

# QuickCSG: Fast Arbitrary Boolean Combinations of N Solids

Matthijs Douze, Jean-Sébastien Franco, Bruno Raffin

August 29, 2018

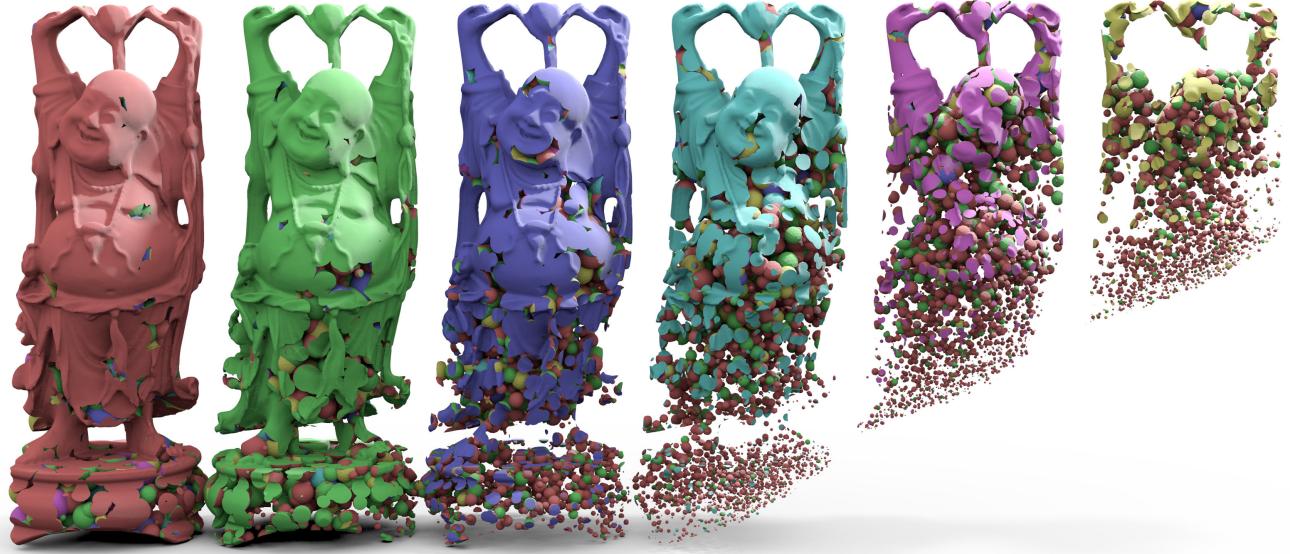


Figure 1: Intersection of 6 Buddhas with the union of 100,000 spheres (total 24 million triangles). Computed in 8 seconds on a desktop machine.

## Abstract

QuickCSG computes the result for general N-polyhedron boolean expressions without an intermediate tree of solids. We propose a vertex-centric view of the problem, which simplifies the identification of final geometric contributions, and facilitates its spatial decomposition. The problem is then cast in a single KD-tree exploration, geared toward the result by early pruning of any region of space not contributing to the final surface. We assume strong regularity properties on the input meshes and that they are in general position. This simplifying assumption, in combination with our vertex-centric approach, improves the speed of the approach. Complemented with a task-stealing parallelization, the algorithm achieves breakthrough performance, one to two orders of magnitude speedups with respect to state-of-the-art CPU algorithms, on boolean operations over two to dozens of polyhedra. The algorithm also outperforms GPU implementations with approximate discretizations, while producing an output without redundant facets. Despite the restrictive assumptions on the input, we show the usefulness of QuickCSG for applications with large CSG problems and strong temporal constraints, e.g. modeling for 3D printers, reconstruction from visual hulls and collision detection.

## 1 Introduction

Solid modeling using boolean operations is an emblematic problem in computer graphics and computational geometry, almost as old as these research topics themselves. It has found its way in every solid modeler in the industry, whether applied to model design for aviation, transportation, manufacturing, architecture, or entertainment. It is also an ubiquitous building block and subject of interest for many fields of research, including computer graphics, computer vision, robotics, virtual reality, and generally any topic where geometric models of subjects of interest are to be manipulated, constructed, truncated or combined.

Since the first introduction of **boundary representations (B-Rep)** Baumgart (1974), the problem has received considerable attention and been the subject of extensive work over more than 40 years. It is all the more striking that, despite the many existing algorithms and variants in this huge corpus, the vast majority of algorithms rely on a common principle and canvas found in the earliest formalizations of the problem Requicha and Voelcker (1985); Laidlaw et al. (1986), summarized hereafter. First and foremost, boolean B-Rep merging algorithms are written for the case of two solids. Second, the computation is divided in three stages: an initial *subdivision* stage, where the boundaries of both objects

are split in two component groups along their intersection with the other object’s boundary. A *classification* stage follows, where each group is classified as belonging inside or outside the other object. In the final *reconstruction* stage, the relevant primitives are gathered and connected to build the final model in accordance to the boolean expression. Note that the subdivision and classification require to intersect and situate all primitives of an object’s boundary with respect to the primitives of the other object’s boundary, which if done naively leads to impractical quadratic-time algorithms. Thus, a third aspect of most algorithms is the use of spatial decomposition structures, often hierarchical, to enable sublinear  $\mathcal{O}(\log m)$  access to each of the  $m$  object primitives. The construction of this data structure becomes the bottleneck of the algorithm, giving it its typically quasilinear time complexity  $\mathcal{O}(m \log m)$  in the number of input primitives Naylor et al. (1990); Hachenberger et al. (2007). An inherent drawback of this dominant approach is that all input solid’s primitives are fully decomposed, but typically only a fraction of those primitives contribute to the output result; the time spent computing hierarchical decompositions for non-contributing primitives is thus useless and can be eliminated, as we will show.

The more general case of  $n$  input solids is classically addressed by combining pairwise operations in arbitrary boolean expressions. The approach has been formalized as Constructive Solid Geometry (CSG) Requicha (1980); Mäntylä (1987), where these expressions map to CSG-trees of boolean operations. Evaluating the result for B-rep solids then relies on combining two boundaries at each node of the tree using an existing two-solid boolean algorithm. This approach however has some significant drawbacks, as they involve computing and storing a set of intermediate results, which are then recombined until the tree expression is fully resolved. The successive recombinations can be error prone, in fact leading to inherently degenerate situations for some expressions. Figure 2 shows for example all possible boolean function outcomes for the case of three solids and illustrates such cases. Another inherent limitation is that certain expressions lead to combinatorial size trees which are impractical to compute. Consider for example identifying the solid whose volume is the intersection of at least  $k$  solids, an operation later referred as *min-k*: this operation involves computing the union of all possible intersections over  $k$  or more solids, yielding a combinatorial CSG tree size.

In this paper, we challenge these dominant views of boolean modeling to eliminate these drawbacks. First, our algorithm directly computes the result of arbitrary  $n$ -ary boolean expressions, avoiding the overhead and intermediate results of binary CSG-tree approaches. This is achieved by directly identifying vertices of the final polyhedron among all relevant vertex candidates, with an efficient vertex classification test using bitvector evaluations of the  $n$ -ary CSG function.

Second, our algorithm performs the classification and subdivision stages simultaneously: as opposed to existing approaches, our hierarchical decomposition and ex-

ploration is performed on the combined set of all input solids primitives and guided by their participation to the final solid B-rep.

Both key aspects of our algorithm are made possible by embedding all input solids in a single KD-tree exploration. The algorithm is able to retrieve the final result directly because each new KD-tree node explored is classified as soon as it is created, by inferring whether its contents is completely inside or outside the final solid, or if it may participate to the final solid boundary instead. The KD-tree is only subdivided in the latter case, pruning large sets of primitives that do not need additional work, and focusing all computational effort and refinement on those space cells containing intersecting primitives participating to the final result.

Consequently, the depth and size of our KD-tree depends on the output model size instead of input model size, leading to gains in time and space complexity that grow with the number of input solids and the input-to-output primitive ratio. This means that our algorithm also outperforms the traditional two-solid boolean algorithms as soon as the result size is smaller than the input size. Finally, as our KD-tree decomposes space into non-intersecting cells that can be processed independently, the classification stage and subsequent subdivision and reconstruction stages naturally lend themselves to parallel evaluation, further substantiating the temporal gain over all state-of-the-art polyhedral boolean evaluation algorithms tested, including recent GPU implementations.

## Contribution summary.

- Complexity: we improve asymptotic upper bounds for running time and memory consumption, both analytically and experimentally, with a simple and parsimonious output-sensitive algorithm;
- Any boolean operation: QuickCSG directly computes the result of arbitrary boolean expressions over a set of input solids, certain classes of which cannot be realistically computed with existing binary boolean approaches;
- Applications on practical use cases: solid modeling for 3D printing, collision detection, and silhouette-based 3D reconstruction from visual inputs;
- Benchmark: we introduce a benchmark comprised of several dozen datasets, made available to the research community with our implementation.

The paper is organized as follows. We first review state of the art of boolean operations (Section 2), then propose a formalization of the  $N$ -solid boolean problem (Section 3). We then explain how vertices (Section 4) and facets (Section 5) of the final polyhedron can be directly identified. We show how the problem can be cast as a KD-tree divide-and-conquer exploration (Section 6), and how the algorithm and this exploration in particular can be performed in parallel (Section 7). We validate the algorithm experimentally in Section 8 and discuss applications in Section 9.

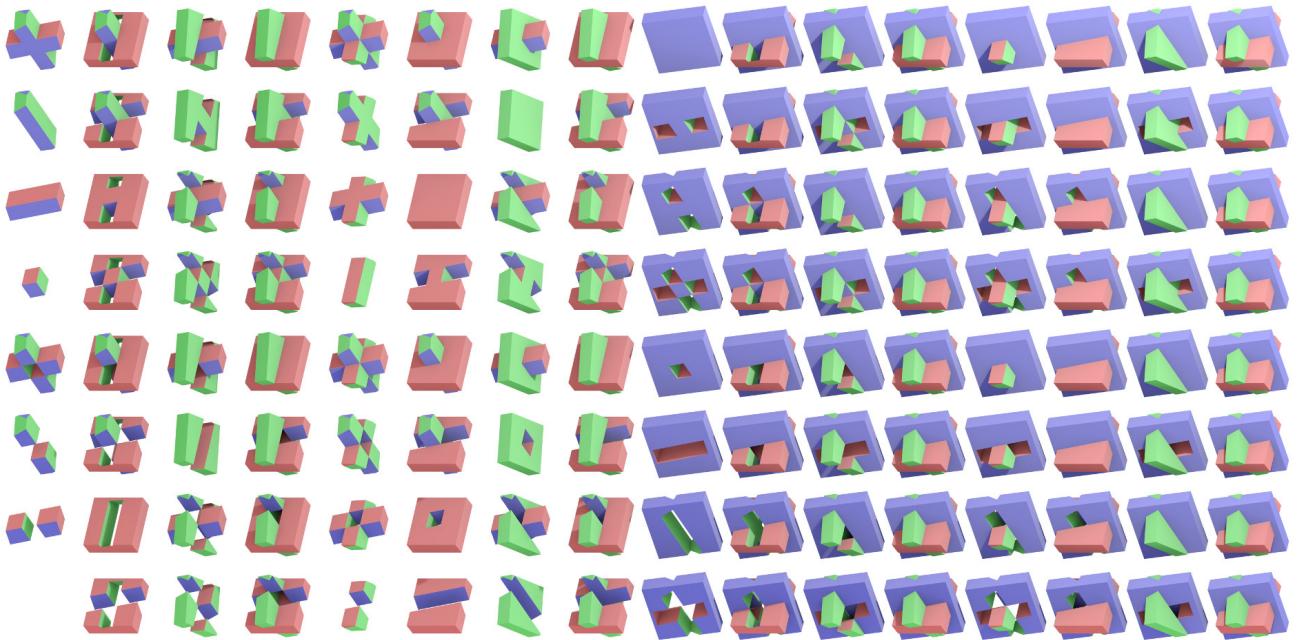


Figure 2: There are  $2^{2^3} = 256$  possible boolean functions of three inputs, of which we show 128, applied to 3D cuboids (the 128 other ones are the same with inside and outside flipped). In our figures, each input solid is assigned a color, which is inherited by the output facets it contributed to the result. Some functions cannot be computed from binary CSG operations without producing degeneracies. For example, the 2nd solid in reading order represents the masking operation  $(\mathcal{P}_1 \cap \mathcal{P}_2) \cup (\mathcal{P}_3 \setminus \mathcal{P}_2)$  with  $\mathcal{P}_2$  in green,  $\mathcal{P}_3$  in red and  $\mathcal{P}_1$  in blue. The union operation in this expression is degenerate because facets of  $\mathcal{P}_2$  appear on both sides.

## 2 Related Work

### 2.1 Boolean Solid Modelling Background

In the 1970's, boolean solid modeling and boundary representations (B-Reps) have been simultaneously pioneered in the context of computer graphics Braid (1975) and computer vision Baumgart (1974). Both proposed discrete representations of solid boundaries as a conjunction of simpler polygonal primitives, either winged edges (Baumgart) or loops (Braid). While many of the ideas are already present in Braid's work, the idea that solids could be specified as a tree of boolean operations (Constructive Solid Geometry or CSG) was theorized by Requicha (1980), and various practical implementations proposed for polyhedral boundaries Requicha and Voelcker (1985); Laidlaw et al. (1986), in particular setting the standard for the aforementioned 3-stage canvas. Robustness was then identified as a recurring issue, due to lack of formal description of degenerate solid configurations, and numerical computation error in near-coincident situations. Most algorithms thereafter, including industrial implementations, thus conform to a set-based formalism with algebraically closed regularized boolean set operations Requicha (1977), ensuring results exclude any non-volume enclosing (dangling) surface primitives. Several works also took on the task of painstakingly accounting for all degenerate relative configurations of solid primitives Hoffmann (1989); Mäntylä (1987), leading to tedious algorithm descriptions. They notably formalize the B-Rep primitive hierarchy as ver-

tex, edges, faces and shells, and the two-polyhedra intersections and degeneracy cases as arising from the possible intersection combination of each primitive type of solid A to each primitive type of solid B. To avoid the complete enumeration, many implementations focus on generic triangle-to-triangle or polygon-to-polygon as their central intersection unit. Even with this simplification, the complexity of dealing with all cases is known to yield unreliable implementations, including in commercial software, as reported in various test cases Wang (2011); Feito et al. (2013). Our algorithm has a significantly simplified core that focuses all classification and subdivision efforts on producing the final output vertices, excluding higher order primitives or intermediate vertices. The final reconstruction stage operates on these vertices, identifying the topologically correct final edges and loops through a posteriori logical vertex-to-vertex reconnections. This improves both the clarity and regularity of the proposed algorithm, and paves the way for tackling the otherwise unaffordable exact topology retrieval in the general N-polyhedron case.

Robustness has remained a dominant issue, with various solutions proposed reviewed by e.g. Hoffmann (2001); Li et al. (2004), such as geometric predicate analysis and fixed or arbitrary precision exact arithmetics. This effort has culminated with Hachenberger's work on CGAL Hachenberger et al. (2007), which uses arbitrary precision arithmetic, with the particularity that it follows Nef's formalism Nef (1978) instead of regularized booleans Requicha (1977), i.e. it explicitly represents dangling primitives. While now standing out as a reference implementation of the research community, it is

notoriously slow, and as most exact schemes, tedious to re-implement, leading to a somewhat paradoxical status: while the perception of the community is that the polyhedral B-Rep boolean problem is solved, free and commercial code is still being crafted and distributed using fragile but fast and memory-efficient geometric predicate evaluations.

Most new contributions in this area are focusing on speeding up or easing the implementation of various algorithmic work cases of the classic two-polyhedron boolean algorithms, with e.g. new specialized data structures Campen and Kobbelt (2010), faster exact arithmetic types Bernstein and Fussell (2009), optimization for particular inputs such as triangular meshes Feito et al. (2013) or polyhedral cones of arbitrary basis Franco and Boyer (2009). Each implementation relies on specific and non-optimal tradeoffs between implementation complexity, speed, memory footprint, input genericity, robustness (or lack thereof). We simultaneously improve over all problematic aspects: our greedy pruning scheme eliminates the need to compute any intermediate geometry and thus improves the complexity, robustness, memory footprint and execution time while enabling multi-arity boolean operations on  $n$  polyhedra, including but not limited to binary boolean trees. As this result strongly relies on the careful use of hierarchical subdivision structures, we specifically review this aspect of prior art in the following section.

## 2.2 Subdivision Structures for Efficient Computation

Because of the need for efficient subdivision and classification stages in the algorithm, a substantial research effort has been devoted to hierarchical structures in the context of boolean solid modelling. Some of the earliest axis-aligned plane-separation structures in this context are the polytrees Carlstrom (1987) and extended octrees, which embed the polyhedral B-Rep primitives in their nodes Brunet and Navazo (1990). Binary space partitions (BSP) of polyhedral B-Rep were devised as a way to more efficiently store the polyhedron, where separating planes are based on input facets Thibault and Naylor (1987). The most common strategy to compute boolean combinations of two solids with these representations is to perform simultaneous traversal of both hierarchies to isolate intersecting primitives Brunet and Navazo (1990), or similarly compute a merged BSP tree itself representing the result Naylor et al. (1990). Recent reference implementations continue to use variants of these seminal approaches, e.g. CGAL uses KD-trees as accelerated axis-aligned plane separating search structures Hachenberger et al. (2007), GTS uses axis-aligned bounding box (AABB) trees Popinet (2006), while Carve CSG uses octrees Sargeant (2011). A number of hybrid variants exist, which seek simultaneous benefit from the access simplicity of the octree structure and the representational flexibility of BSPs Adams and Dutré (2003).

Of significant interest among such hybrid methods, Pavic et al. (2010) examine the boolean CSG binary tree with a single octree to embed all input geometry, subdivi-

vide cells down to a fixed cell size as long as two input solids are volumetrically present, then classify each leaf cell after subdivision by evaluating the CSG boolean tree expression. A key difference with our proposal is that the resulting meshes are stitched with an approximate local triangulation at intersecting leaf cells, while we compute true surface-to-surface boolean contributions, for arbitrary boolean expressions that need not be expressed with a boolean tree. Feito et al. (2013) uses a similar octree subdivision triggered by general two-surface presence, but focuses only on triangular meshes and the two-solid case. Fundamentally for both approaches it can be noted that classification is still independently computed and not used to guide the subdivision.

A compelling case we make in this paper is that separating subdivision and classification stages, as done by all boolean B-Rep algorithms we are aware of, leads to an inherently suboptimal boolean algorithm. In light of the review of prior art, this is because the hierarchical structures proposed are in the vast majority of cases constructed as alternate representations of *each individual input solid*, decomposing its geometric details with trees of logarithmic depth in that input solid's size. In contrast, our algorithm builds a single geometric decomposition, splitting nodes according to a partial classification of their content computed on the fly. Branches not contributing to the resulting solid are pruned away, building a tree whose nodes are focused on the final surface geometry, with a depth logarithmic in the number of intersections present in the *resulting solid* instead. This yields two fundamental improvements over state of the art. First the algorithmic complexity is improved as it is now proportional to the logarithm of the resulting solid size. Second, because this tree classifies final contributions on the fly during subdivision, our algorithm stores only the information of the currently explored tree branch. This frees the algorithm from storing the subdivisions structures of each input solid in preparation for a separate classification stage, as with previous methods.

## 3 N-Polyhedron CSG Formalization

This section introduces the representation of the input solids and the CSG operation.

### 3.1 Definitions and Assumptions

We consider  $n$  input polyhedra  $\{\mathcal{P}_i\}_{i \in \{1, \dots, n\}}$ , whose surfaces are assumed to be closed orientable 2-manifolds embedded in  $\mathbb{R}^3$ , i.e. surfaces with no holes and with a consistent normal orientation. These classical assumptions ensure every polyhedron non-ambiguously defines a closed volume of  $\mathbb{R}^3$ . Each input polyhedron  $\mathcal{P}_i = (\mathcal{V}_i, \mathcal{F}_i)$  is defined by its set of vertices  $\mathcal{V}_i$  and facets  $\mathcal{F}_i$ , each described as a loop of vertex indices whose order is consistent, e.g. given with counterclockwise orientation as seen from its outer region. We assume unicity of vertices, i.e. no vertex coordinates are duplicated

and adjacent loops share common vertices. A polyhedron may have various connected components. Facets are assumed convex and described with a single loop to simplify the explanation and implementation, although the reasoning extends to general, non-convex polygons that may contain holes.

The resulting shape may be complex, as any output facet may be shaped by arbitrary primitives of all inputs. The complexity of possible degeneracies between all types of primitives for two-polyhedron booleans is already quite daunting and error-prone to implement Hoffmann (1989); ?. Generalizing Hoffmann’s analysis to  $n$ -case degeneracies is not desirable, because the combinatorial possibilities of coincidental positioning of vertices, edges and facets are arbitrarily large. As an example, degenerate output vertices may arise from the coincidental positioning of anywhere from 4 to  $n$  input facets chosen among any input solid’s facets, all of which would result in different special cases for reconstructing the vertex neighborhood. In practice, implementations most often get away with using double-precision floating-point arithmetic, as degeneracy cases are shown to be highly unlikely when dealing with noisy inputs, either resulting from an acquisition process Curless and Levoy (1996); Franco and Boyer (2009) or artificially generated with jittering for this purpose. We follow this approach in QuickCSG.

### 3.2 Boolean Functions of $n$ inputs

Instead of the usual CSG tree form of boolean expressions, we provide a framework for arbitrary expressions. We express a boolean solid operation using a boolean-valued function over  $n$  boolean inputs,  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ . We note  $\mathbb{I}_i(x) \in \{0, 1\}$  the indicator function of polyhedron  $\mathcal{P}_i$ , whose value reflects whether a point  $x \in \mathbb{R}^3$  is in polyhedron  $\mathcal{P}_i$ ’s inner volume. The indicator function  $\mathbb{I}_f(x)$  of the final solid  $\mathcal{P}_f$  can then be computed using  $f$ :

$$\mathbb{I}_f(x) = f(\mathbb{I}_1(x), \dots, \mathbb{I}_n(x)). \quad (1)$$

We define the *indicator vector* of point  $x$  as the tuple of its  $n$  indicator functions,  $\mathbf{I}(x) = (\mathbb{I}_1(x), \dots, \mathbb{I}_n(x))$ , and denote the CSG operation as occurring over its indicator vector, *i.e.*  $\mathbb{I}_f(x) = f(\mathbf{I}(x))$ . Note that, as  $\mathcal{P}_i$  is given as a set of vertices and faces  $(\mathcal{V}_i, \mathcal{F}_i)$ , indicator function values  $\mathbb{I}_i(x)$  can be computed by shooting a ray and counting the winding numbers of  $x$  Schneider and Eberly (2003). If a vertex is known to belong to the surface of a polyhedron, we denote the corresponding boolean value as ‘ $s$ ’, see Figure 3. Note that we never compute function  $f$  on  $s$  inputs.

Any indicator function  $\mathbb{I}_f(x)$  can be evaluated from classical binary boolean operators (*e.g.* as a conjunction of disjunctions), but alternative evaluations are also possible based on, for instance, higher arity boolean operators or arithmetic operations. The rationale is to simplify the expression of some operations and speed up the evaluation. The following examples show a few operators whose

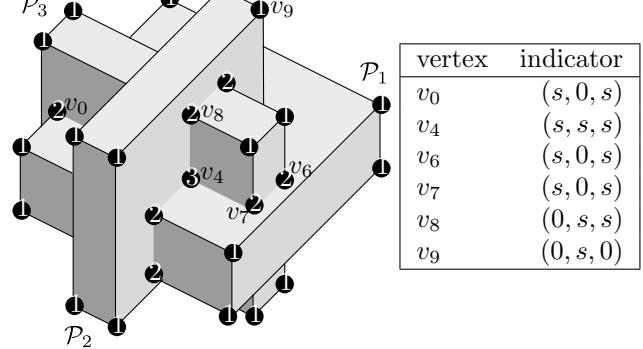


Figure 3: Combination of three solids, with the orders of the vertices (in black circles). Left: the indicator vector for some of the vertices.

$n$ -ary formulation enables efficient evaluations:

$$n\text{-Intersection: } \mathbb{I}_{\cap}(x) = \min(\mathbb{I}_1(x), \dots, \mathbb{I}_n(x)), \quad (2)$$

$$n\text{-Union: } \mathbb{I}_{\cup}(x) = \max(\mathbb{I}_1(x), \dots, \mathbb{I}_n(x)), \quad (3)$$

$$\text{Mutual exclusion: } \mathbb{I}_{\text{xor}}(x) = \mathbb{I}_1(x) \text{ xor } \dots \text{ xor } \mathbb{I}_n(x), \quad (4)$$

$$\text{In } k \text{ or more solids: } \mathbb{I}_{\min-k}(x) = (\sum_i \mathbb{I}_i(x)) \geq k. \quad (5)$$

The  $\min-k$  operation retrieves the solid being part of at least  $k$  input polyhedra. This operation in particular would be tedious to decompose over a binary CSG tree: it requires to evaluate the union of all possible intersections of  $k$  solids, leading to a tree of combinatorial size.

Since the indicator vector can be efficiently represented as a bit vector stored in machine words, evaluating typical boolean functions  $f$  is for most practical purposes a constant time operation in  $n$ , either by directly evaluating a boolean test expression over the machine word, or by building lookup/hash tables for compatible expressions. Any binary tree of CSG operations can be expressed as a single boolean function. Therefore, models built by binary CSG operations can be directly constructed by our algorithm. We will see below how these definitions are used to make final boundary surface decisions.

## 4 Final Polyhedron Vertices

We now analyze the structure of the final polyhedron  $\mathcal{P}_f$ , focusing on its vertices. With primitives in generic (*i.e.* non coincidental) position as assumed, vertices of the final polyhedron can be of only three types (Figure 3):

- *First order vertices* are vertices already present in one of the input polyhedra  $\mathcal{P}_i$ ’s vertex set  $\mathcal{V}_i$ .
- *Second order vertices* result from the intersection of an edge of a polyhedron  $\mathcal{P}_i$  and the facet of another polyhedron  $\mathcal{P}_j$ .

- *Third order vertices* result from the intersection of three facets of three different polyhedra  $\mathcal{P}_i$ ,  $\mathcal{P}_j$ , and  $\mathcal{P}_k$ .

Thus, the order of a vertex is the number of  $s$  bits in its indicator vector. A trivial way to generate all possible vertex candidates is to examine all input vertices, edge-to-facet combinations, and three-facet combinations, and compute the resulting geometric intersections using standard algorithms Schneider and Eberly (2003). Input primitives may intersect at various locations in space, without necessarily participating to the final surface, as determined by the CSG function. We thus propose a classification process to select the candidate vertices participating to the output result. Our description is illustrated on the left column of Figure 4, which summarizes the geometry of vertices of each order, and the notations used. In this figure, orientation information is given in red, and classification information in green. Small green grids are given to break down local subvolume configurations around the vertex  $v$ , and their corresponding classification bits, which are defined hereunder.

## 4.1 Vertex Classification

Intuitively a necessary condition for a vertex candidate to be kept is for it to lay at the border of the final solid, in other words it should be part of a surface transition from inside to outside  $\mathcal{P}_f$ . But this condition is not always sufficient.

The condition is sufficient for a **first order vertex**. By definition, the candidate vertex  $v$  participates in the boundary of one input polyhedron  $\mathcal{P}_i$ , which separates the vicinity of the vertex into two subvolumes, inside and outside  $\mathcal{P}_i$ . For this vertex to lay on the boundary of  $\mathcal{P}_f$ , it must also partition the surrounding volume into inside and outside regions of  $\mathcal{P}_f$ . This information can be obtained by examining how  $f(\mathbf{I}(v))$  transitions at the boundary of  $\mathcal{P}_i$ , *i.e.* when the  $i$ -th bit of the indicator vector, initially a  $s$  bit, is flipped between 0 and 1:

$$f(\mathbb{I}_1(v), \dots, 0, \dots, \mathbb{I}_n(v)) \neq f(\mathbb{I}_1(v), \dots, 1, \dots, \mathbb{I}_n(v)). \quad (6)$$

This conditions is noted **isFinal1( $v$ )**, and tests whether there is a final indicator change when the boundary of  $\mathcal{P}_i$  is traversed. If the two expressions were equal, then the vertex  $v$  would be completely inside (both 1) or outside (both 0)  $\mathcal{P}_f$ . This assumes all other bits  $\mathbb{I}_j(v)$ , with  $j \neq i$ , were computed for vertex  $v$  by *e.g.* ray shooting.

**Second order vertex.** Because facets of two polyhedra  $\mathcal{P}_i$  and  $\mathcal{P}_j$  are involved, the volume surrounding the vertex candidate is locally partitioned in four subvolumes, each of which may be decided to be inside or outside the final polyhedron  $\mathcal{P}_f$  by the CSG function  $f$ . We must therefore examine how each combination of boundary traversals at vertex  $v$  influence the final indicator function, by evaluating the corresponding  $i$  and

$j$  bit-flippings of  $\mathbf{I}(v)$ :

$$\begin{aligned} b_{00} &= f(\mathbb{I}_1(v), \dots, 0, \dots, 0, \dots, \mathbb{I}_n(v)) \\ b_{01} &= f(\mathbb{I}_1(v), \dots, 0, \dots, 1, \dots, \mathbb{I}_n(v)) \\ b_{10} &= f(\mathbb{I}_1(v), \dots, 1, \dots, 0, \dots, \mathbb{I}_n(v)) \\ b_{11} &= f(\mathbb{I}_1(v), \dots, 1, \dots, 1, \dots, \mathbb{I}_n(v)). \end{aligned} \quad (7)$$

We call  $\mathbf{b}(v) = (b_{00}, b_{01}, b_{10}, b_{11}) \in \{0, 1\}^4$  the classification vector of second order vertex  $v$ . Trivially, if all four bits turn out equal, the vertex  $v$  is either completely inside (all 1's) or outside (all 0's) of  $\mathcal{P}_f$  and does not participate to the final surface. Vertex candidates may lay on the final boundary and still not participate to the final surface description if the vertex is introduced in the middle of a final edge. This happens as soon as the bit pattern of  $v$  is symmetric along one of the components  $i$  or  $j$ , which means that traversing the vertex along this border does not change the final primitive participation of the other polyhedron. A sufficient condition can thus be written as the predicate **isFinal2( $v$ )**, which rules out any topological symmetries along the  $i$  or  $j$  components, as underlined in subscripts:

$$((\underline{b}_{00}, \underline{b}_{01}) \neq (\underline{b}_{10}, \underline{b}_{11})) \wedge ((\underline{b}_{00}, \underline{b}_{10}) \neq (\underline{b}_{01}, \underline{b}_{11})) \quad (8)$$

It can be noted that this condition includes the necessary conditions, since complete inclusion or exclusion is also a case of pattern symmetry.

**Third order vertex.** At the intersection locus of three facets from three input polyhedra  $\mathcal{P}_i$ ,  $\mathcal{P}_j$ ,  $\mathcal{P}_k$ , the vertex's neighborhood is locally split in eight subvolumes. The analysis is analogous to second order vertices, and requires examining the influence of crossing the three boundaries. The corresponding 8 combinations of  $i$ ,  $j$  and  $k$  bit-flippings are:

$$\begin{aligned} b_{000} &= f(\mathbb{I}_1(v), \dots, 0, \dots, 0, \dots, 0, \dots, \mathbb{I}_n(v)) \\ b_{001} &= f(\mathbb{I}_1(v), \dots, 0, \dots, 0, \dots, 1, \dots, \mathbb{I}_n(v)) \\ &\dots \\ b_{111} &= f(\mathbb{I}_1(v), \dots, 1, \dots, 1, \dots, 1, \dots, \mathbb{I}_n(v)). \end{aligned} \quad (9)$$

We call  $\mathbf{b}(v) = (b_{000}, \dots, b_{111}) \in \{0, 1\}^8$  the classification vector of third order vertex  $v$ . Similarly to the order-2 case, complete inclusion or exclusion of the volume rules out the vertex, as well as any axis symmetries, which can be jointly evaluated with the predicate **isFinal3( $v$ )**:

$$\begin{aligned} &((\underline{b}_{000}, \underline{b}_{001}, \underline{b}_{010}, \underline{b}_{011}) \neq (\underline{b}_{100}, \underline{b}_{101}, \underline{b}_{110}, \underline{b}_{111})) \\ &\wedge ((\underline{b}_{000}, \underline{b}_{001}, \underline{b}_{100}, \underline{b}_{101}) \neq (\underline{b}_{010}, \underline{b}_{011}, \underline{b}_{110}, \underline{b}_{111})) \\ &\wedge ((\underline{b}_{000}, \underline{b}_{010}, \underline{b}_{100}, \underline{b}_{110}) \neq (\underline{b}_{001}, \underline{b}_{011}, \underline{b}_{101}, \underline{b}_{111})) \end{aligned} \quad (10)$$

If none of these conditions are met, the vertex is completely inside or outside the result polyhedron. If one (resp. two) of the conditions are met, the vertex is on a facet (resp. edge) of the final polyhedron.

**Specificity of the 2-polyhedron case.** Interestingly, in this situation, there are only first and second order vertices with no axis symmetries, *i.e.* all order two vertex candidates participate to  $\mathcal{P}_f$ , which has been analyzed by Franco et al. (2013).

<p>First order vertex</p> <p>Predicate: <math>b_0 \neq b_1</math></p>	<p>Looplets generated on <math>F_1</math></p> <p><math>b_0 = 0</math>      <math>b_1 = 1</math>  <math>(d_{13}, v, d_{12} F_1^+)</math></p> <p><math>b_0 = 1</math>      <math>b_1 = 0</math>  <math>(d_{21}, v, d_{31} F_1^-)</math></p>
<p>Second order vertex</p> <p>Predicate:  <math>((b_{00}, b_{01}) \neq (b_{10}, b_{11}) \wedge</math>  <math>(b_{00}, b_{10}) \neq (b_{01}, b_{11}))</math></p>	<p>Looplets generated on <math>F_1</math> and <math>F_3</math></p> <p><math>b_{00} = 0</math>, <math>b_{01} = 0</math>, <math>b_{10} = 1</math>, <math>b_{11} = 1</math>:  <math>(d_{13}, v, d_{21} F_1^+)</math>, <math>(d_{23}, v, d_{31} F_3^+)</math>  <math>b_{00} = 1</math>, <math>b_{01} = 1</math>, <math>b_{10} = 0</math>, <math>b_{11} = 0</math>:  <math>(d_{12}, v, d_{31} F_1^-)</math>, <math>(d_{31}, v, d_{32} F_3^-)</math></p> <p><math>b_{00} = 0</math>, <math>b_{01} = 1</math>, <math>b_{10} = 1</math>, <math>b_{11} = 0</math>:  <math>(d_{12}, v, d_{31} F_1^-)</math>, <math>(d_{13}, v, d_{32} F_3^-)</math>  <math>b_{00} = 1</math>, <math>b_{01} = 0</math>, <math>b_{10} = 0</math>, <math>b_{11} = 1</math>:  <math>(d_{12}, v, d_{21} F_1^-)</math>, <math>(d_{23}, v, d_{13} F_3^-)</math></p>
<p>Third order vertex</p>	<p>Looplets generated on <math>F_2</math></p> <p><math>b_{00} = 0</math>, <math>b_{01} = 1</math>, <math>b_{10} = 0</math>, <math>b_{11} = 0</math>:  <math>(d_{12}, v, d_{32} F_2^+)</math>  <math>b_{00} = 1</math>, <math>b_{01} = 0</math>, <math>b_{10} = 0</math>, <math>b_{11} = 1</math>:  <math>(d_{23}, v, d_{21} F_2^+)</math>  <math>b_{00} = 0</math>, <math>b_{01} = 1</math>, <math>b_{10} = 1</math>, <math>b_{11} = 0</math>:  <math>(d_{23}, v, d_{21} F_2^-)</math>  <math>b_{00} = 1</math>, <math>b_{01} = 0</math>, <math>b_{10} = 1</math>, <math>b_{11} = 1</math>:  <math>(d_{12}, v, d_{32} F_2^-)</math></p>
<p>For <math>r, s \in \{0, 1\}</math> we define <math>t_{rs}^+ := ((b_{0rs}, b_{1rs}) = (0, 1))</math> and <math>t_{rs}^- := ((b_{0rs}, b_{1rs}) = (1, 0))</math></p> <p>Looplets generated on <math>F_1</math> on quadrant <math>(r, s) = (0, 0)</math>:</p>	<p><math>t_{01}^+ = 0</math>, <math>t_{10}^+ = 0</math>, <math>t_{00}^+ = 1</math>:  <math>(d_{21}, v, d_{31} F_1^+)</math>  <math>t_{01}^+ = t_{01}^+ = t_{10}^+ = t_{11}^+ = 1</math>, <math>t_{00}^+ = 0</math>:  <math>(d_{13}, v, d_{12} F_1^+)</math>  <math>t_{01}^- = 0</math>, <math>t_{10}^- = 0</math>, <math>t_{00}^- = 1</math>:  <math>(d_{13}, v, d_{12} F_1^-)</math>  <math>t_{01}^- = t_{10}^- = t_{11}^- = 1</math>, <math>t_{00}^- = 0</math>:  <math>(d_{21}, v, d_{31} F_1^-)</math></p>

Figure 4: Vertex configurations and their corresponding possible looplets.

## 4.2 Vertex Retrieval Summary

We illustrate in Figure 5 how the set of  $\mathcal{P}_f$ 's vertices may be retrieved using a simple but quartic worst-case complexity algorithm which loops over all 1, 2 and 3-facet combinations, using the previously defined ISFINAL1, ISFINAL2, and ISFINAL3 predicates. These predicates include rayshooting operations to compute the vertex's indicator vector, a linear operation with all input facets. The algorithm uses classical intersection functions Schneider and Eberly (2003): INTERSECT2FACETS computes the intersected edge between two convex facets, as a pair of vertices giving the edge extremities, and INTERSECTSEGMENTFACET computes the vertex representing the intersection of a segment and a facet, if any. Note that the quartic behaviour is in practice mitigated by the fact that the third order loop is only executed if a pair of intersecting input faces was already encountered.

## 5 Final Polyhedron Connectivity

Once the subset of final vertices  $\mathcal{V}_f$  is known through the classification process, the main task left is to identify how vertices are connected together to form the faces  $\mathcal{F}_f$  of the final polyhedron  $\mathcal{P}_f$ . Our method makes a clear distinction between computing all geometric coordinates of final vertices and building the final topology. While the former involves numerical coordinate construction, the latter relies only on orientation and ordering predicates.

The classification vectors introduced previously not only inform us of the participation of a given vertex  $v$  to  $\mathcal{P}_f$ , they also give a snapshot of volume and surface adjacencies around the vertex, as illustrated in Figure 4. We show here how to find all the polygons  $v$  participates in, by introducing the *looplet* construct. We define a looplet of  $v$  as a loop fragment running through this vertex, represented by a symbolic incoming and outgoing edge direction. Its purpose is to compactly represent all partial adjacency information available for individual unconnected vertices, while being independently computable at each vertex.

### 5.1 Surface and Edge Orientation

Surfaces and edges need to be oriented to define looplets. Surface boundaries contributing to the final result  $\mathcal{P}_f$  may change orientation (reversed normal), *e.g.* when participating in a boolean subtraction. For a facet  $F_a$  of any polyhedron, we note the orientation of its contributions to the final surface as either  $F_a^+$  if the contribution conserves initial orientation, and  $F_a^-$  if the orientation of the contribution is inverted. In similar spirit we need to define an intrinsic edge orientation  $d_{ab}$  for every edge adjacent to two faces  $F_a$  and  $F_b$ . For this purpose we distinguish two cases:

- if the edge is pre-existing from an input  $\mathcal{P}_i$ ,  $d_{ab}$  is the edge direction with polygon  $F_a$  on its left and  $F_b$  on its right on the oriented surface.

- if the edge arises from the intersection of  $F_a$  and  $F_b$ , we define the edge direction  $d_{ab} = F_a \times F_b$ , as the crossproduct of the corresponding face normals.

In both cases  $d_{ab} = -d_{ba}$ . With these definitions we can introduce a concise notation for looplets, as  $(d_{ab}, v, d_{ac}|F_a^+)$ , a looplet characterized as a positive contribution in  $F_a$ , centered on vertex  $v$ , with incoming edge direction  $d_{ab}$  and outward edge direction  $d_{ac}$ . Looplets can be symbolically and compactly stored as a vertex reference for  $v$ , an orientation bit, and three ordered face references ( $a,b,c$  in the above example) since the incoming and outgoing directions share the face  $a$  in their adjacencies. In the following, we will describe how classification vectors determine which looplets are present at a vertex, upon which the final polyhedron facets can be built. For this purpose, we break down the presentation of looplet generation cases for each vertex order. This breakdown is illustrated in the right column of Figure 4, where each subfigure shows in green the bit classification state deciding the presence of corresponding looplets, annotated under each figure in blue.

### 5.2 First Order Vertex Looplets

A first order vertex  $v$  that passed the classification test is on the surface boundary of a polyhedron  $\mathcal{P}_i$  and also on the boundary of  $\mathcal{P}_f$ . Each of its adjacent facets thus at least partially participates to  $\mathcal{P}_f$ , and contributes a looplet for this vertex. The directions of this looplet are given by the edges adjacent to the facets of the looplet. If there is no change of orientation at the vertex  $v$ , *i.e.*  $b_0 = 0$  and  $b_1 = 1$ , the edges and facet keep the orientation they had on the initial polyhedron, *e.g.* yielding a looplet  $(d_{13}, v, d_{12}|F_1^+)$  for facet  $F_1$  in Figure 4. In contrast, looplets are inverted in case of surface orientation change, *i.e.* when  $b_0 = 1$  and  $b_1 = 0$ , *e.g.* yielding looplet  $(d_{21}, v, d_{31}|F_1^-)$  for facet  $F_1$ .

### 5.3 Second Order Vertex Looplets

Second order vertices are the intersection of the edge of a polyhedron  $\mathcal{P}_i$  and the facet of a polyhedron  $\mathcal{P}_j$ . As such it involves three facets, one facet labeled  $F_2$  from  $\mathcal{P}_j$ , and two facets  $F_1$  and  $F_3$  from  $\mathcal{P}_i$ , adjacent to the edge yielding  $v$  from intersection with facet  $F_2$ . We here assume facet orientation as in Figure 4, without loss of generality: if the normal of one of the two surfaces is inverted, swapping  $F_1$  and  $F_3$  brings us back to this reference configuration.

The crossing surfaces at the vertex define four boundaries between four subvolumes, each with two possible orientations. In the case of  $F_2$ , each of the two subvolume boundaries may yield one looplet in each orientation, *e.g.* for the top subvolume boundary in Figure 4 the two possible looplets are  $(d_{12}, v, d_{32}|F_2^+)$  and  $(d_{23}, v, d_{21}|F_2^-)$ . One of the two possible looplets is generated for a subvolume boundary, as soon as the classification bits of the two subvolumes it separates have different values. The looplet generated is the one consistent with the orientation of the final surface, with positive normal going from

```

function CSGVERTICES
  Input:  $\mathcal{V}, \mathcal{F}$ : set of vertices and facets of input polyhedra
  Output:  $\mathcal{V}_f$ : corresponding set of final output vertices
  for  $F_1$  in  $\mathcal{F}$  do                                 $\triangleright$  Enumerate input faces
    for  $v$  in  $F_1$  do                       $\triangleright$  Order-1 candidates
      if ISFINAL1( $v$ ) then  $\mathcal{V}_f \leftarrow \mathcal{V}_f \cup \{v\}$ 
    end for
    for  $F_2$  in  $\mathcal{F}$  do                                 $\triangleright$  Order-2 candidates
       $v_1, v_2 \leftarrow \text{INTERSECT2FACETS}(F_1, F_2)$ 
      if  $\{v_1, v_2\} = \emptyset$  then continue  $F_2$  loop
      if ISFINAL2( $v_1$ ) then  $\mathcal{V}_f \leftarrow \mathcal{V}_f \cup \{v_1\}$ 
      if ISFINAL2( $v_2$ ) then  $\mathcal{V}_f \leftarrow \mathcal{V}_f \cup \{v_2\}$ 
      for  $F_3$  in  $\mathcal{F}$  do                                 $\triangleright$  Order-3 candidates
         $v \leftarrow \text{INTERSECTSEGMENTFACET}(v_1, v_2, F_3)$ 
        if  $v \neq \emptyset$  and ISFINAL3( $v$ ) then  $\mathcal{V}_f \leftarrow \mathcal{V}_f \cup \{v\}$ 
      end for
    end for
  end for
end function

```

Figure 5: Brute-force algorithm to find all vertices of the result polyhedron.

the inside ( $b_{**} = 1$ ) to the outside ( $b_{**} = 0$ ) of the final volume.

The decision scheme is analogous for looplets of  $F_1$  and  $F_3$ . Looplets of these facets are always simultaneously decided as they are determined by the same classification bits.

#### 5.4 Third Order Vertex Looplets

The eight subvolumes surrounding  $v$  are separated by the three facets  $F_1, F_2, F_3$ . The configuration shown in Figure 4 assumes that the normals of these three facets form a right-handed trihedron. This is again without loss of generality, should one of the facets have an opposite normal, a single permutation in the order in which facets are considered brings us back to this reference configuration. Because the looplet possibilities are analogous for all three facet planes, we shall only enumerate the configurations for  $F_1$ . The enumeration also has a rotational symmetry within the facet plane, since it is divided in four quadrants by the other two facets. The four quadrants are indexed with  $(r, s) \in \{0, 1\}^2$  tuples. Figure 4 shows that there are four possible looplets for a quadrant, bringing the total possible order-three looplet count to  $3 \times 4 \times 4 = 48$  for a vertex. We focus our description on one quadrant of  $F_1$  where  $(r, s) = (0, 0)$ . To ease the description, we introduce two intermediate boolean predicates  $t_{rs}^+, t_{rs}^-$  for each quadrant  $(r, s)$  of  $F_1$ . They are computed as  $t_{rs}^+ := ((b_{0rs}, b_{1rs}) = (0, 1))$  and  $t_{rs}^- := ((b_{0rs}, b_{1rs}) = (1, 0))$ , indicating whether the surface portion corresponding to the quadrant participates to the final surface as a positive or negative contribution in the plane of  $F_1$ .

Looplet decisions involve examining several quadrant boundary predicates. Concave looplets exist if three of the four quadrant boundaries exist for an orientation and the fourth doesn't, i.e.  $(d_{13}, v, d_{12}|F_1^+) \exists$  exists if  $((t_{00}^+, t_{01}^+, t_{10}^+, t_{11}^+) = (0, 1, 1, 1))$ , and  $(d_{21}, v, d_{31}|F_1^-) \exists$  exists if  $((t_{00}^-, t_{01}^-, t_{10}^-, t_{11}^-) = (0, 1, 1, 1))$ . On the other

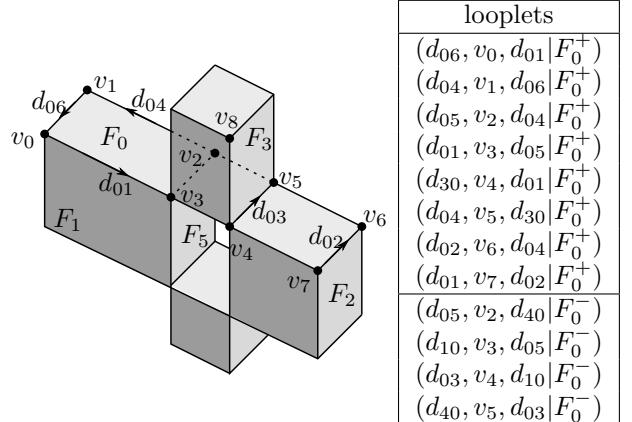


Figure 6: Result of the  $(\mathcal{P}_1 \text{xor} \mathcal{P}_2) \cap \mathcal{P}_3$  operation on the example of Figure 3. The table lists the looplets for facet  $F_0$ .

hand, convex looplets conditions depend only on three quadrant boundaries, the diagonally opposite quadrant in the facet having no influence. Concerning quadrant  $(0, 0)$  in Figure 4, convex looplet  $(d_{21}, v, d_{31}|F_1^+)$  exists if  $((t_{00}^+, t_{01}^+, t_{10}^+) = (1, 0, 0))$ , while the opposing looplet  $(d_{13}, v, d_{12}|F_1^-)$  exists if  $((t_{00}^-, t_{01}^-, t_{10}^-) = (1, 0, 0))$ .

#### 5.5 Retrieving Final Polyhedron Facets

Once all vertices and their looplets have been generated, they can be re-indexed for each facet to generate its contributions to  $\mathcal{P}_f$ . We process each input facet separately, which improves the locality of the algorithm and reduces it to 2D. The corresponding algorithm is in Figure 7 and an example of a result illustrated in Figure 6, where the focus is on contributions of  $F_0$ . Within a finally contributing facet, an arbitrary seed looplet is chosen and its outgoing direction followed, searching for sequentially matching looplets to close the loop. In some occurrences,

two or more looplets may match for a given direction: see for instance looplet  $(d_{06}, v_0, d_{01}|F_0^+)$  in Figure 6, for which both  $(d_{01}, v_3, d_{05}|F_0^+)$  and  $(d_{01}, v_7, d_{02}|F_0^+)$  match. In this case, the FIRST function selects the closest looplet in the search direction  $d_{01}$ , here  $(d_{01}, v_3, d_{05}|F_0^+)$ . The whole process may be repeated until there are no looplets left in the face. For  $F_0$ , this results in the following loops:

$$\begin{aligned} F_0^+ : \quad & v_0 \xrightarrow{d_{01}} v_3 \xrightarrow{d_{05}} v_2 \xrightarrow{d_{04}} v_1 \xrightarrow{d_{06}} v_0 \\ F_0^+ : \quad & v_4 \xrightarrow{d_{01}} v_7 \xrightarrow{d_{02}} v_6 \xrightarrow{d_{04}} v_5 \xrightarrow{d_{30}} v_4 \\ F_0^- : \quad & v_4 \xrightarrow{d_{10}} v_3 \xrightarrow{d_{05}} v_2 \xrightarrow{d_{40}} v_5 \xrightarrow{d_{30}} v_4 \end{aligned}$$

Two convex, diagonally opposing looplets may be triggered for a same vertex, as is the case for *e.g.*  $v_3$  or  $v_4$  in  $\mathcal{F}_1$ . Both negative and positive orientation facets may be generated for a single input facet, and may even be adjacent and share an edge, as for  $F_0$  in our example. Both of these configurations are typical of exclusive-or operations, but may happen with other operations. More generally, the algorithm can generate arbitrary output facets, with non-convex loops, several loops per facet (holes). Remarkably, our vertex-centered framework transparently accounts for all such possibilities.

Since facets may be arbitrarily large (see *e.g.* Figure 21 or Dithering in Figure 22), the cost of FIRST searches may be quadratic if implemented naively. To avoid this, output vertices can be sorted lexicographically by vertex index and offset on the edge, to obtain quasilinear searching, in  $\mathcal{O}(r \log r)$  with  $r$  the number of vertices in the output facet. At this stage the algorithm produced no superfluous geometry: all vertices and edges are required to represent the output polyhedron. However, non-convex and non-0 genus polygons are hard to manipulate, render or even feed back as input to the algorithm. Therefore, we typically tessellate the output facets to triangles or convex polygons, also an  $\mathcal{O}(r \log r)$  operation. This only happen once at finalization and never at an intermediate stage.

## 6 Hierarchical Algorithm

A fully functional approach can be implemented based on the two simple algorithms, CSGVERTICES and CS-GFACETS. CSGFACETS needs little tuning as it already runs quasilinearly over the output primitives identified by CSGVERTICES. However as previously noted the naive CSGVERTICES has an impractical quartic worst-case run time. We thus rely on a hierarchical space exploration to trigger CSGVERTICES only on tightly bounded space regions where output geometry is expected to be found. We choose a KD-tree based exploration for this purpose because of its simplicity, and its favorable reported performance compared to other datastructures for similar tasks Havran (2000).

The KD-tree Bentley (1975) is a binary space partition tree, whose nodes each represent a particular axis-aligned cuboid cell, containing all cells of its child nodes. It is typically built as a search datastructure for a set of points in a d-dimensional space, by recursively subdividing the input point set in two subsets of equal size,

using an axis-aligned split plane, achieving build complexities of  $\mathcal{O}(m \log m)$  time and  $\mathcal{O}(m)$  space with  $m$  input points de Berg et al. (2008). As applied to polyhedra, the KD-tree cells contain a list of polygons, cropped to the cell's bounding box. When polygons sit across a split plane, they are divided in two fragments inherited by both children of the parent node (Figure 8). Typically the cell subdivision is pursued down to a certain depth or until the number of primitives in the cell falls under a chosen bound, below which brute force search is more efficient.

### 6.1 KD-Tree Exploration for the Boolean Problem

Our procedure is similar to KD-tree construction, with the key differences that we never need to store the tree, and that our tree exploration is adaptive and output-sensitive. We want to subdivide only the nodes that may contain order-2 or -3 vertices of the final polyhedron, while pruning others, see Figure 10. This requires classifying KD-nodes during construction for every polyhedron involved. A volumetric cell can be classified as being fully inside or outside a polyhedron, but it can also straddle the polyhedron surface as soon as it contains surface primitives, and be thus undecided. We therefore need to extend the binary boolean logic to ternary as in Pavic et al. (2010). We define three corresponding indicator states  $\mathbb{I}_i(c)$  for a cell  $c$  with respect to polyhedron  $\mathcal{P}_i$ :  $\mathbb{I}_i(c) \in \{0, 1, u\}$  and the cell's indicator vector  $\mathbf{I}(c) = (\mathbb{I}_1(c), \dots, \mathbb{I}_n(c))$  as the tuple of the cell's ternary indicators. In turn, final classifications  $\mathbb{I}_f(c) = f(\mathbf{I}(c))$  of a cell  $c$  are computed with ternary logic, with:

$$f : \{0, 1, u\}^n \rightarrow \{0, 1, u\}. \quad (11)$$

Although the indicator vector of the cell may contain undefined bits, it is often possible to get a definite answer for the cell classification with  $f$ . For example, consider the intersection of  $n$  polyhedra, where  $f_{\cap}(\mathbf{I}(c)) = 1$  iff  $\mathbf{I}(c) = (1, 1, \dots, 1)$ . It suffices for the cell to be known outside of any polyhedron to conclude that the cell is outside the intersection volume, *i.e.*  $f_{\cap}(\dots, u, \dots, 0, \dots) = 0$ . Likewise for unions,  $f_{\cup}(\dots, u, \dots, 1, \dots) = 1$ , regardless of other bits, the cell is known to be inside the union as soon as it is in one of the input polyhedra. Truth tables for ternary versions of usual boolean operations are easy to build. Note that for specific hard functions such as  $f_{\text{xor}}$ , all bits of  $\mathbf{I}(c)$  must be defined ( $\neq u$ ) to compute a definite classification, in other words all order 2 and 3 vertices participate to the final surface. Nevertheless whatever the boolean function  $f$ , it always benefits from other generic pruning features of the algorithm discussed below.

### 6.2 Algorithm Summary

The algorithm is summarized in Figure 11. There are two pruning conditions. First, if the cell final classification  $f(\mathbf{I}(c))$  becomes completely determined, then it does not contain final surface primitives. Second, if the

```

function CSGFACETS
  Input:  $\mathcal{V}_f$ : set of vertices of the output
  Output:  $\mathcal{L}$ : final polyhedron facets as set of loops
   $\mathcal{F} \leftarrow \emptyset$ ,  $\mathcal{V} \leftarrow \{\}$ 
  for  $v$  in  $\mathcal{V}_f$  do
    for  $F$  in ADJACENTFACETS( $v$ ) do
       $\mathcal{F} \leftarrow \mathcal{F} \cup \{F^+\} \cup \{F^-\}$ 
       $\mathcal{V}[F] \leftarrow \mathcal{V}[F] \cup \{v\}$ 
    end for
  end for
  for  $F$  in  $\mathcal{F}$  do
     $l \leftarrow \emptyset$ 
    for  $v$  in  $\mathcal{V}[F]$  do
       $l \leftarrow l \cup \text{COMPUTELOOPLETS}(*, v, *|F)$ 
    end for
    while  $l \neq \emptyset$  do
       $(d_1, v, d|F) = \text{pop}(l)$ 
       $F' \leftarrow []$ 
      repeat
         $F' \leftarrow [F', v]$ 
         $(d', v, d|F) \leftarrow \text{FIRST}((d, *, *|F) \text{ in } l)$ 
      until  $d = d_1$ 
       $\mathcal{L} \leftarrow \mathcal{L} \cup F'$ 
    end while
  end for
end function

```

▷ Set of contributing facets, and their vertices  
 ▷ Collect looplets for all vertices

▷ Keep both orientations of facets  
 ▷ Facet vertex contributions

▷ Process each facet's two orientations  
 ▷ Looplets indexed by incoming direction  
 ▷ Collect looplets for all vertices of  $F$

▷ Looplets left for this facet  
 ▷ Pick and remove a looplet  
 ▷ Build this final facet  
 ▷ Chain looplet vertices

▷ Until back to start  
 ▷ Add this facet to final set

Figure 7: Algorithm to find all facets of the result polyhedron from looplets.

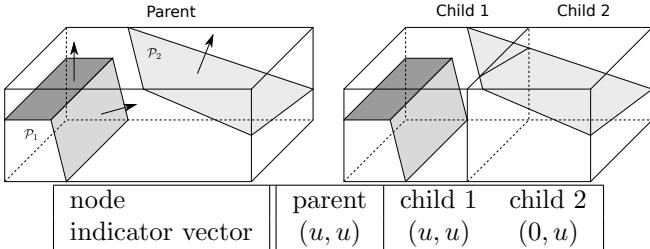


Figure 8: A KD-tree cell containing facets from  $\mathcal{P}_1$  and  $\mathcal{P}_2$  is split. The polygons stored in the KD-tree cells are cropped to the cell.

cell is still undetermined but only one polyhedron participates, then all of its primitives in the cell are part of the final polyhedron and no recursion is needed. This condition is expressed by counting the undefined bits of  $\mathbf{I}(c)$ , where  $\text{UNDEFINED}(\mathbf{I})$  denotes the set of indices of polyhedra whose surface still runs through the cell. As evidenced by the example of Figure 10, for large meshes, most of the explored nodes will fall in these two pruning cases, and the algorithm subdivisions can be seen as converging to the final surface's order-2 and order-3 intersections.

If the number of facets falls under a threshold  $F_{\max}$ , we fall back to CSGVERTICES to report final vertices. Experimentally  $F_{\max} = 20$  is found to be efficient in all situations. Specifics of local indicator evaluation in CSGVERTICES will be discussed in Section 6.4. Finally, if none of the previous conditions were met, the primitive

set can be split, and the two subtrees explored once their indicator status has been updated.  $\text{ISINSIDE}(\mathcal{V}^*, \mathcal{P}_i)$  returns a value in  $\{0, 1, u\}$ : 0 if all vertices of  $\mathcal{V}^*$  are outside  $\mathcal{P}_i$ , 1 if they are all inside,  $u$  otherwise. Since splitting is at the heart of the algorithm complexity, and  $\text{ISINSIDE}$  is performed jointly for the two children, we specifically address their implementation in Section 6.3.

### 6.3 Computing Splits and Node Indicators

Various split heuristics exist and are widely documented in the literature Havran (2000). They determine the tree balancing and the number of split polygon fragments, which may clutter the tree and its performance. Yet the split must be performed as rapidly as possible to keep overall runtime under control. The complexity should be  $O(m)$  with  $m$  the number of primitives. Pre-sorting input vertices along the axis to compute medians without re-sorting each node, or minimizing polygon splits with the Surface Area Heuristic (SAH) MacDonald and Booth (1990), are typical optimizations used to achieve  $O(m \log m)$  tree build performance with small overhead for various tasks Wald and Havran (2006). However, our objective is different, because entire subtrees are to be pruned and the split algorithm can hardly anticipate where. Minimizing the number of splits or finding the median of input polygon vertices is not as important to our algorithm as favoring large pruning possibilities. We have tried several advanced heuristics for pruning, but found experimentally that simply splitting

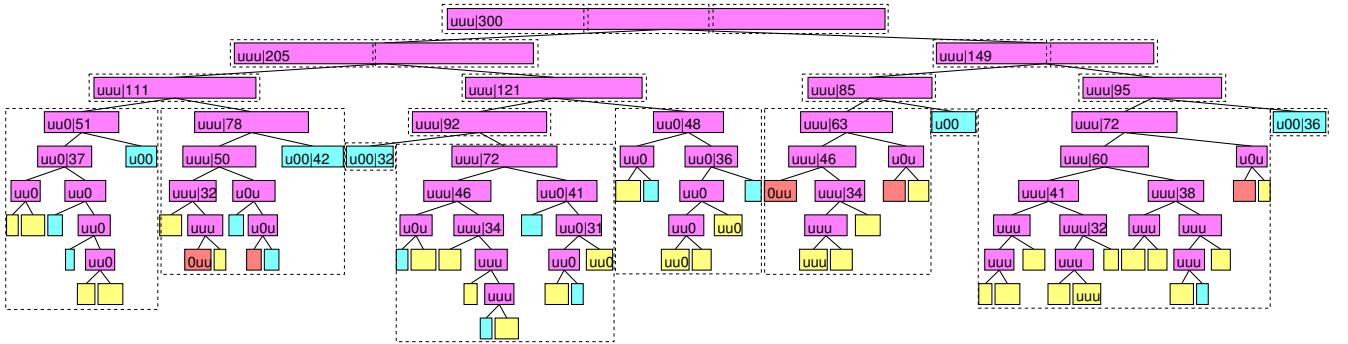


Figure 9: View of the KD-tree for operation  $\mathcal{P}_f = \mathcal{P}_1 \setminus (\mathcal{P}_2 \cup \mathcal{P}_3)$  applied to three simple meshes. Each box represents a node, whose width is proportional to its number of polygons. When space allows, text in the box indicates node's indicator vector and the number of polygons. Color code: █ = node that was split, █ = leaf where vertices were found with CSGVERTICES, █ = facets were just copied to the output, █ = node was found completely inside or outside the mesh. Each dashed box represents a parallel task.

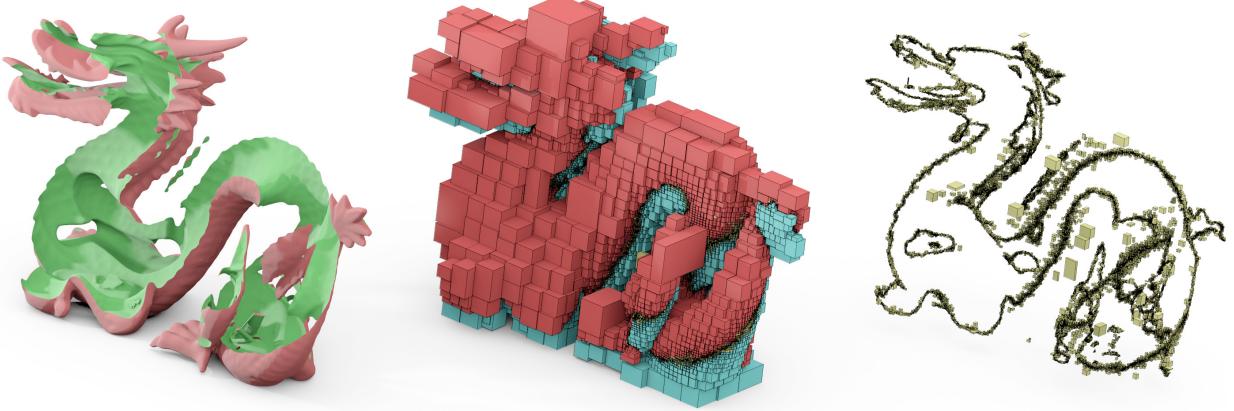


Figure 10: Left: Boolean difference between a red and a green dragon mesh, center: bounding boxes of leaf nodes of the KD-tree, right: the leaves where CSGVERTICES is called (on average, 0.56 order-2 vertices are found on each of these). Color code, center and right: █ = leaf where vertices were found with CSGVERTICES, █ = facets were just copied to the output, █ = node was found completely inside or outside the mesh. The yellow boxes are barely visible in the central representation because the vast majority of output does not require intersection computations.

```

function KDVERTICES
  Input:  $\mathcal{V}$ ,  $\mathcal{F}$ ,  $\mathbf{I}$ : cell's input vertices, facets and indicator vector
  Output:  $\mathcal{V}_f$ : cell's output vertices
  if  $f(\mathbf{I}) \neq u$  then
    return  $\emptyset$ 
  else if  $|\text{UNDEFINED}(\mathbf{I})| = 1$  then
    return  $\mathcal{V}$ 
  else if  $|\mathcal{F}| \leq F_{\max}$  then
    return CSGVERTICES( $\mathcal{V}$ ,  $\mathcal{F}$ )
  else
     $\mathcal{V}^1, \mathcal{F}^1, \mathcal{V}^2, \mathcal{F}^2 \leftarrow \text{SPLIT}(\mathcal{V}, \mathcal{F})$ 
     $\mathbf{I}^1, \mathbf{I}^2 \leftarrow \mathbf{I}$ 
    for  $i$  in  $\text{UNDEFINED}(\mathbf{I})$  do
       $\mathbf{I}^1[i] \leftarrow \text{ISINSIDE}(\mathcal{V}^1, \mathcal{P}_i)$ 
       $\mathbf{I}^2[i] \leftarrow \text{ISINSIDE}(\mathcal{V}^2, \mathcal{P}_i)$ 
    end for
    return KDVERTICES( $\mathcal{V}^1, \mathcal{F}^1, \mathbf{I}^1$ )  $\cup$  KDVERTICES( $\mathcal{V}^2, \mathcal{F}^2, \mathbf{I}^2$ )
  end if
end function

```

- ▷ Cell is completely inside or outside  $\mathcal{P}_f$
- ▷ Pruning, no surface primitive contained
- ▷ Only one active input  $\mathcal{P}_i$
- ▷ Pruning, all cell vertices participate to  $\mathcal{P}_f$
- ▷ Threshold cell size is reached
- ▷ Find cell's order 1,2,3 vertices
- ▷ Cell may still have final vertices
- ▷ Choose and execute split
- ▷ Subnode indicators are updated from  $\mathbf{I}$
- ▷ Enumerate still active  $\mathcal{P}_i$ 's
- ▷ Update i-th bit
- ▷ Update i-th bit

Figure 11: Hierarchical algorithm to find all vertices of the result polyhedron.

in the middle of the bounding box’s largest dimension yields excellent overall performance.

As soon as a the splitting plane is decided, a single  $O(m)$  pass can build the split primitive sets. This pass can also be used to compute new bounding boxes for each sub-tree. Checking whether child nodes intersect the  $i$ -th bounding box provides information to update the  $i$ -th bit of the child node’s indicator vector (which is denoted `ISINSIDE` in the algorithm of Figure 11). If it still intersects, the  $i$ -th bit stays undefined,  $u$ . If no longer active, the child node contains no  $i$ -th polygon fragment, but we still need to determine whether the cell is completely inside or outside the  $i$ -th polyhedron. Shooting a ray outside the current cell would require examining all input primitives and downgrade performance. Fortunately it is possible to answer the question locally by keeping a reference to extremal vertices in the split direction during the splitting pass. Namely, we examine the facets adjacent to the extremal vertex, and if they do not have the same normal orientation, we select the facet closest (most parallel) to the splitting plane for the orientation decision. In the example of Fig. 8, the normal of the  $\mathcal{P}_1$  polygons in child 1 are used to compute bit 1 of the indicator vector of child 2.

## 6.4 Computing the Indicator Vector of Leaf Points

Recall that CSGVERTICES requires ray shooting to compute indicators of candidate vertices in the general algorithm. In the context of the KD-exploration, this rayshooting occurs for calls of CSGVERTICES in leaves, and can be made local to keep the computational time bounded. Given the indicator vector  $\mathbf{I}(c)$  for a leaf node  $c$ , the  $i$ -th bit of the indicator vector  $\mathbb{I}_i(x)$  can be computed as follows:

- if  $x$  is on a facet of polyhedron  $\mathcal{P}_i$ , then  $\mathbb{I}_i(x) = s$  (surface bit),
- if the cell indicator is known  $\mathbb{I}_i(c) \neq u$ , all points inside inherit the indicator bit  $\mathbb{I}_i(x) = \mathbb{I}_i(c)$ ,
- otherwise  $\mathbb{I}_i(v) = u$ , we consider the non-empty set  $P$  of polygons of  $\mathcal{P}_i$  in the node. We shoot a ray from  $x$  to an arbitrary point of one of the polygons to ensure at least one intersection with  $\mathcal{P}_i$ . Then we compute the intersection of this ray with all polygons of  $P$  and keep the intersection nearest to  $x$ . The sign of the dot product of the ray’s direction with the normal at the closest point gives the indicator bit  $\mathbb{I}_i(x)$ .

### 6.4.1 Jittering

To avoid feeding the algorithm degenerate configurations, two workarounds are implemented:

- for CAD meshes, it is useful to randomly translate each mesh independently by a random vector. The vector should be small enough not to change the topology of the output, but still the same order

of magnitude as the input mesh size. The random translation is reverted, ie. the true intersection vertices are recomputed from the faces they are the intersection of.

- the KD-tree is axis-aligned, so axis-aligned facets may also produce degeneracies. A simple workaround is to apply a random rotation to the input and revert this rotation at the end.

No explicit check is done that the random jitter does not introduce new degeneracy: maybe the random rotation aligns another facet with the bounding boxes? This is because the probability computation above shows that such coincidence is almost impossible.

## 7 Parallel Implementation

Performance being a main concern, parallelization is mandatory to benefit from today’s mainstream multi-core computer architectures. Parallelization has been studied in the context of boolean solid modeling, with elementary operations queued and balanced among processors Krishnan et al. (2001), by concurrently computing the result of non-dependent node operations in the CSG tree, or by using the GPU-friendly Layer Depth Image as CSG approximation Wang (2011). KD-tree algorithms have also been parallelized in the context of ray-tracing e.g. Choi et al. (2010) and Shevtsov et al. (2007). The biggest common issue algorithms face is the irregularity of the tree exploration, because work associated to each node of the KD-tree is data-dependent and difficult to predict. A good workload balancing strategy is necessary for optimal resource and core usage. Instead of crafting very specific code, we use the work-stealing paradigm for tree exploration, for which off-the-shelf scheduling algorithms exist.

### 7.1 Work Stealing Principle

Work stealing frameworks allow to express the potential parallelism by delimiting dynamically created tasks that can be executed concurrently. Each processing core maintains a list of tasks. When a core generates a task, it pushes it in its local list. A task of this list is ready for execution once synchronization constraints have been resolved. When a core becomes idle (*i.e.* no local task left), it randomly selects another core and steals part of the tasks ready to be executed in the task list of its target. If no task can be stolen, an other victim is targeted. This scheduling algorithm has proven performance Blumofe and Leiserson (1999). Today, several parallel programming environments are based on work stealing (Cilk, TBB, OpenMP, KAAPI). They come with high level constructions easing the parallelization of common patterns (loop with independent iterations for instance). Their implementations ensures high performance on multi-core processors and shared memory machines. We parallelized QuickCSG with Thread Building Blocks (TBB).

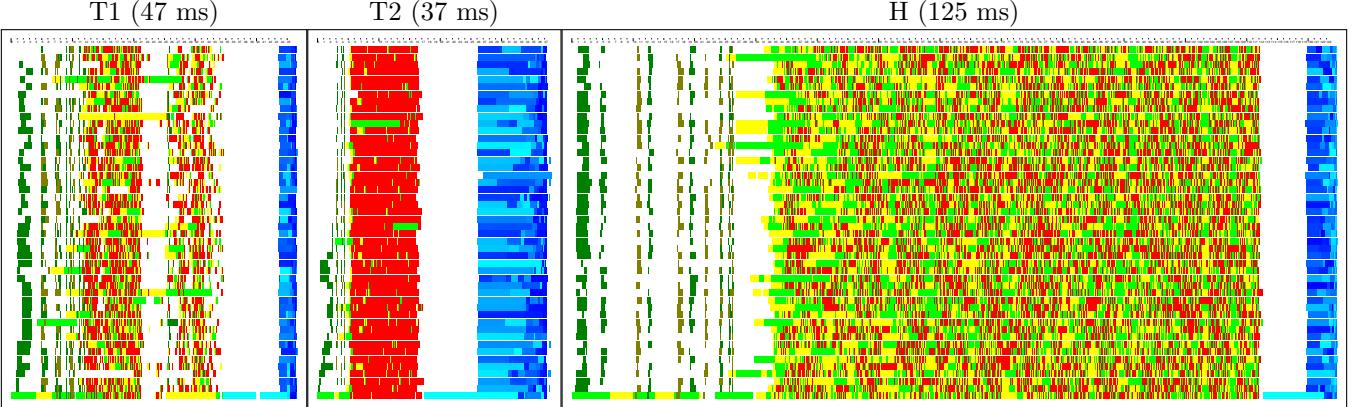


Figure 12: Gantt chart of QuickCSG running on a 48-core AMD machine on 3 different datasets (left:T1,central:T2,right: H). Datasets are detailed in section 8.2. Each line corresponds to a core activity (white: idle). Node splitting is performed in the green and yellow bars, while internal node parallelization is represented by dark green bars. Red bars are leaf computations (calls to CSGVERTICES). A light blue sequential step concatenates results from the different cores before building facets in parallel (calls to CSGFACETS, shades of blue). Notice that total execution times (indicated on top) are degraded due to code instrumentation to gather these statistics.

## 7.2 Proposed Implementation

The recursive nature of the KD-tree construction fits the task model well. We encapsulate the KDVERTICES node processing function in a task. These tasks can be executed concurrently and work stealing ensures they are dynamically spread amongst enrolled cores.

The KD-tree exploration starts with a single task. Enough tasks become available to keep all cores busy only once a certain depth is reached. Meanwhile, many cores will stall. To circumvent this bottleneck, SPLIT calls in the toplevel nodes are parallelized internally, with a simple parallel for, with results accumulated in separate vectors for each thread. Since this is less efficient than the node-level parallelization, this internal parallelization is enabled only in the very upper levels of the tree. Creating a task comes with some overhead, that can become significant for nodes with a light compute load. This is the case for deep nodes where the number of tasks is much higher than the number of enrolled cores. Thus to shave off overheads, we turn to a sequential sub-tree exploration once the number of facets to process in a node is below a given threshold (80). The results, spread in thread-local data structures, are then sequentially concatenated in a global data structure before the call to the CSGFACETS function, which is easily parallelized with a parallel for.

Figure 12 shows a gantt chart for executions of QuickCSG on three different datasets. Though top node splitting is parallelized internally (dark green), it is not as efficient as the node level parallelization once enough nodes have been generated (yellow, green and red bars). The sequential concatenation of results (light blue bars) incurs a non-negligible cost at 48 cores. This step is very memory-intensive, drastically limiting the efficiency of any parallelization. The scene geometry greatly influences the execution. For instance the number of output facets is much higher than the input facets for T2 (central chart), which explains the relatively high cost of the

calls to CSGFACETS (blue bars).

## 8 Experiments

We implemented the QuickCSG algorithm in C++, using 64-bit floats for coordinates. We rely on TBB for thread-stealing implementation, with optimized thread local memory allocations. The parallel implementation reverts to sequential exploration when there are less than 80 faces in the node, and brute-force CSGVERTICES is called as soon as the number of faces drops below 20. We store indicator vectors for nodes and points as 64-bit word bitfields. We use the particularly robust and efficient GLU implementation for the output polygon tessellation.

Similar to Wang (2011); Feito et al. (2013), we observed that many CSG implementations (*e.g.* CGAL, 3DSMax ProBoolean and Carve CSG) crash, generate empty results or refuse to process meshes that they cannot handle completely. Non-robustness to near degeneracies, inability to cope with large and dense primitives, incompatibility of inputs with processing hypotheses are the typical causes. Violation of QuickCSG’s input hypotheses may also occur for certain input meshes, *e.g.* in Figure 13 the input polyhedra are self-intersecting, which affects ISINSIDE, leading to a hole in the bounding box of an incorrectly cancelled node. Regarding numeric robustness, we resort to  $\epsilon$ -coordinate jittering, which, combined to the fact that only output primitives are computed, are observed to vastly reduce and in most cases eliminate the occurrence of degeneracy problems. In both cases QuickCSG fails gracefully: it records the occurrence of errors while outputting the constructed result, even if partially incomplete, as in Figure 13. In the experiments presented, we state when errors have occurred. On QuickCSG’s web page and the supplementary material, we provide the data and command lines that reproduce the experiments on QuickCSG.

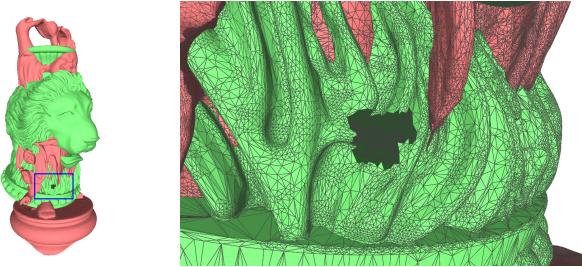


Figure 13: Failure case: when computing the union between two meshes, the algorithm makes a mistake on the mesh position of a KD-tree node because the input mesh is invalid (self-intersecting). Left: the result, right: close-up of the hole in the mesh (the example is “Buddha  $\cup$  Vase-Lion” from Section 8.1.2).

In this section, we compare QuickCSG to state-of-the-art CSG implementations on their own provided benchmarks. Because such benchmarks are typically limited to a few operations on medium to large size meshes, we introduce a new set benchmarks with more meshes and more complex CSG operations. We experimentally probe the main characteristics of QuickCSG: its complexity, parallel performance, and gains of  $n$ -ary versus binary operators.

Unless stated otherwise, we ran the experiments on a i5 CPU 750 at 2.7 GHz (4 cores) with 4 GB of RAM. Reported execution times encompasses the processing from input meshes to the output mesh including tesselization to convex polygons, but excluding startup time and disk I/O. The reported timings are wall-clock times in seconds, measured by the `gettimeofday` function. As timings for several runs are within 1% of each other, we do not report standard deviations.

## 8.1 Comparison With State of the Art

Our comparisons focus on recent boolean methods Wang (2011,?); Pavic et al. (2010); Feito et al. (2013) and one software package (Carve). To the best of our knowledge these implementations are currently the most computationally efficient ones. The related papers provide timing comparisons with other public (CGAL, GTS) Feito et al. (2013) or commercial packages (Rhino, ACIS Wang (2011), Houdini Pavic et al. (2010), and 3DS Max Feito et al. (2013)), which were consistently found to be significantly slower. We thus focus our comparisons on the former, with input data obtained either directly from the authors, or from the Stanford 3D scanning repository, resizing the meshes to the detail level reported in the papers if necessary. In most cases, the authors did not provide their implementation, so we directly compare QuickCSG to the published execution times, with some comments about the differences between the hardware. For these experiments, we use single-core sequential runs of QuickCSG, unless otherwise stated.

### 8.1.1 Carve CSG

Carve is the CSG library<sup>1</sup> used in the Blender modeler. It uses an octree accelerator structure and produces clean output meshes, with few useless vertices. We compare Carve and QuickCSG on the most time consuming benchmarks of the Carve CSG test suite (`test_intersect.cpp`). QuickCSG is 4 to 14 times faster than Carve on these examples handling large meshes or/and many meshes:

Example	$m$	Carve	QuickCSG
21: sphere – translated sphere	19600	0.342	0.077
29: union of 30 rotated cubes	180	0.495	0.077
30: sphere – sphere $\cap$ cube	19606	0.469	0.032
34: cow $\cup$ translated cow	185728	3.601	0.313

### 8.1.2 MeshWorks

MeshWorks<sup>2</sup> is a GPU implementation of an approximate Layered Depth Image algorithm Wang (2011). Because the intersection of surfaces is only resolved up to the resolution of the depth layer images used, several resolutions are tested in their paper. The range of timings in the following table reflect the range from coarse to fine resolution, with the finer scale taking the most time. It is the most relevant to our tests since QuickCSG does not perform any approximation and retrieves the exact topology. QuickCSG, executed on 4 cores, is from 10 to 40 times faster than MeshWorks executed on a nVidia GTX 260 GPU (+ 4 CPU cores), on the examples provided along with the software:

Example	$m$	Wang (2011)	QuickCSG
Dragon $\cup$ Bunny	941k	55.4	3.4
Small dragon – Bunny	347k	3.06 – 8.97	0.253
Buddha $\cup$ Vase-Lion	1.48M	10.68 – 21.81	1.027

We compare both with the provided MeshWorks implementation (first example) and timings from the original paper (two last examples). Notice that the input meshes have small self-intersections which violates the input assumptions, so the QuickCSG output contains holes, see Figure 13.

MeshWorks is optimized for large and detailed meshes. In this case, most faces do not intersect another face. Once identified, these faces can be directly copied to the output. MeshWorks and QuickCSG support this, but it seems that the up- and down-load to the GPU hurts MeshWork’s performance.

### 8.1.3 Hybrid Booleans

The “Hybrid Booleans” method of Pavic et al. (2010) subdivides the input space with an octree, then constructs an approximate output mesh by remeshing the surface at the resolution of leaf nodes. We run QuickCSG on the paper’s test data, and compare it against the reported timings, as the implementation is not available.

<sup>1</sup>Carve 1.4 can be found at <http://code.google.com/p/carve/downloads/list>.

<sup>2</sup>Executable at <http://www2.mae.cuhk.edu.hk/~cwang/projMeshWorks.html>

Depending on the quality settings for Pavic et al. (2010), QuickCSG is 5 to more than 100 times faster. Again the higher times are more relevant to the comparison as QuickCSG retrieves exact topology on all these examples:

Example	$m$	Pavic et al. (2010)	QuickCSG
Chair	1.5k	1.3 – 13	0.003
Sprocket	11k	5 – 47	0.069
Organic	219k	1.6 – 24 (+1)	0.488

The Hybrid Booleans method requires to explore the tree to a predefined depth even for the simplest of input meshes (the "Chair" example), hampering performance. It also generates a large amount of over-tesselated polygons. The "Organic" example is based on a CSG operation with six solids, of the form  $(\mathcal{P}_1 \setminus \mathcal{P}_2) \cup (\mathcal{P}_3 \setminus \mathcal{P}_4) \cup (\mathcal{P}_5 \setminus \mathcal{P}_6)$ . QuickCSG directly computes the result, while it is computed with intermediate meshes in Pavic et al. (2010), hence the additional second for the intermediate mesh computation.

#### 8.1.4 Feito et al

The algorithm from Feito et al. (2013) implements two-component boolean expressions on triangular meshes, which they resolve by exploring an octree with parallel threads. The authors did not provide their implementation, and thus we compare QuickCSG with the times published in the original paper. Compared to our test machine, they use a higher-end processor (Xeon X5550 2.7 GHz), with more cores and memory (12 GB). The evaluation of the original paper relies on combining standard meshes (dragon, armadillo) with a translated version of the same mesh, and averaging the timings results over four CSG operations (union, intersection, and the two possible differences). QuickCSG is 4 to 5 times faster for the same core count: ( $cr = \# \text{ cores}$ ):

Example	$m$	Feito et al. (2013)			QuickCSG	
		1 cr	4 cr	16 cr	1 cr	4 cr
Armadillo	2 × 150k	2.71	1.46	0.68	0.57	0.24
Dragon	2 × 871k	12.64	6.48	2.72	2.61	1.18

Possible reasons for this performance gap could come from the intermediate geometry generated (over-tesselized triangles and vertices that must be merged in a later stage), and an octree fully stored in memory as it is required for ray shooting, unlike our approach which doesn't require storing the tree.

## 8.2 Performance & Comparisons on Huge Datasets

Importantly, all previous tests were performed on the datasets of the original papers, which only process a few, sparse-intersecting inputs, and a small number of boolean operations, *i.e.* in the "comfort zone" of existing algorithms. The algorithm is already shown to significantly outperform them in this favorable situation. To illustrate the even larger gain in the more general situations our algorithm can tackle, we introduce new test

cases involving large meshes, very dense intersection areas, with several dozen operations:

- **T1** is a set of 50 random toruses. We compute the difference between the union of the 25 first toruses with the union of the 25 next ones:  $\mathcal{P}_f = (A_1 \cup \dots \cup A_{25}) \setminus (A_{26} \cup \dots \cup A_{50})$ . This is a typical CSG case, where many facets intersect, but the geometry is regular (small compact facets).
- **T2** is a set of 50 concentric narrow random toruses. The toruses follow the great circles on a sphere, so each torus intersects each other torus in two locations. We compute the volumes where at least two of the toruses are present ( $f_{\min-2}$ ). In this case, most facets intersect another. This generates many disconnected components with many more facets than there are on input.
- **H** is a set of 42 cones with arbitrary bases corresponding to the silhouettes of a piece of rope seen from 42 cameras. The silhouettes define cones whose apexes are the optical centre of the cameras, and that pass through the silhouette's shape on the camera's image planes. An approximation of the piece of rope can be reconstructed by intersecting ( $f_n$ ) the cones Franco and Boyer (2009). The facets are very elongated, and the output mesh has no order-1 vertices.

Table 14 gives some statistics about these three test cases and QuickCSG's performance. The performance timings are broken down in three stages: the "topology" concerns a necessary preprocessing pass of our algorithm over the data for facet normals and adjacencies, and the other timings report the execution time of KDVERTICES and CSGFACETS. The behavior can be different depending on the mesh. For T1 and H, the slowest stage is CSGVERTICES, for T2 it is CSGFACETS, because it generates many facets.

## 8.3 Comparisons

Most existing approaches do not provide their implementation or simply fail in this case. For example, 3DSMax ProBoolean produced an incorrect result for T1, after 12 s of computation. Therefore, we compare results of T1, T2 and H, against one of the fastest and most robust method available, the Carve library. The operations on T1 and H were expressed as trees of binary operations and the operation on T2 as the union of all intersections of 2 meshes to enable Carve to process them. This last operation has  $\binom{n}{2}$  components:

$$\begin{aligned}
 f_{\min-2}(a_1, \dots, a_n) = & \\
 f_{\cup}(f_n(a_1, a_2), f_n(a_1, a_3), \dots, f_n(a_1, a_n)), & \\
 f_n(a_2, a_3), \dots, f_n(a_2, a_n), & \\
 \dots & f_n(a_{n-1}, a_n)
 \end{aligned} \tag{12}$$

Sequential execution times are found to be:

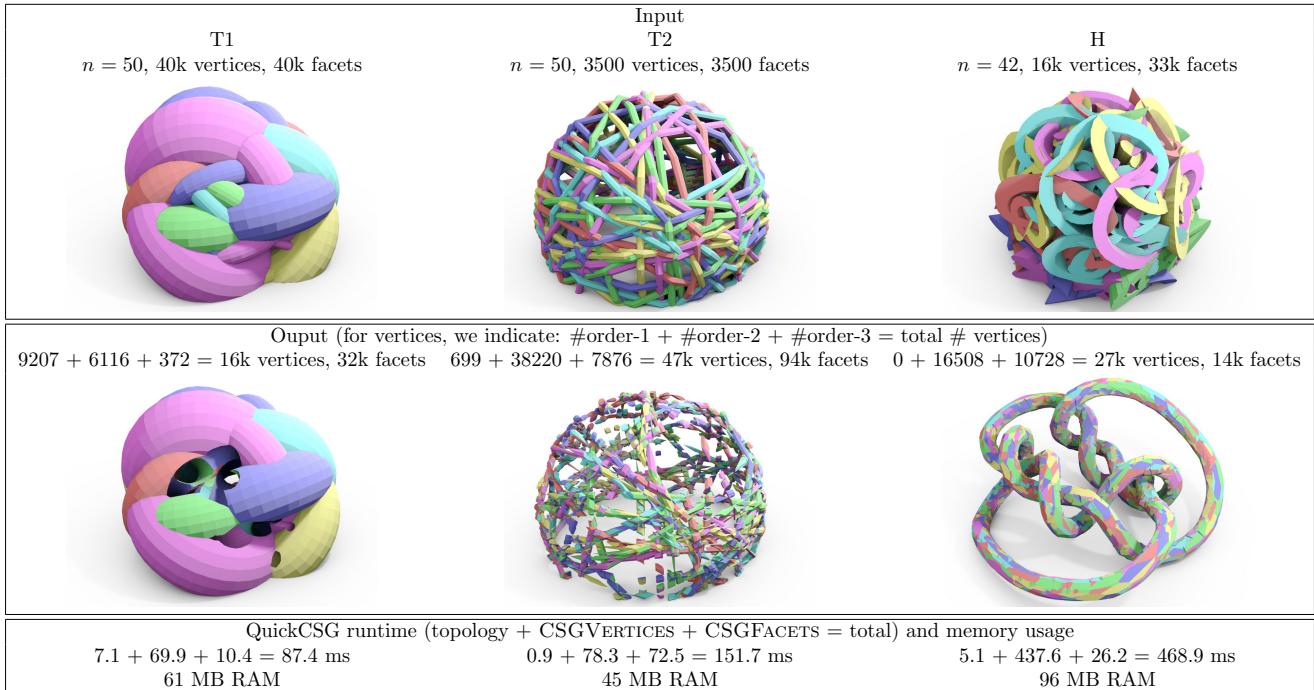


Figure 14: The **T1**, **T2** and **H** test cases and QuickCSG performance. RAM is measured as the maximum resident set size reported by the unix `time` utility.

Example	$m$	Carve CSG	QuickCSG
T1	40000	4.652	0.297
T2	3500	(94.795)	0.596
H	33108	26.330	1.720

Times are given in parenthesis when Carve was only able to provide a partial result. For T1 and H, both QuickCSG and Carve provide the expected results, QuickCSG being about 15 times faster than Carve. For T2, despite its careful handling of degenerate cases, Carve was not able to compute  $f_{\min-2}$  on more than 38 input meshes. Indeed, computing the unions of the intermediate intersections generates many degeneracies because Carve relies on several binary operations of meshes that use exactly the same vertices but with different incidence, which fails for non-exact methods such as Carve.

Additionally, we evaluated how the speedup evolves as a function of size of input meshes, by generating increasingly subdivided versions of the T1 dataset. We report timings in Figure 15 against runs of Carve on the same datasets. Carve fails over one million input polygons in this example. Speedups for QuickCSG: with 100k polygons, mono-core QuickCSG is 30 $\times$  faster than Carve, and 4-core executions are 70 $\times$  faster. With one million input facets, this speedup reaches 60 $\times$  mono-core and 150 $\times$  with 4 cores.

## 8.4 Probing the Characteristics of QuickCSG

We validate some properties of the algorithm to support the claims of the previous sections: namely, it is efficient to directly compute the result of CSG operations on multiple polyhedra, the complexity of the CSG operations

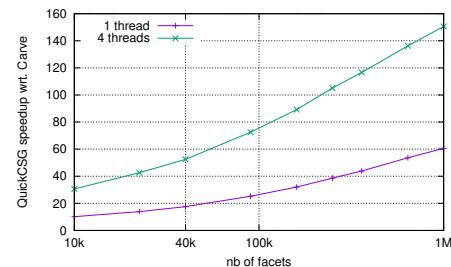


Figure 15: Speed comparison between Carve and QuickCSG for meshes of increasing complexity based on T1 (which corresponds to the point at 40k facets).

is  $\mathcal{O}(m \log h)$ , and the parallelization is efficient.

#### 8.4.1 Comparison with binary CSG operations

We evaluate the performance improvement that QuickCSG can provide by directly computing the final output without intermediate polyhedra, compared to the classical approach that combines polyhedra by pairs. Indeed, any associative boolean operation  $f$  over  $n$  input polyhedra can be expressed as a sequence or binary tree of  $n - 1$  binary operations:

$$\begin{aligned} f(a_1, \dots, a_n) &= f(a_1, f(a_2, \dots, f(a_{n-1}, a_n) \dots)) \\ &= f(\dots f(a_1, a_2) \dots, \dots f(a_{n-1}, a_n) \dots) \end{aligned} \tag{13}$$

Such decompositions produce  $n - 2$  intermediate polyhedra.

Table 1 compares the execution for different combinations. The CSG operations on T1 and H can be expressed with binary operations applied sequentially or

Table 1: QuickCSG execution times when expressing the same CSG operation differently. Tests performed on the T1 and H test cases with 4 threads. Execution times slightly differ from Figure 14 because they are obtained via the Python interface of QuickCSG

T1			H		
ordering	time	errors	ordering	time	errors
single	0.125	0	single	0.524	0
binary tree	0.363	0	binary tree	2.396	9147
sequential	0.619	0	sequential	2.244	50258
5,5,2	0.180	0	8,6	0.674	0
25,2	0.111	0	4,11	0.872	0

through a binary tree. Directly evaluating the equivalent  $n$ -ary operation to compute the output polyhedron clearly outperforms binary operations that produce many facets of intermediate polyhedra that are discarded later on. Producing intermediate polyhedra also increases the probability of errors caused by degeneracies.

We investigate alternate trees of operations by varying the arities at different levels. In Table 1, for example “25,2” in T1 means that we first compute the union of 25 polyhedra and next combine the resulting two polyhedra to get the final result. The results shows that it is generally faster to perform the CSG operation in one pass, except for T1, where the “25,2” ordering gives the best results. In this case the reason could be that operating on fewer meshes at a time means that fewer vertex/cell indicator bits need to be computed for unrelated components.

#### 8.4.2 Evaluation of complexity

We experimentaly probe the expected asymptotical complexity of QuickCSG,  $\mathcal{O}(m \log h)$ , by timing a large number of random operations. We randomly sample 1000 groups of 5 meshes from the “3D Mesh Segmentation Benchmark” collection<sup>3</sup>, consisting of 379 meshes, from 2600 up to 55k facets each. This collection originally comes from the watertight track of the 2007 shape retrieval contest Giorgi et al. (2007). The CSG operation applied for each group is randomly chosen between union, intersection or union of the 3 first minus union of the 2 last (labeled diff3 in the figure).

As in most cases involving complex meshes, the dominant step of the algorithm is the KDVERTICES stage, which our experimental analysis thus focuses on. Even though the total number of split polygons is found to be  $s = \mathcal{O}(m)$ , as shown in Figure 16 (left), in practice complexity plots are significantly clarified by explicitly accounting for the influence of the number  $s$  of splits occurring during the entire exploration, which is volatile from dataset to dataset. Plotting the execution time versus  $(m + s) \log h$  on the right of Figure 16, shows the proportionality relation between both. The fact that the proportionality is verified over all input sizes and heterogeneous boolean operations brings a strong validation to

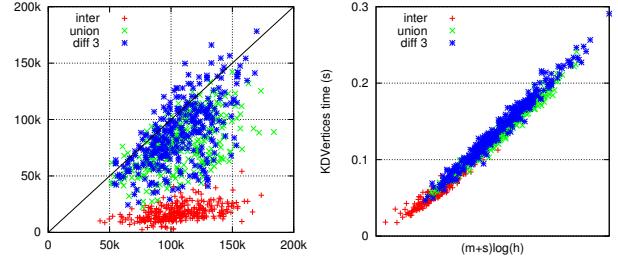


Figure 16: Test on 1000 CSG operations on 5 random meshes. Left: Relationship between the number of split polygons ( $s$ , y-axis) and the number of input facets ( $m$ , x-axis). Right: the runtime for the main computation as a function of  $(m + s) \log h$ .

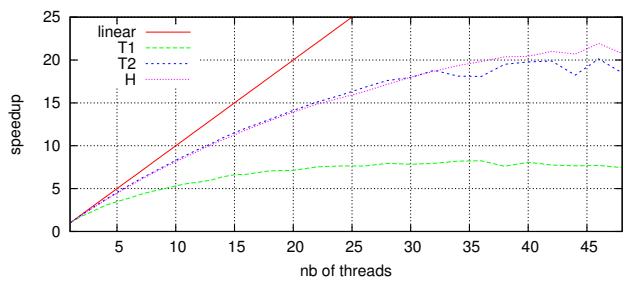


Figure 17: Speedup obtained with more threads, with respect to a sequential run. This is measured on a 48-core AMD Opteron(tm) Processor 6174.

our analysis of the algorithm complexity in Section ??.

#### 8.4.3 Parallel Execution

We evaluate QuickCSG’s parallelization on a machine with four 12-core AMD processors. Figure 17 plots the speedups for the T1, T2 and H tests. The Gantt charts in Figure 12 give a detailed insight about the parallelization behavior at 48 cores. Despite the irregular nature of the computations, parallelization is more than 80% efficient up to 8 cores for T1 and 20 cores for T2 and H. At large core count, the performance is impacted by the exploration of the top KD-tree levels and the data gathering before building facets. The parallelization efficiency depends on the complexity of the input and output meshes. For instance the T1 output consists of a large majority of first order vertices, with few order-2 and order-3 vertices, so there are too few CSGVERTICES tasks in the KD-tree for the distribution over so many cores to be efficient, see Figure 12. The parallel performance improves for large meshes with complex outputs.

## 9 Applications

In this section we demonstrate several use cases where the performance of QuickCSG opens new possibilities for usage of boolean combinations of polyhedra. We first examine the computer vision problem of 3D modeling from a set of silhouettes, then solid modeling in the context of 3D printing, and collision detection in interactive

<sup>3</sup><http://segeval.cs.princeton.edu/>

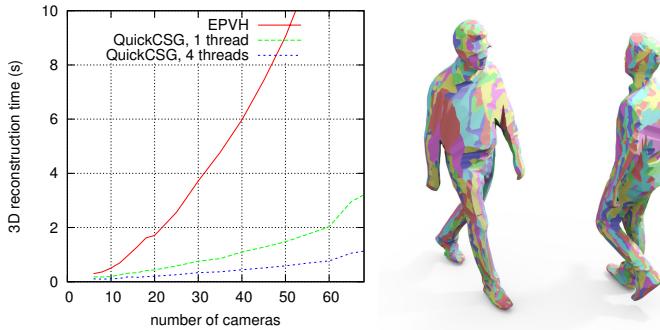


Figure 18: (Left) QuickCSG vs. EPVH runtime. (Right) Resulting visual hull when computed from 68 cameras (22k facets).

systems. We conclude the section on extreme uses of boolean operations, in real-time or over million-polygon inputs.

## 9.1 3D Modeling

Given  $n$  real photographic frames of an object acquired from different camera viewpoints, it is possible to build 3D reconstructions of the object by extracting its silhouettes in the obtained images, and building the *visual hull* of the object. The visual hull is the maximal 3D volume that projects onto the input silhouettes. Previous works have shown that it can be built by intersecting a set of polyhedral viewing cones Baumgart (1974), *i.e.* cones whose apex is the optical center of the cameras, and whose basis is the silhouette itself. Because the quality of the model increases with the number of input views, new dedicated multi-camera acquisition platforms are being crafted with several dozen cameras, as is the case for example of the Kinovis 68-camera studio<sup>4</sup>. These platforms provide huge amounts of complex data, challenging even the fastest existing implementations. In this context, we compare our algorithm with one of the fastest state-of-the-art methods specialized in this task, EPVH Franco and Boyer (2009). We use a 68-camera dataset produced on the Kinovis platform, plotting the execution time of both methods against the number of input cones, see Figure 18. QuickCSG significantly outperforms the dedicated method whatever the number of input views considered, reaching a ten to twenty-fold speed increase above 30 cameras with four threads.

## 9.2 Solid Modeling

With the increasing popularity of 3D printing, there is a demand for fast and convenient modeling tools that enable any user to sketch and put together his own 3D objects for printing. OpenSCAD<sup>5</sup> is a popular software and scene description language used in this context. Solids are modeled through a tree of binary CSG operations from primitive shapes (cylinder, sphere, extruded

2D shapes, etc.). OpenSCAD relies on CGAL Hachenberger et al. (2007) to compute the resulting polygonal mesh. We compare the performance of QuickCSG versus OpenSCAD when both process the same binary tree of operations. Tested on two complex models, Balljoint and Doggie from IceSL Lefebvre (2013), OpenSCAD’s mesh computation requires 16 minutes and 7 minutes respectively while QuickCSG’s only needs 1.46 s and 0.3 s. We printed the Balljoint mesh generated by QuickCSG using Makerware on a Makerbot Replicator 2 (Figure 19). All the balljoints are functional.

It is also possible to compute the whole result at once, by using a boolean function  $f$  that evaluates the binary CSG tree. This approach does not perform as well: 3.8 s for Balljoint. To evaluate the trade-off between binary and all-at-once computation, we traverse the OpenSCAD CSG tree, returning an intermediate sub-tree for each node. For a given node, we collect the result subtrees of its two child nodes. If the total number of meshes in these sub-trees is above some threshold  $G$  (the grouping factor), we call QuickCSG to compute the CSG operation and return a 1-node sub-tree with the CSG result. Otherwise, we return a sub-tree built with the two child mesh results and the binary operation. The two baselines, binary evaluation and all-at-once evaluation are obtained for  $G = 2$  and  $G = \infty$  respectively.

Figure 19 plots the execution time as a function of the grouping factor  $G$ . The optimum occurs at  $G = 8$ , with a 30% gain compared to the binary evaluation: 0.96 s for Balljoint and 0.2 s for Doggie (or 0.52 s and 0.12 s respectively with 4 threads). The binary tree built by OpenSCAD already conveys some spatial clustering. Two intersecting primitive shapes are very likely close in the binary tree. By grouping operations according to their location in the binary tree, we indirectly benefit from a better space partitioning than the one the KD-tree performs when considering all operations at once.

QuickCSG is 500 to 1000 faster than CGAL. Because of the perfect CAD inputs, which exhibit more regularity than acquired datasets, the mesh produced by QuickCSG occasionally suffers from localized degeneracies, even with jittering, unlike CGAL that relies on exact methods. However, when dealing with higher polygon counts, CGAL computation times quickly becomes impractical. The huge performance gap offered by our algorithm leaves room for building an exact version of the method with still significantly faster runtimes.

## 9.3 Collision Detection

Many interactive systems rely on virtual objects simulations that necessitate inter-object collision detection. A vast array of dedicated methods have been designed just for this problem, often based on hierarchical structures that reduce the otherwise quadratic object interpenetration tests Weller (2013). Interestingly however, the existence of generic high-performance CSG tools can provide a new basis to reformulate the problem. Given a set of solids, the set of object interpenetrations subvolumes can be obtained by computing the min-2 operation over all objects in a given scene. Each connected com-

<sup>4</sup><http://kinovis.inrialpes.fr>

<sup>5</sup><http://www.openscad.org/documentation.html>

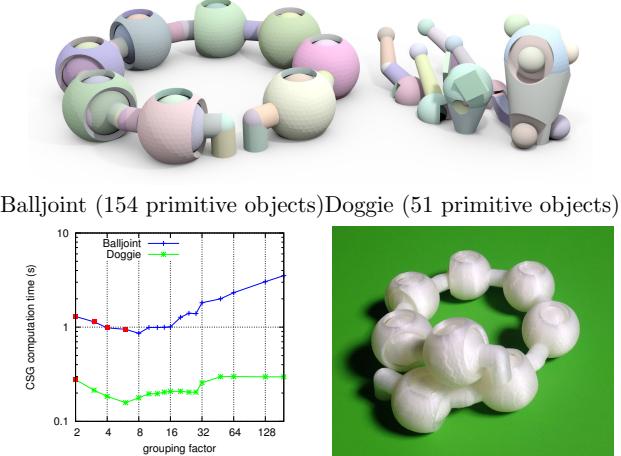


Figure 19: Top: The Balljoint and Doggie models. Down left: QuickCSG runtime (single thread) with different grouping strategies (red squares indicate that there were degeneracy errors). Down right: printed Balljoint. All balljoints are functional.

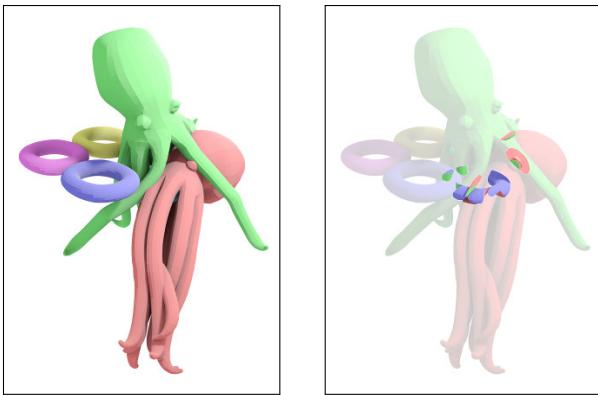


Figure 20: QuickCSG computes min-2 on two flabby octopuses and four rings. (Left) 33k input faces in total. (Right) Interpenetration volumes.

ponent of the output is the interpenetration volume of at least two solids (this is approach is as yet incomplete, since it does not handle self-intersections and interpenetrations of more than 3 solids). We ran QuickCSG on an example of the SOFA physics engine<sup>6</sup>, see Figure 20. QuickCSG on 4 cores takes 15.5 ms (excluding the 6 ms topology pass, which can be run once at the beginning of the animation), while the state-of-the-art LDI method from Allard et al. (2010) computes collisions and interpenetration volumes in 5 ms on a GPU (Quadro 4000). Thus, our generic CPU-only implementation is only a 3-4 factor away of a specialized approximate algorithm running on dedicated hardware.

#### 9.4 Extreme CSG

The speed of QuickCSG makes it well suited for interactive applications. We developed a small Python OpenGL application that animates a set of moving or deform-

ing polyhedra, and combines them with a user-specified boolean function. Performing the min-2 or union of three sets of quasi-parallel boxes (see Figure 21), runs at 30 fps for  $3 \times 10$  boxes, while the output has a large number of triangles ( $10 \times 10 \times 10$  grid of 3D crosses for min-2).

To give an idea of the broader applicability of the method on million-polygon datasets, we test QuickCSG on huge and intrinsically dense and complex examples, see Figure 22.

The Dithering test mixes two dragon meshes  $\mathcal{P}_1$  and  $\mathcal{P}_2$  with a 3D dithering pattern. The pattern is defined as the union of three orthogonal combs:  $D = \mathcal{P}_3 \cup \mathcal{P}_4 \cup \mathcal{P}_5$ . Then the dragon meshes are combined using the pattern as a mask:  $\mathcal{P}_f = (\mathcal{P}_1 \cap D) \cup (\mathcal{P}_2 \setminus D)$ . This is a function that would generate degeneracies by definition, if evaluated as a CSG tree. Indeed, in the intersection volume  $\mathcal{P}_1 \cap \mathcal{P}_2$  it computes  $D \cup \neg D$ . The result is computed in 2.5 s on our test machine (input: 1.74M facets, output: 1.69M).

The Serpent dataset is another such example, built as a fractal, where a tube,  $\mathcal{P}_1$ , is wound around a torus. Then another tube,  $\mathcal{P}_2$ , winds around  $\mathcal{P}_1$  and so on until  $\mathcal{P}_5$ . We compute the min-2 operation. From 31M input facets, QuickCSG outputs a 10M triangle mesh with a topological genus Agoston (2005) of 701, i.e., it can be transformed without tearing into a sphere with this many handles. This dataset overwhelms the memory of our standard test machine, so we computed it on a 12-core Mac Pro machine with 64 GB of RAM in only 15 seconds.

The last example is built from six instances of the Happy Buddha mesh Curless and Levoy (1996), the largest mesh from the Stanford repository. We intersect these with the union of 100,000 random spheres. The spheres were labeled with a greedy graph coloring algorithm to group them into 37 disjoint subsets, so there are a total of 43 input meshes and 24M triangles. The CSG operation computes the union of the 6 Buddhas and intersects this with the union of all spheres:  $\mathcal{P}_f = (\mathcal{P}_1 \cup \dots \cup \mathcal{P}_6) \cap (\mathcal{P}_7 \cup \dots \cup \mathcal{P}_{43})$ . On the Mac Pro this last example runs in 8 s and generates 5M triangles. It is shown on Figure 1.

## 10 Discussion & Conclusion

We have presented a new output-sensitive approach to boolean solid modeling, which generalizes previous known methods to the N-polyhedron case with arbitrary expressions, by directly computing the result with a simplified, vertex-centric algorithm. Thanks to its straightforward divide-and-conquer and pruning scheme, the algorithm achieves a performance breakthrough on datasets of all sizes. This has been extensively verified experimentally against a vast array of state-of-the-art approaches. The speedup not only materializes on typical cases previous algorithms would be used in, it proves to be groundbreaking on extremely dense and large datasets with up to tens of million polygons. The performance speedup in these cases reaches up to three orders of magnitude with respect to commonly available

<sup>6</sup><http://www.sofa-framework.org>.

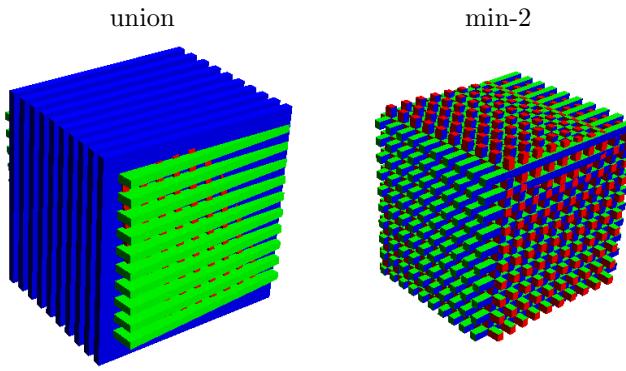


Figure 21: Union and min-2 running at 30fps on  $3 \times 10$  undulating boxes, generating 17k and 27k facets respectively. The number of output facets  $h$  grows as  $h \sim m^3$  with the number of input facets  $m$ .

approaches, when the latter do not fail due to the inability to deal with the large data.

The efficiency and expressiveness of the approach opens new possibilities as it also enables computation of results for arbitrary expressions, some of which simply cannot be processed with existing approaches. For this reason we believe many use cases of boolean modeling for research problems, initially ruled out for feasibility and performance reasons, now become accessible. We have shown some possible applications of the algorithm in the context of solid modeling for 3D printing, computer vision, interactive systems. We make the program, datasets, experimental protocol, and additional results available to the research community in the supplementary material and on the following page: <http://kinovis.inrialpes.fr/static/QuickCSG>.

Many future developments and ramifications of this method are possible. First, the demonstration here was done with polyhedral solids, but the method could be extended to other B-Rep representations, such as parametric surfaces, as the topological analysis shown in the paper is identical with curved facets, edges, and vertices at the intersection of curved edges and facets. Second, although the emphasis here is on performance with fast but non-robust predicates, one can imagine deriving an exact version of the algorithm relying on the Simulation of Simplicity paradigm Edelsbrunner and Mücke (1990), as the algorithm relies on a small number of well identified core geometric constructs and predicates (rayshooting in isFINAL, trihedron orientation test in CSGVERTICES, intersection routines INTERSECT2FACETS, and INTERSECTSEGMENTFACET, vertex ordering in FIRST, axis-aligned plane splitting in SPLIT). The inherent property of the algorithm to compute only final geometric primitives and no intermediate results will necessarily benefit this use case, as a substantial fraction of the speed penalty in converting to exact predicates would thus be avoided. Finally, the fact that the algorithm improves the known upper bound in complexity of boolean solid operations raises the broader theoretical question of its optimality for this problem, which we will investigate in future work.

## References

- ADAMS, B. AND DUTRÉ, P. 2003. Interactive boolean operations on surfel-bounded solids. *ACM Trans. Graph.* 22, 3 (July), 651–656.
- AGOSTON, M. K. 2005. *Computer Graphics and Geometric Modeling*. Springer, London.
- ALLARD, J., FAURE, F., COURTECUISSE, H., FALIPOU, F., DURIEZ, C., AND KRY, P. 2010. Volume Contact Constraints at Arbitrary Resolution. *ACM Transactions on Graphics* 29, 3 (Aug.).
- BAUMGART, B. G. 1974. Geometric modeling for computer vision. Ph.D. thesis, Stanford, CA, USA. AAI7506806.
- BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (Sept.), 509–517.
- BERNSTEIN, G. AND FUSSELL, D. 2009. Fast, exact, linear booleans. *Computer Graphics Forum* 28, 5, 1269–1278.
- BLUMOFER, R. D. AND LEISERSON, C. E. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept.), 720–748.
- BRAID, I. C. 1975. The synthesis of solids bounded by many faces. *Commun. ACM* 18, 4 (Apr.), 209–216.
- BRUNET, P. AND NAVAZO, I. 1990. Solid representation and operation using extended octrees. *ACM Trans. Graph.* 9, 2 (Apr.), 170–197.
- CAMPEN, M. AND KOBBELT, L. 2010. Exact and robust (self) intersections for polygonal meshes. *Comput. Graph. Forum* 29, 2, 397–406.
- CARLBOM, I. 1987. An algorithm for geometric set operations using cellular subdivision techniques. *IEEE Computer Graphics and Applications* 7, 5, 44–55.
- CHOI, B., KOMURAVELLI, R., LU, V., SUNG, H., BOCCINO, R. L., ADVE, S. V., AND HART, J. C. 2010. Parallel sah k-d tree construction. In *High Performance Graphics*.
- CURLESS, B. AND LEVOY, M. 1996. A volumetric method for building complex models from range images. In *SIGGRAPH*.
- DE BERG, M. D., CHEONG, O., KREVELD, M. V., AND OVERMARS, M. 2008. *Computational Geometry: Algorithms and Applications*, 3rd ed. ed. Springer-Verlag TELOS, Santa Clara, CA, USA.
- EDELSBRUNNER, H. AND MÜCKE, E. P. 1990. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM TRANS. GRAPH.* 9, 1, 66–104.

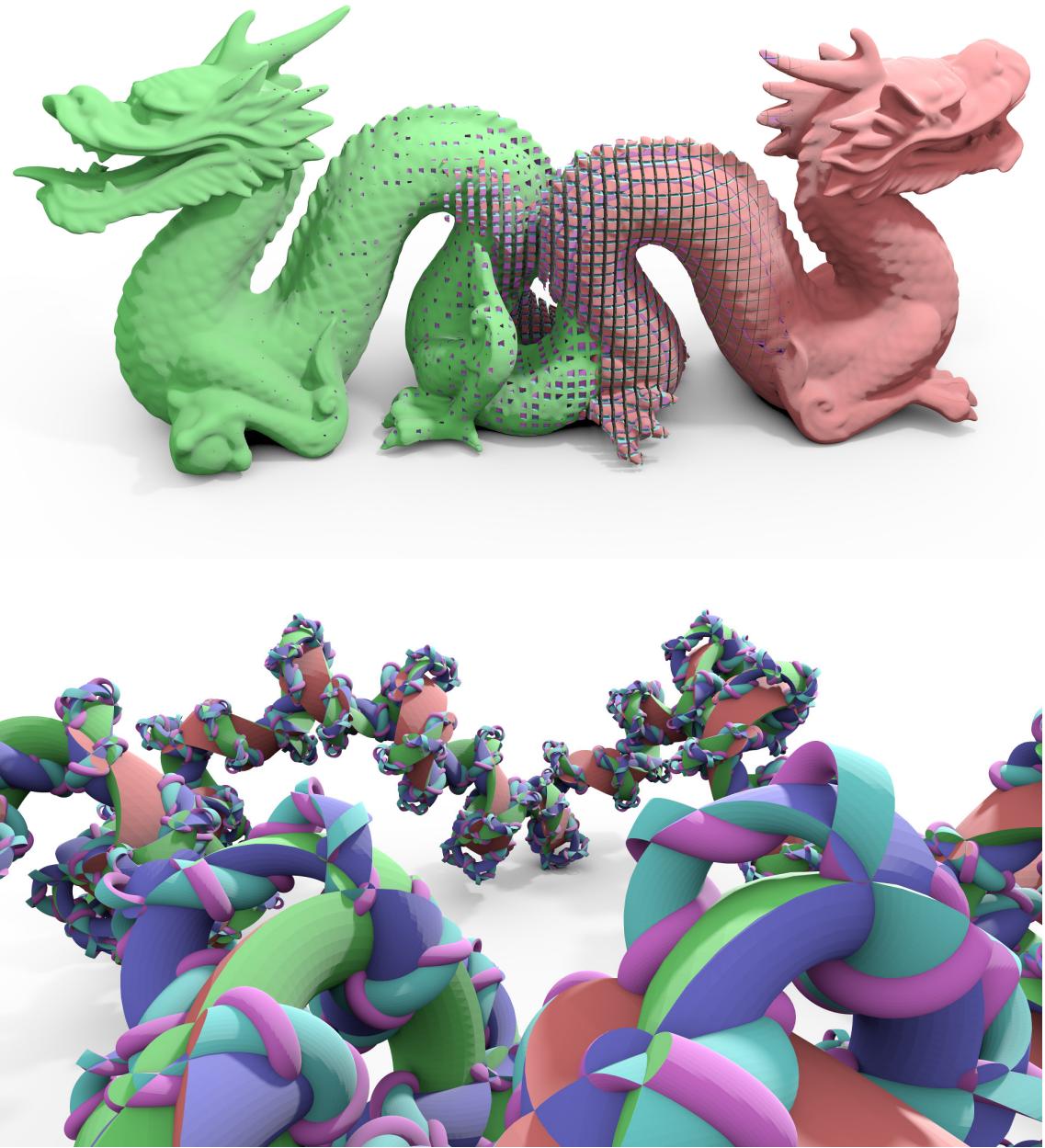


Figure 22: The Dithering (1.5M triangles) and Serpent (10M triangles) tests.

- FEITO, F., OGAYAR, C., SEGURA, R., AND RIVERO, M. 2013. Fast and accurate evaluation of regularized boolean operations on triangulated solids. *Computer-Aided Design* 45, 3, 705 – 716.
- FRANCO, J. AND BOYER, E. 2009. Efficient polyhedral modeling from silhouettes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31, 3 (March), 414–427.
- FRANCO, J.-S., PETIT, B., AND BOYER, E. 2013. 3D Shape Cropping. In *Vision, Modeling and Visualization*. Eurographics Association, Lugano, Switzerland, 65–72.
- GIORGI, D., BIASOTTI, S., AND PARABOSCHI, L. 2007. Shape retrieval contest 2007: Watertight models track. SHREC competition.
- HACHENBERGER, P., KETTNER, L., AND MEHLHORN, K. 2007. Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, Optimized Implementation and Experiments. *Comput. Geom. Theory Appl.* 38, 1-2 (Sept.), 64–99.
- HAVRAN, V. 2000. Heuristic ray shooting algorithms. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.
- HOFFMANN, C. 2001. Robustness in Geometric Computations. *JCISE* 1, 143–155.
- HOFFMANN, C. M. 1989. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- KRISHNAN, S., MANOCHA, D., GOPI, M., CULVER, T., AND KEYSER, J. 2001. Boole: A boundary evaluation system for boolean combinations of sculptured solids. *Int. J. Comput. Geometry Appl.* 11, 1, 105–144.
- LAIDLAW, D. H., TRUMBORE, W. B., AND HUGHES, J. F. 1986. Constructive solid geometry for polyhedral objects. In *Computer Graphics (Proceedings of SIGGRAPH 86)*. Vol. 20. 161–170.
- LEFEBVRE, S. 2013. IceSL : A GPU Accelerated modeler and slicer. In *18th European Forum on Additive Manufacturing*. <http://webloria.loria.fr/~slefebvr/icesl>.
- LI, C., PION, S., AND YAP, C. 2004. Recent progress in exact geometric computation. *J. of Logic and Algebraic Programming* 64, 1, 85–111. Special issue on “Practical Development of Exact Real Number Computation”.
- MACDONALD, J. D. AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 3, 153–166.
- MÄNTYLÄ, M. 1987. *An Introduction to Solid Modeling*. Computer Science Press, Inc., New York, NY, USA.
- NAYLOR, B., AMANATIDES, J., AND THIBAULT, W. 1990. Merging bsp trees yields polyhedral set operations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '90. ACM, New York, NY, USA, 115–124.
- NEF, W. 1978. *Beiträge zur Theorie der Polyeder: mit Anwendungen in der Computergraphik*. Beiträge zur Mathematik, Informatik und Nachrichtentechnik. Lang.
- PAVIC, D., CAMPEN, M., AND KOBELT, L. 2010. Hybrid booleans. *Computer Graphics Forum* 29.
- POPINET, S. 2006. GNU triangulated surface library.
- REQUICHA, A. A. G. 1977. Mathematical Models of Rigid Solid Objects. Tech. Rep. TR-28, Production Automation Project, University of Rochester. Nov.
- REQUICHA, A. A. G. AND VOELCKER, H. 1985. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE* 73, 1 (Jan), 30–44.
- REQUICHA, A. G. 1980. Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv.* 12, 4 (Dec.), 437–464.
- SARGEANT, T. 2011. Carve CSG boolean library, version 1.4.
- SCHNEIDER, P. AND EBERLY, D. 2003. *Geometric tools for computer graphics*. Morgan Kaufmann, San Francisco.
- SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. In *Computer Graphics Forum*. Vol. 26. Wiley Online Library, 395–404.
- THIBAULT, W. C. AND NAYLOR, B. F. 1987. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. ACM, New York, NY, USA, 153–162.
- WALD, I. AND HAVRAN, V. 2006. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *IEEE Symposium on interactive ray tracing*. 61–70.
- WANG, C. L. 2011. Approximate boolean operations on large polyhedral solids with partial mesh reconstruction. *IEEE Transactions on Visualization and Computer Graphics* 17, 6, 836–849.
- WELLER, R. 2013. A brief overview of collision detection. In *New Geometric Data Structures for Collision Detection and Haptics*. Springer Series on Touch and Haptic Systems. Springer International Publishing, 9–46.