Efficient and Robust Plane-Based Boolean Operations on N Meshes

Abstract—Constructive solid geometry (CSG) is widely used in computer aided design and manufacturing. However, boolean operations, which compute the boundary representations of CSG, have suffered from robustness and efficiency problems for more than three decades. We propose a fast and exact boolean operation method which is unconditionally robust for regular set solids. Previously reported boolean methods using plane representations (P-reps) of meshes are robust, but inefficient. Conversely, methods based on boundary representations (B-reps) of meshes are fast but not robust, unless exact arithmetic is applied. Therefore, we choose to combine P-reps with B-reps, which allowed us to use the advantages of both. Our method uses B-reps globally, for efficiency, then P-rep geometry is applied locally to ensure robustness. Comparison experiments with state-of-the-art techniques show that our method is unconditionally exact and robust, and similarly efficient to 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 a series of regularized boolean operations [1], [2]: union, intersection, and difference. CSG can be converted into the widely-used boundary representation (B-rep) through boolean operations.

There are two major types of boolean algorithms, which vary according to how they deal with intersections between primitive meshes. Approximate methods [3], [4], [5] rediscretize the intersection areas, fit vertices approximatively, and rearrange the topology. Conversely, exact methods 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 information, such as face colors and materials.

However, there is always a compromise between the robustness and efficiency of exact boolean operation methods. Many exact boolean operation methods focus on efficiency. Robust exact methods require either the solutions to be arithmetically exact [6], [7] or trivial workarounds to avoid degenerations, such as setting a tolerance [8], [9] or rotating the primitive over small angles [10]. Methods using exact arithmetic are significantly slower, and workarounds are often case-dependent and unreliable. Recently, methods have been developed with unconditional robustness [11], [12] by avoiding geometric constructions (such as computing new vertex coordinates) and using only predicates. The key development is the use of plane-based representations (P-reps) of meshes, rather than boundary representations (B-reps). Sugihara and Iri [13] determined that boolean operations can be performed without geometric constructions if P-reps are used. These plane-based methods are generally faster than exact arithmetic ones, however, they still suffer from performance issues because of the complex conversions between different mesh representations. In addition, the incoherence between different representations means that extra steps are required to reconstruct geometric connectivity, leading to an additional computational burden and more complex topology.

Inspired by these previous studies, we have developed an unconditionally robust boolean operation method, which is as robust as existing P-rep methods, and as fast as non-robust B-rep methods. In our method, P-rep information is embedded into the faces of B-reps, forming hybrid representations of meshes. We avoid geometric constructions throughout our method, and convert all necessary computations into predicates. Generally, the B-rep information is used for coarse tests and efficient neighboring face queries, while P-rep information is used for computation of exact geometries. During intersection computation, we encode the intersections between meshes into sets of planes, then use these planes to determine exact tessellations. Subsequently, we classify each face in the tessellated meshes using local binary space partition (BSP) trees, which also compliment the plane-based representations to guarantee exactness. In addition, while most existing boolean operations can only process two meshes, our method can perform boolean operations with arbitrary meshes in a single call, instead of recursively implementing incremental boolean operations. Thus, our method reduces computation time by avoiding repetitive operations. Experiments have shown that our method is much faster than existing robust exact methods,

- *R. Wang and B. Sheng are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. E-mail:{jhcz,shengbin}@sjtu.edu.cn*
- *X. Jiang is with Autodesk China Research & Development Center. Email: xudong.jiang@autodesk.com*
- *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@ied.edu.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*

and only around two times slower than state-of-art non-robust exact methods.

2 Related Work

Boolean operations have been researched since their inception in the 1980s [14], [15]. Existing methods can be put into two categories: exact methods and approximate methods. Exact methods retain vertex positions, and the topology of the input meshes is preserved as much as possible. The coordinates of intersection points between input meshes are computed by input configurations, to construct the topology around each intersection. However, the coordinates cannot be represented exactly using fixed-precision floating point numbers, hence robustness is a significant problem. Conversely, approximate methods rediscretize the input mesh surfaces using techniques such as voxels, octrees, and Layered Depth Images (LDI). These methods have better performance and robustness than exact methods, but the loss of precision and geometry information is inevitable.

2.1 Exact Methods

Some exact methods [7], [8], [10], [16], [17], are optimized for efficiency, but cannot guarantee robustness in degenerate cases. Most of these methods are implemented by fixed-precision floating point arithmetic, so numerical errors are inevitable. Douze et al. [10] developed a very efficient method for handling very large meshes. In addition, their method can perform non-incremental boolean operations; evaluating the final mesh of CSG as a whole, instead of decomposing the CSG into a series of incremental boolean operations. However, this method can neither deal with coplanar cases, nor guarantee topological accuracy. Indeed, large CSGs are more complex, and thus more likely to contain degenerate cases, and more vulnerable to numerical errors. Efficiency alone is impractical.

Many researchers have attempted to solve the robustness issues of exact boolean operations using arbitrary precision arithmetic [18], [19], [20], [21], [22] and exact interval computation [9], [23], [24]. However, these methods are too expensive in terms of computation time and memory to be practical for large CSGs. For example, CGAL's [25] exact-arithmetic implementation [21] of Nef polyhedra [26] requires conversions between Nef polyhedra and input mesh data structures during evaluation and is more than 50 times slower than non-robust boolean operations [11].

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; better quality meshes require significantly higher resolutions. To accelerate this process, some methods reduce the complexity of the output mesh [27], whereas others preserve non-intersected areas of the input mesh to avoid redundant tessellation [3], [4], [28], [29], [30]. 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 operations. 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 polygonal meshes was first described by Sugihara and Iri [13]. P-reps are beneficial for boolean operations because no new geometric primitive must be constructed to obtain the results‡they can be determined from a subset of the planes from the input polyhedra. This means that the computation can include only geometric predicates, which are much easier to implement accurately and robustly. Berstein and Fussell [11] combined P-reps with binary space partition (BSP) trees [31], [32] to develop a boolean operation method which is unconditionally robust with consistent inputs. Unlike B-rep methods, this 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. This time complexity makes this method impractical for large meshes. Subsequently, Campen and Kobbelt [12] improved this method by localizing BSP operations using an octree. In this method, mesh refinements only take place locally, near intersections, which leads to a much faster method than Berstein and Fussells original.

3 Background and Overview

Our method is designed for boolean operations on an arbitrary number of regular triangular meshes [14]. Input meshes are required to be free of self-intersecting regions, and are represented by B-reps with geometric connectivity (for example, half-edge, or winged-edge). Our method is robust and not sensitive to topological deficiencies, such as holes and self-intersections, and works reliably if deficiencies are not near the regions where meshes intersect.

3.1 Boolean Numerical Errors

The boolean operation on triangular meshes is, in essence, a process of face selection. Namely, given a boolean function, the operation preserves those primitive faces that pass the function, and drops the others to generate the final mesh. Unfortunately, for B-rep meshes, not all of the input faces can be classified collectively, since only parts of the faces near the intersections belong to the final mesh. Therefore, an extra step is required to detect intersections between meshes and perform tessellations. Most of the existing boolean operation methods follow this two-step paradigm; as does our method.

The first step in our method is computing intersections. Input faces are tested in pairs to determine their intersections. Each of the input meshes are tessellated according to the intersection results, ensuring every face is completely inside, outside, or on the boundary of other primitives. The tessellated meshes are categorized as *intersection-free meshes*.

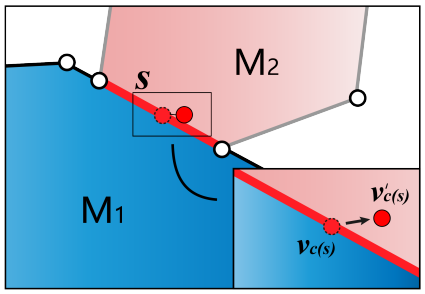


Fig. 1. In this 2D view, face $\boldsymbol{s}$ (the red line segment) of mesh M_2 is on the surface of mesh M_1. However, because the face barycenter $\boldsymbol{v}_{c(\boldsymbol{s})}$ is used to compute the indicator, the coordinates of which contain round-off errors, the point could be moved to $\boldsymbol{v}'_{c(\boldsymbol{s})}$. Thus, $\boldsymbol{s}$ may be falsely classified as being outside of M_1.

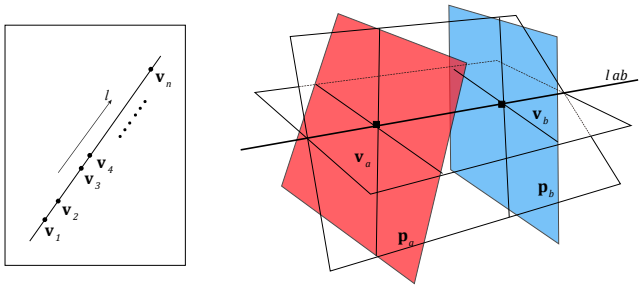


Fig. 2. Geometric configuration of the linear ordering of points. Points $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$ are both on line $\boldsymbol{l}_{ab}$. We convert this problem into the plane ordering of $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ along $\boldsymbol{l}_{ab}$.

Unfortunately, intersection tests are error prone when fixed-precision floating point arithmetic is used. First, degenerate intersections are hard to detect. Second, intersections introduce new vertices into the geometry, the coordinates of which cannot be exactly represented by fixed-precision floating point numbers.

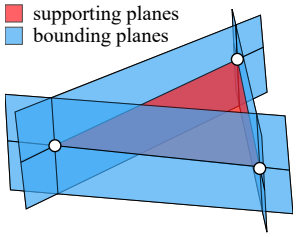
The second step is face classification. A given face $\boldsymbol{s}$ is evaluated to determine whether belongs to the final geometry according to the n-primitive boolean function f:

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

The parameter $\lambda_i(\boldsymbol{s})$ is the space indicator with respect to mesh M_i. To compute $\lambda_f(\boldsymbol{s})$ and classify $\boldsymbol{s}$, the space indicators with respect to all of the primitives ($\boldsymbol{\Lambda}(\boldsymbol{s})$) must be known. Each space indicator has four conditions: completely inside (*in*), completely outside (*out*), on the boundary with consistent normals (*same*) or with opposite normals (*oppo*). The rules of boolean function evaluation are described in [8], [10].

Face classification is also prone to numerical errors. Many methods classify a face according to the indicators of its barycenter using a point-in-polyhedron test [8], [12]. However, the coordinates of barycenters cannot be exactly represented, and can cause incorrect classifications (Fig. 1). This problem is often exacerbated by large inputs (faces and meshes), because many methods [4], [7], [8], [16] rely on the local coherence of the indicators, so classify neighboring faces together. This can improve performance, but can also propagate false classifications to neighboring faces and lead to broader failures.

3.2 Plane-based Geometry



In our method, we avoid numerical errors by embedding P-reps into conventional B-rep boolean algorithms and substituting constructions using B-reps with predicates using P-reps. 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 $\boldsymbol{p}_{s,sp} \cap \boldsymbol{p}^i_{s,b}$. Corner vertex $\boldsymbol{v}^i_s$ is represented by $\boldsymbol{p}_{s,sp} \cap \boldsymbol{p}^i_{s,b} \cap \boldsymbol{p}^{(i+1) \bmod n}_{s,b}$. We use the method of Campen et al. [12] for the exact conversion of triangles to their P-reps. All geometric predicates are computed using filtering techniques proposed by Shewchuk [33], for both exactness and efficiency.

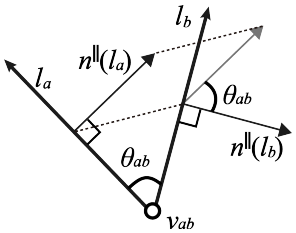
Other commonly used notations in this paper are presented here. The normal of a plane $\boldsymbol{p}$ is denoted as $\boldsymbol{n}(\boldsymbol{p})$. A line $\boldsymbol{l}$ can be represented by the intersection of two planes $(\boldsymbol{p}^0_l \cap \boldsymbol{p}^1_l)$, hence, $\boldsymbol{l}$: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

The plane-based geometric predicates used in our method have mostly been discussed thoroughly in previous work [11], [18]. In the following sections, we focus on two predicates related to sorting, which were specifically designed for our method.

Linear ordering of points

Given a line $\boldsymbol{l}_{ab}$ with two points on it, $\boldsymbol{v}_a$: $(\boldsymbol{p}^0_a \cap \boldsymbol{p}^1_a \cap \boldsymbol{p}^2_a)$, and $\boldsymbol{v}_b$: $(\boldsymbol{p}^0_b \cap \boldsymbol{p}^1_b \cap \boldsymbol{p}^2_b)$, we need to determine the linear order of the two points along $\boldsymbol{l}_{ab}$ (Fig. 2). To solve this problem, we choose one plane that is not parallel with $\boldsymbol{l}_{ab}$ from the plane representation 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. [18]. The chosen planes should have the same orientation with respect to $\boldsymbol{l}_{ab}$ (the dot product between the plane normal and $\boldsymbol{l}_{ab}$ must be positive); unqualified planes need to be flipped.

Circular ordering of lines



During face tessellation, we need to know which edges are neighbors (Fig. 3). This requires circular ordering of directed lines around a vertex. 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 $\boldsymbol{p}_0$ and their intersection point $\boldsymbol{v}_{ab}$, 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 $\boldsymbol{p}_0$. The sign of $\sin\theta_{ab}$ is the same as the triple product $\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{l}_a \times \boldsymbol{l}_b)$. However, directly computing this equation is complicated, because both $\boldsymbol{l}_a$ and

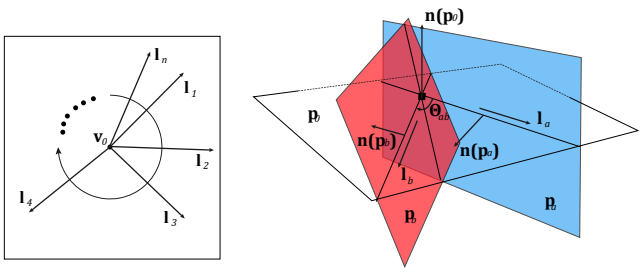


Fig. 3. Geometric configuration of the circular ordering of lines. $\boldsymbol{l}_a : (\boldsymbol{p}_0 \cap \boldsymbol{p}_a)$ and $\boldsymbol{l}_b : (\boldsymbol{p}_0 \cap \boldsymbol{p}_b)$ are within plane $\boldsymbol{p}_0$.

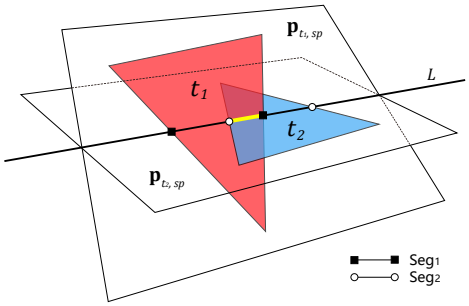


Fig. 4. Seg_1 is the intersection between $\boldsymbol{p}_{t_2,sp}$ and t_1. Seg_2 is the intersection between $\boldsymbol{p}_{t_1,sp}$ and t_2. The intersection between t_1 and t_2 (yellow line segment) is the overlap of Seg_1 and Seg_2.

$\boldsymbol{l}_b$ are implicitly represented by the intersection of planes.

We developed a simpler that requires no explicit computation of $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$. First, we write the P-reps of the two lines as $\boldsymbol{l}_a : (\boldsymbol{p}_0 \cap \boldsymbol{p}_a)$ and $\boldsymbol{l}_b : (\boldsymbol{p}_0 \cap \boldsymbol{p}_b)$. We then orthogonally decompose $\boldsymbol{n}(\boldsymbol{p}_a)$ and $\boldsymbol{n}(\boldsymbol{p}_b)$ along $\boldsymbol{n}(\boldsymbol{p}_0)$:

$$\begin{aligned} \boldsymbol{n}(\boldsymbol{p}_a) &= \boldsymbol{n}^{\parallel}(\boldsymbol{p}_a) + \boldsymbol{n}^{\perp}(\boldsymbol{p}_a) \\ \boldsymbol{n}(\boldsymbol{p}_b) &= \boldsymbol{n}^{\parallel}(\boldsymbol{p}_b) + \boldsymbol{n}^{\perp}(\boldsymbol{p}_b) \end{aligned} \tag{2}$$

Here superscript '$\parallel$' refers to the projection on $\boldsymbol{p}_0$ and '$\perp$' means orthogonal to $\boldsymbol{p}_0$. As we know that $\boldsymbol{n}^{\perp}(\boldsymbol{p}_a)$ and $\boldsymbol{n}^{\perp}(\boldsymbol{p}_b)$ are both parallel to $\boldsymbol{n}(\boldsymbol{p}_0)$, we get:

$$\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}(\boldsymbol{p}_a) \times \boldsymbol{n}(\boldsymbol{p}_b)) = \boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}^{\parallel}(\boldsymbol{p}_a) \times \boldsymbol{n}^{\parallel}(\boldsymbol{p}_b)). \tag{3}$$

By observation we find that the angle between $\boldsymbol{n}^{\parallel}(\boldsymbol{p}_a)$ and $\boldsymbol{n}^{\parallel}(\boldsymbol{p}_b)$ is exactly θ_{ab}. Therefore the sign of the right side of equation 3 is exactly the sign of $\sin\theta_{ab}$. Thus, we come to the following conclusion:

$$sign(\sin\theta_{ab}) = sign(\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{n}(\boldsymbol{p}_a) \times \boldsymbol{n}(\boldsymbol{p}_b))) \tag{4}$$

As the three vectors on the right side are already explicitly represented, we only need to evaluate the sign of a 3×3 matrix, which avoids complicated arbitrary precision floating point arithmetic.

3.3 Method Overview

There are three steps to our method. The first step is to compute the intersections between each triangle face pair. In the second step, the input meshes are tessellated into intersection-free meshes according to these intersections. Lastly, each face is classified and the final mesh is generated.

3.3.1 *Intersection computation*

In this step, we only compute the intersections between pairs of triangles. The triangle-triangle intersection test is largely based on Möllers algorithm [34], because of its efficiency. However, the conventional implementation of Möllers algorithm using vertex-based geometry introduces numerical errors. Therefore, we integrate P-reps into Möllers algorithm. All intersection points and line segments are represented using planes, to avoid explicit computation of the coordinates. In addition, we carefully deal with degenerate cases of triangle intersections, including point intersections, edge intersections, and coplanar situations. 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, it is necessary to tessellate the input meshes and construct the intersection-free meshes. In many existing methods, such as [16], constrained Delaunay triangulation (CDT) is used to perform tessellations. However, in a CSG with more than two meshes, intersections may overlap or intersect with each other, so cannot be used directly as CDT constraints. In addition, as intersections are represented in our method by planes, implementing CDT would be complex and inefficient, as most CDT algorithms (such as those in [35], [36]) are not designed to handle planes and instead require explicit coordinates. Therefore, we first perform intersection refinement to resolve overlapping intersections. We then construct a graph-like structure suitable for P-rep intersections, called a *tess-graph*, to guide the exact tessellation of each face. Once all of the faces are tessellated, we obtain intersection-free meshes. Details are given in §5.

3.3.3 *Face classification*

The purpose of this step is to choose faces from the intersection-free meshes to generate the final mesh. However, literally computing the indicator vector of each face is unacceptably slow for large CSGs. We utilize the geometric connectivity information stored in B-reps to propagate indicators in a flood-filling manner. In addition, many methods use the barycenters of faces together with a point-in-polyhedra test for face classification, which is neither exact nor robust with fixed-precision floating point arithmetic. However, our method classifies faces based on the classification of exactly represented vertices, including input vertices and novel intersection vertices, and carefully deals with coplanar conditions to ensure topological consistency. Details are given in §6.

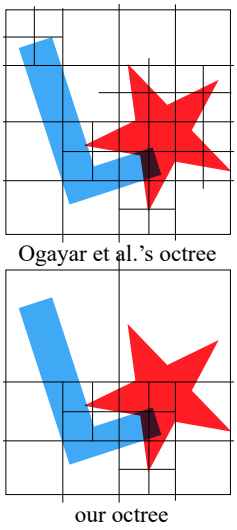
4 Intersection Computation

In this step, the intersections between faces are determined by a triangle-triangle intersection test. We use Möllers algorithm [34], because its efficiency and simplicity. However, a naïve implementation of Möllers algorithm can introduce numerical errors, and may fail to produce the correct results. Therefore, we integrate plane-based geometry to ensure the implementation is exact. In the following sections, we introduce our space division strategy to reduce the number of intersection tests. Then we make a quick review of

Möllers algorithm, and discuss how to embed plane-based geometry. Lastly, we discuss how to deal with degenerate cases.

4.1 Space Division

As intersection detection is performed between each pair of faces, localization is necessary for large CSGs to reduce the number of pairs to be tested. We use an adaptive octree for this purpose. Our implementation is akin to the implementation of Ogayar et al. [16]. The intersections between triangle faces and octree nodes are efficiently detected using the separating axis theorem [37]. 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 following intersection computation is 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 [38], which is seldom used in our method. This simplification saves significant computing time, especially when intersections between primitives are simple, and located in small regions.

4.2 Möller's Vertex-Based Method

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 $\boldsymbol{p}_{t_1,sp}$.
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 operations with large CSGs. We use plane-based geometry to solve this problem, by implicitly representing intersections with planes.

4.3 Plane-Based Intersection

Given a triangle face t, we first convert it to its P-rep: a supporting plane $\boldsymbol{p}_{t,sp}$ surrounded by three bounding planes $\{\boldsymbol{p}^i_{t,b} | \ i = 0,1,2\}$. We integrate the plane-based geometry into each of the three steps of Möllers algorithm as follows.

1) The first step is to compute the orientation of a face with respect to the vertices of its paired face. As the exact coordinates of the vertices of each triangle are known, we can compute the point-plane orientation using vertex-based geometric predicates. This allows us to perform this early rejection efficiently.

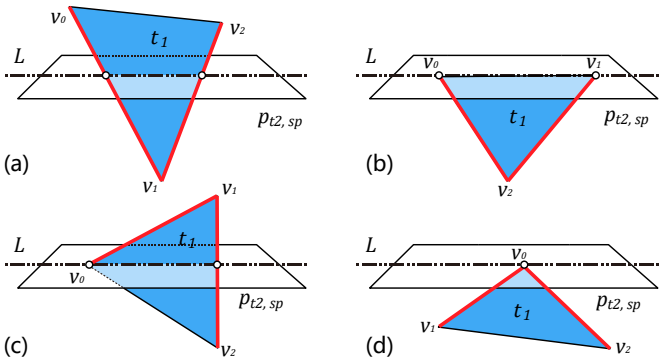


Fig. 5. We denote the signed distance from point v_i to plane $\boldsymbol{p}_{t_2,sp}$ as d_i. The four conditions of intersection between t_1 and $\boldsymbol{p}_{t_2,sp}$ are: (a) $d_0 \cdot d_2 < 0, d_1 \cdot d_2 < 0$; (b) $d_0 = 0, d_1 = 0, d_2 \neq 0$; (c) $d_0 = 0, d_1 \cdot d_2 < 0$; (d) $d_0 = 0, d_1 \cdot d_2 > 0$. End points of Seg_1 are the intersections between $\boldsymbol{p}_{t_2,sp}$ and the related edges of t_1 (bold red lines).

2) If t_1 and t_2 are not coplanar, the end points of Seg_1 and Seg_2 can be implicitly represented by plane triples. For example, Seg_1 is the intersection between t_1 and $\boldsymbol{p}_{t_2,sp}$ and the end points of Seg_1 are the intersections between $\boldsymbol{p}_{t_2,sp}$ and the edges of t_1 (denoted as $\boldsymbol{e}^i_{t_1}, i \in \{0,1,2\}$). Because $\boldsymbol{e}^i_{t_1}$ can be represented as $\boldsymbol{p}_{t_1,s} \cap \boldsymbol{p}^i_{t_1,b}$, the end points of Seg_1 can be represented in the form $\boldsymbol{p}_{t_2,sp} \cap \boldsymbol{p}_{t_1,sp} \cap \boldsymbol{p}^i_{t_1,b}$. Fig. 5 shows all of the possible intersections between t_1 and $\boldsymbol{p}_{t_2,sp}$ and the corresponding $\boldsymbol{e}^i_{t_1}$. The coplanar situation is discussed in §4.5.
3) The intersection between t_1 and t_2 is the overlap between of Seg_1 and Seg_2. It can be computed by sorting the end points of Seg_1 and Seg_2 along the line $\boldsymbol{l}$: $\boldsymbol{p}_{t_1,sp} \cap \boldsymbol{p}_{t_2,sp}$. Because end points are all represented using plane triples, we can use the linear order of points, discussed in §3.2, to sort them.

Each time an intersection is computed, two intersections are generated (one for each triangle), and the newly generated vertices are added into their respective meshes. To avoid repetitive vertices in the final result, we perform vertex repetition elimination in this step; merging coincident vertices even if they come from different input meshes. Because P-reps are used, the comparisons of the vertices are exact.

4.4 Plane-Based Intersection Representation

In our method, the intersection with triangle t is stored as $\mathfrak{I}$: $\{T, \boldsymbol{P}_{ext}, (\boldsymbol{P}_0, \boldsymbol{P}_1), \mathcal{N}\}$. This is the plane-based intersection representation (PBI-rep, see Fig. 6). The first component, T, indicates which triangle $\mathfrak{I}$ lies on. $\boldsymbol{P}_{ext}$ indicates that $\mathfrak{I}$ lies on $\boldsymbol{P}_{ext}$, which also means that it lies on the line $\boldsymbol{p}_{t,sp} \cap \boldsymbol{P}_{ext}$. The two end points of $\mathfrak{I}$ are $\boldsymbol{p}_{t,sp} \cap \boldsymbol{P}_{ext} \cap \boldsymbol{P}_0$ and $\boldsymbol{p}_{t,sp} \cap \boldsymbol{P}_{ext} \cap \boldsymbol{P}_1$. The last component, $\mathcal{N}$, represents the neighborhood of the intersection $\mathfrak{I}$. It indicates that $\mathfrak{I}$ is the intersection of triangle t with $\mathcal{N}$ from the other mesh, where $\mathcal{N}$ can be either a face, or a set of faces that share the same edges.

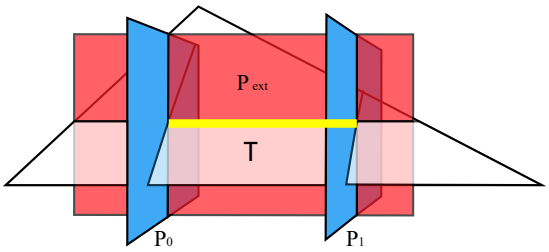


Fig. 6. The geometry of planes in a PBI-rep. The yellow line segment is the intersection represented by $\mathfrak{I}$.

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 $\mathfrak{I}_{12}$ on t_1, $T = t_1$ and $\boldsymbol{P}_{ext} = \boldsymbol{p}_{t_2,sp}$. The third component are the boundary planes of t_2, as discussed in §4.3. The last component is $\mathcal{N} = t_2$. ~~Sometimes, intersections may lie on the mesh edges instead of their faces (Fig. 7). This is called edge intersection. In this case, the second component is a set of adjacent faces. Edge intersection is discussed in §4.5.~~

4.5 Handling Degenerate Case

In most situations, two intersecting triangles intersect on a line segment. However, they can also intersect on a point or a convex area (coplanar case). Even if the intersection is a line segment, the intersection can be on an edge. These degenerate situations prevent us from performing robust boolean operations. In this section, we demonstrate a simple but effective way of dealing with all of these degenerations, which conceals the complexity of intersections, and simplifies later processing.

4.5.1 Point intersection

If two triangles intersect at a single point (such as in Fig. 5d) then the intersection cannot be represented using our four-component description, because the line segment collapses into a single point. In these cases, we simply add the intersection point into the related triangles, which guarantees correct tessellation. No further intersections are introduced.

4.5.2 Edge intersection

When an intersecting line segment lies on the edge of a face, we refer to it as *edge intersection*. The space near the edge intersection is divided by the faces around the intersected edge (typically two faces for a manifold edge), instead of by just one face. Therefore, the neighborhood of the corresponding intersection is a set of faces rather than a single face. For example, in Fig. 7, the neighborhood of the intersection on t_1 is $\{t_2, t_2^*\}$.

Edge intersections will be repeatedly detected. For example, in Fig. 7a, the same intersection on t_1 will be detected twice, because t_1 intersects both t_2 and t_2^* at the same intersection. We solve this duplication~~, together with similar interactions between different intersections~~ in §5. We regard t_2 and t_2^* as *companion triangles*, because they share the same edge intersection. This concept is referred to again in the discussion of coplanar cases.

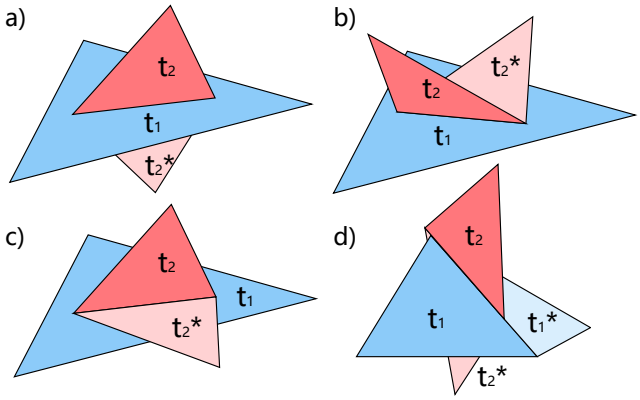


Fig. 7. Different conditions of edge intersection. Triangle faces t_1 and t_1^* are companion faces. Triangle faces t_2 and t_2^* are companion faces. a) t_2 and t_2^* are on different sides of t_1. b) t_2 and t_2^* are on the same side of t_1. c) t_2^* is coplanar with t_1. d) Both t_1 and t_2 have 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).

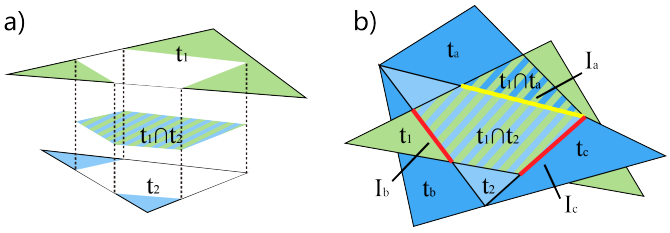


Fig. 8. a) Coplanar cases; t_1 and t_2 intersect in 2D, dividing each other into two areas—a convex overlapping area and exclusive areas. b) Possible configurations of the companion faces. The blue triangles originate from the same mesh. For $\mathfrak{I}_a$, the companion triangle t_a is coplanar with t_1. Therefore, the overlapping areas ($t_1 \cap t_a$ and $t_1 \cap t_2$) do not need to be split by the yellow edge during tessellation, thus $\mathfrak{I}_a$ is invalid. For $\mathfrak{I}_b$ and $\mathfrak{I}_c$, the companion faces are not coplanar, so these intersections can be recorded during an intersection test between t_1 and t_a or t_1 and t_b. There is no need to record the intersection between t_1 and t_2.

4.5.3 Copalnar

Consider two triangle faces, ~~$t_1 \subset M_x$ and $t_2 \subset M_y$~~, which intersect within a common plane. Both t_1 and t_2 divide each other into two ~~areasla~~ convex overlapping area and exclusive areas (Fig. 8a). If we tessellate both t_1 and t_2 according to the boundary of the overlapping area, we can guarantee that the tessellated meshes are intersection-free. Many previous methods [7], [8] use this process to guarantee topological correctness. However, in our method, we do not test whether t_1 and t_2 really intersect in 2D once we determine that they are coplanar. We treat coplanar situations as if they do not intersect at all. In this way, we simplify our method, making it faster and more robust, while having no effect on the topological correctness.

We find that each intersecting line segment is part of the edges of the input meshes in 2D. That means that we can view 2D intersections as special cases of edge intersections. However, there can be up to three edge intersections in one 2D intersection (red and yellow line segments in Fig. 8b). As discussed, edge intersections will be detected twice. Hence, we can rely on the faces of the companion triangles to detect intersections.

However, if the companion triangles are also coplanar, neither of the two triangles will record the intersection. Fortunately, in this case the intersection is not valid. In this instance, valid means that it is necessary to tessellate the intersections, so that the intersections appear as an edge in the final mesh. Supposing the intersection $\mathcal{I}$ is on t_1, in which case:

$$\lambda_f(\boldsymbol{v}_i) \neq constant, \boldsymbol{v}_i \in \boldsymbol{U}(\mathcal{I}) \cap M_x, \tag{5}$$

where f is the CSG function, and $\boldsymbol{U}(\mathcal{I})$ is the neighborhood of $\mathcal{I}$ in $\mathbb{R}^3$. This inequation indicates that the spaces near $\mathcal{I}$ on mesh M_x have different classifications and must be split according to $\mathcal{I}$. Frequently, however, the validation of the intersection is not known until the classification stage, because it is necessary to know the indicator of each point in $\boldsymbol{U}(\mathcal{I}) \cap M_x$. Fortunately, if the companion face of t_2 is coplanar with t_2, both sides of $\mathcal{I}$ on t_1 have the same indicator $\lambda_y = on$ and thus the same λ_f. It also means that $\mathcal{I}$ is invalid and we do not need to record it.

5 Deferred Tessellation

After intersection computation, tessellation is performed on each intersecting triangle face to generate the intersection-free meshes. This step is referred to as deferred tessellation, because the tessellation happens after all of the valid intersections are computed, instead of clipping triangles incrementally at each intersection.

Ogayar-Anguita et al. [16] used CDT to perform deferred tessellation. They treated triangle faces as convex triangulation zones, and intersections as constraints. We first attempted to implement robust plane-based CDT. However, CDT algorithms [35], [39] require the projection of coordinates, or intersection detection between arbitrary connections of vertices within the triangulation zone; both of which are difficult to implement using P-reps. Additionally, CDT produces geometric constructionsłnew edges are generated to guarantee that each subface is a triangle (see Fig. 9d)łwhich we preferred not to use in our method. Moreover, when there are more than two primitives, applying CDT ignores the complexity of the intersections between meshes. For instance, the intersections may intersect each other, introducing new vertices and splitting the original intersecting line segments (see Fig. 9a). This breaks the assumptions of most CDT algorithms, and may lead to incorrect output topologies.

In our method we first perform intersection refinement to eliminate any crossing or overlapping between intersections. We then perform conservative tessellation based on a *tess-graph*, ensuring unconditional topologic correctness. A tess-graph is a graph-like description of the intersections on a given face. The word 'conservative' in this instance means that we do not add any new edges during tessellation, which does not guarantee that the subfaces are all triangular.

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 alone are not sufficient to determine all of the intersection vertices.

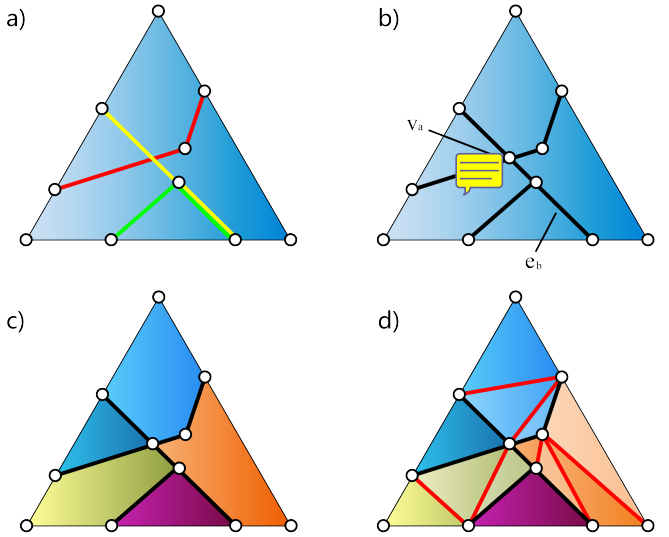


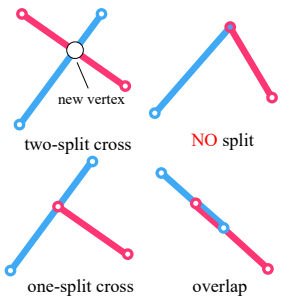
Fig. 9. a) Different colors indicate that the intersections originate from different meshes. The yellow and red intersections intersect at a point. The yellow intersection overlaps with the green intersection. b) After refinement, we introduce a new vertex $\boldsymbol{v}_a$, and merge overlapping intersections 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 precise plane-based representations cannot be constructed for these new edges.

Thus, it is necessary to refine these triangle-triangle intersections before the final tessellation.

Intersection refinement is performed in locally. For each intersecting face t, we collect all of the intersections on t as a set $\boldsymbol{\Gamma}(t)$ and refine them as a set. We also include the three edges of t in $\boldsymbol{\Gamma}(t)$, because the edges also participate in tessellation. In $\boldsymbol{\Gamma}(t)$, edges are represented as PBIs. The neighborhood $\mathcal{N}$ of the edges are set as N/A because they do not have a neighborhood. 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 neighborhood component of the PBI-rep. The new neighborhood component is the union of the original intersections components, meaning that the merged intersection has two (or more) neighborhoods.

Cross and overlap resolving After the two previous steps, there are no overlapping intersections remaining in $\boldsymbol{\Gamma}(t)$. We then check whether any pair of intersections cross or overlap (are collinear to) each other. New vertices can be introduced in this step. The definition of *cross* does not include the situation where two intersec-



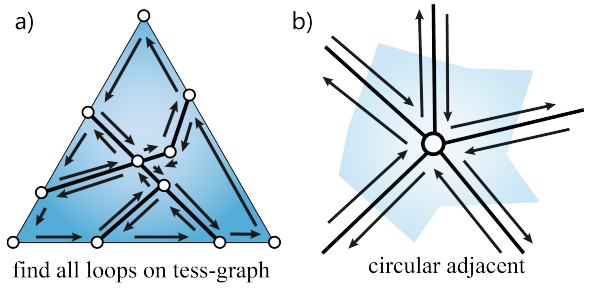


Fig. 10. a) To tessellate a triangle, we find all of the valid loops on a tess-graph. The direction of the loops must be coherent with the triangle normal (assuming here that the normal points to the outside of the paper). b) Each circular adjacent edge pair is an angle of the tessellated polygon.

tions share a single end point because no splitting is required in this situation. When there is crossing or overlapping, at least one of the intersections has to be split. 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 subroutine for determining the linear order of the points (§3.2) is used to sort these splitting points along the splitting intersection. The PBI-reps of the split segments inherit that of their fathers, except for $\boldsymbol{P}_0, \boldsymbol{P}_1$, which identifies the P-reps of the end points*.

Coincidence elimination (revisited) Unfortunately, resolving the overlap may produce new coincident intersections. Therefore, the first step of the process is repeated in the final step.

5.2 Tessellation by Tess-Graph

Intersections are independent data that store coordinates and neighborhood information. To perform tessellation and extract subfaces, we need to organize them to determine topology. We use a tess-graph for this purpose. A tess-graph is the graph description of the tessellated face topology. For each face to be tessellated, we construct a tess-graph from the refined set of intersections. Nodes of the tess-graph represent the end points of intersections, and nondirectional 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 tessellate the face and extract subfaces from it. Subfaces are extracted from valid loops on the tess-graph. A valid loop must 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 (Fig. 10). To facilitate later processes, we also store neighborhood information in the edges of the new faces. The neighborhood information is determined from the corresponding intersections. When all faces are tessellated, the corresponding meshes are intersection-free.

6 Face Classification

We traverse each face of the intersection-free meshes, and determine which belong to the final mesh. The faces are classified by evaluating the indicator vector $\boldsymbol{\Lambda}$ of each face. Our classification method utilizes the space coherence of the face indicator vectors to share the classification results among neighboring faces. Because we store the neighborhood information in the edges which intersect other meshes, the search space of the point-in-polyhedron test is limited. Hence, the indicator computation is very fast. Finally, because P-reps are used, our classification method is exact, and can deal with any degenerate cases.

The space coherence of indicator vectors means that neighboring faces may have the same indicator vector, or most of the components of the indicator vectors. Therefore, we start from a seed face $\boldsymbol{s}_0$ with a known indicator vector $\boldsymbol{\Lambda}(\boldsymbol{s}_0)$, and propagate the indicator to other faces in a flood-filling manner. The trace of the indicator propagation is:

$$\boldsymbol{s}_0 \rightarrow \boldsymbol{s}_1 \rightarrow \boldsymbol{s}_2 \rightarrow \boldsymbol{s}_3 \rightarrow \ldots \tag{6}$$

Intersection-free meshes facilitate this process by accelerating the indicator propagation. From adjacent faces $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, it is straightforward to determine whether $\boldsymbol{\Lambda}(\boldsymbol{s}_1)$ and $\boldsymbol{\Lambda}(\boldsymbol{s}_2)$ are different. The two indicator vectors are the same unless the shared edge $\boldsymbol{e}_{12}$ lies on the boundary of another mesh. In that cases, neighborhood information, say, ~~$\mathfrak{N}(\boldsymbol{e}_{12}) = \{\mathfrak{N}_k\}$~~, is stored on $\boldsymbol{e}_{12}$. Here we use $\mathfrak{N}$ to indicate that the edge may have more than one neighborhood (see §5.1). The neighborhood information indicates which components of the indicator vector differ between $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$. If there is neighborhood information from mesh M_x, the x^{th} indicators $\lambda_x(\boldsymbol{s}_1)$ and $\lambda_x(\boldsymbol{s}_2)$ differ, and can be computed efficiently and exactly with respect to that neighborhood. We outline our classification method in Algorithm 1.

Algorithm 1 Fast Face Classification

Input: Intersection-free hybrid meshes, boolean function f
Output: Classification $f(\boldsymbol{\Lambda}(\boldsymbol{s}_i))$ for all faces $\boldsymbol{s}_i$

```
Select a proper seed face s_0
Compute the seed indicator 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
```

6.1 Indicator Propagation

~~Meshes consist of vertices, edges and faces. It is necessary to perform classification at the face level.~~ However, polyhedron-in-out tests are performed on points, where

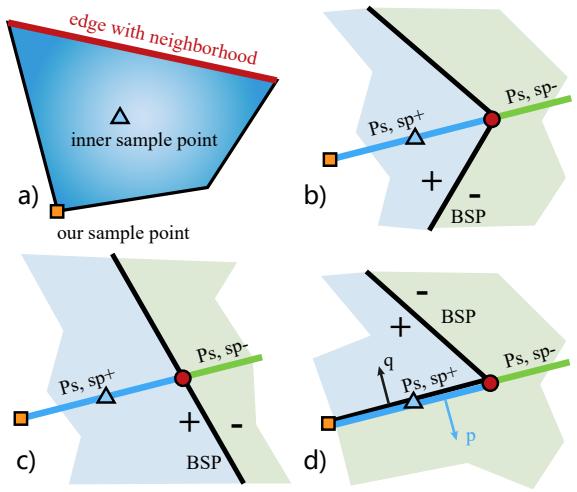
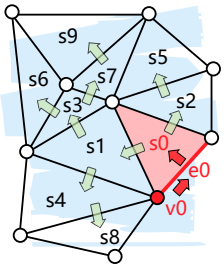


Fig. 11. a) Because the inner sample point includes geometric constructions, we choose the face vertex (orange) instead to compute the face indicator. b-d) Different types of classification. The simple structure of the neighborhood BSP guarantees the same classification result for all of the points on $\boldsymbol{p}^{+}_{\boldsymbol{s},sp}$. 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 indicator should be $oppo$ because p and q are opposite.

gaps exist between them. Conventionally, the ~~face barycenter~~ is used to compute the indicator of the whole face because in intersection-free meshes, the face is classified as a whole, and every inner point of the face will have the same indicator as the whole face. However, the computation of inner point coordinates requires geometric constructions, which introduce errors unless exact arithmetic can be used. Therefore, we use the face (or subface) vertices instead, the exact representations of which are known, for indicator computation.

However, there are two discrepancies between the face and face vertex indicators. First, the vertex indicator is not always equal to the face indicator. Second, the face indicator has two extra classes in the on case$same$ and $oppo$because the face has an orientation.



To deal with these discrepancies, the indicator trace (6) is modified for finer granularity. We start from a seed vertex $\boldsymbol{v}_0$ on a seed face $\boldsymbol{s}_0$. We assume that the indicator vector $\boldsymbol{\Lambda}(\boldsymbol{v}_0)$ is known. Then we use $\boldsymbol{\Lambda}(\boldsymbol{v}_0)$ to compute $\boldsymbol{\Lambda}(\boldsymbol{s}_0)$. In addition, we add an optional edge layer into the trace:

$$\boldsymbol{v}_0 \rightarrow \boldsymbol{e}_0 \rightarrow \boldsymbol{s}_0 \rightarrow \cancel{(\boldsymbol{e}_1)} \rightarrow \boldsymbol{s}_1 \rightarrow \cancel{(\boldsymbol{e}_2)} \rightarrow \boldsymbol{s}_2 \rightarrow \ldots, \quad (7)$$

where $\boldsymbol{v}$ and $\boldsymbol{e}$ are the vertex and edge, respectively. The bracket indicates that propagating to the edge is optional, because if the edge does not contain neighborhood information the indicators on either side are the same, so we do not need to propagate to the edge. There are three basic operations in the trace: $\boldsymbol{v_e} \rightarrow \boldsymbol{e}$, $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$ and $\boldsymbol{s} \rightarrow \boldsymbol{e_s}$, where $\boldsymbol{v_e}$ is the end point of $\boldsymbol{e}$, and $\boldsymbol{e_s}$ is the edge of $\boldsymbol{s}$.

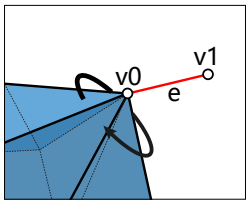
Given the partial orders $on \succ in$ and $on \succ out$, the following relationship is true for indicators within intersection-free meshes:

$$\cancel{\lambda_k(\boldsymbol{v_{e_s}}) \succeq \lambda_k(\boldsymbol{e_s}) \succeq \lambda_k(\boldsymbol{s})}, \quad (8)$$

where $\lambda_k(x)$ is the indicator of x for a certain primitive M_k. This relationship can be inferred from the continuous space assumption and the definition of intersection-free mesh*. It indicates that when we perform $\boldsymbol{v_e} \rightarrow \boldsymbol{e}$ and $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$, from a low-dimension to a high-dimension, we decide whether the on indicator changes to in or out, or remains on. When we perform ~~$\boldsymbol{s} \rightarrow \boldsymbol{e_s}$~~, we decide whether in and out indicators are changed to on. In addition, when considering the orientation of the face, we also need to determine whether the on indicators change to $same$ or $oppo$ during the $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$ operation.

$\boldsymbol{s} \rightarrow \boldsymbol{e_s}$ We need to determine which indicators change to on. We know that when $\lambda_k(\boldsymbol{e_s}) = on$, there will be neighborhood information related to mesh M_k stored on $\boldsymbol{e_s}$. Therefore, in this operation, we need to traverse all of the neighborhoods of $\boldsymbol{e_s}$ and change the corresponding indicators to on.

$\boldsymbol{e_s} \rightarrow \boldsymbol{s}$ From eq. 8, we know that if $\lambda_k(\boldsymbol{e_s}) \neq on$, then $\lambda_k(\boldsymbol{e_s}) = \lambda_k(\boldsymbol{s})$. Conversely, when $\lambda_k(\boldsymbol{e_s}) = on$, $\lambda_k(\boldsymbol{s})$ can be determined using neighborhood information related to M_k. The neighborhood of M_k can be either a face or a set of faces from the mesh M_k. We can build a trivial BSP [31] using these faces. The BSP can be used to compute $\lambda_k(\boldsymbol{s})$ if a point can be sampled from $\boldsymbol{s} \cap \boldsymbol{U}(\boldsymbol{e_s})$, where $\boldsymbol{U}(\boldsymbol{e_s})$ is the neighborhood space of $\boldsymbol{e_s}$. Unfortunately, we cannot guarantee that such a point can be found with precise coordinates.



vertex neighborhood

Our BSP is constructed from a single face, or a set of faces sharing the same edge. This BSP is sufficiently simple that the indicators λ_k of all of the points on the half plane $\boldsymbol{p}^{+}_{\boldsymbol{s},sp}$ are identical. Here $\boldsymbol{p}^{+}_{\boldsymbol{s},sp}$ is half of the supporting plane of $\boldsymbol{s}$ that lies on the positive side of the bounding plane $\boldsymbol{p}^{\boldsymbol{e_s}}_{\boldsymbol{s},b}$ (Fig. 11). Since $\boldsymbol{p}^{+}_{\boldsymbol{s},sp} \cap \boldsymbol{s} \neq \varnothing$, we can compute $\lambda_k(\boldsymbol{s})$ by determining $\lambda_k(\boldsymbol{v}_x, \boldsymbol{v}_x \in \boldsymbol{p}^{+}_{\boldsymbol{s},sp})$. For polygons, at least one corner point lies on $\boldsymbol{p}^{+}_{\boldsymbol{s},sp}$, and the exact coordinates of the corner point are known. That corner point is used as $\boldsymbol{v}_x$. In addition, we assign each BSP splitting plane a normal from 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 neighborhood information related to M_k on $\boldsymbol{e}$. If not, we need to find where $\boldsymbol{v_e}$ lies on M_k. This is straightforward because the intersection-free meshes contain geometric connectivity information. We can consider the connectivity information as the vertex version of neighborhood information, and the same BSP-based classification method is used in $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$ can be used to determine the high dimension indicator $\lambda_k(\boldsymbol{e})$.

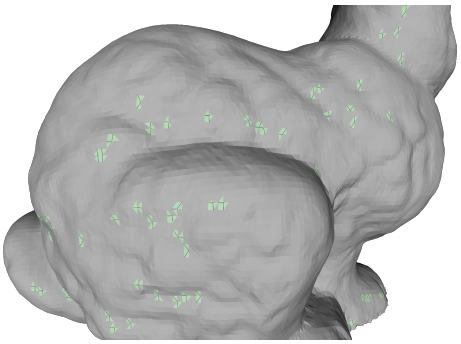


Fig. 12. After perturbation, the self-union of *Bunny* with QuickCSG still suffers from topology problem. The green boundary faces indicate topological deficiencies.

We use the other end point $\boldsymbol{v}'_{\boldsymbol{e}}$ as the sample point for the BSP classification.

6.2 Acceleration by Caching

If the CSG tree is large, with hundreds of primitive nodes, computing $\lambda_f(\boldsymbol{s}_i)$ from $\boldsymbol{\Lambda}(\boldsymbol{s}_i)$ for every face $(\boldsymbol{s}_i)$ can be costly. However, using indicator space coherence, we can reduce the computation time by caching the evaluation results during flood-filling.

The basic caching strategy is to cache the final result, that is, the value of $\lambda_f(\boldsymbol{s}_i)$. We know that the indicator vector will not change if there is no intersection edge encountered during propagation. Therefore, the final classification $\lambda_f(\boldsymbol{s}_i)$ will not change either. Thus, faces which share the same indicator vector can be classified as a set.

We can also perform an intermediate results cache. The boolean expression can be simplified if some components of $\boldsymbol{\Lambda}(\boldsymbol{s}_i)$) are fixed. For example, assume we have a boolean expression $f = M_1 \cup (M_2 \cap M_3 - M_4)$. Given the values of two indicators $\lambda_1(\boldsymbol{s}_i) = out$, $\lambda_2(\boldsymbol{s}_i) = in$, 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 given mesh often intersects with only a few other meshes $\Theta = \{M_{n_1}, M_{n_2}, \cdots, M_{n_x}\}$. Thus, all of the faces in this mesh have the same indicators for $M_i \notin \Theta(M_k)$. Therefore, if we can determine the fixed indicators and simplify the boolean function, then we compute the final indicator for each face in this mesh.

7 Results and Discussion

We implemented our proposed method in C++, and tested a series of models on a laptop with an Intel Core i5 1.5 GHz CPU and 8 GB of RAM. To validate the efficiency and robustness of our method, we performed boolean evaluations on various CSGs with different complexities. We also compared our method with several existing methods, including CGAL [25], Maya [6], [40], Cork [41], QuickCSG [10], Carve [42], and online implementations of Campen and Kobbelts plane-based method [12], [43].

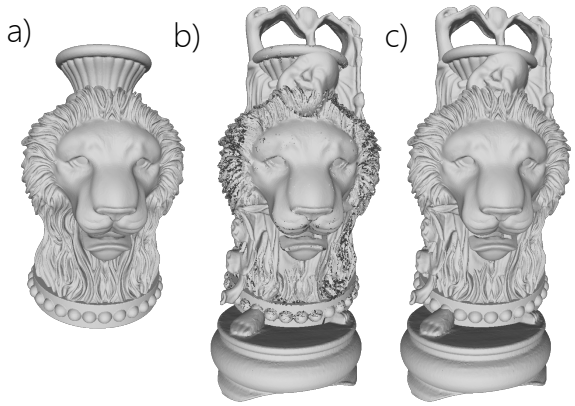


Fig. 13. Results from $Budda \cup Lion$: a) incorrect result using CGAL, b) incorrect result using Cork, and c) correct result using our method.

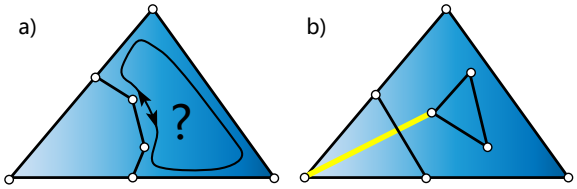


Fig. 14. a) There is ambiguity as to the orientation of the loop in this tess-graph. b) When there is more than one connected component in a tess-graph, auxiliary intersections (yellow line) are used to connect the components.

7.1 Robustness & Performance

Self-union Our method is unconditionally robust for regular set mesh inputs, and even extremely degenerate cases will not lead to failure. To prove this, we tested the self-union of several examples. Table . To prove that, we tested the self-union of several examples. Table shows whether the methods tested gave valid outputs. Here, the word 'valid' means that the outputs are almost the same as the inputs. The example models *Bunny* $\cap$ *Dragon* contain topological deficiencies, such as self-intersection, that cannot be processed using regular set boolean operations. Despite this, we found that our method gave good results in these cases, indicating the robustness of our method. QuickCSG and Cork failed in all cases, because they cannot process models with coplanar faces. By perturbing one of the operands, the results of QuickCSG appear visually acceptable. However, the topologies of these results are inaccurate. Hundreds of their boundary faces do not exist in the original model (Fig. 12).

Binary boolean operations The most common process for CSG evaluation is to perform boolean operations in series. Although our method can evaluate multiple boolean operations simultaneously, we compared the performance of binary boolean operations to determine its efficiency. Table 2 shows the evaluation times of different methods. These results show that our method is very fast, and that it is half the speed of the fastest non-robust method, QuickCSG. Other robust methods, such as Maya and CGAL, are much slower, because they use arbitrary precision arithmetic. Moreover, these robust methods have very strict requirements for the inputs, and do not always deal well with topological

TABLE 1
Results of self-union evaluation using different methods

No.	Model	Face Num.	CGAL	Maya	Cork	Carve	QuickCSG	Our Method
1	Ball	360	✓	✓	✗	✓	✗	✓
2	Head	2.7k	✓	✓	✗	✓	✗	✓
3	Bunny	70k	✗	✓	✗	✓	✗	✓
4	Dragon	277k	✗	✗	✗	✗	✗	✓

TABLE 2
Computation time statistics of binary boolean operations (seconds)

No.	Model	Face Num. 1	Face Num. 2	CGAL	Maya	Cork	Carve	QuickCSG	Our Method
1	Budda ∪ Lion	1.08M	400k	-	-	-	-	3.44	6.88
2	Dragon ∪ Bunny	100k	70k	-	-	-	-	0.613	1.70
3	Armadillo ∪ Armadillo2	150k	150k	487	15.4	7.00	189	0.746	1.62
4	Horse ∪ Corpse	145k	499k	-	38.6	12.6	1.52k	0.630	1.00
5	Budda ∪ Budda2	1.08M	1.08M	-	-	-	-	4.84	-

TABLE 3
Computation time statistics of the evaluations of large CSGs (seconds)

No.	Model	Face Num.	Mesh Num.	CGAL	Cork	Carve	QuickCSG	Our Method
1	Sprocket	11k	52	211	-	4.26	0.132	0.804
2	Ring & Ball	146k	801	-	-	187	-	62.6
3	T1	80k	50	1.00k	18.5	10.4	0.388	20.2
4	T2	7k	50	2.81k	-	16.0	0.804	-
5	H	33k	42	-	-	-	2.22	-
6	Organic	219k	6	-	14.3	63.1	0.580	2.75

deficiencies. These methods simply fail when the inputs are not valid, while our method always attempts to provide a result. With (some) very large CSGs, our method spends the most time on octree construction. This means the other stages, which involve plane-based geometry, take only a small percentage of the time. This further demonstrates the efficiency of our plane-based algorithms*.

CSGs with large number of meshes To identify the ability of the methods to evaluate large CSGs, we tested some CSGs with tens or hundreds of meshes. Only QuichCSG and our method can perform multiple boolean operations directly. Other methods decompose CSG trees into binary boolean operations. The computation times of robust methods like CGAL and Maya were found to be unacceptably long for large CSGs. Conversely, our method showed good performance and stability. During our experiments, Maya gave the correct results in the first few binary boolean operations, but failed in the later ones. This demonstrates a disadvantage of incremental boolean operation methodsłthey can accumulate numerical errors which affect the algorithms stability.

7.2 Implementation

Searching valid loops Searching valid loops In §5.2, we stated that a valid loop on a tess-graph must correspond to its orientation. However, as a tess-graph is nondirectional, it is difficult to check whether a loop orientation agrees with that of the face (Fig. 14a). The graph connections on the triangle edges have unambiguous directions. Thus, starting from these edges, we can guarantee that the loops will have the right orientations. After a valid loop is found, more graph connections will have unambiguous directions, because one connection can participate in two valid loops at most. Therefore, the correct orientation will propagate in a flood-filling manner.

There is no guarantee that the tess-graph will be a connected graph. We solve this problem by inserting auxiliary intersections into the tess-graph, to ensure it is connected. An auxiliary intersection links the corner vertex ($\boldsymbol{v}_i$) of a triangle face which is confirmed to be on the outer component, with the vertex on the inner component. The intersection refinement needs to be performed again if the auxiliary intersections cross any other intersections. This solution is consistent with the principle of no constructions. To guarantee that the auxiliary intersection has a valid PBI-rep, the vertex must be determined from the inner connected component generated by the intersection between the triangle and an edge (denoted as $\boldsymbol{e}_j$) from another mesh. All intersection end points of triangle-triangle intersections are of this type. Therefore, we obtain three vertices with exact coordinates ($\boldsymbol{v}_i$ and the two end points of $\boldsymbol{e}_j$), and, therefore,

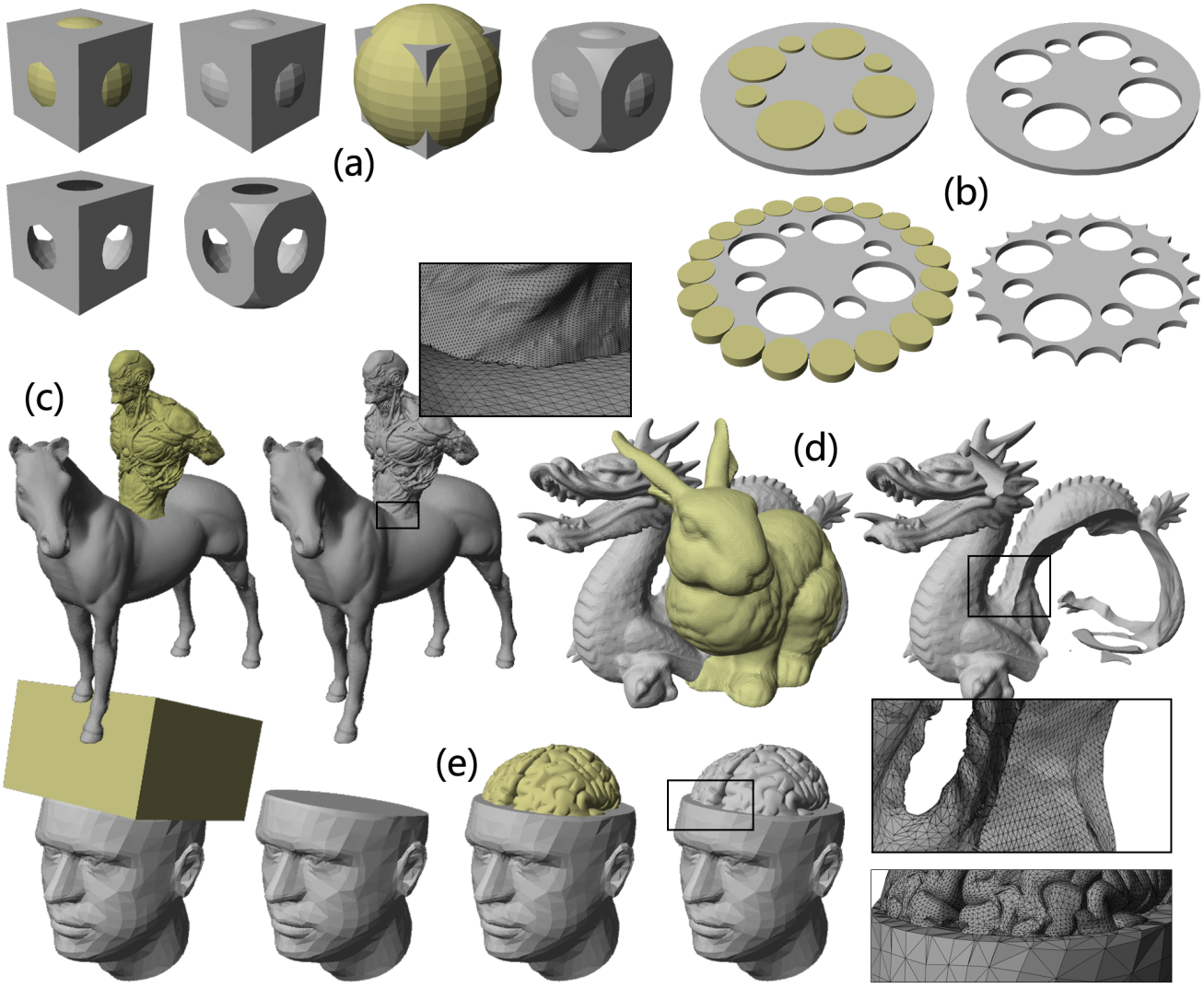


Fig. 15. **********different models: will be replaced

can construct the plane on which the auxiliary intersection lies.

Seed indicator generation Seed indicator generation In §6.1, state that flood-filling starts from a seed vertex $\boldsymbol{v}_0$. The indicator vector of the seed $\boldsymbol{\Lambda}(\boldsymbol{v}_0)$ can be generated by a point-in-polyhedron test [44], using the octree as an acceleration structure [38]. However, a simpler strategy is to use a vertex with known indicators as the seed. The vertex with the maximum x-coordinate is chosen as the seed. Its indicators are either out (in, if the complement is applied on the mesh) or on. The on indicators are generally easy to determine. Exceptions can occur because of coplanar situationsłvertices may not be registered on the mesh, even if they are on the mesh. Fortunately, if a vertex is chosen whose neighboring faces are not coplanar, this will be prevented.

Indicator vectors can propagate among single meshes, and between different meshes across shared vertices and 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 meshes, extra seeds are required to propagate each component. The indicators of these additional seeds can be computed by point-in-polyhedron tests.

Exporting to float-point number Exporting to float-point number The vertices of the final mesh generated by our method are represented as either planes or vertex coordinates. The vertices originating from the input meshes have exact coordinates, however, the vertices newly introduced by the intersection between meshes have only P-reps, and require rounding-off when computing their coordinates. Although our method guarantees the correct topology in the final mesh, rounding errors can cause topological deficiencies. The method of Zhou et al. [7] can be used to solve this problem iteratively.

7.3 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. 2, *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 minimize this problem.

The application of our method is limited to regular set meshes. However, recent reports have proposed that the piece-wise wind number (PWN) is a more powerful method to identify the insides and outsides of meshes [7]. The input requirements of our method may be extended to PWN meshes, that allow topological deficiencies, such as self-intersection and multi-components. We believe that this would be an interesting and valuable extension of our work.

8 Summary

In this paper, we propose a novel method to evaluate CSG models. This method can efficiently perform unconditionally robust non-incremental boolean operations. The novel component of our approach is to embed P-rep information into B-reps. P-reps allow us to strictly follow the principle of no geometry construction to avoid numerical errors. The use of B-reps enables fast neighborhood queries to reduce the computation time. The experimental results show that the performance of our method is competitive with state-of-the-art non-robust methods, while guaranteeing robustness.

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] 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.

[7] Q. Zhou, E. Grinspun, D. Zorin, and A. Jacobson, "Mesh arrangements for solid geometry," in *Tristate Workshop on Imaging and Graphics Posters*, 2016.

[8] 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.

[9] M. Segal, "Using tolerances to guarantee valid polyhedral modeling results," in *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4. ACM, 1990, pp. 105–114.

[10] 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.

[11] G. Bernstein and D. Fussell, "Fast, exact, linear booleans," in *Computer Graphics Forum*, vol. 28, no. 5. Wiley Online Library, 2009, pp. 1269–1278.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] S. Xu and J. Keyser, "Fast and robust booleans on polyhedra," *Computer-Aided Design*, vol. 45, no. 2, pp. 529–534, 2013.

[18] 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.

[19] S. Fortune, "Polyhedral modelling with exact arithmetic," in *Proceedings of the third ACM symposium on Solid modeling and applications*. ACM, 1995, pp. 225–234.

[20] J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha, "Esolidła system for exact boundary evaluation," *Computer-Aided Design*, vol. 36, no. 2, pp. 175–193, 2004.

[21] 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.

[22] 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.

[23] 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.

[24] C.-Y. Hu, N. M. Patrikalakis, and X. Ye, "Robust interval solid modelling," *Computer-Aided Design*, vol. 28, no. 10, pp. 807–817, 1996.

[25] 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/

[26] H. Bieri and W. Nef, "Elementary set operations with d-dimensional polyhedra," in *Workshop on Computational Geometry*. Springer, 1988, pp. 97–112.

[27] 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.

[28] 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.

[29] 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.

[30] C. J. Ogayar, F. R. Feito, R. J. Segura, and M. Rivero, "Gpu-based evaluation of boolean operations on triangulated solids," 2006.

[31] 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.

[32] 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.

[33] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, 1997.

[34] T. Möller, "A fast triangle-triangle intersection test," *Journal of graphics tools*, vol. 2, no. 2, pp. 25–30, 1997.

[35] L. P. Chew, "Constrained delaunay triangulations," *Algorithmica*, vol. 4, no. 1-4, pp. 97–108, 1989.

[36] 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.

[37] 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.

[38] 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.

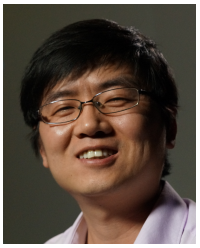
[39] F. P. Preparata and M. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[40] Autodesk Ltd. (2014) Maya 2015. [Online]. Available: http://www.autodesk.com/

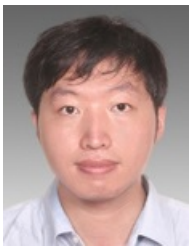
[41] G. Bernstein. (2013) Cork boolean library. [Online]. Available: https://github.com/gilbo/cork/
[42] T. Sargeant. (2011) Carve csg boolean library. [Online]. Available: https://github.com/VTREEM/Carve
[43] L. Kobbelt. (2010) Webbsp 0.3 beta. [Online]. Available: http://www.graphics.rwth-aachen.de/webbsp/
[44] 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.

Xudong Jiang received his Master degree in Computer Science from Shanghai Jiao Tong University in 2014. He is currently working in Autodesk China Research & Development Center. His research interest includes computer-aided geometric design and solid modeling.



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.



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.

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.