Industrial Application of Exact Boolean Operations for Meshes

Martin Schifko*
ECS, Magna Powertrain, Steyr

Bert Jüttler†
Johannes Kepler University of Linz

Bernhard Kornberger ‡
Graz University of Technology

Abstract

We present an algorithm for robust Boolean operations of triangulated solids, which is suitable for real-word industrial applications involving meshes with large numbers of triangles. In order to avoid potential robustness problems, which may be caused by (almost) degenerate triangles or by intersections of nearly co-planar triangles, we use filtered exact arithmetic, based on the libraries CGAL and GNU Multi Precision Arithmetic Library. The method consists of two major steps: First we compute the exact intersection of the meshes using a sweep plane algorithm. Second we apply mesh cleaning methods which allow us to generate output which can safely be represented using floating point numbers. The performance of the method is demonstrated by several examples which are taken from applications at ECS Magna Powertrain.

CR Categories: I.3.5 [Computing Methodologies]: Computer Graphics—Computational Geometry and Object Modeling

Keywords: Boolean operation, triangulated surfaces

1 Introduction

Triangular meshes are commonly used as representation of geometrical objects in manufacturing. Especially at the later stages of the product development pipeline, where frequently components provided by different suppliers have to be combined in the final product, these meshes are often the only available description of the different components.

In order to simulate the manufacturing process, it is essential to be able to perform basic geometric modeling tasks with triangulated solids, e.g. CSG operations, cf. [Hubbard 1996]. In particular, we focus on the problem of Boolean operations of triangulated solids. For our applications, which are related to the simulation of various manufacturing processes in the automotive industry, it is essential that these operations give reliable results for all input meshes, even if (nearly) tangential intersections are present.

The discussion of Boolean operations for meshes has a long history. Already in the mid 80s, the first methods working with arbitrary precision arithmetics appeared. Typically, the method are characterized by a large number of case distinctions [Laidlaw et al. 1986; Ayala et al. 1985]. The plane-based representation of meshes, along with applications to robust modeling operations, was introduced in [Sugihara and Ira 1989].

In principle, the intersection curve of two triangulated solids can be computed using a simple triangle-to-triangle intersection test, see e.g. [Thomas and Prosolvia 1997; Tropp et al. 2006]. In order to obtain an efficient algorithm, however, we aim at minimizing the number of intersection tests. This can be achieved by using one of the various bounding volume methods, such as hierarchies of bounding volumes, octrees, etc. See [Jimenez et al. 2001; Chang et al. 2010] and the references cited therein.

*e-mail:martin.schifko@ecs.steyr.com
†e-mail:bert.juettler@jku.at
‡e-mail:bkorn@geom.at

Another intersection technique for meshes has been described by [Teschner et al. 2003]. In a recent paper dealing with the intersection of triangular meshes, [Park 2004] uses a combination of a space-partitioning method with visibility information for finding the self-intersections of a triangular mesh. The intersection of quadrilateral meshes is considered in [Lo and Wang 2003]. Generally these polygonal algorithm suffer from numerical errors.

Even in the case of NURBS surface, the robust and efficient computation of the intersection is still a challenging task. Using methods of approximate algebraic geometry, it has recently been studied in the frame of the European GAIA II project, see [Dokken 2008].

For real-world applications involving triangular meshes, we need an algorithm which produces a valid result for all input meshes. In a manufacturing environment, robustness and high reliability is required. The associated geometrical computations lead to a great variety of non-trivial problems. On the one hand, this is due to intersections of almost coplanar triangles, which have to be computed robustly. On the other hand, the intersection of two triangular meshes sometimes leads to triangles with one ore more extremely short edges, which cannot be represented correctly using floating point arithmetic.

In his classical textbook, [Hoffmann 2005] presents an algorithm for Boolean operations on boundary representation. In particular, he addresses the need for robust and error-free geometric operations. We use these ideas in our work and apply them in the context of industrial applications.

In a recent paper, [Pavic et al. 2010] present a hybrid method for robustly representing and computing the intersection of triangulated solids. The authors use a local volumetric model and generate a new mesh – which preserves the features of the intersection curve – along the intersection. The method generates an error in the models, which can be controlled by the resolution of the octree-based volumetric model. The size of the examples presented in the paper do not reach the scale needed for our applications, where meshes with millions of triangles occur frequently. The authors did not provide information about memory requirements.

The powerful library [CGAL 2009] provides a package [Hachenberger and Kettner 2005] which implements a B-rep data structure that is closed under Boolean operations. This method relies on exact arithmetic to avoid the well known problems with floating-point arithmetic. The underlying data structure - which is based on Nef polyhedra – is not memory efficient and hence not suitable for Boolean operations involving large meshes. Nevertheless, this implementation of exact operations for triangulated solids is currently considered as a standard reference.

[Bernstein and Fussel 2009] used the concepts of plane based geometry representation and BSP (binary space partitioning, [Thibault 1987; Thibault and Naylor 1987]) in order to construct exact and robust Boolean operators with low algorithmic complexity. This approach has recently been used for intersecting polygonal meshes [Campen and Kobbelt 2010]. The method described in the latter paper performs an exact intersection of two triangulated objects in BSP representation, and generates an exact output in this representation. It does not address the exact conversion of the result into a triangular mesh in detail.

One approach to overcome limitations in numerical accuracy is the

use of multiple precision libraries, and our method is based on it. Our goal is to modify the input geometry as little as possible, since it is the only available geometric information at this stage of the design process. Our method consists of two steps: First, we convert the given vertices of the meshes to points having rational coordinates and we compute the intersection of the meshes using a sweep plane algorithm. Second, we apply a mesh cleaning procedure that allows us to generate output which can safely be represented using floating point numbers.

The remainder of this paper is organized as follows. First we describe the details of the problem and give an outline of our approach. The following section describes the details of the algorithm. Section 4 is devoted to the implementation of the method using filtered exact arithmetic. Finally, in Section 5, we use several realistic examples as well as an artificial example to compare the implementation of our approach with an implementation which uses the 'join' operation of CGAL's NEF Polyhedron package [Hachenberger and Kettner 2005].

2 Problem description and outline

In this section we define the problem and provide an overview of our method. The details will be in the following section.

2.1 The problem

We consider two solid objects A and B in three-dimensional space, $A, B \subset \mathbb{R}^3$. Both are given by their boundary representations, where the boundary surfaces ∂A and ∂B are described by oriented triangular meshes $\mathscr{A}$ and $\mathscr{B}$. In our applications, the input typically consists of car parts, where the numbers of faces is in the range between 10^3 and 10^{10}.

We assume that both solids are different. Consequently, the intersection of any two triangles $\triangle \in \mathscr{A}$ and $\triangle' \in \mathscr{B}$ is either empty, a point, a line segment, or a planar polygon. Each triangle $\triangle = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) \in \mathscr{A}$ (and analogously for $\mathscr{B}$) is described by an ordered list of vertices. The orientation of the triangles is chosen such that the normal vectors $(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)$ (at any inner point of the triangle) point outwards.

Mathematically, we will consider $\mathscr{A}$ and $\mathscr{B}$ as sets of oriented triangles. In the implementation, we use the half-edge data structure representation of CGAL [CGAL 2009]. We assume that the given triangular meshes are both *clean* in the following sense.

1. Both meshes are assumed to be valid boundary surfaces of three-dimensional solid objects. Thus, the meshes do not contain holes or boundary edges.
2. Every edge of any triangle of $\mathscr{A}$ or of $\mathscr{B}$ belongs to exactly two triangles. Consequently, the meshes do not possess any hanging nodes or degenerate triangles.
3. The meshes are assumed to be robust with respect to small perturbations of the coordinates of the vertices. More precisely, if we replace the vertices $\mathbf{v}_i$ by new vertices $\mathbf{v}_i'$, such that $||\mathbf{v}_i - \mathbf{v}_i'|| \leq \varepsilon$, where ε is the accuracy of the chosen geometric representation, then we obtain again a valid boundary representation of a three-dimensional solid.

The last property is needed when using floating point numbers for representing the coordinates of the vertices, since these coordinates are then only known with limited accuracy. The accuracy depends on the size of the bounding box, which includes both the origin of the chosen coordinate system and the two objects, and on the number of significant digits.

We discuss a method for Boolean operations of union (which is often called merging in our applications), intersection and difference of the two solid objects. For instance for the union operation, the boundary representation $\mathscr{M}$ of the union $M = A \cup B$ of the two solids satisfies

$$\partial M = (\partial A \cup \partial B) \setminus M^\circ, \tag{1}$$

where M° denotes the interior of the set $M \subset \mathbb{R}^3$.

The other two Boolean operations intersection and difference can be obtained from the same equation by first inverting the orientation of the faces of the solids (hence replacing A or B with $\mathbb{R}^3 \setminus A$ or $\mathbb{R}^3 \setminus B$, respectively), and secondly inverting the result of the operation. Since the Boolean operations of union, intersection and difference essentially require the same computation, except for some pre- and postprocessing steps, we will only consider the union operation in the remainder of this paper. Moreover, since it is common in the automotive industry to denote the union operation as *merge* operation, we will use this notation from now on.

2.2 The intersection curve of triangular meshes

In order to find the boundary representation of the Boolean operations, we need to analyze the intersection curve $C = \partial A \cap \partial B$ of the two boundary surfaces. The intersection curve is a piecewise linear object, which may be a collection of curves. For simplicity, we will call it "the" intersection curve, even if it may consist of several components.

In the generic situation, this curve is simply a collection of closed curves. In special cases, however, it might also contain open curves and two-dimensional components. Open curves occur if the solids just touch each other along these curves . Two-dimensional components would be present if the solids touch each other in one or several faces.

We represent the intersection curve C by a set $\mathscr{C}$ of line segments which form at least one polyline. More precisely, we use the following conventions.

1. Each closed curve is represented as a closed polyline, which may contain planar loops.
2. Each open curve is represented as a degenerate polyline, where each edge is repeated twice with opposite orientations.
3. Two-dimensional components are represented by their boundary curves, and points are represented as degenerate curves.

Clearly, it may happen that open and closed curves, as well as two-dimensional components, meet in common vertices.

2.3 Outline of the algorithm

In order to generate a boundary representation of the union (merge) operation, we propose the following algorithm.

1. Compute the line segments which form the intersection curve(s) $\mathscr{C}$.
2. Refine the triangular meshes $\mathscr{A}$ and $\mathscr{B}$ such that all edges in $\mathscr{C}$ are contained in the adapted meshes $\mathscr{A}^*$ and $\mathscr{B}^*$. Collect the line segments which form the intersection curve(s) to polylines which match the orientation of the facets of the given meshes.

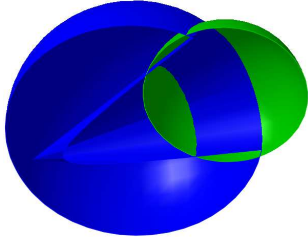

Figure 1: Example 1: The two intersecting surfaces $\mathscr{A}$ (blue) and $\mathscr{B}$ (green). A clipping plane has been used in order to show the interior parts of both objects.

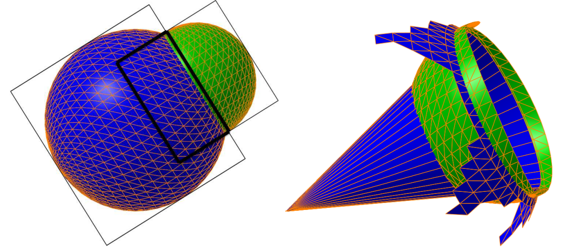

Figure 2: Example 2: The intersection of the two bounding boxes $H(\mathscr{A}) \cap H(\mathscr{B})$ (left) and the triangles contained in the submeshes $\mathscr{A}^+$ and $\mathscr{B}^+$ which are potentially relevant for the intersection curve $\mathscr{C}$ (right).

3. Find the output mesh $\mathscr{M}$ by deleting the sets $\mathscr{A}^* \cap B^\circ$ and $\mathscr{B}^* \cap A^\circ$ of triangles which are contained in the interior A° and B° of the other object A and B, respectively, from the union $\mathscr{A}^* \cup \mathscr{B}^*$.
4. (Optional:) If the mesh is to be exported with limited accuracy (floating point coordinates), then it is cleaned up before exporting. The cleaned mesh is denoted with $\mathscr{M}^c$.

The computations in steps 1-3 are done with filtered exact arithmetic. Consequently, these computations are numerically robust and produce a valid mesh, as no numerical errors are present. The optional cleaning step - which is needed in order to produce floating point numbers - gives an approximation $\mathscr{M}^c$ of the exact boundary representation $\mathscr{M}$ of the merged object $M = A \cup B$.

Example 1 We use a simple example to describe our algorithm. The two solid objects are shown in Figure 1. The first object (shown in blue) has been constructed as the set-theoretic difference of a triangulated ball and a triangulated cone. The second object (shown in green) is simply a ball.

3 Details

Step 1: Computation of the intersection curve $\mathscr{C}$

Let us assume that the meshes $\mathscr{A}$ and $\mathscr{B}$ consist of a and b triangles. Obviously, it does not make sense to perform intersection tests for all ab pairs of triangles. We use axis–aligned bounding boxes, a sweep plane algorithm and dynamical R-trees [Guttman 1984; Manolopoulos et al. 1989] in order to obtain a more efficient algorithm. A sweep plane algorithm provides a sequential processing and therefore a low memory usage. The R-tree increases the efficiency of intersection tests. This is a reasonable compromise between computational efficiency and ease of implementation. Clearly more sophisticated techniques exist and could be used instead, see e.g. [Teschner et al. 2003].

Axis Aligned Bounding Boxes and Preprocessing. We use two- and three-dimensional axis-aligned bounding boxes. First, we denote with $H(X)$ the three-dimensional axis-aligned bounding box of a set $X \subset \mathbb{R}^3$, where X is either a single triangle or a set of triangles. Second, we denote with $R(X)$ the projection of this bounding box into the xy-plane, which is simultaneously the two-dimensional bounding box of the projection of X.

We use the bounding boxes of $\mathscr{A}$ and $\mathscr{B}$ for a simple preprocessing step. Obviously, it suffices to consider only the triangles $\triangle$ in $\mathscr{A}$ and $\mathscr{B}$ having a bounding box $H(\triangle)$ which possesses a non-empty intersection with the intersection of the two bounding boxes,

$$H(\triangle) \cap [H(\mathscr{A}) \cap H(\mathscr{B})] \neq \emptyset. \quad (2)$$

We denote these sets of triangles with $\mathscr{A}^+$ and $\mathscr{B}^+$, respectively.

Example 2 We continue the first example, by restricting both triangular meshes to triangles whose bounding boxes interfere with the intersection of the two global bounding boxes. The resulting subsets of the triangular meshes $\mathscr{A}$ and $\mathscr{B}$ are shown in Figure 2.

Sweep Plane Algorithm and Dynamic R-tree. In order to further restrict the pairs of triangles in $\mathscr{A}$ and $\mathscr{B}$ which we need to consider for mutual intersection, we use a sweep plane algorithm, which takes the two sets $\mathscr{A}^+$ and $\mathscr{B}^+$ as input. We consider a sweep plane, which is parallel to the xy-plane, and moves from the smallest to the largest z–coordinate of the vertices in $\mathscr{A}^+ \cup \mathscr{B}^+$. The algorithm generates a list containing all pairs of triangles $(\triangle, \triangle') \in \mathscr{A}^+ \times \mathscr{B}^+$ with a non-empty intersection.

The *events* of the sweep plane algorithm are the xy-parallel faces of the bounding boxes $H(\triangle)$, $\triangle \in \mathscr{A}^+ \cup \mathscr{B}^+$, sorted according to the value of their z-coordinates. If two events have the same z-value, then the lower faces (which we call $z_{\min}$-values of bounding boxes) are considered first, followed by upper faces (called $z_{\max}$-values). The events are stored in an event queue.

The *status* consists of two *sets of rectangles* $K_{\mathscr{A}}$ and $K_{\mathscr{B}}$, both containing the two-dimensional axis-aligned bounding boxes $R(\triangle)$ of all triangles $\triangle \in A^+$ and $\triangle \in B^+$, which possess a non-empty intersection with the sweep plane. We use a dynamical binary R-tree for representing the status information.

The leafs of the R-tree store sets of rectangles from both sets $K_{\mathscr{A}}$ and $K_{\mathscr{B}}$, where each rectangle has a pointer to the associated triangle in $\mathscr{A}^+$ or $\mathscr{B}^+$. In addition, each leaf and each node of the tree stores the minimum bounding rectangle of all rectangles associated with it or with the leafs of the corresponding subtree. The bounding rectangles at the two children of a node are not always disjoint.

During the execution of the algorithm, the rectangles from $K_{\mathscr{A}}$ and $K_{\mathscr{B}}$ are dynamically inserted into and deleted from the R-tree. Each newly inserted rectangle is inserted into the leaf where it causes the smallest possible enlargement of the bounding rectangle.

If the number of boxes, which are associated with a leaf, exceeds a certain threshold (which is 30 in our implementation), then the rectangle is subdivided into two smaller rectangles. Similarly, leafs are dynamically re-combined, if the number of boxes associated with them falls below a certain threshold.

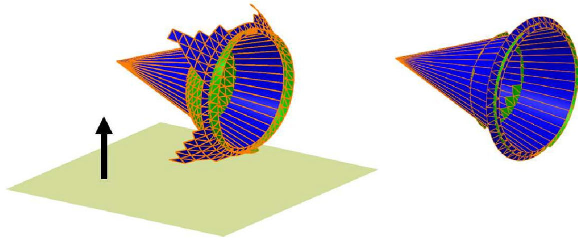

Figure 3: Example 3: The sweep plane algorithm (left) detects all pairs of triangles (right) which contribute to the intersection of both meshes.

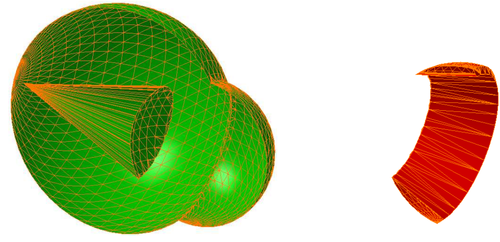

Figure 4: The boundary of the merged solids (left) and the deleted parts of the boundary surfaces (right). Once again, a clipping plane has been used in order to show the interior parts of both objects.

The two types of events are faces of three-dimensional bounding boxes with maximal and minimal z-values, which are handled as follows.

- *Event minimal z-value:* We assume that the event is the lower face of a bounding box $H(\triangle)$ with $\triangle \in \mathscr{A}^+$. The case of $\triangle \in \mathscr{B}^+$ is dealt with in an analogous way. We insert the rectangle $R(\triangle)$ into the set $K_{\mathscr{A}}$. Simultaneously we check if it possesses a non-empty intersection with any of the rectangles $R(\triangle')$ in $K_{\mathscr{B}}$. If this is the case, then we report the intersection $\triangle \cap \triangle'$, provided that it is non-empty.
- *Event maximal z-value:* The bounding rectangle $R(\triangle)$ is deleted from the set $K_{\mathscr{A}}$ or $K_{\mathscr{B}}$.

Example 3 We apply the sweep plane algorithm to the two triangular meshes of Example 1. It finds all pairs of triangles which contribute to the intersection of the two meshes. Note that the first object (shown in blue) is given by a rather coarse triangulation in the conical part, hence the algorithm detects virtually all triangles.

Intersection of the triangles. First we convert all coordinates of all triangles which have an intersection, to rational coordinates. The intersection of two triangles may consist of a point, a line segment or a planar polygon with at most 6 vertices. We compute it using filtered exact arithmetic. The coordinates of the vertices of the intersection curve have again rational coordinates.

Note that the expression swell (i.e. the number digits needed for representing the rational coordinates) is bounded, since the triangles which have to be intersected are already contained in the original meshes $\mathscr{A}$ and $\mathscr{B}$; no further computations with the obtained coordinates of intersection points are needed in this step. See Section 4 for more information on filtered exact arithmetic.

Planar triangle intersections do not need to be computed. The neighboring triangles generate the necessary intersection points of two-dimensional parts of the intersection. We modified our algorithm so that it computes only the boundary curves of the overlapping parts of both meshes if such parts are present. Summing up, the sweep plane algorithm generates pairs of triangles, along with the line segments which form the intersections.

Step 2: Mesh adaptation and assembly of $\mathscr{C}$

Mesh adaptation. Each triangle of the meshes $\mathscr{A}$ and $\mathscr{B}$, which contains a vertex of the intersection curve(s), is further split into smaller triangles, such as the edges of the intersection curve(s) are also edges of the triangulation. In order to obtain triangles which are as regular as possible, we use a constrained Delaunay triangulation within each triangle. However every triangulation algorithm can also be used instead.

The refined meshes, which are now adapted to the intersection curve(s) $\mathscr{C}$, are denoted with $\mathscr{A}^*$ and B^*.

Assembling the intersection curve(s). Finally, for each triangular mesh $\mathscr{A}$ and $\mathscr{B}$, we assemble the computed line segments to obtain the intersection curve(s) $\mathscr{C}$. The orientation is chosen such that it matches the orientation of the facets of the meshes for all facets which contribute to the boundary of the union $M = A \cup B$.

The correct orientation can be decided locally, based on the normal vectors of the intersecting triangles. Note that each component of the intersection curve is obtained twice, once as a curve lying on $\mathscr{A}$, and once as a curve lying on $\mathscr{B}$. If we have planar triangle intersections planar loops on the intersection curve arise. The orientation of each intersection curve determines which part of the planar loop belongs to the curve on $\mathscr{A}$, and $\mathscr{B}$ respectively. By definition, the left-hand side (with respect to the orientation of the triangles) of the intersection curve on the considered solid is part of the merged surface. By traversing the oriented intersection curve and arriving the planar loops, we move around on the left side of the loop.

The areas within the planar loops and the triangles within two-dimensional parts of the intersection need a special treatment. If all their normal vectors point into the same direction, they are part of the intersection. Otherwise they are discarded.

Step 3: Mesh merging

Finally, we remove from both meshes $\mathscr{A}^*$ and B^* the triangles which lie on the right-hand side of the closed intersection curves. In addition, we connect the two meshes along the intersection curve(s), obtaining the single mesh $\mathscr{M}$ which bounds the union of the two objects.

Example 4 After applying Steps 2 and 3 of the algorithm, we obtain the mesh which is shown in Figure 4, left. In addition, the deleted parts of both meshes are also shown.

Step 4: Mesh cleaning

The mesh cleaning problem is also known as "geometric rounding" or "3D snap rounding" problem. The probably best (theoretical) solution which is available so far [Fortune 1998], is still considered to be rather impractical and inefficient by the author.

At this stage, the result of our mesh merging algorithm is still given in exact arithmetic. In order to convert the coordinates back into floating point numbers, we need to perform a mesh cleaning step. In this cleaning step we eliminate small triangles (almost degenerate

triangles) which may be present in $\mathscr{M}$. Otherwise, the conversion to floating point numbers may destroy the correctness of the result, creating degenerate triangles and possibly even local and global self intersections.

We denote with

$$N_\varepsilon(P) = \{\mathbf{x} \in \mathbb{R}^3 : \exists \mathbf{y} \in P : \|\mathbf{x} - \mathbf{y}\| \leq \varepsilon\} \tag{3}$$

the ε-neighborhood of a point set P. The mesh cleaning is based on the following simple observation.

Consider a regular triangular mesh $\mathscr{M}$ which does not possess self-intersections. We assume that $\mathscr{M}$ satisfies the following three conditions.

(i) The ε–neighborhoods $N_\varepsilon(v)$ and $N_\varepsilon(w)$ of any two distinct vertices v, w are disjoint.

(ii) The ε–neighborhoods $N_\varepsilon(v)$ and $N_\varepsilon(e)$ of any vertex v and any edge e not containing v are disjoint.

(iii) The ε–neighborhood $N_\varepsilon(\triangle)$ of any triangle $\triangle$ of $\mathscr{M}$ does not intersect the ε–neighborhood $N_\varepsilon(e)$ of any edge e of $\mathscr{M}$, provided that e and $\triangle$ are disjoint.

Any mesh $\mathscr{M}'$ which is obtained by replacing each vertex v of M by a new vertex v' satisfying $\|v - v'\| \leq \varepsilon$, consists only of regular triangles and does not possess any self-intersections.

The first condition guarantees that the modified mesh does not possess any triangle with edges that collapse into a point and that any two triangles are different. The second condition ensures that the modified mesh does not possess any degenerate triangles with collinear edges. Thus, all triangles are regular. The third condition implies that the modified mesh does not possess self-intersections.

For most meshes, it is possible to infer the first two conditions from the third one. However, this is not always true. For instance, if M represents a tetrahedron, then the third conditions is always satisfied and therefore not useful for certifying the robustness.

We are currently implementing a collection of mesh cleaning operations in order to guarantee that the three conditions are satisfied. In practice, it is sufficient to check the newly created regions of the mesh $\mathscr{M}$, which were not yet contained in either $\mathscr{A}$ or $\mathscr{B}$, since the original meshes are assumed to be clean, as described in Section 2.1. In fact, even if a vertex $\mathbf{v}$ of $\mathscr{A}$ might be closer than ε to a face $N_\varepsilon(\triangle)$ of $\mathscr{B}$ in a region where $\mathscr{A}$ and $\mathscr{B}$ do not intersect. The merged result mesh violates the conditions of $\mathbf{v}$ and $\triangle$. The vertex $\mathbf{v}$ however will never intersect $\triangle$, since the vertices representation of $\mathbf{v}$ and of $\triangle$ have never been modified by any computation through the algorithm. Since we treated the input meshes as exact, we will not have a rounding error in $\mathbf{v}$ and $\triangle$. Thus, in this situation, we do not get any problems, even though the conditions (i)-(iii) are violated.

We choose the constant ε as an upper bound of the rounding error which is created by converting the exact rational numbers into an inexact number type. We identify all edges and triangles in $\mathscr{M}$ which do not satisfy the first assumption and flag them as *small triangles* and *short edges*. The following well-known mesh cleaning operations are currently performed automatically:

1. *Collapsing small triangles:* If all edges have a length less than 2ε, then the triangle is collapsed into a point, located at its barycenter, and the three triangles which share edges with it are collapsed into edges. However, it may happen that the triangles which are adjacent after this step have more than one common edge. If this is the case, then this small triangle is not removed by this approach.

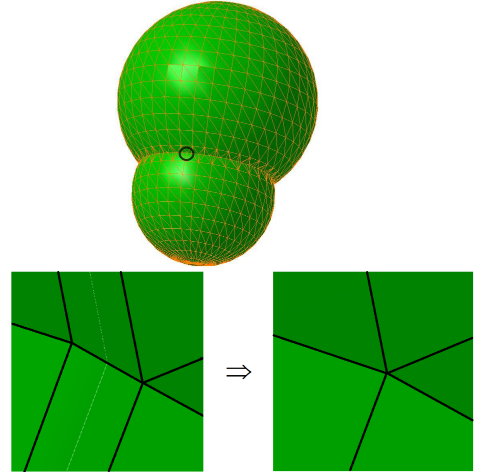

Figure 5: The merged mesh (left) contains a short edge, which is collapsed into a point. The adjacent triangles shrink into edges. The right two figures shows a close-up view.

2. *Collapsing short edges:* If the length of an edge is below 2ε, then it is shrunk to a point and the two neighboring triangles are collapsed to edges. Again, it may happen that the triangles which are adjacent after this step have more than one common edge. If this is the case, then this small edge is not removed.
3. *Enlarging short edges and small triangles:* If the previous two operations fail, then we try to enlarge the remaining small triangles, either by enlarging edges (in the case of small edges) on both ends, or by moving the vertices away from the center of gravity. In order to avoid infinite loops, the number of modifications per vertex is limited.
4. *Flip edge:* The apex angle is almost 180°. If the neighboring triangle of the longest edge is not a small triangle, we flip the longest edge. Now the flipped edge is tested by the method "collapsing short edges".

In our experiments, which we performed with real data sets representing industrial objects (see Section 5), these four simple mesh cleaning operations were sufficient to produce meshes which could safely be exported using floating point numbers. We are currently working on the extension to more complex mesh-cleaning operations, which are based on the full set of constraints (i)-(iii).

Finally the result mesh is tested for self-intersections. This is done by trying to merge the result mesh with itself. In all our examples, no intersections of non-neighboring triangles in the mesh were found, which guarantees that the mesh is free of self-intersections.

Example 5 We apply the mesh cleaning example to the example shown in Figure 4. The algorithm detects a number of short edges and small triangles which automatically removed, see Fig. 5.

4 Using filtered exact arithmetic

We aim at achieving a robust implementation that computes topologically correct results. Floating point arithmetic with limited precision data types such as the built-in data type *double* in C++ is afflicted with numerical errors. While tiny geometric inaccuracies introduced by these errors might be acceptable, topological changes are not. Moreover, implementations of geometric algorithms are

prone to fail if they rely on predicates, that are evaluated using inexact arithmetic, because a wrong result can lead a program into a wrong state.

In fact, our first attempt for computing Boolean operations of triangulated solids used double precision floating point arithmetic. Very quickly we arrived at numerical problems which forced us to abandon this approach.

To achieve both, robustness of our implementation and correctness of its output, we use a so called *kernel* from CGAL [CGAL 2009]. In CGAL, a kernel consists of a collection of basic geometric objects like points, lines, triangles etc., construction operations like the intersection of two objects, and predicates like, e.g., orientation tests. CGAL follows the generic programming paradigm, and thus the used number type can be chosen according to the needs of a specific problem.

We use two different CGAL kernels for our implementation, and their behavior is determined by a nested chain of parametrizations, see Listing 1.

Listing 1: Parametrization of CGAL Kernels

```
typedef Filtered_kernel< Simple_Cartesian<double> >
        Exact_predicates_inexact_constructions_kernel;
typedef Lazy_kernel<Simple_Cartesian<Gmpq> >
        Exact_predicates_exact_constructions_kernel;
```

Concerning the *Exact_predicates_inexact_constructions_kernel*, the part which consists of the template class *Simple_Cartesian*, parametrized by the C++ number type *double*, denotes already a simple CGAL kernel. But this kernel uses cheap (and inexact) double precision floating point arithmetic for both, evaluation of predicates and representation of objects by Cartesian coordinates.

In order to achieve exact evaluation of predicates, one could parametrize this kernel with a slow multi-precision number type instead of *double*. But there is a more efficient way to achieve robustness, which is called arithmetic filtering: The inexact kernel *Simple_Cartesian<double>* is plugged into a *Filtered_kernel* [Fabri and S.Pion 2006], which uses interval arithmetic [Brönnimann et al. 2001] to detect if a predicate, evaluated with the inexact number type, is possibly incorrect.

In cases where the arithmetic filter fails, the predicate is re-evaluated with arbitrary-precision rational arithmetic. Thus, the filtered kernel provides exact predicates although it uses computationally expensive multi-precision arithmetic (hopefully) only rarely. The major drawback of the *Exact_predicates_inexact_constructions_kernel* is that its constructions are computed with inexact arithmetic, such that even exact predicates will possibly fail if they involve constructed objects, such as in the example shown in Listing 2.

Listing 2: Provoking an error in spite of exact predicates

```
typedef CGAL::
   Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef K::Line_2 Line;

Line line1(Point(0,0),Point(1,2));
Line line2(Point(0,1),Point(1,0));
Point p;
assign(p,intersection(line1,line2));
assert(line1.has_on(p) && line2.has_on(p));
```

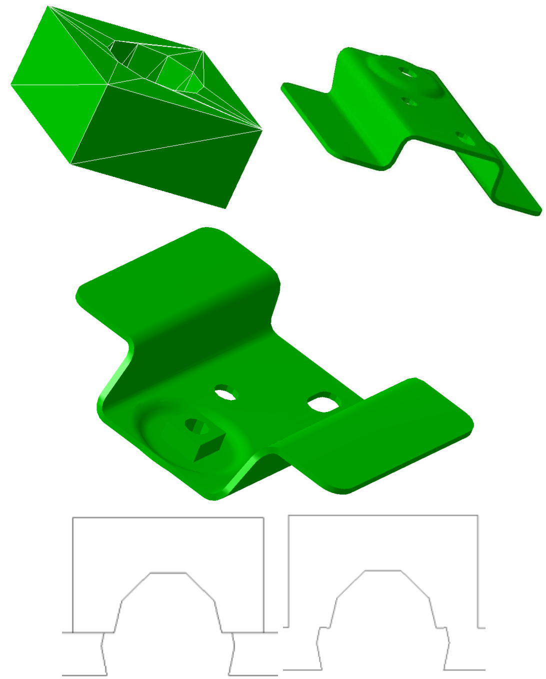

Figure 6: Example 6: The figures in the top row show a nut (left) with a metal sheet (center), merged in a model with 2,304 triangles (Figure below). The last row Figures show a sliced view with a vertical slicing plane of a detail before (left) and after (right) merging the solids.

Therefore, we use the *Exact_predicates_exact_constructions_kernel* in cases where exact constructions are necessary in order to make our implementation robust. The parametrization is also shown in Listing 1. For this kernel, the template class *Simple_Cartesian* is parametrized differently, namely by the class *Gmpq*, which provides an arbitrary precision rational number type based on the GNU Multiple Precision Arithmetic Library [GMP].

Although *Simple_Cartesian<Gmpq>* denotes already an exact kernel, it is, for performance considerations, not used directly. Instead, this kernel is plugged into a *Lazy_kernel* which tentatively uses interval arithmetic for constructions. If later a filter fails, i.e. a predicate, applied to an object, cannot be evaluated safely, then exact rational coordinates are used. The additional cost of this kernel is an increased memory usage as the history of all construction steps for the inexactly evaluated objects has to be stored.

5 Examples

We present several examples which demonstrate the performance of our method. The models are taken from the database at ECS and represent real industrial objects. None of the following examples can be solved with our algorithm by exchanging *Exact_predicates_exact_constructions_kernel* by *Exact_predicates_inexact_constructions_kernel*.

Example 6 This example, which is shown in Figure 6, consists of a box with a hole, representing a nut, which is to be merged with a deep-drawn metal sheet. The two given objects are represented by 112 and 1,976 triangles respectively, and the merged solid is described by a mesh with 2,304 triangles. The computation took 0.25

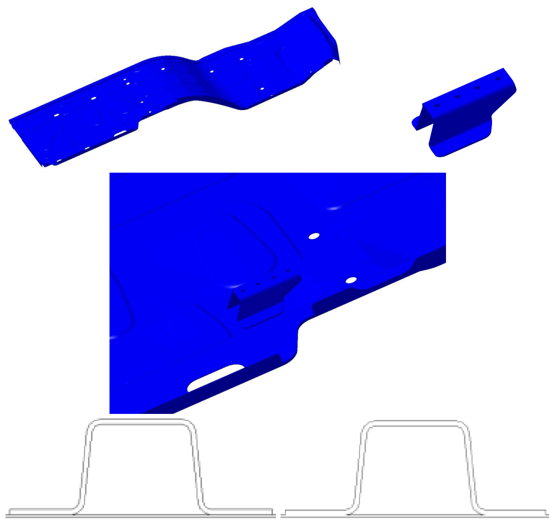

Figure 7: Example 7: Merging of the underbody and panel solids (top row) and sliced views of a detail before (bottom left) and after (bottom right) merging the two solids. In the sliced views, two-dimensional components of the intersections are visible.

seconds CPU time on a Dell Precision™ 490 Workstation with 64-bit quad-core Intel Xeon processors and 8 GB RAM.

Example 7 This example (see Figure 7) consists of a underbody of a car and a panel, which is to be attached to it. The two given objects are represented by 45,234 and 540 triangles respectively, and the merged solid is described by a mesh with 45,774 triangles. The computation took 0.1 seconds CPU time.

Example 8 The next example (Figures 8 and 9) consists of a door of a car and a hinge. The two given objects are represented by 121,829 and 89,166 triangles respectively, and the merged solid is described by a mesh with 217,522 triangles. The computation took 12 seconds CPU time. In order to obtain this result, the thickness of the hinge has to exceed the thickness of the corresponding part of the door (see sliced view). This is achieved by applying a offsetting operation with a small distance to the relevant part of the hinge.

Example 9 Finally we compare the Nef polyhedron-based method in CGAL [CGAL 2009; Hachenberger and Kettner 2005] with our algorithm. We consider again the two solids of Example 1 and approximate them by triangular meshes with a different number of facets. Clearly, the computation time grows with the number of triangles. The running time of our approach depends on the number of triangles of the input object as well as the complexity of the intersection. In Figure 10, the red graph refers shows the computation times needed when using Nef polyhedra, while the blue graph corresponds to our algorithm. The chart shows that the running time of NEF is linear and our runtime - which is much lower - behaves almost linearly.

However, the memory requirements are dramatically different, see Figure 11. Even though we used a computer with 8 GB main memory, we did not have enough memory to compute the next example, where the resulting mesh would have had about 400K triangles, using Nef polyhedra. Using our algorithm it is possible to merge meshes consisting of millions of triangles. Note that the results generated using Nef polyhedra have a different number of triangles, since a remeshing is performed.

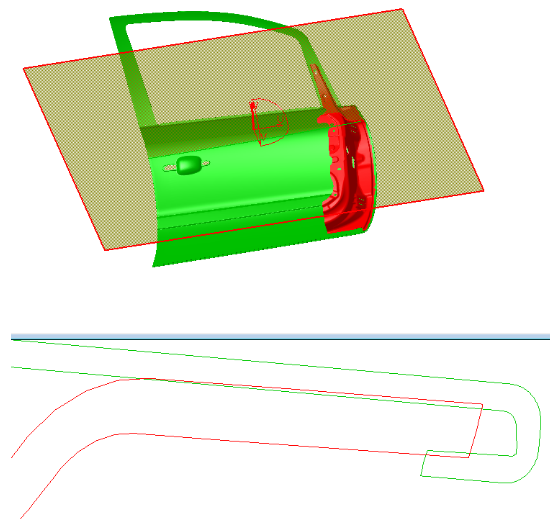

Figure 8: Example 8: Merging of the door (green) and hinge (red) with an enlarged section of sliced view (below). Meshes courtesy of Adam Opel GmbH

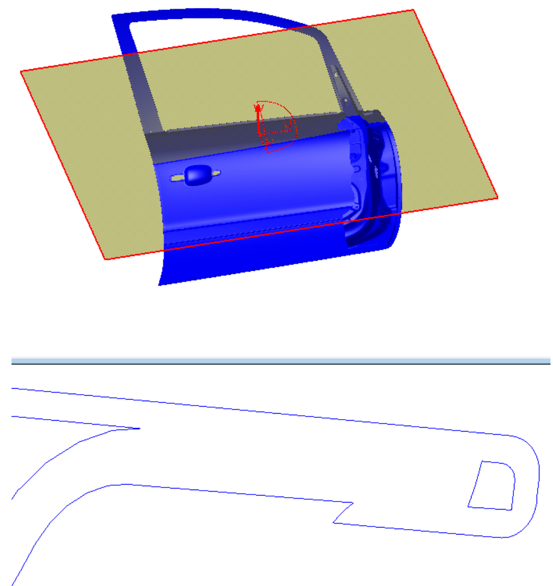

Figure 9: Example 8: Merged mesh (blue) with an enlarged section of sliced view (below).

6 Conclusions & Future Work

We used filtered exact geometric computation in order to formulate and to implement an algorithm for the robust merging of triangulated solids. With the help of suitable libraries from CGAL we were able to implement this algorithm such that data sets arising in real-world applications can be handled within reasonable computation times. The test for self-intersections - which is needed for certifying the result - is still relatively slow and should be improved. However, in practice this is only needed for testing the correctness of the algorithm.

Future work will be devoted to the enhancement of the mesh clean-

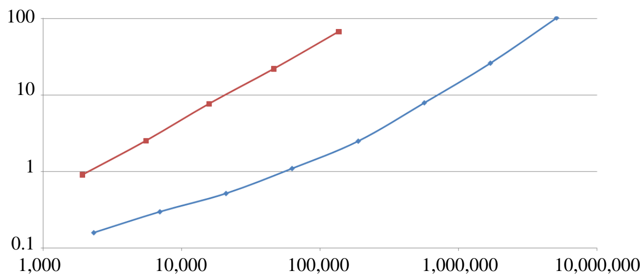

Figure 10: Computation time (vertical axis, in seconds) for merging triangle meshes of various sizes (horizontal axis) representing the two solids in Example 1. The red and the blue graph show the results for NEF polyhedra and using our algorithm.

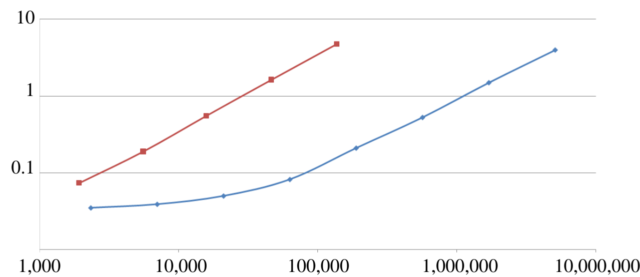

Figure 11: Maximal memory consumption (vertical axis, in GByte) for merging triangle meshes of various sizes (horizontal axis) representing the two solids in Example 1. The red and the blue graph show the results for NEF polyhedra and using our algorithm.

ing procedure, in particular to the remaining conditions which guarantee the absence of self-intersections.

Acknowledgment The authors gratefully acknowledge the support of the Austrian Science Fund through the National Research Network "Industrial Geometry", subprojects S9202 and S9205.

References

AYALA, D., BRUNET, P., R.JUAN, AND I.NAVAZZO. 1985. Object representation by means of nonminimal division quadtrees and octrees. *ACM Trans Graph. 4*, 41–59.

BERNSTEIN, G., AND FUSSEL, D. 2009. Fast, exact, linear Booleans. *Comput. Graph. Forum 28 5*, 1269–1278.

BRÖNNIMANN, H., BURNIKEL, C., AND PION, S. 2001. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Appl. Math. 109*, 1-2, 25–47.

CAMPEN, M., AND KOBBELT, L. 2010. Exact and robust (Self-) intersections for polygonal meshes. In *Proc. Eurographics*, Eurographics Assoc. to appear.

CGAL, 2009. Computational Geometry Algorithms Library. `http://www.cgal.org/`, last accessed: Oct 27, 2009.

CHANG, J.-W., WANG, W., AND KIM, M.-S. 2010. Efficient collision detection using a dual OBB-sphere bounding volume hierarchy. *Comp. -Aided Design 42*, 50–57.

DOKKEN, T. 2008. The GAIA project on intersection and implicitization. In *Geometric modeling and algebraic geometry*, B. Jüttler and R. Piene, Eds. Springer, Berlin, 5–29.

FABRI, A., AND S.PION. 2006. A generic lazy evaluation scheme for exact geometric computations. In *Proc. 2nd Library-Centric Software Design*, 75–84.

FORTUNE, S. 1998. Vertex-rounding a three-dimensional polyhedral subdivision. In *Proc. ACM Symp. Comp. Geom.*, 116–125.

GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`, last accessed: Nov 12, 2009.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data*, 47–57.

HACHENBERGER, P., AND KETTNER, L. 2005. Boolean operations on 3d selective Nef complexes: Optimized implementation and experiments. In *ACM Symposium on Solid and Physical Modeling (SPM 2005)*, 163–174.

HOFFMANN, C. M. 2005. *Geometric and Solid Modeling: an Introduction*. Springer.

HUBBARD, P. 1996. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15*, 179–210.

JIMENEZ, P., THOMAS, F., AND TORRAS, C. 2001. 3D collision detection: a survey. *Comput. Graphics 25*, 269–285.

LAIDLAW, D. H., TRUMBORE, W., AND HUGHES, J. 1986. Constructive solid geometry for polyhedral objects. *SIGGRAPH Computational Graph. 20*, 161–170.

LO, S., AND WANG, W. 2003. An algorithm for the intersection of quadrilateral surfaces by tracing of neighbours. *Comput. Methods Appl. Mech. Eng. 192*, 2319–2338.

MANOLOPOULOS, Y., NANOPOULOS, A., PAPADOPOULOS, A., AND THEODORIDIS, Y. 1989. *R-Trees: Theory and Applications*. Morgan Kaufmann.

PARK, S. C. 2004. Triangular mesh intersection. *The Visual Computer 20*, 448–456.

PAVIC, D., CAMPEN, M., AND KOBBELT, L. 2010. Hybrid Booleans. *Computer Graphics Forum*. to appear.

SUGIHARA, K., AND IRA, M. 1989. A solid modelling system free from topological inconsisteny. *J. Inf. Process. 12*, 380–393.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of VMV'03*, 47–54.

THIBAULT, W. C., AND NAYLOR, B. 1987. Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph 21*, 153–162.

THIBAULT, W. C. 1987. *Application of binary space partitioning trees to geometric modeling and ray-tracing*. PhD thesis, Georgia Institute of Technology.

THOMAS, M., AND PROSOLVIA, C. 1997. A fast triangle-triangle intersection test. *J. Graphics Tools 22*, 25–30.

TROPP, O., TAL, A., AND SHIMSHONI, I. 2006. A fast triangle to triangle intersection test for collision detection. *Computer Animation and Virtual Worlds 17*, 527–535.