Towards Robust Interval Solid Modeling of Curved Objects

by

Chun-Yi Hu

M.S. in Naval Architecture and Marine Engineering, M.I.T., July 1993
M.S. in Mechanical Engineering, M.I.T., July 1993
B.S. in Naval Architecture and Marine Engineering,
National Cheng-Kung University, R.O.C., June 1985

Submitted to the Department of Ocean Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .....
Department of Ocean Engineering
May 1, 1995

Certified by .....
Nicholas M. Patrikalakis
Associate Professor of Ocean Engineering
Thesis Supervisor

Accepted by .....
A. Douglas Carmichael
Chairman, Departmental Committee on Graduate Students

Towards Robust Interval Solid Modeling of Curved Objects

by

Chun-Yi Hu

Submitted to the Department of Ocean Engineering
on May 1, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Solid modelers have been used in computer aided design and manufacturing for more than one decade. However, current solid modelers based on Boundary Representation still encounter numerical instability and theoretical difficulties for ill-conditioned geometric computations involving intersections.

In this thesis, numerically robust geometric representations and algorithms for numerically robust geometric interrogations, including ill-conditioned geometric intersections are developed. Interval polynomial objects are proposed for robust geometrical representations. A robust algorithm (solver) for solving unbalanced non-linear polynomial equation systems is developed. Based on this solver, a robust unified algorithm for general geometric intersections, including overconstrained intersections, is developed. For the effective and robust detection of intersection curve loops, a direct algorithm is developed to determine collinear normal points and isolated tangential contact points of two surfaces. Theory and algorithms are developed for ill-conditioned intersections, such as tangential intersections of curves, overlapping of curves and surfaces, and overlapping of surfaces. The End Point Theorem is presented to verify that if two ideal Bézier patches tangentially intersect along an open curve, the curve must start from and end at boundaries of either of the two patches. This theorem is extended to general C^{∞} surface patches, and is followed by a corollary for surface overlapping.

An nD novel non-manifold data structure for interval polynomial objects (points, curves, and surfaces) is developed to permit Boundary Representation of interval objects. It separates the manifold from the non-manifold parts of the object, and categorizes nodes into six types for Boolean operations. This separation allows the effective use of a new point classification algorithm for non-manifold objects. Based on the robust interval geometrical representations and computations and data structures, algorithms for Boolean operations are developed first for 2D manifold curved regions and then extended to 3D manifold curved solids. These algorithms are further extended to 2D and 3D non-manifold curved objects resulting from Boolean operations. In order to represent the geometric objects, such as trimmed surfaces, resulting from Boolean operations, the Extreme Orientation Theorem is introduced to determine the orientation of a piecewise smooth simple planar closed curve by an extreme point and its derivative at that point. Finally, examples illustrate the robustness and efficiency of the algorithms for geometric intersections and Boolean operations.

Thesis Supervisor: Nicholas M. Patrikalakis
Title: Associate Professor of Ocean Engineering

Acknowledgements

This thesis is dedicated to my parents *Chi-Kuan Fu* and *Tze-Lian Show*, who are not only great architects in Taiwan, but also the great architects of my life. Also to my fiancée, Cherilyn Ho, who is always of the source of encouragement and support to me.

I like to thank the chairwoman, Mrs. Chia-Yun Lee, of the Bethany Children's Home. She encouraged and helped me to pursue graduate studies in America in many ways. I also thank all teachers in the Bethany Children's Home, especially Mrs. and Mr. Chia-Ping Ying, ex-chairwoman and her husband. They instructed and took care of me during my childhood. Many brothers and sisters in the Church of New Garden City in Taipei county, have been supporting me through their prayers and financial assistance. Some people support me financially; I know them only by their names or through a few meetings. I cannot thank any one of them enough for their goodwill and trust. I can only thank them by passing their deeds on to others like me.

I would like to express my gratitude to Professor N. M. Patrikalakis, my thesis supervisor, for his encouragement, his wisdom and especially his expertise, and Professor C. Chryssostomidis, Professor D. C. Gossard, Dr. T. Maekawa and Dr. X. Ye for serving on my thesis committee, their comments, guidance and patience.

I also thank Prof. Franz-Erich Wolter for providing me with useful ideas in the early stages of my work, Dr. Leonidas Bardis for his early implementation of the cell-tuple structure as a directed graph, Dr. Erik Brisson for useful discussions on data structure issues in the early stages of this work, Mr. Michael S. Drooker, Design Laboratory manager, for supplying me with a stable hardware environment, and Mr. Stephen L. Abrams for his programming assistance. Design Laboratory fellows such as Dr. Séamus T. Tuohy, Dr. Evan C. Sherbrooke, and Ms. Jingfang Zhou provided me with many opportunities for stimulating discussions and often encouraged me in my work.

This work was supported, in part, by the M.I.T. Sea Grant College Program, the Office of Naval Research and the National Science Foundation under grant numbers NA90AA-D-SG-424; N00014-91-1-1014 and N00014-94-1-1001; and DMI-9215411.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Solid modelers have been used in computer aided design and manufacturing for more than one decade. Boundary representation (B-rep) is the most commonly used scheme in solid modeling systems. However, current B-rep solid modelers still have the problem of lack of robustness. They encounter numerical instability and theoretical difficulties for geometrical computations. Those difficulties for geometrical computations can occur in Boolean operations, such as ill-conditioned geometrical intersections.

In current B-rep solid modelers, geometric entities (e.g., points, curves, and surfaces) are considered to be ideal mathematical objects. Nonetheless, ideal mathematical objects rarely exist in computer representations. Only a very small fraction of mathematical objects can be represented exactly in computer representations. For example, to represent the real line $\mathbf{R}$, floating point is used in the computer. However, floating point can only represent finitely many real (in fact rational) numbers, while any interval of the real line $\mathbf{R}$ has uncountable number of real numbers in it.

Typically, a geometric object is usually specified by the coordinates of its degree of freedom represented by floating point numbers, and processed based on floating point arithmetic. Unfortunately, as has been pointed out, numerical data represented by floating point numbers are generally only approximate, especially for irrational numbers. Therefore, strictly speaking, representations of geometric objects in floating point are inaccurate. This inaccuracy has the origin in the finite precision of floating point arithmetic. The *discrete* representations in the computer are used for *continuous* geometric entities. The consequences of this inaccuracy are (1) unreliability in geometrical computations and interrogations, such as Boolean operations; and (2) inconsistency between the geometry and topology of geometric objects.

Take the intersection problem for example. Like many other geometrical interrogation

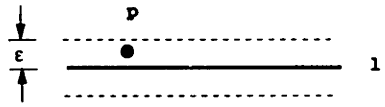


Figure 1-1: A point **p** is considered to be on line **L**, if the distance of **p** and **L** is less than ϵ.

problems, it can also be converted to solving non-linear polynomial equation systems. The non-linear polynomial solver usually operates in floating point arithmetic. Due to the floating point errors in the computation, the roots might be missed. A typical example of this is the solution of ill-conditioned intersection problems, such as tangential intersections and overlaps. Such ill-conditioned problems are very sensitive to the input (e.g., starting point) and computation errors. A small perturbation of the input, as well as an increase and decrease of the numerical precision, may change the result dramatically. The increase of numerical precision can improve the root-finding process, but still *cannot solve* the problem. In fact, even with today's high precision computers, solid modelers are missing intersection points, such as tangential contact points, from time to time. This can sometimes lead to system crash, which is at least frustrating to users and might result in costly expense, too.

Another reason for the lack of robustness of solid modelers is unreliable methods used in geometrical computations and interrogations. Take again the tangential intersection problem for example. Contact points are most probably missed when using conventional balanced nonlinear polynomial solvers.

As has been pointed out by Hoffmann [22], there are three kinds of geometrical failures arising from floating point arithmetic (1) incidence asymmetry, (2) incidence intransitivity, (3) topological violation.

Figure 1-2 shows an example of incidence asymmetry [22]. In this example, $\mathbf{L}_a$, $\mathbf{L}_b$, $\mathbf{L}_c$ and $\mathbf{L}_d$ are four lines. Point **p** is the intersection of $\mathbf{L}_a$ and $\mathbf{L}_b$; **q** is the intersection of $\mathbf{L}_c$ and $\mathbf{L}_d$. Point **q** is incident to point **p**, but point **p** is not incident to point **q**. This is caused by the fact that, in computer programming, a threshold $\epsilon > 0$ is used for determining a small real number as zero. In this manner, a point is considered to be on a line if it is distant from the line less than ϵ. See Figure 1-1 for illustration. Therefore any point in the rectangular areas with width 2ϵ, and line $\mathbf{L}_a$ and $\mathbf{L}_b$ as center line, respectively, is considered as their intersection point, (see Figure 1-2). Similarly, any point in the diamond area generated by lines $\mathbf{L}_c$ and $\mathbf{L}_d$ is considered as their intersection point. Point **q** is in the shaded area formed by lines $\mathbf{L}_a$ and $\mathbf{L}_b$, but point **p** is not in the shaded area formed by lines $\mathbf{L}_c$ and $\mathbf{L}_d$. Hence point **q** is incident to point **p**, while point **p** is not incident to point **q**. Therefore, this is an example of incidence asymmetry.

Figure 1-3 gives an example for incidence intransitivity, **a**, **b** and **c** are three points,

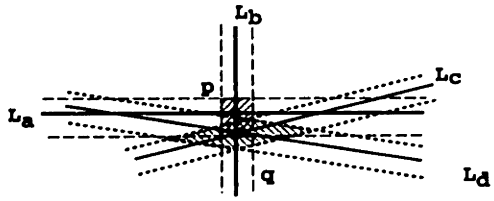
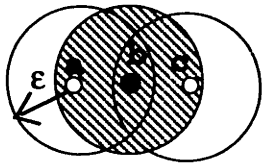


Figure 1-2: An example of incidence asymmetry.



$$\mathbf{a} = \mathbf{b};\ (\ |\mathbf{a}-\mathbf{b}| < \varepsilon)$$
$$\mathbf{b} = \mathbf{c};\ (\ |\mathbf{b}-\mathbf{c}| < \varepsilon)$$
$$\mathbf{a} \neq \mathbf{c};\ (\ |\mathbf{a}-\mathbf{c}| \not< \varepsilon)$$

Figure 1-3: An example for incidence intransitivity.

where $\mathbf{a} = \mathbf{b}$ since $|\mathbf{a} - \mathbf{b}| < \epsilon$; $\mathbf{b} = \mathbf{c}$ since $|\mathbf{b} - \mathbf{c}| < \epsilon$; but $\mathbf{a} \neq \mathbf{c}$, since $|\mathbf{a} - \mathbf{c}| > \epsilon$.

Figure 1-4 gives an example of the problem for topology violation. An ideal closed region in Figure 1-4(a) is not truly closed when represented in the computer (Figure 1-4(b)). It is because there are gaps between curves (edges) and points (vertices) stemming from the inaccurate representation of floating point numbers in computer.

1.2 Previous Work

Robustness of solid modelers has been clearly identified since late 1980, see Hoffmann [23]. There is a great amount of research on robust geometrical representations and computations. Section 1.2.1 summarizes the research on those two topics. This section also summarizes the research for solving systems of non-linear polynomial equations, because it is essential for geometrical computations and interrogations from which robustness problems arise.

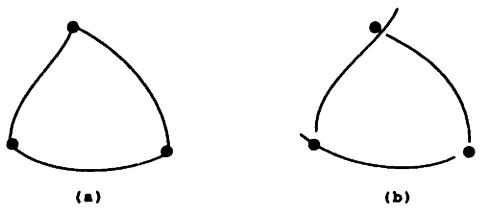


Figure 1-4: An example of topology violation

1.2.1 Robust Geometrical Representation and Computation

Robust Geometrical Representation

Ottmann, Thiemt and Ullrich [48] use exact arithmetic to achieve robust algorithms for intersecting line segments. They only use those linear objects which can be represented exactly and perform their computations with exact numbers. Thus, not only are their geometric entities limited to linear objects, but their representations of linear objects are discrete.

Greene and Yao [18] use the idea of transforming continuous domains into discrete domains to avoid invalid solutions of geometrical computations. In their algorithm, all intersection points and segment vertices are shifted to specified floating point numbers to achieve unambiguous solutions.

Sugihara and Iri [73] employ the same idea but shift the endpoints of segments to grid points. They recognize that the sign of a number can be determined without error if the number is defined from a finite number of computations on finite-bit data. Therefore, in their polyhedral modeler, the plane equations $ax + by + c + d = 0$ only take integers for coefficients a, b, c and adapt arbitrary bits for d as precision is required to maintain robustness.

Milenkovic [41] presents *data normalization* and *hidden variable* methods for robust geometrical computations. The *data normalization* method changes the structure and parameter of geometric entities accordingly so that all numerical tests are confirmably correct. The *hidden variable* method chooses a data structure such that the topology of geometric entities will be consistent for geometrical interrogations. Milenkovic calls it the *hidden variable* method because the topology of the ideal geometric entities is known but their numerical values are not.

Salesin [59], and Salesin, Stofi and Guibas [58], propose ϵ-geometry as a framework for robust geometrical algorithms. Their algorithm finds the range of perturbation for input data in which all perturbed geometries have consistent answers.

Stewart [70] proposes the idea of *local robustness* which is a weaker criterion than robustness and applies it to polyhedral intersections. Local robustness means an algorithm has a consistent set of decisions for all input consisting of exactly the same incidence relationship of points, lines and planes.

Fang, Bruderlin and Zhu use a tolerance-based intuitionistic approach to ensure robustness in solid modeling [15]. They define tolerance-based geometries, for which geometry is represented as tolerance regions which include the space close to the ideal geometry and exclude the space clearly distant from the ideal geometry. The space between these two regions is an ambiguous (undecided) region. Their tolerances can be dynamically changed if ambiguity arises for any geometrical interrogation.

Benouamer, Michelucci and Peroche [3] use lazy rational arithmetic to handle numerical errors in geometrical computation. In their algorithms, only the necessary precise computations are performed without the algorithm having to foresee these computations. They present an *error-free boundary evaluation method* via *lazy rational arithmetic* for polyhedral

solids.

Most of the above research however focuses on linear objects, such as line segments and planes, except Fang, Bruderlin and Zhu [15] who apply their method to quadric and planar surfaces. None of them can treat general free-form Bézier or B-spline curves and surfaces generally used in B-rep solid modeling systems. This thesis attacks this problem based on interval arithmetic.

Robust Geometrical Computation

The problem of lack of robustness for solid modelers in geometrical computations has been widely studied. Geometrical computations are especially sensitive to ill-conditioned geometrical interrogation, such as the computation of the tangential contact points of a curve with a surface, or of two surfaces.

Much literature deals with the computation of singular points in intersection problems, such as [40], [24], [25], [31], [82]. In [40], collinear normal points are used to find the singular points of the intersection of two surfaces. The set of collinear normal points of two surfaces is a superset of the singular points of the intersection of two surfaces. In the end, if the surface distances at collinear normal points are zero, then these points are the actual singular points.

Collinear normal points are also useful for the discovery of intersection loops of two surfaces [64], [62]. Cones and pyramids permit the application of surface subdivisions so that no internal loops exist in subdivided patches, [32], [62], [30]; see Patrikalakis [49] for an overview. Zhou et al [82] use the squared distance functions to compute the stationary points. The set of collinear normal points is a subset of the stationary points of distance functions of two patches, since the distances of the patches at collinear normal points are stationary. However, it includes also the intersection points. Therefore, this is not an efficient method to compute the collinear normal points. Furthermore, in the above literature, problems of curve or surface overlap are not studied. Finally, most of the numerical robustness of above algorithm in floating point was not considered and studied. This thesis develops efficient methods for computing collinear normal points, tangential contact point, tangential intersection curves and overlapping using a robust solver based on interval arithmetic.

The Use of Interval Arithmetic in Geometrical Computation

Interval arithmetic has been applied in geometric modeling, CAD and robotics. For example, Mudur and Koparkar [44], Toth [74], Enger [14], Duff [13] and Snyder [69] applied interval algorithms to geometry processing, and Sederberg and Farouki [63], and Sederberg and Buehler [61] applied interval methods to approximation problems. Tuohy and Patrikalakis [76] applied interval methods to the representation of functions with uncertainty, such as geophysical property maps. Tuohy, Maekawa and Patrikalakis [75] and Hager [21] applied interval methods in robotics. Bliek [4] studied interval Newton methods for design automation and inclusion monotonicity properties in interval arithmetic for solving the consistency problem associated with a hierarchical design methodology.

Maekawa and Patrikalakis [36], [37] extend the interval arithmetic to *rounded interval arithmetic.* They also apply it to the computation of singularities of offset curves and of intersections of two offsets of two planar curves [36]. The rounded interval arithmetic is also used in this thesis as the base for geometrical representations and computations.

1.2.2 Methods for Solving Systems of Non-Linear Polynomial Equations

In computer aided design and computer graphics, free-form curves and surfaces are typically represented parametrically by piecewise polynomials. The governing equations for geometric processing and shape interrogation, in general, reduce to solving systems of non-linear polynomial equations or irrational equations involving non-linear polynomials and square roots of polynomials. Geometrical interrogations can often be converted to solving systems of non-linear polynomial equations. The square root arises, for example, from the normalization of the normal vector and from the analytical expressions of curvatures. The problem involving irrational equations can be reduced to solution of systems of polynomial equations of higher dimensionality through the introduction of auxiliary variables, see Maekawa and Patrikalakis [37], [36]. Alternatively, squaring methods can convert irrational equations to polynomial equations of higher degrees.

Examples of application of non-linear polynomial solver include: (1) to ray trace the offset of trimmed NURBS surfaces(see Hu [26]); (2) to ray trace trimmed rational surface patches (see Nishita et al. [47]); (3) to compute the singular points of silhouette curves in the visual screen (see Hu [26]) and distance function computations (see Zhou et al. [82]); (4) to represent and interrogate functions with uncertainty (see Tuohy et al. [76], [75]); (5) to compute extrema of curvatures of parametric polynomial surface patches (see Maekawa and Patrikalakis [37]), the self-intersection of offset curves and intersections of two offset curves (see Maekawa and Patrikalakis [36], [35]).

There is numerous literature about solving non-linear polynomial equations using global methods, see Kearfott [27], Kearfott [28], Bliek [4], Neumaier [46], Manocha [38] [39], Buchberger [7], Canny [8], Garcia et al.[17], Zangwill [81], Nishita et al. [47], Vafiadou and Patrikalakis [77], Sherbrooke and Patrikalakis [67].[1] These methods can be classified into three categories: algebraic geometry techniques, homotopy techniques and subdivision based-techniques. For more details regarding of categorizations, see [67], [38], and [27].

Among these approaches, subdivision based techniques are preferred for their elegant simplicity and robustness which guarantee to discard regions not containing roots. In particular, they can be easily combined with interval arithmetic to improve numerical robustness. For a summary and applications of subdivision techniques to intersection problems, see Patrikalakis [49]. Following is a brief review of two subdivision methods: Bézier Clipping and Projected-Polyhedron methods.

[1] There also exists a number of *local* numerical techniques which employ some variation of Newton-Raphson iteration or numerical optimization. However, they typically require good initial approximations to roots; such approximations are usually obtained through some sort of global search like sampling, a process which cannot provide full assurance that all roots have been found.

Nishita, et al. [47] develop an adaptive subdivision method, called Bézier Clipping to solve non-linear polynomial systems with two equations and two unknowns. Apart from numerical robustness problems which will be addressed in this thesis, the Bézier Clipping Method presented in [47] works well for well-conditioned non-linear polynomial systems. However for ill-conditioned cases it might report extraneous roots which are not even approximate roots. For counterexamples, see section 3.4.2.

Sherbrooke and Patrikalakis [67] develop the Projected-Polyhedron algorithm and the Linear-Programming to solve systems of n non-linear polynomial equations and n unknowns. They also perform a comprehensive complexity and convergence analysis. However, in general, subdivision methods do not guarantee not missing any roots arising from numerical errors, if floating point arithmetic is used. Maekawa and Patrikalakis [36], [37] coupled Bernstein subdivision with *rounded interval arithmetic* to enhance the robustness of a general solver. These methods guarantee not to miss solutions and are very attractive from the reliability point of view. They are only a constant factor more expensive than plain floating point counterparts. They are also far more efficient than rational arithmetic implementations, and provide a formalized method for conservative rounding of exact operations. Patrikalakis et al. [50] suggest the use of the solver as a core to build a robust solid modeler.

Other interval techniques have received significant attention for the solution of non-linear systems. Among those are primarily interval Newton methods operating in rounded interval arithmetic, combined with bisection to ensure convergence, see Kearfott [28], and Neumaier [46].

This thesis develops a solver for unbalanced non-linear polynomial equation systems. This solver can solve overconstrained non-linear polynomial equations directly and simultaneously.

1.3 Objectives of the Thesis

The overall object of this thesis is to build a framework for a robust solid modeler for curved manifold and non-manifold objects, both in the geometrical representations and operations of geometrical interrogations.

To achieve this goal, the following three intermediate objects are considered: (1) numerically robust geometrical representations, (2) algorithms for numerically robust geometrical interrogations, including ill-conditioned geometrical intersections, and (3) algorithms for Boolean operations for 2D and 3D manifold and non-manifold geometric objects.

Interval polynomial objects are proposed in this thesis for robust geometric representations. A robust algorithm (solver) is developed for the solutions of unbalanced non-linear polynomial equation systems. This solver leads to a robust general unified algorithm for geometrical intersection problems. For surface intersections, a direct algorithm is developed to determine collinear normal points and isolated tangential contact points of two surfaces. Collinear normal points is used for the effective detection of intersection curve loops.

Theory and algorithms are developed for ill-conditioned intersection problems, such as tangential intersection of curves and surfaces, overlapping of curves and surfaces. A theorem

(referred to as End Point Theorem) is presented to verify that if two ideal Bézier patches tangentially intersect along a non-closed curve, the tangential intersection curve must start from a boundary and end at a boundary of either of the two patches. This theorem is extended to any two C^{∞} continuous patches, and is followed by a corollary for surface overlapping.

An n-D novel non-manifold data structure for interval polynomial objects (points, curves, surfaces) is developed. It separates the manifold from the non-manifold parts of the object, and categorizes nodes into six types for Boolean operations. This separation allows the effective use of a new point classification algorithm for non-manifold objects. Based on the robust interval geometrical representations and computations and data structure, algorithms for Boolean operations are developed first for 2D manifold curved solids and then extended to 3D manifold curved solids. These algorithms are further extended to 2D and 3D non-manifold curved solids. In order to represent the geometric objects resulting from Boolean operations, such as trimmed surfaces, a theorem (referred to as Extreme Orientation Theorem) and a related algorithm are developed to determine the orientation of a simple planar closed smooth curve by an extreme point and the curve's derivative. The smoothness can be relaxed to piecewise smoothness.

1.4 Organization of the Thesis

For robust geometrical representations, Chapter 2 explores interval arithmetic and interval geometries, such as interval Bézier curves and Bézier surfaces.

In order to perform geometrical computation robustly, Chapter 3 discusses robust solver for non-linear polynomial equations. In addition, Chapter 3 improves the numerical robustness of the Interval Projected-Polyhedron (IPP) algorithm and extends it to overconstrained and underconstrained non-linear polynomial equation systems.

Based on the solver in Chapter 3, Chapter 4 presents a unified robust algorithm for geometrical intersections. We also discuss ill-conditioned cases, such as computation of tangential contact points of curves and surfaces, and overlapping of curves and surfaces.

Based on the cell-tuple structure, a non-manifold data structure for representing the topology of interval geometries is presented in Chapter 5.

With this data structure, Chapter 6 develops algorithms for non-regularized 2D Boolean operations. This 2D Boolean operators can handle both manifold 2D objects and non-manifold 2D objects.

Chapter 7 further extends 2D Boolean operations to 3D Boolean operations. The 3D Boolean operators can handle manifold 3D objects and non-manifold 3D objects resulting from tangential intersections.

Chapter 8 shows some numerical results for robust intersection algorithms, especially for ill-conditioned tangential intersections and overlappings. It also presents numerical results for Boolean operations on 2D and 3D manifold and non-manifold objects.

Finally, Chapter 9 presents conclusions and recommendations for future research, followed by Appendix A, B and C.

Appendix A presents a theorem and an algorithm to decide the orientation of planar simple closed curves. The orientation of planar simple closed curves is used for representing and visualizing trimmed patches. A point classification algorithm for non-manifold trimmed objects is proposed in Appendix B for Boolean operations. In Appendix C, some issues regarding our implementation experience are discussed.

Chapter 2

Robust Interval Geometric Representations

2.1 Introduction

All state-of-the-art solid modeling systems for free-form objects operate in floating point arithmetic. Usually, a solid object bordered by free-form surface patches is described with a Boundary Representation scheme. The boundary curves and surfaces are frequently expressed in terms of non-uniform rational B-spline (NURBS). NURBS is favored because of its flexibility, generality and its explicit incorporation in data exchange standards. However, assembling a collection of surface patches (of limited precision) to create boundaries of solids frequently yields gaps in boundary parts. These gaps further yield ambiguities in the definition of point sets contained in the interior of the solid, see Figure 2-9(b). This is an example of geometrical failure in current solid modeling systems. As indicated in Chapter 1, this geometrical failure causes topology violation.

The ultimate reason for this failure is the practically *limited precision* of most geometrical representations. This poses theoretical restrictions even for linear objects, see Hoffmann [22] [23]. This thesis adopts interval spline objects for robustly modeling curved solids. Interval spline objects, primarily interval Bézier curves and patches, can retain the advantages offered by Boundary Representations and simultaneously maintain robust representations and geometric processing.

Section 2.2 shows the motivation for searching robust geometrical representations. Section 2.3 reviews the Interval and rounded interval arithmetic for they are the basis for the class of interval spline objects. Section 2.4 discusses the Interval de Casteljau algorithm, because it plays an important role in subdivision methods for solving polynomial systems. Section 2.5 presents the approach for robust curved geometric representations.

2.2 An Example

This section shows one example that motivate the interval geometric representations.

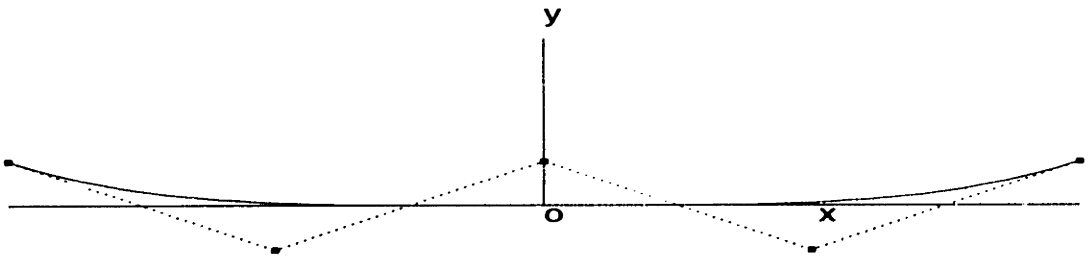


Figure 2-1: Curves $y = x^4$ and $y = 0$ contact tangentially at the origin.

Example 2.1 *Suppose we have a degree four planar Bézier curve whose control points are given by*

$$(-0.5, 0.0625), \quad (-0.25, -0.0625), \quad (0, 0.0625), \quad (0.25, -0.0625), \quad (0.5, 0.0625) \qquad (2.1)$$

as shown in Figure 2-1.

This Bézier curve is equivalent to the explicit curve $y = x^4$ $(-0.5 \leq x \leq 0.5)$. Apparently the curve intersects with x-axis tangentially at $(x, y) = (0, 0)$. However, if the curve has been translated by $+1$ in the y direction and translated back to the original position by moving by $-\frac{1}{3}$ three times during a geometric processing session, the curve will generally not be the same as the original curve in the context of floating point arithmetic (FPA). For illustration, let us assume a decimal computer with a four-digit normalized mantissa, and the computer rounds off intelligently rather than truncating. Then the rational number $-\frac{1}{3}$ will be stored in the decimal computer as -0.3333×10^0. After the processing the new control points will be

$$(-0.5, 0.0631), \quad (-0.25, -0.0624), \quad (0, 0.0631), \quad (0.25, -0.0624), \quad (0.5, 0.0631) \qquad (2.2)$$

If we evaluate the curve at parameter value $t = 0.5$, we obtain (0, 0.00035) instead of (0,0). Therefore there exists a numerical gap which could later lead to inconsistency between topological structures and geometric representations. For example, if these new control points are used for computing intersections with the x-axis, the computer will return no solutions when the tolerance is smaller than 0.00035. The above problem illustrates the case when the error is created during the formulation of the governing equations by various algebraic transformations.

2.3 Interval and Rounded Interval Arithmetic

This section briefly reviews interval and rounded interval arithmetic. It then presents the work on improvement of efficiency for rounded interval arithmetic. This is necessitated by our experience that rounded interval arithmetic often increases the computation time over the floating arithmetic by one order of magnitude.

2.3.1 Interval Arithmetic

An *interval* is a set of real numbers defined by [42]:

$$[a,b] = \{x | a \leq x \leq b\} \tag{2.3}$$

The interval $[a,b]$ is said to be degenerate if $a = b$. Two intervals $[a,b]$ and $[c,d]$ are said to be *equal* if $a = c$ and $b = d$. The *intersection* of two intervals is *empty* or $[a,b] \cap [c,d] = \emptyset$, if either $a > d$ or $c > b$. Otherwise, $[a,b] \cap [c,d] = [max(a,c), min(b,d)]$. The *union* of the two intersecting intervals is $[a,b] \cup [c,d] = [min(a,c), max(b,d)]$. An *order* of intervals is defined by $[a,b] < [c,d]$ if and only if $b < c$. The width of an interval $[a,b]$ is $b - a$ and the *absolute value* is $|[a,b]| = max(|a|,|b|)$.

The interval arithmetic operations are defined by [42]

$$[a,b] \circ [c,d] = \{x \circ y \mid x \in [a,b] \text{ and } y \in [c,d]\}. \tag{2.4}$$

where $\circ$ represents an arithmetic operation $\circ \in \{+,-,\cdot,/\}$. Using the end points of the two intervals, we can rewrite equation (2.4) as follows

$$\begin{aligned}
&[a,b] + [c,d] = [a+c, b+d] \\
&[a,b] - [c,d] = [a-d, b-c] \\
&[a,b] \cdot [c,d] = [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \\
&[a,b]/[c,d] = [min(a/c, a/d, b/c, b/d), max(a/c, a/d, b/c, b/d)]
\end{aligned} \tag{2.5}$$

provided $0 \notin [c,d]$ in the division.

2.3.2 Rounded Interval Arithmetic

If floating point arithmetic is used to evaluate interval arithmetic equations (2.5), there is no guarantee that the roundings of the bounds are conducted conservatively. Floating numbers are represented in the computer by a fixed length. The number of bytes to represent a floating point number depends on the precision of the variable. For example, the IEEE standard for a *double-precision* has 64 bits, 8 bytes wordsize, and is stored in a binary form $(\pm)m \cdot 2^{exp}$, where m is the *mantissa* ($0.5 \leq m < 1$) and exp is the *exponent*. Figure 2-2 illustrates how data is stored in the binary form; a single bit for sign, 11 bits for exponent and 52 bits for mantissa. Since the mantissa is restricted to the range $0.5 \leq m < 1$, the bit for 2^{-1} is not used. The exponent is 1022 biased to ensure the stored exponent is

always positive. For example the number -0.125 is stored as $1011111111000\cdots0$. Most left bit represents the sign $-$, next 11 bits 01111111100 is the biased exponent which is 1020-1022 = -2 and the rest of 52 bits which are all zero represents the mantissa 0.5. Hence $-0.5 \cdot 2^{-2} = -0.125$. If x and x' are consecutive positive double-precision numbers, they differ by an amount ϵ called *ulp* (one Unit in the Last Place), so that $\epsilon = 2^{-53} \cdot 2^{exp} = 2^{exp-53}$. Now it is possible to carry out the operation of interval arithmetic with rounding, so that the computed end points always contain the exact interval as follows

$$
\begin{aligned}
&[a,b] + [c,d] \equiv [a + c - \epsilon, b + d + \epsilon] \\
&[a,b] - [c,d] \equiv [a - d - \epsilon, b - c + \epsilon] \\
&[a,b] \cdot [c,d] \equiv [min(ac,ad,bc,bd) - \epsilon, max(ac,ad,bc,bd) + \epsilon] \\
&[a,b]/[c,d] \equiv [min(a/c,a/d,b/c,b/d) - \epsilon, max(a/c,a/d,b/c,b/d) + \epsilon] \qquad (2.6)
\end{aligned}
$$

Each ϵ in the equations can be obtained by $\epsilon = 2^{exp-53}$ where exp is extracted from *each* computed lower or upper bound. We refer to the definitions given in equations (2.6) as *rounded interval arithmetic.*

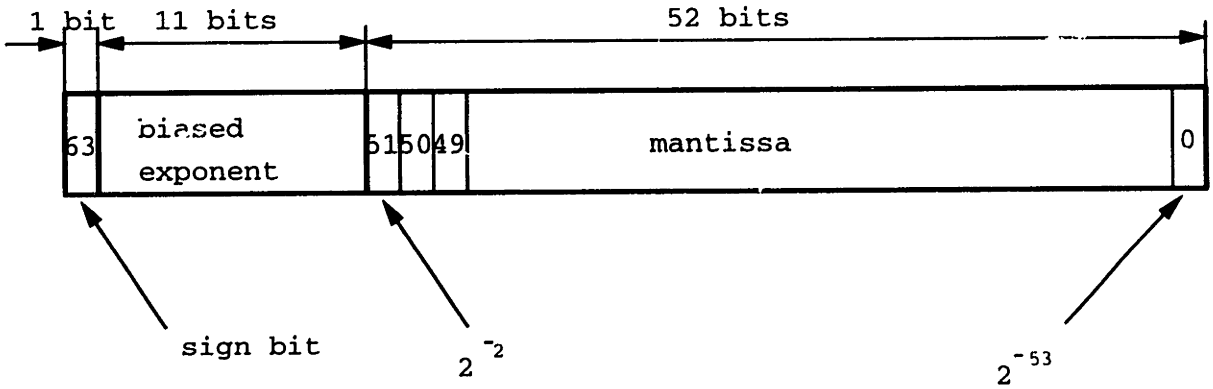


Figure 2-2: IEEE format for binary representation of double-precision floating-point number

When performing standard operations for interval numbers, we always make the lower bound to include its previous consecutive number, which is smaller than the lower bound by *ulp*, and the upper bound include its next consecutive number. We enlarge the result of interval operators by two *ulp* numbers, so the result will be reliable in subsequent operations. Consequently, we can compute the result of interval operations in the least conservative fashion.

2.3.3 The Improvement of Efficiency for Rounded Interval Arithmetic

Rounded interval arithmetic can enhance robustness, yet in our experience, generally computation time increases by a factor of ten to forty over the use of floating point arithmetic. This section introduces bit operators to improve the efficiency for rounded interval

arithmetic. In our experiments, rounded interval arithmetic with bit operators could save two-thirds of computation time of standard rounded interval arithmetic.

Routines `ldexp` and `frexp` are called to extract the ulp of a double-precision and computer-representable number. `frexp` is used mainly to extract exponent `expt` of `d`. The mantissa `man` is ignored. `ldexp` is used to compute the ulp of `d`. The routine that carries out the extraction is:

```
#include <math.h>

double ulp (double d)
{
  double man, ulp;
  int expt;

  man = frexp(d, &expt);
  ulp = ldexp(0.5, expt - 52);

  return ulp;
}
```

The most significant difference between floating point and rounded interval arithmetic is the extraction in *ulp* of a double-precision number for interval arithmetic, see also Hu [26]. This extra routine dramatically slows down overall performance of programs adopting interval arithmetic. Therefore, if we can improve the efficiency in extracting **ulp** from a double-precision and computer-representable number, we can make significant improvements in the use of rounded interval arithmetic in various applications.

In the **ulp** routine, we clearly see excessive work, because of the extraction of the mantissa. Furthermore, the routine `ldexp` is a general one. In our particular case, the mantissa of "ulp" is always `0.5`, the hidden bit is always 1, and bits for mantissa of ulp of `d` are all zero. We could take advantage of this knowledge to save computation time.

In fact, the only work which is useful for us from this routine is to extract the exponent of `d` and subtract `52` from it, and put the result into exponent bits of ulp. The rest of bits of ulp are fixed, since ulp is always positive and the mantissa bits are all zeros, except the hidden one, which is always 1. See fig. 2-3.

Bit operators are among the fastest operations for the computer. So working on bits instead of calling `ldexp` and `frexp` can save substantial amount of computation time.

Here, the way we present how to compute *ulp* with bit operators is only good for IEEE standard 754 double-precision floating-point arithmetic. Nevertheless, it could easily be adopted with minor modifications for triple-precision or quadruple-precision floating-point arithmetic.

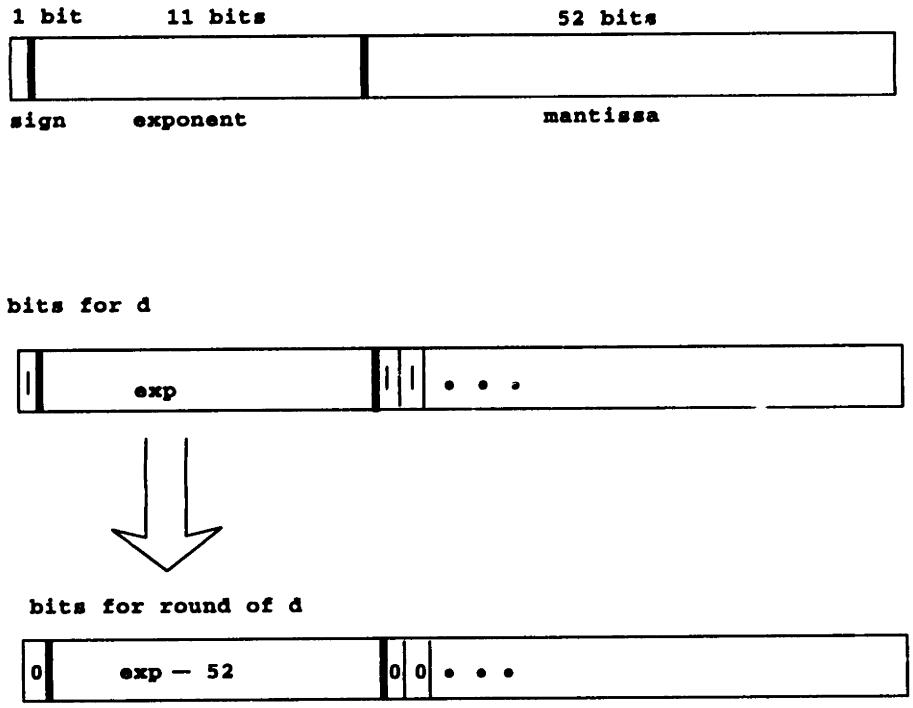


Figure 2-3: The bit operations for the ulp of a double-precision number

2.4 Interval de Casteljau Algorithm

In this section, de Casteljau algorithm is summarized and coupled with rounded interval arithmetic, because it is used to evaluate and subdivide interval polynomial splines. It also plays an important role in the IPP solver in Chapter 3.

The de Casteljau algorithm [16] is a repeated application of affine map of two points:

$$\textit{Suppose} \quad \mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbf{R}^3 \quad \textit{and} \quad t \in \mathbf{R}.$$

Let

$$\mathbf{b}_i^r(t) = (1-t)\mathbf{b}_i^{r-1}(t) + t\mathbf{b}_{i+1}^{r-1}(t) \quad \begin{cases} r = 1, \ldots, \text{n} \\ i = 0, \ldots, \text{n-r} \end{cases} \tag{2.7}$$

where

$$\mathbf{b}_i^0(t) = \mathbf{b}_i (i = 0, 1, \ldots, n).$$

Then $\mathbf{b}_0^n(t)$ is the point with parameter value t on the Bézier curve.

$\mathbf{b}_i$ $(i = 0, 1, \ldots, n)$ are control points. The control points form the *control polygon.* Figure 2-4 illustrates a cubic Bézier curve. The intervening coefficients can be arranged into a triangular array of points, the *de Casteljau scheme.* In equation 2.8 we take the

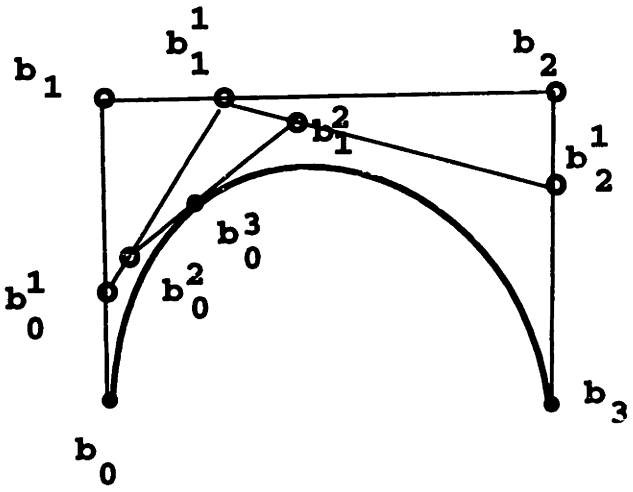


Figure 2-4: The de Casteljau algorithm

cubic case for example.

$$\begin{array}{cccc} \mathbf{b}_0 & & & \\ \mathbf{b}_1 & \mathbf{b}_0^1 & & \\ \mathbf{b}_2 & \mathbf{b}_1^1 & \mathbf{b}_0^2 & \\ \mathbf{b}_3 & \mathbf{b}_2^1 & \mathbf{b}_1^2 & \mathbf{b}_0^3 \end{array} \tag{2.8}$$

Subdivision of Bézier Curves: The de Casteljau algorithm can also provide the control points of subdivided Bézier curves [16]. This subdivision algorithm is heavily used in the IPP solver, as discussed in Chapter 3.

Let t be the variable of a Bézier curve and ranging from 0 to 1, i.e., $0 \leq t \leq 1$. If we subdivide a Bézier curve of degree n at $t = t_0$, $0 \leq t_0 \leq 1$. Then the control points $\mathbf{c}_j$ $(j = 0, 1, \ldots, n)$ of the Bézier curve associated to the range $[0, t_0]$ are:

$$\mathbf{c}_j = \mathbf{b}_0^j(t_0) \quad j \in \{0, \ldots, n\} \tag{2.9}$$

And the control points $\mathbf{c}_j$ of the Bézier curve associated to the range $[t_0, 1]$ are:

$$\mathbf{c}_j = \mathbf{b}_j^{n-j}(t_0) \quad j \in \{0, \ldots, n\} \tag{2.10}$$

The interval de Casteljau algorithm is a repeated linear interpolation of two interval points (rectangles or boxes), illustrated in Figure 2-5. In Figure 2-5, we can observe that the interpolated interval control points can be obtained by linear interpolation of the corner points of the original intervals with a slightly larger area denoted by dotted line, which is the result of rounded interval arithmetic.

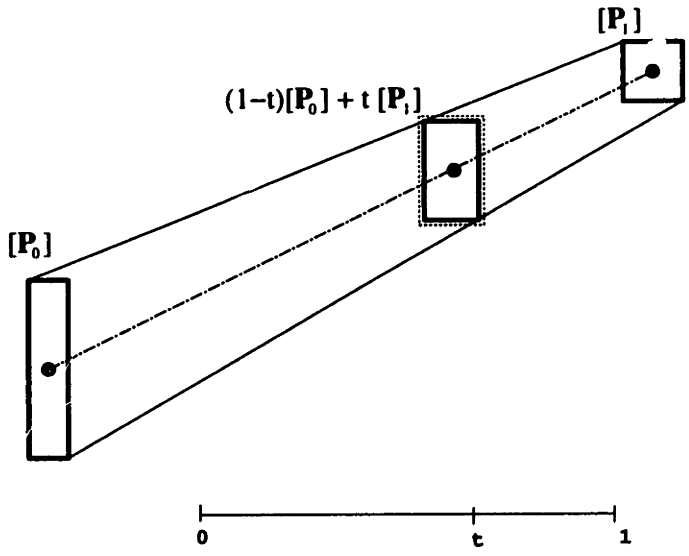


Figure 2-5: Affine map

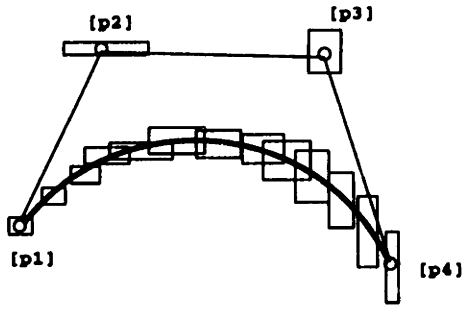


Figure 2-6: An example of an interval polynomial curve.

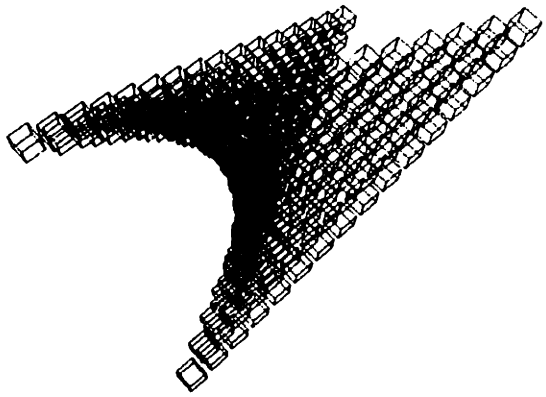


Figure 2-7: An example of an interval polynomial surface patch.

2.5 Robust Representation of Interval Polynomial Splines

Geometrical representations in the context of floating point arithmetic often yield geometrical failure, as discussed in Section 2.1. To overcome these geometrical failures, we use the following interval polynomial objects in the solid modeler.

2.5.1 Interval Polynomial Splines

Interval polynomial objects (curves and surfaces) are polynomial curves/surfaces with interval coefficients. Usually, they are represented by Bézier curves/surfaces with interval control points (see Figures 2-6 and 2-7).

Hence interval polynomial objects differ from polynomial splines in that the real numbers representing control point coordinates are replaced by *intervals*. That means, the classical control points are replaced by rectangle or boxes. Consequently, interval Bézier curves represent slender tubes and interval patches represent volumes typically as thin Bézier shells, if the intervals chosen are small.

In general the control points of the given curve are given in floating point numbers which can be initially treated as degenerate (zero-width) intervals. By using *rounded interval*

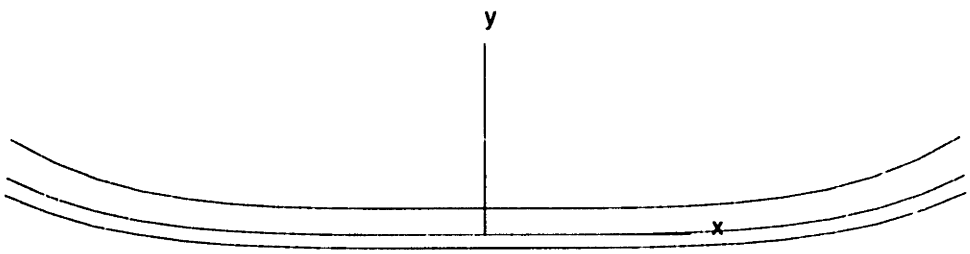


Figure 2-8: A Bézier curve bounded by an interval Bézier curve.

arithmetic in the geometric processing proceeds, the width of the interval grows gradually so that the interval always contain the exact result.

In the remaining of this section, we explain how the interval polynomial spline representations achieve the robustness. The concept of interval polynomial spline comes up naturally when formalizing computational accuracy on computing machinery. Remember that a patch of interval spline is not a true 2D object, but rather an object enclosed by a shell containing a family of patches, whose representations fulfill the error bounds controlled by floating point arithmetic. Figure 2-8 illustrates the bounding interval Bézier curve of the original Bézier curve whose control points are given by (2.1). (In this figure, the height of the interval control points is exaggerated by the factor of 20.) In the computation of the intersection of the curve with the x-axis is computed if the interval curve is used, the algorithm will capture the intersection point. See Figure 2-9(c) for an illustration.

In Figure 2-9 the thickness of sealing boundaries is exaggerated for illustration but in general it may be of the order of 10^{-6} to 10^{-12} when operating in double precision arithmetic. Topological violations can also be avoided by using interval polynomial objects Figure 2-9.

2.5.2 Interval Polynomial Splines v.s. Polynomial Splines

Interval polynomial splines inherit the advantages of polynomial splines, such as (1) economical memory space (only twice as much), (2) representability of large class of free-form objects, (3) acceptability as international standard for data exchange and representation.

It also has the following advantages over polynomial splines (1) automatical control of computation errors, (2) avoidance of topological violation, (3) ability to solve ill-conditioned geometric interrogation problems.

Although the use of interval polynomial splines results in robust geometrical representation, it also destroys some properties of spline objects. This section discusses two issues for interval polynomial splines: one is dimensionality and the other is incidence of interval objects.

One of the issues if the dimensionality of interval objects, as interval objects are of the

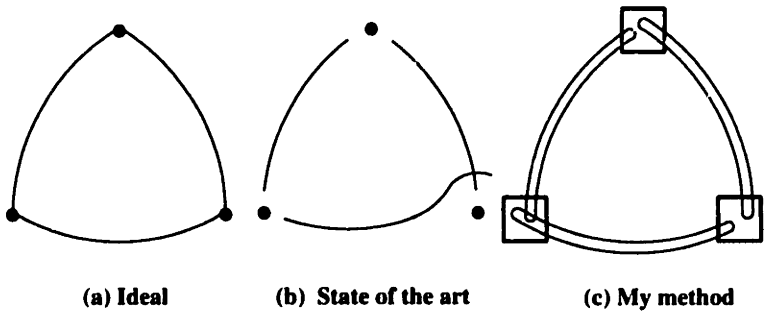


Figure 2-9: (a) Idealized Boundary Representation of a triangular face; (b) Actual numerical Boundary Representation in current CAD/CAM systems with gaps and inappropriate intersections; (c) Conceptual sketch of proposed generalized Boundary Representation in terms of interval polynomial curves / surfaces.

same dimensionality, regardless of their original one. For instance, in 3D space, interval points, curves and surfaces are all volumetric.

However, in some situations, the differentiation of dimensionality of an object is still important. Take point classification for example. We have to find the number of intersection "points" of a ray with the boundary in order to decide whether a point is inside a bounding curve. Another example is the computation of the Euler Formula; we have to distinguish vertices, edges, and faces from each other.

Therefore, in this thesis, the dimensionality of interval objects is defined by the dimensionality of their original objects. This will be further discussed in Chapter 5.

Another issue is the incidence of interval objects. We will discuss this definition in Chapter 4.

Chapter 3

Robust Solver of Non-Linear Polynomial Systems

3.1 Introduction

Many geometrical computations can be transformed to solving a system of non-linear polynomial equations, either balanced (the number of equations and unknowns are the same) or unbalanced systems. An unbalanced system could be *underconstrained* (the number of equations is less than the number of unknowns) or *overconstrained* (the number of equations is larger than the number of unknowns). A balanced system is the most common form in geometric modeling problems. There are plenty of algorithms to solve balanced systems, see Section 1.2.2.

The traditional way to solve overconstrained systems is to convert them to many balanced systems which can be solved sequentially by appropriate algorithms.

For instance, to solve the intersection problem of two planar parametric curves can be converted to a system with two equations and two unknowns. It is a balanced system, and can be solved many algorithms to solved either by generic algorithms [67] [36] [29] [27] [38] [4] or special algorithms [60] [65] [66] [26]. However, solving for the intersection of two parametric curves in 3D space is relatively rarely addressed in the literature, partly, because it is an overconstrained system with three equations and two unknowns.

Such overconstrained systems are solved by converting them (1) into a sequence of balanced and underconstrained problems, or (2) into minimization problems, e.g., see [82] by Zhou, Sherbrooke and Patrikalakis.

The remaining of the chapter is organized as follows. Section 3.2 presents a general solver for unbalanced and balanced non-linear polynomial systems. This solver is used as a kernel for our robust geometrical computations. It also describes the essential idea behind the extension from solving *balanced* systems to solving *unbalanced* systems. By the nature of the subdivision methods, the solver might report multiple roots for one actual root. Hence, a method for consolidating those multiple roots into one root is also discussed. Section 3.3 examines those leftover boxes by the previous subdivision methods, such as

Projected-Polyhedron algorithm and Bézier clipping method. We examine those leftover regions because the previous subdivision methods mistake some extraneous roots as actual roots for some ill-conditioned cases. Section 3.4 gives two such ill-conditioned examples for subdivision methods. One is for 1D and the other is for 2D. An analysis of the failure of subdivision methods for those two examples by the Bézier clipping method is given in the same section. Section 3.5 presents a *Convex-Hull-Cross-Axes* check to overcome this failure. Section 3.6 proves the correctness of this check.

3.2 Rounded Interval Projected-Polyhedron Algorithm

This section presents a robust solver for unbalanced and balanced non-linear polynomial systems.

Maekawa and Patrikalakis [37] [36] extended the Projected-Polyhedron (PP) algorithm [67] to operate in rounded interval arithmetic for numerical robustness. In fact the PP algorithm implemented in rational arithmetic is robust but unacceptably slow for high degree and high dimensional cases. A floating point implementation of the PP algorithm, despite the use of the Bernstein basis, is not always robust when dealing with ill-conditioned roots. Without the cost of rational arithmetic, rounded interval arithmetic, on the other hand, can be considered as a method of conservative rounding of rational arithmetic. Algorithm for Interval Projected Polyhedron (IPP) solver described in this section is an extension of the algorithm described in [36] from solving balanced systems to solving unbalanced systems.

3.2.1 Projected-Polyhedron Algorithm for Unbalanced and Balanced Systems

Suppose we solve a system of nonlinear polynomial equations $\mathbf{f} = (f_1, f_2, \ldots, f_n) = \mathbf{0}$ over the box $S \in \mathbf{R}^m$ ($n > m$, $n = m$, $n < m$) where S is defined by

$$S = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_m, b_m]. \tag{3.1}$$

That is, we wish to find all $\mathbf{u} \in S$ such that

$$f_1(\mathbf{u}) = f_2(\mathbf{u}) = \ldots = f_n(\mathbf{u}) = \mathbf{0}. \tag{3.2}$$

By making the *affine parameter transformation* [16] $u_i = a_i + x_i(b_i - a_i)$ for $i = 1, \cdots m$, we convert the problem to determining all $\mathbf{x} \in [0, 1]^m$ such that

$$f_1(\mathbf{x}) = f_2(\mathbf{x}) = \ldots = f_n(\mathbf{x}) = \mathbf{0}. \tag{3.3}$$

Now furthermore suppose that each of the f_k is polynomial in the independent parameters $x_1, x_2, \ldots, x_m$. Let $d_i^{(k)}$ denote the degree of f_k in the variable x_i; then f_k can be written

in the multivariate Bernstein polynomials:

$$f_k(\mathbf{x}) = \sum_{i_1=0}^{d_1^{(k)}} \sum_{i_2=0}^{d_2^{(k)}} \cdots \sum_{i_m=0}^{d_m^{(k)}} w_{i_1 i_2 \ldots i_m}^{(k)} B_{i_1,d_1^{(k)}}(x_1) B_{i_2,d_2^{(k)}}(x_2) \ldots B_{i_m,d_m^{(k)}}(x_m). \tag{3.4}$$

where $B_{i,l}$ is the i-th Bernstein polynomial. The notation in (3.4) can be simplified by letting $I = (i_1, i_2, \ldots i_m)$, $D^{(k)} = (d_1^{(k)}, d_2^{(k)}, \ldots, d_m^{(k)})$ and writing (3.4) in the equivalent form [67].

$$f_k(\mathbf{x}) = \sum_{I}^{D^{(k)}} w_I^{(k)} B_{I,D^{(k)}}(\mathbf{x}). \tag{3.5}$$

Here we have merely rewritten the product of Bernstein polynomials as a single *Bernstein multinomial* $B_{I,D^{(k)}}(\mathbf{x})$. Bernstein polynomials have a useful identity called *linear precision property*, i.e., t can be expressed as the weighted sum of Bernstein polynomials with coefficients evenly spaced in the interval $[0, 1]$. Using this property, we can rewrite equation (3.5) as follows:

$$\mathbf{F}_k(\mathbf{x}) = \sum_{I}^{D^{(k)}} \mathbf{v}_I^{(k)} B_{I,D^{(k)}}(\mathbf{x}) \tag{3.6}$$

where

$$\mathbf{v}_I^{(k)} = \left(\frac{i_1}{d_1^{(k)}}, \frac{i_2}{d_2^{(k)}}, \ldots, \frac{i_m}{d_m^{(k)}}, w_I^{(k)}\right)^T. \tag{3.7}$$

These $\mathbf{v}_I^{(k)}$s are called the *control points* of $\mathbf{F}_k$. Now the algebraic problem of finding roots of systems of polynomials has been transformed to the geometric problem involving intersection of hypersurfaces. Because the problem is now phrased geometrically, we can use the *convex hull property* of the multivariate Bernstein basis to bound the set of roots. We can structure a root-finding algorithm as follows [51]:

1. Start with an initial box of search.
2. Scale the box and, as we did in converting between equations (3.2) and (3.3), perform an appropriate affine parameter transformation to the functions f_k, so that the box becomes $[0, 1]^m$. However, keep track of the scaling relationship between this box and the initial box of search. This transformation can be performed with multivariate De Casteljau subdivision.
3. Using the convex hull property, find a sub-box of $[0, 1]^m$ which contains all the roots. The essential idea behind the box generation scheme in this algorithm is to transform a complicated $(m + 1)$-D problem into a series of m 2D problems, as follows:
   (a) Project the $\mathbf{v}_I^{(k)}$ of all of the $\mathbf{F}_k$ into m different ordinate planes; specifically, the (x_1, x_{m+1})-plane, the (x_2, x_{m+1})-plane, and so on, up to the (x_m, x_{m+1}) plane.
   (b) In each one of these planes,

i. Construct n 2D convex hulls. The first is the convex hull of the projected control points of $\mathbf{F}_1$, the second is from $\mathbf{F}_2$ and so on, up to $\mathbf{F}_n$.

ii. Intersect each convex hull with the horizontal axis (that is, $x_{m+1} = 0$). Because the polygon is convex, the intersection may be either a closed interval (which may degenerate to a point) or empty. If it is empty, then no root of the system exists within the given search box.

iii. Intersect the intervals with one another. Again, if the result is empty, no root exists within the given search box.

(c) Construct an m-dimensional box by taking the Cartesian product of each one of these intervals in order. In other words, the x_1 side of the box is the interval resulting from the intersection in the (x_1, x_{m+1})-plane, and so forth.

4. Using the scaling relationship between our current box and the initial box of search, see if the new sub-box represents a sufficiently small box in $\mathbf{R}^m$. If it does not, then go to step 5. If it does, then check the convex hulls of the hypersurface in the new box. If the convex hulls cross each variable axis, conclude that there is a root or an approximate root in the new box, and put the new box into a root list. Otherwise the new box is discarded.

5. If any dimension of this sub-box are not much smaller than 1 unit in length (i.e., the box has not decreased much in size along one or more sides), split the box evenly along each dimension which is causing trouble. Continue on to the next iteration with several independent sub-problems.

6. If none of the box is left, then the root-finding process is over. Otherwise, go back to step 2, and perform it once for each new box.

The above root-finding algorithm differs form its previous counterpart in [51] [35] in that (1) the number of Bézier hypersurfaces projected in Step 3 are more (or less), for we have here an unbalanced system. We will discuss the feasibility in Section 3.2.2; (2) previous Projected-Polyhedron algorithm will sometimes report extraneous roots. We developed an additional check, called convex-hull-cross-axes check, to delete those extraneous roots. The check is shown is Step 4. We will discuss this in Section 3.5.

3.2.2 Projected-Polyhedron for Unbalanced Polynomial Systems

In the algorithm described in Section 3.2.1, if we have more equations, we just project more Bézier hypersurfaces onto each (x_i, x_{m+1}) plane. In fact, the more Bézier hypersurfaces we have, the tighter the feasible regions will be for each round of projections, if these equations are independent. Take the example in Figure 3-1. It involves a balanced system with two equations and two unknowns as follows:

$$x^2 + y^2 - 1 = 0 \tag{3.8}$$

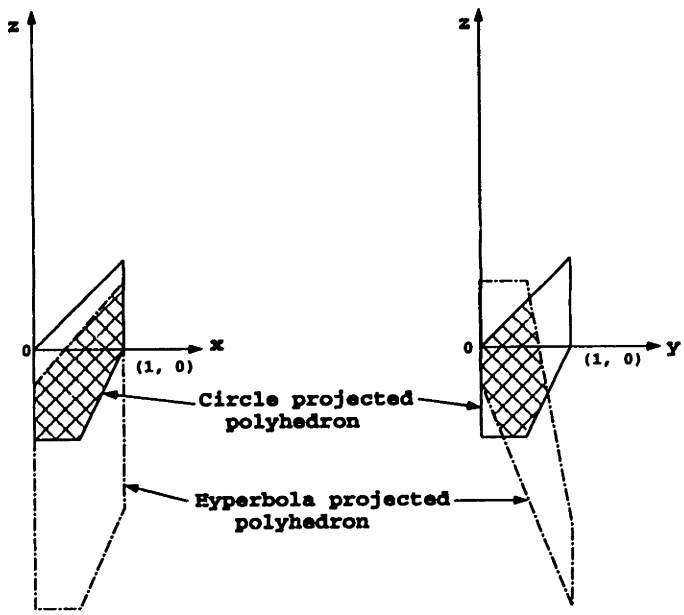


Figure 3-1: Projecting the polyhedra of $(x, y, x^2 + y^2 - 1)$ and $(x, y, \frac{5}{4}x^2 - \frac{5}{2}y^2 - \frac{1}{2})$.

$$\frac{5}{4}x^2 - \frac{5}{2}y^2 - \frac{1}{2} = 0 \tag{3.9}$$

Thus two polyhedra are projected onto (x_i, x_3)-plane, ($x_3 = z$ in Figure3-1) to shrink the feasible range, as indicated in Figure 3-1. By adding one more equation to that system as follows:

$$\frac{x^2}{4} + 4y^2 - 1 = 0 \tag{3.10}$$

the new system becomes an overconstrained system with three equations and two unknowns (intersecting a circular arc, a hyperbolic arc and an elliptical arc). Figure 3-2 shows the shrinkage of the feasible range for the overconstrained system. As we can see from Figure 3-2, the projection of the elliptical convex hull does help in eliminating part of the y-axis, while in Figure 3-1 no part of the x-axis can be eliminated from the projection of circular and elliptical convex hulls. In this manner, solving more equations than unknowns accelerates the root finding process.

If the intersection of the projected convex hulls with the i-th axis does not shrink to 80% or less of the previous i-th interval [67], then each hypersurface is binarily subdivided in i-th direction. Otherwise, the process is reiterated for each direction until the interval box satisfies the tolerance condition [67].

3.2.3 The Advantage of the Extended IPP Solver

The IPP has been implemented in C++ and has been tested extensively. It has three advantages: (1) theoretical and numerical robustness; (2) a global solution method without

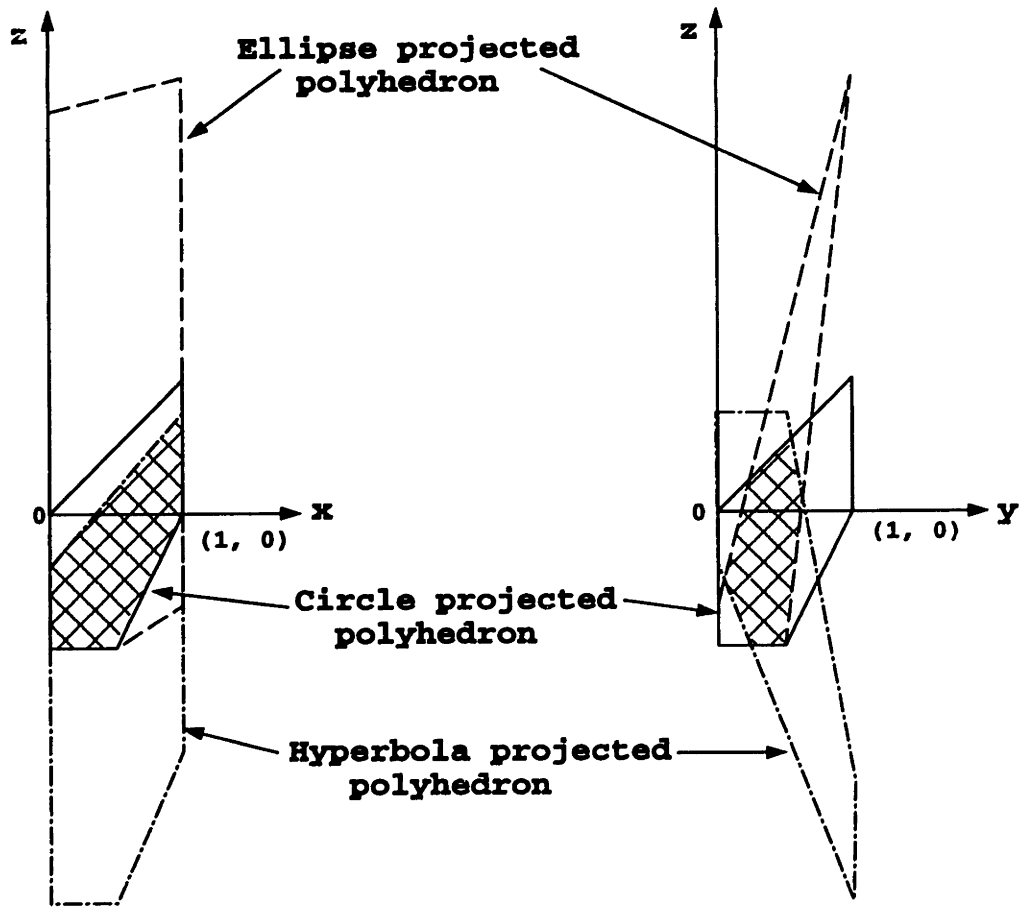


Figure 3-2: Projecting the polyhedra of (x, y, x^2+y^2-1), $(x, y, \frac{5}{4}x^2-\frac{5}{2}y^2-\frac{1}{2})$ and $(x, y, \frac{x^2}{4}+4y^2-1)$.

initial root approximation; (3) solution of overconstrained, underconstrained, and balanced systems of non-linear polynomials.

One example to use the extended root solver, is to compute the tangential intersection of a curve with a surface. The old root solver will take 2530u CPU time to compute the result. By adding one more equation imposing the tangential contact constraint, i.e., the normal of the surface is perpendicular to the tangent of the curve at the intersection point, we have a system of four equations with three unknowns. It only took 15.3u CPU time to compute that one point. The difference in computation time is a factor of 165.4.

Real Solutions of Underconstrained Systems: In underconstrained systems, we have less equations than variables. In the previous subsection, we know that it is possible to project more hypersurfaces than variable planes to find real roots for overconstrained systems. It is also possible to project less hypersurfaces than variable planes onto (x_i, x_{m+1})-planes. The only interaction between those hypersurfaces is the intersection of their projected convex hulls in each (x_i, x_{m+1})-plane. Therefore, we can use the following corollary to conclude the extension of Projected-Polyhedron algorithm to solve underconstrained systems.

Corollary 3.1 *Let* **S** *be an underconstrained polynomial system with m equations and n variables, where $m < n$. If an ϵ-box satisfies the tolerance condition [67], and the final convex hull check, then the ϵ-box contains at least approximate roots.*

Proof: This proof can be done by the same procedures as in Theorem 3.1 in Section 3.6.2. □

3.2.4 Consolidation of Roots

We may want to consolidate several roots, which have approximately the same value, to one root since the solver might report many roots for one actual root due to the nature of the subdivision method. This is important when we deal with topological interrogations, e.g., to identify a point as being outside or inside of a closed region by employing the ray-test method. This consolidating procedure is crucial to the solid modeling system. It can simply be described as follows: for a system of n equations and m variables, if two root intervals overlap in every x_i-axis, for $i = 1, \ldots, m$, then we consolidate these two roots to one root.

3.3 Examination of the Leftover Boxes by Subdivision Methods

Let us call the leftover boxes from the subdivision methods as ϵ-boxes. The goal of this section is to analyze these ϵ-boxes. Section 3.5 will describe the implementation of *Convex-Hull-Cross-Axes* check, to determine which ϵ-boxes contain roots. Section 3.6 will prove the correctness of the *Convex-Hull-Cross-Axes* check.

The essence of the Projected-Polyhedron Algorithm [67] is to exclude those regions which contain no roots. It is an iterative method in which boxes which may contain roots satisfy tolerance conditions. That means all sizes of such boxes are smaller than the prescribed tolerance ϵ. Theoretically what the Projected-Polyhedron algorithm achieves is to determine that there exists a *neighborhood* W for each ϵ-box: ϵ-B, such that there is absolutely no possibility for a root to be in $W \setminus \epsilon$-B [1]. However, it never tells us where the root is.

Section 3.3.1 examines ϵ-boxes because for some ill-conditioned cases, ϵ-boxes may not contain root at all.

3.3.1 Four Cases for Leftover Boxes

In the previous version of the Projected-Polyhedron algorithm, when the bounding box's sizes are smaller than the prescribed tolerance in each dimension, the algorithm stops further subdividing the domain, and assumes that the bounding box contains a root. However, the leftover bounding boxes reported by the Projected-Polyhedron algorithm could be of four cases: (case 1) true roots (for transversal cases), or (case 2) multiple roots (for tangent cases), or (case 3) approximate roots (for nearly tangent cases), or (case 4) no root at all, not even approximate roots (for pathological cases).

In Figure 3-3, examples of all four cases are shown. An example of a transversal case is shown in Figure 3-3(1). This is the most benign case for using subdivision methods, such as the Bézier Clipping method or the Projected-Polyhedron algorithm. That is because each ϵ-box in this case contains one true root. An example of a tangent case is shown in Figure 3-3(2). In this case, it is appropriate to use the Bézier Clipping method or the Projected-Polyhedron algorithm but troublesome. The true tangent root will surely be enveloped by an ϵ-box, but so will be points around its neighborhood, as long as ϵ-boxes pass the tolerance conditions. This case usually produces a lot of ϵ-boxes around the tangent root and results in the use of tremendous time and memory space to complete the computation. In the example of a nearly tangent case, which is shown in Figure 3-3(3), an ϵ-box might contain approximate roots, depending on the tolerance. If the tolerance is loose, an ϵ-box might be reported as an approximate root. However, if the tolerance is set tight enough, then no ϵ-box will be held. This is acceptable because a user can set the tolerance according to the resolution needed in his/her work. In Figure 3-3(4), we show the pathological case, which is similar to the counterexample of Figure 3-4. The ϵ-box contains no root at all (not even an approximate one). The following section shows two counterexamples for subdivision methods to illustrate case 4.

In this thesis, our goal is to identify an ϵ-box containing at least approximate roots.

[1] $W \setminus \epsilon\text{-}B = \{x \| x \in W \text{ and } x \notin \epsilon\text{-}B\}$

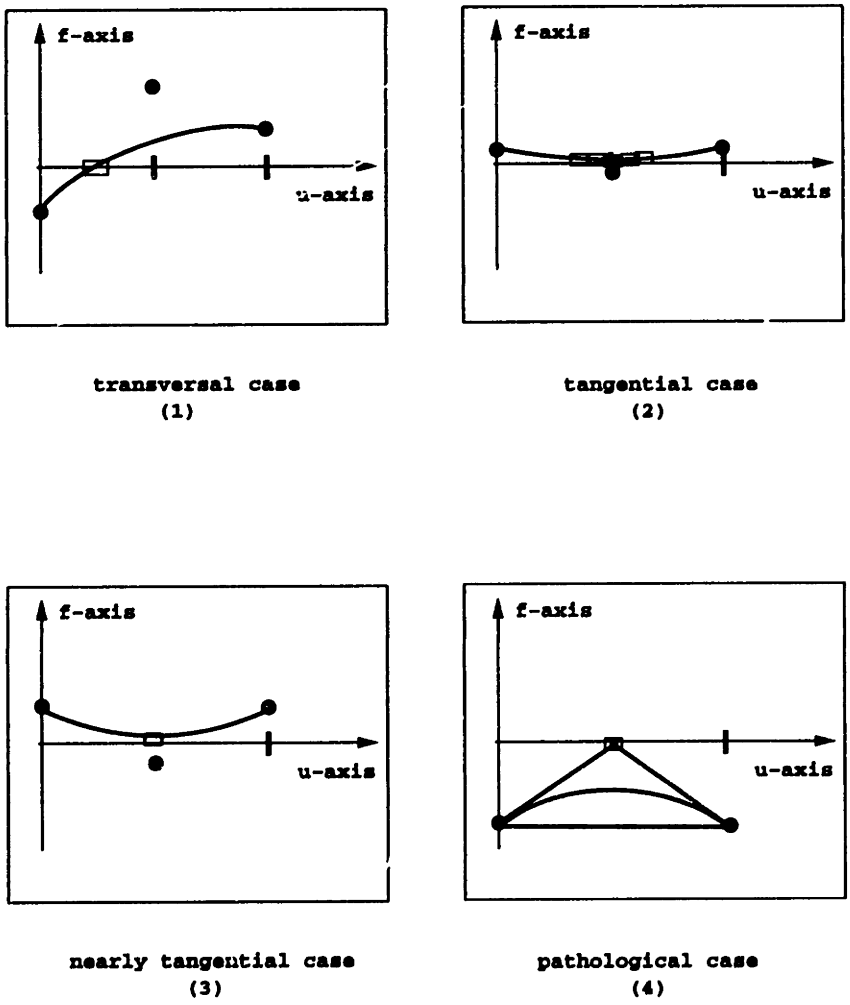


Figure 3-3: Examples of four cases of ϵ-boxes

3.4 Ill-Conditioned Convex Hulls for Subdivision Methods

In this section, we show two examples in which ϵ-box leftover by the Projected-Polyhedron algorithm is far from containing a root. However, no matter how small the tolerance is, the intersection of the projected polyhedron of hypersurfaces with variable axes will satisfy the tolerance condition. Thus, in practice, the subdivision methods will always mistake the leftover interval region for an approximate root. These two counterexamples can be applied to other subdivision methods too, such as Bézier clipping method. The next Section 3.4.1 briefly reviews Bézier clipping method. Section 3.4.2 presents these two counterexamples for subdivision methods.

3.4.1 Brief Review of Bézier Clipping Method

In reference [47], Nishita et al. solved a non-linear polynomial system with two equations and two unknowns to compute ray-patch intersections. They use the convex-hull-intersection technique to clip away those impossible-to-contain-root regions until the region is small enough to *satisfy tolerance conditions.* Then they treated that region as an intersection point, see reference [47] page 342. Nevertheless to pass the tolerance condition is only a *necessary condition*, but not a *sufficient condition.* The necessary condition means any region of parameter domain excluded by the tolerance condition does not contain any root. It is not a sufficient condition because a region passing the tolerance condition does not guarantee to contain a root. A counterexample of the using Bézier Clipping method to solve 2D nonlinear polynomial system is shown in the following subsection.

3.4.2 Counterexamples of Subdivision Methods

Here, we show two counterexamples for subdivision methods, in which subdivision methods will report an extraneous root. The first example is of a 1D system. The second example is of a 2D system.

Example 3.1 *The first example is a system with one equation and one unknown with degree 2, and is shown in Figure 3-4.*

In Figure 3-4, the convex hull of the control points will always intersect the u-axis at exactly one *mathematical point.* (But, the curve itself can be far away from the u-axis.) Therefore, the Projected-Polyhedron algorithm can never discard that point, no matter how small the tolerance is. Accordingly, in the program, that point will be mistaken as one approximate root. Yet it is not a root or an approximate root.

In [47], the 2D Bézier Clipping method is used to solve a system with two equations and two unknowns (see equation 17 in [47]). They convert the two equations $D_1 = D_2 = 0$ into two explicit Bézier surfaces with parameters (s,t) in (s,t,D) coordinate system. To find the root in s direction, they project these two explicit Bézier surfaces onto s-d plane, and intersect their convex hulls with s-axis. If the intersection interval satisfies tolerance conditions, then take that interval region as a root, (see [47], section 3.2 page

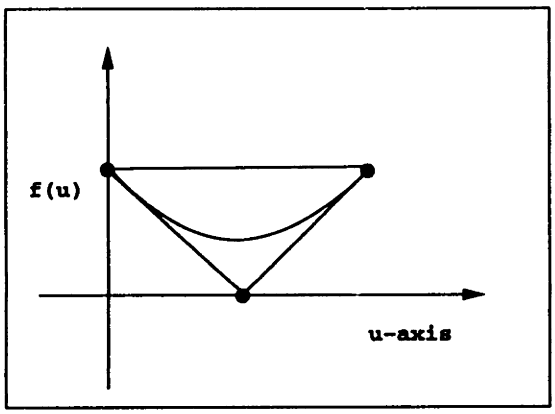


Figure 3-4: A convex hull of control points intersects u-axis at one point

$d^1_{i,j}$	j=0	j=1	j=2
i = 0	-1	-2	-2.5
i = 1	-1	-2	-2.5
i = 2	0	-1	-1.5

$d^2_{i,j}$	j=0	j=1	j=2
i = 0	-0.6	-2	-2
i = 1	0.4	-1	-1
i = 2	0.4	-1	-1

Table 3.1: Data of $d^1_{i,j}$ and $d^2_{i,j}$

342). Otherwise, they clip these two explicit surfaces and repeat the procedure again, until no intersection area is left or the intersection area satisfies tolerance condition. The same procedure is applied to find root in t-direction.

Example 3.2 *Here is a 2D example, with two equations and two unknowns in Bernstein form:*

$$D^1(s,t) = \sum_{i=0}^{2}\sum_{j=0}^{2} B_i^2(s)B_j^2(t)d^1_{i,j} = 0 \tag{3.11}$$

$$D^2(s,t) = \sum_{i=0}^{2}\sum_{j=0}^{2} B_i^2(s)B_j^2(t)d^2_{i,j} = 0 \tag{3.12}$$

where the data of $d^1_{i,j}$ and $d^2_{i,j}$ are shown in Table 3.1.

To find the root for the system, we used two methods: Bézier Clipping and Bézier Clipping with *convex-hull-cross-axes* check.(It is denoted by *BCWCHCA* and incorporated in Step 4 in Section 3.2). The result is shown in Table 3.2.

	Bézier Clipping Method	*BCWCHCA*
root	(s,t) = (1.0, 0.0)	No Root

Table 3.2: Results of two methods used to find the root of the system for a counterexample.

When we substitute $(s,t) = (1.0, 0.0)$, the solution set found by the Bézier Clipping method, back into the system, we find that

$$D^1(1.0, 0.0) = 0.0$$

$$D^2(1.0, 0.0) = 0.4$$

Obviously, $(s,t) = (1.0, 0.0)$ is not a root for the system.

The Bézier Clipping method mistook $(s,t) = (1.0, 0.0)$ for a root, while the *convex-hull-cross-axes* check screens out this root.

3.4.3 Analysis of the Counterexamples

Here we analyze the reason why the Bézier Clipping method fails to identify roots correctly and how our proposed *convex-hull-cross-axes* check works properly.

Let us analyze s-direction first, by referring to Figure 3-5. The convex hull of the projection $\mathbf{g}^1$ on s-d plane is shown in Figure 3-5 (1). It only intersects the s-axis at exactly one point $s = 1.0$. Similarly, the convex hull of the projection $\mathbf{g}^2$ on s-d plane is shown in Figure 3-5(3). It intersects with the s-axis with an interval range containing $s = 1.0$. When intersecting these two convex hulls at the s-axis, one gets a resulting intersection set as one point $\{s \mid s = 1.0\}$. Therefore, no matter how small the tolerance ϵ is, the resulting intersection set will *satisfy the tolerance condition.*

Similarly, for the t-direction, the resulting intersection set of two convex hulls and t-axis is $\{t \mid t = 0.0\}$. Again, no matter how small the tolerance is set, it will satisfy the tolerance condition. Hence the Bézier Clipping method will treat $(s,t) = (1.0, 0.0)$ as a root.

Indeed, the result will nullify $D^1(s,t)$, equation 3.11. When we use the *convex-hull-cross-axes* check, $D^2(s,t)$ will fail to pass the check. That is because d values of the control points of the chopped explicit surface $\mathbf{g}^2$ are (0.4), which will not pass either the s-axis or the t-axis. So, this result is discarded.

3.5 Implementation of Convex-Hull-Cross-Axes Check

This section describes the implementation of the *Convex- hull-cross-axes* check. The check is to confirm *where the roots are.*

The basic idea behind convex-hull-cross-axes method is to check the convex hull of the control points of each *chopped* hypersurface corresponding to the ϵ-boxes to determine

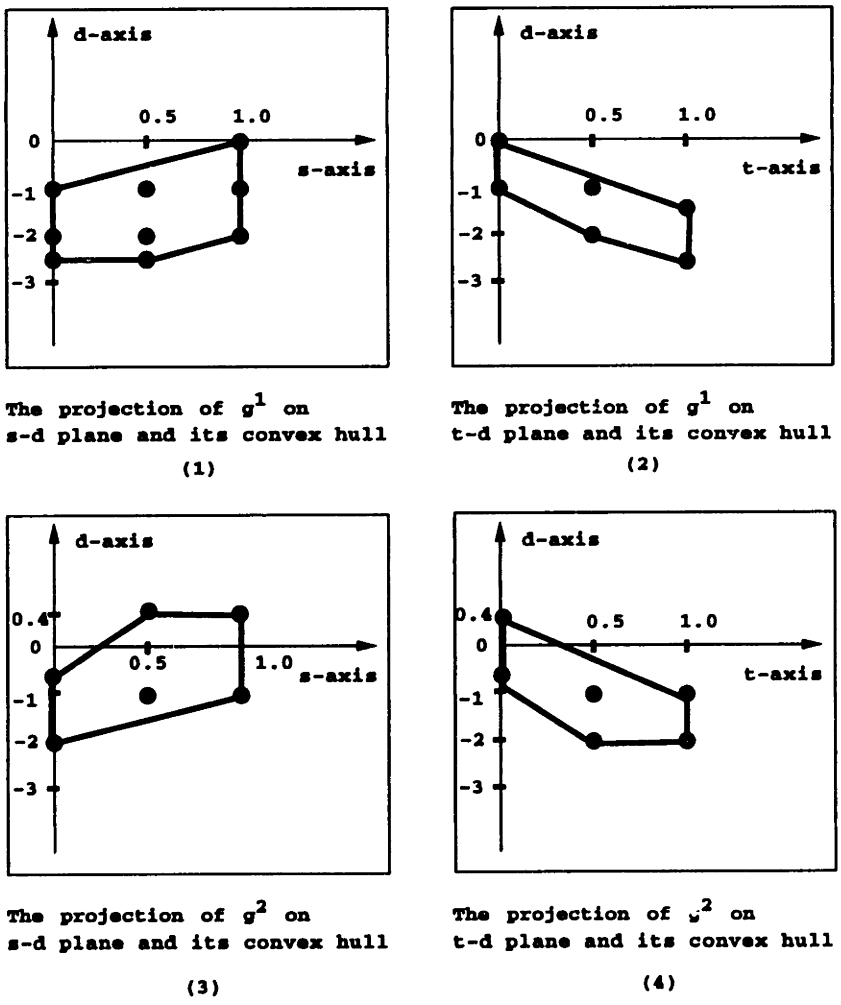


Figure 3-5: Projections of explicit surfaces g^1 and g^2 and their convex hulls.

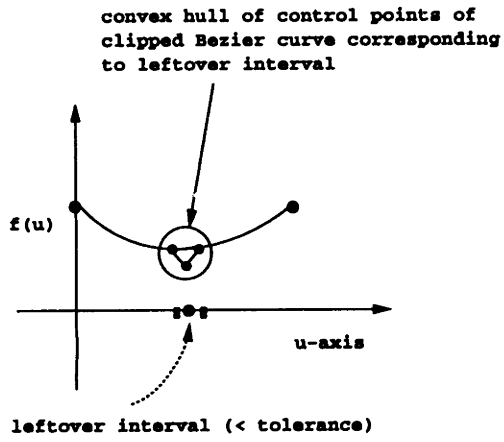


Figure 3-6: Although the leftover region is smaller than the tolerance, the corresponding convex hull of the associated chopped Bézier curve does not cross the u-axis.

whether each projected convex hull intersects each i-axis for each dimension.

At the final stage of the Projected-Polyhedron algorithm, the ϵ-box, with size of interval in each dimension smaller than the tolerance ϵ, is held. According to the De Casteljau algorithm [16], the control points of each chopped hypersurface (of each equation in the non-linear polynomial system) corresponding to the bounding box could be computed. Project control points of the *chopped* Bézier hypersurface onto (x_i, x_{m+1}) plane, for $0 \leq i \leq m$. Check whether the convex hull of those control points crosses x_i axis. We claim that if it crosses **all** of the x_i axis, then ϵ-box contains at least **approximate roots** or true roots. Note that we do not claim ϵ-box guarantee to contain *actual roots* because in the example of Figure 3-3(3), ϵ-box can only contain approximate root. We prove this claim in section 3.6.2.

The implementation is to check whether the ordinates of the projected 2D convex hull for the i-th dimension are not all positive or negative. If they are not all positive or positive, then that convex hull crosses the x_i axis. Otherwise, that convex hull does not cross the x_i axis. If edges of the projected convex hull do cross each i-th axis, then the ϵ-box contains at least *approximate roots*. Otherwise, it contains no root at all and should be discarded.

Take the Bézier curve in Figure 3-4 for instance. The leftover region is of a size smaller than the tolerance. Nevertheless the convex hull of the control points of the chopped curve does not cross the u-axis at all. Therefore, it will be screened out by the *Convex-hull-cross-axes* check. See Figure 3-6 for detail.

3.6 Correctness of the Convex-Hull-Cross-Axes Check

In this section, we prove the correctness of the convex-hull-cross-axes check. Essentially, we prove that, for a system with m equations and m unknowns, an ϵ-box contains at least an

approximate root, if the ϵ-box passes the convex-hull-cross-axes check for all dimensions. Otherwise, ϵ-box contains no roots and should be discarded.

First, Section 3.6.1 presents Lemmas 3.1 and 3.2 to prove the correctness of the convex-hull-cross-axes check for 1D systems. Lemma 3.2 has a tighter bound than Lemma 3.1. However, they use different approaches for proof: Lemma 3.2 uses mathematical analysis methods presented in [67] by Sherbrooke and Patrikalakis; Lemma 3.1 employs the geometry of Bézier hypersurfaces. Hence, we show both lemmas.

Then, Section 3.6.2 presents Theorems 3.1 and 3.3 to prove the correctness of the convex-hull-cross-axes check for m-D systems. Like Lemmas 3.1 and 3.2, Theorem 3.3 has a tighter bound than Theorem 3.1. For the same reason as the one for Lemmas 3.1 and 3.2, we show both theorems.

For the convenience of the proofs, let us first define the following notation. Let $M = \{1, \ldots, m\}$. According to reference [67], to solve a system in Bernstein form with m equations $f^i = 0, i \in M$ and m unknowns $U = (u_1, \ldots, u_m)$,

$$f^j(U) = \sum_{i_1=0}^{n_1^j} \cdots \sum_{i_m=0}^{n_m^j} p^j_{i_1,\ldots,i_m} B_{i_1,n_1^j}(u_1) \cdots B_{i_m,n_m^j}(u_m) = 0 \qquad j \in \{1, \ldots, m\} \tag{3.13}$$

we first convert each equation $f^j(U) = 0$ of the system into a graph $\mathbf{g}^j = (U, f^j)$. These graphs $\mathbf{g}^j, j \in M$ are represented by hypersurfaces:

$$\mathbf{g}^j = (U, f^j(U)) = \sum_{i_1=0}^{n_1^j} \cdots \sum_{i_m=0}^{n_m^j} \mathbf{v}^j_{i_1,\ldots,i_m} B_{i_1,n_1^j}(u_1) \cdots B_{i_m,n_m^j}(u_m) \quad j \in \{1, \ldots, m\} \tag{3.14}$$

where $\mathbf{v}^j = (\frac{i_1}{n_1^j}, \frac{i_2}{n_2^j}, \ldots, \frac{i_m}{n_m^j}, p^j_{i_1,\ldots,i_m})$. Note that $\mathbf{g}^j$ is a hypersurface in $(m+1)$-D space $\mathbf{R}^{m+1}$. Let us denote each axis of $\mathbf{R}^{m+1}$ as i-th axis for $i \in M$ and f-axis for $i = m+1$. For each i-th axis, $i \in M$, the control points $\mathbf{v}^j$ of all m hypersurfaces are projected into the plane W_i spanned by i-th axis and f-axis. Then the m convex hulls of those projected control points are intersected with an i-th axis to find the bounding boxes to bound the solutions.

3.6.1 Proof for One-Dimensional Systems

We start with the simplest system, one equation with one unknown.

Let $f(u) = 0$ be a non-linear polynomial equation of one variable u in Bernstein basis:

$$f(u) = \sum_{i=0}^{n} p_i B_{i,n}(u) = 0 \tag{3.15}$$

Let $\mathbf{R}$ be the graph $(u, f(u))$:

$$\mathbf{R}(u) = \sum_{i=0}^{n} \mathbf{P}_i B_{i,n}(u) \tag{3.16}$$

where $\mathbf{P}_i = (\frac{i}{n}, p_i)^T$ (T denotes transpose). $\mathbf{R}(u)$ is a 2D Bézier curve as indicated in Figure 3-7.

Lemma 3.1 *For a system with one variable u and one equation expressed in equation 3.15, if an interval* $I_o = (l_o, h_o)$ *left over by the Projected-Polyhedron algorithm with tolerance* $\epsilon > 0$ *and passes the convex-hull-cross-axis check, then there exists a constant* $C > 0$ *such that*

$$-(C * \epsilon) \leq f(I_o) \leq (C * \epsilon) \tag{3.17}$$

This lemma means that $f(I_o)$ will converge to zero according to the tolerance ϵ. As stated before, this lemma does not guarantee I_0 to contain a root for tangential cases, see Figure 3-3(3). The proof for Lemma 3.1 is presented later in this section.

The following lemma has tighter bounds than Lemma 3.1.

Lemma 3.2 *Let* $f(u)$, $I_o = (l_o, h_o)$ *and* $\epsilon > 0$ *be defined as before, then there exists a constant* $C > 0$ *such that*

$$-(C * \epsilon^2) \leq f(I_o) \leq (C * \epsilon^2) \tag{3.18}$$

Proof: Let $\bar{\mathbf{R}}$ be the subdivided graph of $\mathbf{R}$ according to I_o. Let $\bar{\mathbf{R}}$ be represented as follows:

$$\bar{\mathbf{R}}(u) = \sum_{i=0}^{n} \bar{\mathbf{P}}_i B_{i,n}(u) \tag{3.19}$$

where $\bar{\mathbf{P}}_i = (\frac{i}{n}, \bar{p}_i)^T$.

Let $\bar{p}_{max}$ be $max\{\bar{p}_i\}$ and $\bar{p}_{min}$ be $min\{\bar{p}_i\}$. From Theorem 5.8 in [67], we know that there exists a constant $\kappa > 0$,

$$-\kappa\epsilon^2 < f(u) - \bar{p}_{min} < \kappa\epsilon^2 \tag{3.20}$$

$$-\kappa\epsilon^2 < f(u) - \bar{p}_{max} < \kappa\epsilon^2 \tag{3.21}$$

By triangular inequality, we get

$$|\bar{p}_{max} - \bar{p}_{min}| < 2\kappa\epsilon^2 \tag{3.22}$$

Since I_o pass the convex-hull-cross-axes check, so we know

$$\bar{p}_{min} < 0 < \bar{p}_{max} \tag{3.23}$$

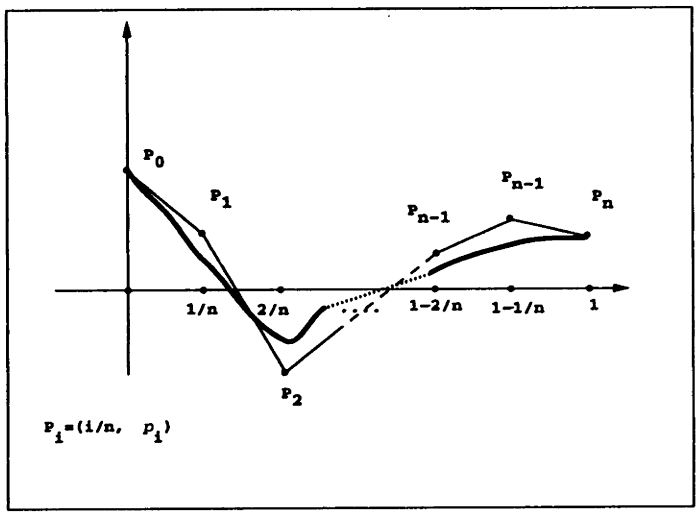


Figure 3-7: The corresponding Bézier curve $\mathbf{R}(u)$ of graph $(u, f(u))$

By convex hull property, we know

$$\bar{p}_{min} < f(u) < \bar{p}_{max} \tag{3.24}$$

From equations 3.22 to 3.24, we have

$$|f(u)| < 2\kappa\epsilon^2 \tag{3.25}$$

We complete the proof by taking $C = 2\kappa$. □

To prove Lemma 3.1, we use the geometrical properties of Bézier hypersurfaces. Let us first define some terminology. We denote the slope of two planar points $\mathbf{P}$ and $\mathbf{Q}$ as $slope(\mathbf{P}, \mathbf{Q})$. Therefore (see Figure 3-7),

$$slope(\mathbf{P}_{i-1},\ \mathbf{P}_i) = \frac{p_i - p_{i-1}}{u_i - u_{i-1}} = n * (p_i - p_{i-1}) \quad for \quad i \in \{1, \ldots, n\} \tag{3.26}$$

The slope of the planar Bézier curve $\mathbf{R}(u)$, denoted by $slope(\mathbf{R})$, is defined by the maximum of the absolute of the slopes of its control polygon lines, i.e. (see Figure 3-7),

$$slope(\mathbf{R}) = max\{|slope(\mathbf{P}_{i-1},\ \mathbf{P}_i)|;\ i = 1, \ldots, n\} \tag{3.27}$$

To subdivide Bézier curves, we apply the de Casteljau algorithm, which is a recursive application of linear interpolations of two points. Let us rewrite the de Casteljau algorithm (operated at a parameter u) for graph $\mathbf{R}(u)$:

$$\mathbf{P}_i^r = (1-u)\mathbf{P}_i^{r-1} + u\mathbf{P}_{i+1}^{r-1} \quad \begin{cases} r = 1, \ldots, \text{n} \\ i = 0, \ldots, \text{n-r} \end{cases} \tag{3.28}$$

where

$$\mathbf{P}_i^0 = \mathbf{P}_i.$$

$\mathbf{P}_i^r = (u_i^r, p_i^r)$ is the r-th level of intermediate points; $\mathbf{P}_i^0 = (u_i^0, p_i^0)$ are original control points and $\mathbf{P}_0^n$ is the point $\mathbf{R}(u)$, see Figure 3-10 for illustration. The slope of the two intermediate points $\mathbf{P}_i^r$ and $\mathbf{P}_{i-1}^r$, based on the fact that $u_i^r - u_{i-1}^r = \frac{1}{n}$, can be easily computed as follows (see Figure 3-10):

$$slope(\mathbf{P}_{i-1}^r, \mathbf{P}_i^r) = \frac{p_i^r - p_{i-1}^r}{u_i^r - u_{i-1}^r} = n * (p_i^r - p_{i-1}^r) \tag{3.29}$$

The following lemma gives the relationship between the slope of a Bézier curve and its subdivision curves.

Lemma 3.3 *Let $\mathbf{P}_i$ be the control points of graph $\mathbf{R}(u) = (u, f(u))$, in equation 3.16. With the de Casteljau algorithm, $\mathbf{R}(u)$ is subdivided at $u = u_0$, $0 < u_0 < 1$. Let $\mathbf{G}_1(u)$ and $\mathbf{G}_2(u)$ be the two resulting subdivided graphs; $\mathbf{G}_1(u)$ corresponds to variable ragne $u : [0, u_0]$, and $\mathbf{G}_2(u)$ to $u : [u_0, 1]$ (see Figure 3-8). Then the slope of $\mathbf{G}_1$ and $\mathbf{G}_2$ are less than or equal to the slope of $\mathbf{R}$, i.e.,*

$$slope(\mathbf{G}_1) <= slope(\mathbf{R}); \qquad slope(\mathbf{G}_2) <= slope(\mathbf{R})$$

Proof:
We prove the first claim. The second one can be similarly proven. If $\mathbf{R}$ is a line, then the slopes of $\mathbf{R}$ and $\mathbf{G}_1$ are the same slope of the line. In this case, the lemma holds trivially. The following proof is for the case of $n > 1$.

Let $\mathbf{P}_i^r$ be the intermediate control points from $\mathbf{R}(u_0)$, defined in equation 3.28 (see Figure 3-8). We can prove this lemma by proving the following claim: (see Figure 3-8)

$$|slope(\mathbf{P}_{i-1}^r, \mathbf{P}_i^r)| \leq max\{|slope(\mathbf{P}_{i-1}^{r-1}, \mathbf{P}_i^{r-1})|,\ |slope(\mathbf{P}_i^{r-1}, \mathbf{P}_{i+1}^{r-1})|\} \tag{3.30}$$

The equality of equation 3.30 occurs when $slope(\mathbf{P}_{i-1}^{r-1}, \mathbf{P}_i^{r-1}) = slope(\mathbf{P}_i^{r-1}, \mathbf{P}_{i+1}^{r-1})$. [2]

We can verify the claim of equation 3.30 by the following self-evident fact: given any three points, (see Figure 3-9), **A**, **B** and **C**, distributed along x-axis in 2D, pick one point **D** on $\overline{AB}$ excluding end points (i.e., **A** and **B**) and another point **E** on $\overline{BC}$ excluding end points; then the slope of segment $\overline{\mathbf{CD}}$ will be between the slopes of segments $\overline{\mathbf{AB}}$ and $\overline{\mathbf{BC}}$; i.e. (see Figure 3-9),

$$min\{|slope(\overline{\mathbf{AB}})|, |slope(\overline{\mathbf{BC}})|\} \leq |slope(\overline{\mathbf{DE}})| \leq max\{|slope(\overline{\mathbf{AB}})|, |slope(\overline{\mathbf{BC}})|\}$$

Therefore

[2] The equality of equation 3.30 will also occur if $u_0 = 0.0$ or $= 1.0$, which has been excluded by the hypothesis.

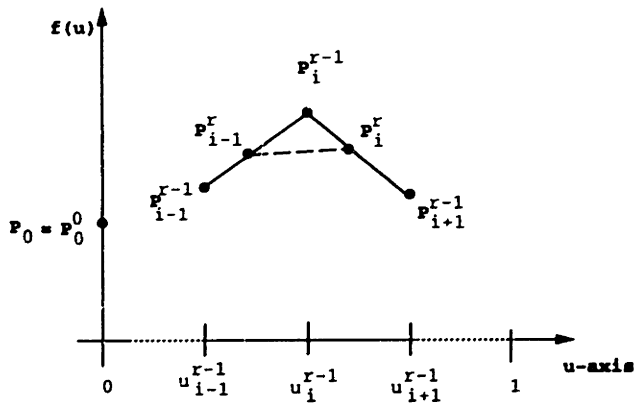


Figure 3-8: The slopes of $(\mathbf{P}_{i-1}^{r-1}, \mathbf{P}_i^{r-1})$, $(\mathbf{P}_i^{r-1}, \mathbf{P}_{i+1}^{r-1})$ and $(\mathbf{P}_{i-1}^{r}, \mathbf{P}_i^{r})$.

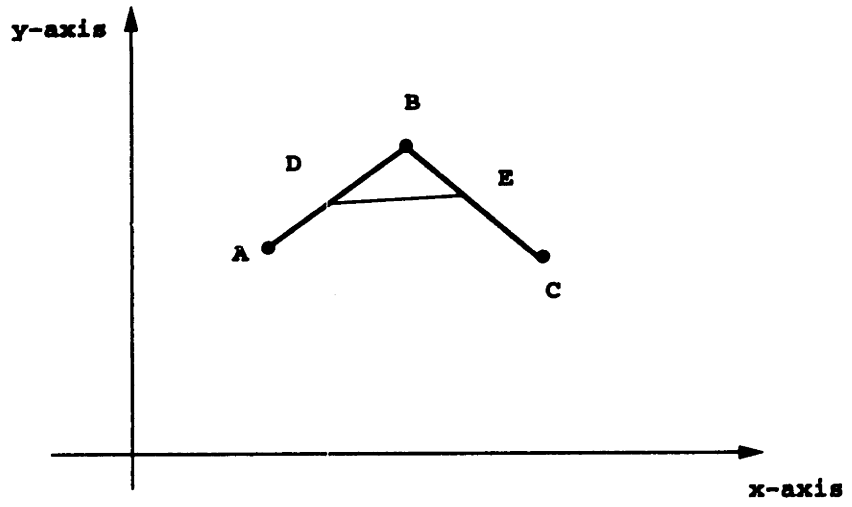


Figure 3-9: The slope of $\overline{\mathbf{DE}}$ lies between the slopes $\overline{\mathbf{AB}}$ and $\overline{\mathbf{BC}}$.

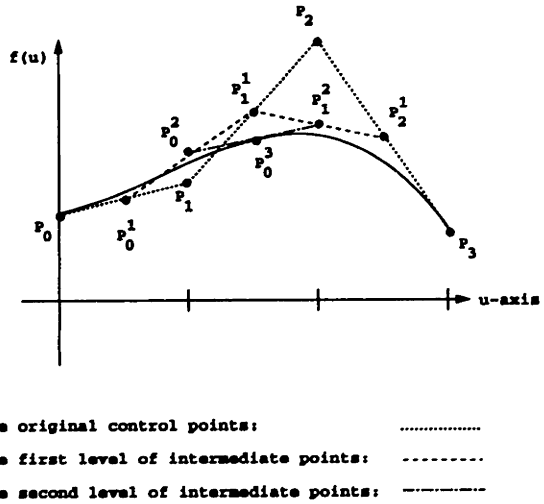


Figure 3-10: The slopes of the subdivided graphs are equal to or smaller than slope of the original graph.

$$|slope(\overline{\mathbf{DE}})| \leq max\{|slope(\overline{\mathbf{AB}})|,\ |slope(\overline{\mathbf{BC}})|\}$$

Hence the claim of equation 3.30 is proven. □

The above lemma also indicates that the control points of a subdivided Bézier hypersurface will fluctuate less along u-axis than the original control points, see Figure 3-10 for illustration.

We now prove Lemma 3.1 for the correctness of convex-hull-axes-check algorithm.

Proof of Lemma 3.1:

It is obvious the slope of $\mathbf{R}(u) = (u, f(u))$, in equation 3.16, defined in equation 3.26. is less than or equal $2nP$, where $P = max\{|\ p_i\ |\}$. This can be seen clearly from equation 3.27.

Let $\mathbf{Q}(u) = \sum_{i=0}^{n} (\gamma_i, q_i)^T B_{i,n}(u)$ be the chopped 2D Bézier curve in the leftover interval $I = [l, h]$. Then, from Lemma 3.3 the slope of $\mathbf{Q}(u), u \in [l, h]$ is no greater than $2n * p$.

$$slope(\mathbf{Q}) \leq slope(\mathbf{R}) \leq 2nP \tag{3.31}$$

By the hypothesis, the leftover interval $I = (h, l)$ has width less than ϵ, i.e.,

$$(l - h) \leq \epsilon \tag{3.32}$$

From equations 3.31 and 3.32, we conclude that the control points of the chopped 2D Bézier curve has the ordinate span of no greater than $\epsilon * 2nP$, i.e. (see Figure 3-12),

$$max\{q_i\} - min\{q_i\} \leq \epsilon 2nP \tag{3.33}$$

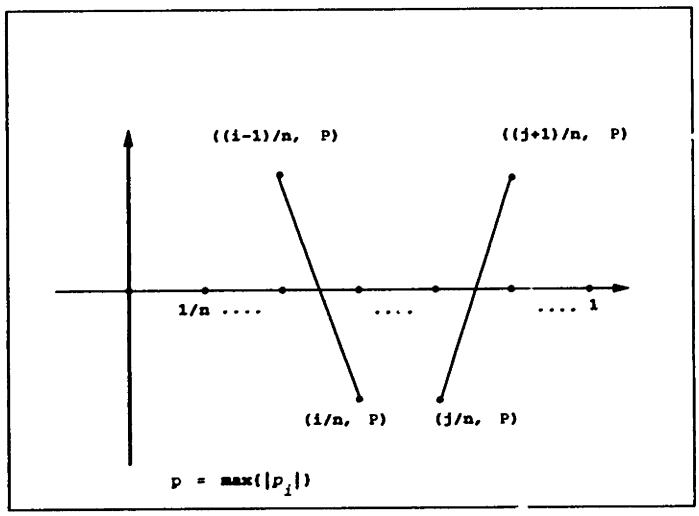


Figure 3-11: The maximum value of slope of graph $(u, f(u))$

By convex hull property, we know that

$$min\{q_i\} \leq f(I_0) \leq max\{q_i\} \tag{3.34}$$

The assumption that I passes the convex-hull-cross-axis check implies that

$$min\{q_i\} \leq 0 \leq max\{q_i\} \tag{3.35}$$

From equations 3.32 to 3.34, we have

$$| f(I_0) | \leq \epsilon C \tag{3.36}$$

where $C = 2nP$. We complete the proof for Lemma 3.1□

The following Lemma proves that for an interval region which does not pass the convex-hull-cross-axis check, there is no root inside that region.

Lemma 3.4 *Let* $\mathbf{R}(u)$ *be the 2D Bézier curve in equation 3.15. An interval* $I = (l, h) \subset [0, 1]$ *is a leftover region by the Projected-Polyhedron method and does not pass the convex-hull-cross-axis check, then*

$$f(I) > 0 \quad or \quad f(I) < 0 \tag{3.37}$$

Proof: This lemma can be easily provided by the convex hull property of Bézier curves[16]. The fact that $I = [l, h]$ does not pass the convex-hull-cross-axis check indicates that the coordinates of the convex hull of the chopped 2D Bézier curve according to the Interval I $\mathbf{Q}(u)$ are all positive or negative. Since convex hulls envelops their corresponding Bézier curve, $f(I)$ is entirely positive or negative.□

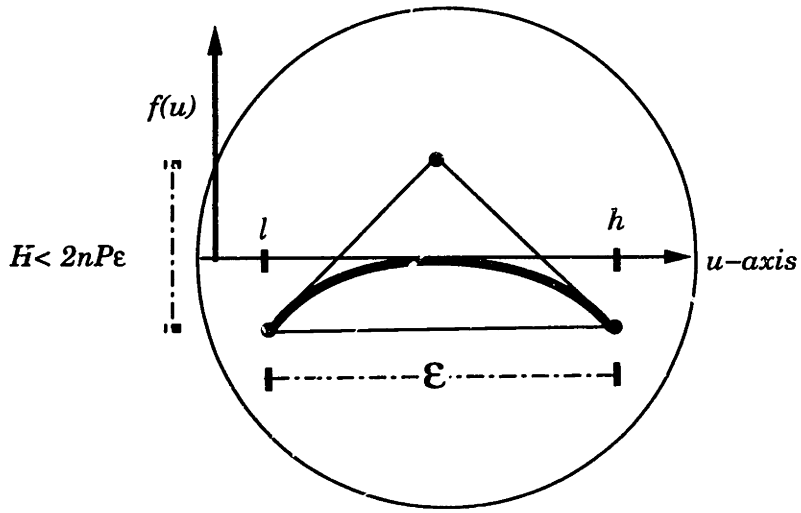


Figure 3-12: Magnified area around I: $|f(I) - 0| < \epsilon 2nP$

3.6.2 Proof for m Dimension Systems

In this section, similar to the previous section 3.6.1, we present two theorems to prove that the ϵ-boxes left over by the Projected-Polyhedron algorithm indeed contains at least an approximate root for a system with m equations with m unknowns. Theorem 3.3 has a tighter bound than Theorem 3.1. However, they use different approaches for proof: The proof for Theorem 3.3 is a direct extension of the one for Lemma 3.2 based on mathematical analysis method [67] by Sherbrooke and Patrikalakis. Theorem 3.1 emploies the geometry of Bézier hypersurfaces. Therefore we present both theorems.

The *convex-hull-cross-axes* check is used to complete subdivision methods, such as Projected-Polyhedron algorithm and Bézier Clipping method. Here, we prove that any ϵ-box which satisfies the convex-hull-cross axes condition contains at least approximate roots (this is used to identify locations of the roots). We also prove that any ϵ-Box which does not pass the convex-hull-cross-axes check will never contain a root, no matter how small the tolerance ϵ is, (this is used to further discard regions not containing a root, in Step 4 of IPP algorithm in Section 3.2).

Theorem 3.3 *For a system with m unknowns $U = (u_i)$ $i \in \{1, \ldots, m\}$ and m equations*

$$f^j(U) = \sum_{i_1=0}^{n_1^j} \cdots \sum_{i_m=0}^{n_m^j} p^j_{i_1,\ldots,i_m} B_{i_1,n_1^j}(u_1) \cdots B_{i_m,n_m^j}(u_m) = 0$$

where $j \in \{1, \ldots, m\}$. If an ϵ-box $\mathbf{B}^m = ((l_i, h_i))_{i \in \{1,\ldots,m\}}$ passes the convex-hull-cross-axes check for each dimension, then there exists a constant $C > 0$ such that

$$-(C * \epsilon) \leq f^j(B^m) \leq (C * \epsilon) \quad for \quad j \in \{1, \ldots, m\} \tag{3.38}$$

Proof: Let $\mathbf{g}^j$ be the hypersurface of f^j in equation 3.14.
We first prove that for each k-th axis $k \in M = \{1, \ldots, m\}$, there exists an constant $C_k > 0$ such that

$$-(C_k * \epsilon) \leq \pi_f(\mathbf{g}^j(B^m)) \leq (C_k * \epsilon) \quad j \in M \tag{3.39}$$

where π_f is the projection function of $(m+1)$D hypersurface $\mathbf{g}^j$ on f-axis.

For an k-th axis, $k \in M$, and for each j-th hypersurface $\mathbf{g}^j$: let cg_k^j denote the projection of the control points of $\mathbf{g}^j$ onto the plane W_k spanned by k-th axis and f-axis. cg_k^j contains points spread on the 2D plane W_k. However, those points will not be scattered all over W_k without any order. Rather, they are posed on node lines $\{u_{k,l}^j = \frac{l}{n_k^j}\}_{l \in \{0,\ldots,n_k^j\}}$ on W_k. On one node line $u_{k,j}^l = \frac{l}{n_k^j}$, there are control points $S_{k,l}^j = \{p_{i_1,\ldots,i_m}^j \| i_k = l\}$ on it, see Figure 3-5 for examples. The maximum absolute value of slopes of these line segments on the span between $u_{k,l}^j = \frac{l}{n_k^j}$ and $u_{k,l+1}^j = \frac{l+1}{n_k^j}$ will not be greater than $2n_k^j P$, where $P = max\{|\ p_{k_1,\ldots,k_m}^j\ | : j, k_i, l \in \{1, \ldots, m\}\}$, since $|\ \frac{(p_{l+1} - p_l)}{\frac{1}{n_k^j}}\ | \leq (P + P) * n_k^j$, where $p_{l+1} \in S_{k,l+1}^j, p_l \in S_{k,l}^j$.

Let $CH_k(\mathbf{g}^j)$ be the set of line segments between consecutive points in cg_k^j, see Figure 3-5 for examples. The slope of $CH_k(\mathbf{g}^j)$, denoted by $slope_k^j$, is defined by the maximum of the slopes of all the line segments in it. It is obvious that $slope_k^j \leq 2n_k^j P$. Therefore, the maximum absolute value of the slopes of m sets $CH_k(\mathbf{g}^j)$, denoted by $slope_k = max\{slope_k^j\}_{j \in M}$, is not greater than $2n_k P$, where $n_k = max\{n_k^1, \ldots, n_k^m\}$.

Let $\overline{\mathbf{g}}_{B^m}^j$ be the chopped j-th graph $\mathbf{g}^j$ according to B^m. Note that this implies

$$\pi_f(\overline{\mathbf{g}}_{B^m}^j) = \pi_f(\mathbf{g}^j(B^m)) \tag{3.40}$$

Let $\overline{CH}_{k,B^m}^j$ be the set of line segments between consecutive points in the projection of $\overline{\mathbf{g}}_{B^m}^j$ on W_k. We claim that for each k-th graph g^j, the slope of $\overline{CH}_{k,B^m}^j$ is not greater than that of $CH_k(\mathbf{g}^j)$. Like the argument in Lemma 3.3, by the property of Bézier hypersurface, the control points of a subdivided Bézier hypersurface will approximate the hypersurface more than the original control points. Therefore, the control points of a subdivided Bézier hypersurface will fluctuate less along each k-axis than the original control points. Hence, the slope of $\overline{CH}_{k,B^m}^j$ is smaller than that of $CH_k(\mathbf{g}^j)$, which is not greater than $2n_k P$.

Since the slope of $\overline{CH}_{k,B^m}^j$ is not greater than $2n_k P$ (from *Step 2.*), and $\pi_k(B^m) < \epsilon$ (from hypothesis)[3], hence the interval ordinate of $\overline{CH}_{k,B^m}^j$ on plane W_k is less than or equal to $\epsilon * 2n_k P$. Let Q_k be the set of the control points of the projection of $\overline{\mathbf{g}}_{B^m}^j$ on W_k. Then,

[3] $\pi_k()$ is the projection function of $(m+1)$ tuple on k-th axis, $k \in M = \{1, \ldots, m\}$

the ordinates of points in $\overline{CH}^j_{k,B^m}$ on W_k are $O^j_k = \{\pi_f(q_k) || q_k \in Q_k\}$. This means that

$$\overline{h}^j_k - \overline{l}^j_k \leq \epsilon * 2n_k P \tag{3.41}$$

where $\overline{l}^j_k = min\ O^j_k$ and $\overline{h}^j_k = max\ O^j_k$. By convex hull property, we know that

$$\overline{l}^j_k \leq \pi_f(\overline{\mathbf{g}}^j_{B^m}) \leq \overline{h}^j_k \tag{3.42}$$

The hypothesis that B^m passes the convex-hull-cross-axis check means that

$$\overline{l}^j_k \leq 0 \leq \overline{h}^j_k \tag{3.43}$$

Therefore, from equations 3.41, 3.42 and 3.43, we have

$$-(\epsilon * 2n_k P) \leq \pi_f(\overline{\mathbf{g}}_{B^m}) \leq (\epsilon * 2n_k P) \tag{3.44}$$

By taking $C_k = 2n_k P$, and from equations 3.40 and 3.44 we obtain equation 3.39.

We now complete the proof of this theorem. Since equation 3.39 is valid for each $k \in M$, we take $C = max\{C_k\}_{k \in M}$, then

$$-(C * \epsilon) \leq \pi_f(\mathbf{g}^j(B^m)) \leq (C * \epsilon) \quad for \quad j \in \{1, \ldots, m\}$$

From equation 3.14, we know that $f^j(B^m) = \pi_f(\mathbf{g}^j(B^m))$.□

Theorem 3.2 *For a system with m unknowns $U = (u_i)$ $i \in \{1, \ldots, m\}$ and m equations*

$$f^j(U) = \sum_{i_1=0}^{n^j_1} \cdots \sum_{i_m=0}^{n^j_m} p^j_{i_1,\ldots,i_m} B_{i_1,n^j_1}(u_1) \cdots B_{i_m,n^j_m}(u_m) = 0 \tag{3.45}$$

where $j \in \{1, \ldots, m\}$. If an interval region $I^m = ((l_i, h_i))_{i \in \{1,\ldots,m\}} \subset [0,1]^m$ does not pass the convex-hull-cross-axes check, then there exists j, such that

$$f^j(I^m) > 0 \quad or \quad f^j(I^m) < 0 \ j \in M \tag{3.46}$$

This theorem is a direct extension of Lemma 3.4 to the (m+1)D case. It can be similarly proven.

Theorem 3.3 *Let U, $f^j(U)$, $\mathbf{B}^m$ be defined as in Theorem 3.1, then there exists a constant $C > 0$ such that*

$$-(C * \epsilon^2) \leq f^j(B^m) \leq (C * \epsilon^2) \quad for \quad j \in \{1, \ldots, m\} \tag{3.47}$$

The theorem is a direct extension of Lemma 3.2 and can be similarly proven.

Chapter 4

Robust Unified Intersection Algorithm

4.1 Introduction

The solutions for geometric intersections of different entities are essential to Boolean operations. In this chapter, we discuss our robust algorithm to solve the following geometric intersection problems: point-to-point, point-to-curve, point-to-surface, curve-to-curve, curve-to-surface, and surface-to-surface problems. In the remaining chapters, geometric entities are interval objects, unless explicitly mentioned otherwise.

Based on the improved and extended IPP solver, this Chapter develops a general unified algorithm to solve various types of intersection problems for both well-conditioned and ill-conditioned cases. The well-conditioned problems include the computation of transversal intersections. The ill-conditioned problems include tangential intersections and overlapping.

This chapter is organized as follows. Section 4.2 presents the general unified algorithm for geometrical intersection problems which can be converted to balanced or overconstrained systems. Section 4.3 defines the incidence for interval objects, such as point-to-point incidence. It also resolves two geometrical failures stemming from floating point arithmetic. Section 4.4 discusses point-to-curve intersection. Section 4.5 discusses point-to-surface intersection. Section 4.6 discusses planar curve-to-curve intersection. It also demonstrates the advantage of the use of overconstrained system solver. Section 4.7 discusses 3D curve-to-curve intersection. Section 4.8 discusses curve-to-surface intersection, which is usually employed to find the starting point for surface-to-surface intersection. Section 4.9 discusses surface-to-surface intersection, which crucial to solid modeling for curved objects. Section 4.9 also resolves surface overlapping problem. To solve the surface overlap, a theorem is developed, referred to as *End Point Theorem*, to describe the condition for two surfaces tangentially intersecting along a non-closed curve. From the *End Point Theorem*, a corollary is derived to describe the condition for surface overlap.

	point	curve	surface
point	p-p		
curve	p-c	c-c	
surface	p-s	s-c	s-s

Table 4.1: Intersections of geometric objects of different dimensions.

4.2 A General Unified Algorithm for Intersection Problems

Geometric entities include points, lines/curves, and polygons/surfaces for solid modeling. Since we deal with curved objects, we only discuss curves and surfaces in this thesis. The types of their intersection problems are listed in Table 4.1. Geometrical intersection problems can usually be converted to the computation of the solution for non-linear polynomial equation systems. All non-trivial intersection problems, p-c, p-s, c-c, and s-c in Table 4.1, are either balanced or overconstrained except for surface-to-surface intersections, which is underconstrained.

Usually, various algorithms are used to solve various geometrical intersection problems. Therefore, geometry engines for solid modeling systems are full of different routines for different algorithms for different intersection problems. This will enlarge the overall modeling systems and complicate the software maintenance. In addition, surface intersection problems alone are difficult enough. Hence, very few geometry engines for curved object intersections in the existing solid modeling systems are well developed in the commercial CAD industry.

Here, we provide a solution to untangle the complication for various intersection problems: the general unified algorithm (GUA). This GUA can solve various intersection problems by using one solver, as long as those intersection problems can be formulated to balanced or overconstrained polynomial equation systems. Sections 4.3 to 4.9 discuss various formulations of different intersection problems.

The GUA uses the improved and extended IPP solver, developed in Chapter 3, as a kernel for the computation of non-linear polynomial equation systems. It only requires different routines to formulate different intersection problems into non-linear polynomial equation systems, but uses one solver to solve them. Section 4.2.1 describes the GUA in detail and Section 4.2.2 summarizes the advantages of the GUA.

4.2.1 The General Unified Algorithm

The GUA adopts the following two steps to solve a geometrical interrogation problem P:

1. Derive an interrogation system W_P of non-linear polynomial equations with n equations and m unknowns, which are sufficient and necessary for P;

2. Solve the interrogation system W_P directly.

As mentioned before, the GUA employs the improved and extended IPP solver, developed in Chapter 3, as a kernel for the computation of non-linear polynomial equation systems. Based on the IPP solver, we devise a high level pseudo-code for the GUA to solve all those p-c, p-s, c-c, and s-c intersection problems (see Table 4.1):

intersect_two_geometries(geometry geom_1, geometry geom_2)
1. ▷ assuming geom_1 is of no higher dimension than geom_2
2. if geom_1 and geom_2 are both points
3. system_of_polynomials = **formulation_pp**(geom_1, geom_2);
4. else if geom_1 is a point and geom_2 is a curve
5. system_of_polynomials = **formulation_pc**(geom_1, geom_2);
6. else if geom_1 is a point and geom_2 is a surface
7. system_of_polynomials = **formulation_ps**(geom_1, geom_2);
8. else if geom_1 is a curve and geom_2 is a curve
9. system_of_polynomials = **formulation_cc**(geom_1, geom_2);
10. else if geom_1 is a curve and geom_2 is a surface
11. system_of_polynomials = **formulation_cs**(geom_1, geom_2);
12. else if geom_1 and geom_2 are both surfaces
13. system_of_polynomials = **formulation_ss**(geom_1, geom_2);
14. roots = **solve_by_IPP**(systems_of_polynomials);
15. return roots;

In the above pseudo-code, **formulation_pp**() is a subroutine that formulates the intersection system of two points. Similarly, **formulation_pc**() is for a point and a curve; **formulation_ps**() is for a point and a surface; **formulation_cc**() is for a curve and another curve; **formulation_cs**() is for a curve and a surface; **formulation_ss**() is for two surfaces. Those formulations are discussed in the following sections. The *system_of_polynomials* can be implemented as a class in C++. The data, such as number of equations, variables, degrees, coefficients of the polynomial system, can be hidden in the class. Finally, the subroutine **solve_by_IPP**() solves a system by calling the IPP solver.

4.2.2 Advantages of the General Unified Algorithm

The GUA has the following advantages that it is (1) numerically robust, (2) of a global solution method, (3) general for intersection problems, and (4) efficient. The listed advantages 1 and 2 are directly inherited from the IPP solver. Here, We want to explain the advantages 3 and 4 in more detail.

Prior to this thesis, there is no scheme to solve overconstrained non-linear polynomial equation systems *directly* and *simultaneously*. There are two standard methods to solve overconstrained systems: one is *minimization method* and the other is *splitting method*. *Minimization method* is to convert overconstrained systems into minimization problems. Thus *minimization method* is not a *direct* method. *Splitting method* is to split overconstrained systems into a series of balanced systems and solve them sequentially. Hence,

splitting method does not solve overconstrained systems *simultaneously.* Different schemes must be adopted to solve balanced and overconstrained systems for existing solid modelers. On the other hand, the GUA can solve balanced and overconstrained systems by one solver. Therefore, the GUA is general for intersection problems.

The GUA is more efficient than minimization and *splitting methods.* Minimization methods require initial approximation value, and they can only find root close to the initial approximation value. On the other hand, GUA does not require initial value, and it can find all roots.

The *splitting method* splits the overconstrained system into several balanced systems and solves them sequentially. In general the first system of the splitting method will contain extraneous roots, since the first system provides a superset of the actual solutions. Take the computation of singular points of surface-to-surface intersection for instance (see Section 4.9.1). The first system provides the collinear normal points. Singular points are collinear normal points, but collinear normal points are not generally singular points. On the other hand, the GUA uses all equations as constraints simultaneously, so it works only on the actual roots.

The idea of GUA can be easily extended to more general geometrical interrogation and computation problems.

4.3 Point-to-Point Intersection

We first define the incidence of points and discuss our solution to *incidence asymmetry* and *incidence intransitivity.*

4.3.1 Incidence of Points

The *incidence* of two interval points is defined by the existence of a real point common to both of them as follows:

Definition 4.1 *Two interval points*

$$\mathbf{A} = ([x_{al}, x_{au}], [y_{al}, y_{au}], [z_{al}, z_{au}])$$

$$\mathbf{B} = ([x_{bl}, x_{bu}], [y_{bl}, y_{bu}], [z_{bl}, z_{bu}])$$

are said to be incident (denoted as $\mathbf{A} = \mathbf{B}$*), if* $\mathbf{A} \cap \mathbf{B} \neq \emptyset$*, that is to say, there is a point* $\mathbf{c} = (x, y, z)$ *such that* $\mathbf{c} \in \mathbf{A} \cap \mathbf{B}$*, or* $x \in [x_{al}, x_{au}] \cap [x_{bl}, x_{bu}]$*,* $y \in [y_{al}, y_{au}] \cap [y_{bl}, y_{bu}]$*, and* $z \in [z_{al}, z_{au}] \cap [z_{bl}, z_{bu}]$.

Hoffmann [23] raised a problem of *incidence asymmetry.* This means that a point can be incident to another point but not vice versa. The above definition of incidence of two interval points will prevent incidence asymmetry of any two interval points. Since, if an interval point $\mathbf{A}$ is incident to an interval point $\mathbf{B}$, then there is a real point $\mathbf{c} \in \mathbf{A} \cap \mathbf{B}$, so $\mathbf{B}$ is also incident to $\mathbf{A}$.

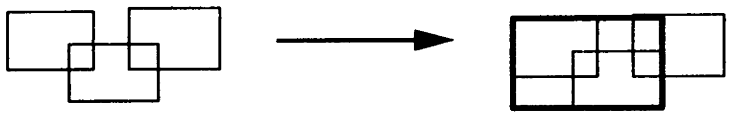


Figure 4-1: Illustration of incidence transitivity of 2-D interval points.

4.3.2 Transitivity of Incidence of Points

One classic geometric failure arising from the floating point error is the *incidence intransitivity*, see Figure 1-3 for an example. However, the incidence intransitivity problem of interval points can be solved by updating the upper bound and lower bound of two incident interval boxes. When two interval points are determined to be incident, we replace these two interval points by a new interval point which is the smallest interval to cover these two incident points. For example, the interval $a = [1.00001, 1.00012]$ is incident to $b = [1.00011, 1.00013]$ since $1.000115 \in a \cap b \neq \emptyset$. As soon as we determine $a = b$, the new interval $c = [1.00001, 1.00013]$ will substitute a and b; i.e., $a' = b' = c = [1.00001, 1.00013]$. Therefore, any interval d incident to a or b is certainly incident to $a' = b' = c$. This process is illustrated in Figure 4-1.

4.4 Point-to-Curve Intersection

Given a point $p = (x_p, y_p, z_p)$, and a parametric polynomial curve $C = C(u) = (x(u), y(u), z(u))$, their intersection is the solution of the following equation system:

$$x(u) = x_p \tag{4.1}$$
$$y(u) = y_p \tag{4.2}$$
$$z(u) = z_p \tag{4.3}$$

This is an overconstrained polynomial equation system and can be solved by the extended IPP solver.

4.5 Point-to-Surface Intersection

Given a point $p = (x_p, y_p, z_p)$, and a parametric polynomial surface patch $S = S(u, v) = (x(u, v), y(u, v), z(u, v))$, their intersection is the solution of the following equation system:

$$x(u, v) = x_p \tag{4.4}$$
$$y(u, v) = y_p \tag{4.5}$$
$$z(u, v) = z_p \tag{4.6}$$

This is an overconstrained polynomial system and can be solved by the extended IPP solver.

4.6 Planar Curve-to-Curve Intersection

For simplicity, the underlying planar interval curve is assumed to be an interval integral Bézier curve.

Extension to the interval rational Bézier curve and the interval non-uniform rational B-spline (INURBS) curve, although tedious and involving higher degree problems during geometric processing, does not present conceptual difficulties. In general, the control points of the given curve are given in single floating point numbers which can be initially treated as zero-width intervals. As the geometric processing proceeds, the widths of the intervals grow gradually so that the interval always contains the exact result, provided that *rounded interval arithmetic* is used for all arithmetic operations.

4.6.1 Transversal Intersection

If we denote the two interval Bézier curves as $\mathbf{p}(u) = (x_p(u), y_p(u))$ and $\mathbf{q}(v) = (x_q(v), y_q(v))$, then the intersection of two interval Bézier curves can be formulated using the identity condition, i.e.,

$$x_p(u) - x_q(v) = 0 \tag{4.7}$$

$$y_p(u) - y_q(v) = 0 \tag{4.8}$$

Equations 4.7 and 4.8 are two simultaneous bivariate polynomial equations, and can be solved by our IPP solver. In the absence of tangential intersection or overlapping of two curves, they are in general well-conditioned and can be solved efficiently.

4.6.2 Tangential Intersection

When the two curves intersect at a point where they have the same tangent, the rate of convergence of the solver [36] drops significantly due to an extensive amount of binary subdivision. An additional equation representing the tangential contact constraint can be added to the system. This additional equation plays an important role in shrinking the width of interval boxes and hence accelerating the convergence rate. It can be expressed as follows:

$$-y_p'(u)x_q'(v) + x_p'(u)y_q'(v) = 0 \tag{4.9}$$

The equation system now becomes overconstrained (with three equations and two unknowns). However, it can be solved by the extended IPP solver. Furthermore, if the two curves intersect each other with second order contact (i.e., same signed curvature) [34], we can further impose the curvature condition and so the system of the equations now involves four equations with two unknowns. The signed curvature of a regular planar curve $\mathbf{r}(t) = (x_r(t), y_r(t))$ is given by

$$\kappa_r(t) = \frac{x_r'(t)y_r''(t) - x_r''(t)y_r'(t)}{((x_r'(t))^2 + (y_r'(t))^2)^{\frac{3}{2}}} \tag{4.10}$$

where the term *regular* signifies that $|\mathbf{r}'(t)| \neq 0$, where $()'$ denotes derivative. Thus, if the two curves $\mathbf{p}(u)$ and $\mathbf{q}(v)$ intersect with second order contact, their signed curvatures should be equal, i.e. $\kappa_p(u) = \kappa_q(v)$:

$$\frac{x'_p(u)y''_p(u) - x''_p(u)y'_p(u)}{[(x'_p(u))^2 + (y'_p(u))^2]^{\frac{3}{2}}} = \frac{x'_q(v)y''_q(v) - x''_q(v)y'_q(v)}{[(x'_q(v))^2 + (y'_q(v))^2]^{\frac{3}{2}}} \tag{4.11}$$

Equation 4.11 involves square roots of polynomials and hence we can not use the convex hull property of the Bernstein polynomial directly to solve the system. However, we can eliminate the square root by squaring out equation 4.11. Therefore equation 4.11 becomes

$$\begin{aligned}[x'_p(u)y''_p(u) - x''_p(u)y'_p(u)]^2[(x'_q(v))^2 + (y'_q(v))^2]^3 = \\ [x'_q(v)y''_q(v) - x''_q(v)y'_q(v)]^2[(x'_p(u))^2 + (y'_p(u))^2]^3\end{aligned} \tag{4.12}$$

Example 4.1 *We illustrate the higher order contact by intersecting a superbola* $y = x^4$ *with the straight line* $y = 0$, *which has third order contact at* $(0,0)$, *see Figure 2-1.*

We solved the intersection problem by treating it respectively as a position, first and second order contact problems, i.e., to solve the system of two variables with two, three and four equations. Figure 4-2 shows the two convex hulls of control points projected in the uw-plane, (w here and in Figure 4-2 represents the value of y). In Figure 4-2, the solid convex hull corresponds to equation 4.7, and the dashed convex hull corresponds to equation 4.8. The asterisk shows the location of the root. It can be seen from the figure that the common region of the two convex hulls on the horizontal axis w=0 extends from u=0.125 to 0.875. Figure 4-3 shows three convex hulls of control points projected in the uw-plane, where the additional dotted convex hull corresponds to equation 4.9. We can see from this figure that the common region of the three convex hulls on $w = 0$ extends from u=0.1667 to 0.8333. Thus the third equation (i.e. the tangent condition) played a role in reducing the initial interval from 75% to 67% of the initial domain. Figure 4-4 shows four convex hulls of projected control points in uw-plane where the additional dot dashed convex hull corresponds to equation 4.12. Now the convex hull based on the fourth equation (i.e. the curvature condition) intersects with w=0 at u=0.175 and 0.825, and reduces the interval from 67% to 65%. Table 4.2 shows the results of our extended IPP algorithm. This particular example was run with a tolerance of 10^{-3}. For presentation purposes only, the intervals are truncated at fourth decimal places in Table 4.2. Note that all the equations are normalized so that $-1 \leq w \leq 1$. If we compare the bounding box for each iteration, we can recognize that the bounding box of four equations is always smaller or equal to that of three equations, and that the bounding box of three equations is always smaller than that of two equations. We can also see from Table 4.2 that it takes 24, 18 and 9 iterations, with two, three and four equations to reach the required tolerance.

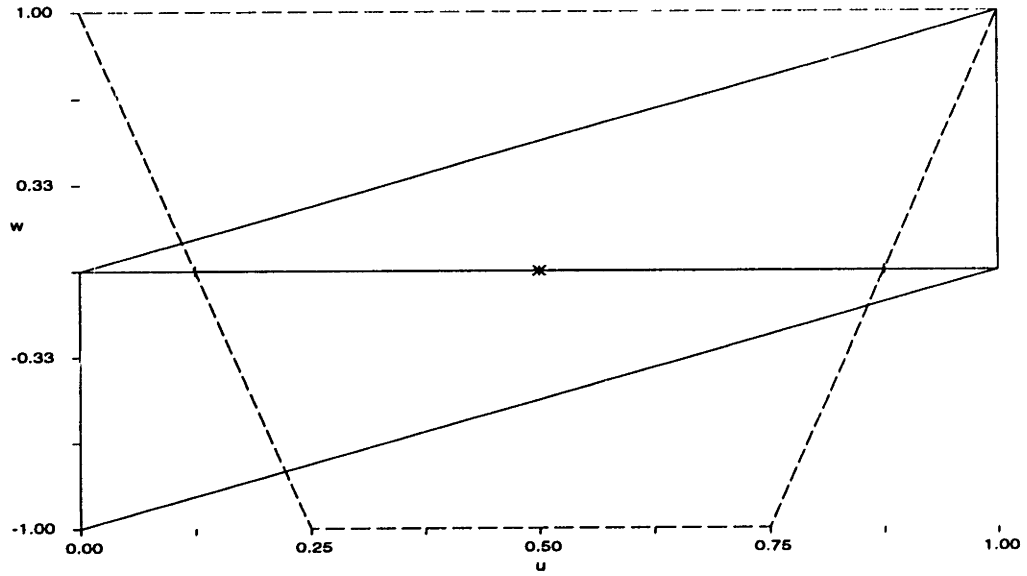


Figure 4-2: Two convex hulls projected onto uw-plane for the example of $y = x^4$ parametrized by u and $y = 0$ by v.

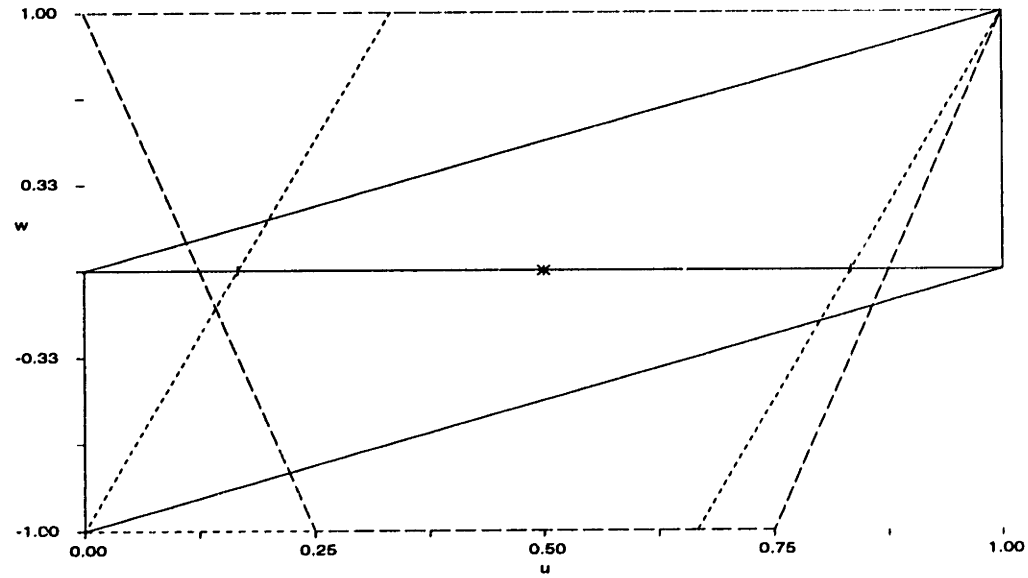


Figure 4-3: Three convex hulls projected onto uw-plane for the example of $y = x^4$ parametrized by u and $y = 0$ by v.

$\epsilon = 10^{-3}$	Two Equations		Three Equations		Four Equations	
Iter	u	v	u	v	u	v
1	[0.0, 1.0]	[0.0, 1.0]	[0.0, 1.0]	[0.0, 1.0]	[0.0, 1.0]	[0.0, 1.0]
2	[0.125,0.875]	[0.5, 1.0]	[0.1666,0.8333]	[0.5, 1]	[0.175, 0.825]	[0.5, 1]
3	[0.5, 0.78125]	[0.5, 0.875]	[0.5, 0.7222]	[0.5, 0.8333]	[0.5, 0.7113]	[0.5, 0.825]
4	[0.5, 0.7109]	[0.5, 0.7812]	[0.5, 0.6481]	[0.5, 0.7222]	[0.5, 0.5634]	[0.5, 0.7113]
5	[0.5, 0.6582]	[0.5, 0.7109]	[0.5, 0.5988]	[0.5, 0.6481]	[0.5, 0.519]	[0.5, 0.5634]
6	[0.5, 0.6186]	[0.5, 0.6582]	[0.5, 0.5658]	[0.5, 0.5988]	[0.5, 0.5057]	[0.5, 0.519]
7	[0.5, 0.5889]	[0.5, 0.6186]	[0.5, 0.5439]	[0.5, 0.5658]	[0.5, 0.5017]	[0.5, 0.5057]
8	[0.5, 0.5667]	[0.5, 0.5889]	[0.5, 0.5293]	[0.5, 0.5439]	[0.5, 0.5005]	[0.5, 0.5017]
9	[0.5, 0.5500]	[0.5, 0.5667]	[0.5, 0.5195]	[0.5, 0.5293]	[0.5, 0.5005]	[0.5, 0.5008]
10	[0.5, 0.5375]	[0.5, 0.5500]	[0.5, 0.513]	[0.5, 0.5195]		
11	[0.5, 0.5281]	[0.5, 0.5375]	[0.5, 0.5087]	[0.5, 0.513]		
12	[0.5, 0.5211]	[0.5, 0.5281]	[0.5, 0.5058]	[0.5, 0.5087]		
13	[0.5, 0.5158]	[0.5, 0.5211]	[0.5, 0.5039]	[0.5, 0.5058]		
14	[0.5, 0.5118]	[0.5, 0.5158]	[0.5, 0.5026]	[0.5, 0.5039]		
15	[0.5, 0.5089]	[0.5, 0.5118]	[0.5, 0.5017]	[0.5, 0.5026]		
16	[0.5, 0.5066]	[0.5, 0.5089]	[0.5, 0.5011]	[0.5, 0.5017]		
17	[0.5, 0.5050]	[0.5, 0.5066]	[0.5, 0.5008]	[0.5, 0.5011]		
18	[0.5, 0.5037]	[0.5, 0.5050]	[0.4999, 0.5008]	[0.5, 0.5009]		
19	[0.5, 0.5028]	[0.5, 0.5037]				
20	[0.5, 0.5021]	[0.5, 0.5028]				
21	[0.5, 0.5015]	[0.5, 0.5021]				
22	[0.5, 0.5011]	[0.5, 0.5015]				
23	[0.5, 0.5008]	[0.5, 0.5011]				
24	[0.5, 0.5008]	[0.5, 0.5009]				

Table 4.2: The box shrinking processes of three methods for intersection between $y = x^4$ parametrized by u and $y = 0$ by v.

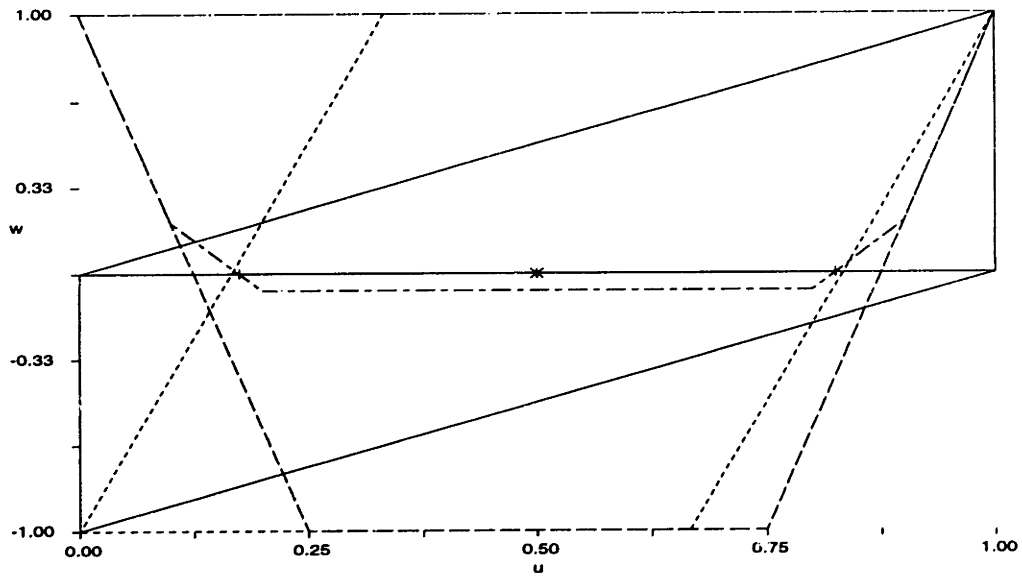


Figure 4-4: Four convex hulls projected onto uw-plane for the example of $y = x^4$ parametrized by u and $y = 0$ by v.

4.6.3 Overlapping

If the two curves overlap partially as illustrated in figure 4-5, the rate of convergence becomes much worse than the tangential intersection cases. Since the curvatures of the two curves are the same for overlapping portions, we can impose the tangent and curvature conditions so that the system of the equations becomes four equations with two unknowns as in the case for intersection of superbola and straight line.

Example 4.2 *We will illustrate this overlapping case using two cubic Bézier curves* $\overline{AB}$ *and* $\overline{CD}$ *whose control points are given by (0,0), (0.8,0.8), (1.6,0.32), (2.4,0.608) and (0.6,0.392), (1.4,0.68), (2.2,0.2), (3,1).*

We first run the solver with a fairly coarse level of accuracy, for example $\epsilon = 10^{-2}$ or 10^{-3}. If we observe a number of boxes overlap one another, as shown in figure 4-6, it is very likely that overlap exists. Figure 4-6 shows the bounding boxes of the two curves with $\epsilon = 10^{-2}$. We can observe that the curve $\overline{AB}$ overlaps with curve $\overline{CD}$ from $u = 0.25$ to $u = 1$ and the curve $\overline{CD}$ overlaps with curve $\overline{AB}$ from $v = 0$ to $v = 0.75$.

General Algorithm for Curve Intersections Since initially the algorithm does not know if the two curves overlap, tangentially intersect or transversally intersect, we should start solving the problem with the overconstrained system of four equations with two unknowns to check for the overlapping case first. If there exists an overlapping region, we can find the overlapping region by the method discussed in Section 4.6.3. We split one of the

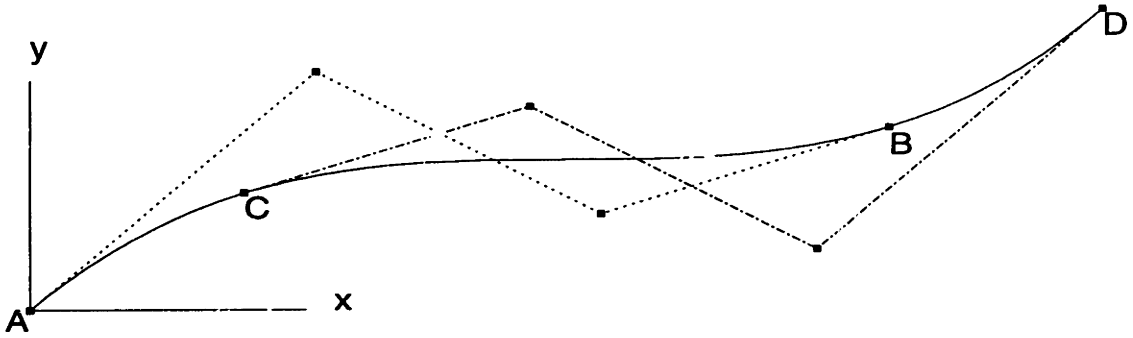


Figure 4-5: Two cubic Bézier curves $\overline{AB}$ and $\overline{CD}$ overlap each other along $\overline{CB}$.

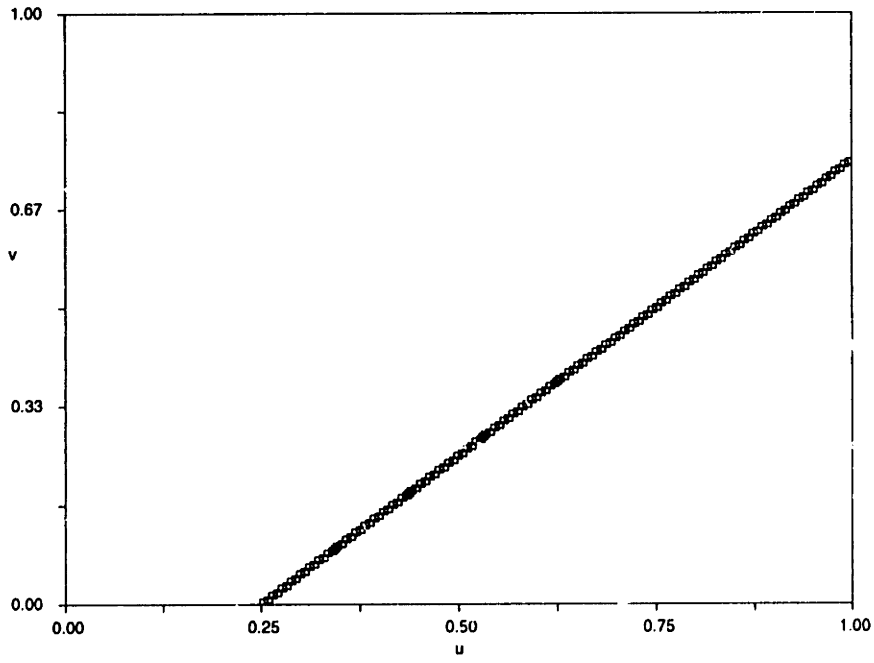


Figure 4-6: The bounding boxes of computing intersection of two overlapping curves

curves into non-overlapping and overlapping segments. Then we use the overconstrained system of three equations with two unknowns to check for the tangential intersection for the non-overlapping segments with the other curve which is not subdivided. If there exists a tangential intersection, we split the segment into two segments at the tangential intersection point. Finally, by using two equations with two unknowns, we check the transversal intersection for all segments. In trimming the curves around tangential contact points, we do not have to use the de Casteljau algorithm; instead we just specify the range of parameter values of those subdivided curves so that those ranges will be away from the tangential points.

4.7 3D Curve-to-Curve Intersection

The system for solving the intersection of two parametric curves in 3D space is an overconstrained system with three equations and two unknowns. As mentioned in Section 4.2, it is relatively rarely addressed in the literature. However, it can be solved by our IPP solver.

Given a parametric polynomial curve $\mathbf{C_1}(s) = (X_{C_1}(s), Y_{C_1}(s), Z_{C_1}(s))$, and another parametric polynomial curve $\mathbf{C_2}(t) = (X_{C_2}(t), Y_{C_2}(t)), Z_{C_2}(t))$, their intersections satisfy the following equation system:

$$X_{C_1}(s) = X_{C_2}(t) \tag{4.13}$$

$$Y_{C_1}(s) = Y_{C_2}(t) \tag{4.14}$$

$$Z_{C_1}(s) = Z_{C_2}(t) \tag{4.15}$$

4.7.1 Tangential and Overlapping Intersection of Curves

Two curves might intersect tangentially. Two examples of tangential curve intersection in the context of interval arithmetic are shown in Figures 4-7 and 4-8. In the example of Figure 4-7, part of the curve is inside of the other; in Figure 4-8, edges of the interval curves cross each other partially. For these pathological cases, the IPP solver might subdivide both curves many times in the area around the tangential contact point(s). Sometimes those fruitless subdivisions in turn increase the computation time by more than 100 times.

For those tangential intersection cases, their tangents are parallel to each other. We can impose the tangential condition to help solve those pathological intersection problems.

For two curves overlapping entirely or partially, we can impose the same condition to solve it. Seemingly the curvature condition can help in this case, but we do not suggest to use it. The reason is that the curvature condition will not speed up the process overall. It might be because the burden of solving one more equation (for curvature condition) offsets the effect of the root shrinking process, (see Section 8.1).

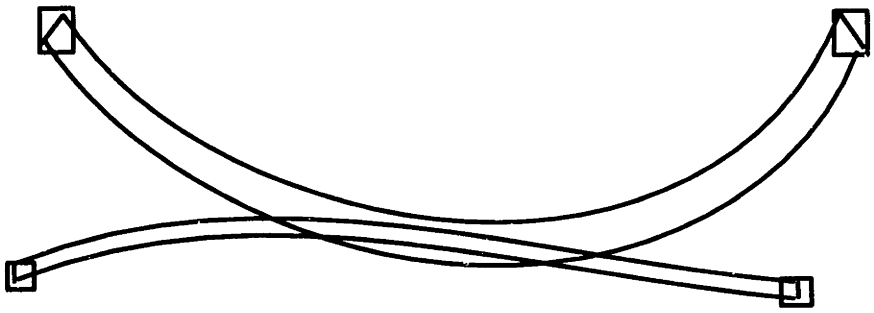


Figure 4-7: An example of pathological case of curve to curve intersection

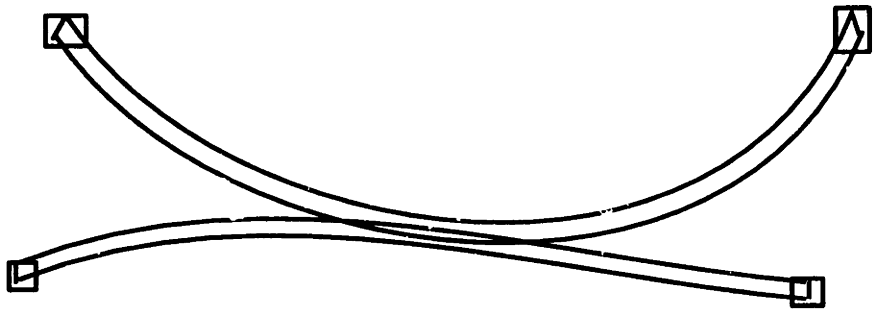


Figure 4-8: An example of pathological case of curve to curve intersection

4.8 Curve-to-Surface Intersection

The curve to surface intersections are used to find the intersections of border curves of one surface patch with the other surface, and vice versa. Those intersections serve as seed points to trace the intersection curves of two surfaces, as discussed in Section 4.9.

Given a parametric polynomial curve $\mathbf{C}(t) = (X_C(t), Y_C(t), Z_C(t))$, and a parametric polynomial surface patch $\mathbf{S}(u,v) = (X_S(u,v), Y_S(u,v), Z_S(u,v))$, their intersections have to satisfy the following system:

$$X_C(t) = X_S(u,v) \tag{4.16}$$

$$Y_C(t) = Y_S(u,v) \tag{4.17}$$

$$Z_C(t) = Z_S(u,v) \tag{4.18}$$

It is a balanced system with three equations and three unknowns. We can solve the system by the IPP solver.

4.8.1 Tangential Intersection of Curve and Surface

A curve might intersect a surface at point(s) where the tangent of the curve is orthogonal to the normal vector of the surface. For this pathological case, the IPP algorithm might subdivide the surface and curve many times in the area around the tangential contact

point(s). We can add the tangential condition to accelerate the root-finding process. At the tangential intersection point, the normal vector of the surface is perpendicular to the tangent of the curve. Let the normal vector of a surface be $(X_N(u,v), Y_N(u,v), Z_N(u,v))$, then

$$X_N(u,v)X'_C(t) + Y_N(u,v)Y'_C(t) + Z_N(u,v)Z'_C(t) = 0 \tag{4.19}$$

where the $\eta'(t)$ means the first derivative of η with respect to parameter t.

4.8.2 Curve on a Surface

A curve might lie on a surface. The IPP solver can find such intersections. However, we can add the tangential condition (equation 4.19) to the above system to speed up the process.

4.9 Surface-to-Surface Intersection

The intersection of two parametric polynomial surface patches $\mathbf{R}(u,v)$ and $\mathbf{S}(t,w)$ can be described as follows:

$$\mathbf{R}(u,v) = \mathbf{S}(t,w) \tag{4.20}$$

It is a underconstrained system with three equations and four unknowns. We can use the IPP solver to solve this system. However, it is too slow when small tolerances are used.

Another method is to use the marching method to find out intersection curves of two parametric surfaces [31], [43], [49]. In order to trace the intersection curve, we have to compute the starting points. An intersection curve branch can be traced if its pre-image starts from the parametric domain boundary in either parameter domain. In Section 4.8, we discussed how to find the border points by intersecting a curve and a surface. However, it is more difficult to find starting points for tracing intersection curve when they are closed in both parametric domains, or the two closed loops degenerate to an isolated point, i.e., the two surfaces intersects by a point. Section 4.9.1 discusses how to find such points. Section 4.9.2 discusses how to find the collinear normal points of two surfaces. Section 4.9.3 discusses the non-isolated collinear normal points and how to trace a collinear normal curve. Section 4.9.4 discusses tangential intersection curves of surface patches and presents a method to locate the starting point of tangential intersection curves. We refer it as *end point theorem*, because it specifies how a non-closed tangential intersection curve end for surface patches. Section 4.9.5 derives Corollary 4.2 from End Point Theorem to describe the condition for surface overlap.

4.9.1 Critical Points of Surface to Surface Intersection

In this section we will show how to compute critical (or significant) points of the intersection of two surfaces. The necessary and sufficient condition for critical points of two surfaces $\mathbf{R}(u,v)$ and $\mathbf{S}(t,w)$ is as follows:

$$\mathbf{R}(u,v) = \mathbf{S}(t,w) \tag{4.21}$$

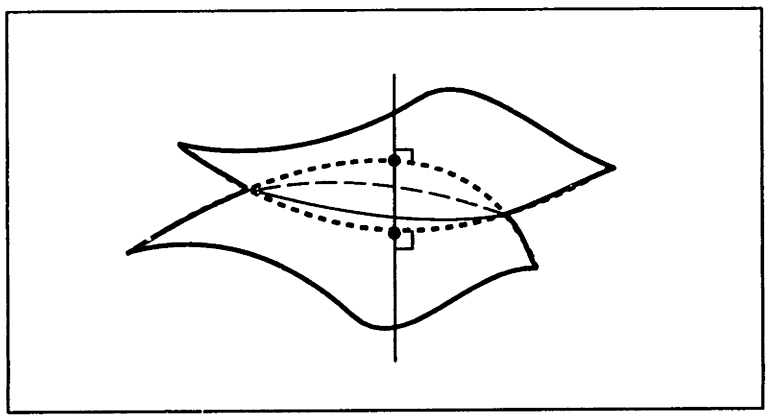


Figure 4-9: A pair of collinear normal points for two surfaces

$$(\mathbf{R}_u(u,v) \times \mathbf{R}_v(u,v)) \bullet \mathbf{S}_t(t,w) = 0 \tag{4.22}$$

$$(\mathbf{R}_u(u,v) \times \mathbf{R}_v(u,v)) \bullet \mathbf{S}_w(t,w) = 0 \tag{4.23}$$

The first (vector) equation 4.21 states that the roots are intersection points. The last two equations 4.22 and 4.23 state that at the intersection points, the two surfaces have the same normal direction. Equations 4.21 to 4.23 form an overconstrained system (five equations and four unknowns). It is traditionally broken into two steps [24], [25], [40] (*splitting method*). The first step is to find the collinear normal points, which is a superset of singular points; the second step is to pick up the actual singular points from the collinear normal points by checking the distance of surfaces at those collinear normal points. However, this system can ce solved by our IPP solver.

4.9.2 Collinear Normal Points of Surfaces

For two surfaces, if there is a line normal to both surfaces, the intersection points of surfaces with this line are called *collinear normal points* of the two surfaces, and this line is called a co-normal line, (see Figure 4-9).

The computation of collinear normal points of two surfaces is crucial for finding the critical points for surface-to-surface intersection problems. It is also important in detecting the occurrences of intersection loops of these two patches, for if two surfaces intersect along a loop, then they must contain collinear normal points. There are several papers on loop detection, [52], [68], [64], [62], [10], [33]. In this section, we present a robust method to compute collinear normal points of two interval Bézier surface patches.

Necessary and Sufficient Conditions for Collinear Normal Points

In the following Lemma, we point out that the condition for the stationary points of the squared distance function of two patches [82] cannot be used to determine collinear normal points efficiently.

Lemma 4.1 *If a pair of points (p, q) on two distinct parametric surfaces are collinear normal points, then they are the stationary points of the squared distance function of the two patches, but not the reverse (necessary but not sufficient condition for collinear normal points).*

Proof: Proof of the necessity ($\Rightarrow$): by the definition of collinear normal points, the distance vector $\vec{pq}$ are normal to both tangent planes of surfaces at p, q. Thus, $\vec{pq}$ is normal to any tangent of surfaces passing through p or q. Hence the inner products of distance vector $\vec{pq}$ and any tangent of the surfaces passing through p or q must be zero. Therefore, they are the stationary points of the squared distance function of the two patches. $\perp$

The above condition is not sufficient. This becomes obvious if we consider non-tangential intersection points of two patches, in which case, the distance function is minimized to zero. □

From Lemma 4.1, we know that employing the stationary point condition for the distance condition to find the collinear normal points of two distinct surfaces will encounter difficulties when two surfaces intersect along a curve or a loop (infinite roots). We have to appeal to other conditions to extract true collinear normal points effectively.

The following Lemma 4.2 presents the necessary and sufficient conditions for collinear normal points of two surfaces. The proof of the lemma is straightforward by the definition of collinear normal points.

Lemma 4.2 *Given two C^1 parametric surfaces $\mathbf{R}$ and $\mathbf{S}$, a pair of points $\mathbf{R}(u, v)$ and $\mathbf{S}(t, w)$ are collinear normal points of $\mathbf{R}$ and $\mathbf{S}$, if and only if the following equations hold:*

$$\mathbf{R}_u(u, v) \bullet (\mathbf{S}_t(t, w) \times \mathbf{S}_w(t, w)) = 0 \tag{4.24}$$

$$\mathbf{R}_v(u, v) \bullet (\mathbf{S}_t(t, w) \times \mathbf{S}_w(t, w)) = 0 \tag{4.25}$$

and

$$(\mathbf{R}(u, v) - \mathbf{S}(t, w)) \bullet \mathbf{R}_u(u, v) = 0 \tag{4.26}$$

$$(\mathbf{R}(u, v) - \mathbf{S}(t, w)) \bullet \mathbf{R}_v(u, v) = 0 \tag{4.27}$$

then, normals of surfaces at the pair of points are aligned.

To find collinear normal points, Sederberg et al. [62] used an interval type method to bound the feasible region for existence of collinear normal points. For those regions which possibly possess collinear normal points, they simply subdivided the two surfaces, and retest the new system until a feasible interval region is small enough or no feasible region is left. Consequently, this test needs expensive computation for tangencies [62].

Instead, we solve the equation system 4.24 to 4.27, by our IPP solver. We test our method by a rigorous example in which two surfaces are almost parallel to each other and have a tangential contact point. The result is shown in example 8.9 in Section 8.2.2.

In the case where there is a collinear normal curve between two surfaces, the equation system 4.24 to 4.27 has infinite roots. One such example is a parabola surface parallel to a plane; see Figure 4-10 for illustration of the collinear normal curve.

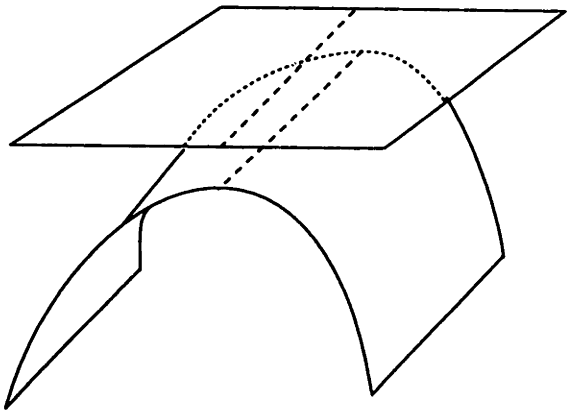


Figure 4-10: The occurrence of a pair of collinear normal curve between a parabola surface and a plane.

4.9.3 Marching on Intersection Curves from Significant Points

In this section, we will discuss how to find the intersection curves of two surfaces by tracing from those significant points. Given two parametric surfaces $\mathbf{R}(u,v) = (X_{\mathbf{R}}(u,v), Y_{\mathbf{R}}(u,v), Z_{\mathbf{R}}(u,v))$ and $\mathbf{S}(t,w) = (X_{\mathbf{S}}(t,w), Y_{\mathbf{S}}(t,w), Z_{\mathbf{S}}(t,w))$, their intersections satisfy the following equation system:

$$X_{\mathbf{R}}(u,v) = X_{\mathbf{S}}(t,w) \tag{4.28}$$

$$Y_{\mathbf{R}}(u,v) = Y_{\mathbf{S}}(t,w) \tag{4.29}$$

$$Z_{\mathbf{R}}(u,v) = Z_{\mathbf{S}}(t,w) \tag{4.30}$$

For the transversal intersection case, we can find the direction of the tangent of the intersection curve by taking the cross product of normals of two surfaces. The reason is because the intersection curve is on both surfaces, so the tangent of the intersection curve is perpendicular to both normals of the surfaces along the intersection curve (see Figure 4-11 for illustration). This direction is used in the tracing process. The starting points for the tracing is obtained by intersection points of one surface with boundaries of the other surface. The direction used for the tracing process is a unit vector τ tangent to both intersecting surfaces, defined by $\tau = \frac{(\mathbf{S}_t \times \mathbf{S}_w) \times (\mathbf{R}_u \times \mathbf{R}_v)}{|(\mathbf{S}_t \times \mathbf{S}_w) \times (\mathbf{R}_u \times \mathbf{R}_v)|}$. Therefore, equations for $\dot{u}$, $\dot{v}$, $\dot{t}$ and $\dot{w}$, (where $(\bullet)$ denotes derivatives with respect to s along the intersection curve parametrized with arc length), can be derived from this vector in a standard manner, as for example in Mortenson [43] (Section 7.4.4). These equations are

$$\dot{u} = \frac{\mathbf{R}_v \times \tau}{\mathbf{R}_u \times \mathbf{R}_v} \tag{4.31}$$

$$\dot{v} = \frac{\mathbf{R}_u \times \tau}{\mathbf{R}_u \times \mathbf{R}_v} \tag{4.32}$$

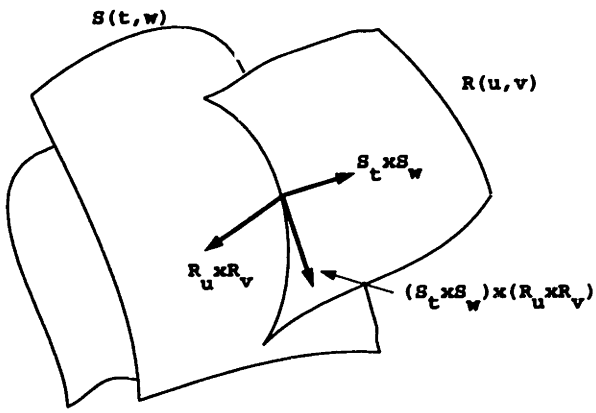


Figure 4-11: The direction of tangent of the intersection curve of two surfaces.

$$\dot{t} = \frac{\mathbf{S}_w \times \tau}{\mathbf{S}_t \times \mathbf{S}_w} \tag{4.33}$$

$$\dot{w} = \frac{\mathbf{S}_t \times \tau}{\mathbf{S}_t \times \mathbf{S}_w} \tag{4.34}$$

These equations form a system of four interval ordinary differential equations (IODE). This system is solved by an interval fourth-order Runge-Kutta method developed specially for this work, Press et al. [54]. The output of this Runge-Kutta IODE integrator is a set of boxes each of which encloses a small segment of the interval intersection curve.

Nevertheless, when two surfaces intersect tangentially along a curve, the normals of the two surfaces are parallel on that curve, and their cross product of normals is zero. So the direction of the tangent can not be decided by their normals. We discuss how to trace the tangential intersection curve of two surfaces in the next subsection.

4.9.4 Tracing of Tangential Intersection Curve of Surfaces

A tangential intersection curve of two surfaces is in fact also a collinear normal curve of the surfaces, (See Figure 4-12).

In order to trace the tangential intersection curve of surfaces, we have to first find the starting points. In this subsection, we prove a theorem stating that a non-closed tangential intersection curve of two Bézier patches must start from a border point and end at another border point as well. We refer this theorem as **end point theorem**. We also give two examples in which the tangential intersection curves contain loops. The first example is a torus and a plane on the top of the torus. Their tangential intersection curve is a circle. The second example is for two cylinders, which have the same radius. One is spaced along x-axis; the other along z-axis. Their tangential intersection curve is shown in Figure 4-13 (with darker line).

Before we prove the **end point theorem**, we need a commonly known fact described in the following lemma.

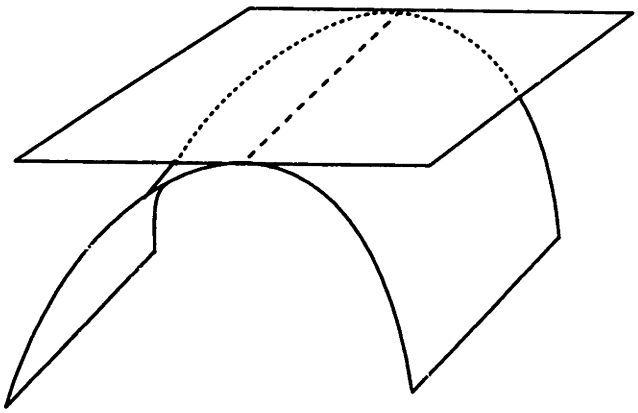


Figure 4-12: A parabola surface and a plane intersect tangentially along a line.

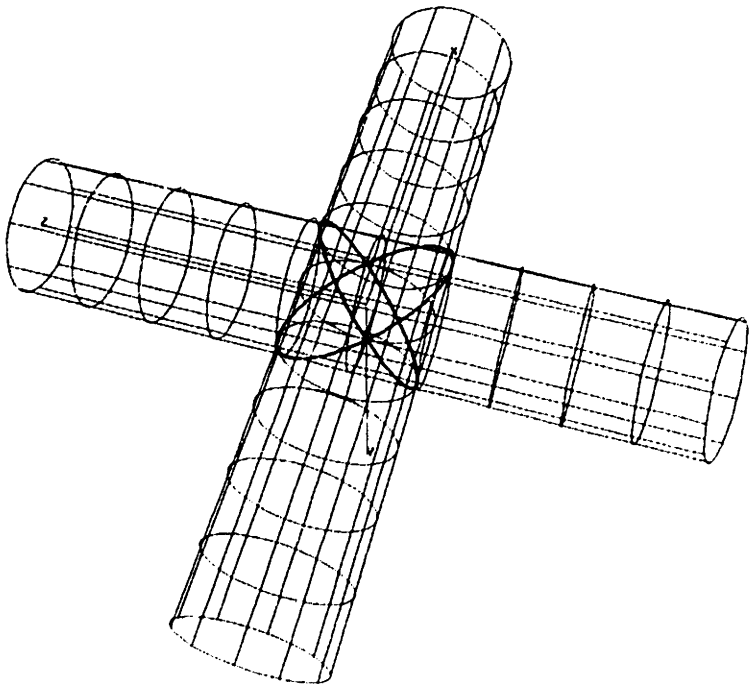


Figure 4-13: The tangential intersection curve of two cylinder contains loops.

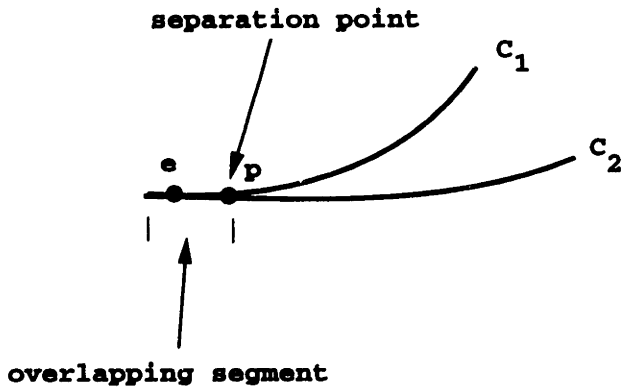


Figure 4-14: It is impossible for two curves to overlap along their common segment and to separate at some point.

Lemma 4.3 *If two C^∞ continuous curve segments $\mathbf{C}_1$ and $\mathbf{C}_2$ overlap along their common part of their segment, they must overlap everywhere. Otherwise, they end at boundary points.*

This lemma means it is impossible that two curves overlap along their common segment and separate from each other at one point, as illustrated in Figure 4-14.

Proof:

We prove this theorem contrapositively. Assume that there exist two C^∞ curves $\mathbf{C}_1$ and $\mathbf{C}_2$, overlap partially. This means that there is a point $\mathbf{p}$ (referred to as separation point) that ends the overlapping segment of $\mathbf{C}_1$ and $\mathbf{C}_2$, as illustrated in Figure 4-14.

Any curve can have two orientations. We parametrize two curves by the arc length appropriately, so they have the same orientations.

Since these two curves are C^∞ continuous, their overlapping segment should be C^∞ continuous, too. Therefore, from the Taylor expansion theorem, we know that there exists an $\epsilon > 0$ for a neighborhood $N_{\epsilon,\mathbf{p}}$, such that any point $q \in N_{\epsilon,\mathbf{p}}$, it lies on both of the two curves. Therefore, $\mathbf{p}$ cannot be the interior point. Hence, we have proven the theorem. □

We now prove the end point theorem.

Theorem 4.1 **(End Point Theorem)**: *If a tangential contact curve of two ideal Bézier patches does not contain a loop, then it must start from a border point and end at another border point as well.*

Proof:

We prove this theorem contrapositively. Assume that the theorem is wrong. That means, there exists a tangential contact curve $\overline{\mathbf{pq}}$ of two Bézier patches $\mathbf{R}$ and $\mathbf{S}$, which does not march from border points to border points of surface patches, as illustrated in Figure 4-15.

We can easily extend curve $\overline{\mathbf{pq}}$ to get a C^∞ continuous curve $\mathbf{C}_1$ on one patch because Bézier patches are C^∞. Likewise, we can get a C^∞ continuous curve on $\mathbf{C}_2$ containg $\overline{\mathbf{pq}}$ on

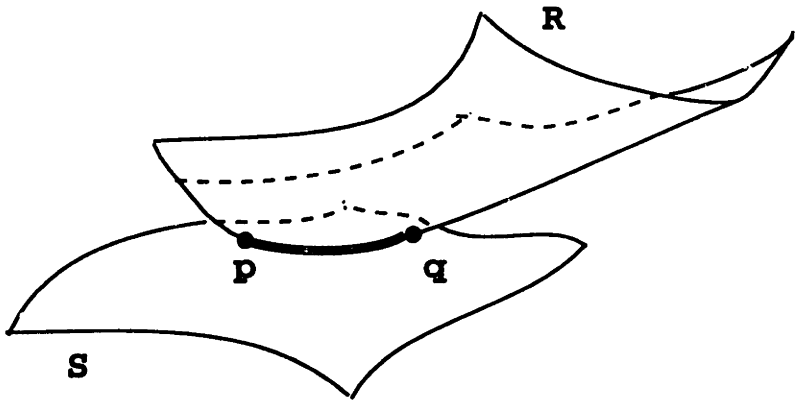


Figure 4-15: It is impossible for two surfaces to contact along a non-closed tangential curve (indicated by $\overline{\mathbf{pq}}$) in the middle of both surfaces.

the other patch. Then $\mathbf{C}_1$ and $\mathbf{C}_2$ overlap only partially (along $\overline{\mathbf{pq}}$) and separate at $\mathbf{p}$ and $\mathbf{q}$. This contradicts Lemma 4.3. □.

This end point theorem can help us locate a starting point to trace the tangential contact curve of Bézier patches by intersecting boundary curves of one patch with the other.

If the tangential contact curve of two Bézier patches is a loop, then inside the loop, there must be a collinear normal point which is not an intersection point of those two patches. We can subdivide these two patches at that collinear normal point to eliminate tangential contact loops.

The tangential contact curve of surfaces must satisfy the following conditions: (1) equation 4.20, (2) the system of equations 4.24 to 4.27, (3) the Hessian of the system in (2) is singular. Condition (1) makes sure that they are intersection points. Condition (2) guarantees that they are indeed collinear normal points. Condition (3) tells us that those points are very likely to be degenerate critical points of the system in condition (2), i.e. a collinear normal curve. The fact that if the collinear normal curve exists, then the direction of the tangent to the collinear normal curve can be derived from condition (3) [82] by singular value decomposition (SVD) [71].

The End Point Theorem can be extended to any two C^∞ continuous patches.

Corollary 4.1 *If a tangential contact curve of two C^∞ continuous patches does not contain a loop, then it must start from a border and end at another border as well.*

The proof for Corollary 4.1 is similar to the one in End Point Theorem.

In summary, to find the starting point of the tangential contact curve of two polynomial surface patches, we can use the boundary curve of one surface to intersect the other surface as starting points and end points. The Jacobian of the system of equations 4.24 to 4.27, is used for two purposes: (1) to detect the occurrence of tangential intersection curve [82]. (2) to find the direction of the tangent to the tangential contact curve to march.

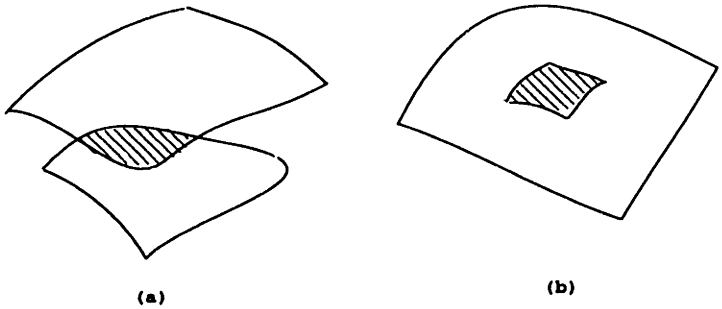


Figure 4-16: (a) Surfaces overlap across boundaries of both patches; (b) One surface fully overlaps within the other surface.

4.9.5 Overlap of Two Surfaces

The overlap of two surfaces is a degenerate singular case of one order higher than that of tangential intersection curve. There is a certain degree of misconception about surface overlap in the CAD community. For example, Sinha et al. [68] shows a figure indicating that two surfaces may overlap over an interior region. That is impossible for such surfaces as quadrics, superquadrics and Bézier patches, whereas this may be possible for general Coons patches and B-spline patches in certain special configurations.

In this subsection, we prove that if two ideal Bézier surface patches overlap, the overlap patch must be bounded by the boundaries of the two Bézier patches, as illustrated in Figure 4-16.

Corollary 4.2 *Given two ideal Bézier patches, if they overlap over a region, then the overlap region must be bounded by boundaries of Bézier patches. In other words, the overlap region cannot end in the middle of both Bézier patches.*

Proof:
Again we prove this corollary contrapositively. Suppose the corollary is not true. Then there exists an overalp region of two Bézier patches that ends in the middle of both Bézier patches.

Let $\mathbf{p}$ be an end point of the overlap region and $\mathbf{p}$ is not on the boundaries of either patch. On the overlap region, we can draw a non-self-intersection curve $\mathbf{C}$ that starting from $\mathbf{p}$ and end on the other point on the overlap region other than $\mathbf{p}$. $\mathbf{C}$ is a tangential intersection curve of the two patches, and contains no loop. Yet one end point $\mathbf{p}$ of $\mathbf{C}$ is not on the boundaries of either patch. This contradicts Theorem 4.1. □

For a point $\mathbf{p}_0$ on overlap of two surfaces, we anticipate that the Jacobian of the equation system equations 4.24 to 4.27 at $\mathbf{p}_0$, to have nullspace of rank 2. (It is possible on rare occasions that there is an overlap part of two surfaces containing p_0 but the Jacobian at $\mathbf{p}_0$ has null space of rank more than 2. This problem might occur if the first-order Taylor expansion of the equation system of equations 4.24 to 4.27 has zero 2nd order derivatives see [53] for details). Fortunately, these cases are extremely rare in practice.

If we find (by singular value decomposition (SVD) [71]) that for several intersection points, the Jacobians have nullspaces of rank 2, then we can practically assume that an overlap of two surfaces occurs.

From Corollary 4.2, we can use the method in Section 4.8 to find the boundaries lying on the other surface. Those boundaries should form a loop. Then we can use this loop as trimming loop to specify the overlapping ideal patch via a trimmed surface. This trimmed surface can come from either of the two involved surface patches.

An example of surface overlapping is shown in Example 8.12 in Section 8.2.3.

Chapter 5

Data Structure

5.1 Introduction

Data structures are devised to represent topological information of geometric objects. To develop a proper data structure for solid modelers is an important issue in solid modeling. B-rep is a natural way to represent solids. It represents a solid by its bounding entities, such as vertices, edges, and faces. The principal use of data structure in B-rep modelers is to store incidence information and provide adjacency between geometric entities. There is a great amount of literature about data structures for B-rep modelers. A review of them can be found in Bardis and Patrikalakis [2]. Since, the geometries in thesis are interval objects, the data structure has to be fit in this context.

The remaining of this chapter is organized as follows. Section 5.2 briefly reviews the cell-tuple data structure developed by Brisson [5] [6] first. Section 5.3 discusses the characteristics and categorizations for nodes of interval objects. Section 5.4 extends this data structure to represent non-manifold interval objects. This data structure can be easily adjusted to represent general $n-$dimensional interval objects.

5.2 Cell-Tuple Structure

In the cell-tuple structure, a d-manifold M is subdivided into k-cells $(k < d) \in C$ such that (1) each k-cell is homeomorphic to open k-disk, (2) each boundary of k-cell is homeomorphic to boundary of open k-disk, (3) each boundary of k-cell is a finite collection of α-cells, (α <k). Two valid examples of subdivided 2-manifold are shown in Figure 5-1 (d) and (e); Three invalid examples are shown in Figure 5-1 (a), (b) and (c).

For two cells c_1 and c_2, $c_1 \prec c_2$ denotes that c_1 is incident to c_2, and that c_1 is exact one dimension lower than c_2. Cell tuples of d-manifold (M, C) are:

$$T_M = \{(c_{\alpha_0}, ..., c_{\alpha_d}) | c_{\alpha_i} \in C, c_{\alpha_0} \prec ... \prec c_{\alpha_d}\}$$

Examples of cell tuples are shown in Figure 5-2. Figure 5-2(a) is a subdivided 2-manifold

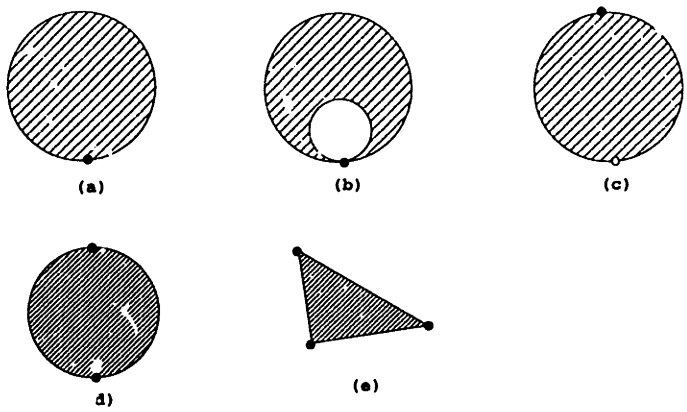


Figure 5-1: Examples of invalid subdivided 2-manifold: (a), (b) and (c), and examples of valid subdivided 2-manifold, (d) and (e).

Q. Figure 5-2(b) lists all cell tuples for Q, and displays 4 cell-tuples. If $t \in T_M$, then t_k denotes the k-cell of the k-th tuple of t.

If (M, C') is a subdivided d-manifold, $t \in T_M$ and $0 \leq k \leq d$, then there is a unique $t' \in T_M$ and a unique k, $(0 \leq k \leq d)$, such that

$t'_k \neq t_k$
$t'_i = t_i$ for all $i \neq k$

In other words, for any cell-tuple t in T_M, there is a unique cell-tuple t' in T_M, such that t and t' only differ by one cell. See Figure 5-2(b) for examples.

Based on this fact, operator $switch_i$ is defined as below:
$switch_i : T_M \rightarrow T_M$ such that
$switch_i(t) = t'$, where

$t'_i \neq t_i$ $t'_k = t_k$ for all $k \neq i$

The incidence graph of subdivided 2-manifold in Figure 5-2(a) is shown in Figure 5-2(c). Since the cell-tuple data structure and the incidence graph can be derived from one to the other, it is implemented by its corresponding incidence graph.

Properties of the Cell-Tuple Data Structure are listed below:

- Cells are mutually exclusive;
- Cells have different dimensions;
- Only manifolds are representable;
- Only closed objects are representable;
- Internal k-loops of n-object are not representable for k<n;

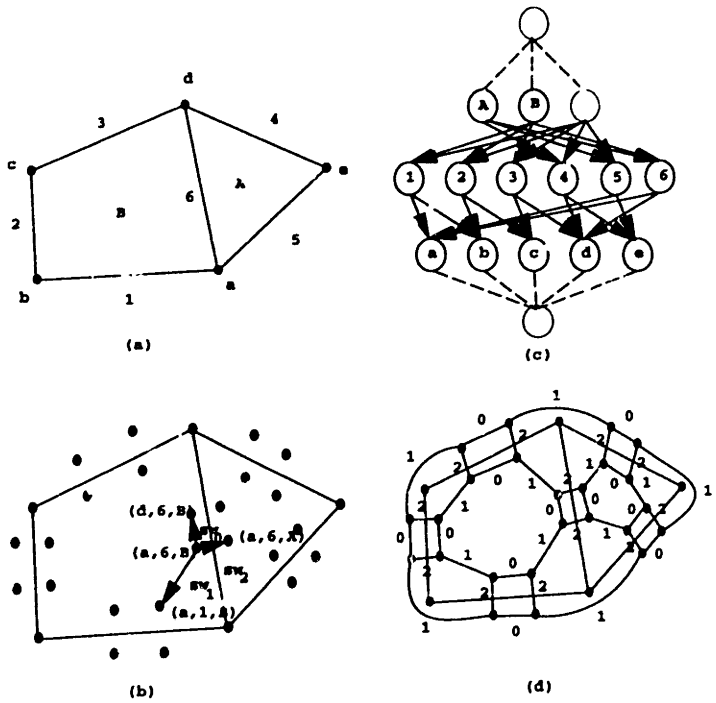


Figure 5-2: (a) is an example of subdivided 2D manifold Q; (b) lists all cell tuples for Q and demonstrates the operator $switch_i$; (c) is an incidence graph for Q; (d) shows an relationship between cell-tuples via $switch_i$ where $i = 0, 1, 2$.

- Artificial cuts are required to represent objects with genus more than 0;
- Ordering information can be derived from the data structure.

5.3 Characteristics and Categorizations of Nodes for Interval Solid Models

In this section, we first discuss the characteristics of nodes for interval objects. We then categorize nodes into different types.

5.3.1 Characteristics of Nodes for Interval Splines

In data structures, each geometrical entity, e.g. vertex, edge, face, or volume is represented by a node. Geometrical description and incidence information are stored in nodes. Conventionally, nodes are classified according to their dimensionalities (Weiler [78], Brisson [6]). The term *adjacency*, the topological counterpart of *incidence*, is stored in the data structure, since it is fundamental and essential to solid modelers. Other relations such as ordering information can be derived[6] from it. The objects in our modeler start as interval objects [1] and are expanded to larger interval objects via rounding processes.

For the sake of clarity, interrogation convenience and reducing incidence pointers of data structure, distinguishing dimensionalities of nodes is an advantage. The problem is how to determine dimensionality for each node. One way is from their representations. For instance, an interval point is represented by Cartesian coordinates specified by interval numbers; a free-form curve is represented by a series of interval control points; a free-form tensor-product surface is represented by a grid of interval control points.

There are two alternatives to decide node's dimensionality. One is to check elongations of geometrical entities. For instance, to decide whether to relegate a range to an interval point or an interval curve, if the ratio of elongations in one direction to the other direction of the range is bigger than a pre-defined constant real number (say 4.0), the range is assigned to the class of curves, otherwise it is relegated to an interval point. The other is to check end points of interval objects. For example, for a surface-to-surface intersection problem, how can we decide the solution set to be a interval point or an interval curve? (Assume that the solution set does not contain loops; loops can be detected and subdivided by collinear normal points, see Chapter 4). We check two end points of the interval intersection points whether they overlap or not. If they overlap, then the solution set should be consolidated to be an interval point. Otherwise, they should be relegated to an interval curve becaue two end points cannot envelop the solution set.

Issues in developing B-rep data structure for interval polynomial curves and surfaces include:

- All cells (vertices, edges, faces) are topologically equivalent;

[1] Recall that in the context of interval arithmetic, every object in 3D is *enveloped* by a 3D volume

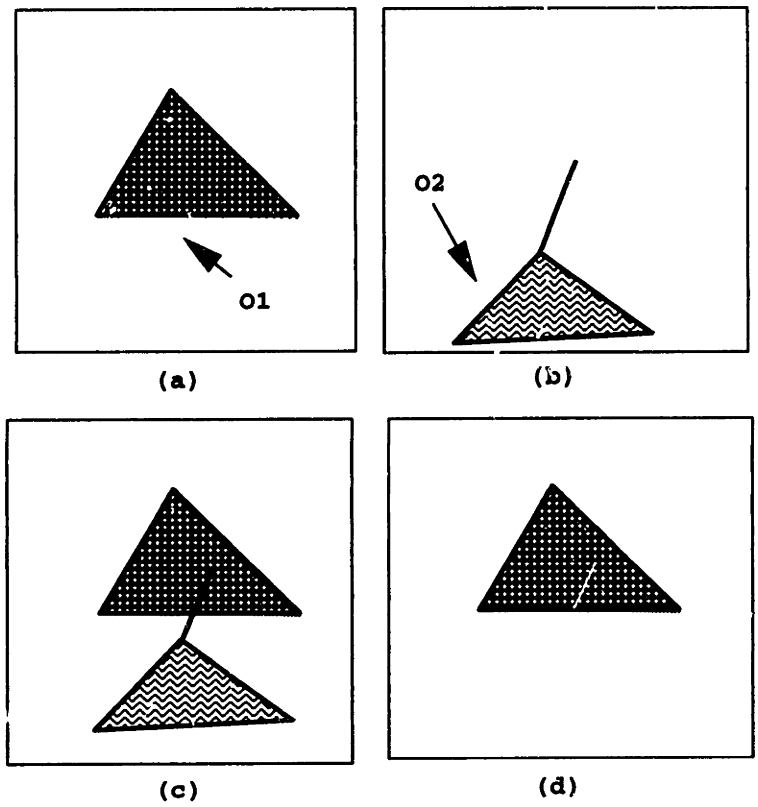


Figure 5-3: The intersection of a 2D bounded manifold with a 2D non-manifold models.

- Cells are not mutually exclusive;
- Data structure should permit representation of non-manifold objects;
- Data structure should permit representation of objects with loops;
- Data structure should permit representation of objects with genus.

5.3.2 Categories of Nodes in Data Structure

Non-manifold objects lead to more complicated topological problems than manifold objects [2]. For example, the difference of object O_2 in Figure 5-3(b) minus the object O_1, in Figure 5-3(a) is not homeomorphic to an open disk, or a line segment, or a point, as indicated in Figure 5-3(d). The reason is the following: we can add one point to the middle of the intersecting spike so that the difference will not be contractible to one point, while an open disk will always be contractible to one point when adding one point to it. Therefore, we need more kinds of nodes for non-manifold objects. They are classified as follows:

[2] In this thesis, the concept of manifold means both manifold and manifold with boundary [6]

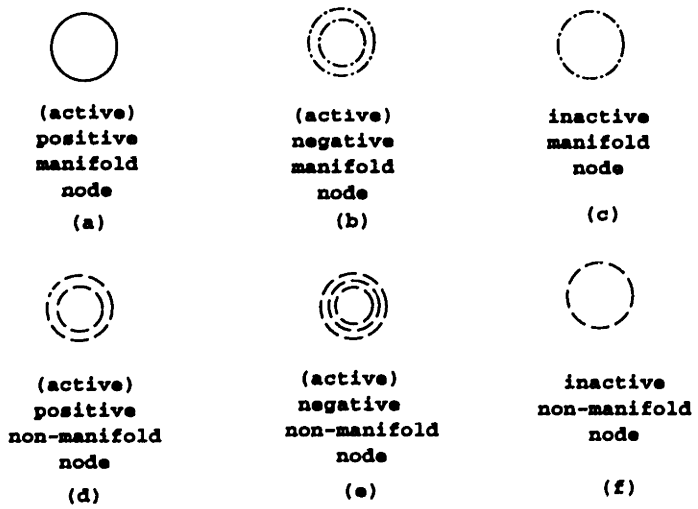


Figure 5-4: Six types of nodes in our data structure.

- active node
  - positive node: elements composing one model;
  - negative node: elements deleted from one model;
- inactive node: elements not belonging to a model;

Inactive nodes stand for entities which are nonexistent temporarily and can be recalled later on. Negative nodes stand for entities which are trimmed from models and they can usually be applied to trimmed surfaces and the resulting non-manifold Boolean difference operations (e.g. the example in Figure 5-3(d)).

Another way to classify nodes is their manifoldness. A node is manifold if its corresponding geometric entity is manifold, while a node is non-manifold if its geometrical entity is non-manifold. So overall, we have six types of nodes: (active) positive manifold nodes, (active) negative manifold nodes, inactive manifold nodes, (active) positive non-manifold nodes, (active) negative non-manifold nodes and inactive non-manifold nodes. When representing data structure graphically, we distinguish them by different symbols for corresponding type of nodes; see Figure 5-4 for illustration. The most commonly used type of node, i.e., the (active) positive manifold node is denoted by a solid circle, (see Figure 5-4(a)). An (active) negative manifold node is denoted by double concentric dashed dotted circles (Figure 5-4(b)), and it is the least used node. An inactive manifold node is denoted by a dashed dotted circle (Figure 5-4(c)). An (active) positive non-manifold node is denoted by double concentric dashed circles (Figure 5-4(d)). An (active) negative non-manifold node is denoted by triple concentric dashed circles (Figure 5-4(e)). An inactive non-manifold node is denoted by a dashed circle (Figure 5-4(f)).

5.4 Data Structure for Non-Manifold Interval Objects

We also develop in this thesis data structure for non-manifold interval objects. which is an extension of incidence graphs from manifold to non-manifold objects. Because the usual definition of incidence is modified, the limitation for cells to be homeomorphic to n-sphere (as in [6]) is relaxed, and the types of nodes are augmented. Also, in the data structure, manifold and non-manifold parts of a object are separated to assist the effective use of non-manifold point classification algorithm, (see Appendix B).

As has been pointed out, two objects are incident if they intersect. Cells in nodes are not necessarily homeomorphic to n-sphere. Cells could include 'holes', or 'handles' (i.e. objects with genus). Two more types of nodes are added to the incidence graph; they are negative nodes and inactive nodes. as discussed in Section 5.3.2. Nodes are also classified according to their manifoldness.

A list is assigned to each node for storing those nodes higher by one dimension incident to it in the counterclockwise order. Another list is assigned for storing those nodes lower by one dimension in the counterclockwise order.

We can always subdivide non-manifold objects into non-manifold parts and manifold parts. In 2D space, the non-manifold parts are usually one-dimensional curves or zero-dimension vertex. In Section 5.3.1, we mentioned that interval objects are all areas in 2D, thus we can treat interval curve segments or interval points as interval areas in 2D. In other words, an interval curve in 2D becomes a curved strip; an interval point becomes a rectangle. Similarly, in 3D space, we treat every interval objects as volumetric entity.

If the non-manifold part is an interval curve, we assign one *1D active* non-manifold node to it as usual and one *2D inactive* non-manifold node to it. Take the non-manifold model in Figure 5-5(a) for example: a triangle with a dangling edge attached to the top vertex. The data structure of the non-manifold model is shown in Figure 5-5(b). The numbers on the left hand side in Figure 5-5(b) stand for the dimensionalities of each rank; e.g. edges e, f, g and h are of one dimension [3]. In Figure 5-5(a) dangling edge h is the non-manifold part for this model, therefore one *1D active* non-manifold node denoted by single dotted circle is assigned to h and so is one *2D inactive* non-manifold node denoted by two dotted circles (see Figure 5-5(b)). The inactive 2D non-manifold node h in Figure 5-5(b) only serves the role to access non-manifold part, while the active 1D non-manifold node h actually contains the geometrical and topological information for dangling edge h in Figure 5-5(a).

Non-manifold models may be formed by some manifold models connected at some vertices. For example, in the the model in Figure 5-6(a), the non-manifold model is composed by two manifold models at the vertex v_3. For the 0D non-manifold interval point $v3$, we have *inactive* non-manifold nodes in dimensions 2 and 1, and an *active* non-manifold node in dimension 0. See Figure 5-6(b) for illustration, the 0D active non-manifold can be traced

[3] In Figure 5-5(b), there are three abstract nodes, indicated by two solid concentric circles for the graph of data structure: firstly, the abstract node in dimension 3 connotes the whole 2D space; secondly, the one in dimension 2 denotes the complement of the model; thirdly, the one in dimension -1 denotes the empty set. This is unnecessary, but useful for manipulating the incidence graph.

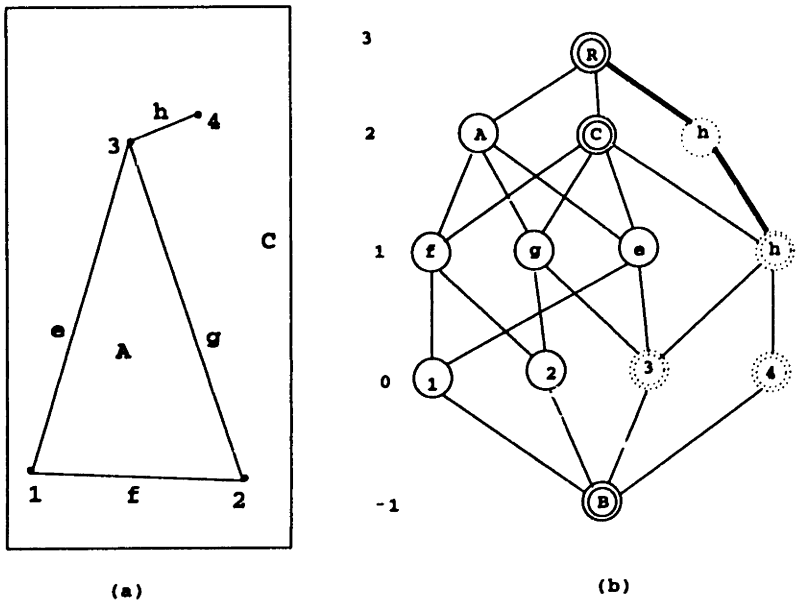


Figure 5-5: Data structures for a 2D non-manifold model

up to be incident to the inactive non-manifold node in dimension 2.

This data structure is easy to comprehend and implement. In addition, the extraction of manifold parts from the data structure is very easy for the use of point classification algorithm for a non-manifold object (see Appendix B).

The difficulty of implementing a data structure for non-manifold B-rep modeler is much greater than for a manifold one. However, thanks to the advent of Objected-Oriented programming languages such as $C++$, this difficulty can be greatly mitigated. The conceptual complexity of the problem can be further mitigated by implementing the incidence graphs from non-manifold objects.

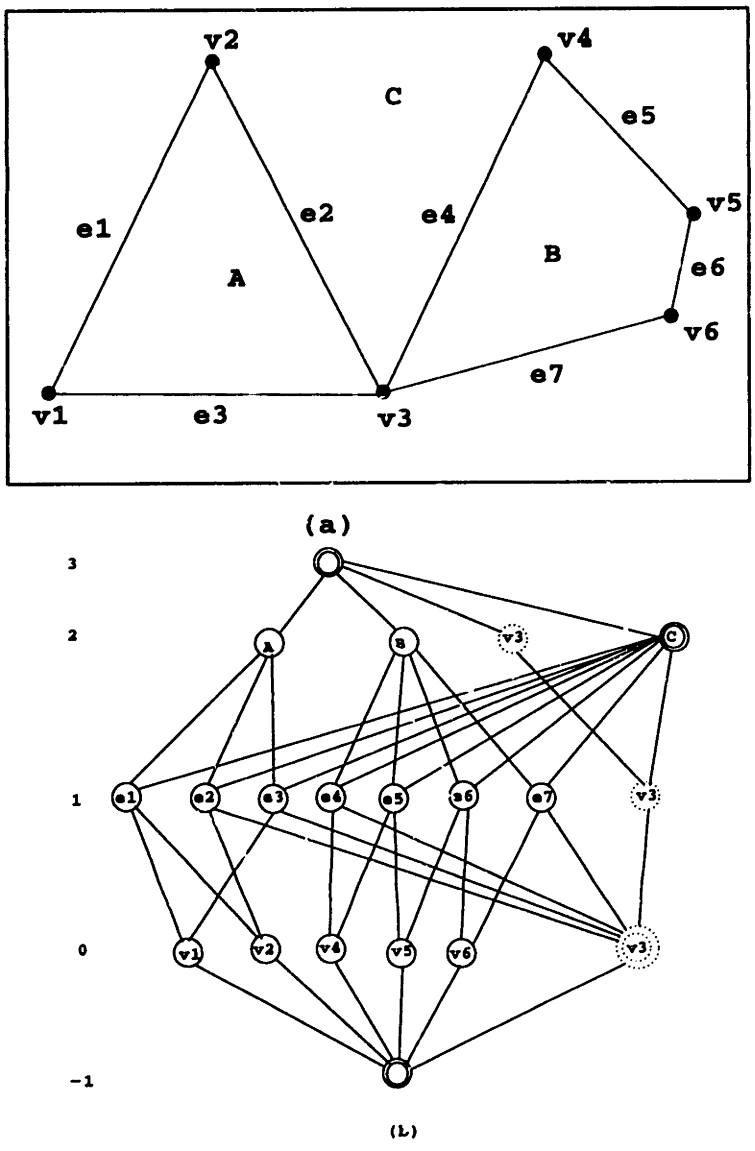


Figure 5-6: A non-manifold model and its data structure.

Chapter 6

Two Dimensional Boolean Operations

6.1 Introduction

Boolean operations, including Boolean operations for manifold and non-manifold objects, are the core operations of solid modeling systems. They have been discussed for example by Requicha [55], Rossignac and Requicha [57] and Gürsöz, Choi and Prinz [20]. Boolean operations are always implemented based on the data structure. Current data structures are only for idealized mathematical objects, see [78] [19] [56].

This chapter describes non-regularized Boolean operations for 2D manifold and non-manifold objects. The algorithms of this Chapter are extended to Boolean operations for more complicated 3D manifold and non-manifold objects in Chapter 7.

We have in Section 5.3 developed a non-manifold data structure for *interval objects* based on incidence graphs and separation of manifold from non-manifold parts. This separation is useful in solving the point classification problem which is essential to Boolean operations for non-manifold objects (see Appendix B). Based on this data structure, after briefly reviewing the Boolean operations in Section 6.2. Section 6.3 presents algorithms for 2D manifold Boolean operations. Section 6.4 extends the 2D manifold to 2D non-manifold operators.

6.2 Definition of Boolean Operations

Let A, B denote regular, compact sets in R^3, as shown in Figure 6-1(a). Boolean operations can be defined as:

$$b(A \cup B) = (bA \cap cB) \cup (bB \cap cA) \tag{6.1}$$

$$b(A \cap B) = (bA \cap iB) \cup (bB \cap iA) \tag{6.2}$$

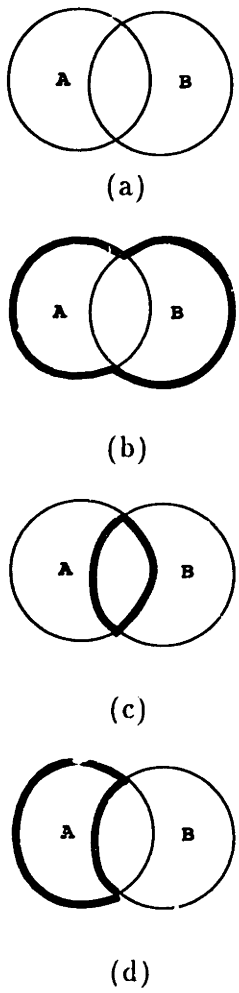


Figure 6-1: (a) The regular compact sets, A and B; (b) the union of A and B, and its boundary: $b(A \cup B) = (bA \cap cB) \cup (bB \cap cA)$; (c) the intersection of A and B, and its boundary: $b(A \cap B) = (bA \cap iB) \cup (bB \cap iA)$; (d) the difference of A minus B, and its boundary: $b(A - B) = (bA \cap cB) \cup (bB \cap iA)$.

$$b(A - B) = (bA \cap \quad) \cup (bB \cap iA) \tag{6.3}$$

where bX is the boundary of the set X, iX is its interior and cX is its complement, (see Figure 6-1 (b), (c) and (d)).

The above defined Boolean operations are set-theoretic operations. They differ from the conventional *regularized Boolean operations* in that lower-dimensional structures are allowed in the objects i.e., they can be applied to both manifold and non-manifold objects. As has been pointed out in [22], although eliminating the lower-dimensional structures by regularized Boolean operations is desirable for defining solids/regions, in some applications, it is also desirable to retain them, possibly even in the interior objects. For example, when considering solids as *domains* in finite element analysis, interior lower-dimensional structures might represent certain constraints on how to discretize the domain, or might define the domain discretization outright.

It should be pointed out here that the above definition of Boolean operations are suitable for objects with any dimensionality, including 2D and 3D, which cases will be discussed in this and the later chapters, respectively.

6.3 Manifold Boolean Operations

In this section, we assume that the input objects and the results of the Boolean operations are 2D, manifold, bounded, and the intersections of the boundaries are finitely many "point boxes". The case in which the input models are manifold and the output models are non-manifold will be handled in Section 6.4.

6.3.1 General Algorithm for Manifold Boolean Operations

The following three steps are used to perform 2D manifold Boolean operations (union, intersection and difference) of two given 2D manifold objects A and B.

1. Intersect A and B and identify the inside or outside status of individual entities (vertices and edges);
2. Refine $A \cup B$;
3. Execute operator-specific operations.

The first step involves intersecting the boundaries of models A and B and identifying the inside or outside status of individual entities (vertices and edges), by point classification algorithms (Appendix B). The second step is to refine the resulting intersection of models, i.e. subdivide models into several sub-models according to their intersection points.

The following steps are used to refine $A \cup B$ in Step 2:

- Start from the intersection points;
- Find all the closed paths such that each of them

- do not connect the subdivided segments,
- only choose the original non-intersecting vertices once,
- do not pick all non-intersecting vertices;

- Remove duplicate cycles.

The result of the refining process is a so-called partition defined as follows:

Definition 6.1 *A* **partition** *of an object M is a collection C of subsets of M such that the union of the elements of C is M and each pair of elements of C have disjoint interiors.*

Since the 2D region is bounded by interval curves and interval points, we have to redefine the interior of a 2D region. Let us redefine the interior point of the region A, first.

Definition 6.2 *Given a region A bounded by interval curves $\{C_i\}_{i\in I}$ and interval points $\{B_j\}_{j\in J}$, a point p of the region A is called an interior point of the region A, if there exists a neighborhood N_p of p such that $N_p \cap C_i = \emptyset$ (for $i \in I$) and $N_p \cap B_j = \emptyset$ (for $j \in J$).*

Now, we define the interior of a 2D region by its interior point.

Definition 6.3 *Given a region A bounded by interval curves $\{C_i\}_{i\in I}$ and interval points $\{B_j\}_{j\in J}$, the interior of A is the set of all the interior points of A.*

With the help of Definition 6.3, Definition 6.1 will also hold for 2D regions bounded by interval curves and interval points. Since two incident regions share the same boundaries which are composed of interval curves and interval points, and their interior set will not have any point in common. Therefore, regions have disjoint interiors.

The third step performs the following operator-specific operations:

- $A \cup B$: output all the closed regions.
- $A \cap B$, output those regions which have edges from B but inside A and edges from A but inside B.
- $A - B$, output those regions whose boundaries are either subdivided from A or subdivided from B but inside A.
- $B - A$, the same as the above procedure, but interchange A and B.

The following terminology is defined for convenience:

Definition 6.4 *Given two planar models A and B, a vertex v is denoted as $\mathbf{v_{AinB}}$, if it belongs to A and is inside B, and it is called a vertex of A inside B. A vertex v is denoted as $\mathbf{v_{AoutB}}$, if it belongs to A and is outside B, and is is called a vertex of A outside B. If two curves are subdivided from one* parent *curve, we call those nodes for these two curves* **brother edge nodes.**

The following procedure is to find cycle paths in a graph of the partitions of M_1 and M_2 with identified intersection points. For the sake of clarity, we take the models in Figure 6-2 for example to explain the procedures.

Algorithm 6.1 *1. Start from intersection points;*

- **Example:** in Figure 6-2(b) $n1$ and $n2$ are both intersection points, so we can start from $n1$ or $n2$ to search loops. Let us take $n1$.

2. *Find the incident objects in an alternative sequence of 1D nodes (edges) and 0D nodes (vertices);*

   - **Example:** Starting from $n1$ (see Figure 6-2(b)), the next node to be inserted into a loop can be one of four 1D nodes $w1$, $w2$, $w5$ or $w6$. Let us take $w6$. Then the next incident node to be chosen has only one choice $v6$. We can continue the similar procedure to insert those nodes into the searching loop $\{n1, w6, v6, e6, v5, w8, n2\}$ until we reach $n2$. We have three nodes $w3$, $w4$ and $w7$ to choose from as the next alternative node incident to $n2$. The next step shows how to choose the next node from them.

3. *Choose nodes other than* **brother edge node** *(see Definition 6.4) , when an edge node encounters its* **brother edge node** *through their common vertex;*

   - **Example:** Taking the previous searching loop $\{n1, w6, v6, e6, v5, w8, n2\}$ for example. From the three candidate nodes $w3$, $w4$ and $w7$ (see Figure 6-2)(b), we will choose either $w3$ or $w4$, but not $w7$ as the next alternative node incident to $n2$, since $w7$ is a *brother* edge of $w8$. Let us take $w4$. So the searching loop becomes $\{n1, w6, v6, e6, v5, w8, n2, w4\}$.

4. *Do not pick up any $v_{M_i outM_j}$ node, once a searching loop picks up one $v_{M_i inM_j}$ node, and vice versa, where $(i \neq j, i, j = 1, 2)$;*

5. *Complete one searching loop, when the starting point is re-visited;*

   - **Example:** If we continue the previous searching loop, searching loop becomes $\{n1, w6, v6, e6, v5, w8, n2, w4, v3, w2, n1\}$. Since $n1$ is the starting point, we complete this searching loop.

6. *Go to step 1, until all options for candidate loops are exhausted.*

   - **Example**: *Pick $n2$ and start the step 1 again.*

In the above Algorithm 6.1, Step 4 can help make the resulting loops to be the difference or the intersection or the union of the two involved models.

Given a partition of objects M_1 and M_2 generated by intersecting their boundaries and for each node (see Figure 6-2), we use point classification (as in the Appendix B)

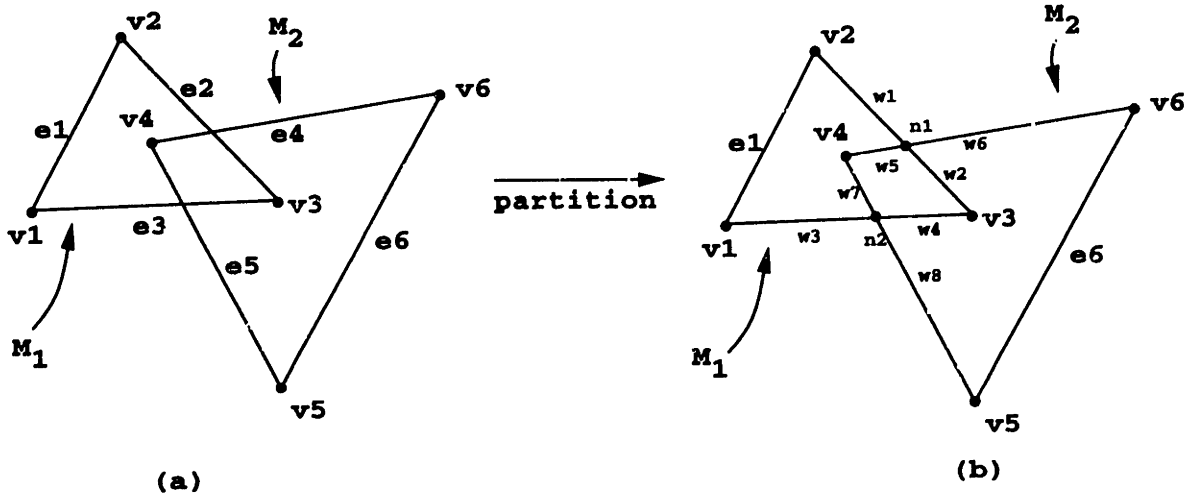


Figure 6-2: Finding the three independent cycles in a graph; step 1, in partition process, two intersection 0D nodes, n1 and n2 are added; edge e2 is subdivided into two edges w1 and w2, so are edges e3, e4, e5; step 2, complete loops include *loop1*: {n2, w3, v1, e1, v2, w1, n1, w5, v4, w7, n2 }; *loop2*: {n2, w4, v3, w2, n1, w5, v4, w7, n2}; *loop3*: {n1, w2, v3, w4, n2, w8, v5, e6, v6, w6, n1}.

to determine their inside or outside status of the other model for each active node. For example, in Figure 6-2(b), vertices $v4, v3$ and edges $w2, w4, w5, w7$ are of inside status, while vertices $v1$, $v2$, $v5$, $v6$ and edges $w3$, $e1$, $w1$, $w6$, $e6$, $w8$ are of outside status.

Let us denote V_i^j as the vertex set of $v_{M_i in M_j}$. Take Figure 6-2(b) for example, $V_1^2 = \{v3\}$, $V_2^1 = \{v4\}$.

If an edge has both vertices in V_1^2, or has one vertex in V_1^2 and the other as an intersection vertex, then that edge belongs to the boundary of the intersection of M_1 and M_2; for instance since $w5$ in Figure 6-2(b) has one end vertex $v4 \in V_2^1$ and the other is the intersection vertex $n1$, $w5$ is one boundary of the intersection of M_1 and M_2, so are $w2$, $w4$ and $w7$.

Similarly, we can denote the vertex of one model, which is outside of the other model.

Let us denote U_i^j as the vertex set of $v_{M_i out M_j}$; for instance, $U_1^2 = \{v1, v2\}$, $U_2^1 = \{v5, v6\}$.

If an edge has both vertices in U_i^j, it belongs to the boundary of the difference of M_i from M_j. For example, since $e6$ has both vertices in U_2^1, $e6$ belongs to the difference of M_2 from M_1. If an edge has one vertex in U_i^j and the other vertex is an intersection vertex, it also belongs to the boundary of the difference of M_i from M_j. For example, since $w6$ has one vertex $v6$ in U_2^1 and the other vertex is the intersection vertex $n1$, $w6$ belongs to the difference of M_2 from M_1.

After the loops are found, we can use them to complete the Boolean operations. This will be discussed in the following Sections.

6.3.2 Intersection Operation

Given two planar models M_1 and M_2 and their partition loops, the following Corollary is used to identify which loops are the intersection loops.

Corollary 6.1 *A loop derived from Algorithm 6.1 is an intersection loop of M_1 and M_2, if and only if*

1. *It has one vertex $v_{M_i inM_j}$, where $i \neq j$, or*
2. *All of its vertices are intersection vertices and the interior of the loop has a vertex common to M_i, $i \in \{1,2\}$.*

Proof: The proof of the sufficiency ($\Leftarrow$) is obvious. It follows from Step 4 of Algorithm 6.1 that no loop of the partition loops generated by Algorithm 6.1 will consist of both kinds of vertices $v_{M_i inM_j}$ and $v_{M_i outM_j}$, $i \neq j, i, j = 1, 2$. For a loop l with a vertex $v_{M_i inM_j}$, $i \neq j$, there are two cases in which this can occur: either (case 1) no edges of l intersect M_j, as indicated in Figure 6-3(a), or (case 2) some edges of loop l intersect M_j, as indicated in Figure 6-3(b). For case 1, the loop l is the boundary of M_i, which is inside M_j, so the loop l is the intersection of M_1 and M_2. For case 2, it only consists of the intersection vertex and $v_{M_i inM_j}$. Any edge between those vertices is in both M_1 and M_2.

A loop l can have all vertices as intersection vertices, without $v_{M_i inM_j}$, as in Figure 6-3(c). From the assumptions, loop l has a point p of its interior common to both M_1 and M_2, so this loop must be one of the boundaries of the intersection of M_1 and M_2. Therefore, loop l is an intersection loop. This completes the proof for the sufficiency.

We prove its necessity ($\Rightarrow$) by proving its contrapositive. Suppose a loop l' contains no $v_{M_i inM_j}$, where $i \neq j$, then l' is composed of $v_{M_i outM_j}$ and intersection points, or intersection points only. If l' contains one $v_{M_i outM_j}$, then it does not belong to M_j, therefore l' can not be the intersection loop. If l' comprises only intersection, it has no vertex of interior common to both M_1 and M_2 by hypothesis; therefore, l' can not be an intersection loop either. This completes the proof for the necessity. □.

6.3.3 Difference Operation

Given two planar models M_1 and M_2 and their partition loops, the following Corollary is used to identify which loops are the difference loops.

Corollary 6.2 *A loop derived from Algorithm 6.1 is a difference loop of M_1 from M_2, if and only if*

1. *it has one vertex $v_{M_i outM_j}$, where $i \neq j$, or*
2. *all of its vertices are intersection vertices and the interior of the loop has no point common to M_i, $i \in \{1,2\}$.*

The proof of Corollary 6.2 is similar to Corollary 6.1. Figure 6-4(a) is used for the proof of Corollary 6.2 as the counterpart for Figure 6-3(a) in the proof of Corollary 6.1; so are Figure 6-4(b) for Figure 6-3(b), Figure 6-4(c) for Figure 6-3(c).

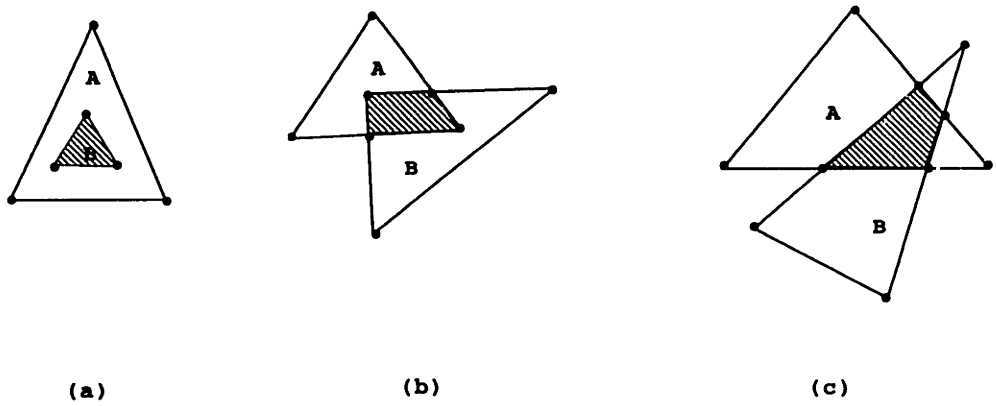


Figure 6-3: Examples of intersection loops for partitions of two manifold objects.

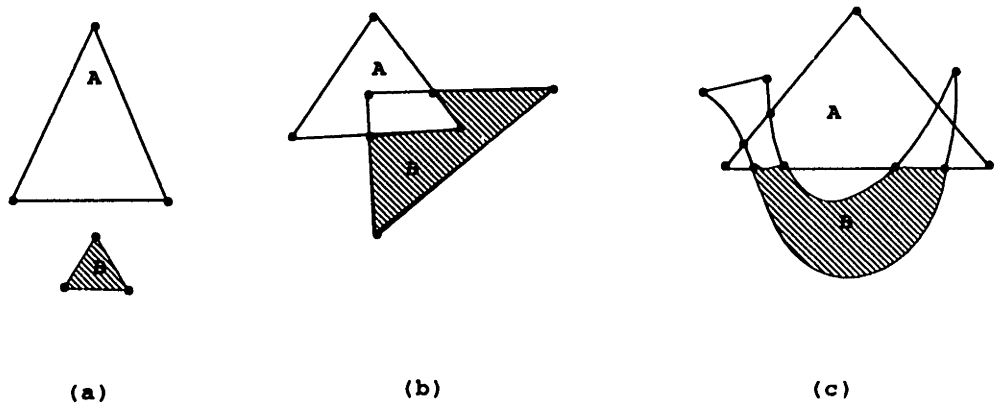


Figure 6-4: Examples of difference loops for partitions of two manifold objects.

6.3.4 Union Operation

Similar to Corollary 6.1 and 6.2, we can have a Corollary for the union of M_1 and M_2. In fact, *the union of M_1 and M_2 is the set of all the loops so that each loop contains only vertices which either are in U_1^2 and U_2^1, or are of intersection vertices.* Take the the loop { $n2, w3, v1, e1, v2, w1, n1, w6, v6, e6, v5, w8, n2$ } in Figure 6-2 for example. Any edge with both vertices being intersection vertices is either inside M_i, $i \in 1,2$ (see Figure 6-3(c)), or outside M_i, $i \in 1,2$ (see Figure 6-4(c)).

6.4 Non-Manifold Boolean Operations

Non-manifold Boolean operations require slightly different approaches than manifold Boolean operations. since subdivided regions in the refinement in Algorithm 6.1 may not be manifold regions. They might be a line segment or a point, thus we can not employ loop-searching to carry out the refinement process.

Algorithm 6.1 for manifold Boolean operations requires small changes to fit the non-manifold Boolean operations. The first step of Algorithm 6.1 for intersecting processes and identifying inside or outside status for each 0D and 1D cell remains unchanged The second step for refining processes is also needed. However, as has been pointed out, it necessitates modifications for the collections of nodes with the same in or out status. Since subdivided parts might comprise non-manifold objects. We discuss them for each Boolean operation separately.

Intersection: Since the non-manifold model results from intersection of two manifold models M_1 and M_2, it might consist of manifold regions, curve segments, points or combinations of manifold regions with curve segments. We need to find them out. The approach to find them is as follows:

- Find closed intersection loops from the intersection points (manifold parts);
- Find intersecting line segments or tangentially intersecting points (non-manifold parts);
- Integrate them into one model, if any non-manifold part is connected to manifold parts.

Difference: A distinct approach is used to extract differences of two models. We unify two intersecting models as one model, then refine the unified model if new regions arise. Afterwards, we remove those elements which are either the intersections of two models or parts of the subtracting model.

Union: To find the union of two models, an approach similar to the one for difference operation is employed. After the boundaries are intersected with each other, the unified model is refined according to the intersections of these two models. Those intersection

nodes which belong to one model and are inside the other model are then removed, and the remaining elements make up the union of the original two models.

Chapter 7

Three Dimensional Boolean Operations

7.1 Introduction

Until now, we have developed (1) a robust geometrical representation, (2) a robust non-linear polynomial equations solver, (3) a robust and unified algorithm for well- and ill-conditioned intersections problems, (4) an nD non-manifold data structure for interval objects, and (5) algorithms for 2D Boolean operations. By employing all of these, this chapter will develop algorithms for 3D Boolean operations for manifold objects, and extend them to non-manifold objects resulting from ill-conditioned intersections.

The remaining of the chapter is organized as follows. Section 7.2 presents an overview of the procedures for 3D Boolean operations. Section 7.3 discusses the surface intersection algorithm for Boolean operations. It also summarizes the difference of our methods from other methods. Sections 7.4 to 7.6 develop algorithms to refine various nodes, such as 0D, 1D, 2D and 3D nodes. Section 7.7 summarizes the algorithms for Boolean operations. Section 7.8 discusses the method for rendering trimmed surfaces.

7.2 Procedures for 3D Boolean Operations

Given two 3D manifold models A and B, the general procedures for 3D manifold Boolean operations can be described as follows:

1. Intersect the boundaries of A and B;
2. Create new nodes for each intersection curve and refine the boundary curves of bounding surfaces of models A and B;
3. Refine each bounding surface of A and B;
4. Refine $A \bigcup B$ into different shells;

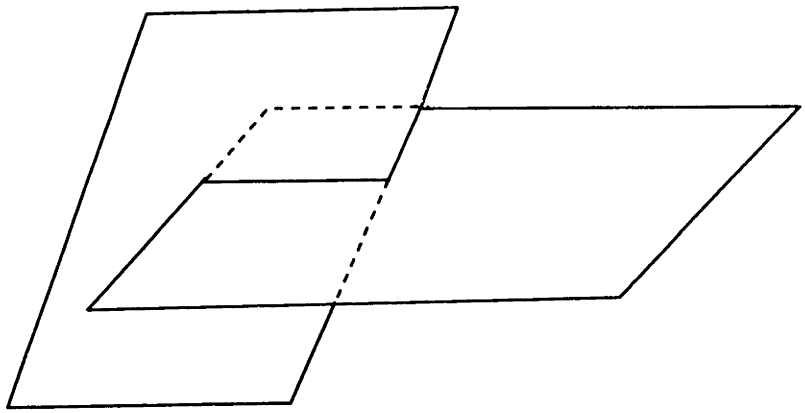


Figure 7-1: The common intersection curve does not divide the surfaces into separate regions.

5. Perform specified operations for union, intersection and difference.

Steps 1 and 2 create new 0D and 1D nodes for intersection curves. In the event of occurrences of boundary intersections, step 2 assigns new nodes to those subdivided bounding surfaces and curves.

In the 2D modeler discussed in Chapter 6, when two curves intersect transversally, we simply subdivide the involved curves at their intersection points. However, in the 3D modeler, we cannot simply subdivide involved surfaces along their intersection curves, because their intersection curves might not divide those surfaces into two separate regions; (see Figure 7-1). Therefore, we cannot subdivide surfaces into separate subpatches until all surface-to-surface intersections are found.

The refinement in Step 2 involves subdivision of boundaries of the two models, based on the intersections of each bounding surfaces of A and B. This refinement is essential to the Boolean operations. It is discussed in detail in Section 7.4.

Step 3 creates new 2D nodes for subdivided surfaces. In this step, we refine the bounding surfaces of models A and B. Because in Step 1, we have already embedded 1D nodes for intersection curves and 0D nodes for their end points into their data structures, we can now identify their intersection nodes from their data structures. A bounding surface can be separated into several regions by intersection curves with other surfaces and its original boundary curves; see Figure 7-8 and Figure 7-10 for examples. In Figure 7-10, surface A is separated into two regions by the intersection curves and its boundary curves. In Figure 7-8, surface A is separated into two regions. One region is bounded only by intersection curves. New 2D nodes are created for those newly divided regions and are incorporated into the data structure.

Step 4 creates new 3D nodes for subdivided shells. In this Step, we refine the $A \cup B$ into several shells. The bounding surfaces of each shell are identified. The shell can be a shell of difference, intersection, or union of A and B. Take Figure 7-2 for example. We can have four resulting shells: (1) the difference of hexahedron A and hexahedron B, (2) the

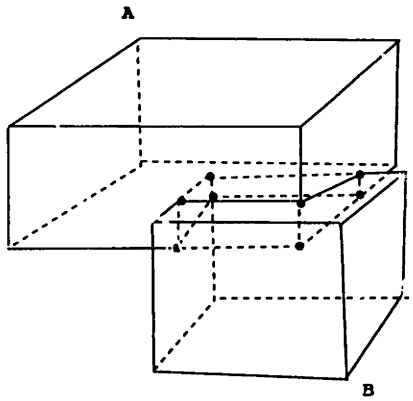


Figure 7-2: Two 3D regular compact objects, A and B.

intersection of A and B, (3) the difference of B and A, and (4) the union of A and B.

Finally, in Step 5, we finish the Boolean operation according to the shells identified from Step 4.

7.3 Intersection of the Boundaries

Geometric intersection is the core issue in Boolean operations. We have described the geometric intersection in detail Chapter 4. Among all the boundary-boundary intersections, the most common intersection is surface-to-surface intersection [31]. We discuss our geometric intersection method for 3D regular compact objects as in Figure 7-2 in this section.

In many solid modelers, the *bottom-up* approach is used to intersect geometries [9] for bounding surfaces of models. In this approach the intersections of lower-dimension geometries are performed before those of higher-dimension geometries. For example, curve-to-surface intersections are executed before surface-to-surface intersections. The main purpose of the *bottom-up* method [9] is to reduce inconsistencies arising from floating point error. On the other hand, it complicates and duplicates the intersection procedure, especially in handling of parameters of parametric entities. Take curve-to-curve intersection for example. Since each curve is incident to two surfaces, each curve has two parametrizations. These two parametrizations might be different. In this case, the resulting parametric values for the intersection points might be different. For instance, for curves A and B, A has two parametrizations A_{p1} and A_{p2}; B has two parametrizations B_{p1} and B_{p2}. The resulting intersection of A_{p1} with B_{p1} is different from that of A_{p2} with B_{p1}. Therefore, to fully investigate the intersection of two curves A and B, we have to intersect four times; A_{p1} with B_{p1}, A_{p1} with B_{p2}, A_{p2} with B_{p1}, and A_{p2} with B_{p2}. The bookkeeping for those four results is significant.

In our work, the inconsistency problems are solved by the combination of interval polynomial objects (Chapter 2) and IPP solver (Chapter 3), which make our geometric computations robust. Hence, for our interval solid modeler, we do not use the bottom-up

approach.

We attach one *parameter* attribute which specifies the parameter information of a node to its associated 2D nodes. The addition of parameter attributes in each 0D, 1D nodes will greatly simplify the topological tracing process. It will also expedite the collection of parameter information of trimming loops to refine surfaces (Section 7.5) for each 0D and 1D node. Take a vertex incident to three surfaces for example. This vertex can be geometrically located from its three incident surfaces via associated parameters. Therefore, in the case, the parameter attribute for this 0D node (vertex) is a list which stores the associated parameters for its three incident surfaces (see Figure 7-3).

There are two kinds of boundary curves for bounding surfaces: one is the original boundary curves of surfaces; the other is the intersection curves of surfaces. The former are usually iso-parametric curves along the boundary surfaces, and their parameter attributes can be specified by two end points in the associated surface's parameter plane. However, for an intersection curve, as in the latter case, we have to keep a list of points in the parameter domain of the associated surfaces.

After all geometrical intersections are executed for the two involved models, we refine these two models according to their intersection set. In the refining procedure, we adopt the idea of the *bottom-up* approach. We refine the lower dimensional entities before the higher dimensional ones. For example, we first refine 1D entities (Section 7.4), which are boundary curves of bounding surfaces of models. Then, we refine 2D entities (Section 7.5), which are the bounding surfaces of models. Finally, we refine 3D entities (Section 7.6), which are the shells of models.

7.4 Refinement of 1D (Curve) Nodes

This section discusses the refinement of 1D nodes required in step 2 of the procedures described in Section 7.2.

During the process of surface-to-surface intersection, the intersection curves will be stored at both intersecting surfaces. After all surface-to-surface intersections are executed, we then recall each intersection curve from bounding patches to refine models. Generally speaking, A 1D node will be created and assigned to each intersection curve. Two 0D nodes will be created and assigned to its two end points.

If a boundary curve of one model is intersected with a bounding surface of the other model at one point, then that boundary curve must be split at that intersection point. At the same time, we have to add one 0D node for that intersection point, two 1D nodes for two subdivided segments, and eliminate the original 1D node. Those new nodes have to be incorporated into the data structure of the model. See Figure 7-3 for an illustration.

An intersection curve comes from two surfaces of two models. Hence, one intersection curve will be used twice to refine both surfaces of two models. This raises the following question: should one or two nodes be assigned to the intersection points? Different answers to this question lead to different data structure.

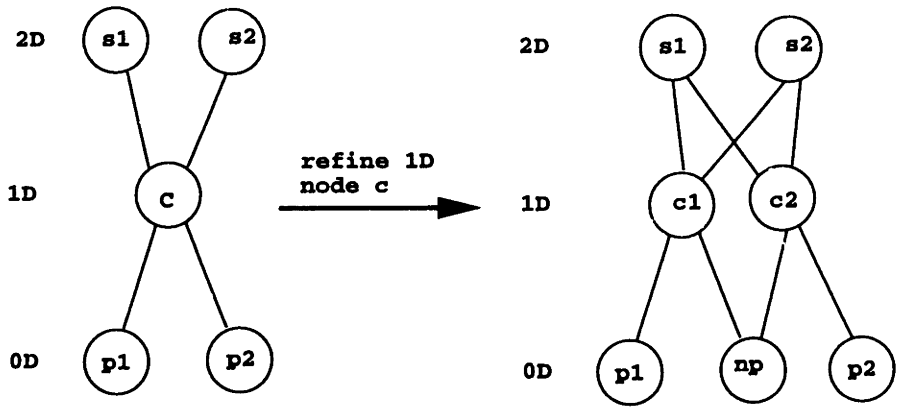


Figure 7-3: Refine 1D node c with 0D node np.

Merging and Refining: If only one new node is assigned to an intersection curve, then both models will have the same intersection curve node. This method is referred to as "*merging*" method. The data structures are merged through those intersection curve nodes. Alternatively, two nodes are assigned for the same intersection curve to their data structures. This method is referred to as "*refining*" method.

When using refining method, the data structures of the two models are refined individually. The refining method can still access the connection of two nodes for one intersection curve by adding one attribute to those two nodes. The attribute is called "same_node". The attribute "same_node" of a node is a pointer to the other node which has the same geometry of that node. To illustrate the difference between "merging" and "refining" methods, see the example shown in Figure 7-4. In Figure 7-4, two original models have two curves. Their data structures are shown in Figure 7-4(a) and (b) respectively. They intersect at one point, thus they both have to be split at the intersection point. Figure 7-4(c) and (d) show the resulting data structure of the "merging" and "refining" methods, respectively. Note that in Figure 7-4 (d), node *np* and node *np'* are the same.

Both "merging" and "refining" methods can work. There is no significant advantage of one over the other. In our implementation of the interval solid modeler, we used the "refining" method. We did so because the refining method keeps those data structures clear for the two models. For example, when point classification is called to identify vertices of a refined model to be inside or outside the other model, we have to know which model consists of which surfaces. If the data structures are merged, it is troublesome to recognize which node belongs to which original model.

Refining boundary curves with intersection curves of surfaces raises another problem, which is illustrated in Figure 7-5.

To demonstrate the problem, let a boundary curve *bc* of bounding surfaces intersect two different intersection curves at one point. These two intersection curves come from two surfaces incident through that boundary curve *bc* (see Figure 7-5).

In Figure 7-5, the boundary curve *bc* is incident to both patches Q and R. The inter-

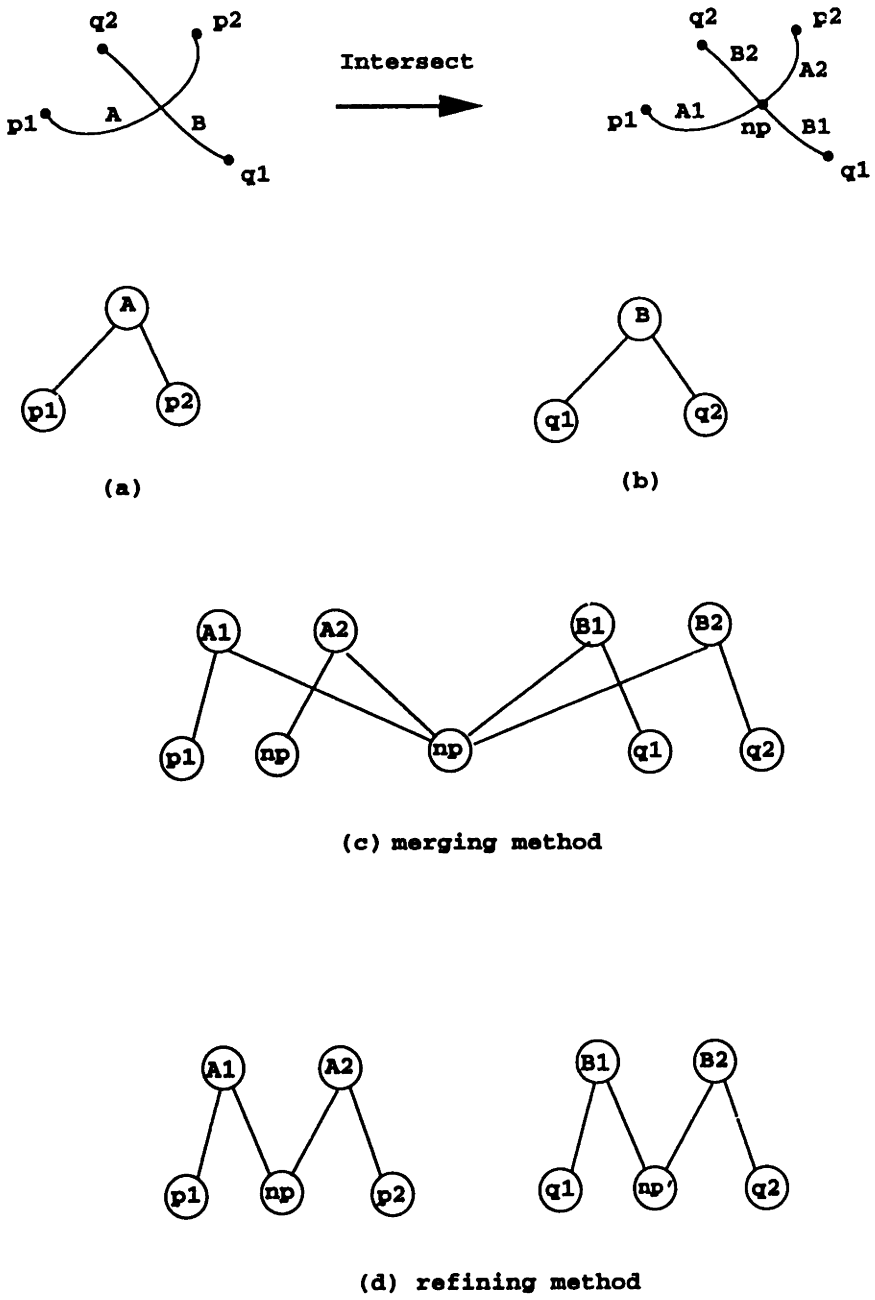


Figure 7-4: An example of "merging" method and "refining" method for two curves. (a) and (b) are data structures for two curves; (c) and (d) are respectively the resulting data structure using merging and refining methods.

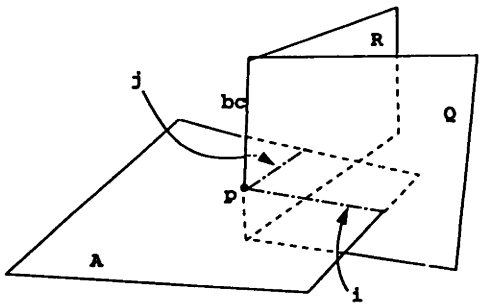


Figure 7-5: The boundary curve *bc* is intersected by two intersection curves i and j.

section curve j comes from patches R and A; the intersection curve i comes from patches Q and A. Both intersection curves i and j intersect the boundary curve *bc* at point p. The question now is: should the boundary curve *bc* be split twice by i and j at p?

The answer is clearly "no". The second splitting at the same point for a boundary curve should be avoided, because it is unnecessary and harmful to the implementation. To prevent that from happening, we check the geometry of the splitting point. Each time before assigning a 0D node to an end point p of the intersection curve, we check whether there is an existing 0D node which has the same geometry as p. If there is such a 0D node, instead of creating a new node for point p, we can associate that node to the point p.

Surfaces might intersect tangentially. For example, two surfaces might come into tangential contact at one point or along a curve. In the former case, we have to create a 0D node for that vertex. That 0D node is non-manifold for the union of the two models involved. In the latter case, the tangential intersection curve must be a collinear normal curve for both surfaces (see Chapter 4), and a 1D node must be assigned to the tangential intersection curve.

7.5 Refinement for 2D (Patch) Nodes

This section discusses step 3 in Section 7.2 in more detail.

As mentioned in Section 7.2, for a 3D modeler, the complete subdivision of one surface into separate subpatches may involve several surfaces from the other model. Therefore, in 3D modelers, we have to complete all surface intersections first, and then identify the separate regions for each bounding surface.

To refine each bounding surface of involved 3D models, first we have to identify individual subpatches of that bounding surface. Then, we refine bounding surfaces into separate subpatches. We will discuss the relationship of the separated subpatches with the intersection curves of the bounding surfaces in Section 7.5.1 and the refinement procedures for bounding surfaces of 3D models in Section 7.5.2

7.5.1 Subpatches of Bounding Surfaces

The following is a list of the relations of subpatches of one bounding surface according to its intersection curves or points, assuming intersection curves for surfaces have no self-intersections:

1. If a bounding surface has several discrete intersection points with other surface(s), then add vertices as many as intersection points to that surface (this will not subdivide the surface into separate subpatches);

2. If a bounding surface S_1 has a simple[1] intersection curve with the other surface S_2 from the boundary to the boundary of the surface S_1, then subdivide the former surface into two separate subpatches, and these two separate subpatches are bounded by the original boundary of the surface S_1 and the intersecting curve;

   Similarly, if a bounding surface has n simple intersecting curves which all cross from the boundary to the boundary of that surface, then subdivide that surface into $n + 1$ separate subpatches;

3. If a bounding surface has n intersecting curves which form a loop, then subdivide that surface into two separate subpatches. One subpatch is a trimmed patch bounded by the loop, while the other subpatch is bounded by the loop and the boundary curves of the surface.

   Similarly, if a bounding surface has several intersecting curves which form m loops, then subdivide the surface into $m + 1$ separate subpatches;

4. As a result of cases 2 and 3, if a bounding surface has several intersecting curves which from m loops and n intersecting curves crossing from its boundary to other boundary, then subdivide the surface into $m + n + 1$ separate subpatches.

In the first case, if the intersection points are not in the boundaries of both bounding surfaces, then it is a degenerate case where two surfaces tangentially contact at several points. Those intersection points are critical points whose computation is very sensitive to the numerical errors, and thus are difficult to find by using floating point arithmetic. In Chapter 4 we discuss how to compute those critical points robustly and efficiently by using interval arithmetic. In this case, both surfaces will not be subdivided into subpatches for those intersection points.

In the second case, if a bounding surface has only one simple intersection curve, the intersection curve must separate the bounding surface into two subpatches, see Figure 7-6 for an example. In Figure 7-6 patch A has only one intersection curve with patch B. Similarly, if a bounding surface is crossed by n simple intersection curves from its boundary to boundary, as shown in Figure 7-7, the intersection curve must separate the bounding surface into (n+1) subpatches, see Figure 7-7 for an example.

[1] "Simple" curve means a curve without self-intersection.

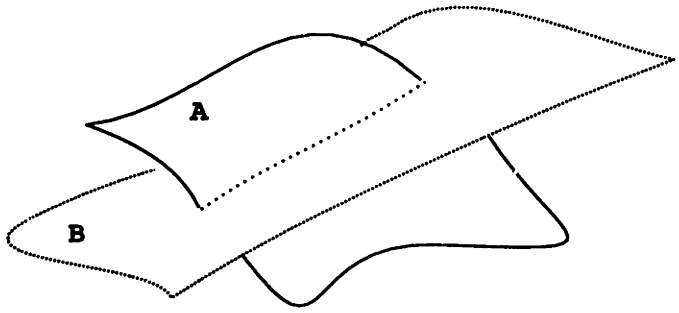


Figure 7-6: Patch A has only one intersection curve with another patch, B.

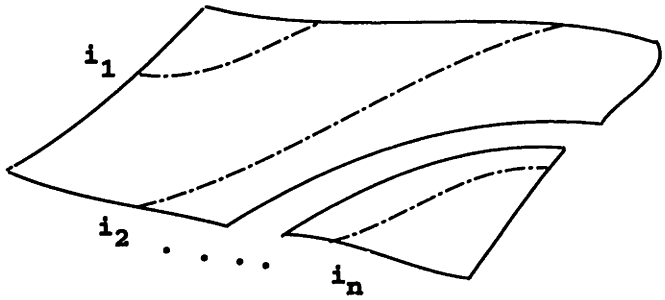


Figure 7-7: A bounding surface is subdivided into $n+1$ subpatches by n intersection curves.

In the third case, if a bounding surface has intersection curves forming a loop,(see Figure 7-8(a)), then the bounding surface (Figure 7-8(a)) has two subpatches; one is bounded by the loop, and the other bounded by the loop and its own boundary curves.

If a bounding surface has multiple loops formed by its intersection curves, see Figure 7-8(b), where $m = 2$, then surface (Figure 7-8(b)) has three subpatches.

In the fourth case, a bounding surface has m loops and n intersecting curves crossing from boundary to boundary. See Figure 7-9 for an example, in which $m = 2$, $n = 1$. Each loop will create one subpatch for the bounding surface. Similarly, so does each intersection curve crossing the bounding surface from boundary to boundary. Therefore, the resulting subpatch is $m + n + 1$.

The above list is not exhaustive for the intersection of bounding surfaces. For example, a bounding surface might have a tangential contact curve as an intersection curve. However, that tangential contact curve will not separate surfaces into subpatches. This tangential contact curve will result in non-manifold objects from Boolean operations, so we have to assign a non-manifold 1D node to it. In this section, we are only interested in cases which will subdivide bounding surfaces into subpatches, because one of the applications of the list is to check the results of refining bounding surfaces. For instance, if a bounding surface has 1 loop and 1 intersecting curve crossing boundary to boundary, we expect that the refining result for that bounding surface will have three subpatches.

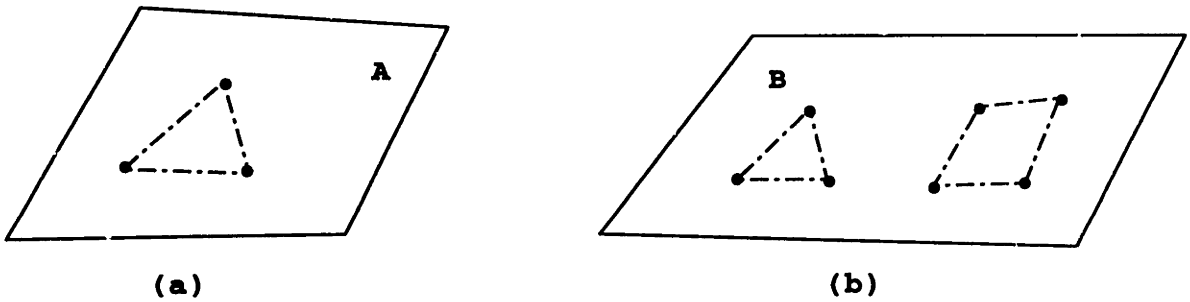


Figure 7-8: (a) A loop is formed by the intersection curves on patch A; (b) two loops are formed by the intersection curves on patch B.

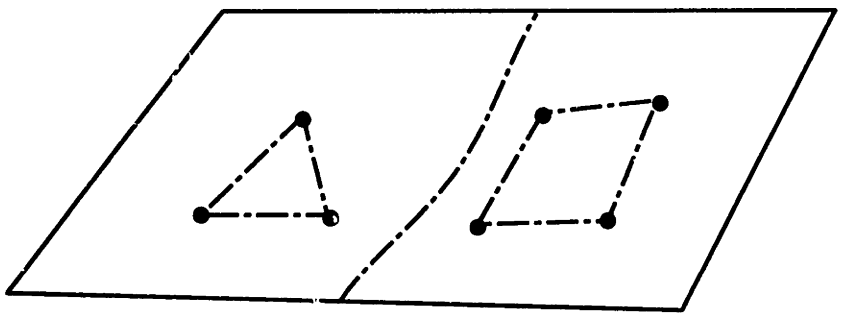


Figure 7-9: A patch with intersecting loop and curves from boundaries to boundaries.

7.5.2 Refining Patches

To illustrate a refining process for a patch, we first take the simplest example, i.e., refining a patch with only one intersection curve, see Figure 7-6. Then we extend the method to refine the general case, i.e., surfaces with multiple intersection curves.

Refine a patch with one intersection curve: To refine a patch A, as in Figure 7-6, execute the following steps:

1. Take a vertex of the intersection curve and check if it has been created:
   (a) If it has not yet been created,
      i. create a vertex node for that vertex;
      ii. subdivide the boundary edge upon which the vertex lies;
      iii. update the parameter attributes of that vertex node;
      iv. update the parameter attributes of those subdivided edges
      v. update the incidences of that vertex node;
      vi. update the incidences of those subdivided edges;
   (b) Otherwise:
      i. update the parameter attributes of that vertex node;
      ii. update the incidences of that vertex node;
   (c) If there is a vertex for an intersection curve left, go to step 1, otherwise proceed to the next step;
2. Create one edge node for the intersection curve;
   (a) update a parameter attribute of the intersection curve;
   (b) update the incidences of the intersection curve;
3. Subdivide patch A into two subpatches:
   (a) collect independent trimming loops via data structure;
   (b) create a 2D node for each trimming loop;
   (c) assign a trimmed surface for each 2D node;
   (d) associate parameter attributes of 0D and 1D nodes in a trimming loop to its newly created 2D node;
   (e) update the incidences of the new 2D nodes.

If an intersection curve touches an isoparametric boundary edge, we refer to that touching point as an *edge vertex.* Otherwise, it is a *non-edge vertex.* For instance in Figure 7-10 point p is an non-edge vertex, but points q and r are edge vertices.

Refine a patch with multiple intersection curves: Here, we refine those patches which have multiple intersection curves, see Figure 7-10:

1. Check two ends of each intersection curve. A vertex can be either an edge vertex or an non-edge vertex.

   For an edge vertex:

   (a) If a vertex node of an intersection curve has been created from the adjacent and previously processed patch:
      i. update the parameter attributes of that vertex and **the two edges incident to this vertex.**
      ii. update the incidences of the vertex node;

   (b) Otherwise:
      i. **create a vertex node** for the edge vertex;
      ii. subdivide the edge incident to this edge vertex;
      iii. update the parameter attribute of the vertex node;
      iv. update the parameter attributes of the two newly subdivided edges;
      v. update the incidences of the vertex node;
      vi. update the incidences of the two newly subdivided edges.

   For a non-edge vertex:

   (a) If a vertex node has been created from another intersection curve:
      i. update its incidences;
      ii. update its parameter attribute;

   (b) Otherwise:
      i. **create a vertex node** for the the non-edge vertex;
      ii. update its incidences;
      iii. update its parameter attribute;

2. Create edge node for each intersection curve;

   (a) update its parameter attributes;

   (b) update its incidences with its end points;

   (c) If there is any intersection curve left, retrieve it and go to Step 1, otherwise proceed to the next step;

3. Separate the patch into several regions:

   (a) collect independent trimming loops via its data structure;

   (b) create a 2D node for each trimming loop;

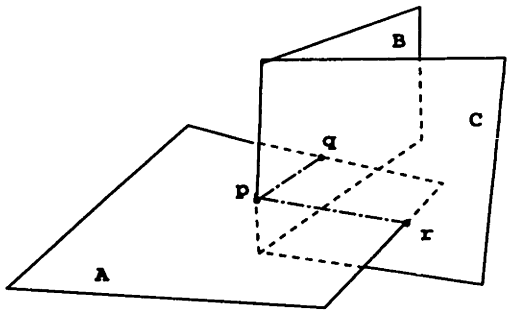


Figure 7-10: Patch A has multiple intersection curves (dash-dotted lines) with patch B and patch C.

(c) assign a trimmed surface for each 2D node;

(d) associate parameter attributes of 0D and 1D nodes in a trimming loop to its newly created 2D node;

(e) update incidences of new 2D nodes.

Parameter attributes of 2D nodes are lists of its 1D nodes and their parameter values. They are used to derive trimming loops. We have to detect those patches in which the intersection curves form a loop or loops, since those holes will result in some trimmed surfaces with inner and outer trimming loops, see Figure 7-8. Those issues are important to rendering the trimmed patches, which is discussed in Section 7.8.

7.6 Refinement for 3D (Shell) Nodes

This section discusses step 4 in Section 7.2 in more detail. The refinement procedures for 3D shells are much more complicated than that for 2D loops.

In the 2D modeler, a 2D manifold has the property that its incident 0D nodes (vertices) and 1D nodes (edges) from a loop, see Figure 6-2 for examples.

While in the 3D case, a manifold shell consists of 1D nodes (edges) and 2D nodes (patches). The question now is: do the 1D and 2D nodes of 3D manifold form a loop? The answer is "no" in general. We discuss this issue in Section 7.6.1, because it is relevant to search the bounding patches for shells. In Section 7.6.2, we develop a general algorithm to search the bounding patches for various shells, such as intersection, difference and union shells.

7.6.1 Loops for 3D Models

In the 3D case, a loop for 3D manifold shells consists of 1D nodes (edges) and 2D nodes (faces). In some situations, 3D loops can be found starting from any 1D node (edge) or 2D node (face) via orientations of faces and edges. Two examples are shown for such loops.

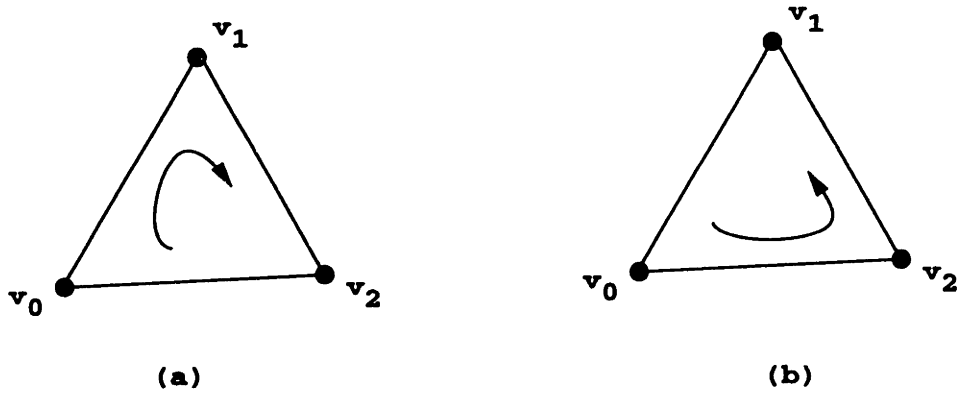


Figure 7-11: Orientations of an triangle.

However such a loop-finding procedure does not work in general. One counter example is given in Figure 7-16.

Orientation of Faces and Edges

Let us review the orientations of faces and edges. For a face with edges and vertices on the boundary, we define two orderings of its vertex set to be equivalent if they differ from one another by an even permutation [45]. The orderings of the vertices of the surface then fall into two equivalence classes. Each of these classes is called an **orientation** of the surface. We often represent an orientation of a surface by drawing a circular arrow. Take an triangle with vertices $\{ v_0, v_1, v_2\}$ for instance as shown in Figure 7-11. In fact, we can check that $\{v_0, v_1, v_2\}$ and $\{v_1, v_2, v_0\}$ are indicated by the same clockwise arrow. Similarly, we can define the orientations of an edge. We define two orderings of its vertex set to be equivalent if they differ from one another by an even permutation. The orderings of the vertices of an edge also fall into two equivalence classes. We often represent the orientation of an edge by drawing an arrow from one vertex to the other.

For convenience, we define the following orderings for a face of a 3D manifold object by using the right hand to circle the ordering of a surface: we decide the orientation of a face by crossing fingers of the right hand.

Definition 7.1 *The ordering of a surface on the boundary of a 3D manifold object is called* **inward** *ordering, if the thumb points toward the inside of the 3D manifold object. By the same token, the ordering is called* **outward** *ordering, if the thumb points toward the outside of the 3D manifold object.*

Imposing an ordering of a surface on its edges means that orderings of edges form the same ordering as that of the surface. See Figure 7-12 for an example. Conversely, we can also impose the ordering of an edge to its incident surface.

Let us see two examples of loops for 3D manifold objects; one is a tetrahedron, the other is a hexahedron. A tetrahedron with vertices $\{ A, B, C, D \}$ is shown in Figure 7-13(a)

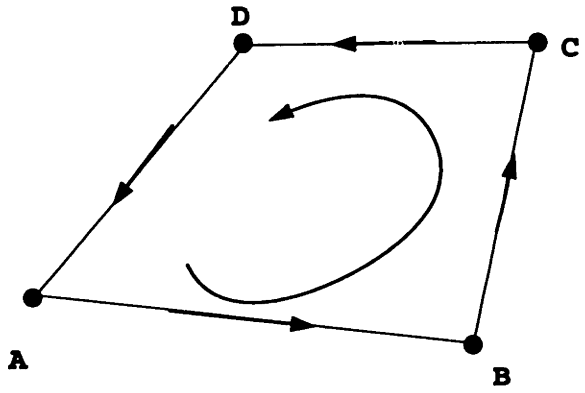


Figure 7-12: Imposing the ordering of a surface on its edges; the orderings of four edges form the same ordering of the face.

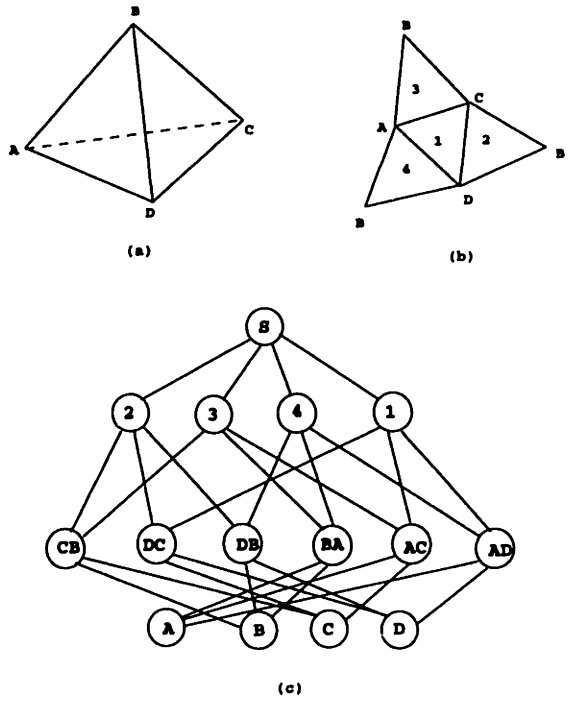


Figure 7-13: (a) A tetrahedron; (b) its resulting planar representation split at vertex B; (c) its incidence graph (data structure).

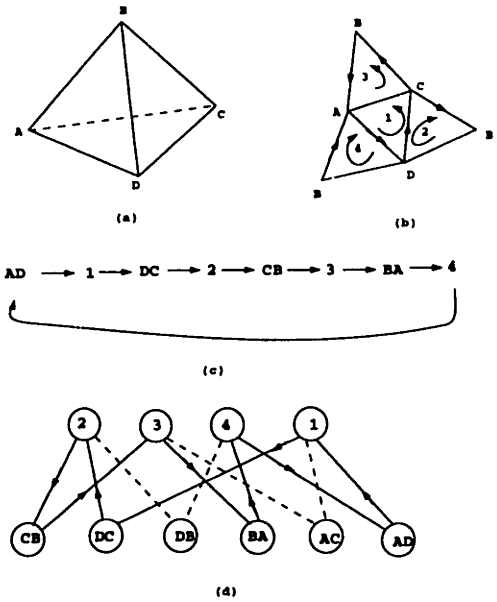


Figure 7-14: (a) A tetrahedron; (b) oriented faces and edges of the tetrahedron; (c) the loop consisting of edges and faces for the tetrahedron; (d) the path, indicated by arrows, of the loop in (c) from its (incidence graph) data structure.

and its four planes are named in Figure 7-13(b). Its incidence data structure is shown in Figure 7-13(c). If we expand the tetrahedron on the 2D plane by splitting vertex B and edges $\overline{BA}$, $\overline{BD}$, $\overline{BC}$, the resulting tetrahedron is illustrated in Figure 7-13(b). The task now is to find a loop consisting of 1D nodes (edges) and 2D nodes (faces) for the tetrahedron; no node is included more than once in such a loop.

Let us begin by taking an edge (1D node) and one of its incident faces (2D node). By assigning an inward ordering on a face, we also obtain the imposing ordering of that face to its edges. For example, we take the edge $\overline{AD}$ and its incident face 1 (see Figure 7-14(b) for illustration). Now the loop consists of $\{\ \overline{AD},\ 1\}$. Take the next edge incident to face 1, in the direction of the ordering of face 1 from $\{\ \overline{AD},\ 1\}$), i.e., edge $\overline{DC}$. Impose the ordering of face 1 to the edge $\overline{DC}$. Take the next face incident to the edge $\overline{DC}$; that is face 2. Impose the ordering of the edge $\overline{DC}$ to the face 2. (Now the loop consists of $\{\ \overline{AD},\ 1,\ \overline{DC},\ 2\}$). If we continue this process, the loop will search back to the starting edge $\overline{AD}$, (see Figure 7-14(c)). Surprisingly, the last face (face 4) added to the loop imposes the same ordering on the starting edge ($\overline{AD}$), as indicated in Figure 7-14(b). The resulting loop is shown in Figure 7-14(c). The corresponding path in the incidence data structure of the tetrahedron is shown in Figure 7-14(d).

This procedure will result in a loop including all four planes of the tetrahedron automatically, no matter which edge we start with.

The same procedure can find a loop for a cube from its data structure, see Figure 7-15.

Unfortunately, this procedure does not work for general cases. See Figure 7-16 for a

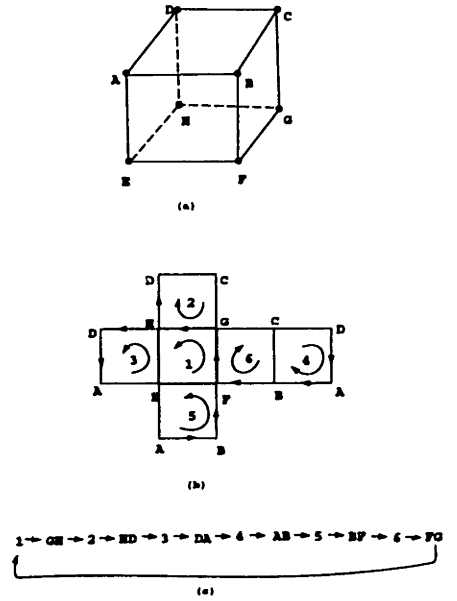


Figure 7-15: (a) A cube; (b) its oriented faces and edges; (c) the loop consisting of edges and faces for the cube.

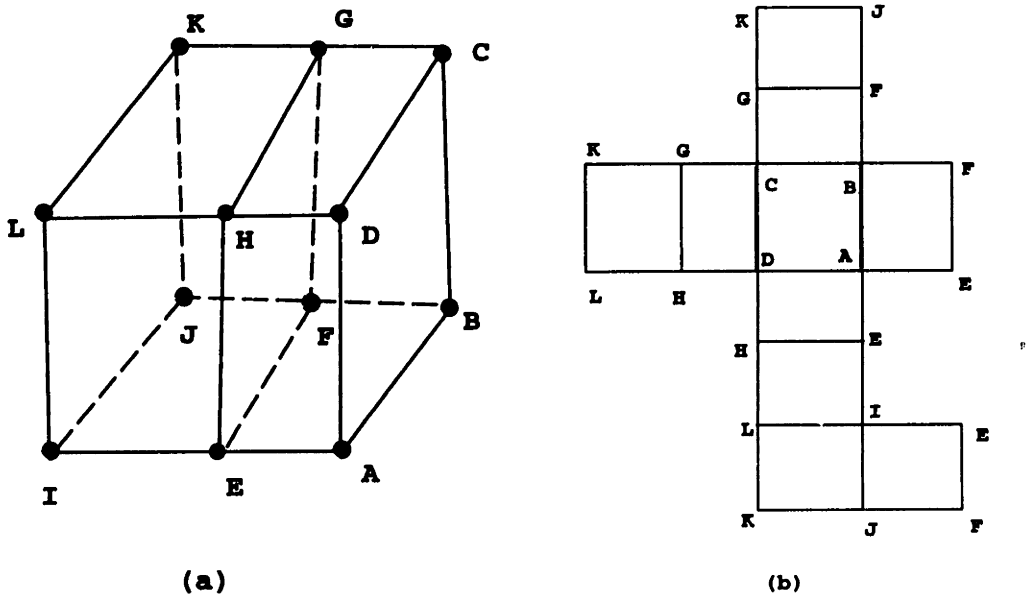


Figure 7-16: (a) A cube; (b) its resulting planar representation split at edge $\overline{KJ}$. Starting with $\overline{AB}$, we will not return to $\overline{AB}$ by employing the loop-finding procedure.

counter-example. If we start with $\overline{AB}$, by employing the aforementioned procedure, we will not return to $\overline{AB}$.

From our experience, this procedure only works for models in which each bounding patch has the same number of edges, such as tetrahedra and cubes. In the next Section 7.6.2, we discuss a way to find the bounding patches for manifold shells for general cases.

7.6.2 Shell Identification

In this section, we assume that the common parts of the two manifold models are also manifold. The case, in which two manifold models intersect at points or along tangential curves introduces non-manifold nodes, will not be discussed here.

In the 3D Boolean operations, after step 3 in Section 7.2, two models are combined into one through their intersection curves. This section discusses how to identify different shells by finding their bounding patches. Shells can be (1) the difference of model A and model B, (2) the intersection of model A and model B, (3) the difference of model B and model A, and (4) the union of model A and model B.

Nodes for intersection curves are called *intersection nodes*. Non-intersection nodes can be either *inside* or *outside* the other model. The *inside* or *outside* status of nodes can be found by the point classification algorithm in Appendix B. The *inside* or *outside* status is important for identifying different shells. For example, an intersection shell is comprised of intersection nodes and *inside* non-intersection nodes. A union shell is comprised of intersection nodes and *outside* non-intersection nodes.

We need to define some terminology for describing different shells. In Definition 6.4, $\mathbf{v_{AinB}}$ denotes a vertex of model A inside model B for 2D case. By the same token, we can define node $\mathbf{nd_{AinB}}$ for 3D case.

Definition 7.2 *Given two 3D models A and B, a node nd is denoted as* $\mathbf{nd_{AinB}}$*, if it belongs to A and is inside B, and is called a node of A inside B. Similarly a node nd is denoted as* $\mathbf{nd_{AoutB}}$*, if it belongs to A and is outside B, and is called a node of A outside B. If nodes are subdivided from one* parent *node, we call those nodes* **brother nodes**.

The brother nodes can be 1D nodes or 2D nodes, since edges and patches can be subdivided.

Definition 7.3 *Two nodes are referred to as (1)* **complement nodes**, *if they belong to the same model and have a different inside or outside status; (2)* **counter nodes**, *if they belong to different models but have the same inside or outside status; (3)* **complement-counter nodes**, *if they belong to different models and have different inside or outside status.*

For example, nd_{AoutB} and nd_{AinB} are complement nodes to each other, and so are nd_{BinA} and nd_{BoutA}. nd_{AoutB} and nd_{BoutA} are counter nodes to each other, and so are nd_{AinB} and nd_{BinA}. nd_{BoutA} and nd_{AinB} are complement-counter nodes to each other, and so are nd_{AoutB} and nd_{BinA}.

A difference shell of model A and model B is comprised of intersection nodes and nodes of A outside B, i.e. $\mathbf{nd_{AoutB}}$ and their complement-counter nodes, i.e. $\mathbf{nd_{BinA}}$. See Figure 7-17 for an example.

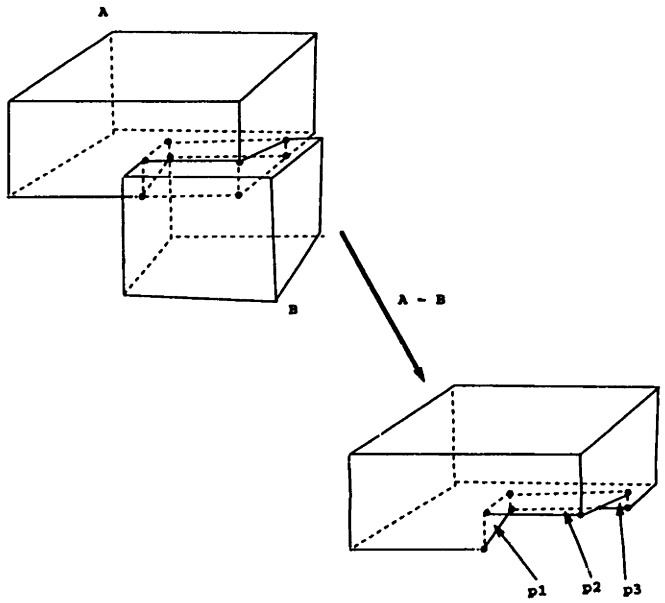


Figure 7-17: On the difference shell of (A - B), patches p1, p2 and p3 are nodes of B inside A; the rest of the patches are nodes of A outside B.

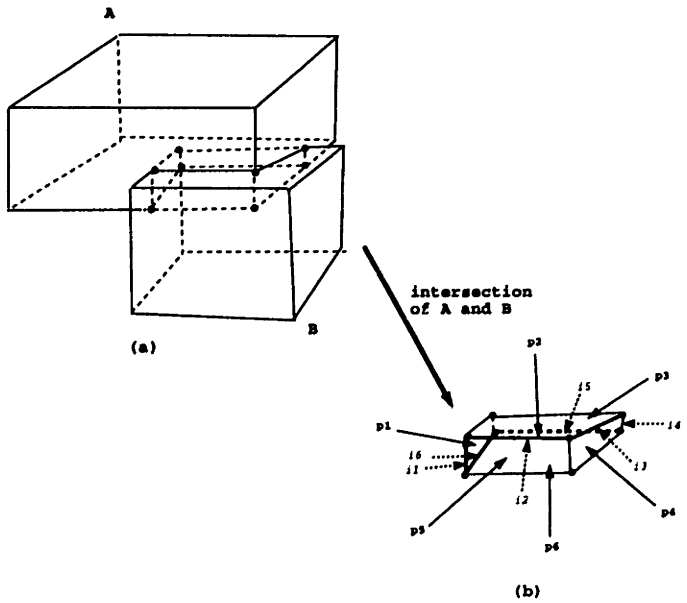


Figure 7-18: On the intersection shell of models A and B, $i1$, $i2$, $i3$, $i4$, $i5$ and $i6$ are intersection nodes (curves). They form a loop on a surface of an intersection shell. On one side of the loop, there are only $nodes_{AinB}$ such as $p4$, $p5$ and $p6$; on the other side of the loop, there are only $nodes_{BinA}$ such as $p1$, $p2$ and $p3$.

Intersection Shells: An intersection shell of models A and B consists of 1D intersection nodes, 1D $nodes_{AinB}$, 1D $nodes_{BinA}$, 2D $nodes_{AinB}$ and 2D $nodes_{BinA}$. See Figure 7-18 for an example. Note that 2D $node_{AinB}$ and 2D $node_{BinA}$ are incident through only 1D intersection nodes; see Figure 7-18 for illustration.

This is useful to find bounding patches of intersection shells of refined models. For instance, to search for the next bounding patch from one patch p, patch p's incident 1D nodes are used. If an incident 1D node of patch p is an intersection node, then the next bounding patch should be a *counter* patch of p; (recall Definition 7.3 for the meaning of counterpatch). Take the intersection shell in Figure 7-18 for example. In search of the next bounding from patch $p2$ through $i3$, $p4$ is the right candidate, not its *brother* node (refer to Definition 7.3 for *brother node*), since $p4$ is a counter patch of $p2$ through $i3$. Therefore, we can get the next patch of intersection shells from one known bounding patch p through its edge e by the following pseudo-procedure: **next_bounding_patch_of_intersection_shell**(p, e)

1. if e is a 1D *intersection* node
2. return p's counter patch through e
3. ▷ otherwise e is not an *intersection* node
4. return a patch of status *in* incident to p through e

▷ means comments in pseudo code. If e is not an intersecting curve, then the next patch will be of status *in* and belongs to the same model of p.

Hence, by identifying a bounding patch of an intersection shell, we can find bounding patches of the same intersection shell by exploiting its incident patches through its edges. Following is a pseudo procedure to recursively collect bounding patches of an intersection shell into a list. In this procedure, l is a empty list of bounding nodes, p is a known bounding patch, e is one of p's edges.

collect_intersection_shell_patches(l, p, e)

1. ▷ base case
2. **if** p and e are in the list l
3. return
4. ▷ otherwise it's the case that either p or e is not in the list
5. if p is not in the list
6. put p in the list l
7. if e is not in the list l
8. put e in the list
9. np = **next_bounding_patch_of_intersection_shell**(p, e)
10. ▷ otherwise, recursively call this routine for ne and np
11. for each edge ne of np
12. **collect_intersection_shell_patches**(l, np, ne)

We get the first bounding patch of an intersection shell from a 1D intersection node. To find bounding patches of intersection shells, employ the following steps and put bounding patches' (2D) nodes into a list along with some of their edge (1D) nodes:

1. Put all 1D intersection nodes (curves) into *list_1d_ind*;
2. Allocate an empty list of shells, *list_shell*;
3. Pick one 1D intersection node, *1d_ind*, from *list_1d_ind*;
4. Go to step 7 if *1d_ind* has been included in one of shells in *list_shell*; otherwise proceed to the next step;
5. Pick one 2D node *2d_nd* of *1d_ind* so that *2d_nd* is of status *in*, and allocate an empty list *l*;
6. Get one intersection shell by calling **collect_intersection_shell_patches**(*l*, *2d_nd*, *1d_ind*) and put it into *list_shell*;
7. Complete the process in search of intersection shells, if there is no 1D intersection node left in *list_1d_ind*; otherwise go to step 3.

In step 4, it is not necessary for the program to check whether *1d_ind* is included in one shell in *list_shell*. Instead, we can set an attribute, for instance *mark*, for each node. The default value for *mark* can be set up as *not_included*; and it is set up as *included* when its node has been put into list *l* in lines 6 and 8 of the routine **collect_intersection_shell_patches()**. Then we do not have to search the list *l* for *1d_ind*. Step 5 will be visited as many times as the number of distinct intersection shells of the two models to extract separate intersection shells. If there is no intersection between two models, then their intersection shell is empty.

Difference Shells: A difference shell of model *A* and model *B* consists of 1D intersection nodes, 1D $nodes_{AoutB}$, 2D $nodes_{AoutB}$ and their complement-counter nodes such as 1D $nodes_{BinA}$, and 2D $nodes_{BinA}$. (Please refer to Definition 7.3 for the meaning of *complement-counter.*) See Figure 7-19 for an example. Note that 2D $node_{AoutB}$ and 2D $node_{BinA}$ are incident through only 1D intersection nodes; see Figure 7-19 for illustration. Analogous to intersection shells, we can get the next patch of intersection shells from one known bounding patch *p* through its edge *e* by the following pseudo-procedure:

next_bounding_patch_of_difference_shell(*p*, *e*)
1. if *e* is a 1D *intersection* node
2. return *p*'s complement-counter patch through *e*
3. ▷ otherwise *e* is not an *intersection* node
4. return a patch of status *out* incident to *p* through *e*

If *e* is not an intersecting curve, then the next patch will be of status *out* and belongs to the same model of *p*.

Hence, by identifying a bounding patch of a difference shell, we can find bounding patches of the same difference shell by exploiting its incident patches through its edges. Similar to the pseudo procedure **collect_intersection_shell_patches()**, we can have a pseudo

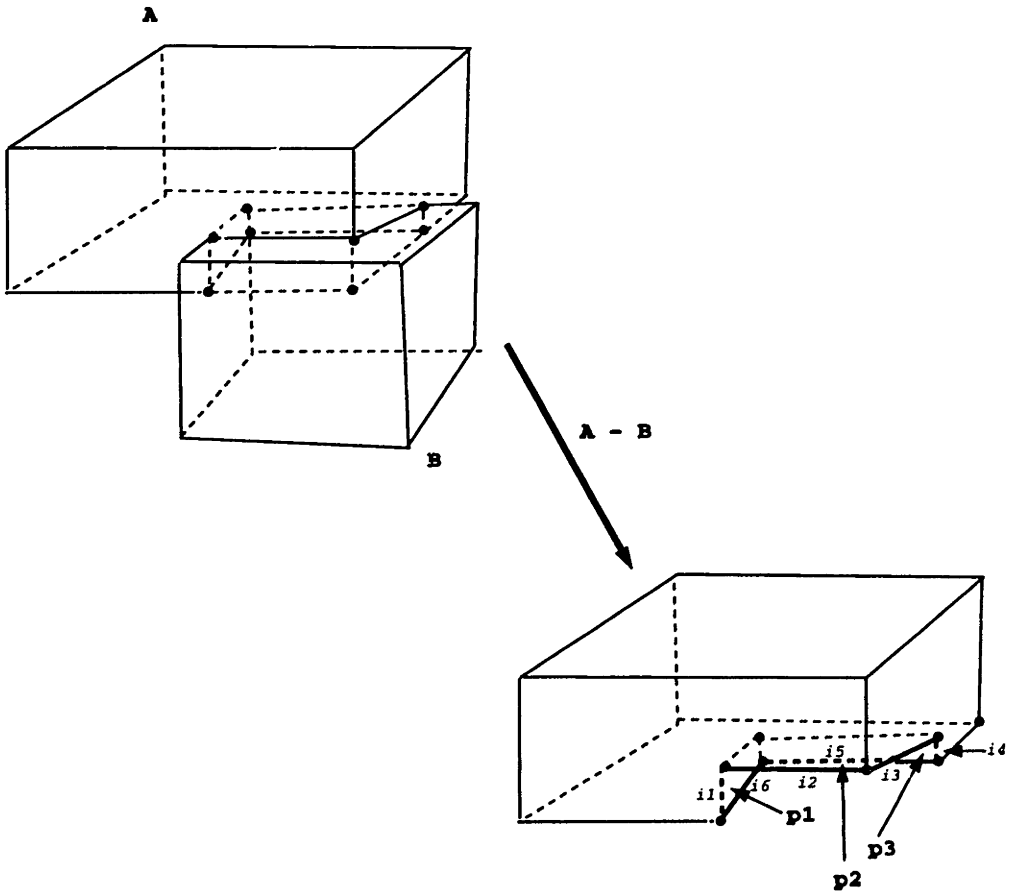


Figure 7-19: On the difference shell of models A and B, $i1$, $i2$, $i3$, $i4$, $i5$ and $i6$ are intersection nodes (curves). They form a loop on a surface of a difference shell. On one side of the loop, there are only $nodes_{BinA}$, like $p1$, $p2$ and $p3$; on the other side of the loop, there are only $nodes_{AoutB}$.

procedure **collect_difference_shell_patches()** to recursively collect bounding patches of a difference shell into a list. The only difference between these two routines is in the line 9. Instead of
np = **next_bounding_patch_of_intersection_shell**(p,e), the line 9 of the new procedure is np = **next_bounding_patch_of_difference_shell**(p,e). Of course, line 12 should be changed to **next_bounding_patch_of_difference_shell**(l, np, ne).

To find bounding patches of difference shells, employ the similar steps as for the intersection shells, and put bounding patches' (2D) nodes into a list along with some of their edge (1D) nodes. These steps differ from their intersection counterparts as follows: (1) in step 5, "status *in*" should be changed "status *out*"; (2) in step 6, instead of **next_bounding_patch_of_intersection_shell**, the procedure **next_bounding_patch_of_difference_shell** should be called; (3) in step 7, "intersection shells" should be changed to "difference shells".

Similar to the intersection case, Step 5 will be visited as many times as the number of distinct difference shells of two models to extract separate difference shells. If there is no intersection between two models, then difference of model A and model B is model A.

Union Shells : A union shell of model A and model B consists of 1D intersection nodes, 1D $nodes_{AoutB}$, 2D $nodes_{AoutB}$ and their counter nodes such as 1D $nodes_{BoutA}$, and 2D $nodes_{BoutA}$. (Please refer to Definition 7.3 for the meaning of *counter*.) See Figure 7-20 for an example. Note that 2D $node_{AoutB}$ and 2D $node_{BoutA}$ are incident through only 1D intersection nodes; see Figure 7-20 for illustration. Analogous to intersection shells, we can write a routine to get a next bounding patch from a known patch and one of its edges.

next_bounding_patch_of_union_shell(p, e)
1. if e is a 1D *intersection* node
2. return p's counter patch through e
3. ▷ e is not an *intersection* node
4. return a patch of status *out* incident to p through e

If e is not an intersection curve, then the next patch will be of status *out* and belongs to the same model of p.

Hence, by identifying a bounding patch of a union shell, we can find bounding patches of the same union shell by exploiting its incident patches through its edges. Similar to the pseudo procedure **collect_intersection_shell_patches()**, we can have a pseudo procedure **collect_union_shell_patches()** to recursively collect bounding patches of a union shell into a list. The only difference between these two routines is in the line 9. Instead of
np = **next_bounding_patch_of_intersection_shell**(p,e), the line 9 of the new procedure is np = **next_bounding_patch_of_union_shell**(p,e). Of course, line 12 should be changed to **next_bounding_patch_of_union_shell**(l, np, ne).

To find bounding patches of union shells, employ the similar steps as for the intersection shells, and put bounding patches' (2D) nodes into a list along with some of their edge (1D) nodes. These steps differ from their intersection counterparts as follows: (1) in step 5,

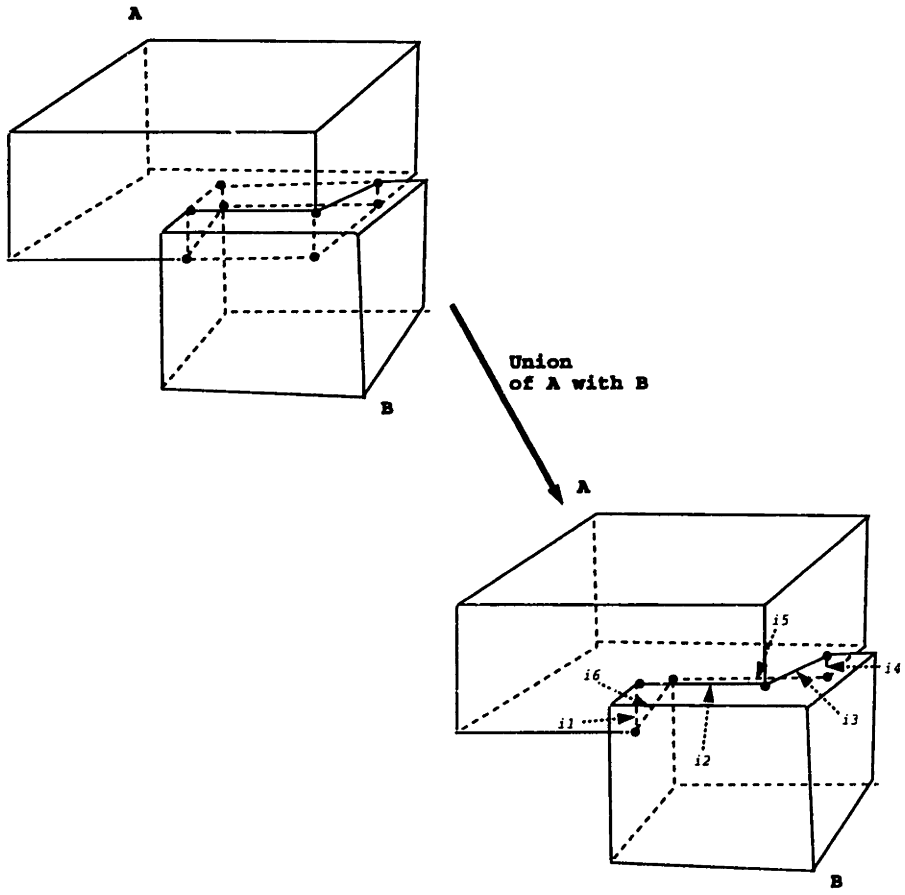


Figure 7-20: On the union shell of models A and B, $i1$, $i2$, $i3$, $i4$, $i5$ and $i6$ are intersection nodes (curves). They form a loop on a surface of a union shell. On one side of the loop, there are only $nodes_{BoutA}$; on the other side of the loop, there are only $nodes_{AoutB}$.

"status *in*" should be changed to"status *out*"; (2) in step 6, instead of **next_bounding_patch_of_intersection_shell**, the procedure **next_bounding_patch_of_union_shell** should be called; (3) in step 7, "intersection shells" should be changed to "union shells".

After collecting patches for a shell, we can examine if those patches form a valid manifold shell by checking their Euler numbers as described in the following.

Euler Number of 3D Manifold Objects: *Genus* is the number of closed patch, on a surface, which does not separate the surface into more than one region. Torus is an example with genus = 1. For a surface, a *hole* is an independent closed path which is not contractible to one point without leaving a surface on which it is embedded; see Figure 7-8 for examples.

For a 3D manifold object without genus, let V be the number of vertices, E the number of edges, and F the number of faces, then the Euler number [45] for 3D manifold objects without genus and holes is

$$V - E + F = 2 \tag{7.1}$$

For a 3D manifold object with genus and holes, let further H be the number of holes, G be the number of genus, and S be the number of shells, then the Euler number [45] for 3D manifold objects is

$$V - E + F - H = 2(S - G) \tag{7.2}$$

For a valid shell, a new model can be built from its patches along with their lower dimensional incident nodes, such as edges and vertices.

7.7 Boolean Operations

This section discusses step 5 in Section 7.2 in more detail. The operations to find shells described in Section 7.6 can be used for most regular manifold Boolean operations. It needs to be slightly modified only for some special cases, such as the case in which no intersection occurs, or the case in which tangential intersection occurs. The resulting model can be non-manifold for the latter case. The non-manifold results are discussed later in this section. In the following discussions, only the the former case is considered.

Recall the general procedures for the 3D manifold Boolean operations described in Section 7.2. Given two manifold models A and B, if they intersect, follow procedures discussed in Sections 7.4, 7.6.2. Otherwise, two models are either (1) separate from each other, or (2) one model is inside the other. Point classification algorithm discussed in Appendix B can be used to detect *in/out* status for the two models.

Intersection Operators : For case (1), there is no intersection shell. For case (2), the intersection shell is the model with *in* status.

Difference Operators : For case (1), the difference shell is the model from which we remove the difference. For example, $A \setminus B = A$. For case (2), the resulting shell has a cavity

inside it.

Union Operators : For case (1), the resulting union model has two separate shells. For case (2), the resulting union model can be represented by the outer shell.

Boolean Operations for Models with Tangential Intersection Surfaces: With the assumption that the input models are manifold, non-manifold objects can only occur in the results of Boolean operations.

Two surfaces tangential intersection can be at points, along curves, or they overlap partially. (See Chapter 4.9 for more detail). The tangential intersection of surfaces can also be categorized as either an: (1) *internal* tangential contact, i.e., one model is *inside* the other model; or (2) *external* tangential contact, i.e., one model is *outside* the other model.

Intersection Operators: For internal tangential contact, a tangential surface intersection is part of an intersection shell. The shell-identification algorithm discussed in Section 7.6.2 can identify which intersection shell the tangential surface intersection belongs to. For example, if model A tangentially contacts model B internally, then the intersection shell will be A. However, for external tangential contact, the intersection of these two models can be the tangential surface intersection itself.

Difference Operators: For both cases of internal and external tangential contacts, the shell-identification algorithm in Section 7.6.2 can extract the difference shell consisting of the tangential surface intersection. However, that node for tangential surface intersection should be set as *negative* (see Chapter 5 for *negative* node).

Union Operators: For internal tangential contact, the union shell of two models are still manifold. However, for external tangential contact, the union shell of two models is non-manifold. For example, two parallel cylinders tangentially intersect along one line externally. The node of the external tangential contact should be set as *non-manifold* node.

7.8 Rendering Trimmed Patches

To visualize a trimmed patch requires not only the trimming loops, but also the differentiation between the trimmed and non-trimmed regions. The most common way to do so is to use orientations of trimming loops. Since a trimming loop is usually represented in a parameter plane, it has two kinds of orientations, clockwise and counterclockwise.

Trimmed regions are usually bounded by an outer clockwise loop with or without inner counterclockwise loops inside it [79]. Conversely, non-trimmed regions are usually bounded by an outer counterclockwise loop with or without inner clockwise loops inside it.

In our implementation, we use *Open Inventor*^{TM} [2] to render trimmed surfaces. A user

[2] *Open Inventor*^{TM} is a registered trademark for Silicon Graphics Inc.

has to provide trimming loops in appropriate orientations for *Open Inventor*^{TM}. The trimming loops can be derived from parameter attributes of 2D nodes. Sometimes adjustment of trimming loops' orientation is needed. In Appendix A, we present an algorithm to find orientations for planar loops.

Chapter 8

Numerical Results

Representing, manipulating and interrogating continuous objects with a computer appears to be comparable to looking at physical realizations of the geometric object with a microscope which allows different magnifications. In such a setting, an approach which permits geometric interrogation with a *variable resolution* is beneficial. Hence geometrical consistency is defined within this context of variable resolution instead of as an absolute truth. Therefore, in our work, resolution can be varied easily in accordance with the user's criteria. For example, when intersecting two Bézier interval curves, the tolerance should be proportional to the width of the interval control points. Therefore, it is favorable to keep the tolerance values variable to satisfy distinct needs for solid modeling systems. In Table 8.1, it shows how the tolerance will affect the resolution of the resulting roots.

In fact, the interval spline and the tolerance in the non-linear polynomial system solver both imply the notion of fuzziness or ambiguity e.g. some resulting geometric interrogations might vary by as much as the interval widths of geometries involved or the tightness of tolerance in the polynomial equations solver. Consequently the interrogation result will further influence the results of the modeling system.

All the examples were run on a graphics workstation running at 150 MHz. We implement our data structures and our algorithms and system in the *C++* language due to advantages of its object-oriented features. This simplifies the code needed for complex geometric and topological manipulations. It certainly makes it easier for future software development. In addition, it also simplifies the data exchange with other modeling systems and therefore enhances compatibility with other systems. This chapter shows 2D (Section 8.1) and 3D (Section 8.2) examples, generated by our robust geometric modeler described in previous chapters.

8.1 Examples for 2D Objects

Example 8.1 *Two cubic Bézier curves transversally intersect each other. The control points for these two Bézier curves are curve* A*:* $(0,-1)$, $(0.25,5.0)$, $(0.5,-5.0)$, $(1,1)$, *and curve* B*:* $(1.6,-0.2)$, $(-4.4,-0.05)$, $(5.6,0.3)$, $(-0.4,0.8)$. *They are shown in Figure 8-1.*

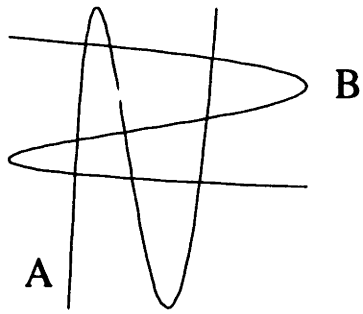


Figure 8-1: Two transversally intersecting curves.

Nine intersection points exist. Let the curve A be parametrized by u, and curve B by v. The nine intersection points are shown in Table 8.1(a) by their parameter values u and v with tolerance of $\epsilon = 10^{-4}$. We have listed only the first three intersection points for Table 8.1(b) and (c) with tolerance of 10^{-8} and 10^{-12}. In Table 8.1, we can see that the tighter the tolerance, the higher precision (resolution) the roots.

Example 2.1 (continued)

We compare the computation time and the number of resulting roots for those three systems ((1) position, (2) position and tangent and (3) position, tangent and curvature conditions) to solve example 2.1, the intersection of $y = x^4$ (parametrized by u) and $y = 0$ (parametrized by v) with the third order tangential contact (see also Figure 2-1 and Table 4.2). Table 8.2 shows the result for example 2.1. We found that at loose tolerance (see Table 8.2(a)), these three systems can be solved in approximately the same computation time and result in the same number of roots. While at tight tolerance (see Table 8.2(b)), the addition of tangent condition can greatly reduce the computation time by nine tenths with respect to system with position condition alone. The addition of tangent condition can also identify the tangential roots more accurately since the tangent condition helps remove those approximate roots around the real root (i.e. origin, in this case). In Table 8.2(b), the system with the position and tangent conditions produces 16 roots, while the system with the position condition alone produce 321 roots. We also compare the computation time with various tolerances in both rounded interval arithmetic and floating point arithmetic for example 2.1, as shown in Table 8.5(c).

Example 8.2 *Two planar Bézier curves intersecting both tangentially and transversally. The control points of these two curves are, Curve A:* $(-0.5,-0.1)$, $(0.0,3.0)$, $(0.5,-3.0)$, $(1.0,1.0)$ *and Curve B:* $(1.5,0.5)$, $(-0.5,0.4964936)$, $(-0.5,0.4)$, $(1.3,-0.2)$. *They are shown in Figure 8-2.*

They intersect at one tangential contact point and at four transversal intersection points. We can solve for these intersections automatically (ie. without manual intervention). After

$\epsilon = 10^{-4}$	u	v
(1)	[0.05832, 0.05842]	[0.12411, 0.12421]
(2)	[0.07734, 0.07744]	[0.40508, 0.40518]
(3)	[0.15455, 0.15465]	[0.96868, 0.96878]
(4)	[0.36357, 0.36367]	[0.95716, 0.95726]
(5)	[0.47115, 0.47125]	[0.46289, 0.46299]
(6)	[0.52306, 0.52316]	[0.08206, 0.08216]
(7)	[0.91839, 0.91849]	[0.04507, 0.04517]
(8)	[0.95319, 0.95329]	[0.55613, 0.55623]
(9)	[0.97968, 0.97978]	[0.89802, 0.89812]

(a)

$\epsilon = 10^{-8}$	u	v
(1)	[0.058378840, 0.058378850]	[0.124158874, 0.124158884]
(2)	[0.077400013, 0.077400023]	[0.405142146, 0.405142156]
(3)	[0.154606406, 0.154606416]	[0.968731633, 0.968731643]

(b)

$\epsilon = 10^{-12}$	u	v
(1)	[0.0583788457428, 0.0583788457438]	[0.1241588818555, 0.1241588818565]
(2)	[0.0774000183281, 0.0774000183291]	[0.4051421518690, 0.4051421518700]
(3)	[0.1546064111049, 0.1546064111059]	[0.9687316377128, 0.9687316377138]

(c)

Table 8.1: Intersections of two Bézier curves for different tolerances ϵ. In (b) and (c), only the first three roots in (a) are shown.

$\epsilon = 10^{-3}$	position cond.	position and tangent cond.	position, tangent and curvature cond.
u region	[0.4990, 0.5009]	[0.4991, 0.5008]	[0.4992, 0.5007]
v region	[0.4990, 0.5009]	[0.4991, 0.5009]	[0.4991, 0.5008]
root number	2	2	2
comput. time	0.2s	0.2s	0.2s

(a)

$\epsilon = 10^{-6}$	position cond.	position and tangent cond.	position, tangent and curvature cond.
u region	[0.4999096, 0.5001125]	[0.4999926, 0.5000068]	[0.4999926, 0.5000068]
v region	[0.4999093, 0.5001124]	[0.4999926, 0.5000067]	[0.4999926, 0.5000067]
root number	321	16	16
comput. time	4.3s	0.6s	0.6s

(b)

Table 8.2: List of root numbers, computation time and final root regions of three methods with various tolerances for intersection between $y = x^4$ parametrized by u and $y = 0$ by v. Root number is the number of roots resulting from the polynomial systems solvers for *one* actual root.

root consolidation (see Section 4.3), only one tangential contact point and four transversal intersection points are identified. The solutions are listed in Table 8.3. If we make the two curves A and B to be non-degenerate interval curves, then the resulting roots have wider range. The new interval control points for A and B are: A ([-0.5, -0.4999999], [-1.0, -0.9999999]), ([0.0, 0.0000001], [3.0, 3.0000001]), ([0.5, 0.5000001], [-3.0, -2.9999999]), ([1.0, 1.0000001], [1.0 1.0000001]) and Curve B: ([1.5, 1.5000001], [0.5, 0.5000001]), ([-0.5, -0.4999999], [0.4964936, 0.4964937]), ([-1.5, -1.4999999], [0.4, 0.4000001]), ([1.3, 1.2999999], [-0.2, -0.1999999]). The solutions for the two interval curves are in Table 8.4.

Example 8.3 *Boolean operations of two manifold models (shown in Figure 8-3). The original two models are bounded by three curves, which are Bézier curves of order 3.*

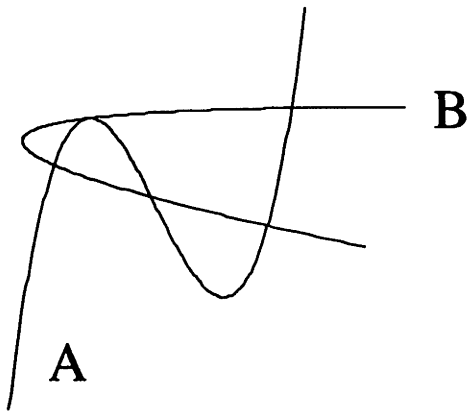


Figure 8-2: Curves intersect tangentially and transversely.

$\epsilon = 10^{-7}$		u (Curve A)	v (Curve B)
transversal	1	[0.15510467, 0.15510472]	[0.7009708, 0.70097107]
tangential	2	[0.26986901, 0.26986911]	[0.34406305, 0.34406315]
transversal	3	[0.48170107, 0.4817011]	[0.8372018, 0.83720202]
transversal	4	[0.87122356, 0.87122357]	[0.93578119, 0.93578139]
transversal	5	[0.95251551, 0.95251552]	[0.10075786, 0.10075792]

Table 8.3: Intersections of curves A (parametrized by u) and B (by v) intersecting tangentially and transversely.

$\epsilon = 10^{-7}$		u (interval Curve A)	v (interval Curve B)
transversal	1	[0.155098974, 0.155108974]	[0.70096536, 0.700975599]
tangential	2	[0.269868971, 0.269869159]	[0.344063004, 0.344063208]
transversal	3	[0.481693192, 0.481703192]	[0.837196555, 0.837203472]
transversal	4	[0.871218493, 0.871228493]	[0.935777798, 0.935784745]
transversal	5	[0.952508310, 0.952518310]	[0.100757069, 0.10076057]

Table 8.4: Intersections of interval curves A (parametrized by u) and B (by v).

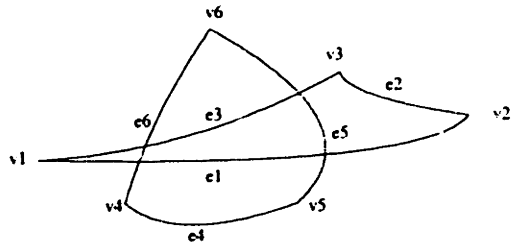
Example 8.4 *Boolean operations of one non-manifold and one manifold object (shown in Figure 8-4 and Figure 8-5). The non-manifold model is of a three-sided region with a dangling edge. The other model is of a three-sided region too. The boundary curves are all quadratic Bézier curves. Figure 8-4 shows the results of the union of two models; Figure 8-5 shows difference of two models.*

The numbers listed in the left sides of Figure 8-3(b) and (c) and Figure 8-4 (c), (d) and (f) are the dimensionalities of the nodes.

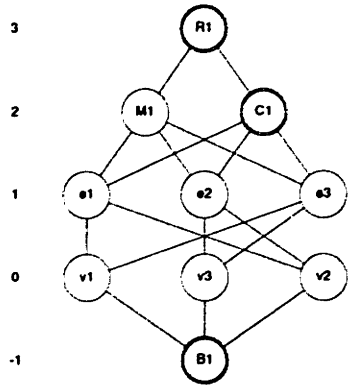
Example 8.5 *Figure 8-6 shows the creation of a non-manifold object resulting from union of manifold objects if the tolerance is specified relatively loose. The top model (Figure 8-6) is bounded by two quadratic Bézier curves and one straight line; the bottom model is bounded by three quadratic Bézier curves.*

The CPU time (in seconds) of above examples for both floating point arithmetic (FPA) and rounded interval arithmetic (RIA) are listed in Table 8.5.

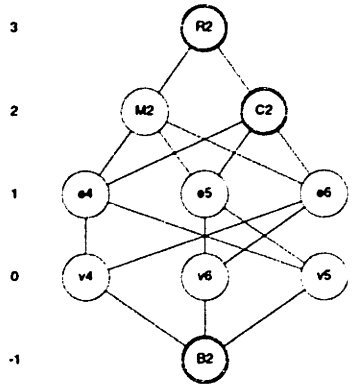
It can be seen from Table 8.5 that: (1) rounded interval arithmetic (RIA) is only one order of magnitude slower than floating point arithmetic (FPA); (2) Table 8.5(c) indicates that the tighter the tolerance is, the more significant the beneficial effect of tangential and curvature conditions becomes in reducing running time. The reason for this effect is that the additional tangent or curvature conditions help shrink the feasible regions more efficiently, so that they eliminate approximate roots around the actual root. Hence, they accelerate the overall root-finding process; (3) Table 8.5(c) shows that, when adding the tangential condition, the FPA solver misses roots. This further justifies the use of rounded interval arithmetic; and (4) the curvature has impact in reducing the number of iterations in the root finding process, see Table 4.2. However, it does not improve the overall computation



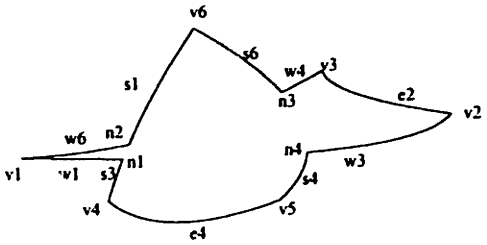
(a) Two manifold objects both bounded by three vertices and three edges.
Model M1 bounded by edges e1, e2 and e3. Model M2 by edges e4, e5, and e6.



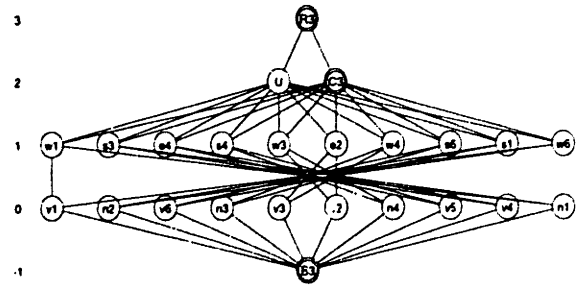
(b) The data structure of M1



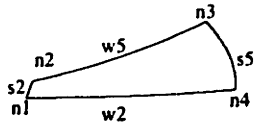
(c) The data structure of M2



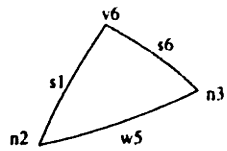
(d) The union of M1 and M2



(e) The data structure of the union

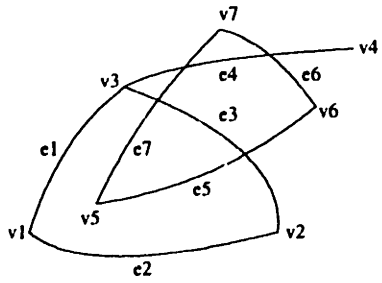


(f) Intersection of M1 and M2

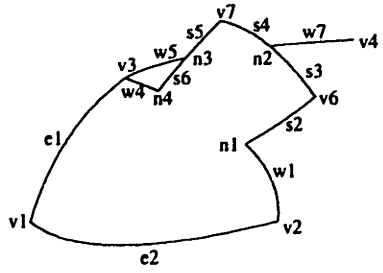


(g) One connected component of M1 - M2

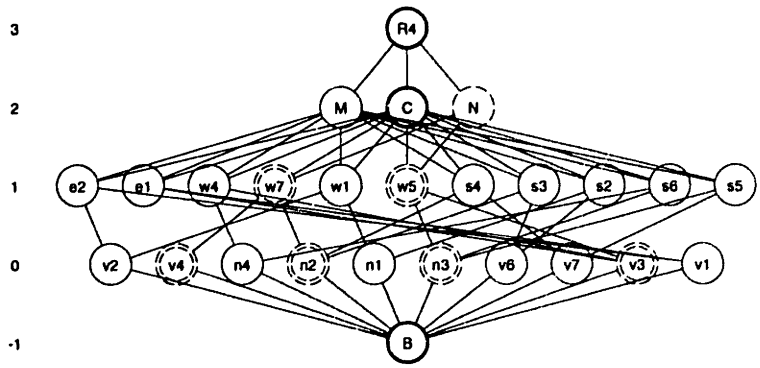
Figure 8-3: An example of manifold Boolean operations on two manifold models.



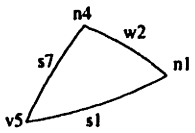
(a)Non-manifold Model M3 and manifold model M4. M3 has edges e1, e2, e3 and dangling edge e4. M4 has edges e5, e6 and e7.



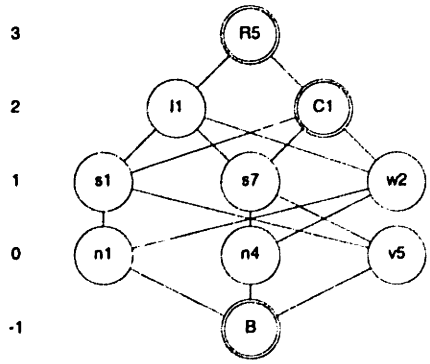
(b) The union of M3 and M4.



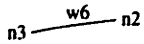
(c) The data structure for the union.



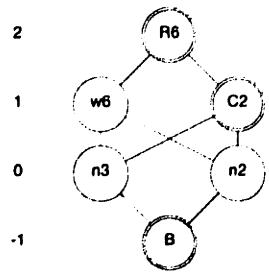
(c) One component of the intersections of M3 and M4.



(d) The data structure for the intersection component.

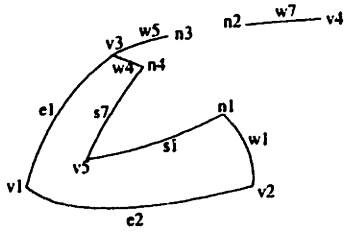


(e) The other intersection component.

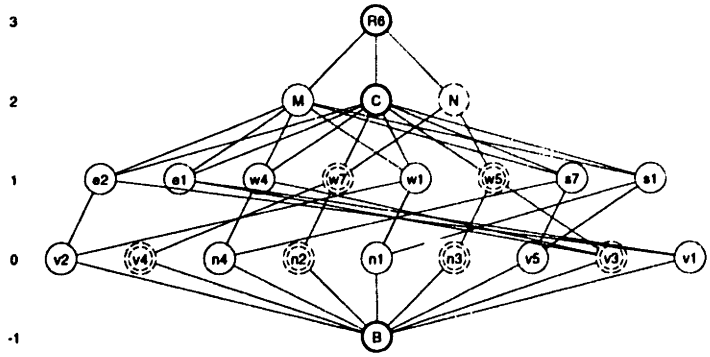


(f) The data structure for the other intersection component.

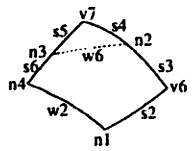
Figure 8-4: An example of non-manifold Boolean operations for one non-manifold and one manifold models.



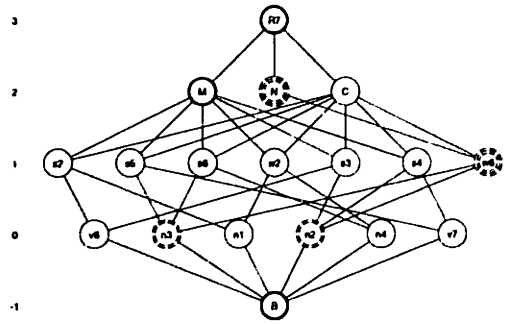
(a) Difference of Model M3 from Model M4.



(b) The data structure for the difference in (a).

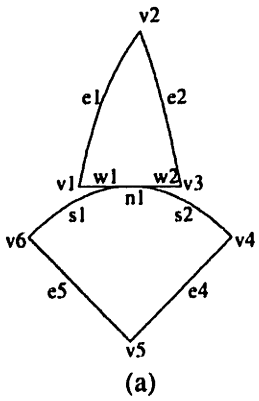


(c) Difference of Model M4 and Model M3.

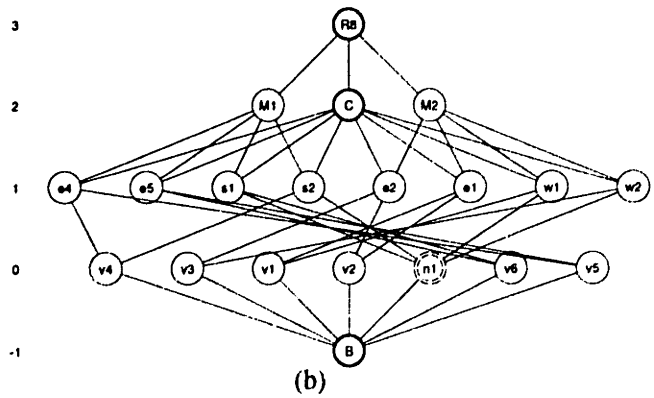


(d) The data structure for the difference in (c).

Figure 8-5: Difference operations for models M3 and M4 in Figure8-4(a).



(a)



(b)

Figure 8-6: A non-manifold object resulting from union of two manifold objects at relatively loose tolerance.

	EX 4.2	EX 8.2	EX. 8.3	EX. 8.4	EX. 8.5
FPA	0.30s	0.07s	0.21s	0.51s	0.00s
RIA	3.70s	1.00s	2.4s	5.50s	0.30s

(a)

EX. 8.1	$\epsilon = 10^{-4}$	$\epsilon = 10^{-8}$	$\epsilon = 10^{-12}$
FPA	0.038s	0.065s	0.091s
RIA	0.145s	0.780s	1.200s

(b)

	EX. 4.1 (with 2 eq.)	EX. 4.1 (with 3 eq.)	EX. 4.1 (with 4 eq.)
FPA ($\epsilon = 10^{-3}$)	0.019s	(No root) 0.005s	(No root) 0.007s
RIA ($\epsilon = 10^{-3}$)	0.200s	0.200s	0.200s
FPA ($\epsilon = 10^{-5}$)	0.026s	(No root) 0.005s	(No root) 0.008s
RIA ($\epsilon = 10^{-5}$)	0.700s	0.400s	0.400s
FPA ($\epsilon = 10^{-6}$)	0.031s	(No root) 0.008s	(No root) 0.007s
RIA ($\epsilon = 10^{-6}$)	4.300s	0.600s	0.700s

(c)

Table 8.5: (a) lists the computation time for some of examples. (b) shows the results from various tolerances for Example 8.1; (c) shows the results from various tolerances for three methods of Example 4.1

time. The reason is that the burden of solving one more equation (for curvature condition) offsets the effect of the root shrinking process. We also tested other cases like intersections of $y = x^3$ with $y = 0$. In general, using the curvature condition was not beneficial in terms of efficiency for tangential contact cases of order more than or equal to 2. Therefore, we suggest the use of position and tangential conditions only for these cases.

8.2 Examples for 3D Objects

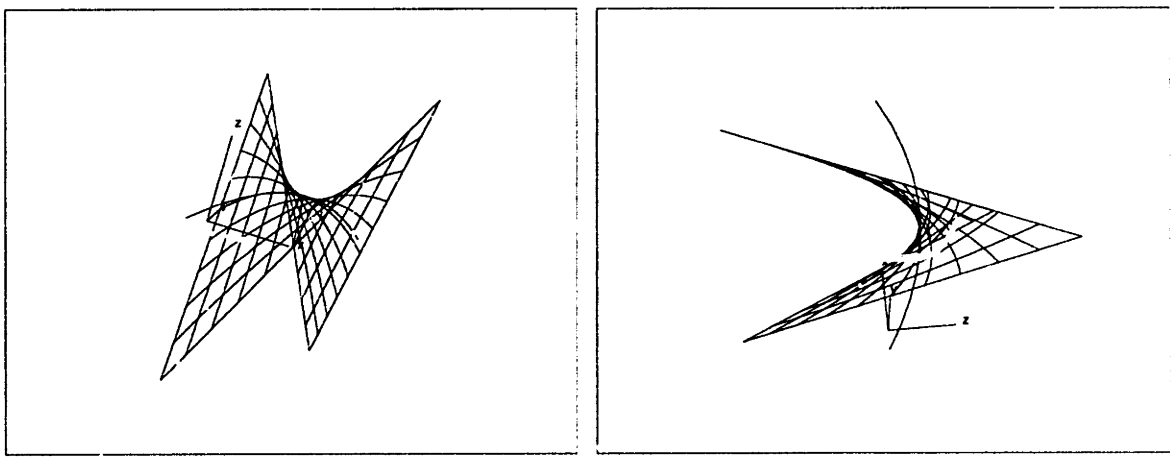
8.2.1 Curve-to-Surface Intersection

Example 8.6 Tangential intersection:

Figure 8-7 shows an example of a quadratic Bézier curve (C1) tangentially intersecting a bi-quadratic Bézier surface (S1). Their control points are listed below:

```
Control points of curve (C1):

(-0.1,  -0.1,  0.0)
( 0.5,   0.6,  0.503476985)
( 1.1,   1.1,  0.0)

Control points of surface (S1):

(0.0,  0.0,  -1.0)
(0.5,  0.0,   0.0)
(1.0,  0.0,   1.0)

(0.0,  0.5,   0.0)
(0.5,  0.5,   1.0)
(1.0,  0.5,   0.0)

(0.0,  1.0,   1.0)
(0.5,  1.0,   0.0)
(1.0,  1.0,   -1.0)
```

Table 8.6(a) shows the result of intersection of surface S1 and curve C1 with tolerance = 10^{-4}; Table 8.6(b) shows the result with tolerance = 10^{-6}. In Table 8.6(a), we can see that using only position equality condition produces loose solutions (57 roots) and takes longer time (15.8s); whereas the additional tangent condition produces more accurate solutions (2 roots) and takes less time (6.0s). The interval is reported as a single root result after root consolidation In Table 8.6. Comparing Table 8.6(a) and Table 8.6(b), we again notice that the tighter the tolerance is, the more significant the effect of the additional tangent condition becomes. This effect has been seen in Table 8.5. In fact, in this case, the additional tangent condition saves more than 96% of computation time, see Table 8.6(b). The reason



(a) (b)

Figure 8-7: Curve C1 intersects surface S1 tangentially at one point. (a) and (b) show the same objects from different views.

$\epsilon = 10^{-4}$	position cond.	position and tangent cond.
u region	[0.5202051058, 0.523057381]	[0.5215977829, 0.521775966]
v region	[0.4701899429, 0.4731015961]	[0.4717673753, 0.4718909151]
t region	[0.4751931042, 0.4775693295]	[0.4764281282, 0.4765757052]
root number	57	2
comput. time	15.8s	4.2s

(a)

$\epsilon = 10^{-6}$	position cond.	position and tangent cond.
u region	[0.5215553358, 0.521851055]	[0.5217074472, 0.521709058]
v region	[0.471665633, 0.4719593502]	[0.4718177428, 0.4718193258]
t region	[0.4763880841, 0.4766332022]	[0.4765147913, 0.4765161375]
root number	730	3
comput. time	190.3s	6.0s

(b)

Table 8.6: List of root numbers, computation time and final root intervals of two methods with various tolerances for intersection between surface S1 parametrized by u, v and curve C1 by t. Root number is the number of roots resulting from the polynomial systems solvers for *one* actual root; interval roots reported are after consolidation.

for this effect is that the additional tangent condition helps shrink the feasible regions more efficiently, and eliminates approximate roots around the actual root. Hence it accelerates the overall root-finding process.

Example 8.7 Tangential and transversal intersections:

Figure 8-8 shows an example of a curve (C2) intersecting a plane (P2) at both tangential and transversal points. Their control points are listed below:

```
Control points of curve C2:

(-0.1,   -0.1,    0.0)
( 0.5,    0.6,    0.503476985)
( 1.1,    1.1,    0.0)

Control points of plane P2:

( 0.0,    0.0,   -1.0)
( 0.5,    0.0,    0.0)
( 1.0,    0.0,    1.0)

( 0.0,    0.5,    0.0)
( 0.5,    0.5,    1.0)
( 1.0,    0.5,    0.0)

( 0.0,    1.0,    1.0)
( 0.5,    1.0,    0.0)
( 1.0,    1.0,   -1.0)
```

In Table 8.7, again we see that the tighter the tolerance is, the more significant the additional tangent condition becomes. Overall, this example takes less time than the previous example because the surface in this example is a plane.

Example 8.8 Curve and surface overlapping:

Figure 8-9 shows an example of overlap between a curve C3 and a surface S3. Figure 8-10 shows the bounding boxes of the roots of surface S3 for the overlap with curve C3. Table 8.2.1 lists the parameter regions for the overlapping curve of surface S3 and curve C3.

Their interval control points are listed below:

```
Curve C3

([0.5999999999999991,   0.6000000000000009], [-2.220446049e-16, 2.220446049e-16], [-2.220446049e-16, 2.220446049e-16])
```

$\epsilon = 10^{-6}$	position cond.	position and tangent cond.
u region	[0.7998360748, 0.7998370748]	[0.799836136, 0.799837136]
v region	[0.4999995, 0.5000005]	[0.4999995, 0.5000005]
t region	[0.8275828565, 0.8275838565]	[0.8275821152, 0.8275845979]
root number	1	1
comput. time	1.2s	0.9s

(a) Transversal root with tolerance = 10^{-6}

$\epsilon = 10^{-6}$	position cond.	position and tangent cond.
u region	[0.4560846374, 0.4560856374]	[0.4560845957, 0.4560855957]
v region	[0.4999995, 0.5000005]	[0.4999995, 0.5000005]
t region	[0.2783750397, 0.2783760397]	[0.2783749819, 0.2783759819]
root number	1	1
comput. time	1.2s	0.9s

(b) Tangential root with tolerance = 10^{-6}

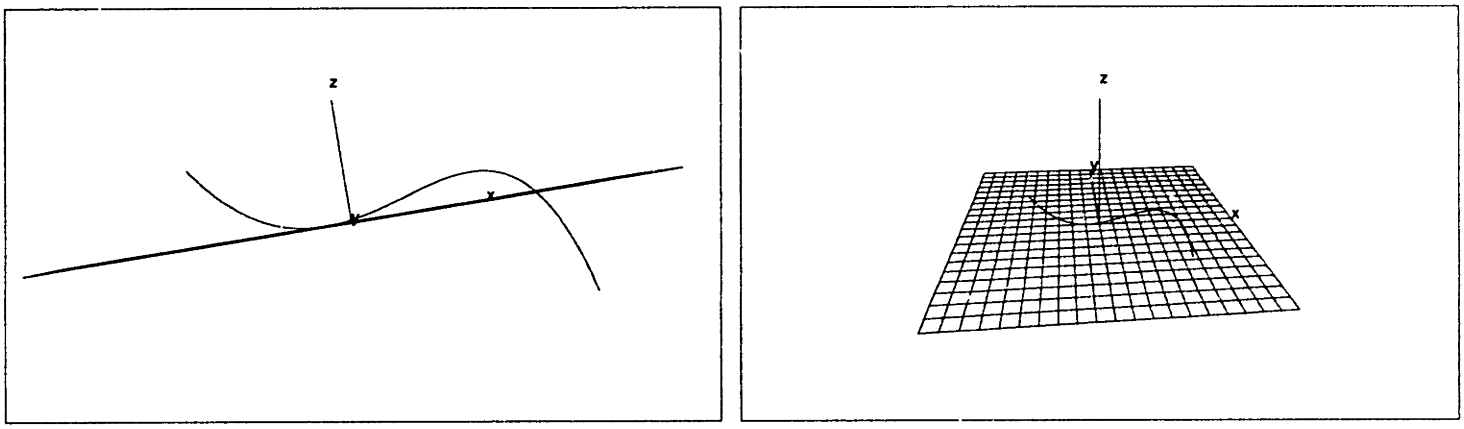
$\epsilon = 10^{-9}$	position cond.	position and tangent cond.
u region	[0.7998366355, 0.7998366365]	[0.7998366355, 0.7998366365]
v region	[0.4999999995, 0.5000000005]	[0.4999999995, 0.5000000005]
t region	[0.827583356, 0.827583357]	[0.8275833553, 0.8275833578]
root number	1	1
comput. time	6.5s	0.9s

(c) Transversal root with tolerance = 10^{-9}

$\epsilon = 10^{-9}$	position cond.	position and tangent cond.
u region	[0.4560850745, 0.4560851164]	[0.4560850952, 0.4560850962]
v region	[0.4999999995, 0.5000000005]	[0.4999999995, 0.5000000005]
t region	[0.2783754523, 0.2783755104]	[0.2783754814, 0.2783754824]
root number	62	1
comput. time	6.5s	0.9s

(d) Tangential root with tolerance = 10^{-9}

Table 8.7: List of root numbers, computation time and final root intervals of two methods with various tolerances for intersection between plane P2 parametrized by u, v and curve C2 by t. They intersect tangentially at another point and transversally at one point. Root number is the number of roots resulting from the polynomial systems solvers for *one* actual root. Interval roots reported are after consolidation.



(a) (b)

Figure 8-8: Curve C2 intersects plane P2 at both tangential and transversal points. (a) and (b) show the same objects from different views.

([0.5999999999999991, 0.6000000000000009], [0.3333333333333326, 0.333333333333334], [-0.1440000000000006, -0.1439999999999993])

([0.5999999999999991, 0.6000000000000009], [0.6666666666666653, 0.666666666666668], [-0.08640000000000037, -0.08639999999999957])

([0.5999999999999991, 0.6000000000000009], [0.9999999999999986, 1.000000000000002], [-2.220446049e-16, 2.220446049e-16])

Surface S3

([0.1999999999999992, 0.2000000000000008], [0.2999999999999988, 0.300000000000001], [0.1596671999999991, 0.1596672000000009])

([0.1999999999999993, 0.2000000000000006], [0.5333333333333318, 0.5333333333333349], [0.213695999999999, 0.213696000000001])

([0.1999999999999995, 0.2000000000000005], [0.7666666666666649, 0.7666666666666685], [0.1209599999999995, 0.1209600000000005])

([0.1999999999999997, 0.2000000000000005], [0.9999999999999988, 1.000000000000002], [-2.220446049e-16, 2.220446049e-16])

([0.4666666666666653, 0.466666666666668], [0.299999999999999, 0.3000000000000009], [0.1774079999999986, 0.1774080000000014])

([0.4666666666666656, 0.4666666666666677], [0.533333333333332, 0.5333333333333345], [0.2374399999999984, 0.2374400000000015])

([0.4666666666666659, 0.4666666666666675], [0.7666666666666654, 0.7666666666666679], [0.1343999999999993, 0.1344000000000007])

([0.4666666666666662, 0.4666666666666673], [0.9999999999999991, 1.000000000000001], [-2.220446049e-16, 2.220446049e-16])

([0.7333333333333316, 0.7333333333333351], [0.2999999999999992, 0.3000000000000007], [-0.443520000000001, -0.4435199999999989])

([0.7333333333333321, 0.7333333333333346], [0.5333333333333323, 0.5333333333333342], [-0.593600000000001, -0.5935999999999987])

([0.7333333333333325, 0.7333333333333342], [0.7666666666666658, 0.7666666666666674], [-0.3360000000000005, -0.3359999999999994])

([0.7333333333333328, 0.7333333333333338], [0.9999999999999996, 1.000000000000001], [-2.220446049e-16, 2.220446049e-16])

([0.9999999999999983, 1.000000000000002], [0.2999999999999993, 0.3000000000000005], [-2.220446049e-16, 2.220446049e-16])

([0.9999999999999989, 1.000000000000001], [0.5333333333333327, 0.5333333333333339], [-2.220446049e-16, 2.220446049e-16])

([0.9999999999999994, 1.000000000000001], [0.7666666666666663, 0.7666666666666669], [-2.220446049e-16, 2.220446049e-16])

([1.0, 1.0], [1.0, 1.0]), [0.0, 0.0])

$\epsilon = 10^{-}2$	u	v	t	time	root number
	[0.495, 0.505]	[0.0, 1.0]	[0.298108, 1.0]	26.8s	128

Table 8.8: Solution fo the overlap between surface S3 parametrized by u, v, and curve C3 parametrized by t.

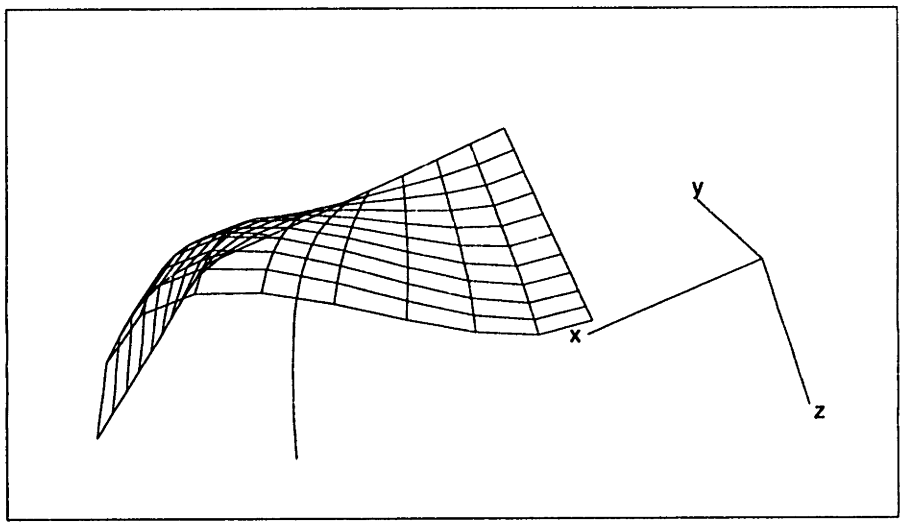
8.2.2 Critical Points

Example 8.9 *Figure 8-11 shows two surfaces, S4 and S5, which are almost parallel to each other and have a tangential contact point in the middle of both surfaces.*

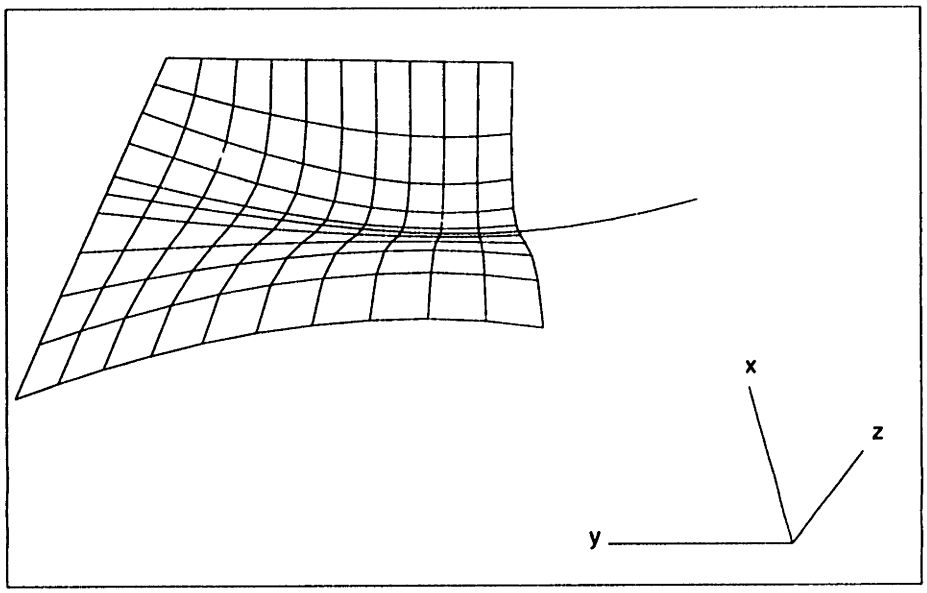
The collinear normal point (in fact, tangential contact point) is found very quickly by our method described in Section 4.9.2. The control points of the two Bézier surfaces are

```
surface S4
(-1.0    -1.0    -0.01)
(-1.0     0.0    -0.01)
(-1.0     1.0    -0.01)


(0.0     -1.0    -0.01)
(0.0      0.0     0.03)
(0.0      1.0    -0.01)

(1.0     -1.0    -0.01)
(1.0      0.0    -0.01)
(1.0      1.0    -0.01)

surface S5
(-0.5     -0.5    0.01)
(-0.5      0.0    0.01)
(-0.5      0.5    0.01)


(0.0     -0.5     0.01)
(0.0      0.0    -0.03)
(0.0      0.5     0.01)

(0.5     -0.5    0.01)
(0.5      0.0    0.01)
(0.5      0.5    0.01)
```

The result only took 4.1 second and is shown in Table 8.9.



(a)



(b)

Figure 8-9: Curve C3 is lying on surface S3. (a) and (b) show the same objects from different views.

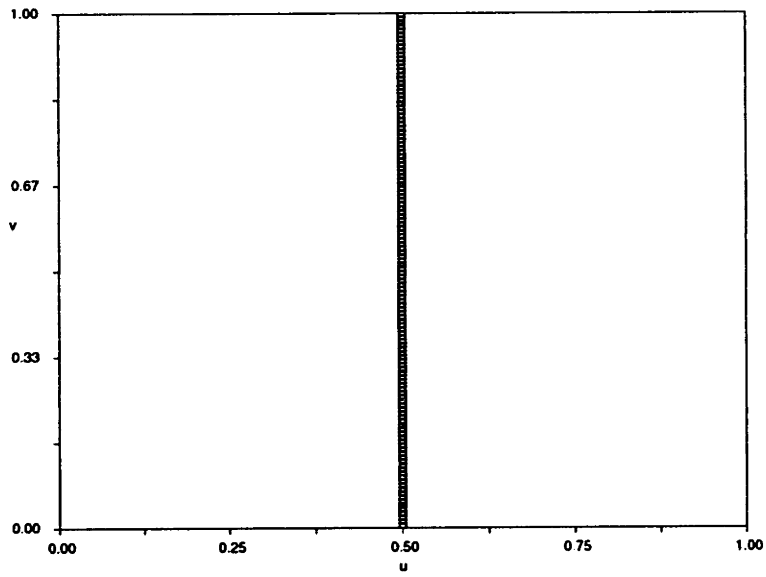
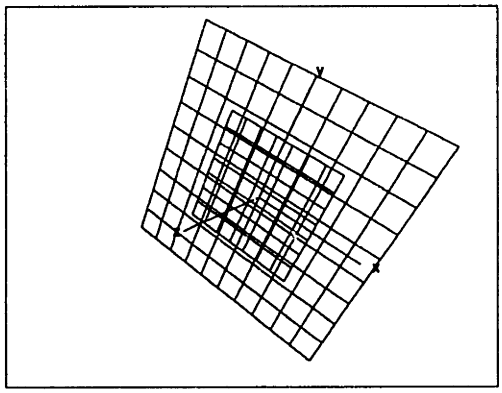


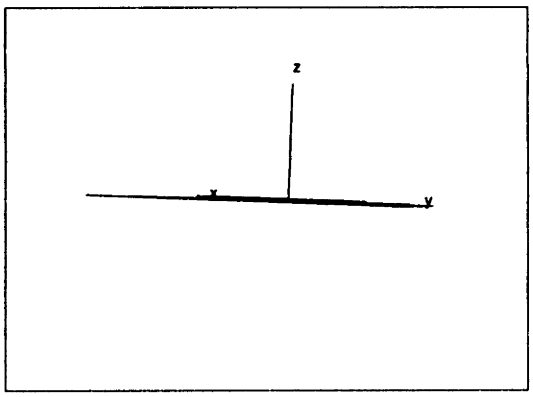
Figure 8-10: The bounding boxes in the parameter domain of surface S3 parametrized by u, v, for the overlap with curve C3.

tolerance	10^{-8}
comput. time	4.1s
surface 1 (u)	[0.499999994, 0.500000005]
surface 1 (v)	[0.499999994, 0.500000005]
surface 2 (t)	[0.499999994, 0.500000005]
surface 2 (w)	[0.499999994, 0.500000005]

Table 8.9: Critical point for intersection of surfaces S4 and S5.



(a)



(b)

Figure 8-11: Two surfaces which are almost parallel to each other have a critical point in the middle of both surfaces. (a) and (b) show the same objects from different views.

$\epsilon = 10^{-}2$	u	v	t	w	time	root number
	[0.0, 0.7413]	[0.495, 0.505]	[0.495, 0.505]	[0.3496, 1.0]	42.6s	96

Table 8.10: Solution for tangential intersection curve of parabolic cylinder S6 parametrized by u, v, and plane P3 parametrized by t, w.

8.2.3 Surface-to-Surface Intersection

Example 8.10 Tangential intersection curve:

Figure 8-12 shows an example of tangential intersection curve of a parabolic cylinder S6 and a plane P3. Their control points are listed below:

```
Surface  S6
-0.5    -0.3     0.25
 0.0    -0.3    -0.25
 0.5    -0.3     0.25

-0.5     0.8     0.25
 0.0     0.8    -0.25
 0.5     0.8     0.25

-0.5     1.3     0.25
 0.0     1.3    -0.25
 0.5     1.3     0.25

Plane P3
-1.0   -1.0   0.0
-1.0    1.0   0.0

 1.0   -1.0   0.0
 1.0    1.0   0.0
```

The solutions are listed in Table 8.2.3. Figure 8-13 shows the bounding boxes in the parameter domain. Note that the bounding boxes of the roots overlap to properly envelop the solution set.

Example 8.11 Transversal intersection curve:

Figure 8-14 shows an example of a transversal intersection curve of two surfaces. In Figure 8-14, surfaces are visualized by a net of interval points on surfaces. The darker line is their

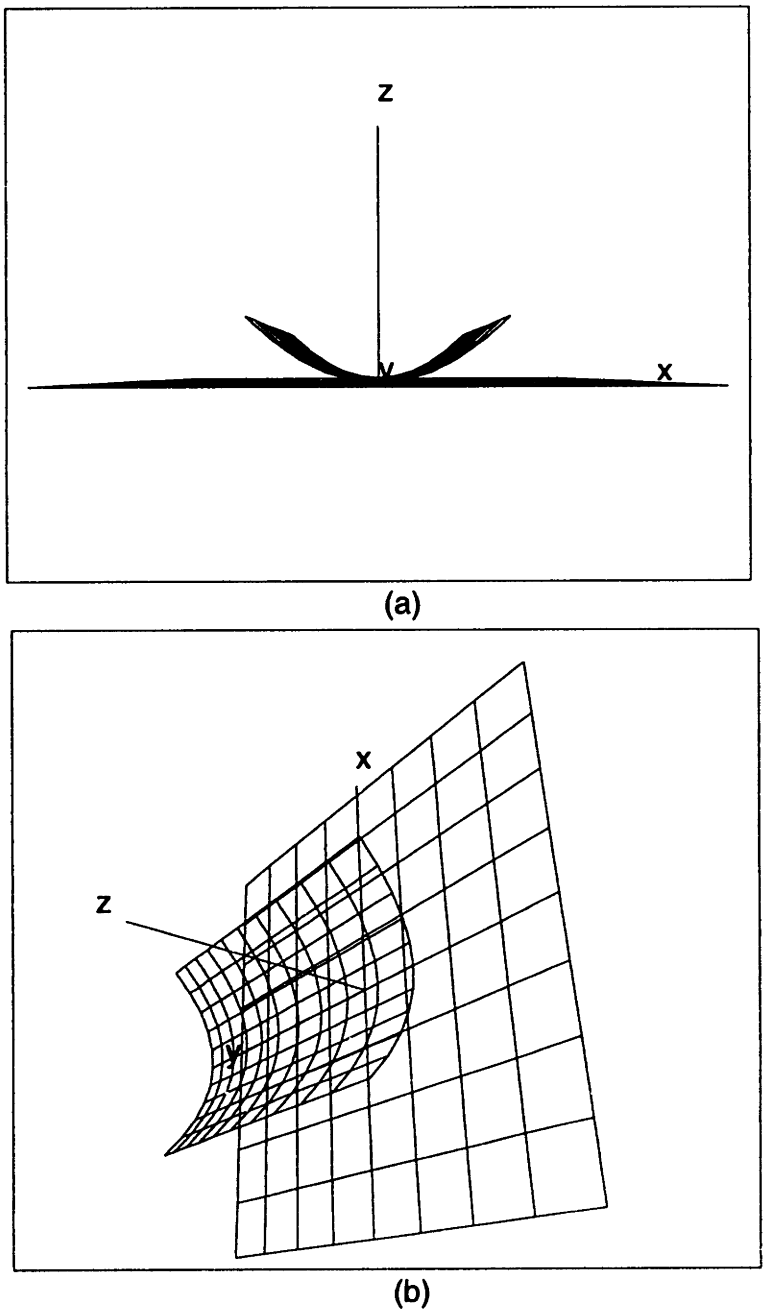
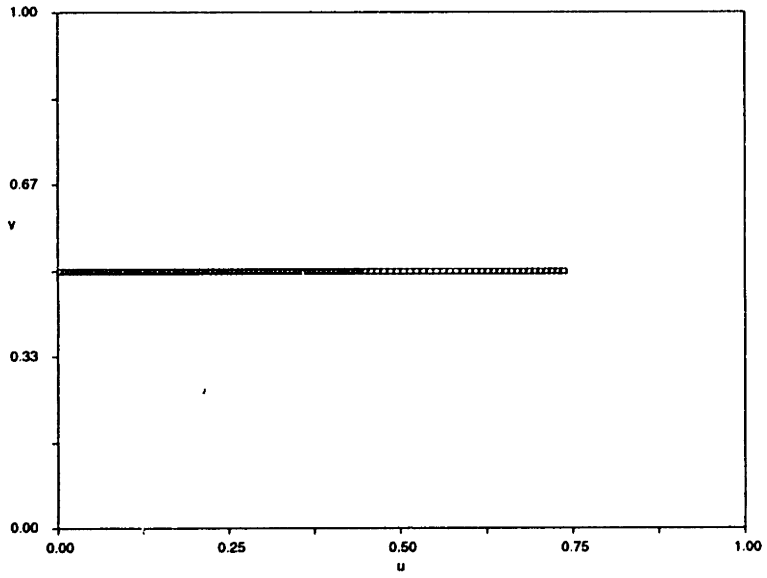
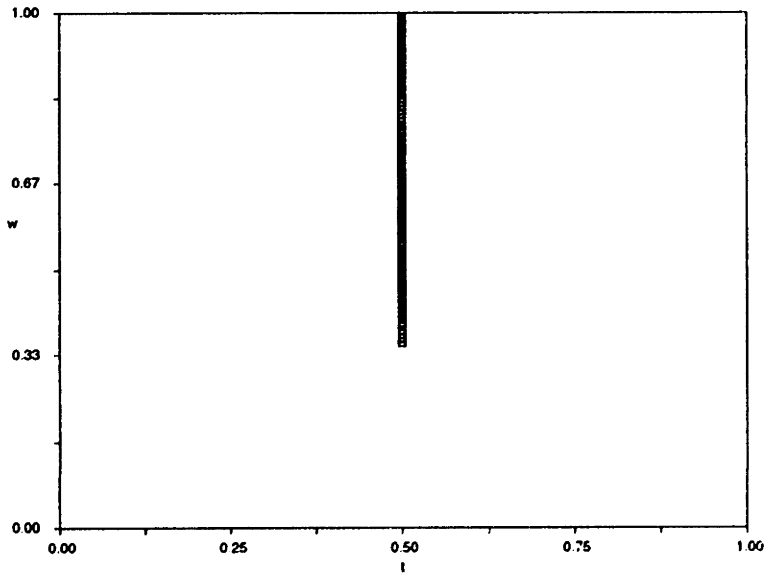


Figure 8-12: Tangential intersection of parabolic cylinder S6 and plane P3. (a) and (b) show the same objects from different views.



(a)



(b)

Figure 8-13: The bounding boxes for tangential intersection curve of parabolic cylinder S6 parametrized by u, v, and plane P3 parametrized by t, w. (a) shows the u-v parameter domain, (b) the t-w parameter domain.

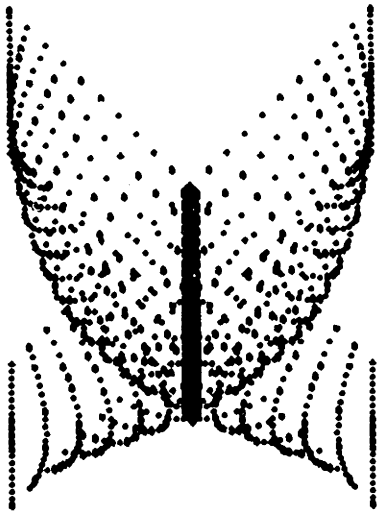


Figure 8-14: Transversal intersection curve of two surfaces.

intersection curve.

Example 8.12 Surface overlapping:

Figure 8-15 shows an example for the overlapping of two surfaces S3 and S7. These two surfaces are subdivided from the same ideal parent surface. Thus we can apply Corollary 4.1 to find the boundaries of their overlapping region. That means their overlapping region is bounded by their boundaries, even though their control points are all interval.

The control points of surface (S3) have been listed previously, so only the control points of surface (S7) are listed below:

Surface S7

([0.0, 0.0], [0.0, 0.0], [0.0, 0.0])

([-2.22044604925e-16, 2.22044604925e-16], [0.1999999999999998, 0.2000000000000002], [-2.220446049e-16, 2.220446049e-16]

([-2.22044604925e-16, 2.22044604925e-16], [0.3999999999999995, 0.4000000000000004], [-2.220446049e-16, 2.220446049e-16])

([-2.22044604925e-16, 2.22044604925e-16], [0.5999999999999991, 0.6000000000000009], [-2.220446049e-16, 2.220446049e-16])

([0.1999999999999998, 0.2000000000000002], [-2.220446049e-16, 2.220446049e-16], [-2 220446049e-16, 2.220446049e-16])

([0.1999999999999997, 0.2000000000000003], [0.1999999999999997, 0.2000000000000003], [0.3599999999999994, 0.3600000000000005])

([0.1999999999999995, 0.2000000000000004], [0.3999999999999994, 0.4000000000000006], [0.4175999999999993, 0.4176000000000007])

([0.1999999999999994, 0.2000000000000005], [0.5999999999999988, 0.6000000000000011], [0.3283199999999991, 0.3283200000000009])

([0.3999999999999995, 0.4000000000000004], [-2.220446049e-16, 2.220446049e-16], [-2.220446049e-16, 2.220446049e-16])

([0 3999999999999992, 0.4000000000000007], [0.1999999999999996, 0.2000000000000004], [0.07199999999999929, 0.07200000000000074])

([0.399999999999999, 0.400000000000001], [0.3999999999999992, 0.4000000000000009], [0.08351999999999904, 0.08352000000000102])

([0.3999999999999987, 0.4000000000000012], [0.5999999999999984, 0.6000000000000014], [0.06566399999999883, 0.06566400000000124])

([0.5999999999999991, 0.6000000000000009], [-2.220446049e-16, 2.220446049e-16], [-2.220446049e-16, 2.220446049e-16])

$\epsilon = 10^-2$	u	v	t	w	root number	time
1	[0.000, 0.000]	[0.328, 0.338]	[0.498, 1.000]	[0.000, 0.429]	64	130.2s
2	[0.331, 1.000]	[0.000, 0.000]	[0.495, 0.505]	[0.000, 0.501]	128	
3	[0.495, 0.505]	[0.000, 0.430]	[1.000, 1.000]	[0.499, 1.000]	190	
4	[0.000, 0.501]	[0.423, 0.433]	[0.331, 1.000]	[1.000, 1.000]	253	

Table 8.11: Solution for four pieces of overlapping curves to form a trimming loop for the surface overlap between surface S3 parametrized by u, v, and surface S7 parametrized by t, w.

([0.5999999999999986, 0.6000000000000014], [0.1999999999999995, 0.2000000000000005], [-0.08640000000000073, -0.08639999999999926])

([0.5999999999999982, 0.6000000000000018], [0.399999999999999, 0.400000000000001], [-0.100224000000001, -0.100223999999999])

([0.599999999999998, 0.6000000000000021], [0.5999999999999981, 0.6000000000000018], [-0.07879680000000111, -0.07879679999999885])

Figure 8-15(a) and (b) show surfaces S3 and S7 from different views. Figure 8-15(c) shows the overlapping patch and S3. Figure 8-15(d) shows the overlapping patch alone.

8.2.4 3D Boolean Operations

Example 8.13 *Boolean operations of a cube and a tetrahedron.*

In the following figures, objects are accompanied with their data structures. However, the data structures are shown only for the visualization purpose, so the numbers of entities (vertices, edges, patches, and shells) can be clear to the reader. We do not intend to show the corresponding nodes to the entities, as we do in the 2D cases (see Figures 8-3 to 8-6). This is because the resulting figures from 3D Boolean operations are much more complicated than those from 2D Boolean operations. It is not only tedious to specify the correspondences of the nodes to the 3D entities, but also difficult to show the correspondences clearly for 3D objects in a 2D screen.

Figure 8-17 shows the cube and its data structure; Figure 8-18 shows the tetrahedron and its data structure.

Figure 8-19 shows the original configuration of the cube and tetrahedron.

Figure 8-20(a) shows the union of the cube and the tetrahedron. Its data structure is shown in Figure 8-20(b).

Figure 8-21(a) shows the difference of the cube from the tetrahedron. Its data structure is shown Figure 8-21(b). Notice that the result is homeomorphic to a torus. This is identified by Figure 8-21(a), in which an passage in the center of the resulting object is shown.

Figure 8-22(a) shows the top of the difference of the tetrahedron from the cube and (b) its corresponding data structure.

Figure 8-23(b) shows the bottom of the difference of the tetrahedron from the cube, and (b) its corresponding data structure.

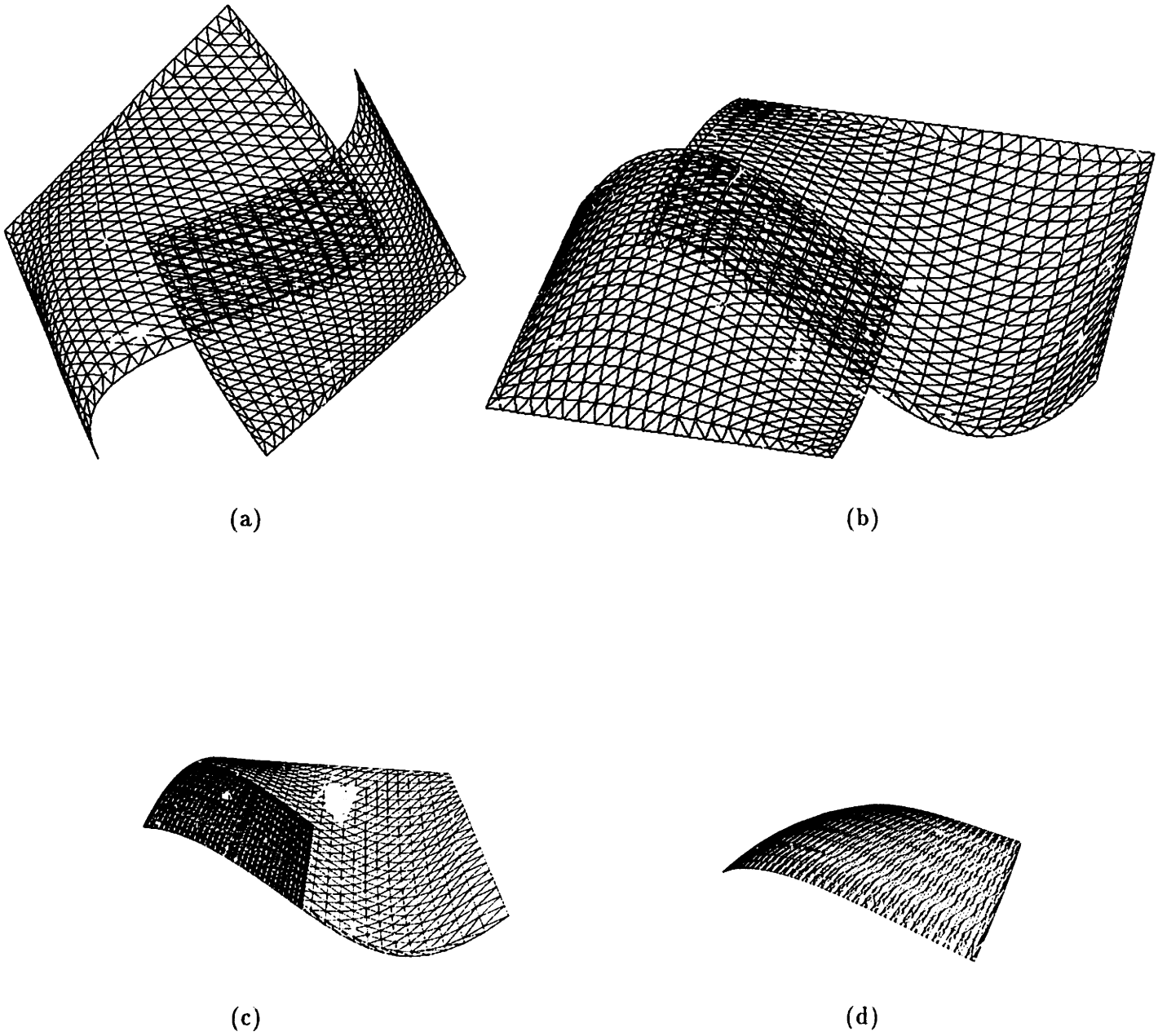
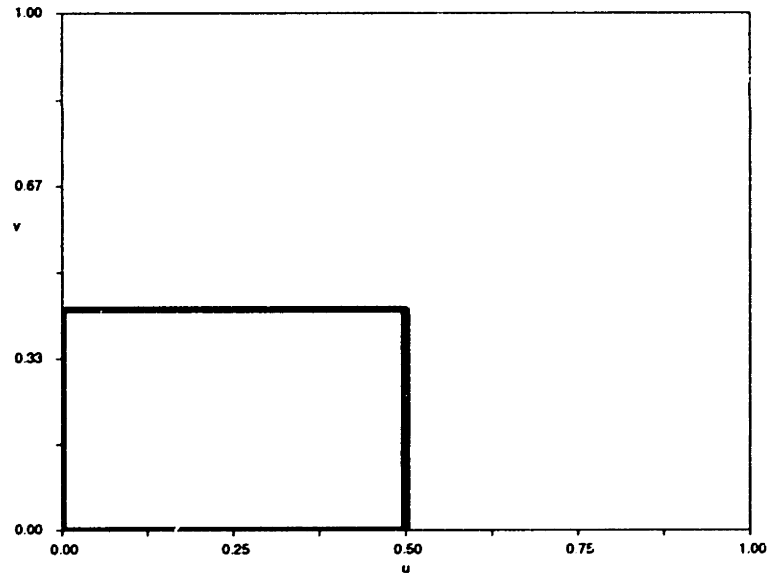
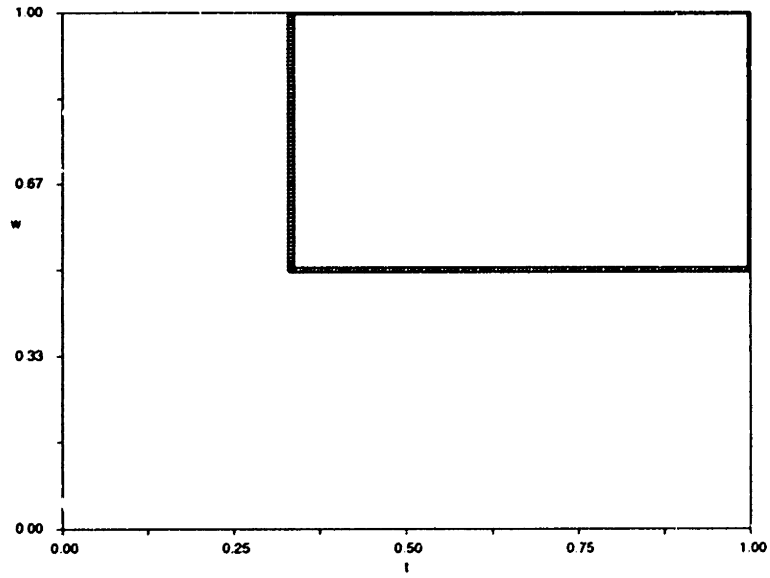


Figure 8-15: Surface S3 and surface S7 overlap partially. (a) and (b) show the same objects from different views. (c) shows the overlapping patch and surface S3. (d) shows the overlapping patch alone.

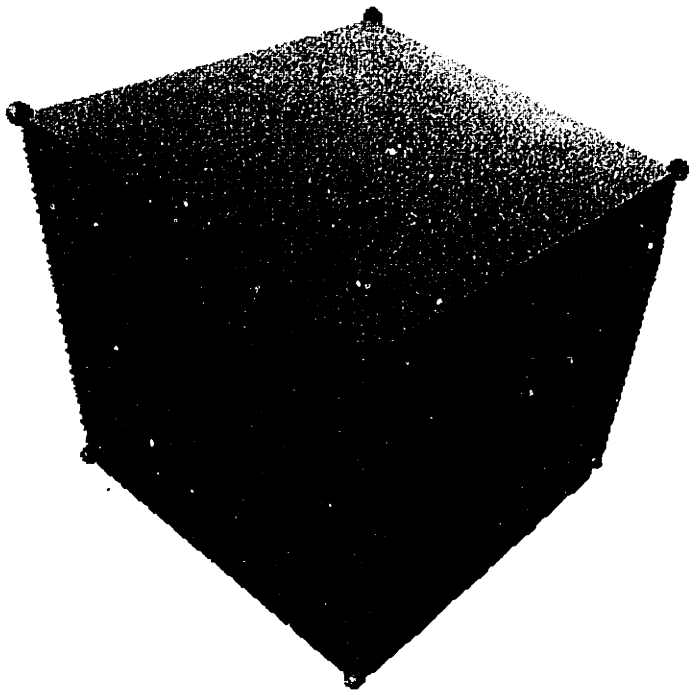


(a)

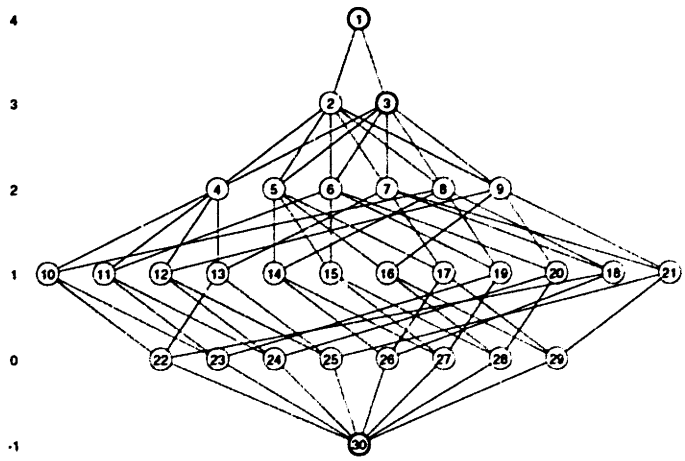


(b)

Figure 8-16: The bounding boxes for the trimming loop of the overlap between surface S3 parametrized by u, v, and surface S7 parametrized by t, w. (a) shows the (u-v) parameter domain of S3. (b) shows the (t-w) parameter domain of S7.

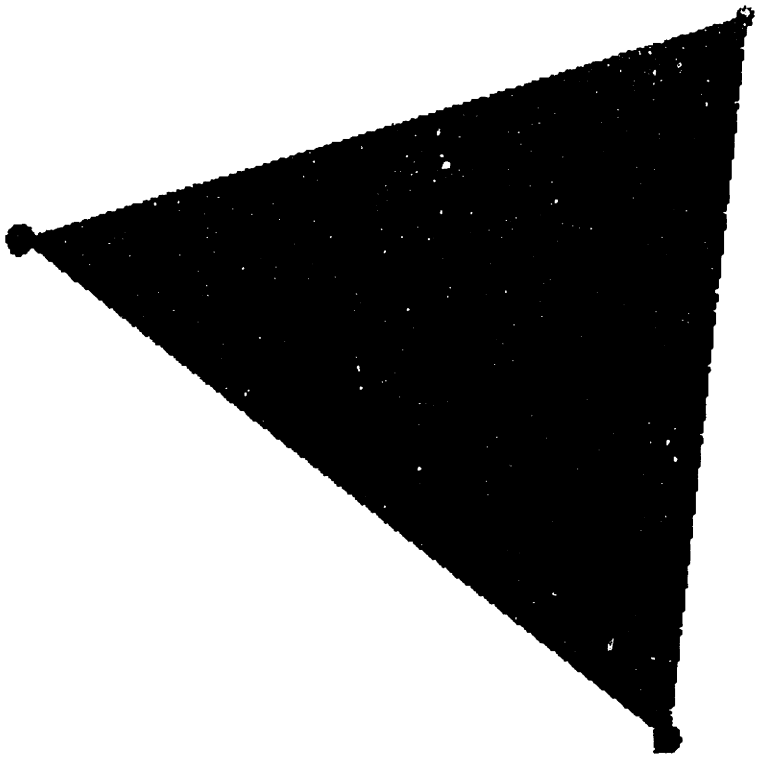


(a)

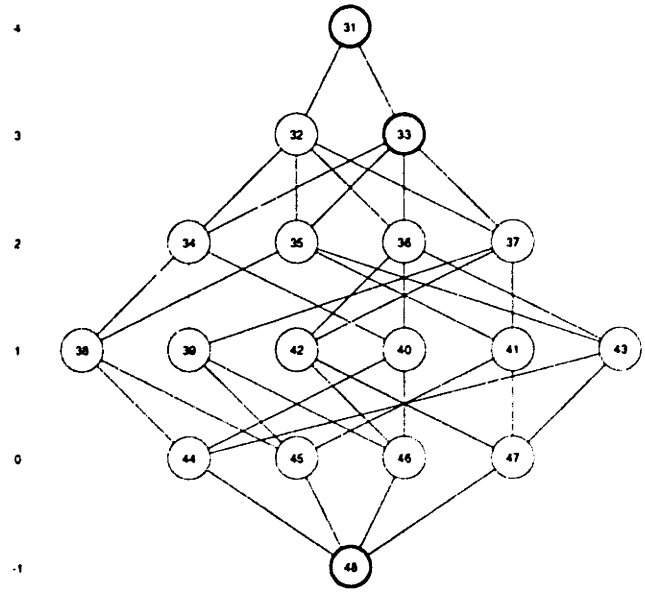


(b)

Figure 8-17: A cube and its data structure.



(a)



(b)

Figure 8-18: A tetrahedron and its data structure.

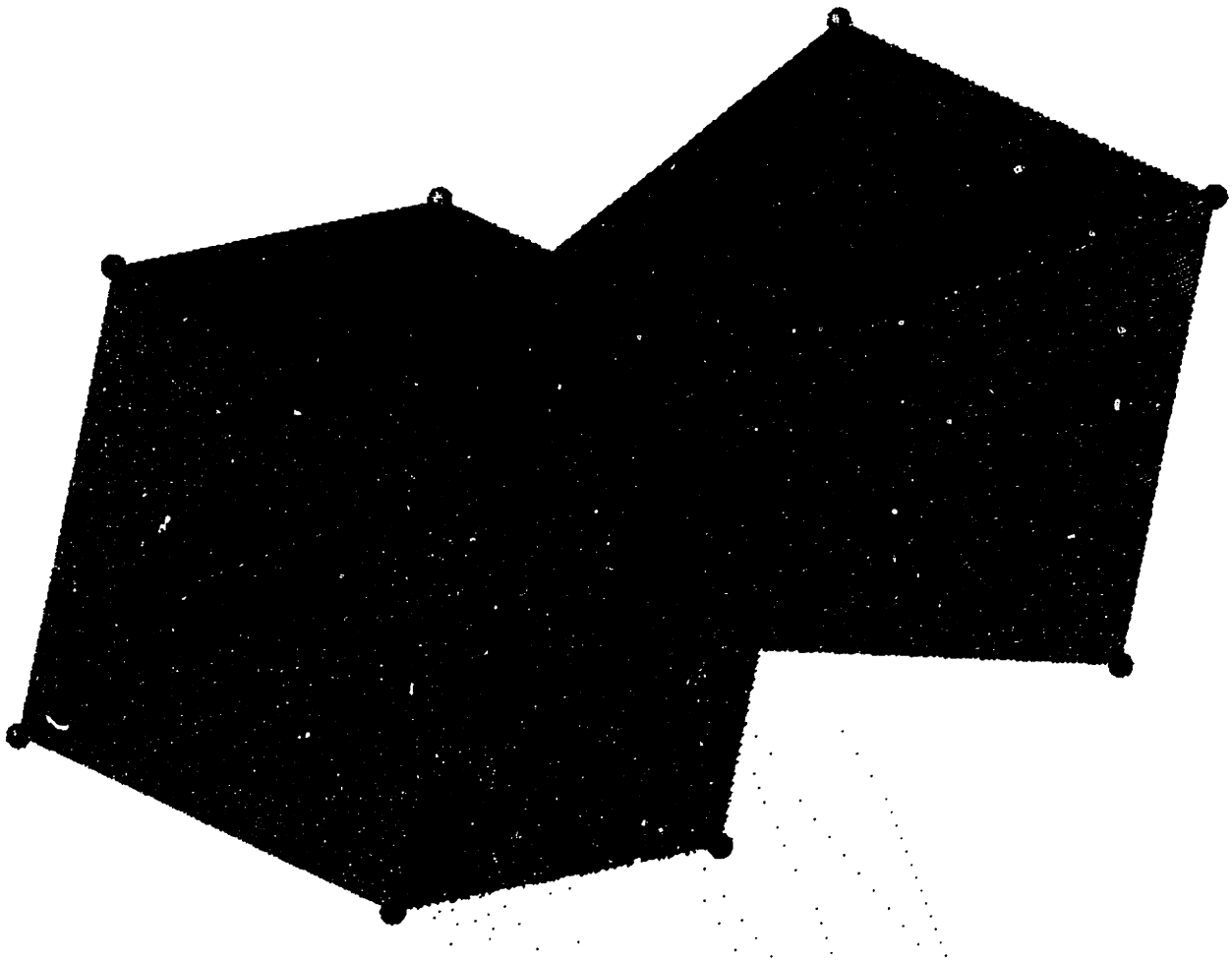
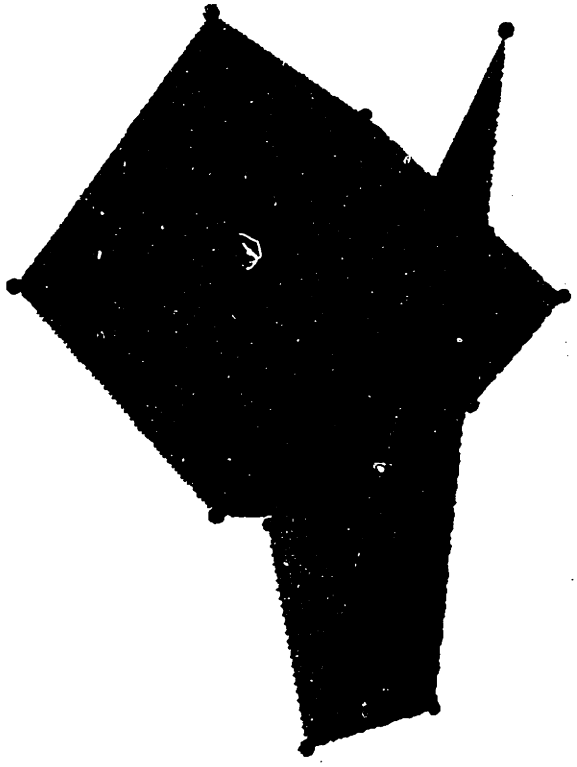
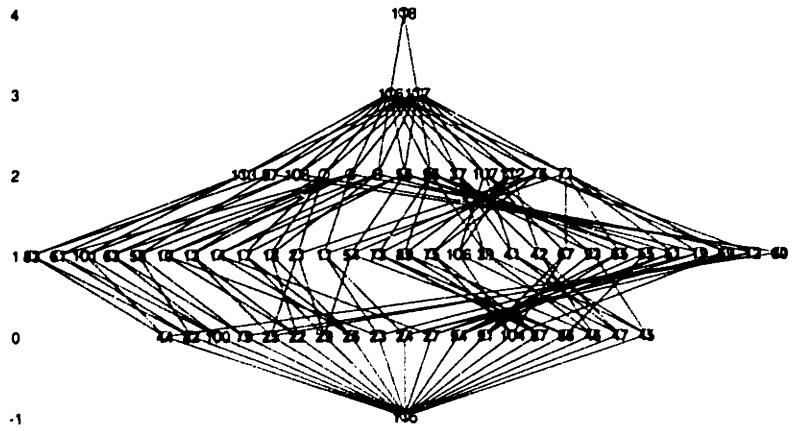


Figure 8-19: The original configuration of the cube and the tetrahedron.

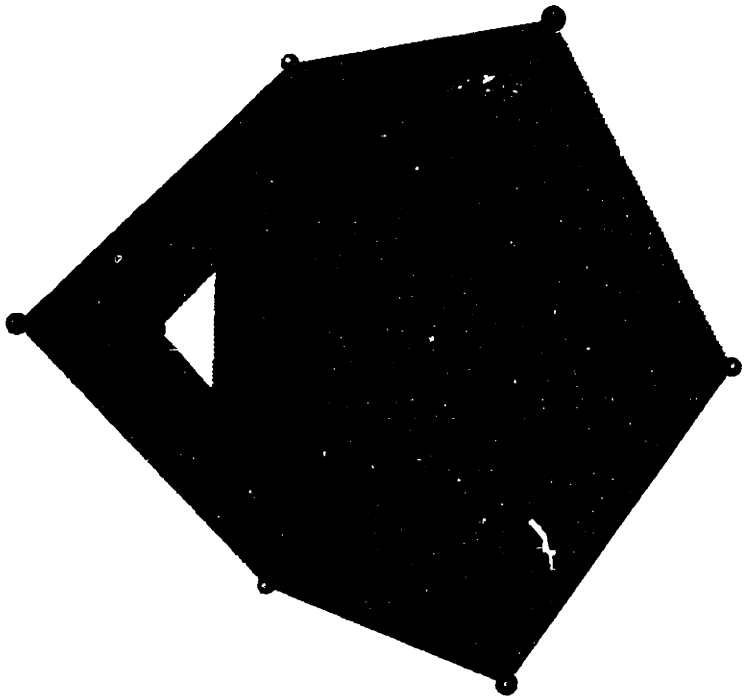


(a)

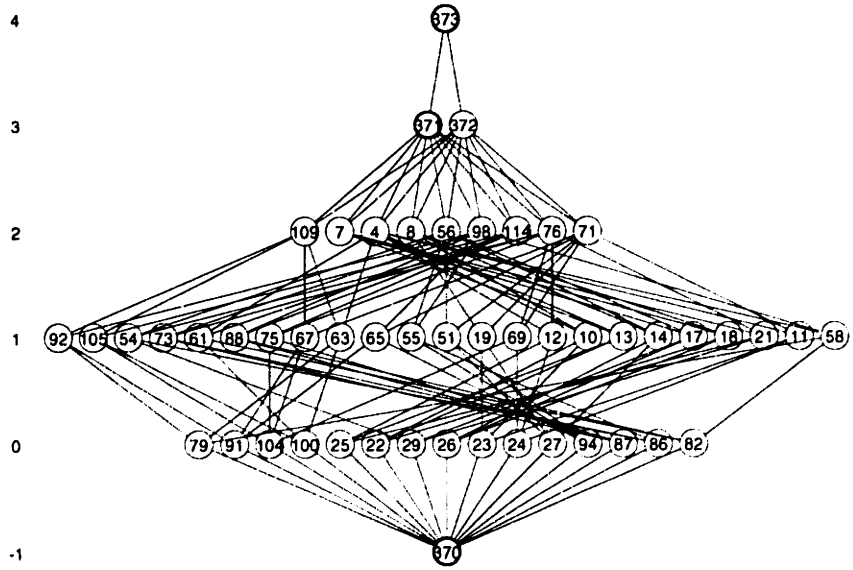


(b)

Figure 8-20: The union of the cube in Figure 8-16 and a tetrahedron in Figure 8-17 and its corresponding data structure.

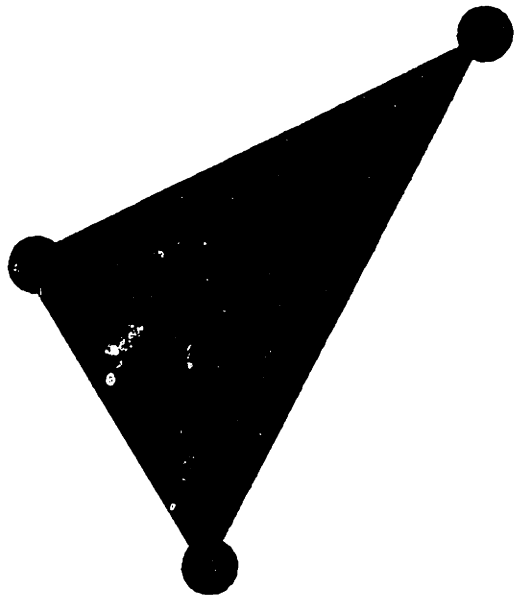


(a)

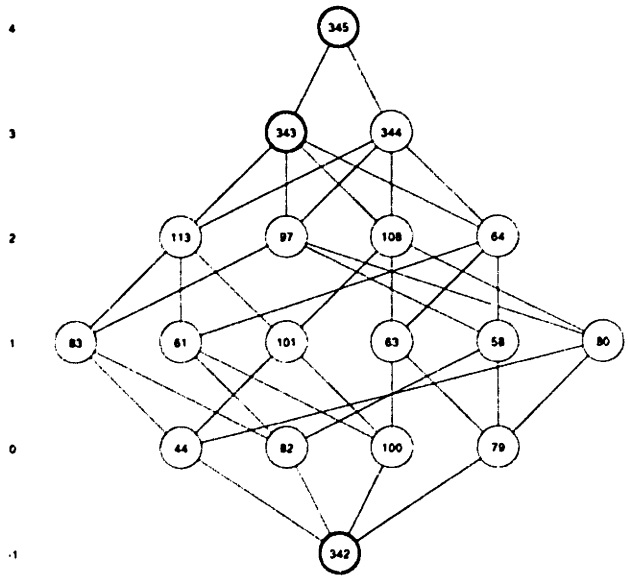


(b)

Figure 8-21: The difference of the cube from the tetrahedron and its data structure. Note that there is an passage in the center (shown as white space) due to the subtraction of the tetrahedron.

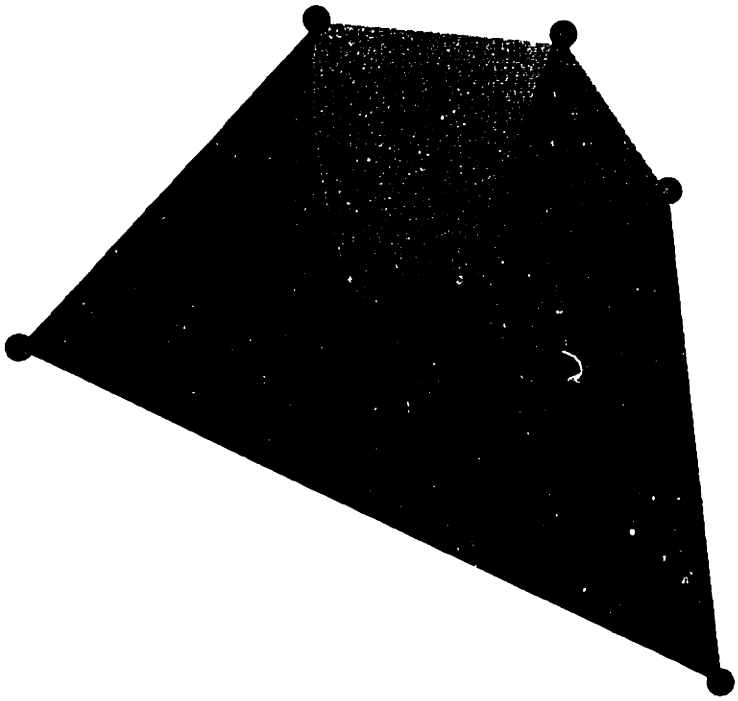


(a)

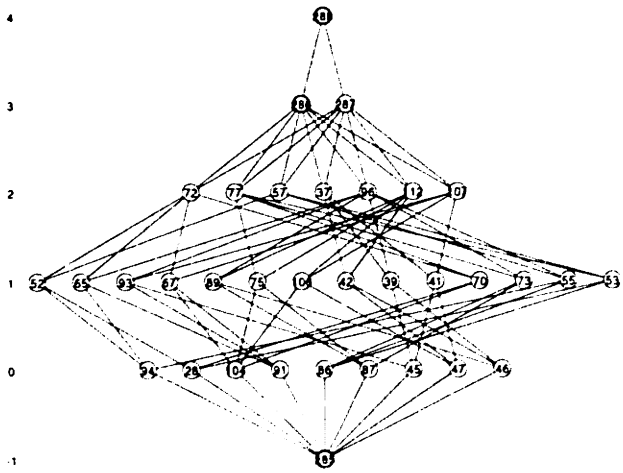


(b)

Figure 8-22: One of shell of the difference of the tetrahedron in Figure 8-17 from the cube in Figure 8-16, and its corresponding data structure.

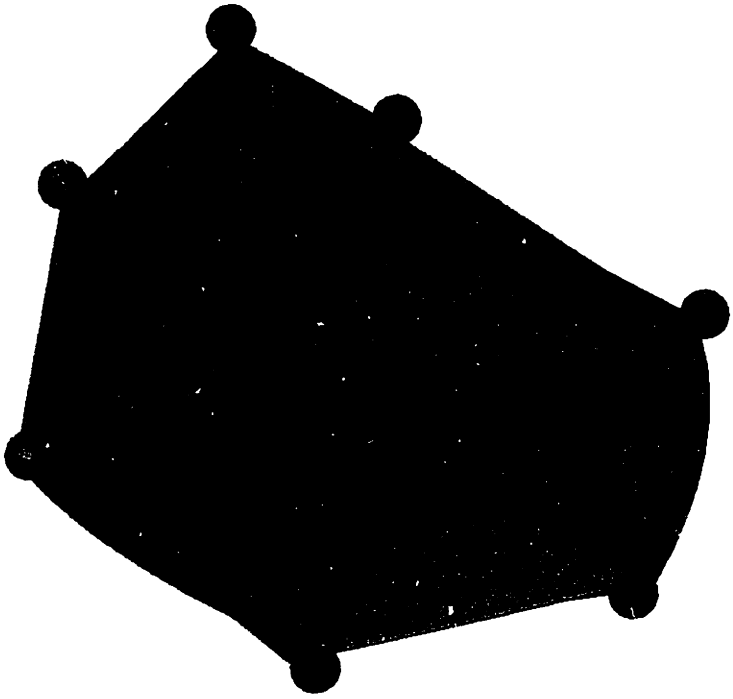


(a)

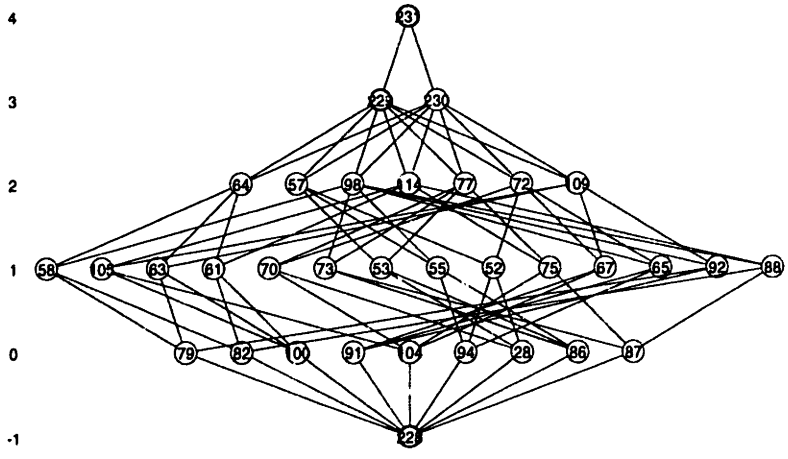


(b)

Figure 8-23: Another shell of the difference of the tetrahedron in Figure 8-17 from the cube in Figure 8-16 and its corresponding data structure.



(a)



(b)

Figure 8-24: Intersection of the cube in Figure 8-16 and the tetrahedron in Figure 8-17 and its data structure.

Figure 8-24(a) shows the intersection of the tetrahedron and the cube and (b) its corresponding data structure.

The dimensionalities of the nodes of the data structures in the above figures are shown in their left sides.

Chapter 9

Conclusions and Recommendations

9.1 Summary

Overall, this work has provided a theoretical and algorithmic framework for developing a robust non-manifold geometric modeler for curved objects.

We think that the lack of robust geometrical representations and robust geometrical computations and methods are the main causes for the robustness problems faced in the existing solid modelers. We explore interval polynomial objects to achieve robust geometrical representations, and develop new algorithms for robust geometrical computations.

We develop algorithms for 2D and 3D Boolean operations to build an experimental solid modeler. The resulting modeler is called *Interval Solid Modeler* (*ISM*). As a whole, we found that our algorithms are robust and more effective than existing methods.

The main contributions and future research are summarized as follows:

9.2 Contributions

The contributions of this thesis are summarized as follows:

1. **Robust geometrical representations:**
   - We developed robust geometrical representations based on interval polynomial objects.
2. **Robust solver** for non-linear polynomial equation systems:
   - We developed a new criterion to verify root containment in leftover intervals by the Bézier clipping and IPP algorithm.
   - We developed a robust solver for *unbalanced* non-linear polynomial systems. It is used to solve ill-conditioned and general intersection problems.

3. **Robust computations for geometrical intersections**

   - We developed algorithms and provide theories, such as *End Point Theorem*, for solving ill-conditioned intersection problems, including tangential contact point, tangential intersection curve and overlap of two objects. These are among the main causes for the lack of robustness in existing solid modeling systems.
   - We introduced a general unified algorithm for geometric intersections which reduces to overconstrained or balanced nonlinear polynomial systems.
   - We solved for the collinear normal points of two surfaces robustly, to eliminate intersection loops between patches.
   - We developed an interval ordinary differential equation (ODE) solver, which permits enveloping of intersection curves of patches.

4. **B-Rep Boolean operations**

   - We developed an n-D non-manifold data structure for interval geometrical objects: interval points, interval polynomial curves, and interval polynomial surfaces.
   - We developed a point classification algorithm for non-manifold objects.
   - We developed and proved an algorithm for determining a point on the boundary of simple planar closed curves. The theorem is used for the point classification algorithm.
   - We developed algorithms for 2D Boolean operations.
   - We developed algorithms for 3D Boolean operations.

5. **Rendering trimmed surfaces**

   - We invented and proved the *Extreme Orientation Theorem* for determining the orientation of simple planar closed curves, which is used to display trimmed patches for curved geometric modelers.

9.3 Future Research

Future research includes:

1. How to further speed up rounded interval arithmetic.
2. How to extend interval polynomial parametric objects to interval splines, like interval NURBS.
3. How to more efficiently detect and represent overlapping of interval polynomial surfaces.

4. How to solve general intersection problems more efficiently. For example, how to test the occurrences of ill-conditioned intersection without actually solving the systems.

5. How to intersect trimmed surfaces even more efficiently (Boolean operations often result in trimmed surfaces).

6. How to efficiently implement interval B-rep solid modelers for practical use. The concept of interval B-rep solid modeler is easy, but its detailed implementation is a difficult task.

7. How to build a user-friendly interface for interval solid modelers for curved objects based on Boolean operations; it is an interesting and profitable line of work for both academia and industry. Maybe Boolean operations should be hidden from users as low-level operations and they are only invoked by high-level operations.

8. How to incorporate interval solid modelers into an intelligent system for both design and manufacturing. For example, the generation of finite element meshes and assembly plans from interval solid modelers.

9. How to choose between an (interval) marching method and IPP solver for surface intersection. To trace intersection curves between surfaces, marching method is commonly used for its simplicity and speed. IPP solver is slower than the marching method in general. It is because marching method uses the first derivatives of the intersection curve; whereas IPP solver finds each root interval individually. However, in our experiments, we notice that the marching method is more efficient than IPP solver, only when loose tolerances are used, (say $\epsilon = 10^{-2}$ or $\epsilon = 10^{-3}$). Otherwise, an interval marching method will generate a lot of points along the intersection curves. Consequently, it appears to take longer time to trace. Hence, the advantage of efficiency for marching method is reduced. Furthermore, those points generated by marching methods do not envelop the solution set, whereas IPP solver generates a solution set to envelop the actual intersection set. To work on a robust solution set is more advantageous than a set of sample points. Therefore, more research needs to be done to make the IPP solver more favorable than the marching method.

10. How to simplify/reduce the representations of intersection curves to make them less memory-intensive.

11. How to develop theory and algorithms for processing data structures of objects with geometric uncertainty at various levels of resolution.

Appendix A

Orientation of a Smooth Simple Planar Closed Curve

A.1 Introduction

To render trimmed surfaces, we have to distinguish trimmed parts from untrimmed parts. The orientations of trimming curves in the parameter space are usually used for this purpose. For example, the IGES standard and the Open Inventor (a graphical software of SGI) need to know these orientations to render trimmed surfaces.

The commonly known method for determining the orientation of a smooth simple [1] planar closed curve is through the integration of the closed curve, as stated in [12] and used in [26], as follows:

Algorithm A.1 *Given a smooth simple planar closed (SSPC) curve U, pick a point* **p** *inside the curve, then integrate the angle around the point* **p** *along curve U. If the integration is non-zero and positive, then the orientation is counter clockwise (CCW); otherwise the integration is negative and the the orientation is clockwise (CW).*

Instead of using *integration* of curves, we use *derivatives* of curves to develop another algorithm to decide the orientation of smooth simple planar closed (SSPC) curves. We present the underlying theorem, referred to as **Extreme Orientation Theorem**, in section A.3. We also prove it with the help of differential geometry and the point classification theorem for planar curves [26].

A.2 Orientation of a SSPC Curve

In this section, we discuss the orientation for SSPC curves, since the trimming loops in the parameter plane usually require that there be no self-intersection for the trimming loops.

[1] The term "simple" curve means the curve does not intersect itself

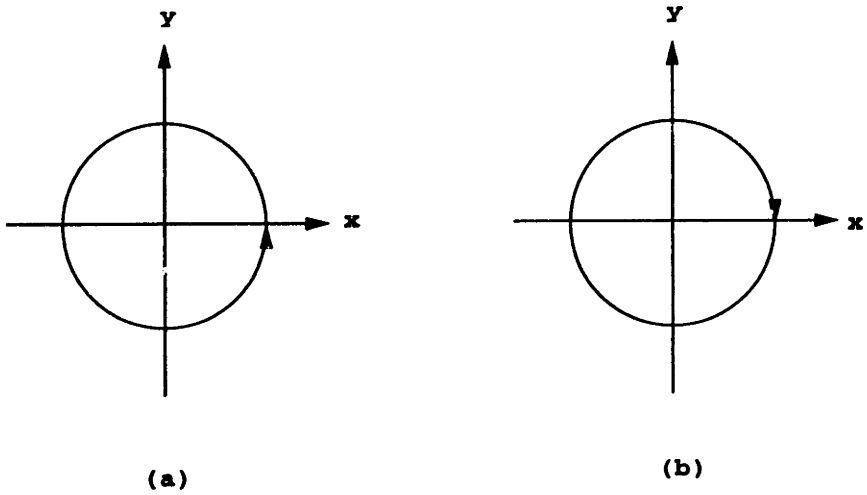


Figure A-1: Two orientations of a circle: (a) CCW orientation and (b) CW orientation.

We assume that the closed trimming curves are smooth; however, this assumption can be relaxed later.

A SSPC curve is topologically equivalent to a circle. Therefore, we use the unit circle centered at the origin with radius 1 as the template for our discussion. There are only two orientations, i.e., counterclockwise and clockwise for a circle, which are illustrated in Figure A-1.

Let us define the unit disk D in the underlying space $\mathbf{R}^2$ by the following set:

$$D = \{p \mid p \in \mathbf{R}^2, \| p \| \leq 1\} \tag{A.1}$$

Let $Int(D)$ be the interior of D:

$$Int(D) = \{p \mid p \in D, \| p \| < 1\} \tag{A.2}$$

Let C be the unit circle. It is obvious that C is the boundary of D. Let D' be the complement of D:

$$D' = \{p \mid p \notin D, p \in \mathbf{R}^2\} \tag{A.3}$$

Without loss of generality, in the remaining of this section, we mainly consider a circle with CCW orientation. The discussions for the circle with CCW can be similarly applied to the circle with CW orientation.

Note that, with the CCW orientation, the derivatives of the upper half of the circle are all toward the left (which implies that x-coordinates decrease along the orientation), whereas at the upper half of the circle with CW orientation the derivatives are all toward the right (which implies that x-coordinates increase), see Figure A-1 for illustration. At the bottom half of the circle with CCW orientation, the derivatives are all toward the right, whereas at the bottom half of the circle with CW orientation, the derivatives are all toward

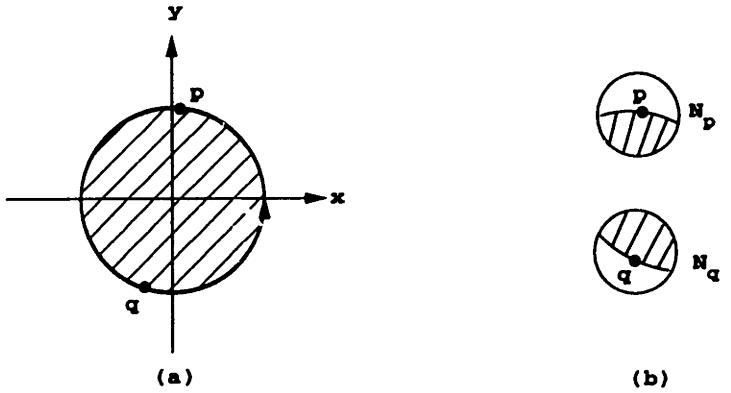


Figure A-2: (a) points **p** and **q** on t ' unit circle C; (b) their neighborhoods N_p and N_q.

the left.

Vice versa, we can gain insight about the circle orientation from its derivatives. Given a point **p** on the circle with CCW orientation, if the derivative of p is toward the right, the inside of the circle is *over* **p**. Otherwise if the derivative of **p** is toward the left, the inside of the circle is *under* **p**. Here, the two terms *over* **p** and *under* **p**, will be defined with the help of the following *Differential Jordan Curve Theorem* [12]. This theorem states that a SSPC curve divides the underlying plane into exactly two connected components. One is $Int(D)$. The other is D' the complement of D, and C is their common boundary.

Theorem A.1 *(Differential Jordan Curve Theorem). Let U be a smooth simple planar closed (SSPC) curve in $\mathbf{R}^2$. Then $\mathbf{R}^2 - U$ has exactly two connected components and U is their common boundary.*

The proof of Theorem A.1 can be found in [12].

Let us first define *neighborhood* of a point $\mathbf{p} \in \mathbf{R}^2$ as an open set $S \subset \mathbf{R}^2$ which contains **p**. Let δ be a positive real number, the set

$$N_{p,\delta} = \{q |\ \ q \in \mathbf{R}^2,\ \ \| p - q \| < \delta\} \tag{A.4}$$

is called the *δ-neighborhood* of **p**. From Theorem A.1, we know that for a point $\mathbf{p} \in C$ (the unit circle), there exists a neighborhood N of **p** such that N is divided into two connected components and $N \cap C$ is their common boundary, see Figure A-2.

We now define the terms *over p* and *under p* as follows:

Definition A.1 *Given an open set $S \subset \mathbf{R}^2$, and a limit point $\mathbf{p} = (x, y)$ of S, we say that* **p is directly under** *S or S* **is directly over p**, *if for any $\delta > 0$, there exists an $\epsilon > 0$ such that $(x, y + \epsilon) \in S \cap N_{p,\delta}$ and $(x, y - \epsilon) \in S' \cap N_{p,\delta}$, where S' is the complement of S. We say that* **p is directly over** *S or S* **is directly under p**, *if for any $\delta > 0$, there exists an $\epsilon > 0$ such that $(x, y - \epsilon) \in S \cap N_{p,\delta}$ and $(x, y + \epsilon) \in S' \cap N_{p,\delta}$. See Figure A-3 for illustration.*

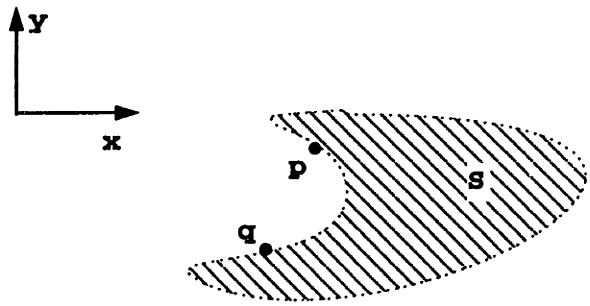


Figure A-3: **p**, a limit point of S, is directly above S and **q** is directly under S.

From Definition A.1, we can derive that for a point **p** to be *directly over* or *directly under* S, **p** must be on the boundary of S. Otherwise, **p** will not satisfy both the $\delta - \epsilon$ conditions.

Definition A.2 *Given an open set* $N \subset \mathbf{R}^2$ *which is subdivided by a curve* D *into two connected components* M_1 *and* M_2*, and a point* **p** *on the dividing curve* D*, if* **p** *is* **directly under** M_1*, we say* M_1 *is* **on the top** *of* N *with* **p**. *If* **p** *is* **directly above** M_1*, we say* M_1 *is* **on the bottom** *of* N *with* **p**. *See Figure A-4.*

We have two remarks regarding Definition A.2. First, in Definition A.2, *on the top* and *on the bottom* are defined only locally with the referenced point, see Figure A-4(b). In Figure A-4(b), M_1 is on the bottom of N with **p**, while M_1 is on the top of N with **q**. Second, point **p** with its derivative parallel to the y-axis, cannot be applied to Definition A.2, since **p** is not *directly under* or *directly above* either of the two components, see Figure A-4(c).

Further, if a point $p \in C$ (the unit circle) has its derivative toward left, then there exists a neighborhood N_p of **p**, which N_p has connected subset of C such that $N_p \cap C$ divides N_p into exactly two components, see Figure A-2. The component, $N_p \cap Int(D)$, is on the bottom of N_p with **p**, while the other component, $N_p \cap D'$ is on the top of N_p with **p**, see Figure A-2(b). On the contrary, if a point $\mathbf{q} \in C$ has its derivative toward the right, then the neighborhood of **p**, N_q which has connected subset of C has the property that $N_q \cap C$ divides N_q into exactly two components. The component, $N_q \cap Int(D)$, is on the bottom of N_q with **q**, while the other component, $N_q \cap D'$ is on the top of N_q with **p**, see Figure A-2(b).

We further observed that the above statements, with regard to the derivatives of points are not limited only to the points on C, the unit circle. These statements also hold for general SSPC curves. See Figure A-5 for an example. In Figure A-5, for those points with derivatives toward the left, the inside of the curve is always *locally directly under* them, while for those points with derivatives toward the right, the inside of the curve is always *locally directly above* them (*locally* means in the neighborhood of points). Let us summarize the observation in the following lemmas, since it will be used later.

Lemma A.1 *For a SSPC curve* $U \in \mathbf{R}^2$ *with CCW orientation, and given a point* **p** *on*

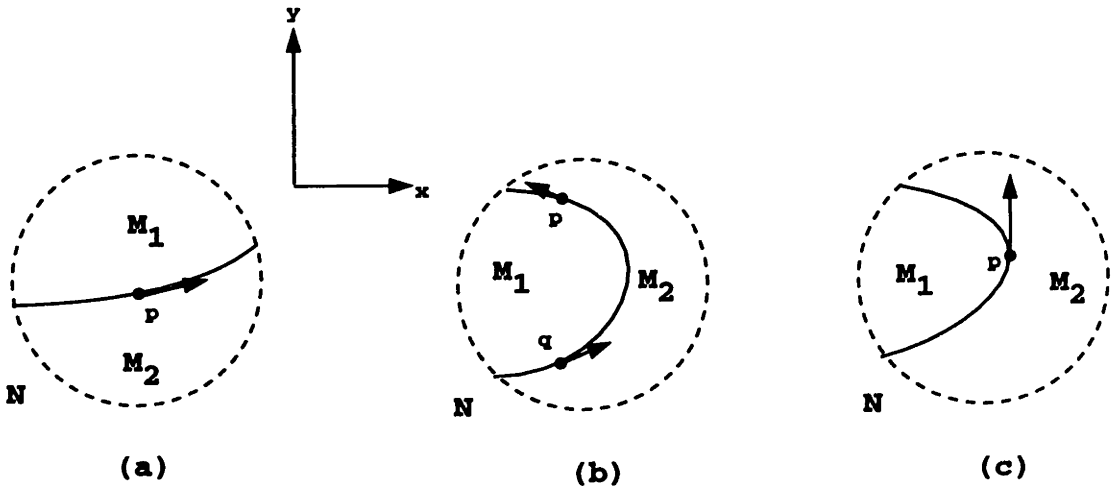


Figure A-4: (a) M_1 is on the top of N with **p**; (b) M_1 is on the bottom of N with **p**; (c) no component is on the top or bottom of N with **p**.

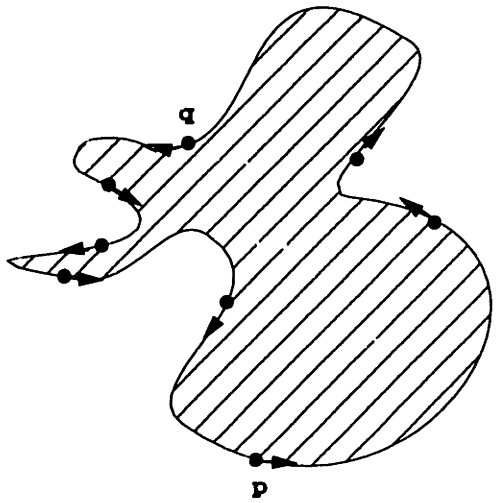


Figure A-5: For a SSPC curve, points with derivatives toward the left are always locally directly above the inside of the curve, while points with derivatives toward the right are always directly locally under the inside of the curve.

U, if **p** *has a derivative toward the right, then* **p** *is* directly under *the inside of U, (or if* **p** *has a derivative toward the left, then* **p** *is* directly above *the inside of U).*

Proof: The orientation of a closed curve can be viewed from another vantage point. If we walk along the curve following the orientation, then we will find that the inside of the curve is always on the left hand side if the orientation is CCW, or that the inside of the curve is on the right hand side if the orientation is CW.

Since U is oriented to CCW, if we stand on **p** and face the direction of derivative of **p**, the inside of U is on the left hand side. If **p** has the derivative toward the right, (see **p** in Figure A-5), the inside of U is on the left hand side. This implies that the inside of U is directly above **p** on the plane $\mathbf{R}^2$. □

Lemma A.2 *For a smooth simple planar closed (SSPC) curve* $U \in \mathbf{R}^2$ *with CW orientation, and given a point* **p** *on U, if* **p** *has a derivative toward the right, then* **p** *is* directly above *the inside of U, (or if* **p** *has a derivative toward the left, then* **p** *is* directly under *the inside of U).*

The proof of Lemma A.2 is similar to that of Lemma A.1.

We need the following planar point classification corollary to prove the *Extreme Orientation Theorem* [26]:

Corollary A.1 *Suppose U is a SSPC simple curve on the 2D plane* $\mathbf{R}^2$, **p** *is a point on* $\mathbf{R}^2$ *and* **p** *is not on the bounding curves. Let* **r** *be a ray emanating from* **p** *and not intersecting the curve U tangentially. If* **r** *intersects U at an even number of points, then* **p** *is outside U. Otherwise, it is inside U.*

A.3 Extreme Orientation Theorem

In this section, we will prove the following so-called *Extreme Orientation Theorem*, which is used to decide the orientation of a SSPC curve.

For convenience, we first define the following terminology:

Definition A.3 *Given a SSPC curve U, point* $\mathbf{p} = (x_\mathbf{p}, y_\mathbf{p}) \in U$ *is called* **maximum-y** *point, if every point on U has a y-coordinate smaller (greater) than or equal to* y_p. y_p *is called the* **maximum y** *of curve U.*

Note that a closed curve might have more than one *maximum-y* point, but it has a unique *maximum y.*

Theorem A.2 *(Extreme Orientation Theorem): Given a SSPC curve U, let* **p** *be a* maximum-y (minimum-y) *point of U. If the derivative of* **p** *is toward the left (right), then U is oriented CCW; otherwise, if the derivative of* **p** *is toward the right (left), then U is oriented CW.*

Proof: We prove this theorem contrapositively. Suppose this theorem is not true. Let $\mathbf{p} = (x_p, y_p)$, a *maximum-y* point of U. **p** has its derivative toward the left, and U is

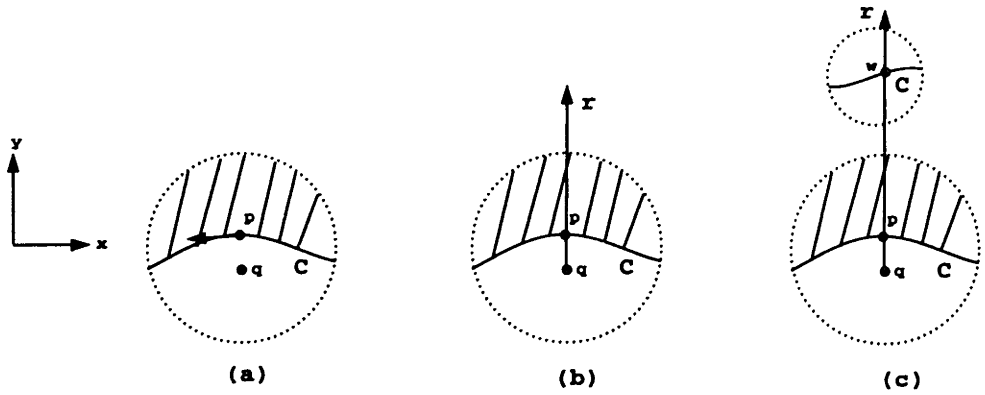


Figure A-6: Shaded areas represent inside of closed curve U.

oriented CW. That implies y_p is the *maximum y* of the curve U. According to Lemma A.2, the inside of U is *directly above* **p**, that means *directly under* **p** is the outside of U (as determined by the Differential Jordan Curve Theorem).

Thus, we can take point **q** outside the curve U and under **p** such that the curve U will not intersect the segment $\overline{\mathbf{pq}}$. Let **q** be $(x_p, y_p - \epsilon)$, where $\epsilon > 0$, see Figure A-6(a).

Then let r be the ray emanating from **q** through **p**, see Figure A-6(b). Note that the ray is toward a vertically upward position and that any point on the ray r has $x - coordinate$ as x_p. Since **q** is in the outside of U and ray r passes curve U at **p** transversally (not tangentially), by Corollary A.1, ray r will intersect U at an even number of points, i.e., at least two points, including **p**. Therefore there must be another point other than **p** at which ray r intersects with curve U. Let w be a point other than **p** at which the ray r intersects with the curve U. Let $w = (x_p, y_w)$.

y_w cannot be less than y_p, because if y_w is less than y_p, then w is either out of the ray r or falls between **p** and **q**, (see Figure A-6(c) for illustration). Therefore, $y_w > y_p$. This contradicts the fact that y_p is the *maximum y* and **p** is the *maximum-y* point of curve U. Therefore, U is oriented CCW. The case where **p** is a *minimum-y* point can be similarly be proven. □

Finally, we present our algorithm to find the orientation of a SSPC curve.

Algorithm A.2 *(Orientation Algorithm): Given a SSPC curve U, the following procedures will decide the orientation of U.*

1. *Find a point p on U with a maximum (minimum) y coordinate;*
2. *Find the derivative of U at* **p**;
3. *If the derivative is toward the left (right), U is oriented CCW, otherwise, if the derivative is toward the right (left), U is oriented CW.*

If a smooth simple planar closed (SSPC) curve is parametrically defined, we can also use the x-coordinate to decide the orientation of the curve.

Definition A.4 *Given a smooth simple planar closed curve U, point* $\mathbf{p} = (x_\mathbf{p}, y_\mathbf{p}) \in U$ *is called* **maximum-x** *(***minimum-x***) point, if every point on U has x-coordinate smaller (greater) than or equal to x_p. x_p is called the* **maximum x** *(***minimum x***) of curve U.*

Corollary A.2 *Given a SSPC curve U, let* **p** *be a* maximum-x *(*minimum-x*) point of U. If the derivative of U at* **p** *is toward up (down), then U is oriented CCW; otherwise, if the derivative of U at* **p** *is toward down (up), then U is oriented CW.*

In Theorem A.2, the condition of smoothness for curve U can be relaxed. As long as curve U is piecewise continuous, Algorithm A.2 can hold. If at the *maximum y* point, U is not smooth (i.e., not G^1 continuous), we can obtain an approximate derivative by comparing two points located in the opposite side of *maximum-y* point and both in the vicinity of *maximum-y* point. In other words, if *maximum-y* point is $U(t_0)$, where t_0 is the parameter of the curve U, then $U'(t_0) \simeq \frac{U(t_0+\epsilon)-U(t_0-\epsilon)}{2\epsilon}$. Similar substitutions can be made for the *minimum-y*, *maximum-x* and *minimum-x* points.

Appendix B

Point Classification

Point Classification for 2D Non-Manifold Objects

Point classification is an important issue in Boolean operations. After two objects are refined by their intersections, the Boolean operations have to identify which part is inside the other model (see Chapter 7). This can be done by testing one point of each refinement region to see if it is inside the other model. If non-manifold models are involved, only slight changes are required in order to permit the use of a similar method.

Point classification in two-dimensional space is posed as: *Given* a closed region C in 2D bounded by a simple curve, and a point **p**, *decide* if **p** is inside region C, outside C or on the curve.

There are two algorithms to solve this problem, both are derived from the Jordan-curve theorem [45] [80]. One is directly based on the following theorem:

Suppose C is a simple closed curve on the 2D plane P and **p** *is a point on P. If the integral of the angle with respect to* **p** *along C is 2π, then p is inside C. If the integral of the angle with respect to* **p** *along C is 0, then* **p** *is outside C.* See Figure B-1(b) for an illustration.

The proof of the above theorem can be found in [11]. It will not be presented here. However, we can give some indications as to how this should be worked out for this proof.

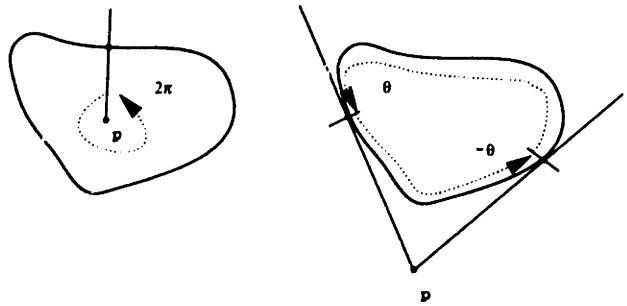


Figure B-1: The integral of the angle with respect to **p**

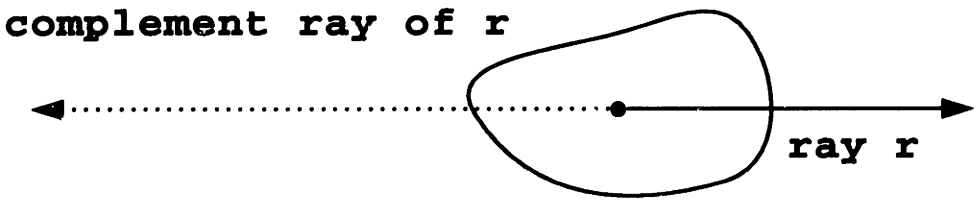


Figure B-2: An example of a ray and its complement ray.

For **p** outside C, the integral of the positive angles is equal to the integral of the negative angles (see Figure B-1), so the sum of them is zero. For **p** inside C, angles will add up to 2π or -2π. The sign depends on the orientation of the parameter of the loop. For a star-shaped loop, the integral angle will always be positive or negative (see Figure B-1), and it will add up to 2π or -2π. However, in the implementation of the point classification, we will not use this theorem directly but instead, we use the following corollary:

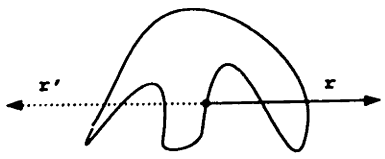
Corollary B.1 *Suppose C is a simple closed curve on the 2D plane P,* **p** *is a point on P and* **p** *is not on C, and* **r** *is a ray emanating from* **p**. *If* **r** *intersects C at an even number of points, then* **p** *is outside C, otherwise it is inside C.*

In the above corollary, a ray is defined as a half infinite line, as shown in Figure B-2. We define a complement ray of a ray r as a ray with the same starting point with r, but with the opposite direction. The proof of the above corollary is derived from the Jordan curve theorem (see [12], and also Theorem A.1). To use the above corollary, we have to solve the ray/curve intersection problem. Also we have to check the tangency condition, in which the ray tangentially intersects the bounding curve. The tangent condition occurs rarely if we choose the direction of the ray to be random. The algorithm for solving the problem of ray/curve intersection can be solved as follows. First, express the ray using the implicit polynomial: $ax + by + c = 0$ where a, b and c are constants. Second, substitute the x and y coordinates of the curve in the above polynomial equation and convert it into a polynomial in the Bernstein basis: $\sum_{i=0}^{n} c_i B_i^n(u) = 0$. Third, find the roots of the Bernstein basis polynomial (e.g. as in [36]), and discard the inappropriate roots. We shall always delete such roots, since the ray is only a half-line; recall the above polynomial equation is for the entire line which includes the ray, and hence roots for the other half-line are always included. Finally, check the number of intersection points.

Next we propose and prove a new algorithm to determine whether a point is on the boundary of a closed region. To do this, we define *parity* of two numbers as: if two integer numbers are both even or odd, they are of same parity, otherwise, they are of different parity.

Corollary B.2 *Suppose a bounding curve and a point is given. The point is on the bounding curve if and only if the ray and complement ray intersect the bounding curve with a different parity number of points as indicated in Figure B-3.*

Proof: Prove the necessity ($\Longrightarrow$). Suppose a point **p** is on the bounding curve. Any line l



r' is the complement ray of r.
r has an odd number of intersection points, while
r' has an even number of intersection points.

Figure B-3: An example of a point on the bounding curve.

through point **p** will intersect the bounding curve with even number of points by the above corollary. The intersection points of ray r on line l and r' the complement ray of r will be the same as that of l, except the point **p** will be counted twice, so the sum of the intersection points of r and r' with the bounding curve is odd, i.e., one number is odd, and the other is even. Therefore, r and r' have nonparity number of intersection points with the bounding curve.

We prove the sufficiency ($\Leftarrow$) by proving its contrapositive. If the point is not on the bounding curve, by the above theorem the point is either inside or outside the bounding curve. By the above corollary, both a ray and its complement ray of that point intersect the bounding curve with same parity number of points (if outside, the intersection number of both ray and its complement with bounding curve are even, otherwise, odd). □

Point Classification for Non-Manifold Objects

We cannot apply Corollary B.1 to non-manifold objects since such objects may include dangling edges. Take Figure B-4 for example, when using the ray test algorithm to identify if point A is inside or outside of model M. Although the number of intersection points with the edges are even, point A is inside of model M. The solution to this problem is that only the edges of manifold parts will be taken into account for the ray test. Even in the case that manifold parts are not connected, this algorithm works well. This suggests also that the data structure should provide easy access to all manifold parts of models.

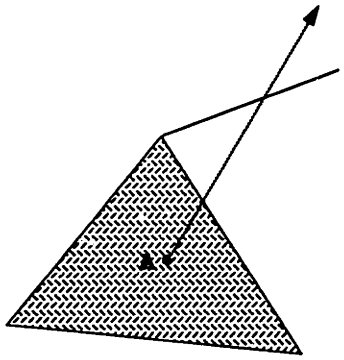


Figure B-4: Ray test for non-manifold: although point A is in inside model M, the number of the intersection of ray with the edges is 2 (even).

Appendix C

Implementation Issues

This section discusses some issues arising from our implementation. Our experience has shown that the complexity for B-rep interval solid modelers lies in the following three aspects: (1) topological consistency, (2) numerical stability , and (3) programming.

Topological Aspect

The great variety of topological entities, such as loop, shell, hole and non-manifoldness, might confuse programmers.

We suggest that programmers, in the debugging stage, would be able to visualize the topological data structure to check if topology is correct. The Euler equation can be used to check if the newly created data structure is topologically correct. For example, at the complementation of a Boolean operation, how do we know the two data structures of the incidence graph in Figure C-1(a) and (b) are correct for a manifold object. They are too complex to check the incidence relations of cells. However, using the Euler equation for a manifold, we can quickly judge which one is topologically wrong.

Numerical Aspect

Numerical errors can be the source which later cause topological violations. Take these two data structures in Figure C-1 for example. The erroneous result in Figure C-1(b) is a real case in our implementation. The error to the data structure in Figure C-1(b) is caused by comparing two 3D points to see if they are geometrically equal. The ideal result is that they should be equal, but they are not. Therefore, this caused the topological error in the data structure (Figure C-1(b)).

In our work, we improved the rounded interval solver which is numerically stable (see Chapter 3).

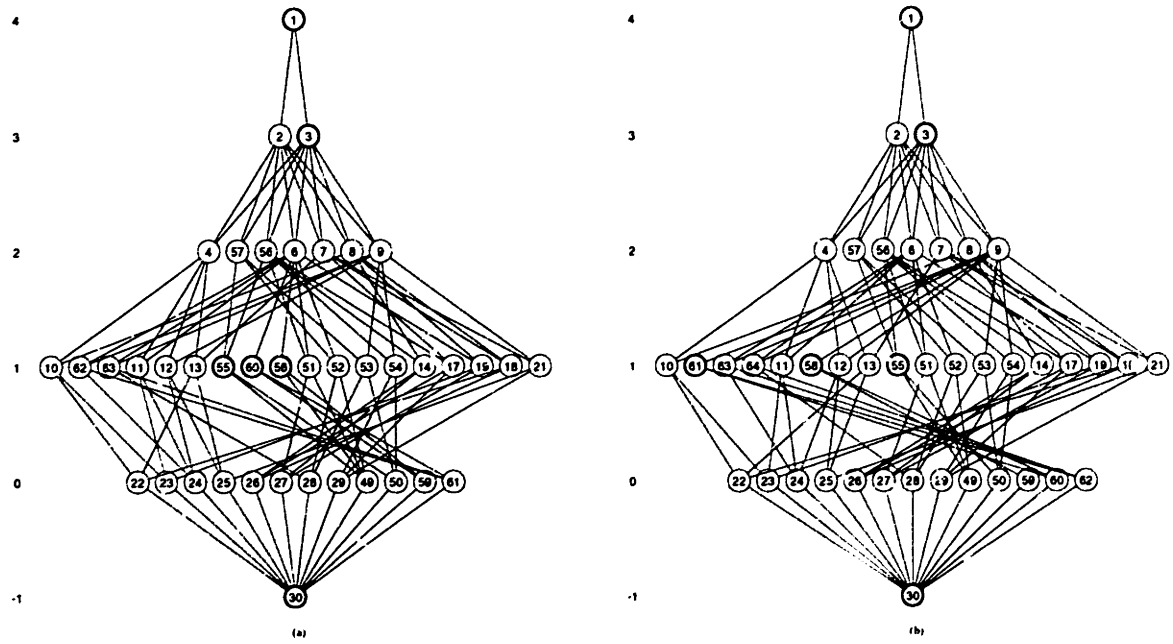


Figure C-1: Two data structures for a manifold object, which one is wrong?

Programming Aspect

The best programming is bug free. Bug-free programming is almost impossible in large and complex CAD systems. The second best programming is *easy to debug*. That means we customize our programs so that the running program can detect the running error by itself and report the errors. Otherwise, the error will cause the running program to crash at some future point far from the real source of the error. We use the following ways to detect running errors:

- use redundancy;
- set check points at routines in experimental and debugging stages;
- program from top to bottom (modular);
- differentiate low-level routines from high-level routines.
- utilize object oriented programming language (OOPL) which is a useful technique for complicated programs, such as solid modelers.

Even though there are debuggers available in some programming languages, for example **dbx** for C and C++, it is still advantageous to program in a manner in which it is convenient to debug later. First, debugging software usually just identify *where the program*

went wrong, while the programs oriented to self-debug can identify *how the program went wrong*. Secondly, the programs oriented to self-debug can stop the running routine when a running error has been caught much earlier than the debug software. These advantages of the programs oriented to self-debug can shorten the implementation processes and greatly alleviate the pain of debugging complex software systems, such as solid modelers.

In an object-oriented programming language style, the programmer deals with *objects*, instead of just a *bunch of data*. Each object's data can be abstracted and can interface with its member functions. One of the great advantages for Object-Oriented languages is *inheritance*. This allows different things to be put in an array or a list. The other advantage of Object-Oriented languages is the *dynamic binding* [1]. This is especially useful for further development without changing the existing routines.

To see an example of how programming can facility the debug later on, let us look at the two data structures in Figure C-1 again. The error message is caught in the routine of the refining process to separate an intersecting surface into two trimmed patches, in which a checker (a few lines of code to check statuses of some data) found one vertex is incident to only one edge; this is impossible for a manifold object. Therefore, the checker stopped the routine and reported the error message. Then we examined the data structure and quickly traced it back to the source of error. Without those checks, the program would probably draw the wrong Boolean results. This is very difficult to debug and would be treated as an error caused by no apparent reason.

In the process of coding, we experienced the great complication involving geometry and topological information in the Boolean operations. To ease those complications, we use the following techniques:

- module software design;
- top to bottom hierarchical software design;
- document each routine.

Documenting each routine divides each routine into three parts: the first part is the statement of the purpose of the routine; the second is the statement of the method and steps to achieve the purpose; the third part is the body of the routine with appropriate comments.

With these techniques in the later phases of this project, we encountered less unknown running errors and accelerated the debugging process considerably.

List or Array

It is sometimes difficult to choose a list or an array in the implementation of data structure. For the situation in which elements are dynamically stored, the list is a better alternative, e.g., root list, node list. For a situation in which a number of elements are known and usually will not change substantially during execution of the program, an array is a better choice, e.g., loop array (a bunch of Bézier curves) and closed regions (a bunch of loops).

Member or Friend Function

In C++ programming [72], when defining a class, a programmer is confronted with the choice of whether to define a function to be a member function or a friend function.

The significant difference between a friend and a member function is that the member function can be called only by an object class, while the friend function might be called by an object which is implicitly conversed to the object class.

Constructors, destructors, and virtual functions must be defined as member functions. An operation which will change the states of classes ought to be defined as a member function. Operators whose operands are of *lvalue* type should be defined as member functions. If implicit type conversion is used for operands of an operation, it is better to define the function as a member function.

Bibliography

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[2] L. Bardis and N. M. Patrikalakis. Topological structures for generalized boundary representations. Technical Report MITSG 94-22, Cambridge, MA: MIT Sea Grant College Program, September 1994.

[3] 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):403–415, June 1993.

[4] C. Bliek. *Computer Methods for Design Automation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1992.

[5] E. Brisson. *Representation of d Dimensional Geometric Objects.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, 1990.

[6] E. Brisson. Representing geometric structures in d dimensions: Topology and order. *Discrete and Computational Geometry*, 9:387–426, 1993.

[7] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory: Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232, 1985. Dordrecht, Holland: D. Reidel Publishing Company.

[8] J. Canny. Generalized characteristic polynomials. *Journal of Symbolic Computation*, 9:241–250, 1990.

[9] J.-M. Chen. *Integration of Parametric Geometry and Non-Manifold Topology in Geometric Modeling.* PhD thesis, Carnegie Mellon University, April 1993.

[10] K.-P. Cheng. Using plane vector fields to obtain all the intersection curves of two general surfaces. In W. Strasser and H. Seidel, editors, *Theory and Practice of Geometric Modeling*, pages 187–204, NY, 1989. Springer.

[11] J. Dieudonne. *Foundations of Modern Analysis.* Academic Press, New York, 1969.

[12] P. M. do Carmo. *Differential Geometry of Curves and Surfaces.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[13] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics*, 26(2):131–138, July 1992.

[14] W. Enger. Interval Ray Tracing - A divide and conquer strategy for realistic computer graphics. *The Visual Computer*, 9(2):91–104, November 1992.

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

[16] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press, San Diego, CA, 3rd edition, 1993.

[17] C. B. Garcia and W. I. Zangwill. Global continuation methods for finding all solutions to polynomial systems of equations in n variables. In A. V. Fiacco and K. O. Kortanek, editors, *Extremal Methods and Systems Analysis*, pages 481–497. Springer-Verlag, New York, NY, 1980.

[18] D. H. Greens and F. F. Yao. Finite-resolution computational geometry. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 143–152, October 1986.

[19] E. L. Gürsöz, Y. Choi, and F. B. Prinz. Vertex-based representation of non-manifold boundaries. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modelling for Product Engineering*, pages 107–130, The Netherlands, 1990. Elsevier Science.

[20] E. L. Gürsöz, Y. Choi, and F. B. Prinz. Boolean set operations on non-manifold representation objects. *Computer Aided Design*, 23(1):33–39, January/February 1991.

[21] G. D. Hager. Constraint solving methods and sensor-based decision making. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 1662–1667. IEEE, 1992.

[22] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[23] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *Computer*, 22(3):31–41, March 1989.

[24] M. E. Hohmeyer. A surface intersection algorithm based on loop detection. In J. Rossignac and J. Turner, editors, *Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 197–207, Austin, TX, June 1991. ACM SIGGRAPH. New York: ACM Press, 1991.

[25] M. E. Hohmeyer. *Robust and Efficient Surface Intersection for Solid Modeling.* PhD thesis, University of California, Berkeley, California, May 1992.

[26] C.-Y. Hu. *Robust Algorithms for Sculptured Shape Visualization.* Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1993.

[27] R. B. Kearfott. Interval arithmetic techniques in the computational solution of nonlinear systems of equations: Introduction, examples, and comparisons. *Lectures in Applied Mathematics*, 26:337–357, 1990.

[28] R. B. Kearfott. Decomposition of arithmetic expressions to improve the behavior of interval iteration for nonlinear systems. *Computing*, 47:169–191, 1991.

[29] R. B. Kearfott and M. Novoa. INTBIS, a portable interval Newton/bisection package (algorithm 681). *ACM Transactions on Mathematical Software*, 16(2):152–157, June 1990.

[30] D.-K. Kim. *Cones on Bezier Curves and Surfaces.* PhD thesis, The University of Michigan, Ann Arbor, MI, 1990.

[31] G. A. Kriezis. *Algorithms for Rational Spline Surface Intersections.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1990.

[32] G. A. Kriezis and N. M. Patrikalakis. Rational polynomial surface intersections. In G. A. Gabriele, editor, *Proceedings of the 17th ASME Design Automation Conference, Vol. II*, pages 43–53, Miami, September 1991. ASME, New York, 1991.

[33] 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, January 1992.

[34] C. Lee, B. Ravani, and A. T. Yang. A theory of contact for geometric continuity of parametric curves. *The Visual Computer*, 8(5-6):338–350, June 1992.

[35] T. Maekawa. *Robust Computational Methods for Shape Interrogation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1993.

[36] T. Maekawa and N. M. Patrikalakis. Computation of singularities and intersections of offsets of planar curves. *Computer Aided Geometric Design*, 10(5):407–429, October 1993.

[37] T. Maekawa and N. M. Patrikalakis. Interrogation of differential geometry properties for design and manufacture. *The Visual Computer*, 10(4):216–237, March 1994.

[38] D. Manocha. Solving polynomial systems for curve, surface and solid modeling. In J. Rossignac, J. Turner, and G. Allen, editors, *Proceedings of 2nd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 169–178, Montreal, May 1993. New York: ACM Press, 1993.

[39] D. Manocha. Solving systems of polynomial equations. *IEEE Computer Graphics and Applications*, 14(2):46–55, March 1994.

[40] R. P. Markot and R. L. Magedson. Solutions of tangential surface and curve intersections. *Computer Aided Design*, 21(7):421–429, September 1989.

[41] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988.

[42] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[43] M. E. Mortenson. *Geometric Modeling.* John Wiley and Sons, New York, 1985.

[44] S. P. Mudur and P.A. Koparkar. Interval methods for processing geometric objects. *IEEE Computer Graphics and Applications*, 4(2):7–17, February 1984.

[45] J. R. Munkres. *Elements of Algebraic Topology.* Addison-Wesley, 1984.

[46] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, 1990.

[47] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *ACM Computer Graphics*, 24(4):337–345, August 1990.

[48] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *ACM Annual Symposium on Computational Geometry*, pages 119–125, June 1987.

[49] N. M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95. January 1993.

[50] N. M. Patrikalakis, W. Cho, C.-Y. Hu, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Towards robust geometric modelers, 1994 progress report. In *Proceedings of the 1995 NSF Design and Manufacturing Grantees Conference, University of California, San Diego, California*, pages 139–140. Society of Manufacturing Engineers, Dearborn, Michigan, January 1995.

[51] N. M. Patrikalakis, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Computation of singularities for engineering design. In T. L. Kunii and Y. Shinagawa, editors, *Modern Geometric Computing for Visualization*, pages 167–191. Tokyo: Springer-Verlag, June 1992.

[52] N. M. Patrikalakis and P. V. Prakash. Surface intersections for geometric modeling. *Journal of Mechanical Design, ASME Transactions*, 112(1):100–107, March 1990.

[53] T. Poston and I. Stewart. *Catastrophe Theory and its Applications.* Pitman, San Francisco, CA, 1978.

[54] W. H. Press et al. *Numerical Recipes in C.* Cambridge University Press. 1988.

[55] A. A. G. Requicha. Progress in solid modeling and its applications. In *Proceedings of the 18th NSF Design and Manufacturing Systems Conference*, pages 761–766, Atlanta, January 1992. SME.

[56] J. R. Rossignac and M. A. O'Connor. SGC: A dimension-independent model for point sets with internal structures and incomplete boundaries. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modelling for Product Engineering*, pages 145–180, Holland, Elsevier Science Publishers, 1990.

[57] J. R. Rossignac and A. G. Requicha. Constructive non-regularized geometry. *Computer Aided Design*, 23(1):21–32, January 1991.

[58] D. Salesin, J. Stolfi, and L. Guibas. Epsilon geometry: Building robust algorithms from imprecise calculations. In *ACM Annual Symposium on Computational Geometry*, pages 208–217, 1989.

[59] D. H. Salesin. *Epsilon geometry: Building robust algorithms from imprecise computations.* PhD thesis, Stanford University, March 1991.

[60] T. W. Sederberg. Algorithm for algebraic curve intersection. *Computer Aided Design*, 21(9):547–554, November 1989.

[61] T. W. Sederberg and D. B. Buehler. Offsets of polynomial Bézier curves: Hermite approximation with error bounds. In T. Lyche and L. L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design*, volume II, pages 549–558. Academic Press, 1992.

[62] T. W. Sederberg, H. N. Christiansen, and S. Katz. An improved test for closed loops in surface intersections. *Computer Aided Design*, 21(8):505–508, October 1989.

[63] T. W. Sederberg and R. T. Farouki. Approximation by interval Bézier curves. *IEEE Computer Graphics and Applications*, 12(5):87–95, September 1992.

[64] T. W. Sederberg and R. J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5(2):161–171, July 1988.

[65] T. W. Sederberg and T. Nishita. Curve intersection using Bézier clipping. *Computer Aided Design*, 22(9):538–549, 1990.

[66] T. W. Sederberg and S. R. Parry. Comparison of three curve intersection algorithms. *Computer Aided Design*, 18(1):58–63, January 1986.

[67] E. C. Sherbrooke and N. M. Patrikalakis. Computation of the solutions of nonlinear polynomial systems. *Computer Aided Geometric Design*, 10(5):379–405, October 1993.

[68] P. Sinha, E. Klassen, and K. K. Wang. Exploiting topological and geometric properties for selective subdivision. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 39–45. New York: ACM, 1985.

[69] J. M. Snyder. Interval analysis for computer graphics. *ACM Computer Graphics*, 26(2):121–130, July 1992.

[70] A. J. Stewart. *The theory and practice of robust geometric computation, or, How to build robust solid modelers.* PhD thesis, Cornell University, 1991.

[71] G. Strang. *Linear Algebra and its Applications.* Harcourt Brace Jovanovich, San Diego, CA, 1988.

[72] B. Stroustrup. *The C++ Programming Language.* Addision-Wesley, Reading, MA, 2nd edition, 1991.

[73] K. Sugihara and M. Iri. A solid modeling system free from topological inconsistency. *Journal of Information Processing*, 12(4):380–393, 1989.

[74] L. Toth. On ray tracing parametric surfaces. *ACM Computer Graphics*, 19(3):171–179, July 1985.

[75] S. T. Tuohy, T. Maekawa, and N. M. Patrikalakis. Interrogation of geophysical maps with uncertainty for AUV micro-navigation. In *Engineering in Harmony with the Ocean, Proceedings of Oceans '93, Victoria, Canada.* IEEE Oceanic Engineering Society, October 1993.

[76] S. T. Tuohy and N. M. Patrikalakis. Representation of geophysical maps with uncertainty. In N. M. Thalmann and D. Thalmann, editors, *Communicating with Virtual Worlds, Proceedings of CG International '93*, pages 179–192. Springer, Tokyo, June 1993.

[77] M. E. Vafiadou and N. M. Patrikalakis. Interrogation of offsets of polynomial surface patches. In F. H. Post and W. Barth, editors, *Eurographics '91, Proceedings of the 12th Annual European Association for Computer Graphics Conference and Exhibition*, pages 247–259 and 538, Vienna, Austria, September 1991. Amsterdam: North-Holland.

[78] K. Weiler. The radial edge structure: A topological representation for non-manifold geometric modeling. In M. J. Wozny, H. McLaughlin, and J. Encarnacao, editors, *Geometric Modeling for CAD Applications*, pages 3–36, Elsevier Science Publishers, Holland, 1986.

[79] J. Wernecke. *The Inventor Mentor.* Addison Wesley, Reading, Massachusetts, 1994.

[80] F.-E. Wolter. *Cut Loci in Bordered and Unbordered Riemannian Manifolds.* PhD thesis, Technical University of Berlin, Department of Mathematics, December 1985.

[81] W. I. Zangwill and C. B. Garcia. *Pathways to solutions, fixed points, and equilibria.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[82] J. Zhou, E. C. Sherbrooke, and N. M. Patrikalakis. Computation of stationary points of distance functions. *Engineering with Computers*, 9(4):231–246, Winter 1993.