Exact and Efficient Booleans for Polyhedra

Cyril Leconte, Hichem Barki, and Florent Dupont
LIRIS Laboratory, UMR CNRS 5205, Université de Lyon, Université Claude Bernard Lyon 1
43 Bd. du 11 novembre 1918, F-69622 Villeurbanne, France

ABSTRACT
Boolean operations are crucial for many domains, especially for those requiring exactness and efficiency. Morphological filtering and indexation of meshes are the most critical applications of Booleans because they need to perform them sequentially. In this context, we propose a new algorithm which fulfils such requirements. We achieve efficiency and exactness through the use of an efficient data structure and the adoption of the exact computation paradigm. Our benchmark showed that our algorithm handles big size polyhedra and outperforms state of the art methods.

Categories and Subject Descriptors
I.3.5 **[Computer Graphics]**: Computational Geometry and Object Modeling
F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—Geometrical problems and computations
J.6 **[Computer-Aided Engineering]**: Computer-aided design (CAD)

General Terms
Algorithms, Performance, Reliability.

Keywords
Boolean operations, polyhedra, exact computation, geometric modeling.

1. INTRODUCTION
Boolean operations on polyhedra are a fundamental task in computational geometry, computer-aided design and manufacturing, computer graphics, etc. Their exact computation and the handling of all degenerate cases is a difficult task because of non-manifold results (the set of polyhedra is not closed under Boolean operations), round-off errors inherent to built-in number types, etc. In computational geometry, the exact computation paradigm is becoming widely used. It allows addressing the non-robustness issues related to round-off errors. However, exact arithmetic is slow in comparison with floating-point one.

Our work is motivated by the need to exact and efficient Boolean operations in order to apply successions of Minkowski operations (addition and subtraction) on meshes as done in mathematical morphology on discrete images. Such morphological filtering techniques will allow characterizing 3d models for indexation purpose.

In this work, we present a new algorithm for the exact computation of Boolean operations on polyhedra. We used exact arithmetic to avoid round-off related issues and efficient data structures to achieve efficiency. It is also known that the set of polyhedra is not closed under Minkowski operations (see fig. 1 for an example with 2D polygons). Therefore, one must process the output of Minkowski operators in order to render it valid for subsequent operations, and take care of efficiency and exactness at the same time.

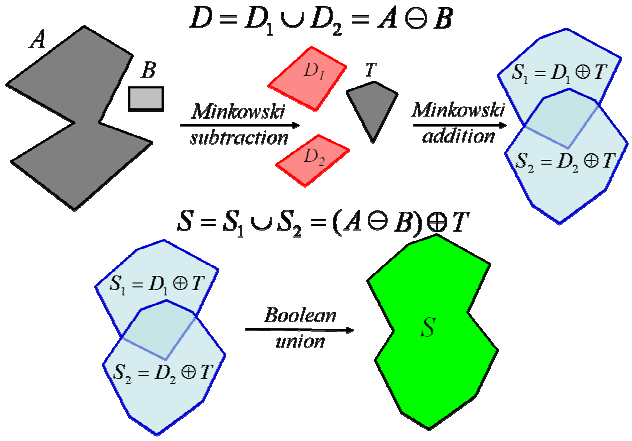

Figure 1. Necessity of computing Booleans when applying Minkowski operations.

2. RELATED WORK
Several approaches have been proposed in literature for the computation of Boolean operations and polyhedra.

An exact approach based on Nef polyhedra was proposed by Hachenberger [1-2] and implemented in CGAL. Even if this approach is robust and is able to handle non-manifold output, it is slow in practice and thus not suited for time-critical tasks. Moreover, it suffers from overflow issues when operating on large size polyhedra. While some approaches have been only dedicated to free-form solids [3-4-5], some others tried to approximate exact Booleans.

Smith and Dodgson proposed an algorithm ensuring a result topologically correct [6]. However, this method uses approximations (perturbations) to determine the relationship between two entities (vertices, edges, facets and solids) and never consider the case where they are coincident. This algorithm may generate geometric artifacts.

A hybrid method using a volumetric model has been presented by Pavić [7], but this method generates errors if the resolution of the octree-based volumetric model does not fit with the size of the facets.

Exact computations have been adopted by Bernstein and Fussell with a method using plane-based convex polygon splitting algorithm and binary space partitioning (BSP) [8]. Recently, Campen and Kobbelt used this method to find the outer hulls of self-intersecting meshes, and Boolean operations [9].

3. OUR METHOD

3.1 Description

In this work, we propose an efficient method to compute the union, the intersection and the subtraction between two polyhedra. The computation of the intersections is based on exact computations [10]. Since the result of Boolean operations is not necessarily two-manifold (faces touching tangentially, double vertices, etc.), we decided to separate the geometry of the non-manifold features. However the resulting mesh cannot be used as an input for another Boolean operation since our algorithm operates only on polyhedron. Our idea is to compute the intersection of the two input polyhedra and build the result by propagation, using the intersection as a bounding line (see fig. 2 for a simple example between two spheres).

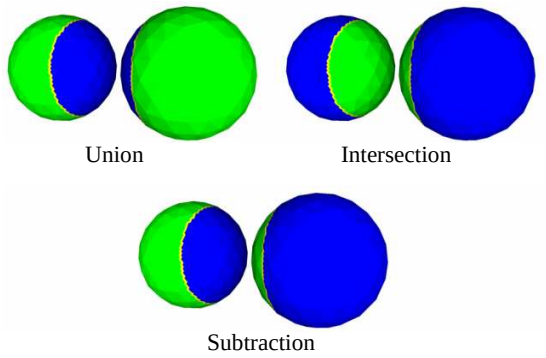

Figure 2. The green zone is the part of the input polyhedra that belongs to the result polyhedron

If some facets of the polyhedra are coplanar, the intersection between the meshes is not a line, but the bounding line we need is included into the intersection. This case is described with more details in the subsections 3.2.3 and 3.2.5.

3.2 The Algorithm

We want to compute a bounding line on the polyhedra and build the result of the Boolean operation around this line, following the surface of the polyhedra. Our algorithm is composed of six steps to compute a Boolean operation.

1- Initialisation of the polyhedra
2- Find every pairs of triangles that intersect
3- Compute the intersections as segments
4- For each intersected triangle, store the intersection segments and orient them
5- Triangulate the intersected facets and add to the solution the triangles that belong to it.
6- Propagate the result to the whole polyhedra, using the connectivity of the facets.

3.2.1 Initialisation

The first step consists of checking that the facets of the two polyhedra are triangular. If not, a triangulation is done for each facet. This is necessary to simplify the computation of the intersections to a single case between two triangles. The triangulation is the only modification done on the two input polyhedra.

3.2.2 Finding the Intersections

The boundary line is made of segments, and each segment is an intersection between two triangles. We then have to find which triangles are concerned by the intersection between the two input polyhedra, and more precisely, we must find every pair of triangles that intersect. It is possible to test every pair but this solution has a *O(nm)* time complexity. That is why we will use an AABB-Tree (Axis Aligned Bounding Boxes Tree). For each facet of one of the input polyhedra, we compute its axis aligned bounding box and we store it into a binary search tree were each node is the bounding box of its two sub-trees (see fig. 3).

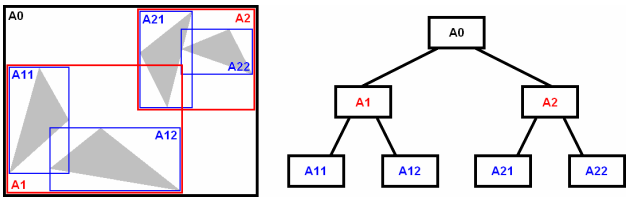

Figure 3. AABB-Tree built on four triangles

The construction of the tree has a time complexity of *O(n log(n))* and the intersection test of an axis aligned box into this tree has *O(log(n))* time complexity. Therefore, the full intersection test between the two polyhedra has *O((n+m)*log(n))* time complexity. The other advantage in using an AABB Tree is that the construction of an axis aligned bounding box for a triangle and the intersection test are efficient. In order to reduce the time complexity, the tree is built on the polyhedron having the smaller number of facets because:

$$\text{If } n < m$$
$$\text{Then } (n+m) * log(n) < (m+n) * log(m)$$

Every pair of facets whom bounding boxes intersect is stored to compute the intersection. In most of the cases, not all the stored pairs of triangles are really intersecting. If the triangles are very close, their bounding boxes could intersect.

3.2.3 Compute the Intersections

For each facet, we need to find and store every intersection with the facets of the other polyhedron. We want to find a bounding line on the two input polyhedra. In most cases, the intersection of two triangles is a segment, but in certain particular cases, it could be a point or a polygon. For the coplanar cases, the three edges of a facet belong to the plane of the other facet. It means that the three neighbouring facets have at least one edge included in the plan. That is why it is not necessary to compute the intersection for the coplanar cases because they are obviously surrounded by other intersections that are easier to compute.

As we want to find a bounding line made of segments, it is also not necessary to consider the cases where the intersection is a point since it does not contribute to the computation of a line.

The algorithm we used to compute the intersection between two triangles is based on the method described by Schneider and Eberly [11]. The first step in the computation consists of finding the position of each triangle with respect to the plan of the other triangle. There are six different cases shown in fig. 4.

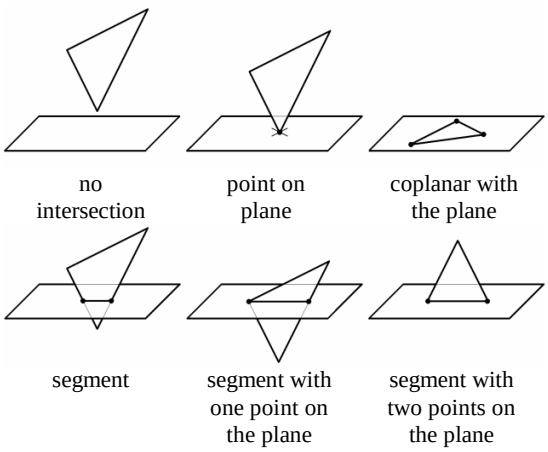

Figure 4. The six possible intersections of a triangle and a plane.

If one of the two triangles is in one of the three first configurations, the intersection is not computed (a coplanar case is always surrounded by configurations of intersection with two points on the plane or by others coplanar cases). With this first result, we know the position of the three vertices of a triangle with respect to the plan, so we know which edges are intersecting the plan. In general, two edges are intersecting the plan. Otherwise, it means that one or two vertices are on the plane, so we directly know the coordinates of the intersection. For each triangle, we compute, if necessary, the two intersection points between the two edges and the plan, and we verify that these points are in the triangle or not. There are three possibilities now. If none of the points are in the triangle, the two triangles do not intersect. If we founded two distinct points included in the two triangles, we have the intersection segment. If we have only one point, it means that we are on the case shown in fig. 5.

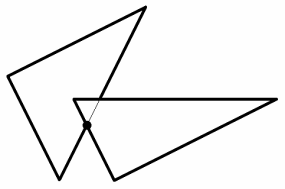

Figure 5. Intersection of two edges.

However, this point is important and must be memorized, because if this is the unique intersection of these triangles, we have to know this point to subdivide correctly these facets.

Every edge on the polyhedra belongs to two facets. If these facets intersect another facet, we would compute the intersection twice. Therefore for each intersected facet, we store the list of the edges which cross the facet, with a reference to the intersection point. We can avoid computing an intersection more than one time by verifying if the point has already been computed.

3.2.4 Orientation of the Segments

The bounding line cuts the two initial polyhedra into two parts, but we need to know which one is part of the result of the Boolean operation and which is not. That is why each segment is stored as two points in a particular order. The cross product between the normal of the triangle and the vector created with the two points must be pointing to the direction of the part of the polyhedron that belongs to the result. To know which part of the intersected triangle belongs to the result, we must take into account the cross product between the normal of the triangles and the Boolean operator (union, intersection or subtraction).

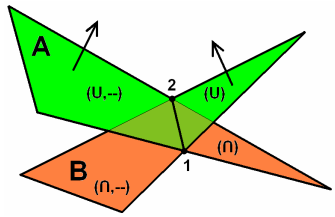

Figure 6. The intersection of two triangles is a segment defined by the points 1 and 2.

The order of the points when we store the segments follows this procedure:

If (N_A x N_B) . (*pt2* – *pt1*) > 0
- If UNION
  - store (*pt1*, *pt2*) for triangle *A*
  - store (*pt2*, *pt1*) for triangle *B*
- Else If INTERSECTION
  - store (*pt2*, *pt1*) for triangle *A*
  - store (*pt1*, *pt2*) for triangle *B*
- Else If SUBTRACTION
  - store (*pt1*, *pt2*) for triangle *A*
  - store (*pt1*, *pt2*) for triangle *B*
- EndIf

Else
- If UNION
  - store (*pt2*, *pt1*) for triangle *A*
  - store (*pt1*, *pt2*) for triangle *B*
- Else If INTERSECTION
  - store (*pt1*, *pt2*) for triangle *A*
  - store (*pt2*, *pt1*) for triangle *B*
- Else If SUBTRACTION
  - store (*pt2*, *pt1*) for triangle *A*
  - store (*pt2*, *pt1*) for triangle *B*
- EndIf

EndIf

3.2.5 The Particular Cases

When the intersection is a case where an edge of one of the triangles is on the plane of the other triangle, is does not mean that there is a part of the bounded line here. We must verify that the two polyhedra are partially overlapping here. The different cases are represented in fig. 7.

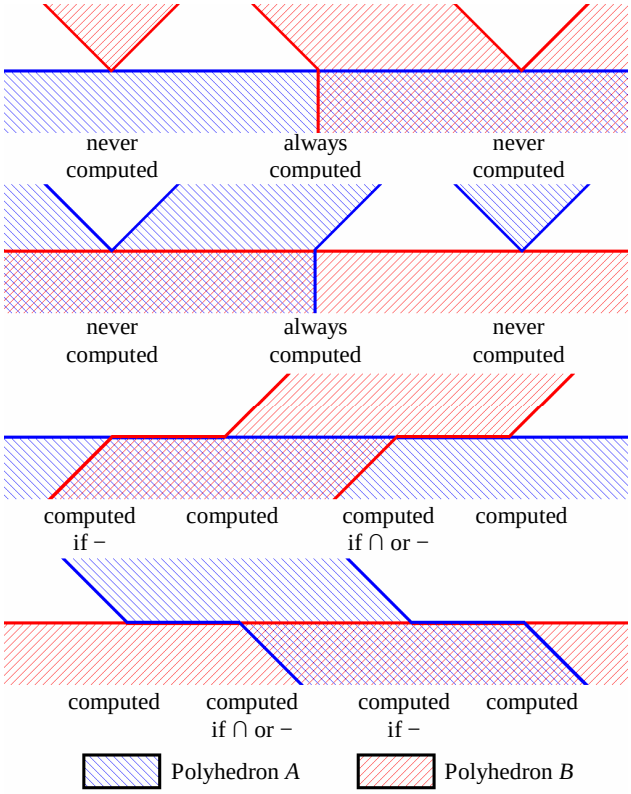

Figure 7. Conditions for computation when a particular case where a contact "edge-triangle" happens.

If the two triangles intersect in an edge, we must consider the position of the two other triangles sharing the same edges. We have to find if the two polyhedra cross in this edge or not. For each facet of the polyhedron *A* sharing the edge, we have to find if they are inside the polyhedron *B* or not. If the two facets are on the same side, there is no intersection. Otherwise, the intersection is part of the bounded line. It is possible that two facets are coplanar. In this case, we must take into account the Boolean operator (union, intersection or subtraction) to determine if the facet of *A* must be considered inside or outside *B*. It is also possible that each facet of *A* is coplanar with two facets of *B*. In this case, we consider that the polyhedra are not overlapping.

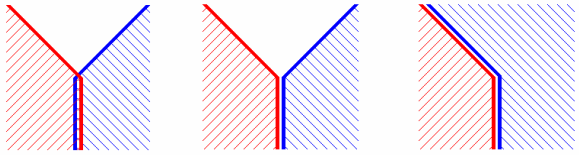

Figure 8. Examples where the two facets intersect on an edge.

3.2.6 Cut the Intersected Facets

The intersected facets are triangulated using the constrained Delaunay triangulation. The segments computed in the part 3.2.3 are constraints. Then, we are able to know which of the two triangles of the triangulation, on each side of a constrained segment, belongs to the result or not. We propagate this information to the neighbouring triangles and reverse it if a constrained edge is crossed, until each triangle is checked. If a facet intersects another, but does not have any constrained segment, we do not validate the facet for now. We use the triangulation to know if the three neighbouring facets belong to the result. If these facets have constrained segments, this information would be false, but it is not important because this information is used only if the facet does not have any constrained segment.

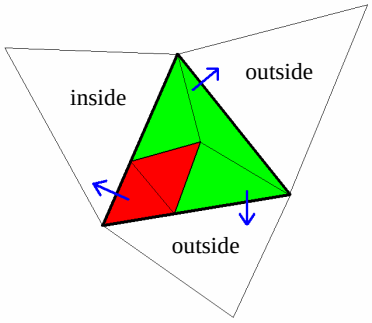

Figure 9. Identification of the three neighbouring facets.

3.2.7 Propagate the Result on the Polyhedra

During the previous step, we determined if the three facets surrounding an intersected facet belong to the result (see fig. 10).

We just have to propagate this information to the whole polyhedra using the connectivity between the facets. If a facet belonging to the result has constrained points, this facet is triangulated.

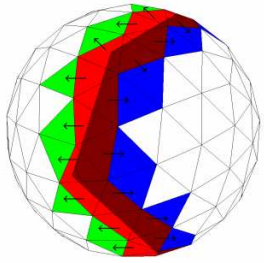

Figure 10. One of the polyhedra before the propagation. The red facets are intersected, the green facets are part of the result, and the blue facets are not.

4. RESULTS

We have tested our algorithm on simple polyhedra to check its correctness. The figure 11 represents the results for the three operations (union, intersection and subtraction) with a star and a cross. We can see that these results are correct. However, there are no particular cases with these polyhedra.

Then, we wanted to check if the particular cases are well computed. The three following examples show what happens when coplanar cases appear. The figure 12 shows the union of two overlapping cubes in a particular case. The top and bottom facets are coplanar and certain facets on the side present a configuration where the triangles intersect on an edge. The

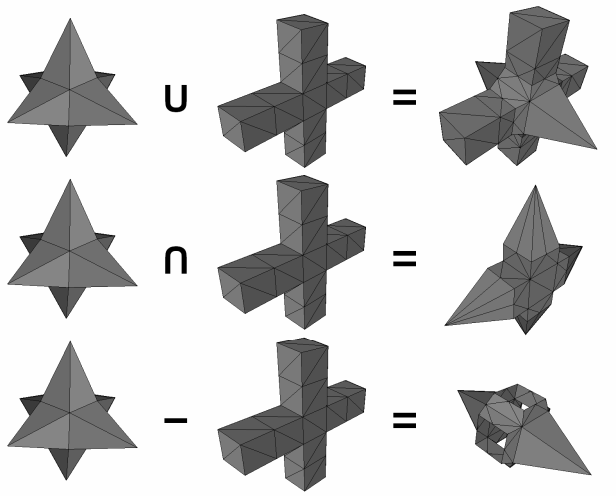

Figure 11. Results for the union, the intersection and the subtraction of two polyhedra (star and cross).

subtraction between these two cubes in figure 13 shows that the two coplanar facets disappeared, proving that the computation of the subtraction is correct in particular configurations. If two cubes are in contact on one facet, the intersection is an empty polyhedron. But for the union of these cubes, the two parts of the facets in contact should disappear. The figure 14 shows this result. To make it easier to be seen, a facet has been deleted.

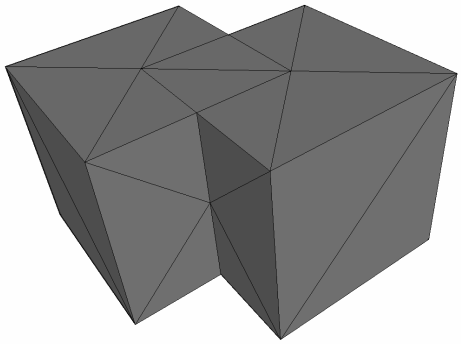

Figure 12. Union with coplanar facets (overlapping cubes).

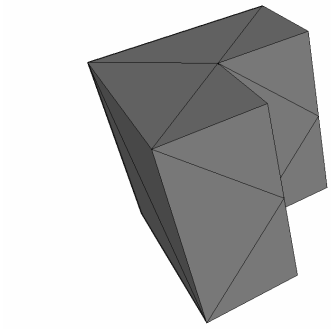

Figure 13. Subtraction with coplanar facets (overlapping cubes).

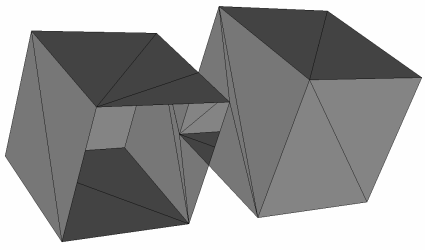

Figure 14. Union with coplanar facets (contact between two cubes). The common facet correctly disappeared.

We have also tested our algorithm on many large 3d models (composed of hundred of thousand of facets). In order to compare our algorithm with others, we have computed Booleans operations with two other methods. The first one is the Nef-based approach implemented in CGAL [1-2], the second one is the algorithm implemented in Moka [13] and based on topologic maps following N. Guiard's method [12]. The table 1 shows the computation time for three different methods. f_A and f_B are the number of facet of the two input polyhedra and f_I is the total number of intersected facets. The polyhedron *A* is an aircraft and the polyhedron *B* is an elephant. The figure 15 shows the result of the three operations with these polyhedra (first column of table one).

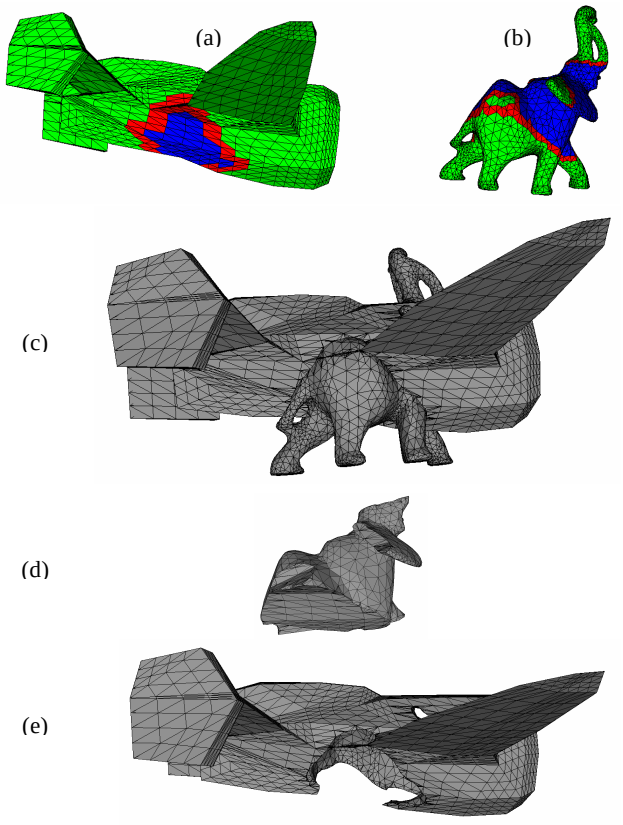

Figure 15. (a) and (b) : aircraft and elephant. The intersected facets are red, the green facets are outside the other polyhedra, the blue facets are inside the other polyhedra.
(c) union. (d) intersection. (e) subtraction.

Table 1. Time results for comparison with other algorithms

		#facets f_A = 4864 f_B = 5558 f_I = 442	#facets f_A = 19456 f_B = 22232 f_I = 904	#facets f_A = 77824 f_B = 88928 f_I = 1827
CGAL (Nef)	**Union**	34.546 s	147.684 s	963.154 s
	Inter	32.672 s	132.435 s	660.487 s
	Sub	33.515 s	138.388 s	783.458 s
Moka	**Union**	0.832 s	3.268 s	11.340 s
	Inter			
	Sub			
Our Method	**Union**	0.380 s	0.930 s	2.511 s
	Inter	0.364 s	0.858 s	2.134 s
	Sub	0.372 s	0.898 s	2.351 s

The union with 311296 facets for the aircraft and 355712 facets for the elephant took 7.805 seconds to be computed and the final result has 567952 facets.

We can see that our algorithm is 90-380 times faster than CGAL. However, contrary to our method, CGAL creates a result without any useless vertices. Moka gives the result of the three operations, but for practical purposes, we generally need only one result. In addition to this, certain parts of our algorithm are common for the three operators. Then, the computation of the three operations would take less than the sum of them computed separately. Moreover, Moka does not ensure that the result is correct.

5. CONCLUSION AND PERSPECTIVES

We proposed an algorithm for the computation of Boolean operations on polyhedral based on an AABB-Tree construction and bounding lines computation.

The particular cases for the intersection are computed separately to ensure a correct result. The treatment of these cases is longer than the regular cases, but they are uncommon in practice. Our approach gives correct results because we used exact number types (no round-off errors).

Experimental results showed that our algorithm outperforms existing methods cited in this paper (based on Nef polyhedra and combinatorial maps). Moreover, our algorithm correctly handled big size polyhedra comprised of hundreds of thousands of facets.

We are working on the generalization of our algorithm to non-manifold input and meshes composed of several disjoint boundaries. All this work is necessary in order to perform morphological filtering on meshes, which in turn will allow proposing new measures for the characterization and indexation of 3d models.

6. REFERENCES

[1] Granados, M., Hachenberger, P., Hert, S., Kettner, L., Mehlhorn, K. and Seel, M., Boolean operations on 3D selective Nef complexes: *Data structure, algorithms, and implementation*. In: Lecture Notes in Comput. Sci., vol. 2832. Springer, Berlin. pp. 654-666.

[2] Hachenberger, P. and Kettner, L. 2005. Boolean operations on 3D selective Nef complexes: optimized implementation and experiments. In *Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling* (Cambridge, Massachusetts, June 13 - 15, 2005). SPM '05. ACM, New York, NY, 163-174.

[3] Biermann, H., Kristjansson, D., and Zorin, D. 2001. Approximate Boolean operations on free-form solids. In *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '01. ACM, New York, NY, 185-194.

[4] Adams, B. and Dutré, P. 2003. Interactive boolean operations on surfel-bounded solids. In *ACM SIGGRAPH 2003 Papers* (San Diego, California, July 27 - 31, 2003). SIGGRAPH '03. ACM, New York, NY, 651-656.

[5] Bajaj, C., Paoluzzi, A., Portuesi, S., Lei, N. & Zhao, W. 2008 Boolean operations with prism algebraic patches. *Comput. Aided Des. Appl.* 5, 730–742

[6] Smith, J. M. and Dodgson, N. A. 2007. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. *Comput. Aided Des.* 39, 2 (Feb. 2007), 149-163.

[7] Pavić, D., Campen, M. and Kobbelt, L. 2010. Hybrid Booleans. *Computer Graphics Forum*. vol.29(1), pp.75-87, 2010.

[8] Bernstein, G. and Fussell, D. 2009. Fast, exact, linear Booleans. In *Proceedings of the Symposium on Geometry Processing* (Berlin, Germany, July 15 - 17, 2009). Eurographics Symposium on Geometry Processing. Eurographics Association, Aire-la-Ville, Switzerland, 1269-1278.

[9] Campen, M. and Kobbelt, L. 2010. Exact and Robust (Self-) Intersections for Polygonal Meshes. In *Computer Graphics Forum*. vol.29, pp.397-406, May 2010.

[10] Pion, S. and Fabri, A. 2006. A generic lazy evaluation scheme for exact geometric computations. In *Proc. 2nd Library-Centric Software Design*, 75-84.

[11] Schneider, P. J. and Eberly, D. 2002 *Geometric Tools for Computer Graphics*. Elsevier Science Inc.

[12] Guiard, N. 2006. Construction de modèles géologiques 3D par co-raffinement de surfaces. *PhD thesis*.

[13] Moka. www.sic.sp2mi.univ-poitiers.fr/moka/