEE Nineteenth Design Automation Conference Proceedings*, pages 501–507, 1982.

[KGI84] F. Kimura and Geomap-III. Designing solids with free-form surfaces. *IEEE Computer Graphics and Applications*, 4:58–72, 1984.

[Kim90] Deok-Soo Kim. *Cones on Bezier Curves and Surfaces.* PhD thesis, University of Michigan, Ann Arbor, 1990.

[KKM97] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate b-rep generation of low degree sculptured solids using exact arithmetic. In *ACM/SIGGRAPH Symposium on Solid Modeling*, pages 42–55, 1997.

[KKMN95] S. Kumar, S. Krishnan, D. Manocha, and A. Narkhede. Representation and display of complex csg models. Technical Report TR95-019, Department of Computer Science, University of North Carolina, 1995.

[KM83] P.A. Koparkar and S. P. Mudur. A new class of algorithms for the processing of parametric curves. *Computer-Aided Design*, 15(1):41–45, 1983.

[KM97] S. Krishnan and D. Manocha. An efficient surface intersection algorithm based on the lower dimensional formulation. *ACM Transactions on Graphics*, 16(1):74–106, 1997.

[KPP90] G.A. Kriezis, P.V. Prakash, and N.M. Patrikalakis. Method for intersecting algebraic surfaces with rational polynomial patches. *Computer-Aided Design*, 22(10):645–654, 1990.

[KPW90] G.A. Kriezis, N.M. Patrikalakis, and F.E. Wolter. Topological and differential equation methods for surface intersections. *Computer-Aided Design*, 24(1):41–55, 1990.

[KS88] S. Katz and T.W. Sederberg. Genus of the intersection curve of two rational surface patches. *Computer Aided Geometric Design*, 5, 1988.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[Las86] D. Lasser. Intersection of parametric surfaces in the bernstein-bezier representation. *Computer-Aided Design*, 18(4):186–192, 1986.

[LR80] J.M. Lane and R.F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):150–159, 1980.

[LTH86] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes. Constructive solid geometry for polyhedral objects. *ACM Computer Graphics*, 20:161–170, 1986.

[Mac02] F.S. Macaulay. On some formula in elimination. *Proceedings of London Mathematical Society*, 1(33):3–27, May 1902.

[Man86] M. Mantyla. Boolean operations of 2-manifolds through vertex neighborhood classification. *ACM Transactions on Graphics*, 5:1–29, 1986.

[Man88] M. Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, Maryland, 1988.

[Man92] D. Manocha. *Algebraic and Numeric Techniques for Modeling and Robotics.* PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[MC91] D. Manocha and J.F. Canny. A new approach for surface intersection. *International Journal of Computational Geometry and Applications*, 1(4):491–516, 1991. Special issue on Solid Modeling.

[MC93] D. Manocha and J.F. Canny. Multipolynomial resultant algorithms. *Journal of Symbolic Computation*, 15(2):99–122, 1993.

[MD94] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves i: simple intersections. *ACM Transactions on Graphics*, 13(1):73–100, 1994.

[Mea84] D. J. Meagher. The solids engine: a processor for interactive solid modeling. In *Proceedings of Nicograph*, 1984.

[Men92] J. Menon. *Constructive Shell Representations for Free-form Surfaces and Solids.* PhD thesis, Dept. of Computer Science, Cornell University, 1992.

[Mil92] P. S. Milne. On the solutions of a set of polynomial equations. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 89–102, 1992.

[ML95] Y. Ma and R. C. Luo. Topological method for loop detection of surface intersection problems. *Computer-Aided Design*, 27(11):811–820, 1995.

[MP93] T. Maekawa and N. M. Patrikalakis. Computation of singularities and intersections of offsets of planar curves. *Computer Aided Geometric Design*, 10, 1993.

[MR86] M. Mantyla and M. Ranta. Interactive solid modeling in hutdesign. In *Proceedings of Computer Graphics, Tokyo*, 1986.

[MT83] M. Mantyla and M. Tamminen. Localized set operations for solid modeling. In *Computer Graphics*, volume 17, pages 279–288, 1983.

[Nat90] B.K. Natarajan. On computing the intersection of b-splines. In *ACM Symposium on Computationl Geometry*, pages 157–167, 1990.

[NM95] A. Narkhede and D. Manocha. Fast polygon triangulation based on seidel's algorithm. In A. Paeth, editor, *Graphics Gems V*, pages 394–397, Academic Press, 1995.

[NSK90] T. Nishita, T.W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics*, 24(4):337–345, 1990.

[OKK73] N. Okino, Y. Kakazu, and H. Kubo. *TIPS-1: Technical Information Processing System for Computer Aided Design and Manufacturing.* Computer Languages for Numerical Control, J. Hatvany, ed., North Holland, Amsterdam, 1973.

[O'N66] B. O'Neill. *Elementary Differential Geometry.* Academic Press, London, UK, 1966.

[Pat93] N.M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95, 1993.

[PFTV90] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1990.

[Pie89] L. Piegl. Geometric method of intersecting natural quadrics represented in trimmed surface form. *Computer-Aided Design*, 21(4):201–212, 1989.

[PK92] J. Ponce and D.J. Kriegman. Elimination theory and computer vision: Recognition and positioning of curved 3d objects from range, intensity, or contours. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 123–146, 1992.

[Pra86] M.J. Pratt. Surface/surface intersection problems. In J.A. Gregory, editor, *The Mathematics of Surfaces II*, pages 117–142, Oxford, 1986. Claredon Press.

[PRS86] A. Paoluzzi, M. Ramella, and A. Santarelli. Un modellatori geometrico su rappresentazioni triango-alate. Technical Report Rept. TR 13.86, Department of Inf. and Systems, University of Rome, Italy, 1986.

[RMS92] J. Rossignac, A. Megahed, and B.D. Schneider. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

[RR87] J.R. Rossignac and A.A.G. Requicha. Piecewise-circular curves for geometric modeling. *IBM Journal of Research and Development*, 31(3):296–313, 1987.

[RR92] A.A.G. Requicha and J.R. Rossignac. Solid modeling and beyond. *IEEE Computer Graphics and Applications*, pages 31–44, September 1992.

[RV82] A.A.G. Requicha and H.B. Voelcker. Solid modeling: A historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24, March 1982.

[RV85] A.A.G. Requicha and H.B. Voelcker. Boolean operations in solid modeling: boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1), 1985.

[RV89] J. Rossignac and H.B. Voelcker. Active zones in csg for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithm. *ACM Transactions on Graphics*, 8(1):51–87, 1989.

[Sal85] G. Salmon. *Lessons Introductory to the Modern Higher Algebra*. G.E. Stechert & Co., New York, 1885.

[Sar83] R F Sarraga. Algebraic methods for intersection. *Computer Vision, Graphics and Image Processing*, 22:222–238, 1983.

[Sat91] T. Satoh. Boolean operations on sets using surface data. In *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 119–127, 1991.

[SB93] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1993.

[Sed83] T.W. Sederberg. *Implicit and Parametric Curves and Surfaces*. PhD thesis, Purdue University, 1983.

[Sed89] T.W. Sederberg. Algorithms for algebraic curve intersection. *Computer-Aided Design*, 21(9):547–555, 1989.

[Seg90] M. Segal. Using tolerances to guarantee valid polyhedral modeling results. In *Proceedings of ACM Siggraph*, pages 105–114, 1990.

[Sei90a] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 211–215, 1990.

[Sei90b] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[Sei91] R. Seidel. A simple and fast randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry Theory & Applications*, 1(1):51–64, 1991.

[SI89] K. Sugihara and M. Iri. A solid modeling system free from topological inconsistencis. *J. Inf. Proc., Inf. Proc. Soc. of Japan*, 12(4):380–393, 1989.

[SKW85] P. Sinha, E. Klassen, and K.K. Wang. Exploiting topological and geometric properties for selective subdivision. In *ACM Symposium on Computationl Geometry*, pages 39–45, 1985.

[SM88] T.W. Sederberg and R.J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5:161–171, 1988.

[SN90] T.W. Sederberg and T. Nishita. Curve intersection using bézier clipping. *Computer-Aided Design*, 22:538–549, 1990.

[SN91] T.W. Sederberg and T. Nishita. Geometric hermite approximation of surface patch intersection curves. *Computer Aided Geometric Design*, 8:97–114, 1991.

[Sny92] J. Snyder. Interval arithmetic for computer graphics. In *Proceedings of ACM Siggraph*, pages 121–130, 1992.

[SP86] T.W. Sederberg and S.R. Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, 1986.

[SS83] J. T. Schwartz and M. Sharir. On the piano movers probelem ii, general techniques for computing topological properties of real algebraic manifolds. *Advances of Applied Maths*, 4:298–351, 1983.

[SWZ89] T.W. Sederberg, S. White, and A. Zundel. Fat arcs: A bounding region with cubic convergence. *Computer Aided Geometric Design*, 6:205–218, 1989.

[Taw91] M. S. Tawfik. An efficient algorithm for csg to b-rep conversion. In *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 99–108, 1991.

[THS89] Sederberg T.W, Christiansen H.N, and Katz S. An improved test for closed loops in surface intersections. *Computer-Aided Design*, 21(8):505–508, 1989.

[Voe74] H. B. Voelcker. An introduction to padl: Characteristics, status, and rationale. Technical Report Research Memo. #22, University of Rochester, 1974. Production Automation Project.

[VP84] T. Varady and M.J. Pratt. Design techniques for the definition of solid objects with free-form geometry. *Computer Aided Geometric Design*, 1(3):207–225, 1984.

[Wal50] R.J. Walker. *Algebraic Curves*. Princeton University Press, New Jersey, 1950.

[Wei85] Kevin J. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.

[Wei86] Kevin J. Weiler. *Topological Structures for Solid Modeling*. PhD thesis, Computer and Systems Engineering, Rensselaer Polytechnic Institute, 1986.

[Wes80] M. Wesley. A geometric modeling system for automated mechanical assembly. *IBM Journal of Research and Development 24*, pages 64–74, 1980.

[Whi94] S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, 1994.

[Wil59] J.H. Wilkinson. The evaluation of the zeros of ill–conditioned polynomials. parts i and ii. *Numer. Math.*, 1:150–166 and 167–180, 1959.

[Wil65] J.H. Wilkinson. *The algebraic eigenvalue problem*. Oxford University Press, Oxford, 1965.

[YA93] J.H. Yang and J. Anderson. Fast, scalable synchronization with minimal hardware support. In *ACM symposium on Principles of Distributed Computing*, pages 171–182, 1993.

[ZS93] A. Zundel and T. Sederberg. Using pyramidal surfaces to detect and isolate surface/surface intersections. In *SIAM Conference on Geometric Design*, Tempe, AZ, 1993.