

# Exact Boundary Evaluation for Curved Solids

by

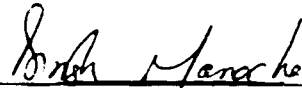
**John C. Keyser**

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

Chapel Hill

2000

Approved by:



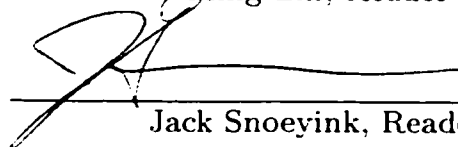
---

Dinesh Manocha, Advisor



---

Ming Lin, Reader



---

Jack Snoeyink, Reader

Copyright © 2000  
John C. Keyser  
All rights reserved

**JOHN C. KEYSER. Exact Boundary Evaluation for Curved Solids.  
(Under the direction of Dinesh Manocha.)**

**ABSTRACT**

In this dissertation, I describe an algorithm for exact boundary evaluation for curved solids. I prove the thesis that accurate boundary evaluation for low-degree curved solids can be performed efficiently using exact computation.

Specifically, I propose exact representations for surfaces, patches, curves, and points. All of the common and standard CSG primitives can be modeled exactly by these representations. I describe kernel operations that provide efficient and exact basic operations on the representations. The kernel operations include new algorithms for curve-curve intersection, curve topology, point generation, and point location. The representations and kernel operations form the basis for the exact boundary evaluation algorithm. I describe each step of the boundary evaluation algorithm in detail.

I propose speedups that increase the efficiency of the boundary evaluation algorithm while maintaining exactness. Speedups fall into several categories, including methods based on lazy evaluation, quick rejection, simplified computation, and incorporation of hardware-supported floating-point methods. I also describe the types of degeneracies that can arise in boundary evaluation and perform a preliminary analysis of methods for eliminating degeneracies.

The representations, kernel operations, boundary evaluation algorithm, and speedups have been implemented in an exact boundary evaluation system, ESOLID, that exactly converts CSG models specified by the Army Research Lab's BRL-CAD system to B-rep models. ESOLID has been applied to a model of the Bradley Fighting Vehicle (provided courtesy of the Army Research Lab). I present results and analysis of ESOLID's performance on data from that model. ESOLID evaluates the exact boundary of Bradley data at speeds less than 1-2 orders of magnitude slower than a similar but inexact boundary evaluation system. ESOLID also correctly evaluates boundaries of objects on which this other system fails.

# Acknowledgments

First of all, I want to thank my advisor, Dinesh Manocha. His advice, encouragement, prodding, and example have challenged me to grow tremendously. I am very grateful for all of the time and effort that he has devoted to me over the past several years, and I am thankful to have been his student.

I would also like to thank the other members of my committee. I thank my readers, Ming Lin and Jack Snoeyink, for the time and energy spent in reading, editing, and helping me to improve my dissertation. I also thank the other members of my committee, Pankaj Agarwal, Fred Brooks, and Steve Molnar, for all of the time and effort they have put in. I thank all of my committee for the comments and suggestions that have challenged and encouraged me in my work.

My graduate school work has been made possible through the support of several organizations. First, I would like to thank the Office of Naval Research for the fellowship that supported me during my first several years of graduate school. That program also gave me the opportunity to work one summer at the Naval Air Warfare Center for Bob Williams, whom I would like to thank for both encouraging me and introducing me to exact computation. I would also like to thank the people at the Army Research Lab for access to the BRL-CAD system, which has been important to my work. I thank the various organizations that (through my advisor) have funded this work over the years.

Perhaps the largest influence on this dissertation has come from the other students that I have had the opportunity to work with. I have found my interactions with all of my fellow UNC students beneficial, but three students in particular have contributed to this dissertation work. First, I want to thank Shankar Krishnan, who played a major role in the early development of this work, and has continued to have an active part over the past several years. I would also like to thank Tim Culver, who has had a major part in both the ideas and implementation of this work. Over the past five years I have benefited, on both an academic and personal level, from the time spent as an officemate with Shankar and then Tim, and I thank them for their friendship



and support. Also, I would like to thank Mark Foskey, who, while working on this project for a shorter time, has also helped out significantly.

For more than 20 years now, I have had the privilege of learning from an excellent group of school teachers. Their encouragement and teaching has helped me reach this point today. Although all of my teachers have played a significant role in my academic development, I want to especially thank Mr. Brown, who first got me really excited about math.

Over my lifetime, my family has been a tremendous source of encouragement and support. I thank my parents, sister, grandparents, and other relatives for all of their love and support over the years. I am especially grateful to my parents for raising me the way they did, and helping me to become the person I am today. Also, I want to thank the members of my church family for their help and encouragement.

During my time in graduate school, I have been blessed to have an extremely caring and supportive wife. I thank Michelle for all of the love, understanding, encouragement, and support she has given me in so many ways. She has brought a great deal of joy and comfort to my life, and is the reason I look forward to returning home each day.

Finally, and most importantly, I thank God. He has blessed me in more ways than I can count, most of all through Jesus. I dedicate this work to Him for “without Him nothing was made that has been made” (John 1:3b).

# Contents

<b>Acknowledgments</b>	<b>iv</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivating Problem . . . . .	1
1.2 Key Considerations . . . . .	4
1.3 Robustness . . . . .	7
1.3.1 Numerical Error . . . . .	7
1.3.2 Degeneracies . . . . .	9
1.4 Boundary Evaluation . . . . .	10
1.5 Thesis . . . . .	12
1.6 Theme . . . . .	12
1.7 New Results . . . . .	13
1.8 Overview of Chapters . . . . .	16
<b>2 Background</b>	<b>18</b>
2.1 Problem Background . . . . .	18
2.1.1 CSG Models . . . . .	18
2.1.2 Boundary Representations . . . . .	20
2.1.3 Robustness Problems . . . . .	25
2.2 Mathematical Background . . . . .	30
2.2.1 Terms and Notation . . . . .	31
2.2.2 Resultants . . . . .	31
2.2.3 Algebraic Numbers and Sturm Sequences . . . . .	33
2.3 Previous Work . . . . .	40

2.3.1	Boundary Evaluation . . . . .	40
2.3.2	Robustness . . . . .	42
<b>3</b>	<b>Representations</b>	<b>57</b>
3.1	Patches . . . . .	57
3.2	Curves . . . . .	61
3.2.1	Curve Representation . . . . .	61
3.2.2	Operations on Curves . . . . .	64
3.3	Points . . . . .	65
3.3.1	Point Representations . . . . .	65
3.3.2	Operations on Points . . . . .	66
3.4	Topology . . . . .	70
3.4.1	Additional Data . . . . .	72
3.4.2	Sufficiency of Topological Data . . . . .	72
3.5	Input Data . . . . .	77
<b>4</b>	<b>Kernel Operations</b>	<b>80</b>
4.1	Curve-Curve Intersection . . . . .	80
4.1.1	Previous Approaches . . . . .	81
4.1.2	Algorithm for Curve-Curve Intersection . . . . .	82
4.1.3	Handling Other Cases . . . . .	88
4.2	Curve Topology . . . . .	91
4.2.1	Previous Work . . . . .	93
4.2.2	Algorithm for Resolving Curve Topology . . . . .	94
4.2.3	Non-Overlapping Bounding Boxes . . . . .	102
4.3	Point Generation and Location . . . . .	102
4.3.1	Point Generation . . . . .	104
4.3.2	2D Point Location . . . . .	105
4.3.3	3D Point Location . . . . .	107
4.4	Finding an Implicit Surface Representation . . . . .	109
4.4.1	Generating Interpolating Points . . . . .	110
4.4.2	Interpolating a Plane . . . . .	110
4.4.3	Checking the Interpolated Surface . . . . .	111
4.4.4	Interpolating Higher-Degree Surfaces . . . . .	112
4.4.5	Surfaces Passing through the Origin . . . . .	113

<b>5</b>	<b>Boundary Evaluation</b>	<b>115</b>
5.1	Overview . . . . .	115
5.2	Generate Intersection Curves . . . . .	116
5.3	Resolve Topology . . . . .	119
5.4	Intersect with Trimming Boundary . . . . .	120
5.4.1	Find Intersection Points . . . . .	120
5.4.2	Point Inversion . . . . .	120
5.5	Determine Curve Correspondence . . . . .	123
5.5.1	Curves with Intersection or Inverted Points . . . . .	126
5.5.2	Curves Without Intersection or Inverted Points . . . . .	129
5.6	Clip Curves to Trimming Boundary . . . . .	130
5.7	Merge Curves . . . . .	136
5.8	Partition Patches . . . . .	138
5.8.1	Break Loops . . . . .	140
5.8.2	Compute Topology . . . . .	144
5.9	Classify Partitions . . . . .	146
5.10	Build Final Solid . . . . .	147
<b>6</b>	<b>Speedups</b>	<b>149</b>
6.1	Lazy Evaluation . . . . .	150
6.2	Quick Rejection . . . . .	151
6.2.1	Interval Arithmetic . . . . .	151
6.2.2	Bounding Boxes . . . . .	152
6.3	Simplifying Computation . . . . .	153
6.3.1	Point Equality . . . . .	153
6.3.2	Curve-curve Intersection . . . . .	153
6.3.3	Reducing Intervals . . . . .	154
6.3.4	Lower-Dimensional Computation . . . . .	154
6.4	Floating-point Evaluations . . . . .	154
6.4.1	Floating-point Filters . . . . .	155
6.4.2	Floating-point Guided Computation . . . . .	156
<b>7</b>	<b>Degeneracies</b>	<b>158</b>
7.1	Types of Degeneracies . . . . .	158
7.1.1	Input Degeneracies . . . . .	159
7.1.2	Unpredictable Degeneracies . . . . .	165

7.2	Detecting Degeneracies . . . . .	166
7.3	Eliminating Degeneracies . . . . .	170
7.3.1	Special Cases . . . . .	170
7.3.2	Symbolic Perturbation . . . . .	173
7.3.3	Numerical Perturbation . . . . .	175
<b>8</b>	<b>Implementation and Performance</b>	<b>187</b>
8.1	The MAPC Library . . . . .	187
8.1.1	Implementation Details . . . . .	188
8.1.2	Performance Results . . . . .	192
8.2	ESOLID . . . . .	198
8.2.1	Implementation Details . . . . .	200
8.2.2	Performance Results . . . . .	207
<b>9</b>	<b>Summary and Future Work</b>	<b>222</b>
9.1	Thesis . . . . .	222
9.2	Future Work . . . . .	223
9.2.1	Increasing Efficiency . . . . .	223
9.2.2	Handling Degeneracies . . . . .	224
9.2.3	Implementation . . . . .	225
9.2.4	Long-Term Directions . . . . .	226
<b>A</b>	<b>Parametric Solids</b>	<b>228</b>
A.1	Polyhedron . . . . .	229
A.2	Truncated Generalized Cone . . . . .	231
A.3	Ellipsoid . . . . .	234
A.4	Torus . . . . .	236
	<b>Bibliography</b>	<b>239</b>

# List of Tables

7.1	Possible degeneracies. . . . .	160
8.1	Timing results for root isolation. . . . .	193
8.2	Details of the difference operations illustrated in Figures 8.6 and 8.7. . . . .	211
8.3	Timing results for example j from Figure 8.7. . . . .	214
8.4	Timings for the examples from Figures 8.9 and 8.10. . . . .	217
8.5	Timing breakdown (under ESOLID) for the examples in Figures 8.9 and 8.10. . . . .	218
8.6	Timings for the examples from Figures 8.9 and 8.10, with and without the incorporation of the PRECISE library. . . . .	221

# List of Figures

1.1	An object in CSG representation. . . . .	2
1.2	An object and its boundary representation. . . . .	3
1.3	Images of the exterior and interior of a Bradley Fighting Vehicle. . . . .	6
1.4	Images of a submarine torpedo room and a pivot assembly. . . . .	7
1.5	Two views of two cylinders that barely interpenetrate . . . . .	11
1.6	Solids evaluated by ESOLID. . . . .	16
2.1	The difference between a non-regularized and a regularized Boolean operation, in 2D. . . . .	19
2.2	Three formats for a CSG representation of an object. . . . .	21
2.3	A pair of circles tessellated. . . . .	22
2.4	A curved parametric patch. . . . .	23
2.5	Three possible patch breakdowns for a cylindrical surface. . . . .	24
2.6	A trimmed parametric patch. . . . .	24
2.7	The winged-edge data structure. . . . .	26
2.8	Determining the intersection of two lines relative to a third. . . . .	27
2.9	An overlapping face degeneracy. . . . .	29
2.10	Two solids in a degenerate position. . . . .	30
2.11	Counting roots in a box using 2D Sturm sequences. . . . .	39
3.1	A trimmed patch. . . . .	58
3.2	Breaking loops. . . . .	59
3.3	Adjacent surfaces in a cylinder. . . . .	60
3.4	A single algebraic plane curve and two associated curves. . . . .	62
3.5	Subdivision of a curve. . . . .	63
3.6	Four ways of representing points. . . . .	66
3.7	A point being cut and halved. . . . .	67
3.8	Contracting and shrinking a point. . . . .	68
3.9	Separating two points. . . . .	69
3.10	Univariate root, bivariate root, and 2D point data structures. . . . .	73
3.11	The data structures for a segment and a curve. . . . .	74
3.12	The data structures for a surface and a patch. . . . .	75

4.1	Forming boxes that might contain roots. . . . .	85
4.2	A few of the possible box hit configurations. . . . .	87
4.3	A curve intersecting the corner of a box. . . . .	88
4.4	Two cases of many tangent box hits by one algebraic plane curve. . . . .	89
4.5	Two types of tangent intersections. . . . .	90
4.6	Intersections at singularities. . . . .	92
4.7	The output of Stage 1 of the curve topology algorithm. . . . .	95
4.8	Curve topology example. . . . .	99
4.9	Curve topology example continued. . . . .	100
4.10	The difficulty of classifying a point with respect to curves. . . . .	103
4.11	An example of point generation. . . . .	105
4.12	The difficulty of point location without monotonic segments. . . . .	107
5.1	A summary of the five main steps in the first stage of the boundary evaluation algorithm. . . . .	117
5.2	A summary of the four main steps in the second stage of the boundary evaluation algorithm. . . . .	118
5.3	Finding intersection points in the domain of a patch. . . . .	121
5.4	Correspondence between curves on different patches. . . . .	125
5.5	The necessity of three points for curve correspondence. . . . .	127
5.6	An example of curve clipping. . . . .	134
5.7	Another example of curve clipping. . . . .	135
5.8	Curve merging. . . . .	137
5.9	Patch partitioning. . . . .	138
5.10	Two loops broken by one patch subdivision. . . . .	140
5.11	Splitting one loop leading to many patches. . . . .	142
5.12	Adjacency across patches. . . . .	143
5.13	The continuation of an intersection curve across patches. . . . .	145
5.14	Adjacency of partitions. . . . .	146
6.1	Floating-point guided computation. . . . .	157
7.1	A borderline degeneracy. . . . .	159
7.2	Examples of degenerate input situations. . . . .	162
7.3	More examples of degenerate input situations. . . . .	163
7.4	Curve tangent to surface degeneracy as seen in the patch domain. . . . .	168



7.5	More input degeneracies as seen in the patch domain. . . . .	169
7.6	Operation leading to overlapping surfaces. . . . .	178
7.7	Random translation before operation yielding incorrect solid. . . . .	179
7.8	Different translations yielding very different output. . . . .	179
7.9	An example of translations unable to capture design intent. . . . .	180
7.10	Examples of expansion and contraction of primitives. . . . .	181
7.11	Examples where numerical expansion and contraction lead to undesirable results. . . . .	182
7.12	A small perturbation in lines resulting in a much larger perturbation of their intersection point. . . . .	183
7.13	The difference between perturbing outward and scaling. . . . .	184
7.14	Propagating extrusion information. . . . .	185
8.1	The organization of MAPC. . . . .	189
8.2	Curve topology algorithm results and timings. . . . .	196
8.3	Sorting points along a curve. . . . .	197
8.4	Arrangement of planar algebraic curves. . . . .	199
8.5	The major parts of ESOLID. . . . .	201
8.6	The results of a difference operation on pairs of primitives. . . . .	209
8.7	The results of Boolean operations on pairs of primitives. . . . .	210
8.8	Timing breakdown for examples from Figures 8.6 and 8.7. . . . .	212
8.9	BRL-CAD examples where both ESOLID and BOOLE worked. . . . .	215
8.10	BRL-CAD examples where ESOLID worked and BOOLE failed. . . . .	216
8.11	Close-up views of Boolean operations where BOOLE fails. . . . .	220
A.1	The trimming curves for a triangular patch of a polyhedron. . . . .	230
A.2	The patch breakdown for a truncated generalized cone. . . . .	232
A.3	The domains of two generalized cone patches. . . . .	234
A.4	The patch breakdown for an ellipsoid. . . . .	235
A.5	The domain of an ellipsoid patch. . . . .	236
A.6	The patch breakdown for a torus. . . . .	237
A.7	The domain of a torus patch. . . . .	238

# Chapter 1

## Introduction

### 1.1 Motivating Problem

Solid modeling systems deal with the representation, creation, and use of models of solid objects. The two primary representations used by current solid modeling systems are Constructive Solid Geometry (CSG) and Boundary Representations (B-reps).

A CSG model is a combination of basic solid objects, called *primitives*. Examples of common primitives include spheres, cones, rectangular blocks, and tori. *Boolean combinations* (union, intersection, and difference) of these primitives are used to construct more complex models. A CSG representation usually stores an object as a binary tree, where leaf nodes are primitives and interior nodes are Boolean combinations of their child nodes. An equivalent storage method for CSG models is set-theoretic expressions. There is a one-to-one correspondence between a valid set-theoretic expression and the binary-tree representation of a CSG object. A simple example of a CSG representation is shown in Figure 1.1. Section 2.1.1 provides more details about CSG representations.

A B-rep model consists of information defining the object's boundary. This includes the geometric data for the points, curves, and surfaces that constitute the boundary and the topological connectivity between these points, curves, and surfaces. Often, the geometric data is stored in the form of *parametric patches*. Patches may be flat or curved, and the set of all patches should cover the entire boundary. In most cases, it is assumed that patches do not overlap. The parameterization of each patch maps each point in the two-dimensional *patch domain* to a three-dimensional point on the object's boundary. An example of a B-rep is shown in Figure 1.2. Section 2.1.2 gives more details about B-reps.

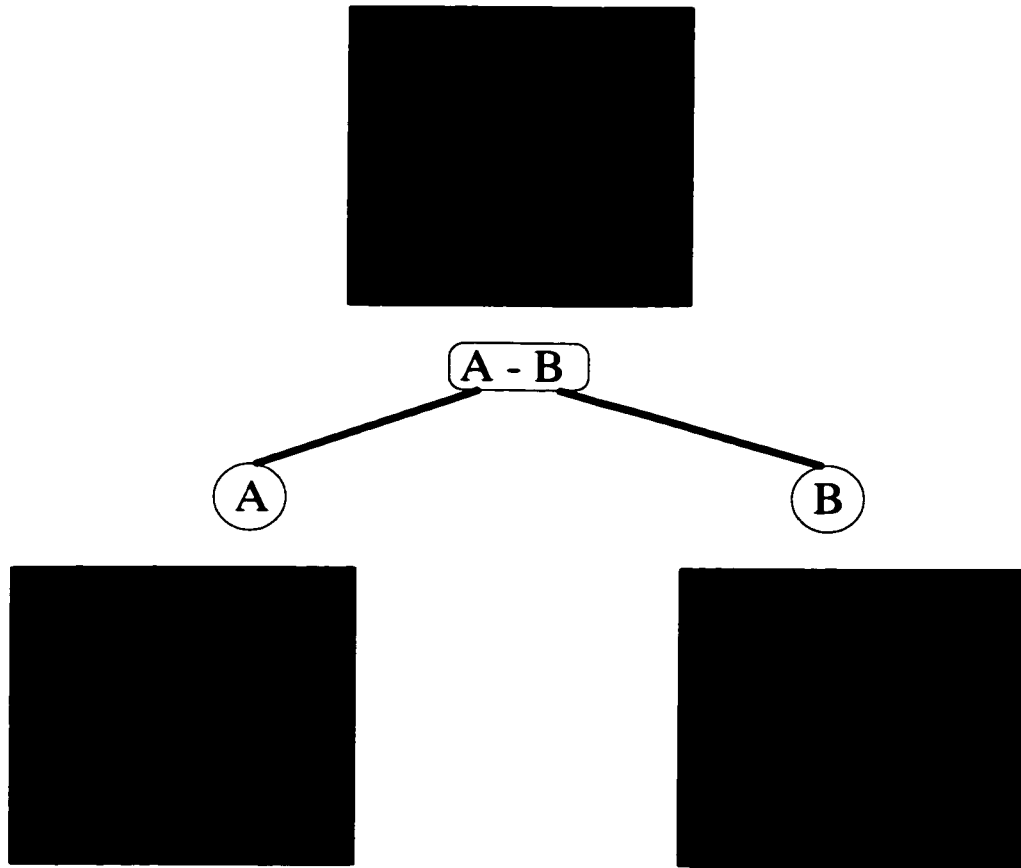


Figure 1.1: An object in CSG representation.

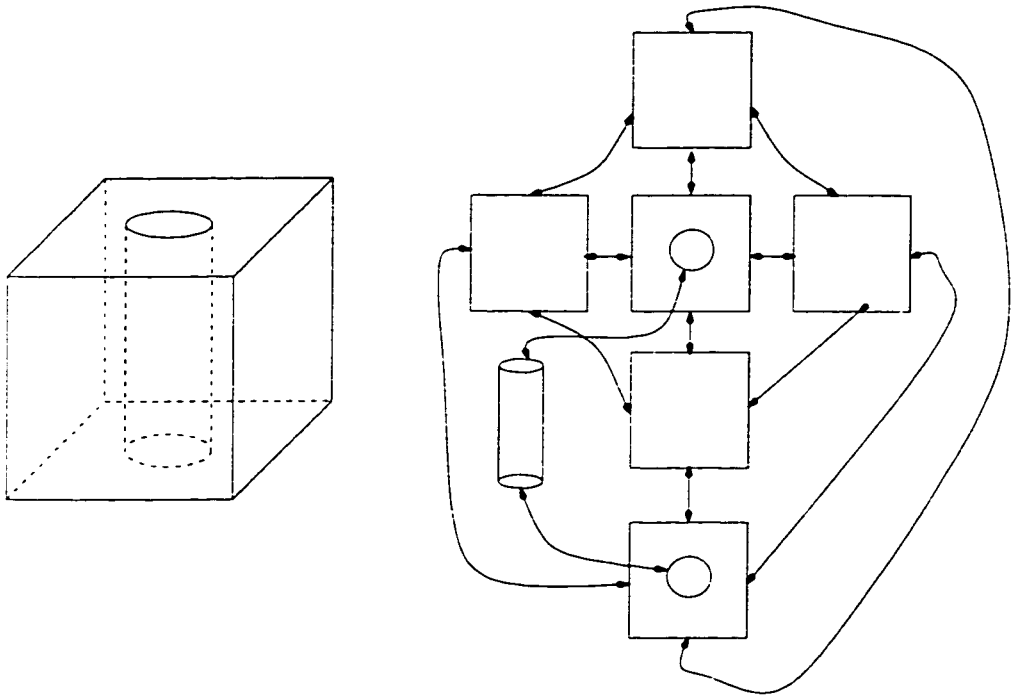


Figure 1.2: **An object and its boundary representation.**

Each representation has its own advantages. CSG provides a compact representation that directly correlates to a design or manufacturing process. Difference operations represent removing material (e.g. by drilling or milling), while union operations represent merging material (e.g. by welding). Certain operations on the model (such as ray casting or testing a point for inclusion) are easier with most CSG representations. B-reps, on the other hand, are more useful for operations such as interactive visualization and mesh generation. B-reps also allow more predictable local modifications to objects, whereas minor changes to the lower levels of a complex CSG representation can cause dramatic and unexpected changes in the overall object.

Because each representation has its own advantages, conversion from one to the other is desirable. CSG to B-rep conversion is more well-defined than B-rep to CSG. A CSG representation defines a unique boundary, and although many B-reps can represent the same boundary, the B-reps will at least be similar to each other. In a B-rep to CSG conversion, on the other hand, a B-rep will correspond to either no CSG representation or to many significantly different CSG representations, depending on the CSG primitives that are available. This dissertation addresses the CSG to B-rep conversion problem.

One method for CSG to B-rep conversion is to convert the primitive objects to B-rep (a relatively simple step) and then compute Boolean combinations of these B-rep objects to obtain a B-rep of the overall object. This computation of Boolean combinations of B-rep objects, called *boundary evaluation*, is the key operation in CSG to B-rep conversion. Note that boundary evaluation is also useful in a strictly B-rep modeling system that provides functionality analogous to that of a CSG-based solid modeling system. This dissertation addresses boundary evaluation.

## 1.2 Key Considerations

Boundary evaluation is well-studied. Some of the previous work addressing this problem is discussed in Section 2.3.1.

Since real-world solids and many CSG primitives are curved, having a curved B-rep is desirable. A *curved solid* is a solid object with a boundary that can be curved - polyhedra (i.e. flat or linear solids) are a subset of the curved solids. Curved B-reps are necessary to represent curved solids exactly, but curved surfaces, and thus curved B-reps, can be much more difficult to represent and manipulate than the planar surfaces of polyhedral B-reps. Furthermore, a curved solid can always be

approximated to within a given precision by a polyhedral B-rep. Therefore, many previous approaches to boundary evaluation have focused only on polyhedral B-reps. However, this dissertation considers curved solids, including polyhedra, represented by curved B-reps.

Because the set of all curved objects is too large for a single practical implementation, this dissertation considers only a subset, *low-degree* algebraic objects. In this context, low-degree refers to objects whose surfaces have an algebraic degree of two to four. Specifically, the surfaces examined are those found among the most common CSG primitives: polyhedra, generalized cones, ellipsoids, and tori. This set of objects, though seemingly small, has been used to create a large number of complex CSG models. As an example, consider the model of a Bradley Fighting Vehicle, which is made up of over 5000 solids (Figure 1.3). This model is composed entirely of Boolean combinations of polyhedra (53%), generalized cones (44%), ellipsoids (2%), and tori (1%). Other complex CSG models, such as the torpedo room of a submarine (Figure 1.4), are also primarily composed of low-degree solids. The only higher-degree CSG primitives in the torpedo room model come from surfaces of revolution formed by rotating an interpolated curve, as seen in the cylindrical rollers in the pivot assembly (Figure 1.4). Such primitives could be approximated by lower degree CSG objects.

Efficiency, accuracy, and robustness are desirable properties of any approach to boundary evaluation, but there are often tradeoffs among these properties. For example, an increase in accuracy or robustness usually leads to a decrease in efficiency.

*Efficiency* refers to the amount of time or computer resources (such as memory) that an algorithm uses as compared to the performance of another algorithm on the same problem. A primary source of inefficiency in boundary evaluation is accurately storing and manipulating curved surfaces. In this dissertation, efficiency is addressed in terms of time.

*Accuracy* refers to how close the computed solution is to the true solution. Numerous potential sources of inaccuracy exist, including numerical errors resulting from fixed-precision arithmetic and errors introduced by approximating geometric data. Accuracy plays an important role in robustness as well, as is seen in Section 1.3.1. The approach outlined in this dissertation uses exact representation and exact evaluation throughout, thus eliminating inaccuracies in the boundary evaluation computations.

*Robustness* is a more complex property than either efficiency or accuracy. For that reason, and because it is a distinguishing feature of the work presented in this

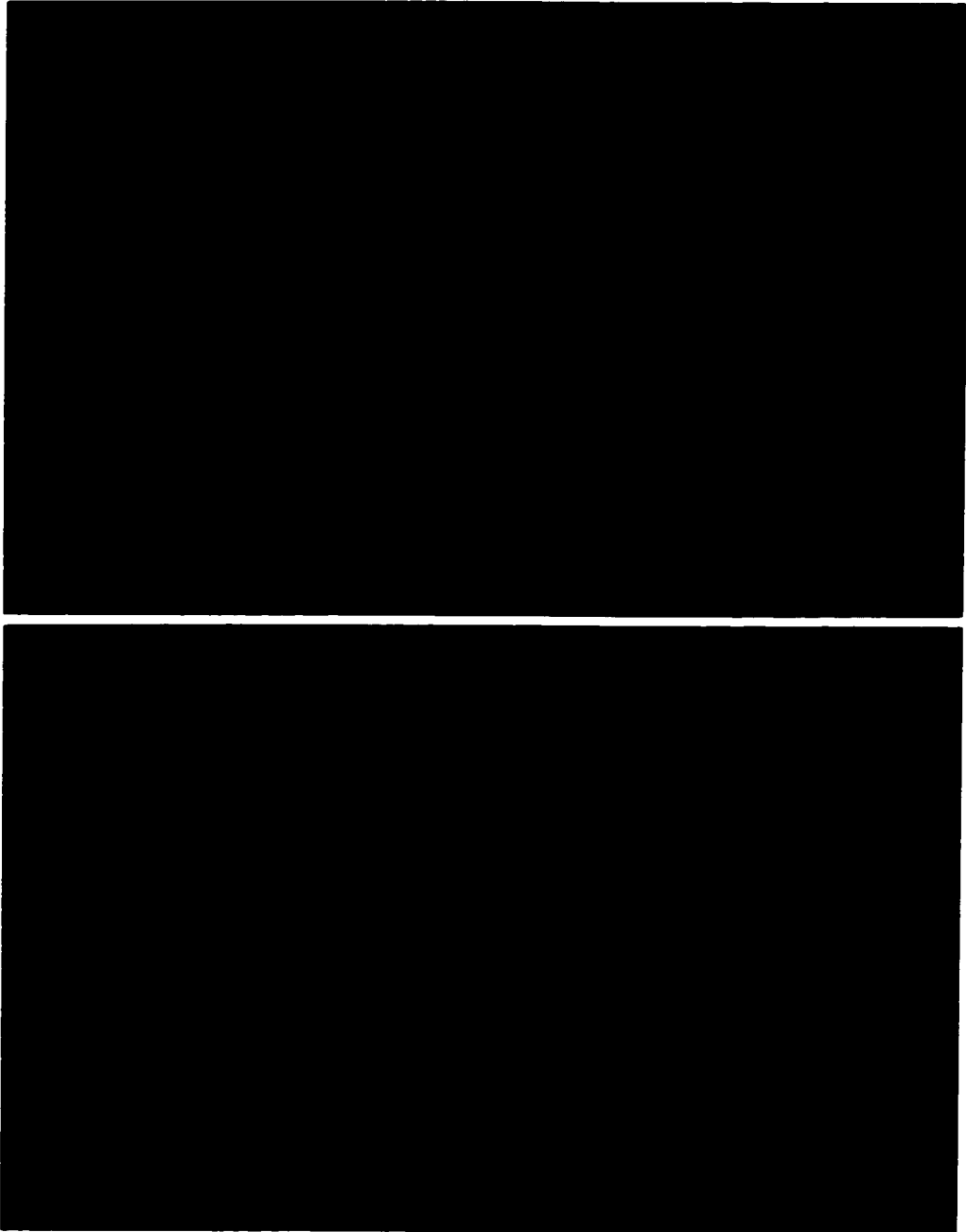


Figure 1.3: **Images of the exterior and interior of a Bradley Fighting Vehicle.** Images are ray-traced. Model courtesy of Army Research Lab.

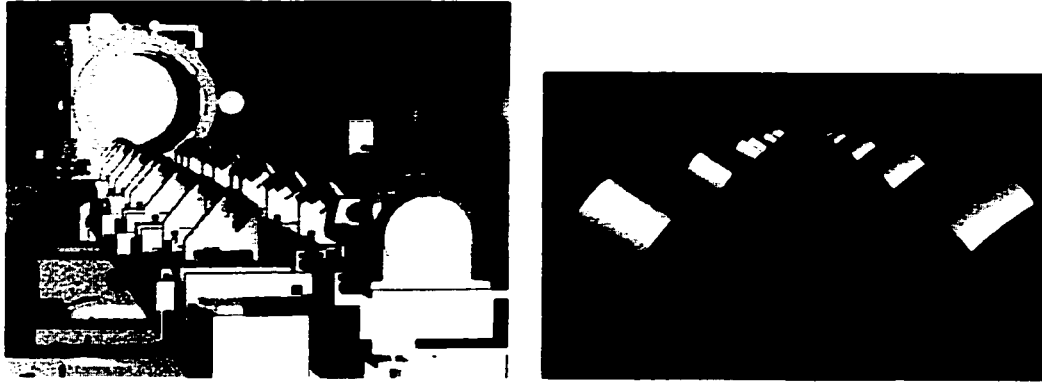


Figure 1.4: **Images of a submarine torpedo room and a pivot assembly.** Model courtesy of Electric Boat, a division of General Dynamics.

dissertation, it is examined at greater length below in Section 1.3.

## 1.3 Robustness

The *robustness problem* refers to the tendency of seemingly well-designed algorithms to fail in practice due to invalid assumptions. It is well-known in geometric modeling, solid modeling, and computational geometry. In geometric computations, there are two major sources of robustness problems: numerical errors and degeneracies.

### 1.3.1 Numerical Error

Most geometric algorithms assume the *Real RAM* model of computation [79]. This model states that all calculations are carried out over the real numbers exactly with unit-time arithmetic operations, but this is not true in practice. Since IEEE floating-point and other fixed-precision numbers are usually supported by computer hardware, they are often used to approximate real numbers, and this approximation creates roundoff error. Further computation with these approximated numbers compounds the error. If exact representations and exact computation are used, arithmetic operations are no longer unit-time.

Numerical error plays an especially important role in geometric computation, because predicates that affect the flow of the algorithm are often evaluated using numerical data. For example, some algorithms for computing the convex hull of a set of points rely on finding the point that creates the smallest angle to a given line.



Numerical inaccuracies in the computation may cause the wrong point to be chosen, which may result in the wrong convex hull being computed or the program crashing. Thus, numerical error may result in completely wrong output, rather than just slightly inaccurate output.

The most common method for dealing with numerical error is the use of tolerances, which attempts to account for error by allowing two values that are within a certain fixed amount of each other to be considered the same. Although the use of tolerances can considerably reduce the problems associated with numerical error, there are still problems with its use in boundary evaluation. For large models, it may be impossible to define a global tolerance that works for all cases, and so a number of different tolerances have to be used. Defining these tolerances and tweaking them until they work can require a large amount of user effort, and tolerances may need to be adjusted after each modification. Furthermore, the use of tolerances suffers from the inherent problem of equality no longer being an equivalence relation. This means that  $A = B$  and  $B = C$  does not imply that  $A = C$ . For example, consider the numbers  $A = 1.47$ ,  $B = 1.50$ , and  $C = 1.53$ . If a tolerance of 0.05 is used, then a program will determine that  $A = B$  and  $B = C$ , since they are only 0.03 ( $< 0.05$ ) apart, but that  $A \neq C$ , since they are 0.06 ( $> 0.05$ ) apart. Since the equivalency of an equality relationship is usually an unstated assumption in algorithms, this problem can easily lead to algorithm failure.

Several other approaches have been used with varying success to reduce or eliminate the problems associated with numerical error. Many of these approaches are reviewed in Section 2.3.2.

To fully eliminate numerical errors, some form of *exact computation* is required. In exact computation, any decision based on numerical data is guaranteed to be made as if all numerical computations are exact [102]. One way to achieve this is with exact arithmetic, where every numerical computation is carried out to whatever precision necessary to specify the number exactly. Problems can arise with exact arithmetic involving irrational numbers, however. Unlike integers, irrational numbers cannot be represented as a finite set of digits and thus cannot be stored directly on a computer.

The primary drawback to exact computation is its inefficiency. Standard computation on floating-point or other fixed-precision numbers can make direct use of computer hardware to produce an answer in a short, fixed amount of time. The output of an arithmetic operation is the same size (e.g. 64 bits in IEEE double-precision arithmetic) as the input. In contrast, exact representations of numbers require vari-

able amounts of memory for storage. Multiplying an  $m$  digit integer by an  $n$  digit integer requires  $m + n$  digits. As more computations are performed, more space is necessary. Additionally, the time required for exact-arithmetic computations on exact numbers is a function of the space those numbers require. In practical applications this additional time can be prohibitive. For example, Karasick et al. found that for a relatively simple problem, Delaunay triangulation, a naive substitution of exact computation for inexact computation yielded running times four orders of magnitude (i.e. 10,000 times) longer [56]. For applications requiring more complex calculations, such as algebraic computations in boundary evaluation, the storage space required can grow more rapidly and the slowdown become even worse. Yu presents some of the theoretical bounds for these storage requirements [105].

Fortunately, methods exist for speeding up exact arithmetic and for performing exact computation without relying solely on exact arithmetic. Some of the previous work in this area is discussed in Section 2.3.2.1. Most of that work has focused on exact computation in the linear domain. In contrast, this dissertation focuses on identifying ways to perform exact computation more efficiently on non-linear algebraic problems like those that arise in boundary evaluation.

Exact computation coupled with exact representations guarantees complete accuracy, thus eliminating robustness problems due to numerical error. This is the approach taken in this dissertation.

### 1.3.2 Degeneracies

A second assumption of many geometric algorithms is that the input data will be in *general position*. General position means that, with probability one, a minor (infinitesimal) perturbation of the input data will not change the branching decisions made in an algorithm. For example, consider an algorithm for finding the intersection between two line segments. If the endpoint of one segment lies on the other, the segments are not in general position, since slight perturbations of the segments can cause the segments to intersect transversely or not to intersect at all. In practice, data often is not in general position. This may be because of numerical error, such as two numbers close together being rounded to the same number, or because of the input itself. Data that is not in general position is called *degenerate*, and the specific degenerate situation is called a *degeneracy*. Unexpected degeneracies can cause algorithms to fail. Note that a geometric algorithm may sometimes construct and then rely on degeneracies as an intermediate step. For example, it may compute the midpoint of

two points, thus constructing a degeneracy of three collinear points. Numerical errors cause such expected degeneracies to be destroyed, creating further problems.

Further details of degeneracies are described in Section 2.1.3.2. Previous approaches for dealing with degeneracies are summarized in Section 2.3.2.2.

Since numerical error can both create and eliminate degenerate configurations, handling numerical error is important for rigorous handling of degeneracies. This dissertation, therefore, focuses on the elimination of numerical error in boundary evaluation by using exact representations and exact computation. The approach outlined assumes that the input data is in general position, although Chapter 7 discusses how degeneracies impact this approach and how they may be handled. A truly robust approach would need to handle both degeneracies and numerical error, so this work is a first step in that direction.

The elimination of numerical error through the use of efficient exact computation is the distinguishing feature of this work. Boundary evaluation of curved solids is the motivating problem.

## 1.4 Boundary Evaluation

Various processes for boundary evaluation follow a common basic approach. First, each patch of the first solid is intersected with each patch of the second, creating intersection curves in the domain of each patch. When all patches have been intersected in this way, each patch is partitioned into some number of subpatches, based on the intersection curves. A decision is then made as to which subpatches should be kept in the final solid, and which should be thrown away. Finally, the subpatches to be kept are joined to form the boundary of the result.

To see the importance of accuracy in dealing with curved solids, consider the example shown in Figure 1.5. In this example, the two cylinders, each of radius 1, interpenetrate by  $10^{-3}$ . Any error in accuracy introduced can cause the cylinders to no longer interpenetrate. An inexact representation is one source of error. Operations such as a rotation, if performed using inexact computation, are another source of error. Even when no additional error is introduced, accurate boundary evaluation (using the algorithm outlined in this dissertation) requires approximately 200 two-dimensional curve-curve intersection computations, some of which must be performed to more than 6 decimal digits of precision. More precision is required when the cylinders interpenetrate less (e.g. up to 16 decimal digits if they interpenetrate by  $10^{-18}$ ). If

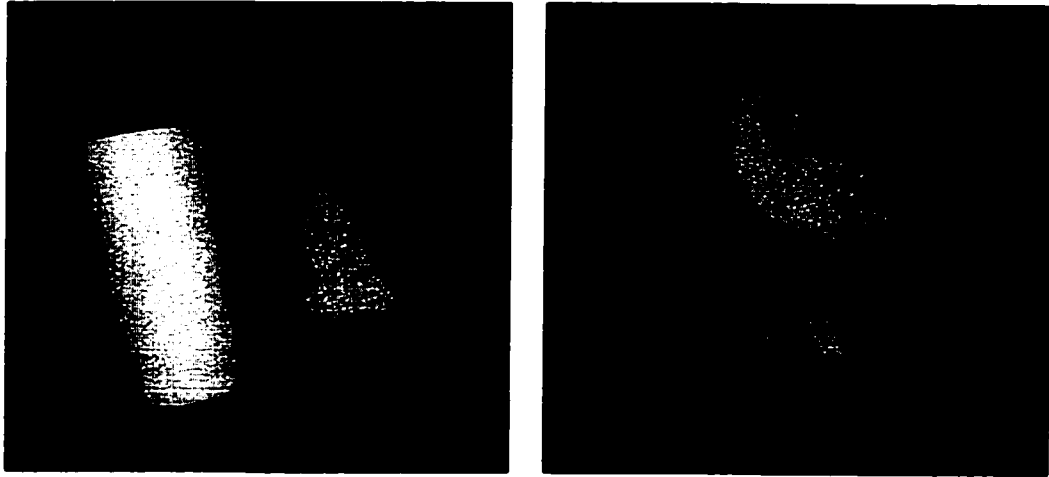


Figure 1.5: **Two views of two cylinders that barely interpenetrate.**

even one calculation is wrong, the data can become inconsistent and the boundary evaluation can fail. IEEE-standard double-precision floating-point numbers cannot represent 16 digits of precision, much less allow guaranteed computation to that precision. For this reason, computations based on fixed-precision numbers, such as the IEEE-standard numbers provided in hardware, are likely to fail on such cases. The use of exact computation and exact representations, on the other hand, ensures consistent and accurate results.

Basic approaches to boundary evaluation have been well-known for several years [12]. A number of approaches have been implemented, and the basic approach is now well understood. The first systematic study of boundary evaluation, by Requicha and Voelcker [82], lists several possible approaches. General descriptions of the process can also be found in books by Hoffmann [43] and Mäntylä [69].

Following the basic work on polyhedral boundary evaluation, development proceeded in two directions. Much of the work focused on expanding the approach to handle curved solids (e.g. [88, 16, 62]). The basic approach described above can be used regardless of whether the patches are planar or curved, but using curved surfaces makes each step more difficult. For example, intersecting pairs of planar faces gives intersection curves that are line segments. Intersecting curved surfaces results in curves that can be extremely complicated to represent and manipulate.

The other direction of development attempted to make the polyhedral operations more robust. As is described in Section 1.3, numerical errors and degeneracies

cause significant problems in implementations of boundary evaluation algorithms. Approaches have been described that focus both on exact numeric computation (e.g. [105, 35]) and on making inexact methods more robust (e.g. [45, 42, 17]).

A small amount of work has been done on accurate and robust computations with curved primitives [74, 32, 48, 49]. Current solid modeling systems that ensure accuracy (i.e. use exact computation) are restricted to polyhedral models. Most industrial solid modeling systems allow curved primitives, but to the best of my knowledge, all such systems are based on fixed-precision arithmetic. Inexact computation can be made more robust (Section 2.3.2.1), but complete accuracy and true robustness require exact methods. Section 2.3.1 discusses previous work on boundary evaluation in more detail.

## 1.5 Thesis

The thesis of this dissertation is the following:

Accurate boundary evaluation for low-degree curved solids can be performed efficiently using exact computation.

The key ideas in this statement have been discussed in the previous sections. The thesis is proved by an implementation of a boundary evaluation algorithm based on exact computation. The use of exact computation on data that (by assumption) is not degenerate yields accurate solutions. A straightforward implementation would be highly inefficient, so a major emphasis of the work is to make exact computation as efficient as possible. The goal is to achieve, on typical real-world examples, running times within one to two orders of magnitude greater than those of a similar approach that uses fixed-precision arithmetic.

## 1.6 Theme

The work presented here follows the approach of *ensuring exactness first, then increasing efficiency*. This means that one first forms an exact method and then works on identifying routines that can be made faster, while maintaining exactness. The method remains exact and is increasingly efficient as individual routines become faster.

Most standard methods for geometric problems follow a different approach of forming a fast but inexact method first, then increasing the accuracy of that method. One major reason for following this other approach is that accuracy-related robustness problems do not become apparent until an implementation is tested on real models. The algorithms used are likely to have been developed assuming the Real RAM model, so assumptions regarding numerical error may not carry through to the implementation.

The application determines which approach is appropriate. The inexact approach is unlikely to yield a fully robust implementation for all cases, but the exact approach is unlikely to be as efficient as the inexact. A decision must be made as to whether speed or robustness is of greater concern.

## 1.7 New Results

The primary result of this work is *a comprehensive description and implementation of an exact algorithm for boundary evaluation for curved solids*. There are no known previous exact algorithms for boundary evaluation for curved solids. Although the algorithm described is geared toward low-degree solids for efficiency reasons, it is also applicable to higher-degree curved solids.

Many subproblems, several of which are related to geometric problems besides boundary evaluation, have been explored during development of the exact boundary evaluation algorithm. Some of the key subproblems addressed in this work are listed below:

- **Exact representation of points, curves, and patches:** The representations used for geometric objects play a large role in the efficiency of operations performed on those objects. Furthermore, the need to maintain exact representations throughout the boundary evaluation algorithm places certain restrictions on the types of representations that are appropriate. This dissertation outlines representations for two-dimensional points and curves and for parametric patches that are exact and are well-suited for efficient use in the boundary evaluation algorithm.
- **Curve-curve intersection and curve topology:** Two of the key operations in the boundary evaluation algorithm are intersecting algebraic plane curves and resolving the topology of algebraic plane curves. These operations are of

interest in general symbolic and algebraic computation as well. This dissertation outlines techniques for both operations that are efficient and well-suited for use with the exact representations outlined previously.

- **Speeding up of exact computation using floating-point arithmetic:** Standard floating-point computations are faster than equivalent exact computations. Although the answer produced by a floating-point computation might not be exact, there still are ways of using floating-point methods to make exact computations faster. This dissertation discusses some of these methods for speeding up exact computations.
- **Library for manipulating algebraic points and curves:** A key component of the boundary evaluation algorithm is efficient and exact manipulation of algebraic points and curves in the plane. A library to do this has been developed in the process of implementing the boundary evaluation algorithm. This library has clear application outside of boundary evaluation, although boundary evaluation is the motivating factor. This dissertation describes the implementation of that library, MAPC, and presents results of its application to problems of interest both within and outside of the context of boundary evaluation.
- **Implementation and application:** The boundary evaluation algorithm has been implemented as part of a system, ESOLID, that forms exact Boolean combinations of low-degree curved solids. ESOLID has been applied to real-world data taken from the Bradley Fighting Vehicle data set. This dissertation describes the implementation of ESOLID and its performance on artificial and real-world data.
- **Curve correspondence and point inversion:** Curve correspondence and point inversion both involve transferring data from one parametric domain to another. Both operations would seem to require computation in more than two dimensions, and both are key operations in the boundary evaluation algorithm. Exact computations in higher dimensions can be extremely slow, however. This dissertation explains ways that both of these operations can be handled in the boundary evaluation algorithm by performing only lower-dimensional computations.
- **Point classification:** The classification of whether a point is within a given patch or within a solid are key operations in later stages of the boundary eval-

uation algorithm. These classifications are highly susceptible to floating-point error in inexact implementations. This dissertation discusses methods for exact point classification that are efficient and work well with the exact representations proposed.

- **Loop detection, patch splitting, and patch partitioning:** A loop occurs when the intersection curve for two solids lies entirely inside the domain of one patch on either solid. Detecting and handling such loops has traditionally been one of the more difficult operations in boundary evaluation programs. Related to loop detection is patch partitioning, or breaking up a patch into subpatches along its intersection curve. This dissertation presents methods for automatic loop detection and for efficient decomposition of loops via patch splitting, as a part of the general approach to patch partitioning. All of these are important operations in the later stages of the boundary evaluation algorithm.
- **Enumeration and description of degeneracies:** Although the boundary evaluation algorithm presented here is not intended to handle degeneracies, it is important to have a clear understanding of what degeneracies are possible and how they affect the algorithm described. This dissertation enumerates potential degeneracies, identifies their effect on the algorithm, proposes possible solutions to some degeneracies on a case-by-case basis, and briefly evaluates more general attempts to deal with degeneracies.

Some examples of the results achieved are shown in Figure 1.6. These examples are taken from the Bradley Fighting Vehicle data set. The models were expressed in CSG format, and ESOLID was used to convert them to B-reps. These models were also processed by the BOOLE system, a floating-point based modeling system developed by Shankar Krishnan [61]. BOOLE was used for timing comparisons and to verify that ESOLID could handle cases on which BOOLE failed.

For the M-16 example (Figure 1.6(a)), six Boolean operations were performed to create the object. ESOLID took 633 seconds on a 300 MHz R12000 processor to perform boundary evaluation while BOOLE took 6.7 seconds. This is approximately a two-orders of magnitude time difference, which is within the time range goal. For the track link example (Figure 1.6(b)), created by eleven Boolean operations, ESOLID took 132 seconds while BOOLE took 27.7 seconds, a difference of less than one order of magnitude. Finally, for the engine hatch example (Figure 1.6(c)), created by sixteen



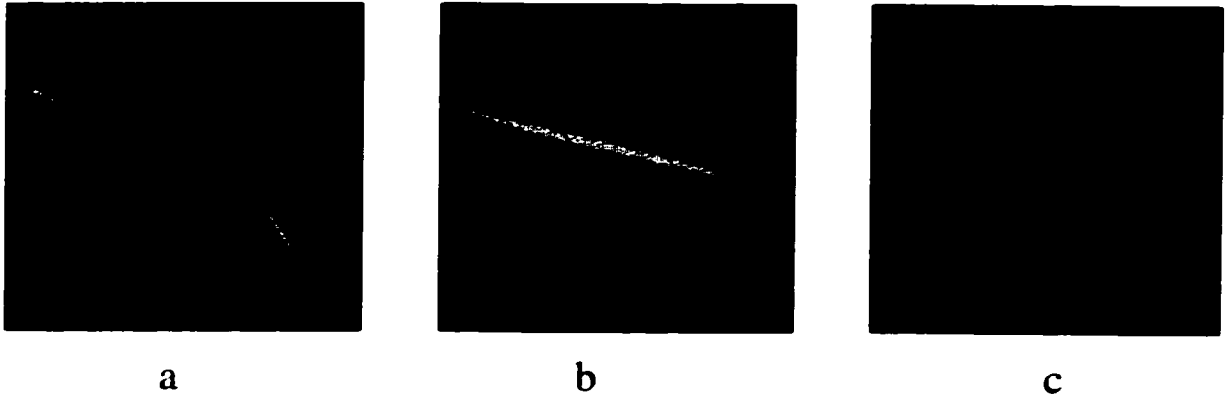


Figure 1.6: **Solids evaluated by ESOLID.** All data is from the Bradley Fighting Vehicle model. At left, an M16 rifle. At center, a link from the tread. At right, an engine access hatch.

Boolean operations. ESOLID took 55 seconds while BOOLE was unable to evaluate the boundary.

## 1.8 Overview of Chapters

The organization of the rest of this dissertation is as follows:

- Chapter 2 describes background material relevant to this work. It first provides a more detailed background of boundary evaluation and the robustness problem. Next it discusses relevant previous work, and it closes with a short discussion of mathematical concepts used in the rest of the dissertation.
- Chapter 3 describes the exact representations for points, curves, and surfaces used in the algorithm. It ends with a discussion of input data.
- Chapter 4 describes several of the basic operations in the boundary evaluation algorithm. These include routines for curve-curve intersection, curve topology resolution, and point location.
- Chapter 5 describes the boundary evaluation algorithm in detail, beginning with an overview of the entire algorithm and followed by a description of each step.
- Chapter 6 describes some of the methods used to increase the efficiency of the exact calculations.

- Chapter 7 enumerates the various types of degeneracies that can arise, and describes how these degeneracies manifest themselves in the boundary evaluation algorithm. This chapter also discusses various ways degeneracies might be handled.
- Chapter 8 describes the implementation of the MAPC library for manipulation of algebraic points and curves and the ESOLID boundary evaluation system, built on top of MAPC. Performance results on example data are also presented.
- Chapter 9 concludes the dissertation with a review of the new results and a discussion of future directions of research related to this work.

# Chapter 2

## Background

This chapter provides background information that is useful for understanding later chapters. The chapter begins by providing background information about boundary evaluation and the robustness problem. Next, an introduction to some of the primary mathematical techniques is given. Finally, a discussion of previous work done on boundary evaluation and the robustness problem is given.

### 2.1 Problem Background

In order to understand the boundary evaluation algorithm, a clear understanding of several general concepts is needed. This section provides that background.

#### 2.1.1 CSG Models

As described in Section 1.1, a CSG model is represented as a sequence of Boolean combinations on primitive solids. The usual method for storing the representation is a binary tree, with leaf nodes representing the primitives and interior nodes representing Boolean combinations of the child nodes. The overall object is at the root node. It is also possible to represent the binary operations in a directed acyclic graph format. In this way, primitives and child objects may be reused instead of copied. An example of the two CSG representations is shown at the top and middle of Figure 2.2. The binary tree representation can be directly translated into a set theoretic expression (and vice versa).

Hoffmann lists the *CSG standard primitives* as the parallelepiped (block), triangular prism, sphere, cylinder, cone, and torus [43]. A CSG based solid modeling system may use more primitives than these. For example, the BRL-CAD [26] system also

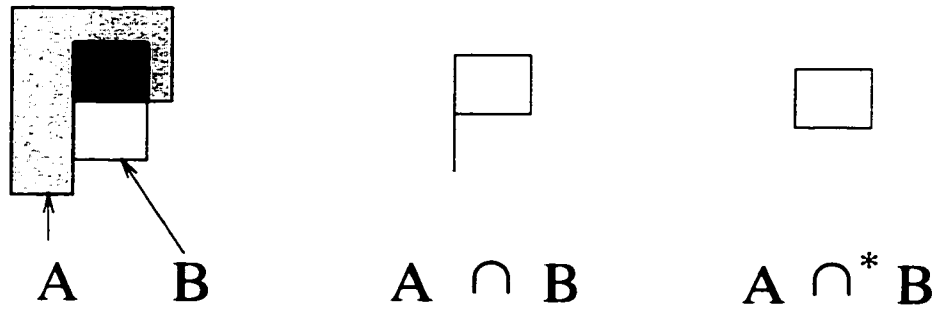


Figure 2.1: **The difference between a non-regularized and a regularized Boolean operation, in 2D.**

allows ellipsoids, generalized cones, tetrahedra, and other primitives [24]. Some of these primitives are just more general versions of the standard primitives. For example, generalized cones include cones and cylinders. The model of the Bradley Fighting Vehicle, shown in Figure 1.3, is built entirely from polyhedra, ellipsoids, generalized cones, and tori. This set of low-degree primitives includes all of the CSG standard primitives and is enough to construct a rather complex model.

The Boolean operations used for representing CSG models are union, intersection, and difference. Rather than the standard definition of these operations, CSG usually assumes a *regularized* operation. A regularized operation is defined as the *closure* of the Boolean operation on the *interior* of the two solids. Regularized operations are usually denoted by an asterisk after the operator's symbol. Thus, regularized union, intersection, and difference are written as  $\cup^*$ ,  $\cap^*$ , and  $-^*$  (or  $\setminus^*$ ) respectively. Regularized operations eliminate problems with extraneous faces, edges, and points that can arise in non-regularized operations. Figure 2.1 shows a two-dimensional example demonstrating the difference between regularized and non-regularized operations. The result of a regularized operation is guaranteed to enclose volume (or be null), although the result might not be a manifold.

Note that the difference between regularized and non-regularized operations is only significant when the input is in degenerate position. More discussion of degeneracies, including definitions of degenerate and general position, is in Section 2.1.3.2. Because the algorithm in this dissertation follows the assumption that input data is in general position, it is not concerned with the distinction between the two cases. The assumption of general position also ensures that if input solids have *manifold*

boundaries, then the output solid's boundary will be manifold. Manifold means that each point on the boundary is (locally) topologically the same as a disc. All common CSG primitives are manifold, so only manifold geometries are considered in this dissertation. Chapter 7 describes how degenerate situations, such as those that might be resolved by using regularized operations, occur, manifest themselves, and may be dealt with.

Besides Boolean combinations, a CSG tree may also incorporate transformations of objects. Conceptually, this means that an object can be designed in its own local frame of reference, and then repositioned in order to be combined (by union, intersection, or difference) with another object in a different reference frame. The most common transformations include translation, rotation, and scaling, and a general transformation matrix can be used to represent a combination of these (or other) transformations. The incorporation of transformation data into a CSG tree inserts new nodes representing the transformation, each with only one child (the object to be transformed). Since transformations can be combined (e.g. by multiplying the transformation matrices), such transformation data may be pushed down the tree to the leaf nodes if the CSG tree is stored as a full binary tree rather than a directed acyclic graph. If this approach is taken, only the initial primitives have to be transformed, which is often easier than transforming more complex objects. This may create a simpler storage scheme, at the cost of possible loss of the design history of the object. Figure 2.2 shows an example of this collapse.

### 2.1.2 Boundary Representations

A B-rep model stores a collection of points, curves, and surfaces that define the boundary of an object. Besides this geometric information, the topological information, which describes how the geometric data is connected, is stored. There are many ways of storing both the geometry and the topology of the model.

The simplest form of B-rep involves only linear geometry. Such models are composed of planar surfaces (faces), line segments (edges), and points (vertices). The faces can be represented using groups of triangles. These B-reps are useful for interactive graphics display, since most graphics hardware is designed to render triangles.

Curved solids can be represented by linear B-reps by tessellating them to a fine level. Curved surfaces can be approximated by a large number of small triangles. Such tessellations may be necessary for display purposes, but can cause problems for accurate boundary evaluation. As shown in Figure 2.3 for a 2D example, the specific

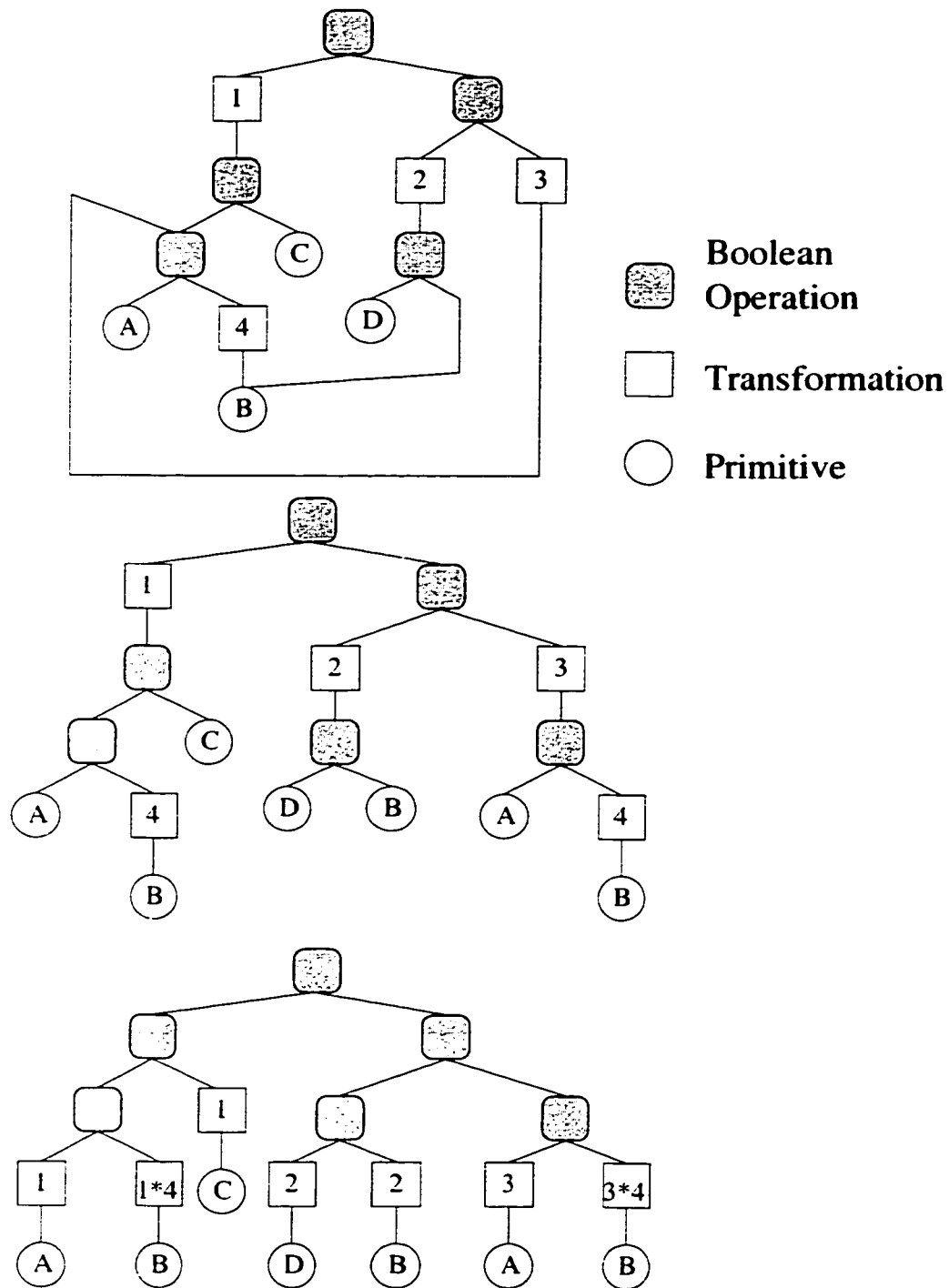


Figure 2.2: **Three formats for a CSG representation of an object.** From top to bottom, as a directed acyclic graph, as a binary tree, and as a binary tree with transformation data collapsed to the leaves.

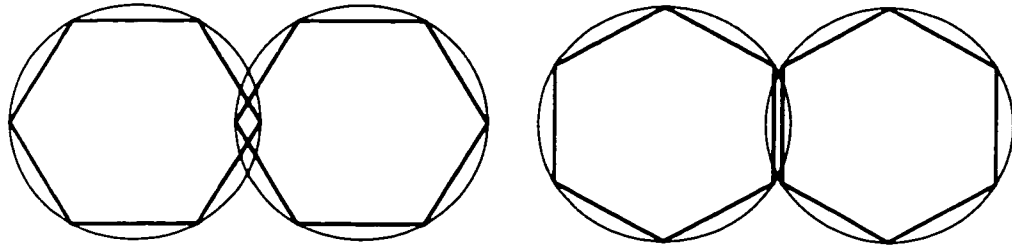


Figure 2.3: **A pair of circles tessellated.** The tessellation at left results in intersecting circles, while the one at right does not.

tessellation used can affect whether two solids are touching. Greater accuracy can be achieved by using more triangles, but this can lead to excessive storage. Furthermore, it may be difficult to determine ahead of time how much accuracy is necessary in a specific situation. Different applications may require different levels of accuracy, so many tessellations of the models may need to be stored. For example, a tessellation appropriate for graphical display and visual inspection may not be accurate enough for finite element analysis.

Because of the shortcomings of linear approximations, nonlinear representations are often used. Faces are defined by curved surfaces, and edges by curves. Note that since lines are just degree one curves, linear B-reps are just a special case of curved B-reps. The usual method for representing the curved faces is by parametric patches. A parametric patch maps a two-dimensional domain into three-dimensional space. Each point in the parametric domain projects to one point in three-dimensional space. Usually only a portion of the two-dimensional parametric domain is defined as being part of the patch. This is called the *patch domain*. A common patch domain is the region  $[0, 1] \times [0, 1]$ , to which any other rectangular domain can be easily transformed. An example of a parametric patch can be seen in Figure 2.4.

There are several ways that a surface may be decomposed into patches. Figure 2.5 shows three different ways that a cylindrical surface can be decomposed. A parametric patch may be divided into two or more subpatches simply by dividing the patch domain. For example, a patch over the region  $[0, 1] \times [0, 1]$  could be subdivided into two patches, over the regions  $[0, 0.5] \times [0, 1]$  and  $[0.5, 1] \times [0, 1]$ , each of which can then be reparameterized to the  $[0, 1] \times [0, 1]$  domain. Also, several different types of rational parametric patches can be used. The particular type of patch used and the particular way a model is divided into patches are dependent on the modeling system

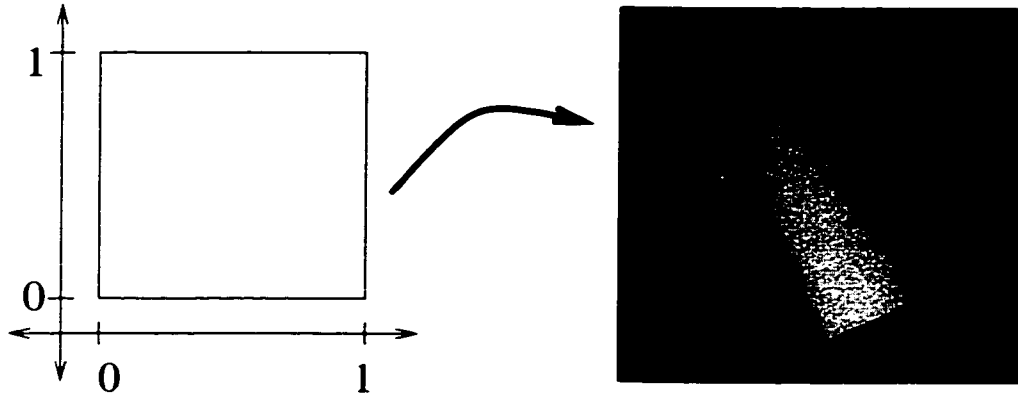


Figure 2.4: **A curved parametric patch.** The domain is shown at left, the 3D projection at right.

being used.

With a rectangular patch domain, there is little control over the edges of the patch, since they must be at a fixed parameter value. In order to represent more complex edges, *trimmed patches* are used. Trimmed patches are formed by defining a closed set of curves in the patch domain, called the *trimming curves*. Only the portion of the patch within the trimming curves is considered part of the boundary. This is called the *trimmed region*. The *boundary* of the patch is defined as the edges formed by the trimming curves in a trimmed patch or the domain boundaries in an untrimmed patch. Figure 2.6 shows an example of a trimmed parametric patch. In this dissertation, a trimmed parametric patch representation for the models is used.

Besides the geometric data, topological data must also be stored. Topological data indicates the connectivity between various parts of the model, and there are several methods for representing that information. Regardless of the method used, the importance of topological information is in allowing all connectivity information to be discovered. Topological data is considered sufficient if it is possible to reconstruct the adjacencies of all faces, edges, and vertices to each other. For example, one must be able to compute the list of all edges around a face, and all vertices adjacent (i.e. connected by a single edge) to a given vertex. This information may be stored explicitly in a topological data structure, or may need to be derived by examining the topological and geometric information. For example, which faces are adjacent to a given face may be stored in a table or graph directly, may be derived from information about the order of faces around each vertex, or may be found by determining which



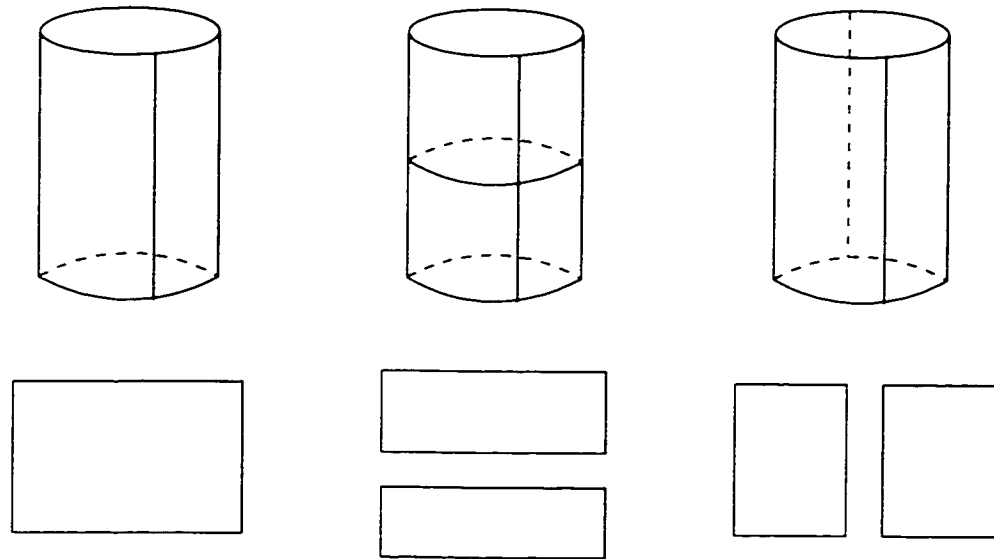


Figure 2.5: **Three possible patch breakdowns for a cylindrical surface.** At the left, as a single patch. In the middle and at the right as two patches.

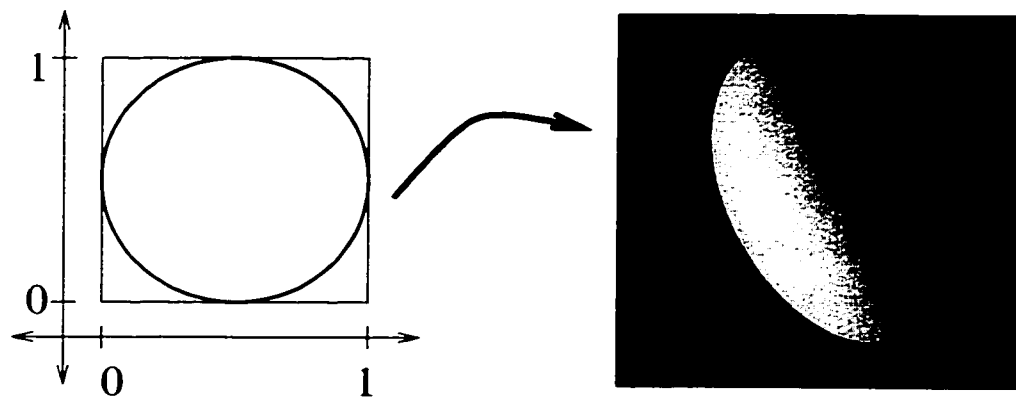


Figure 2.6: **A trimmed parametric patch.** The patch from Figure 2.4 is trimmed by the circular trimming curves shown in the domain on the left. A 3D projection is at right.

other faces, when intersected with this face, yield one of the edges. Obviously, the way the topological data is stored can influence the efficiency of operations on the model.

A popular method for representing topological data is using an edge-based data structure. These data structures have the property that all topological information is associated with the edges of the model. Any further topological information can be derived from this information at the edges. One of the well-known such structures is the winged-edge data structure developed by Baumgart [9]. Weiler [101] gives a proof of the sufficiency of the winged-edge data structure for topological information, and also introduces three other edge-based representations, outlining their sufficiency proofs. Guibas and Stolfi present an equivalent edge-based data structure, the quad-edge [38]. The winged-edge data structure will briefly be described as an example of one type of topological data structure.

The winged-edge data structure stores each edge in a directed manner. The starting and ending vertices are defined, as well as the faces on the left and right. Finally, the previous and succeeding edges are stored, as one traverses clockwise around the adjacent faces. An illustration is shown in Figure 2.7. From this information, all adjacency information is derived. For example, to find the ordered list of edges bounding a face,  $F$ , first one edge,  $E$ , bounding the face is listed. Then, the succeeding edge for  $E$  with  $F$  is followed (e.g. if  $F$  is to the left of  $E$ , then the left succeeding edge is next) and that edge is listed. The process is repeated until  $E$  is reached again. Finding the faces adjacent to a face is similar, except at each edge, the opposing face is listed instead of the edge.

### 2.1.3 Robustness Problems

The robustness problem refers to the tendency of seemingly well-designed algorithms to fail in practice due to unrealistic assumptions. For boundary evaluation, possible failures include invalid (not physically realizable) output and program crashes. There are two main sources of robustness problems, each stemming from an assumption that is often invalid in practice. These sources are numerical errors and degenerate data, and each is discussed separately.

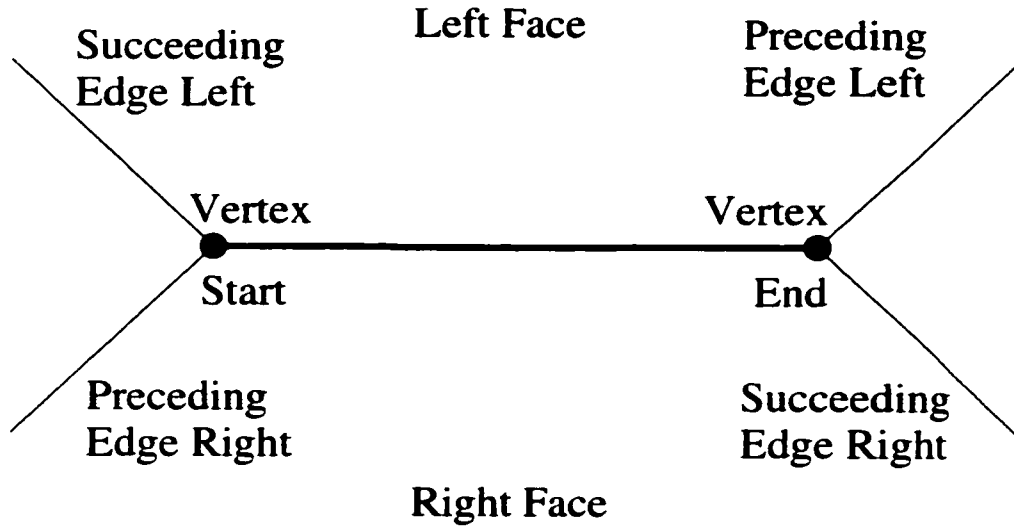


Figure 2.7: **The winged-edge data structure.** An edge of a model is shown. Stored with the edge are the starting and ending vertex, the left and right faces, and the previous and succeeding edges as you travel clockwise around the left and right faces.

### 2.1.3.1 Numerical Errors

As described in section 1.3.1, the assumption of the Real RAM model in algorithm design leads to robustness problems. Usually, because of the efficiency gained through both hardware and compiler support, implementations of algorithms use standard fixed-precision representations for numbers. These representations often require integers to be limited in absolute value and require real numbers to be approximated by fixed-precision floating-point numbers. For applications where the precision requirements are not high, these limits are fine.

In other situations, however, a higher precision is necessary in order to guarantee correctness of the code. This is particularly true for geometric algorithms. A distinguishing feature of geometric algorithms is that numerical data often determines program flow. Even slightly inaccurate numerical results can cause serious problems when those errors are large enough to change the direction a program takes. For example, consider the case of determining on which side of a line lies the intersection of two other lines. This is illustrated in Figure 2.8. A case such as this may arise when determining whether two polygons intersect. The two dashed lines in the figure may be the lines defining two edges of a polygon, with their intersection being the vertex.

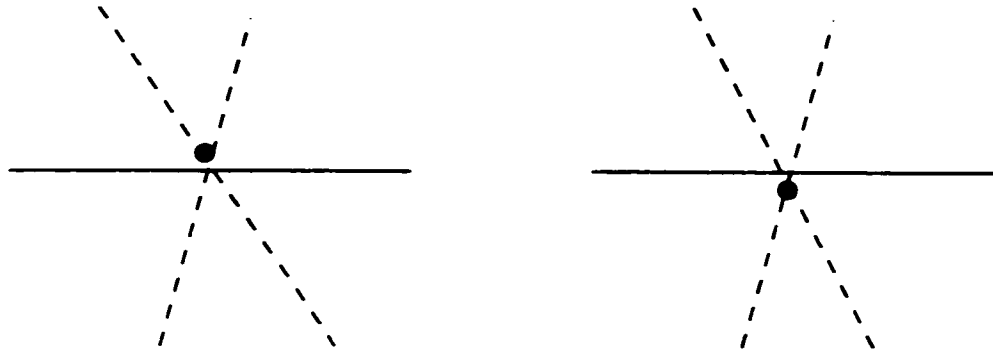


Figure 2.8: **Determining the intersection of two lines relative to a third.** Slight errors in the representations of the lines or in the calculation of the intersection point may yield the wrong answer, possibly resulting in the failure of an entire program.

If the solid line is an edge of the other polygon, then the determination of which side the vertex lies on is key in determining whether the two polygons intersect. If this determination is wrong, a program will produce incorrect output or fail completely.

There are many ways that numerical problems arise. Consider again the example illustrated in Figure 2.8. The case illustrated on the left differs from that on the right only in that one of the dashed lines has a slightly different slope. The first problem that arises is in the representation of one of the lines. Even a small amount of error, say from approximating the slope, can cause the wrong result. A second problem arises in computation of the intersection point itself. Even if the lines are represented exactly, an error in computing the intersection point can cause the lines to appear to intersect on the wrong side of the test line. A third problem arises with redundant geometric data. In the example illustrated, assume that the dashed lines are the edges of a polygon, as described earlier. Assume the polygon representation stores both the line equations for the edges and the coordinates of the intersection point. Due to roundoff or other errors, the true intersection of the lines can differ from the coordinates stored for the vertex. Thus, the geometric information is inconsistent, causing potentially serious problems.

Just as geometric information can conflict with other geometric information, geometric information can conflict with topological information. Numerical errors can be the root cause of this conflict. An example is a case where the topological information describes a valid polygon, while the geometric information (due to roundoff error) in-

dicates that there are self-intersecting sides. Such inconsistency leads to incorrect programs. Inconsistencies can be reduced by eliminating redundant information. For example, when dealing with polygons, only the line equations for the edges might be stored. Vertices are then defined implicitly as the intersections of two lines.

The problems just described show how errors may arise in just one example. More complex situations, such as those involving nonlinear structures, three-dimensional structures, or more complicated queries, magnify the potential problems. In all cases, however, the fundamental problem is due to inexact computations, and thus inexact representations.

Numerical errors can occur at several times during boundary evaluation. Boundary evaluation involves intersecting surfaces, intersecting curves, and determining where points lie relative to surfaces and curves. There are usually a large number of these operations, and numerical errors in any one computation can cause the entire process to fail. Thus, minimizing numerical errors is extremely important when implementing a boundary evaluation algorithm.

A number of methods have been used to deal with numerical errors, some of which are described in Section 2.3.2. The most direct method for dealing with these errors, and the one described in this dissertation, is using exact computation and exact representations. The drawbacks to exact computation have been that it can be extremely slow and that some exact methods (such as those for dealing with algebraic numbers) are very complicated. The algorithms presented in computer algebra books (e.g. [21, 76]) provide examples of such complexities.

### 2.1.3.2 Degeneracies

A second assumption that leads to robustness problems is that data is in general position. As mentioned in section 1.3.2, degeneracies can cause algorithms to fail.

Further distinction can be drawn between three basic types of degeneracies. First are *input degeneracies*. For example, in a convex hull problem, three of the input points might be collinear. Second are *unpredictable degeneracies* that are the results of arbitrary decisions made in the algorithm itself. For example, to intersect two curves, a program might tessellate the curves into chains of linear segments. A degeneracy between the two linear chains is unpredicable, since the algorithm created a degenerate situation from a non-degenerate one. Nothing was inherently degenerate about the problem, and a change in the tessellation would have eliminated the degeneracy. The third type, the *intentional degeneracy* is one that is intentionally



Figure 2.9: **An overlapping face degeneracy.** On the top, the solid to be removed shares a face with the solid being removed from. In a non-regularized operation, it is not clear whether the top face should or should not have a hole in it.

created by the algorithm. An example is when an algorithm forms the midpoint of two points (a degeneracy), then depends on that point being the true midpoint in later computation.

When an algorithm does not anticipate a degeneracy occurring, serious problems occur. Basic tests may fail, causing an immediate crash or causing a program to proceed in a wrong manner, eventually yielding incorrect output or a program crash. It can be difficult to anticipate all of the degenerate situations, and even then, it can be difficult to deal with them. To deal with a degeneracy, both the algorithm and the data structures might need to be modified.

A large number of degeneracies can occur during boundary evaluation. A significant number arise in the input data. Specific input degeneracies that arise are discussed in Chapter 7. There are also a number of unpredictable degeneracies that occur. For example, the particular way that the patches of a B-rep model are parameterized, or the way that a surface is broken up into patches can cause degenerate situations, such as the two solids intersecting along a patch boundary.

The nature of design can lead to some of the input degeneracies in boundary evaluation. For example, consider a degeneracy where a face of one solid overlaps the face of another solid. This is a degeneracy (since a slight modification of either surface would remove the overlap), and it can cause problems in boundary evaluation when it is assumed that two surfaces meet at a curve. Figure 2.9 shows one way that such a degeneracy can arise in a two-dimensional example.

Although a designer might be able to avoid some degenerate cases, others may be mandated by the function of the object. For example, imagine a designer creating a wheel. The designer will wish to place the axle hole directly in the center of the wheel. This creates a degeneracy (two concentric circles - one for the outer rim of the

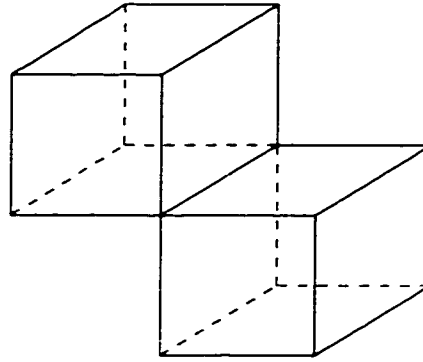


Figure 2.10: **Two solids in a degenerate position.** The union of the two cubes is a nonmanifold object.

wheel, one for the axle hole). This particular degeneracy might not cause a problem for a particular boundary evaluation algorithm, but other designer-mandated input degeneracies will.

Degeneracies can be considered apart from numerical errors as sources of robustness problems, but the two topics are closely linked. Degenerate conditions or conditions that are nearly degenerate are often the conditions most likely to lead to numerical errors. Numerical errors may eliminate certain degeneracies. For example, consider two lines that should be parallel. Due to roundoff error, their linear equations might be slightly modified, enough so that their stored representation is no longer parallel. Numerical errors can also create degenerate situations. For example, imagine two points that are close together, but distinct. Due to limited precision, the coordinates of these points may be rounded to the exact same point. Because numerical errors can both create and eliminate degeneracies, it is important to have a consistent way of addressing issues of numerical error before addressing degeneracies.

Handling degeneracies in an implementation can be difficult. Section 2.3.2 describes some of the approaches used to handle degeneracies.

## 2.2 Mathematical Background

This section provides an introduction to some of the mathematical terms and notation that are used in this dissertation. It addresses only material necessary for understanding the approach described.

## 2.2.1 Terms and Notation

In this dissertation,  $x$ ,  $y$ , and  $z$  are used to denote *three-dimensional coordinates*. A *parametric* surface maps a two-dimensional domain into three-dimensional space. For two dimensions, such as the domain of a parametric patch, the coordinates used in this dissertation are  $s$  and  $t$ . In cases where two patches are considered, one domain uses  $s$  and  $t$ , the other uses  $u$  and  $v$ .

$w$  denotes a homogenizing variable, used to describe coordinates in projective space. A parametric surface that includes a homogenizing variable (i.e. a surface described by  $X(s, t), Y(s, t), Z(s, t), W(s, t)$ ) is said to be a *rational* surface. The 3D coordinates of a point on a rational parametric surface are, in homogeneous coordinates:  $x = X(s, t), y = Y(s, t), z = Z(s, t), w = W(s, t)$ , or in 3D space:  $x = \frac{X(s,t)}{W(s,t)}, y = \frac{Y(s,t)}{W(s,t)}, z = \frac{Z(s,t)}{W(s,t)}$ . That is, the surface being described is the surface at  $w = 1$ .

The implicit form of such a surface is  $F(x, y, z, w) = 0$ , where  $F$  is a homogeneous polynomial (i.e. all terms are of the same degree).  $F$  may also be expressed in non-homogeneous form as  $F(x, y, z) = 0$ . Note that different parametric surfaces may have the same implicit form.

Unless specified otherwise, all functions in this dissertation are polynomials with exact rational number coefficients. Polynomials are expressed in the power basis. The power basis is the traditional basis used to write polynomials, where each term is expressed as the product of the variables, each raised to a power, e.g.  $x^2yz^3$ .

## 2.2.2 Resultants

The theory of elimination deals with methods for eliminating variables from a set of equations, particularly to find common solutions (i.e. common *roots*) of a set of polynomials. Elimination theory has been around since the 1800's. Salmon [87] provides a summary of much of the early work.

Elimination theory is usually concerned with determining the conditions under which a system of  $j$  homogeneous equations in  $j$  variables has a common solution. This is equivalent to looking for solutions of  $j$  non-homogeneous equations in  $j - 1$  variables. Elimination methods extend directly to problems of eliminating some (up to  $j - 1$ ) of the variables when there are  $j$  or more unknowns.

An important part of elimination theory is the *resultant*, also referred to as the *eliminant*. A resultant is defined as an expression involving the coefficients of a set





those of  $g$  in  $n$  rows. Notice that if  $f$  and  $g$  are not univariate, Sylvester's method can still be used to eliminate any one variable from the pair of equations, by treating the remaining unknowns as constants.

For the case of  $j, k = 3$ , Dixon [25] provides a method for expressing the resultant as a single determinant. Dixon's method is an extension of the Cayley-Bezout method [87]. For three equations (nonhomogeneous in two variables):

$$f(s, t) = g(s, t) = h(s, t) = 0 \quad (2.4)$$

a new polynomial is formed:

$$\delta(s, t, a, b) = \frac{1}{(s-a)(t-b)} \begin{vmatrix} f(s, t) & g(s, t) & h(s, t) \\ f(a, t) & g(a, t) & h(a, t) \\ f(a, b) & g(a, b) & h(a, b) \end{vmatrix} \quad (2.5)$$

Notice that every common root of the three equations is also a common root of  $\delta$ , for any value of  $a$  or  $b$ . The Dixon matrix,  $D$ , is simply the coefficient matrix of  $\delta$ , where the rows index the powers of  $a$  and  $b$ , and the columns the powers of  $s$  and  $t$ . Note that the entries of  $D$  can be obtained without explicitly computing  $\delta$ . Since any value of  $a$  or  $b$  can be substituted in  $\delta$ , we must have:

$$D \begin{bmatrix} 1 & t & t^2 & \dots & t^n & s & st & \dots & s^m t^n \end{bmatrix}^T = \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}^T \quad (2.6)$$

The determinant of  $D$  provides the Dixon resultant. Unfortunately, this process only works when the three polynomials are of the same degree with no zero coefficients. Otherwise, the Dixon matrix can become singular, yielding no solution. More recently, the Dixon resultant formulation has been extended to handle certain singular matrix cases, eliminating some of these problems [54].

### 2.2.3 Algebraic Numbers and Sturm Sequences

Because integers can be represented as binary numbers, they can be represented directly on a computer. If the integers are limited in size, they can usually be stored and manipulated directly by hardware. Rational numbers can also be stored exactly and easily by storing one integer for the numerator and one for the denominator. Algebraic numbers (roots of polynomials), on the other hand, do not have such a direct representation on the computer, in general. Thus, special methods must be

used in order to represent and use algebraic numbers.

### 2.2.3.1 Representing Algebraic Numbers

A common format for representing algebraic numbers is to store floating-point approximations to the number. Although the floating-point approximation may be close to the algebraic number, it is (usually) not an exact representation. As described earlier, this inexactness can lead to serious robustness problems.

One way to represent algebraic numbers exactly is to store them as the unique root of a polynomial in an interval. For example, to represent the number  $\sqrt{2}$ , one might store the polynomial  $x^2 - 2 = 0$ , along with the interval  $[0, 2]$ . Since there is only one root of  $x^2 - 2 = 0$  within that interval, the combination of the polynomial and the interval is an exact specification of a unique algebraic number. This is the representation used in this dissertation. Note that for a given algebraic number, there is not a unique representation. Any polynomial with that number as a root can be used, and any interval bounding that one number and no other root of the corresponding polynomial can be used.

Hereafter, the terms *algebraic number* and *root* are used interchangeably. The polynomial used in the representation of the algebraic number is referred to as the *root's polynomial*, and the interval is called the *root's interval*.

The representation described is well known in computer algebra literature. Further discussion of this representation is found in general books on computer algebra (e.g. [21, 76]).

### 2.2.3.2 Univariate Sturm Sequences

In order to effectively manipulate algebraic numbers in the format described, a method is needed for verifying that an interval contains one and only one root of a polynomial. The method used to do this is the *Sturm sequence*. Again, general computer algebra books [21, 76] provide more details, but an overview is given here.

For a polynomial,  $p(s)$ , (with rational coefficients) the Sturm sequence is defined to be the following sequence of polynomials:

$$\begin{aligned} p_0(s) &= p(s) \\ p_1(s) &= p'(s) \\ p_i(s) &= -\text{rem}(p_{i-2}(s), p_{i-1}(s)) \end{aligned} \tag{2.7}$$

where  $rem(a, b)$  means the remainder when  $a$  is divided by  $b$ . The final term of the sequence is the last  $p_i$  that is not identically zero. It should be noted that with the exception of the minus sign, generating a Sturm sequence is the same as a greatest common divisor computation using Euclid's method.

Once the Sturm sequence has been created, each polynomial  $p_i$  is evaluated at a specific value  $a$ . By examining the sign of each of these evaluated terms, one can count the number of *sign permanencies* in the sequence. One permanency is counted every time  $p_i(a)$  has the same sign as  $p_{i+1}(a)$ . The number of sign permanencies in the Sturm sequence of a polynomial is called the *Sturm value*. Assuming  $a$  is not a root of  $p$ , the Sturm value at  $a$  gives the number of real roots of the polynomial less than  $a$ , plus half the number of imaginary (i.e. non-real) roots. Notice that if  $a$  is a root of  $p$ , then  $p_0(a) = 0$ . The handling of zeros in the Sturm sequence determines whether roots at  $a$  contribute to the Sturm value.

As a simple example, consider the equation:

$$p(s) = (s - 1)(s - 3)(s^2 + 1) = s^4 - 4s^3 + 4s^2 - 4s + 3$$

which has two real roots (at 1 and 3), and two imaginary roots. From this, the rest of the Sturm sequence can be generated:

$$p_0 = s^4 - 4s^3 + 4s^2 - 4s + 3$$

$$p_1 = 4s^3 - 12s^2 + 8s - 4$$

$$p_2 = s^2 + s - 2$$

$$p_3 = -32s + 36$$

$$p_4 = -\frac{3}{8}$$

To compute the Sturm value at a number, substitute that number in the  $p_i$ . At  $s = 0$ , the terms of the Sturm sequence are  $(3, -4, -2, 36, -\frac{3}{8})$ , giving a Sturm value of 1. At  $s = 2$  they are  $(-5, -4, -2, 4, -\frac{3}{8})$ , giving a Sturm value of 2. At  $s = 4$ , they are  $(51, 92, 18, -92, -\frac{3}{8})$ , giving a Sturm value of 3. Notice that in each case, the Sturm value correctly gives the number of real roots less than the query value, plus half the number of imaginary roots.

Given an interval,  $[a, b]$ , it is easily verified that the interval contains only one root. By subtracting the Sturm value at  $a$  from the Sturm value at  $b$ , one has a count of the number of real roots within the interval. Sturm values can also be used to isolate individual roots. If an interval contains more than one root, one can

recursively subdivide the interval and count the number of roots in each subinterval until each interval contains at most one root. For a polynomial of degree  $n$ , with coefficients  $a_i, i = 1 \dots n$ , Davenport [21] cites a worst case time for root isolation of  $O(n^6(\log n + \log \sum a_i^2)^3)$ , with average running time of  $O(n^4)$ . Methods for efficient root isolation are discussed in Chapter 6.

### 2.2.3.3 Equality of Algebraic Numbers

A useful operation on algebraic numbers is comparison for equality. Assume that two algebraic numbers are given in the representation described earlier. Let  $f(s) = 0$  and  $g(s) = 0$  be the polynomials of the two roots. Assume for the sake of simplicity that the roots have identical intervals,  $[a, b]$ . If the intervals are not equal, they can be made equal or disjoint via a cut, analogous to that described in Section 3.3.2.2. Note that if the intervals are disjoint, the points cannot be equal. To determine whether the points are equal, find the greatest common divisor of  $f$  and  $g$ ,  $h(s) = \gcd(f, g)$ , which can be done by a method such as Euclid's algorithm (e.g. as in Davenport's book [21]). The points are equal if and only if  $h$  has a root in  $[a, b]$ , which is determined using a method such as univariate Sturm sequences.

### 2.2.3.4 Multivariate Sturm Sequences

Just as univariate Sturm sequences can be used to count the number of roots of a univariate polynomial in an interval, multivariate Sturm sequences can be used to count the number of roots of a system of  $n$  equations in  $n$  variables inside an  $n$ -dimensional box. The common example that arises in this dissertation is counting the intersections of two planar curves (i.e. two bivariate polynomials) within a 2D rectangle.

The work on multivariate Sturm sequences is relatively recent, and key work has been done by both Pedersen [78] and Milne [75]. Pedersen's work is more general, and discusses methods for counting real roots on one side of a hypersurface and within a simplex. The method presented by Milne is restricted to boxes, which makes the formulas simpler. For the boundary evaluation algorithm, boxes are sufficient, and so Milne's approach is discussed here. This section presents the method only – for the development and proofs consult Milne's paper [75].

Just as for univariate Sturm sequences, being able to count the number of roots in a box is enough to perform root isolation. If a box contains more than one root,

it can be recursively bisected and the number of roots counted in each smaller box. This is continued until the roots are all separated.

**General Method:** The general multivariate Sturm method is described here. Immediately afterward, the specialization to two dimensions is described and reinforced with an example.

1. Given:

$$f_i(\mathbf{x}) = 0, i = 1 \dots n. \mathbf{x} = (x_1, x_2, \dots, x_n) \quad (2.8)$$

First define new variables  $u$  and  $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n)$ . Then form a new polynomial:

$$f_{n+1}(u, \mathbf{x}, \boldsymbol{\alpha}) = u + \prod_{i=1}^n (\alpha_i - x_i) = 0 \quad (2.9)$$

2. Next, eliminate the  $x_i$  from the system of equations  $f_i = 0, i = 1 \dots n + 1$ . This can be done using methods such as resultants (Section 2.2.2) or Gröbner bases (Davenport's book [21] gives a description of these). The final result is a single polynomial,  $V(u, \boldsymbol{\alpha})$  called the *volume function*.
3. Treat  $V$  as a univariate polynomial in terms of  $u$ , and compute the univariate Sturm sequence for  $V$ . Evaluating the resulting Sturm sequence at  $u = 0$  leaves a Sturm sequence in terms of the  $\alpha_i$ . This Sturm sequence is called  $M(\boldsymbol{\alpha})$ .
4. Now, for a given point,  $\mathbf{a} = (a_1, a_2, \dots, a_n)$  in  $n$ -dimensional space, the Sturm value is determined by counting the permanencies when  $\mathbf{a}$  is substituted for  $\boldsymbol{\alpha}$ . Here, the Sturm value counts the number of real roots of the original set of equations that make positive volume with  $\mathbf{a}$ , plus half the number of imaginary roots. Positive volume simply means that:

$$\prod_{i=1}^n (\beta_i - a_i) > 0 \quad (2.10)$$

where  $\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_n)$  is a root of the original set of equations.

5. To count the number of real roots inside of an  $n$ -dimensional rectangle,  $[a_1, b_1] \times [a_2, b_2] \times \dots \times [a_n, b_n]$ , define the function:

$$\begin{aligned} E_i(M) &= \frac{1}{2}(E_{i-1}(\alpha_i \leftarrow b_i) - E_{i-1}(\alpha_i \leftarrow a_i)) \\ E_1(M) &= \text{per}(M(b_1)) - \text{per}(M(a_1)) \end{aligned} \quad (2.11)$$

where *per* means the number of permanencies in the Sturm sequence.  $M$ , and the notation  $x \leftarrow y$  means the substitution of  $y$  for  $x$ .

**Specialization:** To understand this process more clearly, the specialization to two dimensions is provided. Note that the 2D operation is also the fundamental operation involved in the boundary evaluation algorithm.

1. Given two polynomials.  $f(s, t)$  and  $g(s, t)$ , first form the polynomial:

$$h(s, t) = u + (\alpha_1 - s)(\alpha_2 - t) \quad (2.12)$$

2. From the three equations,  $f(u, \alpha_1, \alpha_2, s, t) = f(s, t)$ ,  $g(u, \alpha_1, \alpha_2, s, t) = g(s, t)$ , and  $h(u, \alpha_1, \alpha_2, s, t)$ , eliminate  $s$  and  $t$ . Using resultants, this can be done as follows:

$$V(u, \alpha_1, \alpha_2) = \frac{Res_t(Res_s(f, h), Res_s(g, h))}{u^{(deg(f(s,0))deg(g(s,0)))}} \quad (2.13)$$

where  $Res_x(a, b)$  means the resultant of  $a$  and  $b$  obtained by eliminating  $x$ , and  $deg$  stands for the degree of the polynomial. That is, first eliminate  $s$  from the three polynomials in five variables, leaving two polynomials in four variables, then eliminate  $t$  from those, leaving  $V$ . The denominator simply removes an extraneous power of  $u$  that is a result of taking repeated resultants.

3. Now treat  $\alpha_1$  and  $\alpha_2$  as constants and form the Sturm sequence for  $V(u)$ . Then set  $u = 0$ . Call this Sturm sequence  $M(\alpha_1, \alpha_2)$ .
4. Given a point,  $(p, q)$ , the number of permanencies in  $M(p, q)$ , i.e.  $per(M(p, q))$  gives the number of roots that make positive volume with  $(p, q)$ , plus half the number of imaginary roots. The roots that make positive volume are illustrated in Figure 2.11.
5. To count the number of roots inside the rectangle  $[a, b] \times [c, d]$ , use Equation 2.11 to determine the following formula for computing the number:

$$\frac{per(M(a, c)) + per(M(b, d)) - per(M(b, c)) - per(M(a, d))}{2} \quad (2.14)$$

This formula can also be verified by considering how roots in each of the nine regions labeled in Figure 2.11 would be counted.

**Example:** An example of a 2D situation, taken from Milne's paper [75], follows. Assume we are given:

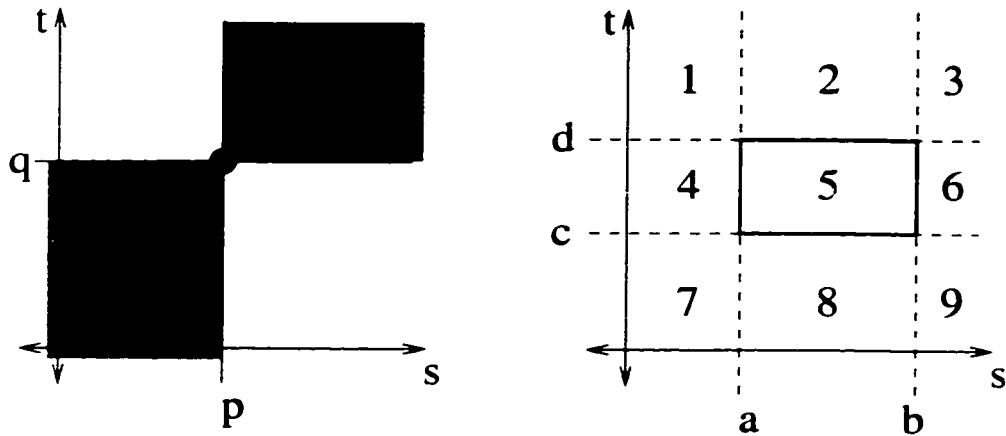


Figure 2.11: **Counting roots in a box using 2D Sturm sequences.** At left, the shaded region shows the region of positive volume with respect to the point  $(p, q)$ . Any roots in the shaded region are counted. At right, the nine regions of the plane associated with a rectangle. Only roots inside region 5 are to be counted. The counting formulas ensure that roots in the other regions are not counted.

$$f(s, t) = s^2 + t^2 - 2$$

$$g(s, t) = s - t$$

The common solutions are  $(-1, -1)$  and  $(1, 1)$ . First form the third polynomial:

$$h(u, \alpha, \beta, s, t) = u + (\alpha - s)(\beta - t)$$

Eliminate  $s$  and  $t$  to obtain:

$$\begin{aligned} V(u, \alpha, \beta) &= (u + \alpha\beta + 1)^2 - (\alpha + \beta)^2 \\ &= u^2 + 2u\alpha\beta + 2u + \alpha^2\beta^2 - \alpha^2 - \beta^2 + 1 \end{aligned}$$

Treating  $V$  as univariate in  $u$ , form the Sturm sequence:

$$u^2 + 2u\alpha\beta + 2u + \alpha^2\beta^2 - \alpha^2 - \beta^2 + 1$$

$$2u + 2\alpha\beta + 2$$

$$4\alpha^2 + 8\alpha\beta + 4\beta^2$$

Setting  $u = 0$ , then, gives the Sturm sequence:

$$M(\alpha, \beta) = \alpha^2\beta^2 - \alpha^2 - \beta^2 + 1$$

$$2\alpha\beta + 2$$

$$4\alpha^2 + 8\alpha\beta + 4\beta^2$$



Assume that we would like to count the number of roots inside the rectangle:  $[0, 2] \times [\frac{1}{2}, \frac{3}{2}]$ . Since  $M(0, \frac{1}{2}) = (\frac{3}{4}, 2, 1)$ , then  $per(M(0, \frac{1}{2})) = 2$ , meaning that two roots make positive volume with the point  $(0, \frac{1}{2})$ . Likewise,  $per(M(0, \frac{3}{2})) = 1$ ,  $per(M(2, \frac{1}{2})) = 1$ , and  $per(M(2, \frac{3}{2})) = 2$ . Thus, by Equation 2.14 we have the number of roots in the rectangle to be:

$$\begin{aligned} & \frac{1}{2}(per(M(0, \frac{1}{2})) + per(M(2, \frac{3}{2})) - per(M(2, \frac{1}{2})) - per(M(0, \frac{3}{2}))) \\ &= \frac{1}{2}(2 + 2 - 1 - 1) \\ &= 1 \end{aligned}$$

Since the root  $(1, 1)$  is in the interval of interest, while the only other real root,  $(-1, -1)$  is not, this is correct.

**Restrictions:** Like univariate Sturm sequences, multivariate Sturm sequences are not well defined along the boundary. Just as with univariate methods, the way that zeros in the sequence are handled determines how roots along the boundary are counted. Unfortunately, multivariate methods can also have problems when a root has the same coordinate value as one of the bounds of the rectangle to be tested. For example, in two dimensions assume  $[a, b] \times [c, d]$  is the rectangle to be tested, to find intersections of  $f(s, t) = 0$  and  $g(s, t) = 0$ . If the common solutions (over  $\mathbb{R}^2$ , not just within the rectangle) of the polynomials are of the form  $(\alpha_i, \beta_i)$ , then there can be problems if  $\exists i : \alpha_i \in \{a, c\}$  or  $\beta_i \in \{b, d\}$ .

## 2.3 Previous Work

This section discusses previous work in boundary evaluation and robustness. Although there has been some overlap between these two areas, much of the work has remained separate. In particular, robustness issues have been addressed mainly in terms of general geometric problems, rather than specifically applied to boundary evaluation.

### 2.3.1 Boundary Evaluation

There has been a great deal of work on subjects related to boundary evaluation. A comprehensive review of all relevant work from every subject is too extensive for this dissertation. This section highlights the relevant, interesting, and widely known previous work.

Boundary evaluation is a topic in the realm of solid modeling, which deals with representing and manipulating models of solid objects. Before the development of solid modeling there was a significant amount of work in geometric modeling, which deals with representing geometric data that might or might not be part of a solid. Geometric modeling has focused on the representation of curves and surfaces, with surface intersection being one of the problems studied.

The study of surface intersection dates to the ancient Greeks. Hohmeyer gives an excellent review of the history of surface intersection in his PhD dissertation [46]. Surface intersection has continued to be an area of research interest. A variety of approaches have been used, including ones based on subdivision, curve tracing, and algebraic representation (the approach taken in this dissertation). These approaches include that of Sarraga [88], Abhyankar and Bajaj [1], Farouki et al. [32], Manocha and Canny [67], Shene and Johnstone [93], and Goldman and Miller [74].

Research in solid modeling began in the 1960s and grew rapidly in the 1970s [81]. CSG and B-rep were formally defined during this time. An overview of early developments in solid modeling is given by Requicha and Voelcker [82]. In their books, both Hoffmann [43] and Mäntylä [69] give comprehensive introductions to the field of solid modeling, including boundary evaluation. Over time, solid modeling and geometric modeling gradually merged, with some of the work in geometric modeling being motivated by solid modeling (including much of the surface intersection work mentioned above [67, 93, 74]). A survey of some of the directions that solid modeling research has taken in recent years is given by Requicha and Rossignac [81].

Boundary evaluation has been an important part of solid modeling systems for many years. Braid treated boundary evaluation (although he did not call it that) in his 1975 paper describing B-reps [12]. Most of the earliest work on boundary evaluation dealt with polyhedral solids. Requicha and Voelcker's work is an example [83]. Later work extended boundary evaluation to curved surfaces. Casale and Bobrow presented one of the first detailed descriptions for boundary evaluation for curved solids [16]. Later work on boundary evaluation for curved solids includes that of Krishnan [62], which motivated the work in this dissertation.

Robustness in boundary evaluation has gained greater attention in recent years. Most of this work has focused on boundary evaluation for polyhedra, where all surface are linear. Some of the approaches use exact computation. These include those by Sugihara and Iri [99], Yu [105], Benouamer et al. [10], Sugihara [98], and Fortune [35]. Other approaches focus on different methods for increasing robustness. These

include those by Hoffmann et al. [45], Fang et al. [31], Higashi et al. [42], Chubarev [17], and Hu et al. [48, 49], Stewart surveys several other methods, and develops a more formalized theory of robustness in relation to polyhedral boundary evaluation [97]. Methods to increase robustness, including some of the approaches mentioned here, are described in more detail in the following section (Section 2.3.2).

There has been a limited amount of previous work toward robust boundary evaluation for curved solids. Some of the work has been in the area of surface intersection. For example, Farouki et al. [32] and Goldman and Miller [74] look at degenerate intersections of quadric surfaces. Yu explores some theoretical bounds on exact arithmetic in boundary evaluation for curved solids [105]. Fang et al. [31] use tolerances to achieve more robust boundary evaluation for solids with quadric surfaces. Hu et al. [48, 49] use interval computations for robust boundary evaluation of more complicated solids.

### 2.3.2 Robustness

This section gives a brief description of some of the previous approaches to the robustness problem. In recent years, the robustness problem has gained increasing attention in the field of computational geometry. The basic issues related to robustness have already been described in Section 2.1.3. As mentioned, the two basic (and interrelated) categories of robustness problems are those due to numerical errors (i.e. precision problems) and those due to degeneracies.

There are several possible definitions for what makes an algorithm robust. Some have devised systems for specifying the type of robustness an algorithm achieves [23]. A number of approaches that are outlined here claim to be robust. Other approaches only claim to increase robustness. Rather than try to examine only approaches that meet certain robustness criteria, this section describes techniques that claim to at least increase robustness.

The tests performed by various methods are often for a particular *predicate*. A predicate is a simple, well-defined test that can be performed on the data and is used to determine the flow of control in a program. It is usually just the sign of some polynomial calculation on input data. An example of a commonly used predicate is the plane orientation test (determining which side of a plane a point lies on), which is usually performed by taking sign of the determinant of a matrix. On the other hand, *constructors* are used to actually generate a new geometric object, and require the value of some polynomial calculation on the input data.

The degree of a computation must be considered when evaluating techniques that address robustness. Most of the classical computational geometry problems have been defined for linear objects (e.g. lines, polygons, polyhedra). Several problems of interest, particularly those in geometric and solid modeling, make use of quadric or higher degree objects. Higher degree objects will, in general, cause execution times and robustness problems to become much worse [73].

There have been several surveys of the robustness problem. Many papers give a brief survey of robustness issues in general before presenting some new technique. Among those that survey the robustness problem (to various levels of detail) are papers by Hoffman [44, 43], Schirra [89], Fortune [34], and Seidel [92].

### 2.3.2.1 Precision Problems

The real RAM model of computation simplifies the design and description of algorithms [33], but can overly simplify analysis, as shown in a paper by Kahan and Snoeyink [53]. Often, implementations of algorithms choose to sacrifice numerical accuracy for the sake of speed. For many types of problems and in many cases this is acceptable. The basic problems of error and error propagation have been well studied in numerical analysis literature. For geometric problems, representations usually involve both numerical and combinatoric data, and choices about combinatoric structure may be made based on numerical data [47]. Thus, error accumulation can lead to incorrect combinatoric information.

The approaches taken to deal with precision problems generally fall into two categories. Some approaches continue to use floating-point or some other finite precision arithmetic. Other approaches use some form of exact computation.

**Floating-point and Finite Precision** The efficiency and ease-of-use of floating-point arithmetic make it attractive for numerical computation. In order to achieve robustness, however, special techniques need to be used. Following are descriptions of some of the numerous techniques used to increase robustness while using finite precision computation.

**Interval Arithmetic.** With interval arithmetic, each number is stored as a interval. The actual number lies somewhere between the two interval bounds. Computations on the number are actually performed as a set of computations on the interval bounds. For example,  $[a, b] + [c, d] = [a + c, b + d]$ . Similar rules define the other interval operations. It is important to round the floating-point interval bounds

correctly in order to ensure that the final interval will contain the answer. Entire books have been written that deal exclusively with the use of interval arithmetic [77].

By storing the interval bounds as floating-point numbers, the interval computations can be performed quickly. There are several drawbacks to using these methods, however. The computed intervals may be much larger than what is actually necessary. For example, using interval arithmetic to calculate  $x(10 - x)$  for  $x = [4..6]$  gives:

$$\begin{aligned} 10 - x &= [10..10] - [4..6] = [4..6] \\ x(10 - x) &= [4..6][4..6] = [16..36] \end{aligned}$$

which is much wider than the actual range of [24..25]. For complex computations, the output interval size may grow too large to be useful. It may not be possible to make the starting intervals small enough for the final computation to have an acceptably narrow interval size. The intervals themselves may not be able to separate two points that are close together. Guaranteeing the equivalence of equality (where  $A = B$  and  $B = C$  implies  $A = C$ ) can be a problem in naive implementations, but it is possible [48]. Despite the drawbacks, interval arithmetic has been used in practice, with some success. Hu et al. have even adapted this approach for use on curved surfaces in a solid modeling system [49].

**Affine Arithmetic.** An extension to interval arithmetic is the use of affine arithmetic [19]. Affine arithmetic reduces the overly conservative bounds sometimes generated by interval arithmetic. It does so by keeping track of correlations between error introduced at each step in a long computation chain. Each number is stored as an affine form:

$$x = x_0 + x_1\varepsilon_1 + \dots + x_n\varepsilon_n$$

where  $\varepsilon_i = [-1..1]$  and represents one source of error. As new potential errors are introduced, new  $\varepsilon_i$  are introduced. This allows one to accurately predict cancellation of error by reuse of the same variable. For example, the previous calculation of  $x(10 - x)$  for  $x = [4..6]$  gives:

$$\begin{aligned} x &= 5 + 1\varepsilon_1 \\ 10 - x &= 5 - 1\varepsilon_1 \\ x(10 - x) &= 25 + 5\varepsilon_1 - 5\varepsilon_1 - 1\varepsilon_2 = 25 - 1\varepsilon_2 = [24..26] \end{aligned}$$

which is much closer to the actual interval, [24..25], than the previous calculation of [16..36].

The use of affine arithmetic is slower than the use of standard interval arithmetic, but in cases where there might be error correlation from one step of a computation to the next, it is beneficial. Affine arithmetic has been used for applications in

computer graphics [19] and surface intersection [22]. Its applicability to more classical computational geometry problems has not yet been looked into.

**Epsilon Geometry.** Another method closely related to interval arithmetic is the epsilon geometry defined by Guibas et al. [40]. Epsilon geometry uses *epsilon predicates* to solve problems. These predicates return intervals that identify regions over which the predicate is “definitely true”, “definitely false”, or “uncertain.” The interval defines a range by which input data can be perturbed in order to reach the “uncertain” interval (and thus, by implication, the range that gives a true or a false return). Thus, if a predicate returns with a “definitely false” region of points  $< 0.3$  that means that this predicate will always return false as long as the data is perturbed by less than 0.3. Here it is not the error in a particular number that is being measured, but rather the allowable error in all numbers for a computation to work correctly. These predicates can be used to determine the amounts by which one would have to perturb data to account for rounding error in a perturbation scheme (Section 2.3.2.2). Unfortunately, there have been few predicates developed, and it is not clear how well this method extends to more complex predicates.

**Tolerances.** Tolerances are one of the most common methods used in practice for increasing robustness. With tolerance methods, tests for equality are done within a certain specified tolerance. Thus, two points that are less than the tolerance value away from each other are considered to be coincident.

There are several potential benefits of tolerance methods [50]. First, by picking a tolerance value large enough, the error buildup due to floating-point arithmetic can be ignored for most cases. If the error buildup is less than half the tolerance, then comparisons between two values are still valid. A second benefit to tolerances is that small features (those smaller than the tolerance) are automatically eliminated. In several applications, the presence of small features in the output can cause problems. A third benefit is that tolerances can sometimes mimic actual physical constraints. For example, manufacturing tools are only able to handle a certain degree of precision - any higher precision in the output is wasted. Also, input data is often created with tolerances in mind. The notion of tolerances for geometric computation (as a way to increase robustness) is different from, although related to, the technical notion of tolerances in computer-aided-design. Design tolerances are tolerance amounts specifically placed by the designer to account for physical constraints such as wear or manufacturing imprecision [31]. There is also work on representing true design tolerances in geometric and solid modeling systems [41] and choosing appropriate design

tolerances [64].

There are several drawbacks to the use of tolerances. First, unless a common tolerance value is used on all machines, there is no guarantee that the output on one machine will be the same as the output on another. Second, it can be difficult or impossible to find a global tolerance value that works well for an entire computation. One part of the data might require a low tolerance and another require a high tolerance. No single tolerance will be appropriate for all requirements. Third, tolerances suffer from the problem of equality not being equivalent (i.e.  $A = B$  and  $B = C$  but  $A \neq C$ ). Fourth, error buildup can still occur to a point where the error is larger than the tolerance, defeating the whole purpose for setting a tolerance. Simply making the tolerance value larger may eliminate small but significant features, make some operations such as subdivision more difficult, and give output that is not accurate.

One potential solution to some of these problems is the use of local adaptive tolerances by Segal [91], which was extended to non-planar cases by Jackson [50]. With local tolerances, different tolerance values are applied to different parts of the model. In areas where higher tolerance values are needed, they are increased as necessary, and topological structures that are close together are merged. This approach is similar to the use of interval arithmetic, in that the tolerances define intervals that are expanded as necessary to account for error buildup. The drawbacks to this approach are that several of the problems with tolerances remain and the tolerance values can grow rather large.

An approach by Fang et al. also uses different tolerances for different parts of the model [31]. In this approach, many tolerances are kept, and decisions are made as to whether geometric objects are apart, coincident, intersecting, or ambiguous (which requires refinement of the interval), based on how the tolerance bounds overlap. A drawback to this approach is that setting up and manipulating the tolerances is difficult. Still, this approach has been applied to Boolean operations on 3D objects bounded by planes and quadrics, with promising results.

**Data Normalization and Hidden Variables.** Milenkovic uses the methods of data normalization and hidden variables to perform robust computation in finite precision arithmetic on linear objects [72]. With data normalization, the input data is modified to meet certain normalization conditions. Namely, points must be at least a certain distance apart and points must be at least a certain distance from a line. Points that are too close together are merged (this is called *vertex shifting*), and edges that are too close to a point are split into two edges that meet at that point (this

is called *edge cracking*). The hidden variable method, on the other hand, does not actually modify numerical data, but operates as if the data were behaving in a certain way topologically. This behavior should be close to the real behavior but does not have to be exact. In effect, topological questions are answered as if the data were slightly different than it actually is, behaving according to some hidden topology. This hidden topology does not ever need to be computed specifically. Using these methods, more robust geometric algorithms are obtained. Milenkovic has extended his work with fixed precision data to begin looking at applications to higher degree objects [73], but the application has been limited.

**Rounding.** Guibas and Marimont [39] have described a successful implementation of a method for rounding line segment arrangements to a grid. In other words, the endpoints and intersection points of the line segments are *snap rounded* to an integer grid. In many ways, this is similar to data normalization. They show that snap rounding can maintain the correct topology among all segments by representing each segment by a short chain of segments with endpoints snapped to the integer grid. Snap rounding holds promise as a way to consistently round geometric computations without altering the overall combinatorial structure.

Methods similar to snap rounding have been used to construct link paths. Link paths are a series of line segments connected at the endpoints. Minimum link paths for certain applications can be shown to require a more bits of precision than the input data. In order to determine a link path in finite precision, Kahan and Snoeyink described a method for constructing link paths where the endpoints are snapped to a predetermined grid [53]. This allows a (not necessarily minimal) link path to be constructed without resorting to higher precision (exact) arithmetic.

**Estimated Error.** Masotti [70] has developed a method for performing floating-point computation using an estimated error. When worst-case error estimates are kept (such as in interval arithmetic), one is guaranteed that the final result is located within a certain bounded interval. However, after several computations, this interval may be so large that it is worthless. Masotti's error estimate is much smaller than the worst-case error, and thus the final answer is more useful. In addition, this approach detects certain ill-conditioned computations. Masotti demonstrates that this approach can be used to perform operations far more robustly than with standard floating-point, although the results are still not guaranteed.



**Exact Computation** With exact computation, all computations are performed such that the outcome of the computation is known precisely. All numerical data is kept to whatever precision is necessary for an algorithm. The size of the individual numbers must be taken into account when performing arithmetic operations on them, so the constant-time assumption of the Real RAM model is violated.

There are numerous implementations of exact arithmetic routines. Exact arithmetic is used in all the common computer algebra systems. Several programming libraries are also available for performing exact arithmetic. One of the most well-developed current libraries is the LEDA library [30]. For a description of a number of other libraries available, see Yap and Dube's paper [102].

Exact arithmetic routines are usually confined to operations on integers or rational numbers. Some packages handle algebraic numbers as well, and this extension appears to handle almost all problems in computational geometry [104]. As long as computations require only rational numbers, exact arithmetic can be used directly to implement a geometric algorithm. Since the numbers and computations are exact, rounding error is not a concern. The drawback to exact arithmetic is efficiency – numbers can take a significant amount of space to store and computations can take a long time to perform. Thus, most work in exact computation deals with ways of making exact computation more efficient.

The standard representation of exact integers is a set of bits that exactly expresses the integer. Rational numbers are stored as two integers (for the numerator and denominator). Algebraic numbers are usually stored as two items: a polynomial equation and an interval. The idea of exact floating-point numbers, in which the mantissa and exponent are kept exactly, has also been investigated. Any representation that allows numbers to be stored exactly (or equivalently, such that they can be determined to any given precision) is acceptable in an exact computation paradigm.

Yap and Dube point out several reasons why exact computation is an important paradigm [102]. Among the reasons they list are that exact algorithms are much more straightforward than a corresponding robust floating-point algorithm, that almost all algorithms (in computational geometry and elsewhere) are developed assuming exact arithmetic, and that exact computation is useful for studying and validating floating-point computation. Yap also points out [104] that as processor speeds increase, the appeal of robustness offered by exact computation may begin to approach the appeal of speed that floating-point offers. Yap's papers [102, 104] provide an excellent overview of many of the concepts and techniques used in exact arithmetic.

A number of techniques have been used to improve the efficiency and applicability of exact computation. Among these techniques are the following methods:

**Interval Arithmetic.** Karasick et al. [56] look into methods for improving exact computation by the use of interval arithmetic. In their approach, they attempt to perform computations using a simplified representation of the rational number (i.e. one requiring fewer bits). Instead of computing with the rational number itself, they bound each rational number with an interval with endpoints that are rational numbers that require fewer bits. Interval arithmetic is used in the computation, and the final result is an interval. For an appropriate predicate, such as sign of a determinant, the computed (interval) answer either is enough to answer the question or indicates that higher precision is needed. If higher precision is needed, then the original intervals are chosen to have more bits of precision. If all attempts at simpler intervals fail, the exact number itself is used.

Since the interval bounds have far fewer bits than the actual number, the computation using interval arithmetic can potentially take far less time than the computation would have had the exact number (with all bits of precision) been used. The usefulness of this approach is limited, however, in that the predicates need to be carefully chosen so that they can be computed efficiently using interval arithmetic.

Another adaptation of interval arithmetic to exact computation is the approach taken by Johnson [51]. Johnson, like Karasick, uses intervals with rational endpoints. Using these intervals, he shows how exact computations can be performed for real numbers that are bounded by an interval, even if the real number itself is not known. Examples of the computations include sign evaluation and substitution into a polynomial. To achieve greater efficiency, Johnson makes use of binary rational numbers, where the denominator is a power of two. Thus, only the power of the denominator needs to be stored, and basic rational operations (+, −, ×) are faster.

**Floating-point Filters.** The idea of floating-point filters was introduced by Fortune and van Wyk as a way to potentially avoid exact computation [36]. Like the interval arithmetic approach of Karasick et al. [56], the idea is to compute an approximate value quickly, then determine whether that value is good enough. With floating-point filters, the computation is performed in floating-point, but an error bound is kept. If the final error bound is too large to precisely determine the answer to a predicate, then the exact computation is performed. Otherwise, the answer determined by floating-point is used. Fortune and van Wyk found that filters dramatically speed up the determination of certain predicates.

**Tuning Computations.** In the same paper [36], Fortune and van Wyk propose a method for *tuning* a computation. By this, they mean that an entire computation for some predicate is parsed, and a complete straight-line program is created to evaluate that computation. The parser estimates the precision required for intermediate computations. This program is then compiled and run. Since the necessary number of bits is predetermined, the computation is performed efficiently and exactly. Fortune and van Wyk see significant improvements from this approach as well. They have implemented a package, LN, that can automatically produce C++ code for evaluating linear geometric primitives [37]. This package includes floating-point filters to increase efficiency.

**Lazy Arithmetic.** Another approach geared toward improving the efficiency of exact computation is lazy arithmetic [10]. With this approach, a number is stored in a floating-point interval format as well as in a directed acyclic graph (dag) format. The graph format is used to trace the history of how the final number is computed. For example, if two numbers are added together, the graph consists of three nodes: one for each input number, with links heading toward an addition node that represents the output. For most computations, only the interval is used. In case the interval is not tight enough, the actual rational number answer is computed by following the dag. As an alternative, a progressive approach can be used, continually starting with tighter intervals and propagating those down (again resorting to the exact computation after a certain point). In the worst case, just as many exact rational operations have to be performed. In the best case, exact rational operations never have to be performed, and the result is determined from the floating-point interval computations alone.

**Precision-Driven Computation.** Another approach, similar to the lazy arithmetic approach, is to use Yap and Dube's precision driven arithmetic [102]. With this approach, a dag is constructed, but instead of propagating information from the input to the output, the desired precision of the output is specified, and that information is propagated backward in the dag to determine to what precision the input needs to be specified. In the worst case, the exact computation needs to be performed, but more typically, the input can be inexact and still give a final output that is within the desired precision.

**Minimizing Intermediate Precision.** Clarkson presents a method for finding the sign of a determinant exactly and efficiently with exact arithmetic [18]. He computes the result exactly, but manages to formulate the problem in such a way that the number of bits of precision required is far less than one would expect. This allows the

approach to be implemented effectively using hardware-based arithmetic, assuming that the original data contains a limited number of bits. A group from INRIA gives a different approach (based on modular arithmetic, see below) for determinant sign evaluation that also minimizes intermediate precision [13]. By implementing a basic geometric predicate in such an efficient manner, an overall geometric operation may be performed in a reasonable amount of time, while maintaining exactness.

The drawback here is that the efficiency is obtained for only one operation – determinant sign evaluation. If similarly efficient approaches can be found for other important operations, then exact arithmetic can be used widely and efficiently for many geometric problems. It is not clear, however, that such efficient implementations of other operations exist.

**Fast Hardware Computation.** Shewchuck has developed a method for quickly finding exact determinations of some predicates using only standard floating-point hardware [95]. Shewchuck uses specialized routines for performing exact arithmetic using floating-point hardware, based on work done by Priest [80]. An adaptive computation method is used such that predicates are determined with less than exact precision, if possible. The efficiency comes from the use of floating-point hardware, the relaxation of storage conditions for high precision numbers, and the use of adaptive computation with the predicates. This approach is promising in that exact computation is performed more quickly and, if the desired predicates are amenable to it, adaptive computation allows individual predicates to be computed quickly.

**Modular Arithmetic** One approach that is commonly used (e.g. [36, 13]) to speed up exact computation is the use of modular arithmetic. Rather than storing integers as long series of bits, the integers are stored modulo a set of prime numbers with small, fixed precision. Then, computation takes place in hardware for each of the modulo numbers. If necessary, the Chinese Remainder Theorem is used to construct the final number. In some cases, the final number does not have to be completely reconstructed, since one might be interested only in some aspect of the number (e.g. the sign). Use of modular arithmetic can give a significant speedup.

### 2.3.2.2 Degeneracies

The manifestation of degeneracy problems in a program is generally the failure of a predicate to determine which branch of a program to take. When there are no degeneracies, the predicate determines which of two branches to take by whether the sign of a polynomial is positive or negative. When degeneracies are present, the

predicate polynomial evaluates to zero, thus making the choice of branch unclear. With inexact computation, a predicate might evaluate to zero even when it shouldn't (effectively creating a degeneracy when there's not one), or a predicate that should evaluate to zero might not (effectively removing a degeneracy that should be there). Although in some cases the loss of a degeneracy might seem like a good thing, it may cause serious problems if the special case handling routines expect to find all degeneracies.

Two main approaches are used to handle degeneracies. These are the use of special cases and the use of symbolic perturbation.

**Special Cases** Usually, computational geometry algorithms are designed assuming that no degeneracy exists. If a degenerate case does arise, it is treated as a *special case* for the problem. This means that a special routine is called when a predicate evaluates to zero.

In order to use special cases, a programmer must determine all potential degeneracies, detect them, and deal with them. Each of these requirements can be difficult. Enumerating all potential degeneracies is difficult, detecting the degeneracies can be difficult and lead to confusing code, and dealing with a detected degeneracy within the framework of the original algorithm can be difficult or impossible. Still, treating degeneracies as special cases is usually the most straightforward and most commonly used method. Yu demonstrates one attempt to handle degeneracies through the use of special cases [105].

**Perturbation Methods** A general approach for dealing with degeneracies is the use of perturbation methods. The idea behind perturbation methods is to modify the input data such that one is guaranteed to have no degeneracies. The goal is to make sure that a predicate never evaluates to zero. There are a number of different techniques for doing so.

Some question the validity of all perturbation methods [92]. One of the most compelling arguments against perturbation methods is that degenerate cases are often there on purpose. Thus, perturbing the data to make it non-degenerate actually loses important information. A second strong argument against perturbation is that the solution determined is not actually the solution of the given problem. Thus, the computed solution might have to be transformed into the solution to the given problem. Such a transformation can be difficult. A third argument against perturbation

methods is that the perturbed problem can be much more difficult to solve than the original problem. A fourth argument against perturbation is that perturbation methods generally require some form of exact computation in order to be successful. This requirement makes implementations of perturbation methods run much slower than those that do not use exact computation.

Despite these drawbacks, perturbation methods remain one of the best known approaches for dealing with degeneracies consistently. A well-chosen perturbation method eliminates all potential degeneracies, meaning that no special cases have to be detected. Since exact computation is used, algorithms can be implemented in their most straightforward sense. Perturbation methods have proven successful for several problems and there is hope that they will be applied successfully to a much wider array of problems.

In a more formal sense, Seidel [92] defines a perturbation scheme:

For input space,  $\mathcal{I}$  and output space,  $\mathcal{O}$ , with  $q \in \mathcal{I}$ , a *perturbation* of  $q$  is any curve  $q(\epsilon) : [0, \infty) \mapsto \mathcal{I}$  with  $q(0) = q$ .

A *perturbation scheme*,  $Q$ , induces for every function  $F : \mathcal{I} \mapsto \mathcal{O}$  a perturbed function  $\bar{F}^Q : \mathcal{I} \mapsto \mathcal{O}$ , defined by

$$\bar{F}^Q(q) = \lim_{\epsilon \rightarrow 0^+} F(q(\epsilon))$$

If  $F$  is continuous at  $q$ , then  $F(q) = \bar{F}^Q(q)$ . Even when  $F$  is not continuous at some  $q$ , usually one can easily recover  $F(q)$  from  $\bar{F}^Q(q)$ . Thus, if the computation is continuous for all inputs, the output of the perturbed problem is exactly the same as that for the unperturbed problem. In cases where the computation is not continuous for all the inputs (as may be the case for degenerate input conditions), the output for the original problem must be recovered through some transformation on the output of the perturbation scheme. In many cases, this transformation is fairly simple although it can be rather difficult, possibly even more difficult than solving the unperturbed degenerate problem. Note that the actual perturbation is never computed. Rather, the computation proceeds treating the perturbation amount as a variable, and the limit of the perturbation is taken at the end.

Following Seidel's outline, there are three primary proposals for perturbation methods:

**Simulation of Simplicity.** Probably the first general perturbation scheme proposed was the Simulation of Simplicity (SoS) scheme of Edelsbrunner and Mücke [27].

With this method, given a set of  $n$  geometric objects, each of which takes  $d$  input parameters, the data is perturbed such that

$$\pi_{i,j}(\epsilon) = \pi_{i,j} + \epsilon^{2^{i \cdot \delta - j}}$$

for  $0 \leq i \leq n-1$ ,  $1 \leq j \leq d$ , and  $\delta \geq d$ . Notice that the amount of the perturbation is different for every  $i, j$  pair. This ensures that each of the original points is perturbed to a different amount

This perturbation can then be used in a predicate. For example, the paper by Edelsbrunner and Mücke uses a point orientation test that is the sign of a determinant. The determinant for the perturbed points gives a polynomial in terms of  $\epsilon$ .

Thus, to find the sign of the determinant as  $\epsilon$  approaches  $0^+$ , one needs to find the sign of the lowest-degree nonzero term of the polynomial. The polynomial's constant term (i.e. its lowest degree term) is nothing more than the determinant of the matrix without any perturbation applied. Thus the only time that the perturbed case requires more work than the non-perturbed case is when the non-perturbed determinant evaluates to zero - i.e. a degenerate case. In these cases, the sign of the determinant is computed from the sign of the smallest nonzero term of the polynomial. Thus, a sign determination is made even when the data is given in degenerate format.

**Linear and Random Perturbation.** Emiris and Canny present a variation on the SoS method that makes use of a linear perturbation [29, 28]. Specifically, the perturbation used is:

$$\pi_{i,j}(\epsilon) = \pi_{i,j} + \epsilon \cdot i^j$$

Notice that in this representation, the perturbation is linear in  $\epsilon$ , improving the efficiency of the approach. Also, the overall scheme can be viewed as a vector,  $\pi_i$ , being perturbed by another vector,  $b_i$ , where the elements of  $b_i$  are just  $i^j$ . Emiris and Canny show that this perturbation scheme can be used in a variety of predicate tests, and apply it to convex hull computation. They perform a rather extensive analysis of time and bit complexity required.

Another approach presented by Emiris and Canny in the same papers is a random permutation of the form:

$$\pi_i(\epsilon) = \pi_i + \epsilon \cdot r_i$$

In this representation,  $r_i$  is a randomly chosen number from a large interval. This perturbation is more efficiently evaluated (assuming the random numbers are chosen with an appropriately smaller number of bits) than the earlier one, but it is not guaranteed to eliminate all degeneracies. Instead, the probability of still having a degeneracy is extremely low. In the unlikely event that a degeneracy remains, it can be detected and the program restarted with a new random choice for a perturbation amount. This method is particularly applicable to cases where the computation is for branching on some arbitrary rational function, as opposed to branching on a determinant, which the earlier method is more suitable for.

**Symbolic Perturbation** Yap's method for symbolic perturbation is far more general than the other perturbation methods [103]. In Yap's formulation, a predicate can be *any* polynomial or rational function, or even an analytic function [92]. Yap's predicate returns a number if that number is not zero, or, if the predicate value is zero, returns either  $0^+$  or  $0^-$ . This is really no different than simply returning the sign of the predicate, just as the other methods do. With Yap's method, a large polynomial is produced, and from it a series of polynomials is found by taking a number of derivatives. The final sequence of polynomials is used to determine the returned value. If the polynomial itself evaluates to zero, then either  $0^+$  or  $0^-$  is returned based upon the sign of the first nonzero polynomial in the sequence. Although this approach is general and powerful, it may be difficult to compute the potentially high derivatives and large polynomial.

**Other Approaches** Other approaches to dealing with degeneracies have been used. These include redundancy elimination and automatic parsing.

**Redundancy Elimination.** Rossignac and Voelcker use a method called redundancy elimination to eliminate sources of degeneracies [84]. With redundancy elimination, data is preprocessed to remove information that is not needed in order to solve problems based on that data (i.e. redundant information). Although the primary focus of such work is for faster implementations, removing some of this extraneous data can eliminate some degenerate cases.

**Automatic Parsing.** An approach taken by Farouki et al. toward dealing with certain types of degeneracy is automatic parsing [32]. An expression is set up to determine when a degenerate case can occur, and the case is automatically formulated as a non-degenerate one instead. In this way, several degeneracies are eliminated by a relatively simple transformation. For this approach to work, the degenerate case



must involve some expression that can be turned into an equivalent expression that yields a non-degenerate case. In Farouki's example, this involves factoring a quadratic polynomial. This can be a useful approach in cases where it applies, but usually there is no simple way to create an equivalent non-degenerate case.

# Chapter 3

## Representations

This chapter describes an exact representation for curved B-rep models. For boundary evaluation, it is assumed that the input is given in this representation, and the output is produced in this representation.

To convert CSG models to B-rep models, first convert the CSG primitives to B-rep. Second, combine the primitive B-reps into the final B-rep. Appendix A describes conversion to B-rep for the CSG primitives this dissertation is concerned with. Section 3.5 describes other aspects of CSG input data, such as transformation matrices.

A solid is assumed to be closed with a manifold boundary. The boundary is represented by a number of trimmed parametric patches, also referred to as the *faces*. The trimming curves are stored in the domain of the patch, and are defined as portions of algebraic plane curves. The trimming curves define the *edges* of the object. The endpoints of these trimming curves define the *vertices* of the object. Representing these vertices requires the representation of points with algebraic coordinates. The solid as a whole is stored as an array of patches, arbitrarily ordered.

The representations described here are exact. The description assumes that an underlying system for *exact rational arithmetic* is provided. Such a system allows for storage and basic arithmetic operations on rational numbers of arbitrary precision. There are several readily available implementations of exact rational arithmetic. LiDIA [11] and LEDA [71] are two examples.

### 3.1 Patches

The *surface* that defines each patch is rational and parametric (Section 2.2.1). So,  $X(s, t)$ ,  $Y(s, t)$ ,  $Z(s, t)$ , and  $W(s, t)$  are specified, along with the implicit form of the

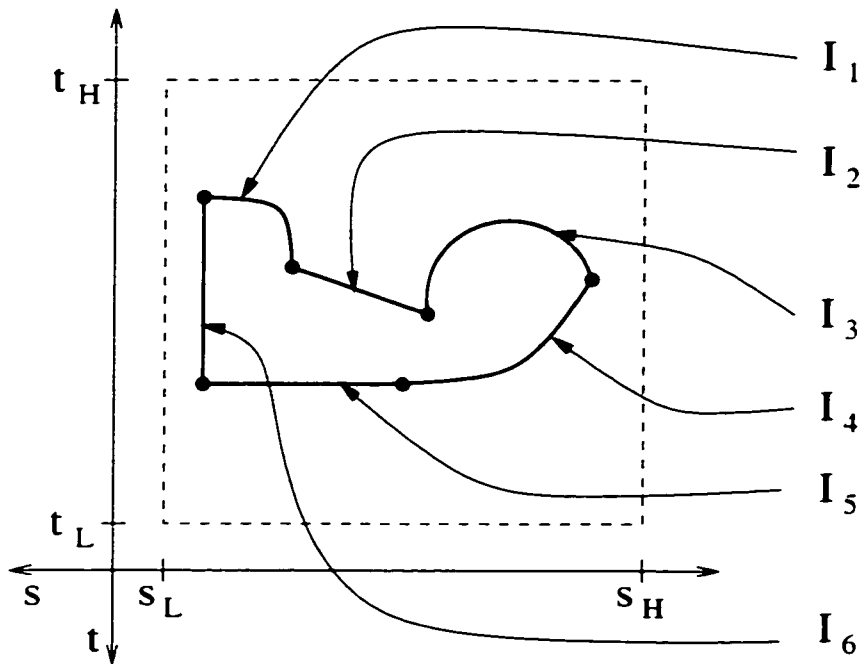


Figure 3.1: **A trimmed patch.** The dashed lines represent the extents of the patch domain. The dark curves denoted by the  $I_i$  are the trimming curves, which form edges of the solid. The circles at the beginning and end of the trimming curves represent the vertices of the solid.

surface.  $F(x, y, z) = 0$ . Such a parametric surface exactly represents all surfaces of the standard CSG primitives (Appendix A), along with many other possible primitives, such as surfaces of revolution.

Each patch is stored as a trimmed parametric patch. Note that a patch is defined by one parametric surface, but a parametric surface may define several patches. Also, different parametric surfaces can have the same implicit form.

The patch domain is defined as a rectangular region  $[s_L, s_H] \times [t_L, t_H]$ , where the limits of the domain,  $s_L$ ,  $s_H$ ,  $t_L$ , and  $t_H$ , are specified as exact rational numbers. Trimmed curves are defined within this patch domain (Section 3.2). The trimming curves are contained entirely within the patch domain. For example, if the trimming curves trace out the boundary of the  $(s, t) = [0, 1] \times [0, 1]$  region of the domain, then the patch domain must be defined by  $s_L, t_L < 0$  and  $s_H, t_H > 1$ . Figure 3.1 gives an example of a patch domain.

The trimming curves within the patch form a single closed loop. Interior loops

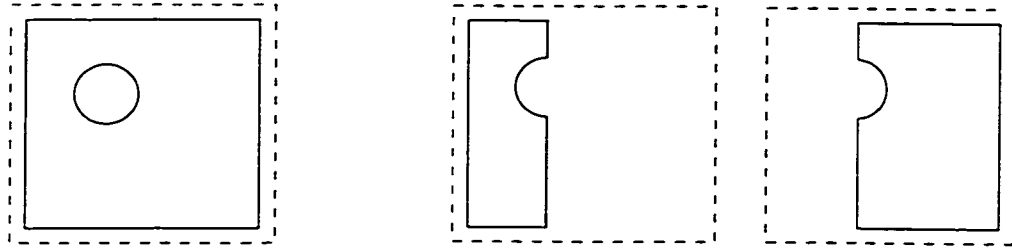


Figure 3.2: **Breaking loops.** At left, one face is represented by a single patch with an interior loop. At right, the same face is broken up into two patches, neither with an interior loop. Solid lines represent trimming curves, dashed lines the extent of the patch domain.

(i.e. holes in patches) are *not* allowed. In the cases where one wishes to represent an interior loop within a patch, the patch is instead broken up into two separate patches (Figure 3.2). The trimming curves are oriented so that when looking at the patch from the exterior of the solid, the trimming curves will go counterclockwise around the patch boundary.

Associated with each trimming curve in a patch is another surface that, when intersected with this patch, yields the algebraic curve that that trimming curve is a part of. This surface is referred to as the *adjacent surface* for a trimming curve. A given patch  $P$  will have some surface  $S_P$  associated with it. For every trimming curve  $C_i$  in  $P$ 's domain, an adjacent surface  $S_i$  is associated, such that  $S_P \cap S_i$  yields the trimming curve  $C_i$ . The adjacent surface may or may not be the surface associated with the adjacent patch. For example, in the representation for a cube, assume each of the six faces is represented by a patch with four trimming curves (one for each adjacent face). The adjacent surface of those trimming curves is the plane associated with the adjacent face. On the other hand, consider a model of a cylinder represented as six patches - one patch for each cap, and four for the curved portion, as in Figure 3.3. Each of the curved patches is adjacent to two other curved patches. Since all the curved patches come from the same surface, the adjacent surface cannot be simply the surface of the adjacent patch. Instead, the adjacent surface is defined to be the plane that separates the two adjacent curved patches, as illustrated in Figure 3.3. Only the implicit form of the adjacent surface needs to be stored. The reason for storing the adjacent surface is shown in the description of the boundary evaluation algorithm itself (e.g. in Section 5.4.2).

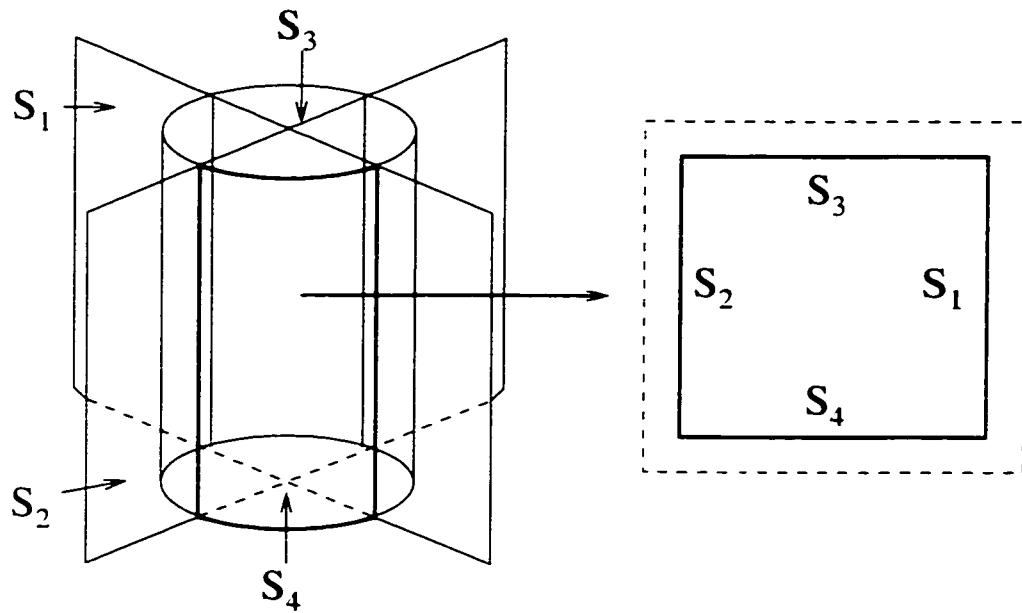


Figure 3.3: **Adjacent surfaces in a cylinder.** At left, a cylinder broken up into six patches (one for each cap, four for the sides). The  $S_i$  are the adjacent surfaces.  $S_1$  and  $S_2$  are the planes that divide the curved faces.  $S_3$  and  $S_4$  are the planes for the top and bottom caps. At right is the domain of one of the curved patches. The trimming curves are labeled with the adjacent surface.

The trimmed parametric patch format described here is a superset of many of the common curved patch representations, such as NURBS and Bezier patches. The surfaces and trimming curves of these other patch formats might need to undergo a change of basis or subdivision into many patches, but they can be converted.

## 3.2 Curves

Trimming curves are stored in the patch domain. No explicit three-dimensional representation of the curve is stored. Each curve is defined as a portion of an algebraic plane curve. The algebraic plane curve is the zero set of a bivariate polynomial with rational coefficients. This algebraic plane curve is assumed to be *regular* within the patch domain. That is, it is assumed that the algebraic plane curve does not have any singularities, such as cusps, self intersections, or isolated point solutions, within the patch domain. As is described in Appendix A, the trimming curves for the B-reps of standard CSG primitives are regular curves. Also, if input solids are in general position, all curves involved in boundary evaluation will be regular.

The representation of curves described here applies to both trimming curves and intersection curves. Intersection curves, formed during boundary evaluation (Section 5.2), become trimming curves of the output B-rep. Intersection curves are formed from the intersection of two patches. The intersection of rational patches does not necessarily have a rational parametric representation with rational coefficients. Thus, curves must be stored implicitly in order to maintain an exact representation.

It is important to remember that the term *curves* is used to refer to these general two-dimensional curves. Only the trimming curves define the *edges* of the solid.

### 3.2.1 Curve Representation

Only certain portions of an algebraic plane curve are of interest. The entire algebraic plane curve is broken into a number of curve *segments*. Each curve segment is defined by two *endpoints*, a *starting point* and an *ending point*, both of which lie on the curve. The representation for these points is described in Section 3.3. The ending point of one segment is the starting point of the next segment. In this way, an orientation is induced on the curve. Hereafter, the term *curve* refers to a series of one or more simply connected segments from a single algebraic plane curve. The *algebraic plane curve* refers to the entire curve independent of which portions are kept. Notice that

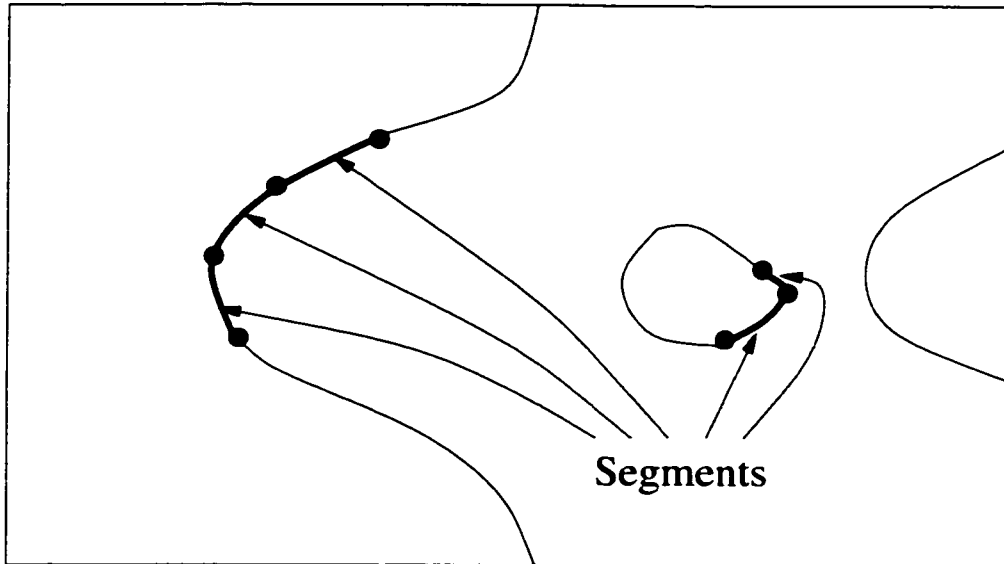


Figure 3.4: **A single algebraic plane curve and two associated curves.** The algebraic plane curve has three components in the region of interest. The heavier lines indicate the two curves. The curve at left is made up of three segments, the curve at right of two.

a single algebraic plane curve may give rise to more than one curve. See Figure 3.4 for an illustration. The *curve's polynomial* refers to the polynomial whose zero set defines the algebraic plane curve. The starting point of the curve is the starting point of the first segment of the curve, and the ending point is the ending point of the last segment.

A curve segment is broken up into many segments by introducing new points along that segment. The curve itself is unchanged, but the number of segments used to define that curve increases. Two restrictions are placed on the curve segments, which might require that the segments be subdivided. The first restriction is that the curve segments be monotonic in both  $s$  and  $t$ . That is, moving from the starting point to the ending point, the curve must be non-increasing (or non-decreasing) in  $s$  and non-increasing (or non-decreasing) in  $t$ . This involves inserting into the curve all of the *turning points*, i.e. the local maxima and minima with respect to  $s$  and  $t$ . *Inserting* points into a curve simply means subdividing the segment that point lies in so that the inserted point becomes the ending point of a segment (and thus the starting point of another segment).

Consider rectangular axis-aligned boxes around each segment, just large enough

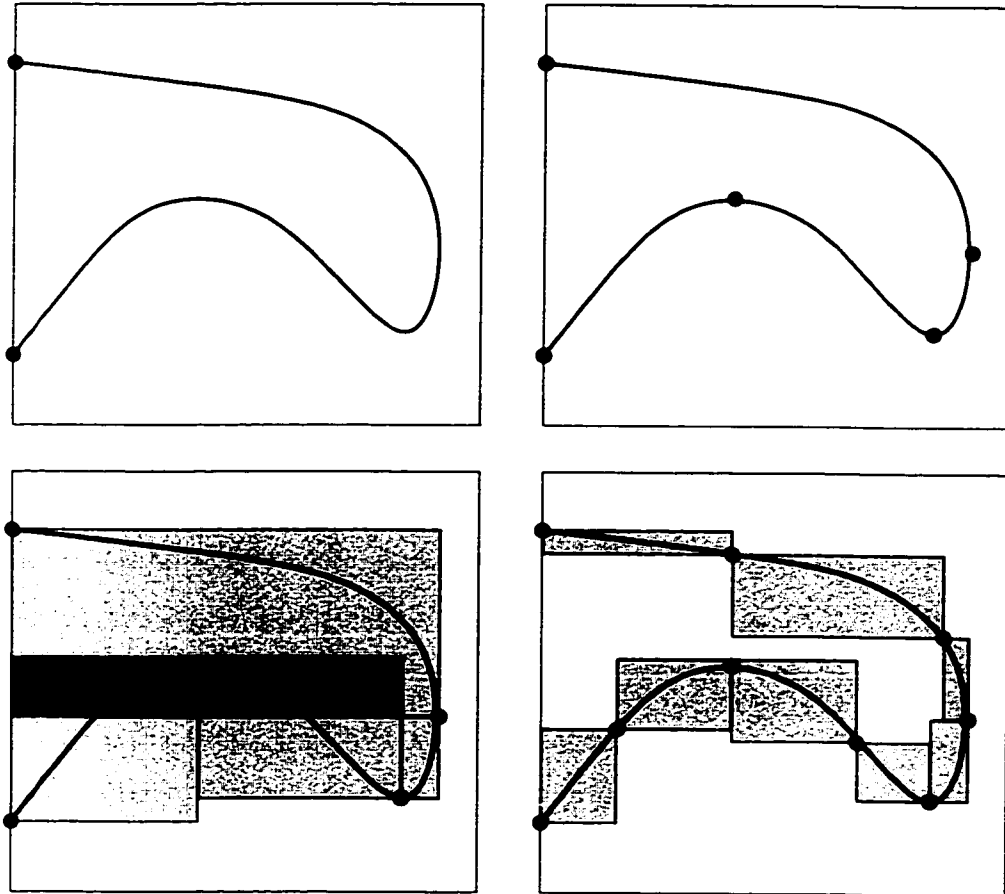


Figure 3.5: **Subdivision of a curve.** The curve in the upper left is divided into monotonic sections as shown in the upper right. The bounding boxes for the curve segments still overlap, as shown in the lower left, and so the segments are further subdivided until the bounding boxes no longer overlap, as shown in the lower right.

to contain both the starting point and ending point. Because the curve is monotonic within the segment, this box bounds the curve segment. The second restriction is that the bounding boxes for all curve segments associated with the same algebraic plane curve do not overlap. This ensures that no portion of the algebraic plane curve passes through the bounding box except for that individual segment. Figure 3.5 shows a curve represented as a single segment, broken up into monotonic segments, and further broken up into segments with nonoverlapping bounding boxes. Section 4.2 describes how an algebraic plane curve is broken up into valid segments.

Each curve segment, and thus the curve, is defined to be open at the starting point, and closed at the ending point. That is, a segment contains the ending point.



but not the starting point. Thus, there is no overlap between the segments. The segments partition the curve.

This representation is capable of storing closed curves, also called loops. For closed curves, the ending point of the last segment of the curve is the same as the starting point of the first segment of the curve.

This representation for curves is appropriate only for curves that are regular within the patch domain (singularities outside the patch domain do not matter). The sequentially connected segments do not allow representations of self intersections or point solutions, and it may not be possible to obtain non-overlapping bounding boxes around cusps. Modifications to the structure (such as allowing zero-length segments to represent isolated point solutions, and implementing a graph structure to handle self intersections) would allow singularities to be represented. Such modifications, however, make operations on the curves more complex.

### 3.2.2 Operations on Curves

All curves can be *reversed*. A curve is reversed by reversing the order of the segments in the curve, and swapping the starting and ending point of each segment. Reversing the curve merely changes the direction that the curve is considered to travel.

For closed curves, the distinction of first and last segment is arbitrary, so a closed curve can always be *rotated* to make a new segment the starting segment. Rotating the curve simply means to change the first segment of the curve to be the last segment, leaving the second segment to be the new first segment.

Having segments that are monotonic with no overlapping bounding boxes makes many basic computations efficient. Section 4.2 describes how an algebraic plane curve is broken up into segments. A few of the basic operations that this representation speeds up are as follows:

- **Curve-curve intersection:** Intersecting two curves is a common operation in the boundary evaluation algorithm. The segment bounding boxes can be used as a quick-reject test to avoid unnecessary computation. If the bounding boxes of the curves' segments do not intersect, the curves themselves do not intersect.
- **Point on a curve query:** Another common operation is determining whether a point lies along a given curve, when the point is already known to lie on the algebraic plane curve. That is, a determination must be made whether this point is on one of the segments that make up the curve. Since the individual

bounding boxes do not overlap any portion of the algebraic plane curve other than that segment, all that is needed is to classify whether or not the point lies in one of the bounding boxes.

- **Sorting points along a curve:** Another operation that arises in the boundary evaluation algorithm is sorting points along a curve. The points are already known to lie on the curve. To sort the points, first determine which segment's bounding box each lies in. Since the segments are ordered from start to end, the order of points relative to points on another segment is known. Within any one segment, the curve is guaranteed to be monotonic in both  $s$  and  $t$ . Thus, the points can be sorted within a segment based only on their  $s$  or  $t$  coordinate.

### 3.3 Points

Like curves, points are defined in two dimensions. In boundary evaluation, points arise from the intersection of algebraic plane curves. The coordinates of these points may be irrational algebraic numbers. This algebraic number can be represented as the unique real root of a polynomial within an interval, as described in Section 2.2.3.1.

It is important to distinguish *points* from *vertices*. A vertex is a point that is formed at the intersection of two trimming curves. A point refers to any two-dimensional point, and may or may not be a vertex.

#### 3.3.1 Point Representations

When points are defined implicitly as the intersection of two algebraic plane curves, i.e. as a common real root of a pair of bivariate equations, the point may be stored as a two-dimensional interval with rational bounds that isolates one root of the pair of equations. One method for guaranteeing that the interval isolates the root is multivariate Sturm sequences, as described in Section 2.2.3.4. Another method, presented in Section 4.1, isolates each coordinate of the point individually, then performs a test to verify that the combination of coordinates isolates the curve intersection. This second approach allows each coordinate to be treated separately as a single algebraic number. In either case, all interval bounds are stored as exact rational numbers.

This root-in-an-interval approach stores any algebraic number, rational or irrational. In many cases, one or both coordinates is a known rational number. One option is to continue to treat the number as before (i.e. as a root within an interval).

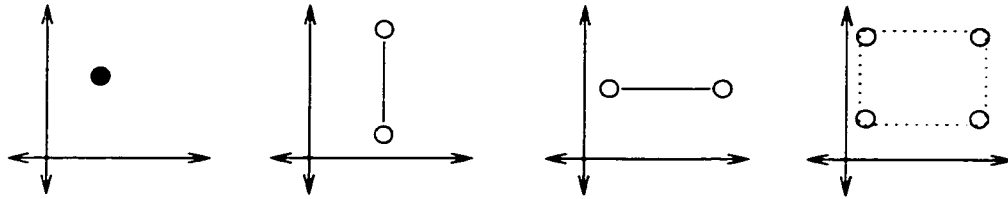


Figure 3.6: **Four ways of representing points.** From left to right: both coordinates known as rational numbers;  $s$  known as a rational number,  $t$  as the root of a polynomial within an open interval;  $t$  as a rational number,  $s$  as a root in an interval; as a root within an open two-dimensional interval.

but make the interval of zero width. All computations involving the coordinate then proceed exactly as they would if the interval were nonzero. Care must be taken that computations on the interval do not implicitly assume that the interval is of nonzero length. Although this approach is elegant, it may lead to significant time performing operations that are unnecessary. If rational coordinates are stored directly, they can be used directly in computations, making them significantly simpler. Storing rational coordinates directly as rational numbers requires a hybrid representation to be used for points, where each coordinate can be stored either directly as a rational number or generally as an algebraic number. This is illustrated in Figure 3.6.

### 3.3.2 Operations on Points

There are a number of basic operations involving points that need to be performed as part of the boundary evaluation algorithm. When coordinates are known as exact rational numbers, operations such as comparison in one dimension are straightforward. When a coordinate is known only as an algebraic number within an interval, the operations are more complex. Basic operations are needed to refine the interval to a smaller width so that appropriate comparisons can be made. Terminology to refer to these basic operations is introduced below.

In Sections 3.3.2.1 and 3.3.2.2, it is assumed that the points have been isolated using a method similar to that presented in Section 4.1. The key point is that the  $s$  and  $t$  coordinates are each known independently from each other. This makes it possible to deal with one coordinate as a single algebraic number, rather than having to consider the entire two-dimensional interval, as is necessary when roots are found using a method such as multivariate Sturm sequences (Section 2.2.3.4). A

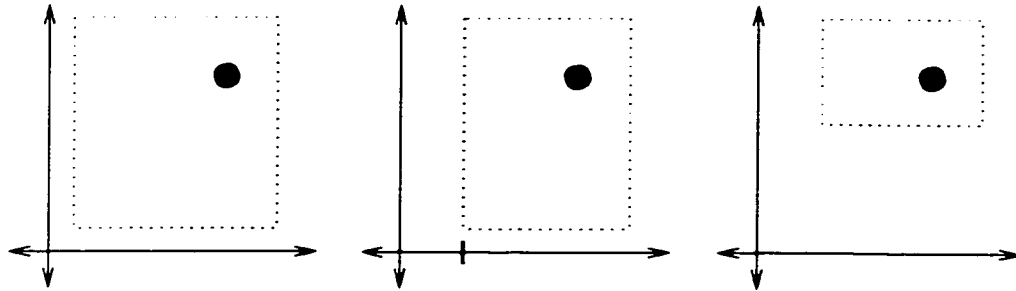


Figure 3.7: **A point being cut and halved.** At left, a point is given by a two-dimensional interval. The actual value is shown by the dot. At center, the point is cut at the  $s$  value indicated, yielding a smaller interval. At right, the point is further reduced by halving in the  $t$  direction.

brief mention of how these operations would be modified for a multivariate Sturm approach is given in Section 3.3.2.3.

### 3.3.2.1 Reducing Intervals

The most basic method of interval reduction is the *cut*. A cut operation in one dimension refines the bounding interval so that it lies on one side or another of a specific rational value, called the *cut point*. Cuts are always defined with respect to only one coordinate direction. When the cut point is defined as the midpoint of the starting interval, the operation is called a *halve*. A cut is performed by examining the Sturm sequence at the cut point (Section 2.2.3.2). Similar to the way that roots can be isolated by Sturm sequences, the Sturm value at the cut point determines which portion of the interval contains the actual algebraic number. If the Sturm value at the cut point is the same as the lower interval bound, then the upper portion of the interval is kept. Otherwise, the lower portion is kept. Figure 3.7 shows an example. Note that it is important to first check that the cut point is not the actual algebraic number being represented. This is done by checking whether the root's polynomial is zero at the cut point. If the cut point is a root, the overall point is represented using a rational number for the coordinate instead of the root of a polynomial. This may also allow the polynomial defining the other coordinate to be simplified.

A second basic operation is a *contract* operation. A contract reduces the bounding interval until it is no larger than a specified tolerance. When this tolerance is expressed in terms relative to the width of the starting interval (e.g. the new interval should be

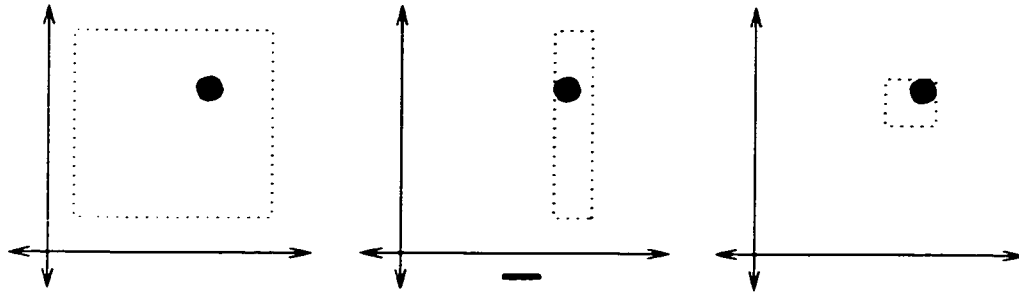


Figure 3.8: **Contracting and shrinking a point.** At left, a point is given by a two-dimensional interval. The actual value is shown by the dot. At center, the point is contracted in the  $s$  direction to a width indicated by the bar underneath the  $s$  axis. At right, the original point is shrunk to  $1/4$  its original size in both the  $s$  and  $t$  dimensions.

$1/10$  the size of the old interval), it is referred to as a *shrink* operation. A contract can be performed by applying a series of halve operations, until the width of the interval is below the specified amount. Since each halve operation reduces the interval by a factor of two, a shrink in one dimension by a factor of  $n$  (i.e. contracting the interval to a width  $1/n$  times the previous interval width) requires  $\lceil \log_2 n \rceil$  halve operations. An example of contracting and shrinking is given in Figure 3.8.

### 3.3.2.2 Comparisons

At times, two distinct points might have overlapping intervals. In order to compare them, it is necessary to *separate* the points. An efficient method for doing this is to first cut each point along the boundaries of the other point. After this, the two points either have nonoverlapping intervals, or have identical intervals. If the points have identical intervals, a shrink operation is performed on each, and the resulting intervals again checked for overlap. This process (cutting and then shrinking) is continued until the two points are separated. Figure 3.9 shows an example of the separation of two points. Once these points are separated, comparison operations can be easily performed. When a comparison is made in only one dimension (e.g. finding which point has the smallest  $s$  coordinate), the cutting and shrinking operations occur in just one dimension.

A final basic operation on points is checking for *equality*. Each coordinate is checked for equality independently. Checking for a coordinate's equality is done in the standard way for algebraic numbers: take the greatest common divisor of the

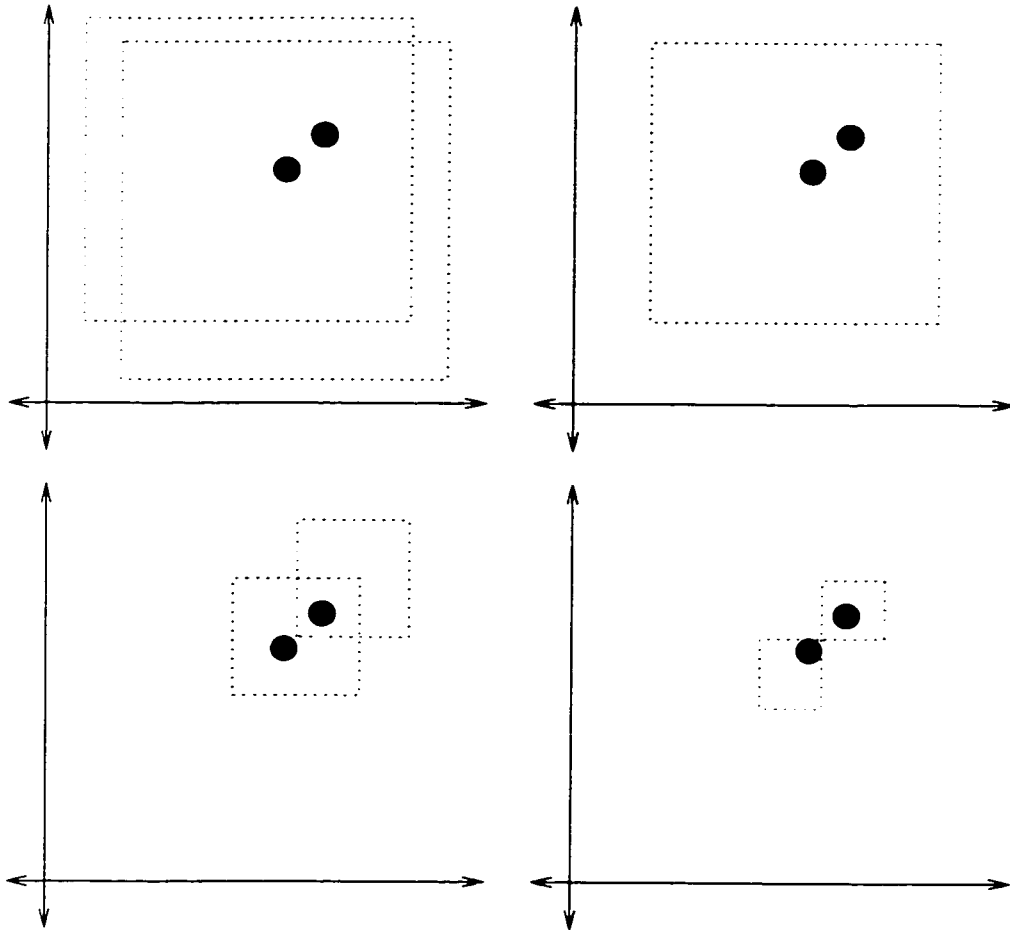


Figure 3.9: **Separating two points.** At upper left, the two points have overlapping intervals. Actual values are shown as dots. At upper right, each point is cut at the bounds of the other's interval. The intervals still overlap, so the points are shrunk, as shown at lower left. The intervals still overlap, so the process is repeated. At lower right, the next set of cuts has separated the two points.

polynomials of the two roots. then check to see whether there is a root of the gcd within the root's interval. These operations can be somewhat time consuming, due mainly to the gcd operation, so it is often more efficient to make a few attempts to separate the points (as described above) before checking for equality. Only after the points are known to be very close to each other is an equality test performed.

### 3.3.2.3 Operations on Points as 2D Intervals

The operations described become somewhat more complicated when the point does not have individual algebraic number representations for each coordinate. This is the case when points have been determined via a multivariate Sturm approach (Section 2.2.3.4). A cut operation, which is the basis for contracting and separating, involves determining Sturm values at two new points. Equality tests cannot be made as described above. Instead, an equality test as described in the next paragraph is used. Because this equality test can be extremely slow, it is especially important to attempt to separate the points before checking for equality.

Each of the two points is defined as the unique real intersection of two algebraic plane curves. Call the curve polynomials  $f_1$  and  $g_1$  for point 1, and  $f_2$ ,  $g_2$  for point 2. Note that one of the polynomials from point 1 may be the same as one of those from point 2. A straightforward approach to this problem would involve looking for simultaneous solutions to all four polynomials. Elimination theory (Section 2.2.2) can be used to achieve this, but such computations can be extremely slow when dealing with several polynomials. Instead, a different check for equality is used. Form a new polynomial,  $F = f_1^2 + g_1^2 + f_2^2 + g_2^2 = 0$ . Notice that since the coefficients of the four point polynomials are real, that  $F = 0$  can have a real solution if and only if  $f_1 = 0$ ,  $g_1 = 0$ ,  $f_2 = 0$ , and  $g_2 = 0$  at that same real solution. Next, find whether there is a real intersection of  $F = 0$  with one of the other polynomials, say  $f_1 = 0$ , within the interval. If there is an intersection, that implies that all four polynomials have a common intersection within that interval, and the two points must be equal. If there is not a common solution to  $F = 0$  and  $f_1 = 0$ , then the two points cannot be equal.

## 3.4 Topology

To this point, only geometry of the representation has been discussed. It is also necessary to represent the topology of the model. Unlike the geometric data, which to some extent has its representation defined by the nature of the problem itself,

there are many ways that the topological structure can be represented. One that has worked well in practice is presented here. The approach presented stores topological data within the patch and the edge. No global topological data structure is stored, although one can be created from the local data. Recall the assumption that the model to be represented is manifold. The topological information described here is appropriate for representing only manifold objects. Representation of non-manifold geometry requires more complex topological data structures and different geometric representations for curves.

First, a word must be said about *shells*. A shell refers to one connected set of boundary surfaces forming a closed manifold. The boundary of a simple object, such as a polyhedron, forms a single shell, while more complex objects may have many shells. For example, imagine a hollow ball. One set of surfaces (i.e. one shell) defines the outermost boundary of the ball, while another set of surfaces (i.e. a second shell) defines the inner boundary surrounding the hollowed out area. In some cases, the topological structure describing the nesting of shells is desired. In the representation presented here, topological shell information is not stored explicitly. Individual shells can be derived from the other topological data, and information on the nesting of shells can be derived from the geometric data, if desired.

Within each patch is an ordered list of trimming curves. These trimming curves are connected and form a counterclockwise loop around each face when viewed from the exterior of the object. The choice of which curve is first in the loop is arbitrary, but the sequential ordering of the curves forms the first piece of topological information. Along with each trimming curve is stored a pointer to the patch that is adjacent along that curve, called the *adjacent patch*. Note that the adjacent patch's surface might not be the same as the adjacent surface (Section 3.1) for that trimming curve. A restriction is made that a patch can not be adjacent to itself. For example, it may be possible to represent the entire curved portion of a cylinder as a single patch (Figure 2.5), but this would require that the patch be adjacent to itself. As mentioned in Section 2.1.2, it is always possible to subdivide a patch into two or more subpatches, and this can be used to create models such that patches are never adjacent to themselves. If no patches are adjacent to themselves before performing boundary evaluation, then no patch will be adjacent to itself in the output.

Within each curve data structure is a pointer to the *associated curve* in another patch domain. Corresponding to each trimming curve is a trimming curve in the domain of the adjacent patch. These two curves define the same curve in three



dimensions, and so a pointer from each to the other is used to store this information.

The topological information listed is sufficient to find all the necessary topological information. A listing of how various adjacency relationships can be derived from a model as presented here is given in Section 3.4.2.

### 3.4.1 Additional Data

For the sake of convenience in computation, some additional data is stored. Within curves, an indicator of the relative orientation of the two curves is kept. As mentioned earlier, each curve has an orientation given by the order of the points along it from the start to the end. Because the curves reside in different parametric domains, “forward” on one curve may correspond to either “forward” or “backward” on the other. Recall that the trimming curves for a patch are in counterclockwise order when viewed from the exterior of the model. This restriction ensures that associated curves must be oriented opposite to each other in the input and output models. In intermediate computations, however, curves may have the same orientation, so this information is stored in the curve structure.

Just as associated curves in different domains refer to the same curve, points in different domains may refer to the same point. Each point, then, may have an *associated point* in a different domain. Even though a point may have an equivalent representation in several other domains, only one associated point needs to be stored. Furthermore, within any one domain, there may be two separate representations for the same point. The ways that this can happen are seen in the discussion of the boundary evaluation algorithm. For this reason, a way to record that two points are coincident is necessary. There are numerous ways of doing this, including a global structure listing pairs of equal points, or by maintaining a circularly linked list of equal points. Like the orientation information for curves, all of this information is used only during intermediate computations.

A diagram outlining all of the data (both geometric and topological) stored is given in Figures 3.10, 3.11 and 3.12.

### 3.4.2 Sufficiency of Topological Data

The sufficiency of the topological information stored is shown here. To do this, it is necessary to show that all nine adjacency relationships can be derived from the information provided. To conform to standard notation, F is used to refer to

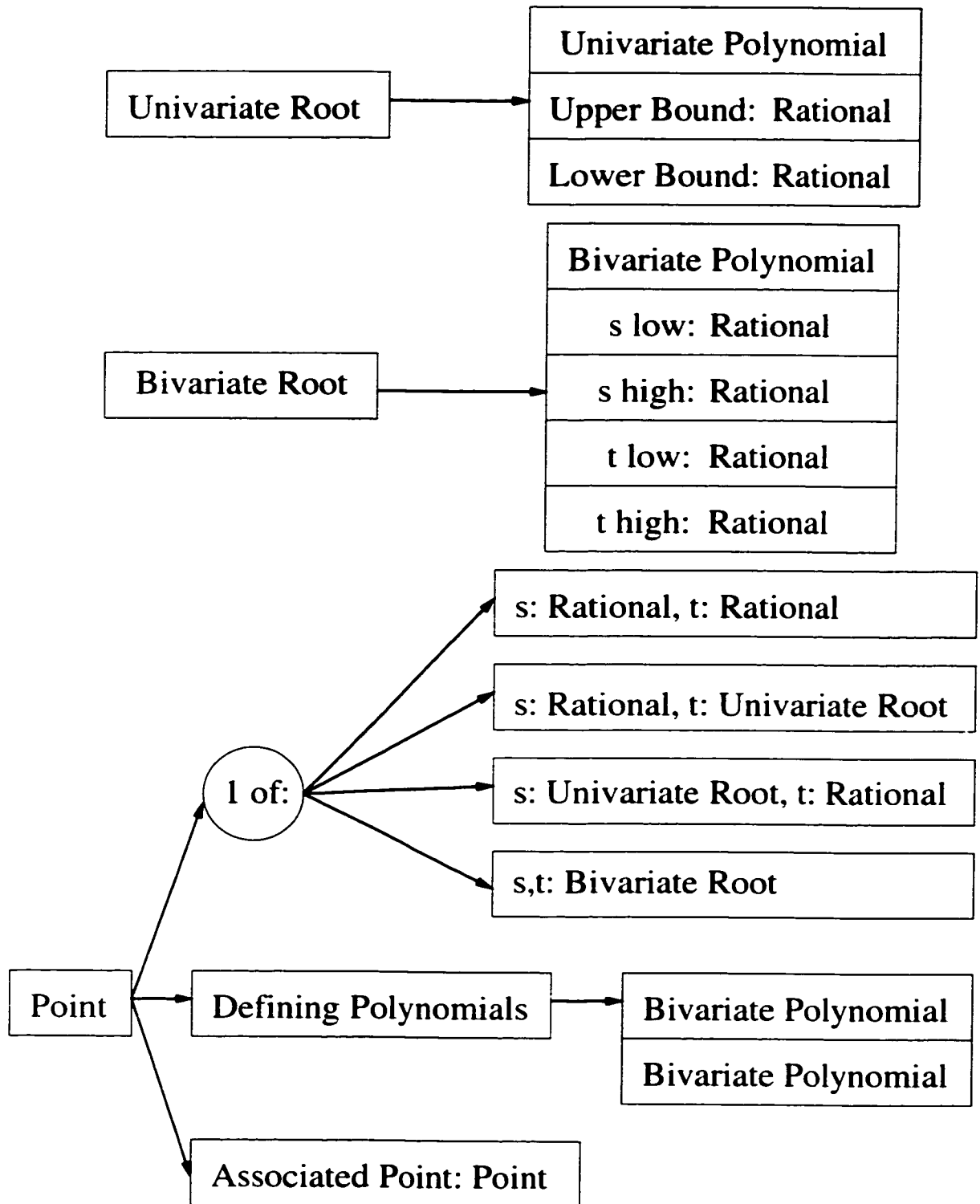


Figure 3.10: Univariate root, bivariate root, and 2D point data structures.

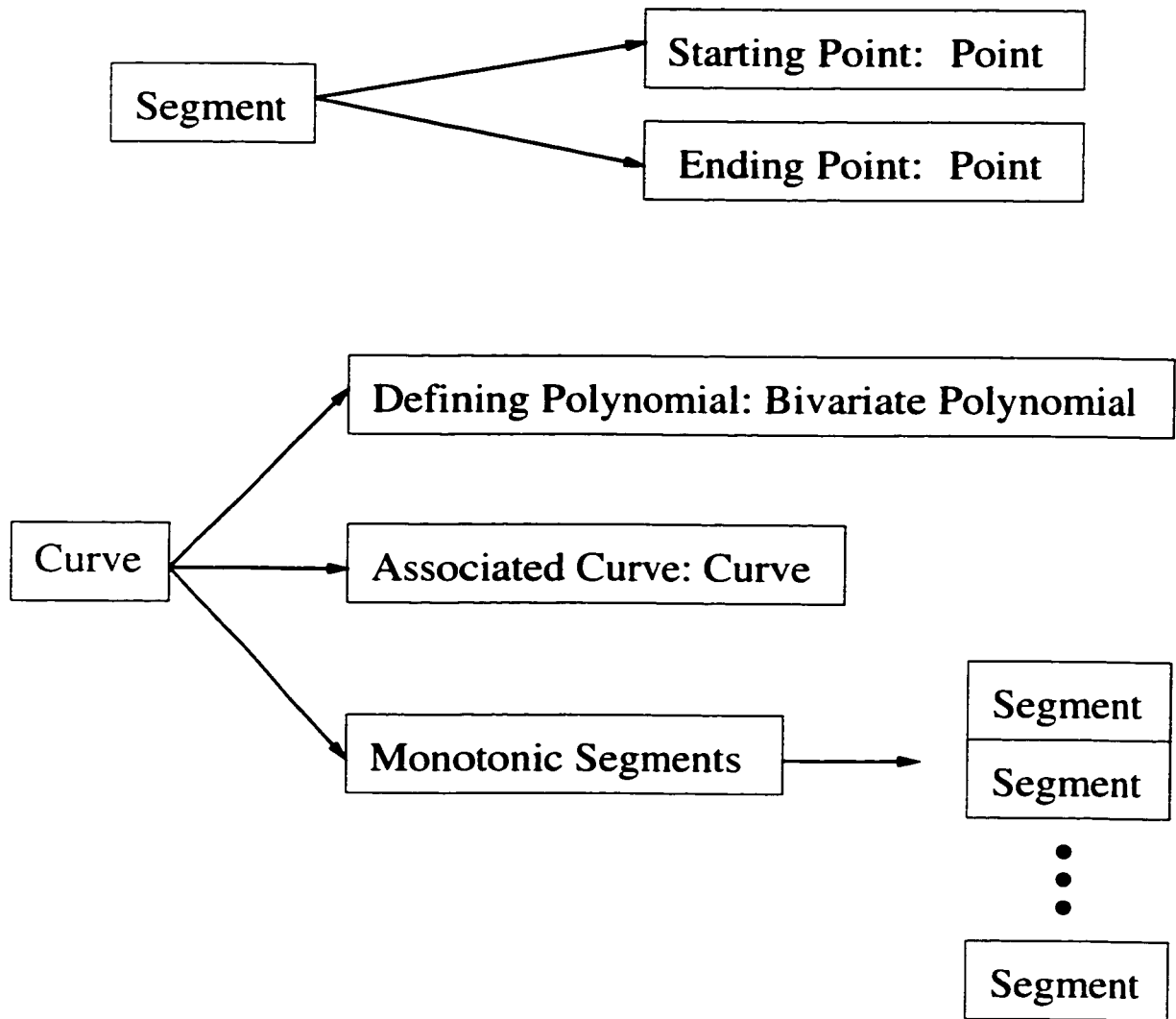


Figure 3.11: The data structures for a segment and a curve.

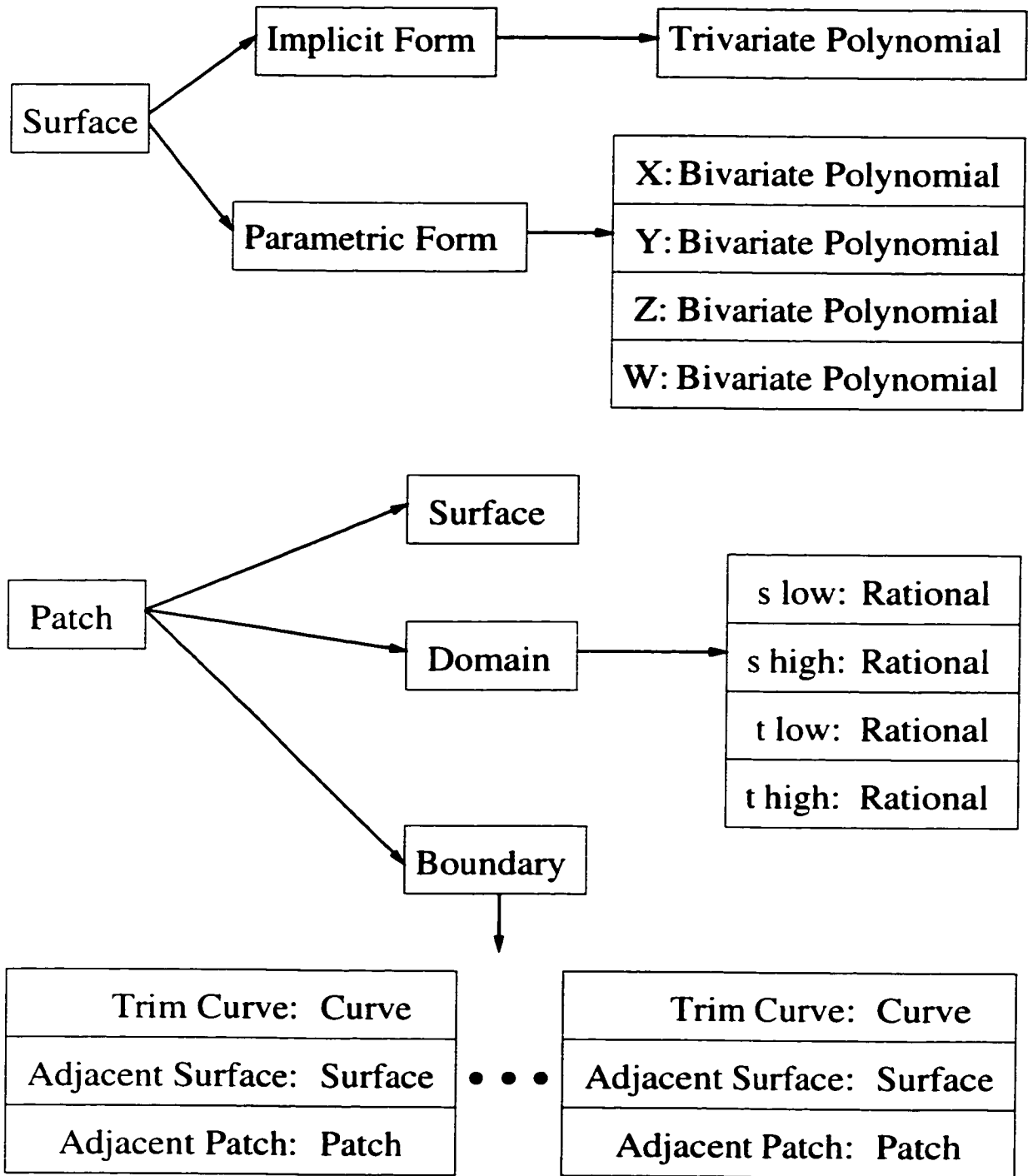


Figure 3.12: The data structures for a surface and a patch.

faces (patches), E to edges (the trimming curves), and V to vertices (points). An adjacency relationship is written as A-B, meaning that given a member of group A (one of {F,E,V}), find an ordered list of adjacent members of group B (again, one of {F,E,V}). For example, F-E means finding an ordered list of the edges adjacent to a face. Showing certain subsets of the nine adjacency relationships is enough to show full sufficiency [101]. All the nine relationships are specified here, however, to demonstrate the method for finding each type. It is assumed that a model is given as a collection of patches, as described above. The order of patches is not important.

Before listing the adjacency relationships, it is important to demonstrate that edges and vertices are specifically identified. Each face has a unique representation, but edges and vertices are defined within the domain of more than one patch. Thus, it must be shown that given one edge (or vertex), that all the other representations of that same edge (or vertex) can be identified. For edges, this is direct. Since we deal with only manifold solids, and since no patch is self adjacent, each edge is represented in exactly two patch domains. The association pointer stored with each curve identifies one, given the other. The identification of all equivalent vertices follows the same approach as the V-E problem, described below.

- **F-F, F-E, and E-V:** These relationships are all stored directly. The patch data structure keeps a circular ordered list of the adjacent faces and the trimming curves, which are the edges. The curve data structure keeps the starting and ending points, i.e. the vertices.
- **F-V:** This is derived directly from the stored F-E and E-V lists. Taking the ending point of each of the trimming curves gives the complete ordered list of vertices around a face.
- **E-F:** Each edge is a part of exactly two faces. For a given trim curve, the associated edge in another patch domain is found. Since each patch contains a list of all trimming curves, the patch list is then iterated through to find the two patches that contain the trimming curve and its associated curve.
- **V-E:** Each vertex is defined as the intersection of two trimming curves, and thus is at the beginning of one trimming curve and the end of another in one domain. Given a vertex,  $V$ , in one domain, the list of trimming curves for all patches is examined until a curve, call it  $A$ , having that point as an ending point is found. The succeeding trimming curve in that same patch, call it  $B$ , then,

has that point as a starting point. Take  $A$ 's associated curve, call it  $C$ , that is in another patch domain.  $C$ 's starting point,  $W$ , is the same as  $A$ 's ending point, in three dimensions. Notice that this is how the equivalent representations of  $V$  are found (i.e.  $V$  and  $W$  are the same vertex, just in different domains). The curve preceding  $C$ , call it  $D$ , will end on  $W$ . Continue with  $D$  just as was done with  $A$ . This proceeds around the vertex in order. Eventually,  $B$  will be reached, at which point all the adjacent edges will have been found in order.

- **V-F:** This is determined in the same way as the V-E case. As one marches around the edges, the patches are examined to determine which one contains each edge. Thus, the adjacent faces are listed in order around a vertex.
- **E-E:** This case is handled by using the V-E approach on both the starting point and ending point of the edge.
- **V-V:** Again, the V-E approach is modified to return this data. As each new edge, such as  $D$  in the description above, is found, a new vertex (such as the starting point of  $D$  in the description) is recorded. In this way, the vertices at the other end of all adjacent edges are recorded.

It is noted that certain of these operations (such as those based on V-E) can be complicated, however they do not regularly arise as part of the boundary evaluation algorithm outlined in Chapter 5. They are presented merely to demonstrate sufficiency. Furthermore, as is seen in later chapters, the running time of the exact boundary evaluation algorithm is heavily dominated by exact numerical computations. The efficiency of combinatorial computations, such as these topological queries, is of relatively little importance. If efficiency in these topological operations is desired, it is a simple matter to augment the topological data. For example, the curve structure can contain a pointer to the patch that it is a part of. This would reduce the time taken for most of the above adjacency queries.

### 3.5 Input Data

Input data must be convertible to the format described above in order for the boundary evaluation algorithm to be applied. Converting input objects in a B-rep format to the format described above is usually straightforward. More often, however, the input data is given in a CSG format, with the goal being conversion to B-rep.

Even among CSG systems, there can be many ways of representing data. For the purposes of this dissertation, a representative CSG-based system has been chosen to use as a basis for input. This is the BRL-CAD system [26, 24], developed at the Army's Ballistics Research Lab. As mentioned in the introduction, the BRL-CAD system has been used for several applications, and large, complex models have been designed with it.

Most input data comes in as floating-point numbers. It should be noted that any floating-point number has an exact representation as a rational number. For example, the number 0.53891 is equal to 53891/100000. Thus, any floating-point number input can be treated as an exact rational number. The implications of this are discussed below. Note also that it is possible, using methods such as continued fractions [59, 96], to generate close, lower precision, approximations to high precision numbers. For the purposes of discussion here, all input is converted exactly.

Before boundary evaluation is performed, the CSG primitives must be converted to the B-rep format just described. Because they include all the CSG standard primitives and can represent complex models, only four basic primitives are considered here: polyhedra, ellipsoids, generalized cones, and tori. Each of these can be represented exactly in the format described, and brief descriptions of these specific representations are provided in Appendix A. Other primitives can also be converted to the representation described.

Besides the primitives and Boolean combinations, CSG data also contains transformations. In BRL-CAD, the transformation data is stored in transformation matrices. Internally, BRL-CAD stores transformation data within the binary tree, rather than collapsing it to the leaves (Section 2.1.1 and Figure 2.2).

Transformation data can be applied directly to an object represented in the format described above. Applying a transformation matrix affects only the surface information in the patch description. All curves and points are defined within the patch domain, and are completely unaffected by a transformation.

A transformation matrix is a  $4 \times 4$  matrix, specifying the new  $X(s, t)$ ,  $Y(s, t)$ ,  $Z(s, t)$ , and  $W(s, t)$  relative to the original parametric form. The transformation matrix is applied to the parametric form of each surface, and the inverse transformation is applied to the implicit form (or vice versa, depending on how the transformation matrix is specified). If the transformation matrix is a combination of the common transforms (translation, rotation, scaling), it is invertible.

As stated before, floating-point data can be converted exactly to an exact ra-

tional representation. In many cases, however, the floating-point data itself is not exact. For example, a number such as  $1/3$  may have been truncated to 0.33333333. A transformation matrix intended to show a rotation by 30 degrees must contain some roundoff error, since the entries of such a matrix would be irrational (and thus not representable by floating-point or exact rational numbers). It should be noted, however, that it is possible to construct a rational number arbitrarily close to an irrational number, thus making it possible to represent a rotation by some number of degrees to any precision desired.

Even if all of the input data is specified exactly as the designer specified, it is unlikely that the designer's intent was to have an absolutely exact value. In the real world, it is not currently possible to specify physical measurements to infinite precision anyway, so an insistence that the designer's input measurements be determined exactly is unfounded. The reason the input data is still treated as exact is to ensure *consistent* computation. Consistency is achieved by dealing with all input in a uniform manner. Treating the input data as exact is just one way of achieving this consistency. Other methods can be used to try to better capture the design intent or to use less precision, but these approaches are outside the scope of this dissertation.

Note that converting an inexact B-rep into the exact representation presented earlier can run into problems if the error in the inexact B-rep causes that B-rep not to be a closed manifold. For example, the trimming curves might be inexact (due to approximation or roundoff error), leading to patches not meeting smoothly along an edge. Such conversion problems, which might be dealt with by not assuming exact input data, again, are outside the scope of this dissertation. With CSG data, since primitives are usually specified by a set of parameters (e.g. center and radius for a sphere), this is not a problem. Minor errors in the parameters only lead to slightly different objects, not invalid ones.

Finally, certain systems may incorporate further information into the model specification. For example, systems may list tolerances on measurements (a way of dealing with uncertain physical constraints), or material property values. Systems that can deal with this additional data are important, but this dissertation is concerned with the boundary evaluation problem only in reference to models as specified earlier in this chapter.



# Chapter 4

## Kernel Operations

This chapter describes some of the the fundamental operations that arise in the boundary evaluation algorithm. The methods presented here are useful for other geometric applications besides boundary evaluation. These operations, along with the basic operations on curves and points discussed in Sections 3.2.2 and 3.3.2, form the *kernel operations* that the overall boundary evaluation algorithm is built on. The specific ways that these operations are used in the boundary evaluation algorithm are described in detail in Chapter 5.

The methods presented in this chapter rely on the mathematical background material described in Section 2.2, as well as the representations (and basic operations on those representations) presented in Chapter 3. This chapter describes methods for intersecting two curves (Section 4.1), resolving the topology of an algebraic plane curve (Section 4.2), generating and locating points (Section 4.3), and finding implicit surfaces (Section 4.4).

### 4.1 Curve-Curve Intersection

Perhaps the most important operation in the boundary evaluation algorithm is that of curve-curve intersection. Specifically, given two algebraic plane curves, the goal is to find all intersections between the curves within a rational rectangle. A *rational rectangle* is defined to be an axis-aligned rectangular region of the plane whose bounds are given by rational numbers. That is, a region of  $\mathfrak{R}^2$  given by  $[a_s, b_s] \times [a_t, b_t]$  where  $a_s$ ,  $a_t$ ,  $b_s$ , and  $b_t$  are rational numbers. The coordinates of the intersection points are algebraic numbers, which may be irrational. Each algebraic plane curve is assumed to be regular (i.e. without singularities such as self-intersections) and is expressed as

the zero set of a bivariate polynomial with rational coefficients. Also, it is assumed that the curves do not have any common components (i.e. there are a finite number of solutions). In the boundary evaluation algorithm, curves are regular and will not have common components except in degenerate situations.

The specification of a rational rectangle comes from the fact that in most cases, one is only interested in roots within a patch domain. This allows roots that are outside of the rational rectangle to be ignored, which can simplify computation. In some cases, one needs to find all intersections between the curves over the entire real plane. Conservative bounds are available that specify the maximum absolute value of a real root of a polynomial (e.g. as in [21]). By using these bounds, a rational rectangle can be determined.

If one or both of the two algebraic plane curves comes from a curve divided into segments as described in Section 3.2, then an additional step is added. Once all intersection points between the algebraic plane curves have been found, each intersection is checked to see whether it is a part of the curve (Section 3.2.2).

#### 4.1.1 Previous Approaches

Finding 2D roots of a pair of bivariate equations is a well-studied problem, usually considered in the more general setting of finding roots of a set of  $n$  equations in  $n$  unknowns. One approach is to use worst-case bit length estimates to guarantee accurate results (e.g. [15, 105]). Another approach involves multivariate Sturm sequences (Section 2.2.3.4). Gröbner Basis methods are commonly used, particularly in general computer algebra systems (e.g. [21, 43]). Finally, a number of other approaches, including those based on interval arithmetic, eigenvalues, and curve subdivision (e.g. [94, 66, 2]) have been explored.

The approach presented here was developed specifically to work well for the boundary evaluation algorithm and the representations described earlier. Unlike some earlier methods, it is designed to find roots only within a rational rectangle. The input plane curves are not required to have a parametric representation, as some other methods require. Interval methods tend to have slower convergence than desired, and can have trouble distinguishing solutions when curves are nearly tangential. Because interval methods do not know a priori how many roots are in a particular region, they may refine only to a limited tolerance, and can potentially report spurious intersection points or merge two nearby intersections into one.

In the new approach, the output points are bounded by a rational rectangle.



as in the point representation (Section 3.3). Although multivariate Sturm sequences (Section 2.2.3.4) can provide the functionality desired, they suffer from two problems. First of all, multivariate Sturm operations are often too slow. The new algorithm, which replaces the multivariate Sturm calculations with a series of univariate Sturm calculations, is significantly faster in practice. Second, multivariate Sturm sequences can have problems whenever a root of the polynomials has one coordinate value the same as that of the rational test rectangle. Although such a condition is a degeneracy, it is usually of the unpredictable variety (Section 2.1.3.2), which makes it difficult to avoid. This type of unpredictable degeneracy can show up in practice, but the new algorithm is unaffected by it.

Although developed independently, the approach presented here is similar to an approach used by Sakkalis [85, 86]. Sakkalis' approach performs root classification by computing the Cauchy index over a region, and the new approach can be seen as an implementation of that same idea. Rather than ordering *box hits* as the new approach does, Sakkalis determines the sign of a ratio of polynomials at some of the box hits in order to classify points. The new approach eliminates some of the minor restrictions of Sakkalis' method (including certain boundary conditions), while Sakkalis' method provides more information about the point itself.

#### 4.1.2 Algorithm for Curve-Curve Intersection

This section describes the algorithm for finding common intersections of two algebraic plane curves, over a limited domain.

**Goal:** Given the algebraic plane curves:

$$\begin{aligned} f(s, t) &= 0 \\ g(s, t) &= 0 \end{aligned} \tag{4.1}$$

and a rational rectangle  $[s_1, s_2] \times [t_1, t_2]$ , return a set of points, as described in Section 3.3, such that each solution of  $f(s, t) = g(s, t) = 0$  that lies within the rational rectangle is described by exactly one point.

## Procedure:

1. First compute the polynomials:

$$\begin{aligned} S(s) &= Res_t(f, g) \\ T(t) &= Res_s(f, g) \end{aligned} \tag{4.2}$$

Notice that if  $f(\sigma, \tau) = g(\sigma, \tau) = 0$ , then  $S(\sigma) = 0$  and  $T(\tau) = 0$ . That is, the  $s$  coordinates of the common solutions of  $f = 0$  and  $g = 0$  are given by the roots of  $S$ , and the  $t$  coordinates are given by the roots of  $T$ .

2. Isolate the roots of  $S$  within the interval  $[s_1, s_2]$  and the roots of  $T$  within the interval  $[t_1, t_2]$ . This can be done using an exact univariate root finding method such as univariate Sturm sequences (Section 2.2.3.2). Each isolated root is represented either as an exact rational number (a *rational root*), or as an interval with rational endpoints (an *interval root*). Let the number of interval roots of  $S$  be  $m_1$ , the number of rational roots be  $m_2$ , and the total number be  $m = m_1 + m_2$ . Let the number of interval roots of  $T$  be  $n_1$ , the number of rational roots be  $n_2$ , and the total number be  $n = n_1 + n_2$ .
3. At this point, any solution of  $f = g = 0$  within the interval will have the  $s$  coordinate specified by one of the  $m$  roots of  $S$ , and the  $t$  coordinate by one of the  $n$  roots of  $T$ . All that remains is to take all  $mn$  combinations of coordinates and determine which, if any, correspond to a solution of  $f = g = 0$ . The  $mn$  combinations fall into one of four categories, depending on whether each root is a rational root or interval root. Three cases are handled quickly:
  - (a) Consider the  $m_2$  rational roots of  $S$  (call them the  $\sigma_i$ ) and the  $n_2$  rational roots of  $T$  (call them the  $\tau_j$ ). For all the  $m_2 n_2$  pairs of roots, check whether  $f(\sigma_i, \tau_j) = g(\sigma_i, \tau_j) = 0$ . If so, then  $(\sigma_i, \tau_j)$  is a root, and is a solution.
  - (b) Consider the  $m_1$  interval roots of  $S$ , and the  $n_2$  rational roots of  $T$  (call them the  $\tau_j$ ). For each of the  $n_2$   $\tau_j$ 's, do the following:
    - i. Compute  $\bar{f}(s) = f(s, \tau_j)$  and  $\bar{g}(s) = g(s, \tau_j)$ .
    - ii. Compute  $\bar{h}(s) = gcd(\bar{f}, \bar{g})$ .
    - iii. Determine whether  $\bar{h}(s)$  has any roots inside the intervals given by the  $m_1$  roots of  $S$ . Univariate Sturm sequences can be used to find any roots. Note that any roots of  $\bar{h}(s)$  must be solutions of  $f(s, \tau_j) =$

$g(s, \tau_j) = 0$ , and any of those roots within the interval  $[s_1, s_2]$  must be one of the  $m$  roots of  $S$ . Any roots found in those intervals give the  $s$  coordinate of a solution (the  $t$  coordinate is given by the  $\tau_j$ ).

(c) The  $m_2$  rational roots of  $S = 0$  and the  $n_1$  interval roots of  $T = 0$  are dealt with as in the previous case, with the coordinates reversed.

4. From this point on, consider only the  $m_1$  roots of  $S = 0$  and the  $n_1$  roots of  $T = 0$ , all represented as intervals. Form  $m_1 n_1$  rational rectangle boxes by combining the root-bounding interval solutions of  $S = 0$  and  $T = 0$ . Each box can contain at most one solution of  $f = g = 0$ .

An illustration of this is shown in Figure 4.1. In the figure, the two curves are shown, along with the intervals that bound roots of  $S$  and  $T$ . Notice that not all roots of  $S$  and  $T$  correspond to an intersection between the two curves within the rational rectangle of interest. The six boxes formed by combining the roots of  $S$  with those of  $T$  are illustrated.

5. At this point, it is still necessary to determine which (if any) of the boxes contain solutions of  $f = g = 0$ . This is done by intersecting each of the curves with the box boundaries.

For each of the  $m_1$  roots of  $S$ , we have a rational upper and lower bound:  $[l_i, h_i]$ . Substitute the lower bound to obtain  $\bar{f}(t) = f(l_i, t)$ . Using a univariate root finding method, determine which roots of  $\bar{f} = 0$  (if any) lie on the boundaries of one of the  $n_1$  boxes that  $l_i$  is a lower boundary of. These roots are called *f box hits*, since they represent the locations where  $f(s, t) = 0$  intersects the box. Likewise, substitute  $l_i$  in to  $g$  and find *g box hits*. Perform the same procedure for the upper bound,  $h_i$  in  $s$ , and then for the lower and upper bounds for the intervals in  $t$ . At the end of this process, all intersections of the curves  $f = 0$  and  $g = 0$  with the boxes has been found. In total, there are  $2(m_1 + n_1)$  univariate equations for which roots must be computed.

6. Finally, *classify* boxes as to whether or not each contains a solution. This is done by ordering the box hits around each box. Figure 4.2 shows some of the potential box hit configurations that can arise. Recall that the curves are assumed to be regular, and thus without self intersections, point solutions, or cusps. For now, also assume that intersections between the two curves are

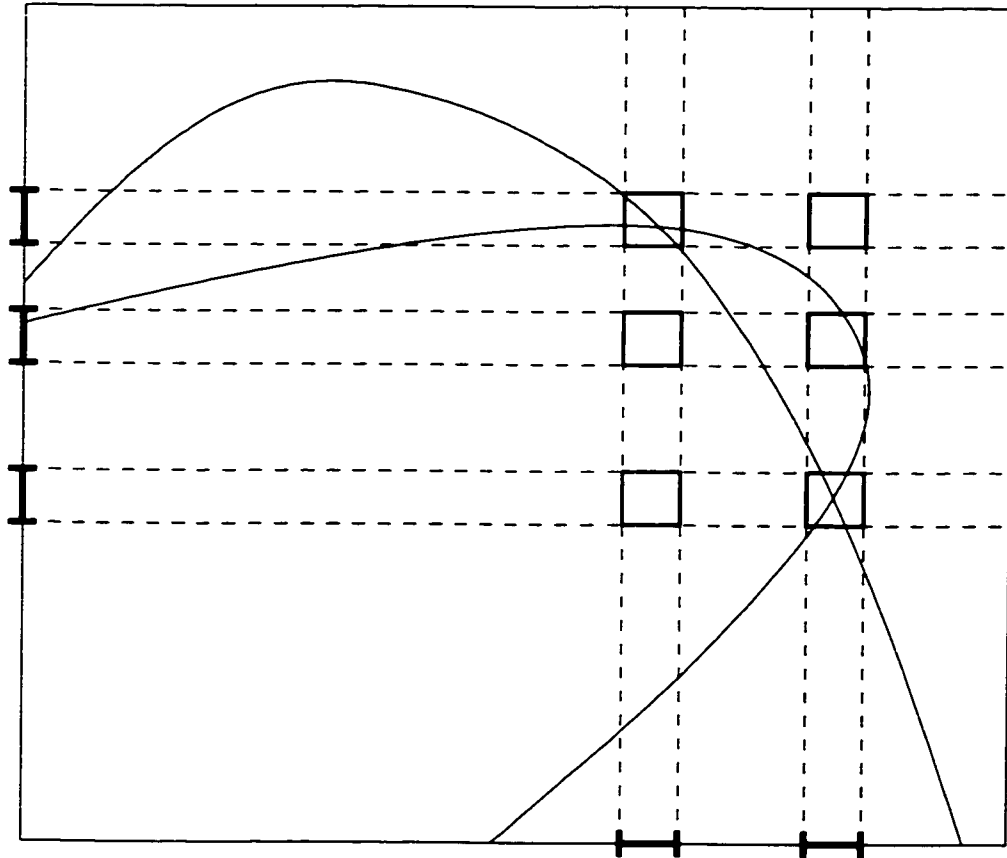


Figure 4.1: **Forming boxes that might contain roots.** Intersections between the two curves can occur only in these boxes. The curves are shown in the rational rectangle of interest. The root-bounding intervals found by isolating roots of  $S = 0$  and  $T = 0$  are shown on the horizontal and vertical axes. Combining the 2 roots in  $s$  with the 3 roots in  $t$  yields the six boxes illustrated in the image.

not tangential. Techniques to handle these special situations are discussed in Section 4.1.3. The possible cases are the following:

- (a) *No hits from  $f$  or no hits from  $g$* : Examples 4.2(a) and 4.2(b) illustrate this. At least one of the two curves does not pass through the box, or has an entire component within the box that does not intersect the other curve. Thus, there is no intersection in the box.
- (b) *Exactly one hit from  $f$  or  $g$* : Example 4.2(c) illustrates this. This means that the curve tangentially hit the box boundary but did not enter it. Thus there is no intersection inside the box.
- (c) *There are exactly two hits from both  $f$  and  $g$* : Determine the ordering (either clockwise or counterclockwise) of the four box hits. The pattern of hits may alternate between the two curves (i.e. an FGFG or GFGF pattern, where F and G describe whether the hit is from  $f$  or  $g$ , respectively). In this instance, as in Example 4.2(d), there must be an intersection inside the box. If the ordering of hits does not alternate (i.e. an FFGG, GGFF, FGGF, or GFFG pattern), then two cases are possible. The curves do not intersect (as in Example 4.2(e)), or the curves meet tangentially (as in Example 4.2(f)). In some applications, it is acceptable to ignore such tangential intersections entirely, in which case these non-alternating cases are never recorded as intersections. In other cases, such events are handled separately, as described in Section 4.1.3.3.
- (d) *There are at least two hits from one curve, and more than two from the other*: Examples 4.2(g) and 4.2(h) are examples of this. At least one of the curves is passing through the box more than once. Shrink the root  $S$  and the root of  $T$  that corresponds to this box, and recompute the box hits. At some point, the box will fall into one of the previous categories, and its status is determined conclusively.

In practice, cases 6b and 6d are very rare, as is case 6c when there is no intersection. Also, note that the tighter the bounds on the roots of  $S$  and  $T$  (i.e. the smaller the interval), the more likely it is that boxes without intersections inside will fall into case 6a.

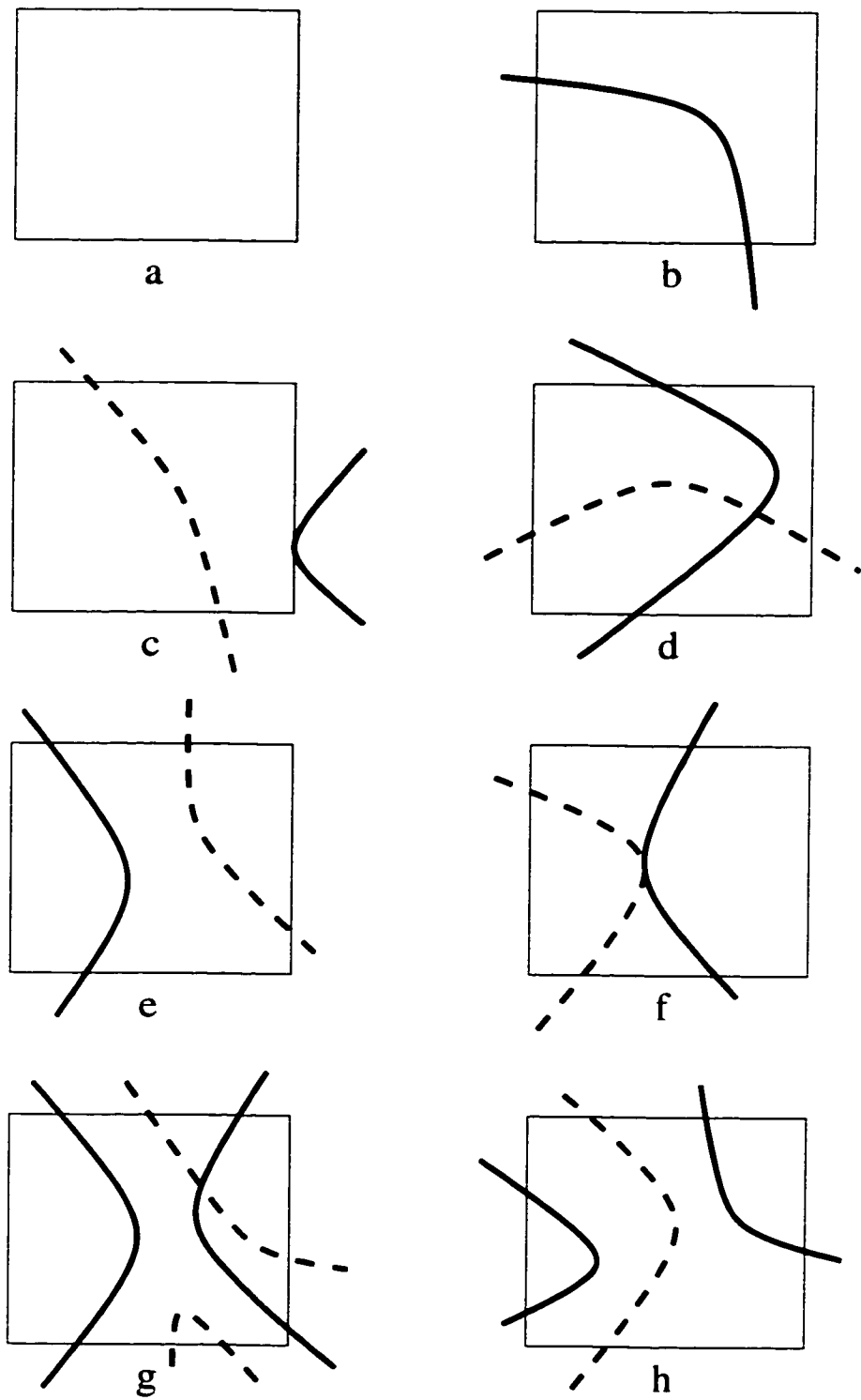


Figure 4.2: A few of the possible box hit configurations.



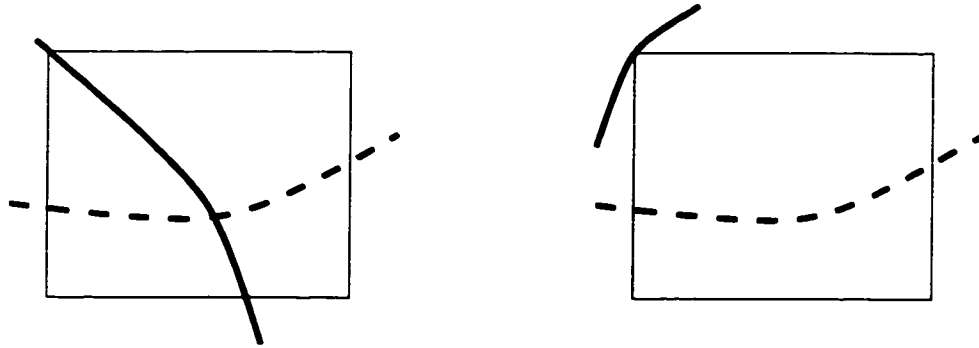


Figure 4.3: **A curve intersecting the corner of a box.** At left, a curve is entering a box at a corner. At right, it is only grazing the box.

### 4.1.3 Handling Other Cases

The algorithm presented in the previous section relies on several assumptions. This section discusses how to handle cases where these assumptions may not be valid.

#### 4.1.3.1 Box Hits At a Corner

It is simple to compute whether  $f = 0$  or  $g = 0$  intersects the corner of the box by substituting the coordinates of the corner into the equations. If a curve intersects a box at a corner, though, there is a question of whether the curve is entering the box or just grazing it. Figure 4.3 shows examples of each case. The question is answered by examining the sign of the slope of the tangent of  $f$  (or  $g$ ) at the corner. So, if the coordinates of the corner are  $(\sigma, \tau)$ , then we want to compute the sign of:

$$\frac{dt}{ds} = -\frac{f_s(\sigma, \tau)}{f_t(\sigma, \tau)} \quad (4.3)$$

The sign of the slope determines whether the curve is entering or grazing the box. If the gradient is horizontal or vertical, simply treat the box as in case 6d above (i.e. shrink the box and recompute box hits – the smaller box should no longer have an intersection at the corner).

#### 4.1.3.2 Many Tangent Box Hits

This is an extremely rare type of unpredictable (Section 2.1.3.2) degeneracy. Examples are given in Figure 4.4. Simply shrinking the box (by shrinking the associated

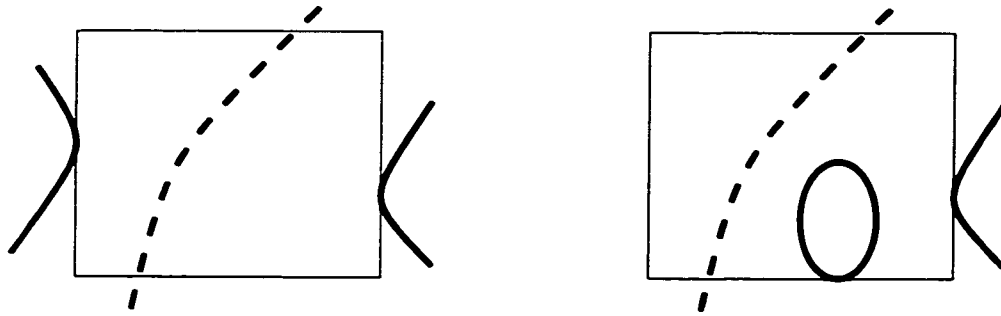


Figure 4.4: **Two cases of many tangent box hits by one algebraic plane curve.**

root intervals) would usually eliminate this problem, but the algorithm as described has no way of distinguishing this case from one where a curve is entering and leaving normally. One solution is to take one box hit from each curve on each box and make sure that that box hit is not a tangency. This is done by examining whether the tangent of  $f$  (or  $g$ ) is horizontal (or vertical, depending on which edge of the box is being tested) at the intersection. That is accomplished by examining whether the partial derivative with respect to  $s$  (or  $t$ ) has a zero at the intersection point. Testing whether a zero of the partial derivative coincides with the box hit is done via algebraic number comparison (Section 2.2.3.3). This is not difficult to implement, but it will decrease the overall efficiency of the algorithm. Because the case is so rare, implementing this solution might not be worthwhile.

#### 4.1.3.3 Tangent intersections

The algorithm assumes that intersections between the algebraic plane curves are transverse. When the curves have only a tangential intersection, the algorithm will fail to detect any intersection. In boundary evaluation, tangent curve intersections only arise as a result of input degeneracies, which in this dissertation are assumed not to occur. This information is provided to discuss how the algorithm will work in a more general setting. Figure 4.5 shows two examples of tangent intersections.

One type of tangency (at right in Figure 4.5) is when one or both of the curves does not intersect the box boundary. This can only occur if an entire closed component of the algebraic plane curve is inside the box. This is unlikely to occur if the boxes are small, and shrinking the box would likely cause the curve to intersect the box. The

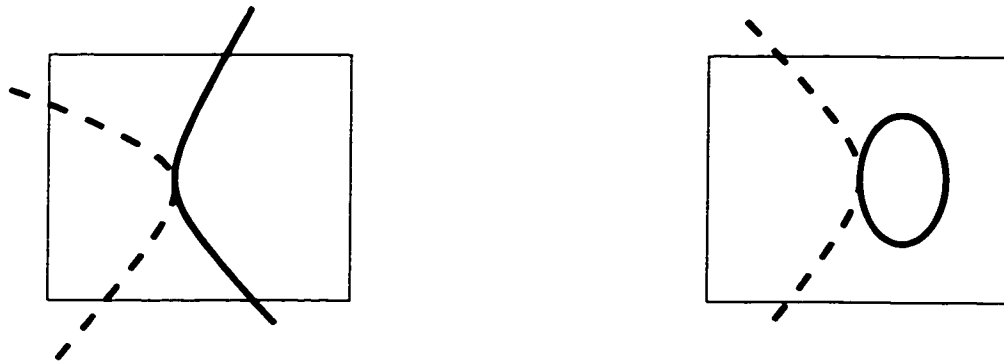


Figure 4.5: **Two types of tangent intersections.** At left, both curves intersect the box in a nonalternating pattern. At right, one of the curves is completely contained in the box.

only way to avoid circumstances like this is to verify that a curve that does not hit the box boundary is indeed not inside the box. A simple interval (or affine) arithmetic computation can usually verify that this is the case (i.e. that  $f \neq 0$  over the box). If the interval arithmetic operation fails to verify this, the box can be shrunk until eventually either the box is shown (by an interval arithmetic test) not to contain a component of  $f = 0$ , or the curve intersects the box. If there is no component inside, then there will be no solution inside the box. If, instead, the curve eventually hits the box, then the second type of tangency situation is encountered. Another possibility is, after a certain amount of box shrinking, to test for an intersection using a multivariate Sturm calculation (Section 2.2.3.4).

The other tangency situation (at left in Figure 4.5) is when both curves hit the box boundary, but do not form an alternating pattern. Usually, this is taken to mean that there is no intersection within the box. In order to rule out a tangent intersection, one solution is to use a method such as multivariate Sturm sequences (Section 2.2.3.4). Because these calculations can be slow, it is often useful to first shrink the box significantly before calling a multivariate Sturm routine. Multivariate Sturm sequence methods can also be avoided entirely if the box is shrunk to a small enough size to meet the gap theorem requirement [15].

#### 4.1.3.4 Non-Regular Curves

An assumption is made in this dissertation that all the algebraic plane curves are regular. Due to the nature of the boundary evaluation algorithm, any non-regular curves that arise are the results of input degeneracies. In fact, the representation for curves described in Section 3.2 is not capable of representing a non-regular curve. Still, the algorithm described is capable of intersecting non-regular curves, except when the actual point of intersection is at a singularity. Figure 4.6 illustrates four examples of intersections at a singularity of a nonregular curve.

For cusps, the ordering of box hits may not be relevant as to whether or not there is an intersection. Intersections at cusps where the ordering alternates are detected by the algorithm (Example 4.6(a)), while those where the box hits do not alternate (Example 4.6(b)) must be handled in the same way as tangential intersections (Section 4.1.3.3).

For curves with an isolated point component (such as  $s^2 + t^2 = 0$  has at  $(0, 0)$ ), there will be no box hits and the intersection is missed entirely (Example 4.6(c)). Similar to the isolated component in the tangential case, the box can be checked via interval techniques, and repeatedly shrunk if necessary. At some point, the box must be either tested with a multivariate Sturm method, or shrunk to a point that it is guaranteed to have a solution by the gap theorem [15].

For curves with self intersections, the box will be shrunk without stopping when the intersection point is at the point of self-intersection (Example 4.6(d)). The algorithm will (by step 6d) repeatedly shrink the interval since there will always be more than two box hits from one polynomial. The gap theorem [15] provides a limit for how much the interval needs to be reduced, but this is often much smaller than desired. Performing a multivariate Sturm test, after a certain amount of shrinking has occurred, is an alternate method for finding an intersection in this case.

## 4.2 Curve Topology

Curve topology refers to the decomposition an algebraic plane curve over a region of the plane. Often, curves are first generated as the zero set of some bivariate polynomial with rational coefficients. A curve topology algorithm allows one to understand the structure of the curve in a region, so that it can be represented and manipulated effectively. The curve topology algorithm presented here generates curves in the format described in Section 3.2 from an algebraic plane curve. Since a curve

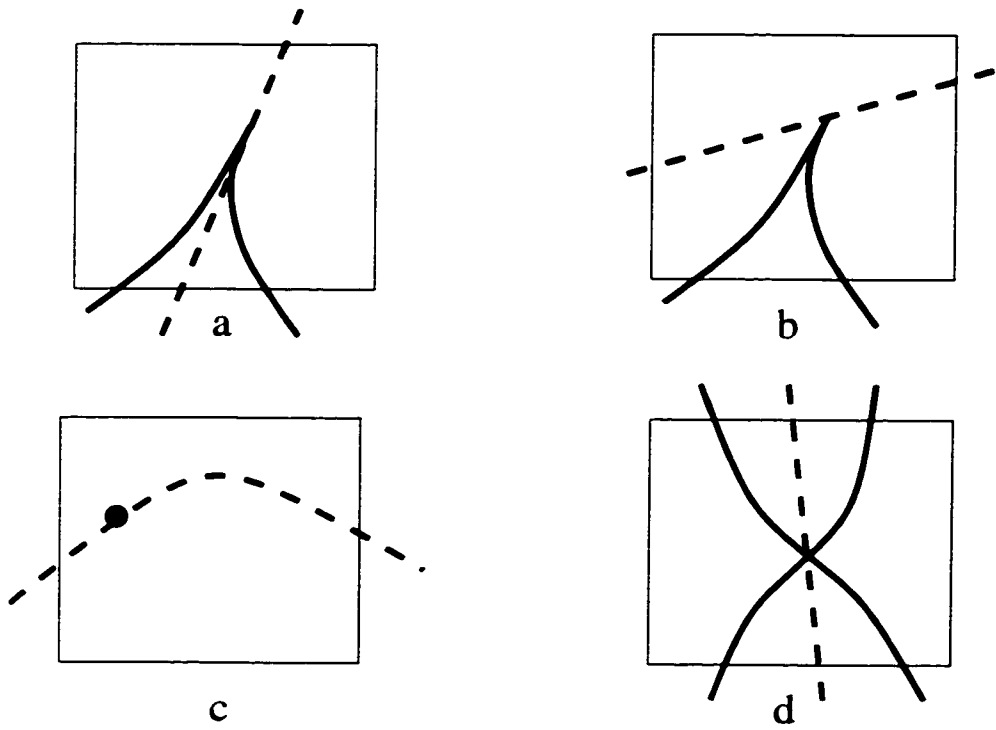


Figure 4.6: **Intersections at singularities.** In (a) and (b), two potential configurations when there is an intersection at a cusp. In (c), an intersection at a point solution. In (d), an intersection at a curve self-intersection.

topology algorithm is run for many, if not all, of the curves generated as part of the boundary evaluation algorithm, having an efficient method that works well with the representations described is important.

The specific problem being dealt with here is this: given a polynomial,  $f(s, t) = 0$ , and a domain,  $[s_L, s_H] \times [t_L, t_H]$ , decompose the curve into monotonic segments of  $f = 0$ , and record the connectivity between the monotonic segments. This process is called resolving the *curve topology*. The algorithm presented here is intended for regular curves, as are assumed within the boundary evaluation algorithm. Discussions of how the algorithm may be modified to work when there are singularities is provided later.

### 4.2.1 Previous Work

Analysis of curve topology has been extensively studied in algebraic geometry, symbolic computation, and geometric modeling. The curve topology algorithm presented here is closely related to the problem of finding a cylindrical algebraic decomposition for a curve, although the cylindrical algebraic decomposition is more general. An overview of the cylindrical algebraic decomposition is given by Arnon, Collins, and McCallum [7, 8]. An example of the use of the cylindrical algebraic decomposition in resolving curve topology can be found in another paper by Arnon [6]. An example of a way to compute the cylindrical algebraic decomposition for 2D curves is presented by Arnborg and Feng [4]. Some work has addressed the problem of interest here more directly. A paper by Kriegman and others [60] is one notable example, although that approach is not exact. Other methods for resolving curve topology include those described by Arnon and McCallum [5] and Sakkalis [86].

Although the algorithm presented here performs the same basic task as these previous approaches, it differs in a few key ways. Some previous approaches require computation over a region that is unbounded in at least one dimension [5, 86]. The algorithm proposed here allows both dimensions to be bounded, saving significant work. Some approaches rely on specific information being computed about the turning points. For example, Sakkalis [86] requires a Cauchy index computation for each point, and Arnborg and Feng [4] require that the point representation be amenable to a spatial subdivision scheme that does not work well with points defined as intervals. The algorithm presented here does not rely on a specific method for computing or representing turning points. Inexact methods, such as those based on curve tracing, have trouble finding small isolated components or correctly resolving topology when

two components are close together. The approach presented here, when exact point-finding methods (such as the ones described in this dissertation) are used, does not have such problems.

### 4.2.2 Algorithm for Resolving Curve Topology

This section describes an algorithm to break down an algebraic plane curve into chains of monotonic segments over a limited domain.

**Goal:** Given a polynomial,  $f(s, t) = 0$ , and a domain,  $[s_L, s_H] \times [t_L, t_H]$ , resolve the topology of the curve.

**Procedure:** The algorithm consists of two stages. First is a preprocessing stage that locates several points on the curve. The second stage is a recursive subalgorithm, that works on connecting the points identified in the first stage.

#### Stage 1.

1. *Isolate turning points:* The *turning points* are defined to be the points where the curve  $f = 0$  has a vertical or horizontal tangent within the domain of interest. The turning points are located by finding the intersections of  $f = 0$  with each of its partial derivative curves,  $f_s = 0$  and  $f_t = 0$ . Turning points can be isolated using the curve-curve intersection method described in Section 4.1. Any other method for isolating the turning points can be used, but the rest of the algorithm benefits from the planar subdivision that occurs in a method such as the one in Section 4.1. Turning points should be separated (Section 3.3.2.2) so that no two have overlapping intervals. Note that singularities, where  $f = f_s = f_t = 0$ , do not happen on regular curves, so turning points are always separable. While turning points often correspond to local maxima and minima in  $s$  and  $t$ , they may also be points of inflection. The rest of this algorithm does not rely on distinguishing among the various types of turning points.
2. *Isolate edge points:* The *edge points* are defined to be the intersections of the curve with the boundaries of the region. The edge points are further classified into one of four types:  $L$ ,  $R$ ,  $T$ , or  $B$  edge points, depending on whether they occur on the left, right, top, or bottom edge, respectively. For example, the  $T$  edge points would be found by computing roots of the equation  $\bar{f}(s) = f(s, t_H)$ .

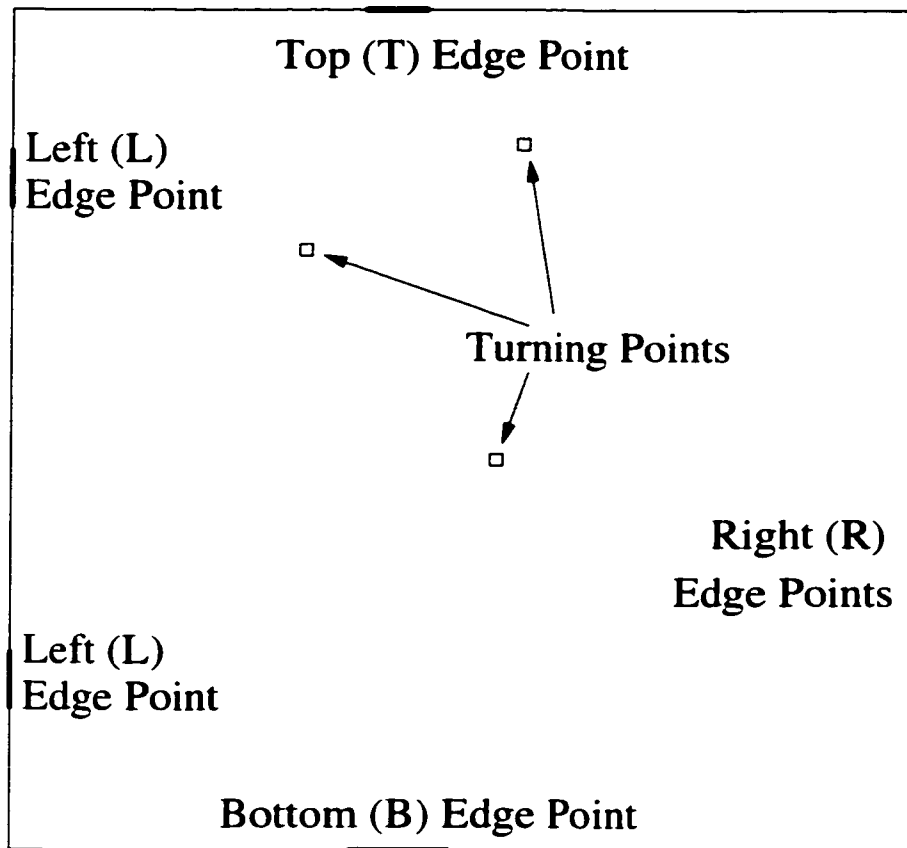


Figure 4.7: **The output of Stage 1 of the curve topology algorithm.** It is also the input to Stage 2. Shown are three turning points and six edge points that were found. Here, the edge points are shown as dark bars representing an interval, and turning points as boxes representing a 2D interval.

for  $s$  in  $[s_L, s_H]$ . Again, these points can be computed by any method and represented in any format. Univariate Sturm sequences and the representation of points described in Section 3.3 is one possibility.

The output of the first stage is a set of turning points, along with four sets of edge points. An example of this, where the points are represented as 1D and 2D intervals, is shown in Figure 4.7. This information is passed into the next stage of the algorithm. The curve topology has not yet been resolved since there may be many ways to connect the points that were computed.

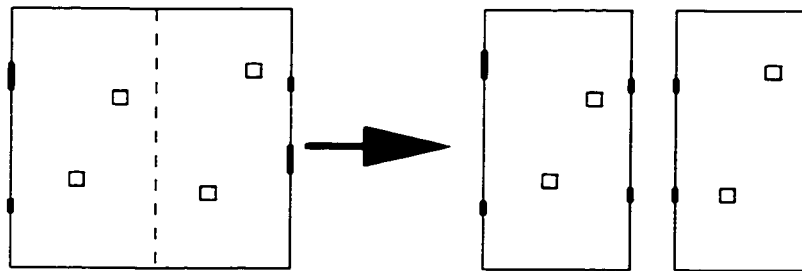


**Stage 2.** The second stage of the curve topology algorithm analyzes the pattern of turning points and edge points, and either determines all connectivity or subdivides and calls itself recursively.

For the purposes of this description, it is assumed that turning points are represented as two-dimensional intervals, and edge points as intervals in one coordinate and exact rational numbers in the other. Any representation that allows the points to be clearly separated from each other and to be cut and shrunk (Section 3.3.2.1) is sufficient. Representing the points as a unique number (e.g. as in floating-point) generally allows even simpler operations.

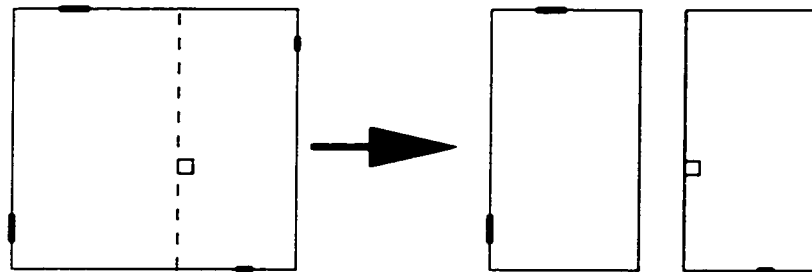
The input falls into one of the following cases:

1. *The region contains many turning points:* Find a horizontal or vertical line that will subdivide the turning points. It is always possible to find such a line, although in the worst case, turning points will have to be shrunk first. Subdivide the region along that line, finding intersections of the curve with the subdividing line. Then call recursively. The intersections with the subdividing line become edge points in the two subregions. The recursive call ensures that given  $n$  turning points originally, the region will be subdivided into  $n$  subregions, each containing exactly one turning point.

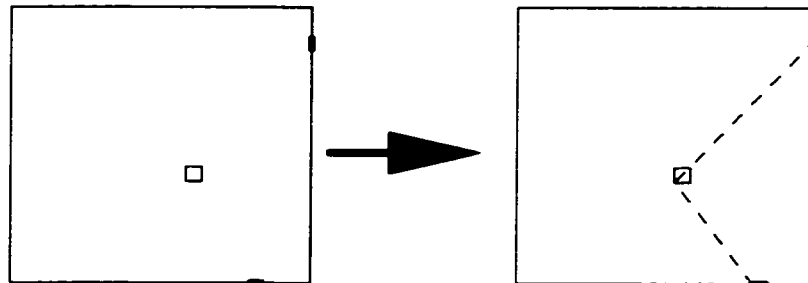


2. *The region contains one turning point and more than two edge points:* The region is subdivided along the boundaries of the turning point's interval. If, after all four interval boundaries have been used, the region still contains more than two edge points, shrink the interval of the turning point (Section 3.3.2.1) and continue subdividing. Note that if the algorithm in Section 4.1 is used to isolate the turning point, then this is never necessary. Eventually, the region will be subdivided into one region with a turning point and exactly two edge points, and other regions containing only edge points. It should be noted that Sakkalis [86] presents a method for determining connectivity in a case like this without

using subdivision, assuming that some additional information is associated with the turning point.

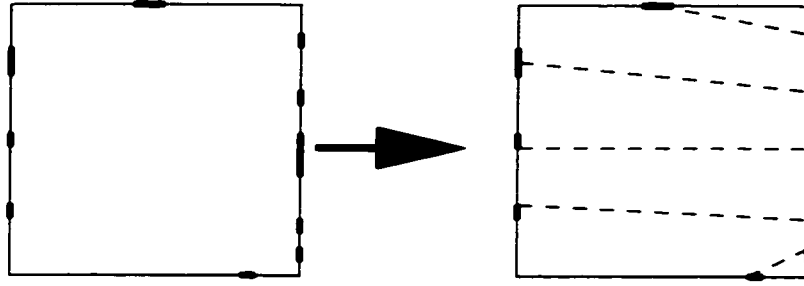


3. *The region contains one turning point and exactly two edge points:* First, note that if a region contains one turning point, it must contain at least two edge points, since the curve is assumed to be regular. The curve must pass from one edge point through the turning point and leave by the other edge point. Thus, the connectivity for this region is determined.

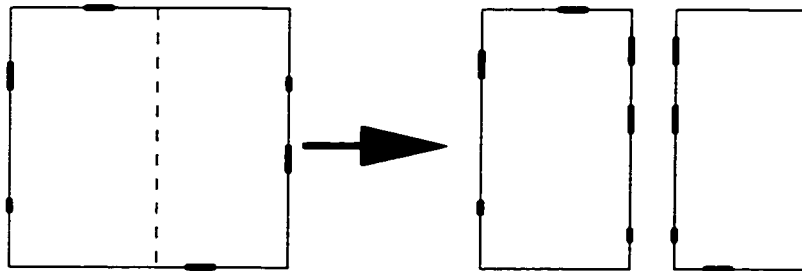


4. *The region contains only edge points:* Let  $numL$  refer to the number of L edge points,  $numR$  the number of R edge points, etc. Two cases are possible:

- (a)  $|numT - numB| = numL + numR$ , or  $|numL - numR| = numT + numB$ : Note that this includes all cases where there are no edge points along one of the edges. In this case, it is possible to connect the edge points to one another directly. Since there are no turning points in the region, and by assumption there are no self intersections in the region, the connections are straightforward.



(b) *Otherwise*: The region must be subdivided. If  $(numT + numB) < (numL + numR)$ , then subdivide with a vertical line that subdivides the T and/or B points, thus ensuring that  $(numT + numB)$  is smaller in the resulting subregions than in the current one. Otherwise, subdivide the L and/or R points with a horizontal line, ensuring that  $(numL + numR)$  is reduced in each subregion. Eventually, with one exception,  $(numT + numB)$  or  $(numL + numR)$  will be at most one, ensuring that the region falls into the previous case. The only exception to this would be when a truly horizontal or vertical line is involved. In such cases, either  $(numT + numB)$  or  $(numL + numR)$  may always be at least two, but the other sum would eventually be zero, again yielding the previous case.



The output of this stage is a connectivity between several points that breaks the curve into monotonic sections. This is either arrived at directly (cases 3 or 4a), or via subdivision (cases 1, 2, and 4b).

An example of the curve topology algorithm being performed on the input from Figure 4.7 can be found in Figures 4.8 and 4.9.

*Subdivision and degenerate cases*: Subdivision involves dividing the region by either a horizontal or vertical line, and recursively running Stage 2 on each portion. Subdividing a region may require certain edge points and turning points to be cut (Section 3.3.2.1). Also, new edge points must be generated along the cut line. This

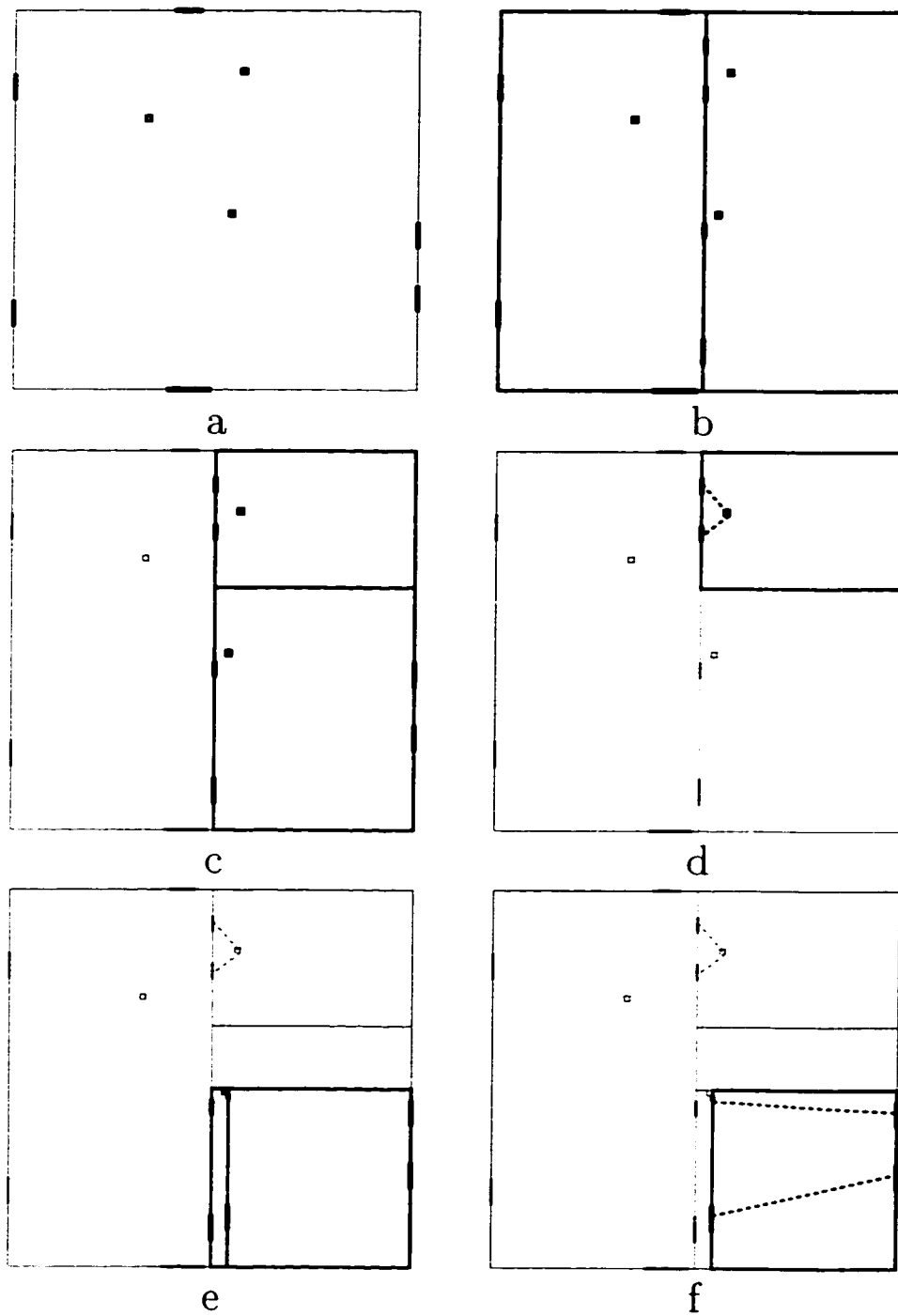


Figure 4.8: **Curve topology example.** In (a), the input to the algorithm. The region is subdivided, following case 1 in (b) and again in (c). In (d), case 3 determines the connectivity in one subregion. In (e), two applications of case 2 have further subdivided the regions. In (f), case 4a determines the connectivity of one region.

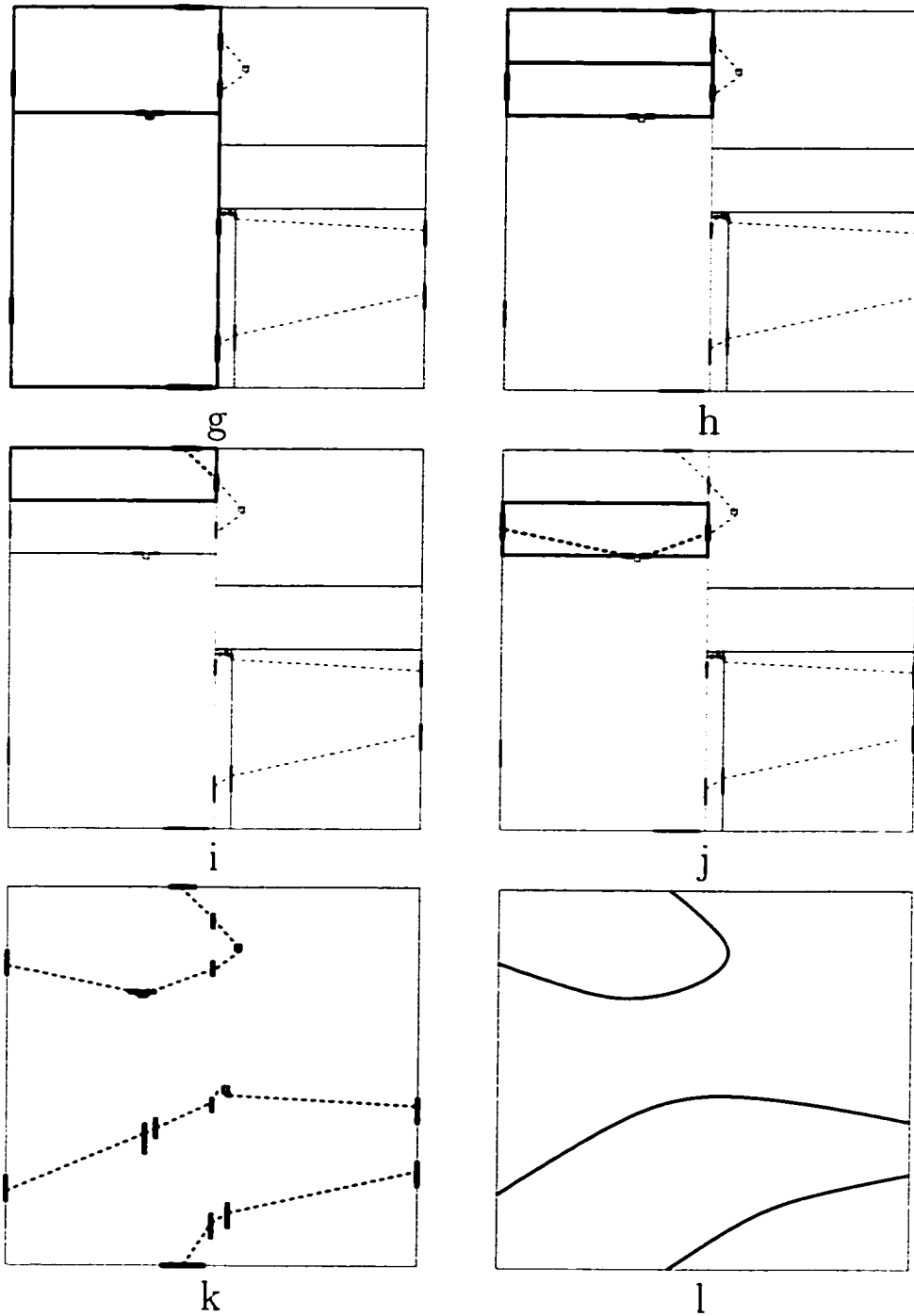


Figure 4.9: **Curve topology example continued.** State (g) is four steps after 4.8(f). (h) is subdivided by case 4b. In (i) and (j), the connectivity is found for each subregion. The algorithm continues until the final output, shown in (k), is found. (l) is the actual algebraic plane curve on which the curve topology algorithm was run.

involves finding the roots of a single univariate equation, just as with finding edge points in Stage 1.

It is possible (though unlikely) that an edge point may lie exactly at the corner of a region. In such cases, it is necessary to examine the sign of the slope of the curve at that point, as in the curve-curve intersection algorithm (Section 4.1). The point is assigned to one of the two subregions based on that slope. Edge points at a corner may be treated as a part of either (but not both) of the two adjacent edges.

Notice that since all turning points are isolated, there is never a tangential intersection along the border of a region, assuming the point is represented as an interval. If the turning point is a single number (such as a floating-point number), it may lie on an edge as a result of step 2. In such a case, a subdivision with a line parallel but slightly (i.e. less than the distance to the nearest edge point) offset from the turning point (instead of the original subdivision) allows that connectivity to be determined conclusively.

**Non-regular curves:** The algorithm as described is capable of handling only regular curves. Extending it to handle curves with singularities is possible, although a representation for non-regular curves must be used. The representation described in Section 3.2 is not sufficient. Handling non-regular curves involves identifying and handling the singular points individually. Singular points are places where  $f = f_s = f_t = 0$ , and are found in Stage 1 of the algorithm. The singular points fall into three categories: isolated point components, cusps, and self-intersections. The type of singularity needs to be identified.

Point solutions require no connectivity information since they are, by definition, isolated. No additional work needs to be done.

Cusps can be treated the same way as turning points, as far as topology. It is important to treat cusps as a single turning point, rather than two turning points (as might arise if  $f = f_s = 0$  and  $f = f_t = 0$  are isolated completely independently). Also, it may not be possible to ensure that the bounding boxes of segments near a cusp point are non-overlapping.

Self intersections occur where a curve has three or more incident branches. These should be treated as a separate type of turning point. Instead of terminating when there are two edge points (case 3), termination occurs when there are  $n$  edge points, where  $n$  is the number of branches leaving that point. It is necessary to classify the number of branches at the point ahead of time.

### 4.2.3 Non-Overlapping Bounding Boxes

In order to fit into the representation described in Section 3.2, the monotonic segments of the curve must have non-overlapping bounding boxes. Often the monotonic sections already have non-overlapping bounding boxes, but in some cases it is necessary to further subdivide the monotonic sections to meet this criterion. Subdivision occurs by locating more points on the curve and inserting those points into the curve (Section 3.2.1) thus creating new, smaller, segments.

Numerous heuristics can be used to subdivide the segments with overlapping bounding boxes, and it is unlikely that any one will be ideal in all circumstances. One that has worked well in practice for a pair of segments is to subdivide each segment at the boundaries of the box surrounding the other segment. For example, if segment  $A$  with bounding box  $[a_1, a_2] \times [a_3, a_4]$ , and segment  $B$  with bounding box  $[b_1, b_2] \times [b_3, b_4]$  have overlapping bounding boxes, then subdivide  $A$  along the lines  $s = b_1$ ,  $s = b_2$ ,  $t = b_3$ ,  $t = b_4$ , and  $B$  along the lines  $s = a_1$ ,  $s = a_2$ ,  $t = a_3$ ,  $t = a_4$ . Each line has at most one intersection with the segment, and finding that intersection point involves only a univariate root isolation.

## 4.3 Point Generation and Location

A basic operation that comes up in the later stages of the boundary evaluation algorithm is *point location* in two and three dimensions. Point location is the process of determining whether a point lies inside or outside a closed loop of curves in 2D, or a closed solid formed by patches in 3D.

Local information regarding which side of individual curves a point lies on is not enough to classify the point with respect to a loop of curves. An example is shown in figure 4.10. Instead, *ray shooting* is used. Ray shooting also forms the basis for point generation.

*Point generation* is the process of finding a 3D point on the surface of a trimmed patch. It comes up in a later stage of the boundary evaluation algorithm. Since points with rational coordinates are easier to work with (i.e. computations are far more efficient) than those stored as intervals, an additional requirement is added that the point have rational coordinates.

Point generation is discussed below, followed by discussions of point location in two and three dimensions. For the point location problem, locating points stored as rational numbers is described, followed by locating points stored as intervals. For

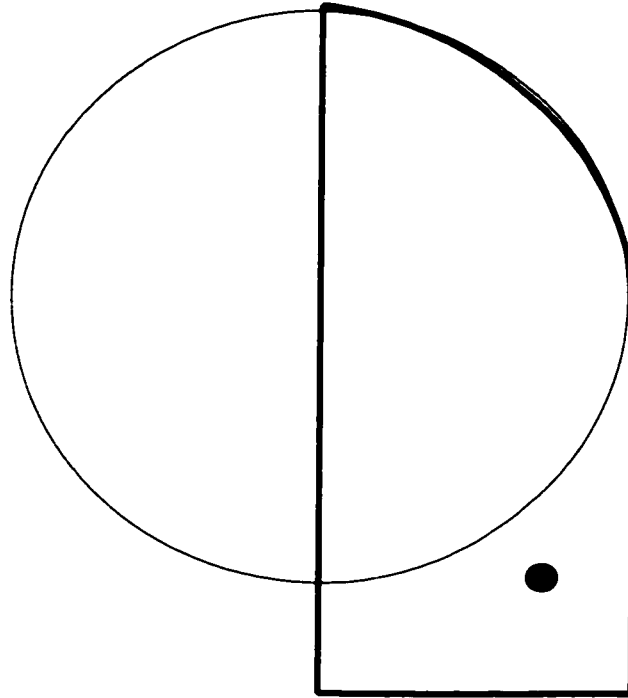


Figure 4.10: **The difficulty of classifying a point with respect to curves.** A loop of curves is defined by the heavy black lines. The point (shown as a solid dot) lies inside the loop of curves, but is outside of the circle that defines one of the curves. Performing point location relative to the circle alone provides no direct indication of the point location relative to the loop of curves.



three dimensions, point location for points with rational coordinates is described, as that is the only type that arises in the boundary evaluation algorithm. Extending the 3D case to handle points as intervals follows the same process as in the 2D case.

### 4.3.1 Point Generation

For point generation, the first step is to generate a point with rational coordinates, that lies within the patch's loop of trimming curves. To perform this computation, intersect a horizontal (or vertical) line with the trimming curves. Any horizontal line that intersects the trimming curves at least twice is acceptable. To find such a line, simply compute the smallest and largest points in  $t$  along the trimming curves. Note that since the trimming curves are made of monotonic segments, these points must be the endpoints of some curve segments. Take the midpoint (actually the midpoint of the rational bounds)  $m_t$  in  $t$  of the largest and smallest points, and let  $L$  be the horizontal line  $t = m_t$ .  $L$  always intersects the loop of trimming curves, except in extremely rare cases where the intervals bounding the maximum and minimum points are large relative to the loop of curves. In that unlikely event, simply reduce the size of intervals around the maximum and minimum points and choose a new line.

$L$  should not intersect the curves tangentially. It is easy to detect such tangential intersections, and choose a different  $L$  if necessary. Since the segments of the curves are monotonic, any tangential intersection occurs at a segment endpoint (where there is a local maximum or minimum in  $t$ ). If these turning points are marked as such, then the tangential intersection is easily detected. A second option is to assume that any time  $L$  hits a segment endpoint,  $L$  hits it tangentially. In either case, create a new line close to  $L$  by choosing a  $t$  value other than  $m_t$  (e.g.  $5/8$  the distance between the maximum and minimum points). Also, in the unlikely event that one of the trimming curves is a horizontal curve at  $m_t$ , choose a new  $L$ .

Intersect  $L$  with all of the trimming curves to find a positive even number of intersections. Order these intersections from the lowest  $s$  value to the highest,  $(p_1, p_2, \dots, p_n)$ . For any  $1 \leq i \leq n/2$ , consider the points  $p_{2i-1}$  and  $p_{2i}$ . The region between these two points must be inside the trimmed region. Each of these points'  $s$  coordinate is either a rational number or a number bounded by a rational interval. Take the upper bound of  $p_{2i-1}$ 's interval (or the rational coordinate itself), and the lower bound of  $p_{2i}$ 's interval (or the rational coordinate itself) and find the midpoint,  $m_s$ . Assuming the original intervals do not overlap (the points should be separated if they do), then  $(m_s, m_t)$  is the point inside the patch. Figure 4.11 shows

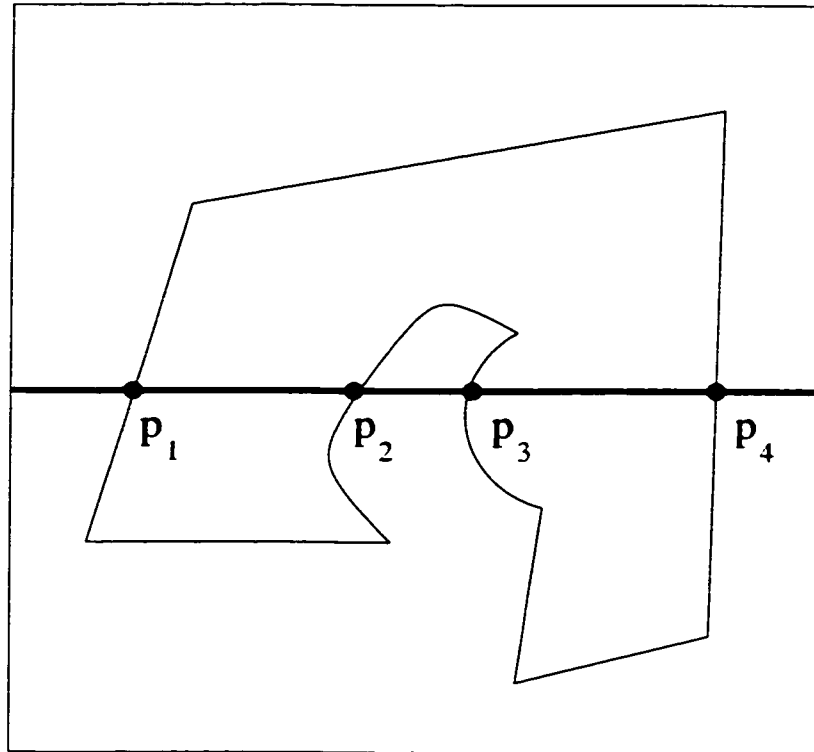


Figure 4.11: **An example of point generation.** The thin lines show the trimming curves, and the thick line shows the horizontal test line. Intersections of the test line with the trimming curves are shown by dots. Any point chosen between  $p_1$  and  $p_2$ , or between  $p_3$  and  $p_4$  is within the trimmed region.

an example of such a case.

Given a point with rational coordinates that is inside the trimmed region of a patch, raise it to three dimensions by substituting the values of  $s$  and  $t$  into the parametric equations. Assuming that the parametric equations have rational coefficients (and that the point is not a base point, where the  $w$  coordinate is zero), the 3D point is a point with rational coordinates that lies on the patch's surface, not at a boundary.

### 4.3.2 2D Point Location

Point location in two dimensions is similar to point generation. First point location when the point has two rational coordinates is described, followed by point location when the point is represented using a 1D or 2D interval.

#### 4.3.2.1 Points with Rational Coordinates

To locate points with rational coordinates, shoot a ray from the point, and count how many times it crosses the loop of curves. An even number of crossings means that the point is outside the loop, an odd number, inside. Because any ray should be sufficient, use a horizontal (or vertical) ray for ease of computation. Assume that the line is horizontal. Also assume that the loop of curves is bounded in the region  $[s_L, s_H] \times [t_L, t_H]$ . For a point at  $(a, b)$ , ray shooting means finding all intersections of  $t = b$  with the curves over either the region  $a < s \leq s_H$  or  $s_L \leq s < a$ .

Tangential intersections are computed in a manner similar to that in point generation (Section 4.3.1). If a tangential intersection is found or is possible, then a different ray is used. In the unlikely event that all four horizontal/vertical rays have tangent intersections, use random rays in any direction. Since such rays are not horizontal/vertical, finding intersections involves slightly more work than a simple substitution followed by univariate root finding.

#### 4.3.2.2 Points stored as intervals.

The situation becomes more complicated when the point is represented as an interval. The basic idea is to check the values at all four (or two) corners of the interval, and see whether all four corners are on the same side of all curves. Since the corners of the interval have rational coordinates, testing any one corner proceeds as above. The difficulty with this is that it is possible to have a point lying on one side of a curve, when all four corners of the interval bounding the curve are on the other side. Figure 4.12 shows an example of such a case.

This difficulty is resolved by the restriction of monotonic segments in the curve representation (Section 3.2.1). First, reduce the interval until it lies in the bounding box of at most one segment in each curve. This usually means cutting the point (Section 3.3.2.1) by the values of the bounding box for each segment within which the point might be contained. Then classify the interval corner points. If the point is located inside the bounding region of at most one segment of the curve, and if the four (or two) corners of the interval are all on the same side of the curve, then the entire interval is on the same side of the curve, and thus the point must be on that side.

Note that if a point does not lie in the bounding box of any segment in the curve, then locating any one point in the interval (e.g. one of the corner points) is sufficient

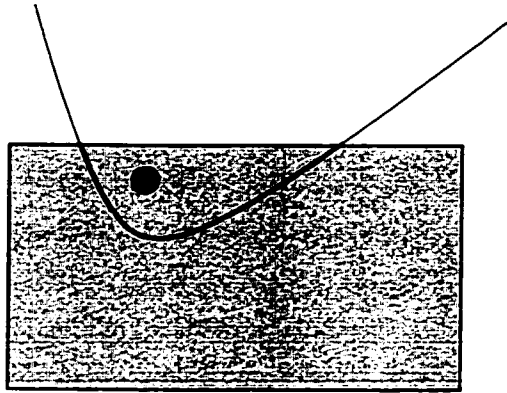


Figure 4.12: **The difficulty of point location without monotonic segments.** The gray region shows the interval bounding the point and the black dot shows the true coordinates. The point lies on the opposite side of a curve from the corner points of its interval.

to locate the entire point relative to that curve.

When the four corner points do not agree, the interval must be “straddling” the curve. In this case, shrink the point (Section 3.3.2.1) until all four corner points lie on the same side of the curve.

It is important to ensure that the point does not lie on the curve itself. If this is the case, no amount of shrinking will make the four corner points yield the same answer. In the boundary evaluation algorithm, the guarantee that the point is not on the curve is a result of the assumption of no input degeneracies.

One further observation is that if horizontal or vertical rays are used, classifying two corner points is usually the same amount of work as classifying a single rational point. For example, if the point is defined as  $s = a, t = [b_L, b_H]$ , then firing a ray along the line  $s = a$  will locate both endpoints of the interval. The endpoints only disagree when there is an intersection with a curve in the region  $b_L < t < b_H$ . Similarly, locating the four corner points of an interval is only twice as much work.

### 4.3.3 3D Point Location

Point location in three dimensions involves determining whether a point is inside or outside of a solid. Again, this is done by shooting a ray from the point and counting how many times it intersects the boundary of the other solid. Like the two-dimensional case, the rays are fired along one of the coordinate directions, to simplify

the computation.

The basic idea is to intersect the ray with each patch domain (there can be many intersections with any one patch). These intersection points are then located with respect to the trimming curves (via 2D point location) to determine whether or not they are within the trimmed region. Each intersection within the trimmed region is counted, and this count is tallied over all patches, with an odd number meaning the point is inside and an even number meaning the point is outside.

The assumption is that the point to be tested has rational coordinates in  $x$ ,  $y$ , and  $z$ , as is always the case in the boundary evaluation algorithm. Assume a ray is shot in the positive  $z$  direction (any other one of the other five signed coordinate directions would also work) from the point with coordinates  $(a, b, c)$ . For a given patch,  $P$ , find all intersection points within  $P$ 's domain. Assume the surface of  $P$  is defined by the parametric equations:  $(X(s, t), Y(s, t), Z(s, t), W(s, t))$ . The line along the  $z$  coordinate through the point is defined as the intersection of the planes  $x = a$  and  $y = b$ . The intersection of these planes with the surface of  $P$  gives the following intersections in the domain of  $P$ :

$$\begin{aligned} x = a &\Rightarrow X(s, t) = aW(s, t) \\ &\Rightarrow X(s, t) - aW(s, t) = f(s, t) = 0 \end{aligned} \tag{4.4}$$

$$\begin{aligned} y = b &\Rightarrow Y(s, t) = bW(s, t) \\ &\Rightarrow Y(s, t) - bW(s, t) = g(s, t) = 0 \end{aligned} \tag{4.5}$$

The intersections of the ray with  $P$ , then, are found by intersecting the curves  $f(s, t) = 0$  and  $g(s, t) = 0$  within  $P$ 's domain.

Once these points are isolated, check that they correspond to values on the correct side of the ray (e.g.  $z > c$  in the above example). This is done by raising the isolated 2D points (that are likely to be intervals) to three dimensions by interval arithmetic on  $z = Z(s, t)/W(s, t)$ . If the three-dimensional interval overlaps  $c$ , then shrink the 2D point until the 3D interval no longer overlaps. Also note that if the entire bounding box of the patch is on one side or the other of  $c$ , then any point in the patch is on that side of  $c$ . Thus, 3D interval operations for each point can often be avoided entirely.

In the boundary evaluation algorithm, the point to be tested is the result of point generation from a patch of solid A, to be located with respect to solid B. Thus if there are any difficulties (such as a potential tangential intersection), a different point from solid A can be easily generated, or another of the six rays can be tested. Testing a

different ray or generating and testing with a new point is likely to be more efficient than performing excessive operations (e.g. many levels of shrinking a 2D point) within 3D point location.

## 4.4 Finding an Implicit Surface Representation

Finding an implicit surface comes up two times during boundary evaluation. It first occurs when defining the surfaces of input solids. As stated in Section 3.1, a patch's surface must have both a parametric and implicit form. It is often easier to define the parametric form of a surface (Appendix A) based on input data, and so *implicitizing* a parametric surface can be important. Even though implicitization is not a part of the boundary evaluation algorithm, it can play a key role in defining input. The second time that implicit surface representations are generated is when splitting patches to break loops. The goal is, given a patch  $P$  with associated surface  $S_P$  and a constant-parameter line  $s = a$  (or  $t = a$ ), find the implicit form of a surface  $S_I$  such that the intersection of  $S_P$  with  $S_I$  is the line  $s = a$  in the domain of  $P$ . This will be referred to as *implicit generation*.

For both cases, the implicit surface is found by interpolation. The basic idea is to generate a set of points known to lie on the surface, and then generate a surface passing through those points. The approach that is described here involves setting up a linear system to solve for the coefficients of the implicit form of the surface.

Implicitization and interpolation are not new concepts. Extensive work in these areas has been done previously. For example, Sederberg [90] and Manocha [68] have described ways of performing surface implicitization using the Dixon resultant (Section 2.2.2). The basic idea in these cases is to form the system of equations:

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

and use a resultant to eliminate  $s$  and  $t$ , leaving a polynomial in  $x$ ,  $y$ , and  $z$ . The difficulty is that the Dixon resultant has problems when the original equations are not of the same degree. Although I am not aware of another specific approach that performs interpolation exactly as described here, interpolation is a well studied problem (particularly in numerical analysis), and it is certain that similar techniques have

been developed, described, and used before. A paper by Manocha and Canny [68], for example, uses interpolation to solve the Dixon resultant, and Zippel [106] describes two methods for polynomial interpolation.

For the CSG solids parameterized as in Appendix A, it is important to realize that implicit generation always yields a plane equation. That is, any parameter line can be determined by an intersection of some plane with the surface. Implicit generation is efficient during boundary evaluation since only such simple surfaces are needed.

#### 4.4.1 Generating Interpolating Points

For the first step of interpolation, generate a series of 3D *sample points* that lie on that surface. In order to avoid degenerate configurations of the sample points, it is best to choose random sample points. In practical implementations, only pseudorandom points can be generated, but this is usually sufficient for specifying a general collection of points.

For the examples of interest in the boundary evaluation algorithm, generate sample points by selecting an appropriate point in the domain of the patch, then raising it to three dimensions. For implicitization, choose any  $(s, t)$  points. For implicit generation, choose any  $t$  value along the specified line  $s = a$ . Note that the patch domain and trimmed region are irrelevant to the generation of sample points, unless they are used in generating the 2D points. It is assumed that all sample points have rational coordinates, as there is never an instance in the boundary evaluation algorithm that would require otherwise.

The 3D coordinates of the points are obtained from the parametric patch:

$$\begin{aligned}x &= X(s, t)/W(s, t) \\y &= Y(s, t)/W(s, t) \\z &= Z(s, t)/W(s, t)\end{aligned}$$

For a set of  $n$  sample points, the coordinates of the sample points are be  $(x_i, y_i, z_i)$  for  $i = (1, 2, \dots, n)$ .

#### 4.4.2 Interpolating a Plane

The most basic interpolating surface is the plane. A plane is defined by three sample points. Although an interpolation method for these points is well known and straight-

forward. it will be described in detail to illustrate the procedure that is followed with higher degree surfaces.

First it is assumed that the plane does not pass through the origin. Handling cases where it does is discussed later. A plane, then, is defined by the equation:

$$C_1x + C_2y + C_3z = 1 \quad (4.6)$$

Substitute the three sample points to yield the equations:

$$C_1x_1 + C_2y_1 + C_3z_1 = 1$$

$$C_1x_2 + C_2y_2 + C_3z_2 = 1$$

$$C_1x_3 + C_2y_3 + C_3z_3 = 1$$

that can be written in matrix form:

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad (4.7)$$

Solve this system directly, using Gaussian elimination, to determine the values of  $C_1$ ,  $C_2$ , and  $C_3$ .

One special case needs to be dealt with. If the three sample points are collinear (as often arises during implicit generation), there are many possible interpolating planes. When the points are collinear (which can easily be checked), form a vector  $A$  pointing from one point to another. Also choose a random vector  $B$ .  $A \times B$  gives the normal vector of a plane passing through these points, and this plane (when anchored to the points) can be used as the interpolating plane. In the implicit generation problem, this plane is acceptable as long as it is not tangent to the surface being split. To verify that the plane is not tangent, examine sample points to either side of the parameter line. If these points (in 3D) are on opposite sides of the plane, then the plane is not tangent.

### 4.4.3 Checking the Interpolated Surface

In some cases, the implicit degree of the surface is not known ahead of time. In such cases, begin by assuming that the surface is a plane, generating three sample points, and interpolating the plane. Then generate a fourth sample point and check the plane



equation to see if that point lies on the plane. If that point satisfies the plane equation, and the point was truly generated randomly, then with probability 1, the surface is a plane. In practice, points will not be truly random, so a number of pseudorandom points might be tested. The more points that are found to lie on the plane, the greater the probability that the surface is truly planar. In practical application, testing only one point is almost always sufficient. If any of these points does not satisfy the plane equation, then the surface cannot be a plane, so interpolate a surface of degree two (as described next). Once that surface has been generated, check new sample points to see whether they satisfy the surface equation, and generate a higher degree surface if necessary. Eventually, all sample points will be found to lie on the surface.

Note that this incremental process ensures that there is not an extraneous factor in the implicit equation. For example, if the surface is actually a plane, but a second-degree surface is interpolated, then the equation of the plane is a factor of the degree two surface. The portion of the degree two surface that is not the true surface is referred to as the extraneous factor. There are other ways of removing an extraneous factor (e.g. symbolic factorization), but the incremental approach is simpler to implement and is efficient for lower-degree (degree four and below) surfaces generally dealt with in this dissertation.

#### 4.4.4 Interpolating Higher-Degree Surfaces

To interpolate a higher-degree surface, a procedure similar to that for the plane surface is followed. Again, assuming that the surface does not pass through the origin, the constant term will be nonzero (assume it is -1). Determine a number of sample points equal to the number of undetermined coefficients. For example, a degree two surface would be:

$$C_1x^2 + C_2xy + C_3xz + C_4y^2 + C_5yz + C_6z^2 + C_7x + C_8y + C_9z = 1 \quad (4.8)$$

Nine coefficients need to be determined, so nine sample points are used.

Each row of the matrix is determined by one sample point, and each column corresponds to some power of  $x$ ,  $y$ , and  $z$ . For example, for the degree two case, the first row, first column would be  $x_1^2$ , the third row, second column would be  $x_3y_3$ , etc. Perform Gaussian elimination on this matrix to determine the values of the coefficients.

In the process of Gaussian elimination, it may occur that one or more rows disap-

pears entirely. In such a case, there are many surfaces of that degree passing through those points. Any of these surfaces is an acceptable solution. One or more of the coefficients will be undetermined. By setting the undetermined coefficients to zero, one at a time, a solution can be found that interpolates all points.

#### 4.4.5 Surfaces Passing through the Origin

When a surface passes through the origin, the constant term of the implicit form must be zero. Thus, the procedure described earlier does not work. This situation is recognized by the failure of Gaussian elimination to find any solution. For example, Gaussian elimination may lead to conflicting solutions, such as  $z = 2$  and  $z = 3$ . In such a case (where it is impossible to find a surface passing through the points), the surface must be passing through the origin.

If the surface is known to pass through the origin, there are a few ways of handling things. One approach is to translate every sample point by the same random amount in the same random direction. Next, compute the surface passing through those sample points. This surface is then translated back the same amount in the opposite direction. That final surface should be a solution for the original sample points, as well as for  $(0, 0, 0)$ .

A second approach is to set up a more general system, fixing the value of one of the other coefficients. For example, in a planar case, the coefficient of  $x$  could be fixed to be  $-1$ . Then, the surface equation would become:

$$-x + C_2y + C_3z = 0 \quad (4.9)$$

Substituting the three sample points in gives the equations:

$$\begin{aligned} C_2y_1 + C_3z_1 &= x_1 \\ C_2y_2 + C_3z_2 &= x_2 \\ C_2y_3 + C_3z_3 &= x_3 \end{aligned}$$

These yield the matrix equation:

$$\begin{bmatrix} y_1 & z_1 \\ y_2 & z_2 \\ y_3 & z_3 \end{bmatrix} \begin{bmatrix} C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (4.10)$$

which can be solved by Gaussian elimination. Higher degree surfaces are similar. This is a seemingly overconstrained system, but it has a solution as long as the coefficient of  $x$  fixed at  $-1$  (or whatever constant value was chosen) is not actually 0. If there is no possible solution to the matrix equation, then the coefficient that was fixed must also be zero, and a different coefficient is fixed to some value. This continues until eventually the matrix system can be solved, thereby providing an interpolating surface.

# Chapter 5

## Boundary Evaluation

This chapter discusses the boundary evaluation algorithm. Using the representations described in Chapter 3 and the kernel operations described in Chapter 4, this chapter describes the steps to perform a Boolean operation between two solids.

The approach used in this algorithm is similar to many previous approaches (Section 2.3.1) to boundary evaluation. Like those approaches, each pair of patches is intersected, and the patches are subdivided based on the intersection curves. Certain subpatches are then stitched together to form the patches of the final solid.

The algorithm presented here differs from previous approaches in the use of exact computation throughout. Exact computation has implications for both the representations used and the operations performed. Computation can be inefficient if an exact operation is naively substituted for an inexact operation. The steps described in the following sections, when coupled with the representations and kernel operations already described and the speedup techniques described in Chapter 6, produce an efficient exact algorithm for boundary evaluation.

In the discussions that follow, it is generally assumed that points are 2D intervals. In almost all cases, if one or both coordinates are known as rational numbers, the operations described simplify significantly.

### 5.1 Overview

The boundary evaluation algorithm is split into two stages. The first stage deals with pairs of patches. The output of this first stage is a set of intersection curves in each patch obtained from the intersections with all of the patches of the other solid. The second stage of the algorithm subdivides the patches in each solid and stitches certain subpatches together to compute the boundary of the final solid.

The first stage consists of five steps, described in detail in Sections 5.2–5.6. First, for each given pair of patches, intersection curves are computed. Second, the topology of the curves is resolved. Third, the intersections of the intersection curves with the trimming boundary are found. Fourth, correspondence between the curves in each patch domain is determined. Fifth, the curves are clipped to the trimming boundaries in both domains.

The second stage consists of four steps, described in detail in Sections 5.7–5.10. First, the various intersection curves are merged together. Second, the patches are divided into partitions, based on the merged curves. Third, the partitions are classified with respect to whether they are inside or outside of the other solid. Fourth, a subset of the partitions is merged together to form the final solid.

The following sections describe the steps in each stage. An overview of how the steps fit together, including where kernel operations are used, is shown in Figures 5.1 (first stage) and 5.2 (second stage).

Overall, the goal of the boundary evaluation algorithm can be stated as follows:

Given two solids,  $S_1$  and  $S_2$ , in general position and in the format described in Chapter 3, and given a Boolean operation  $op \in (\cap, \cup, \setminus)$ , produce a new solid,  $S_F$ , in the same format such that  $S_F = S_1 op S_2$ .

## 5.2 Generate Intersection Curves

If  $S_1$  is composed of  $m$  patches, and  $S_2$  is composed of  $n$  patches, then the first stage operations are repeated as many as  $mn$  times. For each of the first stage operations, assume that the patch from  $S_1$  is  $P_1$ , and the patch from  $S_2$  is  $P_2$ .

Given  $P_1$  and  $P_2$ , the first step is to find the intersection of their associated surfaces. This intersection is represented as an algebraic plane curve in the domain of each patch.

Let the surface associated with  $P_1$  have a rational parametric form given by  $X_1(s, t)$ ,  $Y_1(s, t)$ ,  $Z_1(s, t)$ , and  $W_1(s, t)$ , and an implicit form given by  $F_1(x, y, z) = 0$ . Similarly, let the surface associated with  $P_2$  have parametric form  $X_2(u, v)$ ,  $Y_2(u, v)$ ,  $Z_2(u, v)$ , and  $W_2(u, v)$ , and implicit form  $F_2(x, y, z) = 0$ . Consider the homogenized forms,  $\overline{F}_1$  and  $\overline{F}_2$ , of  $F_1 = 0$  and  $F_2 = 0$ . That is, add a homogenizing variable,  $w$ , so that all terms of  $\overline{F}_1(x, y, z, w)$  and  $\overline{F}_2(x, y, z, w)$  are of full order.

Compute the intersection between the two surfaces in each patch domain. In the

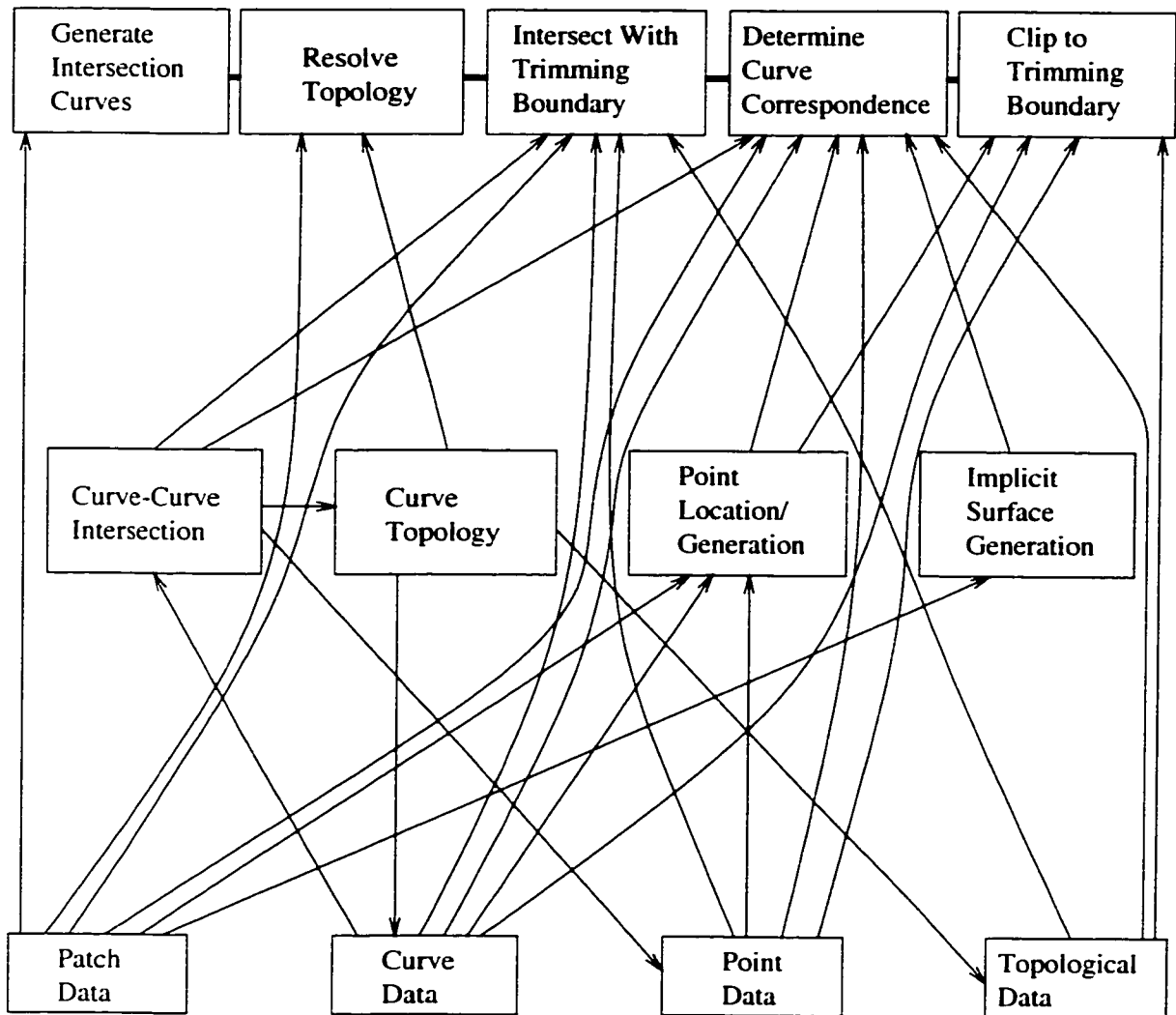


Figure 5.1: **A summary of the five main steps in the first stage of the boundary evaluation algorithm.** These operations are done for each pair of patches (one from the first solid, one from the second solid). Arrows show how the basic data and kernel operations are used in the various steps. At top are the steps in boundary evaluation, in the middle are the kernel operations, and at bottom are the data structures for the input solids.

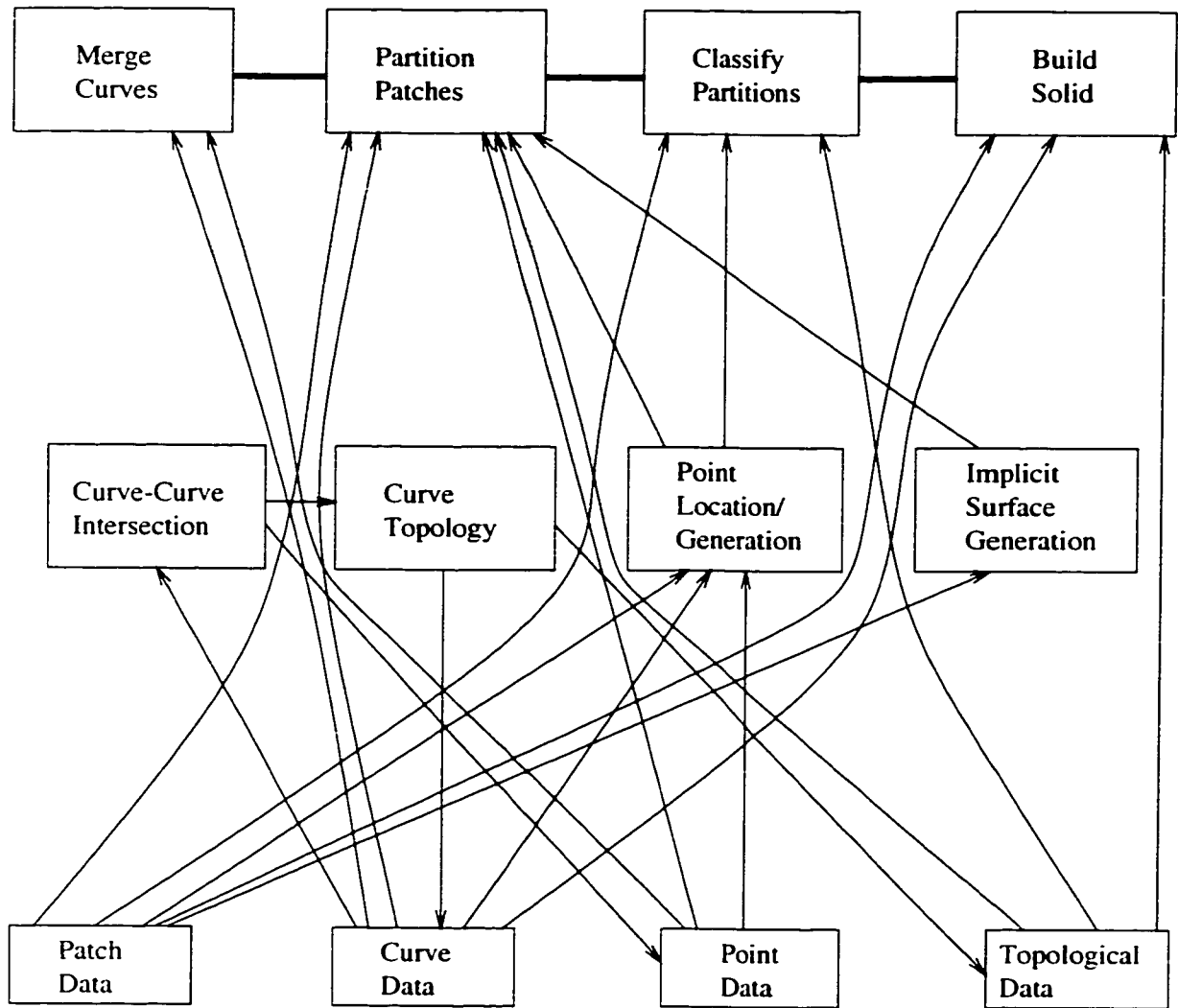


Figure 5.2: **A summary of the four main steps in the second stage of the boundary evaluation algorithm.** Arrows show how the basic data and kernel operations are used in the various steps. At top are the steps in boundary evaluation, in the middle are the kernel operations, and at bottom are the data structures for the input solids.

domain of  $P_1$ , the intersection curve is:

$$\overline{F}_2(X_1(s, t), Y_1(s, t), Z_1(s, t), W_1(s, t)) = f_1(s, t) = 0 \quad (5.1)$$

and in the domain of  $P_2$ , the intersection curve is:

$$\overline{F}_1(X_2(u, v), Y_2(u, v), Z_2(u, v), W_2(u, v)) = f_2(u, v) = 0 \quad (5.2)$$

Notice that this is the same as substituting  $X(s, t)/W(s, t)$ , etc. into  $F_1 = 0$ , then clearing polynomials from the denominator.

When the implicit and parametric forms are all defined as polynomials with rational coefficients,  $f_1$  and  $f_2$  are also polynomials with rational coefficients. Note that the algebraic plane curves  $f_1 = 0$  and  $f_2 = 0$  might not have a rational parameterization.

Assume  $F_1$  has implicit degree  $n$ . Assume that the highest implicit degree of  $X_2$ ,  $Y_2$ ,  $Z_2$ , and  $W_2$  is  $m$ . Then,  $f_2$  has implicit degree at most  $mn$ .  $f_1$  has a similar degree bound. For the standard CSG primitives, the implicit degree of the boundary surfaces ranges from 1 for planes to 4 for tori and certain generalized cones. Likewise, the degree of the parametric form can range from 1 to 4. Thus, the maximum total degree of  $f_1$  and  $f_2$ , for the objects considered here, is 16.

### 5.3 Resolve Topology

The second step is to resolve the topology of each of the algebraic plane curves,  $f_1(s, t) = 0$  and  $f_2(u, v) = 0$ . The procedure for resolving curve topology is discussed in Section 4.2. Resolve the topology of  $f_1$  over the domain of patch  $P_1$ ,  $[s_L, s_H] \times [t_L, t_H]$ , and the topology of  $f_2$  over the domain of patch  $P_2$ ,  $[u_L, u_H] \times [v_L, v_H]$ . The results are sets of *intersection curves*, in the format described in Section 3.2. that represent the intersection of the two patch surfaces in each domain.

Note that when resolving curve topology, there is no information regarding which curve in one domain corresponds to which curve in the other domain. None of the points found during curve topology resolution (such as turning points and edge points) have a known corresponding point in the other domain.



## 5.4 Intersect with Trimming Boundary

The third step is to intersect the intersection curves found in the previous step with the trimming curves. The procedure is described for patch  $P_1$ . The same procedure is used for patch  $P_2$ . This step takes place in two parts. First intersection points are found in the domain of  $P_1$  (Section 5.4.1), then the points are inverted into  $P_2$ 's domain (Section 5.4.2).

### 5.4.1 Find Intersection Points

Intersect each trimming curve with the intersection curves, as follows. Because all intersection curves come from the same algebraic plane curve,  $f_1(s, t) = 0$ , it is sufficient to perform a single curve-curve intersection (Section 4.1) per trimming curve. Let  $T_i$  be the trimming curve, and  $\theta_i(s, t) = 0$  be the trimming curve's polynomial. Compute all intersections between  $f_1 = 0$  and  $\theta_i = 0$  within the domain of  $P_1$ . Each intersection must be contained within exactly one intersection curve, since each point on  $f_1(s, t) = 0$  in the patch domain is on exactly one intersection curve. Test each intersection point to see whether it is contained in the trimming curve  $T_i$  (Section 3.2.2). The intersections that lie on  $T_i$  are called *intersection points*. Those that do not are discarded. The procedure is illustrated in Figure 5.3.

Once each intersection point  $Q$  is found, insert  $Q$  (Section 3.2.1) into both the trimming curve  $T_i$  and the appropriate intersection curve. For each curve, an associated curve in the domain of a different patch will also have a point inserted. Though the actual insertion is straightforward, finding the equivalent point in another domain can be difficult. The term *equivalent*, when used here, means a point that is the same three-dimensional point, but represented in the domain of a different patch. Finding the equivalent point is discussed in Section 5.4.2.

Note that it is not necessary to find the equivalent point in the adjacent patch of  $T_i$ , because that point is automatically determined when that patch is intersected with  $P_2$ . The point in that patch's domain is found equivalent to  $Q$  when patches are partitioned (Section 5.8).

### 5.4.2 Point Inversion

Point *inversion* is the process of taking a point in one patch domain, say  $P_1$ 's, and finding the equivalent point in another patch domain, say  $P_2$ 's. The two points are

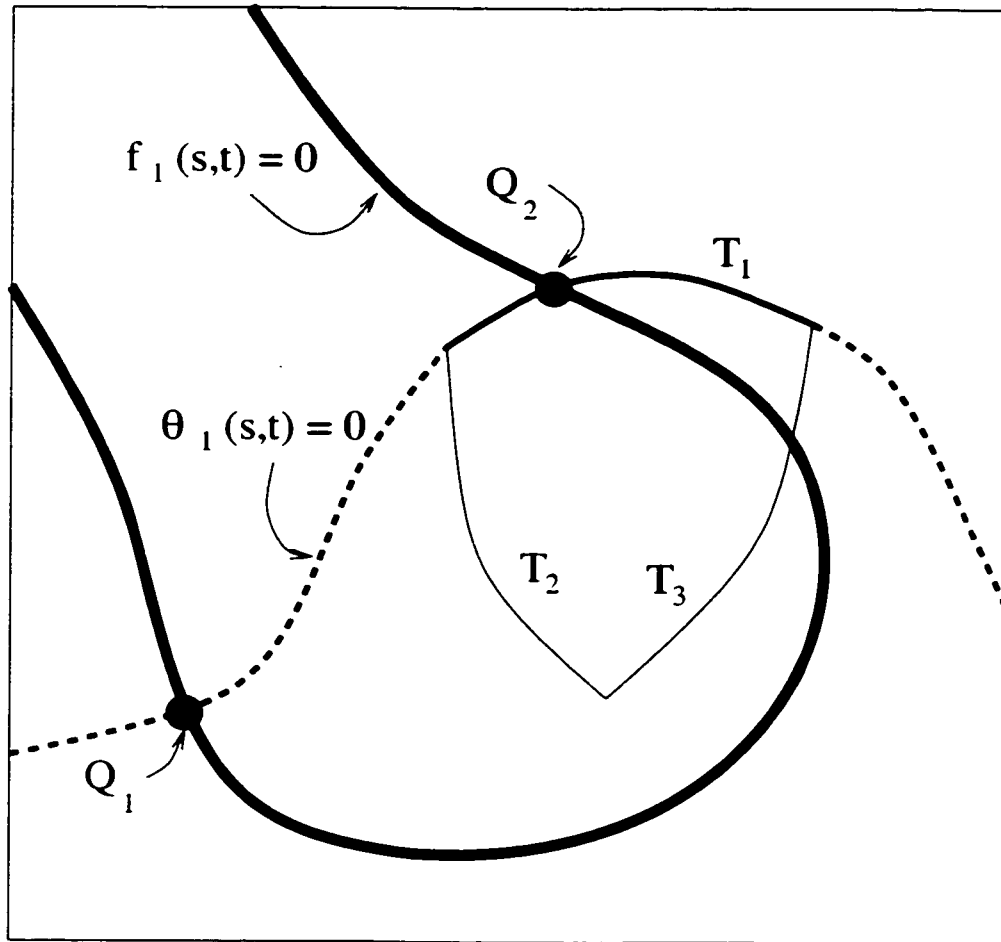


Figure 5.3: **Finding intersection points in the domain of a patch.** The three trimming curves are  $T_1$  (shown with the medium thickness line),  $T_2$ , and  $T_3$ .  $f_1(s,t) = 0$  forms one intersection curve in the domain of interest, as shown with the thickest line. Intersecting  $f_1(s,t) = 0$  with  $T_1$ 's polynomial,  $\theta_1(s,t) = 0$  (shown by the dashed line) produces two intersections,  $Q_1$  and  $Q_2$ . Only  $Q_2$  is located on  $T_1$ .  $Q_2$  is an intersection point for  $T_1$ , and  $Q_1$  is discarded.

then *associated* with each other (Section 3.4). Many approaches to point inversion solve algebraic systems in a higher-dimensional space (such as  $s \times t \times u \times v$ ). While these may be practical in an inexact (e.g. floating-point based) system, the higher number of dimensions usually makes exact computations too slow to be useful. Instead, an alternate approach using lower-dimensional computations will be used here. A set of possible inversion points is created in the other patch domain, then the one that is the corresponding inverse point is chosen.

Recall (from Section 3.1) that with each trimming curve  $T_i$  is an adjacent surface. Let the implicit form of this surface be  $\Omega_i(x, y, z) = 0$ . Let  $\mathcal{S}_1$  be the surface of  $P_1$  and  $\mathcal{S}_2$  be the surface of  $P_2$ . We then know the following:

- $T_i$ 's polynomial is the intersection of  $\Omega_i = 0$  with  $\mathcal{S}_1$ .
- The intersection of  $\mathcal{S}_1$  with  $\mathcal{S}_2$  yields the polynomial of the intersection curves ( $f_1(s, t) = 0$  in  $P_1$ 's domain,  $f_2(u, v) = 0$  in  $P_2$ 's).
- Intersection points (where  $T_i$  meets an intersection curve) occur where  $\Omega_i$ ,  $\mathcal{S}_1$  and  $\mathcal{S}_2$  all meet.

Perform point inversion as follows:

**Goal:** Given an intersection point  $Q$  that is formed from the intersection of  $T_i$  with  $f_1(s, t) = 0$ , find the equivalent point  $Q'$  in the parameter space of  $P_2$ .

**Procedure:**

1.  $Q'$  must lie on the intersection of  $\Omega_i(x, y, z) = 0$  and  $\mathcal{S}_2$ . Construct a new algebraic plane curve,  $f_3(u, v) = 0$ , by substituting the parametric form of  $\mathcal{S}_2$  into  $\Omega_i = 0$  (similar to the process for generating intersection curves in Section 5.2).  $f_3 = 0$  is the intersection of  $\Omega_i(x, y, z) = 0$  and  $\mathcal{S}_2$ , so  $Q'$  must lie on  $f_3(u, v) = 0$ .
2.  $Q'$  must lie on the intersection of  $\mathcal{S}_1$  and  $\mathcal{S}_2$ , thus it must also lie on  $f_2(u, v) = 0$ . Since  $Q'$  must lie on both  $f_2 = 0$  and  $f_3 = 0$ , it must be at an intersection of  $f_2 = 0$  and  $f_3 = 0$ . Find all intersections between  $f_2 = 0$  and  $f_3 = 0$ . Assume that the number of intersections found is  $n$ . One of these  $n$  intersections must be  $Q'$ .

Note that all intersections over  $\Re^2$  must be found, not just those within the domain of  $P_2$ . This is because the point corresponding to  $Q$  can be any of the

intersection points, not just the ones in  $P_2$ 's domain. Conservative bounds on root values exist that bound the range that must be examined to find common roots of the two polynomials.

3. Perform 3D point matching to find which of the  $n$  possible points is the same point (in 3D) as  $Q$ . This is done as follows:
  - (a) For  $Q$  and all the potential matching points, construct a 3D interval that is guaranteed to bound the point found in 2D. For example, if  $Q$  is bounded by the interval  $[s_1, s_2] \times [t_1, t_2]$ , then a three-dimensional interval bounding  $Q$  is found by performing the interval operations:  $x = X_1([s_1, s_2], [t_1, t_2]) / W_1([s_1, s_2], [t_1, t_2])$ , etc.
  - (b) Compare each of the  $n$  3D intervals to  $Q$ 's 3D interval, checking for overlap. At least one interval (corresponding to  $Q'$ ) must overlap. If only one of the  $n$  intervals overlaps with  $Q$ 's interval, then that point must be  $Q'$ .
  - (c) If more than one interval overlaps, shrink the 2D intervals of both  $Q$  and the points that had overlapping 3D intervals with  $Q$ 's. Compute new 3D intervals (which must be smaller than the previous), for those points. Check for overlap of the 3D intervals. Continue this process of shrinking the 2D intervals and computing new 3D intervals until only one interval overlaps with  $Q$ 's interval. That point must be  $Q'$ .
4.  $Q'$  is called the *inverted point*. Check to see whether or not  $Q'$  is located in the domain of  $P_2$ . If it does lie in the domain, then insert  $Q'$  into the appropriate intersection curve in  $P_2$ , otherwise discard it. If  $Q'$  lies in the domain of  $P_2$ , then  $Q$  and  $Q'$  are defined as each other's *associated point* (Section 3.4).

## 5.5 Determine Curve Correspondence

The fourth step in the first stage of the boundary evaluation algorithm is to determine a *correspondence* between the intersection curves in  $P_1$  and those in  $P_2$ . Before the process is described, the notion of correspondence must be explained.

Correspondence means that the two curves are equivalent (i.e. they refer to the same curve in three dimensions), and that the relative direction of each curve is known. Recall that the curve representation provides a direction (from starting point to ending point) on each curve.

Each patch represents a subset of an algebraic surface. The intersection between the two surfaces may have more than one component. All, part, or none of each component may appear in the domain of a patch, and one component of the three-dimensional curve may give rise to many components in the patch domain (each of which is an intersection curve). Thus, even if there are many intersection curves within the two patch domains, each patch might contain a separate subset of the three-dimensional intersection curve. An individual intersection curve can correspond to many, one, or none of the curves in the other domain.

The input to curve correspondence is two sets of curves, one in each patch domain. The topology of each curve has already been resolved and equivalent points found (by intersecting with trimming curves). The output of curve correspondence is a list of which curves are found equivalent, along with their relative directions.

Assume that  $f_1 = 0$  is broken into  $m$  intersection curves (i.e. real components of the algebraic plane curve),  $A_i$ , in the domain of  $P_1$ , and  $f_2 = 0$  is broken into  $n$  intersection curves,  $B_i$ , in the domain of  $P_2$ . The  $A_i$  form the first input set of curves, the  $B_i$  the second. The output of curve correspondence can be written as a table with  $m$  rows and  $n$  columns, where the  $(i, j)$  element states whether moving "forward" on  $A_i$  corresponds to moving "forward," "backward," or "neither" on  $B_j$ . An example of curve correspondence is given in Figure 5.4. For that example, the table listing correspondences would be:

	$B_1$	$B_2$
$A_1$	backward	neither
$A_2$	neither	forward
$A_3$	neither	neither

So, if  $Q$  is a point on  $A_1$ , with an associated point  $Q'$  on  $B_1$ , starting at  $Q$  and moving forward on  $A_1$  is the same as starting at  $Q'$  and moving backward on  $B_1$ .

Note that if an intersection point  $Q$  is on a curve  $A_i$  and its inverted point  $Q'$  is on a curve  $B_j$ , then the entry for  $(i, j)$  may not be "neither", since the curves must have a correspondence. Said another way, if a single point in 3D is found to be on both a curve in  $P_1$  and a curve in  $P_2$ , then those two curves must have a correspondence. Also, just because a pair is marked "neither" does not mean that the curves do not come from the same three-dimensional component of the intersection curve. The neither marking simply means that no common points were found between the two curves, and thus even if a correspondence might exist, it is unnecessary for future

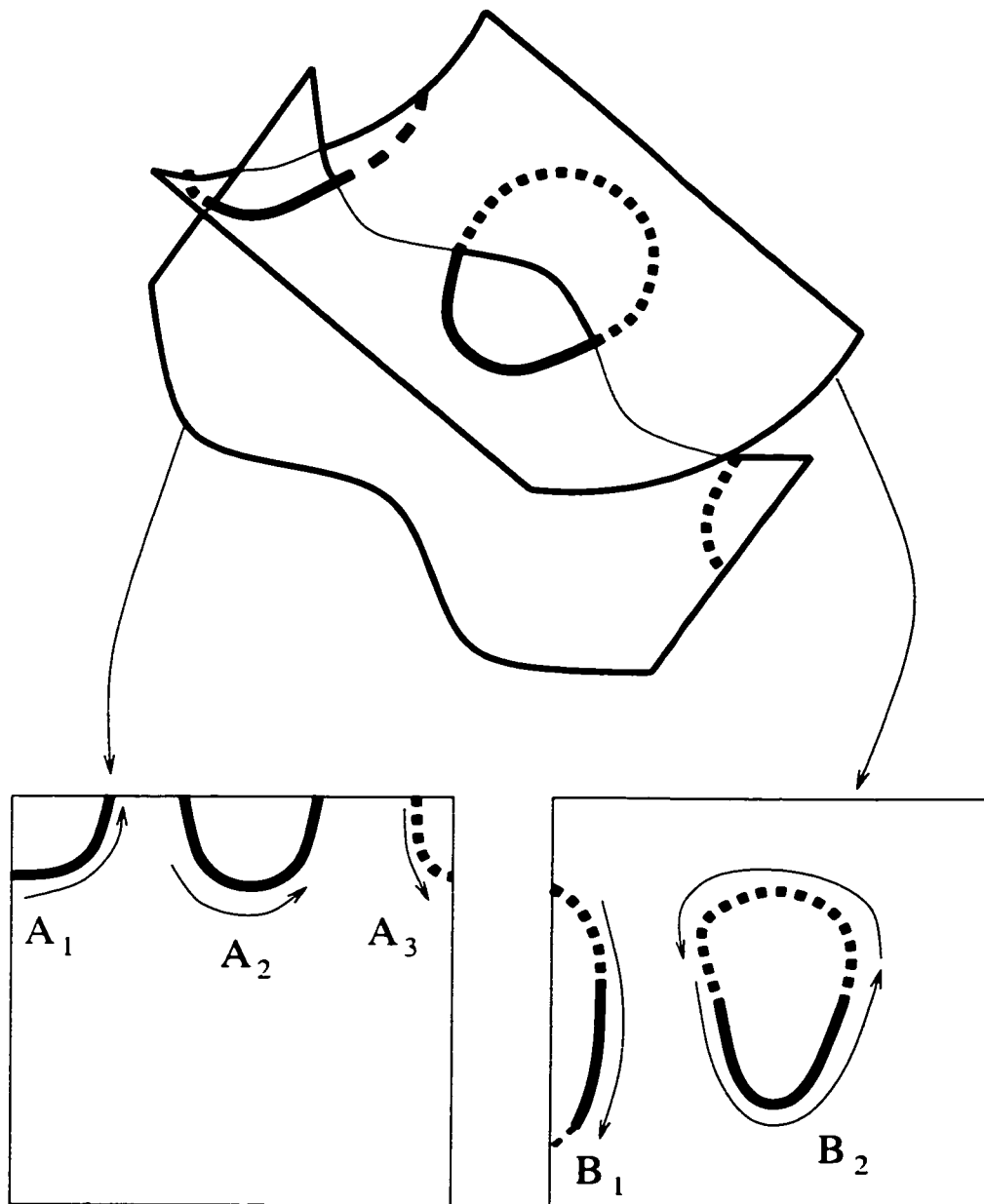


Figure 5.4: **Correspondence between curves on different patches.** Recall that each patch is a subset of a larger algebraic surface. The intersection of the two surfaces results in three intersection curves in the first patch's domain, and two in the second's. The solid lines show the portions of the intersection curves that appear in both patch domains, while dashed lines show the portions that appear in only one patch domain. Curve correspondence identifies which curves in one domain correspond to which in the other. In this case,  $A_1$  and  $B_1$  have backward correspondence.  $A_2$  and  $B_2$  have forward correspondence. All other pairs have neither correspondence.

computation. The process for finding correspondences follows.

The key idea is that curve correspondence can be conclusively determined between two curves if three equivalent points are known on each curve. Consider the curves in Figure 5.5. As the figure shows, having only two points is not enough to determine correspondence between two closed loops. Three equivalent points on each curve is enough, however. A similar condition exists when the curves are not closed loops. Any curve in the patch domain can be part of a larger closed loop (part of which is outside the patch domain), so three points on each curve in each domain are needed in order to conclusively find the correspondence between curves.

Consider each curve from patch  $P_1$  as one of two cases. Section 5.5.1 discusses the case when a curve has at least one intersection or inverted point on it. Section 5.5.2 discusses the case when a curve has no intersection or inverted points. The following sections describe how to find the correspondence for a given curve in  $P_1$ 's domain. It is *not* necessary to perform a similar computation for the curves in patch  $P_2$ , since all information is determined by examining the curves in  $P_1$ . Pairs of curves that do not have correspondence found by the following process will be marked as having "neither" correspondence.

Hereafter, the curve (in the domain of  $P_1$ ) is referred to as  $C$ . An equivalent curve to  $C$  in  $P_2$ 's domain is referred to as  $C'$ .

### 5.5.1 Curves with Intersection or Inverted Points

If  $C$  contains an inverted point, then there must be an equivalent curve  $C'$  that contains the intersection point associated with that inverted point. Likewise, if  $C$  contains an intersection point  $Q$  for which an inverted point  $Q'$  exists, then an equivalent curve  $C'$  must exist (and it contains  $Q'$ ). Ignore intersection points that do not have an inverted point in the domain of  $P_2$ . If  $C$  contains only intersection points that do not have associated inverted points, then  $C$ 's equivalent is clearly outside the trimmed region of  $P_2$ . Discard  $C$  for the remainder of the boundary evaluation algorithm.

For each curve  $C'_i$  with which  $C$  shares at least one associated point, determine a correspondence between  $C$  and  $C'_i$ . To do this, find three equivalent points on each curve (Section 5.5.1.2 below). Once these three points are found, determining correspondence is straightforward.

Label the three points on  $C$ :  $q_1$ ,  $q_2$ , and  $q_3$ , in order from closest to  $C$ 's starting point to farthest from the starting point. Let the associated points on  $C'_i$  be  $q'_1$ ,  $q'_2$ , and

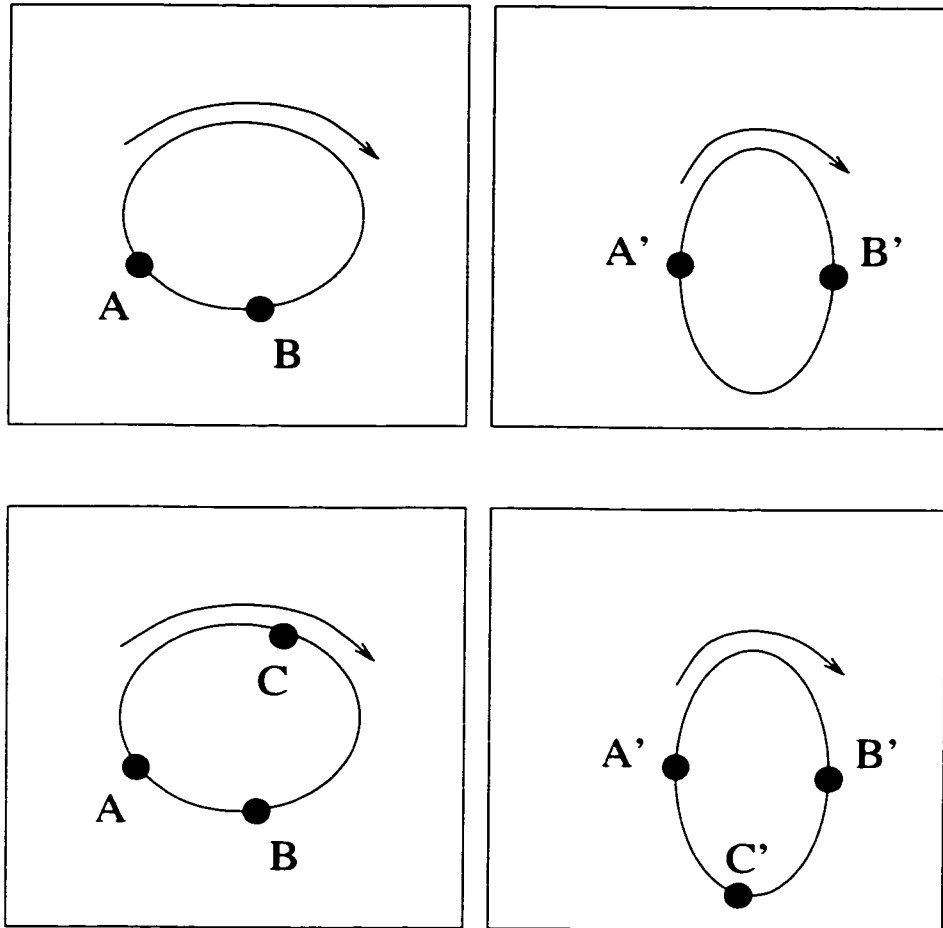


Figure 5.5: **The necessity of three points for curve correspondence.** At top, curve correspondence cannot be determined based on just two points. The arrows show the “forward” direction of each curve in its own domain. As one moves forward from  $A$  to  $B$ , it is not clear whether that is equivalent to moving forward (clockwise) from  $A'$  to  $B'$ , or backward (counterclockwise) from  $A'$  to  $B'$ . At bottom, adding a third point allows correspondence to be determined. On the left patch, moving forward on the curve starting at  $A$  reaches  $C$  before reaching  $B$ . Since one must travel backward in the patch at right in order to reach  $A'$  then  $C'$  then  $B'$ , there must be “backward” correspondence.



$q'_3$ , respectively. Starting at  $q'_1$ , determine whether  $q'_2$  or  $q'_3$  is the first to be reached when proceeding in the forward direction. For the sake of this ordering, assume that  $C'_i$  is a closed curve, so that if the end of the curve is reached, just continue from the start of the curve. If  $q'_2$  is reached first, then the forward direction on  $C'_i$  is the same as on  $C$ , so the correspondence is forward. If  $q'_3$  is reached first, the correspondence is backward.

### 5.5.1.1 Simplifying Observations

A few significant observations help to simplify the overall procedure. An implementation may make use of these to improve efficiency, but the method described above is general and handles all the cases.

- One observation is that *one* point on either  $C$  or  $C'$  (but not both) does not need an associated inverted point on the other curve. For example, if  $q_1$  and  $q_2$  have inverted points on  $C'_i$ , but  $q_3$  has no inverted point, or an inverted point on a different  $C'_j (j \neq i)$ , then consider  $q'_3$  to lie after the end point of  $C'_i$ . In such a situation,  $f_2(u, v)$  extends beyond the domain of  $P_2$ , and  $q'_3$  is assumed to lie somewhere in that area. Notice that this can only be used for one point, since there is no way to order the points in that region.
- A second observation is that if  $C$  contains no intersection points, and only inverted points, and if  $C$  is not a loop, then  $C$  clearly lies outside of the trimmed region. Even if it is a loop, it may still lie outside the trimmed region. In such a case,  $C$  can be eliminated entirely from further consideration. The procedure that is described for clipping curves to the trimmed region (Section 5.6) does this automatically, however, so further discussion will not be given here.
- A third observation is that if  $C$  contains an intersection point that has an associated inverted point on  $C'_i$ , and  $C$  contains no other points associated with ones in  $C'_i$ , then any portion of  $C'_i$  that is equivalent to  $C$  must lie outside the trimmed region of  $P_2$ . The reasoning behind this is complicated and is only summarized here. The basic idea is that the number of intersection points along  $C$  must be even (since it is assumed that there are no tangential intersections). Since only one intersection point inverts to a point on  $C'_i$ , the other one must lie outside of the domain of  $P_2$  (as far as  $C'_i$  is concerned). Thus, at least some of  $C'_i$  must lie outside the trimmed region. Since there is no intersection of  $C'_i$

with  $P_2$ 's trimming curves in the region that would invert to  $C$ , then the entire portion of  $C'_i$  that corresponds to  $C$  must be outside the trimmed region. When this is the case, correspondence does not have to be determined, and regions can be clipped away automatically.

- A final observation, by reasoning similar to the previous case, is that if a curve contains only one inverted point and no intersection points corresponding to the other curve,  $C'_i$ , then this portion of  $C$  lies outside the trimmed region, and its correspondence does not have to be determined.

The net result is that unless there are at least two equivalent points (intersection or inverted) between a pair of curves, correspondence does not have to be determined. So, at most one other point needs to be generated.

### 5.5.1.2 Generating More Intersection Points

In many cases, there will already be three or more intersection or inversion points on  $C$  corresponding to each  $C'_i$ , or the observations in Section 5.5.1.1 make additional points unnecessary. At most one new point needs to be generated.

To generate a new point, intersect  $C$  with a horizontal or vertical line, chosen as in the point generation algorithm (Section 4.3.1). In order to have the inverse point be likely to lie on  $C'_i$ , choose a parameter value for the line that is close to a known inverted point. For example, if two points on  $C$  with equivalent points on  $C'_i$  are known, choosing the midpoint of the points on  $C$  as the parameter value is likely to generate an intersection point that has an equivalent inverted point on  $C'_i$ .

Once the intersection points are found, they should be inverted (Section 5.4.2), which entails interpolating a surface that passes through the parameter curve (Section 4.4). If the inverted point(s) do not lie on  $C'_i$ , then intersect a new line with  $C$  until such a point is found.

The intersection and inverse points generated in this way are used only to determine curve correspondence, *not* to perform curve clipping (Section 5.6).

### 5.5.2 Curves Without Intersection or Inverted Points

If  $C$  has no intersection or inverted points, neither  $C$  nor  $C'$  (if it exists) intersects the trimming boundary in either domain. There are two ways this can happen. First,  $C$  may lie outside of the trimmed region. Second,  $C$  may be a loop contained entirely

inside the trimmed region.  $C'$ , in the other patch domain, must also be in one of these two categories.

If either  $C$  or  $C'$  is outside of the trimmed region, then  $C$  can be discarded for the remainder of the boundary evaluation algorithm. Only when  $C$  is a loop within  $P_1$ 's trimmed region *and*  $C'$  is a loop within  $P_2$ 's trimmed region does the curve matter. In this case, curve correspondence must be determined.

First, check whether  $C$  is a loop. This is a trivial test (the endpoint must be equal to the start point). If  $C$  is a loop, a point on  $C$  is chosen (any segment endpoint will do), and that point is located with respect to the trimmed region (Section 4.3.2). If the point is within the trimmed region, continue. Otherwise, the loop is outside the trimmed region, so discard  $C$ .

To find curve correspondence, first determine whether  $C'$  exists. Recall that in order for  $C$  to matter,  $C'$  must be a loop wholly inside the trimmed region. First check the intersection curves in  $P_2$ 's domain to see whether there are any closed loops with no intersection or inverted points inside the trimmed region. If not, then  $C'$  must be outside the trimmed region of  $P_2$  (if it exists at all), and therefore  $C$  can be discarded. If there is a possible  $C'$ , then generate (and invert) intersection points along  $C$ , as in Section 5.5.1.2. If the inverted points do not lie within the trimmed region of the other patch, then  $C'$  does not lie within the trimmed region of the other patch, and so  $C$  can be discarded. Otherwise, the inverted points lie on one curve in  $P_2$ 's domain. This curve is  $C'$ . Once three equivalent points have been found, which may mean intersecting  $C$  with another line, the correspondence between  $C$  and  $C'$  can be determined.

## 5.6 Clip Curves to Trimming Boundary

The fifth and final step in the first stage of the boundary evaluation algorithm is to *clip* the intersection curves. Clipping means eliminating those segments of the intersection curve that lie outside the trimmed region of either patch. After curve clipping, the remaining intersection curves are the portions of  $f_1(s, t) = 0$  (and  $f_2(s, t) = 0$ ) that lie in the trimmed regions of both patches. The procedure that follows is performed for each intersection curve on each patch.

Let  $C$  (an intersection curve in the domain of patch  $P_1$ ) be the curve under consideration. Assume that  $C$  is broken up into  $n$  segments. Let  $Q$  be the set of intersection points located on  $C$ , with individual points called  $Q_i$ , and  $R$  be the

inverted points located on  $C$ , with individual points called  $R_j$ . For each of the  $n$  segments of  $C$ , maintain a marker as to whether that segment is “good” or “bad.” Set all markers to “good” originally, and change them to “bad” as necessary during the clipping process. The “bad” segments are removed in the last step of the clipping process. Maintain this set of markers for all intersection curves in both patch domains, since clipping removes the regions in both domains.

Because a curve must cross a trimming boundary (in either patch domain) in order to make a transition from outside the trimmed region to inside the trimmed region (or vice-versa), transitions from “good” to “bad” can only happen around one of the  $Q_i$  or  $R_j$ . Note that since each  $Q_i$  and  $R_j$  has been inserted into the curve (Section 5.4), each  $Q_i$  and  $R_j$  must occur at the end of a curve segment. So, given two points,  $A, B \in Q \cup R$ , with no other points in  $Q \cup R$  between  $A$  and  $B$ , all the segments of  $C$  starting at  $A$  and ending at  $B$  will be marked the same (either “good” or “bad”). A transition from good to bad or vice-versa is not possible in that region since the curve does not pass a trimming curve (from either domain) at any point in that region.

In the clipping procedure below, a key operation is *marking bad forward* (or marking bad backward). Both operations take as input a starting point  $A$  that is a segment endpoint of  $C$ . Marking bad forward is marking all segments of  $C$  after  $A$  as “bad” until another segment endpoint  $B$  is reached that is an element of  $Q \cup R$ . If there is no such point  $B$  after  $A$  on the curve, then all segments to the end of the curve are marked “bad.” If the curve is a loop, then the marking bad operation should continue from the starting point of the curve, once the ending point is reached. Marking bad backward is the same process, just in the opposite direction.

Following is the procedure for curve clipping.

- Maintain a status counter. The counter may have the value “out” or “in.”
1. First, handle one special case situation. If all of the following hold for  $C$ :
    - (a) The number of points in  $Q \cup R$  is zero (i.e. there are no intersection or inverted points on the curve).
    - (b) The curve is a loop.
    - (c)  $C$  is inside the trimmed region of  $P_1$ . This is determined by locating any point on  $C$  with respect to  $P_1$ 's trimming curves.
    - (d) There is no corresponding curve to  $C$  in the domain of  $P_2$ . This is determined by finding whether every entry in  $C$ 's row of the correspondence table is “neither.”

Then mark the entire curve bad. This describes a case where there is a loop that is entirely inside  $P_1$ 's trimmed region, but outside the domain of  $P_2$ . Every segment of  $C$  is marked bad, and the remaining steps are skipped for  $C$ .

2. Determine whether the starting point of the curve lies inside or outside of the trimmed region. If the curve is not a loop, the starting point always lies outside the trimmed region, since the curve must start on the boundary of the patch domain, and the entire patch domain boundary lies outside of the trimming curves. If the curve is a loop, the starting point should be located with respect to the trimmed region (Section 4.3.2) to determine whether it is inside or outside. There is a small chance that the point lies on the trimming curves, so if the point location is not immediately clear, rotate the curve (Section 3.2.2) until a point that is clearly inside or outside is found. Such a point always exists, though in the worst case it may be necessary to shrink the points to find it. Note that the good/bad marker associated with each segment must be rotated along with the curve.

Initialize the status counter to "in" if the starting point is inside the trimmed region, or "out" if the starting point is outside.

3. Next, march along the curve, one segment at a time, from the starting point to the ending point. If the current status is "out", mark the current segment bad. This means that the current segment is outside of the trimmed region of  $P_1$ . Before moving to the next segment, examine the endpoint of the current segment to determine whether the segment endpoint is an intersection point,  $Q_i \in Q$ . If so, special action is taken:
  - (a) If the segment endpoint is an intersection point,  $C$  must be passing the trimming boundary of  $P_1$ . Thus, reverse the status, either from "in" to "out" or from "out" to "in."
  - (b) Reaching an intersection point requires that action be taken based on the inverted point. Two cases are possible, and these are distinguished by performing point location (Section 4.3.2). Locate the point associated with  $Q_i$  (i.e. the equivalent inverted point of  $Q_i$ ), relative to the trimmed region of  $P_2$ . Note that the associated point cannot lie on  $P_2$ 's trimming boundary unless there is a degenerate situation, since  $Q_i$  lies on the trimming boundary. Two cases are possible:

- i. *The associated point is outside the trimmed region of  $P_2$* : This includes any case where the associated point is outside the domain of  $P_2$ . In such a case, the segments of  $C$  around  $Q_i$  are clearly equivalent to segments outside of the trimmed region of  $P_2$ . So, mark the segments of  $C$  bad both forward and backward from  $Q$ .
- ii. *The associated point is in the trimmed region of  $P_2$* : In this case, part of a curve in  $P_2$ 's domain must be clipped away. Let the associated point be  $Q'_i$ . Examine the intersection curves in  $P_2$ 's domain to determine which one contains  $Q'_i$ . Call this curve  $C'$ . Mark  $C'$  bad either forward or backward, depending on whether the status on  $C$  is changing from in to out or from out to in. This is where curve correspondence is used. For example, if at  $Q_i$ , the status on  $C$  is moving from in to out, then mark the section equivalent to  $C$ 's forward section bad. If the correspondence between  $C$  and  $C'$  is forward, then mark bad forward on  $C'$ . If the correspondence is backward, then mark bad backward on  $C'$ . If the status had been changing from out to in, this would be reversed.

In this way, all segments outside of the trimming region in patch  $P_1$  are marked bad, and the corresponding sections of curves in  $P_2$  are also marked bad.

Examples of curve clipping are shown in Figures 5.6 and 5.7. These examples show the original intersection curves, the segments remaining after curve clipping in one patch, and the segments remaining after curve clipping in the other patch. Situations far more complex than those in the examples (e.g. many components passing the trimming boundary many times in each domain) are possible.

After this process is performed for all intersection curves in both patch domains, all segments that are outside of at least one trimmed region have been marked "bad." The next step is to find any string of consecutive segments that are still marked "good." Each such string of consecutive "good" segments should then be turned into an intersection curve, referred to as a final intersection curve. Note that the number of final intersection curves can range from zero (if all segments were marked "bad"), to far greater than the original number of intersection curves. Each final intersection curve in  $P_1$  is equivalent to exactly one final intersection curve in  $P_2$ . These curves are marked as *associated* with each other (Section 3.4).

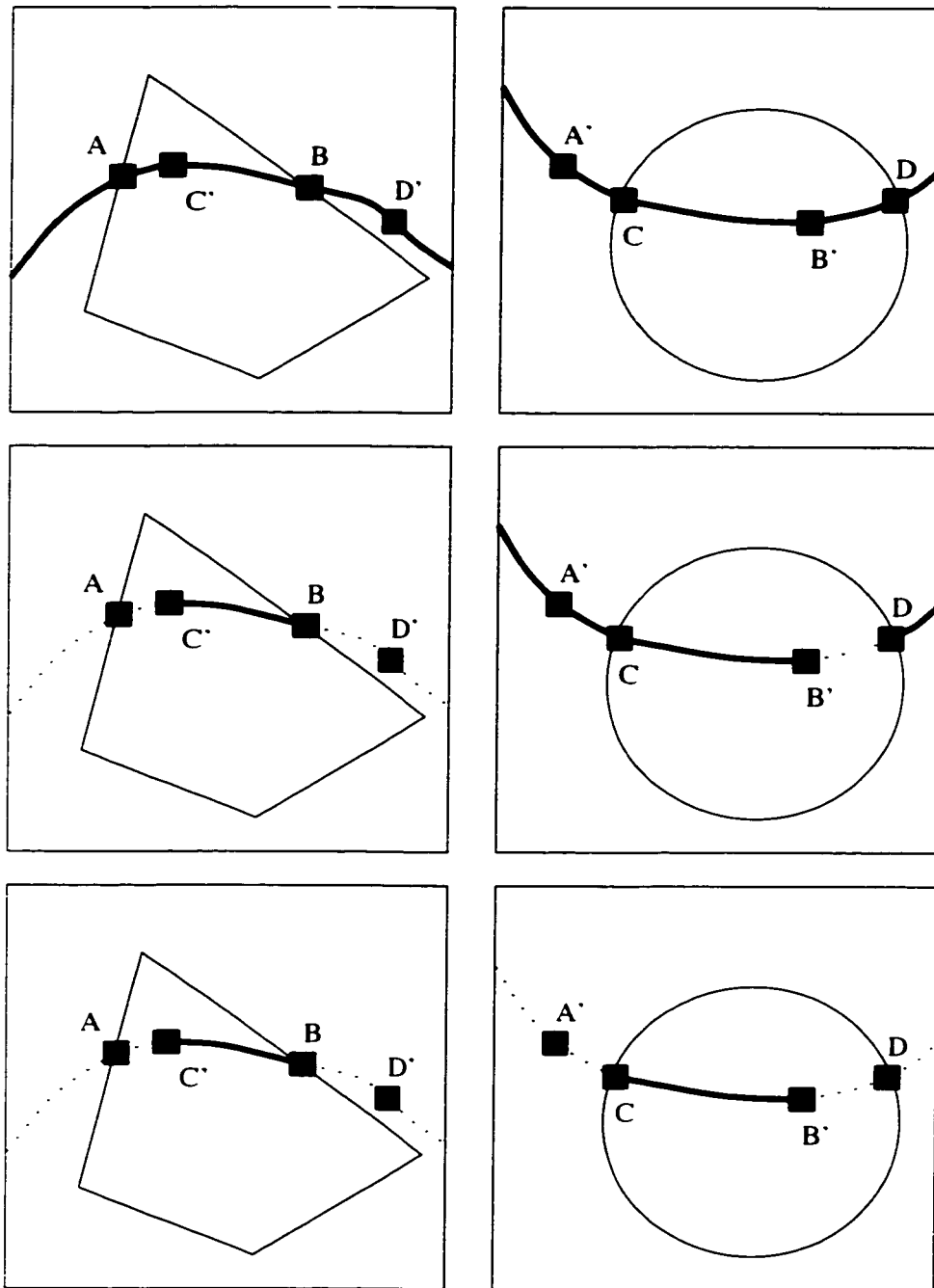


Figure 5.6: **An example of curve clipping.** At top, the original curves in each domain. In the middle, the results after curve clipping in the left-hand patch. At bottom, results after curve clipping in the right-hand patch. Trimming curves are shown with light lines, and dotted lines show portions of intersection curves that are marked bad. Points marked with a ' signify inverted points (e.g. A' is A inverted).

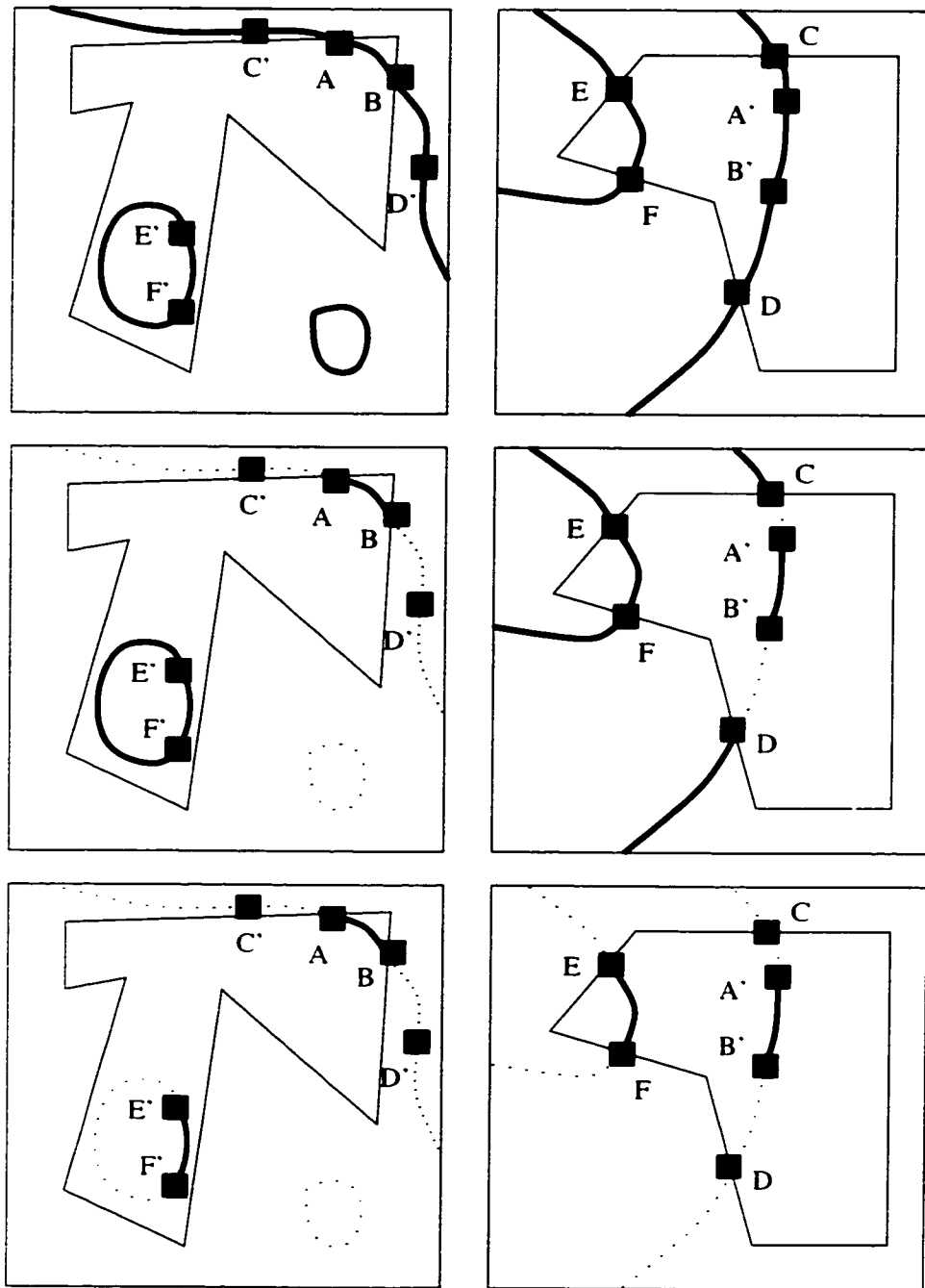


Figure 5.7: **Another example of curve clipping.** At top, the original curves in each domain. In the middle, the results after curve clipping in the left-hand patch. At bottom, results after curve clipping in the right-hand patch. Trimming curves are shown with light lines, and dotted lines show portions of intersection curves that are marked bad. Points marked with a ' signify inverted points (e.g. A' is A inverted).



When clipping is finished, all the final intersection curves are appended to a list of intersection curves maintained in each patch. When all pairs of patches have been intersected, this list will contain all the trimmed intersection curves from that patch intersected with all patches from the other solid. This ends the first stage of the boundary evaluation algorithm. The patches, each with its own trimmed intersection curves, are passed on to the second stage.

## 5.7 Merge Curves

Merging curves is the first step of the second stage of the boundary evaluation algorithm. Unlike the first stage, which dealt with pairs of patches, the second stage generally considers only one patch at a time.

Merging curves combines the adjacent clipped intersection curves found in the first stage to form *joincurves*. A joincurve is a simply connected sequence of clipped intersection curves. That is, the ending point of one curve is the starting point of the next. The overall structure is similar to that of segments in curves, except that the curves do not all come from the same polynomial. Like regular curves, a joincurve can either form a loop or have a separate start point and end point, and cannot self-intersect. The set of trimming curves, for example, can be thought of as a single joincurve that forms a loop. A single curve can be declared a joincurve, and additional curves can be added on to a joincurve (if they share the same starting or ending point as the joincurve). Joincurves can also be merged with each other. Note that if two joincurves have the same starting (or ending) points, then one of the joincurves can be *reversed* so that its starting point is then its ending point. This lets the two joincurves be merged into a single joincurve. Reversing a joincurve means reversing the order of the curves that make up the joincurve, and reversing the individual curves (Section 3.2.2).

After finishing the first stage of the boundary evaluation algorithm, each patch contains a number of clipped intersection curves. Together, these intersection curves represent the trimmed region of the patch intersected with the entire other solid. The intersection curves are combined together to form joincurves. When all possible intersection curves have been merged into joincurves, each joincurve either is a loop or starts and ends on a trimming curve.

Figure 5.8 demonstrates a simple example of curve merging. In the figure, a patch contains six intersection curves. These six curves are merged to form two joincurves.

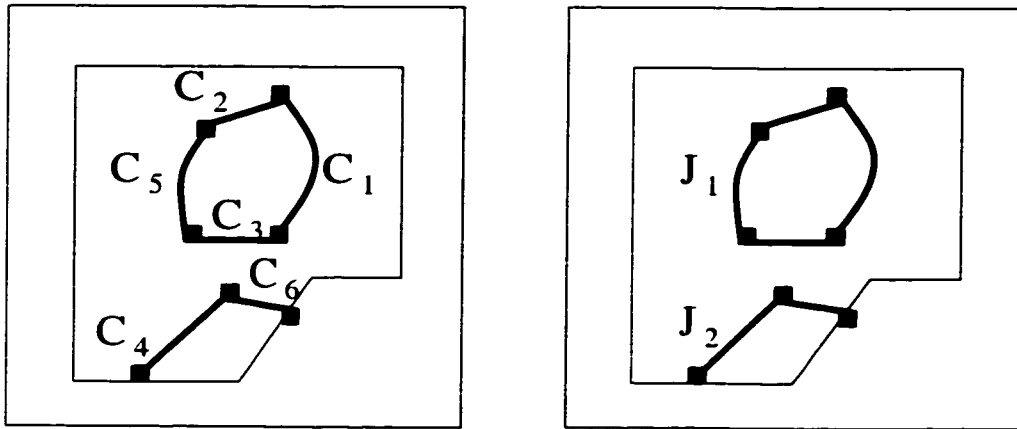


Figure 5.8: **Curve merging.** At left, six clipped intersection curves (designated  $C_i$  and in thick lines) were found in the trimmed region (designated by thin lines).  $C_1$ ,  $C_2$ ,  $C_3$ , and  $C_5$  were merged to form joincurve  $J_1$ , and  $C_4$  and  $C_6$  were merged to form joincurve  $J_2$ , at right.

one of which is made from four intersection curves, the other from two.

To perform curve merging, first declare each intersection curve to be a joincurve. An endpoint (either the starting point or ending point) of a joincurve is declared to be “closed” if the joincurve forms a loop, or if that endpoint lies on a trimming curve (i.e. it is an intersection point). Otherwise the endpoint is declared to be “open.” When all remaining endpoints are closed, curve merging is finished.

Curve merging proceeds by looking for pairs of open endpoints that could potentially be equal. By potentially equal it is meant that the points have overlapping bounding intervals. It is generally not necessary to perform a complete equality test. Every open endpoint is guaranteed to match up with at least one other open endpoint. If  $A$  is an open endpoint, consider all other open endpoints. If there is only one other open endpoint,  $B$ , that has an overlapping interval, then those two points must be equal. The joincurve with endpoint  $A$  should then be merged with the joincurve with endpoint  $B$ . If  $A$  and  $B$  are on the same joincurve, then just mark  $A$  and  $B$  closed, as the joincurve is forming a loop. Note that although this is an  $O(n^2)$  matching operation, in actual implementations  $n$  (the number of open endpoints) will be small, and the asymptotic bound is not significant. In the unlikely event that no pair of open endpoints is found to match (i.e. every endpoint is potentially equal to more than one other endpoint), shrinking the remaining open endpoints and repeating the procedure will eventually lead to all points matching.

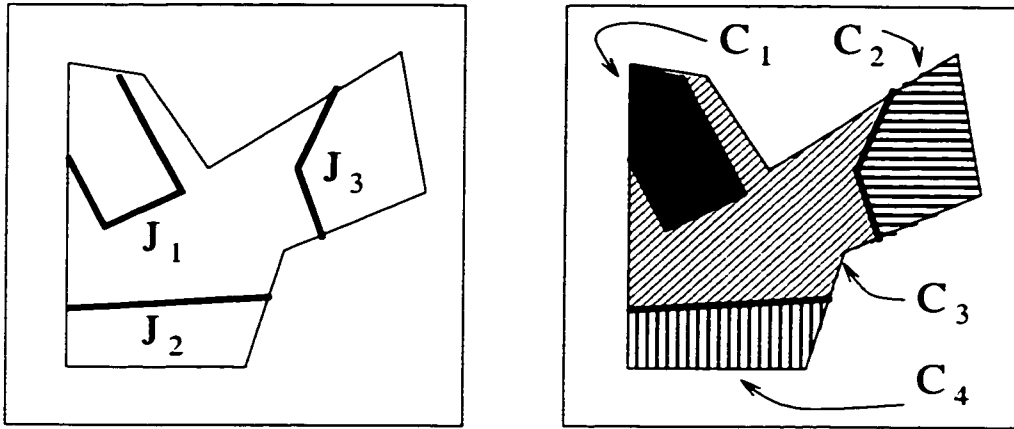


Figure 5.9: **Patch partitioning.** The three joincurves ( $J_i$ ) at left induce a partitioning into four partitions ( $C_i$ ), as shown at right.

## 5.8 Partition Patches

A *partition* is a subregion of a patch that has a boundary defined by the trimming curves and/or joincurves of the patch. If there are no joincurves (because the patch does not intersect with any patch in the other solid), the entire patch is declared to be a single partition. The next step of the boundary evaluation algorithm is to compute each individual partition, and the topological connectivity between them.

First, determine whether any of the joincurves forms a loop. If so, subdivide the patch into two patches, in order to break every loop. Loop breaking is described in detail in Section 5.8.1. The output of loop breaking is two (or more) patches, each of which is treated independently.

With all loops broken, each joincurve starts and ends on the trimming boundary. If there are  $n$  joincurves in the patch, then the patch will be broken up into  $n + 1$  partitions. An example is shown in Figure 5.9.

The immediate goal is to determine the boundary of each partition. This boundary is a list of curves, taken from both the joincurves and the trimming curves. The procedure is as follows:

1. Subdivide the trimming curves at the points of intersection with the joincurves. These points are the endpoints of the joincurves, which are the intersection points found earlier that have already been inserted into the trimming curve. Also, keep a record of which original trimming curve each of the subdivided

curves comes from. This information is used later (Section 5.8.2).

After the trimming curves are subdivided, each trimming curve contributes to the boundary of exactly one partition, and each joincurve contributes to the boundary of exactly two partitions (since it is assumed that there are no degeneracies). One partition contains the joincurve in the forward direction, and one partition contains the joincurve in the reverse direction.

2. Allocate a marker for each trimming curve that registers the trimming curve as either “visited” or “unvisited,” depending on whether it has contributed to a partition yet. Initialize all markers to “unvisited”. Also, allocate registers for a *begin point* and *current point* that point to individual points, and a *previous joincurve* register that points to a joincurve. Finally, for each partition, allocate a list (initially empty) of the curves that make up the partition boundary.
3. Begin to trace out a partition boundary. The partition boundary will be a list of curves. First initialize a new partition:
  - (a) Find a trimming curve marked “unvisited.”
  - (b) Set the begin point to be the starting point of this trimming curve.
  - (c) List this trimming curve as the first curve in the partition boundary.
  - (d) Set the marker for this trimming curve to “visited.”
  - (e) Set the current point to be the endpoint of this trimming curve.
  - (f) Set the previous joincurve to be undefined.
4. Now trace out the rest of the partition boundary. *Repeat* the following until the current point is the same as the begin point:

Examine whether the current point is an endpoint of any joincurve, *other than* the previous joincurve (if defined).

  - If the current point is an endpoint of a joincurve:
    - (a) Add the curves comprising the joincurve to the partition boundary list. If the current point is the starting point of the joincurve, add the curves in order, otherwise reverse the joincurve (Section 5.7) before adding it to the partition boundary list.
    - (b) Set the previous joincurve register to this joincurve.

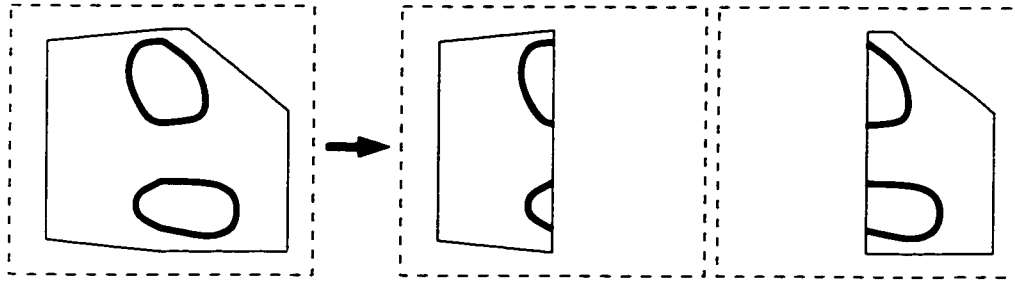


Figure 5.10: **Two loops broken by one patch subdivision.** The joincurves of the original patch are shown at left, in bold. The trimming curves are shown by thin lines and the patch domain boundary by dashed lines. At right, the patch has been broken into two subpatches, neither with a loop.

- (c) Set the current point to the opposite endpoint of this joincurve.
- If the current point is *not* an endpoint of a joincurve:
  - (a) The point is the starting point of a trimming curve. Add this trimming curve to partition boundary list.
  - (b) Set the marker for this trimming curve to “visited.”
  - (c) Set the current point to be the endpoint of this trimming curve.
- 5. Check to see whether any of the trimming curves is still “unvisited.” If so, return to step 3. Otherwise, all partitions have been found.

Once all of the partitions are found for all of the patches in the solid, the topological connectivity between the partitions is determined. Although most of the topological information can be obtained during the partitioning process, it is not significantly less efficient to determine it afterward, and the description is much simpler. This topological information is described in Section 5.8.2.

### 5.8.1 Break Loops

Breaking joincurves that form loops is necessary since a patch is not allowed to have any internal loops (Section 3.1). Loop breaking involves subdividing one patch into two subpatches, each of which contains part of the joincurve. If there are many joincurves, the patches are just further subdivided until all loops are broken. Note that it is possible for a single subdivision to break many loops (Figure 5.10).

Subdivide along one of the the parameter lines. To pick an appropriate parameter value, examine the maximum and minimum extents of the joincurve that forms the loop. Choose a rational value between these extents. Form a line at that parameter value, say  $s = a$  (or  $t = a$ ), that is called the *split line*. Because the rational value is between the maximum and minimum extents, the split line passes through the loop.

Next intersect the split line with the trimming curves and the curves that comprise the joincurve. Insert all intersection points into both the split line and the trimming curve or curve in the joincurve. Form a new surface passing through the split line (Section 4.4), and use this surface to invert the points. Also intersect this surface with the adjacent patches from the same solid in order to invert the points found on the trimming curve. Note that these inversion processes are *not* as complicated as the process described in Section 5.4.2. Because all correspondence information between the curves in each domain is already known, the matching in three dimensions is not necessary. This principle is described in more detail in Section 5.8.2.

Assuming that there are no tangential intersections (and if there are, they are easily detected and a new split line chosen), the split line intersects both the trimming boundary and the loop an even number of times. Clip the split line to the trimmed region. Clipping the split line can be done directly – if the intersections with the trimming curves are ordered along the split line  $i_1, i_2, \dots, i_n$ , then the clipped curves are the segments between  $i_{2j-1}$  and  $i_{2j}$ , for  $j = 1, 2, \dots, n/2$ . These  $n/2$  split curves formed from the split line each become part of the trimmed region for the new subpatches.

Given the  $n/2$  split curves, the patch is subdivided into  $n/2 + 1$  new subpatches. This is illustrated in Figure 5.11. Each split curve forms part of the trimming boundary for two new subpatches. Trace the trimming boundaries for these new subpatches in a manner similar to that described above for determining partitions. Notice that since the subpatches occupy a smaller portion of the parametric space, the patch domain of each subpatch can be reduced, if desired. Reducing the patch domain may save computation if the output of this boundary evaluation is used as input to another boundary evaluation.

The next step is to subdivide the joincurves themselves. First subdivide the joincurve into a number of smaller joincurves, based on the intersection points with the split line. Note that this involves splitting the individual curves in the joincurve into different curves, as well. Each of these joincurves will belong to one of the subpatches. Determine the specific subpatch by choosing any point (except an endpoint) from each

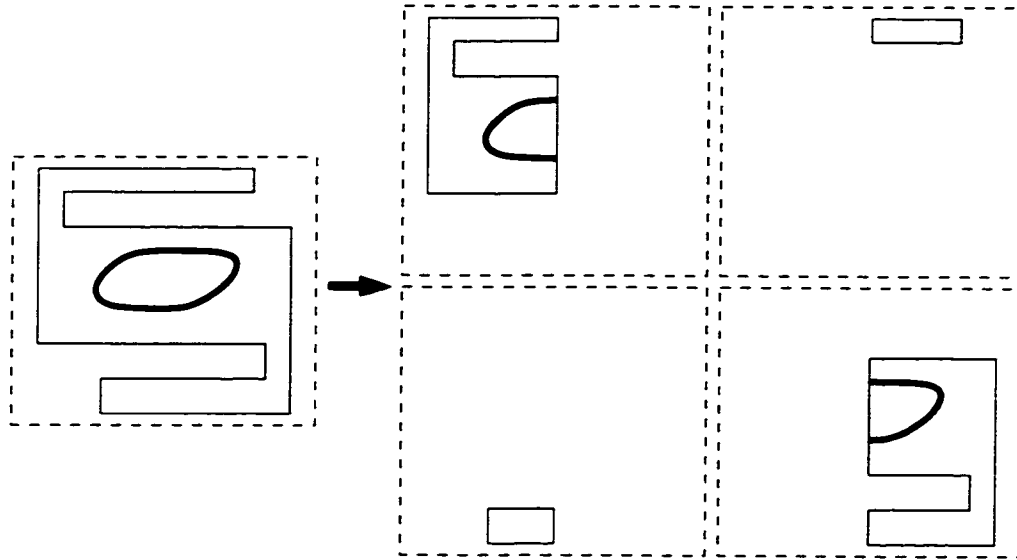


Figure 5.11: **Splitting one loop leading to many patches.** The split line has six intersections with the trimming curves, which are clipped to form three new curves in the trimming region. These three new curves become part of the trimmed boundary of the new subpatches. The single patch is broken into four subpatches.

joincurve and performing point location relative to the trimming boundaries of the subpatches. Store the joincurve in whichever subpatch the point is located in.

Finally, update topological information, both for this patch and for the patches adjacent to the original patch. The adjacency information along each trimming curve must be updated. For any of the original trimming curves (or their subparts) in the patch being split, the adjacent patch is the same as in the original solid. Along any of the new trimming curves in the subpatches (formed from the split curves), the adjacent patch is the other subpatch that shared that split curve. Associate the two curves (one in each subpatch) formed from a split curve.

Another topological change is in the patches that are adjacent to the original patch. Adjust the adjacency information in each of those patches to point to one of the new subpatches. In most cases, this relationship is straightforward. The only area of interest is along the trimming curves that were subdivided as a result of the patch splitting. Divide each of these trimming curves into two separate trimming curves at the inverted point. The adjacency information for each of the two new curves is different (each pointing to a different subpatch). This is illustrated in Figure 5.12.

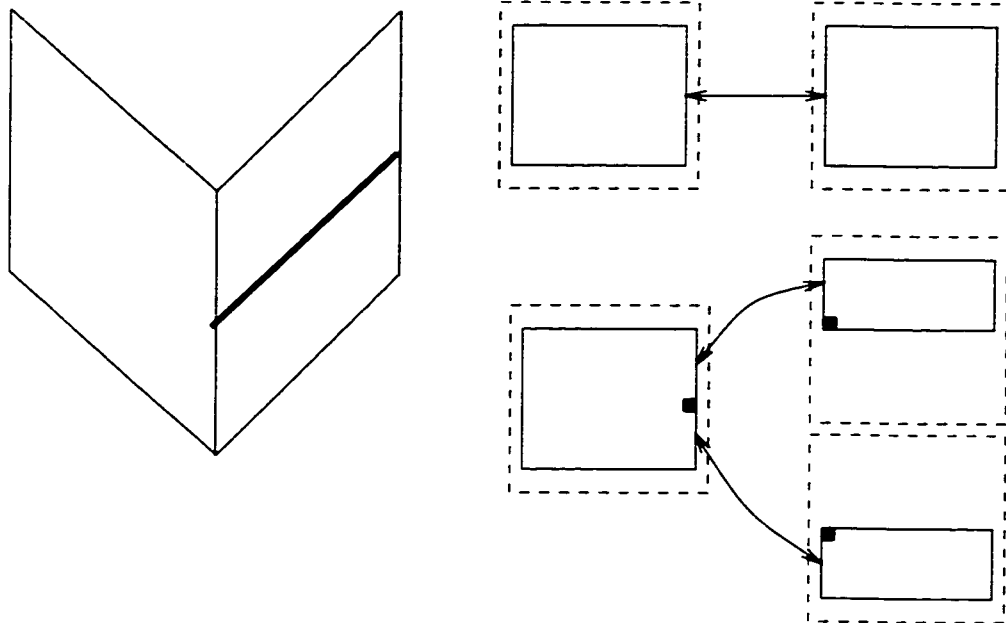


Figure 5.12: **Adjacency across patches.** At left, two adjacent patches in a solid. One of the patches is split along the dark line shown. At right, the topological relationship along adjacent curves. At top, the original patches are adjacent along the curves indicated. At bottom, one trimming curve on the left patch has been split into two trimming curves. The adjacency information is different for the two new curves.



### 5.8.2 Compute Topology

Once all of the patches have been partitioned, update the topological information. Two types of connections need to be determined: connections between partitions from one patch, and connections between partitions from different patches.

The topological information of interest for the partitions is similar to that for the patches, with one additional piece of information. Besides keeping track of which partition is adjacent along each boundary curve (as is done for trimming curves in patches), also keep a record of whether that boundary came from a joincurve or from a trimming curve. The distinction is useful when classifying components (Section 5.9).

For the curves arising from the joincurves, the adjacent partition is simply the other partition from the same patch that involved that joincurve. For the trimming curves, the process is slightly more complicated. Each of the original trimming curves of the patch has been subdivided into a number of smaller trimming curves based on intersections with the joincurves. Because the joincurve is the intersection curve with the other solid, the intersection curve continues to the adjacent patch, so that trimming curve is also subdivided in the adjacent patch (Figure 5.13).

Let  $P_1$  and  $P_2$  be adjacent patches in a solid. Assume that the patches are adjacent along trimming curve  $A$  in  $P_1$ 's domain, and trimming curve  $B$  in  $P_2$ 's domain. After partitioning,  $A$  and  $B$  have been broken up into a number of subcurves,  $A_i$  for  $i = 1, 2, \dots, m$ , and  $B_j$  for  $j = 1, 2, \dots, n$ . It is assumed that the intersection curve with the other solid does not intersect an edge of the solid tangentially, as that is considered an input degeneracy. Thus, the intersection curve carries over to the adjacent patch, subdividing the edge in the same way in each patch. Thus  $m = n$ .

Even though the trimming curves in each patch are subdivided independently of one another, they are subdivided at the same points. The adjacency along all such trimming curves follows directly from knowing the adjacency along the original trimming curve. This is demonstrated in Figure 5.13.

The final result can be stored in a graph structure, if desired (it is helpful for component classification in Section 5.9). Each node of the graph represents one partition. There are two types of edges that will be referred to as *solid edges* and *dashed edges*. Solid edges represent adjacent partitions along original trimming curves. Dashed edges represent adjacent patches along joincurves (i.e. intersection curves with the other solid). An example of such a graph is given in Figure 5.14.

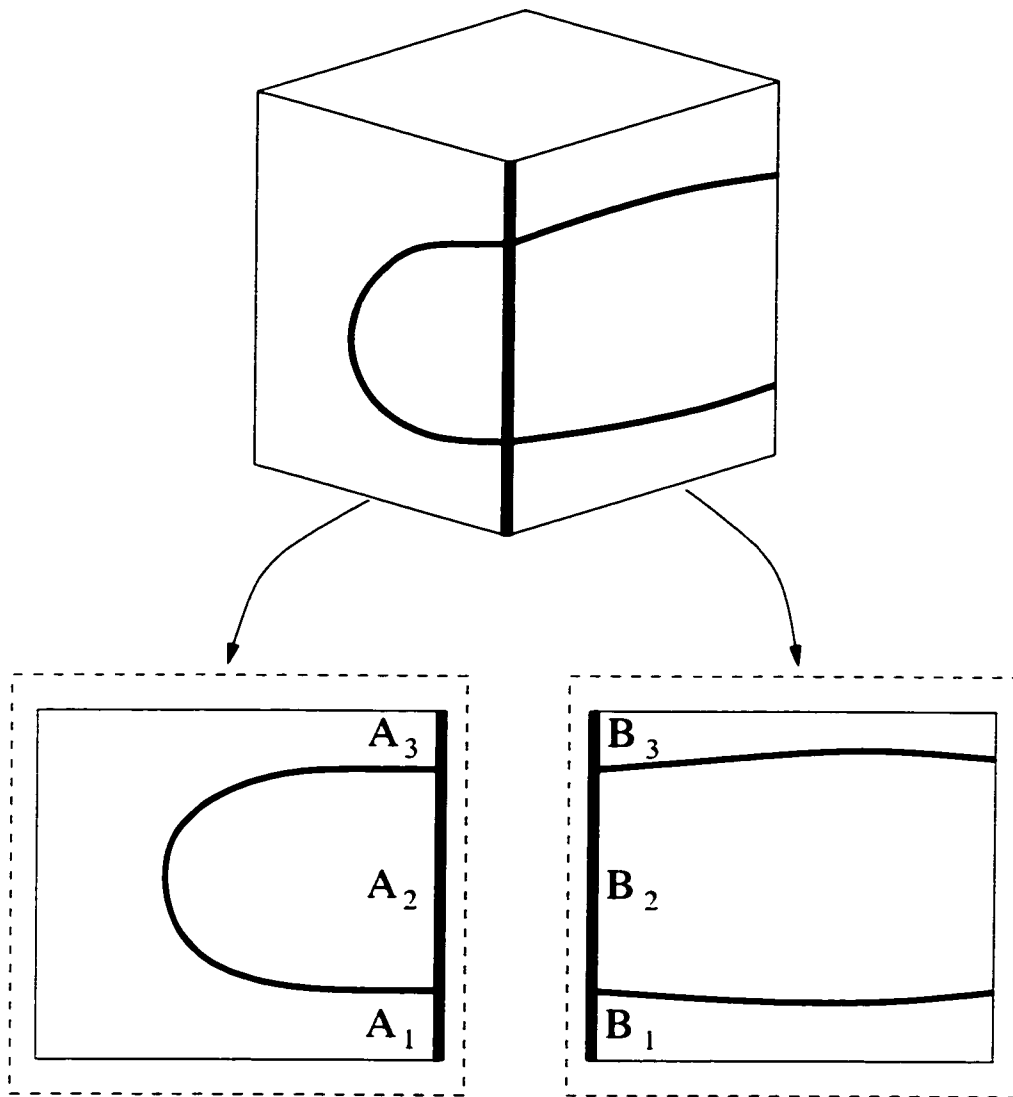


Figure 5.13: **The continuation of an intersection curve across patches.** At top is the original solid, with the intersection curve shown in medium thickness. One edge of the original solid is highlighted in bold. At bottom are the two patches. The domain is shown with dashed lines, the trimming curves with solid lines. The edge highlighted at top is also highlighted in each domain. This edge has been subdivided into equivalent regions in each patch domain, even though the curve was subdivided in each domain independently.

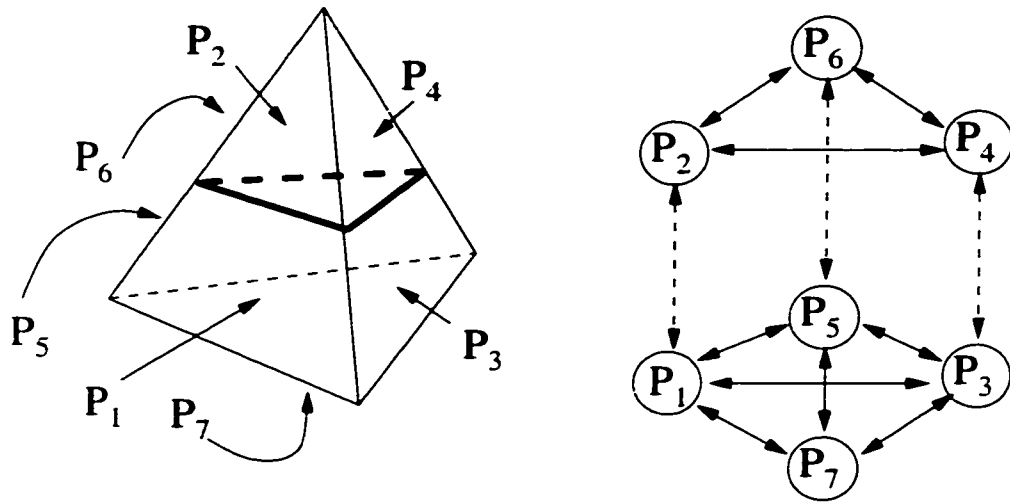


Figure 5.14: **Adjacency of partitions.** At left, a solid after partitioning. The dark curve shows the intersection curve, and the partitions are labeled ( $P_5$  and  $P_6$  are at back.  $P_7$  is underneath). At right, the associated topological graph for the partitions. Solid lines indicate adjacency along original trimming curves, and dashed lines along the intersection curve.

## 5.9 Classify Partitions

The third step of the second stage of the boundary evaluation algorithm is classifying partitions. Classifying partitions means locating the partition with respect to the other solid.

To perform classification, use point generation and point location (Section 4.3). Choose any partition from the first solid,  $S_1$ . Generate a point in that partition, and locate it with respect to  $S_2$ . Likewise, choose any partition from  $S_2$ , and generate and locate a point relative to  $S_1$ . If the point is inside the other solid, the entire partition that it came from must be inside the other solid, and if the point is outside, the entire partition must be outside.

Finally, propagate this information to the other partitions. This is done using the topological structure computed after partitioning. Components connected by a solid edge in the graph have the same location (inside or outside). Components connected by a dashed edge have an opposite location. For example, in Figure 5.14, if  $P_1$  is located outside of the other solid, then  $P_3$ ,  $P_5$ , and  $P_7$  must also be outside, since they are connected by solid edges.  $P_2$ ,  $P_4$ , and  $P_6$  must be inside, since they are connected via dashed edges to partitions that are outside. Any graph routine (e.g.

depth-first search) can be used to propagate this location information.

If any partition is not located in this manner, it means that the original solid is composed of two or more disjoint parts. Recall that a solid may consist of many disjoint collections of surfaces. In such a case, classify one partition from another part of the solid (i.e. one partition that was not successfully located), and propagate the information. Continue until every partition is labeled either inside or outside of the other solid.

## 5.10 Build Final Solid

The final step of the boundary evaluation algorithm is to stitch together partitions to form the final solid. From each solid, choose one set of partitions (either inside or outside). Which set to choose depends on the Boolean operation:

- *Union* ( $S_1 \cup S_2$ ): Keep all partitions from  $S_1$  that lie outside  $S_2$ , and all partitions of  $S_2$  that lie outside  $S_1$ .
- *Intersection* ( $S_1 \cap S_2$ ): Keep all partitions from  $S_1$  that lie inside  $S_2$ , and all partitions of  $S_2$  that lie inside  $S_1$ .
- *Difference* ( $S_1 \setminus S_2$ ): Keep all partitions from  $S_1$  that lie outside  $S_2$ , and all partitions of  $S_2$  that lie inside  $S_1$ .

Once these partitions are selected, convert the data structure for each partition to a patch data structure. The surface and domain of the new patch are the same as for the patch that the partition came from (i.e. the parent patch). It may be possible to reduce the domain, which can save time in the future. The patch boundary is determined as follows:

- The trimming curves are the curves that formed the partition boundary.
- The adjacent surface depends on whether the new trimming curve is (part of) a trimming curve from the parent patch, or (part of) an intersection curve.
  - If the new curve comes from an original trimming curve, then the adjacent surface is the same as the original surface.
  - If the new curve comes from an intersection curve, then the adjacent surface is the surface of the patch (in the other solid) that formed that intersection curve.

- The adjacent patch information depends on whether the topological connection for that curve in the partition is a solid edge or a dashed edge.
  - If the connection is a solid edge (i.e. the curve is from a trimming curve in the parent patch), then the adjacent patch is one from the same solid. The adjacent patch is the patch formed from the adjacent partition along that curve.
  - If the connection is a dashed edge (i.e. the curve is from an intersection curve), then the adjacent patch is from the other solid. In this case, use the associated curve information for the curve. Recall that the curves were associated near the end of the curve clipping stage (Section 5.6). The associated curve is a boundary curve in one and only one of the kept partitions from the other solid. The patch formed from that partition is the adjacent patch.

Thus, the final solid is computed in the same format as the input solids.

# Chapter 6

## Speedups

Direct use of exact computation often results in implementations that are too slow to be practical. This chapter describes a number of speedups that improve the efficiency of computations while maintaining exactness.

The boundary evaluation algorithm described in the previous chapters relies on an exact rational number library. The efficiency of this library has a significant and direct influence on the efficiency of the overall boundary evaluation algorithm. While speeding up exact rational number arithmetic will speed up boundary evaluation, such speedups are outside of the scope of this dissertation.

Exact rational number computation takes more time (in general) on numbers that use more bits than on numbers that use fewer bits. Reducing the number of bits used in an exact computation is the basis for many of the speedups described below. If few enough bits are used, the arithmetic operations can be performed directly in hardware. Assuring that a lower-precision operation is still exact is the key to using it within an exact computation.

Although any individual speedup, on its own, may result in significant efficiency improvement, combining speedups does not necessarily combine the improvements. One reason is that several speedups may eliminate the same unnecessary computations (i.e. the speedups are not independent). For example, floating-point filters (Section 6.4.1) and bounding box comparisons (Section 6.2) both work well on the same types of problems. A second reason why combining speedups may not result in as much gain as desired is that certain speedups produce conflicting goals. For example, lazy evaluation (Section 6.1) encourages intervals surrounding points to be large, while floating-point guided computation (Section 6.4.2) encourages them to be small. The best balance between various speedups is dependent on the problem and the particular implementation, and must be adjusted accordingly. In any case, combining

the speedups described here has resulted in significant efficiency improvements to the boundary evaluation implementation. Some results are described in Chapter 8.

## 6.1 Lazy Evaluation

Lazy evaluation refers to postponing time-consuming operations as long as possible, in the hope that the operation can be avoided entirely or a simpler, less time-consuming operation can be substituted for it. Lazy evaluation fits well with the concept of exact computation being a way to guarantee that any decision based on numerical data is correct. Often, such decisions can be guaranteed using only limited information, and the result of a decision may mean that certain numerical data is no longer needed. If the evaluation of that numerical data had been postponed, then the evaluation operations are avoided entirely.

Lazy evaluation can be incorporated into exact computation in several ways. The most obvious is in the representation of algebraic numbers, and thereby points. Recall that an algebraic number is represented as the root of a polynomial within an interval. As long as the interval is small enough to ensure that it contains only one root of the polynomial, the algebraic number is specified exactly.

Comparisons between algebraic numbers, then, can be first made on the basis of comparing their intervals. Only when the intervals overlap is more computation required. When the intervals overlap, they can often be reduced in size until they no longer overlap. In any case, the interval only needs to be made small enough to make a decision at a particular time. The interval can be reduced in the future, if later computation requires more precision. Thus, the time-consuming operations of shrinking the intervals to a small width and exact algebraic number comparison may be avoided.

The boundary evaluation algorithm uses lazy evaluation to a lesser extent in the definition of curves. Curves are initially broken only as much as is necessary to resolve curve topology. As later computations require, new points can be inserted into the curve. These new points break the curve into smaller segments, thus making it better defined, but only to the extent needed at one time.

Lazy evaluation is also useful in exact computations where algebraic numbers are used to construct some geometric object or other number. Such constructions do not arise in the boundary evaluation algorithm, but they can arise in other problems. An example is when two algebraic numbers must be added. In such cases, a lazy

evaluation approach would store the new number as a tree, with the two original algebraic numbers as leaves. When more precision is required for the new number, the precision of the subtrees (in this case the two leaves containing the original algebraic numbers that were added) must be increased. This approach is used in both the LEDA [14] and Core [55] libraries.

There are other ways of incorporating lazy evaluation into the boundary evaluation algorithm. For example, rather than resolving curve topology, the curves could perhaps be expressed simply as algebraic plane curves, with curve topology resolved only as necessary. It is not clear whether such an approach would actually save computation, however, since curve topology would probably need to be resolved eventually. Other applications of lazy evaluation might involve restructuring the algorithm itself to treat much of the numerical data symbolically.

## 6.2 Quick Rejection

Quick rejection techniques attempt to avoid all computations that do not lead to a useful result. For example, imagine two patches that do not intersect. If the surfaces corresponding to those patches intersect, the boundary evaluation algorithm will go through the entire first stage for that pair of patches, spending significant time, only to have the entire curve clipped away at the last step. If a simple test is performed ahead of time to conclusively eliminate the possibility of an intersection, all that work is avoided.

### 6.2.1 Interval Arithmetic

While interval arithmetic (Section 2.3.2.1) alone does not provide the efficient guaranteed root isolation needed in an exact approach, interval methods (and, similarly, affine arithmetic methods) are useful for quick rejection. An example of interval methods in boundary evaluation is in determining whether the intersection curve between two patches lies in a patch domain. Treating the patch domain as a 2D interval and evaluating the intersection curve over that interval determines whether the algebraic plane curve might pass through the patch domain. This computation eliminates cases that bounding boxes do not. Interval methods can be used in other parts of the algorithm as well, including generating and testing bounding boxes (Section 6.2.2).

Affine arithmetic (Section 2.3.2.1) can provide tighter bounds on computation



than direct interval arithmetic, but can be much slower. Affine arithmetic can be performed efficiently, however, when evaluating a polynomial. Assume that one wishes to evaluate a polynomial,  $f(x)$ , over an interval,  $[a, b]$ . Affine arithmetic represents the interval as  $c + d\varepsilon_1$ , where  $c = (a + b)/2$ ,  $d = (b - a)/2$ , and  $\varepsilon_1 = [-1, 1]$ . A simple variable substitution yields a new polynomial, univariate in  $\varepsilon_1$ , i.e.  $g(\varepsilon_1) = f(c + d\varepsilon_1)$ . Since  $\varepsilon_1^i = [-1, 1]$  for odd  $i$  and  $[0, 1]$  for even  $i$ , a new interval can be easily found from  $g$ . The constant term of  $g$  gives the center of the resulting interval, while the width of the interval is found by summing up the absolute values of the other coefficients. A tighter interval can be obtained by taking into account whether the power of  $\varepsilon_1$  is even or odd. Thus, affine arithmetic is performed by making one variable substitution and then summing the absolute values of the resulting coefficients.

## 6.2.2 Bounding Boxes

The most common method of quick rejection is bounding boxes. A bounding box of a certain geometric object is a rectangle (usually axis-aligned) that is guaranteed to contain that structure. If another geometric object is outside of the bounding box, then the two geometric objects cannot touch. Often, interval arithmetic is a simple way to compute a bounding box.

Bounding boxes have many uses in the boundary evaluation algorithm. For example, a 3D bounding box can be placed around each patch. If two patches' bounding boxes do not overlap, then the patches cannot intersect. As is described in Section 3.2.1, a curve is made up of several segments, each contained within a bounding box. If a curve-curve intersection is to be computed, a test can first be made of whether any of the segment bounding boxes of the curves overlap. Bounding boxes are directly applicable to points, since the interval of each point effectively forms a bounding box. If the intervals around two points do not overlap, the points are not equal. If the points have identical intervals and are the roots of the same polynomials, they are equal.

When performing ray shooting as part of point generation or location (Section 4.3), bounding boxes are also useful. In two dimensions, each segment of a curve can be intersected with a horizontal or vertical line by determining whether the bounding box contains that particular coordinate value. In three dimensions, bounding boxes can quickly eliminate most ray-patch intersection tests.

Another way that bounding boxes are useful is in the approach to point inversion (Section 5.4.2). In that case, matching in three dimensions is performed by comparing

bounding boxes generated in three dimensions. Matching the bounding boxes is significantly faster than performing four-dimensional point inversion.

## 6.3 Simplifying Computation

Simplifying computation refers to substituting fast, simple computations for more complex ones. By analyzing the problem, a simple problem-specific approach can be found to suffice for a more general one.

### 6.3.1 Point Equality

One example of this is in comparing points for equality. At various stages in the boundary evaluation algorithm, points must be checked to see whether they are equal. Exact point comparison involves comparing the coordinates for equality, which involves finding the greatest common divisor of two polynomials and then finding the roots of the resulting polynomial. There are several simpler computations that can often be made first, providing significant speedups. If the points are in the same location in memory, they are the same point. The intervals around points can be used to quickly verify that two points are not equal. There are many cases where one point is guaranteed to be the same as one out of a set of other points. In such cases, finding which point is equal is only a matter of comparing (and possibly shrinking) intervals. By combining these methods, it is often possible to eliminate the test for exact point equality, saving significant time.

### 6.3.2 Curve-curve Intersection

Another example of simplified computation is in curve-curve intersection. For the standard CSG primitives, one of the two curves being intersected during boundary evaluation is often a trimming curve that is a vertical or horizontal line. In such cases, there is no need to perform a complete curve-curve intersection (such as is described in Section 4.1). Instead, the horizontal or vertical line defines the value for one coordinate. This value is substituted into the polynomial of the other curve, and the roots of the resulting univariate polynomial are found, yielding intersection points that have one coordinate as a rational number, the other as an interval. Also, when both curves are lines (not necessarily horizontal/vertical), the intersection point has rational coordinates that can be found directly and exactly.

### 6.3.3 Reducing Intervals

A third example of simplified computation is in reducing the interval surrounding an algebraic number. For an individual algebraic number, the defining polynomial is usually negative on one side of the number, and positive on the other. The only exception is when the number is a root of even multiplicity, in which case the full Sturm sequence must be used. Assuming the usual case, however, the interval around the algebraic number can be cut (Section 3.3.2.1) using only a single sign of a polynomial test. Depending on the sign of the polynomial at the cut value, either the upper or lower end of the interval is changed to the cut value.

### 6.3.4 Lower-Dimensional Computation

A fourth example of simplified computation is in reducing high-dimensional problems to lower-dimensional problems. Patch-patch intersections can be seen as a problem involving seven variables (two for each parametric patch domain, and three for the 3D space). Treating the entire problem using all seven variables at once makes many parts of the algorithm conceptually simpler, however performing seven-dimensional computations takes far too long, particularly with exact computation. Instead, all significant computations in patch-patch intersection are performed in two dimensions (the patch domain). Even in cases where it seems as if more dimensions are necessary, the problem can be reduced primarily to two dimensions. For example, point inversion (Section 5.4.2) is generally seen as at least a four-dimensional problem. Instead, the problem is restructured so that a series of two-dimensional operations are performed, followed by a simple 3D matching algorithm. This approach yields a much faster implementation than the equivalent four-dimensional computation.

Using a lower-dimensional formulation also carries over to the curve-curve intersection algorithm presented in Section 4.1. In that case, the two-dimensional problem is broken (after resultant computations) into a series of one-dimensional problems. The series of one-dimensional computations takes significantly less time than comparable two-dimensional computations (such as multivariate Sturm sequences).

## 6.4 Floating-point Evaluations

Usually, operations using standard floating-point arithmetic are much faster than equivalent exact computations. Besides the direct hardware and compiler support

for floating-point computation, the limited precision obtained by rounding allows computations to be performed quickly. Unfortunately, this rounding also introduces numerical error, making standard floating-point operations seemingly unusable in a system based on exact computation.

At the same time, standard floating-point based computations are usually close to the correct answer. Many systems are based on floating-point computations, and these systems work correctly in many cases. However, the error that accumulates in such systems makes it difficult to ensure accuracy and correctness, particularly in nonlinear geometric computations such as boundary evaluation. The goal is to use the speed of floating-point computation while ensuring exactness. Two ways this is done are described below.

### 6.4.1 Floating-point Filters

As mentioned in Section 2.3.2.1, floating-point filters are one method for speeding up exact computation. The basic idea of a floating-point filter is to perform the computation in floating-point, but keep a bound on the error that is accumulated. The first step is to convert exact data to floating-point data, along with a bound on the error introduced (machine precision). As each floating-point operation is performed, the error bound is increased to account for both propagated error from previous steps and new error introduced by rounding at that step. The final result is a floating-point number and a bound on the maximum error in that number. If that error is small enough to make a particular decision, then exact evaluation has been avoided.

Experience has shown that floating-point filters are not helpful for every exact computation, due to the rate of error growth in nonlinear computations. Furthermore, iterative floating-point routines (such as Newton's method for finding roots) can not be directly adapted to a floating-point filter approach, since they are not a single computation within which the error can be strictly bounded. Nevertheless, floating-point filters can be successfully applied to a few computations within boundary evaluation.

The most important way that floating-point filters are used in boundary evaluation is in determining the sign of a polynomial. This is a key operation in the evaluation of a Sturm sequence, as well as in the process of narrowing the interval surrounding an algebraic number (Section 6.3.3). Finding the sign of a polynomial can be turned into a floating-point filtered operation. The coefficients of the polynomial, as well as the value used in evaluating the polynomial, are converted to floating-point numbers

(with associated error). Error is propagated through the polynomial evaluation, and the final result is a number and an associated error bound. As long as the error bound is smaller than the absolute value of the number, the sign of the polynomial computation is conclusive.

Floating-point filters can be used more extensively. For example, the entire Sturm sequence computation could be performed in floating-point, generating polynomials with floating-point coefficients with bounded error. For fairly simple Sturm sequences, this has proved useful, but more often the error in those coefficients becomes too large to allow further computation to be guaranteed.

### 6.4.2 Floating-point Guided Computation

While floating-point filters are a well-known approach to increasing efficiency, another approach has proved to be just as useful, if not more, for nonlinear computation. This approach will be called *floating-point guided computation*. The basic idea is to use any floating-point method to estimate the result of a computation, then use an exact method to verify that the estimate is close to the true result. Floating-point guided computation does not preclude the use of floating-point filters for the exact portion of the computation. Floating-point guided computation is most applicable in schemes where an exact iterative process is used to gradually close in on a result.

Floating-point guided computation is especially helpful for univariate root finding. As a component of curve-curve intersection, univariate root finding is a key operation in the boundary evaluation algorithm. To use floating-point guided computation, first isolate the roots within some interval using any univariate root finding method. Newton's method or eigenvalue approaches are two possible methods for root isolation. Then, using an exact method such as univariate Sturm sequences, count the total number of roots in the interval. If this number is less than or equal to the number found by the floating-point method, then the floating-point estimates of the roots are temporarily considered valid. Each valid floating-point estimate can be verified by testing a small rational interval around that number. For example, assume that all roots of a polynomial over the range  $[0, 1]$  were to be found. Using a floating-point method, one root is found at the value 0.3589271. The interval  $[0.358, 0.360]$  can then be checked using an exact method to see whether there is a unique root of the polynomial in that interval. If so, then the root has been isolated to a certain precision much faster than it would have for bisection with purely exact arithmetic. Even though the floating-point estimate is not exact, it allows the root to be approximated



Figure 6.1: **Floating-point guided computation.** At left, a floating-point estimate (shown by the x) approximates the actual root (shown by the circle). At center, a bounding interval placed around the floating-point solution contains the actual root. At right, a smaller interval no longer contains the actual root.

much more quickly, yielding an exact interval.

As long as the interval contains the floating-point estimate, the estimate is considered valid. Valid estimates can continue to guide future computations when the interval needs to be shrunk (Section 3.3.2.1) at a later time. At some point (possibly even at the first attempt), the floating-point estimate will not be close enough to the actual root to provide a valid guide. In this case, the floating-point estimate lies outside the interval containing the root, and the interval has to be further reduced by the traditional bisection approach. In the worst possible case, the floating-point estimate is completely invalid (or one of the roots was not found by the floating-point method), and the standard bisection approach to root isolation/interval reduction is used. The only extra work in such a case is in generating the floating-point estimates (which is insignificant in time compared to the exact computations), and in testing the intervals (which is only a small amount of extra work compared to the numerous evaluations by the bisection method). Figure 6.1 illustrates the process of floating-point guided computation.

# Chapter 7

## Degeneracies

Although the focus of this dissertation is on eliminating numerical error, dealing with degeneracies is necessary for any approach to be considered robust. Degenerate situations can arise in many different ways. Far more types of degeneracies are possible when curved surfaces are involved than when only linear surfaces are involved.

This chapter gives a brief discussion of degeneracies in the context of boundary evaluation. This includes the types of degeneracies that are possible, how these degeneracies manifest themselves within the approach to boundary evaluation described earlier, and possible ways that these degeneracies can be handled.

### 7.1 Types of Degeneracies

Degeneracies can be classified into several different categories. As is mentioned in Section 2.1.3.2, degeneracies will be grouped into three different categories. These are:

- *input* degeneracies, which are degenerate configurations of input data.
- *unpredictable* degeneracies, which are degenerate situations that come from arbitrary decisions made in the algorithm itself.
- *intentional* degeneracies, which are intentionally constructed as part of the algorithm.

Recall that with all types of degeneracies, a slight perturbation of the data will remove the degeneracy.

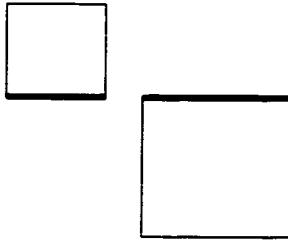


Figure 7.1: **A borderline degeneracy.** The two squares do not meet at a degeneracy, but the faces marked in bold lie on the same line.

The use of exact computation in the boundary evaluation algorithm deals with intentional degeneracies appropriately. All further discussion focuses on dealing with the other two types of degeneracies appropriately.

Certain degeneracies, called *borderline degeneracies* can be viewed as either input degeneracies or unpredictable degeneracies. A borderline degeneracy occurs when the faces and edges of the input solids do not meet in a degenerate configuration, but the infinite extension of the surfaces and curves do. For example, Figure 7.1 shows a 2D example where the faces of two squares lie on the same line. A boundary evaluation algorithm may or may not need to use the extended surface and edge definitions. If it does, it will encounter the same types of situations as would arise with an input degeneracy. For example, a boundary evaluation algorithm intersecting the bold faces from Figure 7.1 could encounter an overlapping surfaces degeneracy.

### 7.1.1 Input Degeneracies

The enumeration of all possible input degeneracies is difficult. Several degenerate conditions have no bearing on the algorithm being used. No general method exists for enumerating all degeneracies that can affect a particular algorithm. Each algorithm must be individually analyzed to determine where a degenerate situation can arise.

For the boundary evaluation algorithm presented here, potential input degeneracies are enumerated by considering the various ways that points, curves, and surfaces can interact in non-generic ways. Following that, the ways that each of these situations can arise in terms of the input solids are described.



	Surface	Curve	Point
Surface	<b>2:</b> Overlapping Surfaces <b>1:</b> Surfaces Tangent Along a Curve <b>0:</b> Surfaces Tangent at a Point	<b>1:</b> Curve Lying on a Surface <b>0:</b> Curve Tangent to a Surface	<b>0:</b> Point Lying on a Surface
Curve		<b>1:</b> Overlapping Curves <b>0:</b> Intersecting Curves	<b>0:</b> Point Lying on a Curve
Point			<b>0:</b> Coincident Points

Table 7.1: **Possible degeneracies.** The entries show the possible types of degeneracies that can arise between an object of the type given in the row and one given by the column. The bold number in front of the degeneracy gives the order of the degenerate intersection. The way that these situations arise is discussed in Section 7.1.1.2

### 7.1.1.1 Degenerate Configurations of Parts of Solids

Each solid is composed of geometric objects of various order: points (order 0), curves (order 1), and surfaces (order 2). When these objects are in general position, only two mutual interactions are possible. Two surfaces can meet transversely along a set of curves, and a curve and a surface can meet transversely at a set of points. No other interactions are possible if the objects are in general position.

Degeneracies occur in two ways. First, a degeneracy occurs when two geometric objects interact that should not (e.g. a point and a surface). Second, a degeneracy occurs when one of the two valid interactions is not transverse (e.g. a curve meets a surface tangentially at a point, instead of the curve passing through the surface at that point). The various types of degeneracies are summarized in Table 7.1.

Curves can be considered the intersection of two surfaces. Recall that each curve in an input solid is an edge formed from the intersection of two faces of the input solid. Similarly, points can be considered the intersection of three surfaces. Generically, two surfaces meet at curves, and three surfaces meet at points (thus we have the two generic interactions). Four or more surfaces do not meet, generically. This idea can

be used to group together many of the types of degeneracies mentioned in Table 7.1 into other categories.

Eight of the ten degenerate intersections can be viewed as two or more surfaces with a non-generic order of intersection. These are:

- *Two surfaces not meeting along a curve:*
  - Overlapping Surfaces
  - Surfaces Tangent at a Point
- *Three (or more) surface meeting at a curve:*
  - Curve Lying on a Surface
  - Overlapping Curves
- *Four (or more) surfaces meeting at a point:*
  - Point Lying on a Surface
  - Intersecting Curves
  - Point Lying on a Curve
  - Coincident Points

The other two types of degeneracy (surfaces tangent along a curve and curve tangent to a surface) occur when there is a seemingly generic intersection (i.e. two surfaces meet at curve or three at a point), but the intersection is tangential instead of transverse. Tangential intersections can occur between the surfaces that cause the other eight degeneracies, as well. For example, two intersecting curves is equivalent to a common intersection of four surfaces. One of those surfaces might intersect another only tangentially at the point. Also, the case of surfaces being tangent along a curve includes cases where even one point along that curve is a point of tangency. This results in an intersection between the two surfaces that contains either a cusp or a self-intersection.

#### **7.1.1.2 Degenerate Positions of Input Solids**

Within boundary evaluation, each of the cases from Table 7.1 can arise from input solids given in a degenerate configuration. Following, examples are given of the types of degenerate solid positions that can result in each of the underlying degenerate configurations. Wherever possible, linear examples are used to illustrate the case.

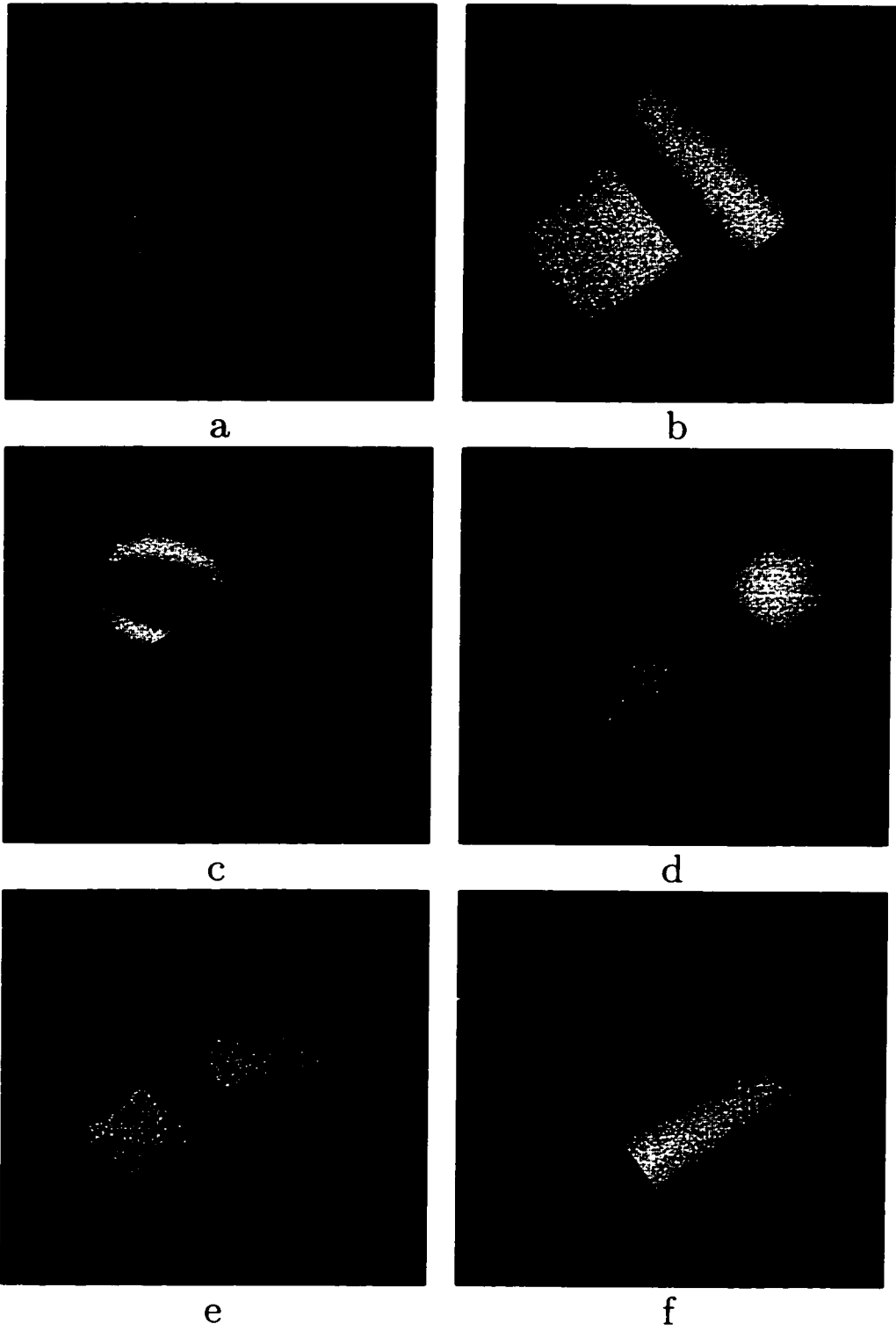
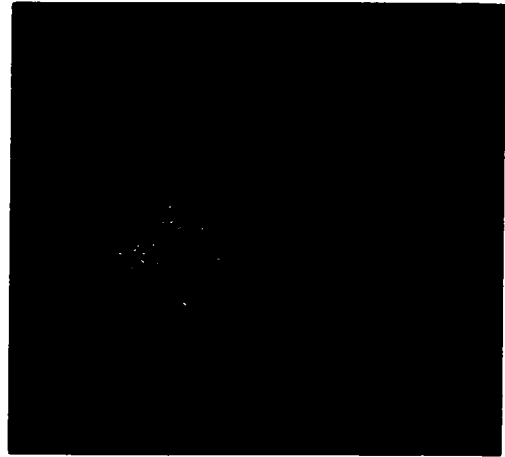


Figure 7.2: **Examples of degenerate input situations.**



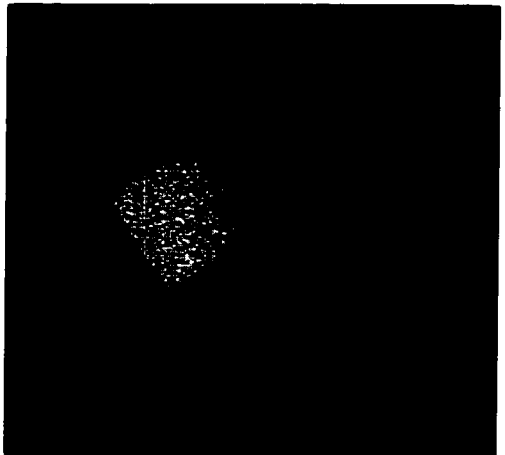
g



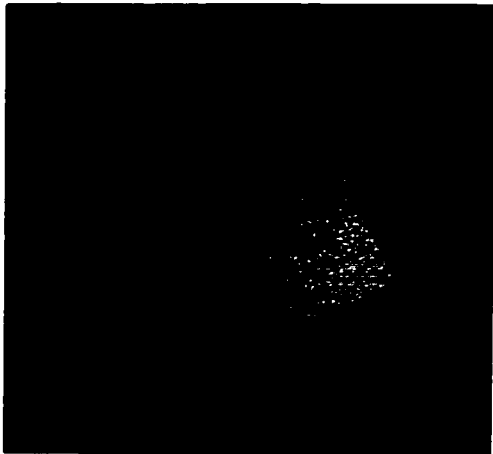
h



i



j



k

Figure 7.3: More examples of degenerate input situations.

- *Overlapping Surfaces:* Instead of intersecting transversely, the two solids have overlapping faces. Example 7.2(a) shows one such configuration.
- *Surfaces Tangent Along Curve:* There are two possible ways this can arise. In one case (Example 7.2(b)), the face of one solid lies entirely on one side of a face from the other solid. The surfaces are tangent along the entire intersection curve. In the other example (Example 7.2(c)), the surfaces are tangent at a point along the intersection curve. This yields an intersection curve that has a cusp or a self-intersection.
- *Surfaces Tangent at Point:* The solids meet at only one point. Example 7.2(d) illustrates this case.
- *Curve Lying on Surface:* As illustrated by Example 7.2(e), an edge of one solid lies on a face of the other solid.
- *Curve Tangent to Surface:* The edge of one solid intersects the face of the other solid only tangentially. Example 7.2(f) illustrates this case.
- *Point Lying on Surface:* Example 7.3(g) shows an example of the vertex of one solid just touching the face of the other solid.
- *Overlapping Curves:* When the edges of two solids overlap, this case occurs, as shown in Example 7.3(h).
- *Intersecting Curves:* When an edge of one solid just touches an edge of the other solid, this case can occur, as illustrated in Example 7.3(i).
- *Point Lying on Curve:* Example 7.3(j) shows an example of the a vertex of one solid lying on an edge of another.
- *Coincident Points:* This case results when a vertex of one solid is exactly the same as a vertex of the other solid. Example 7.3(k) is one such possibility.

Depending on the configuration of the solids, these degeneracies may manifest themselves in different ways. The case of surfaces tangent along a curve illustrates this fact. Particular ways that these degeneracies manifest themselves in the boundary evaluation algorithm are discussed in Section 7.2.

### 7.1.2 Unpredictable Degeneracies

By their nature, it is difficult to determine what types of unpredictable degeneracies may arise. In boundary evaluation, unpredictable degeneracies might arise from the choice of parameterization of the boundary of the solid or from the choice of a direction for ray-shooting. Finding potential unpredictable degeneracies involves analyzing every step of the algorithm to determine assumptions made at each step. Then, each assumption is analyzed to determine how a situation can arise such that the assumption may not hold.

It is important to distinguish unpredictable degeneracies from intentional degeneracies that have not been taken into account. For example, it may be assumed that no two points generated in the boundary evaluation algorithm are equal. Generically, if a set of random points are chosen, no two are equal, so this might seem to be a safe assumption. However, in the boundary evaluation algorithm, equal points are often generated in separate parts of the algorithm (this is what allows curve matching). There are far more subtle cases that can arise. Failure to recognize and account for such possibilities is a failure in algorithm design rather than a failure due to an unpredictable degeneracy.

Following is a list of some of the basic sources of unpredictable degeneracies. This list is likely to be incomplete, but illustrates some types of unpredictable degeneracies.

- *Patch Representations:* The way that a solid is broken up into patches is an arbitrary decision. As is illustrated in Figure 2.5, there are many ways that a cylinder can be broken up into patches. The particular patch breakdown introduces artificial edges and vertices into the solid. These artificial edges and vertices are subject to the same degenerate configurations as the real edges and vertices (the input degeneracies described in Section 7.1.1).
- *Parameterizations of Patches:* Within any one patch, the particular parameterization chosen may lead to a degenerate situation. For example, the parameterization may be such that the trimming curves are horizontal and vertical lines. While this makes some computations easier, problems can arise later on. For example, a horizontal or vertical ray used during point location (Section 4.3.2) might be at the same value as that line, or two separate points might have the same  $s$  or  $t$  coordinate.
- *Point Intervals:* The bounding interval for a point can be chosen arbitrarily. In theory, there is no reason to allow the interval bounding one point to have the

same endpoint(s) as the interval bounding another point. Because points may have been separated by a cut operation, however, two different points might have identical interval endpoints.

- *Points Lying on Cut Value:* Generically, a cut point should never be the true algebraic coordinate of a point. A point chosen at random would not have rational coordinates, much less have those coordinates at the cut value. The design of objects, however, is certainly not random, and points often have rational coordinates, which sometimes may lie at a cut value. An assumption that the point has irrational coordinates or does not lie at a cut value can lead to problems.

Note that the choices that result in an unpredictable degeneracy may be made that way for a specific reason. For example, a parameterization may be chosen because it is simple. A ray direction may be chosen in a coordinate direction to simplify later computation. Even though a different parameterization or a different ray direction could have been used, eliminating problems with the degeneracy, the resulting computation might be much more complicated. As long as the parameterization or ray direction is fixed, a degeneracy can arise.

## 7.2 Detecting Degeneracies

This section discusses methods for detecting when an input degeneracy is present, in the context of boundary evaluation algorithms. In many of these cases, detection requires the boundary evaluation algorithm to be modified. Because unpredictable degeneracies due to patch representation are indistinguishable from input degeneracies, those unpredictable degeneracies will also be detected. The other types of unpredictable degeneracy are usually handled on a case-by-case basis at the same time they are detected. They are discussed briefly in Section 7.3.1.

Below, the input degeneracies described in Table 7.1 are listed either individually or in groups as appropriate:

- *Overlapping Surfaces:* This degeneracy becomes apparent during the first step of boundary evaluation, finding the intersection curve. Since the surfaces overlap, their implicit forms must have a non-constant common factor. Substitution of the parametric form of one surface into the implicit equation of the other causes the implicit equation to vanish. Thus, there will be no intersection curve found.

- *Surfaces Tangent Along Curve:* This appears in different ways depending on whether the tangency is at an isolated point or along the entire intersection curve. If the surfaces are tangent at only one point along the intersection curve, then the intersection curve has a cusp or self-intersection at that point. That can be detected in the curve topology algorithm. If the tangency is along the entire curve, then the degeneracy is not apparent. A modification to the component classification step of the algorithm (as is described in Section 7.3.1) allows such a case to be both detected and handled.
- *Surfaces Tangent at Point:* When surfaces are tangent at a point, the intersection curve between the surfaces is a point solution in the patch domain (e.g.  $s^2 + t^2 = 0$ ). A point solution can be detected when resolving curve topology by finding an  $s$  turning point equal to a  $t$  turning point, but only if the curve-curve intersection method supports point curves. Multivariate Sturm sequences can be used to detect such cases.
- *Three (or more) surfaces meeting at a curve:* This includes the cases of a curve lying on a surface and overlapping curves. First, for a curve lying on a surface, assume that the curve is an edge  $E$  of solid  $A$ , and the surface is a face  $F$  of solid  $B$ . Then, for the two faces from  $A$  that border  $E$ , the intersection curves of  $F$  with those faces overlaps the trimming curves corresponding to  $E$ . This can be detected in the third step of the boundary evaluation algorithm (intersecting with trimming curves). The intersection curve has a non-constant common factor with one of the trimming curves, indicating an overlap. The case of overlapping curves is detected in exactly the same way, since for the curves to overlap, each curve must first be lying on the surface of the other solid.
- *Curve Tangent to Surface:* As in the previous case, consider the faces of solid  $A$  that border edge  $E$ . For each of these faces, the intersection curve has a tangential intersection with the trimming curve corresponding to  $E$ . This is described in Figure 7.4. For solid  $B$ , the face  $F$  has either a cusp, self-intersection, or point solution, similar to the case of surfaces tangent along a curve. Detecting the situation in either solid determines that there is a degeneracy.
- *Four (or more) surfaces meeting at a point:* This includes the cases of a point lying on a surface, intersecting curves, a point lying on a curve, and coincident points. The point lying on a surface case is illustrated, but the other cases are



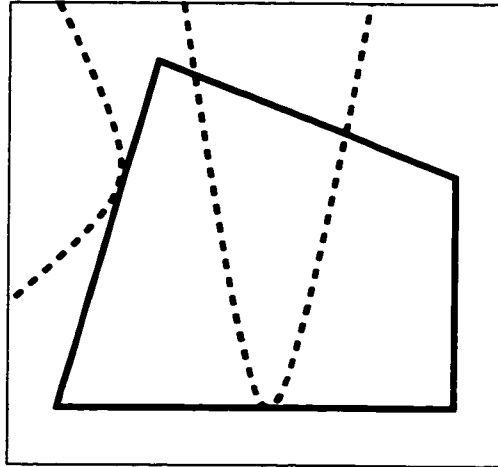


Figure 7.4: **Curve tangent to surface degeneracy as seen in the patch domain.** Different intersection curves are shown by the dashed lines, the trimming curves by the solid lines. Tangential intersections indicate a curve tangent to a surface.

similar. Assume the point is a vertex,  $V$ , of solid  $A$ , and the surface is a face,  $F$ , of solid  $B$ . For the faces of  $A$  that surround that vertex, the intersection curve with  $F$  passes right through the endpoint of one of the trimming curves corresponding to  $V$ . An example is shown in Figure 7.5(a). Notice that within the patch domain, this is equivalent to three planar curves meeting at a point. All such cases of four (or more) surfaces meeting at a point appear as three (or more) planar curves meeting at a common point within a patch domain. For all cases except intersecting curves, this point appears at a trimming curve endpoint in at least one of the domains. For the intersecting curves case, the point lies on a trimming curve, but not at an endpoint. The other two plane curves meeting at the point are two separate intersection curves. This is illustrated in Figure 7.5(b). All of these cases can be detected during the third step of the boundary evaluation algorithm (intersecting with trimming curves). Intersecting curves are found during point inversion, while the other cases are found when performing the intersection in the first patch domain.

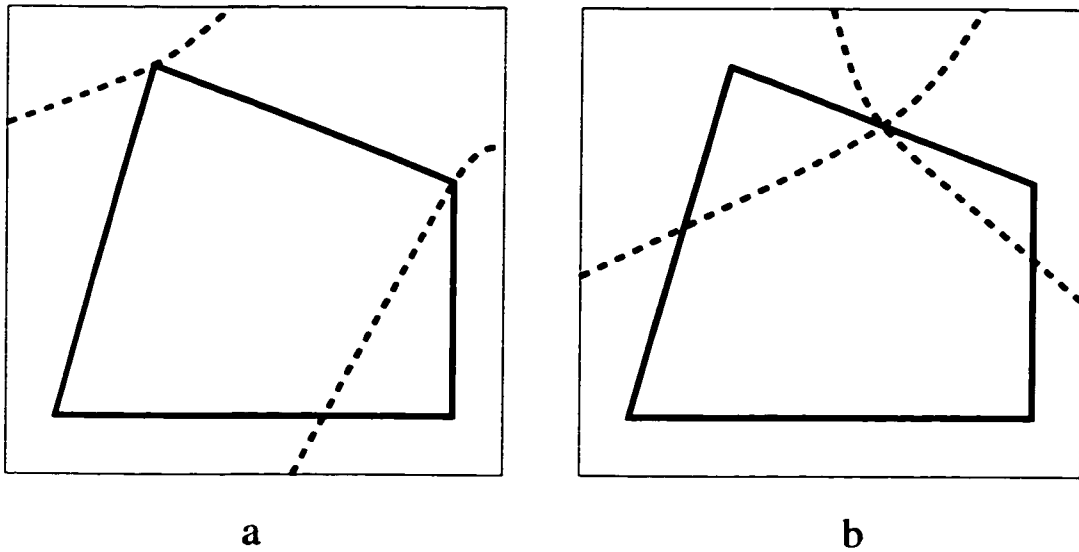


Figure 7.5: **More input degeneracies as seen in the patch domain.** Different intersection curves are shown by the dashed lines, the trimming curves by the solid lines. In (a), when an intersection curve passes through a trimming curve endpoint, a point lying on surface, point lying on curve, or coincident vertex input degeneracy may be occurring. In (b), when three plane curves meet at a point (one trimming curve, two intersection curves), an intersecting curves degeneracy may be occurring.

## 7.3 Eliminating Degeneracies

For this section, assume that the input solid is stored in a CSG format. To have a fully robust system, it is necessary to handle all degeneracies. This means that the system will not crash on any degeneracy, and will produce valid output for all cases. In an ideal system, this output is also a completely accurate result based on the input. Because it is often difficult to enumerate and represent all the degeneracies, in practical implementations, usually either degeneracies are not handled or the output data is slightly different from the completely accurate answer.

One example of the difficulty of representing degeneracies is representing non-manifold data for boundary evaluation. Even when input data is manifold, the output generated may be non-manifold. Unions of the solids in Examples 7.2(b), 7.2(d), 7.2(e), and 7.2(f) are all cases of this. For a boundary evaluation system to accurately handle these cases, it must represent non-manifold data, as well as handle the non-generic intermediate cases that such degenerate situations impose. A system could also choose to handle the degeneracies by approximating the output by a manifold, or some other representation that is close but inaccurate.

Section 2.3.2.2 describes some of the previous approaches to dealing with degenerate data. In this section, three general ways that degeneracies can be handled are described, in the context of the boundary evaluation algorithm. In some cases, it is necessary to distinguish unpredictable degeneracies from input degeneracies, although in general, the methods described handle both types of degeneracies. Also realize that most methods for handling degeneracies require some modification to the boundary evaluation algorithm as described. Some of these modifications are complicated, and they are not described in detail here.

### 7.3.1 Special Cases

The most direct method for dealing with degeneracies is on a case-by-case basis. Unfortunately, it can be difficult to account for all degeneracies, detect all degeneracies, and implement methods for handling each type of degeneracy. Even if all cases can be enumerated, the implementation to detect and handle the cases tends to consist of large amounts of special-case code, obscuring the overall program flow and making software maintenance and debugging difficult. Still, because a case-by-case approach is direct, can be implemented incrementally (handling one degeneracy at a time), and can be added on to an existing non-robust system, it is a reasonable and popular

approach to dealing with degeneracies.

Note that the algorithm described already handles certain special cases. For example, the curve-curve intersection routine described in Section 4.1 ignores point curves. So, an algebraic intersection curve that is a single point in the domain is ignored completely. Because the intersection curve is also ignored in the other patch domain, the corresponding degeneracy is effectively handled.

Minor modifications to the algorithm described allow it to handle other special cases. One example is the degeneracy of surfaces tangent along a curve, where the tangency is along the entire curve, not just at a few points. In this case, all steps of the boundary evaluation algorithm proceed normally, except for classifying partitions (Section 5.9). Normally, at this step, one partition from each solid is classified, and the classification information is propagated to all the other partitions of that solid. The assumption is that partitions adjacent along intersection curves are on opposite sides of the boundary of the other solid. That is, if partitions  $A$  and  $B$  are adjacent partitions of solid  $S_1$ , and  $A$  and  $B$  are separated by an intersection curve, then if  $A$  is inside  $S_2$  then  $B$  must be outside, and vice-versa. When surfaces are tangent along a curve this might not be true. For example, both  $A$  and  $B$  could be outside of  $S_2$ . The boundary evaluation algorithm can be modified so that each component is classified individually, with no information passed between adjacent components. This allows every component to be correctly classified as inside or outside of the other solid, at the cost of more computation.

Similar methods can be used to handle other types of degeneracies. Handling some special cases requires significant changes to the algorithm as a whole. For example, two types of input degeneracies can yield curves that have cusps or self intersections. In order to correctly deal with such curves, kernel operations for curve-curve intersection and curve topology resolution must be modified, the representation of curves must be changed, and the overall boundary evaluation algorithm must be modified. Such complexity led previous researchers to the proposal of perturbations to deal with several degeneracies at once (Section 2.3.2.2). Perturbation methods are discussed further in Sections 7.3.2 and 7.3.3.

Although it can be difficult to implement special cases to handle all degeneracies, particularly input ones, special cases may be useful for dealing with unpredictable degeneracies. General position of the input is a reasonable assumption, although it is seldom guaranteed in practice and it will not necessarily remove unpredictable degeneracies. Fortunately, the unpredictable degeneracies that arise in the boundary

evaluation algorithm tend to be easy to handle by special cases. The special cases for the unpredictable degeneracies mentioned in Section 7.1.2, are:

- *Patch representations:* Since these basically appear the same way as input degeneracies, they must be treated the same way. Perturbation methods are probably more appropriate than special cases for dealing with these.
- *Parameterizations of Patches:* The particular parameterization used for a patch can lead to several unpredictable degeneracies. Two different points may have the same coordinate value in one dimension, which must be taken into account in a variety of point comparison operations. Intersection curves (and trimming curves) may be horizontal or vertical in the patch domain, making computations with them much simpler, but causing them to be treated as a special case. A ray direction might be chosen that overlaps a trimming curve or hits a trimming curve or patch at a trimming curve endpoint. This can either be handled by detecting the case and choosing a different ray direction/test point (which should always be possible), or by simply augmenting the ray test procedure to take into account the other possibilities. Points may be equal due to intentional degeneracies, or may be equal due to unpredictable degeneracies. For example, two curves might intersect at a point that is at a local maximum in  $s$  of one of the curves. That intersection point, then, will also be a segment endpoint, since that local maximum will be one of the turning points found when resolving curve topology. Such unpredictable cases are usually a result of the choice of patch parameterization, and they can be handled by performing an explicit test for point equality.
- *Point Intervals:* Because the intervals surrounding points are assumed to be open, points sharing interval endpoints can still be compared directly. The only important consideration is to take equality into account when performing comparisons (i.e.  $\leq$  vs.  $<$ ).
- *Points Lying on Cut Value:* Every time a point is to be cut, the cut value is first tested to determine if it is the exact solution to the corresponding polynomial. If so, the hybrid representation of the point allows it to be expressed as a rational number instead of an interval.

Another group of degeneracies that can be handled by special cases are the *borderline* degeneracies (Section 7.1.1.1). The boundary evaluation algorithm described

can encounter these cases. For example, the first step of boundary evaluation is to form the entire intersection curve, which is the result of the entire surface of the other solid intersected over the domain of the patch. This intersection may be degenerate, even if the original solids were seemingly in a valid position. Borderline cases are treated the same way as input degeneracies.

### 7.3.2 Symbolic Perturbation

Perturbation methods are used to eliminate a large group of degeneracies at one time. The basic idea is that since a small perturbation of the input data will (by definition) remove any degeneracy, the computation is performed on a perturbed problem rather than on the actual input problem. Symbolic perturbation modifies the data by a variable amount, and the final answer is the result of the limit as that variable goes to zero. Thus, degenerate conditions are not explicitly dealt with.

Previous approaches and a more general discussion of perturbation methods are described in Section 2.3.2.2. In all previous perturbation approaches, the flow of the geometric algorithm is determined by predicates that can be directly expressed in terms of the input data. Each piece of input data is perturbed by some symbolic amount, and thus the predicate is replaced by a function involving these symbolic amounts. Only the sign of the predicate is important, and thus the sign of the new symbolic predicates can be determined by simple computations on the input variables.

A simple example, used by Fortune [35] to deal with degeneracies in an exact polyhedral modeling system, is the plane orientation test. This test determines on which side of a plane a point lies. The point is defined as the intersection of three planes. If the coefficients of the plane equations are of the form  $(a_i, b_i, c_i, d_i)$ , then the plane orientation is determined by the sign of the determinant:

$$\begin{vmatrix} a_i & b_i & c_i & d_i \\ a_j & b_j & c_j & d_j \\ a_k & b_k & c_k & d_k \\ a_l & b_l & c_l & d_l \end{vmatrix} \quad (7.1)$$

When the point lies on the plane (a degeneracy where four planes meet at a point), this determinant evaluates to zero. A perturbation can be induced that shifts all of the planes slightly. The coefficients of the plane equation become  $(a_i, b_i, c_i, d_i + \phi_i \epsilon^i)$ , where  $\phi_i$  is either 1 or  $-1$ , and  $\epsilon > 0$  is the amount of the perturbation. This

perturbation scheme moves the planes either in or out along the normal direction. The plane orientation predicate then becomes:

$$\begin{vmatrix} a_i & b_i & c_i & d_i + \phi_i \epsilon^i \\ a_j & b_j & c_j & d_j + \phi_j \epsilon^j \\ a_k & b_k & c_k & d_k + \phi_k \epsilon^k \\ a_l & b_l & c_l & d_l + \phi_l \epsilon^l \end{vmatrix} = \quad (7.2)$$

$$\begin{vmatrix} a_i & b_i & c_i & d_i \\ a_j & b_j & c_j & d_j \\ a_k & b_k & c_k & d_k \\ a_l & b_l & c_l & d_l \end{vmatrix} + \phi_i \epsilon^i \begin{vmatrix} a_j & b_j & c_j \\ a_k & b_k & c_k \\ a_l & b_l & c_l \end{vmatrix} - \phi_j \epsilon^j \begin{vmatrix} a_i & b_i & c_i \\ a_k & b_k & c_k \\ a_l & b_l & c_l \end{vmatrix} + \phi_k \epsilon^k \begin{vmatrix} a_i & b_i & c_i \\ a_j & b_j & c_j \\ a_l & b_l & c_l \end{vmatrix} - \phi_l \epsilon^l \begin{vmatrix} a_i & b_i & c_i \\ a_j & b_j & c_j \\ a_k & b_k & c_k \end{vmatrix} \quad (7.3)$$

So, whenever predicate 7.1 was to be used originally, it is replaced by the perturbed function 7.2. Since one is only interested in the sign of the predicate, only the sign of 7.3 is of interest, as  $\epsilon$  goes to zero. This means that the sign of 7.2 can be found by evaluating the terms of 7.3 in order of increasing power of  $\epsilon$ . Notice that the first term in 7.3 is identical to 7.1. Thus, if no perturbation is needed, no extra computation is performed. Only when a degeneracy is present are the later terms of 7.3 used to determine a sign of the predicate. Using a perturbation such as this, degeneracies of the form of four planes meeting at a point are dealt with.

Ideally, a similar symbolic perturbation scheme could be followed for the curved surfaces. One such example might be to formulate everything in terms of the surface equations, and then perturb the equations slightly to eliminate degeneracies. A number of problems occur with such an approach, however. While planes can be easily perturbed by translation along a normal direction, such a perturbation is not as well defined for curved surfaces. Adjusting the constant term of an implicit surface equation, however, might achieve a similar result. In any event, solids would need to be defined only in terms of the surface equations, with all edges and vertices defined only implicitly in terms of those surface equations.

A more fundamental difficulty is in defining appropriate predicates. This difficulty applies to any symbolic perturbation scheme for boundary evaluation for curved solids. While the plane orientation predicate is sufficient for an entire polyhedral boundary evaluation algorithm, no similar single predicate is known for boundary evaluation for curved solids. Simply knowing which side of an implicit surface a point lies on is not enough information to perform boundary evaluation. More significantly, three surfaces may meet in 0, 1, or many points, as opposed to three planes, which

(generically) always meet at one point. Thus, any predicate would not be nearly as simple as the determinant of the plane orientation test. Even if a single predicate were found, it is possible that the symbolic expansion of the perturbed predicate would not have a (relatively) simple formulation, like the plane orientation test. This is not to say that no appropriate predicate, or combination of predicates, for curved surfaces is possible, only that no such scheme is apparent.

If such a scheme were to be developed, it would require a significantly different approach to boundary evaluation than the one outlined in this dissertation. The representations and computations used here are not geared toward simple predicates. Predicates for symbolic perturbation are generally defined so that the perturbations on the input can be directly mapped to perturbations in the predicate evaluation. The approach described in this dissertation performs most calculations based on derived data, rather than directly on the input data. For example, the first step of the boundary evaluation algorithm is to form the intersection curve between two surfaces in the patch domain. Thereafter, the intersection curve is used instead of the two surface equations. Points are generated by intersecting with the intersection curve, rather than a three-dimensional intersection of surfaces. This provides significant performance gains (and gives a representation in line with traditional B-reps), but makes the representation and computations incompatible with likely symbolic perturbation methods.

Because it seems unlikely that a practical symbolic perturbation scheme for curved surfaces can be easily obtained, instead a different type of perturbation is proposed. This is numerical perturbation, and it is described in the following section.

### 7.3.3 Numerical Perturbation

While symbolic perturbations modify the input data by symbolic amounts that are used as their limit approaches zero, numerical perturbations induce real changes to the geometric values of the input data. With numerical perturbations, the actual data is modified, resulting in a solution that is truly different than that of the given input. With symbolic perturbations, the usual effect is to create output with correct geometric information, but with topological information slightly modified in order to represent the degeneracy.

The underlying justification for numerical perturbations is the idea of a *global tolerance*. That is, there is an assumption inherent in the input that the input data is correct to within some amount,  $\epsilon$ . This global tolerance (which could also be expressed



as local tolerances associated with each input object) often is used to take into account the inexact nature of real world manufacturing capabilities. Any perturbation of the input data within the amount of the global tolerance is allowed. Thus, the input data can be perturbed however is necessary to allow the program to run, as long as the perturbation is less than  $\epsilon$ .

There are potential problems with numerical perturbations. First, there is the chance that a specific numerical perturbation will not eliminate the degeneracy (or will create another). For example, a vertex of one polyhedron could be perturbed so that it would lie on the face of another polyhedron (creating a degeneracy), or two overlapping faces could be perturbed in the same way by the same amount (thus not removing the degeneracy). If the perturbations are chosen randomly (or even pseudorandomly), it is extremely unlikely that such an event would happen. A second problem is that the perturbed data will have drastically increased bit length. For example, assume a point has one coordinate equal to 0.34852, and there is a global tolerance of  $\epsilon = 10^{-8}$ . Then, a perturbed value for the coordinate might be 0.3485200003. The number of bits required to represent this perturbed value is significantly higher than that needed for the unperturbed value. Since exact arithmetic becomes slower the more bits are needed, this can cause inefficiency. While symbolic perturbations may be structured so that no additional computation is performed when there is no degeneracy, with numerical perturbations, all computations can take more time.

Like symbolic perturbations, numerical perturbations rely on exact computation. If inexact computation is used, degeneracies can still be created or destroyed due to roundoff error and error compounding, and the perturbation scheme will not guarantee the desired effect.

Tessellation of the curved solid can be viewed as a type of numerical perturbation. If the curved solid is tessellated into linear faces that are never farther than  $\epsilon$  from the original solid, then the tessellated solid is just a numerical perturbation of the original. As mentioned in Section 2.1.2, there are drawbacks to such tessellations. Also, such automated tessellations are prone to introducing and maintaining original degeneracies. For example, if two solids have overlapping edges, their tessellations can easily have overlapping edges as well. The linear nature of the tessellation may make computation easier, but does not provide the general handling of degeneracies that is desired.

In the following sections, two methods for numerical perturbations are proposed

for handling degeneracies.

### 7.3.3.1 Random Translation

One numerical perturbation scheme is to perturb the input by applying random translations. Random rotation and scaling can also be applied, so long as each point on the perturbed input solid is no more than  $\epsilon$  away from a point on the original input solid (and vice versa). Since a random translation accomplishes the effect, and is easier to understand, the discussion focuses on it.

Consider the examples of the degeneracies in Figures 7.2 and 7.3. Note that for any of these degeneracies, any translation of one of the solids results in a nondegenerate case. So, translating each solid by a random amount should remove any degenerate condition between them.

It is possible to perturb all the solids in a consistent manner by applying a random translation to the primitives. Perturbing the primitives perturbs each face, edge, and vertex of the primitive itself, as well as all edges and vertices of objects constructed from that primitive. Since each face of the final solid comes from a face of one of the primitives, and each primitive is perturbed only one time, all uses of a primitive solid have a consistent face structure. A potential problem can arise, however, with overlapping surfaces, when the same primitive is used in various parts of the tree. Figure 7.6 shows a 2D example of this. Although a case as obviously problematic as the example shown would probably not be part of a design, similar but less obvious problems occur with only slightly more complex examples.

The problem demonstrated in Figure 7.6 can occur even when the primitives are perturbed. Since the position of B is perturbed only once, it will be in the same position in the first operation as it is in the later one, thus causing overlapping surfaces. The potential for such cases can be detected by examining the CSG definition of an object. If the same primitive contributes to many branches of the CSG tree, this case can happen. At times, such an effect is an intended consequence. In the figure, for example, (A-B) might be one part of a machine, and B might be a different part, with no Boolean operation to be performed between the two pieces. The designer intentionally subtracted B from A to ensure that the two separate parts fit together closely in the final assembly. Any perturbation that causes (A-B) to no longer meet B exactly would be changing the intent of the design.

Another option is to translate the two input objects just before a boundary evaluation is performed. This ensures that each step of the computation is performed

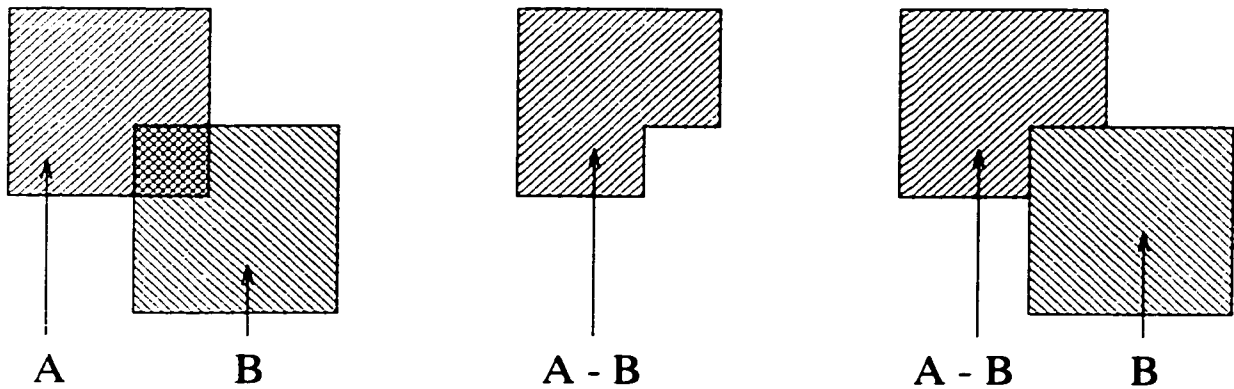


Figure 7.6: **Operation leading to overlapping surfaces.** At left, two solids, A and B, are given. At center is the result of a difference operation (A-B). As seen at right, any time the subtracted object (B) is part of an operation involving (A-B), there will be a problem of overlapping surfaces.

correctly, but allows results that are completely different than what is specified by the primitives. Figure 7.7 shows an example for the objects from Figure 7.6. After forming (A-B), a union with B is performed. A random translation is made just before the operation, guaranteeing success. The resulting object, however, is not what would be expected from the definition. If random translations are to be used, special case detection of situations similar to  $(A-B) \cup B$  should be implemented.

A more general drawback to the use of random translations to remove degeneracies is that the intent of the design is easily lost. Even though the operation might complete, the resulting object can be different from what the designer intended. For example, a designer might assume that operations are regularized, and thus assume that overlapping surfaces are dealt with in that manner. The random translation makes regularization of the operation meaningless, and the designer's intent is lost. Figure 7.8 shows one example of this. Note that such a problem can also be viewed as failure on the designer's part to adhere to the notion of a global tolerance. If the designer had intended the object to be as in the bottom path of Figure 7.8, the solid should have been extruded beyond the tolerance level to guarantee the result. A similar example is shown in Figure 7.9. In this case, no translational perturbation of the primitive will achieve the designer's likely intent of cutting a hole all the way through the object.

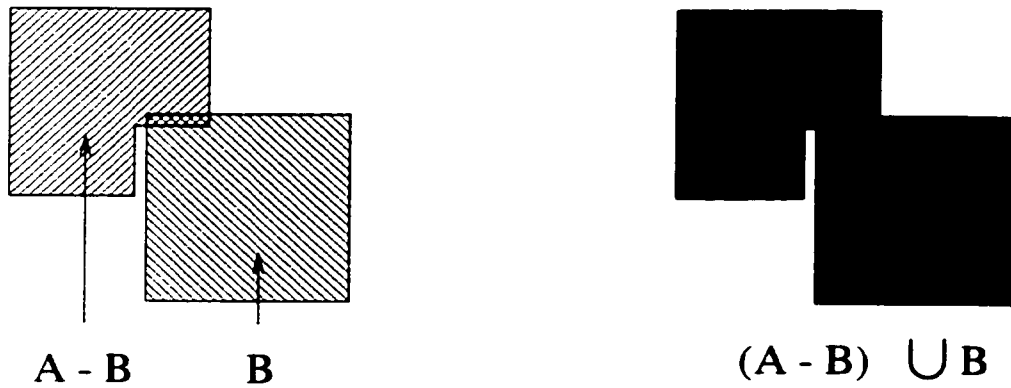


Figure 7.7: **Random translation before operation yielding incorrect solid.** At left, the two solids as in Figure 7.6.  $(A-B)$  and  $B$ , are perturbed slightly before a Boolean operation is performed. A union operation is performed, yielding the solid shown at right. The resulting solid is clearly different than the solid defined by  $(A-B) \cup B$ .

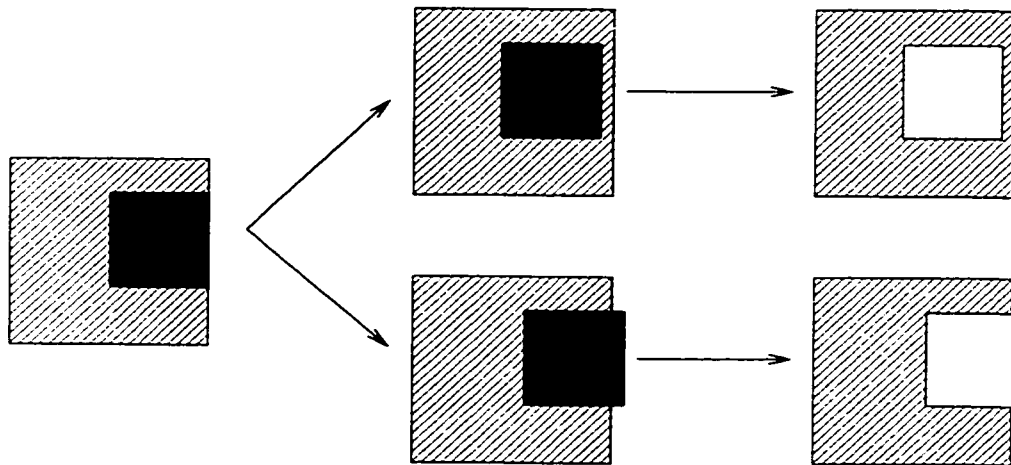


Figure 7.8: **Different translations yielding very different output.** At left, two objects that are input to a difference operation. Depending on the perturbation (shown in the middle), two drastically different output objects are possible (shown at right). The object at bottom is closer to the designer's intent.

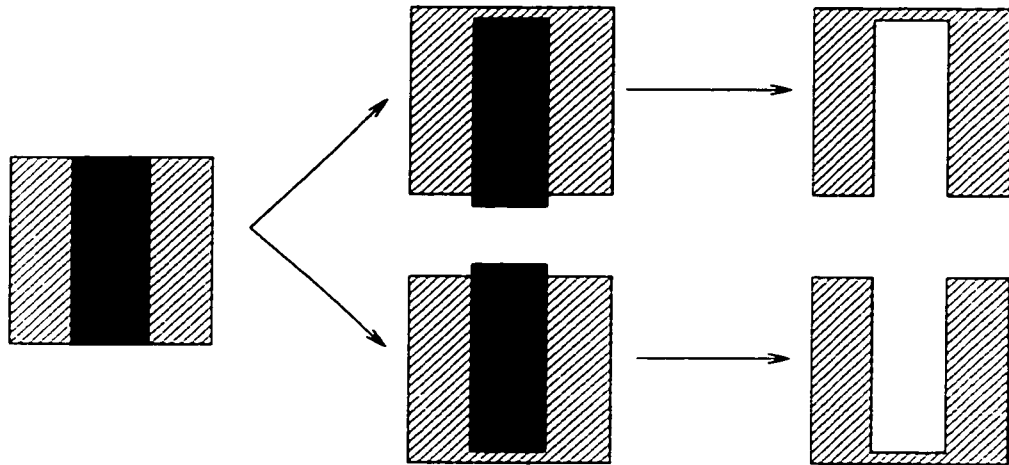


Figure 7.9: **An example of translations unable to capture design intent.** At left, two objects that are input to a difference operation. The two perturbations, shown at middle, produce the output shown at right. Neither output reflects the designer's intent.

### 7.3.3.2 Expanding and Contracting Primitives

A second form of numerical perturbation involves expanding and contracting the solids. As opposed to random translation, the expansion and contraction is a scaling performed in a certain way so as to maintain the designer's intent. For the example in Figure 7.8, for example, the output would be similar to that of the lower path rather than the upper one.

There are significant shortcomings of this method, some of which are described below. The basic principles, however, give a method that eliminates degeneracies, while making the perturbations *more likely* to maintain the designer's intent. This leads to hope that a more complete method can be developed in the future.

This approach follows the general principles outlined by Fortune in [35], where the faces of input polyhedral solids are symbolically perturbed inward or outward in order to remove degeneracies. For a reasonably complex object, such perturbations can change the topological connectivity of the object. Fortune's paper describes how to deal with this problem (which also arises when rounding face planes) in polyhedra by using generalized polyhedra and simplification. For curved surfaces, dealing with the problem is much more complicated. Offsets are not as clearly defined, the topological changes can be much more complicated, and simplification is more difficult.

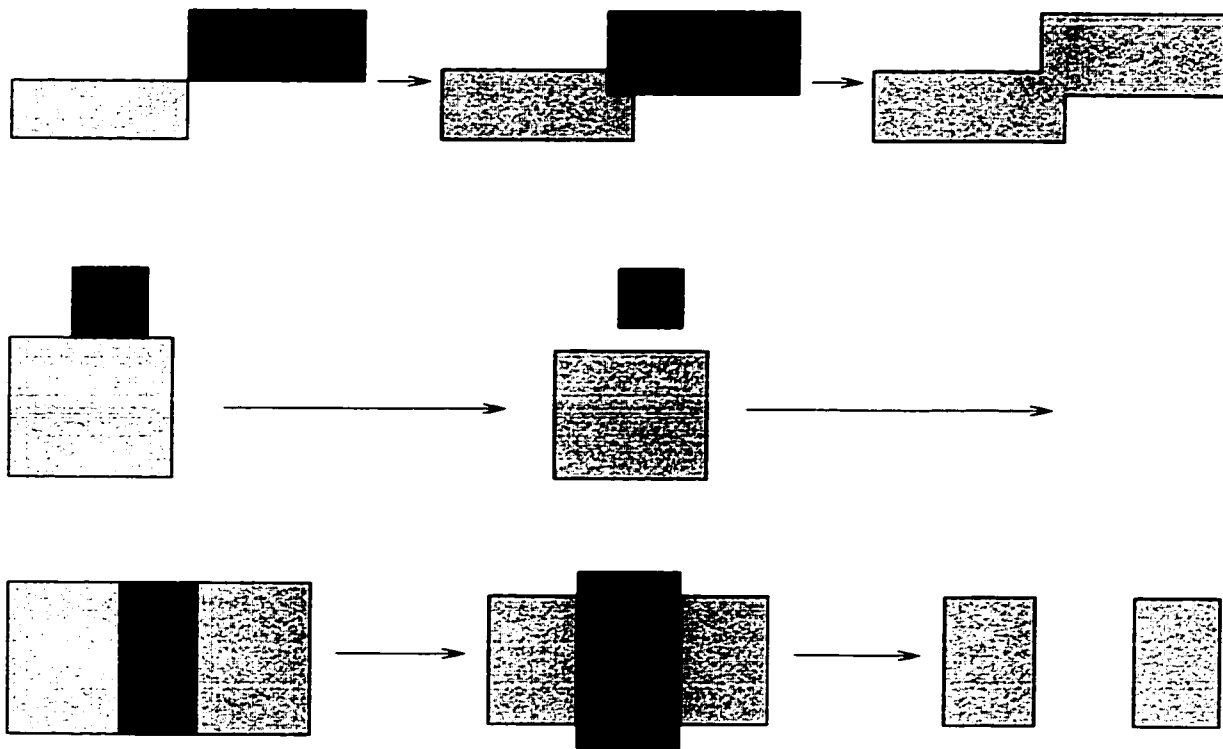


Figure 7.10: **Examples of expansion and contraction of primitives.** The first column shows the input, the second the perturbed solids, and the last the final solids. The top row shows, a union operation, where both solids are expanded. In the middle, an intersection operation requires both solids to be reduced. At bottom, a difference operation contracts one solid and expands the other.

The basic idea of this approach is that for two input solids, expanding and contracting the solids eliminates the degeneracies. Furthermore, the operation being performed dictates whether each solid should be expanded or contracted. For a union, both solids should be expanded, for an intersection, both should be contracted. For a difference operation,  $A - B$ , solid A should be contracted and solid B expanded. Assuming that the degree of expansion or contraction is different for the two solids, no degeneracies will remain. Figure 7.10 shows some 2D examples of how this works, similar to ones shown by Fortune [35].

Note that this is not a perfect method for capturing the designer's intent. Depending on which solid is expanded faster, the output in certain cases can be different than what would be expected. Figure 7.11 shows other 2D examples where the output object does not reflect the original design. If the perturbation were symbolic, the

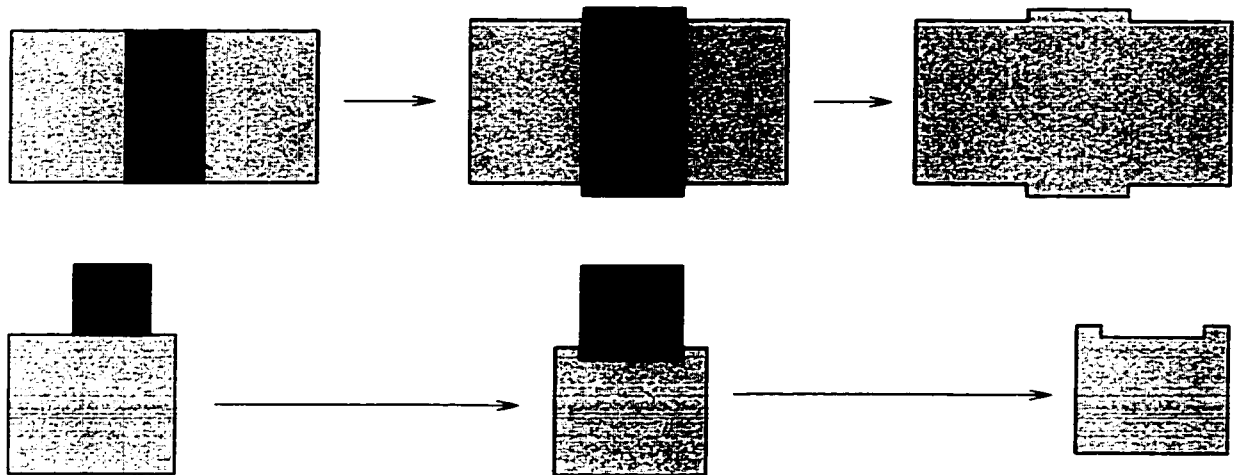


Figure 7.11: **Examples where numerical expansion and contraction lead to undesirable results.** At top, a union operation is performed. The smaller solid is expanded more than the larger one, resulting in extraneous faces. Had the larger solid been expanded more than the smaller one, this would not have happened. At bottom, in a difference operation, the smaller solid is expanded more than the larger one is contracted. This results in a notch that is not in the original specification, and that would not appear had the larger object been reduced more than the smaller one increased.

results would have *symbolic faces*, which are geometrically the same as the original and thus are usually acceptable. For numerical perturbation, however, such extra faces are real, which is often not desired. From personal experience, the more common degenerate cases encountered tend to be of the type shown in Figure 7.10, rather than those of the type in Figure 7.11, and thus the expansion and contraction usually perturb solids in the manner desired.

It is important to realize that offsetting the surfaces by a particular amount may perturb the solid by more than that amount. A simple example is shown in Figure 7.12, where perturbing the lines outward slightly dramatically changes their intersection point. If the lines are perturbed outward by a value  $\epsilon$ , the point may be perturbed by much more than  $\epsilon$ . Similar effects happen when perturbing surfaces.

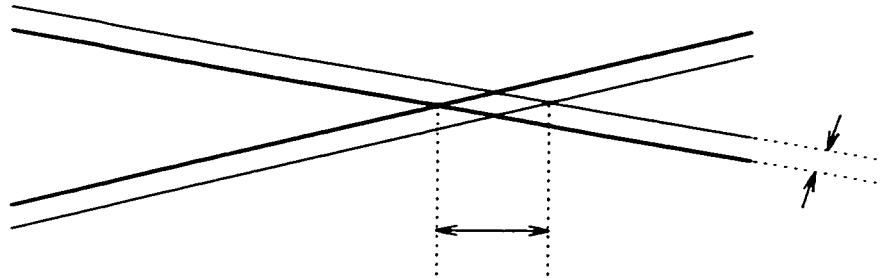


Figure 7.12: **A small perturbation in lines resulting in a much larger perturbation of their intersection point.**

where slight changes in a surface can cause large changes in the edges and points formed by intersections with that surface. Note that even if primitives themselves are not perturbed much, the effect of the perturbation on objects created from those primitives can be significant.

For all but the simplest curved objects, perturbing each of the surfaces of the object as described can be difficult. One approach is to apply a scaling to the entire object, based around the object's centroid (or some other point). For non-convex objects, however, scaling about the centroid may not perturb all faces in the desired direction. Rather than outward or inward relative to the object interior, the surfaces are perturbed outward or inward relative to some point. A 2D example is shown in Figure 7.13. Even though such a scaling is not exactly what is desired, it still eliminates degeneracies, and most of the faces are perturbed in the direction desired. Remember, though, that scaling solids at intermediate stages can lead to the same types of problems described in Section 7.3.3.1.

If, instead of scaling, a true perturbation of surfaces inward or outward is desired, computation can become much more difficult. For curved surfaces, a simple translation of the surface is often not sufficient: instead, an offset surface must be used. Offset surfaces, however, are difficult to compute and operate on. The standard CSG primitives, however, are relatively easy to expand and contract. For example, an ellipsoid can be expanded by just increasing the radius vectors (Appendix A). For this reason, the numerical perturbation approach proposed perturbs the primitives only. By perturbing the surfaces of these primitives, all future computations based on those primitives (i.e. all objects constructed) are also perturbed.

The surfaces of a CSG-defined solid can be perturbed inward and outward by



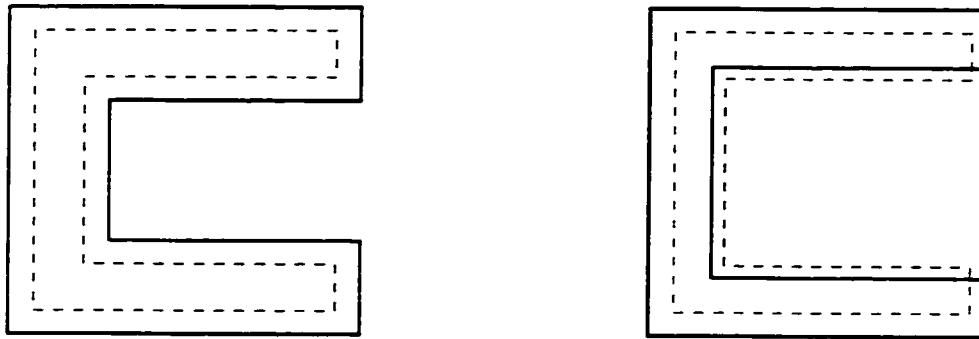


Figure 7.13: **The difference between perturbing outward and scaling.** At left, the original object (shown by the dashed lines) has its faces perturbed outward, resulting in the object shown with the bold lines. At right, the original solid is scaled from a point in the center. Notice that the interior edges of the object are actually perturbed inward, rather than outward.

perturbing the surfaces of the input solids. Assume that the CSG tree has been completely expanded (recall Section 2.1.1 and Figure 2.2). Each primitive is scaled by some amount, and the computation proceeds as usual. The key is to understand how the primitives can be perturbed to eliminate degeneracies, but still provide the general behavior described earlier (which often matches the designer's intent).

Perturbation information can be pushed down the tree. For example, say an object,  $A$  is defined by  $A = B \cup C$ . If we want to push the boundaries of  $A$  outward by one unit, then the boundaries of  $B$  and  $C$  should be pushed outward by one unit.  $B$  and  $C$  would also be pushed outward to compute  $A = B \cap C$ . To compute  $A = B - C$ , however, the boundary of  $B$  would be pushed outward by one unit, and the boundary of  $C$  inward by one unit. For each Boolean operation in the tree, the perturbation information for that operation can be pushed all the way down to the leaf nodes (the primitives). Figure 7.14 shows an example of how the perturbation information can be propagated down the tree.

It is tempting to combine the perturbations for each Boolean operation at the leaves. Each leaf node is then perturbed by the sum of the perturbations from all operations above it. The difficulty with this idea is that the perturbation information from one operation may contradict that from another. For example, if a leaf node had both a union operation and an intersection operation in the tree above it, the perturbation required for one would be in the opposite direction as the perturba-

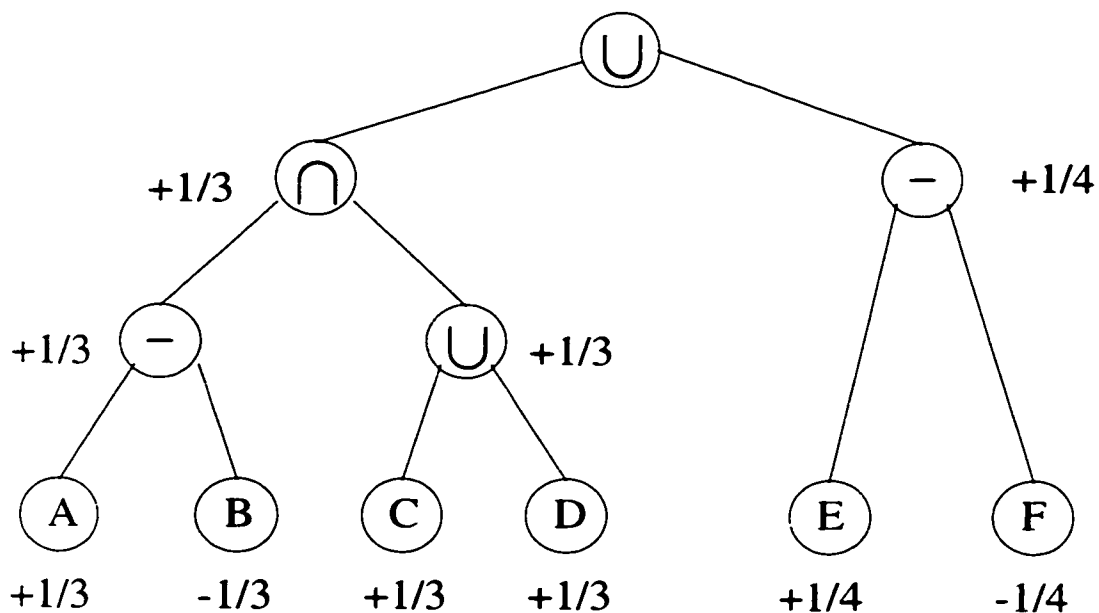
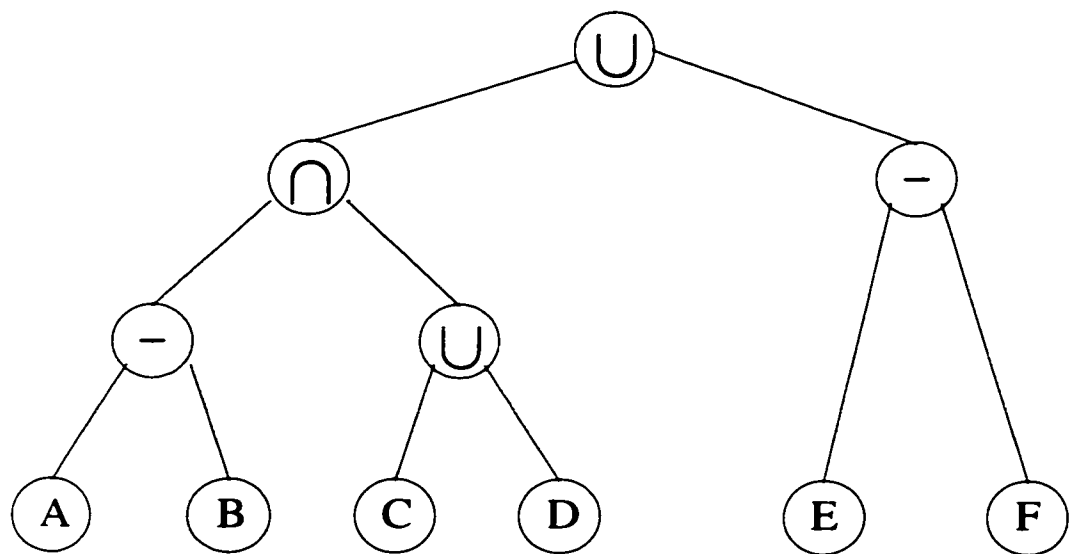


Figure 7.14: **Propagating extrusion information.** At top, the CSG tree defining an object. At bottom, the perturbation information for the top node (a union) is propagated to the leaf nodes. The union requires that both input solids be extruded outward (the left tree by 1/3 unit, the right by 1/4 in this case). In order for those inputs to be extruded that way, the primitive solids defined at the leaf nodes must be extruded by the amounts shown.

tion required by the other. Whichever perturbation is greater will win out, and the primitive will not be perturbed in the appropriate direction for the other solid.

It is easy to imagine schemes that can make use of the aggregate perturbation information passed down from all previous nodes. For example, each previous node could vote on whether the perturbation should be inward or outward. Another idea is to identify which operations are more likely to involve a degeneracy (e.g. by some sort of bounding-box comparison), and use only that perturbation information. Numerous such schemes can be developed, and it is possible that one of them will lead to a system that deals with all degenerate cases reliably. Even if no ideal scheme can be developed, schemes (such as those mentioned) that are *more likely* to give a good perturbation (i.e. one in which the resulting solid reflects the designer's intent) than a bad one can be easily designed.

It is important to remember that the perturbations should not cancel out (otherwise there is effectively no perturbation, and degeneracies can still arise) and the amounts of perturbation should be different for any two solids (or, again, the degeneracies might not be removed). As long as this happens (ignoring the possibility of introducing new degeneracies by the perturbation), the degeneracies are removed, and no computational difficulties due to degeneracies should arise. The only problem is that accuracy is lost.

There are other ways of adapting this scheme slightly. For example, objects could be chosen to be translated toward or away from each other. There are significant problems with this method as well, but the concept of a guided perturbation (rather than a purely random one) to more closely mimic the designer's intent will lead to more effective numerical perturbations.

# Chapter 8

## Implementation and Performance

The representations, algorithms, and speedups mentioned in earlier chapters have been implemented as a part of a boundary evaluation system. This system has been applied to a number of examples, both artificial and from the Bradley Fighting Vehicle model, a CSG model provided courtesy of the Army Research Lab. This chapter discusses some of the details of the implementation, as well as describes the results that have been achieved.

The implementation has been in two major parts. First, a library, MAPC, was developed to handle the underlying two-dimensional operations that form the basis for the entire boundary evaluation algorithm. Although it is specifically geared to boundary evaluation, MAPC also can be applied to other geometric problems. The exact boundary evaluation system, ESOLID, was then built on top of MAPC. ESOLID includes routines to convert data from the BRL-CAD system to the ESOLID format. MAPC and ESOLID are discussed separately in the following sections. In order to understand some of the more general details regarding input data, the material presented in Section 3.5 should be reviewed.

Although I took the lead role in the development of both MAPC and ESOLID, I am not the sole developer. Tim Culver, Shankar Krishnan, and Mark Foskey have each made significant contributions to the code. In total, MAPC and ESOLID consist of over 45,000 lines of code.

### 8.1 The MAPC Library

MAPC, which stands for Manipulation of Algebraic Points and Curves, is a C++ library that exactly represents and operates on algebraic plane curves. MAPC provides

classes for polynomials, 2D bounding boxes, and the points and curves as described in Chapter 3.

Other libraries and systems provide implementations that are similar to portions of MAPC. However, currently no other library or system is known that provides both the exact computation and the geometric data structures provided by MAPC. For example, computer algebra systems can give exact solutions to systems of polynomials, however these do not provide geometric representations for points and curves. Systems that represent points and curves don't use exact representations to do so.

Although MAPC was developed specifically to aid in boundary evaluation, the routines have more general application. MAPC is being used in a program to find the exact medial axis of a polyhedron [20]. An early version of MAPC has been made available for download via the web. Although no record has been kept of downloads or users, from various email responses, it has been downloaded by other people, and at least one person has used the routines in her research work [100].

Section 8.1.1 describes the details of the classes implemented in MAPC. Section 8.1.2 presents timings for several example problems.

### 8.1.1 Implementation Details

The MAPC library consists of over 32,000 lines of C++ code. It is implemented on top of the LiDIA library [11], which provides exact rational number representations and arithmetic. Another rational number library could have just as easily been used. LiDIA was chosen because it provided the functionality being sought, and was faster on some basic tests than similar libraries, such as LEDA [71]. In addition, the LAPACK library [3] is used to find polynomial roots (for increased efficiency, Section 6.4) by the eigenvalue-based methods described by Manocha [66]. Another root-finding method could have easily been used instead. Besides the methods outlined in Section 4.1 for curve-curve intersection, MAPC provides methods based on multivariate Sturm sequences (Section 2.2.3.4). Those routines make more extensive use of LAPACK for floating-point speedups.

A chart showing the primary MAPC classes is shown in Figure 8.1. The functions of each of these classes are described in the following subsections.

Besides the speedups mentioned in Chapter 6, two other speedups have been implemented and incorporated into the MAPC routines. These speedups can be optionally included, and have not been part of the publicly released MAPC code. One speedup, which will be referred to here as `fp-roots`, is a univariate root isolation

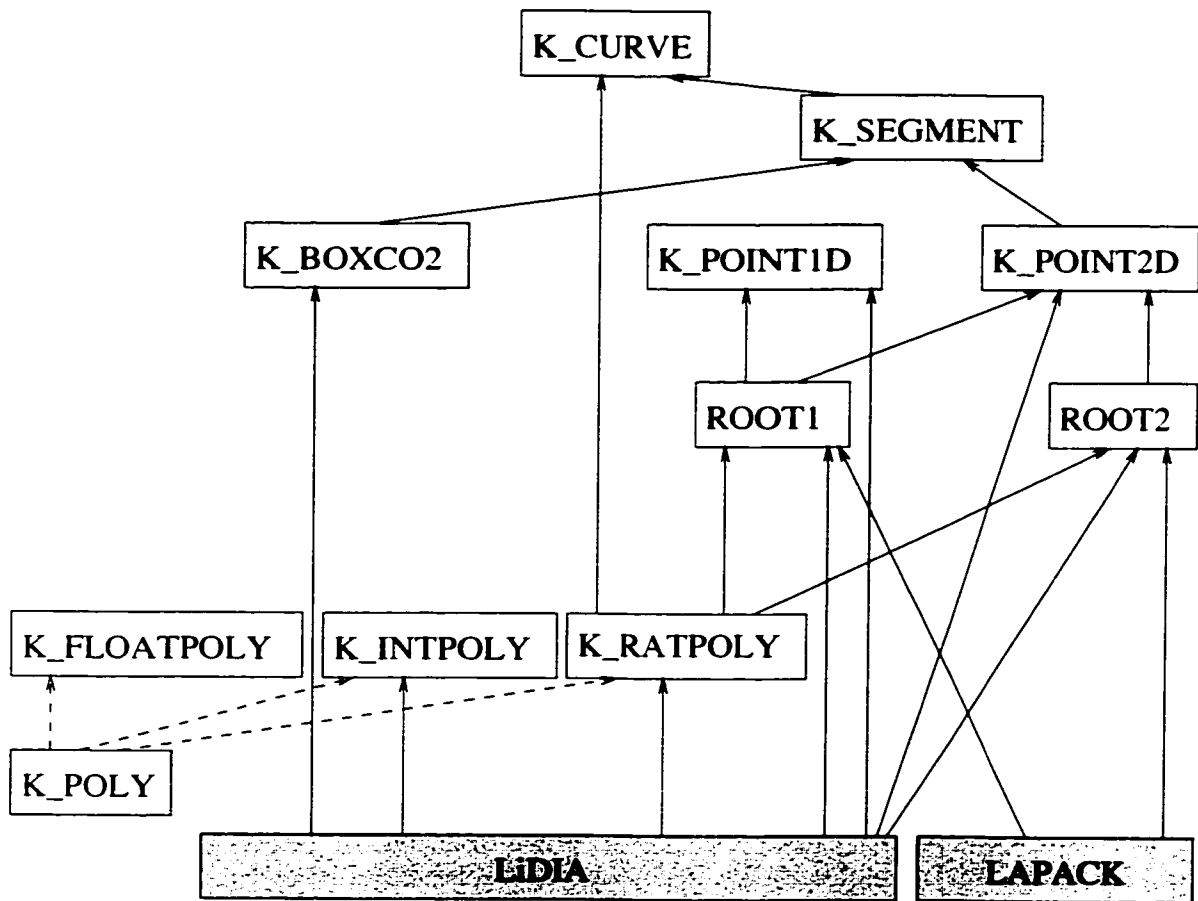


Figure 8.1: **The organization of MAPC.** Primary classes in MAPC, shown by clear boxes. External libraries are shown in shaded boxes. Dashed lines show inheritance. Solid lines indicate a class or library is used as a member variable of another class.

method that performs all computation in standard (inexact) floating-point, but with guaranteed error bounds. Thus, MAPC is able to skip the time-consuming exact Sturm sequence generation under certain circumstances. The other speedup is incorporation of an arbitrary precision floating-point library, PRECISE [63]. PRECISE allows exact computation, but uses arbitrary precision floating-point numbers rather than arbitrary precision rational numbers. It can also perform inexact calculations with more predictable (and controllable) error growth. Although PRECISE could be applied to many parts of MAPC (possibly even replacing all LiDIA bigrational computations), it has been incorporated, thus far, only in speeding up univariate Sturm sequences.

### 8.1.1.1 Polynomials

Polynomials in MAPC are defined by the `K_POLY` class. The base class allows representations of polynomials with any number of variables and of arbitrary degree. Three classes, distinguished by the format of their coefficients, inherit from the `K_POLY` class. These are the `K_FLOATPOLY`, `K_INTPOLY`, and `K_RATPOLY` classes, which have coefficients that are IEEE standard double precision floating-point numbers, LiDIA bigintegers (arbitrary precision integers), and LiDIA bigrationals (arbitrary precision rational numbers), respectively. Conversions between the types are provided. `K_RATPOLY`s are the basic polynomial in almost all MAPC routines. `K_FLOATPOLY`s are used only for implementing floating-point speedups (Section 6.4), and `K_INTPOLY`s are used in the optional multivariate Sturm code.

In the implementation of the polynomial classes, a dense representation for the polynomials is used instead of the sparse representation used in most computer algebra systems. This means that space is allocated for all coefficients up to a certain power, rather than just allocating space for nonzero coefficients. For example, a dense representation of the polynomial  $x^{100} + 1$  would require storing 101 coefficients, while a sparse representation would require only 2 coefficients. In most situations, including general computer algebra systems, the advantage of a sparse representation is undeniable. The polynomials encountered in real-world boundary evaluation examples, however, are almost always dense. The dense representation allows the coefficients to be stored in an array structure that can be slightly more efficient to work with and much easier to write code for.

The MAPC polynomial classes include methods for many of the basic polynomial operations. These include functions for accessing variables, addition, subtraction,

multiplication, division by a known factor or (for univariate polynomials) with a remainder, differentiation, evaluation at a point or over an interval, variable substitution, Bernstein basis conversion, and sign evaluation at a point. More general polynomial functions, such as polynomial factorization, are not included in MAPC. Such functions are useful in general computer algebra systems but are not used in boundary evaluation. The polynomial classes also store the associated Sturm sequence, so that it does not have to be recomputed each time it is needed.

#### 8.1.1.2 Points

MAPC can represent points in one and two dimensions. The underlying classes for these points are the ROOT1 and ROOT2 classes. These classes represent an interval that may contain an *arbitrary number* of roots of a univariate polynomial (in one dimension) or of a pair of bivariate polynomials (in two dimensions). The classes contain all of the information necessary to count the number of roots within the interval, using univariate or multivariate Sturm sequences. In addition, basic routines for isolating roots and reducing interval size are provided. Note that when a 2D point is found using the algorithm described in Section 4.1, the resulting ROOT2 is guaranteed to contain a single root of the polynomials, and is just as easily represented by two ROOT1s (one for each coordinate).

The K\_POINT1D and K\_POINT2D classes represent *individual* points. Each class uses a hybrid representation, allowing a coordinate to be expressed as a rational number when possible. A K\_POINT1D, then, is either a single rational number, or a ROOT1 that contains exactly one root (not on the interval boundary). A K\_POINT2D, like the point description in Section 3.3.1, can have both coordinates be rational numbers, have one coordinate a rational number and the other a ROOT1 that contains exactly one root, not on the boundary, or have both coordinates expressed by a ROOT2 that contains exactly one root.

The ROOT1 and ROOT2 classes are provided in MAPC so that library users can access them for Sturm sequence computations. In boundary evaluation, only the K\_POINT1D and K\_POINT2D classes (which indirectly make use of the ROOT1 and ROOT2 classes) are used, and the multivariate Sturm computations are never used, since a more efficient method for curve-curve intersection in 2D is available.

Several operations are provided for points. Besides root isolation, routines are provided to perform overlap tests, comparison and equality tests, sorting, and interval reduction (Section 3.3.2.1). K\_POINT2Ds store pointers to equivalent points



in other domains, or equal points in the same domain. `K_POINT1Ds` also can be treated as real numbers (rather than just as points), allowing for basic operations (addition, subtraction, multiplication, division, and square root) to be performed on them. Although such operations are provided in MAPC, other representations are more efficient and appropriate for performing these operations [14].

### 8.1.1.3 Curves

MAPC provides a class to represent curves, as described in Section 3.2. A `K_CURVE` is represented by a polynomial and an ordered list of `K_SEGMENTs`. A `K_SEGMENT` consists of a starting point and an ending point, each of which is a `K_POINT2D`.

Closely associated with both curves and segments is the concept of the bounding box. This is represented by the `K_BOXCO2` class, which provides a 2D bounding box that can be either open or closed along the boundaries. Each segment is associated with a bounding box that can be formed by finding a bounding box that contains both segment endpoints (since the segment is assumed to be monotonic). Segment bounding boxes can be merged to find a bounding box for the entire curve. Although only used for bounding boxes in practice, the `K_BOXCO2` can be used for general 2D interval representations.

Several operations are provided for these classes. Operations to merge, intersect, or compare `K_BOXCO2s` are provided. `K_SEGMENTs` can be split in two and subdivided so that their bounding boxes don't overlap. Curves can be intersected, split, refined (by adding a point to them), rotated, and checked for point containment. In addition, routines to sort `K_POINT2Ds` along a `K_CURVE` are included. A `K_CURVE` can also store a pointer to an equivalent curve in another domain, along with the correspondence with that curve.

## 8.1.2 Performance Results

MAPC provides the underlying support for the boundary evaluation system and has applications to other geometric problems. It is important to understand the performance characteristics of MAPC, since they have a major effect on the efficiency of ESOLID.

In this section, some basic timing results are given for the basic computations in MAPC, as well as for a couple of simple applications of MAPC. In all of these cases, it is difficult to determine a comprehensive test. Worst-case bounds often give

<i>Case</i>	1	2	3	4	5
<i>Degree of Curves</i>	2, 2	2, 4	3, 3	4, 3	4, 4
<i>Bits in Coefficients</i>	3, 6	9, 24	23, 20	18, 23	24, 18
<i>Number of Roots</i>	4	2	2	2	1
<i>Time using 2D Sturm</i>	0.51	8.21	20.26	123.29	333.48
<i>Time Using New Algorithm</i>	0.07	0.28	1.01	8.97	36.92
<i>% of Time in Resultants</i>	16	43	17	4	2
<i>% of Time for 1D Roots</i>	30	42	79	95	98
<i>% of Time to Find Box Hits</i>	54	15	5	1	0

Table 8.1: **Timing results for root isolation.** Five test cases are shown for pairs of curves of varying degree and coefficient size, Timings are presented using both a heavily optimized 2D Sturm algorithm and the algorithm of Section 4.1. The time spent in the three main portions of the new algorithm is given in percentages.

timing estimates that are far worse than typical timings seen on real-world data. For example, a degree  $m$  curve and a degree  $n$  curve can intersect in up to  $mn$  real intersections, in general. In practice, the degree four (and higher) curves often encountered in boundary evaluation intersect at most once or twice in the region of interest. Also, for a given problem, input data is often specialized. Different problems may provide completely different input data, yielding completely different performance characteristics. For example, a problem other than boundary evaluation might typically encounter curves that meet in close to the maximum theoretical bound on the number of roots. Determining meaningful tests for problems that vary greatly in complexity for different cases is worthy of a great deal of further study.

The examples presented here provide a general feel for the capabilities of MAPC, the speed of various operations, and the relative time taken by various portions of the code. The examples here do *not* use the speedups provided by `fp-roots` and `PRECISE` [63] mentioned earlier. Either of those speedups would be likely to increase performance further. The timings in this section were first presented in a conference paper [57]. All timings given in this section are in CPU seconds on a 400 MHz Pentium II processor with 128 MB of memory.

### 8.1.2.1 Isolating Roots

Table 8.1 presents example timings for two-dimensional root isolation. Given two polynomials and a 2D interval, the timings are for isolating all roots within that

interval to a range no larger than 0.001 (in each coordinate) of the original interval. The table lists the degrees of the two input polynomials and the number of bits needed to represent their coefficients. Also shown is the number of intersections between the curves in the area of interest (i.e. the number of common real roots of the polynomials in the input interval).

The table lists times both for an implementation of 2D Sturm sequences, and for the algorithm presented in Section 4.1. Although a great deal of time was spent optimizing the 2D Sturm computation (see [58] for some of the optimizations), it proved significantly slower than the new approach on all test cases. Recall, however, that the 2D Sturm method has its own set of advantages and disadvantages. For example, it can handle certain cases (such as singularities) correctly, but has trouble when a root lies on one of the lines forming the boundary.

The bottom portion of the table shows the percentage of time spent in the three major sections of the new algorithm. The three sections are the resultant computation (used to convert the two bivariate polynomials into two univariate polynomials, one for each coordinate), the 1D root isolation (the time to isolate the potential roots in the coordinate directions), and the time for box hits (intersecting the polynomials with the box boundaries, and classifying the roots). Notice that for increasing complexity of the polynomials, the time spent in 1D root isolation begins to dominate. Even though the box hit tests also involve a number of 1D isolations, these are for much lower-degree polynomials, and thus are much faster. A simple derivation shows that when the input polynomials are of degree  $m$  and  $n$ , with bit lengths  $a$  and  $b$ , the univariate polynomials obtained by Sylvester's resultant can have degree  $mn$  and coefficient bit length  $(an + bm) + \log_2(m + n)$ . Another simple derivation shows that the univariate polynomials in the box hit tests have at most degree  $m$  with  $(a + tn) + \log_2 n$  bits per coefficient (or degree  $n$  with  $(b + tm) + \log_2 m$  bits), where  $t$  is the number of bits in the substituted value. This difference in degree and bit length will have a large effect on the time taken to isolate the roots of those polynomials.

Remember that some potential speedups (PRECISE and fp-roots) were not used in these timings. Such speedups directly affect the running time for univariate root isolation, and thus improve the total running time. As the results from [63] indicate, the use of PRECISE can improve the time for univariate root finding by more than an order of magnitude, even making the time to compute resultants dominate the time for 1D root isolation.

### 8.1.2.2 Curve Topology

Figure 8.2 demonstrates the performance of the curve topology algorithm described in Section 4.2. Given an algebraic plane curve (a bivariate polynomial), the curve is broken into a number of pieces, each of which is guaranteed to be monotonic in both of the coordinate directions. Listed in the table are the degree of the equation, the number of bits necessary to represent the coefficients, the number of turning points in the region of interest, and the number of separate components in the region (i.e. the number of curves the algebraic plane curve forms). The images show the algebraic plane curve, the points found on the curve (both turning points and points found in intermediate stages of the algorithm), and lines connecting those points in order. The time taken to resolve curve topology is divided into two portions. As is seen from the numbers, for all cases, the time to compute the turning points clearly dominates the other steps of the algorithm. Thus, speeding up 2D root isolation is of primary importance in speeding up resolution of curve topology.

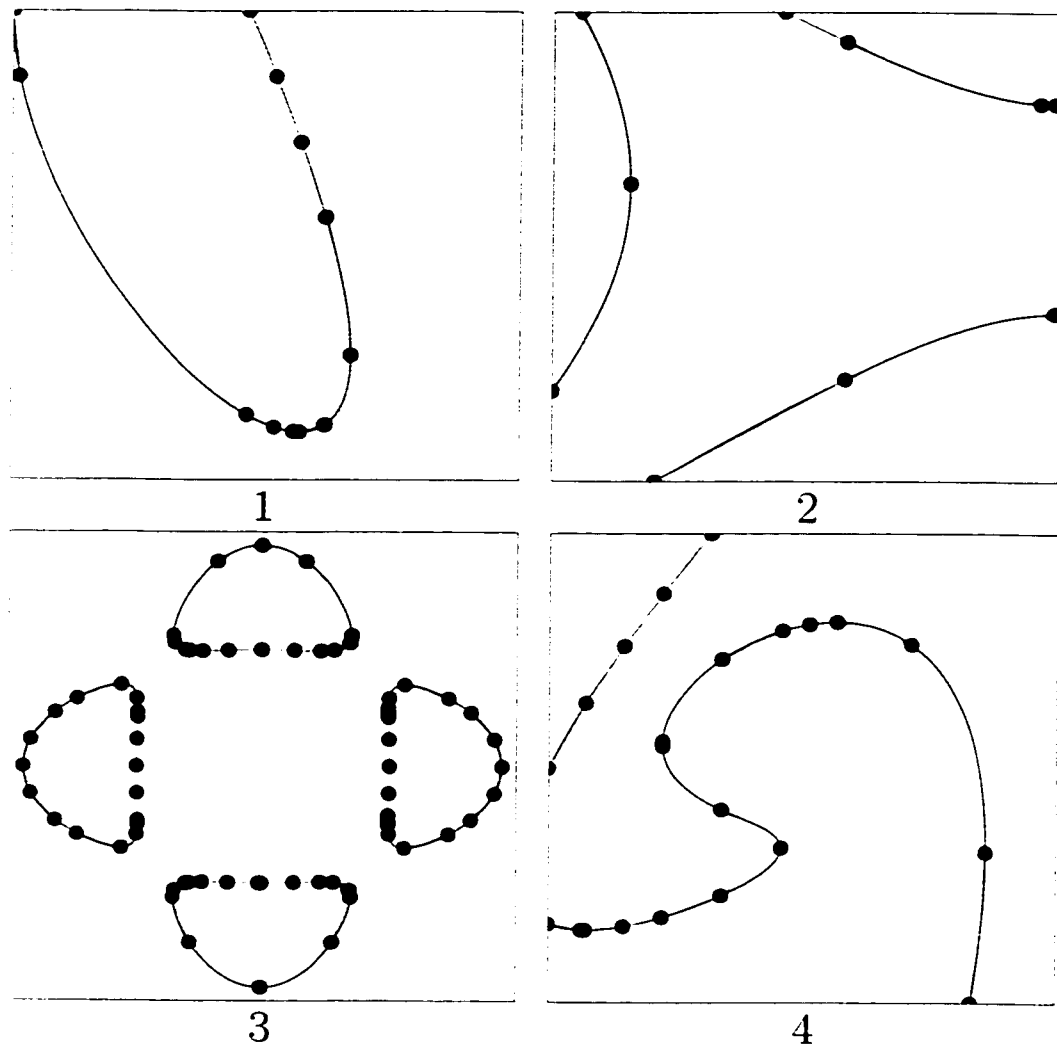
### 8.1.2.3 Sorting Points

Sorting points along a curve is an important part of the boundary evaluation algorithm (the ordering in curve correspondence, see Section 5.5). It is also an important part of finding the medial axis of a polyhedron [20] and is useful for other geometric problems as well. Point sorting is a well-known problem, and several methods have been proposed for it [52]. Point sorting routines are included in MAPC.

Figure 8.3 demonstrates one example of point sorting. In the figure, the degree three curve, shown in bold, is intersected with 25 other curves, shown by thinner lines. The other curves, which range from horizontal and vertical lines to degree five, intersect the bold curve a total of 52 times. These points are sorted along the curve shown in bold. Performing the 25 curve-curve intersections took a total of 102.3 seconds, while sorting the points, including resolving the topology of the curve, took less than one second. Again, 2D root isolation is the major component of the total running time.

### 8.1.2.4 Curve Arrangements

Given a set of curves passing through a domain, the curves partition the domain into a number of distinct regions. Determining that partitioning is referred to as the curve arrangement problem. Each individual region that is created by the partitioning, i.e.



<i>Case</i>	1	2	3	4
<i>Degree of Equation</i>	3	4	4	5
<i>Bits in Coefficients</i>	20	18	5	60
<i>Number of Turning Points</i>	2	3	24	5
<i>Number of Components</i>	1	3	4	2
<i>Time to Find Turning Points</i>	0.15	0.48	0.85	92.27
<i>Time to Run Algorithm</i>	0.04	0.06	0.21	0.09

Figure 8.2: **Curve topology algorithm results and timings.** The figures show four test cases along with the points that the curve topology algorithm finds, connected by lines. The table gives information about each curve within the domain, the time required by the algorithm to isolate the turning points, and the time required to run the topology algorithm.

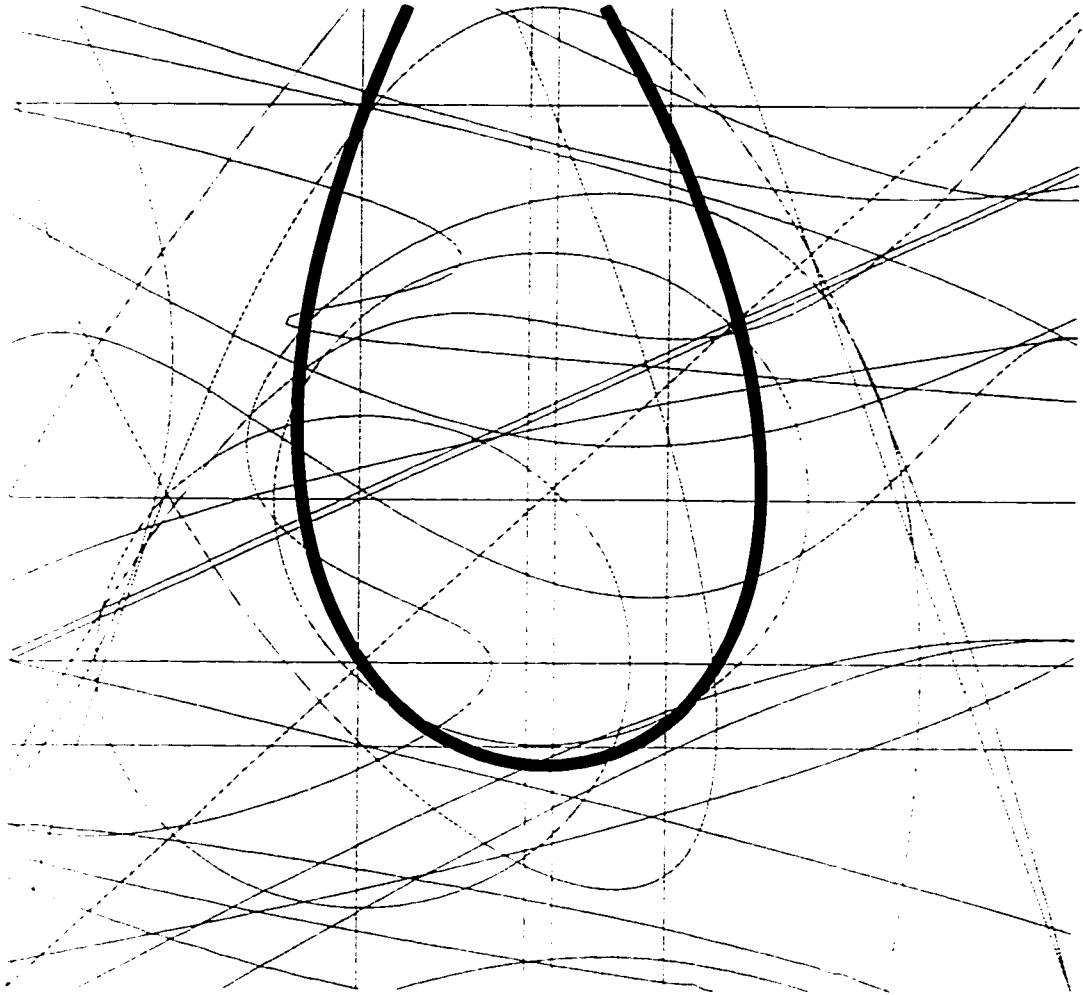


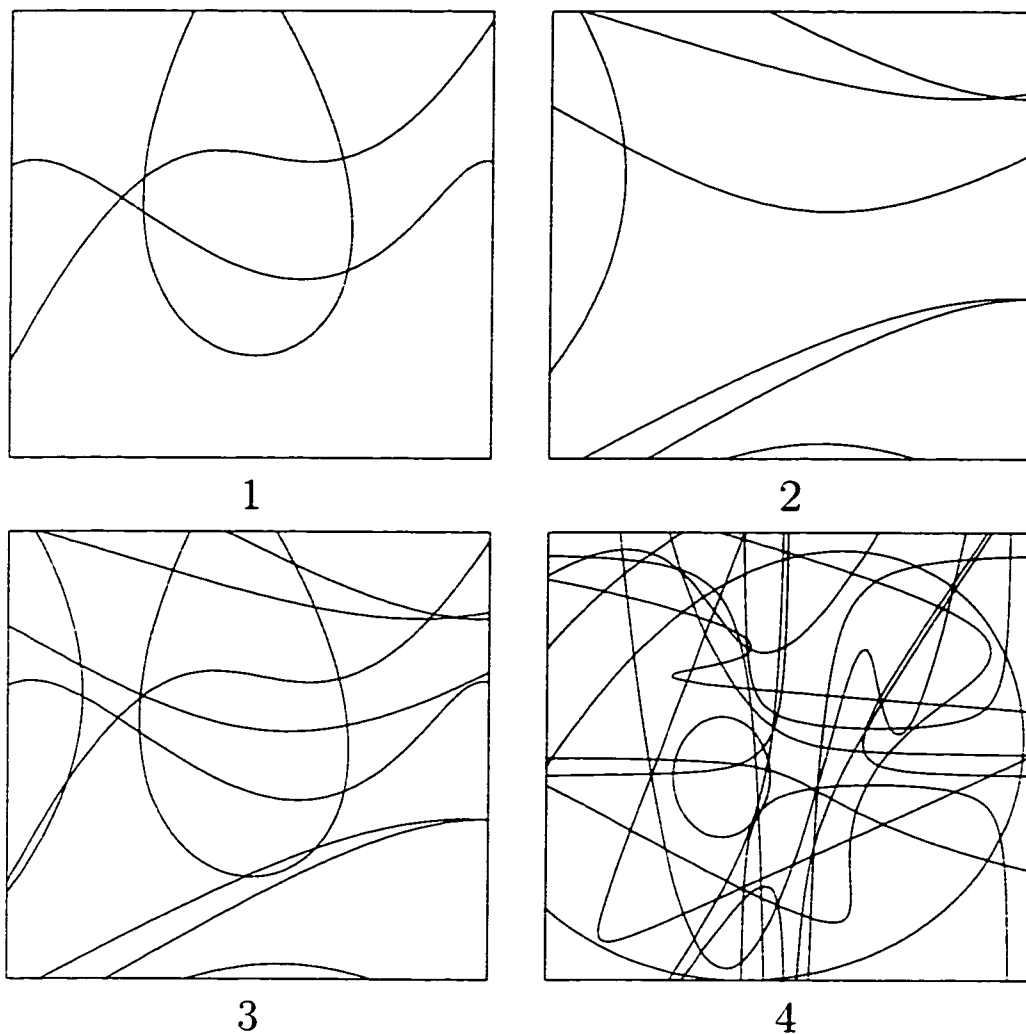
Figure 8.3: **Sorting points along a curve.** The curve along which the points are sorted is in bold. The points to be sorted are the 52 intersections of the bold curve with the other 25 curves shown. Finding all intersections takes 102.3 seconds, and the time to resolve curve topology on the bold curve and sort the points takes less than one second.

a continuous region bounded by curves and/or the domain boundary and without any other curves passing through, is called a face of the arrangement. Determining the arrangement means finding the set of curves bounding each face, along with the connectivity between faces. Most of the patch-level operations within boundary evaluation can be thought of as a curve arrangement problem. The intersection curves with the various patches and the trimming curves partition the region. Boundary evaluation finds the particular faces created by the arrangement that are a part of the final solid B-rep.

Figure 8.4 shows four curve arrangement examples. In each example, curves subdivide the region into a number of faces. The number of curves, as well as the maximum bit-length of the numerators and denominators of the coefficients, are listed in the table. A program was written to evaluate the curve arrangement, given a set of input polynomials (i.e. input algebraic plane curves). The table also lists the number of faces in the arrangement, as well as the total time taken to find all the faces. Finding the faces includes resolving the topology of each curve, intersecting each curve with every other curve and then subdividing the curves, finding all faces, and determining the ordered list of curves around each face, which also gives the connectivity between faces. Again, it should be emphasized that these timings are not the fastest possible. For example, applying PRECISE to the curve arrangement problem yielded speed improvements ranging from over 50% for the faster cases from Table 8.4, to more than 90% (i.e. over an order of magnitude) for the slowest cases.

## 8.2 ESOLID

ESOLID is a system for performing exact boundary evaluation. It includes classes to define patches and solids as indicated in Chapter 3, a program for conversion of CSG primitives and CSG trees from the BRL-CAD format to a directory-based format, and a program to read CSG data from a directory structure and convert the data to B-rep format, with optional output to a B-rep viewer program. Together, these classes and programs are used to perform exact conversions from BRL-CAD data to B-reps.



<i>Case</i>	1	2	3	4
<i>Number of Curves</i>	3	3	6	12
<i>Coeff. Bit size (Num./Den.)</i>	25/1	19/14	25/14	62/17
<i>Number of Faces</i>	9	11	31	171
<i>Time</i>	8.38	16.95	120.89	1142.21

Figure 8.4: **Arrangement of planar algebraic curves.** The figures show a region partitioned into a number of faces by the arrangement of curves. The application finds all subregions, the segments of curves bounding each subregion, and the connectivity between subregions. The table shows the bit-length of the coefficients of the curves (numerator bit-length, denominator bit-length), the number of faces generated by the arrangement, and the total time taken to compute the arrangement. The curves have maximum degree 4.



## 8.2.1 Implementation Details

ESOLID is written in C++, and is built on top of LiDIA and MAPC. ESOLID consists of approximately 13,000 lines of code, in addition to the code for MAPC.

Figure 8.5 gives a diagram of the major portions of ESOLID. The diagram illustrates the major classes and components of ESOLID, along with how each part can form other parts. For example, the `K_SOLID` class is made from `K_PATCH`s, and can be formed from boundary evaluation, from BRL-CAD conversion, or from a combination of `K_PARTITION`s and a `K_GRAPH`. Each of the major sections of ESOLID is described in the following subsections.

### 8.2.1.1 ESOLID Classes

ESOLID defines a new group of classes used to represent solids. This includes the `K_SURF`, `K_PATCH`, `K_PARTITION`, `K_GRAPH`, and `K_SOLID` classes. With the exception of the `K_GRAPH` class, these classes all rely on the classes defined in MAPC.

**Support classes:** Besides the classes shown in Figure 8.5, a few other supporting classes are provided. Three-dimensional intervals, similar to the `K_BOXCO2` class in MAPC, are provided. These allow points to be bounded in three dimensions. Also included is the `T_MATRIX` class to store  $4 \times 4$  transformation matrices. This class includes the ability to transform polynomials and perform an inverse transform, which is necessary when transforming the surfaces of a solid model.

**Surfaces:** The `K_SURF` class defines a surface. It can store both implicit and rational parametric forms of a surface, each defined using `K_RATPOLY`s from MAPC. In addition, a Bernstein basis representation can be computed and stored, for use in Bezier patch output. Very few operations are provided for surfaces. A surface can be modified by a `T_MATRIX` by computing a linear combination of the parametric polynomials and applying an inverse transform to the implicit form. The `K_SURF` class also computes a 3D axis-aligned bounding box for any interval in the parametric domain. This is useful for finding the 3D bounding box of a 2D point (possibly represented by an interval) in the domain, or for an entire patch.

**Patches:** The `K_PATCH` class implements patches, as defined in Section 3.1. A `K_PATCH` stores a wide range of data. A `K_SURF` specifies the patch surface, and a

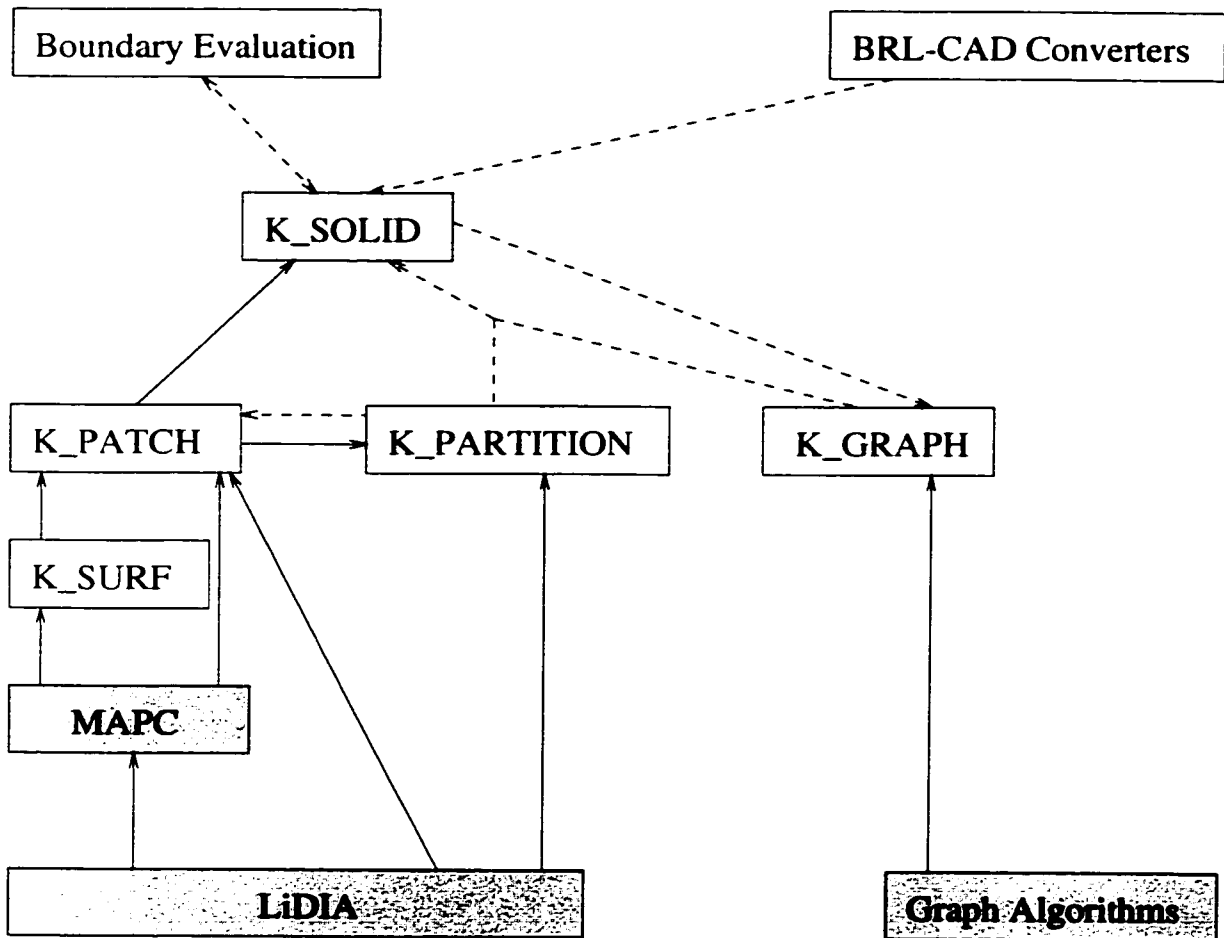


Figure 8.5: **The major parts of ESOLID.** The shaded boxes indicate external libraries used in the code. A solid arrow indicates that one library or structure is a necessary part of another. A dashed arrow from one structure to another means that the source structure can be used to form the destination structure.

2D interval specifies the patch domain. An array of `K_CURVEs` gives the trimming curves. Parallel arrays of pointers to `K_PATCHs` and `K_SURF`s list the adjacent patch and adjacent surface corresponding to each trimming curve. In addition, a `K_PATCH` contains data structures to hold the additional data generated during boundary evaluation. This includes parallel arrays for the intersection curves, similar to those for the trimming curves. In addition, separate structures to store the joincurves formed during curve merging are provided.

Several operations are provided for `K_PATCHs`. Routines are provided to determine whether a point is in the patch domain or patch trim region (i.e. 2D point location), split trimming curves, merge intersection curves, subdivide into many patches in order to split loops of intersection curves, recompute necessary domain boundaries, and output as an (approximated, not exact) trimmed Bezier patch, for display purposes. Most significant is the intersection routine, which performs all of the first stage operations of the boundary evaluation algorithm.

**Partitions:** The `K_PARTITION` class defines all data needed after the `K_PATCHs` are partitioned. Since each partition comes from a single `K_PATCH`, most information regarding which curves form the partition boundary are just references to the appropriate curve from the source `K_PATCH`. A `K_PARTITION` can be used to create a `K_PATCH`, thus allowing the formation of new solids once the appropriate partitions have been selected. `K_PARTITIONs` serve mainly as an intermediate data storage structure. Some of the functions of `K_PATCHs` are duplicated, with the only significant new function being 2D point generation.

**Graphs:** The `K_GRAPH` structure stores the topological structure of an entire solid. It maintains a node for each partition, along with adjacency between the partitions, and a record of whether the adjacency is along a solid edge or dashed edge (Section 5.8.2). The graph structure is computed from a solid once all partitions have been formed during boundary evaluation. 3D point location results are propagated through the `K_GRAPH` data structure to determine which partitions to use in the final solid. Although some basic graph algorithms are provided as part of a `K_GRAPH` (e.g. depth-first search and connected components), it is geared toward boundary evaluation and is not intended for use as a general graph data structure.

**Solids:** The `K_SOLID` class represents an entire solid. The only data stored by the class is the collection of `K_PATCHs` that make up the solid. The routines provided for

K\_SOLIDs include the ability to merge (i.e. join, Section 8.2.1.2) two solids, perform 3D point location, transform a solid by a T\_MATRIX, and output a solid as a collection of approximate trimmed Bezier patches (for display purposes). The fundamental operation in boundary evaluation, performing a Boolean operation, is also defined directly on this class, taking two K\_SOLIDs and the type of operation as input, and returning a single K\_SOLID.

### 8.2.1.2 Boundary Evaluation

The classes described in the previous sections provide all of the basic functionality needed for boundary evaluation. The Boolean operation defined on the K\_SOLID class is effectively the implementation of the boundary evaluation algorithm described in Chapter 5. In order to build a somewhat complex model, however, a CSG tree must be read, and CSG primitives must be converted to B-rep. ESOLID includes a program to do this.

**Primitives:** The program has several modes of operation, including ones to read a single primitive, read pairs of primitives and perform one Boolean operation, or read an entire CSG tree. In all cases, the program reads descriptions of CSG primitives in a high-level format. For example, a box is specified as eight corner points, given in a specific order. Since the CSG tree mode of operation is the most useful, it is the one described hereafter.

As is mentioned in Section 3.5, roundoff error in the input data can modify the nature of the input data. For example, four points that are supposed to form the corners of a face of a cube might not all lie in the same plane. One choice is to accept the data as given. Another choice, which is an optional feature of ESOLID, is to try to understand the designer's intent, and modify the input data slightly. So, in the case of the cube, for example, plane equations can be determined for each of the six faces, based on taking three of the four corner points for the face. New corner points can be found by intersecting those plane equations. If the new corner points are within some small tolerance of the old ones, the new corner points are used instead. Thus, the resulting faces are planar rather than bilinear (as would have been necessary otherwise), making future computation simpler. Although this approach does not treat input as exact, it is still consistent, since the input is only modified once at the time it is read in, and the new values are used consistently in all of the subsequent computations. Besides the approach just mentioned for boxes, a similar

option is provided in ESOLID to treat generalized cones. The vectors describing the cone face axes and centerline are checked for perpendicularity and parallelism, and the radii are compared for proportionality. Again, if the input data is within a small tolerance value, the data can be modified to ensure perpendicularity, parallelism, or proportionality. This helps to prevent a cone that is meant to have a degree two surface from instead being treated as a degree four surface.

**Operations:** Besides primitives, input can include Boolean operations and transformation data. Boolean operations include union, intersection, and difference, and operate on pairs of existing solids (i.e. solids already in memory). Transformations are specified as a  $4 \times 4$  matrix that represents the linear combinations to be applied to the parametric form of a single existing solid.

One other type of operation, the *join*, is allowed. The join operation takes as input a number of solids and blindly merges them together. If two disjoint objects are given as input, the join is equivalent to a union operation. Join operations often merge two objects with overlapping surfaces. This happens because a cavity will have been created in one object by subtracting another object that is later joined. If two objects being joined are not disjoint, the resulting model is probably invalid (since it contains self intersecting surfaces), although no explicit check of this is performed in ESOLID. A join operation is commonly used to group together distinct parts of a model into one model. For example, a description of a car model might join together the models of the engine, body, and frame. Generally, models resulting from a join operation should not be used in later Boolean computations, since they may be invalid.

**Input tree format:** The input CSG tree is stored in a directory tree structure that exactly mimics the CSG tree. For example, to represent  $C = A - B$ , a directory is created for  $C$ . The  $C$  directory has two subdirectories,  $A$  and  $B$ , each of which contains the description of the primitive objects. Within the  $C$  directory is a file specifying that a difference operation should be performed and giving the names of the two subdirectories. The  $C$  directory has two subdirectories,  $A$  and  $B$ , each of which contains the description of the primitive objects.

The boundary evaluation program reads the description file from one directory, and then recursively reads the description of subdirectories. When a leaf directory is reached (i.e. one in which a primitive is specified), a K\_SOLID is formed for that

primitive, and that K\_SOLID is passed back to the parent directory. For a Boolean operation, once a K\_SOLID has been found for each subdirectory, the Boolean operation is performed, and the resulting K\_SOLID is passed back. There is only one subdirectory for a transformation operation and once the K\_SOLID for that subdirectory is found, the solid is translated and passed up to the parent. For a join operation, there can be many subdirectories. The K\_SOLIDs from all of these are merged together to form a K\_SOLID that is, again, passed back to the parent.

**Output:** A K\_SOLID can be optionally output at intermediate stages (i.e. into a file in every subdirectory). The program implemented with ESOLID outputs all data in an approximate trimmed Bezier patch format. This is useful for visualization, but does not reflect the exact representation that is maintained within the program itself. The K\_SURF for each K\_PATCH in a K\_SOLID is converted to a Bernstein basis, and the patch domain is used to define a Bezier patch. The trimming curves in each K\_PATCH are tessellated into a number of linear segments (by intersecting the curves with a number of horizontal and vertical lines). The patch and trimming curves are output to a file that is viewed using an OpenGL program. Since no material properties are read on input or maintained during boundary evaluation, no material properties (such as color) are output.

**Usefulness:** The boundary evaluation program included with ESOLID serves well as a testbed to explore performance characteristics of the ESOLID implementation. It is not well-suited for more general applications, however. The input directory structure is unwieldy for most practical applications. The limitation of output to trimmed Bezier patches, rather than an exact format, makes the program difficult to integrate with a program that would like to use the exact K\_SOLID. For ESOLID to be used in any practical program, a new implementation, based directly on the ESOLID classes (described in Section 8.2.1.1) and designed specifically for that application, should be developed.

### 8.2.1.3 BRL-CAD Converters

Although the program for general boundary evaluation is useful, the input format it uses is not appropriate for real-world CSG data. The boundary evaluation algorithm is intended to be tested on data from the Bradley Fighting Vehicle model, which is in the BRL-CAD [26, 24] format. Thus, routines to convert BRL-CAD data to the

format described earlier are provided by ESOLID.

The BRL-CAD system contains a routine for traversing the internal CSG tree representation. This routine was modified to extract the necessary geometric data for the boundary evaluation program. The program takes an individual named part of a model as input, and outputs the description of that part in the directory format described in Section 8.2.1.2. Although BRL-CAD maintains material properties for each object, this is not passed on in the conversion.

Most of the conversion operations are straightforward. While BRL-CAD represents the CSG tree in a directed acyclic graph structure, it is easily converted to the expanded directory tree structure described earlier. The transformation data and Boolean operations map directly to the tree structure. Although primitives are defined slightly differently in BRL-CAD, the exact data needed for the input primitives in the boundary evaluation program is easily derived.

BRL-CAD assigns names to various objects. A named object can be formed from a series of operations. No name is given to the intermediate stages of object formation, so new names are generated for the directory format. Inspection of the Bradley Fighting Vehicle data set has shown that the vast majority of the operations performed are difference operations, with a smaller number of unions and an even smaller number of intersections involved. This is to be expected, since generally an object is formed by starting with a piece of material and removing unwanted parts (differences). A few pieces might need to be welded together (unions). Intersections have no analogous manufacturing operation, which is probably why they are rarely seen.

The one area where BRL-CAD data needs to be interpreted rather than directly translated is in the representation of joins. Although BRL-CAD supports the notion of joins, it was not clear how to extract this information (i.e. to distinguish joins from unions) within the tree walking routine that was being modified. For this reason, the data is interpreted as follows. Whenever a group of named objects is put together by union/join operations, and this grouped data is not used in any other Boolean operation other than possibly another union/join, the operation is interpreted as a join. Otherwise, the operation is interpreted as a union. It is likely that this reflects the design intent, since named objects are usually individual items, and thus not likely to be unioned. This also prevents any problems with invalid objects being formed by a join and used in subsequent operations.

Finally, BRL-CAD geometric data is provided as single precision floating-point

numbers. These numbers are interpreted as exact rational numbers, as mentioned in Section 3.5. Note, however, that the input data can be optionally modified to attempt to capture design intent, as described in Section 8.2.1.2.

## 8.2.2 Performance Results

Although the classes that are part of ESOLID can be used for other purposes, the fundamental goal of ESOLID is to perform boundary evaluation. This section discusses the performance of the ESOLID system on several boundary evaluation problems.

All of the timings here are given in CPU seconds on a 300 MHz MIPS R12000 processor. The timings include only the times to perform boundary evaluation once the input has been read in to the internal ESOLID format. The timings listed do not include the time taken to read the data from the input files and convert the primitives to a K\_SOLID. Although that time is usually not long, when a surface of degree four is implicitized (as is done for the torus and certain generalized cones), the time can be significant (tens of CPU seconds). Since all that information could potentially be computed or provided ahead of time, it is considered a preprocessing stage that is not part of boundary evaluation. Also, unless specified otherwise, the timings presented here do not include the speedups provided by the PRECISE library or fp-roots. Those speedups can dramatically affect the performance of the MAPC routines (particularly curve-curve intersection), and will usually result in an overall speedup in ESOLID.

### 8.2.2.1 Artificial Cases

This section deals with ESOLID performance on artificial examples. The examples presented here demonstrate the types of input that ESOLID can handle and ESOLID's performance on these basic examples. In these cases, the data has been created by hand, which generally means that the number of digits of precision necessary to specify the data is lower than for a real-world example, the relative configuration of objects is easily understood, and the examples are not complex.

Figures 8.6 and 8.7 show several basic classes of solids that can be handled in ESOLID. Each figure shows the result of a difference operation between two basic primitives. Not all possible primitives are shown. For example, generalized cones can be created, as well as polyhedra that are not parallelepipeds. For the first nine examples, Table 8.2 gives information about the nature of the primitives and the



boundary evaluation, while Figure 8.8 shows a rough breakdown of the timings.

Table 8.2 gives an indication of the importance of accurate computation when performing boundary evaluation. As seen in the table, even for relatively simple examples, a large number of curve-curve intersections can be performed. If even one of these intersections is inaccurate, the entire boundary evaluation may fail. The table also shows the number of algebraic numbers (univariate roots) that are found. Although many of these are part of intermediate computations (including the curve-curve intersection algorithm), an error in any one algebraic number has the potential to cause the entire algorithm to fail.

The table also shows the number of bits of precision to which the boundary evaluation algorithm found it necessary to evaluate the algebraic numbers. If  $n$  bits of precision were used, the intervals surrounding the algebraic numbers were found to a width of no more than  $2^{-n}$ . Note that the rational numbers used to represent the interval require even more bits of precision. The bits of precision table entry indicates the number of bits of precision to which algebraic numbers would need to be evaluated to guarantee correctness in a boundary evaluation algorithm similar to the one used here. IEEE double precision floating-point arithmetic provides at most 53 bits of precision. Thus, even for some of the relatively simple examples presented here (8.6(d), 8.6(e), 8.7(h)), IEEE double precision arithmetic can not provide the precision necessary to *guarantee* correctness. Even for cases such as Example 8.6(f), the accumulation of error makes it unlikely that IEEE double precision arithmetic would provide enough accuracy, in practice, to guarantee results.

The time taken for a Boolean operation is a function of several factors. These include the number of patches on each solid that intersect (or nearly intersect), the degree of the intersecting patches, the number of bits used in the representation of the data, and how close the input solids are to a degenerate configuration (which requires higher bit lengths). Figure 8.8 shows a breakdown of the timings for nine primitive-primitive examples. This figure shows that the time for curve-curve intersection computation (resultant, Sturm, and other curve-curve intersection computation) dominates the time for other computations in all of the longer-running examples. Notice (from Table 8.2) that these are the cases involving higher-degree surfaces (and thus higher-degree intersection curves). Within the curve-curve intersection computation, both the resultant computation and the Sturm sequence calculations take significant amounts of time, and are the primary components of curve-curve intersection in all but the faster examples. While the resultant computation always takes

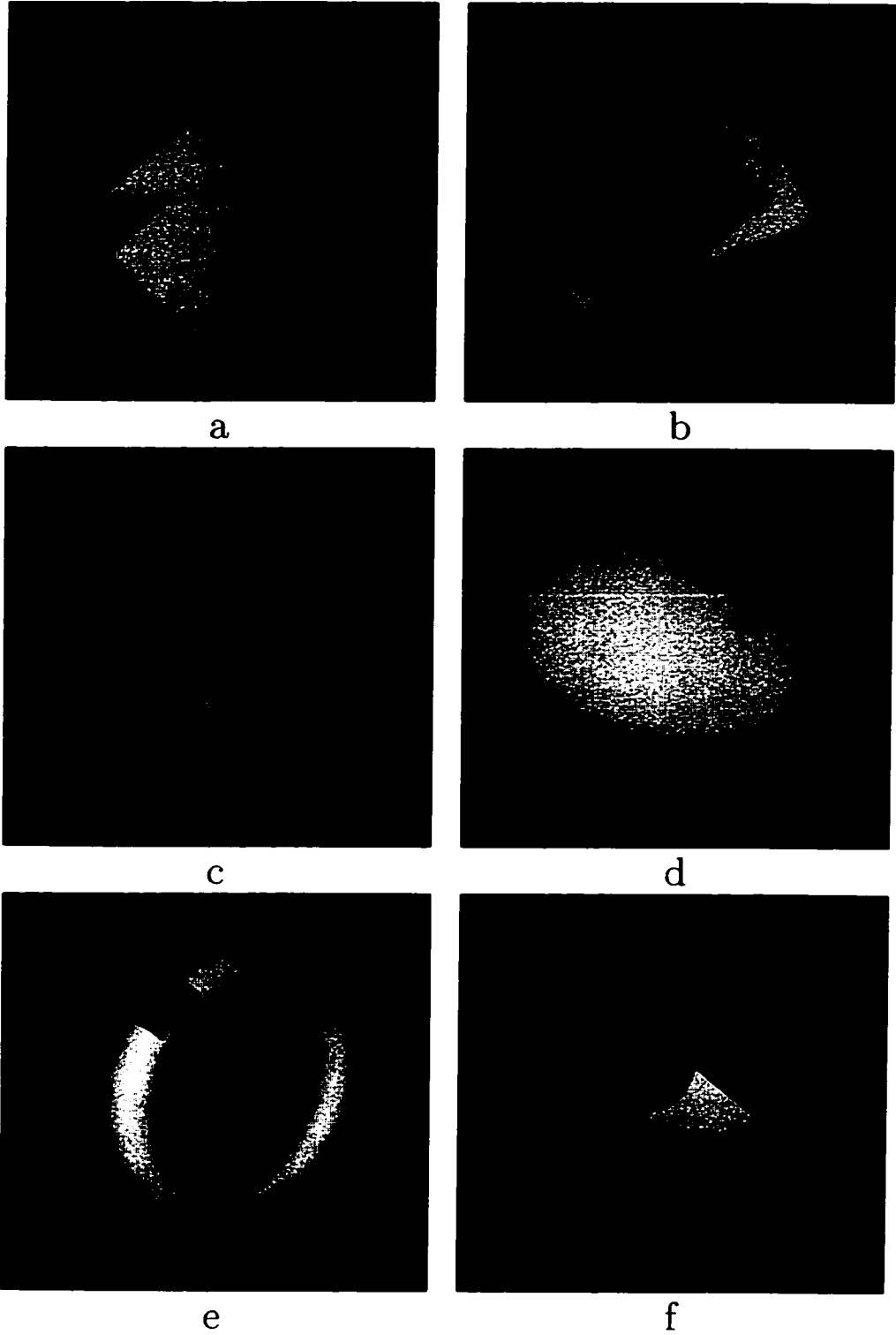


Figure 8.6: **The results of a difference operation on pairs of primitives.**

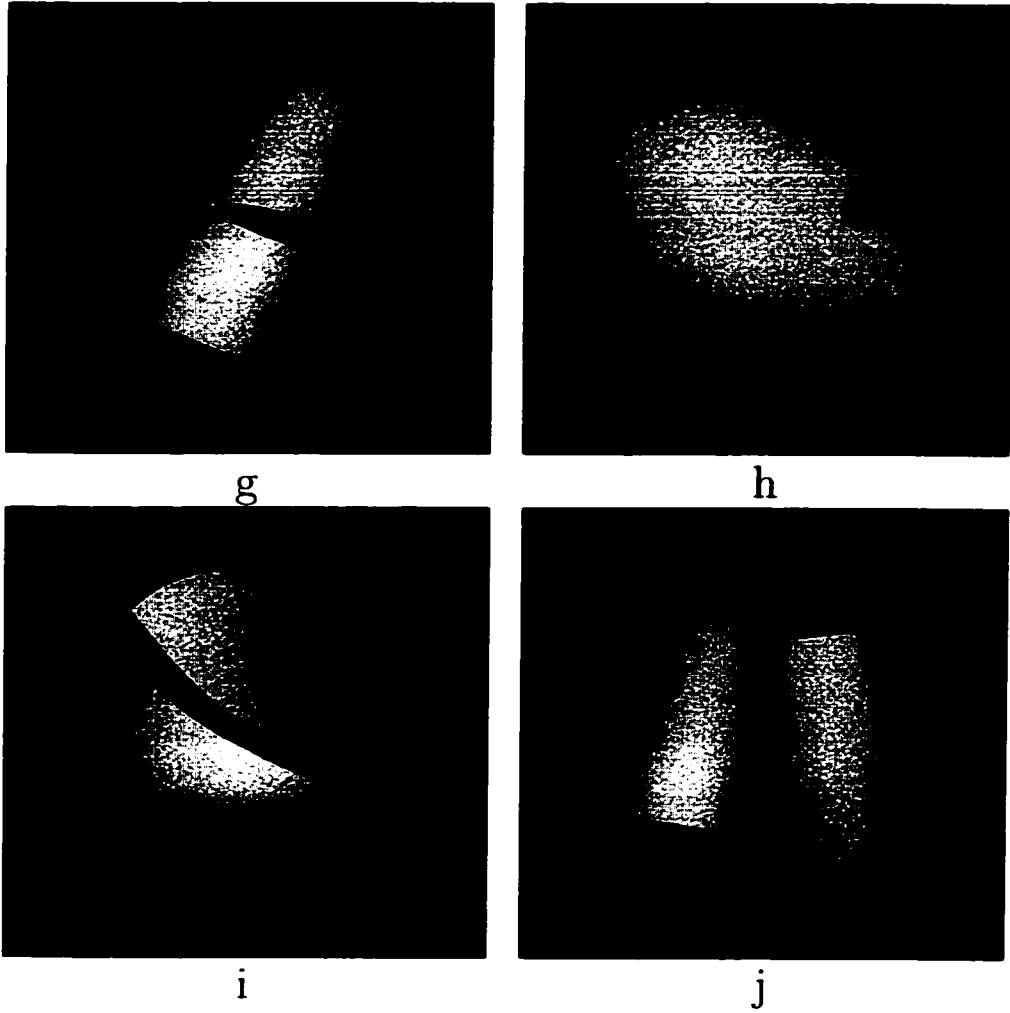


Figure 8.7: **The results of Boolean operations on pairs of primitives.** Examples g, h, and i are the results of a difference operation. Example j is the result of a union operation.

<i>Example</i>	a	b	c	d	e	f	g	h	i
<i>Object 1</i>	box	box	cyl.	ell.	torus	twist	cyl.	ell.	ell.
<i>Object 2</i>	box	twist	box	box	box	cyl.	cyl.	cyl.	twist
<i>Degree of Object Surfaces</i>	1,1	1,2	2,1	2,1	4,1	2,2	2,2	2,2	2,2
<i>Number of Intersecting Patches</i>	6	12	6	8	8	8	4	8	9
<i>Maximum Degree of Intersection Curves</i>	1	2	3	2	4	6	6	6	4
<i>Number of Curve-Curve Intersections</i>	90	551	394	990	447	562	407	881	744
<i>Number of Univariate Roots Found</i>	0	240	353	1268	900	2004	607	2248	1753
<i>Bits of Precision in Algebraic Numbers</i>	-	10	13	59	87	52	31	87	25
<i>Total Time</i>	0.39	1.35	0.90	4.62	8.25	22.05	5.50	49.41	28.83

Table 8.2: **Details of the difference operations illustrated in Figures 8.6 and 8.7.** Object 1 describes the base primitive, while Object 2 describes the primitive being subtracted. The primitives shown are a box (polyhedron), twist (a box twisted so that some faces are bilinear patches), cylinder, ellipsoid, and torus. The degree of the surfaces in the two objects is given, followed by the number of pairs of patches that actually intersect. The maximum degree (in the parametric domain) of the intersection curves is also shown. The total number of curve-curve intersection operations performed is given, along with the total number of univariate roots found (i.e. the number of algebraic numbers found as a root of a univariate polynomial). The maximum number of bits of precision used to represent these algebraic numbers is given, followed by the total time taken to perform the Boolean operation.

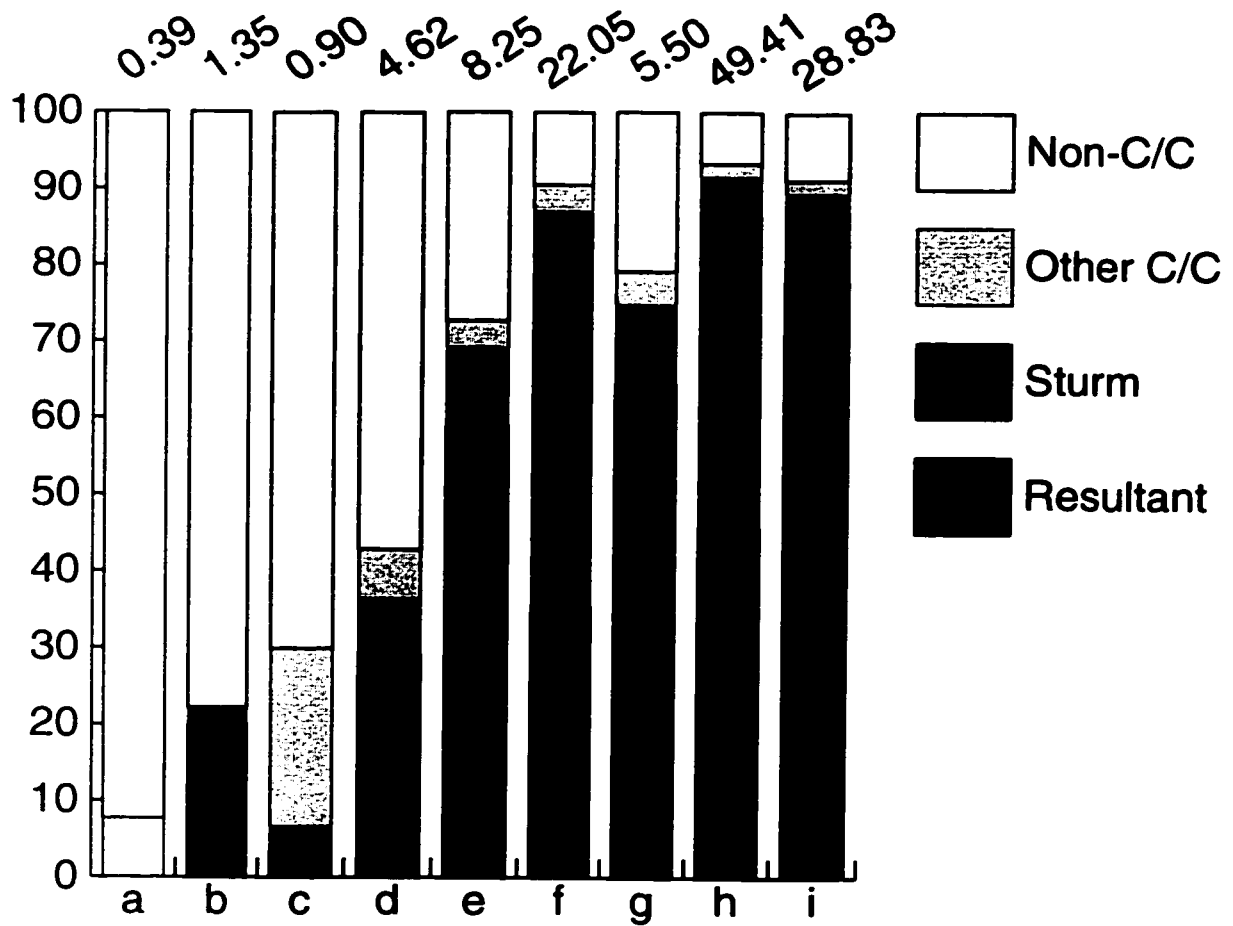


Figure 8.8: **Timing breakdown for examples from Figures 8.6 and 8.7.** For each example, the percentage of the overall boundary evaluation time is shown (labeled along the vertical axis). The two major parts of curve-curve intersection (resultant and Sturm sequence calculations) are shown, along with the times spent in other parts of curve-curve intersection and outside of curve-curve intersection. The example number is shown at the base of each bar, and the total time taken on that example (in seconds) at the top.

a significant amount of time, the Sturm computation varies, occasionally matching or exceeding the time for resultant computation. Recall that many of the speedups discussed earlier (and implemented in ESOLID) have focused on increasing the efficiency of the Sturm portion, which dominates time in a naive approach. From this information, it is apparent that the speedups in Sturm sequences have been effective, shifting the performance bottleneck toward the resultant computation.

Example 8.7(j) shows one example of a nearly degenerate configuration of primitives (also illustrated in Figure 1.5). Two cylinders, each of radius one and at a slight angle to each other, barely interpenetrate. ESOLID correctly computes the Boolean operation, even when the depth of penetration is extremely small. Table 8.3 shows how the time and precision required increases as a function of the depth of interpenetration. This example demonstrates that ESOLID performs correctly on examples that are hard to compute in a system based on IEEE floating-point arithmetic. Such a system can not accurately represent even the input to this problem. Notice that as the depth of penetration becomes smaller (i.e. the cylinders are closer and closer to tangent), the precision required increases, and thus the time taken for boundary evaluation increases. Also notice that while the time for resultant computation increases at a near-linear rate with the depth of penetration, the time for the Sturm computation increases much more rapidly. Therefore, as higher precisions are required, the efficiency of Sturm computations becomes more important.

### 8.2.2.2 BRL-CAD Data

The practical performance of a system can only be judged by testing it on realistic data. For this reason, the ESOLID system has been applied to CSG models from the Bradley Fighting Vehicle. This data, developed in the BRL-CAD system by the Army Research Lab, is considered *real-world* data, since it was developed to be used in a real application. That is, the data was not created for the purpose of testing boundary evaluation.

In this section, examples are used that are taken directly from the Bradley Fighting Vehicle data set. The BRL-CAD conversion routines are applied to convert the data to the directory format, which is then read in and processed by the boundary evaluation program. Unless specified otherwise, the input modification described in Section 8.2.1.2 is used on the BRL-CAD data given here, to try to better capture design intent.

Where possible, the performance of ESOLID is compared with that of BOOLE

Depth of Penetration ( $10^{-x}$ )	Precision Required (bits)	Total Time (s)	Sturm Time (s)	Resultant Time (s)
3	20	8.64	2.19	4.17
6	20	12.45	4.14	5.61
9	25	17.25	7.23	7.47
12	30	22.98	11.13	9.15
15	40	33.21	17.07	11.88
18	52	47.46	24.66	14.46
21	58	60.15	32.64	18.15
24	62	86.76	47.64	22.80
27	68	147.66	99.75	26.37
30	71	120.36	74.79	29.64
33	77	164.01	108.03	34.41
36	117	205.17	143.40	38.34
39	88	446.28	357.63	46.89
42	141	317.55	237.15	49.11
45	96	385.80	296.19	55.80

Table 8.3: **Timing results for example j from Figure 8.7.** The depth of penetration of the two cylinders is given in the first column. Following that is the maximum precision required in the boundary evaluation algorithm to represent the algebraic numbers exactly.  $n$  bits of precision required means that algebraic numbers were determined to an interval of width no smaller than  $2^{-n}$ . The total time to perform boundary evaluation is listed, followed by the time spent in Sturm computations (both generation and evaluation of Sturm sequences), and in resultant computations.

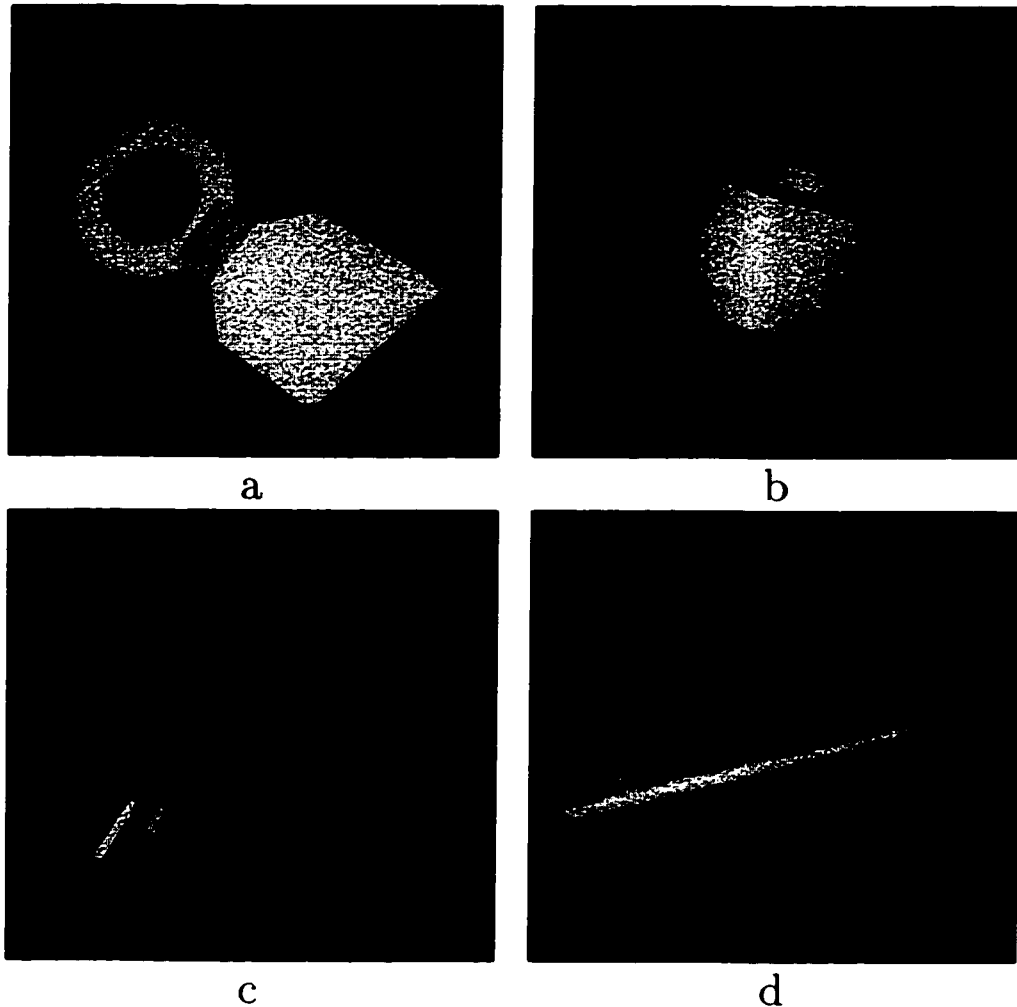
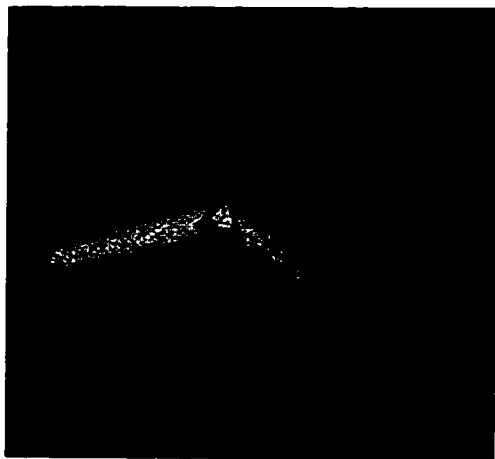


Figure 8.9: **BRL-CAD examples where both ESOLID and BOOLE worked.**

on the same data. The BOOLE system [61], like ESOLID, has been developed to perform boundary evaluation on BRL-CAD data. BOOLE, however, is implemented using inexact floating-point computation. Although local and global tolerances are used in the BOOLE system to deal with some numerical error, they are insufficient to deal with all numerical problems. The inability of BOOLE to deal with all numerical difficulties motivated the work on ESOLID, and so some examples where BOOLE fails but ESOLID succeeds are also included.

Figure 8.9 shows some objects from the Bradley. The boundaries of these objects were successfully evaluated in both ESOLID and BOOLE. Figure 8.10 shows other objects from the Bradley model. The boundaries of these objects were successfully

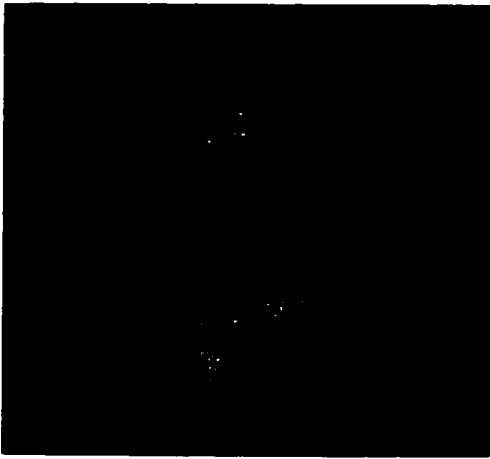




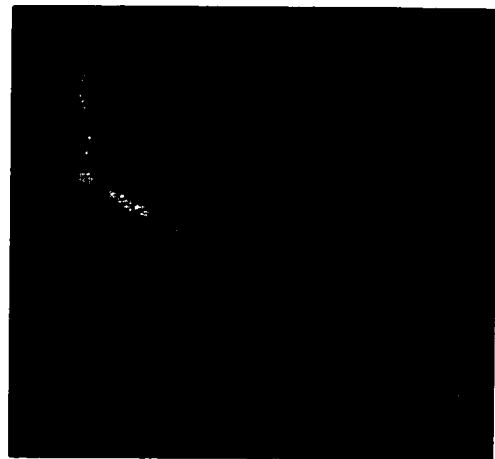
e



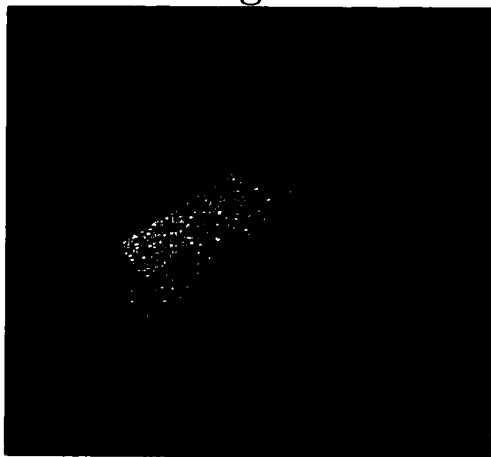
f



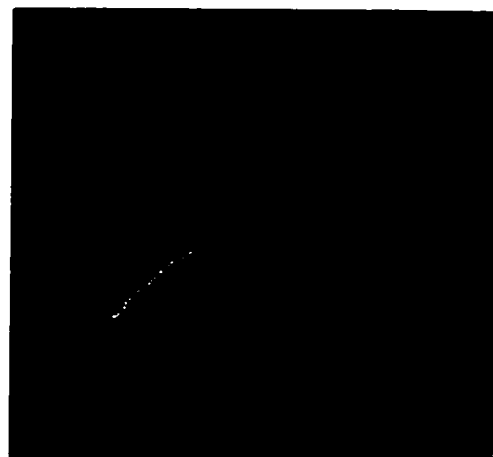
g



h



i



j

Figure 8.10: **BRL-CAD** examples where **ESOLID** worked and **BOOLE** failed.

Example Number	Name	Number of Booleans	ESOLID Time	BOOLE Time
a	Tow Hook	2	10.23	2.23
b	Wheel Assembly	4	12.57	2.81
c	M16 Rifle	6	633.42	6.68
d	Track Link	11	132.48	27.74
e	Relay Mechanism	1	250.74	-
f	Crew Member 3	2	26.37	-
g	Launcher Mount Part	3	63.15	-
h	Support Assembly Part	6	213.72	-
i	Rear Hatch Hinge	7	58.92	-
j	Engine Access Hatch	16	54.78	-

Table 8.4: **Timings for the examples from Figures 8.9 and 8.10.** The number of Boolean operations performed is shown, along with the time taken under ESOLID and BOOLE. A ‘-’ indicates that boundary evaluation failed for that object.

determined in ESOLID, but failed under BOOLE. A discussion of why the failures occurred in BOOLE is given below. Table 8.4 provides further data on these examples, including the number of Boolean operations needed to create the objects, and the time taken in ESOLID and BOOLE. Note that most objects are created by a series of join operations (Section 8.2.1.2). Join operations do not count as Booleans. Thus, the number of Boolean operations may be more or less than is obvious from an external view. For example, Crew Member 3 (Example 8.10(f)), requires only two Boolean operations. The wheel assembly (Example 8.9(b)) requires four Boolean operations, mainly to handle the axle that is hidden in the external view.

As shown in Table 8.4, the time taken by ESOLID is within two orders of magnitude of the time taken by BOOLE, for the examples tested. For three of the four examples, ESOLID is less than one order of magnitude (less than a factor of five, in fact) slower than BOOLE. The goal of achieving times that are within one to two orders of magnitude slower than a fixed-precision implementation (Section 1.5) is clearly achieved.

Table 8.5 provides a breakdown of timings under ESOLID for all of the examples. The data shown in the table confirm that curve-curve intersection continues to play a dominant role in determining the overall running time, just as in the artificial examples in Section 8.2.2.1. In some cases, the time for resultant computations dominates the time for curve-curve intersections, while in other cases, the time for

Example Number	Number Curve-Curve	Number Univar. Roots	Max. Precision (bits)	Total Time	% Curve-Curve	% of Total in Resultant	% of Total in Sturm
a	425	1831	42	10.23	68.0	54.3	5.0
b	637	1106	59	12.57	54.2	46.8	1.9
c	1003	3834	57	633.42	98.2	3.6	96.0
d	4444	13511	75	132.48	74.9	64.6	3.7
e	320	6311	41	250.74	95.1	15.6	76.0
f	315	2259	45	26.37	81.6	71.1	4.9
g	974	5227	65	63.15	81.7	63.6	13.2
h	1162	7116	66	213.72	92.5	35.8	54.7
i	1266	8191	87	58.92	69.1	57.4	5.3
j	1799	5334	69	54.78	64.2	55.0	3.8

Table 8.5: **Timing breakdown (under ESOLID) for the examples in figures 8.9 and 8.10.** The number of curve-curve intersections is given. The number of algebraic numbers found as roots of univariate polynomials are shown, along with the maximum number of bits of precision used to represent these algebraic numbers. The total time is shown, along with the percentage of time spent in curve-curve intersection, the major component of the algorithm. The percentage of total time spent in the two major components of curve-curve intersection, resultant computations and Sturm computations (generation and evaluation of Sturm sequences), is also shown.

Sturm sequence computation dominates. From the data, it appears that for the slower cases, curve-curve intersections, and in particular Sturm sequence calculations, take a greater percentage of the time.

Notice that all examples use a significant amount of precision in determining algebraic numbers. Although computations that do not provide that level of precision can still succeed (as evidenced by the fact that BOOLE worked on several cases), an implementation that does not guarantee that level of precision is prone to failure.

It is not always clear why the BOOLE system fails on certain examples. Although lack of accuracy is certainly a problem (as was determined in earlier studies of BOOLE [62]), other failures may be due to more general programming bugs. Although it is not feasible to examine the inner workings of BOOLE, some information can be gained by examining the error messages printed by BOOLE and observing the examples that BOOLE fails on.

Two such examples are shown in Figure 8.11. Example 8.11(a) shows a close-up view of the area of intersection between the two cylinders forming the Relay Mechanism in 8.10(e). The intersection curve is shown in the patch domain in figure Example 8.11(b). This intersection curve appears to be nearly singular. In fact, the curve is not singular (it has two separate components), and ESOLID correctly resolves the topology of the curve. BOOLE, on the other hand, exits with an error that it has found a singularity or two components that are too close together. It is possible that using much tighter tolerance values would allow BOOLE to successfully evaluate the boundary, but it is also possible that the IEEE floating-point computations in BOOLE could not provide an appropriate level of precision to guarantee correctness. This is a case where the exact computation of ESOLID allows a computation to be performed that would otherwise cause problems.

A second example is shown in 8.11(c) and 8.11(d). It shows one of the Boolean operations from the Crew Member 3 example (8.10(f)). A difference operation is performed on the solids in 8.11(c), resulting in the solid shown in 8.11(d). BOOLE fails on this Boolean operation, with an error that "curves did not close," which indicates that there is a significant error in the intersection curve computations. Although there may be many reasons for such an error to arise, a brief examination of the two solids clearly shows that the two solids are nearly tangential. Thus, a slight error in the position or orientation of either solid can have a significant impact on the intersection between those solids. This type of example is highly prone to numerical error, and it can be surmised that such numerical error causes BOOLE's failure.

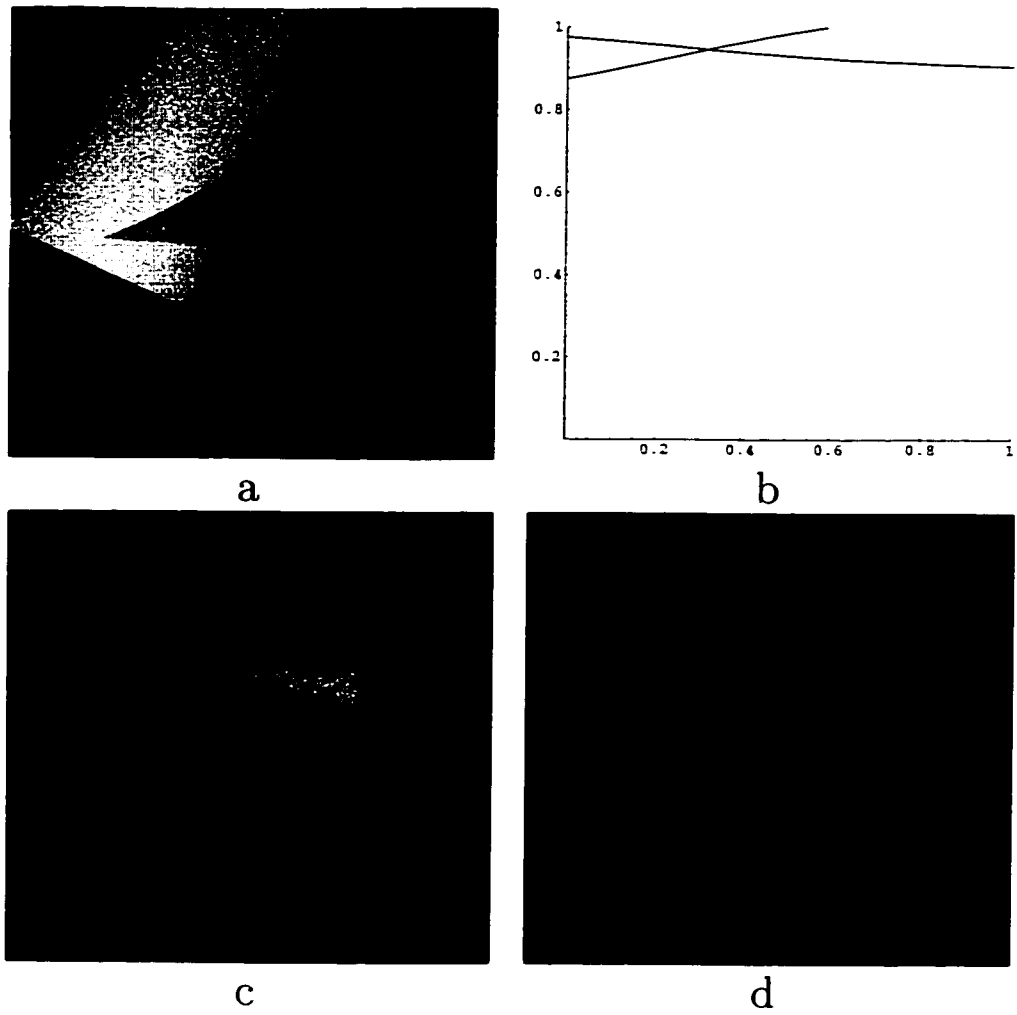


Figure 8.11: **Close-up views of Boolean operations where BOOLE fails.** (a) is a close-up view of the Relay Mechanism (Example 8.10(e)). Note that the intersection between the cylinders is nearly singular. (b) shows a graph of the intersection curve for two patches from the solids shown in (a), over the domain of one of the patches. Even though the intersection curve appears singular, it is actually two separate components. (c) and (d) are from the computation of Crew Member 3 (Example 8.10(f)). A difference operation is performed on the solids shown in b, resulting in the solid shown in (c). The intersection is small, and is formed from two nearly tangential solids.

Example Number	without PRECISE		with PRECISE	
	Total Time	Sturm Time	Total Time	Sturm Time
a	10.23	0.51	10.95	1.62
b	12.57	0.24	12.69	1.44
c	633.42	597.33	42.99	6.93
d	132.48	4.95	137.64	11.55
e	250.74	190.62	73.86	15.36
f	26.37	1.29	28.14	3.63
g	63.15	8.34	61.26	6.36
h	213.72	116.88	105.99	9.90
i	58.92	3.15	63.48	8.55
j	54.78	2.07	58.44	6.66

Table 8.6: **Timings for the examples from Figures 8.9 and 8.10, with and without the incorporation of the PRECISE library.** PRECISE is used to improve the efficiency of Sturm sequence calculations. The total time and the time spent in Sturm computations is shown.

As mentioned earlier, the PRECISE library [63], has been developed to speed up Sturm operations. PRECISE uses error-bounded arbitrary precision (*not* IEEE standard) floating-point operations to perform certain computations. It is particularly useful when the sign of a polynomial is desired, which makes it well-suited for speeding up Sturm computations.

Table 8.6 shows the performance improvement when PRECISE is applied to the BRL-CAD examples. PRECISE affects the performance of only the Sturm sequence portion of ESOLID. As seen in the table, PRECISE can dramatically lower the running time of the most Sturm sequence intensive examples. In the M16 example (8.9(c)) the Sturm code is made almost two orders of magnitude faster. This makes the overall computation more than an order of magnitude faster, bringing the M16 example in ESOLID/PRECISE to less than an order of magnitude slower than BOOLE. PRECISE incurs a certain amount of overhead, which is the reason that several of the low-Sturm time examples actually become slower when run with PRECISE.

# Chapter 9

## Summary and Future Work

This dissertation describes the representations and operations necessary to perform exact boundary evaluation for solids with low-degree curved boundaries. This chapter reviews how the thesis of the dissertation is demonstrated and discusses avenues for future research.

### 9.1 Thesis

The thesis of this dissertation is the following:

Accurate boundary evaluation for low-degree curved solids can be performed efficiently using exact computation.

The thesis is proved as follows:

- Exact representations for curved solids are proposed (Chapter 3).
- On top of these representations, exact kernel operations are proposed (Chapter 4).
- Using both the representations and the kernel operations, an algorithm for exact boundary evaluation is proposed (Chapter 5).
- A number of speedups are used to increase the efficiency of the computations, while maintaining exactness (Chapter 6).
- The representations, algorithms, and speedups are implemented and applied to low-degree real-world data (Chapter 8).

- Exact boundary evaluation is performed in times within two orders (and often within one order) of magnitude of those for a similar inexact approach.
- The exact implementation handles cases on which the inexact approach fails, due to insufficient precision.

The exact representations and computations ensure accurate, exact computation on curved solids. The speedups make the boundary evaluation for low-degree solids efficient, as demonstrated by its application to real-world data. Thus it is shown that accurate boundary evaluation for low-degree curved solids can be performed efficiently using exact computation.

## 9.2 Future Work

There are many avenues available for future work related to exact boundary evaluation on curved solids. This section describes some of the major areas open to further study. Some of these areas are already being explored.

### 9.2.1 Increasing Efficiency

Although the implementation of the boundary evaluation algorithm is already well within the goal of one to two orders of magnitude slower than an equivalent fixed-precision approach (BOOLE), an even faster implementation would be better. Some work in this area is currently being pursued (e.g. PRECISE [63]).

Potential ways of increasing efficiency include:

- **Quick rejection tests:** Although several quick rejection tests are already in use, other quick rejection tests are possible, and the current tests can be made more useful. For example, the bounding box tests (based entirely on affine arithmetic) currently being used are much more conservative than necessary.
- **Floating-point speedups:** Incorporating floating-point methods has already proved valuable in increasing efficiency. Currently, floating-point optimizations are used in only 1D Sturm sequence computations. Floating-point speedups can be applied to other areas of the algorithm as well.
- **Lazy/adaptive coefficient determination:** While algebraic numbers are handled in a lazy manner, the polynomials associated with them are not. The



coefficients of these polynomials are always determined as exact rational numbers. This determination may involve time-consuming resultant computations and interpolations. It may be possible to adaptively determine the coefficients, finding them to higher precision only as necessary. The possibility of using the PRECISE library [63] to do this is being explored.

- **Lazy curve topology:** Resolving curve topology (and in particular, finding turning points) can be a time-consuming process. Topology is resolved for every intersection curve, but is not always needed. A lazy curve topology approach would wait to resolve curve topology until a time when it is actually needed. Such an approach would require the boundary evaluation algorithm to be changed, but could save significant time.
- **Cache coherency:** The memory use characteristics of the ESOLID implementation have not been explored in detail. If a significant amount of time is spent in memory-related operations, it is worth investigating ways that cache coherency can be used to reduce memory access times. Several interesting issues related to this, such as predicting memory usage characteristics of arbitrary-length numbers, could be explored as a result.

### 9.2.2 Handling Degeneracies

Dealing with degeneracies remains a major obstacle in achieving a fully robust implementation. Chapter 7 addresses several issues related to degeneracies, and briefly explores some potential methods for alleviating the problems associated with degeneracies. Completely dealing with degeneracies in a practical manner will require much more work.

One way to approach degeneracies is to handle only certain classes of degeneracies at a time. For example, it would be reasonable to first handle only degenerate cases where the resulting solid is still manifold. Later work could focus on extending the boundary evaluation algorithm to handle degeneracies for all regularized operations.

In any case, exact computation is important in any rigorous approach to handling degeneracies. Thus, the exact algorithm outlined in this dissertation is a basis for future work addressing degeneracies.

### 9.2.3 Implementation

There are several ways that the ESOLID system implementation can be improved. These include:

- **Extended input:** Currently, the methods for providing input are limited. Some direct ways to increase the input sources available include:
  - extending the types of input primitives allowed.
  - adding converters from CSG-based systems other than BRL-CAD, and
  - creating a program that allows interactive creation of input.
- **Integration with other systems:** The library classes, particularly those of MAPC, could be incorporated into other systems. In addition, the exact boundary evaluation implementation could be incorporated into a more general solid modeling system.
- **I/O of internal format:** Currently, ESOLID takes primitive descriptions and a CSG tree as input. Output is either an approximate Bezier patch format (for viewing) or a human-readable format. The human-readable format is useful for testing ESOLID, but cannot easily be read back into ESOLID's internal format. The ability to directly read into and write from the internal ESOLID format would be very useful. A different internal representation (such as a global list of polynomials) is needed to allow such I/O.
- **Parallelism:** The intersection between any patch pair (the first stage of the boundary evaluation algorithm) is highly parallelizable. Each pair of intersecting patches could be given to an individual processor. Also, much of the second stage of the boundary evaluation algorithm (up through patch partitioning) is parallelizable, since each patch can be processed by a separate processor. Such parallelism was implemented in BOOLE [62] with excellent results, and similar results could be expected for ESOLID.
- **Incorporating other libraries:** Other libraries can be incorporated to replace certain computations being performed now. For example, all rational number arithmetic is currently performed in LiDIA [11], but other rational number libraries might provide better performance. Similar changes could be used to replace other computations, such as the floating-point root finding routines.

A different implementation might provide closer approximations to the roots, making the speedup more effective.

#### 9.2.4 Long-Term Directions

Besides the relatively short-term directions for future work proposed above, there are several longer-term and more fundamentally significant areas that can be explored. A few of these are mentioned here.

- **Capturing design intent:** As is mentioned in Section 8.2.1.2, capturing the designer's intent is an important consideration. In fact, it can be argued that capturing the designer's intent is more important than accurately handling the geometry. The methods for determining designer's intent will likely vary depending on the field that the boundary evaluation input comes from, but in any case, significant further study in this area seems warranted.
- **Increasing robustness:** The theme followed in this dissertation has been to ensure exactness first, then increase efficiency. For the near future, however, it is difficult to imagine that many real-world applications will be based on exact representations. Integration of an exact method (such as exact boundary evaluation) into an otherwise inexact program therefore seems like the most likely path for methods to be used. Determining exactly how exact and inexact code can be integrated to achieve greater robustness overall is a topic worthy of further study.
- **Other representations:** This dissertation has focused on the conversion of CSG models to B-rep models. Other model representations, such as skeletal models and volumetric models, are also commonly used. The exact methods described in this dissertation can be used in other conversions, as well. Some work in this area is already being explored [20].
- **Finding general test cases:** One issue which repeatedly arose in the work on this dissertation was how to test an algorithm or routine on a *general* test case. For example, to test the curve-curve intersection routines, it seems useful to have a method for forming completely random polynomials. Real-world data, however, often has a regular and characterizable structure, and thus is not close to random. Data from different real-world applications will have different

structures. For example, one application might generate polynomials where the bit-lengths of all coefficients are nearly equal, while another application might create polynomials where the bit-length of the coefficient of the leading term is far less than that of the last term. The structures of mathematical systems that arise in certain sciences and engineering fields are well understood, and computation methods can be optimized to handle those particular cases. It would be useful to have a similar understanding of the nature of data that arises in real-world boundary evaluation problems.

- **Exact computation in hardware:** One of the reasons why exact computation is so slow, in general, is a lack of direct computer hardware support. While it may not be possible (or even desirable) to have complete hardware support for exact computation, any support can be used to narrow the inherent efficiency gap between exact and inexact computation. Finding what features of exact computation would be useful and feasible to support in hardware is worth significant further study.

# Appendix A

## Parametric Solids

This chapter describes the parameterizations that are used for a few common solids. These include all of the standard CSG primitives, as well as the primitives used by BRL-CAD in the CSG model of the Bradley Fighting Vehicle.

There are many ways to break the boundary into patches for each of these primitive solids. See Figure 2.5 for an example of how a cylinder boundary can be broken down. Within any one patch, many parameterizations are possible.

When choosing a breakdown of the solid boundary into patches, it is generally advantageous to use fewer patches. The fewer patches in each solid, the fewer patch-patch intersections to be computed. However, breaking a boundary into more, smaller, patches allows tighter bounding boxes, and reduces the size of the parametric domain that must be considered for each patch-patch intersection. Overall, experience has shown that fewer patches result in less computation. Practical considerations must still be taken into account. For example, the round sides of a cylinder (i.e. generalized cone) could be represented by a single patch. Such a patch would have an infinite domain and would be adjacent to itself along certain trimming curves, which is undesirable. Using four patches, as described in this chapter, gives a simple, direct parameterization.

When choosing a parameterization for an individual patch, polynomials should be used that have as low a degree as possible, because the degree of the parametric form directly affects the degree of intersection curves in that domain. High-degree intersection curves take longer to compute with than low-degree curves. In the solids below, the parametric form is no more than degree 2 in  $s$  and degree 2 in  $t$  (total degree 4) for all solids. Parameterizations should also be chosen, when possible, such that the trimming curves have a simple (low polynomial degree) form.

The breakdown into patches and the parameterizations presented here are used in ESOLID and work fine in practice. It is certainly possible that better breakdowns into patches or parameterizations exist.

For each case, the breakdown of the boundary into patches, the parametric forms

of the patches, and the trimming curves in each patch domain are described.

The implicit form of the patch surfaces is not given, but it can be found by implicitizing the parametric form (Section 4.4). In many cases, implicitization is not necessary, as the implicit form can be generated directly from the input variables.

The topological connectivity is not directly specified in the solids below, but should be obvious from the breakdown of the boundary into patches. Also omitted from the descriptions below are the specifications of the adjacent surfaces (Section 3.1) for each trimming curve.

The domain of each patch is automatically chosen to be larger than the trimmed region. For all primitives except polyhedra, the trimmed region will fall into the  $[0, 1] \times [0, 1]$  region of the  $s \times t$  parametric domain. Thus, the patch domain will be chosen slightly larger, e.g.  $[-0.1, 1.1] \times [-0.1, 1.1]$ .

## A.1 Polyhedron

Polyhedra are relatively simple to parameterize. Each face is one patch. The description that follows assumes that the polyhedral face is a triangle. The extension to more complex faces is direct and is described below.

Assume that the  $(x, y, z)$  coordinates of each of the three vertices of the face are given. Call these vertices  $A$ ,  $B$ , and  $C$ , in counterclockwise order when seen from the exterior of the solid. (For non-triangular faces, let  $A$ ,  $B$ , and  $C$  be any three adjacent vertices of the face that form a convex angle on the face, given in counterclockwise order.) The vertex information given, then, is:

- The coordinate of point A:  $(X_A, Y_A, Z_A)$
- The coordinate of point B:  $(X_B, Y_B, Z_B)$
- The coordinate of point C:  $(X_C, Y_C, Z_C)$

Assume point  $B$  will be at the origin of the parametric domain,  $A$  at the point  $(0, 1)$ , and  $C$  at the point  $(1, 0)$ . Then, the parametric form of the patch is:

$$\begin{aligned}
 X(s, t) &= X_B + (X_C - X_B)s + (X_A - X_B)t \\
 Y(s, t) &= Y_B + (Y_C - Y_B)s + (Y_A - Y_B)t \\
 Z(s, t) &= Z_B + (Z_C - Z_B)s + (Z_A - Z_B)t \\
 W(s, t) &= 1
 \end{aligned}
 \tag{A.1}$$

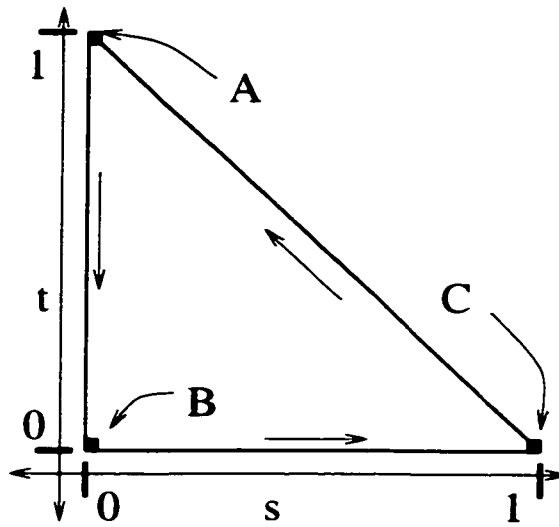


Figure A.1: **The trimming curves for a triangular patch of a polyhedron.** Arrows show the direction of the trimming curves.

The trimming curves are defined as shown in Figure A.1. The horizontal line  $t = 0$  from  $(0, 0)$  to  $(1, 0)$  forms one curve. The next curve is the line  $s + t - 1 = 0$  from  $(1, 0)$  to  $(0, 1)$ . The final trimming curve is the vertical line  $s = 0$  from  $(0, 1)$  to  $(0, 0)$ .

If the face is not triangular, there will be more points that make up the boundary. For each other vertex  $P$ , find the rational parametric coordinates of that vertex by solving the overconstrained linear system from A.1:

$$\begin{aligned}
 X(s, t) &= X_P \\
 Y(s, t) &= Y_P \\
 Z(s, t) &= Z_P
 \end{aligned}
 \tag{A.2}$$

for  $s$  and  $t$ . The trimming curves are modified appropriately. All trimming curves will be lines. Note that  $A$  will still be at  $(0, 1)$ ,  $B$  at  $(0, 0)$ , and  $C$  at  $(1, 0)$ . The other points can have parameter values outside of the  $[0, 1] \times [0, 1]$  region, so the patch domain will have to be modified appropriately.

If more than three points are given, and the points do not all lie in the same plane (as can happen if points that should be in the same plane are rounded off), it is impossible to solve A.2 for the parametric values of additional points. The solid is not a polyhedron, since the face is not planar. If four points not lying in a plane

are given, the face is easily treated as a bilinear patch, rather than as a plane. If more points are given, finding an interpolating parametric surface and determining the appropriate trimming curves (which might not be lines) in that domain becomes much more difficult.

## A.2 Truncated Generalized Cone

A truncated generalized cone is defined by an ellipse forming the base cap and an ellipse forming the top cap. The sides of the generalized cone are formed by connecting the top cap and bottom cap by lines. The lines connect the major axes of the top cap ellipse to that of the bottom cap. The top cap can be a point (yielding the traditional conical shape) or line. In such cases, the cone is no longer truncated. In the description given here, it is assumed that the cone is truncated, so that both the top and bottom caps are ellipses.

Generalized cones can represent right circular cylinders, elliptical cylinders, right circular cones, elliptical cones, and other more exotic objects (with surfaces up to degree four).

Assume that the generalized cone is given by the following information:

- The center of the base ellipse:  $P = (X_p, Y_p, Z_p)$
- The vector from the center of the base ellipse to the center of the top ellipse:  $V_t = [X_t, Y_t, Z_t]$
- The two vectors pointing from the center of the base ellipse to the outer edge along the bottom ellipse axes:

$$V_a = [X_a, Y_a, Z_a], \quad V_b = [X_b, Y_b, Z_b]$$

- The two vectors pointing from the center of the top ellipse to the outer edge along the top ellipse axes:

$$V_c = [X_c, Y_c, Z_c], \quad V_d = [X_d, Y_d, Z_d]$$

This is the information provided by BRL-CAD. The generalized cone is broken into six patches. One patch is used for the top cap, one for the bottom cap, and four for the sides. An example is shown in Figure A.2.



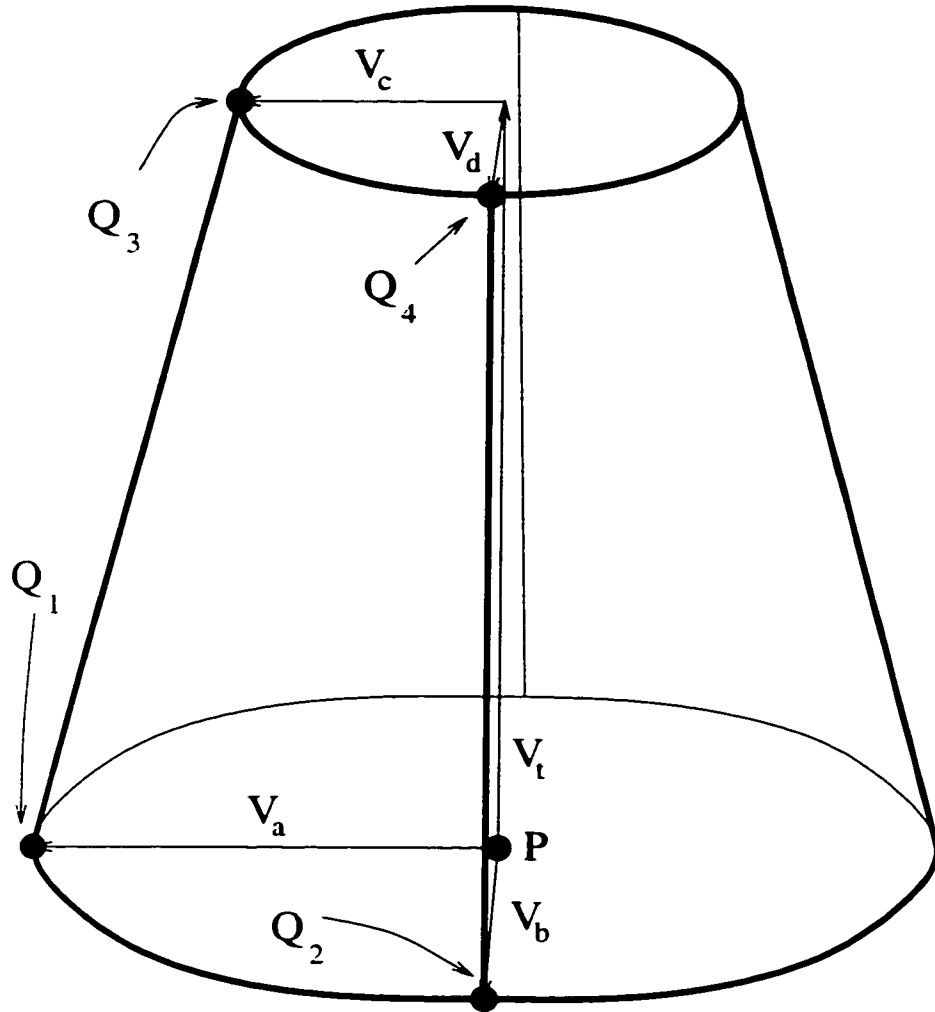


Figure A.2: **The patch breakdown for a truncated generalized cone.** The center point of the base ellipse,  $P$ , is shown. The vector  $V_t$  points from  $P$  to the center of the top ellipse. The vectors  $V_a$  and  $V_b$  point from  $P$  along the major and minor axes of the bottom ellipse, while  $V_c$  and  $V_d$  point along the corresponding axes on the top ellipse. The patch boundaries are outlined. There are four patches forming the sides of the generalized cone. The points  $Q_i$  show the four corner points of one patch.

The four side patches all have a similar parameterization, so only one patch will be described. This parameterization describes the patch at the front left in Figure A.2. The four corners of that patch are labeled by the  $Q_i$ .

$$\begin{aligned}
X(s, t) &= (X_t + X_a - X_c)s^2t + (X_p - X_a)s^2 + 2(X_d - X_b)st + 2X_b s + \\
&\quad (X_t + X_c - X_a)t + (X_p + X_a) \\
Y(s, t) &= (Y_t + Y_a - Y_c)s^2t + (Y_p - Y_a)s^2 + 2(Y_d - Y_b)st + 2Y_b s + \\
&\quad (Y_t + Y_c - Y_a)t + (Y_p + Y_a) \\
Z(s, t) &= (Z_t + Z_a - Z_c)s^2t + (Z_p - Z_a)s^2 + 2(Z_d - Z_b)st + 2Z_b s + \\
&\quad = (Z_t + Z_c - Z_a)t + (Z_p + Z_a) \\
W(s, t) &= s^2 + 1
\end{aligned} \tag{A.3}$$

Note that the point labeled  $Q_1$  in Figure A.2 will be at the point (0,0) in the parametric domain.  $Q_2$  will be at the point (1,0).  $Q_3$  at the point (0,1) and  $Q_4$  at the point (1,1).

The trimming curves for that side patch, then, are as indicated at left in Figure A.3. They consist of the horizontal line  $t = 0$  from (0,0) to (1,0), the vertical line  $s = 1$  from (1,0) to (1,1), the horizontal line  $t = 1$  from (1,1) to (0,1), and the vertical line  $s = 0$  from (0,1) to (0,0).

The top and bottom patches are planar. The parametric form is obtained from the same parameterization as for polyhedra (Equation A.1). For the bottom patch, we have:

$$\begin{aligned}
A &= P + V_a \\
B &= P + V_a + V_b \\
C &= P + V_b
\end{aligned}$$

and for the top patch, we have:

$$\begin{aligned}
A &= P + V_i + V_c \\
B &= P + V_i + V_c + V_d \\
C &= P + V_i + V_d
\end{aligned}$$

The trimming curves for the bottom patch are as indicated at right in Figure A.3. All four trimming curves arise from the same polynomial,

$$\left(x - \frac{1}{2}\right)^2 + \left(y - \frac{1}{2}\right)^2 - \frac{1}{4} = 0 \tag{A.4}$$

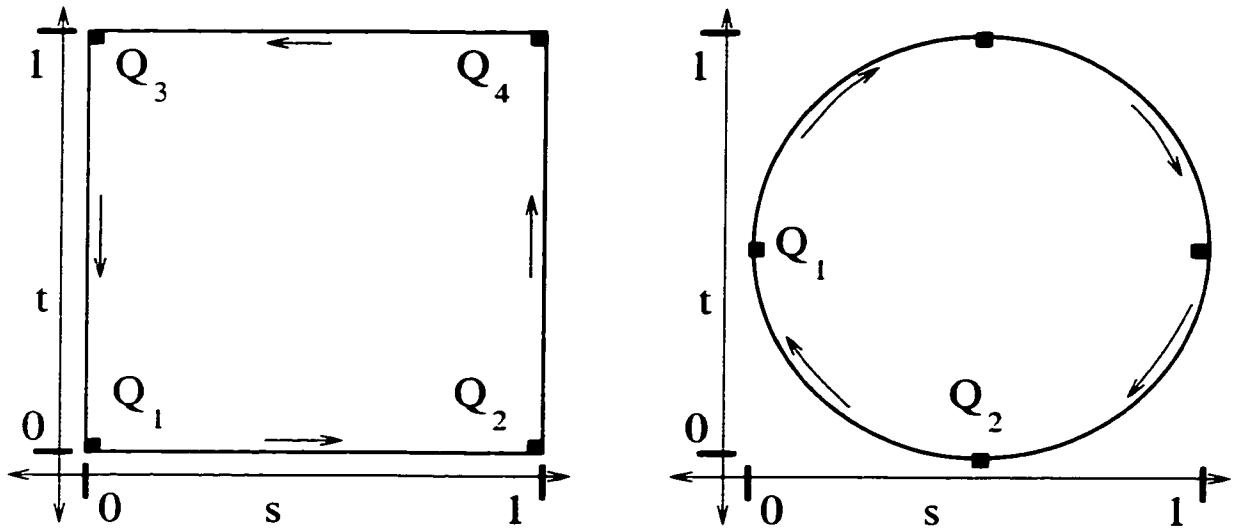


Figure A.3: **The patch domains for a generalized cone.** At left is the patch domain for the front left patch from Figure A.2. At right is the patch domain for the bottom cap from the same generalized cone. The  $Q_i$  show the position of the labeled points from Figure A.2 in this domain. Arrows show the direction of the trimming curves.

The trimming curves proceed from the point  $(\frac{1}{2}, 0)$  to  $(0, \frac{1}{2})$  to  $(\frac{1}{2}, 1)$  to  $(1, \frac{1}{2})$ , and back to  $(\frac{1}{2}, 0)$ . For the top patch, the curves proceed in the opposite order (since the curves travel counterclockwise when viewed from the exterior of the solid).

### A.3 Ellipsoid

Assume that an ellipsoid is described by the following information:

- The center point:  $P = (X_p, Y_p, Z_p)$
- Vectors pointing from the center of the ellipsoid to the ends of the three axes of the ellipsoid:

$$A = [X_a, Y_a, Z_a], \quad B = [X_b, Y_b, Z_b], \quad C = [X_c, Y_c, Z_c]$$

Ellipsoids can be used to represent spheres. An ellipsoid is divided into eight trimmed patches, each being one octant of the ellipsoid. This breakup is illustrated in Figure A.4.

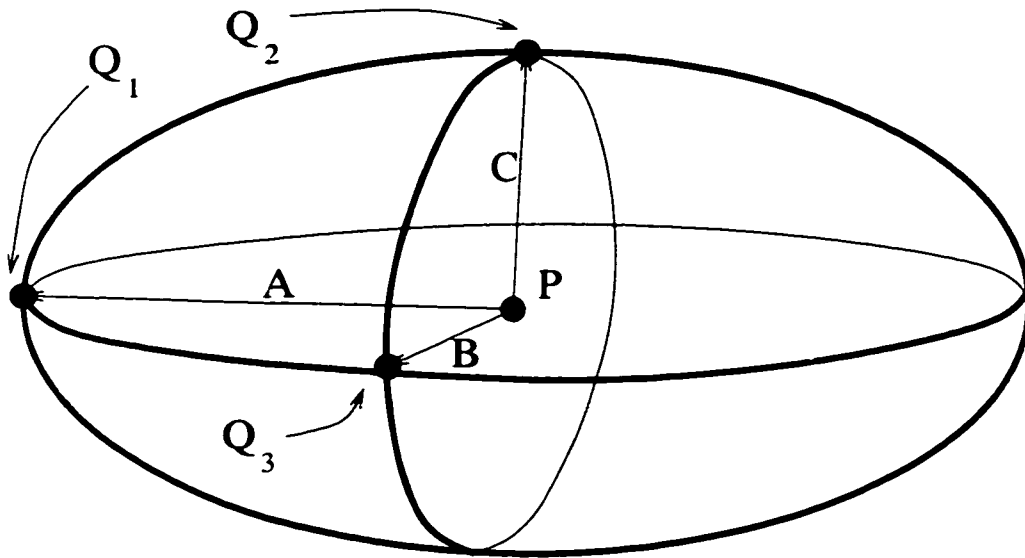


Figure A.4: **The patch breakdown for an ellipsoid.** The center point and three axial vectors are labeled. The  $Q_i$  are three points on one patch.

All eight patches have a similar parameterization. The parameterization of the top, front, left patch (the one containing all the labeled points,  $Q_i$ ) from Figure A.4 is given here:

$$\begin{aligned}
 X(s, t) &= (X_p - X_a)s^2 + (X_p - X_a)t^2 + 2X_b s + 2X_c t + (X_p + X_a) \\
 Y(s, t) &= (Y_p - Y_a)s^2 + (Y_p - Y_a)t^2 + 2Y_b s + 2Y_c t + (Y_p + Y_a) \\
 Z(s, t) &= (Z_p - Z_a)s^2 + (Z_p - Z_a)t^2 + 2Z_b s + 2Z_c t + (Z_p + Z_a) \\
 W(s, t) &= s^2 + t^2 + 1
 \end{aligned} \tag{A.5}$$

For the  $Q_i$  labeled in Figure A.4, notice that  $Q_1$  will be at the point (0, 0),  $Q_2$  will be at the point (1, 0), and  $Q_3$  will be at the point (0, 1). The trimming curves in the patch domain are illustrated in Figure A.5. The curves are the horizontal line  $t = 0$  from (0, 0) to (1, 0), the curve  $s^2 + t^2 = 1$  from (1, 0) to (0, 1), and the vertical line  $s = 0$  from (0, 1) to (0, 0). Four of the patches will have the trimming curves in the opposite orientation.

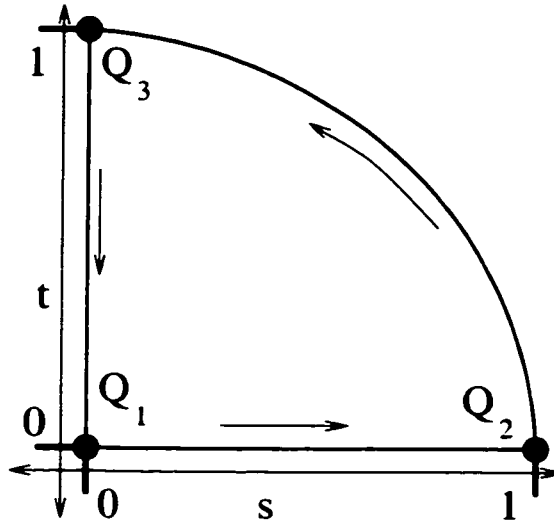


Figure A.5: **The patch domain for an ellipsoid patch.** The trimming curves in the domain of the front, top, left patch from Figure A.4 are shown. The  $Q_i$  are labeled to correspond with those labeled in Figure A.4. Arrows indicate the direction of the trimming curves.

## A.4 Torus

The final primitive considered is the torus. Assume that the information for the torus is given as follows:

- The center of the torus:  $P = (X_p, Y_p, Z_p)$
- The unit vector pointing in the direction normal to the torus—i.e. through the hole in the center:  $N = [X_n, Y_n, Z_n]$
- Two unit vectors perpendicular to the normal vector: they point from the center toward the ring of the torus:

$$V_1 = [X_a, Y_a, Z_a], \quad V_2 = [X_b, Y_b, Z_b]$$

- The radius from the center of the torus to the center of the ring:  $A$
- The radius of the ring:  $B$

The torus is broken into four patches, as shown in Figure A.6.

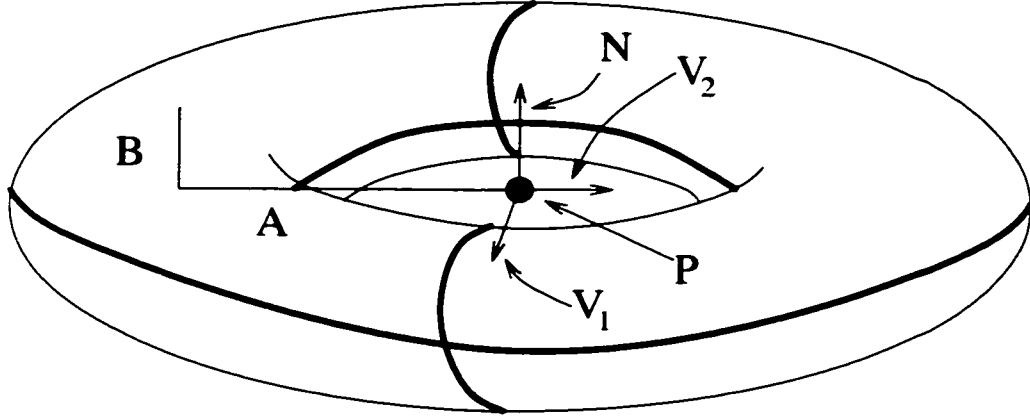


Figure A.6: **The patch breakdown for the torus.** The heavy lines indicate patch boundaries. The center of the torus is at  $P$ .  $N$ ,  $V_1$ , and  $V_2$  are unit vectors.  $N$  points through the center of the torus, while  $V_1$  and  $V_2$  (which are perpendicular to  $N$ ) point toward the  $A$  is the radius from the center of the torus to the center of the ring, while  $B$  is the radius of the ring.

The four patches have a similar parametric form. The parametric form for the patch at the upper right in Figure A.6 is given here.

$$\begin{aligned}
 X(s, t) &= (X_p - AX_b - BX_n)s^2t^2 + (2BX_b)s^2t + (X_p - AX_b + BX_n)s^2 + \\
 &\quad (-2AX_a)st^2 + (4BX_a)st + (-2AX_a)s + \\
 &\quad (X_p + AX_b - BX_n)t^2 + (-2BX_b)t + (X_p + AX_b + BX_n) \\
 Y(s, t) &= (Y_p - AY_b - BY_n)s^2t^2 + (2BY_b)s^2t + (Y_p - AY_b + BY_n)s^2 + \\
 &\quad (-2AY_a)st^2 + (4BY_a)st + (-2AY_a)s + \\
 &\quad (Y_p + AY_b - BY_n)t^2 + (-2BY_b)t + (Y_p + AY_b + BY_n) \\
 Z(s, t) &= (Z_p - AZ_b - BZ_n)s^2t^2 + (2BZ_b)s^2t + (Z_p - AZ_b + BZ_n)s^2 + \\
 &\quad (-2AZ_a)st^2 + (4BZ_a)st + (-2AZ_a)s + \\
 &\quad (Z_p + AZ_b - BZ_n)t^2 + (-2BZ_b)t + (Z_p + AZ_b + BZ_n) \\
 W(s, t) &= s^2t^2 + s^2 + t^2 + 1
 \end{aligned} \tag{A.6}$$

Each patch will have four trimming curves. The curves will be the horizontal line,  $t = -1$  from  $(-1, -1)$  to  $(1, -1)$ , the vertical line  $s = 1$  from  $(1, -1)$  to  $(1, 1)$ , the horizontal line  $t = 1$  from  $(1, 1)$  to  $(-1, 1)$ , and the vertical line  $s = -1$  from

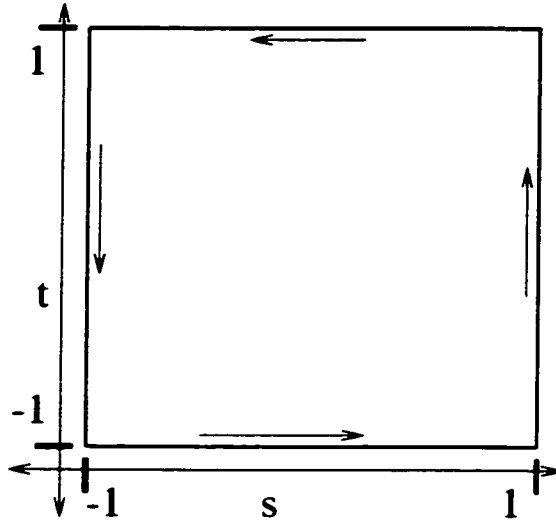


Figure A.7: **The domain of a torus patch.** The trimming curves are shown. Arrows indicate the direction of the trimming curves.

$(-1, 1)$  to  $(-1, -1)$ . This is illustrated in Figure A.7. Two of the patches will have the trimming curves given in the opposite order.

The trimmed patch will be the  $[-1, 1] \times [-1, 1]$  region. A simple linear transformation (substituting  $2s - 1$  for  $s$  and  $2t - 1$  for  $t$ ) will change the domain to the  $[0, 1] \times [0, 1]$  domain. The trimming curves must also be changed. The adjusted trimming curves will be the same as for the cylinder side patches (see the left side of Figure A.3).

# Bibliography

- [1] Shreeram S. Abhyankar and Chanderjit L. Bajaj. Computations with algebraic curves. In P. Gianni, editor, *Proceedings of ISSAC '88*, volume 358 of *Lecture Notes in Computer Science*, pages 274–284. Springer-Verlag, Berlin, 1989.
- [2] S. L. Abrams, W. Cho, C.-Y. Hu, T. Maekawa, N.M. Patrikalakis, E. C. Sherbrooke, and X. Ye. Efficient and reliable methods for rounded interval arithmetic. *Computer Aided Design*, 30(8):657–665, July 1998.
- [3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. *LAPACK User's Guide, Release 1.0*. SIAM, Philadelphia, 1992.
- [4] S. Arnborg and H. Feng. Algebraic decomposition of regular curves. *Journal of Symbolic Computation*, 5:131–140, 1988.
- [5] D. Arnon and S. McCallum. A polynomial time algorithm for the topological type of a real algebraic curve. *Journal of Symbolic Computation*, 5:213–236, 1988.
- [6] D. S. Arnon. Topologically reliable display of algebraic curves. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 219–227. July 1983.
- [7] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: The basic algorithm. *SIAM Journal on Computing*, 13(4):865–877, 1984.
- [8] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition II: The adjacency algorithm for the plane. *SIAM Journal on Computing*, 13(4):878–889, 1984.
- [9] B. G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of AFIPS National Computer Conference*, volume 44, pages 589–596, 1975.
- [10] M. O. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation based on a lazy rational arithmetic: A detailed implementation. *Computer Aided Design*, 26(6):403–416, June 1994.
- [11] I. Biehl, J. Buchmann, and T. Papanikolaou. LiDIA: A library for computational number theory. Technical Report SFB 124-C1, Fachbereich Informatik, Universität des Saarlandes, 1995.



- [12] I. C. Braid. The synthesis of solids bounded by many faces. *Communications of the ACM*, 18(4), April 1975.
- [13] H. Bronnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.
- [14] C. Burnikel, R. Fleischer, K. Melhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proceedings of the 15th ACM Conference on Computational Geometry*, pages 341–350, 1999.
- [15] John F. Canny. *The Complexity of Robot Motion Planning*. The MIT Press, Cambridge, Massachusetts, 1988.
- [16] M. S. Casale and J. E. Bobrow. A set operation algorithm for sculptured solids modeled with trimmed patches. *Computer Aided Geometric Design*, 6:235–247, 1989.
- [17] Alexander I. Chubarev. Robust set operations on polyhedral solids: A fixed precision approach. *International Journal of Computational Geometry and Applications*, 6:187–204, 1999.
- [18] Kenneth L. Clarkson. Safe and effective determinant evaluation. Manuscript, February 1995.
- [19] Joao Luiz Dibl Comba and Jorge Stolfi. Affine arithmetic and its applications to computer graphics. *Anais do VII Sibgraphi*, 1993.
- [20] T. Culver, J. Keyser, and D. Manocha. Accurate computation of the medial axis of a polyhedron. In *Proceedings of the Symposium on Solid Modeling and Applications*, pages 179–190, 1999.
- [21] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra*. Academic Press, London, 1993. 2nd edition.
- [22] Luiz Henrique de Figueiredo. Surface intersection using affine arithmetic. In *Proceedings of Graphics Interface '96*, pages 168–175, 1996.
- [23] Tamal K. Dey, Kokichi Sugihara, and Chandrajit L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.
- [24] Paul H. Dietz, Jr. William H. Mermagen, and Paul R. Stay. An integrated environment for army, navy and air force target description support. Technical Report Memorandum Report BRL-MR-3754, Ballistics Research Laboratory, Aberdeen Proving Ground, MD, 1989.

- [25] A.L. Dixon. The eliminant of three quantics in two independent variables. *Proceedings of London Mathematical Society*, 7:49–69, 1909.
- [26] Phillip C. Dykstra and Michael John Muuss. The BRL-CAD package an overview. Technical report, Advanced Computer Systesms Team, Ballistics Research Laboratory, Aberdeen Proving Ground, MD, 1989.
- [27] Herbert Edelsbrunner and Ernst Peter Mucke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.
- [28] Ioannis Z. Emiris and John F. Canny. A general approach to removing degeneracies. In *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science*, pages 405–413, 1994.
- [29] Ioannis Z. Emiris, John F. Canny, and Raimund Seidel. Efficient perturbations for handling geometric degeneracies. Manuscript, June 1994.
- [30] C. Burnikel et al. Exact computation in LEDA. In *Proceedings of the 11th ACM Conference on Computational Geometry*, pages C18–C19, 1995.
- [31] Shiaofen Fang, Beat Bruderlin, and Xiaohong Zhu. Robustness in solid modeling: A tolerance-based intuitionistic approach. *Computer Aided Design*, 25(9):567–576, September 1993.
- [32] R. T. Farouki, C. A. Neff, and M. A. O’Connor. Automatic parsing of degenerate quadric-surface intersections. *ACM Transactions on Graphics*, 8(3):174–208, July 1993.
- [33] Steven Fortune. Voronoi diagrams and Delaunay triangulations. In D. Z. Du and F. Hwang, editors. *Computing in Euclidean Geometry*, pages 225–265. World Scientific Press, Singapore, 1995.
- [34] Steven Fortune. Robustness issues in geometric algorithms. In *Proceedings 1st ACM Workshop on Applied Computational Geometry*, pages 20–23, 1996.
- [35] Steven Fortune. Polyhedral modelling with multiprecision integer arithmetic. *Computer-Aided Design*, 29(2):123–133, 1997.
- [36] Steven Fortune and Christopher J. van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the 9th ACM Conference on Computational Geometry*, pages 163–171, 1993.
- [37] Steven Fortune and Christopher J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, July 1996.

- [38] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*. 4(2):74–123, April 1985.
- [39] Leonidas J. Guibas and David H. Marimont. Rounding arrangements dynamically. In *Proceedings of the 11th ACM Conference on Computational Geometry*, pages 190–199, 1995.
- [40] Leonidas J. Guibas, David Salesin, and Jorge Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proceedings of the 5th ACM Conference on Computational Geometry*, pages 208–217, 1989.
- [41] James Guilford and Joshua Turner. Representational primitives for geometric tolerancing. *Computer Aided Design*. 25(9):577–586, September 1993.
- [42] Masatake Higashi, Fuyuki Torihara, Nobuhiro Takeuchi, Toshio Sata, Tsuyoshi Saitoh, and Mamoru Hosaka. Face-based data structure and its application to robust geometric modeling. In *Proceedings of the Symposium on Solid Modeling and Applications*, pages 235–246, 1995.
- [43] Christoph M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.
- [44] Christoph M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computar*, 22(3):31–41, March 1989.
- [45] Christoph M. Hoffmann, John E. Hopcroft, and Michael S. Karasick. Robust set operations on polyhedral solids. *IEEE Computer Graphics and Applications*, 9(6):50–59, November 1989.
- [46] Michael E. Hohmeyer. *Robust and Efficient Surface Intersection for Solid Modeling*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California, 1992. CSD-92-681.
- [47] John E. Hopcroft and Peter J. Kahn. A paradigm for robust geometric Algorithms. Report TR 89-1044, Department of Computer Science, Cornell University, Ithaca, NY, 1989.
- [48] Chun-Yi Hu, Nicholas M. Patrikalakis, and Xiuzi Ye. Robust interval solid modelling part 1: Representations. *Computer Aided Design*. 28(10):807–817, October 1996.
- [49] Chun-Yi Hu, Nicholas M. Patrikalakis, and Xiuzi Ye. Robust interval solid modelling part 2: Boundary evaluation. *Computer Aided Design*, 28(10):819–830, October 1996.

- [50] David J. Jackson. Boundary representation modelling with local tolerancing. In *Proceedings of the Symposium on Solid Modeling and Applications*, pages 247–253, 1995.
- [51] J. R. Johnson. Real algebraic number computation using interval arithmetic. In *Proceedings of ISSAC '92*, pages 195–205, 1992.
- [52] J. K. Johnstone. *The sorting of points along an algebraic curve*. Ph.D. thesis. Department of Computer Science, Cornell University, Ithaca, NY, 1987. Technical Report TR 87-841.
- [53] Simon Kahan and Jack Snoeyink. On the bit complexity of minimum link paths: Superquadratic algorithms for problems solvable in linear time. In *Proceedings of 12th Annual ACM Symposium on Computational Geometry*, pages 151–158, 1996.
- [54] D. Kapur, T. Saxena, and L. Yang. Algebraic and geometric reasoning using dixon resultants. In *Proceedings of ISSAC '94*, pages 99–107, 1994.
- [55] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the 15th ACM Conference on Computational Geometry*, pages 351–359, 1999.
- [56] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Transactions on Graphics*, 10:71–91, 1991.
- [57] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. In *Proceedings of the 15th ACM Symposium on Computational Geometry*, pages 360–369, 1999.
- [58] J. Keyser, S. Krishnan, D. Manocha, and T. Culver. Efficient and reliable computation with algebraic numbers for geometric algorithms. Technical Report TR98-012. Department of Computer Science, University of North Carolina, 1998.
- [59] Donald E. Knuth. *The Art of Computer Programming. Volume 2 Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 1998. 3rd edition.
- [60] David J. Kriegman, Erliang Yeh, and Jean Ponce. Convex hulls of algebraic curves. In J. D. Warren, editor, *Proceedings of the International Society for Optical Engineering Volume 1830, Curves and Surfaces in Computer Vision and Graphics III*, pages 118–127. SPIE, Boston, 1992.
- [61] S. Krishnan, D. Manocha, M. Gopi, T. Culver, and J. Keyser. BOOLE: A boundary evaluation system for boolean combinations of sculptured solids. *International Journal of Computational Geometry and Applications*, 2000. To appear.

- [62] Shankar Krishnan. *Efficient and Accurate Boundary Evaluation Algorithms for Boolean Combinations of Sculptured Solids*. PhD thesis, University of North Carolina, 1997.
- [63] Shankar Krishnan, Mark Foskey, Tim Culver, John Keyser, and Dinesh Manocha. PRECISE: Efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation. UNC-CH Technical Report TR 00-008. Department of Computer Science, University of North Carolina, Chapel Hill, NC. 2000.
- [64] J. Lee and G. E. Johnson. Optimal tolerance allotment using a genetic algorithm and truncated Monte Carlo simulation. *Computer Aided Design*, 25(9):601–611, September 1993.
- [65] F.S. Macaulay. On some formulae in elimination. *Proceedings of London Mathematical Society*, 1(33):3–27, May 1902.
- [66] D. Manocha. Solving systems of polynomial equations. *IEEE Computer Graphics and Applications*, pages 46–55, March 1994. Special Issue on Solid Modeling.
- [67] Dinesh Manocha and John Canny. A new approach for surface intersection. *International Journal of Computational Geometry and Applications*, 1(4):491–516. 1991.
- [68] Dinesh Manocha and John Canny. Multipolynomial resultant algorithms. *Journal of Symbolic Computation*, 15:99–122, 1993.
- [69] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland, 1988.
- [70] Glauco Massotti. Floating-point numbers with error estimates. *Computer Aided Design*, 25(9):524–538, September 1993.
- [71] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Report A 04/89, Fachber. Inform., Univ. Saarlandes. Saarbrücken. West Germany, 1989.
- [72] Victor Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. Report CMU-CS-88-168, Department of Computer Science, Cornell University, Pittsburgh, July 1988.
- [73] Victor Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the 5th ACM Conference on Computational Geometry*, pages 197–207, 1989.
- [74] J. Miller and R. Goldman. Combining algebraic rigor with geometric robustness for the detection and calculation of conic sections in the intersection of two quadric surfaces. In *Proceedings of the Symposium on Solid Modeling and Applications*, pages 221–233, 1991.

- [75] P. S. Milne. On the solutions of a set of polynomial equations. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 89–102, 1992.
- [76] Bhubaneswar Mishra. *Algorithmic Algebra*. Springer-Verlag, New York, 1993.
- [77] Ramon E. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, 1979.
- [78] P. Pedersen. Multivariate Sturm theory. In *Proceedings of Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 318–332. Springer-Verlag, 1991.
- [79] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry An Introduction*. Springer-Verlag, New York, 1985.
- [80] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of 10th Symposium on Computer Arithmetic*, pages 132–143, Los Alamitos, California, 1991. IEEE Computer Society Press.
- [81] Aristides A. G. Requicha and Jarek R. Rossignac. Solid modeling and beyond. *IEEE Computer Graphics and Applications*, 12(5):31–44, September 1992. Special Issue on CAGD.
- [82] Aristides A. G. Requicha and Herbert B. Voelcker. Solid modeling: A historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24, March 1982.
- [83] Aristides A. G. Requicha and Herbert B. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1):30–44, January 1985.
- [84] Jaroslav R. Rossignac and Herbert B. Voelcker. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Transactions on Graphics*, 8(1):51–87, January 1989.
- [85] T. Sakkalis. On the zeros of a polynomial vector field. IBM Research Report IBM RC-13303, IBM, 1987.
- [86] T. Sakkalis. The topological configuration of a real algebraic curve. *Bulletin of the Australian Mathematical Society*, 43:37–50, 1991.
- [87] G. Salmon. *Lessons Introductory to the Modern Higher Algebra*. G.E. Stechert & Co., New York, 1885.
- [88] R. F. Sarraga. Algebraic methods for intersection. *Computer Vision, Graphics and Image Processing*, 22:222–238, 1983.

- [89] Stefan Schirra. Precision and robustness. *Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM. Udine, Italy*, September 1996.
- [90] T.W. Sederberg, D.C. Anderson, and R.N. Goldman. Implicit representation of parametric curves and surfaces. *Computer Vision, Graphics and Image Processing*, 28:72–84, 1984.
- [91] Mark Segal. Using tolerances to guarantee valid polyhedral modeling results. In *Proceedings of SIGGRAPH 90*, pages 105–114, 1990.
- [92] Raimund Seidel. The nature and meaning of perturbations in geometric computing. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *STACS 94*, volume 775 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, Berlin, 1994.
- [93] Ching-Kuang Shene and John K. Johnstone. On the planar intersection of natural quadrics. In *Proceedings of the Symposium on Solid Modeling and Applications*, pages 233–242, 1991.
- [94] E. C. Sherbrooke and N. M. Patrikalakis. Computation of the solutions of nonlinear polynomial systems. *Computer Aided Geometric Design*, 10(5):379–405, 1993.
- [95] Jonathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the 12th ACM Conference on Computational Geometry*, pages 141–150, 1996.
- [96] Ken Shoemake. Rational approximation. In Alan W. Paeth, editor, *Graphics Gems V*, pages 25–32. Academic Press, Boston, 1995.
- [97] A. James Stewart. Local robustness and its application to polyhedral intersection. *International Journal of Computational Geometry and Applications*, 4(1):87–118, 1994.
- [98] Kokichi Sugihara. A robust and consistent algorithm for intersecting convex polyhedra. In M. Dæhlen and L. Kjelldahl, editors, *Proceedings of EUROGRAPHICS '94*, volume 13, pages C–45–C–54. Blackwell Association, 1994.
- [99] Kokichi Sugihara and Masao Iri. A solid modeling system free from topological inconsistency. *Journal of Information Processing*, 12(4):380–393, 1989.
- [100] Irina Voiculescu. Personal Communication, 1999.
- [101] Kevin J. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.

- [102] Chee Yap and Thomas Dube. The exact computation paradigm. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 1995.
- [103] Chee-Keng Yap. Symbolic treatment of geometric degeneracies. *Journal of symbolic Computation*, 10:349–370, 1990.
- [104] Chee-Keng Yap. Towards exact geometric computation. *Computational Geometry Theory and Applications*, 7:3–23, 1997.
- [105] J. Yu. *Exact arithmetic solid modeling*. PhD thesis, Purdue University, 1992.
- [106] Richard Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(1):375–403, 1990.