Error-Free Boundary Evaluation Based on a Lazy Rational Arithmetic: A Detailed Implementation

M.O. Benouamer, D. Michelucci, and B. Peroche

Ecole Nationale Supérieure des Mines de Saint-Etienne
Département Informatique Appliquée
158, Cours Fauriel
42023 SAINT-ETIENNE Cedex 2, FRANCE
Tél : (33) 77 42 01 23
Fax : (33) 77 42 00 00
Email : micheluc@emse.fr

Abstract

A new boundary evaluation method is presented. It is based on error-free Boolean operations on polyhedral solids. We describe, in detail, an intersection algorithm that handles, in a straightforward way, all the possible geometric cases. We also describe a general data structure that allows an unified storage of solid boundaries. The intersection algorithm always runs to completion, producing consistent solids from consistent operands. Numerical errors are handled at an algorithm independent level: an original exact arithmetic that performs only the necessary precise computations. Results from our implementation of this CSG solver are discussed.

***Keywords** : Solid Modeling, Non-manifold Polyhedra, Boundary Evaluation, Numerical Errors, Lazy Rational Aritmetic.*

I. INTRODUCTION

Constructive Solid Geometry (CSG) and Boundary Representation (BRep) are two well-established schemes for specifying solid objects. Many solid modeling systems combine the two schemes to satisfy a wide range of applications. Converting a CSG representation of an object to an equivalent boundary representation is known as the *Boundary Evaluation* problem [17]. It entails computing the boundary of a solid that is the result of a set-operation (intersection, union, or difference) applied to two solids with known boundaries. Solids are closed regular subsets of the Euclidean space, and set-theoretic operations are replaced by their regularized versions [22]. Several algorithms exist for solving Boolean operations on polyhedral solids, but not all satisfactorily address the crucial problem of numerical errors that are inherent to floating-point computations [7, 10, 11, 19, 20, 21].

In this paper, we present solutions that we have experimented in the implementation of a solid modeler. Our approach is based on a general algorithm and data structure that naturally accommodate non-manifold geometric cases.

Numerical errors are avoided by the use of a new kind of exact arithmetic (called "Lazy Arithmetic"), that performs only the necessary precise computations without the algorithm having to foresee these computations.

Section **II** discusses the general difficulties in boundary evaluation and gives a quick survey of the known solutions. Section **III** describes the data structure that we have used to store the boundary of polyhedral solids. In Section **IV** we describe our intersection algorithm and the way regularized union and difference operations are solved in terms of intersection and complement. Section **V** gives the principle of the "lazy" rational arithmetic. Section **VI** discusses the way our error-free polyhedral modeler may interface with other modelers that do not rely on an exact arithmetic. Section **VII** closes the paper by presenting some experimental results.

II. GENERAL DIFFICULTIES

II.1. Reducing the amount of geometric computations

To avoid unnecessary geometric computations (i.e. intersection and classification), algorithms quickly seek a sufficient list of candidate face pairs, that includes the actually intersecting pairs. Usually, solids and faces are boxed, and the potentially intersecting faces are determined by testing their bounding boxes for intersection. There exist Multidimensional searching techniques that find the K intersecting box pairs in $O(K+N.log^2N)$ time, where N is the total number of boxes [13]. Other methods use more elaborated spatial directories [12]. For the need of our algorithm we have used classical bounding boxes, as in [10] and [19], even though many other techniques may be devised here.

II.2. Dealing with degenerate cases

As pointed out in [14], geometric algorithms meet troubles when intersecting geometric elements that are not in "general position". Each "special" (i.e. degenerate) case need be represented and treated in a "special" way, leading to complex data structures and algorithms. Moreover, one has to keep in mind that degeneracy often creates opportunities for inconsistencies. This led Edelsbrunner and Mücke [3] to eliminate all degeneracies by slightly perturbing the input data.

Our method has no more than exactly one "special" case to deal with, it is the non-manifold situation (called "isolated" vertex) treated in Section IV.5.

II.3. Dealing with numerical errors

The first attempt to cope with numerical errors consists in the use of some empirical "epsilons". This method cannot always guarantee consistent results as pointed out in [10].

• Guibas, Salesin and Stolfi [6] describe a theoretical framework that allows maintaining intervals in which decisions may be taken safely. But how does one decide outside these intervals ?

• Segal and Sequin [18, 19, 20] impose a minimum separation between each pair of primitives (faces, edges, or vertices): any two primitives that are within less than a chosen minimal distance must be either merged or pulled apart to maintain the minimum separation.

• Hoffmann, Hopcroft and Karasick [7] also impose a minimum separation, but they resort to an additional symbolic reasoning to ensure a decision never contradicts previous ones.

• Milenkovic [15] also places a higher priority on topology. He describes a verifiable implementation of a line arrangement algorithm, that maintains a consistent geometrical data base.

• Mantyla and Sulonen [11] ensure topological consistency in their GWB solid modeler by the strict use of Euler operators. However, these operators do not avoid contradictions between numerical and topological data.

• Sugihara and Iri [21] observe that if the original geometric data are represented in a finite precision, the relative topological configuration of two geometric elements can be computed exactly in some finite precision. They show how it is possible to build an error-free polyhedral modeler based on trihedral primitives.

• Karasick, Lieber and Nakman [8] rely on a rational arithmetic, a clever use of intervals (of integers) and a careful engineering design, to implement their error-free Delaunay triangulation.

Our approach also relies on a rational arithmetic, but unlike the previous approachs, the way numerical errors are avoided is totally transparent from the algorithm point of view. Moreover, the rational arithmetic used here, is computationally far less expensive than traditional pure rational arithmetics, for that only the strictly necessary precise computations are performed, without the algorithm having to presuppose these particular computations (see Section V).

III. BOUNDARY REPRESENTATION

III.1. Rational polyhedral solids

The difficulty of implementing a polyhedral solid modeler depends, to a certain extent, on the underlying representation (i.e. data structures). Our modeler is based on a general data structure that is designed to represent any polyhedral solid whose boundary is an *orientable non-manifold* surface, that is to say there may be boundary points whose neighbourhoods are not homeomorphic to a disk (e.g. Fig. 1). Solids may have a bounded or an unbounded volume, but their boundary must have a finite, bounded surface area. In our solid modeler, the only unbounded solids we deal with are those that are temporarily created when evaluating a *difference* or a *union* node of a CSG tree whose primitives are bounded solids, in terms of *intersection* and *complement* operations. Thus, these particular unbounded solid are always specified as the regularized complements of bounded solids (Sect. IV.6).

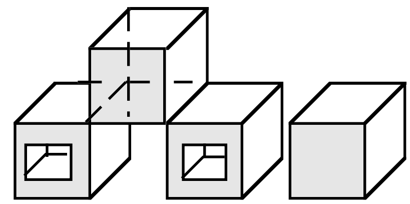

Figure 1. *A non-manifold , multi-connected polyhedron*

From now on, we assume *rational* polyhedral solids, that are polyhedral solids represented by their boundary in which the coordinates of the vertices and the coefficients of the defining face plane equations are all available as *rational numbers*, regardless of the way these numbers are represented. A rational solid is always numerically consistent ; for instance, each vertex coordinate triple satisfies the plane equation of each incident face.

The central topological element of the BRep is a new entity, called "flap", that may be thought of as a "piece" of some face which hangs on some oriented edge. Formally, a flap is an explicit representation of a two-dimensional edge-neighbourhood as defined in [17], that is to say a neighbourhood, with respect to a face `f`, of points in the interior of an edge `e` of `f`. We found the flap entity yields the simplest boundary representation of non-manifold polyhedral solids.

The BRep is made of two lists for edges and faces. A face is specified as a list of flaps, where each flap belongs to (i.e. it points to) a unique face and is incident to a unique edge. Each edge has an even number of flaps incident to it (see Fig. 2).

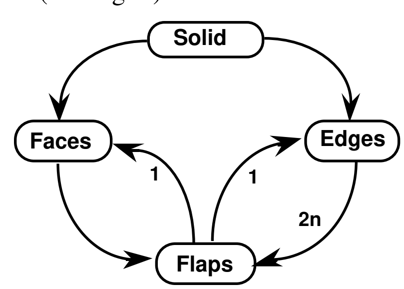

Figure 2. *The adjacency graph of the BRep.*

III.2. Data structure

The data structure chosen for *rational* solids contains the following records:

Solid-record

• A Boolean indicator that says whether the solid is bounded or not (recall that, in our setting, an unbounded solid is the regularized complement of some bounded solid),

• the list of all faces of the solid,

• the list of all edges of the solid,

• the extent of the solid. It is simply the smallest 3D-box whose sides are aligned with the coordinate axes, that contains all the vertices of the solid.

Face-record

• A quadruple *(a, b, c, d)* of rational numbers that define the equation $(a.x + b.y + c.z + d = 0)$ of the face plane. We assume *oriented* planes, so that the gradient *(a,b,c)* always gives an *outward normal* for the face (i.e. a vector directed from the interior to the exterior of the solid). The defining quadruple is stored in a canonical form (for instance, the first non null coordinate is +1 or -1),

• the list of all flaps of the face,

• the extent of the face. It is the smallest 3D-box whose sides are aligned with the coordinate axes, that contains all the vertices of the face.

Edge-record

• Two triples p_1 and p_2 of rational coordinates, that define the *left* and the *right* endpoint of the edge, in the sense that p_1 precedes p_2 in lexicographic (i.e. `xyz`-) order. Edges are unique, they are always stored as ordered pairs (p_1, p_2),

• the list of pointers to all the incident flaps,

• a list of "splitting-points": that are points at which the edge will be split during the intersection algorithm (see Section IV).

Flap-record

• a pointer to the attached edge `e`,

• a pointer to the attached face `f`,

• the *side* `s` of the flap: an integer (±1) defined below.

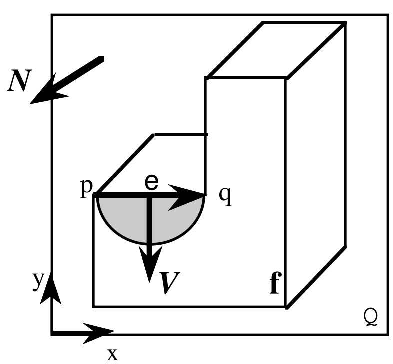

Figure 3. *A flap incident to a manifold edge.*

Consider the front face `f` of the solid shown in Fig. 3, where the edge `e` (= [`p`, `q`]) lies on the plane `Q` of `f`. For a certain value `s` (= ±1), the triple (`e`, `f`, `s`) defines a flap of `f`, incident to `e`. The side value `s` identifies the half-plane of `Q` (delimited by the extension line of `e`) that contains the interior of `f`, in the immediate vicinity of `e`. Let ***N*** be the outward normal of `f`, and ***E*** the tangent vector of `e`, starting at `p` and ending at `q`. From now on, the cross-product $\boldsymbol{V} = s * (\boldsymbol{E} \wedge \boldsymbol{N})$ will be called the *flap-vector* of the flap (`e`, `f`, `s`). This vector is orthogonal to ***E*** and has the following property: for a given point I ∈ (`p`, `q`), any point like $J = p + \varepsilon * \boldsymbol{V}$ is in the interior of `f`, for a sufficiently small ε>0. This unambiguously defines the side value `s`.

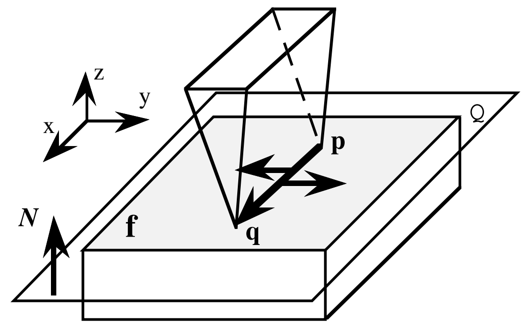

Figure 4. *Flaps incident to a non-manifold edge.*

It may happen a face `f` lies on both sides of an incident edge `e`, as in the non-manifold solid of Fig. 4. In such case, the non-manifold edge `e` is assigned two incident flaps with *opposite* sides (`e`, `f`, ±1).

III.3. What is new in this BRep ?

As pointed out in [17], the *face* entity need be carefully defined. In our BRep we assume "maximal" faces. Such a face is the largest 2D-subset of the solid's boundary, that lies on a single plane so that the solid (i.e. the material) is on the same side of the plane, in the immediate vicinity of the face. The solid shown in Fig. 1 has exactly 20 maximal faces, and the front face (shaded) is made of 24 flaps.

Unlike the "star-edge" data structure used in [7], or the one used in [19], our BRep has neither explicit "shells" for the solid's boundary, nor explicit "loops" for the solid's faces. The boundary of a solid is simply specified as a list of small boundary pieces (the flaps), incident to the solid's edges. Faces need not be connected, they are specified as maximal lists of coplanar flaps. There may be voids in solids or holes in faces, but neither are explicitly stated. Edges are unique and stored as ordered pairs of endpoints. No particular order is assumed on the edges of the solid, or on the flaps that belong to a given face or that are incident to a given edge.

One may find this data structure very rudimentary compared with the variants of the Baumgart's "Winged-Edge" [1]. This is a deliberate choice leading to a simple and general data structure that allows unified treatment and representation of multiple topological situations, including the non-manifold ones. Topological informations that are not explicitly stated may be retrieved, on request, by means of linear scanning or by simple geometric computations.

Remark : Strictly speaking, maximal faces are not treated as a polygonal subsets of the boundary (i.e. ordered sequences of coplanar edges and vertices). They are instead artificial entities that simplify the intersection algorithm. However, faces have an *interior* and an *exterior* in that the parity algorithm holds for point/face classification [22].

Consider the shaded rectangular face `f` of the non-manifold solid of Fig. 4. This face is represented by six flaps, two of which are incident to the non-manifold edge `e` = [`p`, `q`]. Given a point on the plane `Q` of `f`, how can it be classified with respect to `f`? Cast a random ray starting at the test point, and compute the number of times the ray properly intersects the edges attached to the flaps of `f` (if the ray meets the endpoint of some edge, just cast an other ray and redo the whole counting). Since the edge `e` has exactly two incident flaps of `f`, a possible intersection point between the ray and `e` will be computed and counted exactly twice. Thus the parity (i.e. whether it is *even* or *odd*) of the total number of intersection points will be the same as if the edge `e` were omitted in the classification (i.e. as if the face `f` were an ordinary polygon).

III.4. Validity conditions of this BRep

To be valid, the BRep must satisfy the following three conditions :

1) two distinct edges may neither overlap nor intersect at a point which is not a common endpoint ;

2) two distinct faces may intersect only at edges that are listed in the BRep ;

3) an edge must have an *even* number of flaps incident to it. Useless edges, that have exactly two incident flaps (of the same maximal face), must be deleted from the BRep.

IV. INTERSECTION ALGORITHM

We now describe an algorithm for computing the rational BRep of the solid $S = A \cap B$, given two consistent BReps for two rational solids A and B. From now on, all the necessary computations will be supposed error-free , so that no topological inconsistency can arise from numerical errors. Numerical consistency is guaranted by the use of a particular rational arithmetic presented in Section V.

A quick survey

Given two rational solids A and B, with known boundaries ∂A and ∂B, our intersection algorithm follows the general scheme described in [17] ; it roughly proceeds in five steps :

1) it quickly searches for a sufficient list of candidate face pairs, based on the classical method of bounding boxes;

2) each candidate face pair is treated, and all the informations needed for splitting the original boundaries are collected, and adequately stored for later use ;

3) the original boundaries ∂A and ∂B are actually split into non intersecting parts. An intermediate data structure is used to store a superset of $\partial(A \cap B)$: the boundary of a the intersection solid ;

4) each superset element is classified with respect to $A \cap B$ to find the actual members of $\partial(A \cap B)$;

5) the final BRep of $A \cap B$ is generated, then is "cleaned" by deleting the possible "useless" edges.

IV.1. Intersecting candidate faces

Three kinds of candidate face pairs need be distinguished : 1) "transversal" faces, whose planes strictly intersect along some line L ; 2) "coplanar" faces, that lie on the same geometric (i.e. non-oriented) plane ; and 3) "parallel" faces, whose planes are strictly parallel. Obviously, parallel faces can never intersect, and testing whether two faces are strictly parallel is justifiable since we assume error-free computations.

During this step we determine the *cross-edges* between all pairs of candidate transversal faces. A cross-edge between a face `f` of A and a face `g` of B, is the largest open line segment that lies entirely in the interior of each face. The algorithm maintains a global list for all the possible cross-edges, each being stored as a 4-field record (`p`, `q`, `f`, `g`) where `p` and `q` are computed coordinate triples of the *left* and *right* endpoints, and `f` and `g` are pointers to the intersecting faces.

In addition, for each original edge of each solid, we compute all the intersection points between this edge and the faces of the other solid. These points will be used later to split the edge into *homogeneous* sub-edges (Fig. 5). An homogeneous sub-edge of an edge `e` of (say) A, is the largest line segment of `e` that has a constant classification with respect to the other solid `B` : the open segment lies entirely in the exterior, or in the interior, or on the boundary of B.

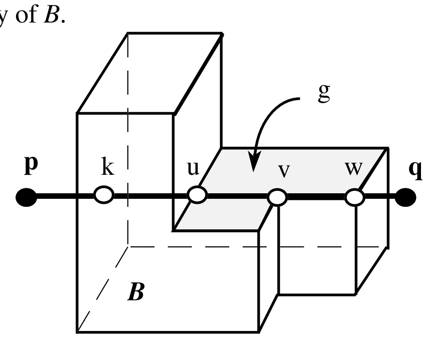

Figure 5. *The edge* e = [p, q] *of the solid A* (*not represented*) *will be split at points* k, u, v, *and* w, *into five homogeneous sub-edges. The sub-edge* [u, v] *need be stored with a reference to the containing face* g *of the other solid B*.

The splitting-points of each original edge `e` are stored as the *left* endpoints of the future sub-edges, within a list attached to `e` (as announced in Sect. III.2). Each splitting-point of `e` is represented as a 2-field record (`p`, `g`), where `p` is a computed coordinate triple, and `g` is a pointer to the possible face of B in the interior of which lies the sub-edge of `e` which starts at `p` (e.g. [`u`, `v`] in Fig. 5).

Splitting-points, as well as cross-edge endpoints, may coincide with endpoints of original edges of A and/or B ; they may also result from a strict intersection between an edge of one solid and an edge or a face of the other solid. The algorithm need not distinguish the possible cases.

IV.1.1 Intersecting two coplanar faces

Intersecting coplanar faces is delicate since faces may be arbitrarily complex. However, our algorithm need not deal with coplanar faces, at all : the necessary work will be done elsewhere, during the intersection of the transversal faces. This is a consequence of our requirement that solids have no "useless" edges, as stated in Section III.4.

IV.1.2 Intersecting two transversal faces

Let `f` and `g` be two transversal faces of A and B, respectively. Since the two faces may intersect only along the intersection line L of their face planes, the face/face intersection can be reduced to a one dimension problem: each face is intersected with the plane of the other face, then the two results are merged to obtain $f \cap g$. This amounts to computing the membership classifications M[L, `f`] and M [L, `g`], then merging them to obtain M[L,$f \cap g$] (see [22]).

Face/plane contacts

Let `f` be a face of A, and `Q` a transversal plane that intersects the plane of `f` along some line L. The possible intersection points between `f` and `Q` will be called face/plane "contacts". A contact between an edge `e` (of `f`) and `Q` may be either a strict intersection point between `e` and `Q` or an endpoint of `e` (see Fig. 6). In both cases, the contact is stored as a 4-field record (*point*, *index*, *tangent-edge-ptr*, *crossing-edge-ptr*), that contains :

• the coordinate triple of the contact ;

• an integer index (0 or +1) that describes the way the edge `e` intersects the plane `Q`. A contact is (by convention) assigned the index value +1 if the edge `e` has one endpoint with a *strictly positive* signed distance, otherwise the index value is set to `0`.

• a pointer to `e` if both endpoints of `e` lie on `Q`: in such case a contact is created at each endpoint of `e`, and the *left* contact carries a pointer to `e` (then called a *tangent-edge*) ;

• a pointer to `e` if the endpoints of `e` lie in opposite sides of `Q` (`e` is then called a *crossing-edge*).

Note that the contact-record has actually two distinct pointer-fields, and at most one of them has a non null value. From now on, the symbol ∅ will stand for any null pointer value.

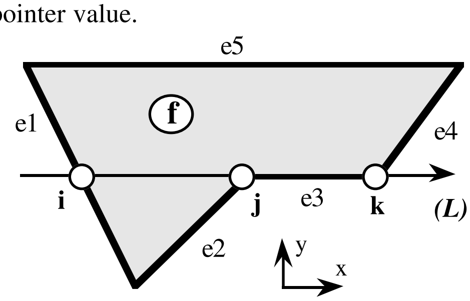

Figure 6. Contacts. Edge `e1` yields (`i`, 1, ∅, `e1`) that stores `e1` as a *crossing-edge* ; `e2` yields (`j`, 0, ∅, ∅) ; `e3` yields two contacts (`j`, 0, e3, ∅) and (`k`, 0, ∅, ∅), the *left* one stores e3 as a *tangent-edge* ; e4 yields (`k`, 1, ∅, ∅) ; e5 yields no contact. Here, we assume that points located "above" the plane Q have positive signed distances.

The pseudo-code below shows how to compute and store (in an intermediate list Lc) all the possible contacts between `f` and `Q`. For each edge `e` of `f`, the signed distances (with respect to `Q`) of both endpoints are computed, and their signs tested to detect a possible contact.

```
IntersectFacePlane (f: Face,  Q: Plane)
{  Lc := ∅;  /* the f/Q contact list */
   for each flap of f do
   {   /* Let e (= [p, q]) the  attached edge */
       α := SignedDistance (p, Q) ;
       β := SignedDistance (q, Q) ;
       if (α and β have opposite signs) then
          {   /* Compute Ω = [p, q] ∩ Q ;  */
              λ := − α  / (β−α) ;
              Ω = p + λ * (q−p) ;
              AddContact  (Ω, +1, ∅, e) }
       else   if (α = 0 and β > 0) then
              AddContact (p, +1, ∅, ∅) ;
       else   if (α = 0 and β < 0) then
              AddContact (p, 0, ∅, ∅) ;
       else   if (α > 0 and β = 0) then
              AddContact (q, +1, ∅, ∅) ;
       else   if (α < 0 and β = 0) then
              AddContact (q, 0, ∅, ∅) ;
       else   if (α = 0 and β = 0) then
              { AddContact (p, 0, e, ∅) ;
                AddContact (q, 0, ∅, ∅) }
       else   /* do nothing */  }
       return Lc
}
```

Face/Plane intersection

Next, all contacts are sorted lexicographically along the line L, by their coordinate triples. Contacts that have identical coordinates (they coincide with endpoints of some edges of `f`) are merged into a single contact which inherits: the common coordinate triple, an index that is the *sum* of the indices, and the possible non null edge-pointer (unique if any) of the merged contacts. The upper part of Fig. 7, below, shows the final index values of the merged contacts.

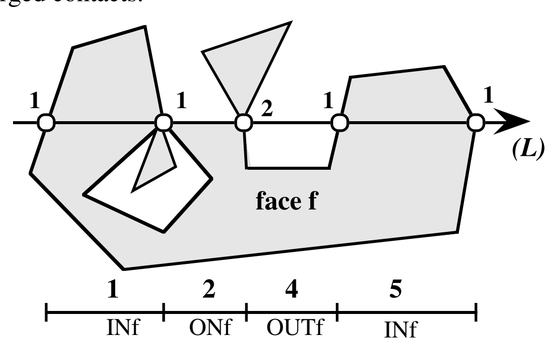

Figure 7. *Classifying L with respect to* `f`.

Now, two consecutive contacts `i` and `j` implicitly define some line segment [`i`, `j`] of L, that has a constant classification with respect to `f`: the open segment lies entirely in the interior, or in the exterior, or on the boundary of f. To classify each segment with respect to f, we just apply the parity principle, as follows: if i carries a non null *tangent-edge* pointer e, then the segment is classified *ON* f (it coincides with the referenced edge e) ; otherwise the segment is classified *IN (OUT)* f if the *sum* of the indices of all the contacts that precede j is *odd (even)*. The index field of each contact-record is reused to store the classification result (See the bottom of Fig. 7).

Merging two Face/Plane intersections

Assume each face has been intersected with the plane of the other face, as shown above. The resulting contact lists are now merged to obtain the largest line segments of *L*, that have a constant classification with respect to f∩g. These homogeneous segments are obtained through a simultaneous scanning of the two lists. The classification of each segment [p, q] (with respect to f∩g) is deduced from its classifications with respect to each face, as follows:

• If [p, q] is *IN* f and *IN* g, it yields a *cross-edge* between f and g. The coordinates of p and q are available. A contact record (p, q, f, g) is created and added to the global cross-edge list. Note that, since faces may be multi-connected, several (maximal) *cross-edges* may exist between f and g. However, each *cross-edge* is computed and stored exactly once.

• If [p, q] is *ON* (say) f and *IN* g, it is a sub-edge of a certain edge e of f, that lies in the interior of g. In such case, a splitting-point record (p, g) is created and added to that edge e.

• If [p, q] is *ON* f and *ON* g, then there are two edges of f and g that overlap along [p, q]. A splitting-point record (p, ∅) is created and added to both edges.

• In each case above, if the contact at p carries a non null *crossing-edge* pointer then the referenced edge e of (say) f strictly intersects the other face g, at p. In such case, a splitting-point record (p, ∅) is created and added to that edge e. If an other contact (of g) exists at p, that carries a non null *crossing-edge* pointer e', then a similar splitting-point record is added to the referenced edge e'.

Geometrically coincident splitting-points of the same original edge are merged into a single record that inherits the common coordinate triple, and the possible non null face-pointer (unique if any) of the merged splitting-points.

The whole intersection process of f and g, is done in *O(N.logN)* time in the total number *N* of flaps of the two faces.

IV.2. Splitting the original boundaries

At this stage, all candidate transversal faces of *A* and *B* have been intersected. All the possible *cross-edges* have been determined, and each original edge of each solid has now a complete list of splitting-points.

In this step, all this information is used to split the original boundaries of *A* and *B* into non-intersecting parts. An intermediate data structure is used to store a superset of *∂(A∩B)*, consisting in all cross-edges and homogeneous sub-edges of the two solids. From now on, both kinds of edges will be called *superset edges*, to indicate that they include the future edges of *A∩B*.

The intermediate data structure

Superset elements to be considered are *vertices*, *edges* and *flaps*. The original faces disappear : what remains from each face is the defining quadruple of the oriented plane it lies on.

• Vertices (original edge-endpoints or new intersection points) are now unique, they are records that store a coordinate triple and a list of pointers to all incident edges.

• Edges (cross-edges or sub-edges) remain unique, they are records that store pointers to the *left* and *right* vertices and a list of pointers to all incident flaps.

• Flaps are unique entities that still need a "side" and an edge-pointer, but the face-pointer is replaced by the plane quadruple of some original face of *A* or *B*. An additional field stores a pointer to the solid (i.e. *A* or *B*) the flap belongs to.

The Fig. 8 below shows the underlying adjacency graph (the numbers above the arrows indicate how many elements may be attached to a given element).

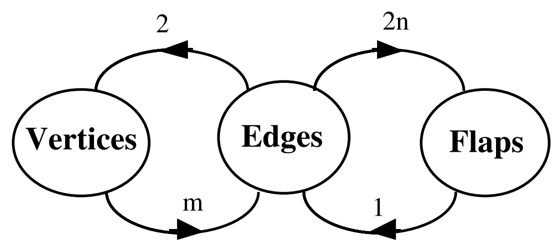

Figure 8. *Adjacency graph of the superset BRep.*

Generating the intermediate data structure

To generate the superset data structure, the pseudo-code below uses the following three primitives:

• AddVertex takes a coordinate-triple and returns a pointer to a unique vertex. Vertices are made unique by means of a *hash-table* that takes coordinate-triples as search keys ;

• AddEdge creates an edge between two vertices and adds it to the incident-edge list of each argument vertex. A pointer to the edge is then returned ;

• AddFlap creates a flap from a tuple *(side*, *edge-ptr*, *solid-ptr*, *plane-quadruple)* and adds it to the incident-flap list of the argument edge. Flaps of distinct solids, that are incident to a shared sub-edge and have identical *sides* and plane-quadruples, are merged into a unique flap. The solid-pointer of the flap is set to a particular value (for instance, ∅) to mark it "shared". Such flaps are detected by means of a *hash-table* that takes plane quadruples as search keys.

The superset BRep generation is done in three steps:

First, each cross-edge is created with four incident flaps, two from each solid, as follows:

```
for each cross-edge record (p, q, f, g)
{  /* Let Qf  and Qg be the face planes  */
    v1 := AddVertex (p) ;
    v2 := AddVertex (q) ;
    e  := AddEdge (v1, v2) ;
    AddFlap (-1, e, A, Qf) ;
    AddFlap (+1, e, A, Qf) ;
    AddFlap (-1, e, B, Qg) ;
    AddFlap (+1, e, B, Qg)
}
```

Next, the remaining superset edges are generated by splitting the original edges of A and B. This requires sorting in lexicographic order, by their coordinate triples, the splitting-point list of each original edge. The following treatment is applied to each solid, successively :

```
SplitSolid (A : Solid)
{   for each original edge e of A  do
    {   Sort the splitting-point list of e  ;
        /* Let (p[i])i=1, n be the ordered list */
        v1 := AddVertex (p[1]) ;
        for i := 1  to (n − 1)  do
        {  v2 := AddVertex (p[i+1]) ;
           h := AddEdge (v1, v2) ;
           for each original flap of A , incident to e do
           { /* the flap has a side value s, and belongs
                to some face f of A, that lies on Qf */
               AddFlap (h, s, A, Qf)
           }
           if  p[i] carries a non null face-pointer g then
           { /* h lies in the interior of a face g of B, that
                lies on Qg */
               AddFlap (-1, h, B, Qg) ;
               AddFlap (+1, h, B, Qg )
           }
           v1 := v2
        }
    }
}
```

IV.3. Classifying the superset elements

In this step, each superset flap is classified with respect to $A \cap B$, to determine whether it belongs or not to the boundary of $A \cap B$. A new field is added to the flap-record to store this membership information (coded *ON* or *OFF*). Before dealing with the classification we first give some necessary definitions :

IV.3.1. Definitions

Shared/Self edges **:** A superset edge will be said *shared* if it has at least one incident flap that belongs to A, and at least one incident flap that belongs to B. An edge which is not shared will be said *self*: its incident flaps all belong to the same solid, and only to it. For the sake of conciseness, flaps that are incident to an edge which is itself incident to a given vertex, will be said incident to this vertex. Similarly, a vertex will be said *self* if its incident flaps all belong to the same solid, and only to it.

Neighbouring flaps : Each flap of (say) A, incident to some edge e of A, may be assigned a *neighbour* at each vertex v of e. The neighbour is defined as the unique flap of A incident to v, that has the same plane-quadruple Q as the considered flap. In addition, no other flap, satisfying these conditions, exists "between" the flap and its neighbour: the two flaps enclose the interior or some original face of A, that lies on Q. Each flap has exactly two neighbours that are incident to the *left* and *right* vertex of the attached edge, respectively (Fig. 9).

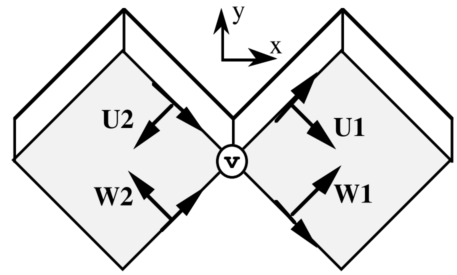

Figure 9. *Each pair* (U_i, W_i) *gives two neighbouring flaps, incident to the same vertex* v.

The candidate flaps (i.e. the possible neighbours) can be easily retrieved by scanning the incident-edge list of v, and retaining for each incident edge e, the incident flaps that belong to the same solid and have the same plane quadruple as the considered flap. In most cases, this leads to only two flaps, one of which is the desired neighbour. Otherwise, the neighbour is determined by sorting radially around v, the coplanar edges that are incident to v. Hoffmann and al. [7] proceed in a similar way.

***Remark* :** There is a convenient way for radial sorting, that specifies angles by means of rational numbers. Each vector $\boldsymbol{V}$ (with rational coordinates x and y) to be sorted, is assigned a rational number in the interval [0, 8[, acting as the polar angle of $\boldsymbol{V}$ with respect to the *x-axis*. This rational number has the following value: (y/x) if $0 \le y \le x$; $(2-x/y)$ if $0 \le y$ and $|x| \le y$; $(4+y/x)$ if $x \le 0$ and $|y| \le |x|$; $(6-x/y)$ if $y \le 0$ and $|x| \le |y|$; $(8+y/x)$ if $y < 0 \le x$ and $|y| \le x$.

IV.3.2. Classifying a superset flap

There are four methods to determine the membership information *(ON/OFF)* of a given flap. They are listed below in increasing computational cost :

- a *shared* flap is always marked *ON* : no computation is needed (recall that a shared flap has a null solid pointer);
- if a flap is incident to a *self* edge of (say) A, that has at least one vertex *outside* the bounding box of the other solid B, then the flap is marked *ON (OFF)* depending on whether the solid B is *unbounded (bounded)* : just test the "boundedness" field of B (see Section III.2) ;
- for a *shared* edge e, form the list of the *flap-vectors* attached to all the flaps incident to e (*flap-vectors* have been defined in Section III.2). Next, sort these vectors radially around e, within the plane Q that is orthogonal to e and that passes through the middle of e. Within this

plane, consecutive *flap-vectors* of the same solid enclose "sectors" (see Fig. 10) that are alternately in the interior or in the exterior of the solid (to decide, consider the outward normal vectors carried by the flaps).

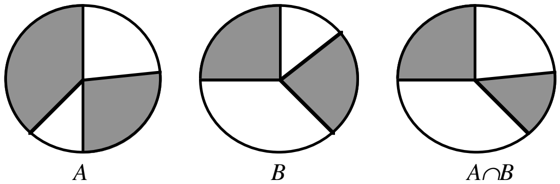

Figure 10. *Cross sections of A and B around a shared edge.*

Next, a Boolean intersection between the sectors of A and those of B, yields sectors of Q that have a constant classification with respect to $A \cap B$. Each flap incident to e is then marked *ON (OFF)* depending on whether the corresponding *flap-vector* delimits some sector which is in the *interior (exterior)* of $A \cap B$.

• it remains to deal with a *self* edge e of (say) A, that has no vertex *outside* the bounding box of the other solid B. In this case, there is no way to compute the membership information by local computations only : some interior point of e (e.g. the middle) need be classified with respect to B. Classically, this amounts to counting the number of times a random ray starting at this point, properly intersects (the interior) of the faces of B. Each flap incident to e is then marked *ON (OFF)* depending on whether the number of ray/face intersections is *odd (even)*, i.e. whether the test point is in the *interior (exterior)* of B .

IV.3.3. Propagating the membership information

There are three simple rules that allow propagating the membership information through the boundary of $A \cap B$:

• flaps that are incident to the same *self* edge have necessarily the same classification ;

• flaps that are incident to the same *self* vertex have necessarily the same classification ;

• two flaps that are neighbours (of each other) at some vertex have necessarily the same classification.

To compute the membership information of all flaps, we proceed as follows: while there still exist flaps not yet classified, select a flap whose classification is the most easily determined, compute the membership information, then propagate it by means of the three rules above. Propagation technique allows a substantial reduction in computation ; it is also exploited in [10] and [19].

Remark: To avoid a redundant computation of the neighbour of the same flap at the same vertex, one may use the so-called "attribute-function" technique. Two fields are added to the flap-record to store pointers to the *left* and *right* neighbours of the flap. Each time the neighbour function is called, it first checks whether the desired neighbour is already computed. If not, the neighbour is determined and stored in the corresponding field.

IV.4. Generating the BRep of $A \cap B$

To complete the algorithm, superset flaps that are marked *OFF* are deleted from the intermediate data structure. The remaining flaps (marled *ON*) are used to build the BRep of final solid $S = A \cap B$. In order to generate (for S) a data structure like the one of A or B (see Section III.2), we need to construct the maximal faces of S. This is done by simply grouping together flaps that have identical plane quadruples.

Finally, the resulting BRep is "cleaned" by deleting the possible "useless" edges the algorithm may have produced. Such edges are easily detected because they have exactly two incident flaps that belong to the same maximal face. These edges are deleted together with their incident flaps.

IV.5. Treating isolated vertices

For the sake of clarity, *isolated* vertices (announced in Sect. II.2) were deliberately omitted in the presentation of the algorithm. An *isolated* vertex of a solid A is the endpoint of some edge of A, that lies in the *interior* of some face f of A. Omitting such a vertex during the intersection of A with an other solid B may lead to an invalid result $A \cap B$. Suppose A is the non-manifold solid shown in Fig. 11, that has an *isolated* vertex v in a face f. Let g be a face of B (not represented), that transversally intersects f. During the intersection of f and g, a *cross-edge* [p, q] is created even though it contains the vertex v, leading to an invalid edge [p, q] for $A \cap B$ (see Sect. III.4).

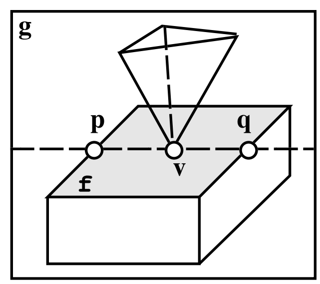

Figure 11. *An isolated vertex* v.

There is a simple method for treating *isolated* vertices. Assume that for each face f of each solid, we maintain a list $\Im$(f) for all *isolated* vertices in f (each being represented by a coordinate triple). Each time the intersection algorithm detects a *cross-edge* [p, q] between a face f of A and a transversal face g of B, it checks whether $\Im$(f) or $\Im$(g) contains some vertex v that lies on the intersection line L of the face planes, somewhere between p and q. If so, the cross-edge is stored as two valid *cross-edges* (p, v, f, g) and (v, p, f, g), then v is deleted from the corresponding list.

The necessary informations to maintain isolated vertex lists are available as transversal faces are intersected. Note that this "special" treatment requires only minor additions to the general algorithm and data structure.

IV.6. Evaluating the boundary of CSG solids

We have actually implemented a solid modeler based on the intersection algorithm above. It takes as input a CSG expression that combines simple and bounded polyhedral solids (like cubes, or polyhedral approximations of cylinders, spheres, cones, or tori) through regularized Boolean operators (∩, ∪, or −) and rigid motions.

To evaluate the whole CSG tree, we used the *incremental* evaluation strategy [17]. The well known De Morgan's laws make it possible to implement the difference and union operators in terms of the regularized intersection and complement.

Complementing a solid consists simply in : inverting the "boundedness" field of the solid ; inverting the orientation of each face plane of the solid (i.e. the signs of the four coefficients) ; and inverting the "side" value of each flap of each face of the solid.

V. A LAZY RATIONAL ARITHMETIC

In a first implementation we used a pure rational arithmetic supplied in [16], that has resulted in a very slow program. The reason stems from that a considerable amount of precise computation is performed even though the exact values are not used subsequently. More precisely, the straightforward use of a rational arithmetic has the following two inconvenients :

First, many numerical data (e.g. signed distances, determinants,...) are computed exactly to make geometric decisions based on a *sign* determination (e.g. testing whether a given point lies on a given plane,...). However, the computed value, itself, need not be known if we assume there is some way in which the *sign* may be safely induced. Karasick and al. [8] use intervals (of integers) to surround their determinants, so that an interval which does not contain zero yields the sign of the underlying determinant with no help of the exact value. Gangnet and al. [4] combine floating-point intervals and exact arithmetic.

Second, many geometric data (e.g. vertex coordinates, and plane coefficients) are computed and stored in data structures, even though they are "thrown" away at the end (e.g. for they are found "irrelevant" to the final result).

To take into account the two points above, we have developped a so-called "lazy"[1] rational arithmetic [2], that is based on the following evaluation scheme : *delay any precise computation until it becomes either useless or indispensable*. Roughly, each number is assigned a double representation : a surrounding *interval* (i.e. two floating-point numbers) that contains the (possibly unknown) exact value, and a *symbolic definition* that allows computing the exact value, when needed.

A definition is either a pure rational number (represented in some way), or it is an unevaluated expression which represents the *Sum*, the *Product*, the *Opposite*, or the *Reciprocal* of other lazy numbers. Strictly speaking, lazy numbers form a *Directed Acyclic Graph* (DAG) rather than a tree, since some nodes may be shared.

Each basic operation is generally done in constant time and space: it simply consists in creating a new DAG node that stores the operator, pointer(s) to the operand(s), and a correct interval that contains the result. The interval is obtained from those of the operands, through simple composition rules rooted from interval arithmetic [9].

In most cases, intervals are sufficient to compare lazy numbers. The exact value of a number is never computed, except in the following three cases: 1) we need to determine the sign of a lazy number whose interval contains zero ; or more generally, when we need to compare two lazy numbers with overlapping intervals ; 2) we need to compute the *reciprocal* of a lazy number whose interval contains zero ; and 3) we need to compute an ancestor (within the DAG) of the considered lazy number.

Such a lazy rational arithmetic is never more expensive than a pure rational one: in the worst case, all computations are performed exactly ; in the best case it performs no exact computation, at all. Anyway, only the necessary precise computations are done, without the algorithm having to foresee these computations. As result, the task of avoiding numerical errors (and thus, inconsistencies they cause) is transparently handled at a lower level, totally independent on the algorithm. The full details on the lazy arithmetic library may be found in [2].

VI. INTERFACING OTHER MODELERS

Our boundary evaluation method relies on error-free Boolean operations, that combine rational solid operands to produce rational solid results. Therefore, to interface with existing modelers, there must be ways to convert consistent rational descriptions of solids to consistent "floating-point" ones, and vice versa.

VI.1. From rational to floating-point solids

This problem is delicate: two distinct rational vertices that are close to each other may produce numerically indiscernible floating-point vertices. Thus, degenerate faces and edges may appear, that destroy the consistency of the BRep. The solution may be the so called "consolidation" devised by Segal and Sequin [18]. Roughly, it consists in reorganizing the BRep so that "too small features" are eliminated: any boundary element (i.e. face, edge, or vertex) that is within a small region around another element, must be either merged with this new element, or pulled apart to maintain a minimum separation from it.

VI.2. From floating-point to rational solids

Our solution consists in *triangulating* the faces of each *primitive* of the input CSG tree. The initial vertices (with floating-point coordinates) are "rounded" (i.e. forced) to the nearest vertices with rational coordinates. Each triangular facet lies on a unique plane whose equation may be computed (exactly) from the three rounded vertices. The resulting BRep is a rational approximation of the initial primitive BRep.

Sugihara and Iri [21] use the dual solution: they assume *trihedral* primitives so that each vertex is specified by three concurrent planes. This method requires smaller integers for the coordinates of new intersection points, but we found it is not convenient for traditional primitives like cones.

[1] The concept of *lazy arithmetic* is different from the *lazy evaluation* paradigm used by some functional langages.

Rational approximations of floating-point numbers may be obtained through one of the following three possible methods :

Using a discrete integral 3D-grid

The initial vertices are snapped to the nearest points of a regular, integral, 3D-grid whose spacing depends on the desired precision (as in [5] or [14]). This method always yields rational numbers that are bounded. Assume for simplicity, that the grid has a spacing of 1, and that the initial coordinates lie in the interval [0, G], for some big enough integer G. Each computed plane equation has coefficients (a, b, c, d) such that: $\mathrm{Max}\ (|a|, |b|, |c|) \leq 2G^2$ and $|d| \leq 6G^3$. Each computed intersection point has rational coordinates p/q such that : $|p| \leq 48G^6$ and $|q| \leq 144G^7$. These bounds do not depend on the size of the CSG tree. We found that for $G = 10^6$ (i.e. for a precision of 1mm in a universe of 1km^3), the biggest integer that may be encountered cannot exceed $144*10^{42}$, which is less than 2^{147}. In these conditions, a fixed length 256-bit integer arithmetic would be sufficient. However, we found more convenient to let open the range of the possible data by allowing *unbounded* integers, represented as lists of digits in some basis.

Using the continued fraction algorithm

The continued fraction algorithm allows any real number x to be represented by a sequence of rational numbers r_k, that converges to x. The sequence is finite if, and only if, x is a rational number. In practice, the expansion is stopped at some rank k that achieves some desired precision ε (i.e. when $|x - r_k| / |r_k| \leq \varepsilon$). The higher the precision is, the finer the rational approximation will be, but the bigger the size of the resulting rational number will be.

Using the binary floating-point format

Floating-point numbers are rational numbers encoded in the specific binary format of the computer at hand. Thus, it is always possible to translate this format to any other format chosen for rational numbers. In this way one may capture the whole precision of a floating-point number. However, this method yields very big rational numbers. On the other hand, one must keep in mind that floating-point numbers are just approximations of the real data.

VII. EXPERIMENTS AND RESULTS

We have implemented our solid modeler in C++, under Domain Os Ver. 10.4, on an HP/apollo Workstation (CPU 68040, 33 MHz and 16 Mo of RAM). For testing purposes, the modeler exists in three distinct versions : a pure floating-point version (F), a pure rational version (R), and a lazy version (L). The goal of this triple version is to estimate how fast is the lazy version compared with the rational version, and how big is the overhead for "laziness" as opposed to the standard floating-point computation. Performance evaluation involves several (possibly conflicting) parameters. Thus, we'll only focuse on two main parameters : the size of the modeled scene and the precision at which the initial floating-point coordinates are rounded to rational numbers (via the continued fraction algorithm CFA).

While the rational and lazy versions always run to completion and produce consistent results, the floating-point version often crashes when the scene contains "special" cases (e.g. horizontal or vertical faces, or very close ties). Sometimes, a slight perturbation of the input geometric data (e.g. a small rotation of the CSG tree) is sufficient to eliminate all degeneracies. Anyway, the expected behaviour of the lazy version is performing only the necessary exact computations (those that are needed to solve the "special" cases).

To compare the three versions, we have performed a serie of tests that consist in computing the intersection of a variable number `n` of cubes centered at the origin and randomly oriented in space, for different precisions. The Figure 12 shows the resulting objects for `n` = 2, 8 and 20. The tables of Fig. 13 (together with the charts of Fig. 14) summarize the execution time (in seconds) of each version, for `n` = 2, 4, ..., then 20, and for an increasing precision $\varepsilon = 10^{-3}$, 10^{-6}, 10^{-9}, then 10^{-12}. The Fig. 14 also includes charts to show the variation of the three possible time ratios (R/F, R/L and L/F).

Obviously, the execution time increases as the scene grows in size (Charts 14a, 14b, 14c). Moreover, increasing the precision considerably affects the rational version (Chart 14b). Rational numbers quickly grow in size as rational operations are performed (they reach 170 digits in their numerators or denominators, radix 32768). Asymptotically, the rational version is 1000 (resp. 3000, 5000, 9000) times slower than the floating-point version, for $\varepsilon = 10^{-3}$ (resp. 10^{-6}, 10^{-9},10^{-12}) (see Chart 14d).

In return, the precision does not affect the lazy version (superposed curves in Chart 14c). Since only a few exact operations are performed, the biggest rational numbers encountred do no exceed 7 digits. Asymptotically, the lazy version runs 100 (resp. 300, 600, 1200) times faster than the rational version, for $\varepsilon = 10^{-3}$ (resp. 10^{-6}, 10^{-9},10^{-12}) (see Chart 14e).

An interesting result is that the L/F time ratio, itself, does not depend on the precision (nearly superposed curves in Chart 14f). For small scenes ($n \leq 6$), the execution time in lazy version is dominated par a fixed cost (due to various initializations). Then, as `n` increases, the L/F time ratio asymptotically decreases towards some constant value (≈ 8), the fixed cost being compensated by the size of the modeled scene.

To complete, we have also performed the classical test described in [10] : consider the unit-cube A centered at the origin, whose sides are aligned with the coordinate axes, and an other cube B obtained by rotating A around each coordinate axis, by a small angle α. The modeler is executed to compute $A \cap B$ for different values of α. Polyhedral solid modelers that accommodate coplanar faces can handle an angle $\alpha = 0$ degrees as well as large angles, but as pointed out in [10], many commercial modelers fail when α drops below 2 degrees. More specifically, above a certain angle α_2, a correct result is obtained, and below a certain angle α_1 ($<\alpha_2$), A and B are so similar that the modeler cannot distinguish them (see also [7], [19], [20]).

Our lazy solid modeler never breaks (there is no critical angle α_2), while α_1 may be made as small as desired, and even zero if we assume the "exact" rounding method described in Section VI : the only limitation is due to the precision (always finite) of the initial floating-point vertex coordinates. We found that in order to make $\alpha_1 = 10^{-4}$, it is sufficient to take $\varepsilon = 10^{-6}$.

VIII. CONCLUSION

Boundary evaluation algorithms are delicate to implement. Tricky degenerate cases are indirect sources for numerical errors, and thus for topological inconsistencies. Despite a careful engineering design, commercial solid modelers often break or fail in producing consistent results. We hope the solutions proposed in this paper will help in reducing the burden of ensuring consistency in the solid modeling area. In particular, we think the concept of lazy arithmetic can broaden the application domain of exact arithmetics, especially in geometric algorithms.

ACKNOWLEDGEMENT

We wish to thank our colleagues P. Jaillon and J. M. Moreau, and also the anonymous referees for their comments and suggestions.

REFERENCES

1 Baumgart, B. *Winged Edge Polyhedron Representation*. Research Report CS-320, I. A. Lab., Stanford University, Oct. 1972.

2a Benouamer, M O, Jaillon, P, Michelucci, D and Moreau, J M. *A Lazy Exact Arithmetic.* Eleventh IEEE Symp. on Computer Arithmetic, June 30 - July 2 (1993), Windsor, Ontario, Canada.

2b Benouamer, M O, Jaillon, P, Michelucci, D and Moreau, J M. *Hashing Lazy Numbers.* Inter. Symp. on Scientific Computing, Computer Arithmetic and Validated Numerics, Sept. 26-29 (1993), Technical University of Vienna (Austria).

2c Benouamer, M O, Jaillon, P, Michelucci, D and Moreau, J M. *A Lazy Solution to Imprecision in Computational Geometry.* Proc. of the Fifth Canadian Conf. on Computational Geometry, Aug. 5-9 (1993), Ontario, Canada, pp. 73-78.

3 Edelsbrunner, H and Mücke, E P. *Simulation of Simplicity: A technique to Cope with Degenerate Cases in Geometric Algorithms*. Proc. of the 4th ACM Symp. on Computational Geometry (1988), pp. 118-133.

4 Gangnet, M and Van Thong, J M. *Robust Boolean Operations on 2D paths*. COMPUGRAPHICS'91, Vol 2, Harold P. Santo Ed., Sesimbra (Portugal), pp. 434-443.

5 Greene, D H and Yao, F. *Finite Resolution Computational Geometry*. 27th Symposium of the Foundations of Computer Science (1986), pp. 143-152.

6 Guibas, L, Salesin, D and Stolfi, J. *Epsilon Geometry: Building Robust Algorithms From Imprecise Computations*. Proc. of the Fifth ACM Symp. on Computational Geometry (1989), pp. 208-217.

7 Hoffmann, C M, Hopcroft, J E and Karasick, M S. *Robust Set Operations on Polyhedral Solids*. IEEE CG&A, 9(6), Nov. 1989, pp. 50-59.

8 Karasick, M, Lieber, D and Nakman, L R. *Efficient Delaunay Triangulation Using Rational Arithmetic*. IBM Research Report RC-14455, #64722, 3/8/89.

9 Kulisch, U W and Milanker, W L. *Computer arithmetic in Theory and Practice*, Academic Press, New York, 1981.

10 Laidlaw, D H, Trumbore, W B and Hugues, J F. *Constructive Solid Geometry for Polyhedral Objects*. Computer Graphics, 20(4), Aug. 1986, pp. 161-180.

11 Mäntylä, M and Sulonen, R. *GWB: A Solid Modeler with Euler Operators*. IEEE CG&A, 2(7), Sept. 1982, pp. 18-31.

12 Mäntylä, M and Tammimen, M. *Localised Set Operations for Solid Modeling*. Computer Graphics, 18(3), 1983, pp. 279-288.

13 Mehlhorn, K. *Data Structures and Algorithms 3: Multidimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.

14 Michelucci, D. *Les représentations par les frontières: quelques constructions; difficultés rencontrées*. Thèse de Doctorat, Nov. 1987, Ecole Supérieure des Mines de Saint-Etienne, France.

15 Milenkovic, V J. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD. Dissertation, CMU-CS-168, Carnegie-Mellon University, 1988.

16 Moreau, J M. *Hiérarchisation et facetisation de la représentation par segments d'un graphe planaire dans le cadre d'une arithmétique mixte*. Thèse de Doctorat, Novembre (1987), Ecole Supérieure des Mines de Saint-Etienne, France.

17 Requicha, A A G and Voelcker, H B. *Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms*. Proc. IEEE, 73(1), Jan. 1985, pp. 30-44.

18 Segal, M and Sequin, C H. *Consistent Calculations for Solid Modeling*. Proc. of the First ACM Symposium on Computational Geometry, 1985, pp. 29-38.

19 Segal, M and Sequin, C H. *Partitioning Polyhedral Objects into Non Intersecting Parts*. IEEE CG&A, 8 (1), 1988, pp. 53-67.

20 Segal, M. *Using Tolerances to Guarantee Valid Polyhedral Modeling Results*. Comp. Graphics, 24 (4), 1990, pp. 105-114.

21 Sugihara, K and Iri, M. *A Solid Modelling System Free From Topological Inconsistency*. Research Memorandum RMI 89-03, Dept. of Math. Engr. and Info. Physics, Tokyo University, Japan, 1989.

22 Tilove, R B. *Setmembership Classification: A Unified Approach to Geometric Intersection Problems*. IEEE Transactions on Computers, Vol. C-29, Number 10, Oct. 1980, pp. 874-883.

TESTS AND RESULTS

n = 2 n = 8 n = 20

Figure 12. Intersecting n cubes centered at the origin and randomly oriented in space

Figure 13. Execution time (in seconds) of each version.

a. Float **b**. Rational **c**. Lazy

d. Rational/Float **e**. Rational/Lazy **f**. Lazy/Float

Figure 14. Charts to show the variation of time and ratios as n varies.

Figure 15. Union, Intersection, and Difference of two groups of four prisms.

a **b**

Figure 16. **a**) Union of 5 cubes C_k inscribed in a regular dodecahedron. C_0 is centred at the origin, while C_k (k= 1,2,3,4) is obtained by rotating C_0 by $\alpha_k = 2k\pi/5$ around the axis (1, φ, 0), where $\varphi = (1+\sqrt{5})/2$ (the golden number). **b**) Same as in **a**, but each cube C_k is replaced by a cubic wireframe, by substracting three prisms.

Figure 17. A fractal object (the so-called Menger's sponge).

Figure 18. An architectural shape modeled as a CSG combination of prisms and cylinders