Technical Report

UCAM-CL-TR-766
ISSN 1476-2986

Number 766

Towards robust inexact geometric computation

Julian M. Smith

December 2009

This technical report is based on a dissertation submitted July 2009 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. Edmund's College.

Some figures in this document are best viewed in colour. If you received a black-and-white copy, please consult the online version if necessary.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Towards robust inexact geometric computation

Julian M. Smith

Summary

Geometric algorithms implemented using rounded arithmetic are prone to robustness problems. Geometric algorithms are often a mix of arithmetic and combinatorial computations, arising from the need to create geometric data structures that are themselves a complex mix of numerical and combinatorial data. Decisions that influence the topology of a geometric structure are made on the basis of certain arithmetic calculations, but the inexactness of these calculations may lead to inconsistent decisions, causing the algorithm to produce a topologically invalid result or to fail catastrophically. The research reported here investigates ways to produce robust algorithms with inexact computation.

I present two algorithms for operations on piecewise linear (polygonal/polyhedral) shapes. Both algorithms are topologically robust, meaning that they are guaranteed to generate a topologically valid result from topologically valid input, irrespective of numerical errors in the computations. The first algorithm performs the Boolean operation in 3D, and also in 2D. The main part of this algorithm is a series of interdependent operations. The relationship between these operations ensures a consistency in these operations, which, I prove, guarantees the generation of a shape representation with valid topology. The basic algorithm may generate geometric artifacts such as gaps and slivers, which generally can be removed by a data-smoothing post-process. The second algorithm presented performs simplification in 2D, converting a geometrically invalid (but topologically valid) shape representation into one that is fully valid. This algorithm is based on a variant of the Bentley-Ottmann sweep line algorithm, but with additional rules to handle situations not possible under an exact implementation.

Both algorithms are presented in the context of what is required of an algorithm in order for it to be classed as robust in some sense. I explain why the formulaic approach used for the Boolean algorithm cannot readily be used for the simplification process. I also give essential code details for a C++ implementation of the 2D simplification algorithm, and discuss the results of extreme tests designed to show up any problems. Finally, I discuss floating-point arithmetic, present error analysis for the floating-point computation of the intersection point between two segments in 2D, and discuss how such errors affect both the simplification algorithm and the basic Boolean algorithm in 2D.

纪念

柳明生

一九六四年 —— 二零零零年

Acknowledgements

Thanks are due to my supervisor, Neil Dodgson, and my second advisor, Malcolm Sabin, for their assistance with my work.

I also thank those who answered my questions or gave help in other ways, including Patrick Campbell-Preston, Ioannis Emiris, Rida Farouki, Dan Halperin, Lutz Kettner, Alan Middleditch, Alan Mycroft, Sylvain Pion, Stefan Schirra and Chee Yap. Thanks also to Ursula Augsdörfer, Rob Hague and Richard Southern for their technical help, and to Ma Jian for pointing out mistakes in my paper [SD07], which stand corrected here.

Finally, I would like to thank to my partner, Sunne, for his patience and tolerance.

Contents

Index of terms

Chapter 1

Introduction

Problems of robustness are a major cause for concern in the implementation of algorithms relating to geometry [Hof89, HHK89, dBvKOS97, Sch99, WDF$^+$98, Far99, Hof01, MY04, Hal04, Yap04]. Geometric operations are prone to many types of error, ranging from unacceptably large errors, or discrepancies such as cracks in the supposed boundary surface, through to strange program behaviour, loss of data or outright crashes [WDF$^+$98]. A major reason for the problems is that geometric algorithms are often a mix of arithmetic and combinatorial computations, arising from the need to create geometric data structures that are themselves a complex mix of numerical and combinatorial data. The extent of interplay between the two types of computation is a critical aspect of geometric and solid modelling, and appears to be without parallel [Hof89, p.8]. Decisions that influence the topology of geometric structures are made on the basis of certain arithmetic calculations that may be affected by numerical rounding error. However, the consequence of such errors is not simply one of minor differences between the correct result and the computed result; the numerical errors may, in fact, lead to inconsistencies in the decisions that adversely affect the algorithm. There are also additional complications insofar that the structure generated has to adhere to certain numerical constraints in order to be valid, so that, for example, the boundaries do not self-intersect. Taken together, these complications hinder or prevent the formation of a structure that is both valid and sufficiently accurate.

The seriousness of the difficulties associated with geometric algorithms cannot be understated. Geometric processing is required for a wide range of applications that handle shape information of one type or other, including those in the fields of CAD, CAE and CAM (Computer-Aided Design, Engineering and Manufacture), NC (numerical control), PDM (Product Data Management) and GIS (Geographic Information Systems). Many geometric operations required by these systems have the potential to fail catastrophically. Even the apparently straightforward operation of determining the convex hull of a finite set of points on the plane is prone to these difficulties [MY04, KMP$^+$08]. One important operation required in many geometric applications that is particularly prone to these difficulties, is the Boolean operation; this dissertation focuses on this operation. Other geometric operations prone to robustness problems include Voronoi cell determination and Delaunay triangulation.

The operations performed are sometimes in the piecewise linear domain, meaning that all shapes are polygonal or polyhedral in nature. Even when an application needs to deal with shapes that have curved boundaries, it may, depending on requirements, be acceptable and expedient to approximate the intended shapes with a faceted approximation. Sometimes operations are required to function in a particular curved domain allowing a specific range of curve and/or surface types. Operations in curved domains have additional complications because of the complexity of the algebraic equations involved; determining the solution to these is a major subject area in its own right [Hof89, PM02]. However, geometric operations in all domains have the potential to be adversely affected by robustness problems relating to consistency if they concern the creation of topological geometric structures.

Because of the issue of robustness, the development of geometric systems can be costly. The first phase of implementing a geometric algorithm is usually followed by a second, time-consuming phase, fine-tuning the system to avoid failure for common inputs. Emphasis is placed, in particular, on ensuring that the system functions satisfactorily for known special cases that are easily understood. However, this adds to the complexity of the system. The difficulty is that for many geometric problems there are several categories of special cases requiring special consideration, so it is not practical to extend special-case handling to work satisfactorily for all problem cases without at the same time introducing new problem cases leading to system failure. Although the enhancements may reduce failure rates, they are unlikely to prevent failure altogether. Furthermore, systems developed this way are difficult to maintain and prone to destabilisation when changes to the system are made [Hof89, pp.8-9].

The fact that geometric systems sometimes fail is costly to users because it hinders full automation. At the very least, users are obliged to tweak perfectly valid input data in order to side-step system failure. While this might be tolerable for small-scale operations, the consequences are more serious in large-scale, multi-user applications, when large numbers of operations are carried out successively without user intervention. The data used may have been created at different times by various teams of people unrelated to the user whose actions trigger the unsuccessful operation.

One such example relates to my own software development experience, working on Cadcentre's Plant Design Management System[AVE06]—a CAD system for process plant and oil and gas industries. Engineers with different responsibilities use the system to specify geometric representations of various parts of a plant being designed: vessels, pipe-work, supporting structures, and heating and ventilation systems for a plant. These engineers in turn rely on instances of generic standard template models of items, each defined parametrically, and selected from a general catalogue database created separately in advance. Once the plant design is complete, a drafter may then run a separate program to create an annotated engineering drawing based on a large part of the plant. The first stage of the drafting program is to generate an approximate, faceted model of the part of the plant, so that it can be drawn in hidden line style. The model is created by performing a series of Boolean operations on polyhedral shape representations, starting with faceted approximations of the basic building block shapes: cuboids, cylinder, tori, etc.. In the original version of the code (now no longer in use) the Boolean operation code was prone to failure, and in certain cases it became necessary to discard the facet structure when it ceased to be meaningful. The failure of just one Boolean operation out of thousands could cause a crucial item, possibly large, to 'disappear' from the drawing. The cost to the client company was not simply the time lost looking out for and responding to errors; there was also the risk that errors would remain undetected. Furthermore, to modify model data at the drafting stage is generally regarded as bad data management practice. These robustness problems ceased once the code was replaced with an implementation of the basic Boolean algorithm that I go on to describe in this document.

The issue of robustness is even more serious when the boundary mesh generated by a CAD package is used by a CFD (Computational Fluid Dynamics) simulation package. This arises when a CAD model of a ship or aircraft is tested for hydrodynamic or aerodynamic efficiency [Far99, SSZ$^+$04, KBF05]. The volume mesh used by the CFD package to represent the vessel exterior must be constructed from the boundary mesh generated by the CAD package. However, the generated boundary mesh is usually unsuitable as it stands, because of surface cracks and other defects to the mesh, and so has to be repaired. Reporting on an interdisciplinary workshop in 1999, Farouki [Far99] points out that the time spent on mesh repair is prohibitively expensive: one to four weeks. In comparison, the other mesh preparation tasks—creating the original surface mesh and creating a volume mesh from the *repaired* surface mesh—take about four hours, and the actual flow analysis takes one hour. Similar difficulties arise with stress analysis tests that require a volume mesh representation of the interior [Hal97].

Placing the issue in a financial context: a 2002 US government report [NIS02] states that the costs due to bugs (or 'an inadequate software testing infrastructure') in CAD, CAM, CAE and PDM software used by the US automotive and aerospace industries amount to US$1.84 billion. It is reasonable to assume that a significant proportion of these costs, borne by developers and users, are related to geometric operations. Furthermore, the consensus among CAD researchers at an MSRI Workshop on Mathematical Foundations of CAD in 1999 was that “the single greatest cause of poor reliability of CAD systems is lack of topologically consistent surface intersection algorithms” [MY04, Lecture 1, p.6].

The intractability of the problem of geometric robustness seems overwhelming. Given the extent of the problem, one might expect industry and academia to make a concerted effort to address the matter in order to resolve it once and for all. For the most part this has not been so. To quote from the introduction to a panel discussion on robustness at the 1998 SIGGRAPH conference [WDF+98]:

> Until recently, most of these issues have been privately debated among friendly colleagues but publicly swept under the rug by both academics and practitioners.... There has been little organized effort to provide methodologies for either avoiding geometric errors or for proving the correctness and accuracy of geometry based computer programs.

The reasons for lack of effort would appear to be socio-political as much as they are technical. To quote from Farouki’s report on CAD/CFD integration [Far99]:

> The problem is partly cultural in nature: The development of CAD systems has been driven by product-release deadlines and by management teams poorly versed in the fundamental mathematical difficulties; the heuristic methods implemented in CAD systems work sufficiently well to make them worthwhile tools in industry, but not without inflicting great pain and exasperation on their users in challenging contexts like the CAD/CFD interface; and the academic CAD research community has largely forsaken many of the fundamental issues and sought refuge in simpler problems that lead to easy publications.

The latter criticism would appear particularly relevant (but not exclusively) to the field of computational geometry, which to a large extent is concerned with complexity of geometric algorithms. Here it is standard to consider the theoretical complexity of an algorithm based on the assumption that it is run on a hypothetical *real random access machine* (or *real RAM*). This stipulates (1) that real values are stored and computed exactly, and (2) that each arithmetic operation is of order unit cost, meaning that any arithmetic operation has an execution time bound over the full range of permitted input [PS85, pp.26-28]. The first assumption does not hold for an implementation based on standard, finite-precision arithmetic, so the problems of robustness are in fact side-tracked. Implementations based on exact computation (discussed shortly) avoid the problems of inconsistency that lead to non-robustness, but for these the second assumption cannot hold. A further issue is that the algorithms are very often described with the assumption that the input data are in ‘general position’—that is to say, no consideration is given to degenerate or ‘borderline’ situations, in which an arbitrarily small change to the numerical input can alter the topological structure of the result. In real life problems, the data may well be borderline, whether by accident or (as is often the case for many applications) by design. The implicit assumption is that the handling of special cases is straightforward, but often this is not so because of the many special cases that have to be considered [MY04, L.8 pp.1-2],[Sch99, p.623]. Only a few of the many algorithms developed in computational geometry have found their way into practice, and the lack of progress is attributed largely to the inappropriateness of the theoretical assumptions just described [FGK$^+$96].

There has, nevertheless, been some research into robust geometric algorithms. Papers published show a build up of interest in the 1980s and into the early 1990s concerning one or other specific operation, usually in the piecewise linear domain, and often limited to the two-dimensional form of the operation; see section 2.4 for more detail. For the most part, the methods concerned are approximate, and as such they generate a result—usually not the true result—considered in some sense acceptable.

Some researchers consider that the exact approach offers the best prospects for implementing algorithms robustly [Hof01]. A simplistic implementation in which every arithmetic value is computed exactly can be very inefficient. However, it is usually not necessary to do this to achieve robustness; it is in fact sufficient to determine exactly the logical states that affect the algorithm path, since this guarantees that the computed result has the correct topology. The term *exact geometric computation* (*EGC*) is used for any computation carried out this way [Yap04, pp.935-936]. The success of the EGC approach depends on being able to perform the essential calculations efficiently. One important technique is the use of *floating-point filters*, whereby values are computed using floating-point arithmetic and compared to a known error bound; this usually ensures that the vast majority of the calculations required are computed efficiently without resorting to representing values exactly [Yap04, pp.939-940].

The last decade has seen the emergence of a research culture committed to the exact approach. A consortium of Computational Geometry departments from Europe and Israel have played a significant role in this movement. In a series of EU-funded projects dating from 1996 they collaborated to create, and subsequently extend, the library known as CGAL: the Computational Geometry Algorithms Library, providing both general tools for the EGC approach and specific geometrical algorithms [CGA08, FT07]. The two most recent projects—ECG and ACS—extend the EGC technique to problems relating to curves and surfaces [CGA08].

Mehlhorn and Yap [MY04, L.1 p.6] use the data in table 1.1 to demonstrate the advantage of using exact geometric computation. Commercial geometric modellers that do not use exact computation run into difficulties for certain cases when computing the union on the plane between an n-sided regular polygon and a copy of itself rotated by α degrees. In

SYSTEM	n	α	TIME	OUTPUT
ACIS	1000	1.0e-4	5 min	correct
ACIS	1000	1.0e-5	4.5 min	correct
ACIS	1000	1.0e-6	30 sec	too difficult!
Microstation95	100	1.0e-2	2 sec	correct
Microstation95	100	0.5e-2	3 sec	incorrect!
Rhino3D	200	1.0e-2	15 sec	correct
Rhino3D	400	1.0e-2	–	crash!
CGAL/LEDA	5000	6.175e-6	30 sec	correct
CGAL/LEDA	5000	1.581e-9	34 sec	correct
CGAL/LEDA	20000	9.88e-7	141 sec	correct

Table 1.1: *Boolean operations on CAD software. From [MY04, L.1 p.6].*

contrast, the CGAL/LEDA code runs successfully for cases with larger n and smaller α that one would expect to be even more problematic.[1]

Although there has been much progress in the use of ECG techniques, at present there are problems of efficiency that prevent its use to any significant degree in commercial applications. Efficiency is problematic when there are serial operations: a chain of operations for which the output of one acts as input to the next. The difficulties are also particularly severe for algebraic problems of high degree, such as are used in many CAD applications [LPY04, p.90].

My work, as presented in this dissertation, is concerned principally with the issue of robustness in the context of approximate arithmetic. In particular, I consider the Boolean operation and a related operation—simplification (in the sense as defined by Fortune in [For95, pp.228-229])—in the piecewise linear domain. I do not discuss operations in curved domains in any detail, not because the topic is unimportant, but because it is crucial in the first instance for robustness issues in the linear domain to be well understood. Two algorithms that I have devised are explained: a 3D Boolean algorithm, and a 2D simplification algorithm. I was solely responsible for devising and implementing the Boolean algorithm for use within Cadcentre's Plant Design Management System [AVE06]; the implementation of this algorithm has been an integral part of that product to this day. An

[1] We should be cautious about what to conclude when comparing the reliability and performance of CGAL/LEDA with those of the commercial modellers, as it would appear that the 2D problem described was converted to 3D form when run on the commercial modellers, but not so when run on CGAL/LEDA—see the presentation file `www.mpi-inf.mpg.de/∼mehlhorn/ftp/EXACUS.pdf` , p.3. The table in [MY04] lists the time spent by ACIS for the case $\alpha = 10^{-6}$ to be 30 minutes, which I assume to be a copying error; the version of the table in the presentation file lists it as 30 seconds.

account of the 3D Boolean algorithm is published in [SD07], and [SD06] contains a briefer description. I devised the simplification algorithm as a variant of the Bentley-Ottmann sweep-line algorithm [BO79], and have implemented it for the purposes of investigation. I describe the requirements of these algorithms, after first discussing the requirements of robust approximate algorithms in a general context; I also describe the issues that affected the designs of these two algorithms, and explain why the techniques that were used for the Boolean algorithm could not also be used in an algorithm for the simplification operation, despite the superficial resemblance of the Boolean and simplification operations. Proofs and other arguments are put forward in support of the topological robustness of both algorithms.

The emphasis within this dissertation is on the theoretical background to both algorithms, though I also describe the achievements and limitations of the implementations of the algorithms. I provide information on the results of the simplification algorithm for specific problems. My account of my experience implementing the Boolean algorithm is necessarily limited, since I no longer work for Cadcentre (now named Aveva), and have no access to the code concerned or any documentation or notes.

Chapter 2 discusses the background to the problem of robustness, and includes a review of methods. Chapters 3, 4 and 5 discuss the Boolean algorithm, and are essentially taken from [SD07], with some modifications; chapter 3 discusses the requirements of a robust algorithm and other considerations that influence the design, while chapter 4 gives a description of the basic Boolean algorithm, including proofs of the topological robustness of the algorithm, and chapter 5 briefly discusses the data-smoothing post-process, giving in broad terms the requirements of that process, and outlining how I implemented it. Chapters 6 and 7 discuss issues relating to simplification; chapter 6 discusses the issues that affect the design of the algorithm, while chapter 7 describes my variant of the Bentley-Ottmann sweep-line algorithm for 2D simplification. Chapter 8 discusses the precision of floating-point arithmetic, and gives an error analysis of the 2D edge-edge intersection calculation. Finally, chapter 9 gives the conclusion, and discusses possible future work.

Chapter 2

Background

This chapter gives the background to the robustness issue for geometric computations. Section 2.1 describes the robustness problem in terms of conflicting levels of abstractions; this leads on to another issue—that of how inexact input should be handled—which is discussed briefly in section 2.2. Section 2.3 describes in some detail how and why algorithms can go wrong when inexact arithmetic is used. Section 2.4 describes both inexact and exact approaches of addressing the robustness issue. Finally, section 2.5 briefly discusses the issue of degeneracy, or marginal data, and the controversy that relates to it, in particular for exact methods.

2.1 Levels of abstraction

In computer science it is understood that the design of a system is aided by the practice of abstraction by which different levels of the system are considered in isolation of each other. Hoffmann [Hof89, pp.4-9] considers three levels of abstraction for a solid-modelling system that help to conceptualise issues:

1. The top-level abstraction is the *user interface* by which the user understands the system. This includes tools for managing models—construction, archiving, editing and deleting—and for querying and analysing a model, including display. The interface can be textual or visual, or indeed both.

2. The next level of abstraction is the *mathematical and algorithmic infrastructure*. At this level, the operations available in the user interface are considered in terms of their geometric/mathematical definitions and the algorithms to implement the operations. The operations considered might include, for example, the Boolean operation, blending, filleting, determining the intersection between two curved surfaces (normally a curve), or a sweep operation.

3. The lowest-level abstraction is the *substratum*. This consists of the arithmetic and symbolic computations as performed on the computer concerned, taking into account (where relevant) the limitations and approximate nature of machine integer and floating-point arithmetic.

It is the second of these three levels of abstraction—the mathematical and algorithmic infrastructure—that is traditionally the most prominent area of research in geometric and solid modelling [Hof89, p.6]. The ideal is for this area to be studied and developed in isolation without considering the other two levels, and indeed, much useful research is based on this assumption. However, this level of abstraction is unduly influenced by the two other layers if there is reliance on inexact arithmetic.

The incompatibility between the algorithmic infrastructure and the substratum is responsible for the main problems of non-robustness that concern us. We shall look into this in more detail shortly. The incompatibility between the user interface and the algorithmic infrastructure is also an issue of concern. The main difficulty here, though, is not the risk of failure, but rather the fact that the result may be 'wrong' in the user's eyes. The model constructed in advance for processing may not agree with the user's mental model of what they specified. This could happen, for example, if there is reliance on some other software—maybe the application that invokes the implemented operation—to set up the initial data. The user may intend to specify, say, exact coincidence between two boundary components, but this might not be achieved because of numerical errors in computing the initial data. Although it is not the main concern of the dissertation, it is appropriate to note how users' expectations of a geometric system have a bearing on the implementation of that system. This is discussed in the next section.

2.2 The implications of imprecise input data

In this section I consider how data imprecision influences the requirements of a geometric system. I do so specifically in the context of a system that computes Boolean operations, since this is relevant to later sections.

Let us first consider in broad terms the requirements for representing shapes within a geometric system if one were to implement Boolean operations in a precise manner. If the result of a Boolean operation is always determined exactly, and the results are available as input for subsequent Boolean operations, the range of shapes it is possible to construct is quite complex. This holds true even if the range of initial shapes is restricted to certain simple solids. For the true Boolean operation, which we call the *pointwise Boolean operation*, the resulting shape is defined point by point: the union, intersection or difference

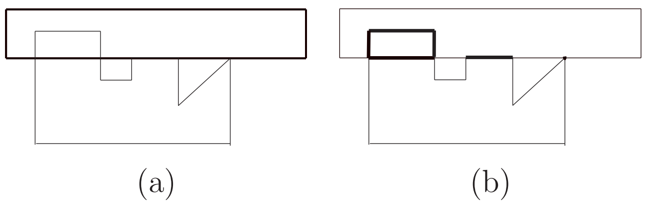


Figure 2.1: *A 2D example showing that the pointwise intersection of two regular shapes need not itself be regular. The original shapes (a) intersect to form a shape that includes a dangling edge and an isolated vertex (b).*

between two shapes is defined simply as the set of points that belong, respectively, to both shapes, to either shape, or to the first shape but not the second. Let us suppose for the sake of argument that the initial shapes allowed—those specified by the application that are not the result of a Boolean or other operation—are restricted to a specific group of solids[1] that are closed, so that the boundary of the solid is considered to be part of the shape. To be precise, we stipulate that an initial shape must be *regular*, meaning that it is equal to the closure of its interior. The pointwise Boolean operation applied to regular shapes does not necessarily generate a regular shape. Figure 2.1 shows one such example in which there are component parts that are not solid. If a shape such as this is subtracted from another shape in a subsequent difference operation, the component parts of the resulting shape (both solid and non-solid) may also have non-solid 'holes'. Any system that implements pointwise Boolean operations correctly must be able to represent shapes such as these. It is possible to do this, provided that all initial data are precise (or considered so), and all internal calculations are exact. Hachenberger, Kettner and Mehlhorn [HKM07] have implemented such a system in 3D within CGAL, in which Nef polyhedra are constructed from planar half-space primitives.

Many applications do not require the ability to represent non-regular shapes, because they only need to handle solids. Furthermore, one can argue it serves no purpose to represent such shapes, if the input is assumed to be imprecise. This is because for any problem that leads to the creation of a non-regular shape it is possible to make the result regular by means of an arbitrarily small modification to the data. If only regular shapes are to be represented, something other than the pointwise Boolean operation must be used. The standard approach taken is to use instead the *regularised Boolean operation* as described in [Req77, Req80, VR93, Mid94]. The regularised form of a particular pointwise Boolean operation is defined as the closure of the interior of the result of the equivalent pointwise Boolean operation. The result of this operation is also regular.

However, it may not be sufficient to rely on regularised Boolean operations in order to fulfill users' expectations. A user may wish to indicate that a solid is to be positioned

[1]I use 'solid' to indicate a space-filling shape, so the argument applies to 2D as well as 3D operations.

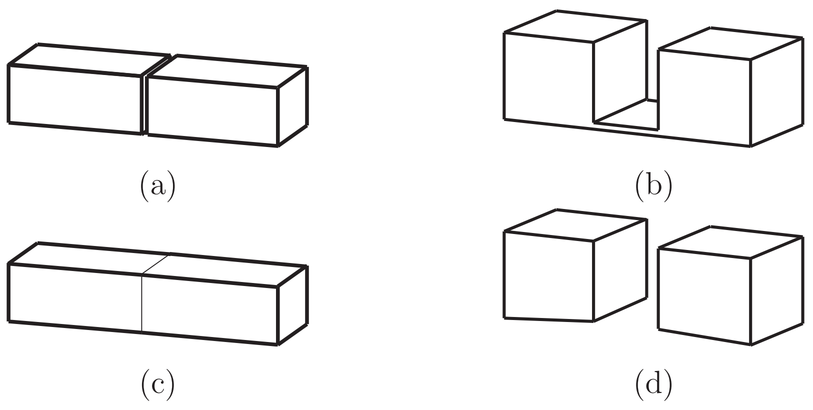


Figure 2.2: *Examples of (a) a gap generated by the union operation, and (b) a sliver generated by the difference operation. (c) and (d) demonstrate the result the user might prefer, without gap or sliver, if it was the user's intention for the faces to coincide exactly.*

end-to-end against another solid, so that their respective surfaces coincide and oppose each other, prior to the two solids being combined in a regularised union operation. This may not be achieved exactly in the input data structure because of numerical errors in the preparation of the data. The items may be incorrectly aligned, or they may be assumed to be incorrectly aligned according to the computations. There may be a gap between the two boundaries, or an overlap, or the boundaries may be skew. Inexactness in the substratum may complicate matters further. In certain circumstances it may be that users are content with a result other than the true result, in which case there is no issue. However, this situation is not satisfactory if users need, at least some of the time, to control positioning precisely in order to achieve a specific result, but find they cannot. In this situation there is an implied additional requirement to resolve gaps and slivers formed by near-coincident faces, as shown in figure 2.2.

If it is accepted that gaps and slivers below a certain thickness must be resolved, the issue then is how exactly should this be achieved. The process of epsilon-regularisation [QS06] is a well-defined operation that can resolve this (if applied exactly), but is inappropriate if rounded edges are to be avoided. An operation making a series of adjustments to 'smooth' the data might seem appropriate. In terms of software development, this may be a pragmatic approach to take, given the absence of suitable alternatives. But this differs from other operations insofar that its task is essentially to tidy up data, rather than emulate some well-defined abstract geometric process. Such an operation is imprecisely defined, and it is not clear what algorithmic rules should be applied, and in what order; furthermore, there is no guarantee that the operation will always terminate to resolve all gaps and slivers, and without introducing new glitches that would make the representation inappropriate.

In short, there are two sources of numerical errors in an approximate arithmetic implementation of a geometric algorithm: those present at the start of the operation in the input data, and those introduced later in the execution of the internal arithmetic. The former redefines the operation being performed and can alter the result; the latter has more serious consequences in terms of non-robustness of the operation.

The standard approach that many systems take to handle both types of numerical error is not to test for exact equality between two values, but to test for equality within a certain tolerance value, absolute or relative—see, for example, [CL01, pp.127-130] on how tolerances are specified for ACIS. This approach is flawed, for reasons I explain in the following section.

2.3 Why geometric computing is problematic

In this section we consider why geometric computations are prone to difficulties if they are implemented using approximate arithmetic. The first subsection looks into how arithmetic errors affect the nature of geometry; the remaining three subsections look at the issue of robustness in different contexts: for the apparently simple convex hull operation, for the Boolean operation that is the main focus of this dissertation, and finally in the context of geometric algorithms in general.

2.3.1 Inaccuracy and the implications of tolerance checking

Let us first look into a simple example of a geometric problem that raises the issue of consistency, presented in [MY04, Lecture 1]. Consider a library able to represent points and (infinite) lines on the plane, which can (1) compute the point of intersection between two non-parallel lines, and (2) determine whether a point lies on a line. By definition, the intersection point between two lines must lie on both lines, but will the computed intersection point be deemed to do so? The standard means of representing a line is by the real triple (a, b, c), that defines the line as the set of points $\{(x, y) \in \mathbb{R}^2 : ax + by + c = 0\}$. The coordinates of the intersection point between two non-parallel lines (a, b, c) and (a', b', c') are $x = (bc' - b'c)/(ab' - a'b)$ and $y = (ca' - c'a)/(ab' - a'b)$; by definition, point (x, y) lies on line (a, b, c) if $ax + by + c = 0$. Suppose that we have a simple, floating-point implementation of the library that determines the point of intersection according to the formula above, and that deems point (x, y) to lie on line (a, b, c) if $ax + by + c$ is computed to be exactly 0. Because of approximation errors the library will often fail to deem the intersection point between two lines to lie on one or other line [MY04, L.1 p.3].

A common approach that programmers take to resolve this issue is to apply arithmetic tests to within a given tolerance. Hence, the point-on-line test operates by determining whether $ax + by + c$ is computed to be of magnitude less than some selected tolerance value, $\varepsilon > 0$. If the line representations are always held in normalised form such that $a^2 + b^2 = 1$, then $ax + by + c$ represents the signed distance from the line, and the test supposedly determines (assuming exact arithmetic) whether the point lies within distance ε of the line. We can think of this as converting points and lines to 'fat' versions of themselves, so that points become discs of radius $\varepsilon_1 > 0$ and lines become bands with half-width $\varepsilon_2 > 0$, where $\varepsilon_1 + \varepsilon_2 = \varepsilon$. This is an artificial interpretation, however, because the intersection between two 'fat' lines is really a rhombus rather than a disc. The geometry that underlies the system is something other than the Euclidean geometry that is largely well understood at both an intuitive and mathematical level. Situations that are impossible in Euclidean geometry are made possible. For example, it is possible for two distinct points both to lie on two distinct lines. So while the use of tolerancing may resolve specific, localised problems that relate to approximate arithmetic, the strategy in itself does not help towards understanding the underlying geometric system. The occurrence of arithmetic rounding makes it difficult to discern any meaningful rules for the new geometry, and the introduction of tolerancing does not resolve that difficulty [MY04, L.1 p.4].

For many applications it may not matter that the underlying geometry is not fully understood. Geometric tests such as the point-on-line test just described may be required for a series of localised computations that have no bearing on each other, in which case it may be sufficient to accept the results of the tests as correct. Difficulties arise when the results of individual tests each have a bearing on the construction of a combinatorial or topological data structure, as we shall see in the following subsection.

2.3.2 The incremental 2D convex hull algorithm

The situation regarding approximate geometric calculations has serious consequences for more complex geometric operations in which inexact calculations influence the construction of any topological structures. This subsection considers the potential problems for what might be considered the simplest operation under this category: that of determining the (2D) convex hull for a finite set of points on the plane. The description given here closely follows Kettner et al. [KMP$^+$08], who describe the robustness problems to a degree of detail not seen in other publications.

An algorithm that constructs a topological structure of any sort can be considered as being driven by a series of arithmetic tests, each of which may influence the path of the algorithm and ultimately on the result. In general terms, any arithmetic test determining a logical state that subsequently influences the construction or modification of a geometric structure or that otherwise influences the path of an algorithm is commonly known as a *predicate* [Sch99, Hof01, Mou04, KN04, MY04, KMP+08]. The following tests, applied on the plane, are examples of such predicates: whether two line segments intersect; whether a sequence of three points appear in anti-clockwise or clockwise order, or are collinear; and whether a point lies inside, outside or on the circle defined by three other points [Sch99, p.602].

The second of these predicates, relating to the ordering of three points, is known as the *orientation predicate*. It is required for any algorithm performing the 2D convex hull operation. The arithmetic value that determines the predicate is the signed area function, defined as:

$$\Delta(\mathbf{p},\mathbf{q},\mathbf{r}) = (\mathbf{q}-\mathbf{p}) \times (\mathbf{r}-\mathbf{p}) = (x_q - x_p)(y_r - y_p) - (y_q - y_p)(x_r - x_p) \tag{2.1}$$

This represents two times the area of the triangle defined by the three points. The three possible states of the orientation predicate are determined by the sign of this value:

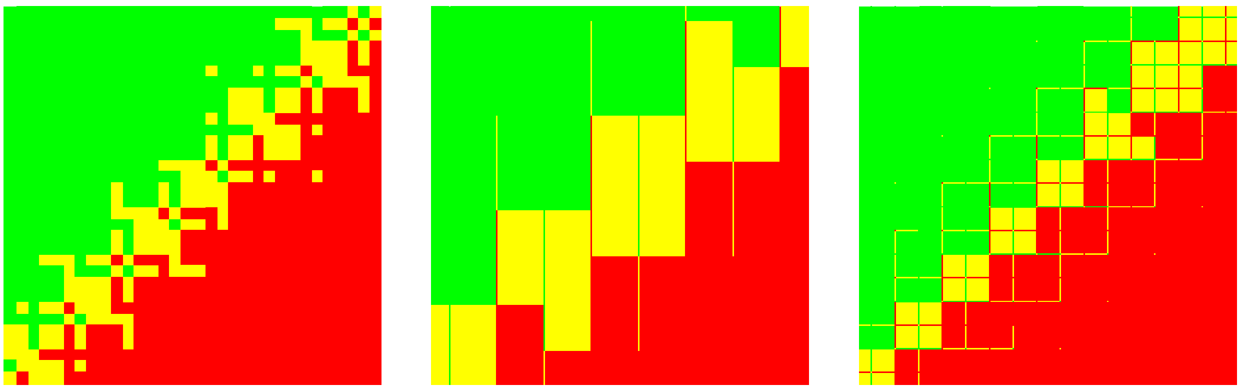
$$\text{ori}\,(\mathbf{p},\mathbf{q},\mathbf{r}) = \text{ sign}\,(\Delta(\mathbf{p},\mathbf{q},\mathbf{r})) \tag{2.2}$$

A positive value of Δ denotes an anti-clockwise order, a negative value denotes a clockwise order, and zero denotes that the points are collinear [KMP+08].

Kettner et al. [KMP+08] present a series of diagrams, reconstructed here in figure 2.3, that show the computed orientation value $\text{ori}\,(\mathbf{p},\mathbf{q},\mathbf{r})$ over a range of $\mathbf{p}$ (with $\mathbf{q}$ and $\mathbf{r}$ remaining fixed). IEEE 754 standard floating-point arithmetic is used.[2] The results form quite a complex pattern for the cases selected, with each region for the three possible results forming 'islands' and 'rays', rather than a simple, contiguous band structure one might expect. The authors go on to give a detailed case study of how the task of determining the convex hull of a finite set of points on the plane can encounter non-robustness problems. They consider the incremental algorithm, in which one point at a time is added, and assume that the signed area function is evaluated as expressed in equation (2.1) using floating point arithmetic. There are other algorithms for determining the convex hull, and the authors note that some are more reliable; nevertheless, a look into the non-robustness problems relating to this algorithm is a useful introduction to the problems that affect many algorithms.

The input to the algorithm is an unstructured, finite set of points, S. The convex hull is the smallest closed, convex polygon that includes all the points. It can be represented simply as a circular list of those points from S that form the strictly convex vertices of that polygon. The hull vertices are assumed to be listed in anti-clockwise order: $L = (\mathbf{v}_0, \mathbf{v}_1, ..., \mathbf{v}_{k-1})$, with indices modulo k. Each adjacent pair of vertices, $(\mathbf{v}_i, \mathbf{v}_{i+1})$, represents an edge of the convex hull. Edge $(\mathbf{v}_i, \mathbf{v}_{i+1})$ and point $\mathbf{r}$ are said to be 'visible' to each other if $\Delta(\mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{r}) < 0$, and 'weakly visible' if $\Delta(\mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{r}) \leq 0$.

[2]The IEEE 754 standard is described in section 8.1.



	(a)	(b)	(c)
p:	$\begin{pmatrix}0.5\\0.5\end{pmatrix}$	$\begin{pmatrix}0.50000000000002531\\0.5000000000000171\end{pmatrix}$	$\begin{pmatrix}0.5\\0.5\end{pmatrix}$
q:	$\begin{pmatrix}12\\12\end{pmatrix}$	$\begin{pmatrix}17.300000000000001\\17.300000000000001\end{pmatrix}$	$\begin{pmatrix}8.000000000000007\\8.000000000000007\end{pmatrix}$
r:	$\begin{pmatrix}24\\24\end{pmatrix}$	$\begin{pmatrix}24.00000000000005\\24.0000000000000517765\end{pmatrix}$	$\begin{pmatrix}12.1\\12.1\end{pmatrix}$

Figure 2.3: *The figure shows the results of* $ori\,((x_p + iu, y_p + ju), \mathbf{q}, \mathbf{r})$ *as computed by the formula given in equations (2.1) & (2.2) for* $0 \leq i \leq 255$ *and* $0 \leq j \leq 255$*, where* $u = 2^{-53}$ *is the increment between adjacent, double-precision floating-point numbers in the range* $[\frac{1}{2}, 1]$*. Red/yellow/green designates a negative/zero/positive result respectively. The results here duplicate those of Kettner et al. [KMP+08].*

The incremental algorithm considered by Kettner et al. for determining the convex hull is as follows: Given a point set S and a vertex $\mathbf{r}$, the point set is augmented to $S' = S \cup \{\mathbf{r}\}$, and L is updated accordingly to represent $C(S')$ in place of $C(S)$. There are two scenarios as to how L is affected:

- $\mathbf{r} \in C(S)$: L remains unchanged;
- $\mathbf{r} \notin C(S)$: $\mathbf{r}$ is inserted into L, possibly replacing a consecutive sub-chain of vertices.

Kettner et al. identify the following properties, possible under Euclidean geometry, that relate to a candidate point to be added to the point set:

- **Property A:** A point is outside the hull if and only if at least one hull edge is visible to it.
- **Property B:** For a point outside the hull, the set of hull edges weakly visible to it form a non-empty, consecutive sub-chain of L; likewise, the set of hull edges that are *not* weakly visible to the point form a non-empty, consecutive sub-chain of L.

The algorithm relies on these properties. Hence, it seeks a hull edge visible to $\mathbf{r}$, to determine whether $\mathbf{r}$ is exterior to $C(S)$. Having found such an edge, it then examines hull edges on either side, until it finds the first hull edge to precede it that is not weakly visible, $(\mathbf{v}_{i-1}, \mathbf{v}_i)$, and also the first hull edge to follow it that is not weakly visible, $(\mathbf{v}_j, \mathbf{v}_{j+1})$. The chain of weakly visible edges, $(\mathbf{v}_i, \mathbf{v}_{i+1})$, $(\mathbf{v}_{i+1}, \mathbf{v}_{i+2})$,...,$(\mathbf{v}_{j-1}, \mathbf{v}_j)$, are then removed from L and replaced by the two edges $(\mathbf{v}_i, \mathbf{r})$ and $(\mathbf{r}, \mathbf{v}_j)$.

However, properties A and B do not necessarily hold when floating-point arithmetic is used to evaluate the predicates. Kettner et al. identify the following situations that can arise:

- **Failure $\mathbf{A_1}$:** A point outside the hull is not visible to any hull edge.
- **Failure $\mathbf{A_2}$:** A point inside the hull is visible to a hull edge.
- **Failure $\mathbf{B_1}$:** A point outside the hull is weakly visible to all edges of the hull.
- **Failure $\mathbf{B_2}$:** A point outside the hull is weakly visible to a non-contiguous set of hull edges.

The failures arise in situations when four points or more are close to collinear. The effect of these failures on the final result can be catastrophic:

- The algorithm computes a convex polygon, but a sample point lies outside the computed hull by a significant distance.
- The algorithm loops indefinitely, so the program hangs or crashes.
- The algorithm computes a non-convex polygon, with a concave vertex of the computed hull lying a significant distance inside the true convex hull.

The first case is a direct consequence of Failure A_1, while the second is a direct consequence of Failure B_1. The third is triggered as a consequence of Failure A_2, which initially creates a hull with a concave vertex adjoining a very short edge; a more substantial concavity is formed subsequently, when the set of edges weakly visible to a subsequent point being processed forms a non-contiguous set.

The use of tolerance checking in evaluating the orientation predicate does not prevent Failure A_1; indeed, the introduction of tolerancing is liable to increase the likelihood of this type of failure [KMP+08].

Kettner et al. [KMP+08] also report that the incremental 3D Delaunay triangulation algorithm can fail, leading to the non-termination of the algorithm.

To conclude, we see that the issue of robustness affects even the 2D convex hull operation. The convex hull operation is not as complex as other operations that generate geometric structures, and these too have robustness problems.

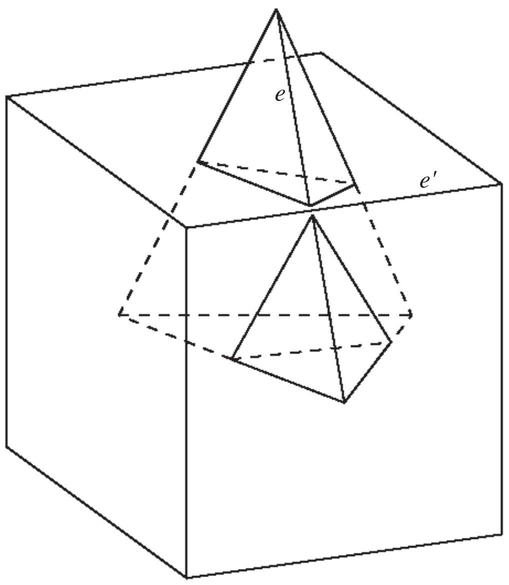


Figure 2.4: *Example in which edges are close together. From [Hof01].*

2.3.3 Problems of consistency in the 3D Boolean operation

We now consider the problem of robustness for 3D Boolean operations on polyhedral shapes in boundary representation form.

Let us start by considering how the operation is usually implemented. A shape is represented by a graph structure indicating the adjacency relationship between the component faces, edges and vertices that form the boundary. The general rule for constructing the result of the operation, assuming that faces do not coincide (and are not computed as doing so), is to determine the intersection curve(s) between the two surfaces, and stitch together at that curve the relevant parts of each boundary. For the union operation, for example, it is the part of the surface exterior to the other shape that is selected. Special rules apply when there are coincident faces. Usually it is assumed that the operation performed is the regularised Boolean operation; for this, the area common to both faces is retained only if it marks the border between the interior and exterior of the result. For details, see, for example, [Hof89].

When the resulting structure is created it is essential for the adjacency data to be consistent. Figure 2.4, originally in [Hof01], shows an example case that can be problematic. Part of the task to determine the intersection curve between the two surfaces is to locate every point where an edge of one shape intersects a face of the other, as these mark the limit points of the intersection edges formed by the intersection between two faces. In the example, the front edge of the tetrahedron, e, is very close to the upper front edge of the cube, e'. In a casual implementation, the point where edge e intersects the upper face of the cube may be computed to be in the interior of that face, while the point where e

intersects the front face may be computed to lie on the bounding edge e'. Hence, there is an inconsistency, because e is considered to intersect the upper face of the cube twice [Hof01, p.5]. Other types of failure are possible, for example, if e is determined not to intersect the front face of the cube, or if e' is determined to intersect one or both of the tetrahedron faces adjoining e.

Edge-to-edge coincidence is not the only borderline case that has the potential of leading to inconsistencies. There are similar issues relating to a vertex close to a face. The decision about whether the vertex lies on the interior or exterior side of the face or on it, has to be consistent with decisions about whether edges connected to the vertex intersect the face. In addition, there are what can be considered as higher orders of coincidence: a vertex coinciding with an edge, a vertex coinciding with a vertex, an edge coinciding with an edge over some length (not just intersecting), an edge coinciding with a face, a face coinciding with a face. Furthermore, if the checks made are tolerance-based, as is often the case, there are other cases to consider. For example, if two facets almost oppose each other to form a sharp wedge, a vertex may lie sufficiently close to both facets to be considered lying on them both, yet it may also lie a significant distance away from the edge connecting the facets [Hof01, p.20].

One particular case of the Boolean operation often cited as problematic is that of combining a cube with a copy of itself that has been rotated in turn by a certain angle about each of the three axes [LTH86, HHK89]. For most geometric software, this operation will fail over a range of angle values because of problems of consistency. If the angle of rotation is sufficiently small, there is no problem (assuming tests are tolerance-based): each face is deemed coincident to the corresponding face of the other solid, and likewise each edge to an edge and each vertex to a vertex. The special handling of coincident faces applies, and each pair of faces are merged. If the angle is sufficiently large, the algorithm can also cope: the data are not coincident or close to coincident, so no inconsistencies arise in computing the points of the intersection curve, where an edge intersects a face. Consider, though, the nature of the problem as the angle of rotation crosses between the two ranges. As the angle decreases, certain pairs of entities will become coincident, according to the computations, but the angle at which they do so will almost certainly not be the same, given that the decisions on coincidence may be determined according to different formulae, and that arithmetic errors may also have an influence. Consequently, over this range there will be inconsistencies that prevent the formation of a valid topology in the result.

2.3.4 Problems of robustness in general

In this subsection, we consider the problems of robustness in general terms.

We have noted that algorithms go wrong when the results of geometric tests form a particular combination that should not be possible under exact Euclidean geometry. For the incremental 2D convex hull algorithm, four types of failure were identified (section 2.3.2, p.31). For the 3D Boolean operation, it was noted that the results of certain groups of tests must be consistent (section 2.3.3, p.32, and figure 2.4). The various combinations of predicate values that must be avoided can be represented in terms of constraints to be applied to the predicate values.

It may help to understand the issues concerned by considering a geometric algorithm as a mapping from one geometric structure to another, representing the input and the result respectively, with each structure having both a topological and numerical part [Sch99, p.601]. Consider the scenario in which the topology of the input structure remains fixed, but the real data values specifying the geometry of the input are allowed to vary. For a particular input topology, the size of the numerical part of the input is fixed at (say) N real values, so that all possible inputs, valid and invalid, form an N-dimensional space.[3] We may consider the input geometry space as being subdivided into regions, each one representing a particular result topology. For a robust algorithm, the border between one such region and another marks a change to the result topology. That border must also mark a change in at least one of the predicate values. However, the changing of just one predicate value may lead to the breaching of one or more of the constraints that apply. Certain other predicate values must also switch at exactly the same boundary in order to ensure there are no regions in input space that lead to an illegal combination of predicate values.

Observing the robustness issue this way, we see that robustness problems cannot easily be resolved by special handling of borderline cases and opting for a particular selection of tolerance values. Because of the need for predicate values to switch at exactly the same boundary, no amount of 'epsilon tweaking'—the practice of experimenting with tolerance values—is likely to achieve full robustness. For a system to be considered robust, there needs to be a clear argument as to why the computed predicate values will always be consistent with each other.

The fact that certain combinations of predicate values have to be avoided is the source of the robustness problem. It is usual for geometric algorithms that certain combinations of predicate values are incompatible. The question arises: for a particular operation, can an algorithm be formulated in such a way so that all combinations of computed predicate values are theoretically possible in exact Euclidean geometry? Such an algorithm, which Fortune calls *parsimonious* [For89, Knu92, She99, Sch99], would have the distinct

[3]It is assumed that the input data will stray, at least to some extent, from the constraints that are supposed to apply. Hence the dimensionality of the input is equal to the number of real values represented, rather than the number of degrees of freedom for the original problem.

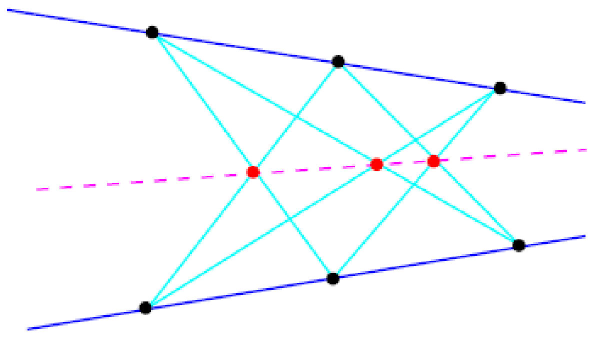


Figure 2.5: *Pappus's theorem: collinearity of the points on the top line and on the bottom line implies collinearity of the three intersection points in the middle. From [Sch98]/[Sch99].*

advantage that it avoids robustness problems altogether.

In principle at least, a non-parsimonious algorithm can be converted into a parsimonious algorithm. Consider a non-parsimonious algorithm evaluating predicate values in sequence. Usually when a predicate is evaluated it is legitimate for it to take the value either true or false. At some stage, though, it may be that the predicate concerned can only legitimately take one specific value, since the other value would have led to an invalid combination of predicate values. If whenever this occurs the predicate is always assigned the only permitted value, the algorithm in its modified form is parsimonious. However, the task of determining an implied predicate value is generally assumed to be complex. For example, the task of determining whether the orientation of three points in 2D is implied from the results of previous orientation tests is said to be NP hard [For89, Gui96, She99, Sch99]. Pappus's theorem provides an example case of an implied relationship that is not immediately obvious: namely, that the three derived intersection points are collinear—see figure 2.5 [Sch99, pp.616-617].

2.3.5 Maintaining boundary integrity

We have seen that geometric algorithms can be affected by the problem of inconsistent decisions, but there is another issue that we also have to be alert to for certain operations. When the result is a shape, it is necessary to maintain the integrity of the result boundary, so that it has no self-intersections or overlaps. Also, the orientation of the boundary should always be correct, so that so that the 'outer' side always marks the shape exterior, and the 'inner' side its interior. The risk of this failing arises not just the Boolean operation, but for almost all types of operation involving shapes. Even transformation operations that shift, rotate, scale or shear a shape can cause this, even though such operations generally do not affect the boundary topology—see figure 2.6. There are similar problems for operations in which the result is a spatial subdivision, in which a region is considered

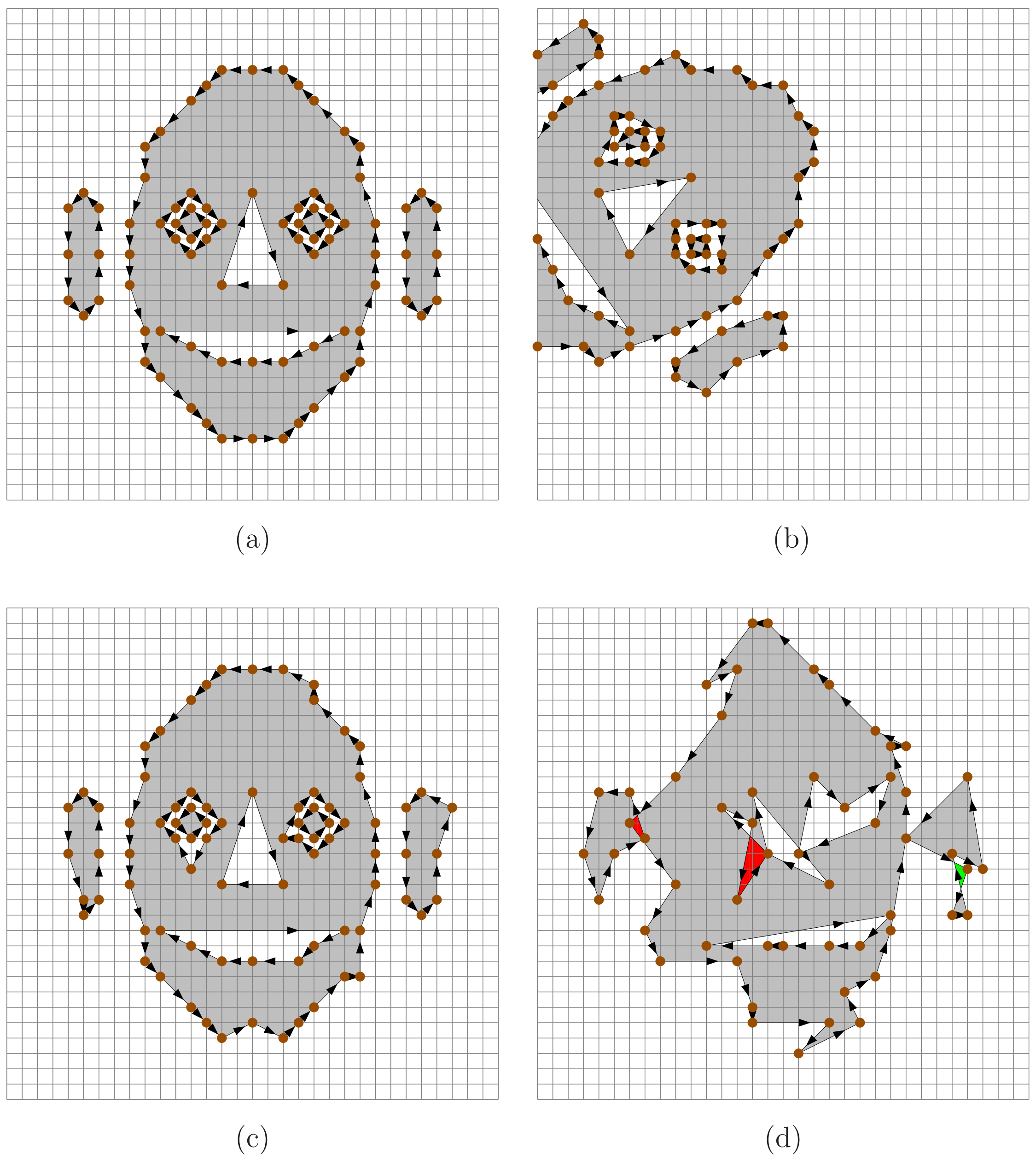


Figure 2.6: *Example showing that simple rotations can lead to geometric errors. A polygonal shape (multi-region with holes) in boundary-representation, originally residing in the region* $[-1, 1] \times [-1, 1]$, *is displaced by the vector* $(1.5 \times 2^{48}, 1.5 \times 2^{48})$, *so that vertices lie at representable points on a grid with a spacing of* $\frac{1}{16}$, *as in figure (a). The shape is distorted when it is rotated about a point (b) and then back (c). The shape representation shown in (d) is the consequence of rotating the shape representation one way and then rotating it back repeatedly, applying in total 32 pairs of rotations, each time applying a different, randomly selected, initial rotation. The boundary in (d) is distorted to such an extent that it intersects itself, hence forming doubly-enclosed and inside-out regions marked red and green respectively.*

subdivided into a collection of mutually exclusive shapes that occupy the whole region. Here the risk is that the cells may overlap, with one or more cells being inside out over the region of overlap.

2.4 Review of methods

There are essentially two approaches to addressing the problems of robustness in geometric computing: making fixed-precision computation robust, and making the exact approach viable [Yap04, p.927]. I describe in this section the advances that have been achieved by these two approaches. My account is largely based on review chapters on the subject by Schirra [Sch99] (also available as [Sch98]) and by Yap [Yap04], and on a review paper of exact methods by Li, Pion and Yap [LPY04]. The account is also influenced by a paper discussing the robustness issue in general terms by Hoffmann [Hof01], and a draft book, available online, covering the subject in a series of lectures by Mehlhorn and Yap [MY04].

2.4.1 Approaches based on inexact arithmetic

For any approximate computation it is accepted that the computed result may be something other than the true result. But what properties must the computed result have in order for it to be considered acceptable? Schirra [Sch99, p.626] points out that there is no general theory on how to deal with imprecision. In numerical analysis, an algorithm is considered *robust* if it produces the correct result for some perturbation of the input, and *stable* if the perturbation is small. In the context of geometric problems it is not clear what should be meant by perturbation. The question arises in particular as to whether the topological part of the data can in some sense be 'perturbed' [Sch99, p.600]. In some situations it may be considered appropriate to allow the result also to be perturbed. For example, a convex hull algorithm may return a polygon that is not quite convex, but within certain bounds close to the true result; this might or might not be acceptable, depending on the application that makes use of the result. Fortune's definition of robustness and stability [For89] has an extra requirement that an implementation based on precise computations would lead to the exact result [Sch99, p.616].

With approximate geometry there is no single way to represent the geometric entities required. The method of representation and the constraints that apply are selected to maintain certain properties and ensure correct behaviour of the algorithm. Yap [Yap04, pp.930-931] identifies different ways of approximating a line on the plane (see also [MY04, L.5 pp.1-3]). One way is to represent the line as a polyline, as first proposed by Greene and Yao [GY86]; for this, a line is approximated by constraining it to pass through specific grid points. A second way is to represent a line as a 'fat line', in which the line is assumed to have finite thickness, and is bounded by two polylines, as in [SS85]. This is a form of 'interval geometry', equivalent to interval arithmetic. A third way is by means of a finite precision representation of the parameter defining the line, (a, b, c) $(ax + by + c = 0)$, as used by Sugihara in [Sug89]; for this, Sugihara advocates that the three parameters be rounded to so many bits, but that c be represented to double the precision of a and b.

Various approaches based on fixed-precision arithmetic have been proposed to address non-robustness problems. Schirra [Sch99, p.622] lists 45 papers and reports dated from 1985 to 1997 that address the robustness issue for geometric operations in the piecewise linear domain. The methods proposed each address a specific operation (or closely related operations); the operations include convex hull determination, operations on polygonal objects and line arrangements, and Delaunay and Voronoi diagrams.

Greene and Yao [GY86], Milenkovic [Mil88] and Hobby [Hob99] have put forward methods for the line segment arrangement problem in which the original input data structure is first mapped onto an integer grid. Greene and Yao break the segments into polylines so that the end-points all lie on the grid. Although the rounding can introduce new incidences not present in the original problem, it can be shown that no new crossings are added [Sch99, p.625]. In his method, Milenkovic carries out a pre-process known as *data normalisation* consisting of a number of epsilon-adjustments in which a vertex may be shifted onto another vertex nearby, and an edge can be cracked at a vertex close to its interior. The adjustments ensure that the modified problem is solved precisely when using imprecise arithmetic [Sch99, p.622].

The *representation and model* approach to robustness, first put forward by Hoffmann, Hopcroft and Karasick [HHK88], formalises the idea that an algorithm should compute the correct solution for a related input [Sch99, pp.616-618]. A distinction is made between a *model* of an abstract geometric object (or set of objects) and its *representation* on computer. An algorithm is considered as a mapping from an input representation to an output representation, while the operation it emulates maps an input model to an output model. An algorithm is called robust if for every allowed input representation there exists an input model that 'corresponds' to the input representation such that the output model of the operation 'corresponds' to the output representation of the algorithm. Proving robustness can be non-trivial. This definition of robustness depends on how 'correspondence' between representation and model is defined. Schirra [Sch99, pp.617-618] makes the point that 'proving' the robustness of an algorithm can be made easier by adopting a generous set of conditions for 'correspondence', and notes concerning the following 'robust' algorithms:

- Polygon intersection on the plane, by Hoffmann, Hopcroft and Karasick [HHK88]:

the output representation can be non-simple (i.e. with a self-intersecting 'boundary'), and the edges can be far from those of the model.

- Hidden variable method for calculating arrangement of lines, by Milenkovic [Mil88]: it is possible for the computed topology to be such that it cannot be realised by straight lines.

- Intersection between convex polyhedron and half-space, by Hopcroft and Kahn [HK89]: the computed result can be far from the real intersection polyhedron.

Guibas, Salesin and Stolfi [GSS89, GSS90] take an approach that uses a theoretical framework known as *epsilon geometry*. For this, a predicate takes a real value, rather than true or false: if the predicate condition does not hold, the predicate value is positive, equalling the smallest perturbation required to make the condition hold; if the predicate condition *does* hold, the predicate value is negative, equalling minus the largest perturbation for which it is guaranteed that the condition will still hold. This approach has been used for the convex hull calculation [GSS90], but not for any more complex operation [Sch99, pp.618-619].

The *topology-oriented approach* achieves consistency by ensuring that whenever a numerical computation leads to a decision that violates topology, the decision is replaced by one that conforms to topology; hence, topological consistency takes priority over numerical accuracy. A parsimonious algorithm is automatically topology-oriented, but without it being necessary to force a change in decision. However, a topology-oriented algorithm generally does not usually test directly for a violation of topology. Rather, a set of rules is followed for which it can be shown that the topological properties required will be satisfied [Sch99, p.619]. Sugihara has been, and continues to be, an advocate of the topologically-oriented approach [Sug08]. He and colleagues have put forward methods for Voronoi and Delaunay calculations, the convex hull problem, and also for the intersection of convex polyhedra [SI89, SIII00]. Topology-oriented methods never crash or loop indefinitely, and are guaranteed to produce a result with the correct topological properties. However, they can exhibit bad geometry. For example, the graph structure of the Voronoi diagram generated by the algorithm described in [SI89] will always be planar, but there is no guarantee that the computed vertex positions will give a planar embedding [Sch99, p.619].

In [Hof01, p.17], Hoffmann classes the topology-oriented approach of Sugihara as belonging to a group of algorithms that are based on *symbolic reasoning*. In earlier publications, Hoffmann, together with Hopcroft and Karasick, discussed the use of symbolic reasoning and reported on their own algorithms in this category [HHK88, HHK89], detailing, in particular, one for polyhedral Boolean operations [HHK89].[4] In the later review paper

[4]Although the title of [HHK89] declares the Boolean algorithm to be 'robust', the final section reports

[Hof01, pp.17-20], Hoffmann emphasises the shortcomings of the symbolic reasoning approach. Recall the case of the Boolean operation in which two edges lie close to each other as shown in figure 2.4, where an edge is deemed to intersect one face in its interior but to intersect the neighbouring face at the edge in between (see section 2.3.3, p.32). Hoffmann argues that for such a situation a decision of incidence should take precedence over one of non-incidence, on the grounds that allowing the opposite degrades performance. However, this is difficult to achieve, given that there is no control in the order that incidence tests are applied. The use of symbolic reasoning is also complicated by the existence of various levels of incidence test and the complications of epsilon-geometry.

One inexact approach that has been of interest in recent years is that of controlled perturbation [HS98, RH99, FKMS04, MOS06]. This was first put forward by Halperin and Shelton for the spherical arrangement problem [HS98], and the approach has also been used for arrangements of polyhedral surfaces [RH99] and for Delaunay triangulations [FKMS04]. Controlled perturbation works by processing a perturbed form of the input data carefully chosen to avoid borderline situations in which predicate values are close to zero and therefore susceptible to inconsistent evaluations. This approach is not suitable for certain applications in which it is important for borderline cases to be treated as such.

2.4.2 Exact geometric computation

In recent years there has been a large growth of interest in the use of *exact geometric computation* (*EGC*) as a means of addressing the robustness problem. An implementation of an algorithm in which the arithmetic calculations are carried out exactly is guaranteed to avoid problems of consistency, since it will always construct a result with exactly the correct topology. In fact, correct topology is guaranteed if the predicates are evaluated exactly, irrespective of how the real values concerned are represented. This fact has influenced the progress of recent years. Hence, the issue of how to represent real values and to what precision is driven by the need to evaluate the predicates exactly, and to do so as efficiently as possible. The term *exact geometric computation* is used to denote any approach in which all predicates are evaluated exactly [Sch98, Yap04].

Although the robustness issue is the primary driving force behind the growing interest in exact geometric techniques, an important additional benefit that has emerged in consequence is a range of library software that provides the basic tools that facilitate EGC in general. Most crucially, they provide the means for representing values exactly and for comparing them exactly, thereby making it relatively straightforward for a programmer

a small ‘critical region of failure’. Hoffmann’s classification of algorithms differs from Schirra’s in [Sch99]; hence, Schirra categorises [HHK88] under the representation and model approach, which he considers to be separate from the topology-oriented approach.

to implement a particular algorithm. The Core library [KLPY99, COR08] provides such tools, as do the kernel libraries that form the basis of LEDA [MN99, LED08] and CGAL [WFZ07, FT07, FGK+96, CGA08]; the full libraries of LEDA and CGAL also provide additional support for a wide range of data structures and algorithms.

By the convention established by Yap [Yap97], a representation scheme for set $S \subset \mathbb{R}$ is classed as exact if it allows an exact comparison between any two representable numbers. This is equivalent to being able to determine the sign of a number, if S is complete to subtraction, i.e. $a - b \in S$ for all $a, b \in S$. This general definition does not specify any details of the representation, but simply reflects what is required to allow an effective EGC implementation.

Different problems need to represent different subsets of the reals, depending on what arithmetic operations need to be represented. Many geometric problems, including those in the piecewise linear domain, deal with real values that are defined in terms of other real values using only the arithmetic operators: $+$, $-$, $\times$, $\div$. Such problems are classed as *rational*. If the expressions are constructed from rational values (or alternatively, from integers), the values they represent are all rational. In principle, it is possible to implement the problem by representing every value involved in rational form using pairs of integer values, indicating the numerator and denominator. All four arithmetic operations and the comparison operation can then be implemented using just integer addition, subtraction and multiplication. The magnitude of the integer values can be very large—beyond what is provided by hardware integer arithmetic—but arbitrary-precision or multiple-precision integer arithmetic can be used instead. A number of packages provide support for large integer arithmetic. LEDA [LED08], for example, provides a number type for arbitrary-precision integers. However, this approach to solving rational problems, though simple, generally requires high numerical precision, which can slow down the computation substantially. For example, the use of this approach for the 2D Delaunay problem can lead to a slow-down factor as large as 10 000 compared to a floating-point implementation [Sch99, pp.607-609].

Efficiency can be improved by the technique of *adaptive evaluation*, also known as *lazy evaluation*. Adaptive evaluation takes advantage of the speed of floating-point arithmetic by making use of it whenever possible. Although floating-point arithmetic can lead to incorrect predicate values, most times it will be correct. For adaptive evaluation to work an additional calculation is required to determine if the floating-point calculation can be relied on. The simplest way to do this is by means of a *floating-point filter*. For all calculations the program computes a value that determines a strict bound on the extent of any numerical error, based on the known precision of individual arithmetic operations. When it is necessary to determine a predicate, the computed value on which it is based is compared to the computed error bound. If the magnitude of the former exceeds the latter, the sign of the computed value can be accepted as the sign of the true value; if not, further calculations are required. Different types of filter are available; thus, one type of filter may achieve a tighter error bound than another, if it is based on a more elaborate formula, but in consequence it has a higher computation cost. A *static filter*, such as the FvW filter of Fortune and van Wyk [FvW96], is efficient because it makes use of information known at compile time; in contrast, a *dynamic filter*, such as the BFS filter of Burnikel, Funke and Seel [BFS98], uses run time information and so is less efficient, but gives a tighter bound [Yap04, p.939]. Also, it is possible to obtain a tighter error bound by using a higher precision floating-point representation of real values for the main calculation. A series of filters may be applied, starting with one that is coarse but efficient, with each successive filter being tighter; it is only when none of the filters gives a definite result that exact rational arithmetic is used to determine the correct decision [Sch99, p.610]. For more information on filters see [MY04, L.10].

One form of dynamic filtering is the use of interval arithmetic, in which upper and lower bounds are maintained. When the bounds for a value are computed, the upper bound must be rounded up, while the lower bound must be rounded down. This requires a switch in the rounding mode of IEEE arithmetic from 'round to nearest' (the default) to (say) 'round to $+\infty$'. The IEEE standard provides both 'round to $+\infty$' and 'round to $-\infty$' options, but it is inefficient to switch between these two rounding modes continually; rather, it is preferable to round to $-\infty$ in effect by computing *minus* the value required [Sch99, p.612].

A very general way of representing values exactly is as a symbolic expression. Taking this approach provides (potentially) a means of handling a wide range of problems—not just rational problems. There is no difficulty in representing values this way, so long as every value concerned is either a well-defined constant or the result of a well-defined arithmetic operation performed on previously defined values. Each expression can be represented as a labelled binary tree, known as an *expression tree*; alternatively, it can be a labelled rooted DAG (directed acyclic graph), known as an *expression DAG*, which allows subexpressions to be multiply referenced. The application code executes the algorithm in the manner required for a hypothetical real RAM machine, specifying the expressions that define each value concerned, and making branching decisions in accordance with the signs of certain of the computed values. When a value is defined, the lower-level library code that handles values only needs to manipulate data structures; no arithmetic calculations are necessary. It is only when an application queries the sign of a value that any arithmetic manipulation is required.

The feasibility of the EGC approach for a particular group of expressions depends ultimately on whether it is always possible to determine the sign of an expression from that group [LPY04, p.91]. The value of an expression can be arbitrarily close to zero, having been defined (say) as the difference between two large values; an expression may even evaluate to zero, as in the case of $\sqrt{3} - \sqrt{2} - \sqrt{5 - \sqrt{24}}$. If an expression represents a non-zero value, it is possible to determine its sign by evaluating it approximately to within a sufficiently high precision. However, it is also necessary to ascertain when an expression evaluates to zero. An approximate evaluation, however precise, is not sufficient on its own to determine that. Fortunately, it is possible to determine zero values for certain classes of problem, provided it is possible to determine a value known as a *zero bound*. The value $b > 0$ is known as a *zero bound* (or *root bound*) for an expression E if it is known that one of the following states is true (but not necessarily which): that (1) E is invalid, (2) $E = 0$, or (3) $|E| > b$.[5] When a zero bound for valid expression E is known to be b, then in order to determine the sign it is sufficient to evaluate E approximately to within an absolute precision of $\frac{1}{2}b$. If E is evaluated approximately as $\tilde{E}$ such that $|E - \tilde{E}| < \frac{1}{2}b$, then $|\tilde{E}| < \frac{1}{2}b$ implies that $E = 0$; otherwise, the sign of E is the sign of $\tilde{E}$ [LPY04, p.91].

The challenge within EGC is to determine a 'constructive zero bound', that is to say, a zero bound that can be computed for any expression from the class $\mathcal{E}$ of permitted expressions. Ideally, the bound should be as large as possible and easily computable. The feasibility of finding a constructive zero bound depends on what expressions are permitted. It is usual to classify a problem by the set of operators permitted to define an expression. The set of permitted initial values is included in the operator set—these are considered to be operators with an arity (operand count) of 0. Given an operator set Ω, $\mathcal{E}(\Omega)$ designates the set of all expressions that can be represented using operators belonging to Ω. Hence, for rational problems, the operator set is $\Omega_1 = \{+, -, \times, \div\} \cup \mathbb{Z}$. Problems covered by the operator set $\Omega_2 = \Omega_1 \cup \{\sqrt{\cdot}\}$ are classed as constructible; a variant of this is $\Omega_2^+ = \Omega_1 \cup \{\sqrt[n]{\cdot}\}$ for integer $n \geq 2$ [LPY04, pp.91-92]. A number of constructive zero bounds have been established for expressions from $\mathcal{E}(\Omega_2^+)$ (and therefore also $\mathcal{E}(\Omega_2)$ and $\mathcal{E}(\Omega_1)$, which are subsets). [LPY04, pp.92-96] gives details for a number of constructive zero bound functions. The constructive zero bound functions are all constructed in a similar manner. For any given expression $E \in \mathcal{E}(\Omega)$, the constructive zero bound $B(E) > 0$ is obtained by a specific formula applied to a set of parameters associated with E: $B(E) = \beta(u_1(E), ..., u_m(E))$. The parameters $u_1(E), ..., u_m(E)$ are obtained by a series of recursive formulae for each arithmetic operator in Ω, based on the equivalent parameters for each operand expression.

Constructive zero bounds have not been found for more complex operator sets that include non-algebraic operations such as $\exp(\cdot)$ or $\ln(\cdot)$ or the trigonometric functions. Consequently, EGC techniques are not considered feasible at present for problems requiring these operators [LPY04, p.92].

[5]Symbol E primarily denotes an expression, but depending on context it can also denote the numerical value represented by the expression.

Li, Pion and Yap [LPY04, pp.90-91] list a number of challenges for development of the EGC approach:

- One particular challenge relates to serial operations, such as the generation of a boundary mesh from a CSG representation of a solid. Each operation in the sequence of operations leads to a higher complexity in the value representations, and hence to longer execution times. One way round this problem is to apply the process of snap rounding [Hob99] to the data between operations, so that the values take on a lower precision. However, since snap rounding modifies the numerical data, this classes it as an approximate method, and as such it can lead to a mismatch between the numerical and topological data, which can be manifested, for example, as a boundary self-intersection.
- A particularly significant challenge is to be able to handle high degree algebraic computations, such as is required by CAD applications (c.f. the ACIS library [CL01]).
- Even for simple problems of limited depth there are special cases for which the computations can slow down to a halt. One example is to determine the Voronoi diagram for a set of points lying on a circle. In this instance the filtering mechanism ceases to be effective.
- As has already been noted, for non-algebraic problems there is no known computable zero bound that can be used to ascertain that an expression has value zero.
- There is at present no computation model for EGC that is suitable for analysing the complexity of a problem.

So, although there have been major advances using the exact approach to resolve the robustness problem that would have seemed barely possible little more than a decade ago, certain issues remain unresolved. It is therefore reasonable to continue investigations into how the approximate approach can help. We note in particular that the approach taken in practice to avoid the problem of increasing number complexity for serial operations is to apply an approximate method, namely snap rounding. Therefore, in order to make this particular approach provably fully robust, we depend on being able to find a robust algorithm for the approximate process of snap rounding.

2.5 Degeneracy

Finally in this section we consider the issue of degeneracy, particularly in the context of exact computations.

The term *degeneracy* is commonly used to describe the situation when a data structure is borderline between being in one state or another [For96, Sch99, Hof01, Yap04]. One example is the union operation between two polyhedral shapes with abutting faces; for this, an adjustment to the data by an arbitrary small amount one way or the other ensures either that the objects overlap or are separated. In strict terms, an input data structure for a particular algorithm is classed as degenerate if the algorithm responds differently for some arbitrarily close variant of the input data. It is usually appropriate to define degeneracy in terms of low-level data, for example when there are three collinear points, four cocircular points, or four coplanar points (in 3D).

In section 2.3 we noted the severe problems of implementing an algorithm using approximate arithmetic because of the inconsistencies that can arise when there is degeneracy or near-degeneracy in the data. Those particular problems do not arise in an exact implementation. In principle, at least, the rules for each of the possible special cases can be derived from first geometric principles. Nevertheless, the special handling required for degenerate cases can potentially complicate the implementation because of the number of special cases to consider.

One way of avoiding having to consider all the special cases is by using *symbolic perturbation*. This approach was first put forward for geometric problems by Edelsbrunner and Mücke [EM90], and subsequently refined by Yap [Yap90] and Emiris and Canny [EC91, EC92]. Let us suppose that for a particular input topology, the numerical input is represented by vector $\mathbf{x}$. One can assume that a series of perturbations can be made to the data to resolve any degeneracies. The first perturbation is scaled by a notional value $\varepsilon > 0$, the second is scaled by ε^2, and so on; hence the original input $\mathbf{x}$ is substituted by a vector polynomial in ε, $\mathbf{x}(\varepsilon)$, which tends to $\mathbf{x}$ as $\varepsilon \to 0$. The solution to the problem is then sought for $\mathbf{x}(\varepsilon)$ as $\varepsilon \to 0$. Consequently, each predicate is determined not by a single real value, but by a real polynomial in ε. Provided sufficiently many perturbations are applied to the numerical input, the value on which the predicate is based is judged to be strictly positive or negative in accordance with the sign of the first non-zero term in the polynomial.

The use of symbolic perturbation is a point of controversy within the computational geometry literature. The fact that the generated result is not necessarily the true result may make it inappropriate for certain applications. For example, in the case of Boolean operations, the application may require that the regularised form of the operation be applied for degenerate cases. Furthermore, Burnikel et al. [BMS94] maintain that for many problems the special handling required for degenerate cases is not particularly complex, and also point out that the use of symbolic perturbation can adversely affect efficiency [For96] [Sch99, pp.623-624].

This completes the coverage of the background to the robustness problem of geometric algorithms. In the next chapter I cover in broad terms what I consider to be the requirements of a robust method.

Chapter 3

Considerations in the Boolean algorithm design

We have noted (see section 2.3) that the consistency we require for the Boolean operation is most unlikely to be achieved by an *ad hoc* implementation that resorts to special case handling when entities are near-coincident. Such a strategy merely increases the complexity of the process, and prevents the process from being sufficiently well understood to resolve matters. The aim in my work has therefore been to devise a method based on as simple an understanding of the operation as is practical. As Fortune stated at the 1998 SIGGRAPH panel discussion on geometric robustness [WDF$^+$98]: "An ideal solution to the problem of numerical robustness would be simple, efficient, and widely applicable." However, he went on to say "No such solution exists or is likely to exist."

In this chapter, I first discuss in general terms the criteria that may be considered important for judging whether an algorithm is robust. I then discuss issues that are specific to boundary mesh Boolean operations, and the particular criteria for robustness I consider within this dissertation.

3.1 General requirements for robustness

We noted in section 2.4.1 the lack of consensus as to what makes an approximate algorithm robust. I list here the conditions that I broadly consider appropriate for classing an algorithm as robust. These conditions are selected with the aim of making the algorithm suitable for the user; as such they do not necessarily agree with the definition of 'robust' given in section 2.4.1, p.37, that is traditionally used in numerical analysis.

An absolute requirement is that for any given 'valid' input, the algorithm generates a 'valid' result. Hence:

1. there is a clear path for the algorithm to follow; that is to say, the algorithm must involve no operation for which there is no satisfactory result (for example, division by zero);
2. the algorithm terminates to give a result;
3. the result is 'valid'.

In addition, there are requirements for accuracy and performance:

4. the result is 'accurate'[1];
5. the algorithm terminates in 'reasonable' time;
6. the amount of memory required by the algorithm is 'reasonable'.

These criteria are selected because they are essentially those required for any piece of software to work. It is not a rigid definition, hence the inverted commas for certain words. Nor is it to be considered necessarily a complete list, as there may be other demands on an algorithm for particular applications. For example, it may be desired that an operation be invariant to transformation.[2] This criterion is unlikely to be achieved with approximate arithmetic, and is not considered any further.

The first three criteria are required for the algorithm to produce a valid result from valid input. The question remains as to what exactly the word 'valid' should be taken to mean, both for input and result data. Designing a robust algorithm increases in difficulty in accordance with how tight a definition for validity is used for the result data, and conversely, how broad a definition is used for the input data. For an operation used in a specific context, the definition of validity for input must suit the available input data, and likewise, the definition of validity for the result must suit the purposes for which it is used. When a computation involves a number of operations, so that the result of one operation acts as input for other operations, the definition of validity for the result of that operation must satisfy the definitions of validity for the input of the operations that process the result. For example, if an operation generates a volume mesh to be used for finite element stress analysis calculations or for fluid flow calculations, this imposes a constraint on the

[1]By the convention of numerical analysis (see [Sch99, p.600]) an algorithm failing to satisfy this criterion may be classed as 'robust' but 'unstable'.

[2]This particular criterion was suggested by Alan Middleditch (December 2006). A computed Boolean operation, $\widehat{\oplus}$, for example, is invariant to transformation if $\widehat{T}(A)\widehat{\oplus}\widehat{T}(B) = \widehat{T}(A\widehat{\oplus}B)$ for any permitted set representations A and B and for any permitted affine transform operation, $\widehat{T}$, such as displacement, rotation and scaling. So the topological structure of the result of an operation will not change if both input structures are identically transformed.

generated mesh that is at least as strong as the minimal constraints that apply to the input for the latter computation; one would also ideally want to impose conditions on the generated mesh to suit the efficiency and stability of the latter computation. It is limiting to design an algorithm to work solely for a specific type of preceding operation and/or a specific type of successive operation; the broader the range of operations allowed before and after the operation concerned, the more generic the algorithm. One point to note is that if a sequence of operations all rely on the same type of data structure both for input and the result, it may be appropriate for the validity conditions of both the input and result to be matched by selecting the same set of conditions throughout.

For an algorithm to progress satisfactorily, generally speaking it is necessary for each lower-level arithmetic and logical operation within the computation to progress to completion to generate a result.[3] Particular concerns for arithmetic operations are that there should be no division by zero, and overflow should not occur. Logical operations specified within the operation concerned must also always complete successfully. For example, if an algorithm relies on selecting an item from a list of items, the list must not be empty; another example is that if the items from two lists are to be paired, so that each item from either list is uniquely paired with an item from the other list, the two lists must have the same number of items.[4]

The accuracy constraint (4) is essential. Were that not so, an algorithm could be classed robust by always producing the same (valid) result irrespective of input data; for example, a Boolean operation algorithm could always return the empty set as the result. The accuracy constraint requires the result to be sufficiently 'close' in some sense to the true solution. How to define 'close' is an open issue that depends on the requirements of the downstream operation. For example, for an algorithm that generates a mesh to approximate a curved surface, if the mesh is required as input for a subsequent shading operation with specular lighting, it may be considered important for the facet normals (and not just the vertex positions) to lie within specified tolerance.

The constraints not to use excessive resources (5 & 6) are also an issue. The ultimate concern for someone implementing an algorithm is that the program will run out of memory, or that the elapsed time will be unacceptably long. For operations where there is no limit on problem size, it is of course inevitable that resource limitations will affect problems above a certain size, irrespective of the amount of available memory and the clock speed of

[3]In strict terms it is not an essential requirement, because an algorithm can take an alternative course of action when a low-order operation fails. However, any course of action for which there is no alternative has to be robust in its own right in order for the method as a whole to be robust. So, for example, for exact geometric computation using filtered exact arithmetic, the filtered arithmetic may 'fail' with 'result not known', but the process as a whole will be robust since the exact arithmetic is always able to generate a result.

[4]These two particular conditions are relevant to the basic Boolean algorithm I go on to describe.

the CPU. Without doubt, certain cases will fail. However, it is not meaningful to declare all algorithms for the operation non-robust because of this. Instead, the relevant question is whether an algorithm 'scales' well, so that if a problem is scaled up by a certain factor, the scale-up of resources required to solve the problem is considered acceptable. Hence, the assumption can be made at a theoretical level that the program is implemented on a machine with unlimited resources. The issue of scalability is traditionally the concern of algorithm analysis and design, as described in, for example, [AH74]; the discipline of computational geometry applies algorithm analysis specifically to geometric algorithms [PS85, pp.6-17]. The cost of an algorithm, both in terms of computation time and the amount of memory required, is measured in accordance with a particular *model of computation* that specifies the costs of individual primitive operations, and likewise the memory required for each component of the intermediate data structure used by the algorithm. The *random access machine model* or *RAM model*, is often the chosen computation model [AH74, 5-11]; this extends naturally to the *real RAM model*, in which real values can be held, [PS85, pp.26-28]. A measure, N, denotes the size of a particular problem in terms of the input data, allowing the time and space complexities of the algorithm to be described using big-O notation.[5]

Although the order of complexity of an algorithm is an aid to understanding its efficiency, it does not necessarily reflect the efficiency of a program that implements the algorithm, because the assumptions made in the model of computation may be wrong. The RAM model is a reasonable approximation for an implementation of an algorithm based on machine arithmetic (up until the memory requirements make it necessary to page data to virtual memory), but not for an implementation based on the exact arithmetic paradigm for which the cost of an arithmetic operation increases with the complexity of the values concerned.

One can argue as to what order of time complexity is acceptable, and whether the difference between an $O(N^2)$ and an $O(N \log N)$ algorithm, say, is sufficiently significant to declare an $O(N^2)$ algorithm non-robust. A more serious concern is that of an algorithm having complexity greater than polynomial order (i.e. not $O(N^k)$, for any positive k), or that fails to terminate at all for certain problems. An algorithm based on an iterative process is potentially a concern in this respect, because the complexity of the intermediate data can slow down progress; indeed, it may even become trapped in an infinite loop, breaching constraint (2). These potential risks place an onus on the algorithm designer to engineer the algorithm so that it is guaranteed to terminate in an acceptable order of time. A similar argument applies to the memory usage constraint (6).

[5]To say an algorithm has time (or space) complexity $O(f(N))$ denotes that there exist positive constants C and N_0 such that for all problem cases P for which $N(P) \geq N_0$, the time (or maximum memory) required to execute the algorithm is no greater than $Cf(N(P))$ [PS85, p.9].

3.2 Requirements for robust Boolean operations

Having discussed the requirements for robustness in general terms, I now consider specific issues relating to Boolean operations on boundary mesh representations, in which the positioning of the mesh is defined in terms of the positions of the vertices, and the computations are performed using approximate arithmetic.

The types of validity considered are topological validity and geometric validity. These are discussed in more detail in the next chapter. In short, a shape representation is considered topologically valid if the components of its boundary mesh are fully connected, making it watertight. The boundary mesh is stored in such a way to be considered oriented, so that each component has an ‘interior’ and ‘exterior’ side directly discernable from the data (details given later). A topologically valid shape representation is also considered geometrically valid if for every boundary mesh component the ‘interior’ side truly represents the shape interior, and the ‘exterior’ side likewise the shape exterior. Examples of valid and invalid representations in 2D are shown in figures 4.1 to 4.3, p.57.

For the methods presented in this dissertation, topological validity of the input is assumed always to hold. To allow topologically invalid shape representations would complicate matters considerably, because it is unclear in such situations how to define the shape represented. The algorithms for Boolean operations presented in this dissertation are *topologically robust*, meaning that the result generated is guaranteed to be topologically valid provided the input shape representations are topologically valid. We can therefore be sure that serial operations will maintain topological validity (assuming all original shape representations are topologically valid).

Maintaining or re-establishing geometric validity, however, is more complicated. We noted in section 2.3.5, p.35, (see figure 2.6, p.36) that transformation operations can make a shape representation geometrically invalid. Worse still, any operation designed to resolve geometrically invalid data, or to prevent the emergence of geometric invalidity in subsequent operations, can itself create the type of geometric invalidity that the operation is trying to avoid. This tendency for geometric errors to arise makes it desirable for algorithms to be able to handle geometrically invalid input data; an algorithm should be able, at the least, to tolerate geometrically invalid input, or even better, resolve it. These matters are discussed later in the context of the algorithms I go on to describe. In particular, section 4.7 describes how the basic Boolean algorithm (described in the next chapter) can be made tolerant to geometric errors, while chapter 6 describes how the simplification algorithm is designed to resolve geometric errors.

We now consider the basic Boolean algorithm in detail.

Chapter 4

The basic Boolean algorithm

This chapter describes in detail the algorithm I devised known as the *basic Boolean algorithm*. The algorithm operates on to two boundary structures, each representing a piecewise linear shape—a polyhedral shape in 3D or a polygonal shape in 2D. If this algorithm were to be implemented in exact arithmetic, then for any fully valid input in general position it would compute the true solution of the specified problem; for degenerate cases it computes the true solution of a symbolically perturbed version of the specified problem. What sets this algorithm apart from other approaches is the fact that an implementation based on approximate arithmetic will always compute a boundary structure, provided both input shape representations are topologically valid; furthermore, the shape representation generated by the algorithm is provably guaranteed to be topologically valid. However, the generated solution might have certain glitches that make it geometrically invalid, or in a marginal state between being geometrically valid and invalid. It is therefore pragmatic to apply a data smoothing post-process to remove these glitches.

The algorithm is discussed mostly in terms of the 3D operation; details relating to the 2D operation are omitted when they are straightforward to deduce from the description of the 3D operation. Certain diagrams relate to the 2D operation.

Section 4.1 gives a brief overview of the method, and discusses the nature of the glitches that need to be smoothed out. Section 4.2 discusses the boundary structure in abstract terms and the conditions for validity; section 4.3 outlines the overall approach, pointing out the similarity between each of the different types of operation involved and their relation to each other. Sections 4.4 and 4.5 explain in detail the operations for determining intersection relationships, the latter section giving specific formulae. Section 4.6 describes the operations for constructing the result of the operation. Section 4.7 discusses how the algorithm can be extended so that it is able to function when geometric errors are present, while section 4.8 touches briefly on the requirements for the triangulation of facets—a process necessary for one variant of the algorithm. Finally, section 4.9 gives mathematical proofs for the topological robustness of the algorithm.

4.1 Overview

The *basic Boolean algorithm* is based on a series of interrelated operations designed to guarantee the generation of a result with correct connectivity, provided the input structures have correct connectivity, but irrespective of the extent of numerical errors in the computations and the input data. Hence the algorithm can be categorised as topology-oriented, as defined in [SIII00]. The operations at the heart of the basic Boolean algorithm are tests that determine, in the 3D case, whether a vertex of one input structure lies in the interior of the other input structure, and whether an edge of one structure intersects a facet of the other. These tests are based on the results of equivalent 2D tests in which the structures are assumed projected onto a plane, the tests being whether a vertex lies in a polygonal region (as projected by a facet), and whether two projected edges intersect. In turn, the 2D tests are based on 1D point-in-interval tests in which vertices and edges are assumed projected onto a line. The result of an individual test between two entities relies only on the data for the entities concerned. This contrasts with many methods for which neighbouring information is also required—see, for example, [FCM87], in which edge-edge test results depend on the orientation of neighbouring facets. Projection methods have been proposed before, by Kalay for point-in-polyhedron testing [Kal82], and Gardan and Perrin for Boolean operations [GP96], and the 2D point-in-polygon-region test used is essentially identical to the winding number method described by Haines [Hai89]. The method is an advance because of the guarantee of correct connectivity in the result of the Boolean operation. There is a consistent pattern of dependency between the operations, from the lower-level relationship tests to the higher-level operations that construct the result. I prove that the operations at each level can be performed (without forcing data) to produce a result satisfying the connectivity constraints, irrespective of the input geometry.

The shapes concerned in the 3D algorithm can be represented as a triangle mesh or a general polygon mesh. The result of the operation is naturally a general polygon mesh, regardless of the type of mesh used to represent the input. Hence, if triangle meshes are used, it is necessary to break up the non-triangular facets of the generated result into triangles, if the result is required for subsequent Boolean operations. The polygon mesh variant of the algorithm is more efficient, but there are theoretical concerns because of the ambiguity as to where the boundary surface lies when the vertices of a polygon are not exactly coplanar. However, the algorithm is always able to proceed without difficulty, despite the ambiguity.

The basic algorithm does not necessarily avoid problems of geometrical correctness. In borderline cases in which boundary components coincide, or nearly so, the result may have gaps and slivers, as shown in figure 2.2, p.26. If the computations were to be carried out exactly, any such gaps and slivers could have arbitrarily small positive thickness, or even zero thickness. The use of approximate arithmetic, however, can perturb the result boundary sufficiently to lead to self-intersections. Even cases that do not lead immediately to a boundary self-intersection can be problematic, because boundary self-intersection can arise from perturbations associated with subsequent operations (such as a shift or rotation or another Boolean operation). I use the term *marginal* loosely to describe data structures such as these that are liable to lead to difficulties: either the structure has become geometrically invalid, due to a small perturbation arising from numerical errors in the computations so far, or else it is liable to lead to geometric invalidity in some structure generated from it, due to numerical errors in subsequent computations. There is also an issue concerning redundant features that may appear in the structure generated by the basic algorithm: coincident vertices, zero-length edges and zero-area facets. Generally, such features are considered undesirable, not least because they introduce unnecessary complexity and inefficiency to the structure.

Since the structure generated by the basic Boolean algorithm can be marginal and also contain redundant features, it is generally preferable to apply some form of data-smoothing post-process that has the effect of 'cleaning up' the structure: removing gaps and slivers and other undesirable features that have a length-scale below a certain size. Ideally, such an operation resolves the marginal nature of the data structure, making it geometrically valid, and also removes redundant features. The data-smoothing operation is covered briefly in chapter 5 and appendix A.

Note that for many applications, the strategy of data smoothing is also likely to be in accord with the requirements of end-users (whether or not they are declared as such). In CAD applications, for example, it is quite common to require surfaces to coincide. A user may very well require that objects assembled within a model connect up; or they may choose to define the complex shape of a single object in terms of a difference operation, with the second shape defining a hole that is to be cut away from the first shape. Both these scenarios may lead to coincident surfaces, at least as the user intends. However, the precise positioning of each surface is determined by a series of approximate arithmetic calculations performed outside the modelling library; hence the exact solution to the problem, as specified through the application interface, is likely to have slivers or gaps the user does not want.

4.2 Topological and geometric validity

Before describing the basic algorithm in detail it is necessary to consider the essential aspects of the shape data structure and the conditions that make it a valid shape repre-

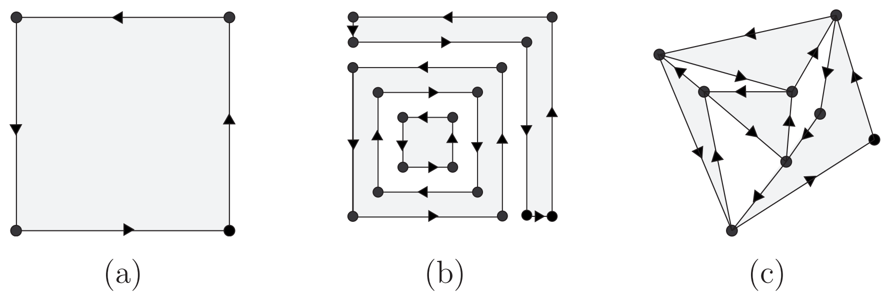


Figure 4.1: *Examples of valid shape representations in 2D space. The edges are directional, with the convention that the shape interior lies to the left.*

sentation. This section describes the shape boundary model in abstract terms to explain the issue of validity—topological validity in particular—rather than as an indication how to implement the model. (For implementable boundary representation data structures see [Bau72, FvDF96].)

There are two categories of constraint that a polygonal or polyhedral data structure needs to adhere to for it to be valid—topological and geometric. These are alternatively known as combinatorial and metric constraints [Req80]. The appendix to [For97] demonstrates one way (different to mine) of specifying the topological constraints for a 'combinatorial polyhedron'.

The topological constraints relate to the connectivity of the boundary components. They must connect to each other so that there are no breaks in the boundary surface which is intended to separate the interior of the shape from its exterior. These constraints are independent of any positional or other geometric data.

The geometric constraints ensure that the boundary surface fully separates the interior and exterior of the shape, and that they lie on the correct side (in accordance with the rules for the data structure). In contrast to the topological constraints, the geometric constraints *do* rely on positional data. The boundary of a representation should not enclose any region the wrong way round, nor enclose it twice. Intersecting boundary components are not permitted, as these generally lead to doubly-enclosed or inside-out regions (except for certain contrived counter-examples).

Figure 4.1 shows examples of shape representations that are valid in the 2D domain, in which the interior lies on the left-hand side of boundary edges. In contrast, figure 4.2 shows examples that break the topological constraints, and figure 4.3 shows examples that adhere to the topological constraints but break the geometric constraints.

For the 2D problem we can formalise the rules for a polygonal shape as follows:

- *Polygonal shape* (data definition): A *polygonal shape* is a collection of *vertices* and *edges*.

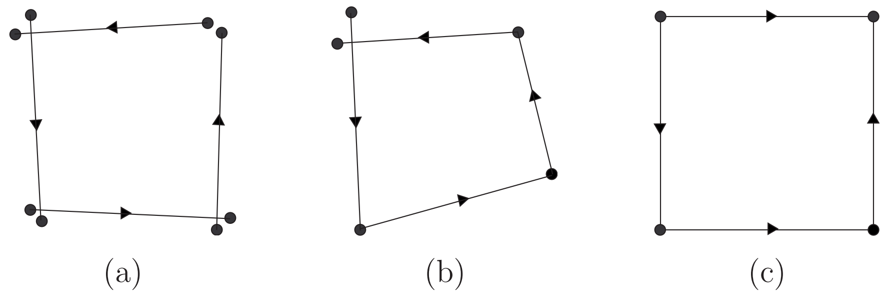


Figure 4.2: *Topologically invalid shape representations. (a) & (b) edges do not all connect; (c) edge directions are inconsistent.*

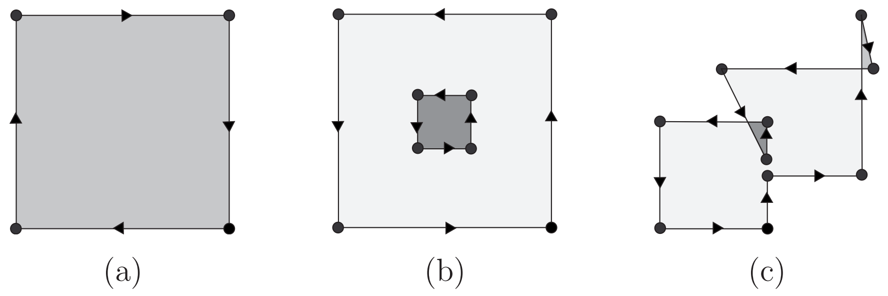


Figure 4.3: *Topologically valid but geometrically invalid shape representations. (a) the inside of the object lies on the wrong side of the edge; (b) the central region is enclosed twice; (c) a doubly-enclosed region exists, and also an inside-out region.*

- *Vertex* (data definition): Each *vertex* has a position in 2D space representing its location.
- *Edge* (data definition): Each *edge* has two links to vertices in the collection, known as the *start-vertex* and *end-vertex*.
- *Shape boundary closure* (topological constraint): If there are n edges for which the end-vertex is vertex P, then there are exactly n edges for which the start-vertex is P.
- *Shape enclosure* (geometric constraint): Each edge separates the interior and exterior of the shape, with the interior to the left and the exterior to the right as one traverses from the start-vertex to the end-vertex.

The topological constraint in effect stipulates that the edges form a number of closed loops (though when vertices act as an end-vertex to more than one edge, it is arbitrary to deem which edge follows on from which). The geometric constraint in effect deems that a loop representing the outer boundary of a contiguous region must go anti-clockwise, and one representing an inner boundary (a hole) must go clockwise. There is no restriction on the connectivity of the shape as a whole, so it is allowed to consist of more than one contiguous region, nor is there any restriction against the shape being non-manifold.

In 3D, the rules for a polyhedral shape represented as a polygon mesh are as follows:

- *Polygon mesh* (data definition): A *polygon mesh* is a collection of *vertices* and *facets*.
- *Vertex* (data definition): Each *vertex* has a position in 3D space representing its location.
- *Facet* (data definition): A *facet* is a collection of *half-edges*.
- *Half-edge* (data definition): Each *half-edge* has two links to vertices in the collection, known as the *start-vertex* and *end-vertex*.
- *Facet boundary closure* (topological constraint): If facet F has n half-edges for which the end-vertex is vertex P, then F has exactly n half-edges for which the start-vertex is P.
- *Shape boundary closure* (topological constraint): If the shape representation (as a whole) has n half-edges for which the start-vertex is vertex P and the end-vertex is vertex Q, then there are exactly n half-edges for which the start-vertex is Q and the end-vertex is P.
- *Planar facet* (geometric constraint): The positions of the end-vertices of the half-edges of a particular facet must all lie on the same plane, known as the *facet plane*.
- *Facet enclosure* (geometric constraint): When viewing a particular facet from one side of its facet plane, to be identified as the *outer side* of the facet, each half-edge separates the interior and exterior of the facet, with the facet interior seen to lie on the left-hand side of each half-edge as one traverses from start-vertex to the end-vertex.
- *Shape enclosure* (geometric constraint): Each facet separates the interior and exterior of the solid, with the exterior lying on the outer side of the facet (as defined above).

Again there is no constraint for the shape to be contiguous or manifold, nor is it required that individual facets should be contiguous or manifold.

The rules for a polyhedral shape represented as a triangle mesh closely resemble those for the polygon mesh:

- *Triangle mesh* (data definition): A *triangle mesh* is a collection of *vertices* and *triangular facets*.

- *Vertex* (data definition): Each *vertex* has a position in 3D space representing its location.

- *Triangular facet* (data definition): A *triangular facet* has three *half-edges*.

- *Half-edge* (data definition): Each *half-edge* has two links to vertices in the collection, known as the *start-vertex* and *end-vertex*.

- *Triangular facet loop* (topological constraint): The half-edges of a triangular facet form a single loop, with the end-vertex of each half-edge being the start-vertex of the next half-edge in the loop.

- *Shape boundary closure* (topological constraint): If the shape representation (as a whole) has n half-edges for which the start-vertex is vertex P and the end-vertex is vertex Q, then there are exactly n half-edges for which the start-vertex is Q and the end-vertex is P.

- *Shape enclosure* (geometric constraint): Each facet separates the interior and exterior of the solid, with the exterior lying on the side of the facet from which the half-edges are viewed to go anti-clockwise.

The geometric constraints listed for the three types of shape representation are descriptive, in contrast to the data definitions and topological constraints given, and no strict definitions of the geometric constraints are presented here.

A shape data representation is said to be *topologically valid* or to have *topological validity* if it adheres to the data definitions and topological constraints. Note that the data definitions and topological constraints have no restrictions on coincidence. Hence a representation can be considered topologically valid even if two vertices have identical positions, or if an edge or half-edge has an identical start-vertex and end-vertex.

For the polygon mesh and triangle mesh it is convenient to refer to an *edge* as a grouping of half-edges for which the start-vertex and end-vertex are the same two vertices (in either order). The two vertices are known as *start-vertex* and *end-vertex* of the edge; when the vertices are distinct, any half-edge with a start-vertex corresponding to the start-vertex of the edge is known as a *forward half-edge* of the edge; if a half-edge has a start-vertex corresponding to the *end*-vertex of the edge it is known as a *backward half-edge*. The shape boundary constraint deems that an edge has an equal number of forward and backward half-edges (if it has distinct vertices).

4.3 Approach taken

The rules for the basic algorithm are designed to achieve topological robustness, and the individual operations are kept simple to achieve this. In particular, symbolic perturbation rules are applied when determining the relationship between two entities. This prevents the need to consider the possibility of entities coinciding. Hence, the algorithm will determine, for example, whether a vertex of one solid is to be considered inside the other solid or outside, but the vertex will never be considered to lie on the surface (even when it does). Similarly, when considering an edge and a facet (one from each solid) the rules deem either that they intersect fully or not at all. This approach, which is in effect a simple example of the technique advocated by Edelsbrunner and Mücke in the context of exact arithmetic calculations [EM90], simplifies the process of constructing the resulting shape. Thus the result boundary consists of the retained parts of the two original boundaries, stitched together (in 3D) by a number of closed intersection curves between the two boundaries.

The basic algorithm for the Boolean operation between two shapes A and B is performed as a series of interdependent operations. There is a similar pattern to each operation and how it depends on lower-level operations: each operation determines how two entities, one from each shape, relate to each other, an *entity* being a vertex, edge (or half-edge), facet, or the entire shape. Hence there are 16 types of operation: one for each combination of the four types of entity. Each operation type is considered as belonging to a particular level from 0 to 6, equal to the sum of the manifold dimensionality values of the two entity types. The result of an operation concerning entities o_A and o_B depends on the results of operations one level below concerning (1) each boundary component of o_A (in turn) and o_B; and (2) o_A, and (in turn) each boundary component of o_B. This general relationship leads to a dependency hierarchy in the form of a double pyramid between the 16 types of operation, as shown in figure 4.4. The functions at levels 0-3 together determine the relationship between a pair of entities, while the operations at levels 3-6 work towards constructing the result.

The operations at level 3 take a pivotal role between the two stages. These determine whether a vertex of one solid lies inside or outside the other solid, and whether an edge of one solid intersects a facet of the other solid. When determining that an edge intersects a facet, the point of intersection is also determined. The information from these calculations enables the construction of the result.

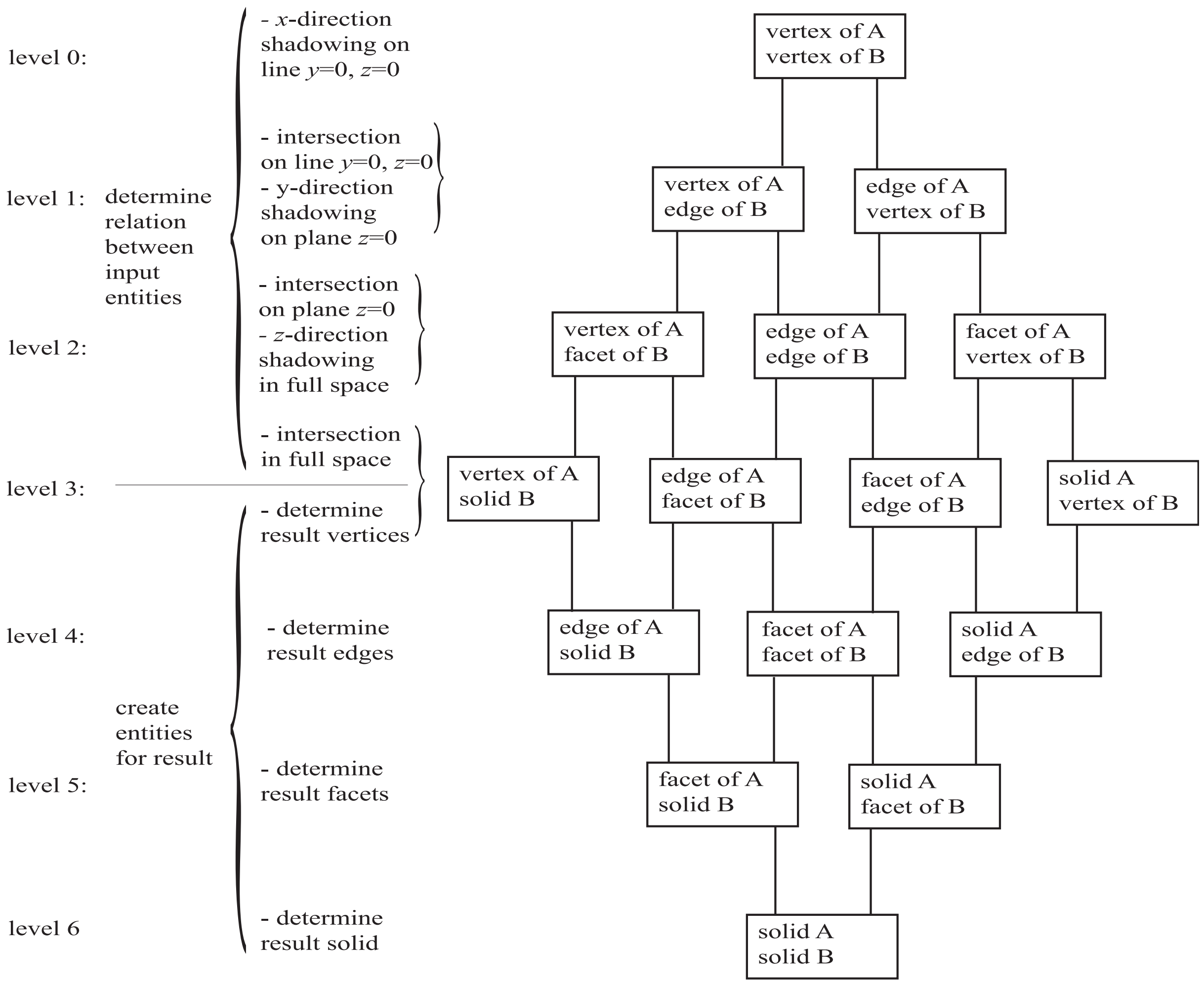


Figure 4.4: *The hierarchy of operations for the basic Boolean algorithm.*

4.4 Determining whether two entities intersect

This section discusses the concept of two entities intersecting, both in full space and in subspace, and how the intersection status between two entities is derived from lower-level intersection calculations. For the sake of consistent terminology we refer to a shape and vertex as ‘intersecting’ when in common parlance one would say the shape ‘contains’ the vertex.

We have already noted that the construction of the result depends on the two types of decisions made at level 3: whether an edge (or half-edge) intersects a facet, and whether a solid ‘intersects’ (contains) a vertex. There are also lower-level intersection relationships between entities. These operate in lower-dimensional space based on the initial coordinates of the Cartesian representation of a point, (x, y, z) or $(\xi^{(1)}, \xi^{(2)}, \xi^{(3)})$. At level 2 the relationship between two entities in 2D space is considered in terms of their x- and y-values and ignoring z. Hence we determine in 2D (x, y) space whether two edges intersect, and likewise whether a facet 'intersects' a vertex. Similarly, at level 1 only the x-coordinate value is considered to determine whether an edge 'intersects' a vertex in 1D space. This idea extends to level 0 and '0D space' at which all coordinates are ignored in the 'test' for intersection; every vertex is considered located at the same point, so any two vertices are considered to 'intersect' at this level.

The intersection status between two entities at level 1, 2 or 3 is determined by considering the intersection status between entities one level below. Consider firstly the task of determining whether two edges intersect in 2D (x, y) space. The initial stage of this task is to note whether the edges overlap when projected onto the line $y = 0$. This is carried out by considering each vertex of each edge in turn and finding out if it 'intersects' the other edge in 1D (x) space. If there are no such vertices, the edges do not overlap, and there can be no intersection between the two edges in 2D space. Ignoring the degenerate cases (which we can do because of the application of the symbolic perturbation rules), the other possible situation is that two of the four vertices are considered to intersect the other edge. The two vertices mark the extremities of the interval over which the two edges overlap in 1D space. In this situation, whether the two edges intersect in 2D space depends on the y coordinate values of the two entities at each of the two extremities. Ignoring again the degenerate cases, there is an intersection if and only if the entity from shape A is above that from shape B at one extremity (in terms of y coordinate values) and below at the other. See figure 4.5 for examples. A similar approach is used to determine if an edge intersects a facet in 3D (x, y, z) space. When the edge and facet are projected onto the plane $z = 0$ they can overlap over a number of segments in 2D (x, y) space. The extreme points of any such segments are identified as 2D intersection points, either where the facet intersects one of the vertices of the edge, or where the specified edge intersects one of the bounding edges of the facet. Whether the edge and facet intersect in 3D is determined by considering the z coordinate values of the edge and facet at the extremities of all the overlapping segments (see figure 4.6).

The decision whether a vertex intersects a facet in 2D (x, y) space is based on inspecting the edges that intersect the vertex in 1D (x) space, of which an equal number go from left to right and from right to left in terms of x coordinate values (see figure 4.7). The vertex and facet intersect when there is a mismatch in the numbers of the two types of edges that cross *above* the vertex in terms of y coordinate values. A similar approach is used for determining whether a vertex intersects a solid in 3D space. There will be an equal number of upwards and downwards facing facets that intersect the vertex in 2D space.

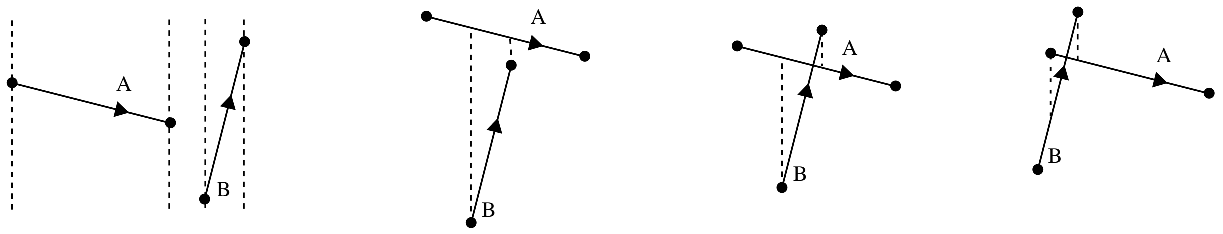


Figure 4.5: *Examples showing how an intersection between two edges in (x, y) space is determined. The two edges intersect only if they overlap in (x) space ($\max x_B \geq \min x_A$ and $\min x_B < \max x_A$) and if $y_B \geq y_A$ at one end of the overlap range in (x) and $y_B < y_A$ at the other (i.e. at $x = \max(\min x_A, \min x_B)$ and $x = \min(\max x_A, \max x_B)$).*

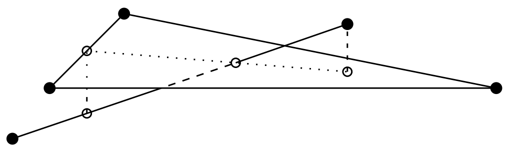


Figure 4.6: *Example showing how an intersection between an edge and a triangular facet is determined in full 3D space. The entities overlap over a segment in (x, y) space; the facet 'shadows' the edge (has a larger z-component) at one end, and the edge shadows the facet at the other.*

The vertex and solid intersect when there is a mismatch in the numbers of the two types of facets that lie *above* the vertex (in terms of z). In 1D (x) space, an edge intersects a vertex if the vertices of the edge lie on either side of the specified vertex (in terms of x values).

An important part of the processing is to determine the point where two entities intersect. The position of a level 3 intersection point between an edge and facet, is the position of the intersection vertex used to construct the result. Positions of lower-level intersections between entities are required to determine the intersection status between entities at the next level up. Although the intersection status between two entities is determined in subspace, it is not sufficient to determine the intersection point in subspace; the subsequent operations need information on where that point lies on each of the two entities in full 3D space. Thus the intersection is represented as two points in full space. For the 2D calculations, the two points have identical x- and y-coordinates, and the z-coordinate values generally differ; for the 1D calculations, the points have identical x-coordinates, and the y- and z-coordinate values each differ, generally. The point(s) of intersection between two entities at any level is derived from two pairs of intersection points determined at the level below by means of linear interpolation. The details of how this is computed are given later.

Figure 4.8 gives an example of how the algorithm determines the intersection point be-

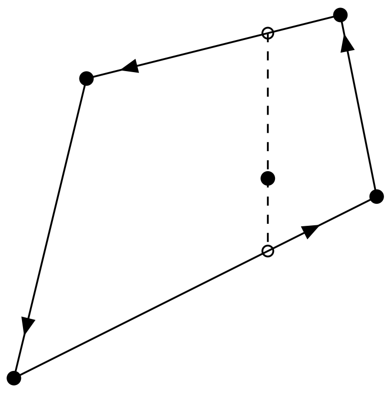


Figure 4.7: *Example showing how an intersection between a vertex and a facet is determined in 2D (x, y) space. The vertex is shadowed in y by a bounding half-edge of the facet going right to left, and shadows another half-edge going left to right.*

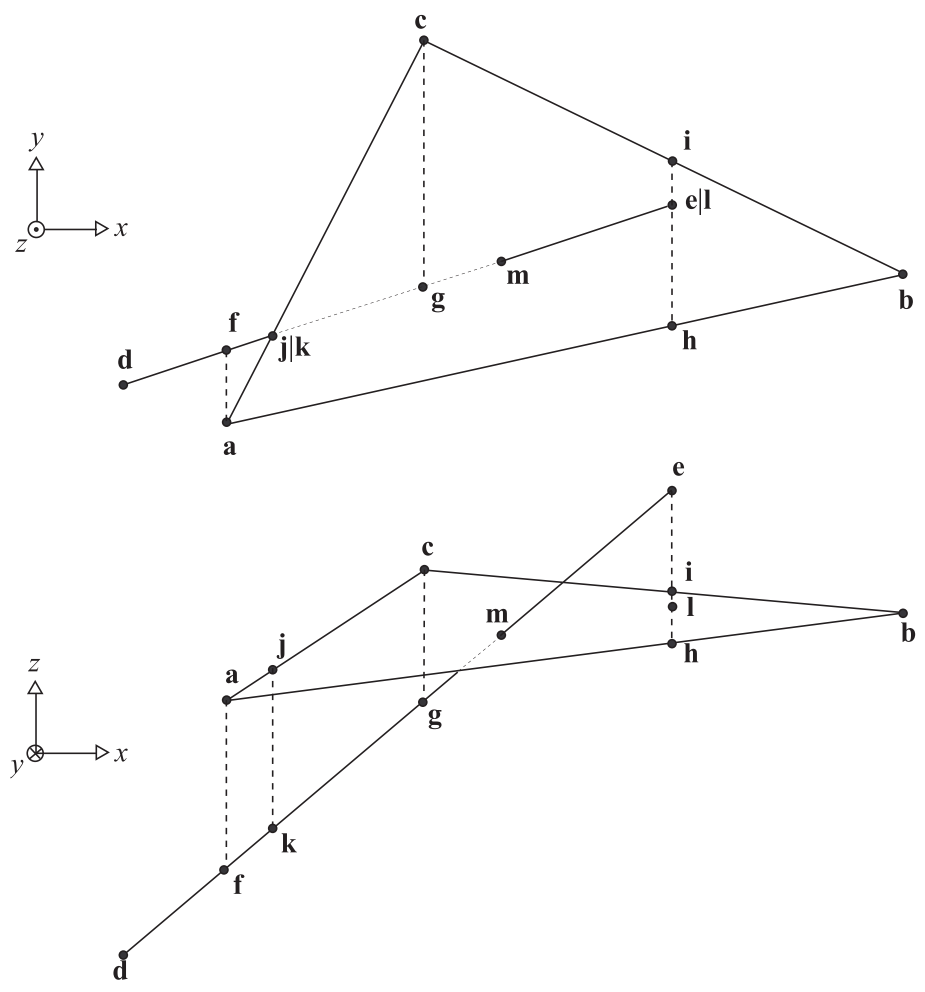


Figure 4.8: *An example showing how the basic algorithm determines the intersection point between a triangular facet defined by points* **a**,**b**,**c** *and an edge defined by points* **d**,**e**. *Points* **f**,**g**,**h**,**i** *are determined by the 1D intersection calculations,* **j**,**k**,**l** *by the 2D intersection calculations, and the result,* **m**, *by the final 3D intersection calculation.*

tween a triangular facet and an edge.

4.5 Formulae for determining intersections

I now demonstrate the relationships between the various intersection calculations in mathematical terms.

A function known as an *intersection function* is used to specify whether two entities are considered to intersect. These are denoted by $X_{ij}(o_A, o_B)$, where o_A and o_B are the entities of A and B, and i and j indicate the manifold dimensionality of o_A and o_B. An intersection function takes an integer value. In normal circumstances, when the shapes concerned satisfy the geometric constraints, an intersection function should take the value -1, 0 or 1, or just 0 or 1 for the case of an intersection function between a vertex and a solid. (We discuss later the significance of intersection function values outside the expected range.) A non-zero function value indicates that the entities intersect; the sign of the function value indicates the nature of the intersection, for example, whether an edge (or half-edge) intersecting a facet enters or exits the solid through the facet. In general terms, $X_{ij}(o_A, o_B)$ operates in $(i+j)$-dimensional space or subspace, as specified by the first $i+j$ coordinate values used in the Cartesian representation of the point. Hence the level 3 intersection functions ($i+j=3$) operate in full 3D space. The level 2 intersection functions ($i+j=2$) operate in effect in the (x, y) plane, and the level 1 intersection functions operate simply on the (x) line. When o_A and o_B intersect, i.e. $X_{ij}(o_A, o_B) \neq 0$, then there are a pair of intersection points, lying on each of o_A and o_B, and their first $i+j$ coordinate values are equal. (Hence at level 3 the two points are equal.) For intersections involving a vertex, the 'intersection point' lying on the vertex is simply the location of the vertex. The level 0 intersection function, $X_{00}(v_A, v_B)$, is considered to operate at the origin and always takes the value 1, the two 'intersection points' being the locations of v_A and v_B.

A function known as a *shadow function* is used to determine if an entity from B shadows an entity from A. $S_{ij}(o_A, o_B)$, a level $i+j$ function where $i+j = 0$, 1 or 2, operates in $(i+j+1)$-dimensional space or subspace as defined by the first $i+j+1$ coordinate values. For o_B to shadow o_A, it must intersect o_A in $(i+j)$-dimensional subspace, and also the $(i+j+1)$th coordinate value of o_B at the intersection point must be greater than or equal to the equivalent coordinate value of o_A at that point. In mathematical terms:

$$S_{ij}(o_A, o_B) = \begin{cases} X_{ij}(o_A, o_B) & \text{if } X_{ij}(o_A, o_B) \neq 0 \text{ and } \xi_B^{(i+j+1)} \geq \xi_A^{(i+j+1)} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

The effect of the condition in the formula is that when the ξ terms are computed to be exactly equal, the situation is treated as though the ξ_B term were strictly greater than ξ_A.

It is as though object B had been adjusted by an arbitrarily small shift in the direction of the positive axis for ξ. It is this symbolic perturbation that makes in unnecessary to treat special cases differently. A consequence of this rule is that the basic operation is not symmetric: the structures representing $A \cup B$ and $B \cup A$ are not necessarily identical.

The intersection function $X_{ij}(o_A, o_B)$ is evaluated as the sum, with appropriate $+/-$ signs, of each of the shadow function values: $S_{i-1,j}(c, o_B)$ for each boundary component c of o_A (if $i > 0$); and $S_{i,j-1}(o_A, c)$ for each boundary component c of o_B (if $j > 0$). The individual formulae are listed in table 4.1. Within this table, ∂A denotes the collection of facets that border shape A, ∂f denotes the collection of half-edges that border facet f, $v_s(e)$ and $v_e(e)$ denote the start- and end-vertex of edge (or half-edge) e.

Note, incidentally, that $X_{02}(v_A, f_B)$ and $X_{20}(f_A, v_B)$ are formulations (not identical) for the *winding number* of the vertex v within facet f on the (x, y) plane [Hai89]. For any point beyond the bound of a shape, the *winding number* relating to the shape is 0, and if one imagines the point being moved in space, the winding number has its value incremented by 1 as one crosses from the outer to the inner side of the boundary (and conversely decreased if one crosses the other way). In 2D, the winding number is a net count of how many times the facet boundary goes round the vertex in an anticlockwise direction. $X_{03}(v_A, B)$ and $X_{30}(A, v_B)$ are formulations for the winding number of the vertex v within the polyhedral shape in full 3D space.

Functions relating to edges also apply to half-edges. Any function relating to a forward half-edge will take the same value as the same function applied to the edge to which it belongs. It can be verified from the formulae that the value of an intersection function or shadow function applied to a backward half-edge is equal to minus the value of the same function applied to the edge to which it belongs; any associated intersection points are also identical.

The intersection point(s) associated with two entities deemed to intersect (by virtue of the fact that $X_{ij}(o_A, o_B) \neq 0$) can be obtained by interpolating intersection point pairs associated with the lower-level intersections that have already been determined. Lower-level intersection point pairs relate to two entities: either $\partial^* o_A$ and o_B, or o_A and $\partial^* o_B$, where $\partial^* o$ designates one of the boundary component entities bordering entity o; the lower-level intersections satisfy the inequality $X_{i-1,j}(\partial^* o_A, o_B) \neq 0$ or $X_{i,j-1}(o_A, \partial^* o_B) \neq 0$. Two lower-level intersection point pairs are required for the interpolation calculation: one where the entity from B shadows the entity from A in the direction of the $(i+j)$th coordinate value and one where the entity from A shadows the entity from B. There is, in fact, always at least one such lower-level intersection of each type (see section 4.9, p.76, for proof); hence, interpolation is always possible. If the intersection point pairs used are $\mathbf{x}_{A+}$ and $\mathbf{x}_{B+}$ where B shadows A and $\mathbf{x}_{A-}$ and $\mathbf{x}_{B-}$ where A shadows B, the intersection point pair between for the original pair of entities in the higher-dimensional space are

level	*formula*	*situation leading to value of* +1
0	$X_{00}(v_A, v_B) = 1$	
1	$X_{01}(v_A, e_B) = S_{00}(v_A, v_e(e_B)) - S_{00}(v_A, v_s(e_B))$	v_A lies within e_B, with e_B going left to right
	$X_{10}(e_A, v_B) = -S_{00}(v_e(e_A), v_B) + S_{00}(v_s(e_A), v_B)$	v_B lies within e_A, with e_A going left to right
2	$X_{02}(v_A, f_B) = -\sum_{h\in\partial f_B} S_{01}(v_A, h)$	v_A lies within f_B, with f_B going anti-clockwise (denoting that it faces upwards)
	$X_{11}(e_A, e_B) = S_{01}(v_e(e_A), e_B) - S_{01}(v_s(e_A), e_B)$ $+S_{10}(e_A, v_e(e_B)) - S_{10}(e_A, v_s(e_B))$	e_A crosses e_B from left to right
	$X_{20}(f_A, v_B) = \sum_{h\in\partial f_A} S_{10}(h, v_B)$	v_B lies within f_A, with f_A going anti-clockwise
3	$X_{03}(v_A, B) = \sum_{f\in\partial B} S_{02}(v_A, f)$	v_A lies within B, with facets facing outwards
	$X_{12}(e_A, f_B) = -S_{02}(v_e(e_A), f_B) + S_{02}(v_s(e_A), f_B)$ $-\sum_{h\in\partial f_B} S_{11}(e_A, h)$	e_A crosses f_B going to the outer side
	$X_{21}(f_A, e_B) = -\sum_{h\in\partial f_A} S_{11}(h, e_B)$ $+S_{20}(f_A, v_e(e_B)) - S_{20}(f_A, v_s(e_B))$	e_B crosses f_A going to the outer side
	$X_{30}(A, v_B) = -\sum_{f\in\partial A} S_{20}(f, v_B)$	v_B lies within A, with facets facing outwards

Table 4.1: *Formulae for intersection functions.*

defined mathematically by the standard linear interpolation formulae:

$$\mathbf{x}_A = \mathbf{x}_{A+} - t(\mathbf{x}_{A+} - \mathbf{x}_{A-}) \tag{4.2}$$

$$\mathbf{x}_B = \mathbf{x}_{B+} - t(\mathbf{x}_{B+} - \mathbf{x}_{B-}) \tag{4.3}$$

with t selected to ensure $\xi_A^{(i+j)} = \xi_B^{(i+j)}$. Hence:

$$t = \Delta_+/(\Delta_+ - \Delta_-) \tag{4.4}$$

where

$$\Delta_+ = \xi_{B+}^{(i+j)} - \xi_{A+}^{(i+j)} \quad (\geq 0) \tag{4.5}$$

$$\Delta_- = \xi_{B-}^{(i+j)} - \xi_{A-}^{(i+j)} \quad (< 0) \tag{4.6}$$

These formulae provide the means by which to compute the intersection point pair. The relationship $\Delta_- < 0 \leq \Delta_+$ guarantees the avoidance of division by zero in equation (4.4).[1]

[1]The formulae as listed do not necessarily represent the arithmetic operations used to compute the result. If $t > \frac{1}{2}$, it is more accurate to determine $\mathbf{x}_A$ and $\mathbf{x}_B$ based on an evaluation of $1 - t$ rather than t. Also, $\xi_A^{(i+j)}$ and $\xi_B^{(i+j)}$ should be computed either according to formula (4.2) or (4.3), but not to

4.6 Constructing the result

Having obtained the level 3 intersection status values, $X_{03}(v_A, B)$, $X_{12}(e_A, f_B)$, $X_{21}(f_A, e_B)$ and $X_{30}(A, v_B)$, and also the point of intersection for each intersecting edge-facet pair (i.e. those edge-facet pairings for which $X_{12}(e_A, f_B)$ or $X_{21}(f_A, e_B)$ is non-zero) it is possible to construct the result.

First, a value known as the *inclusion value*, $I_{ij}(o_A, o_B)$ for $i{+}j{=}3$, is determined for each level-3 pairing of entities. These values are related to the intersection status value in accordance with the particular operation being applied:

$$I_{03}(v_A, B) = c_A + c_I X_{03}(v_A, B) \tag{4.7}$$

$$I_{12}(e_A, f_B) = c_I X_{12}(e_A, f_B) \tag{4.8}$$

$$I_{21}(f_A, e_B) = c_I X_{21}(f_A, e_B) \tag{4.9}$$

$$I_{30}(A, v_B) = c_B + c_I X_{30}(A, v_B) \tag{4.10}$$

where c_A, c_B and c_I depend on the operation type:

$$\text{Union:} \qquad c_A = 1 \quad c_B = 1 \quad c_I = -1 \tag{4.11}$$

$$\text{Intersection:} \qquad c_A = 0 \quad c_B = 0 \quad c_I = +1 \tag{4.12}$$

$$\text{Difference:} \qquad c_A = 1 \quad c_B = 0 \quad c_I = -1 \tag{4.13}$$

The values of I_{03} and I_{30} ensure that vertices are retained when they lie on the appropriate side of the other solid. For geometrically valid shape representations the value X_{03} or X_{30} for each vertex is expected to be either 1 or 0, depending on whether the vertex lies inside or outside the other shape. For the intersection operation, I_{03} or I_{30} is 1 for each vertex

both; $\xi_A^{(i+j)}$ and $\xi_B^{(i+j)}$ are supposed to be identical, but code based on the two formulae will in general yield values that differ slightly. It is acceptable to compute Δ_- and Δ_+ according to the formulae presented here, since the computed values will have the correct sign. Section 7.1, p.106, presents details of one coding option for determining the intersection point between two edges in 2D space, and the error analysis for that method is presented in section 8.2, p.145.

lying inside the other shape, and 0 for each one lying outside, indicating that the inside vertices are to be retained. For the union operation, I_{03} or I_{30} is 1 for each vertex lying outside, indicating that it is to be retained. For the difference operator, $I_{03}(v_A, B) = 1$ for each vertex of A outside B, to indicate that it is to be retained (as for the union operator). Vertices of B lying inside A are retained for the difference operation, but these are indicated by $I_{30}(A, v_B) = -1$ to denote that it is to form a surface facing the opposite way as in B.[2]

The equations for $I_{12}(e_A, f_B)$ and $I_{21}(f_A, e_B)$ also act to drive the construction of the result. For example, $I_{21}(f_A, e_B) = 1$ indicates that the intersection point between f_A and e_B is to be an end-vertex for the part of e_B retained in the result, and $I_{21}(f_A, e_B) = -1$ indicates it is to be a start-vertex.

The first task in the construction is to add the vertices—both retained vertices and intersection vertices:

- retained vertices of A, copied from each vertex v_A for which $I_{03}(v_A, B) \neq 0$;
- new intersection vertices, associated with each edge-facet pairing, e_A from A and f_B from B, for which $I_{12}(e_A, f_B) \neq 0$, and located at the point of intersection;
- new intersection vertices, associated with each facet-edge pairing, f_A from A and e_B from B, for which $I_{21}(f_A, e_B) \neq 0$, and located at the point of intersection;
- retained vertices of B, copied from each vertex v_B for which $I_{30}(A, v_B) \neq 0$;

It is possible that some vertices will have exactly identical positions. No special action is required from the basic operation when this is so—the vertices can remain distinct. Half-edges associated with a particular edge are deemed to intersect a facet at the same vertex at which the edge intersects the facet.

The construction stages at levels 4, 5 and 6 build up respectively the edges of the result, the facets, and ultimately the shape itself. Each operation at these stages has a similar pattern: the manifold dimensionalities of the two input entities, o_A from A and o_B from B, sum to the level number, k, and the result of each operation is a number of entities (possibly none) of manifold dimensionality $k-3$. Furthermore, the effective boundary components of the resulting entities are made up from the results of each operation one level below, relating to one of the two entities and one of the boundary components of the other entity, i.e. to $\partial^* o_A$ and o_B, or to o_A and $\partial^* o_B$.

[2]My explanation of the difference operation was incorrect in [SD07], and is corrected here. My thanks to Ma Jian (Tsinghua University) for pointing out this mistake and also one other in this section.

First consider the three types of operation at level 4. These determine the retained parts of an edge from A, the intersecting edge(s) between two facets from A and B respectively, and the retained parts of an edge from B. The end result of each operation, generally, is a number of edges (possibly zero). Initially, though, the result is computed as a composite edge with a number of start-vertices and end-vertices, each start-vertex and end-vertex being determined without regard (at this stage) as to which segment it should belong to.

The process of determining start- and end-vertices is carried out by considering every subordinate level-3 pairing of entities and assigning a *net end-vertex count* value closely related to the inclusion number for the vertex. The net end-vertex count can potentially be any integer value. In normal circumstances, when A and B are both geometrically valid, the values are expected to be restricted to -1, 0 or 1. The statement that a vertex is assigned 'a net end-vertex count of n' for a particular composite edge is taken to mean:

- 'assign the vertex as an end-vertex n times' if $n > 0$;
- 'assign the vertex as a start-vertex $-n$ times' if $n < 0$;
- no action if $n = 0$.

In detail, the rules are:

- To determine the retained parts of edge (or half-edge) e_A from A:
  - the retained vertex copied from $v_e(e_A)$ is assigned a net end-vertex count of $I_{03}(v_e(e_A), B)$;
  - the retained vertex copied from $v_s(e_A)$ is assigned a net end-vertex count of $-I_{03}(v_s(e_A), B)$;
  - for each facet $f \in \partial B$, the intersection vertex between e_A and f is assigned a net end-vertex count of $I_{12}(e_A, f)$;
- To determine the intersection edge(s) arising from the intersection of facets f_A from A and f_B from B:
  - for each half-edge $h \in \partial f_A$, the intersection vertex between h and f_B is assigned a net end-vertex count of $-I_{12}(h, f_B)$;
  - for each half-edge $h \in \partial f_B$, the intersection vertex between f_A and h is assigned a net end-vertex count of $I_{21}(f_A, h)$;
- To determine the retained parts of edge (or half-edge) e_B from B:

- for each facet $f \in \partial A$, the intersection vertex between e_B and f is assigned a net end-vertex count of $I_{21}(f, e_B)$;
- the retained vertex $v_e(e_B)$ is assigned a net end-vertex count of $I_{30}(A, v_e(e_B))$;
- the retained vertex $v_s(e_B)$ is assigned a net end-vertex count of $-I_{30}(A, v_s(e_B))$.

Each operation creates an equal number start-vertices and end-vertices, making it possible to break down the composite edge into single-segment edges that can be incorporated into the representation of the resulting shape. Any pairing of the vertices is sufficient to ensure the topological robustness of the process as a whole. In normal circumstances, when the geometric constraints are adhered to, the vertices are expected to be collinear and sequenced to alternate between start-vertex and end-vertex. In such circumstances it is appropriate to pair start-vertices and end-vertices accordingly in order to maintain geometric correctness. A suitable way to achieve this pairing is to order both the list of start-vertices and the list of end-vertices in accordance with the direction of the composite edge, and to pair off the vertices accordingly. The direction of the composite edge, $\mathbf{v}$, can be computed from the start- and end-vertex positions, $\mathbf{x}_{s_i}$ and $\mathbf{x}_{e_i}$:

$$\mathbf{v} = \sum_i \mathbf{x}_{e_i} - \sum_i \mathbf{x}_{s_i} \tag{4.14}$$

The start-vertices and end-vertices are then re-ordered so that

$$(\mathbf{x}_{s_{i+1}} - \mathbf{x}_{s_i}).\mathbf{v} \geq 0 \tag{4.15}$$

$$(\mathbf{x}_{e_{i+1}} - \mathbf{x}_{e_i}).\mathbf{v} \geq 0 \tag{4.16}$$

The retained part of a half-edge is assumed to be the same as for the edge to which it belongs, except that for backward half-edges, the start-vertex and end-vertex are switched for each segment.

The two types of operation at level 5 determine respectively the retained parts of a facet from A, and the retained parts of a facet from B. The process determines the half-edges that bound the retained part. In detail:

- To determine the retained parts of facet f_A from A:
  - for each half-edge $h \in \partial f_A$, include the retained half-edge(s) of h;
  - for each facet $f \in \partial B$, include the forward half-edge(s) of the intersection edge(s) between f_A and f.

- To determine the retained parts of facet f_B from B:
  - for each facet $f \in \partial A$, include the backward half-edge(s) of the intersection edge(s) between f and f_B;
  - for each half-edge $h \in \partial f_B$, include the retained half-edge(s) of h.

For the polygon mesh variant of the algorithm the half-edges are simply collected to specify the boundary of the facet polygon. No action is required to determine how the half-edges form loops and regions. For the triangle mesh variant it is necessary to break the polygon region into triangles. The requirements for the triangulation process are covered in section 4.8.

The final stage of the process at level 6 is simply to incorporate all the retained facets from A and all the retained facets from B to form the resulting shape.

4.7 Issues arising from numerical inaccuracy

Geometric validity of the result is *not* assured by the basic Boolean algorithm. When the input structures are geometrically valid the algorithm progresses without problem. However, numerical errors in determining the positions of vertices, whether in the basic operation itself or in other operations, can lead to there being invalid regions and boundary self-intersections in the resulting structure. The consequence of this is that any subsequent basic Boolean operation based on such geometrically invalid data may well lead to unexpected intermediate results. For example, the evaluation of $X_{30}(A, v_B)$ for a particular vertex v_B may lead to a value other than 0 or 1, a value greater than 1 indicating that v_B lies in a multiply-enclosed region of A, a negative value indicating that it lies in an inside-out region. I have already indicated that it is desirable to apply a smoothing operation to the data structure generated by the basic Boolean algorithm in order to make the result generally acceptable for subsequent processing. However, the smoothing operation, cannot be relied on always to make the structure geometrically valid. Consequently, it is insufficient for the basic algorithm to assume that situations like the one described will never occur.

Nevertheless, as section 4.9 will prove, it is always possible for the basic algorithm to progress and generate a topologically valid structure. An example of what can happen in the 2D basic operation is shown in figure 4.9 whereby the result has a 'retained edge' with overlapping segments. Another example of what may happen is that the end-vertex of an existing edge may be assigned a net end-vertex count value outside the range $[-1, 1]$, thus making it necessary for it to be included more than once as end-vertex or start-vertex in

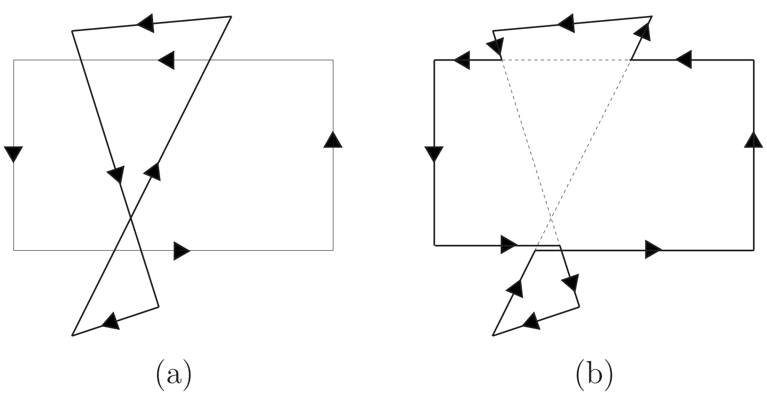


Figure 4.9: *Example showing (a) two shape representations, one geometrically invalid, and (b) the result generated by the union operation. In the result the retained part of the bottom edge of the rectangle is represented twice where it overlaps the inside-out region of the other shape. In this example it is manifested as two overlapping segments (drawn separated for the purposes of illustration).*

the retained part of the edge. However, in order to avoid failure, the algorithm must be implemented to take into account the possibility of such occurrences.

A convenient way of viewing the situation is to regard a particular point in space, $\mathbf{x}$, as being 'included' in a set a certain number of times. This number is the winding number as defined (for the 2D case) in [Hai89]—see section 4.5, p.66. If we name the winding numbers for point $\mathbf{x}$ relating to sets A and B as $a(\mathbf{x})$ and $b(\mathbf{x})$, and the winding numbers relating to the union, intersection and difference as $u(\mathbf{x})$, $i(\mathbf{x})$ and $d(\mathbf{x})$, we expect the following:

$$u(\mathbf{x}) = a(\mathbf{x}) + b(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x}) \tag{4.17}$$

$$i(\mathbf{x}) = a(\mathbf{x})b(\mathbf{x}) \tag{4.18}$$

$$d(\mathbf{x}) = a(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x}) \tag{4.19}$$

Alternatively, the winding number for the result of a general operation, $\phi(\mathbf{x})$, can be expressed using the c constants associated with the operation in question, as specified in equations (4.11) to (4.13):

$$\phi(\mathbf{x}) = c_A a(\mathbf{x}) + c_B b(\mathbf{x}) + c_I a(\mathbf{x})b(\mathbf{x}) \tag{4.20}$$

A particular issue of concern for the polygon-mesh variant of the 3D algorithm is the breaching of the planarity constraint, which is in practice inevitable with the use of approximate arithmetic. It leaves open to question the exact positioning of the shape boundary. A more pragmatic concern is what happens when a facet is nearly vertical. The lower-dimensional calculations consider the facet as projected onto the (x, y) plane, which will be *almost* a line, but it could in fact be quite irregular and geometrically invalid. The topological robustness of the algorithm will enable it to continue but it may lead to invalid regions with a thickness the same order of size as numerical errors that distorted the facet.

One might question the appropriateness of allowing the algorithm to continue when a geometric error is detected, given that the generated result is also likely to be geometrically invalid. Continuation of the algorithm can be defended on two grounds. First, there is a good possibility that the data smoothing post-process will remove invalid regions. The thickness of any region of invalidity in a near-vertical facet when projected onto (x, y) space is expected to be limited in extent, bounded by the largest possible numerical rounding error in the calculations, and the thickness of any region of invalidity in the full 3D result should not exceed that thickness. Consequently, any invalid region is likely to be resolved by the post-process provided the specified distance tolerance value exceeds the upper bound to this thickness. Secondly, any geometric error that persists will be local, and will not prevent the algorithm from operating satisfactorily away from the region of invalidity.

4.8 The requirements of triangulation

For the triangle-mesh variant of the algorithm it is necessary to convert the polygonal region of a retained facet into a set of triangles. This can be considered the 2D manifold equivalent of the 1D manifold task of segmenting a composite edge into a collection of single-segment edges. Much has been published on the triangulation of true polygonal regions, for example [BE92, dBvKOS97]. For full topological robustness, though, it must be possible to form a 'triangulation' even when the polygonal region is geometrically invalid.

The triangulation process consists of a series of operations in which a new edge known as an *internal edge* is added to connect vertices. When the process ends, the original half-edges and the half-edges of the newly added internal edges form a series of triangles. This specifies the process purely in terms of connectivity, without saying where the edges should be located. Such a strategy ensures topological robustness, since it guarantees that the structure generated satisfies the topological constraints. The ideal option, however, is always to select an internal edge from the *interior* of the polygonal region, and which does not intersect or touch other edges (including internal edges) except at their respective ends.

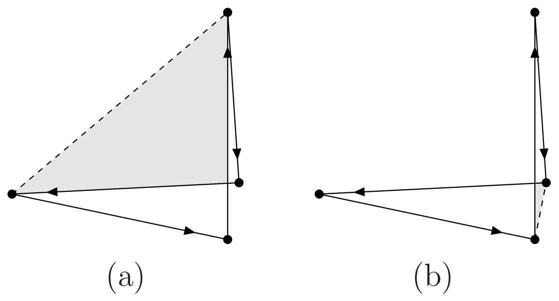


Figure 4.10: *Example showing that a geometrically invalid quadrilateral can be triangulated one of two ways. Whichever way is selected, the two triangles face opposite directions and overlap. However, option (b) is preferable because the region of overlap (drawn shaded) is smaller.*

It is always possible to do this when the region is geometrically valid. Such an approach ensures that every triangle faces the same direction as the original polygonal region, and that no triangle overlaps another, thereby maintaining geometric validity.

This approach is not possible for a geometrically invalid region; doubly enclosed regions need to be covered by two overlapping triangles, and inside-out regions by a triangle facing the opposing direction. Furthermore, the region of invalidity is in general extended. This is demonstrated in the case of a geometrically invalid quadrilateral, as shown in figure 4.10. A topological quadrilateral, regardless of its geometric validity, can only be triangulated in one of two ways, splitting it into two triangles at one of the diagonals. However, for a geometrically invalid quadrilateral it is inevitable for the two triangles, which have to face opposite ways, to overlap in the shape exterior where the winding number is 0. Note, though, that the second option shown is preferable, because the area of overlap it introduces is smaller.

In general terms, it is preferable for the triangulation process to operate in such a way that minimises how far any geometrically invalid region is extended.

4.9 Proofs relating to topological robustness

The topological robustness of the scheme guarantees that the basic Boolean operation will always generate a topologically valid result from topologically valid input structures, irrespective of any geometric errors in the data. Even replacing the vertex position data with random data would generate a topologically valid (though meaningless) result. I demonstrate this through a series of theorems with proofs relating to operations at each level of the basic algorithm. Taken together they show that the basic algorithm is able to progress without ambiguity and without having to invent or discard data, and also that the output satisfies the expected topological constraints. In the wording of each theorem it is taken as read that the input structures satisfy the topological constraints (though not necessarily the geometric constraints).

The basic algorithm is executed as a fixed number of operations, with each operation requiring a finite amount of work, so there is no concern about the basic algorithm failing to terminate due to looping or data complexity.

The operations up to level 3 determine the intersection status between entities. It needs to be proved that it is always possible to compute the intersection status values, and that the intersection point(s) can always be computed for any pair of entities deemed to intersect. The latter calculation, following equations (4.2) to (4.6), is of particular concern since it relies on there being suitable solutions to lower-level calculations upon which to base the interpolations.

Consider the intersection status calculations up to level 3. The term *level k intersection data* ($k = 0$, 1, 2 or 3) is used to refer to all intersection status values at that level, i.e. all values of $X_{ij}(o_A, o_B)$ for $i{+}j{=}k$, and also all intersection point pairs, $\mathbf{x}_A$ and $\mathbf{x}_B$, for which $X_{ij}(o_A, o_B) \neq 0$ at level k. It is true to say that all the intersection data at level 0 are available, because $X_{00}(v_A, v_B)$ invariably equals 1, and $\mathbf{x}_A$ and $\mathbf{x}_B$ are simply the locations of v_A and v_B. The following theorem demonstrates that intersection data can be obtained successively for levels 1, 2 and 3:

Theorem 1 *For k = 1, 2 or 3, if all the intersection data at level $k{-}1$ are available, then all the intersection data at level k can be deduced.*

Proof. $X_{ij}(o_A, o_B)$ for $i{+}j{=}k$ is computed, according to the formulae shown in table 4.1, from terms $S_{i-1,j}(\partial^* o_A, o_B)$ and $S_{i,j-1}(o_A, \partial^* o_B)$, where $\partial^* o$ designates one of the boundary components by which entity o is defined. Each of the S terms is readily available from equation (4.1), because the equivalent X term, $X_{i-1,j}(\partial^* o_A, o_B)$ or $X_{i,j-1}(o_A, \partial^* o_B)$, is known, and the intersection points are available if it takes a non-zero value.

Whenever for a particular pair of entities, o_A and o_B, $X_{ij}(o_A, o_B)$ ($i{+}j{=}k$) is computed to be non-zero, the intersection point pair, $\mathbf{x}_A$ and $\mathbf{x}_B$, need to be computed by linear interpolation. This is possible when there are two intersection point pairs at level $k{-}1$, each associated with a non-zero value of $X_{i-1,j}(\partial^* o_A, o_B)$ or $X_{i,j-1}(o_A, \partial^* o_B)$: one for which A shadows B, and one for which B shadows A.

To prove that this is indeed the case, let us assume it *not* to be true. So among all the level $k{-}1$ pairings of entities between $\partial^* o_A$ and o_B and between o_A and $\partial^* o_B$ there would either be (1) no level $k{-}1$ intersection point pairs for which B shadows A, or (2) no level $k{-}1$ intersection point pairs for which A shadows B. For case (1) (which includes the possible case of there being no level $k{-}1$ intersection point pairs at all), the $S_{i-1,j}(\partial^* o_A, o_B)$ and

$S_{i,j-1}(o_A, \partial^* o_B)$ terms that make up the formula for X_{ij} would all take the value 0, in which case X_{ij} would be 0. For case (2), the $S_{i-1,j}$ and $S_{i,j-1}$ terms would each be identical to the equivalent $X_{i-1,j}$ and $X_{i,j-1}$ terms; substituting the expressions given in table 4.1 for the $X_{i-1,j}$ and $X_{i,j-1}$ terms into the modified expression for X_{ij} would again yield X_{ij} to be 0 (taking into account, for certain of the cases, that A and B adhere to the topological constraints). This contradicts the stipulation that $X_{ij} \neq 0$. Therefore there is always at least one level $k-1$ intersection point pair involving $\partial^* o_A$ and o_B or o_A and $\partial^* o_B$ for which A shadows B, and also at least one for which B shadows A. Hence it is always possible to determine the intersection point pair by means of the formulae specified in equations (4.2) to (4.6). □

Each of the level 4 calculations determines a composite edge (possibly empty), which is subsequently broken up into standard edges each with one start-vertex and one end-vertex. This break up is possible because there is always the same number of start-vertices and end-vertices, as proved by the next theorem:

Theorem 2 *Each composite edge computed at level 4 has exactly the same number of start-vertices and end-vertices.*

Proof. Define $E_{ij}(o_A, o_B)$ for $i+j=4$ to be the number of end-vertices minus the number of start-vertices computed when determining the composite edge constructed from entities o_A and o_B. Consider the rules for constructing the composite edge. The net end-vertex count assigned for each retained vertex and each intersection vertex contributes to E_{ij}, so E_{ij} is simply the sum of all the net end-vertex counts:

$$E_{13}(e_A, B) = I_{03}(v_e(e_A), B) - I_{03}(v_s(e_A), B) + \sum_{f \in \partial B} I_{12}(e_A, f) \tag{4.21}$$

$$E_{22}(f_A, f_B) = -\sum_{h \in \partial f_A} I_{12}(h, f_B) + \sum_{h \in \partial f_B} I_{21}(f_A, h) \tag{4.22}$$

$$E_{31}(A, e_B) = \sum_{f \in \partial A} I_{21}(f, e_B) + I_{30}(A, v_e(e_B)) - I_{30}(A, v_s(e_B)) \tag{4.23}$$

Substituting the I terms with the formulae given in equations (4.7) to (4.10), then substituting the X terms with the formulae given in table 4.1, confirms that $E_{ij}(o_A, o_B) = 0$ for each case. □

Each operation at level 5 generates a polygonal retained facet or part-facet (possibly empty) satisfying the facet boundary constraint, as proved by the next theorem. This is required for both the polygon and triangle mesh variants of the algorithm.

Theorem 3 *The half-edges computed to border the retained part of a facet satisfy the facet boundary constraint, with each vertex acting as start-vertex for all the half-edges the same number of times it acts as end-vertex.*

Proof. Consider facet $f_A \in \partial A$. Recall that the half-edges deemed to border the retained part of f_A are obtained from the retained parts of each half-edge bordering f_A and from the forward half-edge(s) of the intersection edge(s) between f_A and each facet bordering B. The vertices potentially included in the bounded half-edges are:

- retained vertices from A from each vertex from f_A;
- intersection vertices between each half-edge bordering f_A and each facet bordering B;
- intersection vertices between f_A and each edge e from B.

We show that for all three types of vertex, any one is instanced the same number of times as start-vertex and end-vertex by the half-edges bordering the retained facet.

Concerning a retained vertex, v, from f_A: If f_A has n half-edges that have v as their end-vertex, then by virtue of the facet boundary closure constraint it also has exactly n half-edges that have v as their start-vertex. In the resulting structure, v is assigned a net end-vertex count of $I_{03}(v, B)$ for the retained part of each half-edge that ends at v, and $-I_{03}(v, B)$ for the retained part of each half-edge that starts at v.

Concerning an intersection vertex, v, between f_A and an edge from B: If B has n half-edges going from vertex v_0 to v_1, referred to as h, then because of the shape boundary closure constraint it also has exactly n half-edges going from v_1 to v_0, referred to as h^*. These half-edges all form one edge, e, and we assume h to be the forward half-edge. Each of the half-edges concerned borders a facet from B, which potentially intersects f_A, and it is the forward half-edges that form the border of the retained part of f_A. For those facets bordered by h, the composite intersection edge has a net end-vertex count for the intersection vertex v of $I_{21}(f_A, h)$. For those bordered by h^*, the count is $I_{21}(f_A, h^*)$. Substitution shows in fact that $I_{21}(f_A, h^*) = -I_{21}(f_A, h)$. Hence for the retained part of f_A, the number of half-edges ending at v equals the number of half-edges starting at v.

Finally, concerning an intersection vertex, v, between a half-edge $h \in \partial f_A$ and a facet $f \in \partial B$: The retained part of h (which borders the retained part of f_A) has a net end-vertex count for v of $I_{12}(h, f)$. The composite intersection edge between f_A and f has a net end-vertex count for v of $-I_{12}(h, f)$, hence so too do the forward half-edge(s) that border the retained part of f_A. So the retained part of f_A has the same number of half-edges starting and ending at v.

The proof relating to the retained parts of a facet from B is similar. □

It finally needs to be shown that the result satisfies the shape boundary closure constraint, with the number of half-edges going from vertex P to vertex Q equalling the number of half-edges going from Q to P. For the triangle-mesh variant of the algorithm the internal edges created by the triangulation process have two matching half-edges belonging to the two triangles on either side; the bounding edges of a triangulated polygonal facet are simply copies of the half-edges that bound the facet in polygonal form. It therefore suffices to show that the half-edges generated by the polygon-mesh variant of the algorithm match up.

Theorem 4 *The polygonal facets computed to border the resulting solid satisfy the shape boundary constraint, with the number of half-edges going from vertex P to vertex Q equalling the number of half-edges going from Q to P.*

Proof. First consider an edge e belonging to one of the input shapes, with forward and backward half-edges h and h^*, and the facets that include either h or h^* as part of their boundary. The retained part of an original facet bordered by h will itself be bordered (in part) by the retained part of h, which is in fact the forward half-edges of the retained part of edge e. Similarly, the retained part of an original facet bordered by h^* will be bordered (in part) by the backward half-edges of the retained part of e. Note that h and h^* are instanced equally many times as part of a facet boundary in the original structure. Therefore, the forward and backward half-edges of each segment of the retained part of e are instanced equally many times as part of a facet boundary in the *resulting* structure.

Finally consider the composite intersection edge between facets f_A and f_B originating from A and B. Recall that the forward half-edges of the segments from the composite intersection edge form part of the boundary of the retained part of f_A; likewise, the backward half-edges form part of the boundary of the retained part of f_B. Hence the forward and backward half-edges of each segment of the intersection edge are instanced once each as part of a facet boundary in the resulting structure. □

This completes the chapter on the basic Boolean algorithm. The chapter that follows briefly describes how a data-smoothing can be used to resolve unsatisfactory marginal and redundant features that might appear in the result, and gives a brief account of how the algorithm was successfully incorporated within CAD software. Later on, chapter 8 applies error analysis to the 2D edge-edge intersection calculation used in the basic Boolean algorithm (both 2D and 3D), and which is also used in the simplification process (to be described later).

Chapter 5

Post-processing for the basic Boolean algorithm

The structure generated by the basic Boolean algorithm to represent the result of the Boolean operation is prone to contain features that are generally speaking unsatisfactory. This brief chapter discusses how a data-smoothing process can generally make the resulting structure acceptable. The first section describes how the post-process operates in principle, while the second discusses the issues that arose on implementing this process for commercially released software.

5.1 The data-smoothing post-process

The use of symbolic perturbation rules by the basic Boolean algorithm makes the special handling of degenerate cases unnecessary. Indeed, it is the avoidance of the need for special case handling that keeps the algorithm simple and straightforward, enabling full topological robustness to be guaranteed. However, a consequence of the basic algorithm is that the generated structure can have invalid or marginally valid geometry when the input is degenerate or close to degenerate. The data-smoothing post-process must modify the generated structure to make it satisfactory.

Consider, for example, the problem of determining the union of two axis-aligned cubes touching each other, with the top facet of one coinciding exactly with the bottom facet of the other, as shown in figure 5.1(a). The lower cube is designated as B, so according to the symbolic perturbation rules it is considered to have been shifted as shown in figure 5.1(b), and the two boundaries are considered to intersect. The polygon mesh variant of the basic Boolean operation yields a single-shell structure, with topology as depicted in figure 5.2(a). In this example, both retained vertices and intersection vertices coincide, certain edges

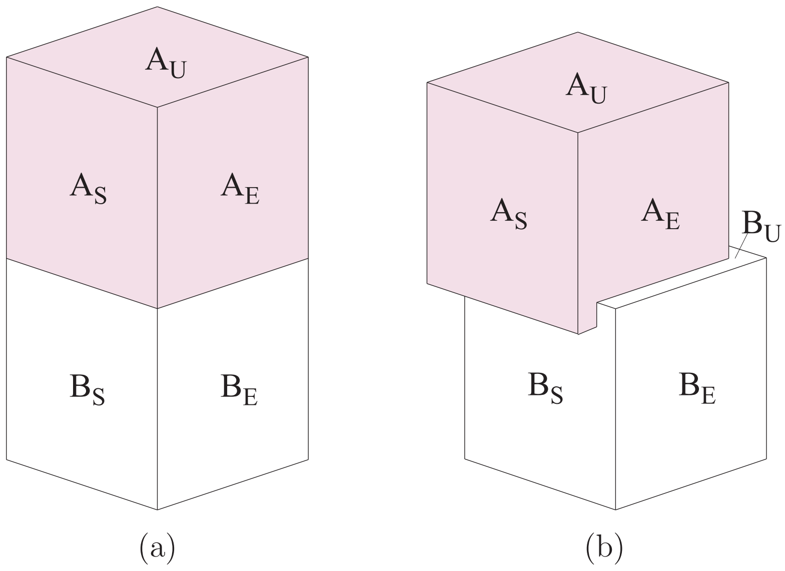


Figure 5.1: *Symbolic perturbation. (b) demonstrates how the basic Boolean algorithm considers the layout shown in (a), with cube A on top of cube B. B is assumed to be shifted in the positive axis directions. Facets are labelled according to the shape of origin and the facet normal direction, assuming an east-north-up coordinate system.*

also coincide, other edges are of zero length, and the retained parts of the two coinciding original facets (A_D and B_U) are in fact zero-area facets. Ideally the structure should be modified to that shown in figure 5.2(b). Conversely, if the upper cube is designated B, the two boundaries are considered *not* to intersect, and the result of the basic operation is simply an assembly of the two original boundary shells. In this case, the zero-thickness 'gap' between A and B needs to be removed, again to form a structure similar to that shown in figure 5.2(b), but with A and B interchanged. Other topological structures can be generated by the basic operation when the facets do not quite coincide, and often these too have to be resolved by the smoothing process.

The most obvious way to implement data-smoothing is as a series of adjustments to the structure, each one designed to resolve some inappropriate aspect of the data. With a polygonal mesh, an obvious first operation is to merge vertices that are coincident or nearly coincident. In consequence, some edges may have the same start- and end-vertex, and these can be removed without breaking the topological validity of the structure. Coincident and opposing half-edges that belong to the same facet can be considered to cancel each other out, and so be removed; likewise, coincident and opposing facets can be removed. Another useful operation in the data-smoothing process is edge-cracking, whereby an edge is split in two if a vertex or another edge is close by. This eases the handling of another operation: the partial cancellation of facets, needed when facets oppose each other but coincide in only part of their respective regions. Edge-cracking also enables the partial cancellation of half-edges within a facet. Some of the operations

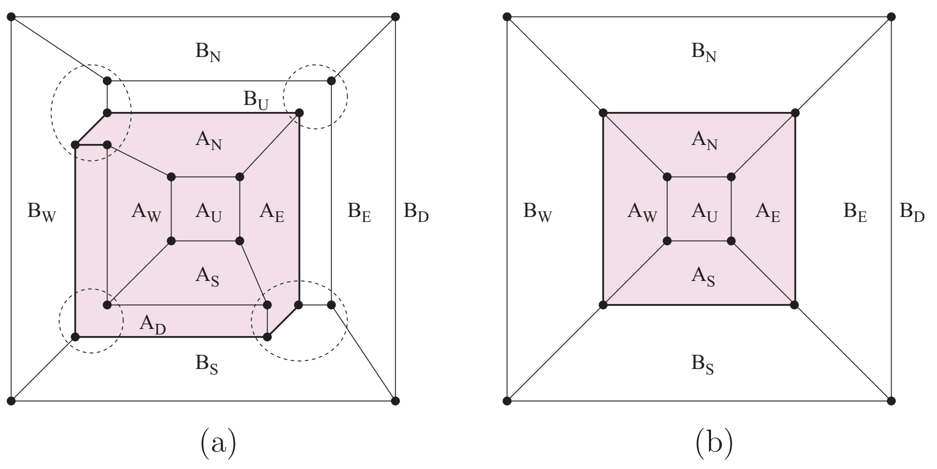


Figure 5.2: *The topological structure created when computing the union of two cubes as shown in figure 5.1. (a) shows the structure generated by the basic algorithm, and (b) shows the structure as it is expected to be after data-smoothing. Vertices that are identically positioned following the basic operation are encircled.*

are those required for data normalisation, as described in [Mil88].

These operations can form the basis of the data-smoothing process, and in fact they were used in a commercial implementation of the overall algorithm. For the record, the process is described in a little more detail in appendix A; the implementation is discussed briefly in the next section. Although the process can be (and was) tuned to be acceptably reliable in terms of a commercial software development, it is not fully robust. One problem is that the operations often make facets non-planar due to distortion. Other problems can arise in theory, at least: multiple operations may distort the boundary too much (as with data normalisation [Mil00]); self-intersections may arise that cannot be resolved by the operations; the method may not terminate in reasonable time; and so on.

5.2 Implementation

I devised the Boolean algorithm—both the basic algorithm and the data-smoothing post-process—and successfully implemented it for use within Cadcentre's widely circulated Plant Design Management System [AVE06]. The product uses Boolean operations principally to convert CSG models of industrial plant components to polyhedral approximations for the use in subsequent operations. The general polygonal mesh variant of the algorithm was implemented, since it was found to be more efficient than an initial trial version based on the triangular mesh. The algorithm has been used extensively in the released product since 1997. It is used most heavily for generating hidden-line engineering drawings, and also to assist in clash detection and surface rendering.

I implemented the data-smoothing process to make the structure suitable for downstream processing. It applies the operations described in the previous section, namely vertex merging, removal of zero-length edges, edge-cracking, half-edge cancellation, and facet cancellation (full or partial). The operations are applied using by default a tolerance of 0.1mm, in principle until no further operations are possible (but see below). Tests carried out at the time of implementation showed it to be more efficient to apply the data smoothing process once at the end of a sequence of basic Boolean operations rather than after each individual operation.

Although there is no theory to support full robustness of the data-smoothing process, it has turned out to be reliable to a sufficiently high degree. In the five years after initial release, users reported only one fault in the released product relating to the Boolean operation. This was a problem of non-termination in the smoothing operation, apparently complexity-related, arising for one particular case in which there were four coincident, almost vertical facets. The problem was resolved in subsequent releases by enforced early termination in the data smoothing process. This led to the formation of shard-like artifacts in the result for the one known problem case. Though this is strictly speaking a failure of the operation, the artifact problem is considerably less serious than that of non-termination. It was in fact possible to resolve the artifacts generated in this particular case by applying data smoothing after each execution of the basic Boolean algorithm; however, the general decline in performance that would have been incurred by taking that approach was not justified, given the rarity and the non-fatal nature of the artifact problem.

In my experience, it was much easier to achieve a high degree of reliability for the combined basic Boolean algorithm and data-smoothing process than it was to achieve even a moderate degree of reliability when refining a standard implementation of the Boolean operation that uses inexact arithmetic. With a standard implementation it is impractical to guarantee the consistency between low-level computations needed to guarantee a topologically valid result (see sections 2.3.3, p.32, and 2.3.4, p.33). For a system that is not topologically robust, allowance must be made for the possibility of a generated structure with connectivity errors. Either the generated boundary has to undergo a repair process, or the downstream processes have to be designed to be tolerant to connectivity faults in the input. If the process concerned is also the Boolean operation, with the old result used for input, then that further complicates the implementation of that operation. However, with the Boolean algorithm I present, downstream processes have to be tolerant only to possible geometric errors. The basic Boolean algorithm itself satisfies that requirement.

The data-smoothing process as it was implemented is not the only, or even in all likelihood, the best way of doing so. How best to implement the smoothing operation is clearly an important issue, since the success of the Boolean operation as a whole depends on the success of this operation, but it is one that remains open at present.

I have not covered the data structure representation in detail for the basic Boolean algorithm and smoothing process. Both processes are amenable to a range of possible representations, and it remains an open issue as to which might be preferred. For the basic Boolean operation, however, it is an unnecessary complication to have a representation that requires the facets of the 3D boundary to be formed into shells and lumps, or for the half-edges that bound a facet to be formed into loops and regions (c.f. ACIS [CL01]).

Chapter 6

Simplification—requirements and issues

So far I have presented the basic Boolean algorithm for performing Boolean operations on polyhedral solid representations. It has the advantage over other algorithms using standard machine arithmetic in that correct connectivity in the input guarantees correct connectivity in the result. However, marginal features, in particular gaps and slivers, that arise from the exact operation can be sufficiently distorted by the use of rounded arithmetic to make the result geometrically invalid. The data-smoothing post-process described in the last chapter, acts to resolve marginal data; though not fully robust, in practice it is highly reliable, making it well suited for use within the CAD commercial product for which it was designed.

It remains an open issue as to what approach(es) are to be preferred for resolving geometrically invalid and marginal data produced by the basic Boolean operation. The formation of geometrically invalid shape representations is inevitable not just for the basic Boolean algorithm, but for geometric operations in general, if approximate machine arithmetic is used (see section 2.3.5, p.35). A marginally geometrically valid feature can cross over to being marginally geometrically invalid, and the greater the amount of processing required, the greater the possibility that the extent of the geometric error will exceed the ‘marginal’ status, however we choose to define that. Since data-smoothing is a sequence of adjustments rather than a precisely defined operation, the exact details of how it is implemented can greatly affect the process in terms of speed, distortion in the final result, and the risk of complexity and non-termination. The behaviour of the operation is likely to be strongly influenced by the nature of the input data expected to be handled. It is difficult to envisage how a data-smoothing operation could be engineered to ensure that it *always* produces a satisfactory result in reasonable time (assuming sufficient memory).

A particular shortcoming of data smoothing is that it does not aim to resolve non-

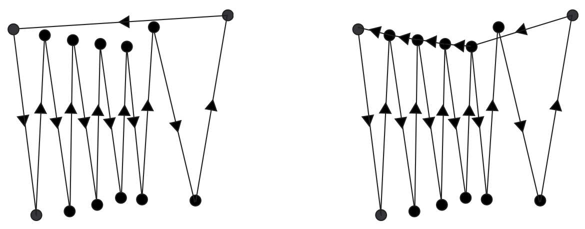


Figure 6.1: *2D example showing that the data-smoothing process can make a shape representation geometrically invalid. A series of adjustments starting at the left and working right drag down the top edge by a total distance exceeding the data-smoothing tolerance value. A vertex that was within tolerance distance of the original top edge is no longer within tolerance distance of the adjusted edge, so no further adjustment is made.*

marginal, geometrically invalid data. This is potentially a concern, because the data-smoothing process itself can *create* geometrically invalid regions even when the initial structure is fully geometrically valid. Furthermore, there is no limit to the extent of the error; whatever criteria we select to define data as marginal, there will always be cases for which the result includes a region that is geometrically invalid to a non-marginal degree. Figure 6.1 shows an example in 2D of how it is possible for the shape boundary to be distorted significantly by a series of adjustments.

Since geometric operations of all types are liable to lead to geometric errors, and tolerance-based adjustments intended to resolve marginal data are potentially liable to degrade the data further, it is reasonable to seek some means of resolving geometric errors that is not based on the assumption that all geometric errors are marginal. One well-defined operation that can resolve geometric errors in theory (assuming an exact implementation) is the process of *simplification* as defined by Fortune [For95, pp.228-229].[1] This operation modifies the boundary for the shape representation in such a way that regions that are multiply enclosed become singly enclosed, and regions that are enclosed in inside-out mode cease to be enclosed. The action required to carry out the simplification process is similar to that of the Boolean operation: the boundary components are broken up where they intersect other boundary components, and the boundary for the result is constructed from the boundary components after splitting.

Ideally, we would like an algorithm for simplification using floating-point arithmetic that is guaranteed to generate a fully valid shape representation (see section 3.1, p.47). Considering the similarity between simplification and the Boolean operation, it might appear reasonable to suppose that an approximate arithmetic algorithm for simplification could be devised that benefits from the techniques used for the basic Boolean algorithm. Recall

[1]The term ‘simplification’ is very often used to refer to a totally different process in which a boundary mesh is approximated by a coarser mesh in order to speed up rendering and transmission of mesh data—see, for example, [PS97]. Throughout this document I use the term in the sense defined by Fortune.

that the basic Boolean algorithm benefits from the 'simulated simplicity' usually only available for exact algorithms (see [EM90]); furthermore, the formulaic nature of the algorithm gives a wide degree of freedom as to how to implement the algorithm, in terms of what order to carry out the computations and what data to cache. However, a closer examination of the issues shows there to be no obvious equivalent algorithm with the same benefits. The crucial difference between the two operations is that while the simplification process needs to determine *all* boundary self-intersections, the basic Boolean algorithm operates by determining only the intersection between two boundaries, without considering whether either boundary has self-intersections.

This chapter introduces the simplification process and discusses the issues that influence the design of a floating-point implementation, in preparation for the following chapter that describes an algorithm for simplification. Section 6.1 describes the process and its uses. Section 6.2 discusses the arrangement problem [Hal04], which is closely related to the simplification process, and is helpful in understanding the issues. Section 6.3 discusses the problems relating to consistency and cell formation that can arise when performing simplification. Section 6.4 explains why the basic Boolean algorithm is not affected by the same problems. Finally, section 6.5 discusses how a simplification algorithm would be expected to handle coincident data.

6.1 The simplification process and its uses

In general terms, simplification takes as input a bounded oriented surface in 3D (or curve in 2D) that is topologically 'closed' in the sense of being without frontier, but which may intersect itself. Given such a surface, any point in space not on the surface has a well-defined winding number in relation to the surface (as in [Hai89], see section 4.5, p.66). For the standard form of the operation, the result is defined as the set of points in space for which the winding number with respect to the input surface is strictly positive. See figure 6.2(a)-(c) for an example of how simplification in this form works on a 2D polygonal representation. As with the Boolean operation, one may assume the result to be regularised, so that it is clearly defined which parts of the original surface are included in the result.[2] This standard form of the simplification operation can be regarded as a minimal 'best fix' to a boundary representation, converting multiply-enclosed regions to singly enclosed (interior) regions, and inside-out regions to unenclosed (exterior) regions. In terms of structure, the result is represented by its boundary, and is constructed from

[2]If the convention taken is to represent only closed regular shapes [Req77, Req80], a surface point is included in the result if and only if for any real $\varepsilon > 0$ there exists a non-surface point within distance ε deemed included in the result by virtue of its winding number.

those parts of the original ‘boundary’ that separate a region to be included in the result (by virtue of its winding number) from one that is not.

Simplification can also be used to implement other tasks. In particular, it can be used to implement all the types of Boolean operation. The union operation can be implemented by simply assembling the boundary components of the two input shapes into one representation and then applying simplification in its standard form. The difference operation can be implemented in a similar way, except that the components copied from the boundary of the second shape must have their orientation reversed. The intersection operation, like the union operation, can be implemented by assembling boundary components in their original orientation, but a variant form of simplification is required in which regions of winding number strictly greater than 1 are selected [For95, p.228]. The symmetric difference operation $(A \setminus B) \cup (B \setminus A)$ can be implemented using another variant form of simplification, in which regions with an odd-valued winding number are selected; figure 6.2(d) shows the result of this variant form of simplification when applied to figure 6.2(a). Implementing Boolean operations by assembling the two boundaries and simplifying has the advantage that the algorithm acts to resolve geometrically invalid regions; in contrast, the basic Boolean algorithm is merely tolerant to such regions, leaving them geometrically invalid, as was shown in figure 4.9, p.73.

The simplification process does not resolve marginal data, and is not designed to do so. In principle, however, the simplification process can form the basis for other operations, such as offsetting, filleting and epsilon-regularisation. The last of these processes is capable of resolving marginally valid data. This is discussed briefly in section 9.2.3, p.166.

6.2 The arrangement structure

The shape generated by the exact simplification process is closely related to the cellular structure formed by the supposed boundary of the initial (possibly geometrically invalid) shape representation. Indeed, the resulting shape can be defined in terms of this implied structure. Although it is not necessary to represent the cellular structure in full in order to perform the simplification process, it is helpful to understand the simplification process in terms of the task of determining the cellular structure. The formation of cells by the initial ‘boundary’ is a special case of the arrangement problem, [Hal04, FHK$^+$07], which I now describe.

In general terms, an *arrangement* is the subdivision of a geometric space induced by a finite collection of geometric objects. It is usual to consider only collections of hypersurfaces; these may be linear or curved, bounded or unbounded, and with or without a border [Hal04].

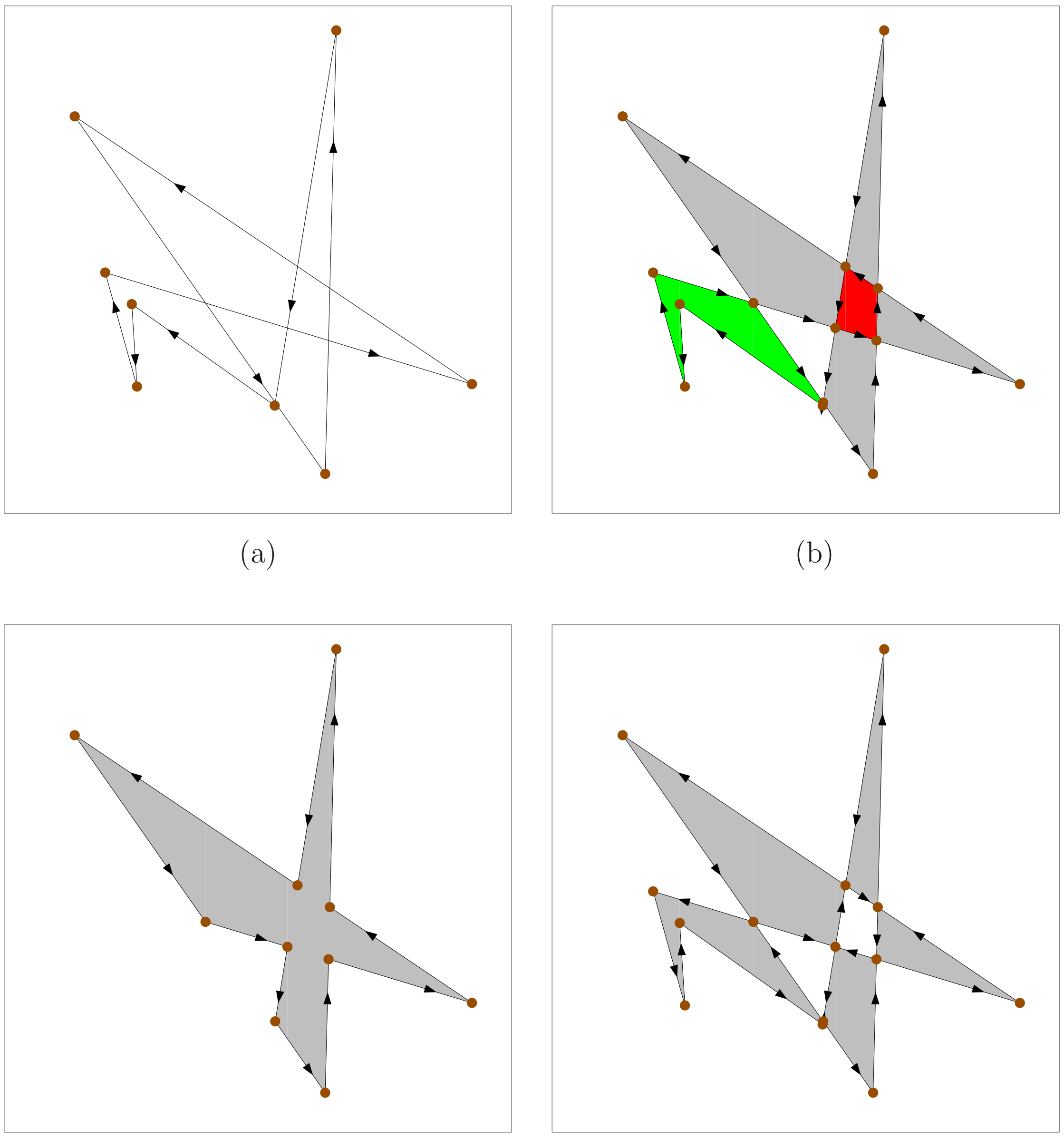


Figure 6.2: *Example showing in principle the simplification process for a geometrically invalid shape representation. The initial topological octagon is shown in (a). In (b), all boundary intersection points are determined, and the edges concerned are split at each intersection point. The remaining edges form a plane graph, with each face having a well-defined winding number (green for -1, white for 0, grey for 1, red for 2). (c) shows the result of the standard operation, in which cells with winding number* > 0 *are retained; (d) shows the result of the variant operation in which cells with odd winding number are retained. In all cases, the result is constructed from those edges separating a retained face from a discarded face, the direction of each edge being set so that the retained face lies to the left.*

Let $\mathcal{S}$ be a finite collection of hypersurfaces ($(d-1)$-manifolds) in $\mathbb{R}^d$. The hypersurfaces in $\mathcal{S}$ induce a decomposition of $\mathbb{R}^d$ into a number of regions, each defined by a particular subset $\mathcal{S}^* \subseteq \mathcal{S}$:

$$A(\mathcal{S}^*) = (\bigcap_{H \in \mathcal{S}^*} H) \cap (\bigcap_{H \in \overline{\mathcal{S}^*}} \overline{H}) \tag{6.1}$$

where $\overline{\mathcal{S}^*} = \mathcal{S} \setminus \mathcal{S}^*$ and $\overline{H} = \mathbb{R}^d \setminus H$. Note that this includes the region defined by the empty subcollection, Φ, which simplifies to $A(\Phi) = \bigcap_{H \in \mathcal{S}} \overline{H}$.[3] Each region $A(\mathcal{S}^*)$ that is not empty can be considered subdivided into cells, in which each cell is a maximal connected subset of the region concerned. The arrangement induced by $\mathcal{S}$ is defined as the assembly of all the cells from all the regions defined. In general, for non-degenerate cases, these cells have manifold dimensionality $0, 1, ..., d$ [Hal04].

The d-cells are formed from $A(\Phi)$, and each d-cell is bordered by $(d-1)$-cells that separate it from neighbouring d-cells. In general, each $(d-1)$-cell originates from the region for a single-component sub-collection of $\mathcal{S}$, but it may originate from the region for a larger sub-collection in degenerate cases when the components of the sub-collection overlap. Each $(d-1)$-cell is a subset of one of the hypersurfaces (at least) from $\mathcal{S}$.

Let us now consider the simplification process in terms of the arrangement problem. For the exact process, the resulting shape can be constructed from the arrangement structure induced by the $(d-1)$-manifold components of the presumed 'boundary'; each component is assumed to be closed, so that it includes the $(d-2)$-manifold border. The boundary for the result of the operation is constructed from $(d-1)$-cells within the arrangement. It is not necessary to compute the structure of d-cells in order to perform the operation; however, for each $(d-1)$-cell in the arrangement it is necessary to obtain the winding number of the d-cells on either side, and some form of navigation of the implied structure is required to compute this value. The orientation of each of the boundary components is of importance when determining the d-cell winding numbers. By convention, the original boundary components are oriented to face the supposed shape exterior; for a geometrically invalid shape representation they indicate the direction of traversal that results in a decrement in winding number. The $(d-1)$-cells are considered to inherit the orientation of the component from which it originates; for a $(d-1)$-cell that originates from two or more boundary components that overlap, the increments and decrements for each component are considered to apply in aggregate across the $(d-1)$-cell. Having determined the winding numbers, the resulting shape is the closure of the assembly of those d-cells for which the winding number is appropriate. The result is represented by the shape boundary, built from the $(d-1)$-cells in the arrangement that separate an included and excluded d-cell; each $(d-1)$-cell is oriented to face the excluded d-cell.

[3]The regions $A(\mathcal{S}^*)$ form a decomposition, since for any point $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{x} \in A(\mathcal{S}^*)$ holds true uniquely for $\mathcal{S}^* = \{H \in \mathcal{S} : \mathbf{x} \in H\}$.

Our concern is with how to devise a robust algorithm for simplification if the real arithmetic calculations performed are approximate. Provided the implied structure represents a valid cellular subdivision with winding numbers, it is a straightforward logical task to determine a representation for the result. However, for an approximate implementation it is possible for the implied structure to have logical faults that prevent it from truly representing a subdivision; when this happens it becomes impossible to assign a winding number to each cell in a consistent manner. The need to avoid such errors is essentially a topological constraint, which is additional to the constraints that apply to a boundary representation, such as those described in section 4.2, p.55, for the 2D and 3D linear cases. The next section describes in detail the errors that can occur for the 2D linear case, and how they affect the assignment of winding numbers.

6.3 Problems in implementing 2D simplification

For the simplification process in the 2D linear domain, the representation of the shape boundary at the start and end of the operation is a collection of directed edges, as described in section 4.2, p.56. This can be considered as a *simple graph* with nodes and connecting edges, with the nodes representing vertices, and the (graph) edges representing (model) edges. The arrangement, however, is an embedded *planar graph* that includes faces to represent the 2D polygonal cells (see [MN99], for example).[4] While a simple graph edge acts simply to connect two nodes, a planar graph edge also separates two faces. Furthermore, while the edges that connect to a node have no implied order in a simple graph, in a planar graph they have a specified cyclical ordering that is relevant to the structure.

An edge can be considered as two directed half-edges going in opposing directions, the half-edges form an implied structure known as a *doubly-connected edge list* or *DCEL* as described in [dBvKOS97, pp.29-33]. An example of such a structure is shown in figure 6.3. The half-edges form both vertex loops and face loops. All half-edges that start at a particular vertex are considered to form one vertex loop, with each half-edge being succeeded by the neighbouring half-edge to the right (thereby forming a clockwise ordering). A face loop is formed by half-edges that have the same face to the left, with each half-edge being succeeded by the half-edge that starts at its end-vertex.[5] A face may have more than one face loop; every face has exactly one outer loop, except for the unbounded face that extends to infinity, which has none.

[4]In graph theory, a 'planar graph' is a simple graph that can be embedded onto the plane, but without any indication how. I use the term for a structure for which the face formation is known and represented.

[5]For paired half-edges h_A and h_B, the half-edge that succeeds h_A in a face loop succeeds h_B in a vertex loop.

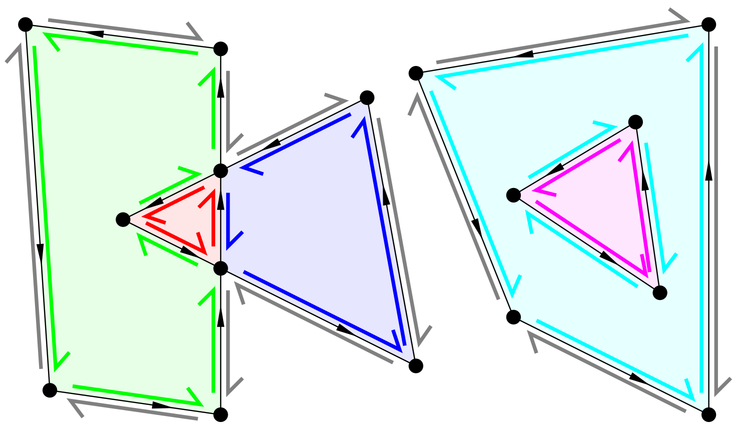


Figure 6.3: *An example showing a doubly-connected edge list (DCEL). Half-edges are drawn as hooked arrows, and each is coloured according to which face it borders. In this example the half-edges form eight face loops bordering six faces; the face that extends to infinity (half-edges drawn in grey) and the face with half-edges coloured cyan (pale blue) each have two face loops.*

As with the Boolean operation, it is necessary for the simplification process to split an edge at each point where it intersects another edge. However, this action alone is not sufficient, because it merely extends the original (simple) graph representing the boundary; it does not determine the planar graph required to represent the arrangement and thus the cellular subdivision. For an exact implementation, the splitting of edges at intersection points will lead to a boundary representation without (internal) intersections, and from that it is possible to determine the faces formed. However, there are problems for an approximate implementation. Firstly, the cyclical ordering of edges about a vertex is not necessarily well-defined as it would be in the exact domain. Furthermore, the location of intersection points is often not exact. This changes the geometry of the boundary, and new boundary self-intersections can form; it can also leave open to question how the intersection points associated with one particular edge should be ordered along that edge.

6.3.1 Triples of data entities

When several entities lie close to each other, computing the relationship between pairs of entities using rounded arithmetic can lead to 'impossible' combinations that one would not expect with exact geometry. I show here that this can arise even for just three entities. I make no assumption about coding details here; later, in section 7.2, I give real examples that lead to invalid arrangements, based on a particular coding for the vertex-edge shadow relationship and the use of IEEE arithmetic.

Consider the situation in which two edges overlap over a range in x. Under the rules of exact arithmetic, if one edge is above the other at each of the two extremes of the x range (i.e. has a higher y-coordinate value for the x-coordinate concerned), then it is

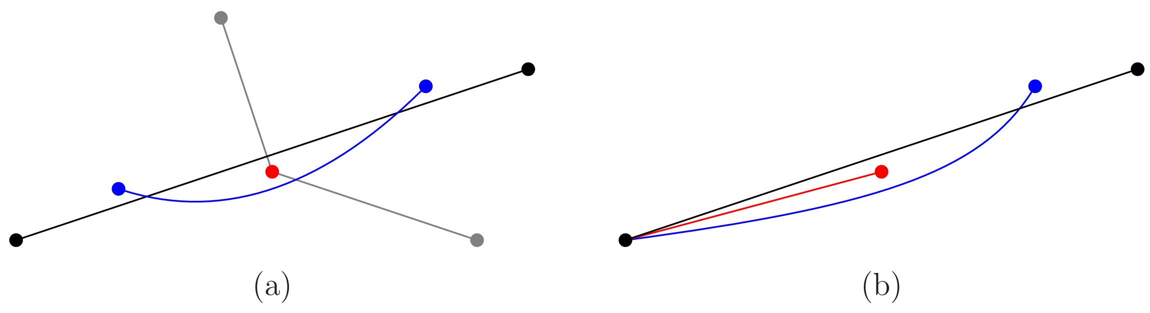


Figure 6.4: *Examples of situations that are impossible with exact geometry, which might arise when rounded arithmetic is used. All edges are in fact straight, even though some are drawn curved. Though the blue edge is determined to be above the black edge, the red vertex is determined to be above the blue edge and below the black edge.*

not possible for a vertex in the x coordinate range to lie both above the upper edge and below the lower edge. However, we have to be alert to the possibility of such a situation arising according to the rounded arithmetic calculations, as in figure 6.4(a). The same argument applies to two edges with a shared end-vertex, as in figure 6.4(b). The latter case is a particular concern because it shows that the computations will not necessarily give a definitive cyclical ordering of edges about a common end-vertex.

Next, consider the situation in 2D in which there are three intersecting edges, so that for each pair of edges there is a single point of intersection internal to both edges. With exact geometry, the three points of intersection are either all identical, as shown in figure 6.5(a), or else they are all distinct, as in figures 6.5(b)&(c). It is not possible for two of the intersection points to be identical and the third one distinct. Furthermore, if the intersection points are distinct, there are constraints on their geometric positioning. Suppose that the three edges are listed in order of gradient, i.e. dy/dx, with a vertical edge (if present) placed last. Suppose also that the intersection points are listed in lexicographical order, according to x coordinate values in the first instance, and then according to y coordinates in the event of a tie. It follows that the intersection point between the first and last edges, as ordered by gradient, will lie lexicographically in between the other two intersection points.

This geometric constraint on the lexicographical ordering of the intersection points in turn places a topological constraint on the ordering of points on each of the three edges. Let us suppose we name the 3 edges in order of gradient as AB, CD and EF, where $A \prec B$, $C \prec D$ and $E \prec F$, where '$\prec$' designates lexicographic precedence. Let us designate P as the intersection between CD and EF, Q as the intersection between EF and AB, and R as the intersection between AB and CD. This gives just two possible orderings of points in the three edges when P, Q and R are distinct: $ARQB$, $CRPD$ and $EQPF$, as in figure 6.5(b), or $AQRB$, $CPRD$ and $EPQF$, as in figure 6.5(c). For either case, the

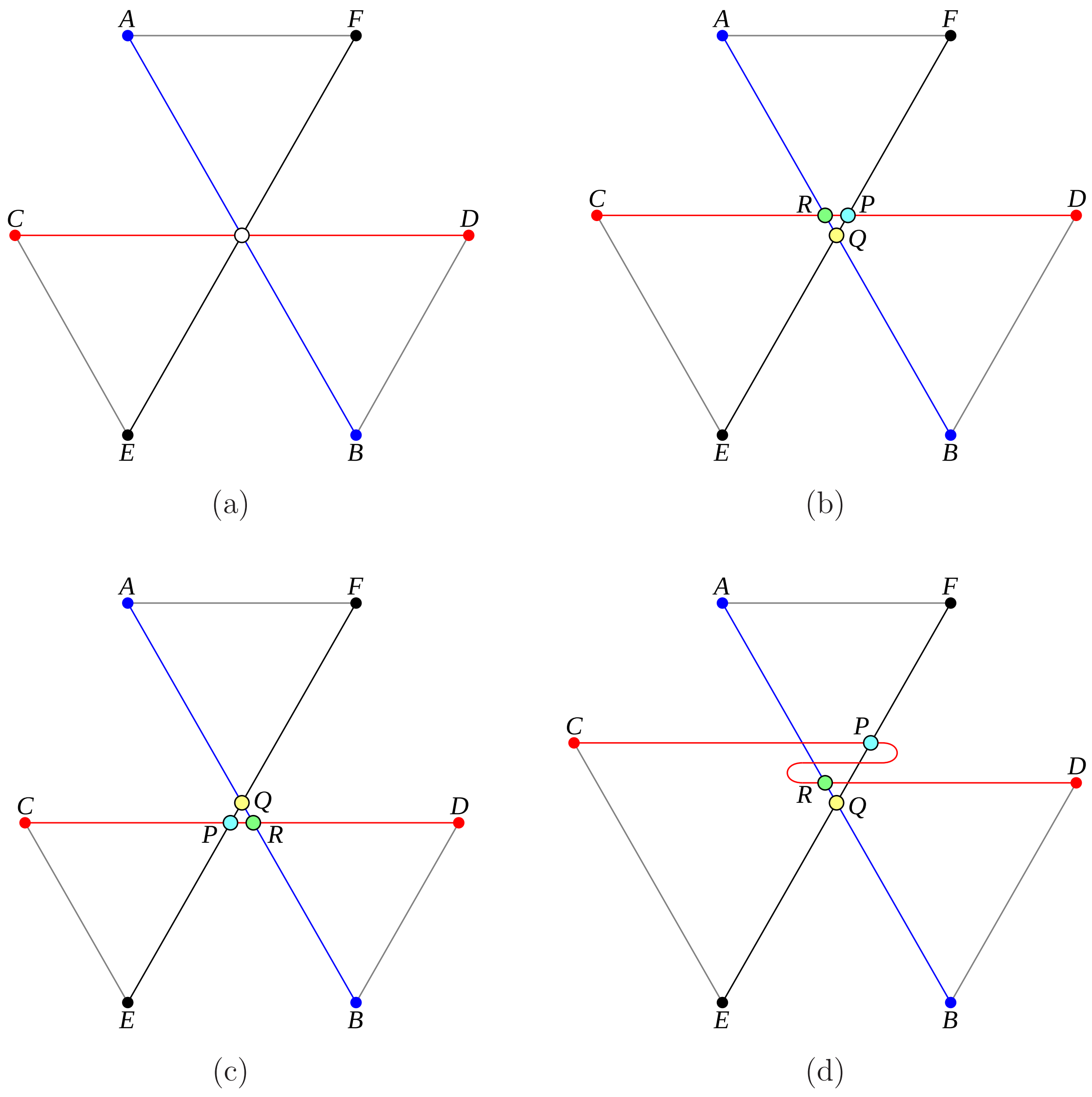


Figure 6.5: *(a), (b) and (c) show the three possible configurations that are possible when three edges intersect, with each pair of edges intersecting at a single point interior to both edges. (d) is a configuration that is not possible with exact geometry. The points of intersection are marked with an encircled dot (the extra line crossings in (d) being artefacts).*

points P, Q and R are ordered in a clockwise orientation.

When the approximate arithmetic calculations deem two edges to intersect, the computed point of intersection will in general not be exact, in which case it will lie off one or both edges. Furthermore, each calculation concerning the intersection between a particular pair of edges is independent of any other such calculation. Hence, when three edges intersect each other, we cannot guarantee that the three points of intersection, if they are distinct, are lexicographically ordered in one of the two permitted orders; nor can we guarantee that their orientation order is correct. A further difficulty is that the intersection point computations do not provide information concerning how the points of intersection relating to one specific edge are to be ordered. Lexicographical ordering, or any other ordering based on numerical result, may lead to an invalid planar subdivision structure. Figure 6.5(d) shows an example of an invalid structure that could be constructed from three edges that intersect close to each other.

A situation such as shown in figure 6.5(d) is unsatisfactory, not just because it looks wrong, but because it leads to logical problems. For any boundary structure for which the cyclical ordering of half-edges is supposedly known it is possible to identify facet loops by adhering to a 'keep left' policy. We do 'know' the cyclical ordering of all the vertices in figures 6.5(b), (c) and (d): the original vertices are connected only by two edges; and the ordering of the four edges about an intersection vertex is clear from the type of intersection between the two original edges.[6] The 'keep left' policy works fine for figures 6.5(b) and (c). Figure 6.5(b) has single-loop faces $AFPDBQECR$ (the external face), $ARPF$, $BDPQ$, $CEQR$ and PRQ. If the original edges are directed as $\overrightarrow{AB}$, $\overrightarrow{BD}$, $\overrightarrow{DC}$, $\overrightarrow{CE}$, $\overrightarrow{EF}$ and $\overrightarrow{FA}$, then the faces as identified have respective winding numbers of 0, 1, 1, 1 and 2. However, carrying out the same exercise for figure 6.5(d) gives face loops $AFPR$, $ARDBQECPF$ and $BDRQPCEQRPQ$. This face loop formation has a nonsensical geometry; more significantly, it has a bad topology that prevents the process of simplification from generating any result, accurate or inaccurate. The winding number for the face on the left-hand side of the directed edge $\overrightarrow{RQ}$ is supposed to be one greater than that for the face on the right-hand edge, but in this example that cannot be so because the half-edges on either side both belong to the third of the face loops listed. Consequently, half-edges cannot be assigned winding number values needed to allow the simplification process to complete.

The rules for topological validity given in section 4.2 only suppose a simple graph structure. Under these rules, the structure in figure 6.5(d) is classed as topologically valid because each vertex acts as a start-vertex to an edge as many times as it acts as an end-vertex. So we see that not only does the simplification process impose on us the need for a notional cell structure to manage the arrangement calculation; it also imposes a need for the cell structure to adhere to a valid cell topology in addition to the topological constraints identified in section 4.2. This holds irrespective of how geometric rounding errors are managed.

To conclude, we have seen that triples of entities—two edges and a vertex, or three edges—have the potential to lead to inconsistencies, and at worst, to introduce bad cell topology that would make it impossible for the algorithm to progress. It is this difficulty that hinders the devising of an approximate arithmetic simplification algorithm resembling the basic Boolean algorithm and with the same benefits.

[6]The sign of the intersection function $X_{22}(e_A, e_B)$ as defined in table 4.1 tells us which edge crosses the other from left to right.

6.4 Why the basic Boolean algorithm is not affected

We have observed the complications in determining the arrangement formed by the boundary of a polygonal shape representation, if approximate arithmetic is used. The intersection calculations for three edges may lead to a bad cell formation, and this presents problems for implementing the simplification process, which needs to determine the winding numbers on either side of each edge. The 2D Boolean operation can also be expressed in terms of the arrangement induced by the boundaries of the two initial shape representations. However, the basic Boolean algorithm is an implementation of the Boolean operation that does not rely on determining the arrangement; therefore, it is immune from problems relating to invalid cell formation in the structure assumed to represent the arrangement. We briefly consider here why the basic Boolean algorithm is able to do this, and why a similar technique cannot be used for the simplification process.

In order to determine the arrangement structure, it is necessary to compute the intersection status for every possible pair of edges. With the basic Boolean algorithm, however, it is only necessary to compute the intersection status for pairs of edges originating from different input objects. When two edges from the same input object intersect, they are not split at the intersection point; they are processed separately, irrespective of their status. These edges (or their retained parts) may well continue to intersect undetected in the representation of the result. In consequence, the potential problem relating to three intersecting edges does not arise. Two of the three edges must originate from the same object; hence, only two out of the three possible edge pairings are tested for intersection.

Let us consider the Boolean operation in its exact form as a pointwise mapping from the winding numbers for the two input objects, $a(\mathbf{x})$ and $b(\mathbf{x})$, to the winding number for the resulting object, $\phi(\mathbf{x})$ (disregarding points on the boundary of either input object, which are subject to regularisation). The Boolean operation concerned defines the mapping for the four cases for which $a(\mathbf{x})$ and $b(\mathbf{x})$ equal 0 or 1. Because of the possibility of geometrically invalid input objects, we are obliged to extend the definition of the mapping to apply to all integer value pairs for $a(\mathbf{x})$ and $b(\mathbf{x})$. In principle at least, we are free to choose how to extend the mapping for non-standard cases. Recall, however, that for the basic Boolean algorithm, the mapping is expressed by equation (4.20), p.73:

$$\phi(\mathbf{x}) = c_A a(\mathbf{x}) + c_B b(\mathbf{x}) + c_I a(\mathbf{x}) b(\mathbf{x}) \tag{6.2}$$

where the constants c_A, c_B and c_I depend on the Boolean operation type, as specified in equations (4.11) to (4.13). Let us consider the significance of this formula as we imagine the point $\mathbf{x}$ crossing a single edge belonging to the first object, A, from its exterior side to its interior side. The change in location leads to an increment in $a(\mathbf{x})$ by 1, while

$b(\mathbf{x})$ remains unchanged. In consequence, $\phi(\mathbf{x})$ is incremented by $c_A + c_I b(\mathbf{x})$. This value represents the number of times that the edge concerned (or a portion of it) must be represented within the computed result of the operation. We may note that this number is independent of the value $a(\mathbf{x})$ on either side of the edge. Hence, there is no need to determine self-intersections for the boundary of object A. Conveniently, this has the effect of sidestepping any problem that might arise from the computed intersection status between these two edges being at variance with the computed intersection status for the other two edge pairs. The same argument holds true for the boundary of object B, if we imagine $\mathbf{x}$ crossing a single edge from B. It is this feature that enables the basic Boolean algorithm to consider the status of each boundary component individually, in isolation of the structure to which it belongs.

The disadvantage of the basic Boolean algorithm is that the generated result may be geometrically invalid, because $\phi(\mathbf{x})$ may take a value other than 0 or 1. Unfortunately, it is not possible to use any other formula for the mapping of $a(\mathbf{x})$ and $b(\mathbf{x})$ to $\phi(\mathbf{x})$ without also making it necessary to determine boundary self-intersections in the original shape representations.

By similar reasoning, it is not possible to devise a simplification algorithm that is able to avoid computing the intersection status for all edge pairs. We may consider the operation as a mapping from the winding number for the input object, $a(\mathbf{x})$, to the winding number for the resulting object, $\phi(\mathbf{x})$. We require that geometrically valid objects remain unaltered; hence 0 must map to 0, and 1 to 1. We want the increment of $\phi(\mathbf{x})$ with respect to $a(\mathbf{x})$ to be independent of $a(\mathbf{x})$; the only solution to this is $\phi(\mathbf{x}) = a(\mathbf{x})$. This is unsatisfactory because it represents a simple copying of the object concerned and therefore fails to resolve any geometric invalidity.

6.5 Coincident data

Given that there is no obvious way to implement the 2D linear simplification process using a formulaic approach of the basic Boolean algorithm, some other approach is required. In this section I consider what rules are appropriate for handling coincident data, in preparation for the description of the sweep-line algorithm given in the chapter that follows.

The basic Boolean algorithm made use of symbolic perturbation, which resolved the problem of degeneracy, and also ensured that the implementation was relatively straight forward. Use of symbolic perturbation for the simplification process is more complicated, and does not bring the same benefits. Symbolic perturbation also destroys the symmetry of the operation, which was a price worth paying for the basic Boolean algorithm because of the benefits. I therefore considered it appropriate for the algorithm for simplification to perform special case handling whenever two entities coincided in a degenerate manner in full 2D space, or more precisely, whenever the floating-point calculations deem them to coincide.

Nevertheless, symbolic perturbation is used for the simplification process in one respect. This relates to the ordering of points to be processed for a sweep-line implementation of the process. For a sweep-line algorithm, a line is considered to sweep across the plane, and the data structure relating to the algorithm is modified as the line crosses the points to be processed, known as *event points*. It is an established technique to assume the sweep line to be symbolically perturbed, so that the algorithm may determine in which order the points should be processed when the line reaches two or more points simultaneously [dBvKOS97]. If the sweep line is a vertical line going left to right, but considered to have an infinitesimally small backwards tilt, the event points are processed in order of ascending x-coordinate value, with any event points sharing the same x-coordinate value being processed in order of ascending y-coordinate values. This is known as *lexicographic ordering*, which can be viewed as ordering points according to ascending $x + \varepsilon y$, where ε represents a symbolic, infinitesimally small positive value.[7]

I now consider the different types of coincidence in full space, and how they are to be handled for simplification. I describe the algorithm not in terms of the simplification process per se, but rather in terms of determining the cellular subdivision and the winding number for each face formed. Once the cellular subdivision is determined it is a straightforward logical task to determine the simplified shape. Determining the cellular subdivision is largely the same as determining the arrangement structure. The main difference is that extra processing is required to handle the winding numbers of faces and also the increment in winding number across edges. It is also possible for the essential structure of the cellular subdivision to differ from that for the true arrangement, since in certain circumstances (described shortly) it is possible for edges in the cellular subdivision to cancel each other out.

Listed below are the types of coincidence, both non-degenerate and degenerate, that can arise in the 2D problem. I explain how coincidence is determined, and describe the actions required to resolve data locally. The tests and actions to be applied are essentially those that would apply for the exact problem, though the tests for coincidence differ in detail in that they rely on approximate arithmetic and may not yield the correct answer.

Vertices coinciding. Though this may be considered degenerate for a pure arrangement problem, it is of course standard for the problem of simplification because of the topological constraints on the input. Vertices that are located at (exactly) the same point are

[7]In [dBvKOS97], the sweep line is horizontal and goes from top to bottom; hence, points are processed in order of descending $y - \varepsilon x$.

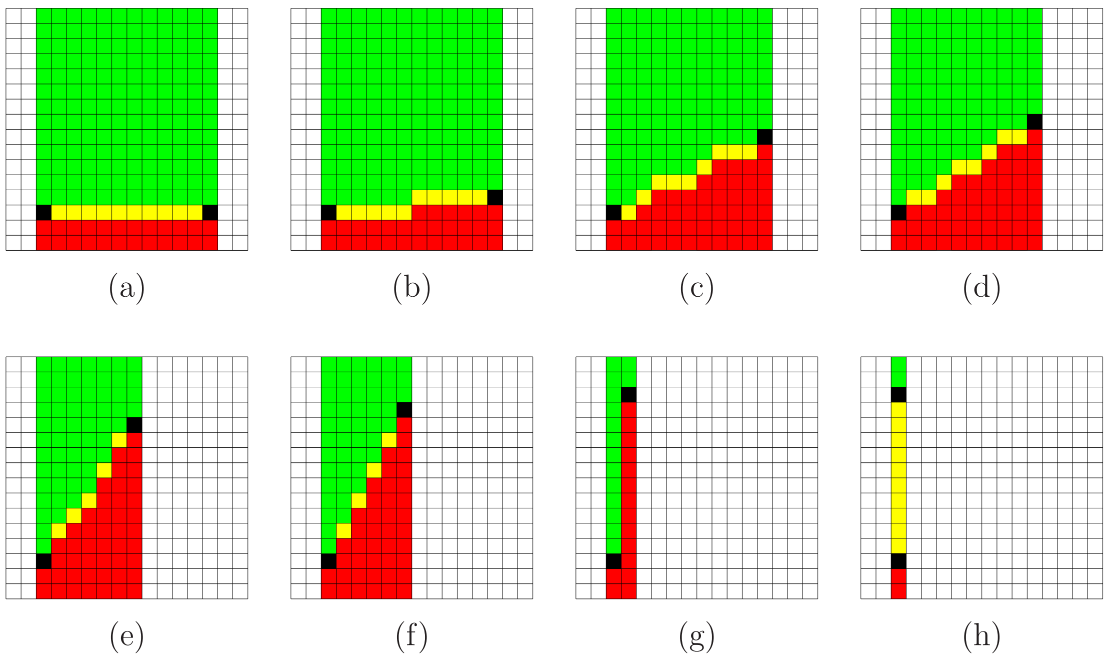


Figure 6.6: *A series of example cases on a 16×16 grid of representable points, showing the two vertices that define an edge in black and the points deemed to lie on the edge interior in yellow. Except for the vertical edge shown in (h), there is exactly one representable point on the edge for each representable x value within range. All other representable points with an x value within range are identified as being either below or above the edge; these are marked red and green respectively.*

considered identical. Hence, all edges with a vertex at the same point are to be considered connected at the common vertex. Any edge starting and ending at the same point is zero-length and is discarded.

A vertex lying on the interior of an edge. There is no fixed method to determine whether a vertex lies on an edge, and two methods may well yield different results for a specific case. The rule arbitrarily adopted to signify that vertex (x_P, y_P) lies on a non-vertical edge connecting (x_L, y_L) and (x_R, y_R) where $x_L < x_R$ is that y_P should be exactly equal to the interpolated y value (as computed) on the edge at $x = x_P$. The code for the interpolation calculation is given later, in section 7.1.1. For a vertical edge with $x_L = x_R$ and $y_L < y_R$, the vertex is considered to lie on the edge if $x_P = x_L$ and $y_L < y_P < y_R$. See figure 6.6 for example cases. Whenever a vertex is deemed to lie on an edge interior according to these rules, the edge is replaced by two edges, linking the vertex concerned to each end-vertex of the original edge.

Two edges intersecting each other at a point interior to both edges (non-degenerate). This situation is determined by the vertex-edge shadow relationships—i.e. whether a vertex lies below or above a vertex. Cases involving coincident vertices need to be excluded in order to avoid looping; cases whereby a vertex lies on an edge interior are

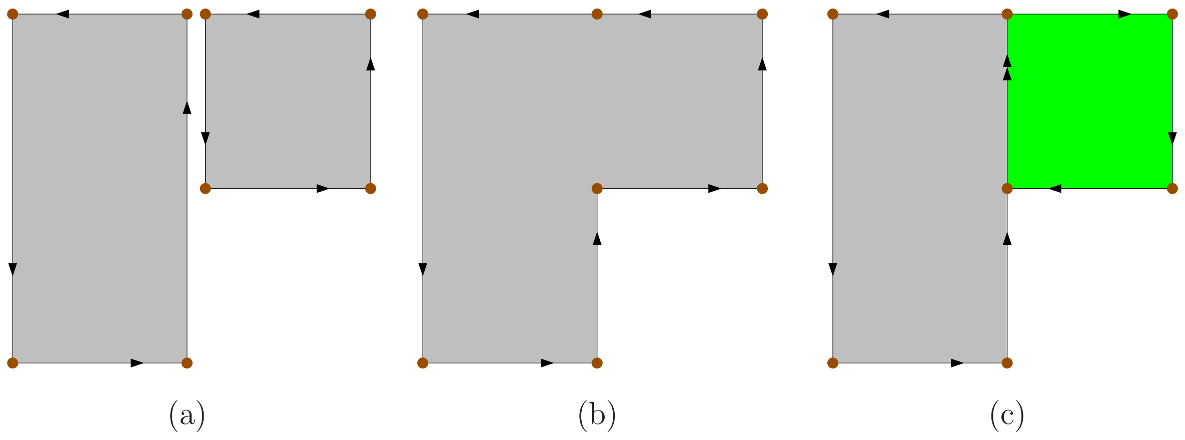


Figure 6.7: *Merging of coincident edges by simplification process. (a) The initial set of boundary edges. (The two partially coinciding edges and the two coinciding vertices are drawn apart for illustration.) (b) In consequence of the boundary self-intersection process, the longer of the overlapping edges is split at the vertex, and the upper edge that remains cancels the other original edge. (c) In this example the edges of the right-hand rectangle are reversed before the boundary self-intersection process. The two edges are merged to give a single edge of multiplicity* 2 *(marked by a double arrow-head).*

also excluded, as these can be handled in advance by the vertex-on-edge operation just described. When two edges are determined to intersect, a new vertex is created at the point of intersection; also, each edge is replaced by two edges, linking the new vertex to each of the original four vertices.

Two edges coinciding over a finite-length segment, in part or in full. This situation implies a vertex-vertex or vertex-edge type degeneracy occurring at each end of the overlapping segment. The advance processing of these two degeneracies by the operations described above will lead to there being two edges coinciding in full over the segment; that is to say they will both be defined by the same pair of points to be their start- and end-point, maybe going in the same direction, or maybe going in opposing directions. The initial action relating to these fully coinciding edges is to combine them into one edge; this is as one would expect for the pure segment arrangement problem, in which a segment acts solely to demarcate the boundary between two cells formed by the arrangement.

However, there is a further matter to consider concerning the role of an edge, and how that role is represented. Recall that an edge acts to demarcate between two regions of different winding number. So far we have considered an edge as marking an increment in winding number by the value 1 as one crosses from the right-hand side of the edge to the left (as seen when going from start- to end-point). If we suppose the two original edges both mark an increment in winding number by 1, then there are two scenarios for the merged edge: if the original edges go in the same direction, the merged edge marks a winding number increment of 2; if they go in opposing directions, then there is no difference in winding number across the merged edge. This makes it necessary to be able to represent edges with an arbitrary winding number increment value. Sometimes it may be necessary to merge an already merged edge with a third edge, and so on. Generally speaking, when merging two edges, the winding number increment of the merged edge is equal to either the sum or difference of the winding number increments of the two edges being merged, according to whether they go in the same or opposing directions. The increment in winding number represented by a particular edge is referred to as its *multiplicity*.

Considering that the purpose of the boundary is to demarcate between regions with different winding numbers, an edge with zero multiplicity is redundant. Hence, when edges are merged to create one with zero multiplicity they are considered to cancel each other out, and no edge is left remaining. Figure 6.7 shows what can happen to coinciding edges, initially with multiplicity value 1, during the boundary self-intersection process. Note that because of the removal of edges of multiplicity 0, the boundary self-intersection process for a polygonal representation is not exactly equivalent to the segment arrangement problem.

This completes our initial consideration of the simplification process. Simplification can be used to perform a number of tasks; most notably, it can be used to tidy up shape representations that have become geometrically invalid due to data degradation—a task that cannot be carried out using the basic Boolean algorithm. We have seen that it is necessary for a simplification algorithm to maintain a planar subdivision with a valid cell formation—a constraint that was not necessary for the basic Boolean algorithm; consequently, it is not feasible to create a robust algorithm for simplification with the formulaic simplicity of the basic Boolean algorithm. The chapter finished with a consideration as to how the simplification process can handle coinciding entities. We are now in a position to consider the implementation of a simplification algorithm in more detail.

Chapter 7

An approach to 2D simplification

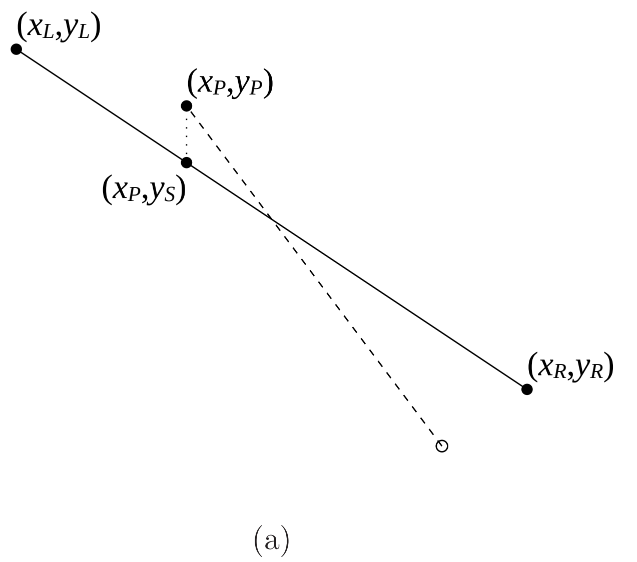
Having considered issues that influence the design of an algorithm for simplification, I now present in some detail a sweep-line algorithm for the process.

The algorithm presented is one of a number of candidate algorithms I investigated. All algorithms considered were found to have problems. Certain algorithms tried cannot strictly be classed as robust by the criteria listed in section 3.1, p.47. They behave unacceptably in certain circumstances: either the algorithm concerned is liable to iterate indefinitely, or alternatively, it can produce a result beyond acceptable error bounds. The main algorithm presented is immune from the first problem, since it is not iterative; also, it generates a result for which there is a bound on numerical error proportional to the complexity of the problem.[1]

The algorithm presented in this chapter is a modified version of the Bentley-Ottmann sweep line algorithm [BO79]. In its original form, the Bentley-Ottmann algorithm assumes exact arithmetic. The variant form requires certain modifications, and certain additional actions have to be specified for anomalous situations not possible for the exact form of the algorithm. At any stage of the algorithm, computations are based on the structure as it currently exists, rather than on the original structure; hence, any change to the structure, such as splitting an edge at an intersection point, influences the downstream processing. The one respect in which the algorithm is not fully robust is that in certain circumstances the result may not be geometrically valid. This is because numerical inaccuracies in the computations may cause the resulting structure to breach the constraints required for geometrical validity.

Section 7.1 of this chapter gives details of code used for determining vertex-edge shadowing and edge-edge intersection, and also for representing polygonal sets. Section 7.2 demonstrates that the anomalous cell formation problems discussed hypothetically in section 6.3.1 are in fact a reality for the intersection code. The sections that follow explain

[1]The numerical accuracy of this algorithm is discussed in section 8.4, p.153.



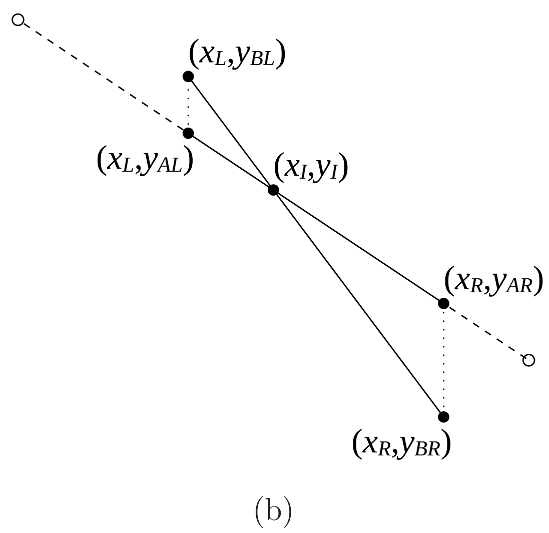


Figure 7.1: *Example showing the two stages to computing an intersection point: (a) given x_P (from point (x_P, y_P)), determine y_S such that point (x_P, y_S) lies on the segment connecting (x_L, y_L) to (x_R, y_R); and (b) determine the intersection point, (x_I, y_I), between two segments with identical range in x, connecting (x_L, y_{AL}) to (x_R, y_{AR}) and (x_L, y_{BL}) to (x_R, y_{BR}).*

the algorithm in stages: first, section 7.3 describes the Bentley-Ottmann algorithm in its original form, under the assumption that computations are exact; then section 7.4 describes a hypothetical, variant form of the Bentley-Ottmann algorithm, which resembles the approximate algorithm to be presented, except that it is still assumed (for the sake of argument) that computations are exact; the handling of degenerate cases is considered in detail in this section; section 7.5 then explains the anomalous situations that can arise for the algorithm just described if it is implemented in floating-point arithmetic; this leads on to section 7.6, which explains the action the approximate algorithm takes to handle anomalous data. Section 7.7 discusses the results of an experimental implementation of the algorithm as applied to the boundary self-intersection problem—the crucial first stage of the simplification process. Finally, section 7.8 briefly discusses other algorithms that were considered for the simplification process, and why they are not robust.

7.1 Essential code details

7.1.1 Determining intersection points

For the 2D simplification process, as with the 2D basic Boolean algorithm, it is necessary to determine whether two edges intersect; when they do, the point of intersection must also

```
//================================================================
double yAtX(const Point2& l, const Point2& r, double xP)
//================================================================
{
// For a given segment l->r ( l.x() < r.x() ) and x value
// (xP) in range ( l.x() <= xP <= r.x() ), compute the value
// yS such that the point (xP,yS) lies on the segment
  double dxL = xP - l.x();  // >= 0
  double dxR = xP - r.x();  // <= 0
  bool dxLLess = dxL < -dxR;
  double lambda = ( dxLLess ? dxL : dxR ) / ( r.x() - l.x() );
  double yS = ( dxLLess ? l.y() : r.y() )
              + lambda * ( r.y() - l.y() );
  return yS;
}
```

Table 7.1: *Code for determining y-value in segment.*

be determined. This section gives details of C++ code for determining the intersection point. The code is suitable for both the simplification process and the basic Boolean algorithm, since the essential difference between the algorithms is not how the point is calculated, but rather the conditions that determine when the calculation needs to be performed. For the simplification algorithm, the calculations described here are carried out only when preliminary calculations deem there to be no degeneracy in the data; for the basic Boolean algorithm, the symbolic perturbation rules that apply make it unnecessary to test for degeneracy.

There are two stages to calculating the intersection point between two segments. The first stage is to determine the y-value on a non-vertical segment for a specific x-value; the second is to determine the intersection point itself, based on data obtained from two instances of the first calculation.

For the first stage we are given two end-points of a segment, (x_L, y_L) and (x_R, y_R), where $x_L < x_R$, and a value x_P (taken from point (x_P, y_P)) such that $x_L \leq x_P \leq x_R$, as shown in figure 7.1 (a). The value y_S is computed such that the point (x_P, y_S) lies on the segment. The C++ code to compute y_S is essentially as shown in table 7.1.[2] Note that y_P is computed in one of two ways, depending on whether x_P is determined to lie closer to x_L

[2]In the code tests that I describe later it was necessary to change the code as listed in tables 7.1 and 7.2 so that the compiled program would execute all floating point arithmetic operations in accordance with the IEEE 754 standard. The differences are explained in section 7.7, p.131.

```
//================================================================
void interpolate(double xL, double yAL, double yBL,
                 double xR, double yAR, double yBR,
                 Point2& intPt)
//================================================================
{
// Given two segments (xL,yAL)->(xR,yAR) and (xL,yBL)->(xR,yBR)
// where yAL <= yBL and yBR <= yAR, with one of the inequalities
// being strict, determine the point of intersection, intPt
  double dyL = yBL - yAL;
  double dyR = yBR - yAR;
  bool dyLLess = dyL < -dyR;
  double dx  = xR  - xL;
  double dyA = yAR - yAL;
  double dyB = yBR - yBL;
  bool dyALess = fabs(dyA) < fabs(dyB);
  double lambda = ( dyLLess ? dyL : dyR ) / ( dyL - dyR );
  double xI = ( dyLLess ? xL : xR ) + lambda * dx;
  double yI = ( dyLLess ? ( dyALess ? yAL : yBL )
                        : ( dyALess ? yAR : yBR ) )
              + lambda * ( dyALess ? dyA : dyB );
  intPt.setXY(xI, yI);
}
```

Table 7.2: *Code sample for determining intersection point between segments.*

than x_R; this ensures that y_S is computed correctly when $x_P = x_L$ or x_S.

The code for the second stage, in which the intersection point is determined, is essentially as shown in table 7.2. This uses two instances of data created in the first stage—an approach that provably avoids division by zero (see explanation relating to equation (4.4), p.67). Thus, the algorithm starts with two x-values, x_L and x_R with $x_L \leq x_R$, which are the limits of the range in x for which the two original segments overlap; if one of the segments is vertical, then $x_L = x_R$. y_{AL} and y_{BL} represent the y-values of the two segments A and B at $x = x_L$, and likewise y_{AR} and y_{BR} at $x = x_R$, as shown in figure 7.1 (b). At least one of the points (x_L, y_{AL}) and (x_L, y_{BL}) is an original segment end-point, and the other may be an interpolated point determined by the first calculation; likewise for points (x_R, y_{AR}) and (x_R, y_{BR}).

The code can evaluate the intersection point in one of four ways, depending on (1) whether

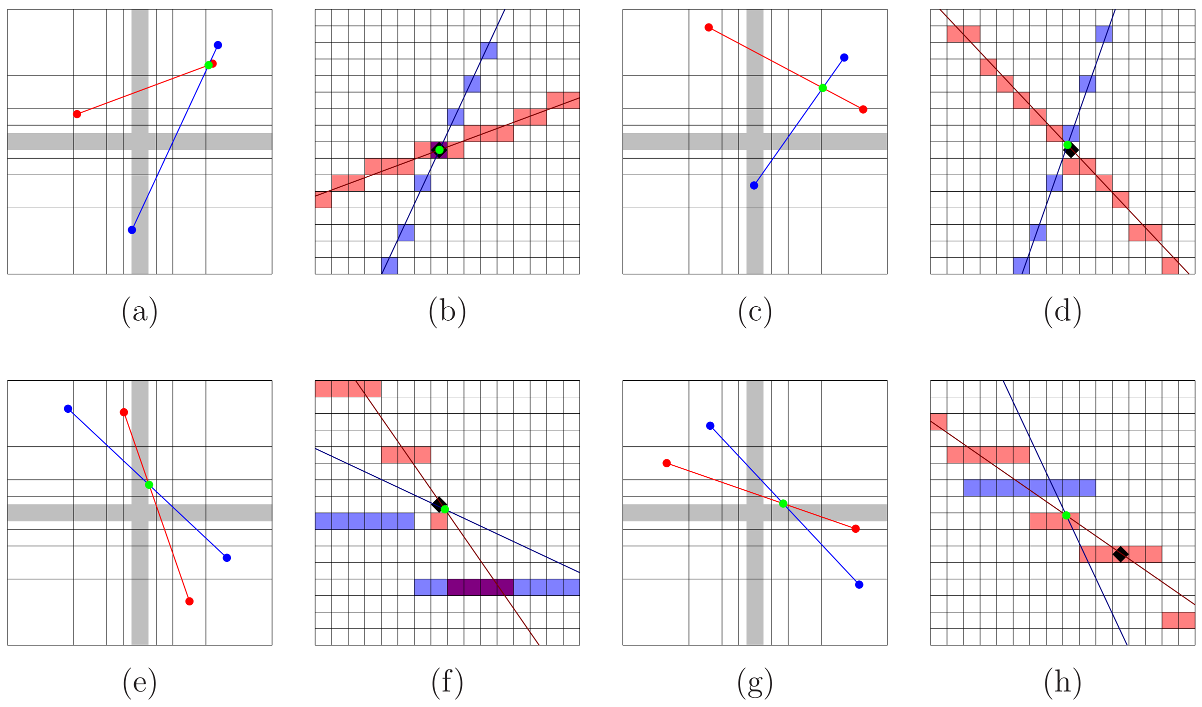


Figure 7.2: *Four example cases showing the result of the intersection calculation. The first of each pair of diagrams shows the two segments in red and blue and the intersection point in green, displayed within the square $[-1, 1] \times [-1, 1]$; the horizontal and vertical lines mark a change in the spacing between representable points. The subsequent diagram of each pair shows the grid of representable points around the intersection point. The points deemed to be on either segment are shaded respectively red and blue, or magenta if on both; the computed intersection point is marked with a black diamond. The calculations determining which points lie on a segment are affected by the precision of intermediate values, hence the jumps in y value for both segments seen in (f) and (h). The exact locations of the two segments and of the true intersection point are superimposed. The example cases shown in (e) and (g) are a transpose of each other, with x- and y-coordinate values switched; however, the computed intersection points shown in (f) and (h) are not a transpose of each other.*

the intersection point is determined to be closer to the left than the right ($y_{BL} - y_{AL} < y_{AR} - y_{BR}$), and (2) whether the absolute slope for segment A is less than that for B ($|y_{AR} - y_{AL}| < |y_{BR} - y_{BL}|$); this strategy ensures respectively (1) the correct evaluation of x_I when $y_{BL} = y_{AL}$ or $y_{AR} = y_{BR}$, and (2) the correct evaluation of y_I when there is a horizontal segment, with either $y_{AR} = y_{AL}$ or $y_{BR} = y_{BL}$.

Figure 7.2 shows four cases of segment intersection calculations. For each pair of diagrams, the first diagram shows the particular case as displayed in the square $[-1, 1] \times [-1, 1]$; the second diagram in each case shows a close-up view at the point of intersection. Note that the computed point of intersection is not necessarily a point that is deemed to lie on both edges, or even just one; indeed, there may not necessarily exist a point deemed to lie on both edges.

7.1.2 Representing polygonal sets and edge collections

I devised and implemented the C++ class `PolySet2` for managing shape representations, including geometrically invalid ones. It is also suitable for handling more general edge collections—a polygonal set representation is simply a collection of edges that form the (supposed) boundary. The class was used in all versions of the simplification algorithm investigated.

The interface to the class allows an edge collection to be constructed by adding edges successively in any order. It also provides the means to read each edge of a constant (i.e. completed) edge collection in sorted order.

The policy for simplification, as stated in section 6.5, p.102, is to merge edges with an identical start-point/end-point pair and combine the associated multiplicity values; if the multiplicity values cumulate to zero, the edge is regarded as cancelled. The `PolySet2` class does exactly this. Adding an edge through the interface leads to a search for a pre-existing edge within the collection with an exactly identical start-point/end-point pair, which if found is merged with the new edge.

The `PolySet2` class makes use of a segment class, `Segment2`, to store the two points of an edge in lexicographic order. We refer to these as the *leftmost point* and *rightmost point*. Note that for a vertical segment they will be the lowermost and uppermost points respectively. Edges are represented as a collection of segments, each one mapped to the integer value indicating the multiplicity of the edge. Edges that go from rightmost to leftmost point are represented by a negative multiplicity value. The C++ Standard Library template class `std::map` is used to store the collection. The `map` template requires the key class, `Segment2`, to have an order of precedence, as defined by the comparator operator '<', by which the collection is ordered. Lexicographical ordering is used, in accordance with the leftmost point in the first instance, and then the rightmost point. The iterator for the class visits the collection edges in this order, making it particularly suited for sweep-line algorithms going from left to right. A partial listing of the code for the polygonal set representation class is given in table 7.3, showing in particular the mechanism for adding an edge to the collection.

This method of representing a polygonal shape or edge collection does not explicitly store vertices or the linkage of edges through a shared vertex. This information is not needed by the algorithm. The representation of a particular point as the leftmost or rightmost point of one or more edges is taken to indicate the existence of a vertex at that point, with the edges assumed to be linked to each other at the vertex.

```
#include <map>
#include "Point2.h"
#include "Segment2.h"
class PolySet2
{
public:
  ...
  int addEdge(const Point2& ptS, const Point2& ptE, int mult=1)
  // add edge from point ptS to point ptE with multiplicity mult
  // argument mult is optional - default value = 1
  // returns resulting multiplicity, taking into account
  //   pre-existing edge for this segment
  {
    int result = 0;
    if ( ptS < ptE ) // comparator for Point2 is lexicographic
    { result = addOrderedEdge(Segment2(ptS, ptE), mult); }
    else if ( ptS != ptE ) // note: null edge not processed
    { result = -addOrderedEdge(Segment2(ptE, ptS), -mult); }
    return result;
  }

private:
  int addOrderedEdge(const Segment2& seg, int mult)
  {
    int result = _edges[seg] += mult;
          // seeks existing edge for segment, or creates new one
    if ( result == 0 )
    { _edges.erase(seg); } // not retained for zero multiplicity
    return result;
  }

  std::map<Segment2,int> _edges;
};
```

Table 7.3: *Code (in part) for the polygonal set representation (or edge collection) class* `PolySet2`*.*

7.2 Example cases of inconsistent data

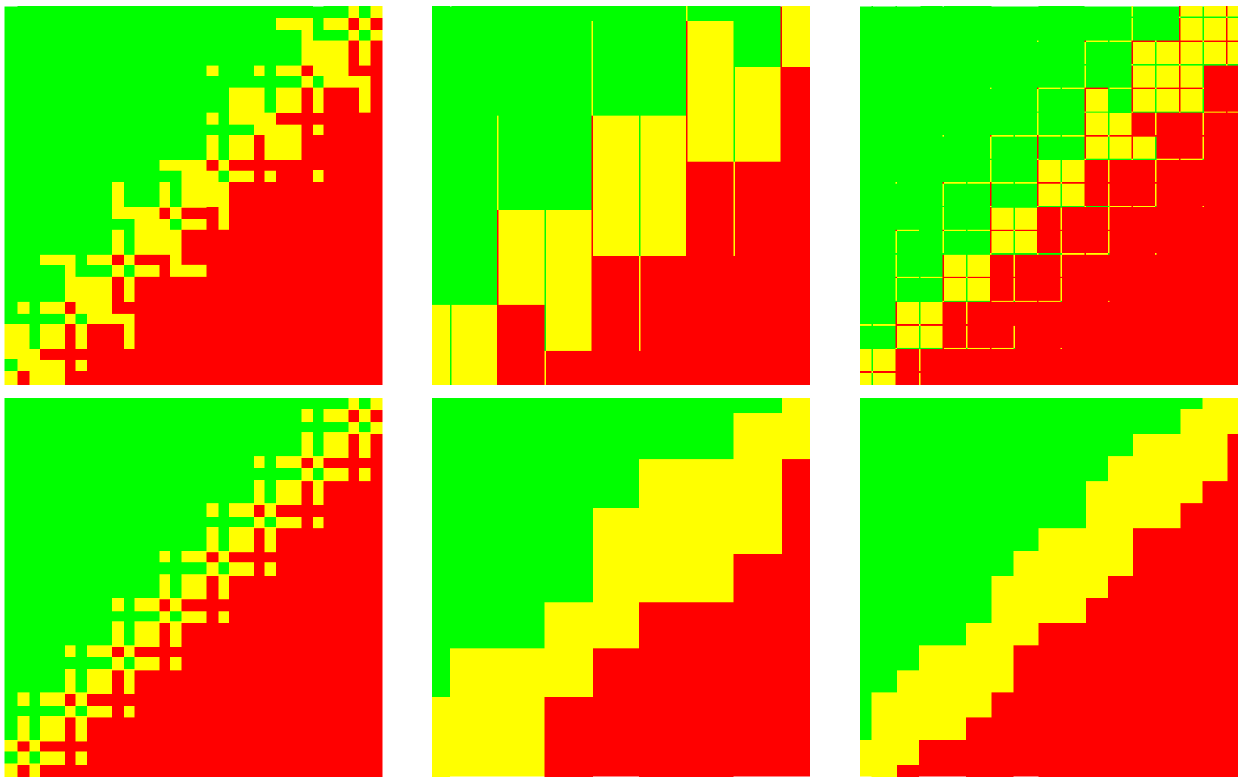
I pointed out in section 6.3.1 that when using approximate arithmetic there is nothing to prevent inconsistent combinations of predicate relationships that would not be possible in the exact domain. Furthermore, when such situations arise, the planar structure of the subdivision can be lost, which prevents progress with simplification. In this section I demonstrate that examples of such cases can occur when using the code given in section 7.1.1 to determine shadow relationships and intersections.

We saw in section 2.3.2, p.29, following Kettner et al. [KMP+08], that computing the orientation predicate based on the cross product formula can lead to situations that are impossible under Euclidean geometry, which in turn leads to robustness problems for the incremental convex hull algorithm. The method used for determining vertex-edge shadowing is in effect another way of computing the orientation predicate. All three cases that were demonstrated in figure 2.3, p.30, have $x_p < x_q < x_r$, so the calculation of the orientation of **p**, **q**, **r** is equivalent to determining whether point **q** lies above, on or below the segment connecting **p** and **r**. Figure 7.3 repeats the plots of the orientation of the three points computed using the cross product formula, as shown originally in figure 2.3, together with the orientation computed using the shadow function `yAtX()` as listed in table 7.2 for the same three cases. We see that cases (b) and (c) are somewhat better behaved using the shadow function, with the region of points deemed collinear with **q** and **r** forming a contiguous band that cleanly separates the other two regions. The difference in nature is due to x_q being closer to x_r than to x_p, causing `yAtX()` to select **r** as the reference point in place of **p**. **p** remains the reference point for case (a), and as we see, the pattern of regions is similar (but different in detail) for the two methods of computing the orientation. For both methods, all three regions have peninsulas and islands. Furthermore, for both methods there are points that form a strict anti-clockwise order with **q** and **r** but are *computed* to have a strict *clockwise* order, and likewise the other way round.

By examining such diagrams for these and other data it is possible to determine example cases that lead to inconsistencies of the type discussed in section 6.3.1. Listed below is a set of points from which a number of inconsistent relationships can be derived. These make use of the value $u = 2^{-53}$ (1.11×10^{-16}), which is the spacing between representable double-precision numbers in the range $[\frac{1}{2}, 1]$:[3]

- $A = (0.5 + 9u, 0.5 + 23u) = (0.500000000000000999, 0.500000000000002554)$

[3]See section 8.1 on floating-point arithmetic, which gives the definition of u, known as the unit roundoff value. The separation distance between adjacent representable values in the ranges $[-L, -\frac{1}{2}L]$ and $[\frac{1}{2}L, L]$ is uL, where L is an integer power of 2.



(a)

$$p: \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \quad q: \begin{pmatrix} 12 \\ 12 \end{pmatrix} \quad r: \begin{pmatrix} 24 \\ 24 \end{pmatrix}$$

(b)

$$\begin{pmatrix} 0.50000000000002531 \\ 0.5000000000000171 \end{pmatrix} \quad \begin{pmatrix} 17.300000000000001 \\ 17.300000000000001 \end{pmatrix} \quad \begin{pmatrix} 24.00000000000005 \\ 24.0000000000000517765 \end{pmatrix}$$

(c)

$$\begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \quad \begin{pmatrix} 8.000000000000007 \\ 8.000000000000007 \end{pmatrix} \quad \begin{pmatrix} 12.1 \\ 12.1 \end{pmatrix}$$

Figure 7.3: *This figure displays the orientation predicate values* $ori\left((x_p + iu, y_p + ju), \mathbf{q}, \mathbf{r}\right)$ *computed in two ways. The upper displays are a repeat of those in figure 2.3, p.30, computed using the cross product formula (see caption for details); for the lower displays, the predicate values are determined by the code for calculating the vertex-edge shadow relationship between point* $\mathbf{q}$ *and the segment connecting* $\mathbf{p}$ *and* $\mathbf{r}$*.*

- $B = (0.5 + 23u, 0.5 + 25u) = (0.500000000000002554, 0.500000000000002776)$
- $C = (9, 0.5)$
- $D = (9, 9)$
- $E = (9, 32)$
- $F = (32, 32 - 32u) = (32, 31.9999999999999964)$
- $G = (32, 32)$

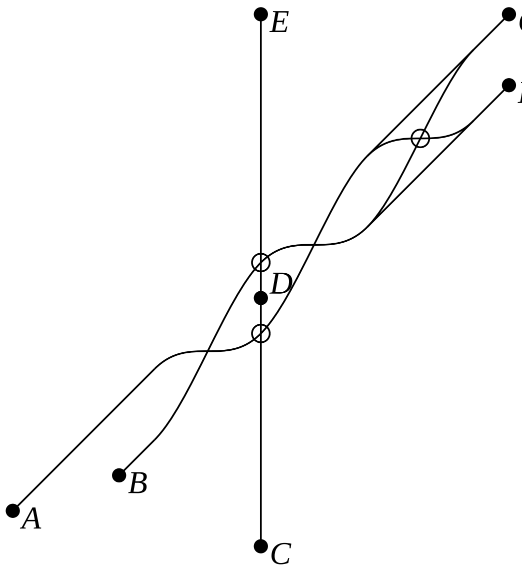


Figure 7.4: *The invalid topology (as computed) relating to straight-line segments connecting the points* $A = (0.5 + 9u, 0.5 + 23u)$, $B = (0.5 + 23u, 0.5 + 25u)$, $C = (9, 0.5)$, $D = (9, 9)$, $E = (9, 32)$, $F = (32, 32 - 32u)$ *and* $G = (32, 32)$, *where* $u = 2^{-53}$. *Not drawn to scale. Hollow circles depict computed intersection points between segments.*

From these points the following vertex-edge shadow relationships are computed:

- B lies below both AF and AG
- F lies below both AG and BG
- D lies above both AF and AG
- D lies below both BF and BG

We see here examples of impossible relationships as shown in figure 6.4. D lies above AG and below BF, even though BF is below AG. Also, AG lies below DG, which lies below BG, which lies below AG, thus giving an inconsistent cyclical ordering of edges about vertex G.

The following edge-edge intersection relationships are computed:

- both $\overrightarrow{AF}$ and $\overrightarrow{AG}$ cross $\overrightarrow{CE}$ left-to-right at $(9, 9 - 16u) = (9, 8.99999999999999822)$
- both $\overrightarrow{BF}$ and $\overrightarrow{BG}$ cross $\overrightarrow{CE}$ left-to-right at $(9, 9 + 16u) = (9, 9.00000000000000178)$
- $\overrightarrow{AF}$ crosses $\overrightarrow{BG}$ left-to-right at $(9.09090909090909172, 9.09090909090909172)$

So the three edges AF, BG and CE are computed to intersect each other in an incompatible way, akin to what is shown in figure 6.5(d), and as such they fail to form a valid cell structure. The relationships between the three edges AG, BF and CE are also such that they fail to form a valid cell structure.

The computed relationships described are shown in figure 7.4.

7.3 The Bentley-Ottmann sweep-line algorithm

To help explain the sweep-line algorithm based on the use of approximate arithmetic, I first describe the Bentley-Ottmann sweep-line algorithm in its original form, in which all arithmetic calculations are assumed to be exact. First published in [BO79], the Bentley-Ottmann algorithm is also described in [dBvKOS97, pp.20-29] and [Sun06]. The algorithm was initially devised to report the points of intersection between segments in the plane [BO79], but it extends naturally to apply to the problem of determining the cell structure of the arrangement [dBvKOS97, pp.29-33], and hence to related operations such as the Boolean operation and the simplification process. It has a time complexity of $O(n + I)\log n$ where n is the number of segments in the input data and I is the total number of intersecting pairs of segments [dBvKOS97, p.28]. The Bentley-Ottmann algorithm is fully robust if implemented using exact arithmetic or according to the exact geometric computing paradigm (see section 2.4.2). An implementation based on exact computation is available in the CGAL library, for the broader problem of determining the arrangement of curves [WFZ07].

For the Bentley-Ottmann algorithm, an infinite vertical line, known as the *sweep line*, is considered to sweep across the plane from left to right. Two structures are maintained as the sweep line crosses the input data; these are known as the *sweep-line structure* and the *event queue*.[4]

The *sweep-line structure* is a list of *sweep-line edges*, consisting of those edges in the original structure that traverse the notional line. As such, it can be considered as a record of the arrangement in the locality of that line as currently positioned. The algorithm operates by maintaining an ordered list of the edges within the sweep-line structure, their order in the list being in accordance with their physical ordering, i.e. in terms of ascending y. The structure can be regarded as a 1D cellular structure, representing a cross section of the arrangement.

The *event queue* is a set of *event points* that affect the sweep-line structure each time the sweep line notionally crosses one such point. The points concerned are the edge limit-

[4]In [dBvKOS97], the sweep line is a horizontal line progressing downwards, and the sweep-line structure is known as the status structure.

points (both leftmost and rightmost) of the original edges and the intersection points between two or more edges that are yet to be processed. Each event point is considered in turn in lexicographic order (see section 6.5, p.100), and the event queue is updated accordingly. The event points are held in lexicographic order within the structure representing the event queue, and are removed one by one as they are processed; thus, when the processing of an event point is complete, the next to be selected is always the first one listed within the structure. It is not essential for the event queue structure to hold *all* event points that have yet to be processed at any given time; it is only essential for an event point to be present in the queue when the time has come for it to be selected.[5]

In the original form of the algorithm, as described in [dBvKOS97], sweep-line edges are instances of complete edges from the input structure. All leftmost and rightmost points of the input edges are considered to be in the event queue at the start, but no intersection points are initially included; these are added to the event queue as they are discovered. When an event point is processed there are three possible types of action:

- Any edge in the sweep-line structure that ends at the event point is removed from the sweep-line structure.[6] If the algorithm is used to generate a structure, such as a cell formation, a copy of the edge is sent to the output.
- If two or more edges in the sweep-line structure pass through the event point, the event point is considered to be an intersection point, and the edges are swapped within the sweep-line structure. If the algorithm is used to report intersections, the point is reported as such at this stage.
- Any edges in the input structure starting at the event point are copied into the sweep-line structure. The point of insertion within the list of sweep-line edges follows those edges that pass below the event point and precedes those that pass above, and the inserted edges are ordered according to gradient.

Once the sweep-line structure has been modified in accordance with the event point in question, neighbouring pairs of edges are tested to see if they intersect; only edges that were not neighbouring previously need to be tested. If two neighbouring edges do intersect, the intersection point is inserted into the event queue.

[5]Details of how to represent the two structures are not included in this account, since they are not of direct concern for the robustness issue. The description of the algorithm in [dBvKOS97, p.25] advocates the use of a balanced binary search tree for the sweep line in order to optimise position searching.

[6]In the context of a sweep line algorithm, saying an edge 'starts at / passes through / ends at' point P means that P is the leftmost point / an interior point / the rightmost point of the edge. This is unrelated to the edge direction defined in section 4.2 indicating which side of the edge marks the shape interior, or cell with the higher winding number. Hence, an edge may well 'start at' its end-vertex and 'end at' its start-vertex.

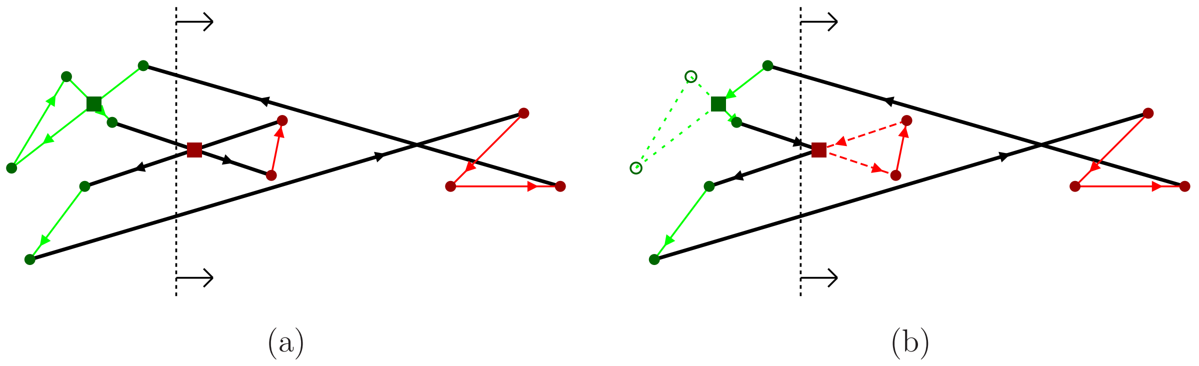


Figure 7.5: *Example case showing how the Bentley-Ottmann sweep-line algorithm operates. (a) illustrates the arrangement problem in the pure form, showing the situation after the first seven event points (including one intersection point) have been processed. Original end-points are marked circular; the intersection points so far detected are marked square. Edges and points already processed are marked green; those yet to be processed are marked red; the sweep-line edges are marked black. The sweep line, shown dotted in black, lies notionally between the last event point to be processed and the next. (b) shows the situation for the modified form of the algorithm, as applied to the simplification process. Sweep-line edges known to intersect are in effect shortened to end at the point of intersection; the remaining edge parts, drawn dashed in red, are placed in a buffer and made available for processing in the same manner as for input edges. The hollow circles and dotted lines in green are the points and edges discarded by simplification.*

Note that if the processing of an event point introduces a new sweep-line edge that intersects an existing sweep-line edge that is *not* an immediate neighbour, the intersection point concerned will not be detected, and so will not be added to the event list at this stage. This does not matter, however, because they eventually will become neighbouring edges, once all the edges lying in between have either terminated or else crossed one of the two edges concerned. Figure 7.5(a) shows an example case of the arrangement problem, with certain of the intersection points not yet detected (and therefore not marked).

This is an incomplete description of the Bentley-Ottmann algorithm, which has avoided discussion as to how to handle degenerate data. Potentially, a point may be a multiple event point, acting as leftmost, rightmost and/or intersection point to arbitrarily many edges.[7] Degenerate cases are considered in detail in the description of the variant form of the algorithm that follows.

[7]De Berg et al. [dBvKOS97, p.26] describe how the algorithm handles a multiple event point, but do not consider overlapping segments.

7.4 A modification to the Bentley-Ottmann algorithm

It is now helpful to consider a variant of the Bentley-Ottmann algorithm. This variant still assumes exact arithmetic with no errors. I present this variant of the algorithm not as a method to be implemented, but rather as a prototype for the implemented, approximate algorithm I describe shortly.

7.4.1 The sweep-line process

The variant form of the algorithm essentially resembles the original just described, except that whenever two adjacent sweep-line edges are detected to intersect, the edges are split at the point of intersection. This is carried out for each of the two edges by re-setting the rightmost point to be the point of intersection, which in effect shortens the sweep-line edge, and by introducing a new edge that connects the point of intersection to the original rightmost point. A buffer edge collection is used to hold the new edges created this way. The edges within the buffer edge collection are considered equivalent to input edges that are yet to be incorporated into the sweep-line structure; buffer edges and remaining input edges are selected in lexicographical order as though originating from the same collection.

The consequence of splitting edges at intersection points is that it ceases to be necessary to handle an intersection point as an event point. An edge intersected at an interior point is split at that point, usually in advance of it being processed as an event point (apart from one exceptional situation discussed shortly). Figure 7.5(b) shows an example of how this variant of the sweep-line algorithm works for simplification.

7.4.2 Handling degenerate data

Section 6.5 lists the essential requirements for handling coincident data.

The requirement for edge splitting is as follows: an edge is to be split at any of its internal points that is also an event point, an event point being either a limit-point of another edge or else as the *sole* point of intersection between this edge and another edge. The consequence of this rule is that when an edge has a region of overlap with another edge, that overlap will cause the edge to be split at at most two internal points (notwithstanding the effect of other edges).

There are many possible ways for degeneracy to arise. An event point may act as leftmost or rightmost point to an arbitrary number of edges, and likewise as an interior point to an arbitrary number of edges. This section explains the mechanism by which edges are split at the required points, in particular for degenerate cases.

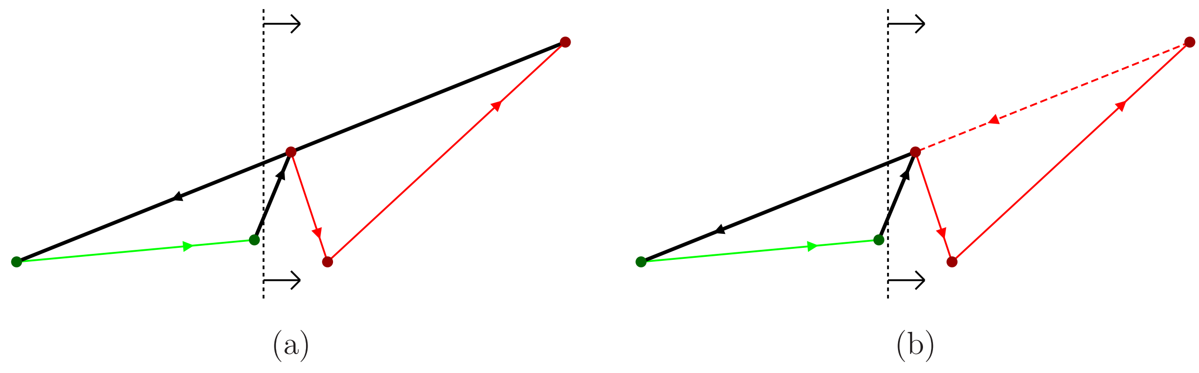


Figure 7.6: *In (a), out of the two sweep-line edges, drawn in black, the longer one passes through the rightmost point of the shorter one. It is shortened to end at this point, with the remainder stored as a buffer collection edge, drawn dotted in red in (b).*

Let us first consider the situation in which one or more edges end at a particular point, and exactly one other edge passes through the point. (Other edges may start at this point, but have no effect on the process if present.) At some stage of the algorithm, in advance of the point being processed as an event point, the edge that passes through the point will be included in the sweep-line structure, and within that structure it will neighbour one of the edges ending at the point. The test used to determine whether two neighbouring sweep-line edges intersect includes a preliminary test to determine if the rightmost of one edge lies in the interior of the other. The relationship between these two edges will therefore be detected; consequently, the edge passing through the point is shortened to end at the point, in the usual manner for intersected edges; the edge ending at the point is not affected. An example case is shown in figure 7.6.

Essentially the same mechanism is used for other examples of intersection points, *provided* there are at least two non-overlapping edges that either pass through or end at the point. Since the edges coalesce towards the point from the left, eventually two will neighbour each other on the sweep line, and the intersection test mechanism will split either or both these edges, if they pass through the point. There may be further intersection tests applied as other pairs of edges are tested, and eventually all edges passing through the point will be tested for intersection and be split at that point. This all occurs before the point of intersection is processed as an event point. Figure 7.7 shows an example of a multiple event point, in which the edges passing through the point are all split.

This leaves one situation in which an intersected edge is not split in consequence of the intersection test. This arises when there are a number of edges starting from a particular point, but with no edges ending at it, and with exactly one passing through it; the situation can also arise with several edges passing through the point, provided they are all collinear and overlap over a segment that includes the point. In this situation, the algorithm will first encounter the point concerned as an event point to be processed. It

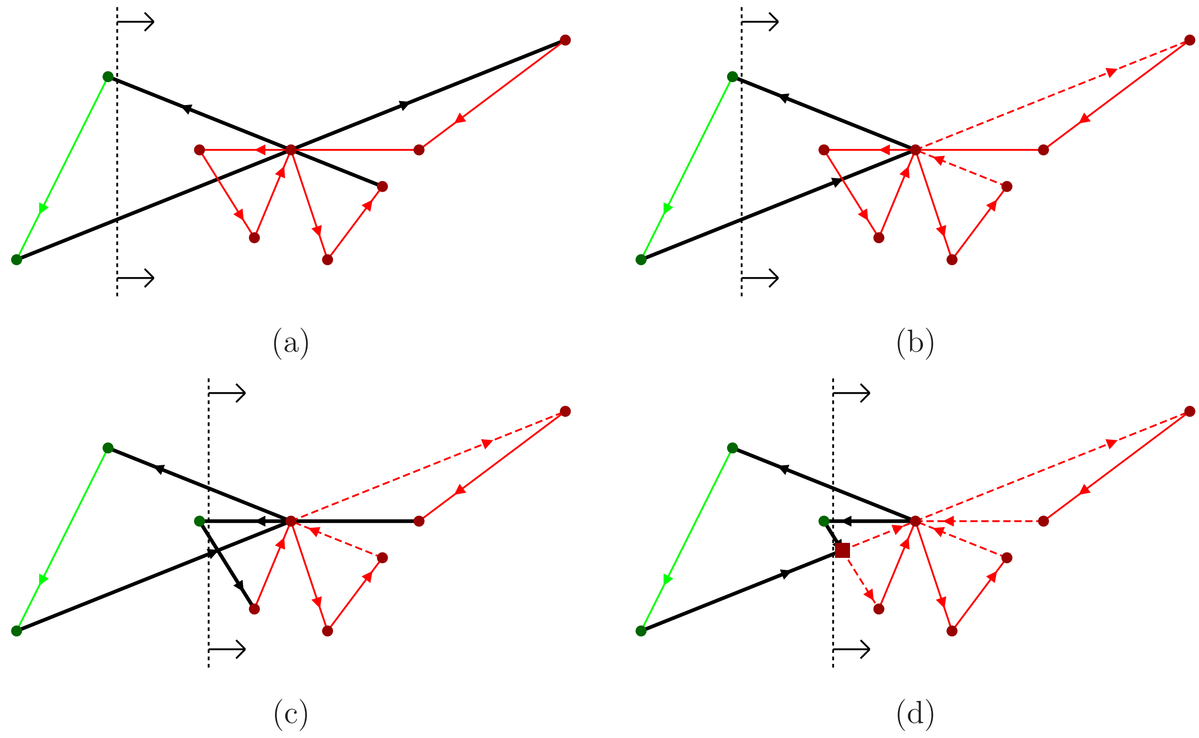


Figure 7.7: *An example of a more complex event point. A non-degenerate intersection between the two sweep-line edges, as shown in (a), leads to both edges being shortened, as in (b). When the third edge to pass through the point is placed on the sweep line, as in (c), a degenerate intersection is detected, and this edge too is shortened, as in (d).*

will recognise that there are edges starting at the point, but will not know initially about the edge passing through. As is standard, the algorithm will determine whether successive sweep-line edges pass over, under or through the event point, principally for the purpose of determining where within the sweep-line structure the new edges should be inserted. When the edge is detected to pass through the point, special action is taken in effect to split this edge. For this, the edge itself is removed from the sweep-line structure, while the part-edge to the left is sent to the output. The part to the right is re-inserted into the sweep-line structure, together with the original edges that start at the event point; as is standard, these are all sorted within the sweep-line structure according to gradient. An example case is shown in figure 7.8.

The mechanisms just described are also appropriate for handling overlapping edges, as discussed in section 6.5, p.102 and shown in figure 6.7. Edges occupying exactly the same segment may co-exist within the sweep-line structure during the process. The only extra action required is that exactly coincident edges should be merged on output.[8]

[8]The class `PolySet2` described in section 7.1.2 merges exactly coincident edges, as required.

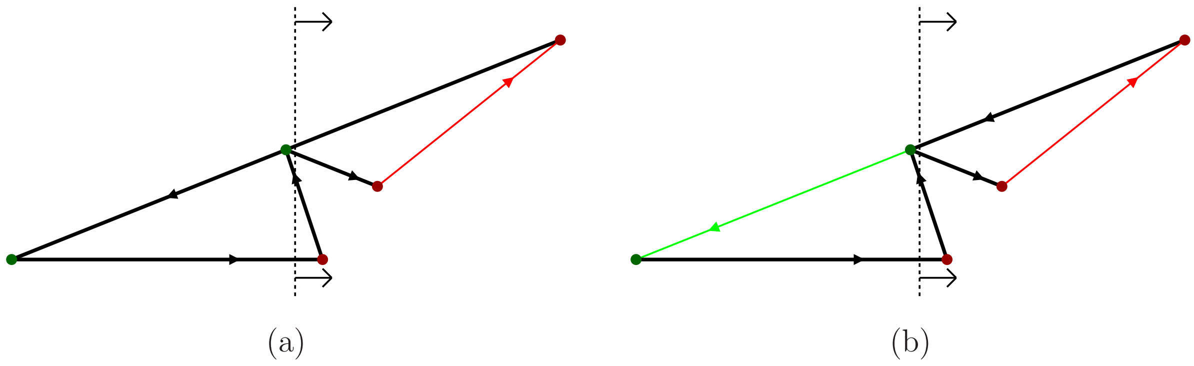


Figure 7.8: *Two edges start at a point that is interior to another. The point is processed as an event point, requiring that the two edges that start there be added as sweep-line edges, as shown in (a). The upper edge is detected to pass through the point, it is in effect split at that point, with the part to the left sent to output, and the part to the right re-inserted as a new sweep-line edge, as in (b).*

7.4.3 Output of edges

The remaining aspect of the algorithm requiring explanation is how an edge is processed for output, once the processing of its rightmost point terminates its role as a sweep-line edge.

We can consider the process described so far simply as a means of determining the cellular structure formed by the initial boundary structure. If every sweep-line edge is sent to the output structure with the multiplicity value assigned to it, then the output simply represents this cellular structure (though the information concerning the faces is lost).

The simplification process is an extension of the process of determining the boundary structure. It differs only at the stage when a sweep-line edge is sent to the output structure. As was stated in section 6.1, p.89, and shown in figure 6.2, p.91, the process of simplification is in effect the selection of regions in the original structure with an appropriate winding number. For each edge within the structure generated to represent the boundary self-intersection, it is possible to derive the winding number of each cell that lies on either side. From this it can be determined whether the cell on either side forms part of the result interior, and hence, whether the edge is part of the result boundary, and in which direction it goes.

The sweep-line structure represents a 1D cross-section of the cell formation at the sweep line for a specific x-value. The edges of the sweep-line structure are ordered according to their respective y-values for the x-value concerned, so an interval between two edges adjacent on the sweep-line structure represents the intersection between the sweep line and a particular face. The winding number of that face equals the sum of the signed multiplicity values of all the sweep-line edges that precede the cell.

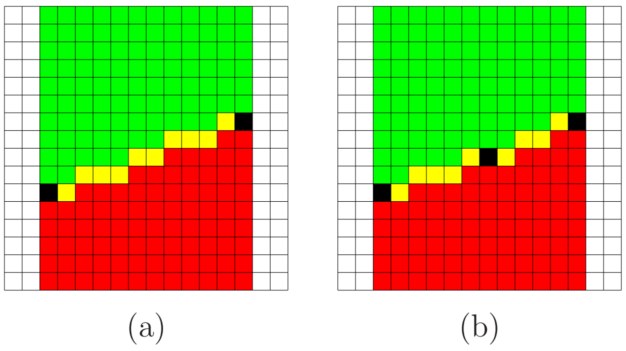


Figure 7.9: *(a) shows the set of points deemed to lie below/on/above an edge, coloured red/yellow/green respectively. (b) shows the same for two edges, generated by splitting the original edge at a particular point. The point sets are not identical for the two cases, even though the split point is deemed to lie on the original edge.*

This completes the description of the modified form of the Bentley-Ottmann algorithm in the context of exact arithmetic. We are now in a position to consider how the algorithm can be implemented using approximate arithmetic.

7.5 The effect of rounded arithmetic

The above description of the variant form of the Bentley-Ottmann algorithm is based on an implicit assumption that all arithmetic computations are exact. It makes certain assumptions about the behaviour of the algorithm which, while always true under exact arithmetic, does not necessarily apply under rounded arithmetic. This section describes what can happen, and how that hinders the implementation of the algorithm. Section 7.6 then describes how the approximate algorithm handles such situations.

7.5.1 The effect of edge-splitting on the data and the algorithm

The action of splitting an edge at an intersection point can affect the relationship (as computed) between the edge and a specific point—that is to say, whether that point is deemed to lie above, on or below the edge concerned. This is clearly the case if the computed intersection point is not deemed to lie on the original edge, but it can be the case even when the point does lie on the edge—see figure 7.9. Such changes affect the topology of the arrangement that underlies the collection of edges, and hence the behaviour of the algorithm. Pairs of edges that were previously deemed separate may subsequently be considered to intersect; alternatively, they may continue not to intersect, but their physical ordering in terms of y-values within the range they overlap in x may switch, with what was considered the upper edge becoming the lower edge. The exact effect of the change in data upon the algorithm behaviour depends on the status of the edges concerned—that is to say, whether the edges affected are input edges yet to be processed or buffer edges, or sweep-line edges, or output edges. For the sake of keeping the explanation simple, in the rest of this section the term 'input edge' is used to mean either an unprocessed input edge or a buffer edge.

Consider the situation when splitting of a sweep-line edge affects the relationship between that edge (now two edges) and another edge. There are three scenarios to consider concerning a pair of edges *after* an edge split, in which one (at least) is a new edge formed by the split:

- One edge (at least) out of the two is an input edge.
- One edge (at least) out of the two is an output edge.
- Both original edges are sweep-line edges.

Example cases are shown in figure 7.10 in which two sweep-line edges are split at an imprecisely computed point of intersection. Let us consider each scenario.

When one edge out of a pair is an input edge, the effect of any change in relationship between these two edge is benign—this includes cases in which the edge to the right of a split point has a change in status from sweep-line edge to buffer edge. The change in relationship affects the final result, but there is no reason for the change to hinder the progress of the operation or to generate an invalid result. Information on the original relationship was either never computed by the algorithm, or else discarded following the change in status for an edge. Figure 7.10(a) and (b) shows a hypothetical example case, before and after the split, in which several edge pairings come under this category.

When one edge out of a pair is an output edge, the algorithm is able to progress without difficulty. Edges that have been sent to the output are 'forgotten' by the algorithm, and so can no longer affect it. However, this scenario is liable to lead to the generation a geometrically invalid result; figure 7.10(c) and (d) shows one such example case. This possibility is unfortunate, given that the intended purpose of the simplification process is to resolve a geometrically invalid shape representation.

A more serious situation arises when there is a change to the relationship (as computed) between a pair of sweep-line edges. An example case is shown in figure 7.10(e) and (f). The computed relationship between the two edges may well not agree with the ordering of the edges in the sweep-line structure; the algorithm must therefore be re-specified to accommodate the possibility that such inconsistencies may arise. It is inappropriate simply to switch the order of the edges within the sweep-line structure, or to ignore the change in the shadow relationship between a point and a sweep-line edge, as this is likely to lead to a topologically invalid result.

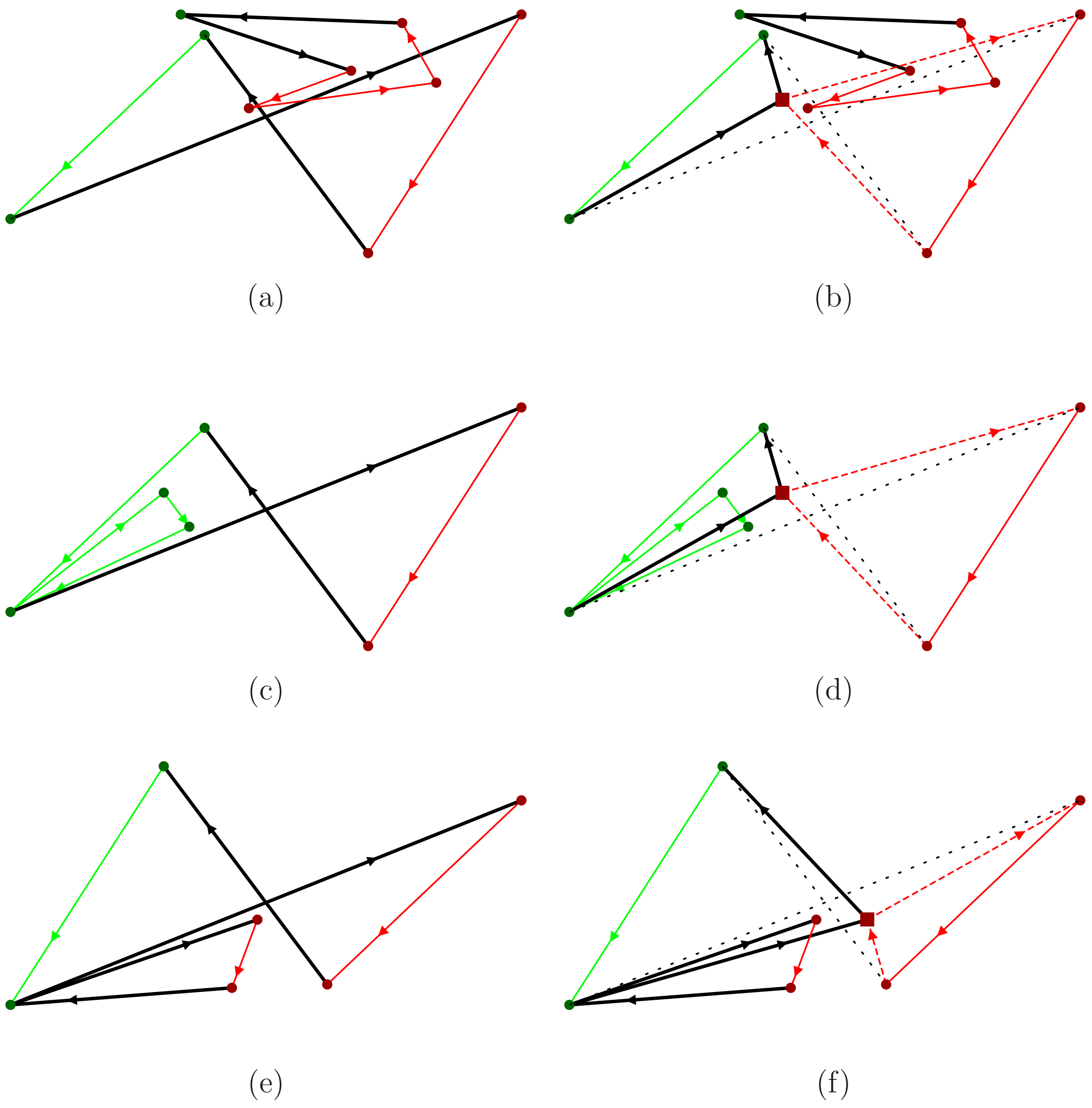


Figure 7.10: *Three example cases in which an inaccurately positioned intersection point leads to a change in the relation between certain pairs of edges. For each case, the situation is shown before and after the two intersecting edges are split. In (a) and (b), several pairs of edges are affected; each pair affected includes one edge (at least) that is an input edge, or else a buffer edge (after the split). The change in data affects the progress of the algorithm and produces a different result, but this influence is benign because the result is equally valid. In (c) and (d), a sweep-line edge moves in relation to edges that have already been output. The algorithm is able to continue without problem, but since it no longer holds output edge data, it takes no action concerning the intersection between the modified sweep-line edge and the output edge; consequently, the result generated will be geometrically invalid. In (e) and (f), the physical ordering between two sweep-line edges is reversed, making it inconsistent with the logical ordering of edges in the sweep-line structure.*

7.5.2 Grouping sweep-line edges with respect to the event point

A crucial issue affecting a rounded arithmetic implementation of the Bentley-Ottmann algorithm is the positioning of the event point with respect to each sweep-line edge. At the start of processing of an event point, $\mathbf{p}$, each sweep-line edge with leftmost and rightmost points $\mathbf{l}$ and $\mathbf{r}$ satisfies the constraint $\mathbf{l} \prec \mathbf{p} \preceq \mathbf{r}$; hence $x_l \leq x_p \leq x_r$. The sweep-line edges can therefore be categorised as those ending at $\mathbf{p}$, those passing through $\mathbf{p}$, those that pass under, and those that pass over $\mathbf{p}$. In the description of the algorithm there is an underlying assumption that the logical ordering of the edges in the sweep-line structure reflects the physical ordering of these edges in terms of ascending y-coordinate values on the sweep line itself, just prior to the sweep line reaching $\mathbf{p}$. Hence, any edges passing under $\mathbf{p}$ should all be listed first, followed by all edges (if there are any) that end at or pass through $\mathbf{p}$, and finally by all edges (again if present) that pass over. While this will always hold when the computations are exact, it is not the case when approximate arithmetic is used. Inconsistent ordering of the sweep-line edges may be brought about by edge splitting, in the manner just described, or the inconsistency may be inherent in the original data, as was discussed in sections 6.3.1, p.94, and 7.2, p.112.

In an exact implementation, any sweep-line edges that end at or pass through $\mathbf{p}$ are removed as a contiguous block within the list, to be replaced (possibly) by a new contiguous block of sweep-line edges that start at $\mathbf{p}$. This ensures that the logical ordering continues to reflect the physical ordering on the sweep line once the sweep line has passed $\mathbf{p}$. However, let us consider the consequences if the edges do not form the expected contiguous blocks prior to the processing of $\mathbf{p}$. An edge that does not pass through $\mathbf{p}$ will remain present on the sweep line after $\mathbf{p}$ has been processed as an event point. The cell that supposedly lies above this edge prior to processing $\mathbf{p}$ should therefore be the cell that lies above it after processing; the same applies to the cell lying below the edge. However, if the winding number of the cells both above and below this edge are computed in the manner described in section 7.4.3, p.121, the values obtained before and after $\mathbf{p}$ is processed will not necessarily be the same. Since the winding number values for these cells influence the decision as to whether the edge forms part of the boundary of the simplified shape, if these values change, there is no definitive answer as to whether the edge should be included.

7.5.3 Formation of downwards vertical edges

An edge in the input connecting vertices $\mathbf{l}$ and $\mathbf{r}$ ($\mathbf{l} \prec \mathbf{r}$) becomes a sweep-line edge once $\mathbf{l}$ is processed as an event point. In the original form of the algorithm, it remains a sweep-line edge up until $\mathbf{r}$ is processed as an event point. In the variant form of the algorithm, it may cease to act as a sweep-line edge beforehand, if it is split at a point, $\mathbf{q}$, that is presumed to

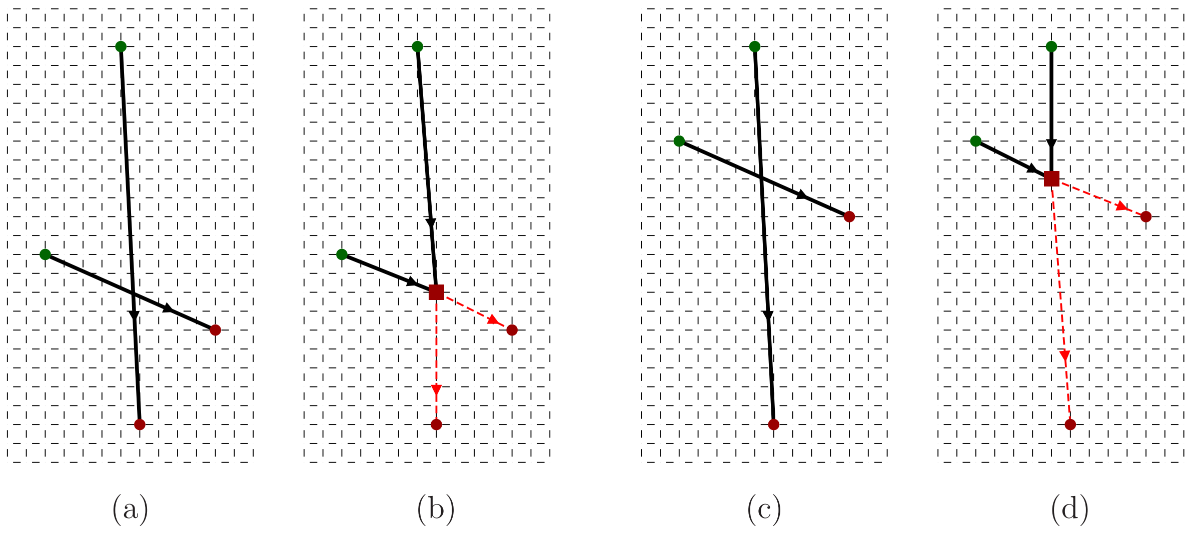


Figure 7.11: *Two before-and-after cases, in which splitting an edge with a steep negative gradient leads to the creation of a vertical edge going downwards, with the split point lexicographically outside the range defined by the two limit points. The downwards vertical edge can be either the new edge to the right that is sent to the buffer, as in (a) and (b), or the new edge to the left that replaces the original sweep-line edge, as in (c) and (d).*

lie in the interior of the edge. If a split occurs, the edge is replaced by an edge connecting either $\mathbf{l}$ to $\mathbf{q}$, or $\mathbf{q}$ to $\mathbf{r}$. The former action is taken when $\mathbf{q}$ has yet to be processed as an event point, as in figures 7.5, p.117, and 7.6, p.119; the latter is taken when the split occurs in consequence of $\mathbf{q}$ being processed as event point, as in figure 7.8, p.121. Either action ensures that a sweep-line edge invariably connects a point that has already been processed as event point with one that as yet has not.

The point $\mathbf{q}$ is selected either because it has been computed to be intersection point between that edge and another (figure 7.5), or because it is a pre-existing event point that has been detected to lie on the edge (figures 7.6 and 7.8). For an exact implementation, $\mathbf{q}$ lies exactly on the edge. Since $\mathbf{l} \prec \mathbf{r}$, it follows that $\mathbf{l} \prec \mathbf{q} \prec \mathbf{r}$; hence, both edges formed by the split are in the correct lexicographic order.

It does not necessarily follow that $\mathbf{l} \prec \mathbf{q} \prec \mathbf{r}$ for an implementation using approximate arithmetic. The handling of pre-existing event points is not a problem. The rules used to test whether a point lies on an edge (section 6.5, p.101) ensures that for any edge, no point outside the lexicographic range defined by the two limit points is ever deemed to lie on the edge. However, it is possible for a computed intersection point between two edges to lie outside the lexicographical range for one of the edges. As I show later, in section 8.5, p.156, the intersection point between two edges can be computed in such a way that it always lies in the axis-aligned bounding box for each edge; hence, $x_l \leq x_q \leq x_r$ and $\min(y_l, y_r) \leq y_q \leq \max(y_l, y_r)$ for both edges concerned. This is not sufficient, however, to prevent a breach in the lexicographic constraint $\mathbf{l} \prec \mathbf{q} \prec \mathbf{r}$. Figure 7.11 shows that a breach can arise for edges with steep negative gradients, because one of the two edges formed can be a vertical edge going downwards. It is therefore necessary for an algorithm based on inexact arithmetic to be able to handle vertical edges that go downwards.

7.6 The algorithm using rounded arithmetic

I have considered in detail the variant form of the Bentley-Ottmann algorithm, as it would operate if implemented using exact arithmetic, and also the shortcomings of that form of the algorithm if it were to be implemented using inexact arithmetic. I now describe the actions required that enable the algorithm to function using inexact arithmetic.

7.6.1 Handling downwards vertical edges

As noted in the last section, a downwards vertical edge may emerge (when previously there was none) when two edges are split at a computed intersection point; for a sweep-line edge from $\mathbf{l}$ to $\mathbf{r}$ ($\mathbf{l} \prec \mathbf{r}$) split at intersection point $\mathbf{q}$, either the new sweep-line edge, $\mathbf{l}$ to $\mathbf{q}$, or the buffer edge, $\mathbf{q}$ to $\mathbf{r}$, may emerge as a downwards vertical edge.

When the buffer edge is downwards vertical ($x_q = x_r$, $y_q > y_r$) the situation can be handled by reversing the two vertices, and negating the multiplicity value. This is achieved automatically if the buffer edge collection is represented as a `PolySet2` class object (see section 7.1.2, p.110).

When the new sweep-line edge is downwards vertical ($x_q = x_l$, $y_q < y_l$) it is essential *not* to reverse this edge. Point $\mathbf{l}$ is potentially connected to edges that have already been sent to output, while $\mathbf{q}$ will be connected to buffer edges, to be handled like input edges yet to be processed, and possibly also to true input edges; hence the respective roles of $\mathbf{l}$ and $\mathbf{q}$ as leftmost point and rightmost point must be retained, even though $\mathbf{q} \prec \mathbf{l}$. As is standard, the intersection point $\mathbf{q}$ will be added as an event point to be processed. This represents a backwards lexicographic progression, but there is no backwards progression in x. This aspect of the approximate arithmetic implementation causes no problem for the execution of the algorithm. However, a possible consequence is that the result may contain overlapping vertical edges.[9]

[9]It is possible to merge vertical edges for a specific x-coordinate value by means of a 1D simplification process. The test program discussed later merges vertical edges in this way. This extra action resolves geometric errors in the output due to coincident vertical edges and reduces the amount of data, but it does not affect (in absolute terms) the robustness of the algorithm. Other forms of geometric error in the output are not affected by such action.

7.6.2 Handling incorrectly grouped sweep-line edges

The approach taken when the three categories of sweep-line edge do not form separate groups in the order expected is as follows: Every edge in the sweep-line structure is examined in turn, starting from the first, until one is found that either ends at the event point being processed, or that does not (according to arithmetic computations) pass under the event point. Any edges preceding the one found are left as they are, as edges that pass under the event point. Similarly, every edge in the sweep-line structure is examined in turn starting from the *last*, until one is found that either ends at the event point or is determined to pass over the event point. Any edges considered before this one are left alone, as edges that pass over the event point. This leaves a collection of edges, which in an exact implementation would consist only of edges that end at or pass through the event point, but may in fact include edges that pass over or under the event point. The algorithm treats all edges in this collection that do not end at the event point as edges that pass through the event point. Hence each one is split at the event point, with the portion to the left being sent to output, and the portion to the right being treated as a new sweep-line edge to be added to the sweep-line structure. The justification for this approach is based on the assumption that the edges that do pass over or under the event point do so by a small distance. This accuracy of this approach is discussed in section 8.4, p.154.

7.6.3 Robustness of the method

In section 3.1, p.47, I set conditions by which an algorithm might be considered robust: that the algorithm terminates within a specified time bound, that it uses memory resources within a specified bound, that the result is valid, and that the result is accurate to within a specified error bound.

It can be demonstrated that the algorithm maintains topological validity of the shape representation, both while determining the cell structure, and also while determining the simplified shape representation from the cell structure. Recall that the condition for topological validity of a polygonal shape is that a vertex must act as start-vertex to an edge in the boundary representation as many times as it acts as end-vertex (section 4.2, p.58). Since multiple edges may be included in the structure, the count must take into account the multiplicity value for each edge. We may observe that the process of determining the cell structure is a series of topological operations, whereby either an edge is split at a specific point, or else two edges with identical limit points are merged, with the multiplicity values aggregated. Both these operations maintain topological validity. Recall also that the process of determining the simplified shape representation operates by retaining only those edges of the computed arrangement for which the cell on one side is to be considered internal for the representation of the result, and the cell on the other side external; each retained edge is oriented so that the internal cell lies to the left (section 7.4.3, p.121). From this it follows that each vertex acts as start-vertex and end-vertex to identically many edges.

The algorithm is not iterative, and always terminates in polynomial time: $O(n(n+I))$ at worst, where n is the number of input edges and I is the number of intersection points; the maximum internal memory required is $O(n)$.

The issue of accuracy of the algorithm is discussed later, in section 8.4, p.153. This shows that there is a bound on the geometric error of the result, dependent on the complexity of the problem. The complexity is measured in terms of the maximum number of adjustments to an original edge, which is $O(n)$ at most. The derived error bound, which is proportional to the complexity measure, is defined in terms of distortion of the original boundary within the result, and also the precision to which the decisions made relating to the result may be considered acceptable.

The test results described in the next section show that it is possible for geometric errors to appear in the output; hence, we cannot say that the algorithm is fully robust.

We may therefore class the sweep-line simplification algorithm as topologically robust, if the efficiency and accuracy properties described above are accepted as sufficient. This makes it superior to other algorithms I investigated, which cannot be so classed; these algorithms were prone in certain circumstances either to iterate excessively, or alternatively, to generate a result with an unacceptably high geometric error. Two trial algorithms that were found to be problematic are briefly described in section 7.8.

7.7 Tests

The algorithm just described for performing the boundary self-intersection and simplification processes on 2D polygonal shape representations is one among several that were implemented and tested. In order to test these algorithms easily and interactively upon a suitable range of shape representations I developed a program capable of constructing the data required and invoking the algorithms. The purpose of the tests was primarily to investigate the robustness of the algorithms, rather than their efficiency. Since the tendency is for geometric algorithms to fail only in rare cases, the cases that formed the basis for the tests were not 'typical' ones likely to arise in applications; on the contrary, for the most part they were highly contrived examples considered likely to increase the possibility of the algorithm failing or showing up other shortcomings. The generated shape representations were all topologically valid, as required by the simplification operation, but tended to be geometrically invalid with a high degree of complexity, with many pairs of boundary edges intersecting or in close proximity. Furthermore, large shifts were often applied to the data, in either or both the x and y directions; hence, the length scales of manifested features would be not much larger than the spacing between representable points.

The principal aims of the tests can be categorised as follows:

- To test for unexpected conditions arising from the computations. This is good practice in the development of software for the purpose of identifying coding bugs, but here it was important also for identifying faulty arguments that had been assumed to support the robustness of a candidate algorithm. Especially important were tests for conditions that prevented the algorithm from progressing, or that manifested a breach in the expected topological constraints in intermediate data or in the final result.
- To check that there is no excessive propagation of new vertices and edges in certain situations. For exact computations in which the initial shape representation has n edges, the number of new vertices in the result of the boundary self-intersection, arising from non-degenerate intersections between pairs of edges, is bounded by $O(n^2)$; the number of edges in the resulting shape representation is therefore also bounded by $O(n^2)$. There is no automatic guarantee in general that this bound will apply when the computations are inexact.
- To check that there is no excessive use of resources, namely CPU time and memory. While not primarily concerned with efficiency, we need to be alert to complexity of non-polynomial degree of any algorithm.
- To check that the extent of numerical error relating to the result of an operation is not excessive.

All code was written in C++, and developed using the Gnu compiler `gcc` version 3.4.4 20050721 (Red Hat 3.4.4-2). The principal program used for the tests was developed as a GUI event-driven application using the Qt 3 toolkit [BS04]. For the most part, operations are controlled by a command-line input, thus making it straightforward to add new functionality as and when required. Commands were introduced to create and manipulate shape representations, to display the data in text format, to save and restore data, both temporarily in core and permanently on disc file, and ultimately, to invoke specific versions of the simplification algorithm. Within the program, commands are primarily invoked interactively through a display window; however, an ‘execute’ command exists to invoke a sequence of commands as stored on file, making it possible for the command sequence to be applied repeatedly. Many assertion checks were placed within the code for the simplification algorithms, forcing the program to abort on failure, in order to give assurance that the algorithm was functioning as expected.

Steps were taken to ensure that all real-valued computations adhered to the IEEE 754 standard for floating point arithmetic, as described later in section 8.1 [IEE85]. This was achieved by (1) switching off all compiler optimisation, and (2) ensuring all code is written to specify at most one floating-point operation per statement [Mon08].[10] The latter constraint makes it necessary to re-write the code for `yAtX()` and `interpolate()` as listed in tables 7.1 and 7.2. For example, the following line from `yAtX()`

```
double lambda = ( dxLLess ? dxL : dxR ) / ( r.x() - l.x() );
```

was re-written as:

```
double dx = r.x() - l.x();
double lambda = ( dxLLess ? dxL : dxR ) / dx;
```

The program maintains and displays the *current shape representation*, which is the result of the last operation to create or modify a shape representation. The boundary edges of this structure is displayed in vector graphics form on a screen window using the Qt graphics system. A number of fast keys allow easy zooming in and out and scrolling of the current shape on display. There is also within the program a separate graphics system, invoked by a specific command, in which a PostScript [PS86] display file of the current shape is generated and stored on file. The results of the program displayed in this document are generated by this means, as in figures 2.6, p.36, and 6.2, p.91. The boundary edges are displayed in vector graphics form. Arrow heads denote the direction of each edge, and a multiple count of arrow heads denotes the multiplicity. Optionally, interior regions are also shaded, each in a colour specific to the winding number of the region.

One shape representation used for testing is the random *topological polygon*. This is constructed from a cyclical sequence of vertices selected at random from within the unit-sized disc. Optionally, the pseudo-random number generator is reseeded at the start of the construction, in order to ensure that the same shape representation is generated each time. The example case shown in figure 6.2 uses the 8-sided random polygon.

[10]Prof. Alan Mycroft helped me to determine whether generated assembler code was IEEE 754 compliant, and gave useful background information. Monniaux's paper [Mon08], which came to our attention later, confirms these issues for Intel's IA32 architecture. Using `gcc` with compiler option `-ffloat-store` also achieves IEEE 754 compliance, provided there is one statement for each floating-point operation. Kettner et al. [KMP^{+}08] report taking similar steps to achieve IEEE 754 compliance for the three-point orientation test based on the cross product calculation. See section 8.1, p.145, for further discussion.

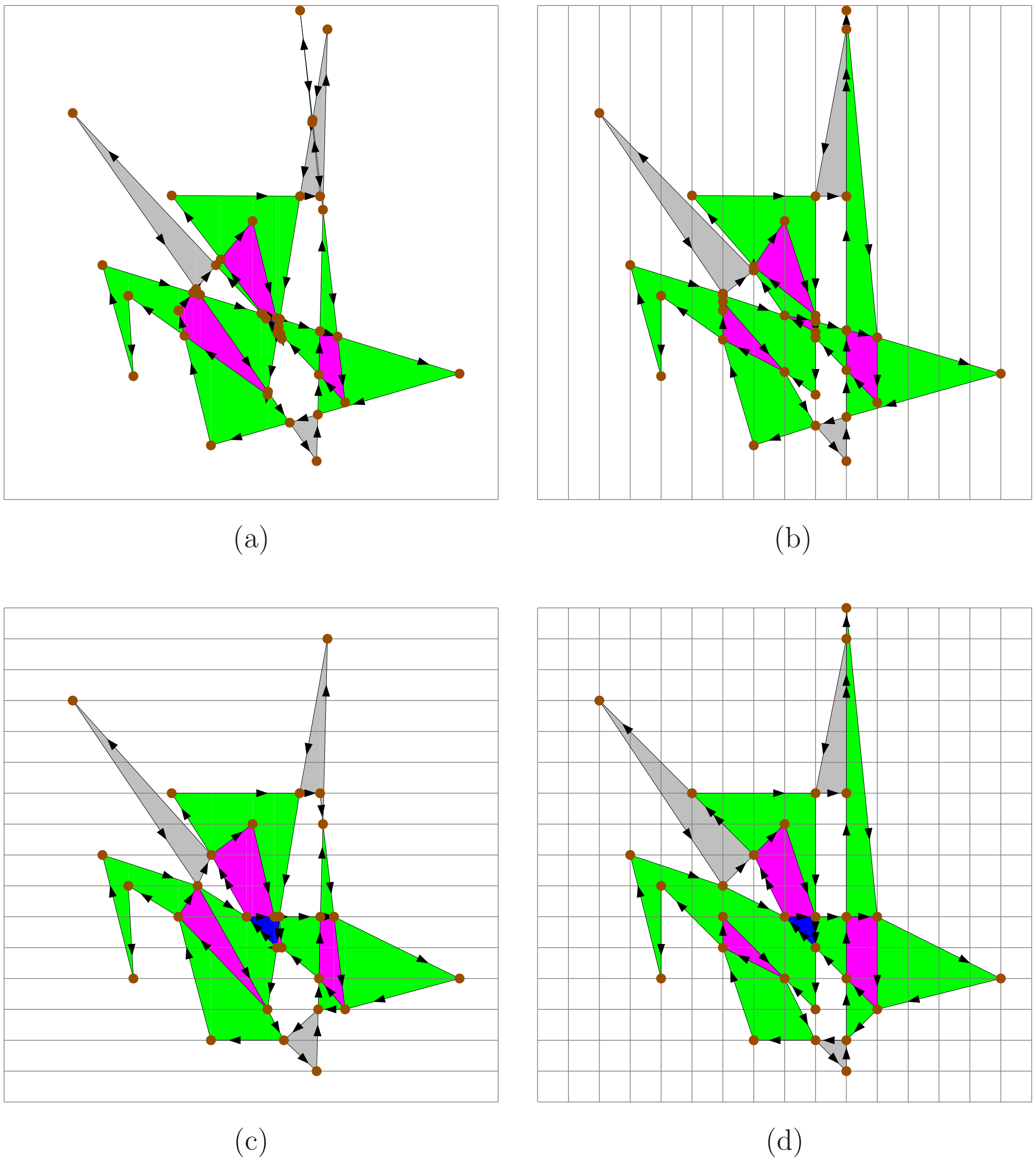


Figure 7.12: *The result of the boundary self-intersection operation upon the 16-sided random polygon, with (a) no displacement of the original data, (b) displacement in the x direction only, (c) displacement in the y direction only, and (d) displacement in both the x and y directions. The displacement applied here for both directions is 1.5×2^{49} (8.44×10^{14}), leading to a spacing of 0.125 between representable coordinate values.*

Figure 7.12 shows results of the boundary self-intersection based on the 16-sided random polygon; no displacement of the original data is applied in figure 7.12(a), while for the other cases displayed there is displacement in either or both the x and y directions. For all the tests I show only the cellular structure formed by the boundary self-intersection operation, since it is essentially this operation that is of concern; the simplified shape boundary is formed from a subset of the edges for the cellular structure (see section 7.4.3, p.121).

Another shape representation used for testing is the *polygonal star*—a regular shape representation based on the set of n evenly-spaced points on the unit circle. As with the random polygon, the star has many (indeed more) pairs of intersecting edges. However, there is also a high concentration of intersecting edges and of intersection points close to the centre, which one might assume is more likely to highlight any problems present in the algorithm. Furthermore, the regularity present in the original structure and therefore expected in the final result makes clearer the nature and the extent of the numerical error. Figure 7.13(a) shows the result of the boundary self-intersection based on the 12-sided polygonal star when no displacement is applied; the cases with displacement are shown in figure 7.13(b), (c) and (d).

The algorithm showed no serious problems. As one would expect, as the data becomes more complex, or the grid of representable points becomes coarser, the extent of data degradation in the result worsens. One may observe both in figures 7.12 and 7.13 that applying the large displacements distorts the original edges, resulting in a change to the topology of the result. In figure 7.13(a), for example, we see there are 12 small regions of winding number 4, marked yellow. In (b), (c) and (d) we see that some of these regions are enlarged, while others are squeezed out of existence, leaving behind multiple edges. As expected, the boundary self-intersection operation never failed in any of the experiments, nor generated a topologically invalid result.

Figure 7.14 shows the structure created by the boundary self-intersection operation upon the 64-sided star when a displacement of 1.5×2^{47} (2.11×10^{14}) is applied in both the x and y directions; figure 7.15 shows the central part of the generated structure in more detail. We see that the structure has degraded quite substantially at the centre, and that the outer parts are starting to deteriorate at this level of coarseness; the right-hand side, being downstream of the sweep process, has degraded more.

The algorithm does not always generate a geometrically valid result. This is the consequence of data changing as the result is computed. In the example just shown there is one edge intersected by two others in the operation result. Figure 7.16(a) shows the part of the structure that contains the intersections, while figure 7.16(b) shows the structure after a second application of the boundary self-intersection operation.

To complete the discussion on the modified Bentley-Ottmann algorithm, we revisit the table from [MY04, L.1 p.6] presented in chapter 1, p.19. Recall that this considers example

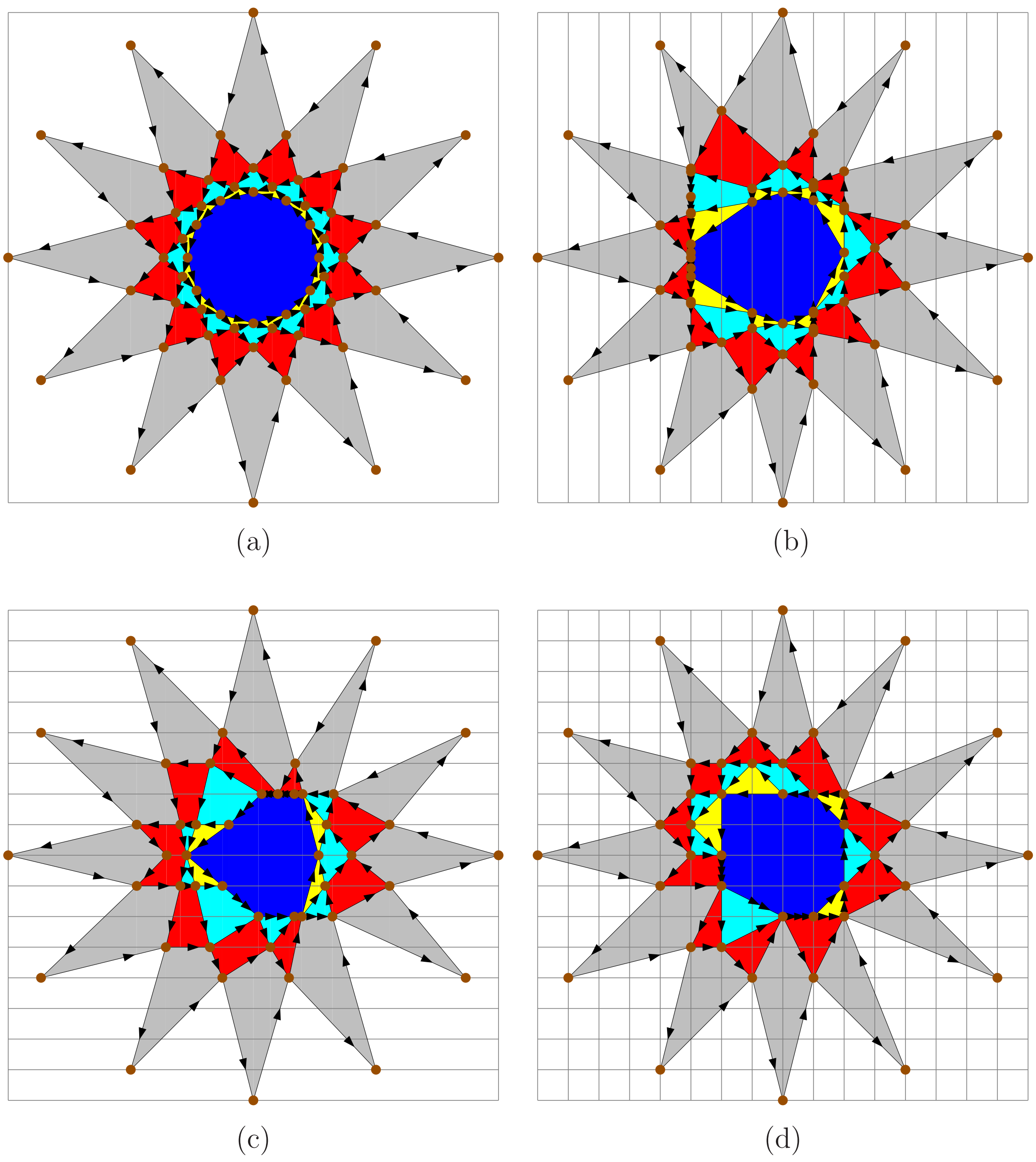


Figure 7.13: *The result of the boundary self-intersection operation upon the 12-sided polygonal star, using the same range of displacement options as for figure 7.12.*

cases of how geometric software copes with computing the union on the plane between an n-sided regular polygon and a copy of itself rotated by α degrees, listing the time taken and the success or otherwise of the operation. Table 7.4 is a repeat of that table, but with an extra column giving how long it takes for the modified Bentley-Ottmann algorithm to perform the same operation. The algorithm terminates to give the correct result in all cases. The times for the original software and for the algorithm presented are not strictly comparable.[11]

[11]The tests were run on different machines. There is no indication concerning the type of machine on

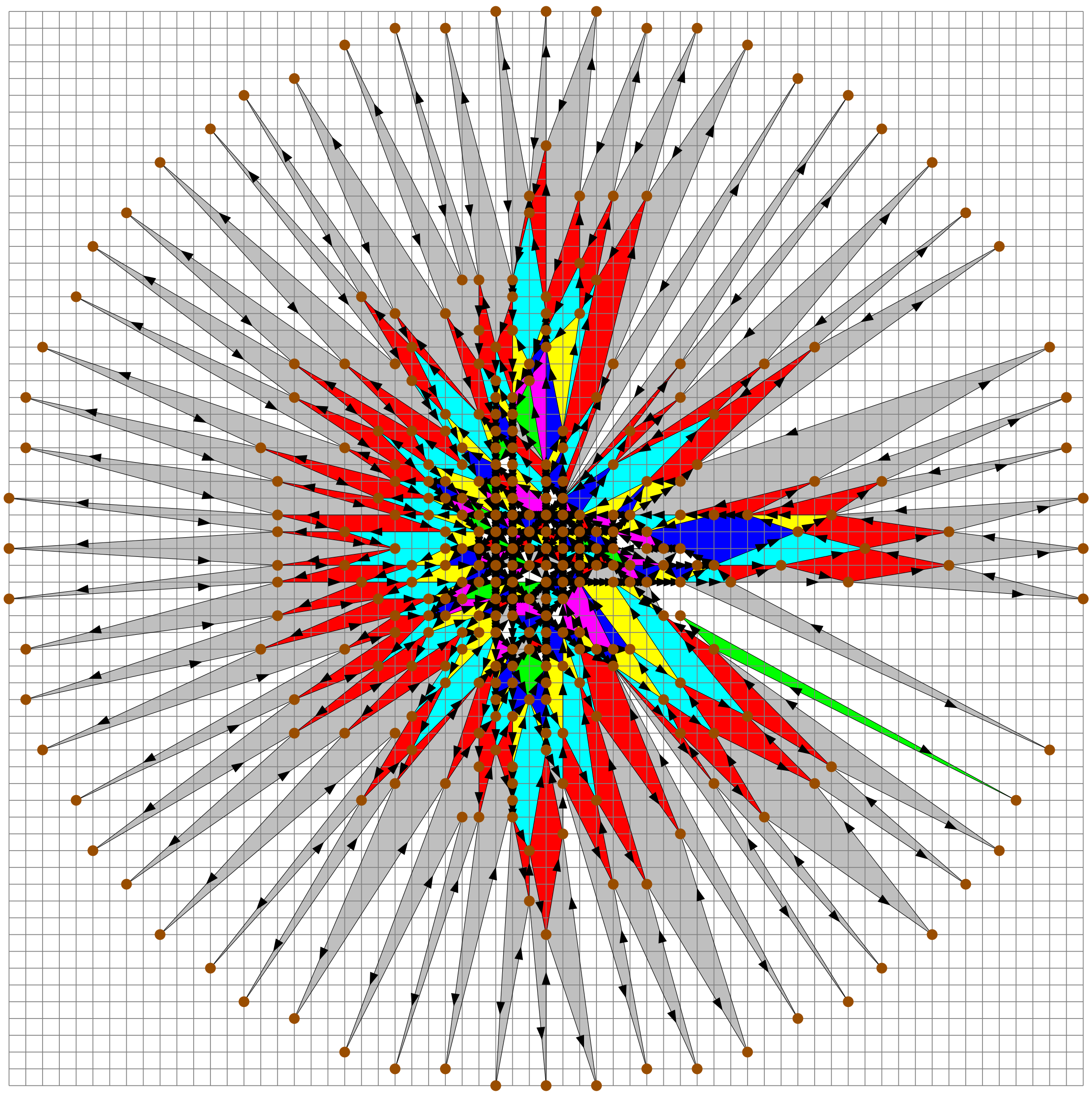


Figure 7.14: *The result of the boundary self-intersection operation upon the* 64*-sided star, with an x and y direction displacement of* 1.5×2^{47}.

which the original software was timed (the original table was first presented in 2001); my implementation of the modified Bentley-Ottmann algorithm was run on a 400MHz Pentium II, and compiled without optimisation (to ensure adherence to IEEE 754). Though the computer I used is not 'state of the art' for 2008, it may be reasonable to assume it to be about 10 times as fast as the computer used for the original tests. It would appear that the three commercial modellers worked on the 3D form of the specified problem (see footnote to the discussion in chapter 1, p.20.) One may expect the algorithm to compare well with the software systems listed in terms of speed (and reliability) for the near-degenerate cases concerned. ACIS is a tolerance-based system (presumably likewise for Microstation95 and Rhino3D) so special action is required for the cases considered, and LEDA/CGAL is slowed down because of the need to use filtered arithmetic, and presumably exact arithmetic too.

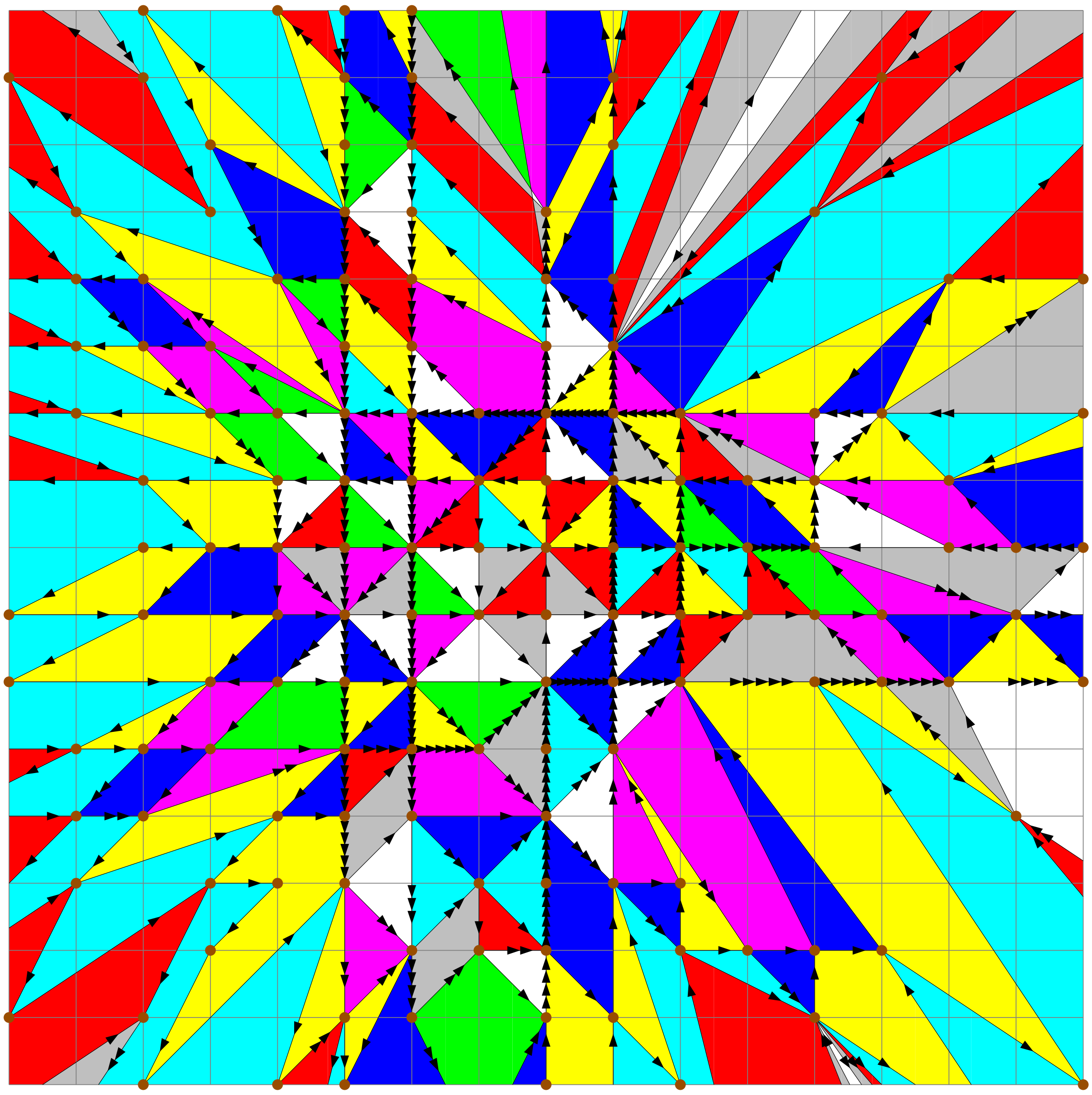


Figure 7.15: *As in figure 7.14, in detail.*

7.8 Other approaches considered

Various approaches were considered for implementing an algorithm for the simplification process. In my investigations the aim has been to devise an algorithm that may be classed as robust according to the requirements laid down in section 3.1, p.47. Two potential problems are particularly important to identify, because they rule out an algorithm from being classed as robust: those relating to severe inefficiency, and those relating to severe inaccuracy in the result.

The sweep algorithm described in section 7.6, p.127 is able to avoid these two particular problems; the algorithms I tried based on other approaches were not. I briefly describe two approaches to demonstrate the difficulties in devising a robust algorithm.

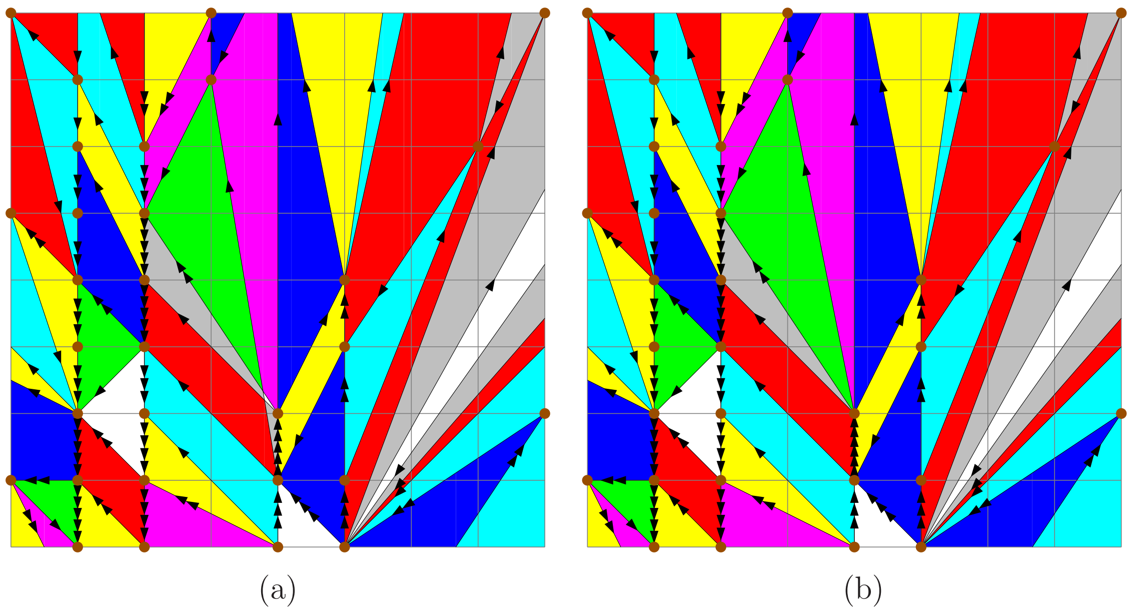


Figure 7.16: *As in figure 7.14. (a) shows the parts of the generated structure with intersecting edges; (b) shows the same area after the boundary self-intersection operation is applied a second time.*

SYSTEM	n	α	TIME	OUTPUT	B-O TIME
ACIS	1000	1.0e-4	5 min	correct	0.5 sec
ACIS	1000	1.0e-5	4.5 min	correct	0.6 sec
ACIS	1000	1.0e-6	30 sec	too difficult!	0.5 sec
Microstation95	100	1.0e-2	2 sec	correct	0.1 sec
Microstation95	100	0.5e-2	3 sec	incorrect!	0.1 sec
Rhino3D	200	1.0e-2	15 sec	correct	0.2 sec
Rhino3D	400	1.0e-2	–	crash!	0.2 sec
CGAL/LEDA	5000	6.175e-6	30 sec	correct	2.3 sec
CGAL/LEDA	5000	1.581e-9	34 sec	correct	2.3 sec
CGAL/LEDA	20000	9.88e-7	141 sec	correct	9.0 sec

Table 7.4: *Repeat of table 1.1, together with the times taken for the same problem to be solved using the modified Bentley-Ottmann algorithm. Original table from [MY04, L.1 p.6].*

One approach tried was a very simply specified algorithm for determining intersection points. It is defined by the following pseudocode:

```
while boundary edge collection contains 2 edges that intersect do
    determine intersection point
    split each edge into 2 edges, with new vertex at intersection point
end while
```

In this algorithm, edges connected at their limit points are are not considered to intersect, but edges are considered to intersect when the limit point of one edge is deemed to lie in the interior of the other. Once all boundary self-intersections are resolved, a subsequent process determines the winding numbers on either side of each edge.

The algorithm described for determining all boundary self-intersections is not particularly efficient. An exact implementation would run in $O(n^2)$ time, compared to $O(n \log n)$ for Bentley-Ottmann for problems of bounded complexity. In functional terms this algorithm would be suitable if implemented in exact arithmetic, since the geometry does not change on splitting the edges, and no new intersections can emerge. We cannot rely on this for an approximate implementation, because each time an intersecting edge-pair is processed, the repositioning of the edges (now split) may lead to new intersecting edge-pairs. It is possible for this to happen repeatedly, leading to a laddering effect as in figure 7.17. This particular effect is liable to occur when the spacing between x-coordinate values is much larger than that between y-coordinate values, or equivalently, when the edges are close to vertical. For certain more complex cases the process fails to terminate in reasonable time. Without a resolution of all boundary self-intersections it is not possible (nor meaningful) to go on to determine the winding number values.

Another algorithm tried was a sweep-line algorithm in which all intersection calculations are carried out on the original data, as opposed to the data currently maintained (as in the sweep algorithm just described). An algorithm that performs all intersection calculations based on original data might be considered superior on grounds of accuracy. For such an algorithm, the error bound on the distortion of an edge would be no greater for an edge intersected several times than for an edge that is intersected only once. In contrast, for the sweep-line algorithm just considered that relies on modified data the distortion may accumulate. Hence, if k intersections are found that are associated with a particular original edge, the representation of that edge in the result (a $(k{+}1)$-segment polyline) may be distorted as much as k times the maximum distortion for an edge found to have just one intersection.

For the sweep-line algorithm that relies on original data a strategy must be devised on how to handle any discrepancy that may arise between the logical ordering and physical positioning of edges with respect to the event point. The strategy I devised was to force the edge concerned to be cracked at the event point. This turned out not to be suitable,

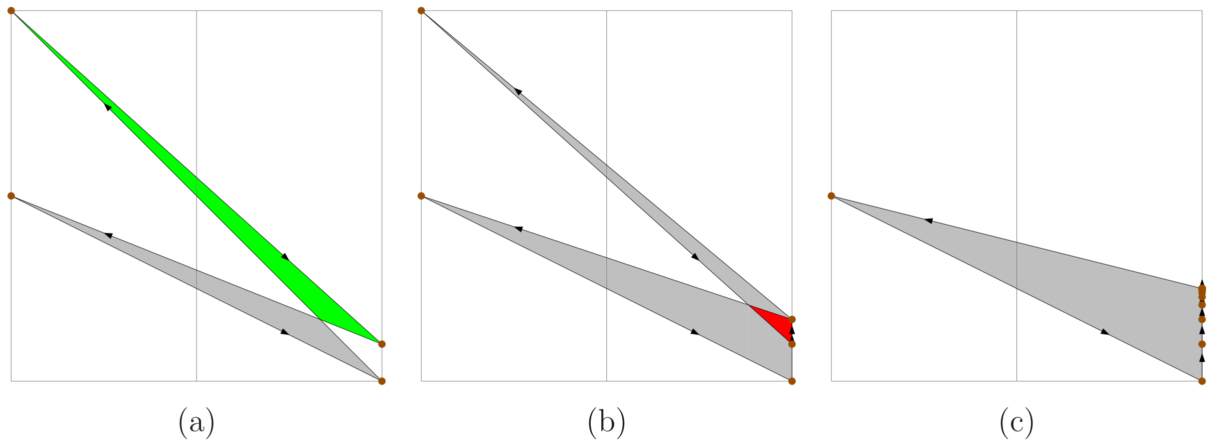


Figure 7.17: *Example of the laddering effect that can occur with the simplistic algorithm in which pairs of intersecting edges are split until no intersections remain. In the original structure (a) the x-coordinate values used are displaced so that the spacing between representable values is half the span (but not the y-coordinates). On splitting the one pair of intersecting edges at the computed intersection point, a new pair of intersecting edges emerges, as in (b). All boundary intersections are resolved only after processing intersecting edge pairs a further 48 times, as shown in (c).*

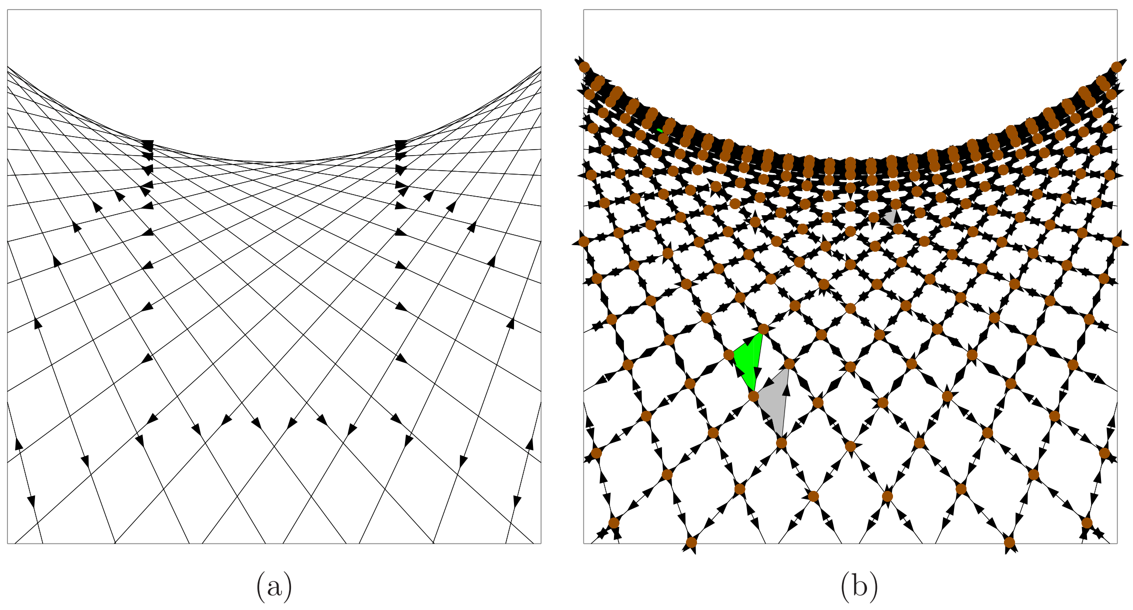


Figure 7.18: *(a) shows an initial data structure consisting of a star-shaped topological polygon assembled with a near-exact copy with minor distortions and with edges reversed. (b) shows the result produced by the sweep algorithm in which the intersection calculations are based on original data. The edges from the two polygons are very close that they appear coincident in (a). The result in (b) should also have pairs of near-coincident edges, and for the most part this is so. However, the diagram shows three cases where an edge is seen to be highly distorted in consequence of the policy of cracking edges that are wrongly positioned with respect to a particular event point.*

because there is potentially no limit to the extent of the distortion applied to an edge that is cracked. See figure 7.18 for an example case showing this problem. Here, the algorithm decides on a particular ordering of very short edges that follow on from an intersection point, but it becomes clear that the opposite ordering would have been appropriate only with the handling of much larger edges that follow the short edges.

The shortcomings of the two algorithms just described demonstrate the difficulties in devising robust algorithms. It is not sufficient to devise a variant form of either algorithm just described that resolve the particular problem case, if that algorithm fails to terminate in reasonable time for other cases, or gives rise to large errors in the result.

This concludes the chapter describing how the Bentley-Ottmann sweep-line algorithm can be modified to perform the 2D simplification process using inexact arithmetic. The form of the algorithm that relies on the boundary structure as currently maintained is topologically robust. Although the result may be marginally geometrically invalid, the algorithm has an advantage over the basic Boolean algorithm insofar that it acts to resolve any geometric errors in the input structure. Certain aspects of the code were described, and the chapter finished by noting the difficulties of other approaches. The one aspect not yet considered is the extent of the error in the result. This is discussed in the next chapter.

Chapter 8

Arithmetic precision and error analysis

This chapter considers the issue of accuracy for 2D computations, both for the basic Boolean algorithm and the simplification algorithm.

The basic Boolean algorithm guarantees a topologically valid result because of the dependency relationship of operations. Indeed, topological robustness is guaranteed irrespective of how erratic the arithmetic operations $+, -, \times, \div$ might be; the only requirements are that such operations should always return the same result for the same input data, and that no failure (such as overflow) should occur. Nevertheless, it is important to know what guarantees there are about the accuracy of the computed result. Errors arise from the arithmetic calculations used for determining whether and where two entities intersect; the use of error analysis, as explained in this chapter, is a useful tool for analysing the extent of any such error.

The 2D simplification algorithm described also relies on interpolation calculations. Indeed, the 2D simplification algorithm and the 2D basic Boolean algorithm perform essentially the same calculation for determining where two segments intersect on the plane. The same error analysis therefore applies to both operations for this particular calculation. The analysis is considered in detail in 8.2. However, the simplification process as a whole differs in two respects: firstly, the determination of the location of an intersection point on a particular edge may depend on a sequence of intersection calculations relating to that edge, because of the need to use modified data at each stage; secondly, there is sometimes the need for data to be adjusted, as a special action required for situations that in theory would be impossible for an exact computation, but which can occur for an inexact computation. These special actions are also a source of error, and for a candidate algorithm to be considered acceptable there must be a clear argument for a reasonable bound on that error. Section 8.4, p.154 specifically considers this issue for the simplification algorithm.

The most obvious approach to take is for all real values in the calculations to be represented using IEEE floating-point, single or double precision, as is readily available in programming languages and on machine hardware. The implementation of the simplification algorithm considered here uses double precision. An alternative approach would be to use integer arithmetic, or some form of fixed-point real arithmetic (which can be implemented using integer values). Whatever approach is taken, any error analysis of the algorithm depends on how real values are represented, and also on the precise rules for the rounding of values in the calculations. We also need to be alert to the possibility of failure, for example, due to overflow or division by zero.

The analysis presented in this chapter is based for the most part on the assumption that all real values are represented and that all real-valued calculations are carried out in accordance with the ANSI/IEEE standard 754-1985 as in 'IEEE Standard Binary Floating-Point Arithmetic' [IEE85]; real values are assumed to be all double precision, or alternatively all single precision (but not a mixture of the two). Section 8.1 describes the principles of IEEE floating-point arithmetic, and how error analysis can be used for calculations using such arithmetic. Section 8.2 presents an analysis providing an error bound on the segment intersection calculation, while section 8.3 reports on the extent of the error in practice. Section 8.4 discusses overall accuracy for both the basic Boolean algorithm and the sweep-line simplification algorithm. Finally, section 8.5 briefly considers other issues such as overflow.

No consideration is given to the accuracy of the data-smoothing process. The smoothing process differs from the basic Boolean and simplification algorithms in that it seeks to adjust the boundary to resolve marginal features; each local adjustment to the boundary can introduce error of magnitude up to the specified tolerance in addition to any rounding error. The tolerance specified needs to be larger, and may be significantly larger, than any rounding error that may have accumulated in this and in previous operations. If the specified tolerance is ε, we can say the boundary is distorted by at most $k\varepsilon$, where k is the maximum number of adjustments that might affect the boundary at any one place. To place a limit on the boundary distortion, we would need to determine a bound on k or on the total number of adjustments. It is not immediately clear how one would do that for specific cases; furthermore, judging from the one reported problem case in the commercial CAD product (see section 5.2, p.83) there would appear to be no global limit to the number of adjustments required to resolve marginal data.

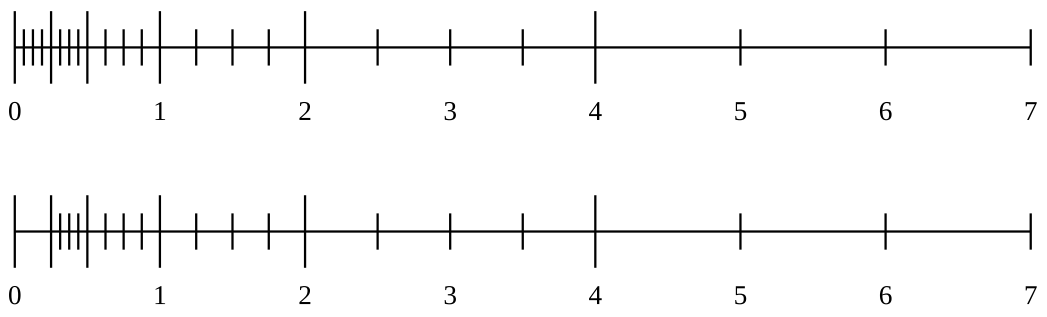


Figure 8.1: *The positive and zero values of the floating-point system $F(2, 3, -1, 3)$ in standard form (top) and normalised form (bottom).*

8.1 Floating-point arithmetic

I discuss here the principles of floating-point arithmetic, and how in general the issue of precision influences the error in algorithms implemented in floating-point arithmetic. This topic is covered in a number of texts, e.g. [Hig02, Gol91, MY04]; the description given in here is largely influenced by [Hig02].

The finite set $F(\beta, t, e_{\min}, e_{\max}) \subset \mathbb{R}$ designates the *floating-point system* as specified by the natural numbers $\beta \geq 2$ and $t \geq 1$, known respectively as the *base* (or *radix*) and *precision*, and integers $e_{\min}$ and $e_{\max}$ ($e_{\min} \leq e_{\max}$) giving the *exponent range*. F includes all numbers of the form:

$$y = \pm m \times \beta^{e-t} \tag{8.1}$$

for integer values m and e in the respective ranges $0 \leq m \leq \beta^t - 1$ and $e_{\min} \leq e \leq e_{\max}$; m is known as the *mantissa* (or *significand*) and e the *exponent*. There is also the concept of a *normalised floating-point system*, for which the constraint for m is $\beta^{t-1} \leq m \leq \beta^t - 1$; the number 0 is considered also included in the normalised system as a special case.

The floating-point system can alternatively be specified by:

$$y = \pm \beta^e \sum_{i=1}^{t} \frac{d_i}{\beta^i} \tag{8.2}$$

with integer d_i representing the ith digit of the mantissa, lying in the range $0 \leq d_i \leq \beta - 1$; for a normalised (non-zero) value there is the additional constraint that $d_1 \neq 0$. Values within a system that are *not* in the normalised system are known as *subnormal*. The numbers represented by the system $F(2, 3, -1, 3)$ in both the standard and normalised form are shown in figure 8.1 [Hig02, pp.36-38].

Non-zero normalised numbers lie in the *normal range* of F, defined as $[-\beta^{e_{\max}}(1-\beta^{-t}), -\beta^{e_{\min}-1}] \cup [\beta^{e_{\min}-1}, \beta^{e_{\max}}(1-\beta^{-t})]$. Subnormal numbers lie in the *subnormal range* $(-\beta^{e_{\min}-1}, \beta^{e_{\min}-1})$. Number 0, though part of the normalised system, lies in the subnormal range. The union of the two ranges, $[-\beta^{e_{\max}}(1-\beta^{-t}), \beta^{e_{\max}}(1-\beta^{-t})]$, is known simply as the *range* of F [MY04, Lecture 2, p.5].

The quantity $\epsilon_M = \beta^{1-t}$, known as *machine epsilon*, has the significance that $1+\epsilon_M$ is the smallest representable number > 1 (assuming $1+\epsilon_M$ lies in the normal range, as is the case for floating-point systems normally considered). For y lying in the range $\beta^e \leq |y| \leq \beta^{e+1}$ within the normal range, the spacing between two neighbouring numbers is $\epsilon_M \beta^e$. It follows that the spacing between a representable number, y, and its neighbour is no greater than $\epsilon_M |y|$ provided y lies in the normal range.

The function $fl : \mathbb{R} \to F$ mapping a real argument to the closest representable number is known as the *rounding function*, and the quantity $u = \frac{1}{2}\epsilon_M = \frac{1}{2}\beta^{1-t}$ is known as the *unit roundoff*. It follows that for any real value of x lying in the normal range there exists a value δ such that

$$fl(x) = x(1+\delta) \qquad |\delta| < u \tag{8.3}$$

Also, for any $|x| \leq L = \beta^e$ for integer e ($e_{\min} \leq e \leq e_{\max}$) there exists a value Δ such that

$$fl(x) = x + \Delta \qquad |\Delta| \leq uL/\beta \tag{8.4}$$

Either of these two relationships can be used in error analysis to place an error bound on particular computed value.

In the analysis that follows, we shall assume the use of a binary floating-point system ($\beta = 2$), which is the norm for computer systems today. Calculators use base $\beta = 10$, and in earlier times IBM 3000 series computers used base 16.

Almost all computer hardware today is able to comply with the IEEE standard 754-1985 for binary floating-point arithmetic, as published in [IEE85]. 32-bit single precision numbers and 64-bit double precision numbers, of type `float` and `double` in C and C++, are based respectively on the floating-point systems $F(2, 24, -125, 128)$ and $F(2, 53, -1021, 1024)$; subnormal values are included. Consequently, the value for u is respectively 2^{-24} (5.96×10^{-8}) and 2^{-53} (1.11×10^{-16}). In addition, there are 80-bit double-extended precision numbers, known as type `long double` in C/C++; the floating-point system is not precisely specified for these, but the constraint on precision is that $t \geq 64$ [Kah97].

The IEEE standard states that for the default mode, values are rounded to the nearest representable number, with the mantissa being rounded to an even value in the event of a tie.[1] So, if op represents a particular binary operation $+$, $-$, $\times$ or $\div$, the equivalent floating-point operation $\widehat{\mathrm{op}}$ on values $a, b \in F$ returns the value:

$$a \mathbin{\widehat{\mathrm{op}}} b = fl(a \mathbin{\mathrm{op}} b) \tag{8.5}$$

provided the exact result $a \mathbin{\mathrm{op}} b$ lies in the range of F [Hig02, pp.37-40].

Although IEEE 754 has brought standardisation to computer hardware, compilers and programming languages are considered lacking in their support for the standard [Mon08, Sev98]. Arithmetic operations as specified in a program do not necessarily adhere to the standard, and so may be interpreted differently on different platforms. Furthermore, if an expression such as `x+y` or `x<1` is specified in two places in a program, and no change to `x` or `y` is specified in the mean time, it does not necessarily follow that the two instances of the expression will yield the same result. One cause for such discrepancies is the use of 80-bit internal floating-point registers on the Intel platforms with IA32 architecture (386, 486, Pentium etc.) [Mon08]. Compilers can produce code that is more compact and efficient, and with more local accuracy, by retaining variables in the registers, rather than by storing register values in memory. However, it is not always possible for compilers to perform all operations within the register, so certain variables may be stored in a 64-bit memory cell. With optimisation, all variables are susceptible to this problem, including arguments and return values of function calls that are 'inlined' by the compiler. Even without optimisation, the compiler is allowed to retain in the register 'temporary' variables—the unnamed intermediate variables implied by statements that incorporate more than one floating-point operation. It is beyond the scope of this dissertation to consider this matter any further. The analysis presented here assumes adherence to IEEE 754, and that all real values are represented to the same precision. As stated in section 7.7, p.131, all code was built so that it adhered to IEEE 754 and used only double-precision floating-point numbers.

8.2 Error analysis for 2D intersection calculation

This section considers the numerical accuracy of the code presented in section 7.1.1, p.106, for the critical operation of determining the point of intersection between two segments on the plane. I show that when two segments are deemed to intersect, the computed point of intersection always lies within a specific distance from either segment, equal to a fixed multiple of the arithmetic precision of the original vertex data. The analysis is based on the assumption that the compiled program adheres to IEEE 754; hence, every

[1] The other rounding options available are to round towards $+\infty$, $-\infty$ or 0. The round-to-∞ options are useful for implementing interval arithmetic.

expr.	*formula*	*range*	*rel. error*	*abs. error*
x_L		$[-L_X, L_X]$	0	0
y_L		$[-L_Y, L_Y]$	0	0
x_R		$[-L_X, L_X]$	0	0
y_R		$[-L_Y, L_Y]$	0	0
x_P		$[-L_X, L_X]$	0	0
Δx_L	$x_P - x_L$	$[0, L_X]$	$\pm u$	$\pm\frac{1}{2}uL_X$
Δx	$x_R - x_L$	$(0, 2L_X]$	$\pm u$	$\pm uL_X$
λ	$\Delta x_L/\Delta x$	$[0, \frac{1}{2}]$	$\pm 3u$	$\pm\frac{5}{4}u$
Δy	$y_R - y_L$	$[-2L_Y, 2L_Y]$	$\pm u$	$\pm uL_Y$
Δy_L	$\lambda\Delta y$	$[-L_Y, L_Y]$	$\pm 5u$	$\pm\frac{7}{2}uL_Y$
y_S	$y_L + \Delta y_L$	$[-L_Y, L_Y]$		$\pm 4uL_Y$

Table 8.1: *Errors on determining the y-value for a given x-value in a segment. The absolute error bound of y_S is ascertained by considering relative error bounds up to the computation of λ, and the absolute error bounds beyond that. The error bounds of intermediate expressions that are not relevant to this calculation are printed faint.*

expression is evaluated as a binary operation upon subexpression values that have been rounded to the relevant floating-point system—single or double precision—before being rounded itself.

Either or both formulae for the bounds on rounding (equations (8.3) and (8.4)) can be used as appropriate. It is appropriate to determine a bound on *absolute* error for addition and subtraction, because the subtraction of two nearly equal values of the same sign can make the relative error arbitrarily large. Conversely, for division it is appropriate to determine a bound on *relative* error, because a small denominator can lead to arbitrarily large absolute error. Either error bound is suited for multiplication. When the bound on error is small compared to the value (as is often the case) it is assumed that the compound effect of the errors can be ignored. So, for example, if the relative error in the evaluations of expressions a and b is bounded respectively by positive values $e_a, e_b \ll 1$, the bound in relative error for $a.b$ can be assumed to be $e_a + e_b + u$ when in fact the true bound is $(1 + e_a)(1 + e_b)(1 + u) - 1$.

Consider the first stage of the calculation—determining the y-value on a segment for a specific x-value—as shown in figure 7.1(a), using the code in table 7.1, p.107. For the sake of the analysis, we can assume without loss of generality that all values are computed in accordance with the Boolean variable `dxLLess` being `true`. The analysis presented is based on the assumption that $x_P - x_L \leq x_R - x_P$. The error analysis for the case when `dxLLess` is `false` is identical, except that certain values are negated and/or swapped. The

expr.	*formula*	*range*	*rel. error*	*abs. error*
x_L		$[-L_X, L_X]$	0	0
x_R		$[-L_X, L_X]$	0	0
y_{AL}		$[-L_Y, L_Y]$		$\pm 4uL_Y$
y_{BL}		$[-L_Y, L_Y]$		$\pm 4uL_Y$
y_{AR}		$[-L_Y, L_Y]$		$\pm 4uL_Y$
y_{BR}		$[-L_Y, L_Y]$		$\pm 4uL_Y$
Δx	$x_R - x_L$	$[0, 2L_X]$	$\pm u$	$\pm uL_X$
Δy_L	$y_{BL} - y_{AL}$	$[0, 2L_Y]$		$\pm 5uL_Y$
Δy_R	$y_{BR} - y_{AR}$	$[-2L_Y, 0]$		$\pm 5uL_Y$
Δy_A	$y_{AR} - y_{AL}$	$[-2L_Y, 2L_Y]$		$\pm 9uL_Y$
Δy_B	$y_{BR} - y_{BL}$	$[-2L_Y, 2L_Y]$		$\pm 9uL_Y$
$\Delta\Delta y$	$\Delta y_L - \Delta y_R$	$(0, 4L_Y]$		$\pm 12uL_Y$
λ	$\Delta y_L / \Delta\Delta y$	$[0, \frac{1}{2}]$		
Δx_I	$\lambda \Delta x$	$[0, L_X]$		
Δy_{AI}	$\lambda \Delta y_A$	$[-L_Y, L_Y]$		
x_I	$x_L + \Delta x_I$	$[-L_X, L_X]$		
y_I	$y_{AL} + \Delta y_{AI}$	$[-L_Y, L_Y]$		

Table 8.2: *Errors on determining the intersection point between segments.*

inclusive inequality ($\leq$ rather than $<$) is assumed in order that it can be applied to both cases. Table 8.1 lists each expression involved in the computation, including any implicit expression not named in the code, together with the binary operation by which it is derived (if it is not an input value), its range, and the bounds on the relative and absolute errors. It is assumed that the three points lie within the box $[-L_X, L_X] \times [-L_Y, L_Y]$ where L_X and L_Y are integer powers of 2, and that all values in the calculation are in the normal range. The absolute error of the computed value of y_S is bounded by $4uL_Y$, i.e. four times the maximum spacing between adjacent floating-point numbers within the range $[-L_Y, L_Y]$ ($2.38 \times 10^{-7} L_Y$ for single precision and $4.44 \times 10^{-16} L_Y$ for double precision). Appendix B explains in detail how the error bounds for each variable are derived.

Next let us consider the second stage of the calculation in which the interpolation point is determined, as shown in figure 7.1(b), and using the code in table 7.2, p.108. We assume that the y-values as computed by the first calculation are such that it is segment A that lies below B to the left, and above it to the right, i.e. $y_{AL} \leq y_{BL}$ and $y_{BR} \leq y_{AR}$, with at least one of the inequalities being strict. If segment A, say, is vertical and B is not, y_{AL} and y_{AR} take respectively the y-values of the lower and upper end-points of A, and y_{BL} and y_{BR} both take the y-value of segment B at $x = x_L = x_R$.

For the error analysis we can assume without loss of generality that the Boolean variables `dyLLess` and `dyALess` are `true`; hence, for the sake of the analysis we assume that $y_{BL} - y_{AL} \leq y_{AR} - y_{BR}$ and $|y_{AR} - y_{AL}| \leq |y_{BR} - y_{BL}|$. Table 8.2 gives details of the error bounds; again it is assumed that all geometry is contained within the box $[-L_X, L_X] \times [-L_Y, L_Y]$ where L_X and L_Y are integer powers of 2. (Certain symbols used for table 8.1 are re-used to have a different meaning.)

Of the y-coordinate values y_{AL} and y_{BL}, one at least will be taken from an original vertex, and so will be exact; the other is likely to be derived by interpolation as in stage 1. No assumption can be made as to which of these two values is exact (without causing a loss of generality). The same applies to the y-coordinate values y_{AR} and y_{BR}.

One can make some initial progress in determining error bounds, in the same way as for table 8.1. However, it is not possible to determine relative error bounds for terms relating to y-coordinate values, and ultimately we cannot determine relative *or* absolute error bounds for λ and subsequent terms. This difficulty arises because $\Delta\Delta y$, though strictly positive, can be arbitrarily small. In fact, the *true* value of $\Delta\Delta y$ could well be negative or zero. The segments have been deemed to intersect by virtue of the signs of the *computed* values of Δy_L and Δy_R, when in fact it is the *exact* values that determine whether they truly intersect. Thus the code is determining a supposed intersection point between the two segments when in fact they might not intersect at all: the lines they lie on may be parallel or coincident, or they may intersect out of range of the two segments, possibly by a substantially large distance. This makes it meaningless to attempt to determine an error bound on the evaluation of λ, which may be very large or infinite, or of the intersection point (x_I, y_I) which depends on λ.

We can accept this discrepancy, because the general aim in our computations is to produce a result that is sufficiently accurate, rather than the correct result. What is important, though, is that the computed intersection point should lie within a bounded distance of both original segments in all circumstances (including the cases when there is no true intersection point). Thus, when each segment is split at the computed intersection point, the boundary it represents is distorted by no more than this bounded distance.

To be able to continue our analysis for λ and the expressions that follow in the table, it is convenient to disregard the true value of λ, which ceases to be meaningful for segments that are actually parallel or close to parallel. Instead, we compare the computed value of each expression with a value derived from the computed values of Δy_L and Δy_R but for which all other operations are exact. We determine bounds on the difference between these two values, which we label the *adjusted absolute error bound* or *adjusted relative error bound*, depending on whether we are considering the absolute or relative difference. We define a value $\overline{\lambda}$ in terms of the computed values of Δy_L and Δy_R, which we label $\widehat{\Delta y_L}$ and $\widehat{\Delta y_R}$:

expr.	*formula*	*range*	*comparison value*	*adjusted rel. error*	*adjusted abs. error*
$\Delta\Delta y$	$\Delta y_L - \Delta y_R$	$(0, 4L_Y]$	$\widehat{\Delta y_L} - \widehat{\Delta y_R}$	$\pm u$	$\pm 2uL_Y$
λ	$\Delta y_L / \Delta\Delta y$	$[0, \frac{1}{2}]$	$\overline{\lambda}$	$\pm 2u$	$\pm\frac{3}{4}u$
Δx_I	$\lambda\Delta x$	$[0, L_X]$	$\overline{\lambda}\Delta x$	$\pm 4u$	$\pm\frac{5}{2}uL_X$
Δy_{AI}	$\lambda\Delta y_A$	$[-L_Y, L_Y]$	$\overline{\lambda}\Delta y_A$		$\pm\frac{13}{2}uL_Y$
x_I	$x_L + \Delta x_I$	$[-L_X, L_X]$	$x_L + \overline{\lambda}\Delta x$		$\pm 3uL_X$
y_I	$y_{AL} + \Delta y_{AI}$	$[-L_Y, L_Y]$	$y_{AL} + \overline{\lambda}\Delta y_A$		$\pm 7uL_Y$

Table 8.3: *Errors on determining the intersection point between segments.*

$$\overline{\lambda} = \frac{\widehat{\Delta y_L}}{\widehat{\Delta y_L} - \widehat{\Delta y_R}} \tag{8.6}$$

We are able to derive adjusted error bounds of the computed values; these are given in table 8.3.

Having obtained ranges for the computed values of x_I and y_I we must now determine a limit on the extent to which the computed intersection point (x_I, y_I) distorts the two segments. To proceed with the analysis we define certain computed values (marked with a hat) in terms of certain exact values (with no markings), making use of a series of terms, α_i, each representing some unspecified value in the range $[-1, 1]$:

$$\widehat{\Delta y_L} = \Delta y_L + 5\alpha_1 uL_Y \tag{8.7}$$

$$\widehat{\Delta y_R} = \Delta y_R + 5\alpha_2 uL_Y \tag{8.8}$$

$$\widehat{x_I} = x_L + \overline{\lambda}\Delta x + 3\alpha_3 uL_X \tag{8.9}$$

$$\widehat{y_I} = y_{AL} + \overline{\lambda}\Delta y_A + 7\alpha_4 uL_Y \tag{8.10}$$

where $\overline{\lambda}$ is as defined in equation (8.6). The signed distances of the point (x_I, y_I) from the segments A and B are respectively:

$$e_A = \frac{(\widehat{y_I} - y_{AL})\Delta x - (\widehat{x_I} - x_L)\Delta y_A}{\sqrt{\Delta x^2 + \Delta y_A^2}} \tag{8.11}$$

$$e_B = \frac{(\widehat{y_I} - y_{BL})\Delta x - (\widehat{x_I} - x_L)\Delta y_B}{\sqrt{\Delta x^2 + \Delta y_B^2}} \tag{8.12}$$

Substituting equations (8.9) and (8.10) into equation (8.11) gives

$$e_A = \frac{u(7\alpha_4 L_Y \Delta x - 3\alpha_3 L_X \Delta y_A)}{\sqrt{\Delta x^2 + \Delta y_A^2}} \tag{8.13}$$

From this equation it is possible to derive a bound on $|e_A|$ by taking advantage of the property that for real values c and s satisfying $c^2 + s^2 = 1$, the absolute value of $Ac+Bs$ is bounded by $\sqrt{A^2+B^2}$. Taking c and s to be $\Delta y_A/\sqrt{\Delta x^2 + \Delta y_A^2}$ and $\Delta x/\sqrt{\Delta x^2 + \Delta y_A^2}$, we may conclude that

$$|e_A| \le u\sqrt{9\alpha_3^2 L_X^2 + 49\alpha_4^2 L_Y^2} \le u\sqrt{9L_X^2 + 49L_Y^2} \le \sqrt{58}uL \tag{8.14}$$

where $L = \max(L_X, L_Y)$.

The algebra required to determine a bound for e_B is more complex. Making use of equations (8.6) to (8.10) and the definitions for exact terms in table 8.2, equation (8.12) can be converted to

$$e_B = \frac{u((5(1-\overline{\lambda})\alpha_1 + 5\overline{\lambda}\alpha_2 + 7\alpha_4)L_Y\Delta x - 3\alpha_3 L_X \Delta y_B)}{\sqrt{\Delta x^2 + \Delta y_B^2}} \tag{8.15}$$

This can be re-expressed as

$$e_B = \frac{u(12\alpha_5 L_Y \Delta x - 3\alpha_3 L_X \Delta y_B)}{\sqrt{\Delta x^2 + \Delta y_B^2}} \tag{8.16}$$

Therefore

$$|e_B| \le u\sqrt{9L_X^2 + 144L_Y^2} \le \sqrt{153}uL \tag{8.17}$$

Hence we can conclude, that given two segments lying fully in the box $[-L, L] \times [-L, L]$ that are determined by the code to intersect, the computed intersection point lies within distance $\sqrt{153}uL$ or $12.37uL$ of both segments ($7.37\times10^{-7}L$ for single precision and $1.37\times 10^{-15}L$ for double precision). Note that uL is the maximum spacing between adjacent floating-point numbers lying in the range $[-L, L]$ (which encompasses all eight coordinate values specifying the two segments). In consequence of the earlier analysis we can also say that for a topologically valid polygonal set lying fully in the box $[-L, L] \times [-L, L]$, the winding number computed for a point in relation to this set will be correct provided the point is no closer than $4uL$ (i.e. four times the maximum spacing between floating point numbers) to any edge (segment) that defines the boundary.

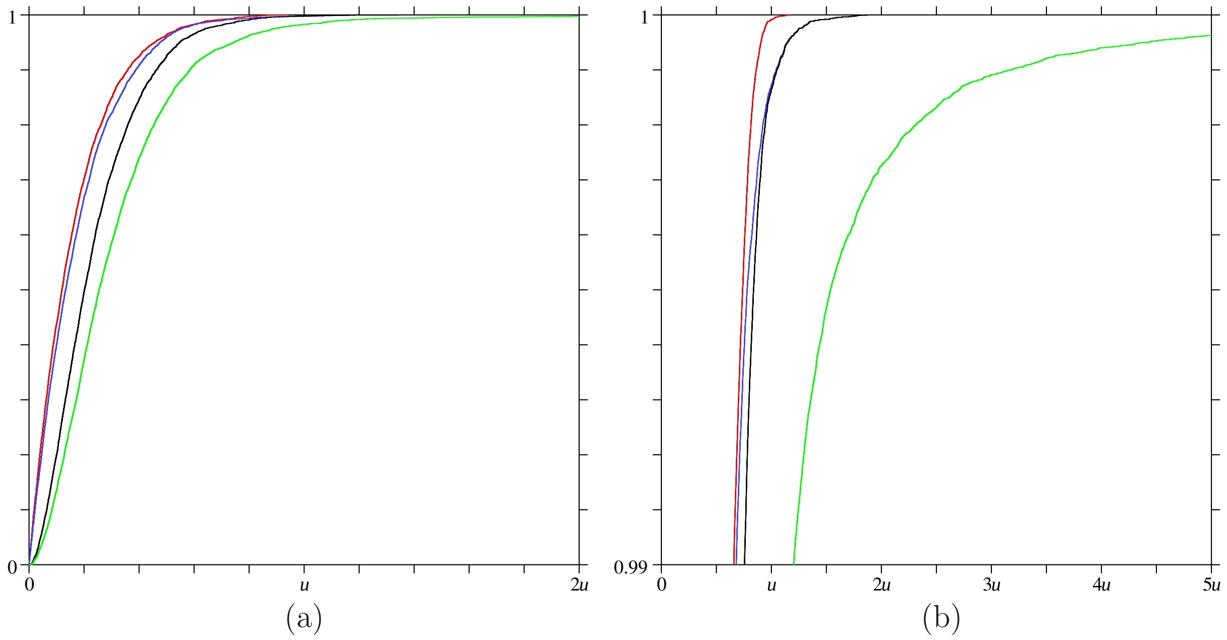


Figure 8.2: *(a) (and (b)): Cumulative distribution graph, based on 5,000 (resp. 200,000) randomly selected segment intersection calculations confined to* $[-1, 1] \times [-1, 1]$*, showing the proportion of cases for which a particular error measure is less than distance* d *over the range* $0 \leq d \leq 2u$ *(resp.* $0 \leq d \leq 5u$*); only the highest 1% of values are displayed in (b). The error measures displayed are the distances of the computed intersection point from the segment with lesser absolute slope (red), from the segment with greater absolute slope (blue), the maximum of these two distances (black), and the distance of the computed intersection point from the true intersection point (green). The 'true' intersection point and error measures were computed using* `long double` *floating-point arithmetic with a 64-bit mantissa, compared with 53 bits for the* `double` *arithmetic used for the test computation.*

8.3 Extent of error in practice

The error analysis of the previous section showed that the computed intersection point lies within distance $12.37uL$ of both segments. In practice, however, the error is much less than this.

The code for computing the intersection point between two segments, using double precision floating point arithmetic ($u = 2^{-53}$), was tested for a large number of randomly selected cases. For each computation, the two segments were created from two pairs of points selected at random from the box $[-1, 1] \times [-1, 1]$, so that L may be considered to equal 1; if the segments did not intersect, new random values were selected until they did. As with the tests for the simplification sweep algorithm, the code was written and compiled to ensure that the compiled program adhered to the IEEE 754 standard. Figure 8.2 plots the cumulative distribution of the error measures. The appropriate error measure for consideration is the larger of the two values giving the projected distance of the computed intersection point from either segment. In 99.87% of cases, the error computed this way was less than the maximum spacing between floating point numbers for the input coordinate values, $u = 2^{-53}$, and for 99.99% it was less than $1.5u$ (see figure 8.2(b)). For 10,000,000 randomly selected intersection point computations, this error measure was found never to exceed $2.57u$. As expected (see section 8.2, p.148), there was no indication of a bound on the distance of the computed intersection point from the true intersection point (exceeding $324u$ in one instance).

8.4 Accuracy of operations

Having looked at the numerical error in determining the intersection point between two segments, it is now appropriate to consider the accuracy of the sweep algorithm for simplification, and also the basic Boolean algorithm. In order to discuss the accuracy of these operations it is necessary to establish a means of defining their accuracy. It is not meaningful for any accuracy metric to be based on the geometry of the result, given that the result is a structure that may be geometrically invalid. Nor is it sufficient for the metric to consider only how the generated structure representing the result compares with the true geometric result. To see this is so, consider the intersection operation between two mutually exclusive shapes, for which the true result is the null set; it may be acceptable for the computed result to be something other than a representation of the null set, but only if the separation distance between the two input shape boundaries is within tolerance bounds.

In the arguments that follow concerning the accuracy of algorithms I make use of an accuracy metric I devised for inexact operations that takes advantage of the fact that for the algorithms being considered, each boundary component of the result originates from part or all of a boundary component from the input. According to this metric, an algorithm processing boundary representations in $\mathbb{R}^d$ is an *epsilon-tolerant algorithm* for some value $\varepsilon > 0$ if:

- every $(d-1)$-manifold boundary component in the result lies within distance ε of the boundary component in the input data from which it originates;
- the decision as to whether any part of any $(d-1)$-manifold boundary component in the input should be included in the result, and if so, its orientation, is either the correct decision (according to exact geometry), or is one that could be achieved by having boundary components (including itself, possibly) moved by a distance no greater than ε.

The second condition allows a boundary component to be considered to be on either side of another boundary component when it is less than distance ε away.

I confine discussion here to the 2D basic Boolean algorithm and sweep-line simplification algorithm. These operations rely both on the intersection calculation between two segments and the shadow relationship between a point and a segment. I assume for the sake of argument that a computed intersection point between two segments always lies within distance α of both segments, and that the computed shadow relationship between a point and segment is always correct when the distance between the two exceeds α. The value $\alpha = \sqrt{153}uL$ has already been determined as a suitable bound for a floating-point implementation. It is useful to represent the error bound symbolically without reference to this bound, so that the arguments presented here still apply if a smaller bound is determined, or alternatively, if a different sequence of arithmetic operations is used, or a different form of approximate arithmetic, or both.

We can say the basic Boolean algorithm in 2D is epsilon-tolerant for $\varepsilon = \alpha$. Every intersection vertex incorporated in the result arises from having determined that two original boundary edges intersect; hence it can be no more than distance α from either edge. It follows that the retained part of an edge in the result is at most distance α from the location of the original edge. The decision as to whether any part of an edge from one input object is to be retained in the result will be correct if it lies at least distance α from any edge from the other input object.

Consider now the simplification algorithm. For this algorithm it is necessary to compute the intersections successively, so that each intersection point calculation is based on the arrangement of edges after the previous intersection point calculation. Let us first assume, for the sake of argument, that there are no edges that need to be adjusted in consequence of inconsistencies in the logical and physical ordering of sweep-line edges. For both the simplification algorithm and the boundary self-intersection algorithm (which is the first stage of the simplification algorithm), every edge handled during the process can be designated an *adjustment count number* k as follows: For any edge from the original structure, $k = 0$; for any edge split due to a computed intersection, or due to a vertex being found to lie on the edge interior, the value of k for each of the two edges created is 1 greater than the value of k assigned to the original edge. Since each edge split may lead to the new edges lying up to a distance α away from the old edge, it follows that any edge handled in the process is at most $k\alpha$ away from the edge from which it originated. If for the final structure we designate $k_{\max}$ to be the maximum value of k for all the edges in the final structure, then we can say that no point on any edge will have migrated by a distance more than $k_{\max}\alpha$.

The simplified shape is determined from the computed boundary self-intersection. The decision as to whether to include an edge in the simplified shape, and with which orientation, will be correct for the particular configuration after the boundary self-intersection process if every vertex is at least distance α from every edge for which it is not a limit point. Because of the migration of the boundary, the decision will be correct for the *original* data if components are at least distance $(k_{\max} + 1)\alpha$ apart. Hence, the algorithm is epsilon-tolerant for $\varepsilon = (k_{\max} + 1)\alpha$.

Now let us consider the consequences of having to adjust edges, so that rules can be established for determining suitable values of $k_{\max}$ when this happens. Recall that when an event point is processed, certain of the sweep-line edges need to be split at the event point. A sweep-line edge deemed to pass through the event point is always split at that point; as just explained, the argument used for intersected edges applies in this situation. In general, a sweep-line edge deemed to pass over the event point is not split at the event point, except if it is succeeded in the sweep-line structure by an edge deemed not to pass over; similarly, an edge deemed to pass under the point must be split if it is preceded by an edge deemed not to pass under. There are two reasons why this situation may arise: One is because of an inconsistent set of relationships between entities that cannot exist under Euclidean geometry; the other is because of change to the geometry due to inaccurate determination of the intersection points.

Section 6.3.1, p.94, discussed the possibility of inconsistent ordering of entities. We may suppose that a vertex found to be on the wrong side of an edge will be at most distance α from the edge. Therefore, the splitting of the edge at that vertex will lead to a movement in the edge data of at most α. This can be handled by assuming the value of k for the new edges to be 1 greater than the k value assigned to the original edge.

The situation regarding the movement of edges due to splitting needs more consideration. It is a feature of the algorithm in its exact form that after the processing of an event point, no sweep-line edge fully intersects either neighbouring edge in the sweep-line structure (though they may touch at limit points, and non-neighbouring sweep-line edges may intersect fully). This is assured because whenever edges are inserted into or removed from the sweep-line structure, the algorithm tests for intersections between pairs of sweep-line edges that have just become neighbours and splits them at the intersection point if they do. The approximate form of the algorithm also takes that action. However, the splitting of an edge due to it intersecting one neighbouring edge on the sweep-line structure may lead it to cross over the other neighbouring edge, or move it further away from that edge on what is logically the wrong side.[2] A series of edge splits due to intersections, or alternatively, due to the adjustment process, can have a cumulative effect on movement of the edge, as shown in figure 8.3.

[2]The new sweep-line edges created by splitting at an intersection point are not tested for intersection because this makes the process iterative, with the risk of non-termination, or very slow termination and severe data degradation—see section 7.8.

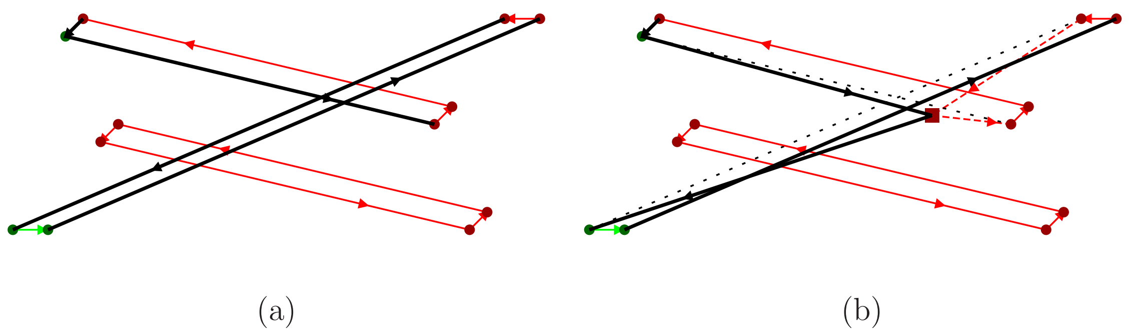


Figure 8.3: *An example case, before (a) and after (b), in which an inaccurately positioned intersection point between two neighbouring sweep-line edges leads to one of the truncated edges intersecting its other neighbour in the sweep-line structure. In this example, the retained edge will undergo further splitting due to a second intersection, so the second intersection point may migrate even further (up to 2α) on the wrong side of the neighbouring sweep-line edge. When each intersection point is processed as event point, this edge will be split at that point.*

The system of assigning adjustment count values needs to be extended to take into account the possibility of an edge being split at a vertex lying some distance away from an original edge on which it is supposed to lie. One option would be to assign an adjustment count value also to vertices. On creation, an intersection vertex would inherit the larger count value associated with the new edges attached to it. When an edge is split at a vertex which has a higher adjustment count value than the edge concerned, the vertex adjustment count value must be assigned to both edges formed by the split.

We thus have the means of computing for any given case (in parallel with the operation itself) an upper bound on ε such that we can say that the operation is epsilon-tolerant for that case. Unfortunately, we do not have a bound on $k_{\max}$, and hence ε, that can be applied to an infinite group of cases. We can say that if the exact form of the algorithm is executed, then $k_{\max}$ is bounded by n, the number of edges in the input. This is not necessarily useful to us, since the exact algorithm never has to adjust edges, but we might consider it gives a rough indication.

The bound on ε of $(k_{\max}+1)\alpha$ translates to $12.37(k_{\max}+1)uL$, or $1.37\times10^{-15}(k_{\max}+1)L$ for double precision arithmetic. For the highly atypical star-shaped polygon cases tests discussed in section 7.7, p.129, which were chosen as suitably destructive tests, the estimated bound on ε far exceeds the length scale of the initial data structure. For the cases displayed in figures 7.13(b)-(d), p.134 ($uL = \frac{1}{8}$ and $k_{\max}$ expected to be 8), and in figure 7.14, p.135 ($uL = \frac{1}{32}$ and $k_{\max}$ expected to be 60), the bound derived for ε is 13.92 and 23.58 respectively, compared with a length scale of 2. The pertinent question to consider is whether the bound on error is acceptable for typical problems in reality. Let us suppose that for a particular set of problems an edge is not intersected or adjusted more than 1,000 times. This gives a relative error bound close to 10^{-12}; for 1,000,000 intersections and adjustments the relative error bound is 10^{-9}. These are well within the accuracy bounds for many real-life problems.

8.5 Other arithmetic issues

We consider here the remaining issues related to the use of IEEE rounded arithmetic.

Division by zero is not an issue in the basic Boolean algorithm or in the simplification process. Division arises only in the determination of the two values known as λ in each of the two stages. For both stages, and for both problems, there is a guarantee that the denominator is non-zero because of the checks made either on the denominator itself, or on the terms subtracted from each other to evaluate the denominator. Note, incidentally, that since IEEE floating-point arithmetic is a standard form floating-point system that includes sub-normal numbers, the subtraction of one value from another (distinct) value will never lead to a result of value zero.

Overflow problems will never occur in any of the division operations concerned, because the numerator will never be greater than the denominator. Overflow will never occur in any of the multiplication operations either, because one of the terms concerned is guaranteed to be less than 1. There is a theoretical possibility of overflow due to subtraction; however, this can only arise from having coordinate values of magnitude greater than $\frac{1}{4}$ of the largest representable floating-point number. The largest representable number is 2^{128} (3.40×10^{38}) for single-precision, or 2^{1024} (1.80×10^{308}) for double-precision. Most problems to be solved do not require coordinate values approaching this magnitude.

The method of calculating the intersection point guarantees that the intersection point as computed lies within in the intersection of the axis-aligned bounding boxes enclosing each of the two segments, so that

$$\max(\min_A(x), \min_B(x)) \leq x_I \leq \min(\max_A(x), \max_B(x)) \tag{8.18}$$

$$\max(\min_A(y), \min_B(y)) \leq y_I \leq \min(\max_A(y), \max_B(y)) \tag{8.19}$$

8.6 Summary

In this chapter I first described the established principles of floating-point arithmetic and of error analysis for numerical computations, before going on to apply error analysis to my method for determining the intersection point between two segments on the plane. I showed that given a topologically valid polygon, the algorithm will correctly classify the enclosure (or winding number) of a point with respect to the polygon provided its distance from any edge is no closer than four times the maximum separation distance between adjacent floating-point numbers within the range of the relevant coordinate values. Furthermore, when two segments are determined to intersect, the computed point of intersection lies within a distance of either segment that is less than 13 times the maximum separation distance. However, tests show that this distance very rarely exceeds two times the separation distance. I concluded with a discussion on other issues relating to a floating-point implementation of segment intersection algorithm: in particular, zero divide and overflow.

Chapter 9

Conclusion and future work

9.1 Conclusion

In this dissertation I have discussed geometric algorithms in the context of robustness. In particular, I have presented two algorithms I devised that operate on piecewise linear (polyhedral/polygonal) shapes with boundary representations: the first is the basic Boolean algorithm for the Boolean operation in 3D (chapter 4); the second is a variant of the Bentley-Ottmann sweep-line algorithm for the simplification process in 2D (chapters 6 and 7). Both algorithms are topologically robust, meaning that the result is guaranteed to have valid topology provided all input also has valid topology. For both algorithms it is assumed that some form of inexact arithmetic, such as floating-point, is used.

Consideration of robustness has been crucial in this dissertation. Some authors use the word 'robust' loosely, in a relative sense, to indicate that an algorithm is likely to behave problematically less often than other algorithms. However, it is not sufficient simply to reduce the failure rate, when the aim is to produce geometric software that is consistently reliable, irrespective of the nature and scale of the application. Rather, it is necessary to show that an algorithm is robust in some absolute sense, so that it can be stated what exactly is guaranteed, and in what circumstances. For that reason, it was important to consider the requirements for a robust algorithm (chapter 3), and how they influenced the design of a particular algorithm. The general requirements I identified for a robust algorithm are for the result to be both valid and accurate, and for the algorithm to be efficient in terms of time and memory usage (section 3.1, p.47). For a generated geometric shape representation to be considered fully valid, it must satisfy both topological (combinatorial) and geometric (arithmetic) constraints (4.2, p.55).

Full robustness was not achieved either for the Boolean operation or the simplification process. For each algorithm considered for the process of simplification in 2D it was always possible to devise cases for which the algorithm would fail according to the strict criteria for full robustness; either it would generate a structure in which an unconnected pair of edges intersected, and were computed to intersect, or else the algorithm would iterate in an excessive manner. Consequently, the algorithms described can only be classed as topologically robust. The possibility of a geometrically invalid result imposes the need for any downstream process (whether or not for the same type of operation) to be tolerant to invalid input geometry, so that a topologically valid result is guaranteed even in these circumstances. Both the Boolean algorithm and the simplification algorithm satisfy this requirement (4.9, p.75; 7.6.3, p.128).

The basic Boolean algorithm can produce marginal features such as gaps and slivers in which entities coincide or nearly coincide. This is particularly likely to arise within CAD applications, for which it is often intended for surfaces or edges to be coincident. When the distance between entities is of the same order as the precision of the computations, the generated structure is borderline between geometrically valid and invalid, and a representation may cross over from being valid to invalid due to arithmetic errors (introduction to chapter 6, p.87). Furthermore, for certain applications, particularly CAD, end-users expect the near coincidence of entities to be treated as exact coincidence, and that some adjustment should be made accordingly (2.2, p.25). The assumed precision of the input data may be larger than the precision of the computations internal to the basic Boolean algorithm. The data-smoothing post-process (chapter 5 and appendix A) was implemented in CAD software as a pragmatic means of resolving features that are marginal to within a specified length scale tolerance, irrespective of the reason for their presence. The smoothing process is iterative, and in practice it was necessary to force early termination of the process for the very rare cases when it would not terminate naturally in reasonable time. Since the smoothing process can fail to resolve geometrically invalid and marginal data, both in theory (introduction to chapter 6, p.88) and, in very rare cases, in practice (5.2, p.83), in strict terms the combined operation is still classed only as topologically robust. In practical terms, this drawback can be tolerated, given the extreme rarity of failures of this nature, and the fact that the downstream processes concerned were tolerant to geometric errors in the input data.

The process of simplification closely resembles the Boolean operation in the way it operates, but it is more powerful insofar that it can (assuming exact arithmetic) fix a geometrically invalid shape representation, making it geometrically valid. It was not possible to devise an algorithm with a formulaic approach resembling that used for the basic Boolean algorithm (6.3, p.93), so another approach was required. For simplification in 2D, I presented a variant of the Bentley-Ottmann sweep algorithm [BO79]. This variant algorithm was designed specifically to cope with situations not possible under exact arithmetic, and therefore not covered by the original algorithm (7.6, p.127).

A test program was developed in order to try out the sweep-line algorithm and other candidate algorithms for 2D simplification (7.7, p.129). It was possible to devise cases that resulted in a geometrically invalid result. Nevertheless, in contrast to other methods tried, this method satisfies other requirements for robustness, thereby making it topologically robust; furthermore, it should be noted that the cases that led to a geometrically invalid result were highly atypical, with both a large displacement and a high degree of complexity.

I carried out error analysis for computing the intersection between two segments when floating point arithmetic is used, as required by the sweep-line algorithm for simplification (and also the 2D basic Boolean algorithm) (8.2, p.145, and appendix B). From that, I was able to deduce a bound on error for the simplification algorithm, based on a count of the number of edge splits affecting any part of the computed boundary (8.4, p.152).

9.2 Discussion and future work

Finally, I discuss future work required in order to resolve robustness problems for the Boolean operation and for data smoothing.

9.2.1 Achieving geometric robustness for 2D simplification

The ideal achievement would be to devise an algorithm guaranteed to generate output that is geometrically valid (as well as topologically valid). Often, the downstream processes for an operation are externally supplied, and there is no reason to suppose that these processes will be tolerant to geometrically invalid input. A geometrically robust algorithm for simplification in particular would be beneficial, since it could be relied on always to resolve invalid geometry; hence, any process that is not geometrically robust could be made geometrically robust by applying simplification as a post-process.

I experimented with a number of variants of the modified Bentley-Ottmann algorithm for 2D simplification. However, for each variant algorithm it was always possible to devise cases (usually highly atypical ones), that led either to geometric errors or a more serious form of failure. A major part of the problem in devising a suitable algorithm is the need to keep within the constraints of efficiency and accuracy. An approach that involves iteration is unlikely to be satisfactory because of the possibility of the algorithm failing to terminate, or to do so in polynomial time, for certain cases.

While I have not found an algorithm that is fully robust, that does not mean that one does not exist. I continue to seek a solution (see below). However, an alternative line for future research is to look for a proof that a fully robust algorithm is impossible.

I am currently investigating the feasibility of an approach in which an edge is split at a vertex whenever a modification causes the edge to cross (or appear to cross) the vertex concerned. For this approach, it is necessary to maintain a record of the vertices lying respectively above and below each edge. An experimental implementation I tried that maintained such data only for sweep-line edges was not satisfactory, because of the loss of crucial data when an edge is split. An algorithm that maintains *all* vertex-edge relationship data (requiring a re-writing the `PolySet2` class) would appear more promising, and is worthy of investigation. This approach is closely related to the process known as vertical decomposition, as applied to a set of segments in the plane [BY98, pp.40-42].[1]

An alternative approach to consider is one in which a change of topology, such as the splitting of an edge, does not lead to a change in geometry. An integer-based linear geometry devised by Luby for graphics applications [Lub88] satisfies this condition. By the rules of this geometry, the segment $\mathbf{ab}$ connecting points $\mathbf{a}$ and $\mathbf{b}$ on an integer grid is a monotonic sequence of points on the grid from $\mathbf{a}$ to $\mathbf{b}$ in which each point is succeeded by an immediate horizontal or vertical neighbour; a zero-length segment $\mathbf{aa} = \{\mathbf{a}\}$. Any segment $\mathbf{ab}$ has the property that for any point $\mathbf{p} \in \mathbf{ab}$, $\mathbf{ap} \cup \mathbf{pb} = \mathbf{ab}$. Also, if any two segments cross each other, the intersection is itself a segment according to this geometry.

There is no known equivalent of Luby geometry in 3D. Certain obvious analogues investigated by Sabin did not carry forward into 3D [Sab99], but as he notes, this does not necessarily imply that no such analogue exists.

9.2.2 Complicating issues for 3D operations

For both the Boolean operation and the simplification process on 3D polyhedral shape representations there are two options (at least) as to which type of boundary mesh to use: the triangle mesh and the general polygon mesh described in section 4.2, p.58. As with the 2D polygon representation, the geometry is assumed to be defined by vertex position data. The triangle mesh precisely defines an oriented surface, possibly with self-intersections and overlaps. In contrast, the polygon mesh is over-defined, and in general, the vertices for a polygonal facet will routinely breach the planar facet constraint for geometric validity (4.2, p.58). If the vertices of a facet are not coplanar, the geometry of that facet is not well-defined. This complicates any analysis of the algorithm concerned. It also complicates, at the very least, the design of any algorithm for polygon mesh if there is a requirement for it to be classed as geometrically (as well as topologically) robust. The

[1]Vertical decomposition determines the structure whereby each segment limit point, and likewise, each intersection point between two segments, is linked both to the closest segment directly above and the closest segment directly below.

planar facet constraint for geometric validity would have to be relaxed so that it would only have to be adhered to within tolerance, thereby allowing ‘slightly’ non-planar facets.

Despite this difficulty concerning the polygon mesh representation, it was possible to implement the basic Boolean algorithm using a polygon mesh. Topological robustness of the algorithm is assured, as was proved by the theorems of section 4.9, p.75. However, if there are non-planar facets in the original data that are both nearly vertical and nearly coincident with each other, the basic Boolean algorithm can (depending on circumstances) generate quasi-random shard-like artifacts, the geometry of which is highly sensitive to input data. Although such features are undesirable in their own right, they will in normal circumstances be resolved by the data-smoothing post-process due to facet cancellation, if the process is applied.[2]

Let us now consider the simplification process. The question arises as to whether an algorithm for 3D polyhedral shape representations can be devised that resembles the sweep line algorithm that was presented for 2D polygonal shape representations. There are certain aspects of the 2D algorithm that we would expect to carry over to an algorithm for the 3D operation. Thus, a 3D simplification algorithm would be based on an algorithm for the arrangement of the (supposed) boundary edges. It would operate by means of a sweep plane, and maintain a data structure intended to represent the cross-section of the arrangement formed by the sweep plane. Just as the sweep-line structure for the 2D algorithm is, in effect, a 1D cellular structure, the sweep-plane structure for the 3D algorithm would need to represent a 2D cellular structure. That structure would be an embedded planar graph, with nodes, edges and faces representing respectively the edges, faces and (3D) cells of the arrangement in cross-section.

The strategy that worked for the 2D simplification algorithm was to modify the boundary model progressively each time two boundary components were determined to intersect, and to base all arithmetic computations on the structure in its current, rather than initial, state. It is reasonable to suppose that a similar strategy is required in 3D, but the exact details of such a strategy are not clear. One aspect of the 3D arrangement problem that sets it apart from the 2D arrangement problem, and also from the 3D basic Boolean algorithm, is that it needs to be able to determine the intersection point between three intersecting faces.

The feasibility of devising a 3D simplification algorithm that is topologically robust (at least) may depend on the type of boundary mesh used for shape representations. A

[2]The artifacts can occur because if small, arbitrary perturbations are applied to the vertices of a vertical facet that is geometrically valid, the polygon formed by projecting onto the (x, y) plane can be geometrically invalid. Although the data-smoothing process usually resolves such artifacts, the reported problem case arose in consequence of a series of basic Boolean algorithm operations involving shapes with near-vertical facets—see section 5.2, p.84.

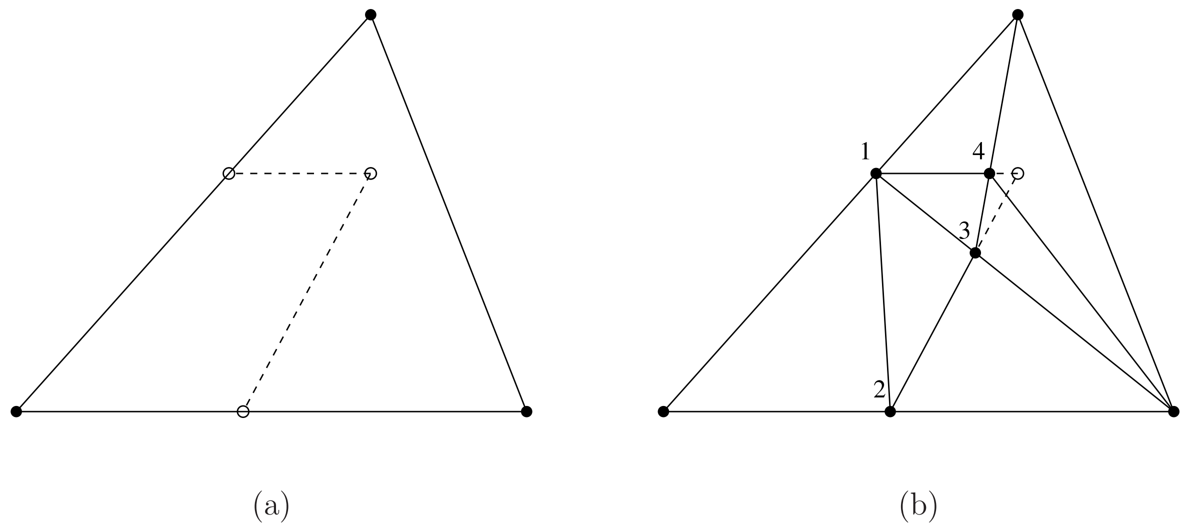


Figure 9.1: *Example showing the laddering effect that could occur in an algorithm for 3D simplification based on the use of triangle meshes. (a) shows a triangle that initially intersects two triangles that neighbour each other (not shown); the two intersection edges are shown dashed. Each time an edge is determined to intersect a triangle, the edge is split at the intersection point, and the triangles that neighbour the edge are also split; however, the edges formed by splitting a triangle sometimes also intersect a triangle. (b) shows the situation following the determination of four intersection points in the order marked.*

problem with using a triangle mesh is that every time an edge is found to intersect a triangle, extra edges must be inserted in order to split both the triangle concerned and the triangles that neighbour the edge. Depending on the algorithm, and on the order in which intersections are computed, the formation of new edges may continue indefinitely. Figure 9.1 shows how this may happen. If a topologically robust algorithm were to be devised for the polygon mesh representation, there is a possibility that undesirable artifacts may emerge when there are near-vertical, non-planar facets in the input, as was the case for the basic Boolean algorithm without data smoothing.

9.2.3 Addressing the need to resolve marginal geometry

We have observed that for CAD applications there is an implicit requirement to resolve any marginal geometry in shape representations constructed by Boolean operations. The principal reason for resolving marginal geometry is that it provides the means for the user to specify coincidence between two entities—for example, between two faces—by ensuring that the entities concerned lie within some given tolerance distance of each other (2.2, p.25); this allows for imprecision in the advance computations that position the objects concerned. A secondary reason is that it allows the resulting structure to be reprocessed to a certain extent (for example, by transformations) before running the risk of becoming geometrically invalid (introduction to chapter 6, p.87).

The operation to resolve marginal geometry is imprecisely specified and open-ended. Rules can be specified for determining whether or not a shape is marginal, but there is no hard rule as to what changes should be applied to resolve marginality.

A data-smoothing process is a practical approach to resolving marginal data; however, it is a challenge, at the very least, to make the process fully robust. Adjustments are required if a pair of entities not logically connected coincide to within the specified tolerance. The adjustments made are arbitrary, so the final result depends on the order in which the tests and adjustments are applied. The preferred technique is to merge the entities so that they become connected, splitting up one or both if necessary, as this satisfies the first requirement to treat near-coincident entities as coincident. Forcing entities apart can also resolve the marginal relationship, though this goes against the first requirement; however, when there is a chain of marginal relationships between pairs of entities, the marginal status between a specific pair of entities may well be resolved this way. Note that there are two additional requirements for data smoothing that ideally should apply: (1) that the process should not apply too excessive a distortion to boundary components; and (2) that the final structure formed by the process should not be geometrically invalid or marginal. It may be impractical to satisfy both these requirements for certain pathological cases.

The adjustments in the data-smoothing process I devised are made without reference to the geometric validity of the data; hence, there is no distinction between marginally valid and marginally invalid data. This conveniently avoids the problem of determining geometric validity, which is not well-defined for a polygon mesh. Furthermore, I argued that a data smoothing algorithm relying solely on tests for marginal data is unlikely to be fully robust, since adjusting one geometric entity to make it coincide with another has the potential to form a region that is geometrically invalid to a non-marginal degree (introduction to chapter 6, p.87).

Given the difficulty of achieving full robustness by the smoothing process, it is reasonable to consider other approaches. One possible approach would be by means of an algorithm resembling the sweep algorithm for the simplification problem, but which makes use of tolerance based checks. Ideally, one would like the algorithm to be designed to avoid the formation of geometrically invalid regions; this is clearly a challenge, given the difficulty of achieving this goal without the use of tolerances. The algorithm would have the advantage of being a single-pass algorithm, rather than an iterative algorithm with no guarantee of convergence. A concern, though, is the potential for boundary drift to cause a substantial distortion to the representation for pathological cases. Although this is potentially a problem also for the arrangement problem without tolerance based checks, as seen in figure 7.14, p.135, the distortion is unlikely to be substantial for realistic examples if the precision is sufficiently high; in general, double precision should be sufficient (8.4, p.155). However, for a tolerance-based simplification algorithm, the tolerance value selected may be sufficiently large to make data adjustment a significant possibility; indeed, it is essential to select a tolerance value likely to lead to data adjustment if there is a requirement to treat nearly coincident geometry as exactly coincident. Hence, the potential for large-scale distortion is much more likely than for the simplification algorithm described, which avoids tolerance-based checks.

Both data smoothing and tolerance-based simplification have the disadvantage that the adjustments made to the boundary are arbitrary. This problem could be avoided by an approach based on a mathematically well defined operation. One such operation that resolves marginal gaps and slivers—arguably the form of marginal data that is of most concern—is that of offsetting. It is possible to resolve gaps of magnitude up to ε (> 0) by offsetting a shape boundary by $\frac{1}{2}\varepsilon$; conversely, slivers can be resolved by offsetting by $-\frac{1}{2}\varepsilon$. In order to resolve both gaps and slivers of magnitude ε, and to ensure that the boundary for the most part returns to its original position, it is necessary to apply a sequence of offsets: either $\frac{1}{2}\varepsilon$, $-\varepsilon$, $\frac{1}{2}\varepsilon$, or $-\frac{1}{2}\varepsilon$, ε, $-\frac{1}{2}\varepsilon$. (The result generated by these sequences is not always the same.) An exactly defined offset operation would not be appropriate, as it would result in a shape in which the original edges and vertices are in effect rounded; this takes the shape outside the piecewise linear domain, and furthermore, it is unlikely to be what the user would want. An operation resembling offsetting that for the most part retains the angle of corners and (in 3D) of edges may be preferred.[3] In order to work, this approach would require a robust algorithm for determining the modified offset. It would be necessary, as a first step, to determine the cyclical ordering of edges about a vertex; hence, the algorithm would be an extension of the arrangement algorithm. A pseudo-offset operation might not be suitable for certain applications if there is a requirement, for example, for near-coincident edges (in 3D) to be merged. Thus, if two shapes are positioned not quite end to end, the result may have very thin faces (originating from the end faces) at a sharp angle to neighbouring faces.

9.2.4 Operations on curved shape representations

Given that many geometric modelling systems operate in a curved domain, an important question to consider is whether some form of robustness—full or topological—can be achieved for the Boolean operation or for simplification in any of the curved domains.

[3]For this modified form of offsetting, if in 2D a vertex corner of angle ϕ is offset by ε, the vertex would need to be shifted by distance $\varepsilon \operatorname{cosec}\frac{1}{2}\phi$. This distance tends to infinity as $\phi \to 0$, so vertices of angle below a certain threshold value would have to be filleted.

Unfortunately, there are some serious difficulties that make it appear unlikely that even a topologically robust algorithm can be devised, at least for general geometric modelling systems in 3D with a wide range of functionality.

The feasibility of devising a robust algorithm is influenced by the range of shapes permitted by the representation scheme. Thus, the issues for a 3D system dealing only with spherical surfaces (for molecular modelling, for example) differ from those for a more general system allowing a wider range of shapes. Hence, it is meaningful to refer to curved domains in the plural, with each domain identified by the family of surfaces (for the 3D problem) or curves (for 2D) that can be used to represent the geometry of each boundary component. For any system, the extent of the geometry permitted by the representation scheme must be sufficient for the functionality provided by the system. Consider the ACIS library [CL01]. The primitive objects specified by the user can have planar, cylindrical, conical, spherical and toroidal surfaces; in addition, the user can specify spline surfaces, which are in effect a patchwork of rational parametric surfaces. However, the operations of surface offsetting and of blending between two surfaces extend the family of permitted surfaces even further.

For the algorithms I presented in the linear domain, the geometry of the represented shape is assumed to be specified by the vertex positions (though as just discussed, it is *over*-specified for the 3D polygonal mesh). The inexact arithmetic of the computations can modify the geometry, but when that happens, the new geometry is accepted as definitive.

A similar strategy may be feasible for the simpler curved domains in 2D, provided it is possible to define the geometry of any edge by a set of control points. For example, if each edge lies on a rational parametric curve, it can be represented by Bézier control points with weights. When an edge is split at an intersection point, the control point data for the two new edges can be recomputed. For a similar technique to be feasible in 3D, a first requirement is that each face to be a patch represented by control points. However, a further requirement is that when a face is split due to intersections with other faces, it must be possible to break down each part into a collection of similarly defined patches (c.f. the triangulation process in the linear domain).

For the more complex curved domains in 3D required for modelling systems such as ACIS, the standard form of boundary representation model maintains a representation of the surface geometry for each face. See [CL01, Appendix B (pp.361-382)] for a summary of the ACIS data structure. The representation scheme also provides query functions giving the curve geometry for each edge and the position for each vertex. It is hard to see how a topologically robust Boolean algorithm could be devised using such a method of representation, given the complexity of the structure. The issue is further complicated if positioning transforms to shape representations are allowed, as is the case for ACIS. A particular concern is how any algorithm should manage inconsistencies in the data. If curve and vertex geometry data are maintained separately from surface geometry data, allowance must be made for data drift, with an edge or vertex not positioned precisely on a face it is presumed to border. Conversely, if curve and vertex geometry data are derived from surface geometry data, there are serious difficulties if the topology implied by the surface geometry differs from the topology stored in the representation.

9.3 Reflection

This research started with the hypothesis that it is possible to create a strictly robust Boolean algorithm that uses inexact arithmetic. A number of promising lines of attack were considered and investigated. In each case, the goal of strict geometric robustness was not attained. In the best algorithms, failure was only observed in extreme, pathological cases.

The easier goal of strict topological robustness has been achieved, and has been proven to have been achieved in all possible cases. Whether strict geometrical robustness can be achieved is still an open problem. It remains to future research to devise an algorithm to achieve this, or a proof that it cannot be achieved at all.

Appendix A

The data-smoothing process

The data-smoothing process simplifies the shape representation by a series of adjustments. Certain of these adjustments operate within a specified distance tolerance value, δ. The adjustment operations are as follows:

Vertex adjustment. Each vertex is checked against a collection of accepted vertices in turn. The collection is initially empty, and a vertex is added to the collection only if it does not lie within distance δ of any vertex already in the collection. If a vertex being checked is found to lie within distance δ of a vertex in the collection, it is discarded as a vertex within the structure, and any edge in the structure connected to that vertex as a start- or end-vertex is subsequently assumed to be connected instead to the vertex found nearby. The effect is to merge the two vertices into one, with a single identity, and located at the first vertex to be accepted.

Edge cracking (by vertex). If an edge has a vertex (not its start- or end-vertex) lying within distance δ from its interior, the edge is split into two at the vertex, *provided* both new edges are shorter than the original. (This extra condition is necessary to prevent infinite looping.)

Edge cracking (between edges). If two edges (involving four distinct vertices) lie within distance δ of each other at their interiors, a new vertex is created, located close to the respective points from each edge interior that lie closest to the other edge; if any existing vertex (including any of the four vertices involved) lies within distance δ of the allocated position, the new vertex is adjusted to this vertex. Both edges are split at the new vertex (or the vertex it adjusts to).

Half-edge cancellation. If two half-edges bordering the same facet oppose each other, with the start-vertex of each acting as the end-vertex of the other, they are both considered removed. If they belong to different edges, the half-edges that pair the removed half-edges are merged into one edge.

Facet cancellation. Two facets in contact and facing opposite directions cancel each other out (in part or full). This is performed by swapping boundary half-edges between the two facets. The details of this process are described later.

Zero-length edge removal. Any edge with identical start-vertex and end-vertex is removed.

Unreferenced vertex removal. Any vertex not acting as a start- or end-vertex to any edge is removed.

Empty facet removal. Any facet with no bounding half-edges is removed.

The processes described are applied iteratively. For the original design of the data-smoothing algorithm, the intention was for adjustments to be applied indefinitely until the conditions for adjustment no longer held. However, in the implemented version it was necessary to impose termination prematurely because of the failure to terminate in very rare circumstances.

The facet cancellation operation requires further explanation. The process is designed primarily for the situation when two facets lie on the same plane and face the opposite direction to each other. However, it also has to function when the facet planes do not quite coincide exactly. The plane equation for a facet is not held as primary data, but is derived from vertex position data and then cached locally.[1] The process is carried out after vertex adjustment and edge cracking.

For each candidate pair of facets, f_1 and f_2, a check on the box limits and an approximate comparison of plane normals is first made, so that detailed processing is not carried out unnecessarily for pairs of facets that are not expected to cancel. If the facets oppose each other, a list of potentially swappable half-edges is formed from both facets. These consist of:

- any half-edge bounding f_1 that is deemed to lie in the interior of f_2;
- any half-edge bounding f_2 that is deemed to lie in the interior of f_1;
- any half-edge bounding f_1 that counteracts one that bounds f_2, so that the start-vertex for f_1 is the end-vertex for f_2 and the end-vertex for f_1 is the start-vertex for f_2.

Half-edges are included in the list of potentially swappable half-edges only on condition that both start- and end-vertex lie within distance δ of the computed best-fit plane of the other facet. The calculation for determining whether a half-edge lies in the interior a facet

[1]For a facet that is not quite planar, the normal of the plane is taken to be the direction that maximises the apparent area of the facet, and the plane is positioned to pass through the mid-point of the vertices.

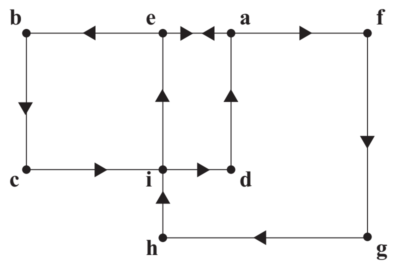


Figure A.1: *Case demonstrating partial cancellation between facets* **abcd** *(facing the viewer) and* **efgh** *(facing away), which are reduced to* **ebci** *and* **afghid** *respectively.*

is performed in 2D within the normal plane, using standard ray tracing techniques based on the mid-point of the half-edge. Out of the list of potentially swappable half-edges, a maximal subset is selected which form closed loops when considering the half-edges of f_1 and the *reverse* form of the half-edges of f_2.[2] The half-edges in the maximal list are then swapped, so that one that bordered f_1 is re-assigned to belong to f_2, and one from f_2 to f_1.

An example how facet cancellation works can be demonstrated by the case shown in figure A.1 where the original facets are the rectangles **abcd** and **efgh**:

- The original facet boundaries are defined by the sets of half-edges: $\partial f_1 = \{\mathbf{ab}, \mathbf{bc}, \mathbf{cd}, \mathbf{da}\}$ and $\partial f_2 = \{\mathbf{ef}, \mathbf{fg}, \mathbf{gh}, \mathbf{he}\}$.
- After edge cracking, $\partial f_1 = \{\mathbf{ae}, \mathbf{eb}, \mathbf{bc}, \mathbf{ci}, \mathbf{id}, \mathbf{da}\}$ and $\partial f_2 = \{\mathbf{ea}, \mathbf{af}, \mathbf{fg}, \mathbf{gh}, \mathbf{hi}, \mathbf{ie}\}$.
- The list of potentially swappable half-edges consists of **id**, **da** & **ae** from ∂f_1 and **ie** from ∂f_2.
- Since these four half-edges (with the half-edge from ∂f_2 reversed) form a complete loop, all these half-edges are re-assigned to the facet from which they did *not* originate, leading to $\partial f_1 = \{\mathbf{eb}, \mathbf{bc}, \mathbf{ci}, \mathbf{ie}\}$ and $\partial f_2 = \{\mathbf{ea}, \mathbf{af}, \mathbf{fg}, \mathbf{gh}, \mathbf{hi}, \mathbf{id}, \mathbf{da}, \mathbf{ae}\}$.
- Half-edge cancellation between **ea** and **ae** within ∂f_2 leads to $\partial f_2 = \{\mathbf{af}, \mathbf{fg}, \mathbf{gh}, \mathbf{hi}, \mathbf{id}, \mathbf{da}\}$.

In other words, the two facets end up as the polygons **ebci** and **afghid**.

[2]The algorithm is designed under the assumption that edges on the border will have been detected in advance by vertex adjustment and edge cracking operations, though that might not be the case when a facet is badly warped as a result of rounding errors or previous operations within the normalisation process. Generally, the list of swappable half-edges should form closed loops, but it is necessary to find the maximal subset in case they do not, since otherwise the operation would lead to a breaking of the topological facet boundary closure constraint.

Appendix B

Determination of error bounds

Tables 8.1 and 8.2 (section 8.2) listed error bounds for each of the expression values that need to be computed when determining the point of intersection between two segments on the plane. This appendix gives details as to how the error bounds are obtained.

Each computed value is the result of an arithmetic operation ($+$, $-$, $\times$ or $\div$) upon two values that have already been computed or given. If the intended operation in its exact form is $c = a \text{ op } b$, the computed evaluation is represented by $\widehat{c} = \mathit{fl}(\widehat{a} \text{ op } \widehat{b})$. In many cases it is possible to determine an error bound (relative or absolute) on $\widehat{c}$ by first determining an error bound on the exact expression $\widehat{a} \text{ op } \widehat{b}$ and then taking into account the effect of applying the rounding function $\mathit{fl}()$. For a particular computed value v, let L_v represent a known bound on the absolute value of v, and e_v and E_v represent respectively known relative and absolute error bounds on v. Assuming the error bounds are small compared to the values we can say:

- The absolute error for the exact evaluation of $\widehat{a} + \widehat{b}$ or $\widehat{a} - \widehat{b}$ is bounded by $E_a + E_b$.
- The relative error for the exact evaluation of $\widehat{ab}$ or $\widehat{a}/\widehat{b}$ is bounded by $e_a + e_b$.
- The absolute error for the exact evaluation of $\widehat{ab}$ is bounded by $|b|E_a + |a|E_b$, which in turn is bounded by $L_bE_a + L_aE_b$.

From the relative or absolute error bound of the exact evaluation it is possible to determine the equivalent error bound for the *rounded* value in accordance with equation (8.3) or (8.4). Hence, u is added to the relative error bound, and $\frac{1}{2}uL$ is added to the absolute error bound, where L is the smallest integer power of 2 known to bound the result.

The error bounds for the first stage, as listed in table 8.1, are determined as follows:

- $\Delta x_L = x_P - x_L$ — **exact evaluation**: no error; **rounded value**: relative error bound: u, absolute error bound: $\frac{1}{2}uL_X$

- $\Delta x = x_R - x_L$ — **exact evaluation**: no error; **rounded value**: relative error bound: u, absolute error bound: uL_X
- $\lambda = \Delta x_L / \Delta x$ — **exact evaluation**: relative error bound: $2u$, absolute error bound: u (derived from relative error bound); **rounded value**: relative error bound: $3u$, absolute error bound: $\frac{5}{4}u$
- $\Delta y = y_R - y_L$ — **exact evaluation**: no error; **rounded value**: relative error bound: u, absolute error bound: uL_Y
- $\Delta y_L = \lambda \Delta y$ — **exact evaluation**: relative error bound: $4u$, absolute error bound: $\frac{5}{4}u|\Delta y| + \lambda u L_Y \leq 3uL_Y$; **rounded value**: relative error bound: $5u$, absolute error bound: $\frac{7}{2}uL_Y$
- $y_S = y_L + \Delta y_L$ — **exact evaluation**: absolute error bound: $\frac{7}{2}uL_Y$; **rounded value**: absolute error bound: $4uL_Y$

The error bounds for the second stage, for the expression variables listed in table 8.2, up to the calculation of $\Delta\Delta y$, are determined as follows:

- $\Delta x = x_R - x_L$ — **exact evaluation**: no error; **rounded value**: relative error bound: u, absolute error bound: uL_X
- $\Delta y_L = y_{BL} - y_{AL}$ — **exact evaluation**: error bound: $4uL_Y$ (since at least one operand has no error); **rounded value**: absolute error bound: $5uL_Y$
- $\Delta y_R = y_{BR} - y_{AR}$ — **exact evaluation**: error bound: $4uL_Y$ (since at least one operand has no error); **rounded value**: absolute error bound: $5uL_Y$
- $\Delta y_A = y_{AR} - y_{AL}$ — **exact evaluation**: error bound: $8uL_Y$; **rounded value**: absolute error bound: $9uL_Y$
- $\Delta y_B = y_{BR} - y_{BL}$ — **exact evaluation**: error bound: $8uL_Y$; **rounded value**: absolute error bound: $9uL_Y$
- $\Delta\Delta y = \Delta y_L - \Delta y_R$ — **exact evaluation**: absolute error bound: $10uL_Y$; **rounded value**: absolute error bound: $12uL_Y$

As explained in section 8.2, we cannot progress by simply determining the bound on error between the computed and exact expression values. Instead, we determine the *adjusted* relative or absolute error bound:

- $\Delta\Delta y = \Delta y_L - \Delta y_R$ — **exact evaluation**: no adjusted error (by definition); **rounded value**: adjusted relative error bound: u, adjusted absolute error bound: $2uL_Y$

- $\lambda = \Delta y_L/\Delta\Delta y$ — **exact evaluation**: adjusted relative error bound: u, adjusted absolute error bound: $\frac{1}{2}u$ (derived from adjusted relative error bound); **rounded value**: adjusted relative error bound: $2u$, adjusted absolute error bound: $\frac{3}{4}u$

- $\Delta x_I = \lambda\Delta x$ — **exact evaluation**: adjusted relative error bound: $3u$, adjusted absolute error bound: $\frac{3}{4}u|\Delta x| + \overline{\lambda}uL_X \leq 2uL_X$; **rounded value**: adjusted relative error bound: $4u$, adjusted absolute error bound: $\frac{5}{2}uL_X$

- $\Delta y_{AI} = \lambda\Delta y_A$ — **exact evaluation**: adjusted absolute error bound: $\frac{3}{4}u|\Delta y_A| + 9\overline{\lambda}uL_Y \leq 6uL_Y$; **rounded value**: adjusted absolute error bound: $\frac{13}{2}uL_Y$

- $x_I = x_L + \Delta x_I$ — **exact evaluation**: adjusted absolute error bound: $\frac{5}{2}uL_X$; **rounded value**: adjusted absolute error bound: $3uL_X$

- $y_I = y_{AL} + \Delta y_{AI}$ — **exact evaluation**: adjusted absolute error bound: $\frac{21}{2}uL_Y$; **rounded value**: adjusted absolute error bound: $11uL_Y$

The analysis just presented considers each evaluation in isolation. This approach keeps the analysis simple, but it can in certain circumstances produce an unnecessarily large error bound because it fails to take into account the cancellation of errors. The adjusted absolute error bound for y_I is a case in point, for which I was able to derive a bound of $7uL_Y$, compared to the bound of $11uL_Y$ determined by the step-by-step analysis. This is shown in the equations below, in which each α_i denotes an unspecified value in the range $[-1, 1]$:

$$\widehat{y_{AL}} = y_{AL} + 4\alpha_1 u L_Y \tag{B.1}$$

$$\widehat{y_{AR}} = y_{AR} + 4\alpha_2 u L_Y \tag{B.2}$$

$$\begin{aligned}\widehat{\Delta y_A} &= \widehat{y_{AR}} - \widehat{y_{AL}} + \alpha_3 u L_Y \\ &= \Delta y_A + (4\alpha_2 - 4\alpha_1 + \alpha_3) u L_Y \end{aligned} \tag{B.3}$$

$$\widehat{\lambda} = \overline{\lambda} + \tfrac{3}{4}\alpha_4 u \tag{B.4}$$

$$\begin{aligned}\widehat{\Delta y_{AI}} &= \widehat{\lambda}\widehat{\Delta y_A} + \tfrac{1}{2}\alpha_5 u L_Y \\ &= \overline{\lambda}\widehat{\Delta y_A} + \tfrac{3}{4}\alpha_4 u \widehat{\Delta y_A} + \tfrac{1}{2}\alpha_5 u L_Y \\ &= \overline{\lambda}\Delta y_A + (4\overline{\lambda}\alpha_2 - 4\overline{\lambda}\alpha_1 + \overline{\lambda}\alpha_3 + \tfrac{1}{2}\alpha_5) u L_Y + \tfrac{3}{4}\alpha_4 u \widehat{\Delta y_A} \\ &= \overline{\lambda}\Delta y_A + (4\overline{\lambda}\alpha_2 - 4\overline{\lambda}\alpha_1 + \tfrac{5}{2}\alpha_6) u L_Y \end{aligned} \tag{B.5}$$

$$\begin{aligned}\widehat{y_I} &= \widehat{y_{AL}} + \widehat{\Delta y_{AI}} + \tfrac{1}{2}\alpha_7 u L_Y \\ &= y_{AL} + 4\alpha_1 u L_Y + \overline{\lambda}\Delta y_A + (4\overline{\lambda}\alpha_2 - 4\overline{\lambda}\alpha_1 + \tfrac{5}{2}\alpha_6) u L_Y + \tfrac{1}{2}\alpha_7 u L_Y \\ &= (y_{AL} + \overline{\lambda}\Delta y_A) + (4\overline{\lambda}\alpha_2 + 4(1-\overline{\lambda})\alpha_1 + \tfrac{5}{2}\alpha_6 + \tfrac{1}{2}\alpha_7) u L_Y \\ &= (y_{AL} + \overline{\lambda}\Delta y_A) + 7\alpha_8 u L_Y \end{aligned} \tag{B.6}$$

Bibliography

[AH74] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[AVE06] Vantage Plant Design Management System. http://www.aveva.com/media_centre/library/datasheets/vpd_pdms.pdf, 2006.

[Bau72] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[BE92] Marshall W. Bern and David Eppstein. Polynomial-size nonobtuse triangulation of polygons. *International Journal of Computational Geometry and Applications*, 2(3):241–255, 1992.

[BFS98] Christoph Burnikel, Stefan Funke, and Michael Seel. Exact geometric predicates using cascaded computation. In *In Proc. 14th Annu. ACM Sympos. Comput. Geom*, pages 175–183, 1998.

[BMS94] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. On degeneracy in geometric computations. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 16–23, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28(9):643–647, 1979.

[BS04] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 3.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic geometry.* Cambridge University Press, UK, 1998. Translated from the French version (Ediscience International) by Hervé Brönnimann.

[CGA08] CGAL, Computational Geometry Algorithms Library, 2008. http://www.cgal.org.

[CL01] J. Corney and T. Lim. *3D Modeling with ACIS*. Saxe-Coburg Publications, 2001.

[COR08] The CORE project homepage, 2008. http://www.cs.nyu.edu/exact.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*. Springer, 1997.

[EC91] Ioannis Emiris and John Canny. A general approach to removing degeneracies. In *SFCS '91: Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 405–413, Washington, DC, USA, 1991. IEEE Computer Society.

[EC92] Ioannis Emiris and John Canny. An efficient approach to removing geometric degeneracies. In *SCG '92: Proceedings of the eighth annual symposium on Computational geometry*, pages 74–82, New York, NY, USA, 1992. ACM.

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

[Far99] Rida T. Farouki. Closing the gap between cad model and downstream application.
http://www.siam.org/news/news.php?id=743, 1999.

[FCM87] J. Flaquer, A. Carbajal, and M. A. Mendez. Edge-edge relationships in geometric modelling. *Computer-Aided Design*, 19(5):237–244, 1987.

[FGK$^+$96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The cgal kernel: A basis for geometric computation. In *FCRC '96/WACG '96: Selected papers from the Workshop on Applied Computational Geometry, Towards Geometric Engineering*, pages 191–202, London, UK, 1996. Springer-Verlag.

[FHK$^+$07] E. Fogel, D. Halperin, L. Kettner, M. Teillaud, R. Wein, and N. Wolpert. Arrangements. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, pages 1–66. Springer-Verlag, Mathematics and Visualization, 2007.

[FKMS04] Stefan Funke, Christian Klein, Christian Mehlhorn, and Kurt Schmitt. Controlled perturbation for delaunay triangulations. In *In SODA 05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1047–1056. ACM, 2004.

[For89] Steven Fortune. Stable maintenance of point set triangulations in two dimensions. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 494–499, 1989.

[For95] S. Fortune. Polyhedral modelling with exact arithmetic. In *Proc. 3rd Symp. Solid Modeling*, pages 225–234. ACM Press, NY, 1995.

[For96] S.J. Fortune. Robustness issues in geometric algorithms. In *Applied computational geometry: towards geometric engineering*, pages 9–14. Springer, 1996.

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

[FT07] Efraim Fogel and Monique Teillaud. Generic programming and the cgal library. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, pages 313–320. Springer-Verlag, Mathematics and Visualization, 2007.

[FvDF96] J. D. Foley, A. van Dam, and S. K. Feiner. *Computer Graphics: Principles and Practice*. Reading, Mass. ; Wokingham : Addison-Wesley, 2nd edition, 1996.

[FvW96] Steven Fortune and Christopher J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph*, 15:223–248, 1996.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[GP96] Yvon Gardan and Estelle Perrin. An algorithm reducing 3D boolean operations to a 2D problem: concepts and results. *Computer-Aided Design*, 28(4):277–287, 1996.

[GSS89] Leonidas J. Guibas, David Salesin, and Jorge Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 208–217, New York, NY, USA, 1989. ACM.

[GSS90] Leonidas J. Guibas, David Salesin, and Jorge Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. In *SIGAL '90: Proceedings of the International Symposium on Algorithms*, pages 261–270, London, UK, 1990. Springer-Verlag.

[Gui96] Leonidas J. Guibas. Implementing geometric algorithms robustly. In *FCRC '96/WACG '96: Selected papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering*, pages 15–22, London, UK, 1996. Springer-Verlag.

[GY86] Daniel H. Greene and F. Frances Yao. Finite-resolution computational geometry. In *27th IEEE Symposium on Foundations of Computer Science*, pages 143–152, 1986.

[Hai89] Eric Haines. Essential ray tracing algorithms. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 33–77, London, UK, 1989. Academic Press Ltd.

[Hal97] Marc Halpern. Industrial requirements and practices in finite element meshing: A survey of trends. In *6th International Meshing Roundtable, Sandia National Laboratories*, pages 399–411, 1997.

[Hal04] Dan Halperin. Arrangements. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. CRC Press LLC, Boca Raton, FL, 2004.

[HHK88] Christoph M. Hoffmann, John E. Hopcroft, and Michael S. Karasick. Towards implementing robust geometric computations. In *Symposium on Computational Geometry*, pages 106–117, 1988.

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

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[HK89] John E. Hopcroft and Peter J. Kahn. A paradigm for robust geometric algorithms. Technical report, Cornell University, Ithaca, NY, USA, 1989.

[HKM07] Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn. Boolean operations on 3d selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. *Computational Geometry: Theory and Applications*, 38(1-2):64–99, September 2007.

[Hob99] John D. Hobby. Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199–214, 1999.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers, Inc., 1989.

[Hof01] C.M. Hoffmann. Robustness in geometric computations. *Journal of Computing and Information Science in Engineering*, 1:143–156, 2001.

[HS98] Dan Halperin and Christian R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comput. Geom. Theory Appl.*, 10(4):273–287, 1998.

[IEE85] IEEE Standard 754-1985 for binary floating-point arithmetic. Technical report, The Institute of Electrical and Electronic Engineers, Inc., 1985.

[Kah97] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. http://www.cs.berkeley.edu/˜wkahan/ieee754status/ieee754.ps, 1997.

[Kal82] Yehuda E. Kalay. Determining the spatial containment of a point in general polyhedra. *Computer Graphics and Image Processing*, 19(4):303–334, August 1982.

[KBF05] David J. Kasik, William Buxton, and David R. Ferguson. Ten CAD challenges. *IEEE Computer Graphics and Applications*, 25(2):81–92, 2005.

[KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 351–359, New York, NY, USA, 1999. ACM.

[KMP$^+$08] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry: Theory and Applications*, 40(1):61–78, 2008.

[KN04] Lutz Kettner and Stefan Näher. Two computational geometry libraries. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 65, pages 1435–1463. CRC Press LLC, Boca Raton, FL, 2004.

[Knu92] Donald E. Knuth. *Axioms and Hulls*, volume 606 of *Lecture Notes in Computer Science*. Springer, 1992.

[LED08] LEDA. http://www.algorithmic-solutions.com, 2008.

[LPY04] Chen Li, Sylvain Pion, and Chee Yap. Recent progress in exact geometric computation. *J. of Logic and Algebraic Programming*, 64(1):85–111, 2004. Special issue on "Practical Development of Exact Real Number Computation".

[LTH86] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes. Constructive solid geometry for polyhedral objects. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 161–170, August 1986.

[Lub88] Michael G. Luby. Grid geometries which preserve properties of Euclidean geometry: A study of graphics line drawing algorithms. In R.A.Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 397–432. NATO ASI series F vol 40, Springer Verlag, 1988.

[Mid94] A.E. Middleditch. "The Bug" and Beyond. In *CSG 94—Set-theoretic Solid Modelling Techniques and Applications*, pages 1–16. Information Geometers, 1994.

[Mil88] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, pages 377–401, 1988.

[Mil00] Victor J. Milenkovic. Shortest path geometric rounding. *Algorithmica*, 27(1):57–86, 2000.

[MN99] Kurt Mehlhorn and Stefan Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[MOS06] Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2006.

[Mou04] David M. Mount. Geometric intersection. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 38, pages 857–876. CRC Press LLC, Boca Raton, FL, 2004.

[MY04] Kurt Mehlhorn and Chee Yap. Robust geometric computation (tentative title).
http://cs.nyu.edu/˜yap/book/egc, 2004.

[NIS02] The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, National Institute of Standards and Technology, May 2002.

[PM02] N.M. Patrikalakis and T. Maekawa. *Shape Interrogation for Computer Aided Design and Manufacturing.* Springer, 2002.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Springer, 1985.

[PS86] Adobe Press and Adobe Syst. *PostScript Language Tutorial and Cookbook.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[PS97] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications, 1997.

[QS06] Jianchang Qi and Vadim Shapiro. Epsilon-topological formulation of tolerant solid modeling. *Computer-Aided Design*, 38(4):367–377, 2006.

[Req77] A.A.G. Requicha. Mathematical models of rigid solids. Tech. Report PAP Tech. Memo 28, Univ. of Rochester, 1977.

[Req80] A.A.G. Requicha. Representations for rigid solids: Theory, methods, and systems. *Computing Surveys*, 12(4):437–464, 1980.

[RH99] Sigal Raab and Dan Halperin. Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. In *In Proc. 15th Annu. ACM Sympos. Comput. Geom*, pages 163–172, 1999.

[Sab99] M.A. Sabin. Explorations in 3D integer-based linear geometry. Technical report, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, 1999.

[Sch98] Stefan Schirra. Robustness and precision issues in geometric computation. Research Report MPI-I-98-1-004, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1998.

[Sch99] Stefan Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V., 1999.

[SD06] J. M. Smith and N. A. Dodgson. A topologically robust Boolean algorithm using approximate arithmetic. In *22nd European Workshop on Computational Geometry (EUROCG 2006)*, 2006.

[SD07] J. M. Smith and N. A. Dodgson. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. *Comput. Aided Des.*, 39(2):149–163, 2007.

[Sev98] Charles Severance. IEEE 754: An interview with William Kahan. *Computer*, 31(3):114–115, 1998.

[She99] Jonathan Richard Shewchuk. Lecture notes on geometric robustness. Technical report, Dept. of Electrical Eng. and Computer Science, Univ of California at Berkeley, 1999.

[SI89] Kokichi Sugihara and Masao Iri. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. In *Proc. 1st Canadian Conf. Computational Geometry CCCG*, Montréal, Canada, August 1989.

[SIII00] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation—an approach to robust geometric algorithms. *Algorithmica*, 27:5–20, 2000.

[SS85] M.G. Segal and C.H. Sequin. Consistent calculations for solids modelling. In *Proc. 1st ACM Sympos. Comput. Geom.*, pages 29–38, 1985.

[SSZ$^+$04] Xiaowen Song, Thomas W. Sederberg, Jianmin Zheng, Rida T. Farouki, and Joel Hass. Linear perturbation methods for topologically consistent representations of free-form surface intersections, 2004.

[Sug89] K. Sugihara. On finite-precision representations of geometric objects. *J. Comput. Syst. Sci.*, 39(2):236–247, 1989.

[Sug08] Kokichi Sugihara. Toward superrobust geometric computation. In *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 11–12, New York, NY, USA, 2008. ACM.

[Sun06] Dan Sunday. The Intersections for a Set of 2D Segments, and Testing Simple Polygons. http://www.geometryalgorithms.com/Archive/algorithm_0108/algorithm_0108.htm, 2006.

[VR93] H.B. Voelcker and A.A.G. Requicha. Research in solid modeling at the University of Rochester. In Les A. Piegl, editor, *Fundamental Developments* *of Computer-Aided Geometric Modeling*, pages 203–254. Academic Press, 1993.

[WDF$^+$98] Kevin Weiler, Tom Duff, Steve Fortune, Chris Hoffman, and Tom Peters. Is robust geometry possible? (panel). In *SIGGRAPH '98: ACM SIGGRAPH 98 Conference abstracts and applications*, pages 217–219, New York, NY, USA, 1998. ACM.

[WFZ07] Ron Wein, Efi Fogel, and Baruch Zukerman. 2D Intersection of Curves. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.3 edition, 2007.

[Yap90] Chee-Keng Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40(1):2–18, 1990.

[Yap97] Chee-Keng Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1-2):3–23, 1997.

[Yap04] Chee K. Yap. Robust geometric computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. CRC Press LLC, Boca Raton, FL, 2004.