Available online at www.sciencedirect.com
ScienceDirect
Computer-Aided Design xx (xxxx) xxx–xxx
COMPUTER-AIDED DESIGN
www.elsevier.com/locate/cad

A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic

J.M. Smith*, N.A. Dodgson

Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, United Kingdom

Received 21 June 2006; accepted 14 November 2006

Abstract

We present a topologically robust algorithm for Boolean operations on polyhedral boundary models. The algorithm can be proved always to generate a result with valid connectivity if the input shape representations have valid connectivity, irrespective of the type of arithmetic used or the extent of numerical errors in the computations or input data. The main part of the algorithm is based on a series of interdependent operations. The relationship between these operations ensures a consistency in the intermediate results that guarantees correct connectivity in the final result. Either a triangle mesh or polygon mesh can be used. Although the basic algorithm may generate geometric artifacts, principally gaps and slivers, a data smoothing post-process can be applied to the result to remove such artifacts, thereby making the combined process a practical and reliable way of performing Boolean operations.

Keywords: Boolean operations; Boundary mesh representations; Robustness

1. Introduction

Problems of robustness are a major cause for concern in the implementation of algorithms relating to geometry [1–4]. Most geometric algorithms are a mix of numerical and combinatorial computations, and the approximate nature of the former often leads to inconsistencies that hinder the ability to construct a satisfactory result [2]. For operations such as the Boolean operation between shapes in boundary representation form, in which geometric calculations drive the construction of the result, the inconsistencies arising from numerical error can lead to connectivity faults, such as breaks in the supposed boundary. The inaccuracies in the calculations can also create geometric errors, often in the form of boundary self-intersections.

Within computer-aided design systems it is often necessary to perform Boolean operations (and indeed other operations) on shapes in boundary representation form. For example, the Boolean operation is required when converting a shape specified in CSG (constructive solid geometry) format into boundary format for subsequent operations where the boundary representation is required, such as rendering, hidden line removal, clash detection and the determination of mass properties. Often large numbers of operations are performed without user intervention on data already prepared, so the failure of just one operation can be catastrophic. The response traditionally taken by commercial system designers to such problems has been to invest time and effort tuning the system to avoid catastrophic errors in applications that are considered typical [2].

A number of approaches to the problem have been advocated in academia. Hoffmann [3] categorises these into three strategies, based on exact arithmetic, symbolic reasoning and interval computation. Many have come to regard the use of exact arithmetic as the most promising approach because it avoids the problems described. Most advances are in the piecewise-linear domain for which rational arithmetic can be used [5–8]; for advances in the curved domain see [9, 10]. Performance can be badly affected if exact arithmetic is used exclusively, so filtered exact arithmetic is usually preferred, whereby exact arithmetic is used to determine a course of action only when approximate arithmetic fails to give a definite answer [5,11]. A particular advantage of the exact arithmetic approach is that it can be used for a wide range of problems. Indeed, certain libraries such as LEDA [12] and CGAL [8] support filtered exact arithmetic, which eases the implementation of exact arithmetic algorithms.

* Corresponding author. Tel.: +44 1223 763679; fax: +44 1223 334678.
E-mail address: jms222@cl.cam.ac.uk (J.M. Smith).

0010-4485/$ - see front matter © 2006 Published by Elsevier Ltd
doi:10.1016/j.physd.2007.01.005

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

Of particular relevance to this paper are the methods based on symbolic reasoning. These are designed to guarantee that the computed result has a valid topology or connectivity irrespective of the extent of any numerical errors in the computations. Sugihara [13] uses such an approach, which he classes as topology-oriented, for Voronoi and Delaunay calculations, the convex hull problem, and also for the intersection of convex polyhedra.

This paper presents a previously unpublished algorithm for Boolean operations between piecewise-linear boundary representation shapes. The method described can also be used for the related problem of determining the intersection between two piecewise-linear surfaces (or curves in 2D). We present the algorithm principally in the 3D (polyhedral) domain, though it can also operate in the 2D (polygonal) domain, and it seems reasonable to suppose it could be extended to work in higher finite-dimensional domains.

The main part of the algorithm, which we call 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. The operations at the heart of the basic Boolean operation 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, [14], 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 [15], and Gardan and Perrin for Boolean operations [16], and our 2D point-in-polygon-region test is essentially identical to the winding number method described by Haines [17]. Our method is an advance because of the guarantee of correct connectivity in the result of the Boolean operation. We demonstrate a consistent pattern of dependency between the operations, from the lower-level relationship tests to the higher-level operations that construct the result, and provide proofs 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. For the triangle mesh variant of the algorithm it is necessary to break up non-triangular facets in the result into triangles if the output is required to be in the same format, for example for subsequent Boolean operations. The polygon mesh variant of the algorithm is potentially more efficient, but there are theoretical concerns, in particular that it is ambiguous as to where the surface boundary lies when the vertices of a polygon are not exactly coplanar.

The basic algorithm described does not necessarily avoid problems of geometrical correctness. The result can have gaps and slivers that make the result borderline between being valid and invalid in geometric terms. The gaps and slivers can have arbitrarily small positive thickness, or even zero thickness, if computed exactly, and the use of approximate arithmetic in the computations can perturb the result boundary sufficiently to cause 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). The output may have other aspects that could be considered undesirable, namely coincident vertices, zero-length edges and zero-area facets. For most applications it is therefore preferable to apply a data-smoothing post-process to simplify the structure, removing gaps and slivers and other undesirable features below a certain thickness or size.

In this paper we focus on the basic Boolean algorithm. The main part of the paper – Section 2 – describes the basic Boolean algorithm and provides proofs that the operation is topologically robust. Section 3 briefly discusses the data-smoothing post-process, giving in broad terms the requirements of that process and outlining one way of implementing it. Section 4 briefly describes an implementation of the Boolean algorithm as a whole by the first author within a commercial environment, while Section 5 covers possible future work.

2. The basic Boolean algorithm

2.1. 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 representation. 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 [18,19].)

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 [20]. The appendix to [21] demonstrates one way (different to ours) 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

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

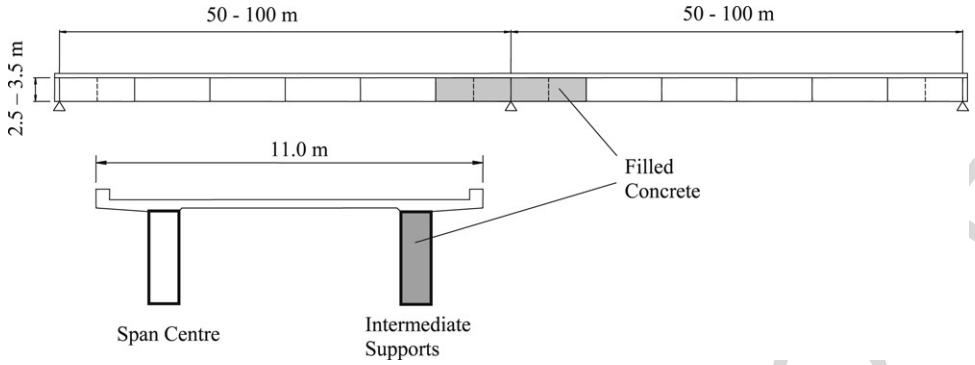

Fig. 1. Examples of valid shape representations in 2D space.

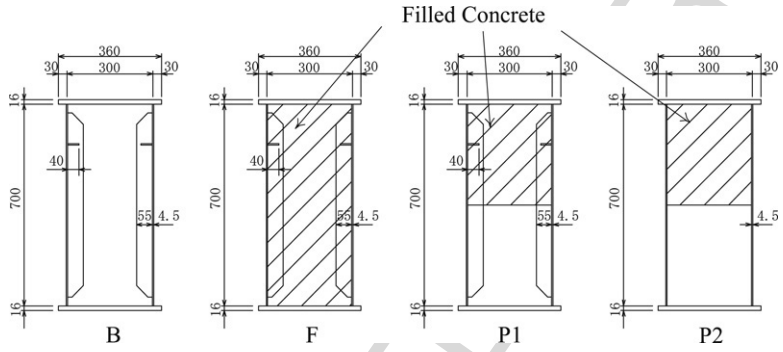

Fig. 3. Topologically valid but geometrically invalid shape representations. (a) The inside of the object lies on the wrong side of the edge; (b) the central region is enclosed twice; (c) a doubly-enclosed region exists, and also an inside-out region.

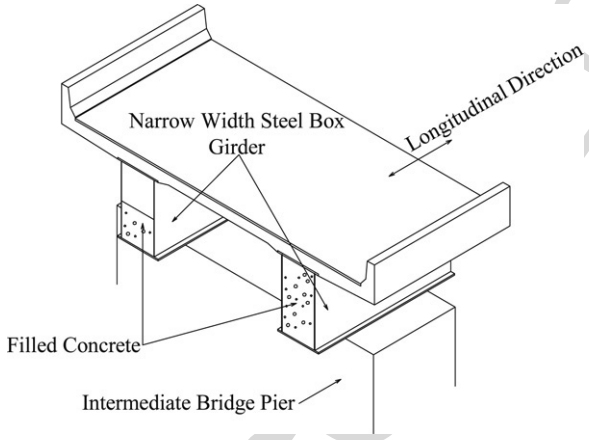

Fig. 2. Topologically invalid shape representations. (a) and (b) Edges do not all connect; (c) edge directions are inconsistent.

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).

Fig. 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, Fig. 2 shows examples that break the topological constraints, and Fig. 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*.
- *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.

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

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* 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 have 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, those half-edges for which the start-vertex is the start-vertex of the edge are known as *forward half-edges* of the edge, and the remainder are known as *backward half-edges*. The shape boundary constraint deems that an edge has an equal number of forward and backward half-edges (if it has distinct vertices).

2.2. 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 resembles the technique advocated by Edelsbrunner and Mücke in the context of exact arithmetic calculations [22], 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

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

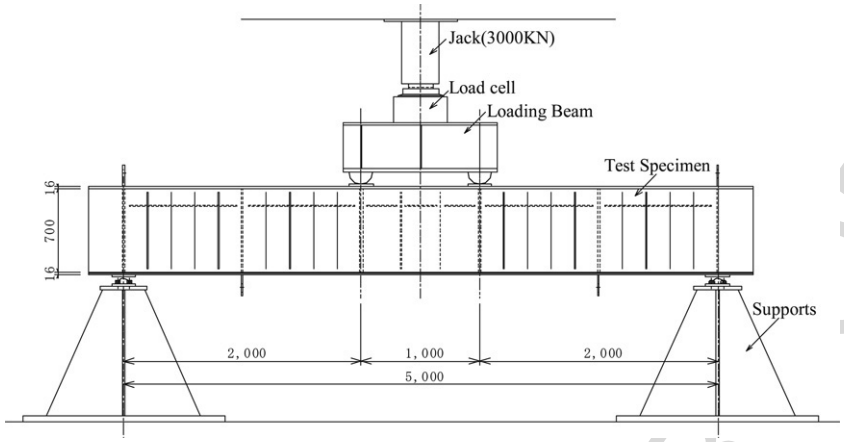

Fig. 4. The hierarchy of operations for the basic Boolean algorithm.

form of a double pyramid between the 16 types of operation, as shown in Fig. 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.

2.3. 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 first 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 Fig. 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 Fig. 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 Fig. 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. 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

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

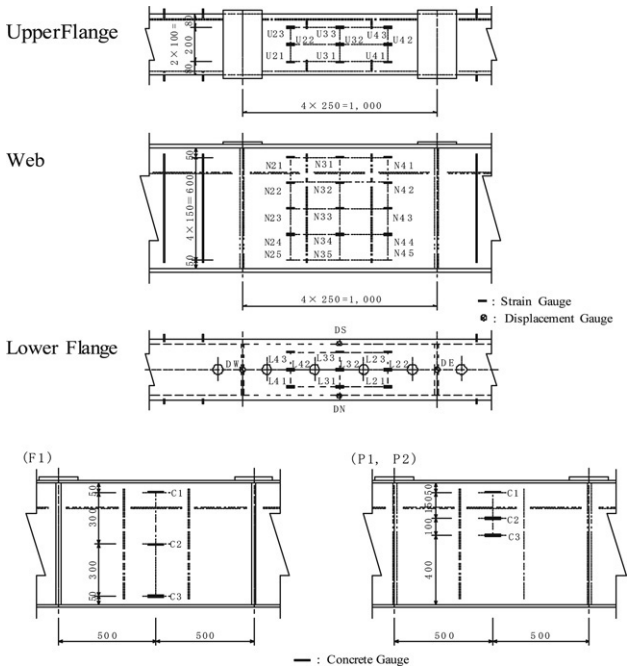

Fig. 5. Examples showing how an intersection between two edges in (x, y) space is determined. The two edges intersect only if they overlap in (x) space ($\max x_B \geq \min x_A$ and $\min x_B < \max x_A$) and if $y_B \geq y_A$ at one end of the overlap range in (x) and $y_B < y_A$ at the other (i.e. at $x = \max(\min x_A, \min x_B)$ and $x = \min(\max x_A, \max x_B)$).

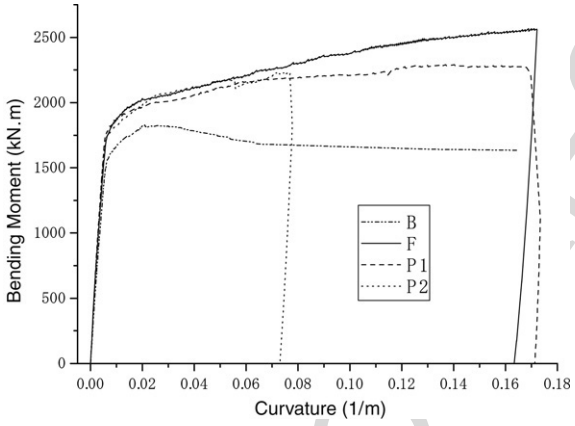

Fig. 6. Example showing how an intersection between an edge and a triangular facet is determined in full 3D space. The entities overlap over a segment in (x, y) space; the facet 'shadows' the edge (has a larger z-component) at one end, and the edge shadows the facet at the other.

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

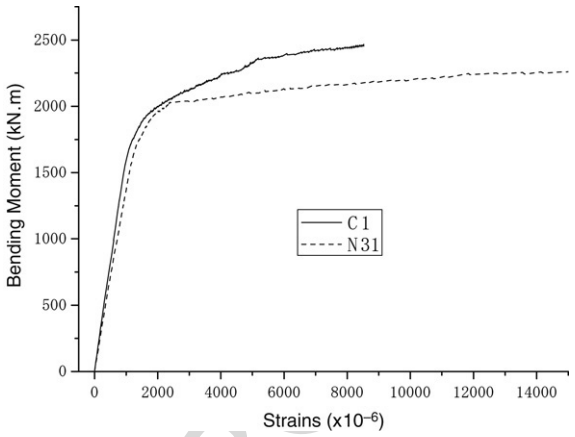

Fig. 7. Example showing how an intersection between a vertex and a facet is determined in 2D (x, y) space. The vertex is shadowed in y by a bounding half-edge of the facet going right to left, and shadows another half-edge going left to right.

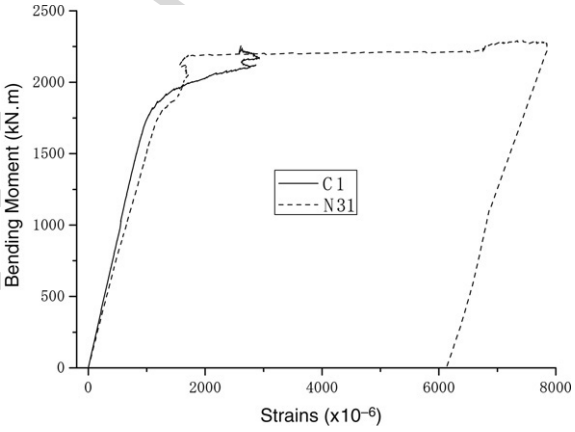

Fig. 8. An example showing how the basic algorithm determines the intersection point between a triangular facet defined by points **a**, **b**, **c** and an edge defined by points **d**, **e**. Points **f**, **g**, **h**, **i** are determined by the 1D intersection calculations, **j**, **k**, **l** by the 2D intersection calculations, and the result, **m**, by the final 3D intersection calculation.

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.

Fig. 8 gives an example of how the algorithm determines the intersection point between a triangular facet based on points **a**, **b**, **c** and an edge based on points **d**, **e**.

2.4. Formulae for determining intersections

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

Functions known as *intersection functions* are 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.

Functions known as *shadow functions* are 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:

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

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 it 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 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 [17]. This number is a net count of how many times the facet boundary goes round the vertex in an anticlockwise direction. Similarly, $X_{03}(v_A, B)$ and $X_{30}(A, v_B)$ are formulations for the winding number of the vertex v within the polyhedral shape in full 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 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, associated with non-zero values of $X_{i-1,j}(\partial^* o_A, o_B)$ and $X_{i,j-1}(o_A, \partial^* o_B)$ ($\partial^* o$ designating one of the boundary component entities bordering entity o). Two such 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. (We prove later that there will always be at least one lower-level intersection of each type.) 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 the original pair of entities in the higher-dimensional space are defined by the standard linear interpolation formulae:

$$\mathbf{x}_A = \mathbf{x}_{A+} - t(\mathbf{x}_{A+} - \mathbf{x}_{A-}) \quad (2)$$

$$\mathbf{x}_B = \mathbf{x}_{B+} - t(\mathbf{x}_{B+} - \mathbf{x}_{B-}) \quad (3)$$

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

$$t = \Delta_+(\Delta_+ - \Delta_-) \quad (4)$$

where

$$\Delta_+ = \xi_{B+}^{(i+j)} - \xi_{A+}^{(i+j)} \quad (\geq 0) \quad (5)$$

$$\Delta_- = \xi_{B-}^{(i+j)} - \xi_{A-}^{(i+j)} \quad (< 0). \quad (6)$$

The relationship $\Delta_- < 0 \leq \Delta_+$ guarantees the avoidance of division by zero in Eq. (4).

2.5. 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.

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

Table 1
Formulae for intersection functions

Level	Formula	Situation leading to value of +1
0	$X_{00}(v_A, v_B) = 1$	
1	$X_{01}(v_A, e_B) = S_{00}(v_A, v_e(e_B)) - S_{00}(v_A, v_s(e_B))$ $X_{10}(e_A, v_B) = -S_{00}(v_e(e_A), v_B) + S_{00}(v_s(e_A), v_B)$	v_A lies within e_B, with e_B going left to right 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 faces 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 faces facing outwards

These values are related to the intersection status value in accordance with the particular operation being applied:

$$I_{03}(v_A, B) = c_A + c_I X_{03}(v_A, B) \quad (7)$$

$$I_{12}(e_A, f_B) = c_I X_{12}(e_A, f_B) \quad (8)$$

$$I_{21}(f_A, e_B) = c_I X_{21}(f_A, e_B) \quad (9)$$

$$I_{30}(A, v_B) = c_B + c_I X_{30}(A, v_B) \quad (10)$$

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

$$\text{Union:} \quad c_A = 1 \quad c_B = 1 \quad c_I = -1 \quad (11)$$

$$\text{Intersection:} \quad c_A = 0 \quad c_B = 0 \quad c_I = +1 \quad (12)$$

$$\text{Difference:} \quad c_A = 1 \quad c_B = 0 \quad c_I = -1. \quad (13)$$

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 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 inside B, to indicate that it is to be retained (as for the intersection operator). Vertices of B lying outside 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.

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 $\partial^* o_B$.

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

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

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_A 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 of 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 $\mathbf{x}_{e_i}$:

$$\mathbf{v} = \sum_i \mathbf{x}_{e_i} - \sum_i \mathbf{x}_{s_i}. \tag{14}$$

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

$$(\mathbf{x}_{s_{i+1}} - \mathbf{x}_{s_i}) \cdot \mathbf{v} \geq 0 \tag{15}$$

$$(\mathbf{x}_{e_{i+1}} - \mathbf{x}_{e_i}) \cdot \mathbf{v} \geq 0. \tag{16}$$

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 of the triangulation process are covered in Section 2.7.

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.

2.6. 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, but numerical rounding errors relating to the vertex positions, either in the basic algorithm itself or in subsequent operations, can create invalid regions and boundary self-intersections in the resulting structure. A consequence of that could be that in a subsequent operation the basic algorithm would, for example, evaluate $X_{30}(A, v_B)$ to something other than 0 or 1, designating that vertex v_B lay in a multiply-enclosed region of A if the value exceeds 1, or alternatively in an inside-out region for a negative value. We 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 mentioned will never occur.

Nevertheless, as Section 2.8 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 Fig. 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

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

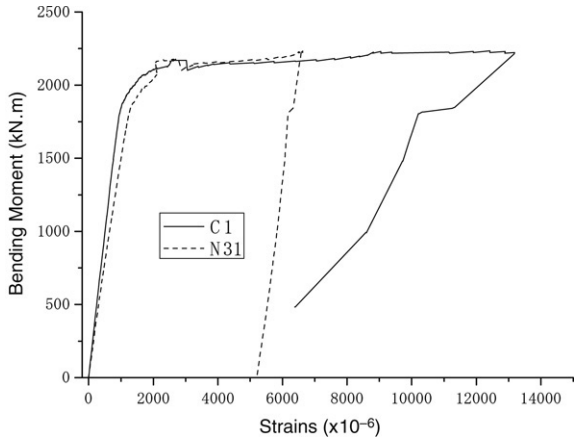

Fig. 9. Example showing (a) two shape representations, one geometrically invalid, and (b) the result generated by the union operation. In the result the retained part of the bottom edge of the rectangle is represented twice where it overlaps the inside-out region of the other shape. In this example it is manifested as two overlapping segments (drawn separated for the purposes of illustration).

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 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 [17], and it is incremented in value by 1 each time one crosses the boundary from the outer side to the inner side. 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:

$$u(\mathbf{x}) = a(\mathbf{x}) + b(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x}) \quad (17)$$

$$i(\mathbf{x}) = a(\mathbf{x})b(\mathbf{x}) \quad (18)$$

$$d(\mathbf{x}) = a(\mathbf{x}) - a(\mathbf{x})b(\mathbf{x}). \quad (19)$$

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 Eqs. (11)–(13):

$$\phi(\mathbf{x}) = c_A a(\mathbf{x}) + c_B b(\mathbf{x}) + c_I a(\mathbf{x})b(\mathbf{x}). \quad (20)$$

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.

2.7. 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 [23, 1]. 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 new edges known as *internal edges* are 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. 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 Fig. 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,

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

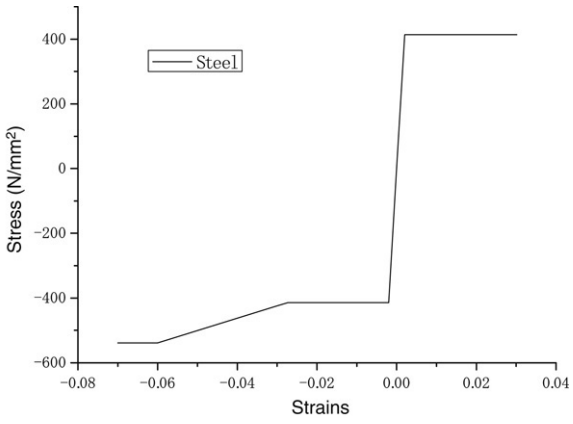

Fig. 10. Example showing that a geometrically invalid quadrilateral can be triangulated one of two ways. Whichever way is selected, the two triangles face opposite directions and overlap. However, option (b) is preferable because the region of overlap (drawn shaded) is smaller.

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.

2.8. 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. We 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 Eqs. (2)–(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 *intersection data at level k* ($k = 0, 1, 2$ 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 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 Eq. (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 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 Eqs. (2)–(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:

$$E_{13}(e_A, B) = I_{03}(v_e(e_A), B) - I_{03}(v_s(e_A), B) + \sum_{f \in \partial B} I_{12}(e_A, f) \quad (21)$$

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

$$E_{31}(A, e_B) = \sum_{f \in \partial A} I_{21}(f, e_B) + I_{30}(A, v_e(e_B)) - I_{30}(A, v_s(e_B)). \quad (23)$$

Substituting the I terms with the formulae given in Eqs. (7)–(10), then substituting the X terms with the formulae given in Table 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. □

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

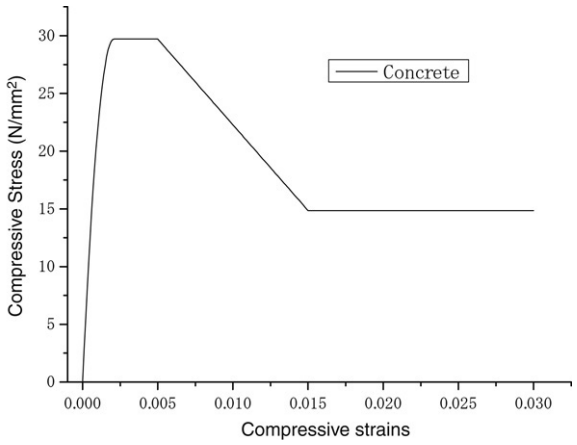

Fig. 11. Example showing the topological structure created when cube A is on top of cube B. (a) shows the structure generated by the basic algorithm, and (b) shows the structure as it is expected to be after data-smoothing. Facets are labelled according to the shape of origin and the facet normal direction assuming an east–north–up alignment of the coordinate system. Vertices that are identically positioned following the basic operation are encircled.

3. 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 face of one coinciding exactly with the bottom face of the other. If the lower cube is the one designated as B, then in consequence of the symbolic perturbation rules the two boundaries are considered to intersect, and the polygon mesh variant of the basic Boolean operation yields a single-shell structure with topology as depicted in Fig. 11(a). In this example, both retained vertices and intersection vertices coincide, certain edges 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 Fig. 11(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 Fig. 11(b), but with A and B interchanged. Other topological structures can be generated by the basic operation when the faces 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 are those required for data normalisation, as described in [24].

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 that we discuss 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. The operations can distort facets to make them non-planar; multiple operations may distort the boundary too much (as with data normalisation [25]); self-intersections may arise that cannot be resolved by the operations; the method may not terminate in reasonable time; and so on.

The details of how best to implement the smoothing operation and whether some provable form of robustness can be achieved remain an open issue. It is clearly an important issue, since the success of the Boolean operation as a whole depends on the success of the data-smoothing post-process, but it is beyond the scope of this paper to consider these matters further.

4. Implementation

The first author devised and successfully implemented the Boolean algorithm for use within Cadcentre's widely circulated Plant Design Management System [26]. 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.

A data smoothing process was implemented to make the structure suitable for downstream processing. It works by applying the operations mentioned 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 0.1 mm, in principle until no further operations are possible (see below). Tests carried out at the time of implementation

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

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.

As we mentioned in the last section, the data-smoothing process implemented does not have a theoretical backing, so it is not provably fully robust, but it has turned out to be sufficiently reliable. Users only ever reported one fault in the released product that turned out to be related to the Boolean operation in its entirety. This was a problem of non-termination in the smoothing operation, apparently complexity-related, which occurred for one particular case involving four coincident, almost vertical facets. The problem was resolved in subsequent releases by enforced early termination in the data-smoothing process, though this led to shard-like artifacts in the result for that one known problem case. An alternative fix – applying data smoothing after every basic Boolean operation – would have resolved the less serious problem of the artifacts as well, but the decline in performance generally that would have been incurred was not justified given the rarity (and non-fatal nature) of the artifact problem.

5. Discussion and future work

We have presented an algorithm for performing Boolean operations on polyhedral solid representations using approximate arithmetic, and proved it to be topologically robust insofar that correct connectivity in the input guarantees correct connectivity in the output. In general it is desirable to apply a data-smoothing process to the output to remove artifacts, in particular gaps and slivers, which can be problematic because they can distort to form boundary self-intersections.

A clear advantage of the method we present over others that use standard machine arithmetic is the guarantee that the result will always have correct connectivity. The method as a whole is not fully robust geometrically, because the output of the basic operation has to undergo data-smoothing, an 'engineering' solution not backed by rigorous theory.

Methods based on exact arithmetic avoid the particular robustness problems, both topological and geometric, associated with methods that use standard machine arithmetic, but they have the disadvantage that number representations become increasingly complex with successive computations. This can slow down processing, particularly for cascaded operations where the output of one operation is the input of another [25].

The basic Boolean algorithm is straightforward because of its formulaic nature. It operates by considering pairs of entities in the input, so can be implemented to run in $O(n \log n)$ time for structures that do not have significant complexity [1].

Though we have described the basic Boolean algorithm in terms of a 3D operation, it can in principle be implemented in any finite-dimensional domain. It should also be possible to extend the algorithm to the related problem of determining the overlay between two piecewise-linear cellular subdivisions.

The overall method is not fully robust because of the data-smoothing, but in practice the smoothing process very rarely caused trouble in the version that was implemented commercially. The experience of the author who implemented the algorithm is that it is a lot easier to implement the smoothing operation to a high degree of reliability than to implement the Boolean operation to the same degree (or even a lesser degree) of reliability. The smoothing operation is essentially a sequence of local modifications, so potential problems are limited to failure of the algorithm to terminate, or alternatively failure to resolve all geometric errors, and excess distortion to the shape boundary. In contrast, designing a reliable Boolean algorithm is fraught with difficulties if one takes the strategy of constructing the final result directly from the input data. To work satisfactorily such an algorithm has to take into consideration a wide range of special cases, including combinations of special cases. While this can be (and has been) achieved for exact arithmetic implementations, an implementation based on approximate arithmetic, and which out of necessity has to test for coincidence within tolerance, is liable to run into problems of consistency. For this reason such an implementation is likely to be susceptible to a range of problems, including connectivity errors, when special cases interact in unfortunate ways. Consequently, 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 the same Boolean operation, then that indeed further complicates the implementation of that operation. With the Boolean algorithm we present, downstream processes have to be tolerant only to possible geometric errors. The Boolean algorithm itself satisfies that requirement.

We have not covered the data structure representation in detail. The algorithm is amenable to a range of possible representations, and it remains an open question as to which might be preferred. We make the comment, however, that for the basic Boolean operation it is an unnecessary complication to have a representation that requires the 3D boundary to be formed into shells and lumps, or for facet boundaries to be formed into loops and regions.

We are currently investigating whether a variant of the basic Boolean operation could be used to determine surface self-intersections, as opposed to the intersection between two distinct surfaces. The hope is that the algorithm concerned would provide mechanisms both for geometric validation and for a boundary correction process that would be backed by theory, thus making the operation geometrically robust as well as topologically robust. Such an operation could be used to implement Boolean operations, and also operations such as offsetting, filleting and ε-regularisation.

The question remains whether the technique described here could be extended to apply to the curved surface domain. It should be possible to handle subdividable surfaces by refining polyhedral bounds to each surface, though such a technique may not be practical when surfaces are nearly coincident.

References

[1] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry. Springer; 1997.
[2] Hoffmann C. Geometric and solid modeling: An introduction. Morgan Kaufmann; 1989.

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005

[3] Hoffmann C. Robustness in geometric computations. Journal of Computing and Information Science in Engineering 2001;1:143–56.
[4] Hoffmann C, Hopcroft J, Karasik M. Robust set operations on polyhedral solids. IEEE Computer Graphics & Applications 1989;9(6):50–9.
[5] Fortune S, Van Wyk CJ. Efficient exact arithmetic for computational geometry. In: Proceedings of the 9th ACM symposium on computational geometry. 1993. p. 163–72.
[6] Fortune S. Polyhedral modelling with exact arithmetic. In: Proc. 3rd symp. solid modeling. NY: ACM Press; 1995. p. 225–34.
[7] Mehlhorn K, Näher S. The LEDA platform of combinatorial and geometric computing. Cambridge University Press; 1999.
[8] CGAL. http://www.cgal.org/; 2006.
[9] Keyser J, Culver T, Foskey M, Krishnan S, Manocha D. ESOLID—a system for exact boundary evaluation. Computer-Aided Design 2004;36: 175–93.
[10] Emiris IZ, Kakargias A, Pion S, Teillaud M, Tsigaridas EP. Towards an open curved kernel. In: Proc. 20th annual ACM symposium on computational geometry. 2004. p. 438–46.
[11] Brönnimann H, Burnikel C, Pion S. Interval arithmetic yields efficient dynamic filters for computational geometry. In: Proc. 14th annu. ACM sympos. comput. geom. 1998. p. 165–74.
[12] LEDA. http://www.algorithmic-solutions.com/enleda.htm/; 2006.
[13] Sugihara K, Iri M, Inagaki H, Imai T. Topology-oriented implementation—an approach to robust geometric algorithms. Algorithmica 2000;27:5–20.
[14] Flaquer J, Carbajal A, Mendez MA. Edge-edge relationships in geometric modelling. Computer-Aided Design 1987;19(5):237–44.
[15] Kalay YE. Determining the spatial containment of a point in general polyhedra. Computer Graphics and Image Processing 1982;19(4):303–34.
[16] Gardan Y, Perrin E. An algorithm reducing 3d boolean operations to a 2d problem: Concepts and results. Computer-Aided Design 1996;28(4): 277–87.
[17] Haines E. Essential ray tracing algorithms. In: Glassner AS, editor. An introduction to ray tracing. London (UK): Academic Press; 1989. p. 33–77.
[18] Baumgart BG. Winged edge polyhedron representation. Tech. rep. Stanford (CA, USA); 1972.
[19] Foley JD, van Dam A, Feiner SK. Computer graphics: Principles and practice. 2nd ed. Reading (MA), Wokingham: Addison-Wesley; 1996.
[20] Requicha A. Representations for rigid solids: Theory, methods, and systems. Computing Surveys 1980;12(4):437–64.
[21] Fortune S. Polyhedral modelling with multiprecision integer arithmetic. Computer-Aided Design 1997;29(2):123–33.
[22] Edelsbrunner H, Mücke E. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. ACM Transactions on Graphics 1990;9(1):66–104.
[23] Bern MW, Eppstein D. Polynomial-size nonobtuse triangulation of polygons. International Journal of Computational Geometry and Applications 1992;2(3):241–55.
[24] Milenkovic VJ. Verifiable implementations of geometric algorithms using finite precision arithmetic. Artificial Intelligence 1988;377–401.
[25] Milenkovic VJ. Shortest path geometric rounding. Algorithmica 2000; 27(1):57–86.
[26] Vantage plant design management system, http://www.aveva.com/media_centre/library/datasheets/vpd_pdms.pdf; 2006.

Please cite this article in press as: Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Computer-Aided Design (2007), doi:10.1016/j.physd.2007.01.005