An Introduction to the Robustness Issue

D. Michelucci

École Nationale Supérieure des Mines de Saint-Étienne
158 Cours Fauriel, 42023 Saint-Étienne cedex 02, France
micheluc@emse.fr

Abstract

Rounding of floating point arithmetic causes failures or inconsistencies in geometric computations. For Computational Geometry, the only solution seems to be Exact Computing. CADCAM and Computer Graphics have investigated several approximate approachs.

Keywords: Robustness, Inaccuracy, Interval, Fuzzyness, Exactness, ϵ Heuristic, Lazyness.

1 The problem

Due to inaccuracy of floating point arithmetic, geometric algorithms can enter in infinite loops, take inconsistent decisions and crash, or produce inconsistent topologies which will confuse other methods [34, 11]. This lack of robustness especially occurs with the most sophisticated methods, typically from Computational Geometry in opposition to brute force ones, like ray casting; and with the most sophisticated data structures, typically BReps in opposition to voxel arrays or CSG trees. It is due to the fact that sophisticated methods propagate intermediate results (like $a < b$ and $b < c \Rightarrow a < c$) which are sometimes wrong due to inaccuracy; sophisticated data structures like BReps must fullfill integrity constraints (eg to be a planar graph), but are built step by step as a succession of geometric decisions, sometimes wrong due to inaccuracy. Even when Euler operators are used to ensure topological consistency, it is not enough to guarantee the consistency between answers to topological and geometric queries (does this intersection vertex geometrically lie on the surface it is topologically supposed to ?) and between geometric queries: for instance if points P_1, P_3, P_5 are aligned, and P_2, P_4, P_6 too, then points $P_1P_2 \cap P_4P_5$, $P_2P_3 \cap P_5P_6$, $P_3P_4 \cap P_6P_1$ must also be collinear (Pappus' Theorem): it is unlikely that this kind of property can be encoded in the data structure itself, and floating point arithmetic precludes its verification, since it already precludes verification of trivial identities like $\left(\sqrt{2}\right)^2 = 2$ or $a + b - a = b$.

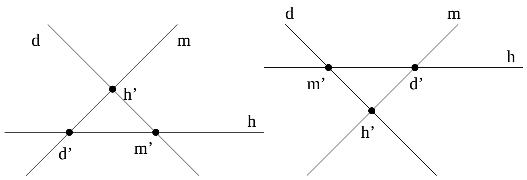

Figure 1: *The two generic configurations.*

To give a simple example of inconsistency, consider the three lines of Fig. 1. h is nearly horizontal, m climbs up and has slope about 45 degrees, d goes down and has slope about -45 degrees. Let the three intersection points be $h' = d \cap m$, $m' = d \cap h$, $d' = m \cap h$. Excluding the degenerate case for simplicity, there are two possible configurations, and in both h' abscissa is between d' and m' ones. With a small enough triangle, coordinates will differ only by their least significant bits corrupted by rounding; so *fp* arithmetic will sometimes produce non consistent triangles impossible to draw.

For a long time, programmers have hoped that some heuristics would be sufficient (section 2) against inaccuracy. Today Computational Geometers think that the only way to save CG methods from inaccuracy ravages is Exact Computation [6, 34] (section 3). Exact computation is not realistic in Computer Graphics and CADCAM; people prefer to account for inaccuracy at the modelling stage (section 4.1), or to use more reliable methods and data structures: it is the "Approximate Computation Paradigm" (section 4).

2 Heuristics

2.1 The ϵ trick

The most popular and the oldest trick is the ϵ heuristic, which declare equal numbers the relative difference of which is smaller than a prescribed threshold classically called ϵ. In practice, this trick works not so bad with the active understanding of users. But the fine tuning of ϵ is a never ending process since it introduces new inconsistencies; for instance equality transitivity is lost since one may find a, b, c such that $a =_\epsilon b, b =_\epsilon c$, but $a \neq_\epsilon c$, where $=_\epsilon$ stands equal up to ϵ.

2.2 Careful programming

Some computer scientists prefer to avoid the ϵ heuristic and have settled a set of tricks:
– Check first in the data structures before computing; for instance, before computing the power of a vertex relatively to some line (surface), verify first if the line (the surface) is topologically incident to the vertex from the involved data structures.
– Handle in a special way some particular cases, for instance the intersection point between a vertical line and an oblique one. In this case, assign the abscissa with the abscissa of the vertical line, not with the expression $\Delta x/\Delta$. Idem for special planes or surfaces.
– Never use several (algebraically equivalent) formulas for the same value, since floating point arithmetic would yield different results.
– The computation of $ab \cap cd$, $ab \cap dc$, $ba \cap cd$, $ba \cap dc$ (they are, for instance, segments in $2D$) generally give slightly different results. Before computing an intersection, it is worth systematically orienting segments, so that $a <_L b$[1] and $c <_L d$ and so on, by exchanging vertices in order to ensure: $ab \cap cd = ab \cap dc = ba \cap cd = ba \cap dc$.
– Use numerical *input* data rather than *derived* (thus corrupted) numerical data.
– Prefer non redundant data structures, to limit the probability of contradictions.

Quoting C. Hoffmann [11]: "*Conceptually we view these heuristics as attempts to reduce the logical interdependence of decisions that are based on numerical computations.*" M. Iri and K. Sugihara [14] have used this kind of approach for computing Voronoï's diagrams. They ensure that their program will never crash because of inaccuracy, that the resulting graph is correct when there is no numerical difficulty, and otherwise that the graph is connected with all vertices having degree 3, like a correct Voronoï's diagram. It is impressive. However remains a great problem: there is strictly no guaranty that another program, mathematically correct, and using this Voronoï's diagram as an input, will not crash.

These stratagems show wits, but they can only avoid the most obvious inconsistencies. Avoiding more convoluted ones (for instance the non respect of Pappus's theorem) and avoiding contradictions between several pieces of software written by different programmers using different conventions, notations and formulas, is an impossible task.

3 Exact computation

Exact Computation seems the only way for classical CG methods to work [6, 34]. Often, it only needs an exact rational arithmetic, sometimes the square root, and rarely a general exact algebraic arithmetic. Even when a rational arithmetic is sufficient, naively using a rational package is too slow, and people use filters, presented in 3.1 and 3.2. When rational arithmetics are not enough, CGers typically use gap arithmetics (see 3.3), rather than others solutions from Symbolic Algebraic Computation (D_5, Gröbner bases, resultants).

3.1 The LN library

The LN package of S. Fortune and C. van Wyk [9] proceed in two steps: First the program is pre-compiled and the minimum number of digits needed for the exact arithmetic (the longest integer generated by the algorithm, knowing the data range and the arithmetic expressions in the program) is determined. For each test in the program, *C++* code is automatically generated: to compute the test in standard *fp* arithmetic, using references to original data only; to test if *fp* value is greater than the maximum possible error for the expression; finally, to call the exact, long integer library to evalu-

[1] $a <_L b \Leftrightarrow a_x < b_x$ or $a_x = b_x$ and $a_y < b_y$.

ate the expression when the sign of the *fp* value is not reliable.

Second, the program is then compiled and linked with the exact library. Note that every test must be made with reference to original data. This permits a static (ie before running time) computation of the maximum possible error for each expression when evaluated in *fp* arithmetic; so the error bound has not to be computed at run time with intervals or whatever method. It speeds up execution, in a remarkable way, but it is not always very convenient for the user [4]; it forbids on-line and reentrant algorithms, in which computation depth is not *a priori* known, and it causes a proliferation of types: for instance input points and intersection points cannot be of the same type; this proliferation is a programmer's burden, and sometimes a compiler's one.

3.2 Lazy arithmetic

The lazy arithmetic computes with lazy rational numbers. A lazy rational number is first represented by an interval of two *fp* numbers, guaranteed to bracket the rational number, be it known (exactly evaluated) or not; and then by a symbolic definition, to recover the exact value of the underlying rational number, if need be. The definition is either a standard representation of a rational number (for example 2 arrays or lists of digits in some basis, for numerator and denominator), or the sum or the product of two other lazy numbers, or the reciprocal or opposite of another lazy number. Thus each lazy number is the root of a tree, whose nodes are binary (sum or product) or unary (opposite or reciprocal) operators, and whose leaves are usual rational numbers; actually, lazy numbers form a directed acyclic graph rather than a tree, since any node or leaf may be shared. Each operation is generally performed in constant time and space: a new cell is allocated for the number, its interval is computed from the intervals of the operand(s), and the definition field is filled (operation type, and pointers to the operand(s)). Intervals are more often than not sufficient during computations; the only cases in which they become insufficient and thus the definition has to be "evaluated" (ie with rational arithmetic) are: when one wants to compare two lazy numbers the intervals of which overlap, when one wants a lazy number sign or reciprocal the interval of which contains 0. A possible evaluation method is the natural and recursive one. Using such a lazy library is transparent: classical geometric methods need not to be modified. The lazy library also provides hashing of lazy numbers, using modular arithmetic [22].

The lazy version of a boolean solver between polyhedra is 10 or 15 times slower than the pure floating point version (when the latter succeeds) and 100 or 1000 times faster than the pure rational version.

Contrarily to LN, the lazy library is fully dynamic and so equally applies to on-line and reentrant algorithms: the computation depth needs not to be known *a priori*. In compensation, LN when usable should be (2 or 3 times) faster than the lazy library.

3.3 Gap arithmetic

The ϵ trick is based on a correct mathematical intuition which has given rise to gap arithmetics. The latters are exact arithmetics which exploit gap theorems, like Canny's one: *Let x_1, $x_2 \ldots x_n$ be the solutions of an algebraic system of n equations and n unknowns, having a finite number of solutions, with maximal total degree d, with relative integer coefficients smaller or equal to M in absolute value. Then, for all $i \in [1, n]$, either $x_i = 0$ or $|x_i| > \epsilon_c$ where*

$$\epsilon_c = \frac{1}{(3Md)^{nd^n}}$$

This theorem gives a way to prove *numerically* that a number is zero: compute a (guaranteed) interval containing it, with range smaller than ϵ_c. As soon as the interval does not contain 0, the number is clearly not 0 and its sign is known. Otherwise, if the interval contains 0 and has range less than ϵ_c, the number can only be 0. See [18] for other gap theorems or references. Note that gap thresholds are much smaller than the ones used in the ϵ heuristic, and that some bigfloat library is needed.

3.4 Exact algebraic arithmetic

Recently in CADCAM field, D. Manocha and some of his students [17] used an exact algebraic arithmetic to reliably compute intersections between algebraic parameterized surfaces of low degree (2-4: eg quadrics and torii).

They use Dixon's resultants for Elimination, with Milne's multivariate Sturm's sequences in order to locate roots. A number of improvements is needed to speed up this approach: for instance modular arithmetic speeds up the calculation of resultant coefficients, and intervals with rational endpoints that isolate roots are computed as lazily as possible. For the moment, it is doubtful that extensions to implicit surfaces and to higher standard degrees are possible, due to the intrinsic exponential cost of the involved symbolic computations.

Other exact algebraic arithmetic use Gröbner's bases [11], D_5 ideas [7, 10], resultants [32]. See [21] for a real quadratic arithmetic.

3.5 Pros and cons

Good reasons to use exact computations are: when usable (roughly, when an exact rational arithmetic is sufficient), it really solves the robustness problem, at the arithmetical level, ie classical CG methods do not need to be modified. It is even possible to remove degeneracies (another burden for the programmer) at this arithmetic level, with symbolic, infinitely small, perturbation. It opens interesting new problems, like exactly computing signs of determinants [3].

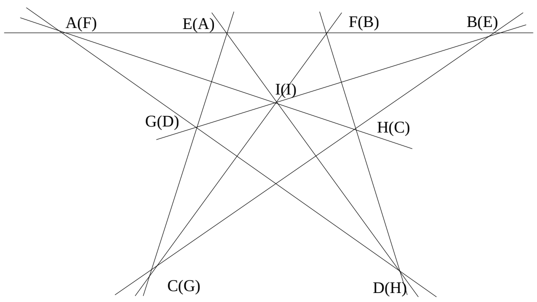

Figure 2: *Place 9 points $A, B \dots I$ so that only the following subsets (and all couples) are collinear: ABEF, ADG, AHI, BCH, BGI, CFI, DEI, DFH. This configuration is not realizable in $\mathbb{Q}^2$; it is in $\mathbb{Q}[\sqrt{5}]^2$.*

However, maybe the true reason is: it postpones heartbreaking reappraisals. Exact computations have serious limitations: algebraic arithmetics are very slow, relatively to *fp* ones, but intersection between curved lines or surfaces, or just rotations, introduce irrational numbers; moreover some configurations (Fig. 2, from [11]) need algebraic numbers to be exactly represented; last, incremental modifications of shapes, or on-line creations (use this intersection point between two circles as the centre of a new circle) increase algebraic degrees; geometric rounding becomes sooner or latter unavoidable to break the exponential degree growth, and also to communicate with the outside *fp* world. But rounding polygons and polyhedra without introducing self intersections is NP-complete [23]; Fortune's solution [8] is to accept self-intersecting polyhedra. Admittedly, but it means classical CG methods cannot be used...

4 Approximate approachs

CADCAM and Computer Graphics have tried several approximate approachs, I will mention: fuzzy boundaries in section 4.1, CSG, interval analysis and recursive space subdivision in section 4.2, CSG and marching methods in 4.3, ray tracing and ray representations in 4.4, discretization in section 4.5.

4.1 Fuzzy boundaries

The ϵ heuristic loses the order transitivity (it is possible to have $a =_\epsilon b$, $b =_\epsilon c$ and $a \neq_\epsilon c$), so inconsistencies remain possible. In such a case, a solution is to give up the distinction between a, b and c, and to merge them into another larger entity, actually an interval. The example in Fig. 1 will become something like in Fig. 3.

This approach has been investigated in solid modelling by M. Segal [27], by D. Jackson [15], by Patrikalakis's team [13]. In $3D$, geometric elements (vertices, edges or arcs, surfaces) are

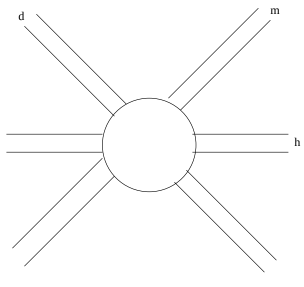

Figure 3: *Three lines with their halos, incident to a fuzzy point (the circle).*

surrounded by a thin halo of imprecision; two distinct and not adjacent elements must not have overlapping halos. During (typically) the computation of some boolean set operation, two elements the halos of which overlap must be cut or merged to restore the data structure consistency.

One can notice that two close but non overlapping entities have to be merged when a third entity that overlaps both former ones is introduced. One can deplore this information loss (the distinction between the first two entities has been lost though they are not modified), and fear that existing geometric algorithms will not spontaneously withstand such a non monotonic logic. But this is the spirit of this approach.

The main advantages of this approach are that it applies not only to "linear" problems but also to algebraic ones, and that it does not rely on an exact arithmetic; so it is fast. Moreover, it is intuitive. Finally, it can handle inaccurate data from sensors, and machining tolerances, in a natural way: up to now, this is the only approach that can represent fuzzy data.

When we want to know if the halos of two geometric entities overlap, their distance can be computed in several but algebraically equivalent ways; with a first formula, one may find that the elements do not overlap, but they will with another formula. Thus it is not clear for the moment that this approach is completely free of contradictions and that it definitively solves the robustness problem. Patrikalakis et al [13] do not claim that it does: "*With the interval representations of objects, topological violations due to numerical perturbation of fp arithmetic can often be avoided.*" Not always? The non monotonicity of these methods does not facilitate their proof and study, too.

4.2 CSG, interval, subdivision

Interval Analysis [16] can compute conservative (and rather accurate [13, 24]) bounds for a function range on an interval. When the interval for $f(B)$ where $B = [x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$ does not contain 0, one knows whether the box B is inside or outside the primitive object $f(x, y, z) \geq 0$. When the box is cut by the boundary: $f(x, y, z) = 0$, other sophisticated tests [28] from Interval Analysis can detect if the surface is simple enough in the box, for instance if one coordinate is an implicit function of the two other coordinates. It is also possible to detect if a box contains a single intersection curve between two surfaces, simple enough, or a regular intersection point between three surfaces. In a cell containing a single surface (respectively a single intersection curve between two surfaces), it is also possible to bracket it between two (respectively four) planes.

Otherwise, but if the box is too small according to an *a priori* threshold, the box is divided in 2 or 8 depending on the implementations, and the sub-boxes are studied the same way. Filiations between boxes may be stored in an octree. Such a method find boxes strictly inside CSG object, strictly outside, cut by a boundary in a simple way, or residual. Such residual boxes have smaller size than the prescribed threshold, and they usually contain or are very close to singularities or near-singularities. The robustness of this method may be obvious. SVLIS modeller [2] uses such a method. These methods can be used beyond $\mathbb{R}^3$: this "dimensionality paradigm" (the name is due to C.M. Hoffmann [12]) has been exploited by J. Woodwark for Feature Recognition, by K.D. Wise and A. Bowyer for Spatial Planning [33], by C.M. Hoffmann for Surface Interrogations [12]. Unfortunately, it seems difficult to account for free form surfaces in this framework, more fitted for CSG.

4.3 CSG and marching methods

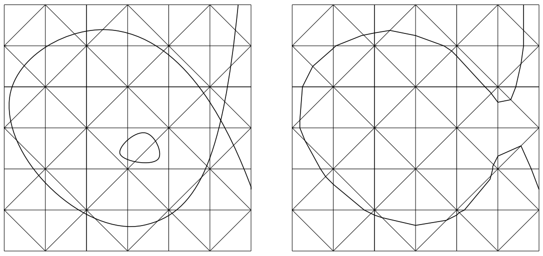

Figure 4: *A 2D curve and its piecewise linear approximation.*

To approximately triangulate objects defined by CSG trees within a given tolerance μ [25, 31], the space $\mathbb{R}^3$ is first partitioned with a regular cubic lattice, sided μ. Each cube is then partitioned into (5 or 6) tetrahedra; for all vertices $v = (x, y, z)$ of the lattice, the value of CSG tree at v is computed: for a primitive described by an inequality $f(x, y, z) < 0$, it is $f(v)$; for nodes $A \cap B$ and $A \cup B$, it is respectively $max(A(v), B(v))$ and $min(A(v), B(v))$ where $A(v)$ and $B(v)$ recursively stand for the value of A and B CSG trees in point v. The object surface cut a given tetrahedron when the values in the 4 vertices have opposite signs. These 4 values define, by linear interpolation, a unique linear map $l(x, y, z)$ from $\mathbb{R}^3$ to $\mathbb{R}$, and the plane $l(x, y, z) = 0$ is considered as a good enough approximation of the object contour inside the tetrahedron: it gives a triangle or a quadrilateral. The same is done for all tetrahedra. This technique is illustrated in $2D$ in Fig. 4. Marching methods are not sensitive to inaccuracy: in the worst cases, a vertex value is close to 0, and *fp* evaluations may yield a wrong sign for the value, but the only and immaterial consequence will be to move the approximation surface a little.

The true object topology and the one of its linear piecewise approximation may be different. Small components, with size less than the threshold, can be missed. In the vicinity of singularities and quasi-singularities of the true object boundary, the approximation remains manifold. This filtering can be considered as an advantage, a simplification. Geometrically (in opposition to topologically), the object and its approximation are close, up to μ.

Of course, it is faster to not consider all lattice cells, eg by some interval computations [5, 30] or by using continuity: once a starting tetrahedron crossed by the surface is known, the sides by which the contour surface leaves the tetrahedron are easily computed and the contour surface is then followed in the neighboring tetrahedron. It is also possible to better approximate the intersection curve between two surfaces in a cell. All the variants and optimizations are beyond the scope of this article, the main thing being marching methods reliability is preserved. Thus an approximate BRep (and all its precious informations) can be obtained from a CSG tree, without having to perform boolean set operations on BReps, which is a very unreliable process. Unfortunately again, free form surfaces do not naturally enter this framework.

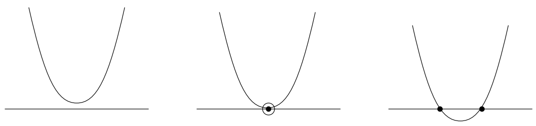

Figure 5: *Three indiscernible cases.*

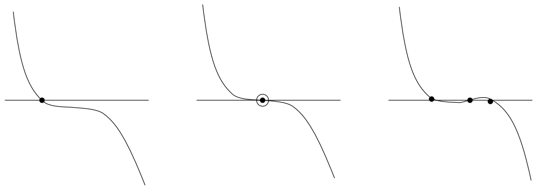

Figure 6: *Three indiscernible cases.*

4.4 Ray tracing and RayReps

Ray Casting is a very robust method. The difficult part is the numeric resolution of algebraic equations, like $F(t) = 0$, by interval analysis [16] or whatever numerical methods. Obviously *fp* and interval arithmetics cannot reliably decide in some ambiguous intervals: for instance they cannot distinct between the three cases in Fig. 5. Idem for the three cases in Fig. 6. However, the main thing is not to make a mistake on the parity of the number of roots in such ambiguous intervals, that is to say not to confuse a case in Fig. 5 (even parity) with one in Fig. 6 (odd parity). It is easily achieved. Assuming the parity is correct, mistakes have immaterial consequences on the final picture since they occur only when the ray is tangent or almost tangent to a surface. Thus the only effect is to move slightly and locally the object outline. Useless to indicate, a ray tracer never crashes due to these numerical errors, and mistakes are not propagated from pixels to pixels. This robustness against errors contrasts with the CG methods behaviour, or boolean operations between BReps.

Figure 7 shows an extreme (and rather artificial) case, where an ellipsoid is so thin it can be missed by rays. The classical solution is to change the modelling, and to use a disk (possibly carrying some thickness function) rather than an ellipsoid.

Ray tracing gives rise to the "ray representation" (rayreps for short): the object is sampled by an array of parallel lines. They become fashionable data structures in CADCAM [19, 20, 26, 1] due to their simplicity, versa-

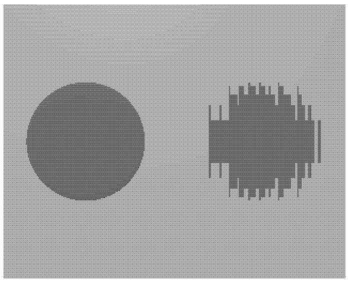

Figure 7: *Two ray-traced ellipsoids, radius* 1, *thickness* 10^{-5} *and* 10^{-7}.

tility and robustness. A rayrep can be computed by *any visualization method*: ray tracing but also the well-known Z-buffer method, or by merging two other rayreps with the same family of lines. It is possible to compute this way boolean operations between two rayreps (which have possibly been computed with different methods). Finally, to account for a new kind of geometric object, it suffices to implement the corresponding visualization routine.

An inconvenient of rayreps is anisotropy: surfaces parallel or nearly parallel to ray directions are less sampled than the ones perpendicular or nearly perpendicular to the ray direction. The obvious solution is to use a triple rayrep, ie three rayreps with three orthogonal directions, like Ox, Oy and Oz. A triple rayrep [1] induces a regular cubic lattice in which a marching method can then built an approximate triangulation of the boundary. Here again, an approximate BRep can be safely obtained from a CSG tree, without unreliable computations of boolean operations over BReps. Moreover, triple rayreps also treat sculptured solids (the boundary of which is made of free form surfaces) in a very natural way.

4.5 Discretization

Boundary representations are basically used to "evaluate" more or less accurately the boundary of a *CSG* object. It is the standard and historical way. Discretization is another solution: the space is represented by a $3D$ array of points, ie "voxels". This discrete representation makes trivial the most frequent geometric problems (estimating mass properties, interference detection, boolean operation, etc) and it virtually removes the inaccuracy problem. Nowadays, Computer Tomography and Magnetic Resonance Imaging make it possible to acquire such image data in $3D$. At the other end, from such a voxel-based representation, Rapid Prototyping [29] can produce real tactile plastic prototypes for manufacturers, chemists or biologists with "printing in $3D$", ie with stereolithography. Moreover, at this level of precision, the voxel-based representation is also the most precise one: this is in contrast with the not so old reluctance of some theorists for this discrete representation, which they considered as a trivial and very rough approximation of "exact" models. Last, the voxel-based representation is always the simplest one, obviously.

It is worth comparing the history of space representation with the one of pictures. In the beginning of Computer Graphics and CAD-CAM, more than twenty years ago, pictures were usually not represented by discrete representations, ie $2D$ arrays of pixels, but by BReps, because discrete representations were too cumbersome at this time, and available devices only provided wire frame display for which BReps are best suited. Related algorithms, for removing hidden parts for instance, already had trouble with inaccuracy. Nowadays, pictures are represented by discrete representations, and everybody has forgotten these algorithms and their inaccuracy problems. One can wonder if, similarly, the time has not come for discrete representations of space to supplant boundary representations of solids, and to remove the inaccuracy problem in geometric computations.

5 Conclusion

Unrobustness of geometric computations is still an open issue. Today CGers investigate Exact Computing: fortunately, exact arithmetics on integer or rational numbers are very often sufficient for CG. They are not for CAD-CAM, and algebraic arithmetics are too expensive. Thus people prefer approximated approachs which can be classified in two trends:
– use some exact "reference description" (eg CSG-like, or feature-based) and evaluate it when needed, up to some prescribed accuracy, with some reliable method from section 4.2, 4.3, 4.4, 4.5.
– account for inaccuracy in the geometric

model itself, typically maintain some fuzzy or interval BRep (section 4.1).

Unfortunately, lack of space prevents a more accurate comparison.

References

[1] M.O. Benouamer and D. Michelucci. Bridging the Gap between CSG and Brep via a Triple Ray Representation. In *Solid Modeling*, 1997.

[2] A. Bowyer. *SVLIS – Introduction and User Manual*. Information Geometers Ltd, 1995.

[3] H. Brönnimann, C. Burnikel, and S. Pion. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Symp. on CG*, 165–174, 1998.

[4] J.D. Chang and V. Milenkovic. An Experiment Using LN for Exact Geometry Computations. In *Canad. Conf. on CG*, 67–72, 1993.

[5] L.H. de Figueiredo and J. Stolfi. Adaptive Enumeration of Implicit Surfaces with Affine Arithmetic. In *Eurographics Workshop on Implicit Surfaces*, 161–170. 1995.

[6] T. Dubé and C.K. Yap. The Exact Computation Paradigm. In World Scientific Press, *Computing in Euclidean Geometry*, 1995.

[7] D. Duval. Handling Algebraic Numbers in Computer Algebra. In *ISSAC'89*, 1989.

[8] S. Fortune. Polyhedral Modelling with Exact Arithmetic. In *Solid Modeling*, 225–233, 1995.

[9] S. Fortune and C. Van Wyk. Efficient Exact Arithmetic for Comp. Geometry. In *Symp. on CG*, 163–172, 1993.

[10] T. Gomez-Diaz. *Quelques applications de l'évaluation dynamique*. PhD thesis, Université de Limoges, 1994.

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

[12] C.M. Hoffmann. A Dimensionality Paradigm for Surface Interrogations. *CAGD*, 7:517–532, 1990.

[13] C.-Y. Hu, N. Patrikalakis, and X. Ye. Robust Interval Solid Modelling. *CAD*, 28(10):807–817, 819–830, 1996.

[14] M. Iri and K. Sugihara. Construction of the Voronoi Diagram for One Million Generators in Single-precision Arithmetic. In *Canad. Conf. on CG*, 1989.

[15] D. Jackson. Boundary Representation Modelling with Local Tolerances. In *Solid Modeling*, 247–253, 1995.

[16] R.B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer, 1996.

[17] J. Keyser, S. Krishnan, and D. Manocha. Efficient Brep Generation of Low Degree Sculptured Solids using Exact Arithmetic. In *Solid Modeling*, 1997.

[18] K. Mehlhorn, C. Burnikel, R. Fleischer, and S. Schirra. A Strong and Easily Computable Separation Bound In *SODA97*, 702–709, 1997.

[19] J. Menon, R.J. Marisa, and J. Zagajac. More Powerful Solid Modeling through Ray Representations. *IEEE Comp. Grap. & App.*, 14(3):22–35, 1994.

[20] Jai Menon and Herbert Voelcker. On the Completeness and Conversion of Ray Representations of Arbitrary Solids. In *Solid Modeling*, 175–186, 1995.

[21] D. Michelucci. A Quadratic non Standard Arithmetic. In *Canad. Conf. on CG*, 1997.

[22] D. Michelucci and J-M. Moreau. Lazy Arithmetic. *IEEE Tran. on Comp.*, 46(9):961–975, 1997.

[23] V.J. Milenkovic and L.R. Nackmann. Finding Compact Coordinate Representations for Polygons and Polyhedra. *IBM J. of Research & Development*, 34(5):753–769, 1990.

[24] J-M. Muller and M. Daumas. *Qualité des calculs sur ordinateur* Masson, 1997.

[25] R.M. Persiano and A. Apolinário. Boundary Evaluation of CSG Models by Adaptative Triangulation. In *CSG 94*, Information Geometers Ltd, 1994.

[26] M.G. Prisant. Application of the Ray-Representation to Problems of Protein Structure and Function. In *CSG 96*, Information Geometers Ltd, 33–47, 1996.

[27] M. Segal. Using Tolerances to Guarantee Valid Polyhedral Modeling Results. *SIGGRAPH '90*, 24(4):105–114, 1990.

[28] J.M. Snyder. Interval Analysis for Computer Graphics. *Comp. Grap.*, 26(2):121–130, 1992.

[29] P. Stucki, J. Bresenham, and R. Earnshaw. Computer Graphics in Rapid Prototyping Technology. *IEEE Comp. Grap. & App.*, 15(6):17–19, 1995.

[30] G. Taubin. An Accurate Algorithm for Rasterizing Algebraic Curves. In *Solid Modeling*, 221–230, 1993.

[31] R.F. Tobler, T.M. Galla, and W. Purgatofer. ACSGM–an Adaptative CSG Meshing Algorithm. In *CSG 96*, Information Geometers Ltd, 17–31, 1996.

[32] Chionh Eng Wee and Ronald N. Goldman. Elimination and Resultants. *IEEE Comp. Grap. & App.*, 69–77, 1995.

[33] K.D. Wise and A. Bowyer. Using CSG Models to Map where Things Can and Cannot Go. In *CSG 96*, Information Geometers Ltd, 359–376, 1996.

[34] Chee K. Yap. Robust Geometric Computation. In *Handbook in CG*. CRC Press, 1997.