Efficient and Robust N-Mesh Boolean Operations Using Hybrid Representations

Rui Wang, Bin Sheng Hongbo Fu, Ping Li, and Enhua Wu

Abstract—Constructive solid geometry (CSG) is a widely used modeling tool. It constructs complex models by combining primitive solids with boolean operations. However, boolean evaluation, which computes the final model of CSG, has suffered from robustness problem for more than three decades. Some methods are fast but not robust, while some methods are robust but rely on exact arithmetic which is slow. Recent works tried to implement fast and robust boolean evaluation methods using plane-based representations (P-reps) of polyhedrons instead of vertex-based representations (V-reps) of polyhedrons. While they have achieved remarkable improvements, they are still slow compared with non-robust methods. We propose a fast and robust boolean method. It uses hybrid representations of polyhedrons, which combines P-reps with V-reps. The P-rep information is used for exact predicates computation and the V-rep information is used for coarse tests and fast topology query, which allows us to take advantages of both the efficiency of V-reps and the exactness of P-reps. Comparison experiments with the state-of-the-art show that our method is unconditionally robust for solid inputs, and is similarly efficient to existing non-robust methods.

Index Terms—boolean operations, CSG evaluation, plane-based geometry

1 Introduction

Constructive solid geometry (CSG) has long been a popular modeling tool for computer-aided design and computer-aided manufacturing (CAD / CAM). Complex models are constructed by combining primitives using regularized boolean operations [1], [2]: union, intersection, and difference. A CSG can be converted into the boundary representation (B-rep) through boolean evaluation. There are two major types of boolean evaluation methods, which vary according to how they deal with intersections between primitives. Approximate methods [3], [4], [5] rediscretize the intersection areas, fit vertices approximatively, and rearrange the topology. Conversely, exact methods [6], [7], [8] do not change vertex positions, and maintain as many input elements (such as faces, vertices, and topology) as possible. In many applications, exact methods are preferred for their accuracy. Additionally, the clear mapping between the surfaces of the input and output meshes in exact methods simplifies the inheritance of surface information, such as face colors and materials.

However, there is always a compromise between robustness and efficiency for exact boolean algorithms. Robust methods often require exact arithmetic [8], [9], which is significantly slower than normal floating-point arithmetic. Other methods only guarantee **quasi-robust** [10] by using unreliable techniques, such as epsilon-tweaking [11], [12], [13] and numerical perturbation [7]. Recently, robust plane-based methods have been developed [14], [15] based on binary space partition (BSP) merging algorithms [16], [17]. The robustness of these methods is guaranteed by the theorem of Sugihara and Iri [18] that boolean evaluations can be performed without **constructions** [10] if polyhedrons are represented based on planes instead of vertices. By only use **predicates**, boolean methods are easier to be robust. These plane-based methods are generally faster than others which using exact arithmetic. However, they still suffer from performance issues because of the high computation complexity of BSP algorithms. In addition, the incoherence between BSP representations and B-reps of polyhedrons requires that extra steps for conversion and connectivity reconstruction, leading to worse performance.

Inspired by these previous studies, we develop a robust boolean method, which is unconditionally robust given consistent solid inputs, and is comparable efficient as non-robust methods. In our method, we combine P-reps with V-reps, forming hybrid representations of solids. Generally, the V-reps information is used for coarse tests and efficient neighboring face queries, while P-reps information is used for exact predicates (see Fig. 1). To avoid using exact arithmetic and ensure efficiency, we avoid constructions throughout our method.

Our method is different from the previous BSP-based methods [14], [15]. While their methods are largely based on BSP theorem of [16], [17], our method is more like vertex-based methods such as [8], [12] and is significantly faster than these BSP-based methods. Our method is not a simple improvement of vertex-based methods, but a systematical solution for robustness of boolean evaluations. During intersection computation, we encode the triangle intersections into sets of planes, and then use these planes to determine exact tessellations. Subsequently, we classify each face in the tessellated meshes using small local BSP trees, which also compliment the P-reps to guarantee exactness.

- *R. Wang and B. Sheng are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Email:{jhcz,shengbin}@sjtu.edu.cn*
- *H. Fu is with the School of Creative Media, City University of Hong Kong. Email: hongbofu@cityu.edu.hk*
- *P. Li is with the Department of Mathematics and Information Technology, The Hong Kong Institute of Education. Email: pli@eduhk.hk*
- *E. Wu is currently a research professor at State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences. Email: ehwu@umac.mo*

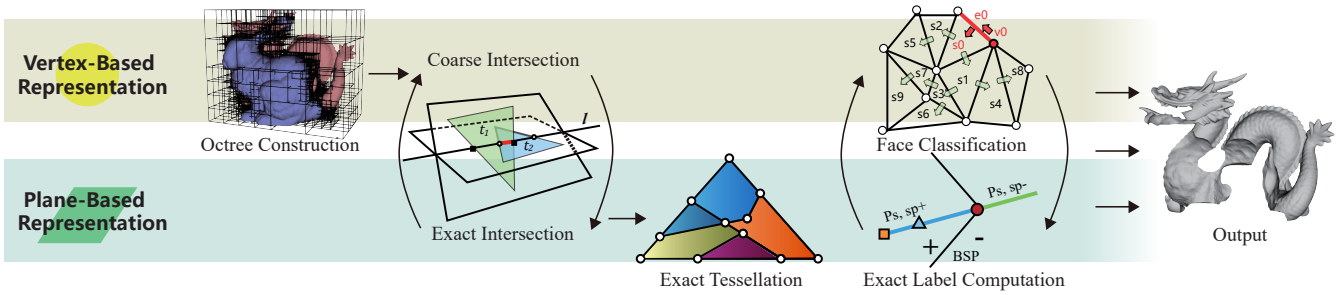


Figure 1. The illustration of our framework for boolean evaluation using hybrid representations. Vertex-based computation has its advantage of efficiency, and plane-based is necessary to guarantee exact geometry predicates.

In order to guarantee performance, we try to minimize the usage of plane-based exact predicates in these stages. In addition, while most existing boolean methods can only process two meshes in each time, our method is **variadic** [8], which can evaluate boolean expression with multiple input meshes immediately. Thus, our method reduce evaluation time of large CSGs by avoiding repetitive computation. Experiments have shown that our method is much faster than existing robust methods, and only around two times slower than non-robust methods.

2 Related Work

Boolean methods have been researched since their inception in the 1980s [11], [19]. Existing methods can be put into two categories: exact methods and approximate methods. Exact methods retain vertex positions, and the topology of the inputs is preserved as much as possible. The coordinates of intersection points between input meshes are computed by input configurations. However, the coordinates cannot be represented exactly using fixed-precision floating point numbers, hence exact arithmetic is needed for robustness. Conversely, approximate methods rediscretize the input mesh surfaces using techniques such as voxels, octrees, and Layered Depth Images (LDI). These methods generally have better performance and easier to be robust than exact methods, but the loss of precision and geometry information is inevitable. In this section, we first introduce previous work of these two categories, and then discuss plane-based boolean methods, which have close relation with our method.

2.1 Exact Methods

Some exact methods [6], [7], [12], [20], [21], are optimized for efficiency, but cannot guarantee robustness. They are implemented by fixed-precision floating-point arithmetic, so numerical errors are inevitable, which often leads to inconsistent results. Douze et al. [7] developed a very efficient method for handling very large meshes and arbitrary input primitives. However, this method makes general position assumption and cannot deal with coplanar situations.

On the other hand, some methods focus on robustness. The state-of-art method [8] try to handle a large range of topology deficiencies and guarantee to give solid outputs, with the cost of many extra computations such as self-intersecting test and vertex-rounding iterations. As robustness problem results from numerical errors, many researchers have attempted use arbitrary precision arithmetic [8], [9], [22], [23], [24], [25], [26] and exact interval computation [13], [27], [28]. However, exact arithmetic are too expensive in terms of computation time. For example, CGAL's [29] exact-arithmetic implementation [25] of Nef polyhedra [30] is more than 50 times slower than non-robust boolean operations [14]. Our method try to balance the performance and robustness, which guarantees the solid outputs under valid inputs without the use of exact arithmetic.

2.2 Approximative Methods

Because it is difficult to achieve efficiency, accuracy, and robustness simultaneously, some methods sacrifice accuracy for greater efficiency and robustness. Most such methods are based on volumetric representations of meshes. However, the quality of the resulting mesh depends on the resolution of the volume grid, and better quality requires significantly higher resolutions. To accelerate this process, some methods reduce the complexity of the output mesh [31], whereas others preserve non-intersected areas of the input mesh to avoid redundant tessellation [3], [4], [21], [32], [33], [34]. Moreover, with the development of general-purpose computing on graphics processing units (GPGPU), many of them utilize the computational power of graphics hardware for boolean evaluations. These methods have good performance, and are suitable for interactive applications. However, the fundamental problem of approximate methods arises from the grid-dependent nature of volumetric calculations. They inevitably suffer from geometric detail loss and unwieldy topological changes.

2.3 Plane-Based Methods

The concept of plane-based representations (P-reps) of polyhedrons was first described by Sugihara and Iri [18]. By using P-reps, boolean evaluations can be performed with only predicates, which are much easier to implement robustly. Berstein and Fussell [14] combined P-reps with BSP trees [16], [17] to develop a boolean method which is unconditionally robust with consistent inputs. Unlike vertex-based methods, their method does not rely on exact arithmetic to be robust. However, the merging of two BSP trees is has $O(mn)$ time complexity, where m and n are the size of the trees. Such high time complexity makes this method impractical for large meshes. Subsequently, Campen and

Kobbelt [15] improved this method by localizing BSP operations using an octree. In this method, mesh refinements only take place locally near intersections, which leads to better performance. However, their method splits the intersection regions and non-intersection regions, and requires extra stitching step to reconstruct topology. Also, the performance of localized BSP merging is still poor compared with the vertex-based methods. Our method uses plane-based geometry to guarantee no error introduced during boolean evaluation, while reduces the performance impact from plane-based computation with V-reps information, which is much cheaper to use.

3 Background and Overview

Our method is designed for boolean evaluations on arbitrary number of inputs primitives [19]. Primitives are triangular solid meshes free of self-intersecting. Geometry connectivity is also required as inputs. Even if there are topological deficiencies, our method still try to give acceptable answer. Our method is not sensitive to topological deficiencies which are not near the regions where primitives intersect.

In the following, we first discuss the causes of non-robustness of boolean methods. Then we analyze the cause of inefficiency of previous plane-based methods and discuss how to avoid these problems. Finally, we give an overview of the proposed method.

3.1 Robustness of Boolean Methods

The boolean evaluations on mesh primitives are, in essence, a process of face selection. Namely, given a boolean expression, the operation collect those faces that satisfy the expression to generate the final mesh. Whether a certain face $\boldsymbol{s}$ belong to the final mesh is determined by its inclusion labels and the boolean expression f:

$$\lambda_f(\boldsymbol{s}) = f(\boldsymbol{\Lambda}(\boldsymbol{s})) = f(\lambda_1(\boldsymbol{s}), \lambda_2(\boldsymbol{s}), \cdots, \lambda_n(\boldsymbol{s})), \quad (1)$$

where $\lambda_i(\boldsymbol{s})$ is the inclusion label with respect to primitive M_i. Each label has four conditions: completely inside (*in*), completely outside (*out*), on the boundary with consistent normal vector (*same*) or with opposite normal vector (*oppo*). To compute $\lambda_f(\boldsymbol{s})$, the labels with respect to all of the primitives (as $\boldsymbol{\Lambda}(\boldsymbol{s})$) must be known. The evaluation of boolean expression with inclusion labels are discussed in [7], [12]. If and only if $\lambda_f(\boldsymbol{s}) = same$, $\boldsymbol{s}$ is on the surface of the final mesh.

Unfortunately, not all of the input faces can be classified collectively. For some faces near the intersections, only parts of them belong to the final mesh. Therefore, an extra step before face classification is required to detect intersections between meshes. Then input meshes are tessellated according to these intersections to ensure that every face is completely inside, outside, or on the boundary of other input meshes.

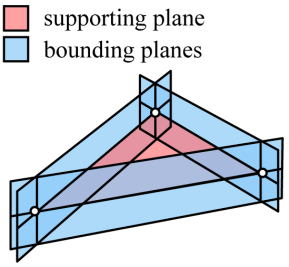
Most of the existing boolean methods follow such two-step paradigm—intersection computation and face classification, as does our method. Non-robustness comes from three stages of such boolean methods. 1) The intersection between faces may not be exactly computed. 2) Face tessellation produces inconsistent topology. 3) The computation of the inclusion labels is not consistent with the location of the faces. These problems all come from the numerical errors during geometric computation. Therefore, if these three stages can be performed in exact ways, boolean evaluations are robust.

3.2 Plane-based booleans

Comparing with applying exact arithmetic, using plane-based geometry [15] is more promising to produce efficient exact computations. Therefore, in our method, we use P-reps as the key to avoid numerical errors.

3.2.1 Plane-based representation

Using P-reps, each face $\boldsymbol{s}$ with n edges is represented by a supporting plane $\boldsymbol{p}_{s,sp}$ on which the face lies, and a set bounding planes $\{\boldsymbol{p}^i_{s,b} \mid i = 0, 1, ..., n-1\}$. Each edge line $\boldsymbol{e}^i_{\boldsymbol{s}}$ is represented by the intersection $\boldsymbol{p}_{s,sp} \cap \boldsymbol{p}^i_{s,b}$. Vertex $\boldsymbol{v}^i_s$ is represented by the intersection of $\boldsymbol{p}_{s,sp}$ and two consecutive bounding planes. We use the method of Campen et al. [15] for the exact conversion of triangles to their P-reps. The exact plane-based predicates are sped up using numerical filtering techniques by Shewchuk [35].



Other commonly used notations in this paper are presented below. The normal of a plane $\boldsymbol{p}$ is denoted as $\boldsymbol{n}(\boldsymbol{p})$. A line $\boldsymbol{l}$ of P-reps can be represented by the intersection of two planes $(\boldsymbol{p}^0_l \cap \boldsymbol{p}^1_l)$, hence, $\boldsymbol{l}$: $(\boldsymbol{p}^0_l \cap \boldsymbol{p}^1_l)$. The positive direction of the line $\boldsymbol{l}$ is defined by $\boldsymbol{n}(\boldsymbol{p}^0_l) \times \boldsymbol{n}(\boldsymbol{p}^1_l)$. A point $\boldsymbol{v}$ of P-reps can be represented by non-trivial plane triples $(\boldsymbol{p}^0_v \cap \boldsymbol{p}^1_v \cap \boldsymbol{p}^2_v)$, hence, $\boldsymbol{v}$: $(\boldsymbol{p}^0_v \cap \boldsymbol{p}^1_v \cap \boldsymbol{p}^2_v)$.

3.2.2 Efficient embedding

Under P-reps, vertex is represented with a much higher precision than under V-reps. A plane is represented by four parameters. Then a vertex takes twelve parameters under P-reps compared with only three parameters under V-reps. This makes geometric computations significantly slower under P-reps. Even though numerical filters can be applied to speed up, pure plane-based methods, such as [18], [22], are significant slower than vertex-based methods. On the other hand, a hybrid representation can bring both the efficiency of V-reps and the exactness of P-reps. And plane-based geometric computation should be avoided as much as possible to reduce computation cost.

BSP-based boolean algorithms are substantially based on planes, therefore are suitable to implement with P-reps [14], [15]. However, BSP merging is a pure plane-based algorithm. Also, BSP structure has additional two drawbacks which make it slow: a) BSP algorithms have high time complexity. b) BSP structure destroy geometry connectivity information, thus algorithms benefiting from connectivity require extra effort to reconstruct connectivity. Therefore, as an exact but not efficient representation, BSP should not substitute V-reps in total, but can serve as a compliment

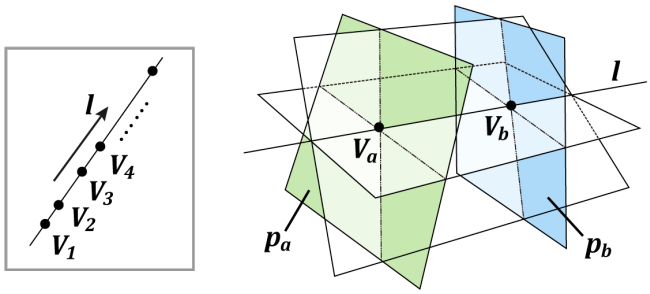


Figure 2. Geometric configuration of the linear ordering of points. Points $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$ are both on line $\boldsymbol{l}_{ab}$. We convert this problem into the plane ordering of $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ along $\boldsymbol{l}_{ab}$.

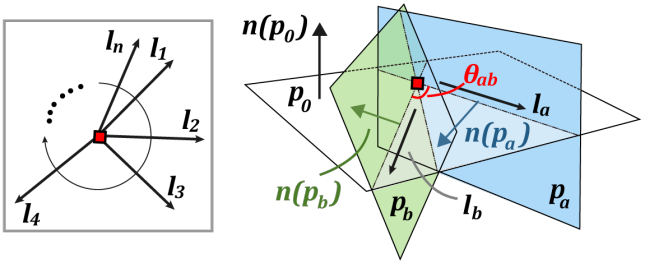


Figure 3. Geometric configuration of the circular ordering of lines. $\boldsymbol{l}_a\colon (\boldsymbol{p}_0 \cap \boldsymbol{p}_a)$ and $\boldsymbol{l}_b\colon (\boldsymbol{p}_0 \cap \boldsymbol{p}_b)$ are within plane $\boldsymbol{p}_0$.

of V-reps to help plane-based geometric computation. Also, BSP structure has to be localized to minimize its size.

In our method, we use V-reps for coarse tests and fast connectivity queries and P-reps for exact predicates. The framework of our method is based on [6], which is a vertex-based method. Plane-based algorithms are embedded into the processing. This embedding is not a simple plane-based implementation of vertex-based algorithms. We develop three efficient algorithms which is optimized under P-reps: triangle-triangle intersection tests, triangle tessellation and polygon classification. These three algorithms are corresponding to the three sources of non-robustness discussed in §3.1. The major difficulty is to guarantee the efficiency and exactness simultaneously. For this purpose, we avoid any computation under high precision, which means our method is designed to be implemented using only common double-precision floating-point arithmetic.

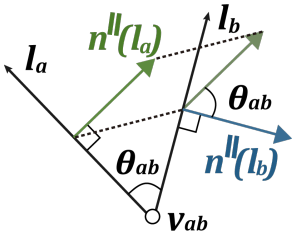
3.2.3 Mapping to three diemensions

Many vertex-based algorithms, such as 2D triangulation and face classification, is hard to implement under P-reps because they requires projection from three dimensions to planes or lines. Projections are performed on vertex coordinates which are absent under P-reps. Since plane-based predicates [14], [22] are usually performed under three dimensions, we have to find the equivalent three dimensional problem, whose projection to low dimension is the problem we want to compute. In the following, we discuss three such plane-based algorithms used in our method.

Point-line orientation
Within the plane $\boldsymbol{p}_0$, the point-line orientation is computed by the line parameters and point coordinates under V-reps. Under P-reps, we map this problem to point-plane orientation by picking a plane $\boldsymbol{p}_{\boldsymbol{l}}$ which satisfy three requirements: 1) the line is in $\boldsymbol{p}_{\boldsymbol{l}}$; 2) $\boldsymbol{p}_{\boldsymbol{l}}$ is not parallel with $\boldsymbol{p}_0$; 3) $\boldsymbol{n}(\boldsymbol{p}_0) \times \boldsymbol{n}(\boldsymbol{p}_{\boldsymbol{l}})$ has the same orientation as the line.

Linear ordering of points
Given a line $\boldsymbol{l}$ with two points on it, $\boldsymbol{v}_a\colon (\boldsymbol{p}_a^0 \cap \boldsymbol{p}_a^1 \cap \boldsymbol{p}_a^2)$, and $\boldsymbol{v}_b\colon (\boldsymbol{p}_b^0 \cap \boldsymbol{p}_b^1 \cap \boldsymbol{p}_b^2)$, we need to determine the linear order of the two points along $\boldsymbol{l}$ (see Fig. 2). To solve this problem, we choose one plane that is not parallel with $\boldsymbol{l}$ from the P-rep of each point, then convert this problem into one of determining the linear order of planes, which can be solved using the method of Banerjee et al. [22]. The chosen planes should have the same orientation with respect to $\boldsymbol{l}$ (the dot product between the plane normal and $\boldsymbol{l}$ must be positive), and unqualified planes need to be flipped.

Circular ordering of lines
During face tessellation, we need to know which intersections are neighbors (see Fig. 3). This requires circular ordering of directed lines around a vertex. The Lines can be sorted in a divide-and-conquer way, based on the relative order of each pair of lines. Thus, this problem is converted to one where, given two lines $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$ in a plane $\boldsymbol{p}_0$, circular order of the two lines needs to be computed. We can compute the order by the sign of $\sin\theta_{ab}$, where $\theta_{ab} \in (-\pi, \pi)$ is the angle from $\boldsymbol{l}_a$ to $\boldsymbol{l}_b$ in the top-view of $\boldsymbol{p}_0$.



We know the sign of $\sin\theta_{ab}$ is the same as the sign of $\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{l}_a \times \boldsymbol{l}_b)$. However, directly computing this equation requires extra precision to explicitly compute $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$. Fortunately, we found a efficient solution which only needs to compute the sign of a 3×3 determinants, whose elements are all floating-point numbers.

Theorem 1. *Given two directed lines* $\boldsymbol{l}_a\colon (\boldsymbol{p}_0 \cap \boldsymbol{p}_a)$ *and* $\boldsymbol{l}_b\colon (\boldsymbol{p}_0 \cap \boldsymbol{p}_b)$ *within plane* $\boldsymbol{p}_0$*, the following relation always stands:*

$$sign(\sin\theta_{ab}) = sign(\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}(\boldsymbol{p}_a) \times \boldsymbol{n}(\boldsymbol{p}_b))) \quad (2)$$

Proof. First, $\boldsymbol{n}(\boldsymbol{p}_a)$ and $\boldsymbol{n}(\boldsymbol{p}_b)$ are orthogonally decomposed along $\boldsymbol{n}(\boldsymbol{p}_0)$:

$$\begin{aligned} \boldsymbol{n}(\boldsymbol{p}_a) &= \boldsymbol{n}^{\parallel}(\boldsymbol{p}_a) + \boldsymbol{n}^{\perp}(\boldsymbol{p}_a) \\ \boldsymbol{n}(\boldsymbol{p}_b) &= \boldsymbol{n}^{\parallel}(\boldsymbol{p}_b) + \boldsymbol{n}^{\perp}(\boldsymbol{p}_b), \end{aligned} \quad (3)$$

where the superscript $\parallel$ refers to the component parallel with $\boldsymbol{p}_0$ and $\perp$ means the component orthogonal to $\boldsymbol{p}_0$. Since $\boldsymbol{n}(\boldsymbol{p}_0)$ is orthogonal with $\boldsymbol{p}_0$, we get:

$$\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}(\boldsymbol{p}_a) \times \boldsymbol{n}(\boldsymbol{p}_b)) = \boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}^{\parallel}(\boldsymbol{p}_a) \times \boldsymbol{n}^{\parallel}(\boldsymbol{p}_b)). \quad (4)$$

On the other hand, the angle between $\boldsymbol{n}^{\parallel}(\boldsymbol{p}_a)$ and $\boldsymbol{n}^{\parallel}(\boldsymbol{p}_b)$ is exactly θ_{ab}. Therefore,

$$sign(\sin\theta_{ab}) = sign(\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}^{\parallel}(\boldsymbol{p}_a) \times \boldsymbol{n}^{\parallel}(\boldsymbol{p}_b))). \quad (5)$$

By (4) and (5), the theorem is proved. □

3.3 Method Overview

Before going into details, we give an overview of our method. While the framework is similar to vertex-based boolean method, we address the major differences in each stage and briefly explain why these differences are necessary.

3.3.1 Intersection computation

We compute the intersections between pairs of triangles. The triangle-triangle intersection algorithm is largely based on Möller's algorithm [36]. However, the conventional vertex-based implementation of Möller's algorithm introduces numerical errors. Our plane-based intersection algorithm implicitly represents intersections using planes to avoid errors. In addition, we carefully deal with all of the degenerate situations, including point intersections, edge intersections, and coplanar intersections. Furthermore, octree is used to speed up the process. Details are provided in §4.

3.3.2 Deferred tessellation

Once all of the intersections between triangles are determined, input meshes are subdivided, so that all intersections occurs on edges and vertices. In many existing methods (e.g., [6], [8]), 2D constrained Delaunay triangulation (CDT) [37], [38] is used to implement this stage. In our method, intersections are represented by planes. Therefore, projection is not allowed and the 2D CDT has to be mapped to an equivalent 3D problem. However, a 2D CDT needs to construct edges between arbitrary pair of vertices, whose equivalent 3D operation is to construct new planes. The newly constructed planes have to contain arbitrary pair of vertices, thus require extra precision to represent.

To avoid this problem, we choose to tessellate in a more conservative way that do not add any new edges. As a consequence, the subdivided faces are no more triangles but polygons. We first perform intersection refinement to resolve intersections between intersections. We then construct a graph-like structure, called *tess-graph*, to guide the exact tessellation of each face. Details are given in §5.

3.3.3 Face classification

The purpose of this step is to collect faces that pass the boolean expression from the tessellated meshes to generate the final mesh. However, literally computing the inclusion label vector of each face is unacceptably slow for large CSGs. We utilize the connectivity information to propagate inclusion labels in a flood-filling manner. Typically, the seed face label is deduced by the label of a point on that face. However, the computation of point label can be incorrect without exact arithmetic. We exactly computed face label by a local BSP constructed according to the faces of neighborhoods. Our algorithm does not requires exact arithmetic. Also, the constructed BSP is typically small, which resulting in good performance. Details are given in §6.

4 Intersection Computation

In this step, the intersections between faces are determined by triangle-triangle intersection tests. We use Möller's algorithm [36] because of its efficiency and simplicity. To ensure exact intersection computation, we integrate plane-based geometry into this algorithm. In the following, we first describe our space division algorithm, which reduces the number of intersection tests. Then our plane-based intersection algorithm is introduced. Lastly, we discuss how to deal with degenerate situations.

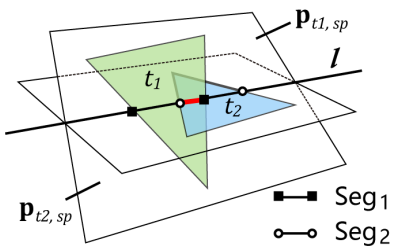
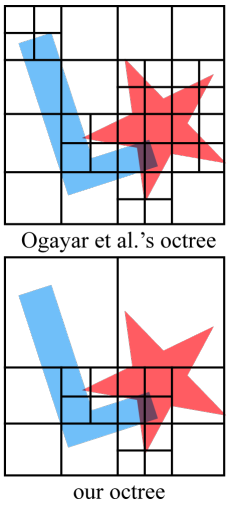


Figure 4. Seg_1 is the intersection between $\boldsymbol{p}_{t_2,sp}$ and t_1. Seg_2 is the intersection between $\boldsymbol{p}_{t_1,sp}$ and t_2. The intersection between t_1 and t_2 (red line segment) is the overlap of Seg_1 and Seg_2.

4.1 Space Division

As intersection detection is performed between each pair of faces, space division is necessary to reduce the number of testing pairs. We use an adaptive octree for this purpose. Our implementation is akin to the implementation of Ogayar et al. [6]. The intersections between triangle faces and octree nodes are efficiently detected using the separating axis theorem [39]. Octree leaves are classified into two types. If all faces that intersect a leaf belong to the same mesh, we regard it as a *normal cell*. Otherwise, it is regarded as a *critical cell*, within which the triangle-triangle intersection tests are performed.



The difference between our octree and that of Ogayar et al. is that we do not subdivide any normal cell, no matter how many faces it contains. Subdividing normal cells is only beneficial for the point-in-polyhedron test [40], which is seldom used in our method. This simplification saves significant computing time, especially when intersections between primitives are sparse located in small regions.

4.2 Plane-Based Intersection Test

We first make a quick review of Möller's vertex-based algorithm. Then we introduce our implicit representations of intersections using planes. After that, we discuss how to implement Möller's algorithm using plane-based geometry.

4.2.1 Review of Möller's algorithm

Möller's algorithm computes the intersection between two triangles t_1 and t_2 in three steps as shown in Fig. 4:

1) An early rejection is performed by testing whether t_1 intersects $\boldsymbol{p}_{t_2,sp}$, the supporting plane of t_2. The same test is also carried out between t_2 and $\boldsymbol{p}_{t_1,sp}$.

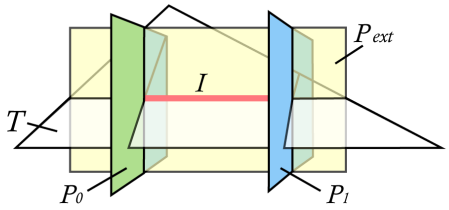


Figure 5. The geometry of planes in a PBI-rep. The red line segment ($\mathfrak{I}$) is the intersection represented by $\mathfrak{I}$.

2) The intersection between t_1 and $\boldsymbol{p}_{t_2,sp}$, denoted as Seg_1, and the intersection between t_2 and $\boldsymbol{p}_{t_1,sp}$, denoted as Seg_2, are computed separately.
3) The intersection between t_1 and t_2 is determined by computing the overlap between Seg_1 and Seg_2 .

The non-robustness of this algorithm stems from computing the coordinates of the intersection vertices (the end points of Seg_1 and Seg_2). Although implementation of this algorithm with arbitrary precision arithmetic produces exact coordinates, it is too costly for boolean evaluations with large CSGs.

4.2.2 Plane-based intersection representation

In our method, an intersection line segment $\mathfrak{I}$ is stored as $\{T, \boldsymbol{P}_{ext}, \boldsymbol{P}_0, \boldsymbol{P}_1, \mathcal{N}\}$. This is the plane-based intersection representation (PBI-rep, see Fig. 5). The first component, T, indicates which triangle $\mathfrak{I}$ lies on. $\boldsymbol{P}_{ext}$ indicates the plane that $\mathfrak{I}$ lies on. T should not be in $\boldsymbol{P}_{ext}$. Thus the first two components indicate that $\mathfrak{I}$ lies on the line $T \cap \boldsymbol{P}_{ext}$. Then two end points of $\mathfrak{I}$ are $T \cap \boldsymbol{P}_{ext} \cap \boldsymbol{P}_0$ and $T \cap \boldsymbol{P}_{ext} \cap \boldsymbol{P}_1$. The last component, $\mathcal{N}$, represents the neighborhood faces of $\mathfrak{I}$. $\mathcal{N}$ can be a single face, or a set faces from one or more input primitives.

For example, two triangle faces, t_1 and t_2, originating from meshes M_i and M_j respectively, intersect. Two intersections are generated, $\mathfrak{I}_{12}$ on t_1 and $\mathfrak{I}_{21}$ on t_2. For $\mathfrak{I}_{12}$, $T = t_1$ and $\boldsymbol{P}_{ext} = \boldsymbol{p}_{t_2,sp}$. $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ are boundary planes of t_2, which will be discussed later. The last component $\mathcal{N} = t_2$ in general situation. Sometimes, $\mathfrak{I}_{12}$ may lie on the edge of t_2 (see Fig. 7). This is called *edge intersection*. In this situation, the $\mathcal{N}$ is the set of all the faces from M_j adjacent to that edge. Edge intersection is discussed in §4.3.

4.2.3 Plane-based implementation

To implement Möller's algorithm by plane-based geometry, we first convert each triangle to its P-reps: a supporting plane $\boldsymbol{p}_{t,sp}$ surrounded by three bounding planes $\{\boldsymbol{p}^i_{t,b} |\ i = 0, 1, 2\}$. The substrates of the intersection algorithms are point-plane orientation [14] and linear order of points (§3.2), both of which can be implemented by plane-based geometry.

In the first step, the computation of signed distances involves only the vertices coordinates and the supporting planes of triangles. Therefore, if the early rejections occur, the bounding planes are not needed at all. In addition, according to the conversion method of Campen et al. [15], the supporting plane is represented by four double-precision floating-point numbers. The precisions of the first three parameters and the last parameter are L_a and L_d respectively.

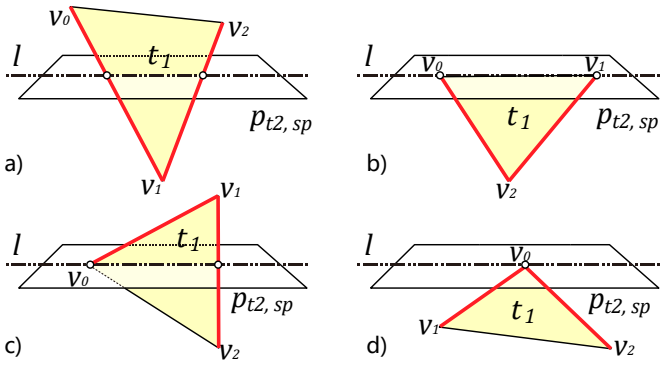


Figure 6. We denote the signed distance from point v_i to plane $\boldsymbol{p}_{t_2,sp}$ as d_i. The four conditions of intersection between t_1 and $\boldsymbol{p}_{t_2,sp}$ are: (a) $d_0 \cdot d_2 < 0, d_1 \cdot d_2 < 0$; (b) $d_0 = 0, d_1 = 0, d_2 \neq 0$; (c) $d_0 = 0, d_1 \cdot d_2 < 0$; (d) $d_0 = 0, d_1 \cdot d_2 > 0$. End points of Seg_1 are the intersections between $\boldsymbol{p}_{t_2,sp}$ and the related edges of t_1 (bold red lines).

They hold the relation $L_d = L_a + L + 1$, where L is the precision of the coordinates of input vertices. This means we can compute the signed distance exactly within double-precision. With these two facts, our early rejection does not involve plane-based computation, thus is very efficient.

If early rejection is not triggered and t_1 and t_2 are not coplanar, Seg_1 and Seg_2 are computed. The end points are intersections between the supporting plane of one triangle and an edge of the other triangle. By computing the bounding planes, the end points can be implicitly represented by plane triples, $\boldsymbol{p}_{t_2,sp} \cap \boldsymbol{p}_{t_1,sp} \cap \boldsymbol{p}^i_{t_1,b}$, where $\boldsymbol{p}^i_{t_1,b}$ is the bounding plane of the related edge. Fig. 6 shows all of the possible intersection conditions between t_1 and $\boldsymbol{p}_{t_2,sp}$. The coplanar situation is discussed specifically in §4.3.

To avoid repetitive vertices in the final result, we perform vertex repetition elimination in this step. Because P-reps are used, the coincidence tests of the vertices are exact.

4.3 Handling Degenerate Situations

In most situations, intersections between triangles are line segments. However, they can also intersect on a point or a convex area. Even if the intersection is a line segment, the intersection can be on a primitive edge. These degenerate situations prevent us from performing robust boolean operations. In this section, we demonstrate our simple but effective ways of dealing with these degenerations, which conceals the complexity of intersections, and simplifies later processing.

4.3.1 Point intersection

If two triangles intersect at a single point (e.g., Fig. 6d) then the intersection cannot be represented using our PBI-rep. In this situation, we simply add the intersection point into the related triangles, guaranteeing correct tessellation. No intersection line segment is introduced.

4.3.2 Edge intersection

When a triangle intersects with the edge of other triangle, the generated intersection is referred as *edge intersection*. The neighboring faces of the intersection is the set of all the faces that shared the edge rather than a single face. For example,

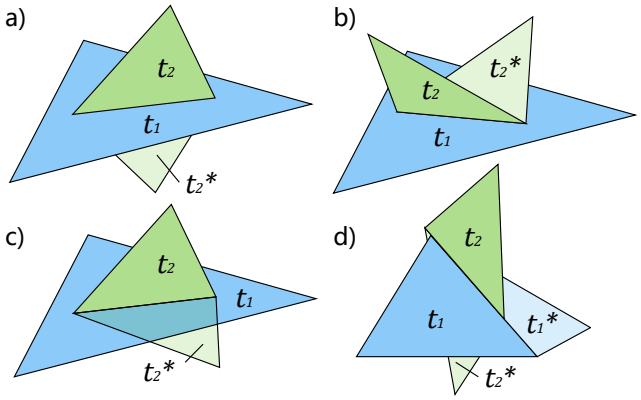


Figure 7. Different conditions of edge intersection. Triangle faces t_1 and t_1^* are companion faces. Triangle faces t_2 and t_2^* are companion faces. a) t_2 and t_2^* are on different sides of t_1. b) t_2 and t_2^* are on the same side of t_1. c) t_2^* is coplanar with t_1. d) Both t_1 and t_2 have companion faces: the intersection is an edge intersection for both t_1 and t_2 (instead of only for t_2 is the previous three conditions).

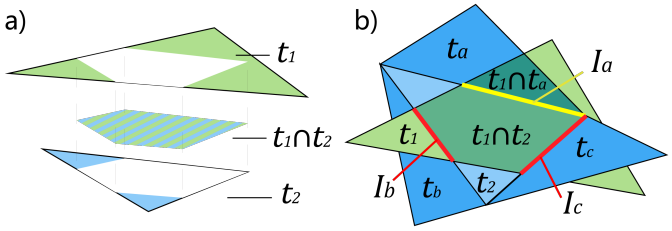


Figure 8. a) Coplanar situation. t_1 and t_2 intersect in 2D, dividing each other into a convex overlapping area and a exclusive area. b) Possible configurations of the companion faces. The blue triangles originate from the same mesh. For the yellow segment, it is not necessary in the final mesh, and it will not be detected. For the red segments, the companion triangles are not coplanar with t_1, so they can be detected during intersection tests between t_1 and the companion triangles (t_a and t_b).

in Fig. 7a, the neighboring faces of the intersection on t_1 are t_2 and t_2^*. The same intersection on t_1 will be detected twice, because t_1 intersects both t_2 and t_2^* at the same line segment. In fact, edge intersections are often detected for multiple times. We solve this duplication in tessellation stage (see §5). We regard t_2 and t_2^* as *companion* triangles, because they are the neighboring faces of the same edge intersection. This concept will be referred to again in the discussion of coplanar cases.

4.3.3 Copalnar

Consider two triangle faces, t_1 and t_2, which intersect within a common plane. Both t_1 and t_2 divide each other into two areas–a convex overlapping area and a exclusive area (see Fig. 8a). If we tessellate both t_1 and t_2 according to the boundary of the overlapping area, we can guarantee correct topology. This strategy is adopted by many previous methods [8], [12]. Conversely, in our method, we treat coplanar situations as if they do not intersect at all. We do not need any 2D intersection test. In this way, we avoid the processing of complex situations of 2D intersections, while having no side effect on the topological correctness.

We find that t_1 is actually clipped by edges of t_2 (see 8b). That means that we can view 2D intersections as special cases of edge intersections. There can be up to three edge intersections in one 2D intersection (red and yellow line segments). As discussed, edge intersections will be detected for multiple times by the companion triangles. Hence, we can rely on them to detect the intersections.

However, if all the companion triangles are coplanar, none of them will detect the intersection (the yellow segment). Fortunately, the intersection is not necessary in this situation. The neighboring faces of the intersection are all within the same plane. If such intersection is on the surface of the final mesh, the surface in the neighborhood of this intersection must be a plane. Therefore, the intersection is not necessary to be an edge in the final model and we can omit it.

Because we do not process coplanar intersections, the coplanar areas (the stripe areas) may have different tessellations in different primitives. It seems that inconsistent topology can occur in the final mesh. In fact, we do not need to worry about that, because faces from a certain primitive in such coplanar area are collected or abandoned together. The coplanar area, if on the surface of the final mesh, will inherit only one of the tessellations.

5 Deferred Tessellation

Tessellation is performed on each intersecting triangle. This stage is referred as deferred tessellation, because the tessellation happens after all of the intersections between triangles are computed, instead of clipping triangles incrementally at each intersection.

Many methods use CDT to perform tessellation on triangles. They treated triangle faces as convex triangulation zones, and intersections as constraints. However, CDT algorithms [37], [41] are performed in 2D. Even after mapping to three dimensions, implementing a plane-based CDT requires introduction of new planes. These new planes are used to guarantee that each subface is a triangle (see Fig. 9d). We find double-precision is not enough to store the coefficients of these planes. Therefore, we choose not to add any new edges, performing minimal tessellation on each triangle. In this way, the subdivided faces are no more triangles, but general polygons.

In our method we first perform intersection refinement to ensure that intersections intersect each other only on end points. After that, we perform our minimal tessellation based on *tess-graph*, a graph-like description of the intersections on a given face.

5.1 Intersection Refinement

Intersection vertices can be introduced by the intersection of three triangles, in addition to intersections between an edge and a triangle. Triangle-triangle intersection tests only compute the latter. Thus, it is necessary to refine these triangle-triangle intersections before the final tessellation.

Intersection refinement is performed on the scope of each intersecting triangle. For triangle face t, we collect all of the intersections on t as a set $\mathbf{\Gamma}(t)$ and refine them together. We also include the three edges of t in $\mathbf{\Gamma}(t)$, because the edges are also involved in the step of subfaces extraction. In $\mathbf{\Gamma}(t)$, the three edges are represented by PBI-reps. The

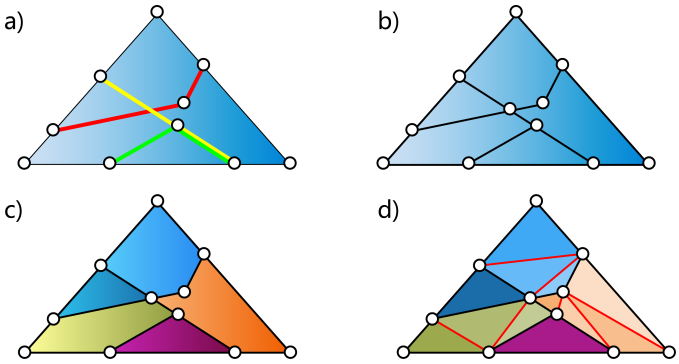
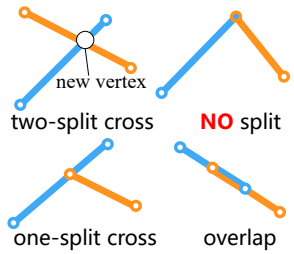


Figure 9. a) Different colors indicate that the intersections originate from different meshes. The yellow and red intersections intersect at a point. The yellow intersection overlaps with the green intersection. b) After refinement, we introduce a new vertex $\boldsymbol{v}_a$, and merge overlapping intersections into a single edge $\boldsymbol{e}_b$. c) Our tessellation method does not guarantee that all of the faces are triangular. d) If triangulation is performed, new edges (red lines) are introduced and double-precision is not enough to hold the plane coefficients of their P-reps.

neighboring faces $\mathcal{N}$ are set as N/A because they do not have a neighboring face. Other PBI-rep components can be determined by the P-reps of the edges. The refinement of $\boldsymbol{\Gamma}(t)$ is done using only plane-based geometric predicates, in the following three steps.

Coincidence elimination We merge coincidence intersections that have the same end points. Intersections are undirected, so intersections with inverse end points are also coincident. The PBI-rep of the merged intersection is inherited from either of the original intersections, except for the neighboring faces $\mathcal{N}$. The neighboring faces of the merge intersection is the union of all the neighboring faces of original ones. It means from now on, an intersection may have neighboring faces from different primitives.

Intersection resolving Each pair of intersections is tested if they intersect in any location other than end points. The tested intersections should be from different primitives, since primitive inputs are free of self-intersecting. The three conditions of intersections are illustrated on the right. When two intersections intersect, at least one of them is split. New vertices may be introduced, whose P-reps are intersections between the common plane and the two $\boldsymbol{P}_{ext}$ of the PBI-reps of the two intersections. Because one intersection may be in contact with more than one other intersection, this splitting is deferred until all of the splitting points are found. The PBI-reps of the split segments inherit that of their father's, except for $\boldsymbol{P}_0, \boldsymbol{P}_1$, since the split segments have different end points.



Coincidence elimination (revisited) If two intersections are collinear, resolving their overlap may produce new coincident intersections. Therefore, the first step of the process is repeated here.

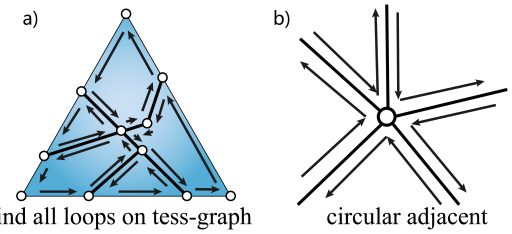


Figure 10. a) To tessellate a triangle, we find all of the valid loops on a tess-graph. The direction of the loops must be coherent with the triangle normal (assuming here that the normal points to the outside of the paper). b) Each circular adjacent edge pair is an angle of the tessellated polygon.

5.2 Tessellation by Tess-Graph

To perform tessellation and extract subfaces, we use tess-graph, which is the graph description of the tessellated face topology. For each intersecting triangle face, we construct a tess-graph according to the refined set of intersections. Nodes of the tess-graph represent the end points of intersections, and connections between nodes represent intersections. The construction of a tess-graph is straightforward, and readers should be able to determine the details.

After the tess-graph is constructed, we extract subfaces by valid loops. A valid loop is a loop in tess-graph which satisfy two criteria: 1) the direction of the loop should correspond with the face normal, and 2) consecutive connections on the loop should be adjacent by circular order. Each valid loop corresponds to an intersection-free face. After all of the valid loops are determined from the tess-graph, the corresponding face is tessellated (see Fig. 10). To facilitate face classification, we also store neighboring faces in the edges of the new faces. Because the edges have one-to-one correspondence to the connections in tess-graph , and the connections have one-to-one correspondence to the PBI-reps, the neighboring faces of edges is determined from the corresponding PBI-reps.

6 Face Classification

We traverse each face of the primitives, and determine which belong to the final mesh. The faces are classified by evaluating their inclusion label vector. Our classification method utilizes the space coherence of the label vectors, and shares the classification results among neighboring faces. Because we store the neighboring faces in the edges where intersections occur, the difference of labels between adjacent faces are computed according to the BSP constructed with these neighboring faces. For arbitrary adjacent faces $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, the label vectors $\boldsymbol{\Lambda}(\boldsymbol{s}_1)$ and $\boldsymbol{\Lambda}(\boldsymbol{s}_2)$ differ if their shared edge $\boldsymbol{e}_{12}$ lies on the surface of any other primitive. In that cases, some neighboring faces should be stored on $\boldsymbol{e}_{12}$. The owner primitives of these neighboring faces indicate which labels differ. If there are neighboring faces from mesh M_x, the labels $\lambda_x(\boldsymbol{s}_1)$ and $\lambda_x(\boldsymbol{s}_2)$ differ. We outline our classification method in Algorithm 1.

6.1 Label Computation

In methods like [6], [12], the triangle barycenter is used to compute the inclusion label of the face. This is because

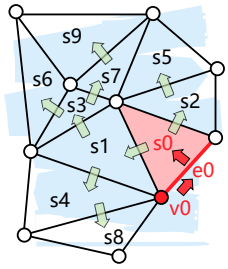
Algorithm 1 Fast Face Classification

Input: Tessellated primitives and boolean expression f
Output: Classification $f(\mathbf{\Lambda}(\boldsymbol{s}_i))$ of each face $\boldsymbol{s}_i$

```
Select a proper seed face s_0;
Compute the seed label vector Λ(s_0);
PROPAGATE( s_0 , Λ(s_0));

function PROPAGATE( s , Λ(s))
    Compute f(Λ(s));
    for each neighboring face s_{s,i} do
        if s_i has been classified then
            continue;
        end if
        if there are PBI-reps J_k on e(s_{s,i}, s) then
            compute Λ(s_{s,i}) by Λ(s) and J_k;
            PROPAGATE( s_{s,i} , Λ(s_{s,i}));
        else
            PROPAGATE( s_{s,i} , Λ(s));
        end if
    end for
end function
```

the face can be classified as a whole, and all inner points of the face has the same label vector. In methods like [8], points inside the cell is used for label computation, and the face labels are decided by the labels of the two sides of the face. However, all these methods require to construct new vertices, whose coordinates require rational numbers to store, or round-off errors may be introduced. Incorrect label computation may occur because of such errors. To avoid this, we use the vertices instead, the exact coordinates of which are known.



However, there are two discrepancies between the face labels and vertex labels. First, the vertex label is not always equal to the face label. Second, the face label has two subclasses in the *on* case, *same* and *oppo*, because the face has an orientation.

To deal with these discrepancies, we start from a seed vertex $\boldsymbol{v}_0$, which is one of the end point of an edge $\boldsymbol{e}_0$ of a face $\boldsymbol{s}_0$. We assume that the label vector $\mathbf{\Lambda}(\boldsymbol{v}_0)$ is known. Then we use $\mathbf{\Lambda}(\boldsymbol{v}_0)$ to compute $\mathbf{\Lambda}(\boldsymbol{e}_0)$, and then $\mathbf{\Lambda}(\boldsymbol{s}_0)$. The trace of label propagation is:

$$\boldsymbol{v}_0 \to \boldsymbol{e}_0 \to \boldsymbol{s}_0 \to \boldsymbol{s}_1 \to \boldsymbol{s}_2 \to \dots \quad , \tag{6}$$

There are three basic operations: $\boldsymbol{v_e} \to \boldsymbol{e}$, $\boldsymbol{e_s} \to \boldsymbol{s}$ and $\boldsymbol{s}_i \to \boldsymbol{s}_j$, where $\boldsymbol{v_e}$ is the end point of $\boldsymbol{e}$, $\boldsymbol{s_s}$ is the edge of $\boldsymbol{s}$, and $\boldsymbol{s}_i$ is adjacent to $\boldsymbol{s}_j$.

Theorem 2. *Given the partial orders on $\succ$ in and on $\succ$ out, the following relationship is true for labels within the tessellated primitives:*

$$\lambda_k(\boldsymbol{v_{e_s}}) \succeq \lambda_k(\boldsymbol{e_s}) \succeq \lambda_k(\boldsymbol{s}), \tag{7}$$

where $\lambda_k(x)$ is the label of x for a certain primitive M_k.

Proof. We prove this theorem by contradiction. When $\lambda_k(\boldsymbol{v_{e_s}}) \succeq \lambda_k(\boldsymbol{e_s})$ is not satisfied, it means for a certain label λ_k, $\lambda_k(\boldsymbol{v_{e_s}})$ is *in* or *out*, but $\lambda_k(\boldsymbol{e_s}) \neq \lambda_k(\boldsymbol{v_{e_s}})$. Let us assume that $\lambda_k(\boldsymbol{v_{e_s}}) = in$ and $\lambda_k(\boldsymbol{e_s}) = on$ or *out*.

Because $\boldsymbol{v_{e_s}}$ is inside of closed regular set (solid) M_k, according to the continuity of space, any point in $U(\boldsymbol{v_{e_s}})$, which is the neighborhood of $\boldsymbol{v_{e_s}}$, should be inside of M_k. And because $\boldsymbol{e_s^\circ} \cap U(\boldsymbol{v_{e_s}})$ is not empty (superscript $\circ$ means interior), $\boldsymbol{e_s}$ should be inside of M_k, which contradict our assumption. □

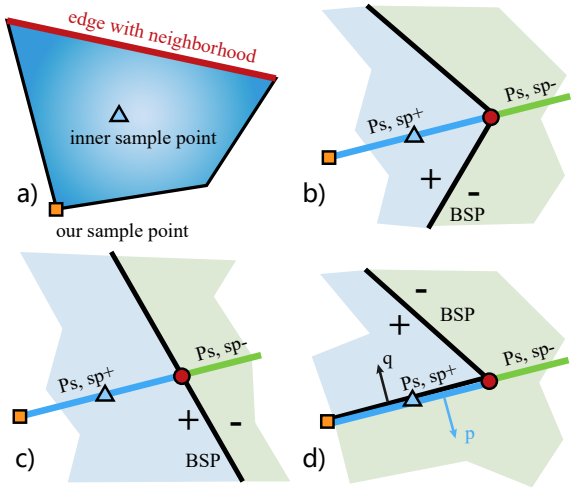


Figure 11. a) Because the inner sample point includes geometric constructions, we choose the face vertex (orange) instead to compute the face label. b-d) Different types of classification. The simple structure of the neighborhood BSP guarantees the same classification result for all of the points on $\boldsymbol{p}^+_{\boldsymbol{s},sp}$. In d), the classification result is *on* so it is necessary to determine the orientation from the normal of the BSP splitting plane q and the face normal p. In this cases, the label should be *oppo* because p and q are opposite.

It indicates that when we perform $\boldsymbol{v_e} \to \boldsymbol{e}$ and $\boldsymbol{e_s} \to \boldsymbol{s}$, from a low-dimension to a high-dimension, we decide whether the *on* label changes to *in* or *out*, or remains *on*. In addition, when considering the orientation of the face, we also need to determine whether the *on* labels change to *same* or *oppo* during the $\boldsymbol{e_s} \to \boldsymbol{s}$ operation.

$\boldsymbol{e_s} \to \boldsymbol{s}$: From theorem 2, we know that if $\lambda_k(\boldsymbol{e_s}) \neq on$, then $\lambda_k(\boldsymbol{e_s}) = \lambda_k(\boldsymbol{s})$. Conversely, when $\lambda_k(\boldsymbol{e_s}) = on$, we can build a trivial BSP [17] using these the neighboring faces which belongs to M_k stored in $\boldsymbol{e_s}$. The BSP can be used to compute inclusion state $\lambda_k(\boldsymbol{s})$ if a point can be sampled from $\boldsymbol{s}^\circ \cap \boldsymbol{U}(\boldsymbol{e_s})$, where $\boldsymbol{U}(\boldsymbol{e_s})$ is the neighborhood space of $\boldsymbol{e_s}$. However, we cannot guarantee that such a point can be found with exact representation of double-precision.

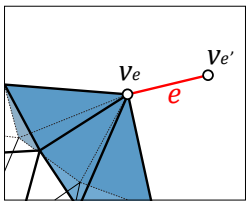
Theorem 3. *The BSP is constructed by the neighboring faces around a certain edge $\boldsymbol{e}$. Then the inclusion states of all the points on interior of the half plane $\boldsymbol{h}$ are the same, if 1) $\boldsymbol{e}$ is within $\boldsymbol{h}$, and 2) $\boldsymbol{e}$ is the boundary of $\boldsymbol{h}$.*

Proof. The correctness of the theorem is obvious because the supporting planes of the neighboring faces always cross the edge $\boldsymbol{e}$ and extend to infinity. □

Let $\boldsymbol{h}^+_{\boldsymbol{s}}$ be the half of the supporting plane of $\boldsymbol{s}$ which lies on the interior side of $\boldsymbol{e_s}$ (see Fig. 11). By theorem 3, the label of points on the half plane $\boldsymbol{h}^+_{\boldsymbol{s}}$ are identical. Since

$\boldsymbol{h}_{\boldsymbol{s}}^{+} \cap \boldsymbol{s} \neq \varnothing$, we can compute $\lambda_k(\boldsymbol{s})$ by determining $\lambda_k(\boldsymbol{v}_x)$, where the vertex $\boldsymbol{v}_x \in \boldsymbol{h}_{\boldsymbol{s}}^{+}$. For polygons, we can always find at least one such vertex, which is represented exactly (by planes or vertex coordinates). In addition, we assign each BSP splitting plane a normal vector by the normal of the triangle it contains, from which we can decide the orientation (*same* or *oppo*) when $\lambda_k(\boldsymbol{s}) = on$.

$\boldsymbol{v_e} \rightarrow \boldsymbol{e}$: If $\lambda_k(\boldsymbol{v_e}) = on$, we first check whether $\lambda_k(\boldsymbol{e}) = on$, by checking whether there is neighboring faces from M_k on $\boldsymbol{e}$. If not, we need to find where $\boldsymbol{v_e}$ lies on M_k. Because our tessellated meshes contain connectivity information, we can easily find all neighboring faces of $\boldsymbol{v_e}$. After that, the similar BSP-based classification method (as in $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$) can be used to determine the high dimension label $\lambda_k(\boldsymbol{e})$. We use the other end point $\boldsymbol{v}'_{\boldsymbol{e}}$ as the sample point for the BSP classification.



$\boldsymbol{s}_i \rightarrow \boldsymbol{s}_j$: We need to determine which labels change if shared edge contains neighboring faces from other primitives. We can know which labels differ by the owner primitives of these neighboring faces. And the recomputation of these labels is exactly the same as in $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$.

6.2 Acceleration by Caching

If the CSG tree is large, with hundreds of primitive nodes, evaluations of boolean expression by inclusion label vectors of each face can be costly. However, using label space coherence, we can reduce the computation time by caching the evaluation results.

The basic caching strategy is to cache the final evaluation results. Faces which share the same label vector can be classified together. We can also perform an intermediate results cache. The boolean expression can be simplified if some components of label vector are known. For example, assume we have a boolean expression $f = M_1 \cup (M_2 \cap M_3 - M_4)$. Given the values of two labels $\lambda_1(\boldsymbol{s}_i) = out$, $\lambda_2(\boldsymbol{s}_i) = in$, the expression can be rewritten as $f(\lambda_1 = out, \lambda_2 = in) = out \cup (in \cap M_3 - M_4)$. Using the combination rules, we can simplify the expression to $f(\lambda_1 = out, \lambda_2 = in) = M_3 - M_4$. In a large CSG, a certain primitive often intersects with only a few other primitives $\Theta = \{M_{n_1}, M_{n_2}, \cdots, M_{n_x}\}$. All of the faces in this primitive have the same labels with respect to primitives not in Θ. Therefore, if we can determine these fixed labels and simplify the boolean function, then we compute the final label in this primitive based on the simplified expression.

7 Results and Discussion

We implemented our proposed method in C++, and tested a series of models on a PC with an Intel Core i5 CPU and 8 GB of RAM. We compared our method with several existing methods, including CGAL [29], Cork [42], QuickCSG [7], Carve [43], and online service of Campen and Kobbelt's plane-based method [15], [44]. Among them, CGAL and Carve use exact arithmetic. To ensure fairness, we turn off any features of multi-thread computation during the comparison.

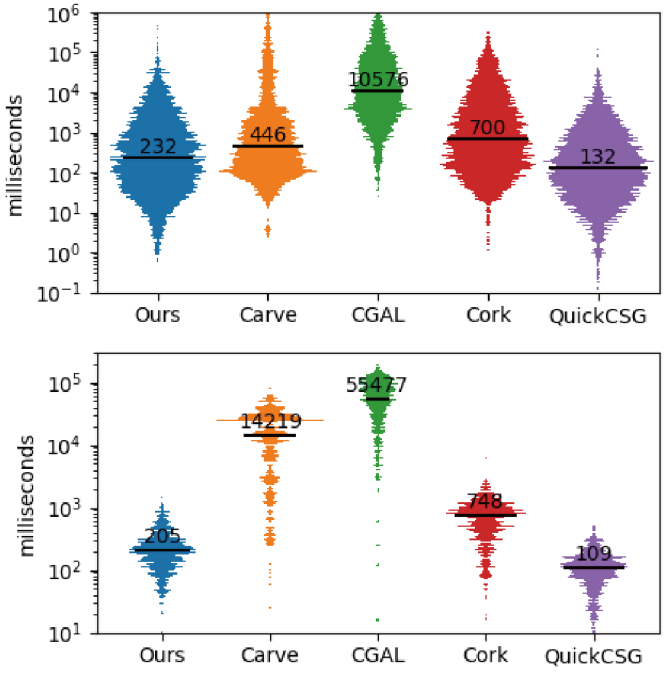


Figure 12. The performance of test on Thingi10k (top) and Barki et al.'s dataset (bottom). We outline the median of processing time for each method. The different histogram areas result from the different success rate.

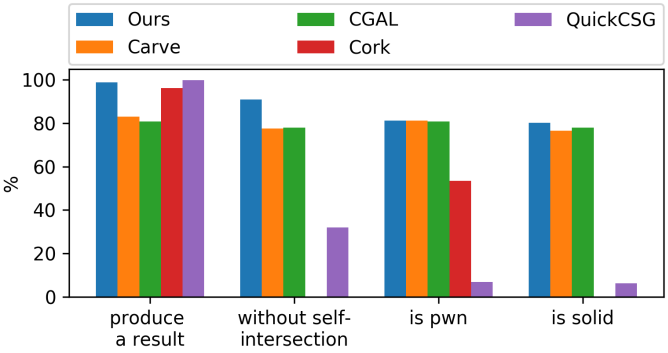


Figure 13. The result meshes of our method for the self-union of Thingi10k are cleaner than state-of-art methods. The Cork and QuickCSG assume general position and cannot handle self-union at all, so they failed to produce solids in most of the cases.

7.1 Self-Union on Thingi10K Dataset

We reproduce the experiments of [8] on the Thingi10K dataset on website thingiverse.com. The dataset currently contains 9956 .stl models, which are heavily biased towards 3D printing modeling by amateurs or semi-professionals. Since .stl models are triangle streams rather than meshes, we merge the exact coincident vertices to reconstruct connectivity. After that, we check the cleanness of all of the models. Among the 9956 meshes, 4509 meet the requirements of solid. The face numbers of these meshes generally follows a log-normal distribution with the average of 17k. We limit our comparison to these solid meshes.

The comparison results are available for CGAL, Cork, Carve and QuickCSG. Web service of Campen et al. [15], [44] does not support batch processing. Therefore, we made a few tests manually, and roughly a half of them failed. The

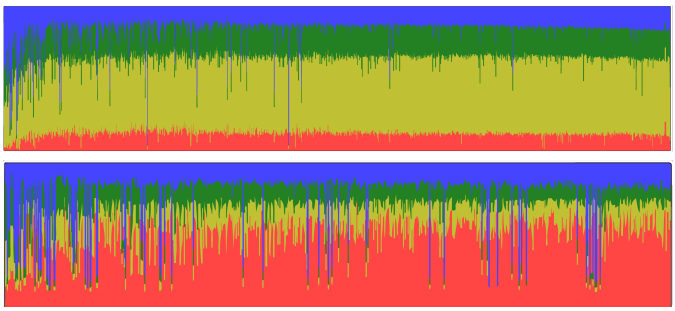


Figure 14. Performance profile of tests on Thingi10k (top) and Barki et al.'s dataset (bottom). The red, yellow, green, blue areas represent the time percentage of octree construction, intersection computation, tessellation and classification, respectively. Each one-pixel column represents a test. All the tests are sorted by total processing time.

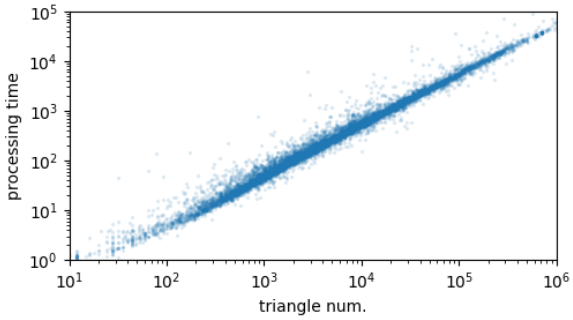


Figure 15. For the self-union tests on Thingi10k, the linear relation between processing time and triangle number is apparent.

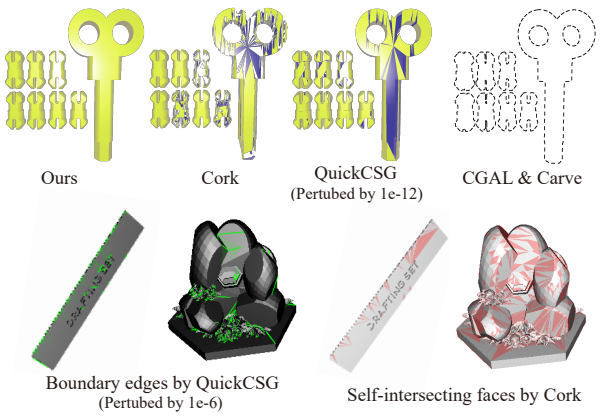


Figure 16. *First row*: the results of self-union of *components*. QuickCSG (adding perturbation of 1e-12) and Cork cannot produce valid results. CGAL and Carve simple crashed and do not give a result. *Second row*: In our self-union tests, QuickCSG tends to produce results with many boundary edges (green lines), and Cork tends to produce many self-intersecting faces (red faces).

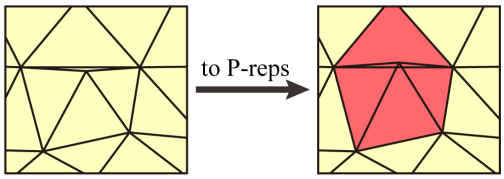


Figure 17. After coordinates approximation, the 'thin' triangle in the middle intersect adjacent faces and results in self-intersection on the mesh (red triangles).

implementation of recent works [8], [9] are not available. Both methods are based on exact arithmetic. The performance of method in [8] is claimed to be close to Carve (with parallel computation) by the results on the same dataset Thingi10k in their paper.

The computation time of our method outperform most of other methods (see Fig. 12 *top*). Even compared with the fastest QuickCSG, which is not exact, our method takes only about two times of the processing time. We profile each stage of our method, and the results indicate the bottleneck is the intersection computation (see Fig. 14 *top*). Also, from Fig. 15, the linear relation between triangle number and processing time implies the ability of our method to handle large boolean evaluations.

Our method produces results in 98.8% of the tests. The cleanness of resulting meshes are tested by checking self-intersecting (we use VCG [45]) and the total signed incidence of every edge [8]. If the total signed incidence of every edge of a mesh is zero, then the mesh is a piecewise-constant integer *generalized winding number* (PWN) [8] mesh. These two requirements are sufficient conditions of solid. According to Fig. 13, over 80% of our results are solids, which outperform all other methods. We also notice very low rates of success for Cork and QuickCSG. This is because both methods assume general position, which is not the case for self-union. Even after perturbation, they can hardly produce solid results (see Fig. 16).

The failure cases and unclean results of our method do not impair the robustness of our algorithms. The reasons lie on the degenerate faces and close-to-degenerate faces from the inputs. The normal vectors of degenerate faces cannot be defined so face classification may fail when encounter such faces. Also, close-to-degenerate faces can become degenerate faces or lead to self-intersecting (see Fig. 17) after conversion to their P-reps. This is because the conversion method [15] requires coordinates rounding-off. In our implementation, the IEEE-754 single-precision floating-point coordinates (precision=24) are rounded to the precision of 18∼20.

7.2 Binary Boolean Operations

The most common situations of boolean evaluations in CAD are binary evaluations. It is because people are used to incrementally refine their models by adding or subtracting one shape each time. Although our method is variadic, we perform comparisons of binary boolean evaluations. We reproduce the experiments in [8] on the dataset of Barki et al. [9]. The dataset of Barki et al. contains 26 triangle meshes. All of them are closed and manifold, and 10 of them contain local self-intersecting. We exhaustively perform union, intersection, and both asymmetric differences for all pairs, producing $26 \times (26-1)/2 \times 4 = 1300$ results.

The configuration of such tests satisfy the general position assumption, so Cork and QuickCSG can produce correct result in more cases than during self-union tests. But we address that such tests are performed only for completeness and coplanar faces are important situations in CSG modeling. The results (see Fig. 12 *bottom*) shows that our method is slightly slower than QuickCSG, and much faster

than other exact methods. We notice the bottleneck shifts from intersection computation to octree construction in this time (see Fig. 14 *bottom*). This is because compared with self-union tests, the binary boolean evaluations generally process fewer intersections. We also find that Carve has shows much worse performance in these cases than in self-union tests, which indicates that Carve does not localize the boolean evaluations within intersection areas.

7.3 Variadic Boolean Operations

To identify the ability of the methods to evaluate large CSGs, we reproduce the experiments of Douze et al. [7]. Some of variadic test cases are not presented because of the 8GB memory limitation. Since only QuickCSG and our method are variadic, comparisons with the other methods are performed by decomposing CSG into binary boolean operations. We also add an extreme CSG with 801 primitives to prove the effect of label caching (§6.2).

The performance of our method (see Table 1) is not that good in these cases, because our method exhaustively compute all intersections between primitives, which is not necessary. The QuickCSG is much faster than our method, but generally produces results with worse qualities. Except QuickCSG, our performance is better than other methods in most time. The evaluation time of *Ring* & *Ball* is 38.1s in our first try. We find that each primitive usually intersects with less than 10 other primitives. Therefore, we applied the label caching, and the total computation time drops to only half of the original.

7.4 Implementation

Tess-graph with multiple components. There is no guarantee that the tess-graph will be a connected graph. If a tess-graph contains more than one connected component, we need to merge identical valid loops, which allows us to generate polygons with non-zero genus. To find identical loops, we construct an auxiliary connection C_{ext} for each inner component, which connects a vertex $\boldsymbol{v}_o$ on the outer component and a vertex $\boldsymbol{v}_i$ on the current inner component (see Fig. 19a). After that, we search all the connections belongs to the current inner component that intersection C_{ext}, and find the one (as C_1) which is the nearest to $\boldsymbol{v}_o$. Then we search all the connections not belongs to the current inner component that intersection C_{ext}, and find the one (as C_2) which is the nearest to C_1. Finally, by figuring out in which valid loops C_1 and C_2 are, we find the identical pair of loops.

To guarantee that C_{ext} has an exact P-rep within double-precision, the vertex of triangle is chosen as the $\boldsymbol{v}_o$. On the other hand, $\boldsymbol{v}_i$ must be the vertex generated by the intersection between an triangle and an edge (as $\boldsymbol{e}$). All intersection points introduced by triangle-triangle intersections are of this type. In this way, we obtain three vertices with exact coordinates ($\boldsymbol{v}_o$ and the two end points of $\boldsymbol{e}$). Therefore, we can construct the plane on which C_{ext} lies by the same method of generating supporting planes of triangle faces (see Fig. 19b).

Seed label generation. The label vector of the seed vertex can be generated by a point-in-polyhedron test [46], using the octree as an acceleration structure [40]. However, a simpler strategy can be adopted which chooses a vertex with known labels as the seed. The inclusion labels of vertex with the maximum x-coordinate are either out (in, if the complement is applied on the mesh) or on. The on labels can be determined by connectivity. Exceptions can occur because of our ignorance of coplanar situations—vertices may not be added to the primitive, even if they are on surface of it. Fortunately, if seed vertex is chosen whose neighboring faces are not all coplanar, exceptions can be avoided.

Label vectors can propagate within a certain mesh, and between different meshes across shared edges. Therefore, in most cases, only a single seed vertex is needed for classification. However, if there are more than two connected components among the tessellated meshes, extra seeds are required. The labels of these additional seeds have to be computed by point-in-polyhedron tests.

Exporting to float-point number. The vertices of the final mesh are represented as either planes or vertex coordinates. The vertices originating from the input meshes have exact coordinates, and the vertices newly introduced by the intersection between meshes have only P-reps, and require round-off when computing their coordinates. Although our method guarantees the correct topology in the final mesh, round-off errors can still cause topological deficiencies. This is called **vertex rounding** problem. We can adopt the method of Zhou et al. [8] to solve this problem iteratively. For the consideration of efficiency, we do not include this step during our comparison experiments.

7.5 Limitations and Future Work

We found that performance of our method was poor for CSGs that contained a lot of meshes within a small area (Table. 1, *T1*). In these cases, our method computes many intersections that will not appear as edges in the final mesh, leading to unnecessary tessellation. Optimization may be explored to alleviate this problem.

The inputs of our method are limited to solids. However, recent work [8] have proposed that the piecewise-constant integer *generalized winding number* (PWN) is a more powerful method to identify the insides and outsides of meshes. The input requirements of our method may be extended to PWN meshes, that allow self-intersection. We believe that this would be an interesting and valuable extension of our work.

8 Summary

In this paper, we propose a novel boolean method. Our method can efficiently perform variadic boolean operations, and is robust with solid inputs. The novel component of our approach is to combine P-reps with V-reps. P-reps allow us to strictly follow the principle of no geometry construction to avoid numerical errors ,and the V-reps enables coarse tests and fast neighborhood queries to reduce the need of slow plane-based computation. The experimental results show that the performance of our method is competitive with state-of-the-art non-robust methods, while guaranteeing robustness under consistent inputs.

Table 1
Computation time statistics of the evaluations of large CSGs (seconds)

No.	Model	Face Num.	Mesh Num.	CGAL [29]	Cork [42]	Carve [43]	QuickCSG [7]	Our Approach†				
								Total	Step 1	Step 2	Step 3	Step 4
1	Organic	219k	6	-	14.3	63.1	0.580	2.75	0.892	1.32	0.397	0.118
2	T1	80k	50	1.00k	18.5	10.4	0.388	14.4	0.691	2.71	8.11	2.87
3	T2	7k	50	2.81k	-	16.0	0.804	5.52	0.162	1.11	3.29	0.746
4	Sprocket	11k	52	211	-	4.26	(0.132)*	0.386	0.093	0.105	0.149	0.034
5	Ring & Ball	146k	801	-	-	187	(1.10)	20.0	1.04	3.55	8.61	6.68

† The *step 1, 2, 3, 4* are octree constrcution, intersection computation, tessellation and classification respectively.
* The bracket indicates that although QuickCSG gives the answer, the result meshes are full of topology deficiencies that contain thousands of boundary edges.

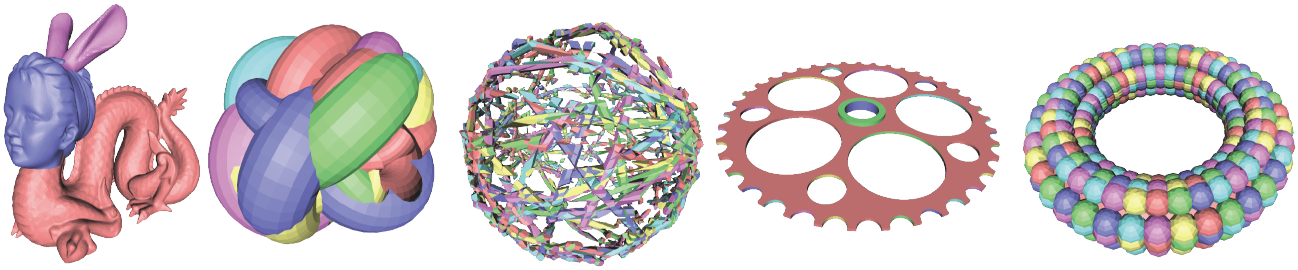


Figure 18. The result meshes of our method for variadic boolean evaluations. From the left to the right is Organic, T1, T2, Sprocket and Ring & Ball respectively. The performance of our method is better on cases with loosely distributed primitives (T2, Sprocket, Ring & Ball).

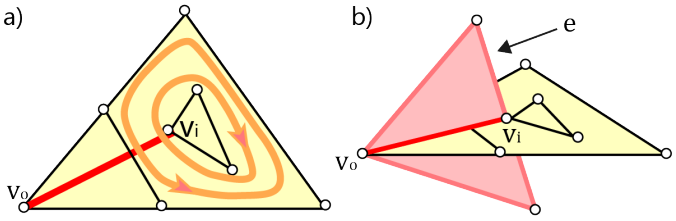


Figure 19. a) We construct an auxiliary connection to connect each inner component with outer component (the red line segment). By checking the intersections on the auxiliary connection, we find that the orange loops are identical. b) We construct the P-rep of the auxiliary connection by v_o and the exact coordinates of the end points of e.

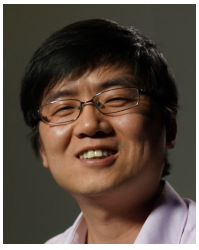
REFERENCES

[1] A. Requicha, "Mathematical models of rigid solid objects," 1977.
[2] R. Tilove and A. A. Requicha, "Closure of boolean operations on geometric entities," *Computer-Aided Design*, vol. 12, no. 5, pp. 219–220, 1980.
[3] C. C. Wang, "Approximate boolean operations on large polyhedral solids with partial mesh reconstruction," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 6, pp. 836–849, 2011.
[4] D. Pavić, M. Campen, and L. Kobbelt, "Hybrid booleans," in *Computer Graphics Forum*, vol. 29, no. 1. Wiley Online Library, 2010, pp. 75–87.
[5] H. Biermann, D. Kristjansson, and D. Zorin, "Approximate boolean operations on free-form solids," in *Siggraph*, vol. 1, 2001, pp. 185–194.
[6] C. Ogayar-Anguita, Á. García-Fernández, F. Feito-Higueruela, and R. Segura-Sánchez, "Deferred boundary evaluation of complex csg models," *Advances in Engineering Software*, vol. 85, pp. 51–60, 2015.
[7] M. Douze, J.-S. Franco, and B. Raffin, "Quickcsg: Arbitrary and faster boolean combinations of n solids," Ph.D. dissertation, Inria-Research Centre Grenoble–Rhône-Alpes, 2015.
[8] Q. Zhou, E. Grinspun, D. Zorin, and A. Jacobson, "Mesh arrangements for solid geometry," in *Tristate Workshop on Imaging and Graphics Posters*, 2016.
[9] H. Barki, G. Guennebaud, and S. Foufou, "Exact, robust, and efficient regularized booleans on general 3d meshes," *Computers & Mathematics with Applications*, vol. 70, no. 6, pp. 1235–1254, 2015.
[10] J. R. Shewchuk, "Lecture notes on geometric robustness," in *Interpola-tion, Conditioning, and Quality Measures. In Eleventh International Meshing Roundtable*, 1999.
[11] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes, "Constructive solid geometry for polyhedral objects," in *ACM SIGGRAPH computer graphics*, vol. 20, no. 4. ACM, 1986, pp. 161–170.
[12] F. R. Feito, C. J. Ogáyar, R. J. Segura, and M. Rivero, "Fast and accurate evaluation of regularized boolean operations on triangulated solids," *Computer-Aided Design*, vol. 45, no. 3, pp. 705–716, 2013.
[13] M. Segal, "Using tolerances to guarantee valid polyhedral modeling results," in *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4. ACM, 1990, pp. 105–114.
[14] G. Bernstein and D. Fussell, "Fast, exact, linear booleans," in *Computer Graphics Forum*, vol. 28, no. 5. Wiley Online Library, 2009, pp. 1269–1278.
[15] M. Campen and L. Kobbelt, "Exact and robust (self-) intersections for polygonal meshes," in *Computer Graphics Forum*, vol. 29, no. 2. Wiley Online Library, 2010, pp. 397–406.
[16] B. Naylor, J. Amanatides, and W. Thibault, "Merging bsp trees yields polyhedral set operations," in *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4. ACM, 1990, pp. 115–124.
[17] W. C. Thibault and B. F. Naylor, "Set operations on polyhedra using binary space partitioning trees," in *ACM SIGGRAPH computer graphics*, vol. 21, no. 4. ACM, 1987, pp. 153–162.
[18] K. Sugihara and M. Iri, "A solid modelling system free from topological inconsistency," *Journal of Information Processing*, vol. 12, no. 4, pp. 380–393, 1990.
[19] A. A. Requicha and H. B. Voelcker, "Boolean operations in solid modeling: Boundary evaluation and merging algorithms," *Proceedings of the IEEE*, vol. 73, no. 1, pp. 30–44, 1985.
[20] S. Xu and J. Keyser, "Fast and robust booleans on polyhedra," *Computer-Aided Design*, vol. 45, no. 2, pp. 529–534, 2013.
[21] A. Updegrove, N. M. Wilson, and S. C. Shadden, "Boolean and

smoothing of discrete polygonal surfaces," *Advances in Engineering Software*, vol. 95, pp. 16–27, 2016.
[22] R. P. Banerjee and J. R. Rossignac, "Topologically exact evaluation of polyhedra defined in csg with loose primitives," in *Computer Graphics Forum*, vol. 15, no. 4. Wiley Online Library, 1996, pp. 205–217.
[23] S. Fortune, "Polyhedral modelling with exact arithmetic," in *Proceedings of the third ACM symposium on Solid modeling and applications*. ACM, 1995, pp. 225–234.
[24] J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha, "Esolida system for exact boundary evaluation," *Computer-Aided Design*, vol. 36, no. 2, pp. 175–193, 2004.
[25] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel, "Boolean operations on 3d selective nef complexes: Data structure, algorithms, and implementation," in *Algorithms-ESA 2003*. Springer, 2003, pp. 654–666.
[26] P. Hachenberger and L. Kettner, "Boolean operations on 3d selective nef complexes: Optimized implementation and experiments," in *Proceedings of the 2005 ACM symposium on Solid and physical modeling*. ACM, 2005, pp. 163–174.
[27] S. Fang, B. Bruderlin, and X. Zhu, "Robustness in solid modelling: a tolerance-based intuitionistic approach," *Computer-Aided Design*, vol. 25, no. 9, pp. 567–576, 1993.
[28] C.-Y. Hu, N. M. Patrikalakis, and X. Ye, "Robust interval solid modelling," *Computer-Aided Design*, vol. 28, no. 10, pp. 807–817, 1996.
[29] P. Hachenberger and L. Kettner, "3D boolean operations on nef polyhedra," in *CGAL User and Reference Manual*, 4.7 ed. CGAL Editorial Board, 2015. [Online]. Available: http://doc.cgal.org/4.7/Manual/
[30] H. Bieri and W. Nef, "Elementary set operations with d-dimensional polyhedra," in *Workshop on Computational Geometry*. Springer, 1988, pp. 97–112.
[31] G. Varadhan, S. Krishnan, T. Sriram, and D. Manocha, "Topology preserving surface extraction using adaptive subdivision," in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. ACM, 2004, pp. 235–244.
[32] H. Zhao, C. C. Wang, Y. Chen, and X. Jin, "Parallel and efficient boolean on polygonal solids," *The Visual Computer*, vol. 27, no. 6-8, pp. 507–517, 2011.
[33] J. Hable and J. Rossignac, "Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 1024–1031.
[34] C. J. Ogayar, F. R. Feito, R. J. Segura, and M. Rivero, "Gpu-based evaluation of boolean operations on triangulated solids," 2006.
[35] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, 1997.
[36] T. Möller, "A fast triangle-triangle intersection test," *Journal of graphics tools*, vol. 2, no. 2, pp. 25–30, 1997.
[37] L. P. Chew, "Constrained delaunay triangulations," *Algorithmica*, vol. 4, no. 1-4, pp. 97–108, 1989.
[38] L. De Floriani and E. Puppo, "An on-line algorithm for constrained delaunay triangulation," *CVGIP: Graphical Models and Image Processing*, vol. 54, no. 4, pp. 290–300, 1992.
[39] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 1996, pp. 171–180.
[40] S. F. Frisken and R. N. Perry, "Simple and efficient traversal methods for quadtrees and octrees," *Journal of Graphics Tools*, vol. 7, no. 3, pp. 1–11, 2002.
[41] F. P. Preparata and M. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.
[42] G. Bernstein. (2013) Cork boolean library. [Online]. Available: https://github.com/gilbo/cork/
[43] T. Sargeant. (2011) Carve csg boolean library. [Online]. Available: https://github.com/VTREEM/Carve
[44] L. Kobbelt. (2010) Webbsp 0.3 beta. [Online]. Available: http://www.graphics.rwth-aachen.de/webbsp/
[45] Paolo Cignoni (p.cignoni@isti.cnr.it). (2015) Visualization and computer graphics library. [Online]. Available: http://vcg.isti.cnr.it/vcglib/
[46] C. J. Ogayar, R. J. Segura, and F. R. Feito, "Point in solid strategies," *Computers & Graphics*, vol. 29, no. 4, pp. 616–624, 2005.

Rui Wang is currently a postgraduate student at the Department of Computer Science and Technology, the Shanghai Jiao Tong University. His main research interests include real-time computer graphics and virtual reality applications.

Bin Sheng received his BS degree in computer science from Huazhong University of Science and Technology in 2004, MS degree in software engineering from University of Macau in 2007, and PhD Degree in computer science from The Chinese University of Hong Kong in 2011. He is currently an associate professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. His research interests include virtual reality, computer graphics, and image-based techniques.



Hongbu Fu is an Associate Professor in the School of Creative Media, City University of Hong Kong. He received the PhD degree in computer science from the Hong Kong University of Science and Technology in 2007 and the BS degree in information sciences from Peking University, China, in 2002. His primary research interests fall in the fields of computer graphics and human computer interaction. He has served as an associate editor of The Visual Computer, Computers & Graphics, and Computer Graphics Forum.

Li Ping received his Ph.D. from The Chinese University of Hong Kong. He is currently a Lecturer at The Education University of Hong Kong. His research interests include image/video stylization, learning analytics, big data visualization, and creative media.

Enhua Wu received the BS degree from Tsinghua University in 1970, and the PhD degree from the University of Manchester (UK) in 1984. He is currently a research professor at the Institute of Software, Chinese Academy of Sciences, and Fellow of China Computer Federation. He has also been teaching at the University of Macau since 1997, where he is currently an Emeritus Professor. His research interests include realistic image synthesis, virtual reality, and scientific visualization. He has served as an associate editor of The Visual Computer, Computer Animation and Virtual Worlds.