Bridging the Gap between CSG and Brep via a Triple Ray Representation

M.O. Benouamer and D. Michelucci
École des Mines, F42023 Saint-Étienne 02
micheluc@emse.fr

Abstract

Computing intersections between algebraic surfaces is an essential issue for Brep-based modellers, and a very difficult one. The more often, existing methods are not reliable, and reliable ones are hairy. We think there is another and simple-minded way which avoids this problem without loss of practicalities. The key idea is computing a *triple ray representation* by zbuffer, raytracing or whatever, and then using the popular *marching cubes* algorithm with some local improvements.

1 The gap between CSG and Brep

Breps [Hof89] describe solid objects by their boundary: surface patches, edges and vertices with their connectivity relations. They typically use free-form patches, carefully sewn together to form the consistent boundary of a solid which is then called a free-form (or sculptured) object. The high geometric coverage of free-form surfaces and their design flexibility are very appealing. In the other hand, Boolean operations on solid objects are an essential practicality for end users. Unfortunately, performing Boolean operations on Breps involve computing the intersection between algebraic surfaces, which is a very difficult task. Existing methods are often not reliable, and when they are, they are anyway exceedingly complicated: see [Pat93, KM96, HPY96].

The CSG model [Hof89] represent solid objects by a tree whose nodes carry Boolean operators and leaves carry algebraic half-spaces (algebraic inequalities: $f(x, y, z) \leq 0$). In contrast to Breps, the CSG representation does not suffer from reliability problems, and the surface to surface intersection problem is not a crucial issue. The raytracing method permits to visualize CSG objects and to convert them to ray representations (rayrep for short). The recursive space subdivision method permits to evaluate (*ie* to voxelize, or tesselate) them as in the SVLIS modeller [Bow95] or in Taubin's method [Tau93]. As long as a CSG modeller does not rely on tesselation, the latter can even be locally inconsistent without affecting the modeller. Note the *divide and conquer* approach basically relies on the possibility of quickly and simply classifying a point with respect to an algebraic half-space (by evaluating and testing the sign of the corresponding formula $f(x, y, z)$). It is then possible to compute, by an *interval arithmetic* (or some variant), ranges of the function f for boxes (a box is a point whose coordinates are intervals): a box B is classified *inside* if $f(B) < 0$ and *outside* if $f(B) > 0$. Otherwise the box is subdivided (into 2 or 8 smaller ones, according to implementations). Such a classification test is not available for free-form objects.

Unfortunately CSG does not support the full range of free-form objects. Several attempts have been made to combine appeals of CSG and Brep:

- Using *soft objects* is mainly restricted to the Animation field for the moment [IS'95].
- In the CAD/CAM field, J. Menon & B. Guo [MG96] use a restricted set of free-form surfaces, with a low degree implicit form (2 or 3): each free-form patch is assigned a companion tetrahedron which contains the patch, and whose vertices are, in some way, its *control points*. These tetrahedra permit to edit patches in an intuitive and interactive way. This modeller allows a *bilateral* CSG/Brep conversion.
- A. Pasko & V. Adzhiev & A. Sourin & V. Savchenko [PASS93] describe the interior of all geometric objects: algebraic half-spaces, Boolean operations, sweeps, some kind of deformations and blends, free-form volumes (sometimes called cuboids) [MPS96]..., by a semi-algebraic inequality $f(x, y, z) \leq 0$.

None of the previous solutions fully integrate in the CSG model all the free-form objects used in Brep-based modellers.

- A last approach combines CSG and Brep in that free-form primitives are accepted at leaves of the CSG tree. However, the simplicity of the pure-CSG scheme is lost: these modellers face the surface to surface intersection and the robustness problems. Recent works illustrating this tendency are due to S. Krishnan & D. Manocha [KM96], and to C.-Y. Hu & N. Patrikalakis & X. Ye [HPY96]. No doubt for us that these modellers are masterpieces, *tours de force* of geometric computing. But they are too much complicated. Moreover, they do not cover all the possible cases: sweeps (occurring for instance in NC-milling), blends or Minkowski sums.

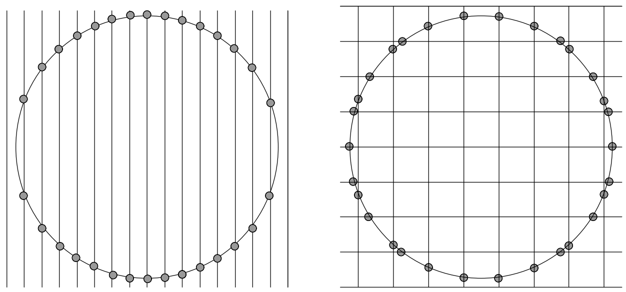


Figure 1: *Left: Sampling only along one direction misses curve portions. Right: this drawback is removed when sampling along two directions, without increasing the number of rays.*

2 Bridging the gap between CSG and Brep

2.1 The principle

To summarize the CSG/Brep dilemma: either we use CSG-based modellers, but we cannot enjoy free-form primitives, or we use Brep-based modellers to enjoy free-form surfaces and primitives, but we then face the fearful surface to surface intersection problem. How to solve this dilemma?

The approach we promote is simple-minded, general and easy to implement. Solid objects are represented by CSG trees including classical Boolean operators and primitives (algebraic half-spaces), extended to:

• objects obtained by sweeping any primitive (actually any CSG node or leaf) whose motion is defined by some function $t \rightarrow M(t)$ where t is in $[0,1]$ and $M(t)$ is any *invertible* 4x4 matrix,

• free-form primitives, described by a collection of free-form surface patches, that is guaranteed to form a consistent boundary of a solid object. The (u,v)-parameter domain of a patch is the simple, initial one: either the square $[0,1]^2$ for *tensor product* patches, or the triangle: $u \in [0,1]$ and $v \in [0, 1-u]$. *Trimmed patches* could be considered, but they are not essential in our context: since free-form objects with trimmed patches are Boolean combinations of free-form objects with untrimmed patches (*ie* with simple parameter domains), it suffices to keep them under the corresponding CSG form.

So we opt for the classical CSG model, but accept free-form primitives and sweeps as possible CSG primitives.

Next, we compute three separate orthogonal ray representations where rays are parallel to X-axis for the first, Y-axis for the second, and Z-axis for the third. Such a triple rayrep (3rayrep for short) is precisely what is needed to bridge the gap between CSG and Brep, since it allows simply computing a polyhedral approximation for the boundary of the CSG solid, via some variant of the popular "marching cubes" algorithm (MC for short) [LC87].

To summarize: sample, then reconstruct.

2.2 Single rayrep versus triple rayrep

While J. Menon & R.J. Marisa & J. Zagajac [MMZ94] and J. Menon & H.B. Voelcker [MV95] promote the use of a single rayrep, our approach is based on a *triple* rayrep. In our implementation of an NC-milling simulation module, we have first used a single rayrep. As result, we met problems with surface pieces which happen to be parallel or nearly parallel to rays or with sharp corners lying between rays. In such a case, the first solution that comes in mind is to increase the resolution of the rayrep (*ie* to decrease the ray-grid step), but we found that a 3rayrep with $3 \times n^2$ rays gives better results than a single rayrep with $(\lceil\sqrt{3}\rceil n)^2$ rays. A visual inspection of the produced Brep is our first argument. Moreover, to quote [MMZ94]: "The discretization of the boundary of the object A is sensitive to the ray direction, since ray intercepts of "on" segments or nearly "on" segments provide a poor sampling of ∂A...".

We think a 3rayrep is much less ray-direction sensitive than a single rayrep: Given the same total number of rays, ray hits are likely to be better distributed over the surface in a 3rayrep than in a single one (see Figure 1). Finally the 3rayrep is more convenient as an input for the MC algorithm.

2.3 Previous works

The concept of *multiple* rayreps is directly inspired and precursed by J. Menon & R.J. Marisa & J. Zagajac [MMZ94], and by J. Menon & H.B. Voelcker [MV95] who have already pointed out the appeals of (enhanced) rayreps. However the latter paper focuses on the *completeness* of the rayrep, not on its particular use to bridge the gap between CSG and Breps, nor on that it is a simple way to avoid the robustness problems encountered with Boolean operations on Breps. In [MMZ94], the authors suggest in passing (page 29) the use of a multiple rayrep, but they actually use a single rayrep for evaluating sweeps and Minkowski sums. Though they do not treat free-form surfaces and primitives (except those in J. Menon & B. Guo's format [MG96], which have a low degree implicit form and are easily raytraced), and do not resort to the zbuffer and MC methods, our work is clearly in their wake.

More recently, R.F. Tobler & T.M. Galla & W. Purgatofer [TGP96] have used raytracing and MC methods for meshing CSG trees with implicit surfaces at leaves. However, they do not consider free-form surfaces or primitives, nor sweeps.

2.4 Advantages of our approach

• The difficult and unreliable process of surface to surface intersection (necessary for direct CSG evaluation) is no longer needed. It is replaced by easy-to-implement, reliable and classical tools: zbuffer, raytracing, rayrep merging and marching cubes. We already have all we need.

• Actually, any "raytracable" or "zbufferable" object is an acceptable primitive for our extended CSG model. For instance, $3D$ data images (from Tomography, Scannography or whatever) may also be accepted as primitives.

• For the modeller to support a new kind of primitive, it suffices to provide the corresponding routine for zbuffering or raytracing it (and, if possible, a routine for computing a bounding box). Note if the modeller handles n types of objects, only n ray/object intersection routines are needed, instead of the $n(n-1)/2$ possible object/object intersection routines. Thus extensions are easy.

• An enhanced rayrep or 3rayrep can even be *complete* if the sampling is sufficient, as already pointed out by J. Menon & H.B. Voelcker [MV95]: for instance, a polynomial of degree d is completely defined by $(d+1)$ points, and a limited bandwidth signal can be faithfully reconstructed from regularly spaced samples when the Nyquist condition is fulfilled.

• Since a faceted Brep is generated, it may be viewed from any viewpoint, in contrast to T. Van Hook [Van86] who rather generates a viewpoint dependent image.

• For well-formed objects, this Brep can even be the *exact* one (see 4.3) if the sampling is sufficient, and if some local refinements (like those devised in [BF95, TGP96]) are performed.

• Even when completeness is not achieved in a 3rayrep, it is always simple and consistent, in contrast to Breps.

• As long as a CSG-based modeller does not rely on the Brep (for example for set-member classification or Boolean operations), the Brep may even be locally inconsistent: for instance, we can ignore the popular "ambiguous" cases of the MC algorithm, possibly leaving holes in the Brep, or displaying ambiguous cubes with some special color to warn the user. This will not pose any reliability problem: This is in contrast to the fragility of Brep-based modellers, where such casualness will invariably lead to failures!

• The 3rayrep may be seen as a virtual *voxmap* (*ie* a matrix of voxels) which is more accurate and more compact than a traditional one.

• Finally, hardware implementations of the zbuffer method are available. On the other hand, research is in progress for implementing in hardware the raytracing method as well, for instance the Ray Casting Engine (RCE) handles quadratic half-spaces [MMZ94]. Thus interactivity and even real-time should be achievable in the near future.

3 Computing a ray representation

3.1 The rayrep data structure

Fundamentally, a rayrep is a matrix of *dexel* lists. Each list represents the intersection of a given ray with the CSG object, sorted by increasing depth values. Each dexel stores: the *entering* hit, the *exiting* hit, and the *material* present in the dexel (possibly none). Storing the materials allows heterogeneous objects with multiple layers of distinct material. Each hit corresponds to an intersection point between the ray and the boundary of some primitive. The hit data structure stores: the depth value (*ie* the abscissa along the ray of the intersection point), a pointer to the intersected surface, and additional fields like: (u, v)-parameters in case of free-form surfaces, or application dependent informations like tool movement identifiers needed for NC-milling simulation and verification. These informations are sufficient to recover the coordinates of the hit point and its surface normal, when needed (one may prefer to explicitly store normals in data structures).

There are of course several ways to implement these data structures in the computer.

The dexel entity was first introduced by T. Van Hook for real-time shaded NC-milling display [Van86]. It is also used by T. Saito & T. Takahashi [ST91] in their *G-buffer* method.

3.2 Available methods

To build a rayrep one can use any of the following methods, and freely combine them according to the nature of the geometric objects at hand:

- Raycasting.
- Zbuffer.
- Rayrep merging, to trivially perform Boolean operations on two rayreps with the same ray-grid [MMZ94].
- Recursive Space Subdivision [Sny92, Tau94, Bow95]: This method is especially well suited for primitives whose interior is expressed by an available implicit formulation: $f(x, y, z) \leq 0$, and for CSG trees based on such primitives.
- Marching cubes: Of course this method not only gives a rayrep, but also a faceted Brep! Again, it is more suited for implicit primitives and induced CSG trees, and for 3D image data.

We restrict ourselves to the first three methods. Obviously zbuffer is quite imperative for free-form surfaces and primitives, since it is then an order of magnitude faster than raytracing. Conversely, raytracing is imperative for objects known only by an inequality $f(x, y, z) \leq 0$ (say *soft objects*). On the other hand, some primitives like cubes, quadrics and torii have both an implicit form and a parametric form, so we can freely choose.

3.3 Combining zbuffer and raytracing

To compute a rayrep, one can use either raytracing or zbuffer. Several combinations may be considered, but the simplest one resorts to merging two rayreps with the same ray-grid: for each node A *op* B in the CSG tree, compute the rayrep of A: R_A, the one of B: R_B, then merge R_A and R_B according to the Boolean operator *op*, in the straightforward way. Note the computation of R_A and R_B, when A and B are CSG primitives, can be performed *equally* with raytracing or zbuffer. This combining method is very flexible, and works on two active rayreps at a time (when merging R_A and R_B, R_A is updated to account for R_B), thus it is not very space consuming.

Remark 1: In NC Milling simulation, only one rayrep need be maintained, since the CSG tree has here the special form: $(((...(A - B_1) - B_2) - B_3) \ldots) - B_n$. In our implementation of a NC-milling simulation module, we actually start with the 3rayrep of the initial material block, then iteratively subtract dexels resulting from successive tool movements.

Remark 2: When computing a rayrep by raytracing, we need cast only rays which are in relevant windows of the ray-grid, *ie* S-bounds defined by S. Cameron [Cam91]. Similarly, when using zbuffer, typically for a free-form surface, one can eliminate patches or triangles of the tesselation, that are outside the relevant boxes. Obviously one cannot eliminate back faces or patches.

The whole previous section holds for the computation of a single rayrep. When computing a 3rayrep, the simplest approach is to naively use three times the same method. But it is also possible to use the first rayrep to speed up the computation of the two others. Using a rayrep to speed up raytracing is classical [Van86]: the idea is to trace the ray from pixel to pixel into the rayrep, testing each object present in the pixel for intersection with the ray, until the hit is met.

3.4 Zbuffering free-form surfaces and primitives

A recent method for zbuffering free-form surfaces is due to S. Kumar & D. Manocha & A. Lastra [KML95]. We differ from it in two details: by the fact we cannot eliminate back patches and triangles, and by the treatment for preventing

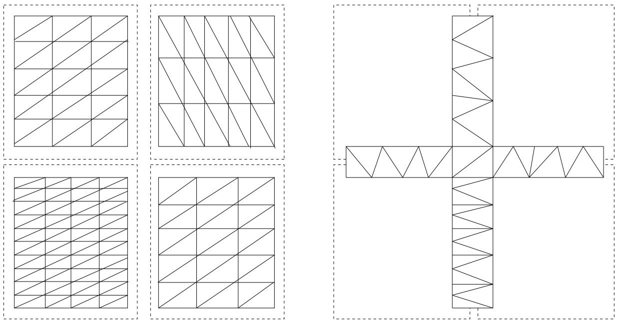


Figure 2: *Left: tesselating the interior of 4 contiguous patches. Right: sewing triangles along the shared edges in real space, and around the shared vertex.*

cracks between contiguous patches. We have preferred the following simple-minded approach (Figure 2):

First, we tesselate each patch independently, keeping a thin strip along the boundary of the parameter domain: for instance, for tensor product patches with domain $[0,1]^2$, we tesselate (roughly like D. Manocha et al) the part $[\epsilon, 1-\epsilon]^2$ with typically $\epsilon = 0.005$. We proceed similarly for triangular patches. Next, we use the non tesselated strip along edges for sewing triangles between contiguous patches. We also generate sewing triangles between corners of contiguous patches.

3.5 Zbuffering and raytracing sweeps

3.5.1 Special cases

A special case is when the surface of the sweep is defined by an available algebraic equation $f(x,y,z,t) = 0$, where t is the time variable. The projection onto the (x,y,z)-space of this hypersurface obeys: $f(x,y,z,t) = \frac{\partial f}{\partial t}(x,y,z,t) = 0$. Using, say, *Sylvester's resultant*, it is possible to eliminate the t variable between the two equations, and obtain a matrix with polynomials entries in x,y,z, whose determinant nullity defines the sweep. Using the matricial form of the resultant, instead of symbolically expanding its determinant, is an idea due to D. Manocha & J. Canny [MC91]. By replacing, for each ray, x,y,z by their values $x = x_0 + \lambda a$, $y = y_0 + \lambda b$, $z = z_0 + \lambda c$, where λ (the abscissa along the ray) is the unique unknown, we obtain an equation in λ, expressed as the nullity of the determinant of a square matrix with polynomial entries in λ. We then proceed as usual, computing an *eigen-decomposition* of the matrix to find λ and then x,y,z. Finally, we find t by computing the *kernel* of the matrix: t is generally needed to check whether it really belongs to some interval.

If the surface of the moving object is described by some equation $f(x,y,z) = f(X) = 0$, and the motion by a function: $t \to M(t)$ where $t \in [0,1]$ and $M(t)$ is an *invertible* 4×4 matrix (typically with polynomial entries in t), then the surface of the sweep is defined by: $f(XM(t)^{-1}) = 0$. This equation can be obtained with some symbolic computations, yielding an equation similar to the one derived previously. These symbolic computations are a little costly but they are done only once and for all rays.

3.5.2 General case

There is a brute-force and general method, not relying on a particular shape of the moving object. This method can be combined equally with either raytracing or zbuffer.

Let A be the moving object, the motion of which is described, like above, by some function: $t \to M(t)$, with $t \in [0,1]$. The main idea is to approximate the sweep by the discrete union: $A(0) \cup A(\Delta t) \cup A(2\Delta t) \ldots \cup A(1)$ where $A(t)$ is the object obtained by applying $M(t)$ to A.

For instance, to simulate material removal in multi-axis milling, developers often discretize the sweeps of tools along NC tool-paths. For each linear tool movement, the swept volume is replaced by a certain number of discrete "instances of motion" to avoid the computational expense of raytracing complex swept volumes [HO94]. The discretization step (together with the ray-grid step) is chosen according to some machining tolerance and other parameters owing to specific milling requirements (like cutter geometry and feed rate). This approach is common in practical NC-milling softwares. It allows real-time and realistic simulation without sacrificing accuracy. The Figure 3 illustrates this approach in 3-axis milling.

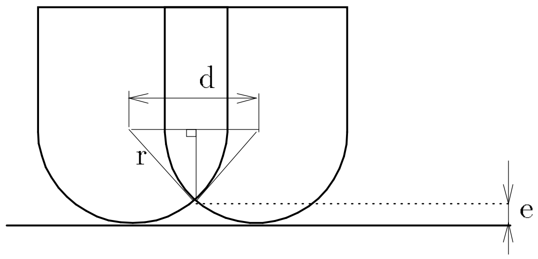


Figure 3: *Computing the discretization step d for tool movements: Let r be the tool radius and e a user-defined tolerance for the "undercut" error due to discretization. Considering two consecutive instances of motion yields* $(\frac{d}{2})^2 + (r-e)^2 = r^2$, *thus* $d = 2\sqrt{e(2r-e)}$ *or, equivalently,* $\frac{d}{r} = 2\sqrt{1-(1-\frac{e}{r})^2}$. *Now, if* c_1 *and* c_2 *are the initial and final locations in the tool movement, the linear sweep between* c_1 *and* c_2 *is decomposed into* $(n+1)$ *evenly spaced tool positions* $c_i = c_1 + \frac{i}{n}(P_2 - P_1)$, $i = 0, n$, *so that* $nd = ||c_2 - c_1||$.

Note it is possible to improve the computed hit of the ray. If the hit involves $A(k\Delta t)$, then the *exact* hit is between $A((k-1)\Delta t)$ and $A((k+1)\Delta t)$: resample this interval. When the sweep is also described by an available system of equations, another improvement is to use the computed hit as an initial guess for some Newton iterations.

As already noticed in [MMZ94], when A is lengthy to raytrace, a good solution is to compute, once and for all, a rayrep (or a 3rayrep) for A. Then the rayrep is used to speed up the intersection between A and *any* ray –not only rays with X-, Y- or Z-direction: see [Van86] and Section 3.3.

4 From 3rayrep to Brep

4.1 The marching cubes algorithm

The MC algorithm was initially designed for surface reconstruction from 3D medical data (obtained by Computer Tomography or Magnetic Resonance techniques). Fundamentally, the space is partitioned into small cubes, each vertex of which carries the value of a certain scalar field (say a temperature or a density). The goal is to construct an isosurface whose points have field-values less or equal to a given threshold. Each cube is examined to trace the isosurface by comparing the field-values at each pair of vertices forming an edge of the cube. If one vertex satisfies the threshold (such vertex is denoted IN) while the other does not (OUT vertex) then clearly the isosurface intersects the current edge (at least once), somewhere between the two vertices. The key assumption in the original MC algorithm is that cubes are small enough so that there is at most one intersection point on each edge. Moreover, since field-values are known only at cube vertices, intersection points are usually computed by interpolating vertex field-values. Finally, intersection points, in each cube, are connected together to form polygonal faces of the isosurface (Figure 4).

The original MC algorithm suffers from the well known "ambiguous cubes". Such a cube has a face with two IN vertices and two OUT vertices, vertices with the same status being on a diagonal of the face. The ambiguity comes from there are two distinct ways to connect the four intersection points (Figure 4). To treat this problem, B. Wyvill & C. McPheeters & G. Wyvill [WMW86] examine the average field-value at the *center* of the ambiguous face, while J. Wilhelms & A. Van Gelder [WG90b] use the more sophisticated *Gradient Heuristic* method.

Storage and performance issues are also addressed by using explicit *octrees* [WG90a], or variable size cubes [MS93]. More recently, R.F. Tobler & T.M. Galla & W. Purgatopher [TGP96] combine octrees and raytracing in their adaptative CSG meshing algorithm, baptized ACSGM. They use normals at vertices to deal with ambiguous voxels.

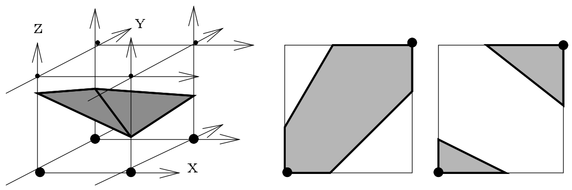


Figure 4: *Left: A possible configuration for a marching cube and an associated local triangulation: Bold vertices denote "in" vertices. Middle and Right: two possible ways to connect intersection points in an ambiguous face.*

R.F. Tobler & T.M. Galla & W. Purgatopher [TGP96] use raytracing to get the *exact* intersection points between rays and edges of the marching cubes. However they consider only implicit surfaces at leaves of their input CSG trees. Moreover they resort to a recursive subdivision by means of an octree, whereas we prefer working on a (possibly recursive) matrix of pixels. Our choice is motivated by the fact that, anyway, the first levels of the octree have generally to be expanded (especially for NC-milling applications), and that using arrays permits enjoying the zbuffer method.

In the original MC algorithm [LC87] W. Lorensen & H. Cline enumerate 256 possible configurations where a surface intersects a cube (assuming it intersects each edge at most once). This number is reduced to 14 really distinct (among which 6 ambiguous) cases, by considering symmetry. Thus, to triangulate the isosurface, they use a *look-up table*: an 8-bit index (one bit per vertex) is created for each case, serving as a pointer into an *edge-table* that gives all the edges intersected in a given cube configuration. B. Wyvill & C. McPheeters & G. Wyvill [WMW86] avoid the look-up table by simply connecting all the intersection points in a cube to their centroid (even such a triangulation method does not work for polygons in general, the authors claim that it works well for polygons involved in their algorithm). As in [LC87], we have used a look-up table based triangulation in our current implementation.

4.2 The MC algorithm in the context of 3rayreps

Our first implementation of the MC algorithm is very close to the original one, except in the following points:

• The 3rayrep permits to examine only the relevant marching cubes, *ie* those that are actually traversed by the isosurface.

• A new kind of ambiguity arises when there are more than one intersection point along an edge of a marching cube. This occurs when the resolution is not sufficient, or in presence of non manifold situations (even they do not appear in machinable objects). Our first implementation assumes this problem does not appear with a reasonable sampling, and we just tag ambiguous cubes to warn the user.

• Finally, combining three rayreps rises consistency problems due to numerical errors: see 5.2.

4.3 Towards the exact Brep

We plan to obtain the *exact* Brep by the following approach: Voxels (*ie* marching cubes) are subdivided until they are "small" enough or "simple", by computing a local 3rayrep (typically $32 \times 32 \times 32$), the CSG tree being simplified to its *active part*. The active part may be computed in several ways. The first and the simplest one restricts the CSG tree to the primitives whose boundary intersects the edges of the voxel. This simplification is fast and the more often sufficient, but may sometimes miss thin or small objects. A more conservative method makes use of Cameron's S-bounds [Cam91] or interval analysis [Mit90].

A voxel is "simple" if it contains no boundary, or only one surface, or only a single intersection curve and its (typically two) incident surfaces, or only one vertex and its (typically three) incident surfaces. The boundary in a simple voxel can be approximated by a more accurate triangulation, as J. Bloomenthal & K. Ferguson [BF95] or R.F. Tobler & T.M. Galla & W. Purgatofer [TGP96] already did. Concerning the "small" residual voxels, which contain singularities (where the jacobian matrix has not full rank) or near-singularities, we may simply opt for a reasonable triangulation (for instance the averaging method in [WMW86]), or just tag them to warn the user.

When there are no such residual voxels (it is the case when the object has no detail smaller than the size of a minimal voxel), or when they are small enough to be assimilated to vertices, we may argue we have actually got the "exact" Brep, at least topologically, since we can recover intersection curves between surfaces, maximum pieces of surfaces (*ie* trimmed patches of free-form surfaces, and pieces of implicit surfaces), and all their connectivity relations: this is nothing else than the exact Brep.

5 About inconsistencies

Our approach is simple and robust. To be honest, we now detail the remaining possible sources of inconsistency:

we show how to solve them, or that they have immaterial consequences.

5.1 Passing through two contiguous triangles

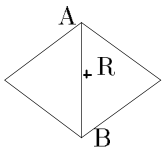


Figure 5: *The hit point R is very close to edge AB. Due to inaccuracy and ill luck, R may be found at the left of AB, and at the left of BA too. Conversely R may be found at the right of both AB and BA.*

Preventing cracks has already been addressed in Section 3.4. But another classical misfortune (we were "unlucky" enough to meet) may happen. Due to numerical floating-point inaccuracy, computing whether a point P lies on the *left* or on the *right* side of an *oriented* edge AB may contradict the result of classifying P with respect to BA (Figure 5): a ray R, very close to an edge AB, can either miss the two incident triangles or hit both. A similar problem is possible with the zbuffer method. A well known solution is to always consider *the same oriented edge* in the two incident triangles (*ie* do not use edge AB in the first triangle and edge BA in the second!), for instance sort endpoints in *lexicographic* order of their coordinates.

5.2 Inconsistencies in 3rayreps

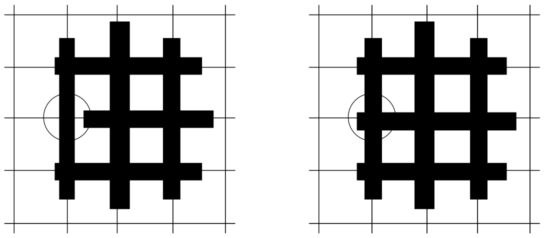


Figure 6: *Left: the circled vertex is found OUT for the X-ray, but IN for the Y-ray. Right: a local perturbation eliminates the inconsistency (exaggerated in the Figure).*

Combining three rayreps inevitably rises another kind of inconsistency, due exclusively to numerical errors. In our 3rayrep, the marching cubes have their vertices (implicitly) defined as the intersection of three rays with distinct directions (X, Y, Z). Thus, when classifying the vertices (to determine their IN or OUT status with respect to a given material) it may happen that the same vertex is found IN with respect to some X-ray, while it is found to be OUT with respect to some Y- and/or Z-ray(s) (Figure 6). To ensure consistency, our solution is brute-force and simple: it consists in a preprocessing step which eliminates possible ambiguous vertices all at once. Each time an ambiguous vertex is detected, the dexels immediately around the vertex (at most six, two from each direction) are slightly shortened or lengthened so that the vertex becomes consistent, possibly merging some dexels to preserve consistency in the dexel lists.

Note we have not met this inconsistency yet, but we mention it for the sake of completeness. Anyway the solution is short and easy.

5.3 Inconsistencies with implicit primitives

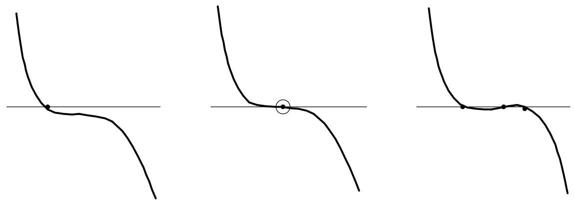


Figure 7: *Numerically indiscernible cases, with an odd number of roots.*

Raytracing implicit primitives boils down to solving a polynomial equation $f(t) = 0$, t being the abscissa along the ray. We recursively solve $f'(t) = 0$, stopping the recursion when reaching degree 1 or 2. In each interval defined by successive roots of f', f has at most one root which may be localized by *dichotomy* and then made more precise by some Newton iterations.

It is essential to avoid miscalculating the parity (even or odd) of the number of ray/surface intersections, which is achieved by our method. Apart this, confusions between two "odd" cases (Figure 7), or confusions between two "even" cases (Figure 8) are inevitable in some cases. Note that an *interval arithmetic* will detect the indetermination, contrarily to the blind floating-point arithmetic, but anyway it will not be able to solve it (Figure 9).

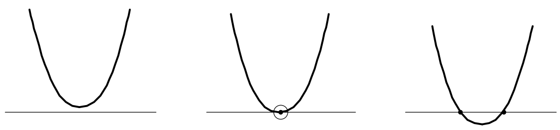


Figure 8: *Numerically indiscernible cases, with an even number of roots.*

These confusions have immaterial consequences: they occur when the ray is tangent or almost tangent to the surface. Their sole effect is to slightly and locally move the outline of the object.

Another error occurs when raytracing exceedingly thin primitives (like the two ellipsoids of the Figure 10). For instance a very thin ellipsoid will be sometimes missed by the ray, sometimes not, apparently at random. Note the modeller does not crash, and that, again, this problem will occur even with an interval arithmetic: the latter will detect the indetermination (contrarily to the floating-point arithmetic) but will not be able to solve it (Figure 9). Actually, this problem is irrelevant, since such objects are rather modelled by laminar ones (here disks instead of ellipsoids).

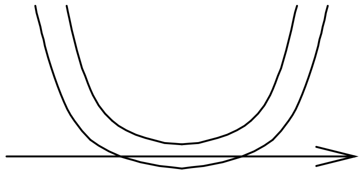


Figure 9: *Interval arithmetic detects that this interval quadratic polynomial may have 0 or 2 single roots, or a double root, but gives no way to decide.*

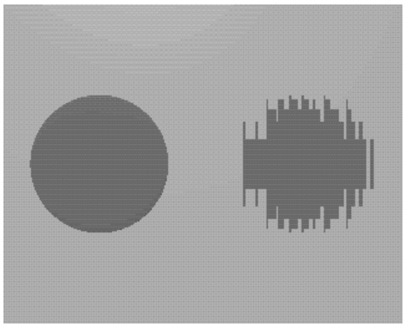


Figure 10: *Two raytraced ellipsoids, with radius* 1*, and thickness* 10^{-5} *for the left one,* 10^{-7} *for the right one. The latter is so thin that some intersections are missed.*

6 Examples of NC-milling simulation

Our first implementation was realized for the simulation needs of a French commercial software, WorkNC, for NC-milling. The Figures 12, 13 and 14 give experimental results. For details see annotations therein. The Figure 12 shows the effect of ray-direction when using a *single rayrep* or a *3rayrep* with the same resolution (*ie* the number of rays). The Figure 13 illustrates the effect of the ray-grid resolution in 3rayreps. The Figure 14 involves a 1.2 meter long part. All experiments are realized on a PC (Pentium 90 processor with 32 Megabytes of Memory), which of course restricts the maximal resolution.

All the shown examples are rendered by flat shading. However, since we supply normals at vertices, the Phong shading may be used as well. The tesselation can be rendered from any viewpoint, in real- or interactive-time on current Graphic Workstations. Moreover, our method also allows wireframe display by producing planar *constant* X-, Y- or Z-*contours* that are very useful in NC-milling applications (Figure 14.(c)). In contrast, contours generated by Y. Huang & J.H. Oliver [HO94] are less accurate than ours since they use only a single rayrep.

Remark: Actually, the 3rayreps themselves (rather than Breps) are stored on disk, which allows multi-stage simulation processes: a milled part may be reconstructed from the saved 3rayrep and then milled with other NC paths and tools.

7 Interval analysis and 3rayrep quality

What is the quality of the approximation of a given 3rayrep ? Interval methods may help answer this question, by providing conservative bounds inside each voxel of the (virtual) voxmap induced by the 3rayrep.

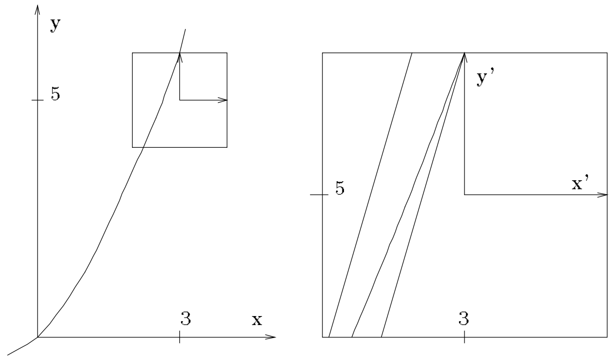


Figure 11: *Interval analysis can delimit a curve between two lines, inside a pixel.*

For simplicity, we discuss only a $2D$ example. Consider a pixel $x \in [x_0 - \frac{\Delta x}{2}, x_0 + \frac{\Delta x}{2}]$, $y \in [y_0 - \frac{\Delta y}{2}, y_0 + \frac{\Delta y}{2}]$ crossed by an algebraic curve $f(x,y) = 0$. We want to get an idea of the areas of the pixel covered by the regions $f \geq 0$ and $f \leq 0$. To fix our ideas, let us consider the example (Figure 7):

$$f(x,y) = y^2 - x^2(x+1) = 0 \ , \ \Delta x = \Delta y = 2 \ , \ x_0 = 3 \ , \ y_0 = 4 \ .$$

It is convenient to introduce two new variables x' and $y' \in [-1, 1]$, then replace x by $x_0 + \frac{\Delta x}{2} x' = 3 + x'$, and y by $y = y_0 + \frac{\Delta y}{2} y' = 5 + y'$. Actually, x', y' are the local coordinates within the pixel. Next, we evaluate $f(x,y)$ inside the pixel by:

$$\begin{aligned} f(x,y) &= f(3+x', 5+y') \\ &= (5+y')^2 - (3+x')^2(3+x'+1) \\ &= -11 - 33x' + 10y' + (-x'^3 - 10x'^2 + y'^2) \\ f(x,y) &\in -11 - 33x' + 10y' + [-11, 2] \end{aligned}$$

Thus, $-22 - 33x' + 10y' \leq f(x,y) \leq -9 - 33x' + 10y'$. So, inside the pixel, the curve is located between the two lines: $y' = 3.3x' + 0.9$ and $y' = 3.3x' + 2.2$. It is then easy to compute bounds for area of the pixel parts covered by $f(x,y) \geq 0$ and by $f(x,y) \leq 0$.

This technique resembles the *affine interval arithmetic* proposed by L.H. de Figueiredo and J. Stolfi [dFS95], though they don't use it to delimit curves (surfaces) between parallel lines (planes) inside a pixel (voxel). This technique may be easily extended to $3D$ and to parametric forms.

8 Improvements, open questions and further works

We are currently implementing a more general prototype to test more deeply our ideas:

- Updating: when the CSG to raytrace is modified, during (say) the design stage, it is better not to redo all the work from scratch, since a few minutes may be needed. It is possible to compute a "background" 3rayrep for the non modified part of the object, and recompute the 3rayrep for the modified part only. Merging two 3rayreps with the same ray-grid is easy and fast. In this way, it is possible to achieve refreshing rates compatible with interaction, if not real-time.

• Aliasing and undersampling problems: even when ambiguous voxels are subdivided, a too large initial ray-grid step may cause the raytracing or the zbuffer method miss thin or small objects. A solution is to stamp and to force the subdivision of voxels containing thin or small objects. The latters may be detected by using conservative bounds like the S-bounds of Cameron {Cameron:1991:EBC, or interval analysis [Mit90]. We have not experimented it yet, for we have not met this problem in our NC-milling application.

• Curves: *Wire-like* objects are sometimes represented by curves. Of course, the naive raytracing method will miss such curves. A solution is to assign them a thickness of (say) 2 pixels. One may find this is cheating, but assigning thickness to streets or rivers in geographical maps is usual: it is here the same idea. There are no problems with surfaces representing *laminar* objects, since the raytracing method will not miss them. Note J. Bloomenthal and K. Ferguson [BF95] have already polygonized non-manifold implicit surfaces.

• Other kinds of primitives: for instance, as already pointed out in [MMZ94], rayreps enable the computation of Minkowski sums, offsets, blends and fillets.

9 Conclusion

We have proposed a very simple framework for robust free-form solid modelling. Our approach naturally suppresses the necessity of sophisticated (and often unreliable) methods for intersecting and classifying complex curves, surfaces and objects: all the difficulty is handled by raytracing or zbuffering. This is done without loss of practicalities: since it is now sufficient to raytrace or zbufferize a primitive, it becomes easy to handle a new kind of objects in geometric modellers. Our approach relieves developers of the burden of surface to surface intersection and robustness-related issues. We believe it significantly helps in repairing the divorce between CSG and Brep models. We are conscious that our approach is brute-force and simple-minded, when compared with the most sophisticated ones. However, since the hardware is in constant improvement, brute-force methods become viable and take over because of the simplicity of implementation.

Acknowledgements

We would like to thank the anonymous referees for their remarks, and the LISSE team (École des Mines de St-Étienne, France) for helpful discussions.

References

[BF95] J. Bloomenthal and K. Ferguson. Polygonization of non-manifold implicit surfaces. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Computer Graphics Proceedings, Annual Conference Series, pages 309–316. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[Bow95] A. Bowyer. *SVLIS – Introduction and User Manual.* Information Geometers Ltd, 47 Stockers Avenue, Winchester, SO22 5LB, UK, second edition, 1995.

[Cam91] S. Cameron. Efficient bounds in constructive solid geometry. *IEEE Computer Graphics and Applications*, 11(3):68–74, May 1991.

[dFS95] L.H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. In *Proceedings Eurographics Workshop on Implicit Surfaces*, pages 161–170. INRIA, 1995.

[HO94] Y. Huang and J.H. Oliver. NC milling error assessment and tool path correction. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 287–294. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

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

[HPY96] C.-Y. Hu, N. Patrikalakis, and X. Ye. Robust interval solid modelling. part 1: Representations. part 2: Boundary evaluation. *CAD*, 28(10):807–817, 819–830, 1996.

[IS'95] *Implicit Surfaces'95.* Inria, Grenoble, France, 18-19 avril 1995. Proceedings of Eurographics Workshop.

[KM96] S. Krishnan and D. Manocha. Efficient representations and techniques for computing b-reps of csg models with nurbs primitives. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 101–122, Winchester, UK, April 1996.

[KML95] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large-scale NURBS models. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 51–58. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[MC91] D. Manocha and J.F. Canny. A new approach for surface intersection. *International Journal of Computational Geometry and Applications*, 1(4):491–516, 1991. Special Issue on Solid Modelling.

[MG96] J.P. Menon and B. Guo. A framework for sculptured solids in exact csg representation. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 141–157, Winchester, UK, April 1996.

[Mit90] Don P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface '90*, pages 68–74, May 1990.

[MMZ94] J. Menon, R.J. Marisa, and J. Zagajac. More powerful solid modeling through ray representations. *IEEE Computer Graphics and Applications*, 14(3):22–35, May 1994.

[MPS96] K.T. Miura, A.A. Pasko, and V.V. Savchenko. Parametric patches and volumes in the functional representation of geometric solids. In Information Geometers Ltd, editor, *Proceedings of*

CSG96, pages 217–231, Winchester, UK, April 1996.

[MS93] H. Muller and M. Stark. Adaptive generation of surfaces in volume data. *The Visual Computer*, 9(4):182–199, January 1993.

[MV95] J. Menon and H. Voelcker. On the completeness and conversion of ray representations of arbitrary solids. In Chris Hoffman and Jarek Rossignac, editors, *Solid Modeling '95*, pages 175–186, May 1995.

[PASS93] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1993.

[Pat93] N.M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95, jan 1993.

[Sny92] J.M. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121–130, july 1992.

[ST91] T. Saito and T. Takahashi. NC machining with G-buffer method. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 207–216, July 1991.

[Tau93] G. Taubin. An accurate algorithm for rasterizing algebraic curves. In *Second Symposium on Solid Modeling and Applications, ACM/IEEE*, pages 221–230, May 1993.

[Tau94] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer Graphics and Applications*, 14(2):14–23, mar 1994.

[TGP96] R.F. Tobler, T.M. Galla, and W. Purgatofer. Acsgm–an adaptative csg meshing algorithm. In Information Geometers Ltd, editor, *Proceedings of CSG96*, pages 17–31, Winchester, UK, April 1996.

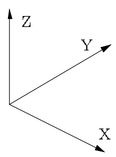
[Van86] Tim Van Hook. Real-time shaded NC milling display. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 15–20, August 1986.

[WG90a] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation extended abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 57–62, November 1990.

[WG90b] J. Wilhelms and A. Van Gelder. Topological considerations in isosurface generation extended abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24, pages 79–86, November 1990.

[WMW86] B. Wyvill, C. McPheeters, and G. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

a



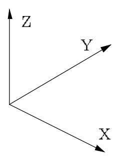
b

c

d

Figure 12: *The workpiece has size:* $109 \times 120 \times 68$ *mm, the NC path consists of* $21,553$ *tool movements, and the tool is a ballend with a radius of* 4 *mm. (a) shows a single rayrep with a resolution of* 163×179 Z*-rays. The rayrep computation took about 0.6 mn. (b) shows a 3rayrep with resolution* $110 \times 121 \times 69$*, which involves approximately the same number of rays (29,249). Raytracing took about 1.8 mn. MC algorithm took 6 seconds to build the faceted Brep. (c) shows a wireframe display of constant-*X *contours. (d) shows constant-*Y *contours. Note the poor sampling in regions where the surface is nearly parallel to* Z *axis.*

a



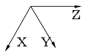
b

c

d

Figure 13: *The same part, tool and tool path as in Figure 12. But the 3rayrep has a finer resolution of* $146 \times 161 \times 91$*, which involves a total of* $51,443$ *rays. Raytracing took about* 4.6 *mn. MC algorithm took* 11 *seconds. (a) shows the NC tool path generated by WorkNC (*$21,553$ *tool movements). (b) shows the tesselated 3rayrep. Note the better aspect of the surface than in Figure 12.a and 12.b. (c) shows a zoomed detail of the upper part. (d) shows the milled part from another viewpoint.*

a



b

c

d

Figure 14: *The workpiece has size* $600 \times 1200 \times 150$ *mm. The tool is a ballend of radius* 10 *mm. (a) shows the NC path with* $9,141$ *tool movements. Due to symmetry, the simulation has been performed only on half the part. (b) shows the 3rayrep of a roughing result, with resolution* $301 \times 251 \times 76$*. Raytracing took about* 3.23 *mn. MC algorithm took* 36 *seconds. The total number of marching cubes is* $5,625,000$*, only* 11% *of which are actually crossed by the surface. (c) shows a detail of the upper left corner in (b) through constant* X *and* Y *contours. (d) is a flat shading display of the same detail.*