ACM Symposium on Solid Modeling and Applications (2004)
G. Elber, N. Patrikalakis, P. Brunet (Editors)

Handling Degeneracies in Exact Boundary Evaluation

Koji Ouchi and John Keyser

Department of Computer Science, 3112 Texas A&M University
College Station, TX, 77843-3112

Abstract

We present a method for dealing with degenerate situations in an exact boundary evaluation system. We describe the possible degeneracies that can arise and how to detect them. We then present a numeric perturbation method that is simpler to implement within a complex system than symbolic perturbation methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling G.4 [MATHEMATICAL SOFTWARE]: Reliability and robustness

1. Introduction

1.1. Motivation

Degenerate configurations of solids are an unavoidable part of solid modelers. Solids are often specified, either accidentally or intentionally, in degenerate configurations. Degenerate configurations are a fact of nature, for example, a sphere sitting on a table intersects the table tangentially at a point.

Unfortunately, many algorithms, at least in their initial formation, make an assumption that input will be given in general position. Roughly, this assumption states that no minor perturbation of the data will cause the solids to intersect in a manner that is topologically different. In order to deal with degenerate configurations of solids, a system must be able to accurately detect and handle the degenerate cases. Often this additional code is overwhelming, taking far more work than the basic algorithm alone would.

For solid modelers, degeneracies remain a problem. One solution is that designers using a solid modeling system can be trained to avoid placing objects in such degenerate configurations. However, such an approach is likely to fail in many instances, and one would like a program to still produce usable output in such a situation. Another option is to systematically add degeneracy detection and handling code to a modeler. While this can be very useful (and can allow a wider variety of objects to be represented), it also creates more complexity in algorithms and representations, e.g. by needing to handle and work with non-manifold representations. A final approach (and the one we follow here) is to perturb the data so that all computation is done on objects in general position. It is important in these cases, however, that the perturbation results in objects that are physically realizable and maintain the designer's intent.

1.2. Exact Computation

The method presented here assumes an exact computation [DY95] framework for the solid modeling system. Exact computation eliminates the problems that can arise with numerical error. Handling numerical error is a necessary precondition for any attempt to deal with degeneracies. This is because inexact computation and numerical error can both create degeneracies (e.g. two numbers could be rounded to the same number) and remove degeneracies (e.g. two numbers that should come out to the same value actually come out differently). Note that exact computation (where enough precision is used to guarantee that every decision made is correct) does not necessarily mean exact arithmetic (where every computation is done to full precision). Using techniques such as lazy evaluation and filtering can also allow exact computation, at much less computational cost.

Our work is based on the exact solid modeler, ESOLID [KCF*04]. To our knowledge, this is currently the only system that supports exact boundary evaluation for solids with curved surfaces. ESOLID uses exact representations for all solids, and performs exact computations to evaluate the boundary of the solids. Input is taken from a CSG-style tree, and output is a boundary representation of the final solids. Primitive solids include polyhedra, ellipsoids, generalized cones, and tori. ESOLID has been successful at exact boundary evaluation for solids (with low algebraic degree surfaces) in general position. However, the algorithms ESOLID uses are specifically designed assuming general position. ESOLID therefore fails for almost all degenerate situations.

1.3. Goal

Overall, the problem we are addressing can be stated as follows: *Given a CSG tree and a physical tolerance within which we are*

free to adjust the surfaces of the input solids, compute a boundary representation robustly, maintaining the designer's intent.

We emphasize that we are using exact computation and handling degeneracies to achieve greater robustness and consistency. The goal is not necessarily to have an exact solution to the input problem—in the real world it is very rare that the output solids would need to be found with absolute precision. However, without accounting for numerical error and degeneracies, programs are subject to crashes or inconsistent output, neither of which is a desirable condition. We would like to have a program that can reliably produce a set of valid output, for a wide variety of input.

1.4. Main Results

We present a method for dealing with degeneracies in an exact computation-based solid modeler. First, we classify the types of degenerate configurations possible between objects. Then, we describe a numerical perturbation scheme that can be used to handle degenerate situations. This approach requires very little modification to existing exact solid modeling, yet eliminates the problems associated with degeneracies.

2. Previous Work

Dealing with robustness issues has been an active area of work for quite some time, now. Much of the need for robustness was highlighted by the work of Hoffmann et al. [HHK89].

Exact computation as a method for dealing with numerical error has been addressed within the solid modeling community. Much of the earliest work focused on polyhedral solids, with only limited work on curved solids. Among the work on exact computation in solid modeling is that of Sugihara and Iri [SI89], Yu [Yu91], Benouamer et al. [BMP94], Sugihara [Sug94], and Fortune [For97]. The work presented here builds off of previous work on exact solid modeling [KCMK00, KCF*04]. More general work supporting exact computation includes the development of the LEDA [MN99], CORE [KLY99] and CGAL [FGK*00] libraries.

For dealing with degeneracies, special-case code has been predominant. Examples can be seen in both solid modeling textbooks [Hof89] and research papers [Yu91]. For curved solids, degeneracies become more complicated. A great deal of effort has focused just on handling the intersections of quadric surfaces, such as in Farouki et al. [FNO93] and Geismann et al. [GHS01].

Perturbation methods have arisen as a more general way of dealing with degenerate situations. These approaches use a symbolic perturbation to move the surfaces by an infinitesimal amount. The major drawbacks are that the schemes must either use symbolic computation, which is extremely expensive, or define every predicate directly from the input, which may be impractical. The earliest perturbation scheme was probably that of Edelsbrunner and Mucke [EM90]. Emiris and Canny varied the perturbation used to a simpler one, and applied it to a wider variety of cases [EC91, ECS97]. Yap provided an even more generalized perturbation approach [Yap90]. Seidel provides a summary of these techniques, along with a critique of the general perturbation scheme [Sei94]. Fortune actually describes the use of symbolic perturbation for linear solids [For97].

3. Classifying Degeneracies

Degeneracies can be classified into three categories. These are:

1) *Input degeneracies*: degenerate configurations of input data, where a minor change in the solid changes the nature of the output. For example, two objects might meet tangentially at a point.

2) *Arbitrary degeneracies*: degenerate situations arising from arbitrary algorithmic decisions. For example, an algorithm might choose to shoot a ray in an arbitrary direction.

3) *Intentional degeneracies*: degeneracies constructed as a part of the algorithm, and that must be maintained. For example, an algorithm might create a point that is the midpoint between two others. Later computations might assume that the three points are collinear (a degenerate configuration of the three points).

Intentional degeneracies are correctly dealt with if exact computation is used; we will not discuss them further except to note that the intentional degeneracies at one stage can appear as input degeneracies at the next. Arbitrary degeneracies in general are difficult to predict and avoid, and are not dealt with here directly. Randomization can usually be used to avoid these situations, but possibly at a cost in efficiency. Falling between input and arbitrary degeneracies are "borderline" cases where the algorithm uses a technique that exposes a degenerate case, but for which another algorithm might not encounter difficulty. For example, two surfaces might meet in a degenerate configuration, but outside of the trimmed region of the patches they belong to—some algorithms might encounter this degeneracy, others might not.

3.1. Enumerating Input Degeneracies

Possible input degeneracies are enumerated by considering how surfaces, curves and points interact.

A solid is composed of objects of various orders: points (order 0), curves (order 1) and surfaces (order 2). Generically, two surfaces meet at curves and three surfaces meet at points. Four or more surfaces cannot meet generically.

When objects are in general position, only two different types of mutual interactions are possible: 1) two surfaces meet transversely along a set of curves, and 2) a surface and a curve meet transversely at a set of points. Degeneracies occur in two ways: 1) when two objects interact that should not (e.g. a surface intersects a point), and 2) when an interaction is between a valid pair of objects but is tangential instead of transverse. The types of possible degeneracies are summarized in Table 1.

4. Detecting Degeneracies

Degeneracies are detected during boundary evaluation by checking for occurrences of the irregular interactions classified in Section 3. While the description below is broad, it may be helpful to be familiar with the exact boundary evaluation approach used in ESOLID [KCF*04] in order to understand how these cases arise specifically. ESOLID finds and represents points in the 2D domain, using the MAPC library [KCMK00]. MAPC represents points (with real algebraic coordinates) using a set of bounding intervals, guaranteed to contain a unique root of the defining polynomials.

Objects are represented in terms of polynomials (with rational

	Surface		Curve		Point	
Surface	**2**	Surfaces overlap				
	1	Surfaces tangent along a curve	**1**	A curve lies on a surface		
	0	Surfaces tangent at a point	**0**	A curve is tangent at a point	**0**	A Point lies on a surface
Curve			**1**	Curves overlap		
			0	Curves intersect tangentially	**0**	A Point lies on a curve
Point					**0**	Points coincide.

Table 1: *Types of degeneracies. The entries show degeneracies between surfaces, curves and points. The order of each degenerate intersection involved is in bold: points (0), curves (1), surfaces (2).*

coefficients). Thus, degeneracy detection often can be thought of as discovering when systems of polynomials interact in non-generic ways.

1) *Surfaces Overlap*: If surfaces overlap then their implicit representation must have a non-trivial common factor. Thus, a degeneracy is detected when substitution of the parametric form of one surface into the implicit form of the other surface causes the implicit form to vanish.

2) *Surfaces Tangent along a Curve*: There are two possible realizations of this degeneracy. If the tangency is at a point then the intersection curve has a cusp or self-intersection at the point of tangency. Such singularities appear when analyzing the topology of the intersection curve. If the tangency is along the entire intersection curve of the two surfaces (i.e. they never cross) then the degeneracy is not obvious from within the patch domain. For this one instance only, the degeneracy cannot be detected by local polynomial evaluations. Fortunately, the overall boundary evaluation algorithm can be easily modified to handle such cases smoothly at a higher level, without causing any potential problems in the polynomial computations. The details of this are omitted here.

3) *Surfaces Tangent at a Point*: If surfaces are tangent at a point then the intersection curve of two surfaces shrinks to a single point, which is a singularity.

4) *A Curve Tangent to a Surface*: Within the patch domain, this appears as an intersection curve intersecting a trimming curve tangentially.

5) *Three (or More) Surfaces Meet at a Curve*: This case include two subcases *A Curve Lies on a Surface* and *Curves Overlap*. In both cases, in at least one patch domain, the degeneracy will appear as the intersection curve overlapping the trimming curve. In other words, the intersection curve and trimming curve intersect in a positive dimensional component.

6) *Four Surfaces Meet at a Point*: This includes four subcases: *A Point Lies on a Surface*, *Curves Intersect Tangentially*, *A Point Lies on a Curve* and *Points Coincide*. In any of these cases, the intersection curves intersect the trimming curves at an endpoint or previously found intersection point.

To summarize, barring the trivial cases *Surfaces Overlap* and the second realization of *Surfaces Tangent along a Curve* that are handled through other mechanisms, degeneracies are detected when one of the following happens in 2D:

A) Three or more curves intersect at a single point. This includes the cases of singularities (where the curve and its two derivative curves intersect at a common point), B) two curves intersect tangentially and C) two curves share a positive-dimensional component.

To detect these cases, we need an exact polynomial system solver that works under any condition. We have developed and implemented a method based on the Rational Univariate Reduction which allows this. The details are omitted here.

5. Numerical Perturbation

A perturbation method is a general approach for dealing with degeneracies. The idea is to perturb the input data slightly such that one is guaranteed to have no degeneracies.

There are two types of perturbation methods: symbolic and numeric. It is important to note that exact computation is a *necessity* for both symbolic and numeric perturbation, since otherwise the perturbation scheme will not guarantee even a consistent result. Symbolic perturbations modify the input data by symbolic amounts and then take their limit to zero, whereas numerical perturbations change the actual input data. Thus, the resulting solid is different than that of the original input.

The validity of numerical perturbation is supported by the idea of existence of a global tolerance. That is, there is an assumption that the input data is correct within some amount, ε, which may be expressed in relative or absolute terms. Any perturbation of the input data within ε is allowed, as that data could potentially be a valid input. The input data can be perturbed how ever much is necessary to allow the program to run as long as the perturbation is less than ε.

There are two potential problems with numerical perturbations.

First, there is a chance that a specific numerical perturbation will not eliminate the degeneracies, or even worse, will create another. If the perturbation is chosen randomly, it is extremely unlikely that such an event will happen. Assuming we can detect degeneracies, we can always find when this has occurred and use another perturbation instead.

A second problem is that, by perturbation, the bit length of data could be drastically increased. For example, the number 1 might be perturbed to the number 1.000001. Longer bit lengths can lead to much longer running times. From first-hand experience with real-world data, however, tolerance values are often of the order a number is specified to (e.g. 3.141595 ± 0.000005), so the perturbation often does not increase bit length significantly.

We also note that even if bit lengths are increased, the use of floating-point filters can eliminate many of the numerical efficiency problems in all computations except near the original degeneracy. That is, aggressively filtering means that little additional time needs to be spent, except to actually resolve the area around the degeneracy (where the filters are likely to fail).

5.1. Expanding and Contracting Primitives

Expansion and contraction of primitives offers the opportunity to capture design intent. The expanding and contracting follows the principles proposed first by Sugihara and Iri [SI89], and later adapted by Fortune [For97], where the surfaces of input solids are symbolically perturbed inward or outward in order to remove degeneracies. The topological complexity issues are avoided, however, by only perturbing input primitives, and allowing computation to proceed as normal (this follows Sugihara and Iri). Our approach differs from that of Sugihara and Iri in that it can be applied to non-polyhedral solids, and in that we propagate our information through the tree only to resolve specific degeneracies, and in such a way as to better capture the designer's intent.

Solids are expanded or contracted depending on the operation applied to those solids. For a union, both solids should be expanded. For an intersection, both solids should be contracted. For a difference operation $A - B$, solid A should be contracted and solid B expanded. This approach is far more likely to capture a designer's intent, as shown in Figure 1.

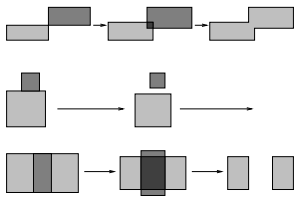


Figure 1: *Examples of expansion and contraction of primitives. The first column shows the input, the second the perturbed solids, and the last the final solids. From top to bottom are shown a union, an intersection, and a difference.*

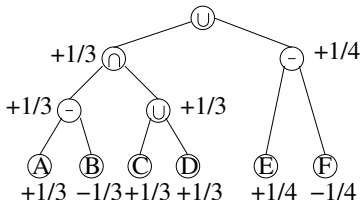


Figure 2: *The perturbation information at the top node is propagated to the leaf nodes.*

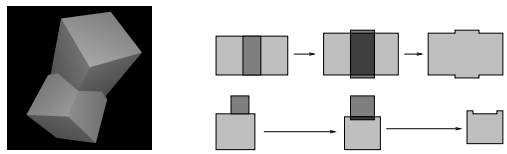
It is important to note that a small amount of perturbation applied to input surfaces may result in a larger perturbation of the eventual solid. When perturbing surfaces, slight changes in a surface can cause large changes in edges and points formed by intersections with that surfaces, particularly when the intersections are nearly tangential.

By perturbing only the surfaces of the input primitives, we avoid the problems of perturbing curved surfaces in general, and the complex topological issues that can arise in more complicated solids. All standard CSG primitives (boxes, generalized cones, ellipsoids, and tori) can be perturbed "outward" or "inward" easily and cleanly, usually by just changing a few parameters.

We determine that a perturbation needs to be applied only when we encounter a degeneracy. Since this may occur at a high level in a CSG-tree, we need to propagate this perturbation information down to the leaves (i.e. the input primitives). Assume without loss of generality that we wish to propagate an expansion down the tree. Each union or intersection node would also propagate an expansion downward. That is, if $A \cup B$ or $A \cap B$ need to be expanded, both A and B would be expanded. With a difference operation $A - B$, however, A would be expanded, and B would be contracted. See the examples in Figure 2.

5.2. Shortcomings of Numerical Perturbation

There are a few shortcomings to the numerical perturbation approach, which we list here.



a b

Figure 3: *a. Examples where perturbing inward and outward does not remove the degeneracy. b. Examples where numerical expansion and contraction lead to small undesirable faces.*

First, there are a limited number of cases for which expansion or contraction will not work. An example of this is shown in Figure 3a. It is likely that a slight adjustment in the perturbation, including translation, would eliminate this deficiency.

Second, it is possible with perturbation schemes to create small, unintended features in objects. While these will indeed be small, and should not affect the overall topology of the solid, they can be annoying to deal with in subsequent computation. An example is shown in Figure 3b.

Regardless of these cases, first-hand experience with real-world degenerate data has shown that the cases similar to those presented in Figure 1 are far more likely to occur than those in Figure 3.

Finally, it should be noted that if there are multiple degeneracies in a tree, it is possible that different degeneracies might require a primitive to be expanded in different directions. In this case, by allowing perturbations of different orders of magnitude for the different operations, both operations can be satisfied. Since one degeneracy must precede the other in the tree, the perturbation from the higher level affects both input solids at the lower-level degeneracy. As long as the solids are perturbed by an amount sufficient to resolve their own degeneracy, but not so much as to change the perturbation direction required by the higher degeneracy, the numerical perturbation will work. This is likely to lead to much higher bit complexity, however.

a b c

Figure 4: *Real-world examples. a. Cargo hatch, b. Commander hatch, c. Engine*

6. Implementation

6.1. A System Overview

We have implemented a solid modeler with numerical perturbation. Our modeler is build on top of ESOLID [KCF*04], the exact solid modeler. It takes the set of CSG primitive solids as an input (as ESOLID), perturbs them, and performs binary operations. CSG primitives are perturbed by scaling inward or outward by a specified amount. The centroid of any type of primitive solid is fixed.

6.2. Real-World Examples

We show the result from examples taken from a model of a Bradley Fighting Vehicle that is developed by the Army Research Lab with their BRL-CAD solid modeler [DM89]. ESOLID is used to convert parts from CSG format to an exact B-rep format. Whenever a degeneracy is detected, ESOLID either aborts or loops. If ESOLID aborts then numerical perturbation is applied. Perturbation is propagated through the entire CSG tree and eventually all the input primitive solids are perturbed. Then, the computation restarts.

The experiments are performed on three parts shown in Figure 4 using a 3.0 GHz Intel Pentium CPU with 6 GB of memory.

Tables 2 to 4 show experiments on the examples in Figure 4. Rows 3 and 4 give the number of times the basic bivariate and univariate root-finding routines were invoked while computing. MAPC performs root isolation and sign evaluation (of a polynomial at a given value) by using floating point filters. When the filter fails, exact methods are invoked. The number of such root isolation and sign evaluations is shown, along with the percentage of the time that the floating point filter fails (and thus exact methods must be used). Column 1 shows the result when ESOLID (i.e. without perturbation) is used for operations on solids not in degenerate configuration, while column 2 shows the performance of ESOLID on a perturbed version (by $\frac{1}{1024}$) of those same solids. Column 3 gives the results for the entire part, including the perturbed solids that have removed the degeneracies.

The part *cargo hatch* is obtained by joining 11 solids. ESOLID fails to model 4 of them because of degeneracies. The part *commander hatch* is obtained by joining 5 solids. ESOLID fails to model 1 of them because of a degeneracy. In both examples, all the degeneracies can be eliminated by numerical perturbation. The part *engine* is obtained by joining 14 solids. One of them has a degeneracy that makes ESOLID loop. Thus, we ignore this part. Among the other 13 solids, ESOLID fails on 3 of them because of the degeneracies.

	w/o pert.	w/ pert.	all solids
time (msec)	36570	50487	141474
# CSG boolean op's	17	17	34
# bivariate root-finding	25137	25764	50902
# univariate root-finding	23809	26344	56800
# root isolation	32616	36879	82024
% of exact computation	6.30	3.88	7.33
# sign evaluation	486139	573022	1288838
% of exact computation	6.37	7.37	11.33

Table 2: *Experiments on cargo hatch*

	w/o pert.	w/ pert.	all solids
time (msec)	95658	701096	722385
# CSG boolean op's	6	6	12
# bivariate root-finding	5331	4103	6636
# univariate root-finding	12354	4772	8635
# root isolation	19889	8761	13032
% of exact computation	18.87	17.62	11.85
# sign evaluation	359768	136517	208196
% of exact computation	3.66	7.85	6.15

Table 3: *Experiments on commander hatch*

	w/o pert.	w/ pert.	all solids
time (msec)	224803	279560	1233390
# CSG boolean op's	15	15	33
# bivariate root-finding	18252	17827	36456
# univariate root-finding	30222	29154	51079
# root isolation	49370	48878	86664
% of exact computation	14.46	16.42	20.38
# sign evaluation	826664	821022	1501026
% of exact computation	3.59	4.49	7.95

Table 4: *Experiments on engine*

In summary, on these real-world cases, we note first that numerical perturbation allows us to compute the result while maintaining designer's intent. The perturbed versions do, indeed, run slower than the unperturbed versions and require a higher percentage of sign evaluations of polynomials using exact arithmetic, rather than floating-point filters (due to increases in bit-length). For two examples these are only modest increases. For one example, however, the time requirement was more significant. We believe that with more aggressive floating-point filters, we can reduce this time increase (for the non-degenerate cases) to within a very small factor, especially for the one extreme example.

7. Conclusion

We have presented a method for handling degeneracies in an exact boundary evaluation scheme. We have described an enumeration of degeneracies, along with a method for detecting each of these degenerate situations. We have also proposed a numerical perturbation scheme that can be used to eliminate the computational problems associated with a degeneracy, in a way that is likely to be consistent with a designer's intent. We have presented the results of our implementation on a number of example cases, to show that it works. In the implementation, we use an initial version of our exact polynomial system solver.

7.1. Future Work

Among our directions for future work are the following:

Currently, we have not integrated the locate degeneracy/perturb approach into an automated loop. This integration would be necessary for a complete system for handling degeneracies.

Again, as mentioned earlier, there are a few cases where numerical perturbation does not solve the problem. We need to address these cases in a different way. Data cleanup must be done. Finally, we have not addressed the situation where multiple degeneracies cause conflicting information in perturbation directions. Determining a good way to handle such cases will be important in dealing with very large scale examples.

Acknowledgements

This work was funded in part by NSF/DARPA CARGO award DMS-0138446 and NSF ITR Award CCR-0220047. The Bradley Fighting Vehicle was provided courtesy of the Army Research Lab.

References

[BMP94] BENOUAMER M. O., MICHELUCCI D., PEROCHE B.: Error-free boundary evaluation based on a lazy rational arithmetic: A detailed implementation. *Computer Aided Design 26*, 6 (1994), 403 – 416.

[DM89] DYKSTRA P. C., MUUSS M. J.: *The BRL-CAD Package An Overview*. Tech. rep., Advanced Computer Systesms Team, Ballistics Research Laboratory, Aberdeen Proving Ground, MD, 1989.

[DY95] DUBÉ T., YAP C.: The exact computation paradigm. In *Computing in Euclidean Geometry*, Du D., Hwang F., (Eds.), 2nd ed., Lecture Notes on Computing. World Scientific, 1995, pp. 452 – 492.

[EC91] EMIRIS I. Z., CANNY J. F.: A general approach to removing degeneracies. In *Proc. 32nd FOCS* (1991), IEEE, pp. 405 – 413.

[ECS97] EMIRIS I. Z., CANNY J. F., SEIDEL R.: Efficient perturbations for handling geometric degeneracies. *Algorithmica 19*, 1/2 (1997), 219 – 242.

[EM90] EDELSBRUNNER H., MÜCKE E.: Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics 9*, 1 (1990), 66 – 104.

[FGK*00] FABRI A., GIEZEMAN G.-J., KETTNER L., SCHIRRA S., SCHÖNHERR S.: On the design of cgal a computational geometry algorithms library. *Software – Practice & Experience 30*, 11 (2000), 1167 – 1202.

[FNO93] FAROUKI R. T., NEFF C. A., O'CONNOR M. A.: Automatic parsing of degenerate quadric-surface intersections. *ACM Transactions on Graphics 8*, 3 (1993), 174 – 208.

[For97] FORTUNE S.: Polyhedral modeling with multiprecision integer arithmetic. *Computer-Aided Design 29*, 2 (1997), 123 – 133.

[GHS01] GEISMANN N., HEMMER M., SCHÖMER E.: Computing a 3-dimensional cell in an arrangement of quadrics: Exactly and actually! In *Proc. 17th SoCG* (2001), ACM, pp. 264 – 273.

[HHK89] HOFFMAN C. M., HOPCROFT J. E., KARASICK M. S.: Robust set operations on polyhedral solids. *IEEE Computer Graphics and Applications 9*, 6 (1989), 50 – 59.

[Hof89] HOFFMAN C. M.: The problems of accuracy and robustness in geometric computation. *IEEE Computer 22*, 3 (1989), 31 – 41.

[KCF*04] KEYSER J., CULVER T., FOSKEY M., KRISHNAN S., MANOCHA D.: ESOLID - a system for exact boundary evaluation. *Computer-Aided Design 36*, 2 (2004), 175 – 193.

[KCMK00] KEYSER J., CULVER T., MANOCHA D., KRISHNAN S.: Efficient and exact manipulation of algebraic points and curves. *Computer-Aided Design 32*, 11 (2000), 649 – 662.

[KLY99] KARAMCHETI V., LI C., YAP C.: A Core library for robust numerical and geometric computation. In *Proc. 15th SoCG* (1999), ACM, pp. 351 – 359.

[MN99] MEHLHORN S., NÄHER M.: *LEDA - A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[Sei94] SEIDEL R.: The nature and meaning of perturbations in geometric computing. In *Proc. 11th STACS* (1994), LNCS 775, Springer, pp. 3 – 17.

[SI89] SUGIHARA K., IRI M.: A solid modelling system free from topological inconsistency. *Journal of Information Processing 12*, 4 (1989), 380 – 393.

[Sug94] SUGIHARA K.: A robust and consistent algorithm for intersecting convex polyhedra. In *Proc. of EUROGRAPHICS '94*, Dæhlen M., Kjelldahl L., (Eds.), Computer Graphics Forum, Vol. 13, No. 3. Blackwell Association, 1994, pp. C–45 – C–54.

[Yap90] YAP C.: Symbolic treatment of geometric degeneracies. *Journal of Symbolic Computation 10*, 3/4 (1990), 349 – 370.

[Yu91] YU J.: *Exact Arithmetic Solid Modeling*. Ph.D. thesis, Department of Computer Science, Purdue University, West Lafayette, IN, 1991.