

96-DETC/DFM-1296

COMPUTING BOOLEAN COMBINATIONS OF SOLIDS COMPOSED OF FREE-FORM SURFACES

Shankar Krishnan

Dept. of Computer Science
Univ. of North Carolina
CB#3175 Sitterson Hall
Chapel Hill, North Carolina 27599-3175
USA
e-mail: krishnas@cs.unc.edu

Dinesh Manocha¹

Dept. of Computer Science
Univ. of North Carolina
CB#3175 Sitterson Hall
Chapel Hill, North Carolina 27599-3175
USA
e-mail: manocha@cs.unc.edu

ABSTRACT

We present efficient and accurate algorithms for Boolean combinations of solids composed of sculptured models. The surface of each solid is represented as a collection of trimmed and untrimmed spline surfaces and a connectivity graph. Based on algorithms for trapezoidation of polygons, partitioning of polygons using polygonal chains, surface intersection of high degree spline surfaces and ray-shooting, the boundaries of the resulting solids and its connectivity graph after the Boolean operation are computed. The algorithm has been implemented and its performance measured on a number of industrial models.

INTRODUCTION

The field of solid modeling deals with design and representation of physical objects. The two major representation schemata used in solid modeling are constructive solid geometry (CSG) and boundary representations (B-rep). Both these representations have different inherent strengths and weaknesses, and for most applications both are desired. Currently, most solid modelers are able to support solids composed of polyhedral models and quadric surfaces (like spheres, cylinders etc.) and their Boolean combinations. The field of geometric modeling has been developed to model classes of piecewise surfaces based on particular conditions of shape and smoothness. Such models are referred to as *sculptured models*. Over the last few years, modeling using free-form surfaces has become quite indispensable 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

¹Supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, NSF Grant CCR-9319957, NSF Grant CCR-9625217, ONR Contract N00014-94-1-0738, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Center for Computer Graphics and Scientific Visualization

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. As a result, some solid modelers use polyhedral approximations to these surfaces and apply existing algorithms to design and manipulate these polyhedral objects. Not only does this lead to data proliferation, but the resulting algorithms are inefficient and inaccurate.

In this paper we restrict ourselves to solid models composed of piecewise parametric or spline surfaces. The techniques presented can also be generalized to all algebraic surfaces. 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 or Bézier surfaces [KS88, SN91, Nat90, Hoh92, MC91, KPW90, BHHL88, BK90, KM94]. However, the algebraic degree of the resulting curve can typically be very high (up to 324 for a pair of bicubic Bézier surfaces) and the *genus* is also non-zero [KS88]. Hence, it is very difficult to represent the intersection curve analytically and the current methods are aimed at computing approximations to the intersection curve. The main techniques for approximation can be classified into subdivision, analytic and marching methods. Currently, techniques based on marching methods combined with loop detection, handling singularities and component jumping are able to robustly compute the intersections between a pair of surfaces, for non-degenerate intersections [Hoh92, KM94, SN91].

In this paper, we present algorithms for representation and computation of Boolean operations on CSG models. Every CSG object is built from a set of primitive objects which are of a simpler structure, and we assume that each of these primitives is an *oriented solid* representable as a collection of parametric surface patches. An example of a CSG tree is shown in fig. 1. The class of models that can be used include all polyhedral models, primitive solids like cones, spheres, cylinders, generalized prisms and pyramids, and tori. Each intermediate and final solid in the CSG hierarchy is represented as a collection of *trimmed* surface patches. Every *trimmed* patch contains a closed *trimming* curve on its domain which unambiguously determines which part of the surface is to be retained. We also create a *connectivity* graph that maintains the connectivity information between the various surfaces composing the solid (in other words, we maintain both geometric and topological information about the solid).

Overview of Algorithm: The entire algorithm proceeds in two steps. Initially, the complete intersection curve between the two solids (at each level in the CSG tree) is determined. If the set operation on the two solids is well-defined, the intersection curve partitions the surface of the solid. The second step of the algorithm uses the graph struc-

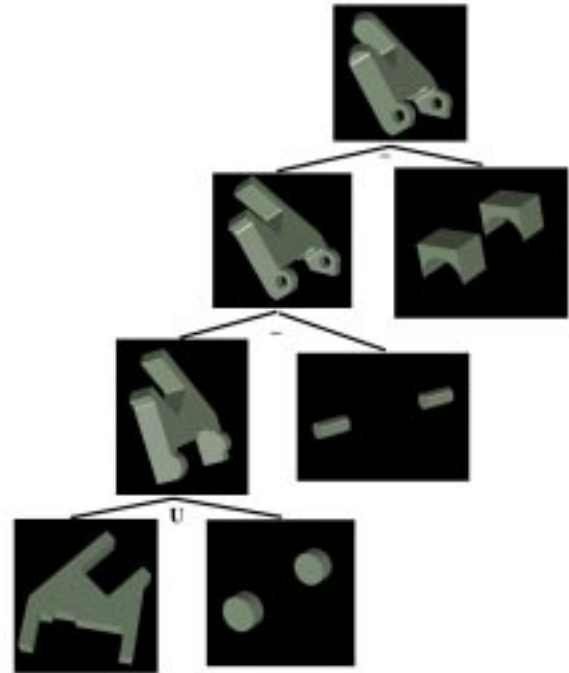


Figure 1. Part of a CSG tree of the roller model

ture of each solid, and depending on the set operation, computes the new solid and its associated graph structure. The overall algorithm makes use of a number of recently developed algorithms for linear programming, trapezoidation of polygons, partition of polygonal domains and ray-shooting to compute the resulting solid efficiently.

The rest of the paper is organized in the following manner. Brief descriptions of some of the algebraic, geometric and numeric algorithms used by our algorithm are presented in section 2. Section 3 discusses the various representation issues and data structures that are needed. A brief description of our new surface intersection algorithm is given in section 4. The entire algorithm is sketched for one Boolean operation in section 5. Section 6 describes the implementation of our algorithm on certain industrial models.

BACKGROUND

The CSG to B-rep conversion algorithm is based on a number of geometric, algebraic and numeric algorithms. These include trapezoidation of simple polygons, partitioning a simple polygon using non-intersecting polygonal chains, ray-surface intersection during *component classification* and inverse power iterations for curve tracing during the computation of the intersection curve of two parametric

surfaces. This section briefly discusses these algorithms.

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 line segments in 2D. The expected time complexity of this algorithm is $O(n \log^* n)$. This algorithm can be used for fast polygon triangulation, as and 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.

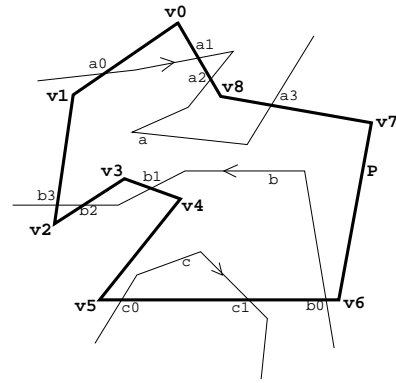


Figure 2. Partitioning a polygon with polygonal chains

- **Trapezoidation of the polygon:** Let S be a set of non-horizontal, non-intersecting line segments of the polygon. The randomized algorithm is used to create the trapezoidal decomposition of the $X - Y$ plane arising due to the segments of set S . This is done by taking a random ordering s_1, \dots, 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, \dots, 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 the two vertices of the original polygon lie on the same side. 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.

Partitioning a Simple Polygon using Non-Intersecting Chains

Given a simple polygon P and a number of non-intersecting polygonal chains C , the output of the algorithm is a list of disjoint regions/partitions of the polygon due to the chains. We make no assumptions on C except that each

chain $c_i \in C$ in itself should partition P . Let us assume for the sake of simplicity that the chain does not form a loop inside the polygon. We handle this case as a special case in our algorithm, and hence details are omitted. The main idea in this algorithm is the fact that since the chains are non-intersecting, each of the partitioned region starts and ends at intersection points (with the polygon) of the same chain. Fig. 2 shows a simple polygon P and three non-intersecting chains a , b and c . Since the vertices of each chain are given in a specific order, we shall assume that to be the direction of the chain (see fig.). The algorithm works in two steps.

- Find all the intersection points of each chain with the polygon and number them according to the order in which they occur. This problem can be solved in time $O(N \log^2 N)$ [CEGS94], where N is the total number of segments in the polygon and chains. We associate three fields with each intersection point - the chain corresponding to each intersection point (*type*), the number of the intersection point within the chain (*rank*), and whether the chain was coming in or going out of the polygon at this point (*in_or_out*). For example, the intersection point a_1 in the figure has *type* = a , *rank* = 1 and *in_or_out* = *out* as its three fields. The *in_or_out* field is actually not necessary because *rank* has that information; however, we use it for ease of description.
- Given all the intersections, we traverse the polygon in some order starting from an arbitrary vertex. We use a stack as a data structure to compute the partitions. Let us assume that we start traversing the polygon from vertex v_0 in an anticlockwise order for the example given in the figure. Given this traversal, we can order the intersection points around the polygon. In the example, the order would be $a_0, b_3, b_2, b_1, c_0, c_1, b_0, a_3, a_2, a_1$. As we proceed from vertex v_i to v_{i+1} in the

polygon, we retrieve all the intersection points of the various chains with the edge (v_i, v_{i+1}) in order. If q is an intersection point, and let p be the point on the top of the stack. To determine if q is *pushed* or p is *popped*, the following condition is checked.

```

if (q.type ≠ p.type)
  push(q);
if ((q.rank - p.rank == 1) && (q.in_or_out == out)
    && (p.in_or_out == in))
  pop(p);
if ((p.rank - q.rank == 1) && (p.in_or_out == out)
    && (q.in_or_out == in))
  pop(p);
push(q);

```

If p is popped, then p and q form a partition of the polygon. The corresponding partition is read out and the chain of vertices between p and q is appended to the vertex (this chain will be a part of the partition that involves this vertex) currently on top of the stack (the one that was below p). Some details about reading out the proper partitions once a vertex is popped are omitted. After traversing the polygon completely once, we would have obtained all the partitions.

Ray-Surface Intersection

Computing the intersection of a ray with a surface is needed to answer point classification queries for a solid. Basically we shoot a ray from the point in an arbitrary direction and determine the number of intersections of the ray with the solid. If the number of intersections is odd, the point is inside, otherwise it is outside. We use some recent algorithms for these intersections based on eigenvalue computations [MD94].

Given a surface $\mathbf{F}(s, t)$, we compute its implicit representation using resultant methods [Dix08] and obtain a matrix formulation $\mathbf{M}(x, y, z, w)$. We substitute the parametrization of the curve, say $\mathbf{G}(u) = (\overline{X}(u), \overline{Y}(u), \overline{Z}(u), \overline{W}(u))$ of degree d , and obtain a univariate matrix polynomial $\mathbf{M}(u)$. The problem of intersection computation reduces to computing the roots of the non-linear matrix polynomial $\mathbf{M}(u)$. The polynomial which are in Bernstein basis can be converted to the power basis by the transformation $\overline{u} = \frac{u}{1-u}$. The resulting matrix $\mathbf{M}(\overline{u})$ can be represented as

$$\mathbf{M}(\overline{u}) = \overline{u}^d M_d + \overline{u}^{d-1} M_{d-1} + \dots + \overline{u} M_1 + M_0. \quad (1)$$

where M_i 's are matrices of order $2mn$ with numeric entries. Furthermore, the roots of the matrix polynomial, $\mathbf{M}(u)$,

have one-to-one correspondence with the eigendecomposition of

$$C = \begin{bmatrix} 0 & I_n & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & I_n \\ -\overline{M}_0 & -\overline{M}_1 & -\overline{M}_2 & \dots & -\overline{M}_{d-1} \end{bmatrix} \quad (2)$$

where $\overline{M}_i = \mathbf{M}_d^{-1} \mathbf{M}_i$ [GLR82]. In case \mathbf{M}_d is singular or ill-conditioned, the intersection problem is reduced to a generalized eigenvalue problem [MD94]. Algorithms to compute all the eigenvalues are based on QR orthogonal transformations [GL89]. They compute all the real as well as complex eigenvalues. Algorithms to compute eigenvalues in a subset of the real or complex domain are presented in [MD94].

Inverse Power Iterations for Curve Tracing

We use marching methods to trace the intersection curves. At each iterations, we pose the problem as an eigenvalue problem and use local methods to compute points on the curve. Power iteration is a fundamental local technique used to compute eigenvalues and eigenvectors of a matrix. Given a diagonalizable matrix, \mathbf{A} , we can find \mathbf{X} ($= [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]$) such that $\mathbf{X}^{-1} \mathbf{A} \mathbf{X} = \text{Diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ and $|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$. Given a unit vector \mathbf{q}_0 , the *power method* produces a sequence of vector \mathbf{q}_k as follows:

$$\mathbf{z}_k = \mathbf{A} \mathbf{q}_{k-1}; \quad \mathbf{q}_k = \mathbf{z}_k / \|\mathbf{z}_k\|_\infty; \quad s_k = \mathbf{q}_k^T \mathbf{A} \mathbf{q}_k;$$

where $\|\mathbf{z}_k\|_\infty$ refers to the infinity norm of the vector \mathbf{z}_k . s_k converges to the largest eigenvalue λ_1 and \mathbf{q}_k converges to the corresponding eigenvector \mathbf{x}_1 .

The basic idea of power iterations can be used and modified to obtain the eigenvalue of a matrix A that is closest to a given guess s . It actually corresponds to the largest eigenvalue of the matrix $(\mathbf{A} - s\mathbf{I})^{-1}$. Instead of computing the inverse explicitly (which can be numerically unstable), we use *inverse power* iterations. Given an initial unit vector \mathbf{q}_0 , we generate a sequence of vectors \mathbf{q}_k as

$$\text{Solve } (\mathbf{A} - s\mathbf{I}) \mathbf{z}_k = \mathbf{q}_{k-1}; \quad \mathbf{q}_k = \mathbf{z}_k / \|\mathbf{z}_k\|_\infty; \quad s_k = \mathbf{q}_k^T \mathbf{A} \mathbf{q}_k;$$

We use inverse power iterations to trace curves. We formulate the curve as the singular set of a matrix polynomial and reduce it to an eigenvalue problem. Given a point on the curve, we approximate the next point on the curve by taking a small stepsize in a direction determined by the

local geometry of the curve. We use this point as our guess and use inverse power iterations to converge back to the curve. Details of the curve tracing algorithm using inverse power iterations are presented in [KM94].

REPRESENTATION OF A SOLID

Every solid is represented as a set of *trimmed* parametric surface patches which define the solid boundary. We represent each patch as a tensor-product Bézier patch $\mathbf{F}(s, t)$ given by

$$\mathbf{F}(s, t) = \frac{\sum_{i=0}^m \sum_{j=0}^n \mathbf{V}_{ij} B_i^m(s) B_j^n(t)}{\sum_{i=0}^m \sum_{j=0}^n w_{ij} B_i^m(s) B_j^n(t)},$$

where $\mathbf{V}_{ij} = \langle x_{ij}, y_{ij}, z_{ij} \rangle$ are the *control points* of the patch and $B_i^m(s) = \binom{m}{i} s^i (1-s)^{m-i}$ is the Bernstein polynomial. This can be viewed as a mapping from the unit square $0 \leq s, t \leq 1$ in the (s, t) -plane (also called the *domain* of the Bézier patch) to three space (\mathfrak{R}^3).

Surface patches of different solids may intersect during set operations. The intersection curves introduced in this process are used to partition the patches into different regions. Only a few of them are retained after the boolean operation. Each such region is referred to as a *trimmed patch*. Each trimmed patch is represented using the original parametrization along with a trimming curve defined on the domain. The intersection curves, which actually form the trimming boundaries of most of the trimmed patches, are generally very high degree curves and are not parameterizable (using rational polynomial functions) exactly in general. Therefore, most of the existing algorithms approximate these intersection curves as piecewise linear chains. The trimming region thus obtained is a polygon on the domain of the patch. However, this approximation could affect the accuracy (and sometimes, even correctness) of the solid models generated by the algorithm. In addition to the piecewise linear representation, we also maintain an accurate analytic representation of the intersection curve in the domain space of the patch (as the singular set of a bivariate matrix polynomial).

Every solid S is associated with a *connectivity graph*, (S) which is an undirected graph describing the neighborhood information between the patches constituting the solid. Let F_1, F_2, \dots, F_n be the patches belonging to S ; every patch F_i is associated with a vertex v_i in (S) . An edge exists between v_i and v_j if and only if F_i and F_j are *neighboring patches* (i.e., F_i and F_j share some portion of their boundary in \mathfrak{R}^3). In the domain, every trimming polygon (of patch F_i) is divided into several components such that each component is the common boundary between F_i and

a neighboring patch F_j . Fig.7(a), (b), (d) and (e) show the connectivity graphs for a cube and a cylinder. This adjacency information is often referred to as the *topology* of the solid model. The actual geometric surface and curve descriptions are referred to as the *geometry* of the solid model. The topological information serves as the framework into which the geometric information is placed.

COMPUTING INTERSECTION CURVE OF TWO SURFACE PATCHES

Evaluating the intersection of parametric and algebraic surfaces is a recurring operation in computer graphics, geometric and solid modeling, and computer-aided design. An efficient surface intersection algorithm that is numerically reliable, accepts general surface models, and operates without human supervision is critical to many solid modeling applications. The intersection of two surfaces can be complicated in general, with a number of closed loops and self-intersections (singularities). A good surface intersection algorithm, should, in theory, be able to detect all such features of the intersection curve and trace them correctly in an efficient manner. It is desirable to design an algorithm that is as robust and efficient as possible, minimizing the potential for missed loops and other curve features which plague many of the present schemes.

Previous Work: There is a significant body of literature addressing the surface intersection problem. Some recent surveys include [Pat93, Pra86, Hof89]. Surface intersection algorithms can be broadly classified into four major categories: Subdivision methods [LR80], interval arithmetic [Sny92], lattice evaluation [RR87], analytic [Sed83, Sar83] and marching methods [Far86, BHHL88, KPW90]. More recently, techniques have been designed that combine features of different categories and are generally referred to as *hybrid methods*.

Our approach uses a *combination of analytic and marching methods*. The components of an intersection curve consist of boundary segments and closed loops (see fig. 4). Start points on the boundary segments are obtained by curve-surface intersections [SN91]. Many techniques have appeared over the last few years to detect closed loops on the intersection curve [Hoh91, KPP90, ZS93] based on bounds on *gauss maps* and subdivision of surfaces. These algorithms work very well to isolate cases with no loops only if the surfaces are relatively flat. However, in the presence of small loops or singularities, they tend to become slow. In terms of tracing, 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



Figure 3. Intersection of the Utah teapots and the intersection curve

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].

Main Result: We briefly describe algorithms for efficient computation of surface intersection. Depending on the application requirement, the algorithm can be fine-tuned to provide better robustness guarantees at the expense of execution time. The algorithm has been implemented using double-precision arithmetic. Given combinations of NURBS surfaces, the algorithm decomposes them into Bézier patches. It uses spatial techniques like bounding-box tests and linear programming to reduce the number of pairwise intersections. Given a pair of Bézier and/or algebraic surfaces, it finds a starting point on each component of the intersection curve, decomposes the domain, marches along the curve choosing appropriate step sizes to prevent component jumping, finds the singularities and all branches at each singularity. It has been applied to the intersection of high-degree parametric surfaces and performs well. It typically takes about 8 – 10 seconds to compute intersection curves of about 100 pairs of bicubic Bézier surfaces on a SGI Onyx workstation.

Overview : Our algorithm formulates the intersection problem algebraically, computes the projection of the intersection curve as an algebraic plane curve and evaluates it. This is based on a classical theorem from algebraic geometry which states that every algebraic space curve, after suitable linear transformation, can be mapped birationally to a plane curve. The projection is represented as the singular set of a bivariate matrix polynomial and the algorithm uses matrix computations to trace the curve. Tracing is a geometric operation and evaluating the curve in the plane reduces its *geometric complexity*. The loops of the intersection curve are characterized algebraically and computed using curve-surface intersections followed by complex tracing. The tracing algorithm employs *domain decomposition* and the use of *inverse power iterations* (see section 2.4). Domain decomposition provides a natural procedure to compute the step size and prevent *component jumping*. This is a major advantage of our tracing algorithm since incorrect

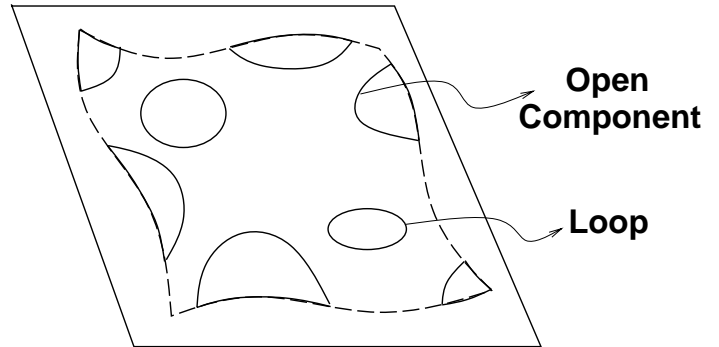


Figure 4. Wireframe of two surfaces intersecting in multiple components

connectivity of the various curve components was considered a drawback with traditional marching methods. We also show that we need not require the precomputation of singular points before the tracing step. Our algorithm can detect the presence of singularities during the tracing and decomposition phase. The accuracy of the result is improved by performing minimization on a distance-based energy function. To speed up the algorithm we also use existing simple geometric tests to isolate cases consisting of no loops or singularities. Fig.3 shows the intersections of two teapots (Utah teapots) and their intersection curve. Each teapot is composed of 32 Bézier patches. In all the figures displaying intersection curves, we have used spheres of a small constant radius to represent the curve clearly as traced out by the algorithm.

Intersection Computation

The intersection curve in the domain of one of the surfaces is defined as the *singular set* of the matrix polynomial [KM94]. The main advantages of this matrix representation of the intersection curve are in terms of 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 are well known [GL89] and good implementations are available as part of numerical libraries like EISPACK and LAPACK.

Start Points: The marching algorithm needs start points on each component of the intersection curve. 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 fig.4. These are points on the plane curve where one of the parameter values of the Bézier surface is 0 or 1 and can be computed using eigenvalue methods [MD94]. The other components

are closed loops. The start points on the open components are computed using curve-surface intersections.

The difficulty in identifying start points on closed components lies in the fact that loops (see fig. 5(a)) have no such simple characterization as the one for open components. However, we show that we can use a simple algebraic property that would guide us to some point on every loop.

Loop Characterization: We initially use Hohmeyer's algorithm based on Gauss maps and linear programming to check for absence of loops [Hoh91]. If the loop detection criterion is not satisfied, we use the algorithm presented below for computing start points on the loops.

The intersection curve 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, the intersection curve is defined as a continuous set consisting of real and complex components. We state here without proof the following lemma which we use to detect starting points on all loops.

Lemma 1. *If the intersection 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.*

Proof: See [KM94]

The domain of the intersection curve in the complex space is shown in fig. 5(b). The third axis corresponds to the imaginary components of v . It represents a continuous component of the intersection curve. The *dark* curve is the intersection curve in the complex space and the *white* curve is the part of the curve that lies in the real plane.

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, *solidA* and *solidB*. We shall denote the resulting solid as *solidAB*. Let the number of patches in *solidA* be m and those in *solidB* 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 *solidAB* is to find the curve of intersection between the two solids. Since each solid is composed of a set of 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 the Bézier patches have the *convex hull*

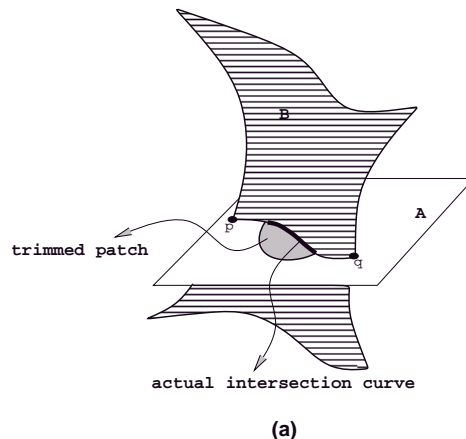


Figure 6. Intersection of trimmed surfaces

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. The next stage of pruning uses linear programming. We formulate the 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 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 $2(d_s + 1)(d_t + 1)$. The running time for the entire process, thus, is $O(mnd_s d_t)$. Typically, d_s and d_t are much smaller than m and n . Therefore, the running time is $O(mn)$. 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 [Sei90].

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 the previous section to compute the complete intersection curve of the two patches (ignoring the trimming curves). The intersection curve so obtained from the algorithm is represented as a piecewise linear chain in parameter space. Fig.6 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 the

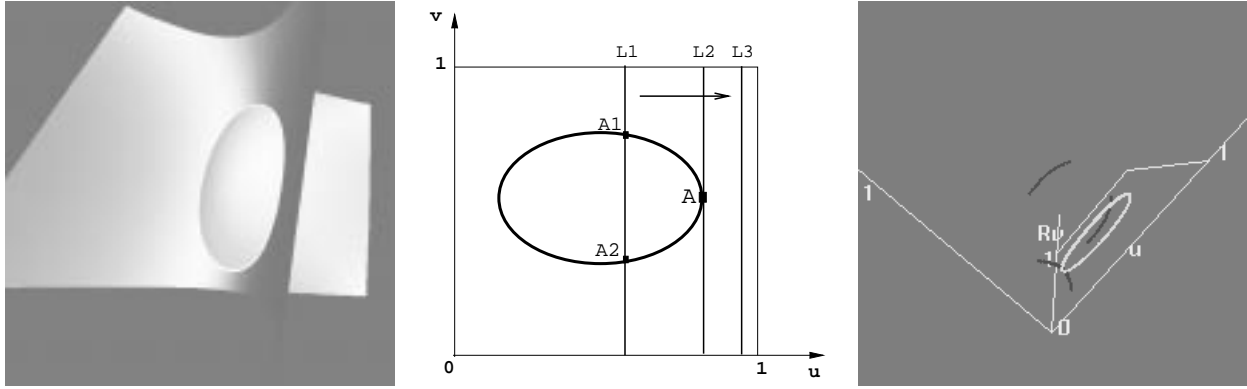


Figure 5. (a) Two surfaces intersecting in a loop (b) Characterization of loops based on complex tracing

figure. 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, 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 $O(\log n)$ point location queries [Sei91] (see section 2.1). The intersection points obtained by this process is only an approximation to the true intersection points. In order to refine this estimate, we use the analytic representation of the curves involved and perform local iterative methods. The accuracy of the results obtained by this two-step process is usually adequate for most models.

By applying this algorithm on all pairs of patches, we obtain a set of curves in the domain of every patch. Let us consider a single patch $F \in \text{solidA}$ and denote the set of intersection curves lying in F by C_F . Since each solid is closed and C^0 continuous, the intersection curve between them 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.

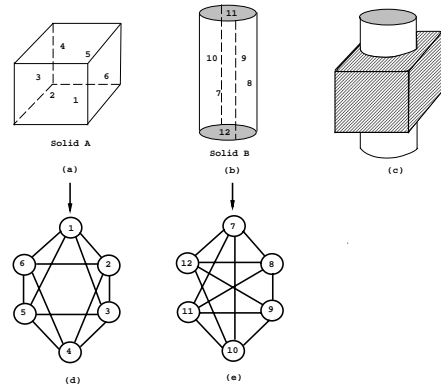


Figure 7. (a) A cube (solidA) (b) A cylinder (solidB) (c) Cylinder used to drill a hole right through the cube (d), (e) Connectivity graphs of the cube and cylinder

Computation of the New Solid

Fig.7 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 new solid and its graph. The set of all curves in a single surface partitions that surface (because solids are closed and compact). For example, in the figure, the closed curve on patch 4 of *solidA* partitions it into two regions 4a and 4b. We compute all the partitions using our *partition_polygon* routine (see section 2.2). Every curve $i \in C_F$ (we will refer to it as an *edge* of C_F) also induces an adjacency relationship between a patch in *solidA* and one in *solidB*. Therefore, any new edge in the connectivity graph of the resulting solid uniquely corresponds to such an intersection curve (any intersection curve $i \in C_F$ is

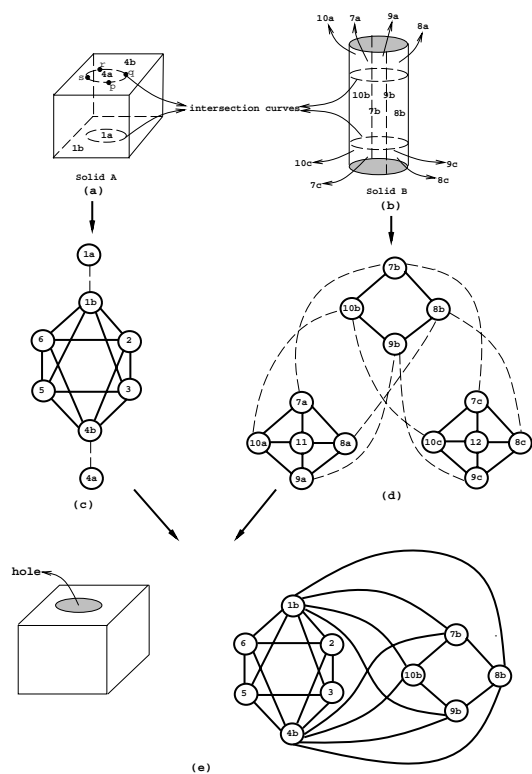


Figure 8. (a),(b) Intersection curves on cubes and cylinders (c),(d) Updated connectivity graphs based on partitions (e) Connectivity graph of final solid

actually represented as chains in the domain of each intersecting patch). This relationship is very important in the construction of the connectivity graph for the solid resulting from the set operation. We construct a table T that given a partitioned region of the patch $F \in \text{solid}A$ and an intersection curve $i \in C_F$, provides the two regions of the patch in $\text{solid}B$ (that intersected with F to give i). For example, regions $7a$ and $7b$ are adjacent to region $4a$ by the curve rs (see fig.8(a) and (b)).

The partitions induced by the intersection curves change the structure of the connectivity graph. Fig.8(c) and (d) show the updated graphs for the cube and the cylinder. We determine all the adjacencies to a partitioned region by traversing along its boundary. For example, region $7a$ has $8a$, $10a$ and 11 as its neighbors from edges that were part of some edge in 7 (marked as solid lines in fig.8(d)) and $7b$ as its neighbor from the edge corresponding to the intersection curve (marked as dotted lines in fig.8(d)). The edge from vertex $7a$ to $7b$ does not actually exist in the graph. It is shown here just to illustrate the adjacency within the patch. Now each connected component of this new graph has the property that all the patches corresponding to them are ei-

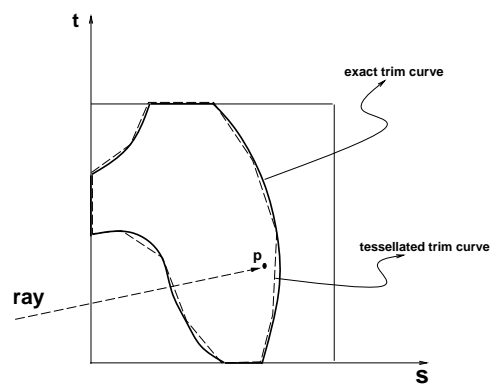


Figure 9. 2D Classification

ther completely inside the other solid or completely outside (*containment classification*).

We use ray-surface intersection (section 2.3) to find out if a point lies inside another solid or not. If the solid is closed and not self-intersecting (as we have assumed), then this query is answered by computing the number of intersections of a semi-infinite ray, emanating from that point, with the solid. If the number is odd, then the point is inside, otherwise it is outside the solid. The algorithm described in section 2.3 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 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. We use the trapezoidation of the trimming polygon (using Seidel's algorithm) to perform logarithmic time point location queries. However, 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 fig. 9. 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). If the distance of the point in question from the trimming boundary is within a specified tolerance limit, we perform *singular value decomposition* (SVD) to determine the sign of the matrix determinant. SVD is a very stable numerical algorithm, and hence its result is reliable. Containment classification is a highly time consuming operation, and its complexity is very sensitive to the degree of the surface patch and the trimming curve.

We now construct a new graph Π whose vertices are connected components of the old graph , , and there is an

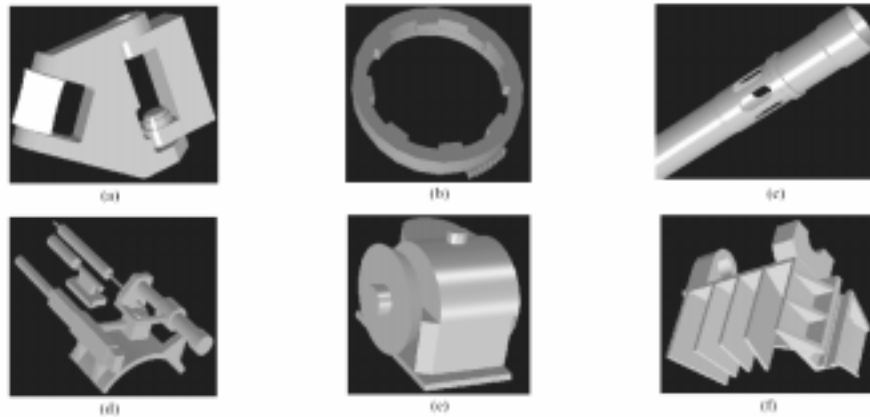


Figure 10. Application of our System to Different Solids

edge between two components if they lie on either side of the intersection curve. From the construction of Π it is clear that if one vertex lies inside the other solid, all its neighbors lie outside and vice-versa. Therefore, *performing exactly one inside-outside test is sufficient to determine the containment classification of each component*. We perform the classification test on one patch of a connected component of the graph (or vertex of Π) and propagate the information by a breadth-first search of Π . The particular set operation performed on the solids determines if a component is part of the new solid or not. The rules that govern this choice are

- *Union*: A component in *solidA* is part of the new solid if it lies **outside** *solidB*. A component in *solidB* is part of the new solid if it lies **outside** *solidA*.
- *Intersection*: A component in *solidA* is part of the new solid if it lies **inside** *solidB*. A component in *solidB* is part of the new solid if it lies **inside** *solidA*.
- *Difference*: A component in *solidA* is part of the new solid if it lies **outside** *solidB*. A component in *solidB* is part of the new solid if it lies **inside** *solidA*.

One way to avoid constructing the connectivity graph is by performing inside-outside tests on each portion of a patch (obtained after partitioning using the intersection curve) and determining its membership in the resulting solid (based on the operation performed). This is a much simpler but expensive algorithm because the classification test is very costly for curved geometries. Maintaining the graph structure allows us to perform just two such tests per set operation, significantly improving the running time.

Now that all the components that form the new solid are determined, the only thing left to do is to compute its connectivity graph. There is no change to the graph struc-

ture within a solid, *ie.*, there are no edges added or deleted within a component. The only new edges are those across components that are formed by intersection curves. We can find all the adjacency information from the table T that we had constructed. Fig.8(e) shows the new solid and its connectivity graph for the difference operation.

IMPLEMENTATION DETAILS

The system has been implemented on a high-end SGI-Onyx workstation with a single processor configuration. The system consists of two main parts

- the surface intersection code between a pair of parametric surfaces, and
- the supporting geometric routines for CSG.

The input to the intersection code are two parametric patches, and the output is a piecewise linear approximation of their intersection curve. The accuracy of this output directly depends on the *stepsize* used by the numerical tracing algorithm. Our implementation uses EISPACK routines (in Fortran) for various matrix computations. The intersection code is invoked at every level in the CSG tree. We found that the choice of stepsize was critical in preventing excessive accumulation of floating point error and avoiding data proliferation while performing the set operations on multi-level CSG trees. Large errors could result in inconsistencies in the topology of the final solid.

Geometric algorithms implemented on fixed precision arithmetic introduce different types of computational errors. Therefore different comparison tests have to be made using tolerances. Due to the wide range of input values encountered, setting a fixed tolerance is not sufficient. Currently our system normalizes the patches (scales down the control points) before applying the intersection algorithm,

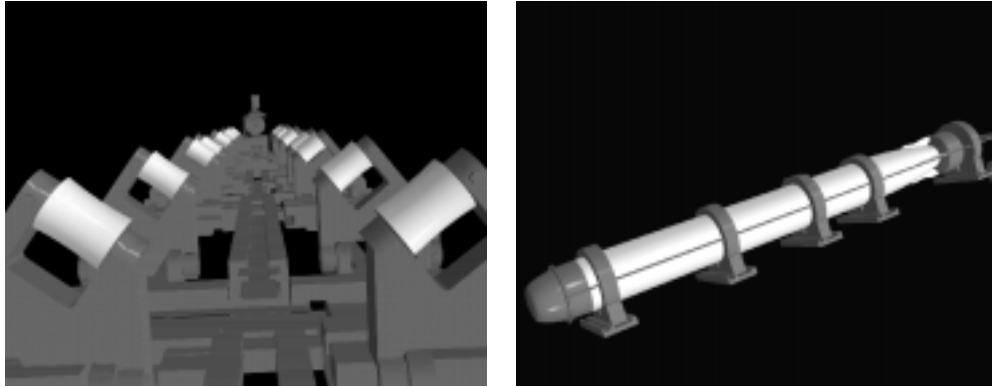


Figure 11. (a) A view of the pivot model (b) Torpedo model

but still the tolerance adjustments are not adequate. This is a potential cause for failure in geometric operations like detecting degenerate overlaps, merging intersection curves, polygon partitioning, etc.

The system has been tested and its performance measured on a number of models including an industrial model provided by Electric Boat Division, General Dynamics (see Fig. 10). Details for two of the models are given below (Fig. 11).

- *Pivot model*: The complete pivot model is generated from 168 CSG trees. The height of these trees vary from about 3 to 30. It consists of about 4400 trimmed patches (some of whose degree is 5×2).
- *Torpedo model*: This model consists of 71 CSG trees and its boundary representation is made up of about 650 trimmed patches. The surface of the torpedo consists of surfaces of revolution of degree as high as 11×2 .

ACKNOWLEDGEMENTS

We thank Mike Hohmeyer for his implementation of Seidel's linear programming algorithm. We are very grateful to G. Angelini, J. Boudreaux and K. Fast of Electric Boat for providing us with a CSG model of a submarine torpedo.

REFERENCES

[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.

[Cas87] M. S. Casale. Free-form solid modeling with

trimmed surface patches. *IEEE Computer Graphics and Applications*, pages 33–43, January 1987.

[CEGS94] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica II*, pages 116–132, 1994.

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

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

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

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

[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.

[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.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins Press, Baltimore, 1989.

[GLR82] I. Gohberg, P. Lancaster, and L. Rodman. *Matrix Polynomials*. Academic Press, New York, 1982.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling*.

- Morgan Kaufmann, San Mateo, California, 1989.
- [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.
- [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.
- [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.
- [KM94] S. Krishnan and D. Manocha. An efficient surface intersection algorithm based on the lower dimensional formulation. Technical Report TR94-062, Department of Computer Science, University of North Carolina, 1994.
- [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.
- [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.
- [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.
- [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.
- [Men92] J. Menon. *Constructive Shell Representations for Free-form Surfaces and Solids*. PhD thesis, Dept. of Computer Science, Cornell University, 1992.
- [Nat90] B.K. Natarajan. On computing the intersection of b-splines. In *ACM Symposium on Computational Geometry*, pages 157–167, 1990.
- [Pat93] N.M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95, 1993.
- [Pra86] M.J. Pratt. Surface/surface intersection problems. In J.A. Gregory, editor, *The Mathematics of Surfaces II*, pages 117–142, Oxford, 1986. Clarendon Press.
- [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.
- [Sar83] R F Sarraga. Algebraic methods for intersection. *Computer Vision, Graphics and Image Processing*, 22:222–238, 1983.
- [Sed83] T.W. Sederberg. *Implicit and Parametric Curves and Surfaces*. PhD thesis, Purdue University, 1983.
- [Sei90] 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.
- [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.
- [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.
- [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.
- [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.