Computer-Aided Design 45 (2013) 705–716

Contents lists available at SciVerse ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

Fast and accurate evaluation of regularized Boolean operations on triangulated solids

F.R. Feito [a,*], C.J. Ogayar [a], R.J. Segura [a], M.L. Rivero [b]

[a] *Department of Computer Science, University of Jaén, EPS Jaén, 23071, Spain*

[b] *Department of Computer Science, University of Jaén, EUP Linares, 23700, Spain*

ARTICLE INFO

Article history:
Received 27 January 2012
Accepted 22 November 2012

Keywords:
Boundary representations
Geometric algorithms
Boolean operations
Triangular meshes

ABSTRACT

In this paper we present a robust and accurate method for evaluating regularized Boolean operations on triangulated solids. It allows the exact evaluation of the regularized union, intersection, difference and symmetric difference simultaneously. Moreover, this approach is simpler than other methods, including those that provide an approximate evaluation or only a rendering of the result. It is based on a simple data structure and on the use of an octree which facilitates the division of the geometry into subsets for distribution among several threads, and accelerates the spatial queries needed during the process. This method is designed to be used in a multithreaded environment and it can also be implemented using an out-of-core approach. We also present some experimental results, and a comparison with other systems that also provide an exact evaluation of the Boolean operations.

1. Introduction

Boolean operations between solids are used in geometric modeling for the manipulation of objects and the creation of new ones. These operations, such as union, intersection, difference, and symmetric difference, are applied to the boundary of 3D objects in the same way as with other sets. New complex solids can be obtained as a combination of others, which are usually simpler. This form of 3D modeling has been used extensively in fields such as industrial engineering (CAD/CAM), virtual reality, and Computer Graphics [1,2]. Boolean operations are the basis of Constructive Solid Geometry. Moreover, they can be used for simulation purposes, such as drilling, machining or collision detection.

The polygonal mesh is the most popular boundary representation for 3D objects. More specifically, the triangle mesh is the most versatile polyhedral representation among B-Rep representations [3]. Triangle meshes have multiple applications. They are used in CAD, virtual reality and videogames to represent polyhedral solids. This representation scheme is a standard in several areas, due to its simplicity. It can represent almost any object with a given level of detail. Moreover, it can be directly processed with graphics hardware.

A Boolean operation between polygonal models can present several problems. Many commercial CAD systems have difficulties in resolving operations with very complex models. Occasionally they fail to give a correct result (which in most cases is an empty set), or they refuse to give a result at all. Some systems are unable to resolve a Boolean operation between solids even with a medium complexity (100k–200k triangles). Numerical stability and memory consumption are the main problems, although the performance can be also an issue. Typical CAD systems operate on a data structure derived from a winged-edge, and the intersections are calculated per edge. Frequently, some hierarchical data structure is used for classifying geometric entities, such as BSP. In this case, a large number of faces in the solids can lead to an insufficient memory problem and a poor performance in most cases. Other systems avoid these drawbacks by converting the polygonal meshes into volumetric representations. After resolving the Boolean operation with voxels in a straightforward manner, the polygonal result is extracted from the volumetric result. There are other frameworks that give approximate results by using direct rendering or by performing local operations only on intersection zones [4–6]. In any case, those systems only provide approximate results in order to maximize the performance. Moreover, they tend to suffer from additional issues and special cases.

In this paper we present a method for evaluating regularized Boolean operations on triangulated meshes. The algorithm presented here is based on a classical approach for B-Reps [7–10]. We focus on triangle intersection and tessellation. The first step consists of intersecting the two solids involved in the Boolean operation (mesh refinement). The second step performs the classification of the triangle set resulting from the previous step. These new meshes are conveniently tessellated so that every triangle from both meshes is totally inside, outside or on the boundary of the other mesh. Because of that, the problem of triangle classification is

* Corresponding author. Tel.: +34 953 212446; fax: +34 953 212472.
E-mail addresses: ffeito@ujaen.es (F.R. Feito), cogayar@ujaen.es (C.J. Ogayar), rsegura@ujaen.es (R.J. Segura), mlina@ujaen.es (M.L. Rivero).

0010-4485/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2012.11.004

reduced to a point-in-solid test. The final step selects a triangle set that meets the conditions of the selected Boolean operation, and then builds the resulting solid. Our approach is optimized in several ways, therefore it is more efficient than other methods. It can operate with complex meshes without memory issues, in contrast to BSP-based methods. One of the main aspects of the algorithm is the use of an octree, which accelerates the spatial queries needed during the process. Also, by using an octree the data set can be divided into subsets for distribution among several threads of execution. This allows us to use a multicore implementation or even an out-of-core approach for working with very complex solids. The implementation tested clearly outperforms other Boolean evaluation systems that also provide an exact solution and preserves all features of the solids.

The rest of the paper is organized as follows. The next two sections present some previous work and background. Then, some preliminary steps are presented, which include key aspects of our approach. After that, the mesh refinement process, the classification of the mesh and the Boolean evaluation are presented. Finally, implementation and results are analyzed, and the paper is concluded with some final considerations about the work.

2. Previous work

Boolean operations between solids have been researched in Computer Graphics for years [11,12]. Many approaches have been used for resolving Boolean operations and ultimately CSG trees. However, since this problem is very complex, not every method is suitable for all situations. The representation scheme determines the algorithm to be used. In this work we focus on a specific type of B-Rep, that is, triangle meshes. Other representations for 3D objects, such as volumetric structures, curved surfaces or complex non-manifold B-Reps, require different solutions.

First methods compute Boolean operations directly over the input elements, such as faces, edges and vertices [13,7,14,11,10]. These algorithms tend to suffer from robustness problems, therefore volumetric methods are often preferred. However, a volumetric representation has a limited precision, and the loss of geometric features is unavoidable. Typical methods use a Marching Cubes algorithm to extract features from the volumetric result [15,16]. A proposed solution for the first problem is a system based on exact arithmetic [17], although the performance can be several orders of magnitude slower. Other works propose the use of hybrid techniques which use voxels at a coarser level and geometry below the voxel resolution [5,6]. However, the extraction of features can be problematic sometimes or they are resolved with a complex approach.

Further, a large number of recent papers about Boolean operations between solids are focused on the visualization of the result [4,18] or an approximate evaluation [19–22]. This is because the performance is critical in modern CAD systems [23], and the amount of data to be processed can be very large. Approximations can be made at different levels, such as arithmetic [21], surface definition and representation [19,5,24,25,23], and visualization instead of evaluation [4,18,26]. The purpose of this last approach is not valid for the evaluation of Boolean operations, because no valid geometric data can be recovered from the GPU after the rendering process. Approximate methods can be desirable in some applications such as collision detections, biomedical analysis or video games. However, there are applications that do need an exact result instead of an approximation, such as architectural design, industrial design, CAE, machining and prototyping, among others.

3. Background

The proposed algorithm is based on some classical approaches presented by Requicha et al. [11], Hubbard [7] and others. It operates directly over the triangles from the meshes, therefore it does not need a complete winged-edge data structure, which is the classical B-Rep representation for polygonal solids. The algorithm has three main steps. The first step consists of intersecting the two solids involved in the Boolean operation. The second step performs the classification of the triangle set resulting from the previous step. The final step selects a triangle set and builds a new solid.

First, a decomposition of each solid is performed in order to obtain a resulting triangle set that meets several conditions: each triangle of a solid must not intersect with other triangle, and it must be completely inside, completely outside or completely on the boundary of the other solid. To summarize the whole process in a sentence, each triangle from one object is intersected with the surface of the other solid, and the appropriate tessellation is performed when there is an intersection. When the triangulation of each solid is adapted to the intersection regions with the other solid, the classification of each triangle is performed. Every triangle from solid *A* is logically on the boundary of *A*, and the same applies for *B*. To test whether a triangle from *A* is inside, outside or on the boundary of *B*, a point-in-solid test is carried out using the triangle barycenter. This simplification is based on the fact that every triangle is entirely contained or not in the solid to be tested, due to the triangulation performed in the first step. The result of this test determines the classification of the triangle with respect to the other solid, that is, inside, outside or on the boundary. When all triangles have been classified, a subset that meets several conditions can be selected for each Boolean operation.

One of the main objectives of the algorithm is accuracy. Several studies have focused on the approximate evaluation of Boolean operations [19,15,16,5,24,25,23]. These techniques have a limited precision, although they usually have a good performance. Their main weakness is the loss of geometric detail and features. The proposed algorithm does not make any simplification in order to get an exact result. The rest of the systems tested in this work are also designed to give exact results. A contribution of this work is to propose an algorithm that is accurate and also fast.

Robustness is also a main objective of the proposed method. Geometric computation software tends to fail due to complex data structures and a large number of special cases of geometric operations. As stated by Hoffman [27], there are several layers in geometric computation software. These layers range from very simple computations (vector addition, inner product, etc.) to higher level operations like face–face intersection, topology management, and ultimately component operations such as a Boolean operation between meshes. All these operations can have their singular robustness issues, which logically affect the overall robustness of the system. Lower levels must be absolutely reliable. In this sense, the main limitation is the numerical precision. To alleviate this, exact arithmetic could be used, but with the drawback of a noticeable performance loss. Both our system and others like CGAL can benefit from the use of extended precision. Higher-level computations (like triangle–triangle intersections) present other robustness issues, most of them due to special or degenerate cases. The proposed implementation uses intersection tests which are designed to handle all possible special cases (segment–triangle, triangle–triangle, ray–octree, etc.).

The last cause of robustness problems is topology management. CAD systems are among geometry software with the highest complexity. Classical approaches use a winged-edge based structure for representing polyhedral solids [10]. The more sophisticated and informative the data structures, the less reliable the resulting code tend to be [27]. For a complete winged-edge structure or similar, an inconsistency in the adjacency data is a source of severe failure for the Boolean operation algorithm. Our algorithm does not maintain explicit information about edges, only vertices and triangles. Therefore, the consistency of the structure is very straightforward to maintain. This simplification reduces the robustness problem, although some additional topological tests have to be performed during the evaluation of the operation (e.g. check for isolated vertices). However, these additional steps are straightforward. In any case, any problems that may arise will never produce an abnormal termination of the software.

The method uses an adaptive octree that accelerates spatial queries. This structure also facilitates the division of the geometry into subsets for distribution among several threads or nodes in a cluster. In several steps of the algorithm the triangles are handled independently of the rest of the mesh. This allows us to perform several operations on the mesh in a parallel manner, using a multicore implementation or even an out-of-core approach for working with very complex solids. It is also possible to treat a part of the mesh while the rest is stored on secondary memory (in this case the octree must be partitioned and stored also). This helps to overcome some memory limitations which other methods suffer, allowing us to handle very large meshes.

In addition to the above, this method does not perform any domain transformation, such as using volumetric elements, or any feature extraction. This is because no geometric detail is lost during the process. The tessellation of every triangle is maintained until the last step of the algorithm, and the connectivity between triangles is reconstructed during the tessellation. This method works optimally with 2-manifold meshes connected at the vertices, which are known as indexed face sets. However, unconnected meshes (triangle soups) and many non-manifold meshes can be used also. The connectivity information is optional for a triangle mesh, although it accelerates the classification of entire zones of the surface, as will be shown later in the paper. The result of the Boolean operation does never include elements like isolated vertices or edges, although it can still produce a non-manifold result. This includes loose triangles or slivers, which are eliminated by using regularized operations [28]. This only means the modification of the conditions used to select the triangles from each tessellated mesh.

4. Preliminary steps

The first step of the algorithm consists of the building of the data structure for the B-Reps. Each solid is represented by a triangle mesh, which is stored as an indexed face set. In this way, there is an indexed list of vertices and a list of triangles which reference each vertex using its index. This is one of the simplest forms for representing a triangle mesh, and allows us to share the vertices among neighboring triangles (see Appendix for more details).

The algorithm can also operate with triangle soups, that is, sets of isolated triangles that do not share their vertices. But in this case, the algorithm will be unable to group triangles with the same status. This neighboring information accelerates the classification of entire zones of the surface, as will be shown later in the paper. Therefore, in order to maximize the performance, the triangle soups must be transformed into indexed face sets. It must be noted that the algorithm is able to evaluate a Boolean operation with triangle soups, but in this case the performance is suboptimal. Also, many non-manifold meshes can also be used with the algorithm. For example, if an edge is shared by more than two triangles, the neighboring information will be incomplete. This is because no explicit representation of edges is used and the faces connectivity information is limited to a maximum of three neighbors (this information is embedded directly into the triangles). In this case the algorithm will still work, but using a greater number of classification zones.

To transform a triangle soup into an indexed face set, a vertex welding must be carried out. By using a spatial sorting, vertices with the same coordinates will be stored consecutively in a list and duplicates can be eliminated in a straightforward manner. The comparison of the vertices is made using a tolerance value (epsilon). This tolerance value must be chosen according to the numerical precision of the input data. At this point, the set of vertices is indexed and the triangles are modified to use indices to vertices.

Another preliminary step is the setup of connectivity information between faces. This information is the link between triangles and determines the neighbors of each face of the mesh. This allows the performing of a fast topological query in order to establish groups of triangles that define a zone of the surface with the same classification status. The establishment of zones is carried out later. In order to perform this step, the meshes must be converted to indexed face sets. If the Boolean operation is carried out using any unprocessed triangle soup, then no classification zones can be determined for that mesh. This severely affects the overall performance. The last preliminary step is the building of the octree. This data structure accelerates the spatial queries needed during the Boolean evaluation. It also allows us to decompose the space and to distribute the data among several threads (or nodes in a cluster). Several parts of the algorithm can benefit from a parallel execution. The octree stores in each leaf node the identifiers of the triangles of each mesh that intersect the volume represented by that node. The octree is the same for both meshes, so it is adjusted to cover all triangles involved in the Boolean evaluation. Two major steps are accelerated by the use of the octree: the triangle–triangle intersection test and the point-in-solid test. The construction of the octree is performed incrementally. At first, it is constructed to include only those triangles which are directly involved in the intersection of the meshes. Thus, a node from the octree is divided only if it contains triangles from both meshes (see Fig. 1). This optimization accelerates the construction and the traversal of the octree for the first part of the Boolean evaluation. Later, if the point-in-solid test is needed in the triangles classification step, the octree must be completed in order to include all of the triangles from both meshes, whether they are involved in the intersections or not. This will be explained further.

The octree can be configured in order to achieve a good balance between the building time of the structure and the performance of the spatial queries used in several steps of the Boolean evaluation. The memory usage of the octree is also affected by this configuration. In general, building the structure requires $O(N\log(N))$ for N triangles. The construction of the octree begins with the common axis-aligned minimum bounding box of the two solids. Each node is subdivided until the termination condition is met. This is determined by two parameters: the maximum depth of the octree, and the maximum number of triangles per leaf node. The two conditions apply, but the depth of the octree has the highest priority. Generally speaking, with a depth of 8–10 and a limit of about 50 triangles per node the performance is near optimal, having a reasonably low calculation time for the structure. However this ultimately depends on the topology of the meshes.

5. Mesh refinement

The mesh refinement process is the first main step of the evaluation of a Boolean operation. The goal is to obtain a resulting triangle set that meets the following conditions: each resulting triangle of the decomposition of a solid must not intersect other triangle from the other solid. Furthermore, each resulting triangle must be completely inside, completely outside or completely on the boundary of the other solid. This resulting triangle set allows us to classify every triangle from each solid with respect to the other solid. These classifications can be performed using a single point from each triangle; more specifically, the barycenter is used for obtaining each triangle inclusion state. During the mesh

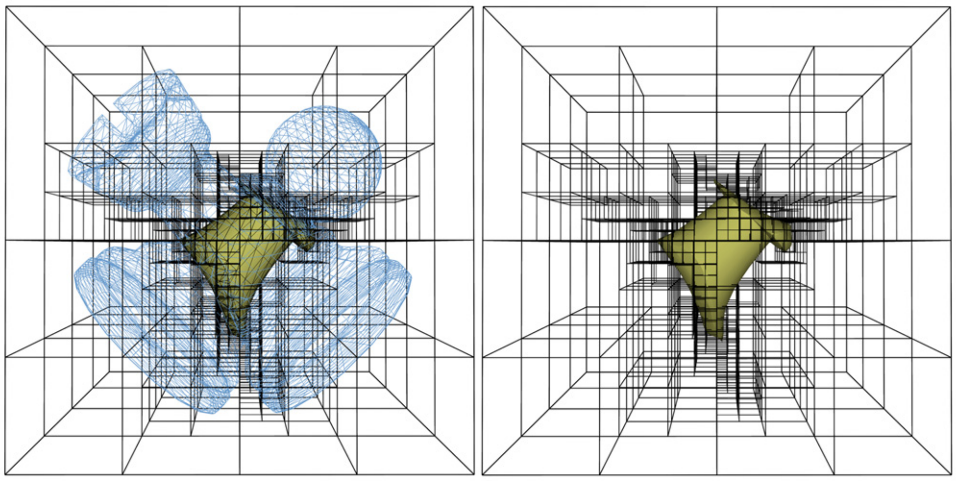


Fig. 1. Example of the octree used in a Boolean intersection. At first, only the nodes that have triangles from both meshes are subdivided.

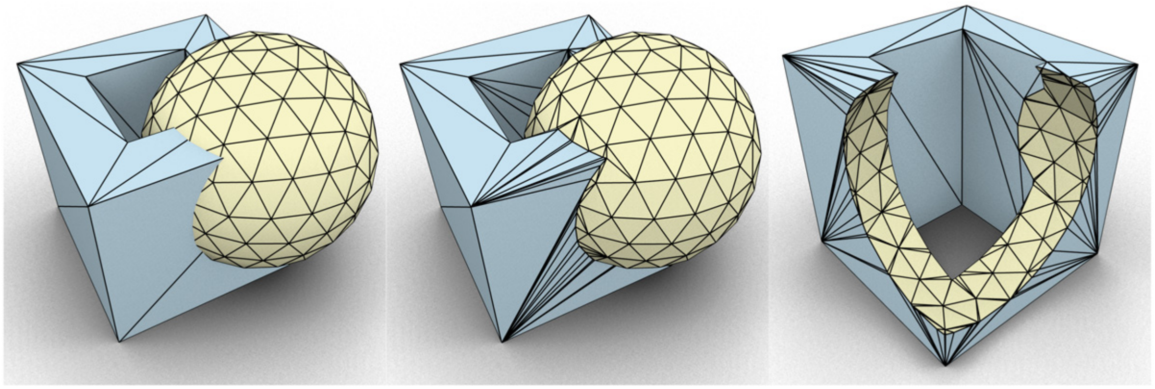


Fig. 2. Mesh refinements used for a Boolean difference.

refinement each triangle from one mesh is intersected with the boundary of the other mesh. With this process two new meshes are obtained. The triangulation of these meshes is adapted to the intersection zone between the two original solids. Fig. 2 shows a sample.

With each triangle–triangle intersection new vertices are introduced into the B-Rep structure. This applies for both meshes during the intersection process. These new vertices logically belong to the frontiers of both meshes, and they are the base for the new triangulations. Each triangle affected by an intersection must be decomposed into new triangles (and vertices). The bottleneck of this stage is the triangle–triangle intersection algorithm, because it must be performed for every combination of two triangles from both meshes. This process is accelerated by the use of the octree. At this point, the octree stores in each leaf node the identifiers of the triangles that intersect that node. When a triangle–triangle test is to be performed, the nodes of the octree where the triangle to be intersected is included have a list of triangles from the other mesh that are also included in those nodes. With this optimization the triangle intersection step cost is dramatically reduced from $O(n * m)$ to $O(k)$, being n and m the number of triangles of the two meshes, and k the average number of triangles per leaf node (see Fig. 1).

Different methods are used for subdivision of triangles. These essentially focus either on establishing section lines [8] or cutting faces [9]. Our algorithm belongs to the second type, as it sets the intersection points for each pair of triangles in order to divide them both afterwards. Given two triangles T_1 and T_2, first we establish whether both triangles are coplanar or not. If the triangles are coplanar the algorithm can be simplified to a 2D intersection, which is more efficient. In any case, we check the intersection of every edge of T_2 on triangle T_1 [29], and then perform the same operation but reversing the roles of T_1 and T_2. For each pair of triangles our algorithm will carry out two main operations. First, an intersection test is performed and the intersection points are calculated (if the test is positive). Second, each triangle is tessellated into new triangles. The 3D case is significantly more complicated. In our algorithm, an edge–triangle intersection test is performed by checking the sign of the vertices of the edge with respect to the triangle. This sign is actually the sign of the volume of a tetrahedron formed by the vertex and the triangle. The sign is $+1$ if the vertex is under the plane that contains the triangle, -1 if above. The sign can be 0 if the volume of the tetrahedron is also 0 (this means that the four points are coplanar). If the signs of the vertices of the edge to be tested for intersection are different, the edge intersects the plane that contains the triangle. Also, it is necessary to determine if the intersection point is inside the triangle using barycentric coordinates. If a sign is 0, the corresponding end of the edge is on the plane of the triangle, and if both signs are 0 the edge and the triangle are coplanar [30].

Given an edge $E = (a, b)$ and a triangle $T = (V_0, V_1, V_2)$, E intersects T if the following conditions are satisfied: First, the vertices of the edge have different signs or they are equal to zero with respect to the triangle. Second, one vertex (a or b) is inside the angular zone of the tetrahedron formed by the triangle and the other vertex (b or a). Fig. 3 shows the construction of this tetrahedron. In this sample, an *original face* of the tetrahedron is a face that contains vertex a.

In order to determine whether a point is inside the angular zone of a tetrahedron, it is enough to check the three signs of that

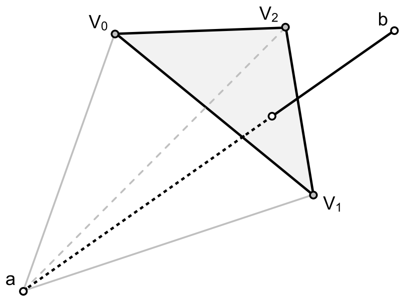


Fig. 3. Vertex b is inside the angular zone of tetrahedron $[V[0], V[1], V[2], a]$.

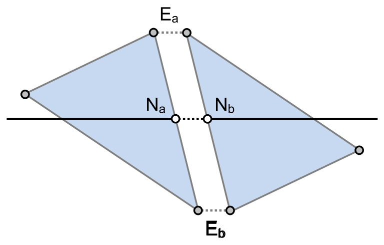


Fig. 4. Neighbor triangles, which share vertices E_a and E_b, are intersected by the same segment. This produces two new vertices N_a and N_b that are actually the same, and must be joined in the data structure. For the sake of clarity the triangles are displaced from each other.

point with respect to each original face of the tetrahedron, that is, $vol0 = svol(a, V[0], v[1], b)$, $vol1 = svol(a, V[1], V[2], b)$, $vol2 = svol(a, V[2], V[0], b)$. Function *svol* calculates the signed volume of the given tetrahedron, which can be -1, $+1$ or 0 for a degenerated tetrahedron. The three signs must be the same (all of them $+1$ or all of them -1), except for the unusual case of zero (vertex–vertex, vertex–edge and coplanar intersections). This approach is similar to the algorithm described in [30]. Several situations and special cases can occur in the intersection between an edge (segment) and a triangle in 3D [31,8,32].

It must be noted that the edge–triangle intersection algorithm is designed to provide additional information that will be used for the triangle division. This information is linked to the triangles, and allows the welding of the new coincident vertices. Fig. 4 shows an example of neighbor triangles which are intersected by the same segment (an edge from the other mesh). These triangles share two of their vertices and therefore they share an edge. As the triangles from each mesh are treated separately, two intersections are produced and so two new vertices are created. However, these new vertices are actually the same, because they occupy the same position in space and are the intersection with the same edge of the same triangle. Therefore, one of them must be replaced by a reference to the other vertex. Since the matching vertices are located on the same edge and they are produced by the intersection with the same triangle of the other mesh, it is straightforward to weld them. Furthermore, to increase the robustness a floating-point comparison is carried out between the vertices using a tolerance value. Also at this point, the neighboring information of the new triangles is calculated.

Once the intersection points and their associated configuration have been determined, we will proceed to insert them in the list of vertices of the triangle to be tessellated. However, if an intersection point coincides with any of the vertices of a triangle (original vertices) it must not be added to the list of triangle vertices in order to avoid repetitions. In this case, a reference is used. The tessellation of a triangle is performed carefully, and it depends on the way the intersections have occurred. To ensure the robustness, all possible cases of the edge–triangle intersection have been taken into account, including the coplanar cases. Fig. 5 shows an example of a coplanar case.

The division of a triangle depends on the configuration of the intersection points calculated before. It must be noted that the meshes refinement process requires a double pass through the tessellation step. This is because if a triangle intersects with another triangle causing its subdivision into new ones, those new triangles can affect the tessellation of the first. Fig. 6(a) shows an example of this. The blue triangle must be tessellated again in order to add an edge along the intersection with the red triangle. This ensures that the interior of every triangle produced by the tessellation of the blue triangle will not intersect with the interior of the red triangle. The same process must be performed for the red triangle.

6. Mesh classification

The classification of the new triangles is the second main step of the Boolean evaluation process. There are two triangulated meshes as a result of the previous step. At this point, these meshes are conveniently tessellated so that every triangle from both meshes is totally inside, outside or on the boundary of the other mesh. This allows us to determine the classification of each triangle using only a single point located on the triangle. To test whether a triangle from mesh A is inside, outside or on the boundary of mesh B, a point-in-solid test is carried out using the barycenter of the triangle. The goal is to classify every triangle from both meshes with respect to the other (with an in, out or on state).

The previous step of the Boolean evaluation (tessellation of the meshes) performs an early Boolean classification of the new

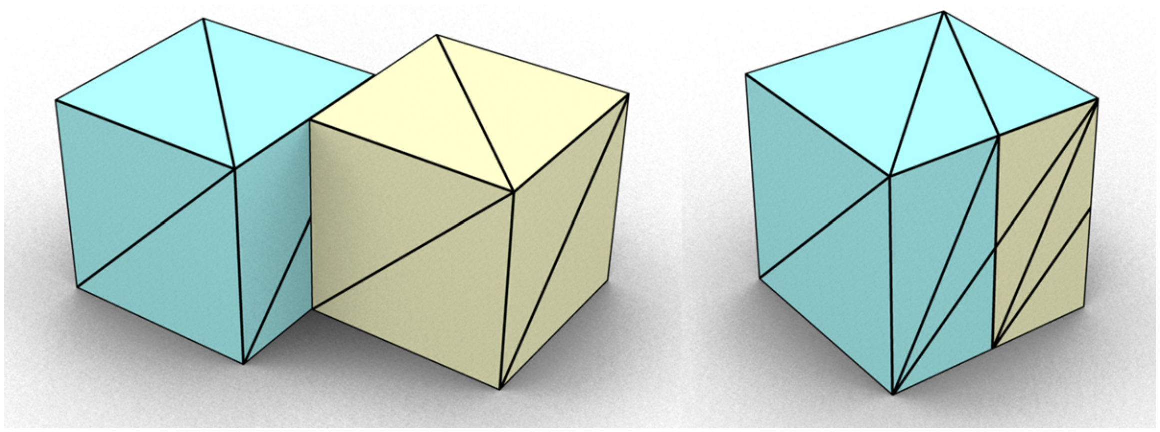


Fig. 5. Example of a tessellation with a coplanar case.

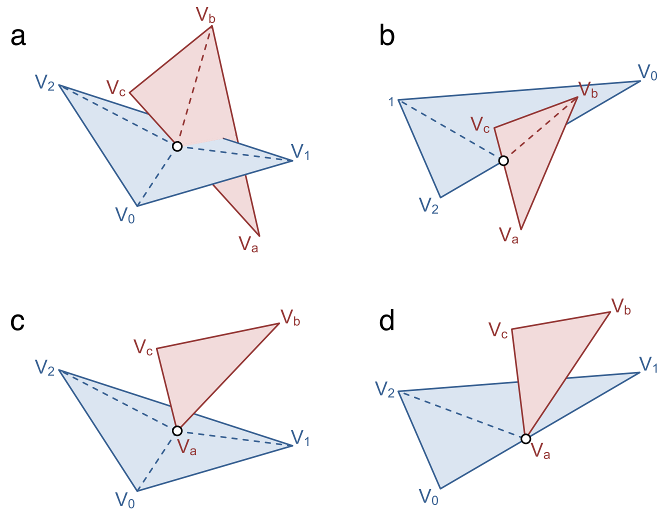


Fig. 6. Examples of triangle division.

triangles. When a triangle from a mesh is intersected by the surface of the other mesh, we can calculate the common edges that belong to the intersection line shared by both tessellated meshes. This line can be a simple point for some configurations. By using the lines of intersection, we can also calculate neighboring information between triangles from different meshes. Thus, an intersection zone is the set of triangles which have at least an edge belonging to an intersection line. Each intersection line has associated an intersection zone, which includes triangles from both tessellations. This information allows us to classify every triangle that belongs to the intersection zone in a straightforward manner, because it has neighbors from the other mesh that define the position and orientation of the intersecting surface. Additionally, there are triangles of a solid that are not divided because they are totally inside or outside the other solid from the beginning. Those triangles are located together in groups which represent a part of the surface of the solid. These zones of the surface are normally surrounded by the triangles involved in the tessellation, which are located over the intersection line between the two solids. Using the neighboring information of the triangles, we can build groups of triangles that will be separated only by intersection lines. Fig. 7 shows the tessellations carried out for two intersecting solids. The figure on the right shows how each group is painted with a different color, representing a part of the tessellated mesh whose triangles have the same classification with respect to the other solid.

However, there are configurations that do not allow to directly classify some parts of the surface of a mesh. If there are isolated parts of a mesh (bodies) that do not intersect the other mesh, their inclusion state cannot be determined without additional information. In this case, for a Boolean operation between meshes *A* and *B*, if mesh *A* does not intersect mesh *B*, we cannot establish whether *A* is inside *B* or *B* is inside *A*. Moreover, if the meshes are triangle soups, we cannot use any neighboring information to determine which parts of the surface are delimited by intersection lines. In this case, only triangles that belong to the intersection zone can be classified in a straightforward manner. This is the main reason to convert a triangle soup to an indexed face sets prior to performing the Boolean evaluation.

The configurations cited above force the algorithm to perform an additional step. Every triangle, surface part or isolated body that cannot be classified using a direct contact with the other solid, must be classified using a point-in-solid. For a group of connected triangles that is delimited by one or more intersection lines, a single point from that part of the surface is used to classify the entire part with a point-in-solid test. The barycenter of any of those triangles can be used. This is because that group of triangles is totally inside, outside or on the surface of the other solid. Fig. 7 (right) shows how each group is painted with a different color, representing a part of the solid that can be classified as a single point. This simplification dramatically increases the performance.

Several approaches can be used to perform the point-in-solid tests [33]. In a previous work [34] we used the simplicial covering of the solids. This method is very fast when it is implemented on GPU and the number of tests is high enough. Further, it does not need any spatial classification structure. However, as the number of tests to be performed in this work is rather low, we use the Jordan Curve Theorem [35]. This method performs a ray-surface intersection test in order to obtain the parity of the number of intersections. It needs the octree calculated in previous steps for performing as few ray-triangle intersections as possible. This method is more efficient than the previous when there are few point-in-solid tests to carry out. However, the octree calculated in the meshes refinement step does not include all the triangles from both meshes, only those are near the intersection zones. In order to use the octree with the Jordan-based point-in-solid test,

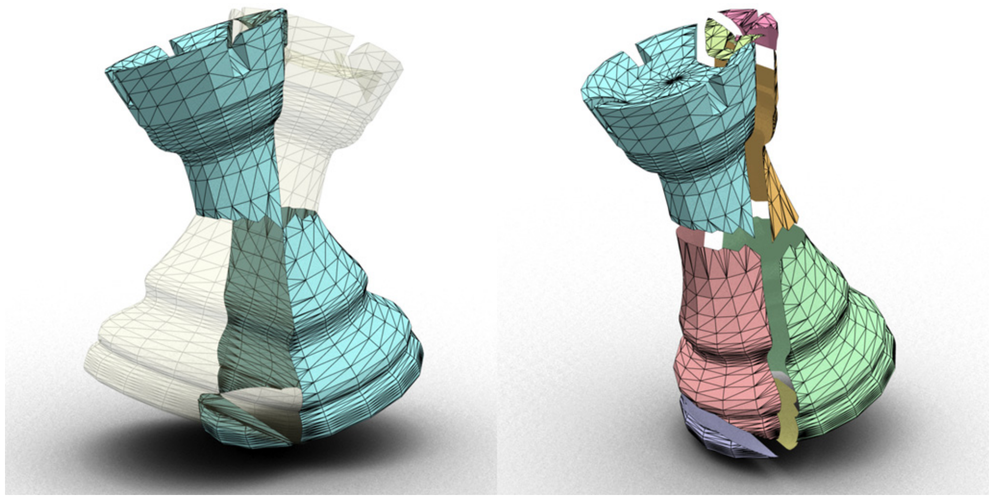


Fig. 7. A blue rook—yellow rook Boolean operation (left). Groups of triangles obtained with the tessellation of blue rook (right). Each group is represented with a different color and has the same inclusion state for all of its triangles. For the sake of clarity, each group is displaced from the center of the solid. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

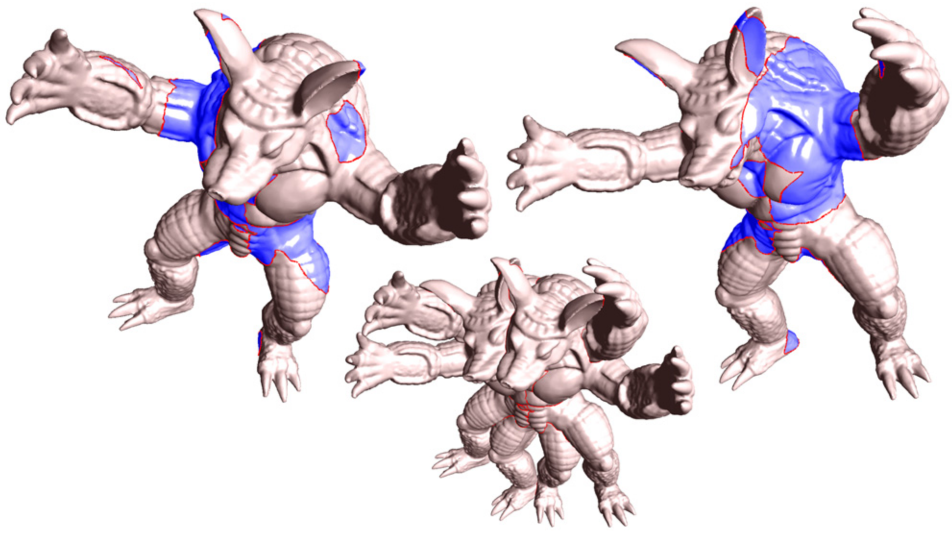


Fig. 8. A view of the classification of two instances of the armadillo (150 000 triangles) in a Boolean operation. The zones of each surface that are inside the other solid are marked in blue. Each intersection zone is delimited by a perimeter marked in red. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

it must be completed to include all of the triangles. The octree implementation used in our tests can be calculated in two phases in an incremental manner. This optimization allows an important performance gain and less memory usage when the Boolean operation does not require any point-in-solid test.

7. Boolean evaluation

At this point the meshes used in the Boolean operation are tessellated so that every triangle from each solid is completely outside, completely inside or on the boundary of the other solid. Also, all the triangles from each solid are also classified with respect to the other solid. This last step consists of selecting a triangle set that meets several conditions and building a new mesh with that triangle set [36]. This set is the result of the Boolean operation. Fig. 8 shows the classification of the surfaces of the two meshes prior to performing the final selection of triangles.

Let T_A be the set of triangles from the tessellation of the mesh A, and T_B the set of triangles from the tessellation of the mesh B. Let T^{-1} be a set of triangles with their normals inverted and $n(t)$ the normal vector of the triangle t. Each Boolean operation is defined with several conditions that the triangles must verify. An additional condition is also included in order to get a regularized result [28]: let W be a set (universe) and T a topology on W (the collection of all open subsets of W). In the topological space (W, T) a subset X of W is a closed (regular) set if it equals the closure of its interior: $X = kiX$. Thus, the regularized Boolean operators imply the selection of the following sets of triangles:

- $\{A \cap^* B\} : \{T_A \text{ in } B\} \cup \{T_B \text{ in } A\} \cup \{T_A \text{ on } B / n(t_i) = n(t_j); t_i \in T_A, t_j \in T_B\}$. Regularized intersection.
- $\{A \cup^* B\} : \{T_A \text{ out } B\} \cup \{T_B \text{ out } A\} \cup \{T_A \text{ on } B / n(t_i) = n(t_j); t_i \in T_A, t_j \in T_B\}$. Regularized union.
- $\{A -^* B\} : \{T_A \text{ out } B\} \cup \{(T_B \text{ in } A)^{-1}\} \cup \{T_A \text{ on } B / n(t_i) \neq n(t_j); t_i \in T_A, t_j \in T_B\}$. Regularized difference.
- $\{A \Delta^* B\} : \{T_A \text{ out } B\} \cup \{(T_A \text{ in } B)^{-1}\} \cup \{T_B \text{ out } A\} \cup \{(T_B \text{ in } A)^{-1}\}$. Regularized symmetric difference. It is the same as $(A -^* B) \cup (B -^* A)$.

For example, to evaluate a regularized Boolean difference, the following triangle set is selected: the triangles from A that are outside B, the triangles from B that are inside A (reversing their normals), and the triangles from A that are on the boundary of B and have a different normal vector. With this triangle set we build a B-Rep that represents the new solid.

The final result will be obtained by copying the triangles that meet the conditions of the Boolean operation onto a new mesh. Note that all Boolean operators use the same refined meshes obtained after the tessellations of the original solids. The only difference is the last step of the algorithm, that is, the final triangle set selection. For this reason, all possible Boolean evaluations (regularized union, intersection, difference and symmetric difference) can be obtained with almost the same processing cost as a single one, because the last step is very fast (only a selection and a triangle set copy operation). This can be useful in some situations, such as an optimized CSG evaluation.

After the selection of the triangles set for building the final result, an additional check must be performed in order to assure that the mesh will be regular (closed). Because of the data structure used for this algorithm, no dangling edges or vertices can result from a Boolean operation. Our method focuses only on triangles, and the vertices are inserted or shared in the data structure only when there is at least one triangle that needs it. However, arithmetic precision issues can arise with the intersection and tessellation of very small or stretched triangles, especially when not using extended precision. In order to solve these cases, a straightforward connectivity check is performed on each triangle of the resulting mesh. Only space-filling volumes are allowed (closed bodies with a finite boundary). Thus, any triangle with less than three neighbors will be removed from the mesh. This step must not be carried out if the operands of the Boolean evaluation are open meshes, because they are not regular from the beginning. Logically, in this case the correctness of the result cannot be guaranteed.

8. Results and discussion

We have tested our Boolean operation method with several triangulated meshes. They have different polygon quantities and different topological properties. All of the meshes have a 2-manifold representation. The implementation of the algorithm has been completely written using C++ and OpenMP. We have also tested other systems in order to compare the quality of the results and the overall performance: CGAL [37], the GNU triangulated surface library (GTS) [38], MicroStation V8 and 3ds Max 2010. All

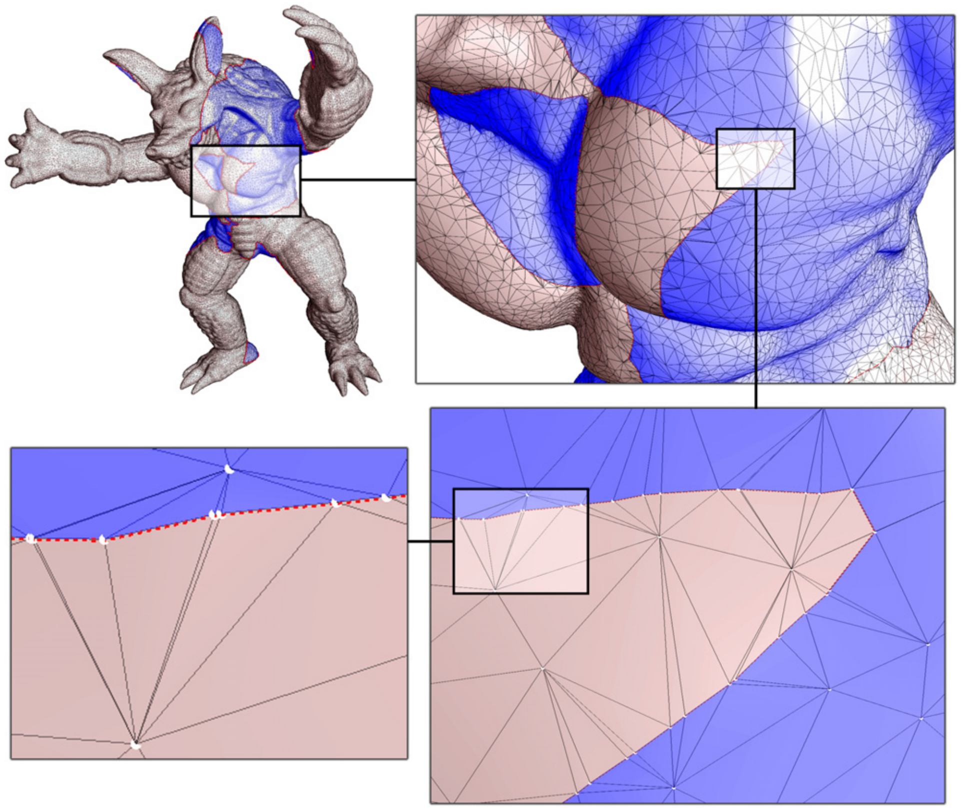


Fig. 9. Detailed view of the tessellation of one instance of the armadillo. As it can be seen, the narrow triangles are treated correctly. The dashed red line marks the perimeter of the intersection zones. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

these tests have been carried out with a Dual Intel Xeon X5550 2,66 GHz with 12 Gb of memory.

The data structure used to represent the meshes is as follows. To store the indexed face set, the vertices are stored as an array of vectors. The triangle list is an array of integer-based indices which reference the associated vertices of each triangle. During the mesh refinement new vertices and triangles are inserted into the structure. There may be duplicated vertices which must be removed in order to ensure that the resulting mesh will be a valid representation of a solid. When removing duplicated vertices their linked triangles must be adjusted in order to reference the correct vertices. Duplicated vertices are removed after the tessellation step, in order to improve the performance.

Several parameters can be adjusted in order to get optimal results. Floating point precision is very important for maintaining the robustness of the method. This is because nearly zero point-in-solid tests can determine that a point and the associated triangle are exactly on the surface. Also, if the precision for the epsilon used in the operations is insufficient, problems can arise when intersecting coplanar faces from both solids, or when working with very narrow triangles. Fig. 9 shows an example of narrow triangles resulting from the tessellation of one instance of the armadillo. In the tests we have used double precision arithmetic, which provides correct results and has a good performance. For operating with more complex meshes extended/exact precision arithmetic could be needed. If it is the case, the performance would be much slower. However, in our tests double precision was sufficient to get correct results. This also applies to CGAL and GTS. All the systems tested provide an exact result, and they do not perform any simplification that leads to a loss of geometric detail.

The octree can be configured in order to achieve a good balance between the building time of the structure and the performance of the spatial queries used in several steps of the Boolean evaluation. The memory usage of the octree is also affected. The octree stores in each leaf node the identifiers of the triangles of each solid that intersect the volume represented by that node. The construction of the octree is performed using several threads of execution. Each node of the octree is subdivided until the termination condition is met (then the node is closed). During this process each thread is always working on a node that is not fully subdivided and has not reached the termination condition (the node remains open). With an octree depth of 8–10 the performance is very good, having a reasonably low structure calculation time. Fig. 10 shows the octree used in the test with the armadillo.

Table 1 shows the performance times of the tests, which include every type of Boolean operator. Each result is calculated as the average of the time of all Boolean operations: union, intersection, *A*–*B* and *B*–*A*. Each test has been performed with two instances of the same mesh, each one with a different position in space. The overlapping between meshes has been maximized, in order to achieve a high number of triangle intersections. The symmetric difference is not available as a single operation in MicroStation and 3ds Max, therefore is not included in the tests. It must be noted that the symmetric difference has the same performance as the rest of the operations in our system.

As expected, the algorithm depends on the number of polygons. The times of the proposed method include the entire process as described in previous sections, including the creation of the octree, the calculation of the neighboring data of the triangles, the mesh refinement, the triangle classifications, the Boolean evaluation, the regularization test and the final mesh creation. The implementation takes advantage of several CPU threads at different stages of the algorithm. As a consequence, it benefits from a large number of triangles, since the higher the number of calculations to be carried out, the better the performance gain factor over a single-threaded version.

Now, a comparison between the tested systems is presented. We focus on the quality of the results and on the performance and

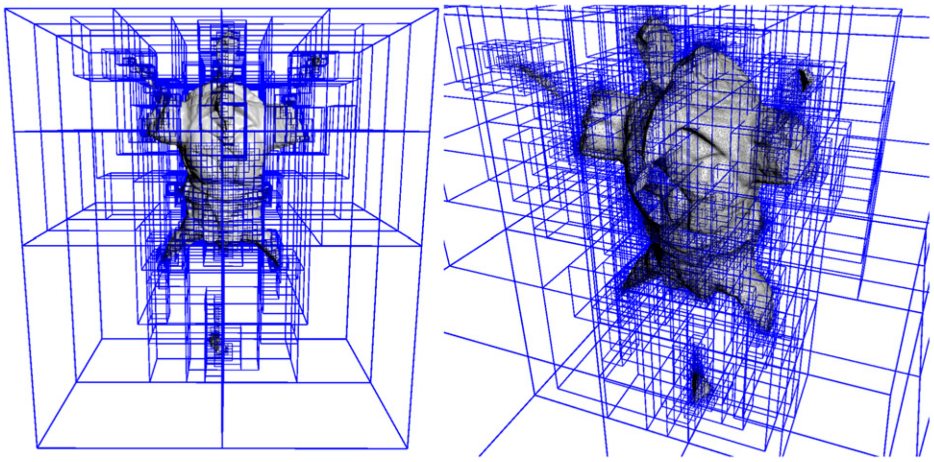


Fig. 10. The octree used for the test of the armadillo (the Boolean intersection is shown). For the sake of clarity, only the nodes that contain triangles from both meshes are shown. It can be noticed how the octree is adapted to the intersection between the two solids in the first step of its construction.

Table 1
Time in seconds for several Boolean operations between two instances of the same solid. Each result is calculated as the average of the time of all Boolean operations: the union, the intersection and the differences (*A*–*B* and *B*–*A*). The new method has been tested with several multithreading configurations: from 1 thread (th) to 16 threads.

Object	Triangles per mesh	MicroStation V8-XM	3ds Max 2010 (ProBoolean) (s)	CGAL	GTS (s)	New			
						1th (s)	4th (s)	8th (s)	16th (s)
Rook	2 256	2.06 s	0.35	4.08 s	0.07	0.13	0.10	0.10	0.11
Bishop	5 814	8.15 s	0.51	8.85 s	0.13	0.19	0.13	0.11	0.10
Ring	16 384	47.5 s	0.82	25.24 s	0.36	0.38	0.27	0.24	0.22
Golfball	46 205	276.38 s	3.96	73.37 s	0.94	1.00	0.59	0.44	0.34
Armadillo	150 000	Fail	5.54	Fail	3.84	2.71	1.46	0.95	0.68
Sculpture	277 512	Fail	8.47	Fail	6.29	4.39	2.15	1.34	0.89
Dragon	871 414	Fail	84.28	Fail	23.01	12.64	6.48	4.07	2.72

the robustness of all the systems. We have obtained a wide variety of results. Two commercial packages have been used: MicroStation V8 and 3ds Max 2010. MicroStation gives correct results with low-complexity meshes. However it is unable to perform Boolean operations with larger solids, aborting the operation with an error. Moreover it is quite slow, and the performance is severely affected by an increment of the number of triangles to be processed. 3ds Max has been tested with the ProBoolean option, which is more advanced than the standard Boolean option. It is quite fast, although some operations ended with incorrect results. It is worth noting the failure with the ring test, which ended with an open mesh. This object is generated procedurally with evenly distributed vertices and without narrow triangles, so no robustness problems were expected.

We have also used two geometric processing libraries: CGAL [37] and GTS [38]. CGAL provides tools for processing polygons using arbitrary precision arithmetic for Boolean evaluations. Although it is designed to allow the use of exact arithmetic, this has not been used in order to do a performance comparison with the other systems on equal terms. With a topologically correct input, the results are good. The performance of CGAL is acceptable for low-complexity tests, but rather slow with a medium complexity. For the most complex tests it failed with every type of Boolean operator. On the other hand, GTS performs very well, and it can handle large meshes correctly. The results have a very high quality. It is very strict with the topology, so any inconsistency of the structure will cause the cancelation of the Boolean operation. It uses an *AABB* tree to accelerate the intersection algorithms, so this software is fast. However, the *AABB* tree does not allow the division of the geometry into separable subsets for working in parallel.

Our system provides correct results with a topologically correct input. Figs. 11 and 12 show the results of some Boolean operations. The method also works with open meshes and many non-manifold solids, in contrast to other systems. Logically, in any of these cases the correctness of the result is not guaranteed. Fig. 13 shows an example of a non-manifold solid that can be successfully used with our algorithm. On the other hand, the ability to work with open meshes is useful for handling data provided by 3D scanners, or objects built with a non-CAD software which does not guarantee the correctness of the topology. The robustness of our method at a low level is the same as of other systems. CGAL, GTS and our system can benefit from using exact arithmetic with simple computations (vector addition, inner product, etc.), so this aspect does not make a difference. We have used double precision arithmetic with CGAL, GTS and our system. This provides correct results while offering a good performance. All the systems tested are designed to provide an exact result, in the sense that they do not perform any simplification that leads to a loss of geometric detail. Contrary to the above, at a high level the robustness is not the same for all systems [27]. Both MicroStation and CGAL present issues when working with large meshes. This is due to the use of very elaborate data structures which complicate the topology management, so the Booleans operations tend to be more error-prone. Also 3ds Max (using ProBoolean) has robustness problems. Some of the straightforward operations ended with incorrect results, such as the tests of the ring and the armadillo. Our system is more robust at the topology management level than the other systems. The data structure is simple, and its consistency is very straightforward to maintain. However, some topological queries must be performed with a slight additional cost, but as it can be seen in Table 1 the overall performance is very good. Fig. 12 shows the details of some Boolean operations performed with our system.

The performance of the systems tested is very different. MicroStation presents the worst scalability, and it is the slowest system for performing Boolean operations. Since it is CAD software, it is focused on complex data structures aimed for design. It must be noted that it can perform Boolean operations with solids that are more complex than a triangle mesh. This also happen with CGAL,

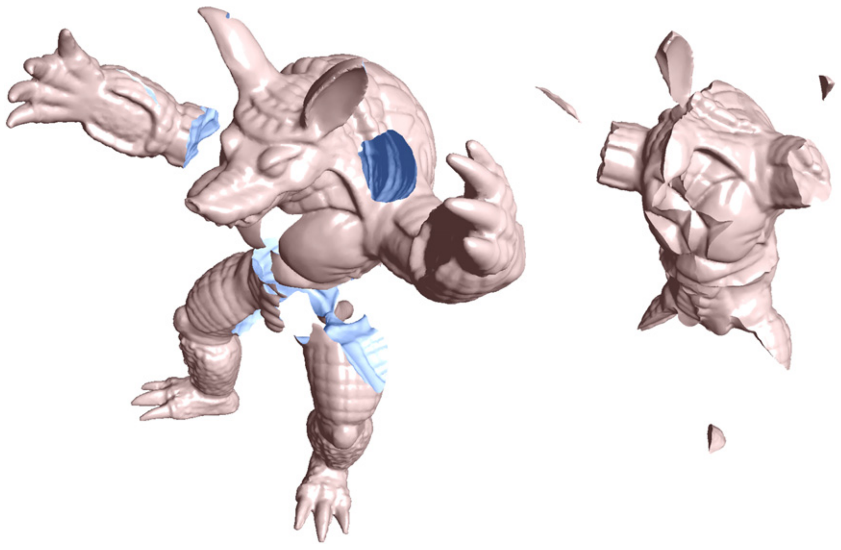


Fig. 11. Examples from the armadillo test (see also Fig. 8). Left: difference (armadillo *A*–armadillo *B*). Right: intersection between armadillo *A* and armadillo *B*.

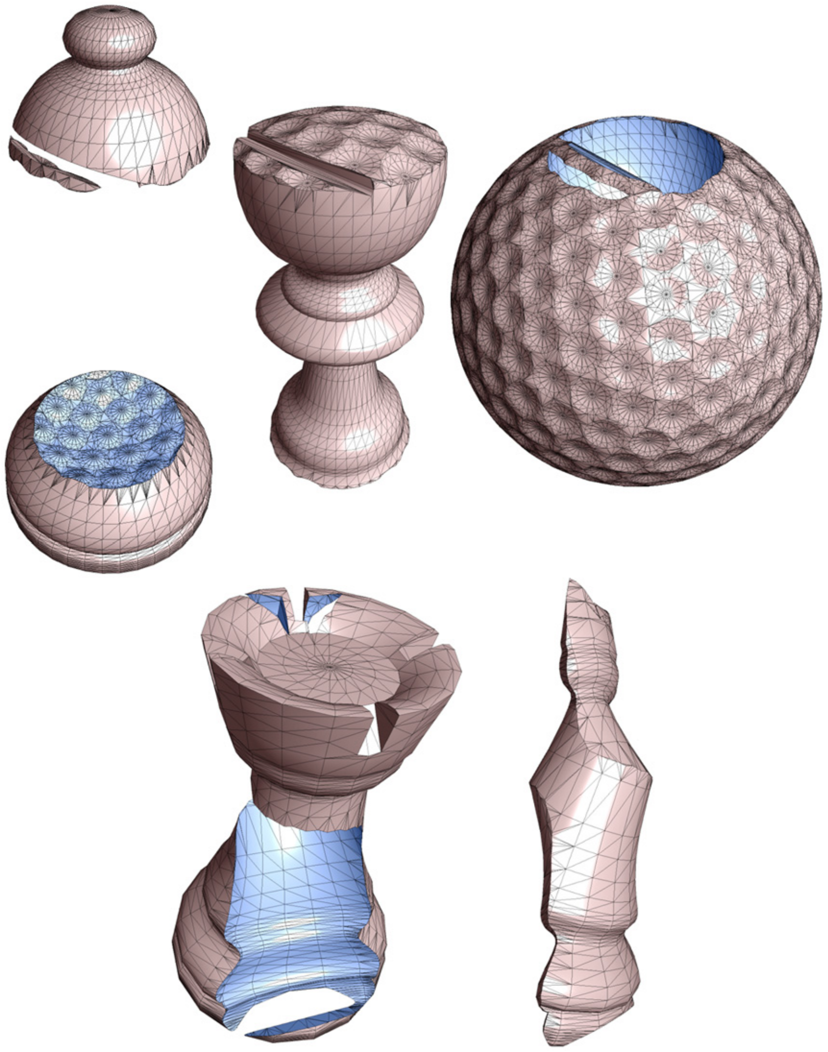


Fig. 12. Some Boolean operations performed with our system. From top to bottom and from left to right: bishop–golf ball, bishop and golf ball intersection, golf ball–bishop, tower *A*–tower *B*, tower *A* and tower *B* intersection.

which is also designed for generality in the use of geometric algorithms. For this reason, the performance of CGAL is not very good. 3ds Max is a general-purpose modeler aimed for design and rendering, and the Boolean operations are not well supported by the basic Boolean commands, which are also quite slow. However the ProBoolean module works better, it is faster and more reliable. This

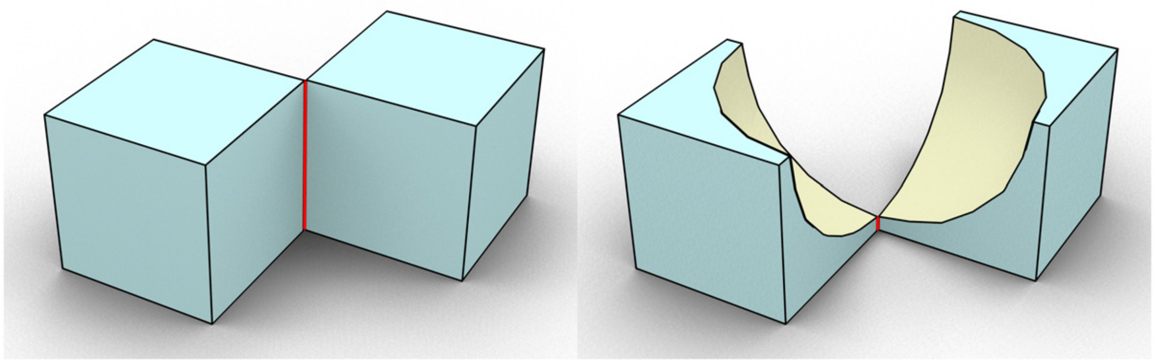


Fig. 13. An example of a non-manifold solid. The red edge (at the center) cannot be shared by all the adjacent triangles. Although this limits the neighboring information, the algorithm can still complete the operation. The solid on the right is the result of the Boolean difference with a sphere. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

is the option that has been tested. GTS and our proposal are the fastest methods, and they must be compared with more detail.

The performance of our method is better for the majority of cases. Because of the nature of the algorithm, the computing time includes all of the Boolean evaluations. Thus, every Boolean operator has the same performance, and we can obtain four resulting meshes at the same time (union, intersection, difference and symmetric difference). This is also true for GTS, because for each solid it also calculates the sets of triangles that are totally inside or totally outside of the other solid. GTS uses an *AABB* tree to accelerate the intersection between triangles. This structure is a bit faster to calculate than the octree of our system. This is the reason GTS is faster with small solids. With medium and large meshes, our method is faster even with only one thread of execution. The octree is worthwhile when the number of spatial queries is highly enough to compensate for the construction time. This also depends on the number of threads used. Additionally, the octree allows us to divide the geometry into separable subsets for working with several threads. It also allows the use of an out-of-core approach for working with very large meshes. The rest of the systems tested cannot take advantage of multithreading, including GTS. Table 1 shows how the benefit of using several threads depends on the number of triangles of the solids. With small meshes there is little work for each thread. Moreover, the octree contains fewer nodes to distribute among threads, and the mutual exclusion objects (mutexes) used to manage shared data adversely affect the performance gain expected. However using multithreading with larger meshes clearly increases the performance. The only drawback is that some nodes of the octree contain more data than others, which affects the workload ratio of each thread. This affects the performance gain factor expected. It must be noted that none of the other systems tested use multithreading.

9. Conclusions

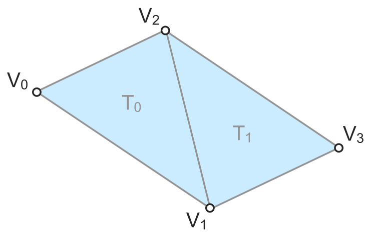
In this paper, we have presented an efficient, accurate and robust method to evaluate regularized Boolean operations on triangle meshes. It is based on a straightforward data structure and on the use of an octree which accelerates the spatial queries needed during the process. Although it is designed to work with 2-manifold solids, it can also provide correct results with some non-manifold meshes. It allows the exact evaluation of the regularized union, intersection, difference and symmetric difference simultaneously. By using an adaptive octree the solids can be divided into subsets for distribution among several threads of execution. This method can also be easily implemented using an out-of-core approach that allows to work with very high-complex meshes and to share the work among the nodes of a cluster. The performance is better that the other systems that also provide an exact evaluation of the Boolean operations.

Acknowledgments

This work has been partially supported by the Ministerio de Ciencia e Innovación and the European Union (via ERDF funds) through the research project TIN2011-25259 and by the University of Jaén through the research project UJA2010/13/08 sponsored by Caja Rural de Jaén.

Appendix

Next, we present the main algorithm steps written in pseudocode and some details about the data structure. Fig. A.1 shows a sample of an indexed face set. In our approach we have also used an array for storing the neighboring information of the triangles. These connections are coded as indices to other faces of the mesh. The first index is for a neighbor on the edge $\{V_0, V_1\}$, the second is for a neighbor on the edge $\{V_1, V_2\}$, and the third for a neighbor on the edge $\{V_2, V_0\}$. A value of -1 means that there is no connectivity for a given edge.



$V = [\ x_0, y_0, z_0\ , x_1, y_1, z_1\ , x_2, y_2, z_2\ , x_3, y_3, z_3\ ]$

$F = [\ 0, 1, 2\ , 1, 3, 2\ ]$

$F_c = [\ -1, 1, -1\ , -1, -1, 0\ ]$

Fig. A.1. Example of an indexed triangle set. V is the array of vertices, F is the array of faces (indices to vertices), and F_c is the array of face connections.

```
BooleanEvaluation (meshes, operator, resultMesh)
  For each m in meshes do
    If m.isTriangleSoup () then
        m.weldVertices ()
      m.calculateNeighbors ()
  octree.setupOnlyIntersectionsNodes (meshes)
  refineMeshes (octree, meshes, refMeshes)
  classifyMeshes (octree, meshes, refMeshes)
  resultMesh = new Mesh
  For each refMesh in refMeshes do
      For each t in refMesh.getTriangles () do
        If checkBooleanCondition
        (t, operator, meshes) then
```

```
              Add t to resultMesh
    regularizeMesh (resultMesh)
  refineMeshes (octree, meshes, refMeshes)
    For each m in meshes do
      refMesh = new Mesh
      For each t in m.getTriangles () do
          triList = octree.getNearestTriangles (meshes, m, t)
          For each nt in triList do
              newTris = triangleTriangleIntersection(t, nt)
              If newTris is not empty then
                 Add newTris to refMesh
                 classifyTrianglesWithIntersectionLine
                 (newTris, nt)
                 Else Add t to refMesh
      Add refMesh to refMeshes
  classifyMeshes (octree, meshes, refMeshes)
    For each refMesh in refMeshes do
      triList = refMesh.getClassifiedTriangles ()
      For each ct in triList do
          classifyUnclassifiedNeighboringTriangles
          (ct, refMesh)
      triList = refMesh.getUnclassifiedTriangles ()
      If triList is not empty then
          If octree.isHalfBuilt () then
              octree.completeFullSetup (meshes)
          For each ut in triList do
              classifyTriangle (octree, ut, meshes)
```

References

[1] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry. Berlin: Springer; 1997.

[2] Krishnan S, Manocha D, Gopi M, Culver T, Keyser J. Boole: a boundary evaluation system for Boolean combinations of sculptured solids. Internat J Comput Geom Appl 2001;11(1):105–44.

[3] Mäntylä M. An introduction to solid modeling computer. Rockville (Maryland): Science Press; 1988.

[4] Hable J, Rossignac J. Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes. In: Proceedings of Siggraph'05. 2005, p. 1024–31.

[5] Pavić D, Campen M, Kobbelt L. Hybrid Booleans. Comput Graph Forum 2010; 29:75–87.

[6] Varadhan G, Krishnan S, Zhang L, Manocha D. Reliable implicit surface polygonization using visibility mapping. In: Symposium on Geometry Processing. 2006, p. 211–21.

[7] Hubbard PM. Constructive solid geometry for triangulated polyhedra. Tech. report CS-90-07. Brown CS. 1990.

[8] Mäntylä M, Tammine M. Localized set operations for solid modelling. Comput Graph 1983;17(3).

[9] Pilz M, Kamel HA. Creation and boundary evaluation of CSG models. Eng Comput 1989;5:105–18.

[10] Yamaguchi F, Tokieda T. A unified algorithm for Boolean shape operations. IEEE Comput Graph Appl 1984;4(6).

[11] Requicha AAG, Voelcker HB. Boolean operations in solid modeling: boundary evaluation and merging algorithms. IEEE Proc 1985;73(1):30–44.

[12] Rossignac JR, Requicha AR. Solid modeling. In: Webster J, editor. Encyclopedia of electrical and electronics engineering. John Wiley & Sons; 1999. Webster.

[13] Ayala D, Brunet P, Juan R, Navazo I. Object representation by means of nonminimal division quadtrees and octrees. ACM Trans Graph 1985;4(1): 41–59.

[14] Laidlaw DH, Trumbore WB, Hughes JF. Constructive solid geometry for polyhedral objects. SIGGRAPH Proc 1986;20(4):161–70.

[15] Kobbelt LP, Botsch M, Schwanecke U, Seidel H-P. Feature sensitive surface extraction from volume data. In: SIGGRAPH'01 Proceedings. 2001.

[16] Lorensen WE, Cline HE. Marching cubes: a high resolution 3D surface construction algorithm. In: SIGGRAPH'87. 1987, p. 163–169.

[17] Keyser J, Culver T, Foskey M, Krishnan S, Manocha D. ESolid: a system for exact boundary evaluation. In: SMA'02. 2002, p. 23–34.

[18] Hable J, Rossignac J. CST: constructive solid trimming for rendering BReps and CSG. IEEE Trans Vis Comput Graphics 2007;13(5):1004–14.

[19] Adams B, Dutré P. Interactive Boolean operations on surfelbounded solids. ACM Trans Graph 2003;22(3):651–6.

[20] Biermann H, Kristjansson D, Zorrin D. Approximate Boolean operations on free-form solids. In: Siggraph'01. 2001, p. 185–194.

[21] Smith JM, Dodgson NA. A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic. Comput Aided Des 2007;39(2).

[22] Qin Xujia, Wang Weihong, Li Qu. Practical Boolean operations on point-sampled models. In: ICCSA (1). 2006, p. 393–401.

[23] Zhao H, Wang CC, Chen Y, Jin X. Parallel and efficient Boolean on polygonal solids. Vis Comput 2011;27(6–8).

[24] Pfister H, Zwicker M, van Baar J, Gross M. Surfels: surface elements as rendering primitives. In: SIGGRAPH'00. 2000, p. 335–42.

[25] Wang CCL, Leung Y-S, Chen Y. Solid modeling of polyhedral objects by layered depth-normal images on the GPU. Comput Aided Des 2010;42(6):535–44.

[26] Rappoport A, Spitz S. Interactive Boolean operations for conceptual design of 3D solids. In: SIGGRAPH'97. 1997, p. 269–278.

[27] Hoffmann CM. Robustness in geometric computations. J Comput Inf Sci Eng 2001;1:143–56.

[28] Requicha AAG, Tilove RB. Mathematical foundations of constructive solid geometry. General topology of regular closed sets. Tech. memo. 27. Production Automation Project, Univ Rochester, Rochester, NY. March 1978.

[29] O'Rourke J. Computational geometry in C. 2nd ed. Cambridge University Press; 1998.

[30] Segura RJ, Feito FR. Algorithms to test ray–triangle intersection: comparative study. In: Proceedings of the WSCG'01. 2001.

[31] Hoffmann CM. Geometric and solid modeling. An introduction. Morgan Kauffmann Publishing; 1989.

[32] Rivero ML, Feito FR. Boolean operactions on general planar polygons. Comput Graph 2000;24(6):881–96.

[33] Ogayar CJ, Segura RJ, Feito FR. Point in Solid Strategies. Comput Graph 2005; 29(4).

[34] Ogayar CJ, Feito FR, Segura RJ, Rivero ML. GPU-based evaluation of Boolean operations on triangulated solids. In: SIACG'06. 2006.

[35] Veblen O. Theory on plane curves in non-metrical analysis situs. Trans Amer Math Soc 1905;6(1):83–98. Providence (RI) American Mathematical Society.

[36] Kuratowski K, Mostowski A. Set theory. Amsterdam: North-Holland; 1976.

[37] CGAL, Computational geometry algorithms library. 2011. http://www.cgal.org.

[38] GTS, The GNU triangulated surface library. 2012. http://gts.sourceforge.net/.