Efficient, Robust and Exact Boolean Operations on N Triangular Mesh Primitives

Abstract—Constructive solid geometry (CSG) is widely used for computer aided design and manufacturing. However, boolean operations, which compute the boundary representation of CSG, have been suffered from robustness and efficiency problem for more than three decades. In this paper, we propose a fast and exact boolean operation approach which is theoretically robust for regular meshes. This is done by introducing plane-based geometry into boolean operations and strictly follow the principle of no geometry constructions during processing. Different from previous BSP-based methods, our method is with only one-time conversion to the plane representation in intersection areas, instead of a series of complex conversions between different representations. Also, by using geometry connectivity and space neighboring information, our method reuse the computation iteratively, saving much computation time. Comparison experiments with state-of-the-art techniques show that our method is unconditionally exact and robust, while is competitively efficient with non-robust methods and much faster than the robust ones.

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

1 Introduction

Constructive solid geometry (CSG) has long been a popular modeling tool of computer-aided design and computer-aided manufacturing (CAD/CAM). It constructs complex models 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, which is an classical topic with history of over three decades.

In general, there are two major types of boolean algorithms according to how they deal with intersection between primitive meshes. Approximate methods redicretize the intersection areas, fit vertices approximatively and rearrange topology. On the other hand, exact method do not change vertex positions, retain as many input elements (e.g., faces, vertices and topology) as possible. In many applications, exact methods are preferred for their accuracy. Also, using exact methods, the clear mapping between the surface of input and output meshes makes it easier for the inheritance of information like face colors and materials. However, for exact boolean operation methods, there is always a bargain between robustness and efficiency. Recent exact boolean operation methods try to find faster ways to compute boundaries of CSG. However, for robustness, the answers are often exact arithmetic [3, 4] or trivial workarounds, like setting a tolerance [5, 6] or rotating the primitive with small angles to avoid coplanar cases [7]. However, none of the solution is good enough. While methods using exact arithmetic are significantly slower, workarounds are often case dependent and not reliable for complex CSGs.

We develop an unconditionally robust and exact boolean operation method for triangle meshes without using exact arithmetic. The robustness of our method is based on the observations by Sugihara and Iri [8] that boolean operations can be performed without geometry constructions if representations are based on planes. This idea of using plane-based representation (P-rep) has been developed by Berstein and Fussell [9] by combining it with binary search partition (BSP) techniques [10, 11]. Later, Campen et. al. [12] improved the method using octrees to resolve intersections locally, avoiding inefficient large BSP tree construction. Despite of that, Campen et. al.'s method still suffers from performance problem because of the complex conversions among B-reps, P-reps and BSP representations. Also, the incoherence between BSP representation, where boolean operations actually happens, and B-reps, which are input and output meshes, requires extra steps to connect the reconstructed intersection areas with the unchanged areas, leading to extra computation burden and complex topology.

Our method is based on B-reps of meshes with geometry connectivity (e.g., halfedge [13]). We embed P-rep of faces into the B-rep, forming a hybrid representation, which allows us to take the advantage of both representations for robustness and efficiency. This hybrid representation requires only one-time conversion from B-rep to P-rep in intersection areas of meshes. Moreover, our method can perform boolean operations with more than two primitives in one call, instead of recursively doing incremental boolean operations. Despite of extra complexity, non-incremental boolean operation may save much computation time by avoiding repetitive computation. The major difficulty of embedding P-rep into B-rep is the incoherence between the two representations. Many geometry operations (e.g., coordinates projection and compute the barycenter coordinate of triangle) available on B-rep can be very complex

- *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 thus inefficient on P-rep. On the contrary, all geometry predicates have to be performed under P-rep for exactness and robustness. We solve this problem by various methods to constrain the computation within plane-based geometry predicates. The experiments show that our method is much faster than existing robust exact methods, while only about two to three times slower than the state-of-art non-robust exact methods.

2 Related Work

Boolean operations have been researched for over three decades since it inception in the 1980s [14, 15]. Previous works of boolean operations can be classified into two categories: exact methods and approximate methods. Exact methods retain vertex positions. Also 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 intersection. However, the coordinates cannot be represented exactly using fix-precision float-point numbers and the robustness is always a problem. On the other hand, approximate methods rediscretize the input mesh surfaces using techniques such as voxels, octrees and Layered Depth Images. They can reach better performance and robustness, but loss of geometry information and precision is inevitable.

2.1 Exact Methods

For exact methods, some [4, 5, 7, 16, 17] pursues efficiency but cannot guarantee robustness in degenerate cases. Most of them are implemented by fix-precision float-point arithmetic and numerical errors are inevitable. Douze et al. [7] implemented very efficient method that can handle very large meshes. Also, they can perform non-incremental boolean operations: evaluating the final result mesh of CSG all at once instead of decomposing the CSG into a series of incremental boolean operations. However, their method can neither deal with coplanar cases nor guarantee accuracy of topology. In fact, large CSGs are more complex and more likely to contain degenerate cases, and more vulnerable under numerical errors. Efficiency only is far from being practical.

Many researchers have attempted to solve robustness issue for exact boolean operations using arbitrary precision arithmetic [18–22] and exact interval computation [6, 23, 24]. However, these methods are too expensive in computation time and memory to be practical for CSGs with massive faces. For example, the current standard of robust exact boolean operations method is CGAL's [25] exact-arithmetic implementation [21] of Nef polyhedra [26]. It requires conversions between Nef polyhedra and input mesh data structures during evaluation and is more than 50 times slower than non-robust booleans [9].

2.2 Approximative Methods

Since efficiency, accuracy and robustness are hard to be satisfied at the same time, starting from [27], some sacrifice accuracy for better efficiency and robustness. Most of such methods are based on conversion to volumetric representations. However, the quality of result mesh depends on the resolution of the volume grid and better quality requires dramatically higher resolution. To accelerate this process, some try to reduce the complexity of the output mesh [28] and others try to preserve non-intersected areas of the input mesh to avoid redundant tessellation [29–33]. Also, with the development of general-purpose computing on graphics processing units (GPGPU), many of them utilize the grand computation power of graphics hardware for boolean operations. These methods often have good performance and are suitable for interactive applications. However, the fundamental problem of approximate methods is owing to grid-depend nature of volumetric calculations: they inevitably suffers from geometric detail loss and unwieldy topology changes.

2.3 Plane-Based Methods

The concept of plane-based representation of polygonal meshes was first described by Sugihara and Iri [8]. P-rep provides important advantage that for boolean operations, no new primary geometry information has to be constructed to obtain the result polyhedron—it is composed of a subset of the planes of the input polyhedra. Berstein and Fussell [9] combined the P-rep with binary space partitioning (BSP) [10, 11] to develop an unconditionally robust boolean operation method. Compared to volume-grid based methods, their BSP-based method increases mesh complexity more moderately and the input topology can be reconstructed better. However, it requires conversions between input data structures and P-rep which are inefficient from both storage and time views. Also, the merge of two BSP trees is an $O(mn)$ time complexity process, where m and n are the size of the trees. It makes this method impractical for very large input meshes. Campen and Kobbelt [12] improved their method by localization using an octree. And the refinement only takes place locally near intersections.

3 Background and Overview

Our method is designed for boolean operations on arbitrary number of regular triangle meshes [14]. Input meshes are required to be without self-intersecting, represented with B-reps with geometry connectivity (e.g., halfedge, winged-edge, etc.). Despite of these requirements, our method is not sensitive to topology deficiencies like holes and self-intersections and works correctly deficiencies are not near the intersection areas between meshes.

3.1 Background

The boolean operation of triangle meshes is in essence a process of face selection. Namely, given a boolean function, it reserves those primitive faces that pass the function and drop the others to generate the final mesh. Unfortunately, for B-rep meshes, not all the input faces can be classified as a whole, since only part of the faces belong to the final mesh. Therefore, we need an extra step aiming at detecting intersections between meshes and performing tessellation. Most of the boolean operation methods follow this two-step paradigm, which consists of intersection computation and face classification. So is our method.

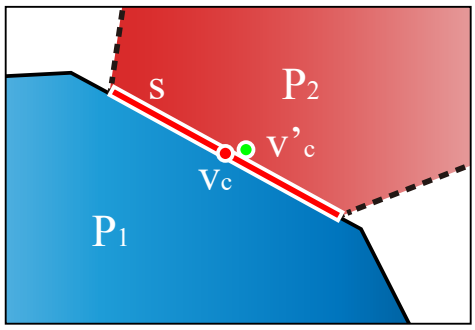


Fig. 1. In a 2D view, face s from P_2 is on the surface of mesh P_1. However, because we use face barycenter v_c to compute the indicator, the coordinates of v_c contain round-off error and move to v_c', we might falsely classify s as outside of P_2.

During the intersection computation, input faces are tested in pairs to compute their intersections. Each input meshes are tessellated according to the intersection results, making every face be either inside, outside or on the boundary of other primitives. The tessellated meshes are what we called *intersection-free meshes*. Unfortunately, under fix-precision float-point arithmetic, intersection tests are error-prone: first, degenerate cases of intersection are hard to detect; second, when there are intersections, new vertices are introduced into the geometry, whose coordinates cannot be exactly represented by fix-precision float-point numbers.

During face classification, we evaluate whether a given face s belongs to the final geometry according to the n-primitive boolean function f :

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

The parameter $\lambda_i(s)$ is space indicator with respect to mesh P_i. To compute $\lambda_f(s)$ and classify s, one has to know the space indicators with respect to all the primitives (denoted as $\mathbf{\Lambda}(s)$). A space indicator has four conditions: completely inside (*in*), completely outside (*out*), on the boundary with consistent normals (*same*) or opposite normals (*oppo*). The rules of boolean function evaluation are described in [5, 7].

Face classification also suffers from robustness problems. Many methods classify face according to the indicators of its barycenter, which can be computed by point-in-polyhedron test [5, 12]. However, coordinates of barycenters cannot be exactly represented and could generate false classification (Fig. 1). In addition, for the large amount of faces and input meshes, many methods take the benefits of the local coherence of indicators, classifying neighboring faces together [4, 5, 16, 29]. Despite of the performance improvement, it can make the robustness problem worse because the false classification may propagate to neighboring faces.

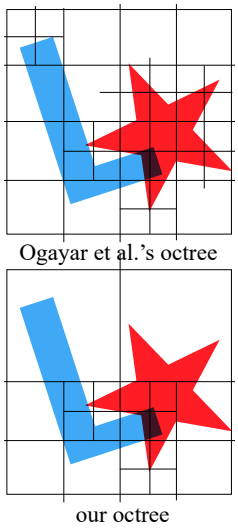
We develop an exact boolean operation method based on the two-step paradigm, being unconditional robust and as fast as possible. For efficiency, the robustness of method does not rely on arbitrary precision arithmetic. Instead, we apply the plane-based techniques from BSP-based method [12]. However, BSP structure does not support geometry connectivity query in nature, which is essential for classification acceleration. Also, we do not want complex conversions between different representations. Therefore, our method is based on B-rep and we embed P-rep into intersection areas to guarantee the robust geometry computations.

3.2 Method Overview

There are generally four stages for our boolean method. The first three stages are to compute intersection, which can be seen as pre-processing, processing and post-processing. After that, the input meshes are tessellated into intersection-free meshes with hybrid representations. The last step is to classify each face and get the final result mesh of boolean operations.

3.2.1 *Space division*

As intersection detection is performed between each pair of faces, localization is necessary to filter invalid pairs beforehand. Many space division methods can be used in this step. We use the adaptive octree for its simplicity and efficiency. Our implementation is akin to the implementation in Ogayar et al.'s method [16]. Intersection between triangle faces and octree leaves are efficiently detected using the separating axis theorem [34]. Octree leaves are classified into two types: if all faces that intersect a leaf belong to the same primitive, we call it a *normal cell*. Otherwise, it is a *critical cell*, within which the following intersection computation is performed.



The difference between our octree and Ogayar et al.'s is that we do not subdivide any normal cell no matter how many faces it contains. This is because subdividing normal cells benefits only the point-in-polyhedron test [35], which seldom uses in our method. This simplification can save much computing time, especially when intersections between primitives are not complex and located in small regions.

3.2.2 *Triangle-triangle intersection*

Our triangle-triangle intersection test is largely based on Möller's algorithm [36], which is very efficient to compute intersection between two triangles. However, naive implementation of Möller's algorithm using vertex-based geometry brings numerical errors. Therefore, we integrate P-rep into Möller's algorithm. All intersection points and line segments are represented using plane intersections to avoid explicitly computing the coordinates. Also, we carefully deal with degenerate cases of triangle intersections, including point intersection, edge intersection and coplanar situation. Details are provided in §5.

3.2.3 *Tessellation*

Once all intersections between triangles are detected, we need to tessellate the input meshes and construct the intersection-free meshes. In many methods like [16], constraint Delaunay triangulation (CDT) is applied to perform tessellation. However, for a CSG with more than two

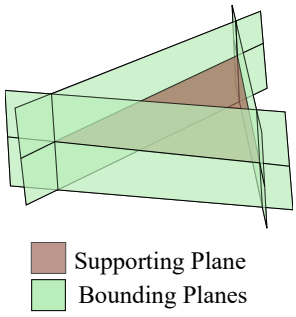
meshes, intersections may overlap or intersect with each other, and cannot be used for constraints of CDT directly. In addition, as our intersections are represented by planes, implementation of CDT are complex and inefficient, as most CDT algorithms (e.g. [37], [38]) is not designed to handle planes and requires explicit coordinates. Therefore, we first perform intersection refinement to resolve intersecting intersections. After that, we construct a graph-like structure suitable for P-rep intersections, called *tess-graph*, to guide the exact tessellation of each face. After all faces are tessellated, we get our intersection-free hybrid meshes, where 'hybrid' means the meshes are represented by both B-rep and P-rep. Details are shown in §6.

3.2.4 Face classification

This step is to choose faces from the intersection-free meshes to generate the final mesh. However, literally computing all the space indicators of each face is unacceptably slow for large CSG mesh. We utilize the geometry connectivity to propagate indicators in a flood-filling way, which is based on the spatial coherence of face indicators. In addition, many methods use barycenters of faces together with point-in-polyhedra test for face classification, which is not exact nor robust with fix-precision float-point arithmetics. Douze et al. [7] classifies faces using space indicators of vertices, but failed to deal with coplanar conditions because of coplanar leads to ambiguity of $same$ and $oppo$ in such configurations. Different from them, we classify faces based on the classification results of exactly represented vertices, including input vertices and newly introduced intersection vertices, and carefully deal with coplanar conditions to ensure topology consistency. Details will be shown in §7.

4 Plane-based Geometry

Using Plane-based representation, each face $\boldsymbol{s}$ with n edges is represented by a supporting plane $\boldsymbol{p}_{s,sp}$, where the face lies, and a list bounding planes $\{\boldsymbol{p}^i_{s,b} \mid i = 0, 1, ..., n-1\}$. Each edge line $\boldsymbol{e}^i_{\boldsymbol{s}}$ is represented by intersection $\boldsymbol{p}_{s,sp} \cap \boldsymbol{p}^i_{s,b}$. Corner vertex $\boldsymbol{v}^i_s$ is represented by intersection $\boldsymbol{p}_{s,sp} \cap \boldsymbol{p}^i_{s,b} \cap \boldsymbol{p}^{(i+1) \bmod n}_{s,b}$.



Other commonly used notations in this paper are presented here. A plane $\boldsymbol{p}$ can be represented by four scalar coefficients $a(\boldsymbol{p}), b(\boldsymbol{p}), c(\boldsymbol{p}), d(\boldsymbol{p})$. The normal of $\boldsymbol{p}$ is denoted as $\boldsymbol{n}(\boldsymbol{p})$, which is a normalized vector consisting of the first three plane coefficients. A line $\boldsymbol{l}$ can be represented by intersection of two planes $(\boldsymbol{p}^0_l \cap \boldsymbol{p}^1_l)$, or in short, $\boldsymbol{l}\colon (\boldsymbol{p}^0_l \cap \boldsymbol{p}^1_l)$. 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 $(\boldsymbol{p}^0_v \cap \boldsymbol{p}^1_v \cap \boldsymbol{p}^2_v)$, or in short, $\boldsymbol{v}\colon (\boldsymbol{p}^0_v \cap \boldsymbol{p}^1_v \cap \boldsymbol{p}^2_v)$.

We use Campen et al.'s method [12] for exact conversion of triangles from their vertex-based representation to plane-based representation. During the conversion, the coordinates of input vertices is rounded to certain precision which is dependent on the precision used to represent plane coefficients during boolean operation. If IEEE 754 double precision float-point number is used, 20-bit precision is allowed, which is enough for most CAD applications. As the precision of input coordinates is finite, we can determine in advance the upper boundary of precision needed to perform a precise decision, with which we can use fixed precision predicates to improve performance. For efficiency, we implement all geometry predicates using filtering techniques proposed by Shewchuk [39].

Plane-based geometry predicates used in our method include checking coincidence and co-orientation of planes, orientation of a plane with respect to a point, and whether three planes intersect in a unique point, etc. Most of them are well-discussed in previous work [9, 18]. In the following, we focus on two advanced predicates, which are also important building blocks of our method.

Linear ordering of points Given an line $\boldsymbol{l}_{ab}$ with two points $\boldsymbol{v}_a\colon (\boldsymbol{p}^0_a \cap \boldsymbol{p}^1_a \cap \boldsymbol{p}^2_a)$ and $\boldsymbol{v}_b\colon (\boldsymbol{p}^0_b \cap \boldsymbol{p}^1_b \cap \boldsymbol{p}^2_b)$ on it, 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 not parallel with $\boldsymbol{l}_{ab}$ from the plane representation of each point. Then we convert this problem into linear order of planes and solve it using the same technique as Banerjee et al. [18]. The chosen planes should have the same orientation with $\boldsymbol{l}_{ab}$ (the dot product between plane normal and $\boldsymbol{l}_{ab}$ has to be positive) and unqualified planes have to be flipped.

Circular ordering of lines During face tessellation, we need to know the which edges are neighboring (Fig. 3), that is, 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. Therefore, this problem is converted to that given two coplanar lines $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$ with their intersection point $\boldsymbol{v}_{ab}$, compute the circular order of the two lines. Supposing the common plane of $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$ is $\boldsymbol{p}_0$, 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 $\boldsymbol{l}_b$ under the top view of $\boldsymbol{p}_0$. The sign of $\sin\theta_{ab}$ is the same as triple product $\boldsymbol{n}(\boldsymbol{p}_0) \cdot (\boldsymbol{l}_a \times \boldsymbol{l}_b)$, where $\boldsymbol{n}$ means the normal direction. However, explicit computing this equation is complicated because both $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$ are implicitly represented by intersection of planes.

We develop an easier way that requiring no explicit computation of $\boldsymbol{l}_a$ and $\boldsymbol{l}_b$. First, we write the P-reps of two lines as $\boldsymbol{l}_a\colon (\boldsymbol{p}_0 \cap \boldsymbol{p}_a)$ and $\boldsymbol{l}_b\colon (\boldsymbol{p}_0 \cap \boldsymbol{p}_b)$. Then we orthogonally decompose $\boldsymbol{n}(\boldsymbol{p}_a)$ and $\boldsymbol{n}(\boldsymbol{p}_b)$ along the normal the common plane $\boldsymbol{p}_0$:

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

Here superscript '$\parallel$' means the projection on $\boldsymbol{p}_s$ and '$\perp$' means orthogonal to $\boldsymbol{p}_0$. As we know $\boldsymbol{n}^{\perp}(\boldsymbol{p}_a)$ 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 simple geometry reasoning, we find that the angle between $\boldsymbol{n}^{\parallel}(\boldsymbol{p}_a)$ and $\boldsymbol{n}^{\parallel}(\boldsymbol{p}_b)$ is exactly θ_{ab}. So the sign of the right side of equation 3 is exactly the sign of $\sin\theta_{ab}$. It means we have the following nice conclusion:

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

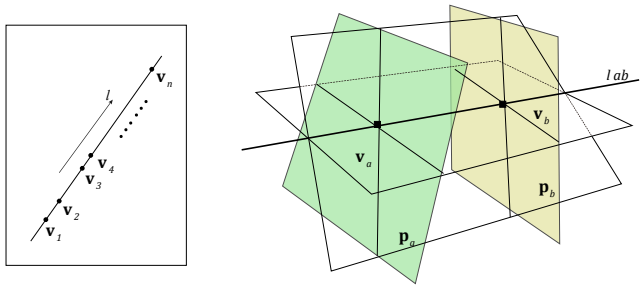


Fig. 2. The configuration of linear ordering of points. Points $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$ are both on line $\boldsymbol{l}_{ab}$. We convert this problem into the plane ordering of $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ along $\boldsymbol{l}_{ab}$.

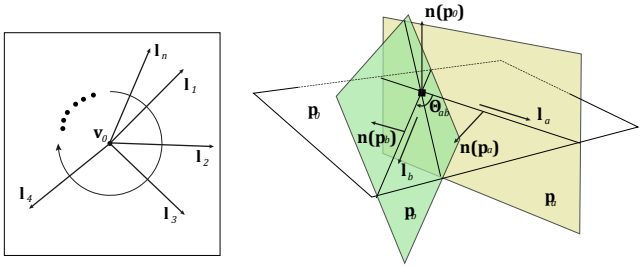


Fig. 3. The configuration of circular ordering of lines. $\boldsymbol{l}_a$: $(\boldsymbol{p}_0 \cap \boldsymbol{p}_a)$ and $\boldsymbol{l}_b$: $(\boldsymbol{p}_0 \cap \boldsymbol{p}_b)$ are coplanar in $\boldsymbol{p}_0$.

Since all the three vectors on the right side are already explicitly represented, we only have to evaluate the sign of an 3×3 matrix and avoid complicated high precision float-point arithmetic.

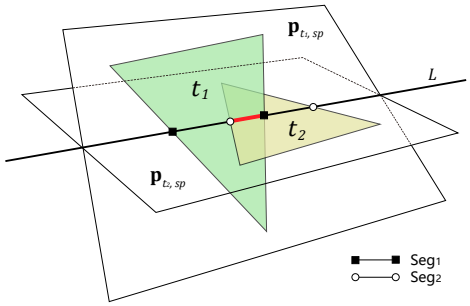


Fig. 4. Seg_1 is the intersection between $\boldsymbol{p}_{t_2,sp}$ and t_1. Seg_2 is the intersection between $\boldsymbol{p}_{t_1,sp}$ and t_2. The intersection between t_1 and t_2, which is the line segment in red, is the overlap of Seg_1 and Seg_2.

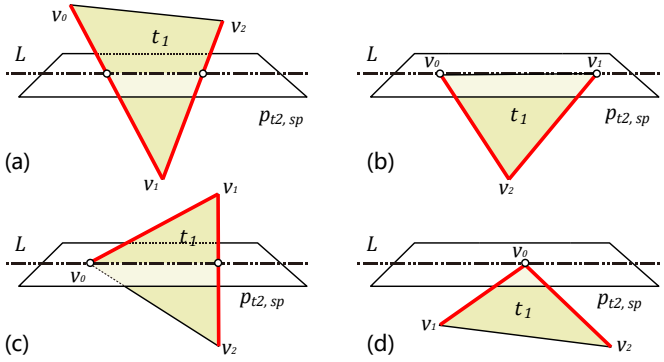


Fig. 5. We denote the signed distance from point v_i to plane $\boldsymbol{p}_{t_2,sp}$ as d_i. All the four conditions of intersection between t_1 and $\boldsymbol{p}_{t_2,sp}$ are: (a) $d_0 \cdot d_2 < 0$, $d_1 \cdot d_2 < 0$; (b) $d_0 = 0$, $d_1 = 0$, $d_2 \neq 0$; (c) $d_0 = 0$, $d_1 \cdot d_2 <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

5 Exact Intersection Computation

In this step, intersections between faces are computed through triangle-triangle intersection test. We adopt Möller's algorithm [36] on account of its efficiency and simplicity. However, a naive implementation of Möller's algorithm may fail to produce correct results because of numerical errors. Therefore, we integrate plane-based geometry to make it unconditionally exact and robust. In the following, we first make a quick review of Möller's algorithm, and then discuss the way to embed plane-based geometry. After that, we discuss how to deal with degenerate cases.

5.1 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 done 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 separately computed .
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 is from computing the coordinates of intersection vertices, which is the end points of $Seg1$ and $Seg2$. Although implementation with arbitrary precision arithmetic produces exact coordinates, it is too costly for boolean operations of large CSG. We use plane-based geometry to solve this problem by implicitly representing intersections with planes.

5.2 Plane-Based Intersection

Given a triangle face t, it is firstly converted 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 tree steps of Möller's algorithm as following:

1) The basic subroutine of the first step is given two faces t_1 and t_2, computing the orientation of of a face with respect to vertices from the other face. We find that since the exact coordinates of each triangle corners are known, we can compute the point-plane orientation within the precision of plane coefficients—namely, using simple fix float-point arithmetic. This allow us to perform this early rejection efficiently.
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. Take Seg_1 as an example. Seg_1 is the intersection between t_1 and $\boldsymbol{p}_{t_2,sp}$ and the end points of Seg_1 are intersections between $\boldsymbol{p}_{t_2,sp}$ and edges from t_1 (denoted as $\boldsymbol{e}^i_{t_1}$). 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

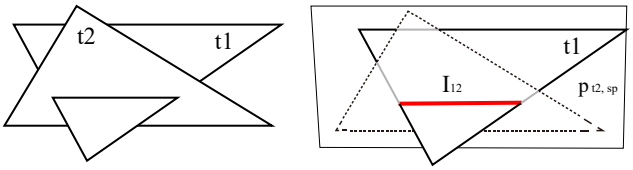


Fig. 6. Triangle t_1 and t_2 are from mesh P_i, P_j respectively. They intersect on line segment, generating intersection $\mathfrak{I}_{12}$ on t_1 and $\mathfrak{I}_{21}$ on t_2. The PBI-rep of $\mathfrak{I}_{12}$ is $\{t_1, \boldsymbol{p}_{t_2,sp}, (\boldsymbol{p}^0_{t_2,b}, \boldsymbol{p}^1_{t_2,b}), \{(P_i, \mathcal{N}_i), (P_j, \mathcal{N}_j)\}\}$.

of $\boldsymbol{p}_{t_2,sp} \cap \boldsymbol{p}_{t_1,sp} \cap \boldsymbol{p}^i_{t_1,b}$. In Fig. 5, we list all possible intersecting situations between t_1 and $\boldsymbol{p}_{t_2,sp}$. The two intersection edges $\boldsymbol{e}^i_{t_1}$ for each situation are also presented. The coplanar situation will be discussed later in §5.4.

3) The intersection between t_1 and t_2 is the overlap area of Seg_1 and Seg_2. It can be easily 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 §4 to sort.

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

5.3 Plane-Based Intersection Representation

In our method, intersection on triangle t is denoted as $\mathfrak{I}$: $\{t, \boldsymbol{p}_{ext}, (\boldsymbol{p}^0, \boldsymbol{p}^1), (P, \mathcal{N})\}$, which is called plane-based intersection representation (PBI-rep). The first component t indicates $\mathfrak{I}$ is on triangle t, and second one $\boldsymbol{p}_{ext}$ indicates $\mathfrak{I}$ lies on $\boldsymbol{p}_{ext}$, which also means it lies on 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 $(P, \mathcal{N})$ represents the neighborhood of the intersection $\mathfrak{I}$. It indicates $\mathfrak{I}$ is intersection of triangle t with $\mathcal{N}$ from mesh P, where $\mathcal{N}$ can be either a edge or a face.

For example, in Fig. 6, two triangle faces t_1 and t_2 from mesh P_i and P_j intersect. Then it generates two intersections for each of the two triangles— $\mathfrak{I}_{12}$ on t_1 and $\mathfrak{I}_{21}$ on t_2. For $\mathfrak{I}_{12}$, the first two components are t_1 and $\boldsymbol{p}_{t_2,sp}$. The third component is boundary planes from t_2 which has already been discussed in §5.2. The last component is (P_j, t_2), meaning the neighborhood of $\mathfrak{I}_{12}$ is the face t_2. Sometimes intersections may lie on the mesh edges instead of faces (Fig. 7), which is called 'edge intersection'. In that case, the second component of neighborhood is the corresponding edge. We postpone the discussion of edge intersection to §5.4.

5.4 Degenerate Case Handling

While two triangles, if intersecting, intersect on a line segment in most situations, 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 triangle edges. These degenerate situations prevent us from performing robust boolean operations. In this section, we offer simple but effective way to deal with all these degenerations and conceal the complexity of intersections behind the concise PBI-rep.

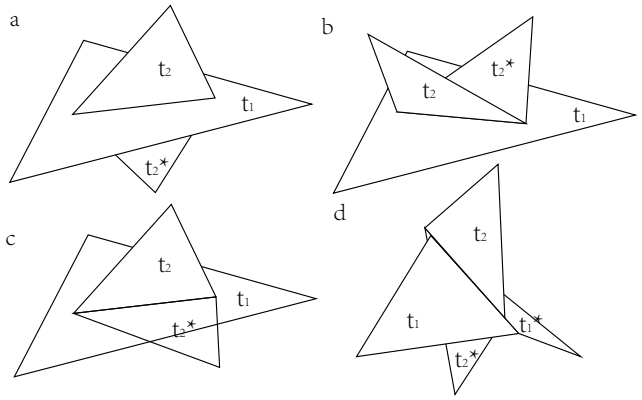


Fig. 7. Different conditions of edge intersection. Triangle faces t_1 and t^*_1 are companion faces. Triangle faces t_2 and t^*_2 are companion faces. a) t_2 and t^*_2 are in different sides of t_1. b) t_2 and t^*_2 are in the same side of t_1. c) t^*_2 is coplanar with t_1. d) Both t_1 and t_2 have companion faces: the intersection is edge intersection for both t_1 and t_2 (instead of only for t_1 is previous three conditions).

5.4.1 Point intersection

If two triangles intersect on a single point (e.g., Fig. 5d), the intersection cannot be represented using our four-component description since the line segment collapses into a single point. In these cases, we simply add the intersection point into the related triangles to guarantee correct tessellation, and no intersection is introduced.

5.4.2 Edge intersection

When intersection line segment lies on an edge (Fig. 7). we call it *edge intersection*. The space near edge intersection is divided by faces around the intersected edge, instead of just one face. If intersection is on the edge $\boldsymbol{e}_{k,i}$ from mesh P_k, the neighborhood of the intersection is then $(P_k, \boldsymbol{e}_{k,i})$, where we use $\boldsymbol{e}_{k,i}$ to represent all faces from P_k adjacent to $\boldsymbol{e}_{k,i}$.

Another thing needs to be noticed is that there will be repetitive detection of edge intersections. For example, in Fig. 7a, the same intersection on t_1 will be detected twice because t_1 intersect both t_2 and $t^\star_2$ in the same place. We solve this duplication together with other interactions among different intersections in §6. We call such t_2 and $t^\star_2$ as *companion triangles* in an edge intersection, a concept which will be referred again in the discussion of coplanar cases.

5.4.3 Copalnar

Consider two triangle faces t_1 and t_2 intersect within a common plane. Both t_1 and t_2 will divide each other into two areas—a convex overlapping area and exclusive parts (Fig. 8a). Apparantly, if we tessellate both t_1 and t_2 according to the boundary of overlapping area, we can guarantee that the tessellated meshes will not cross the meshes where t_1 and t_2 are from. Many methods [4, 5] claim that tessellating faces according to 2D-intersections is necessary. However, in our method, we do not test whether t_1 and t_2 really intersect in 2D once we find they are coplanar. We treat coplanar

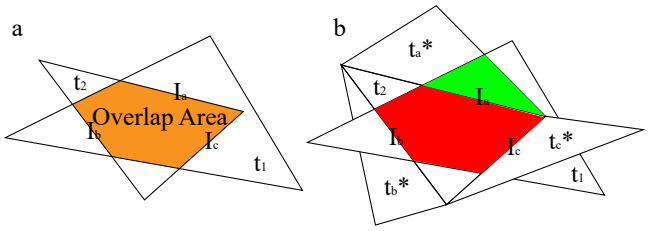


Fig. 8. a) Coplanar cases, t_1 and t_2 intersect in 2D, forming three intersections $\mathcal{I}_a$, $\mathcal{I}_b$ and $\mathcal{I}_c$ on t_1. b) The possible configurations of the companion faces. For $\mathcal{I}_a$, the companion triangle t_a^* is coplanar with t_1. Therefore, the red and the green areas do not need to be split during tessellation, and $\mathcal{I}_a$ is invalid. For $\mathcal{I}_b$ and $\mathcal{I}_c$, the companion faces are not coplanar, so these intersection can be recorded during intersection test between t_1 and the companion faces t_b^* and t_c^*. In any configurations, their is no need to record the intersection.

situations as they do not intersect at all. In this way, we simplify our method, making it more robust and fast, while doing no harm to the topology correctness.

We find that each intersection line segment in 2D is on the edges from input meshes. That means we can view 2D intersection as a special case of edge intersection. The only difference is that there can be up to three edge intersections in one 2D intersection (Fig. 8b). As we have discussed, edge intersection will be detected twice. That means we can rely on the companion triangle faces 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. The word 'valid' means the intersections will appear as an edge in the final mesh, thus is necessary to put into tessellation. Supposing the intersection is on triangle t_1, this definition equals to:

$$\lambda_f(\boldsymbol{x}_i) \neq constant, \boldsymbol{x}_i \in \boldsymbol{U}(\mathcal{I}) \cup t_1 \tag{5}$$

, where f is the CSG function, and $\boldsymbol{x}_i \in \boldsymbol{U}(\mathcal{I})$ is the points from the neighborhood of intersection $\mathcal{I}$. This inequation indicates the two sides of $\mathcal{I}$ have different classification and must be split according to $\mathcal{I}$ during tessellation. However, in most time, the validation of intersection is not known until classification stage, because we have to know the indicators of points in $\boldsymbol{U}(\mathcal{I}) \cup t_1$. Fortunately, if companion faces are both coplanar with t_1, both sides of $\mathcal{I}$ have the same indicators and thus the same classification. It means $\mathcal{I}$ is invalid and we do not need to record it either.

6 Deferred Tessellation

After intersection computation, tessellation is performed on each intersecting triangle face and generate our intersection-free meshes. We call this stage as deferred tessellation because the tessellation happens until all valid intersections are computed.

In previous work, Ogayar-Anguita et. al. [16] use Constrained Delaunay Triangulation (CDT) to perform deferred tessellation. They treat triangle faces as convex triangulation zones and intersections as constraints. Our first try is to implement a robust plane-based CDT. However, CDT algorithms [37, 40] usually require coordinates projection or intersection detection between arbitrary connections of

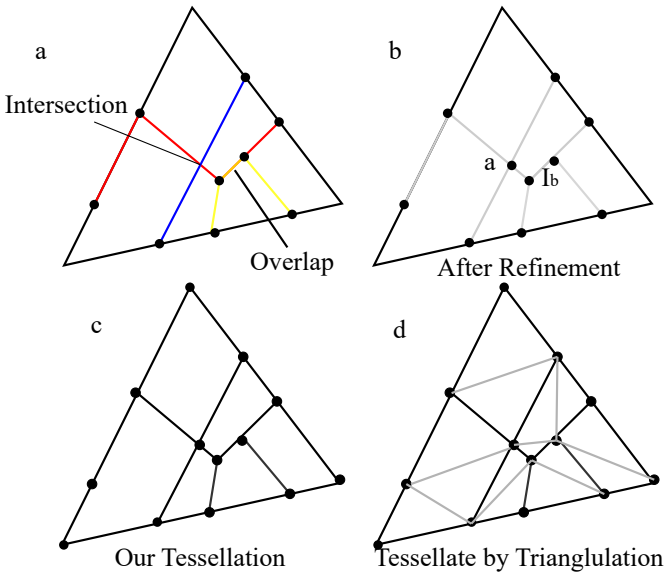


Fig. 9. a) Different colors indicate the intersections are from different meshes. The blue intersections and the red ones intersect at a point. Also, the yellow intersections overlap with the red ones. b) After refinement, we introduce a new vertex a, and merge overlapping intersections into a new one $\mathcal{I}_b$. c) Our tessellation does not guarantee to obtain triangles. d) If triangulation is performed, new edges (gray ones) are introduced and we cannot give the precise plane-based representation for these new edges.

vertices within triangulation zone. Because vertices may be implicitly represented in our method, these geometry predicates are hard to guarantee computation efficiency and precision simultaneously. Moreover, when there are more than two primitives, simply applying CDT omits the complexity of intersections between meshes. The intersections may intersect each other, introducing new vertices and splitting original intersection line segments (Fig. 9a). This breaks the assumption of most CDT algorithms and may lead to incorrect output topology.

To solve these problems, we first perform intersection refinement to eliminate any crossing or overlapping between intersections. After that, we perform conservative tessellation based on *tess-graph*, ensuring unconditional topology correctness. An tess-graph is a graph-like description of intersections on a certain face. The word 'conservative' means we tessellate face into polygons, not triangles. This is because if triangles are required, we need construct new bounding planes for these new triangles, which violating our principle of no geometry construction (Fig. 9d).

6.1 Intersection Refinement

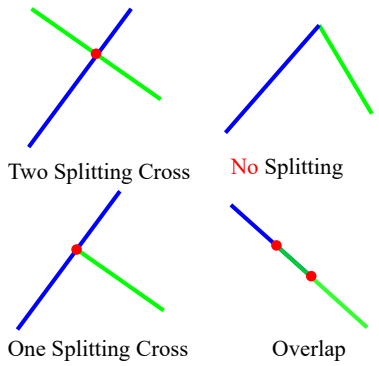
The intersection vertices are not only introduced by intersection between an edge and a triangle, but also introduced by intersections of three triangles. In order to find all intersection vertices, it is not enough by only triangle-triangle intersection tests. To make tessellation easier and more robust, we have to refine these intersections and resolve the intersections between intersections.

Intersection refinement is performed in a local scope. For each intersecting face t from mesh P_t, we collect all intersections on t as set $\boldsymbol{\Gamma}(t)$ and refine them together. For

simpler processing, we also include the three edges of t into $\mathbf{\Gamma}(t)$, since edges also participate in tessellation. In $\mathbf{\Gamma}(t)$, edges are represented as PBI. The neighborhood components of edges are set as $None$ because they do not have a neighborhood edge or face. Other PBI-rep components can be determined by the P-reps of edges. The refinement on $\mathbf{\Gamma}(t)$ is done using only plane-based geometry predicates by the following three steps:

Coincidence elimination We merge coincidence intersections which have the same end points. Intersections are undirected so intersections with inverse end points are also coincident. The PBI-rep of the new intersection is inherited from either of the old one's except the neighborhood component. The new neighborhood is the combination of the merged ones', meaning the intersection has two (or more) neighborhoods.

Cross and overlap resolving After the process of two previous steps, there are no overlapping intersections in $\mathbf{\Gamma}(t)$. We check whether any pair of intersections cross or overlap (collinear) each other. The definition of *cross* does not include the situation that two intersections share end points, since no splitting is required in this situation. When there is crossing or overlapping, at least one of the intersections have to be split. Because one intersection may may 'touch' more than one other intersections, the splitting is deferred until all splitting points are found. The subroutine linear order of points (§4) is used to sort these splitting points along the splitting intersection. The PBI-reps of split segments inherit their father's except the third component which identifies the end points.



Coincidence elimination (again) Unfortunately, the overlap resolving may bring new coincident intersections. Therefore the first step is run again in the final step.

6.2 Tessellation by Tess-Graph

So far, intersections are only independent data that store the coordinates and neighbouring information. To perform tessellation and extract subfaces, we need to organize them to reveal the topology. We use 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 according to the refined intersections. Nodes of tess-graph represent end points of intersections and undirectional connections between nodes represent intersections. The construction of tess-graph is straightforward and the reader will not have problem filling the details.

After we have tess-graph, we tessellate face and extract subfaces according to it. This is done by extract valid loops on tess-graph. A valid loop that corresponds to a subface must satisfy two criterions: 1) the direction of loops should be coherent with the face normal, and 2) consecutive connections on the loop should be adjacent by circular order. Each valid loop corresponds to a intersection-free face. After we find out all valid loops on tess-graph, we finish tessellating the corresponding face. To facilitate the later process, we also store neighborhood information into the edges of the new faces. Because the edges of new faces are generated from intersections, the neighborhood information of the edges is inherent from the corresponding intersections.

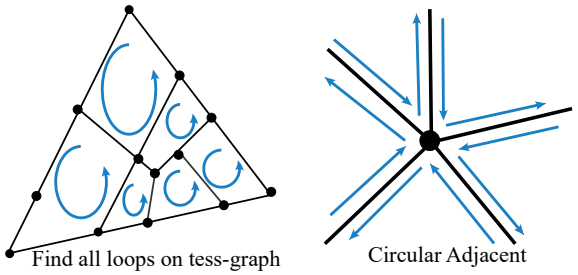


Fig. 10. *Left*: we find all valid loops on tess-graph to tesselate a triangle. The direction of loops have to be coherent with the triangle normal (assume the normal points to the outside of paper). *Right*: each circular adjacent edge pair is an angle of the tessellated polygon.

When all faces are tessellated, meshes are intersection-free. We further attach the PBI-reps of intersections to the edges generated from tess-graph connections which corresponds to an intersection. Finally, we get the intersection-free hybrid meshes, with some edges contains neighborhood information which indicates there are intersections between meshes.

7 Exact Face Classification

We traverse each face in the intersection-free meshes, and classify which one belongs to the final mesh. Classification is done by evaluating the indicator vector of each face. The basic idea of our classification method is to utilize the space coherence of face indicator vectors to reuse the classification results. Because we store the PBI-reps which contains neighborhood information in the edges which intersect other meshes, the search space of point-in-polyhedron test is limited to only a few faces, which makes the indicator computation very fast. Also, by using P-reps, our classification method can deal with any degenerate cases.

7.1 Framework

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

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

Our hybrid meshes facilitate this process by accelerating the indicator propagation. Given adjacent faces $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, it is straightforward to know whether $\mathbf{\Lambda}(\boldsymbol{s}_1)$ and $\mathbf{\Lambda}(\boldsymbol{s}_2)$ are different. The two indicator vectors are the same unless the shared edge $\boldsymbol{e}_{12}$ lies on the boundary of some other

meshes, which means there is neighborhood information, say, $\mathcal{N}(\boldsymbol{e_{12}}) = \{(P_k, \mathcal{N}_k)\}$, stored on $\boldsymbol{e}_{12}$. The neighborhood information indicates which components of indicator vector differ between $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$. If there is a neighborhood $(P_k, \mathcal{N}_k) \in \mathcal{N}(\boldsymbol{e_{12}})$, the k^{th} indicators $\lambda_k(\boldsymbol{s}_1)$ and $\lambda_k(\boldsymbol{s}_2)$ can be computed efficiently and exactly according to $\mathcal{N}_k$ (§7.2). 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 $\boldsymbol{s}_i$

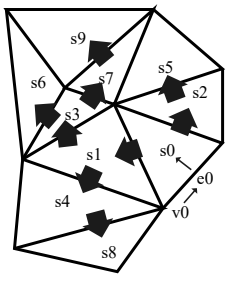
```
Select a proper seed face s_0
Compute the seed indicator vector Λ(s_0)
PROPAGATE( s_0 , Λ(s_0))

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
```

7.2 Indicator Propagation

Meshes consist of vertices, edges and faces. Classification is required to perform on the face level. However, polyhedron in-out test is performed based on points. Gaps exist between the them. Conventionally, face barycenter is used to compute the indicator of the whole face. This is because in intersection-free meshes, face is classified as a whole, and every face inner point will have the same indicator as the face's. However, the computation of inner point coordinates requires geometry construction, which introduces errors without exact arithmetic. Therefore, we use the vertices instead, whose exact representations are known, for indicator computation.

However, there are two gaps between face indicator and vertex indicator. First, the indicator of vertex does not always equal to indicator of face. Second, face indicator has two conditions in *on* case—*same* and *oppo*, because the face has its orientation.



To overcome the gaps, we discuss our indicator trace (6) in finer granularity. In the beginning, we start from a seed vertex $\boldsymbol{v}_0$ on seed face $\boldsymbol{s}_0$. We assume 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 the edge layer into the trace. Then it changes to:

$$\boldsymbol{v}_0 \to \boldsymbol{e}_0 \to \boldsymbol{s}_0 \to (\boldsymbol{e}_1) \to \boldsymbol{s}_1 \to (\boldsymbol{e}_2) \to \boldsymbol{s}_2 \to \dots \quad (7)$$

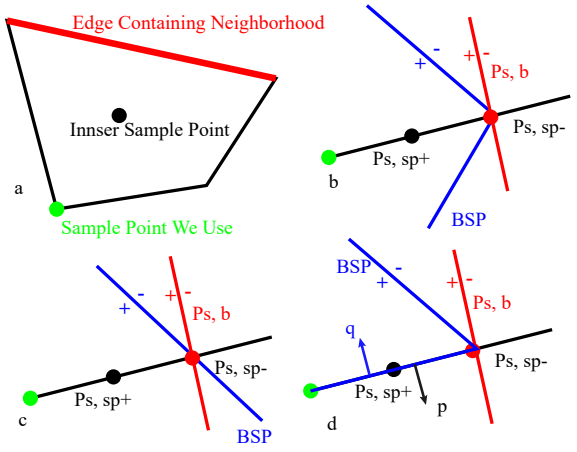


Fig. 11. a) Because we cannot use inner sample point includes geometry construction, we choose the vertices (green one) instead to compute the face indicator. b-d): Different situations of classification. The simple structure of neighborhood BSP guarantees the same classification result of the points on $\boldsymbol{p}^{+}_{\boldsymbol{s},sp}$. In *a*, the BSP contains two half-planes because it is constructed from two neighborhood faces. In *d*, the classification result is *on* so we need to determine the orientation by the normal of half-plane q and the face p. In this condition, the indicator should be *oppo* since p and q are opposite.

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

Given the partial order $on \succ in$ and $on \succ out$, our key observation is that the following relation always stands for indicators:

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

, where $\lambda_k(x)$ means the indicator of x against a certain primitive P_k. This relation can be inferred from continuous space assumption and the definition of intersection-free mesh. It tells us when we perform $\boldsymbol{v_e} \to \boldsymbol{e}$ and $\boldsymbol{e_s} \to \boldsymbol{s}$, we decide whether the on indicator changes to in or out or remains on. When we perform $\boldsymbol{s} \to \boldsymbol{e_s}$, we decide whether in and out indicators change to on. In addition, considering the orientation of face, we also need to figure out whether the on indicators changes to $same$ or $oppo$ during $\boldsymbol{e_s} \to \boldsymbol{s}$ operation.

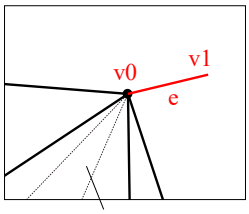
$\boldsymbol{s} \to \boldsymbol{e_s}$ We need to find which indicators change to on. We known when $\lambda_k(\boldsymbol{e_s}) = on$, there will be neighborhood information related to mesh P_k stored on $\boldsymbol{e_s}$. Therefore, in this operation, all we need is to iterate over neighborhoods of $\boldsymbol{e_s}$ and change corresponding indicators to on.

$\boldsymbol{e_s} \to \boldsymbol{s}$ From eq. 8, we know if $\lambda_k(\boldsymbol{e_s}) \neq on$, then $\lambda_k(\boldsymbol{e_s}) = \lambda_k(\boldsymbol{s})$. On the other hand, when $\lambda_k(\boldsymbol{e_s}) = on$, we figure out $\lambda_k(\boldsymbol{s})$ using neighborhood information related to P_k. We know the corresponding neighborhood $\mathcal{N}_k$ and can be either a triangle face or an edge from mesh P_k. If $\mathcal{N}_k$ is a face, we can build a trivial BSP [11] using the face on the space near $\boldsymbol{e_s}$. If $\mathcal{N}_k$ is an edge, we can still build BSP using the faces adjacent to $\boldsymbol{e_s}$. In both cases, the BSP can be used

to compute $\lambda_k(\boldsymbol{s})$ if we can sample a point from $\boldsymbol{s}$ that near $\boldsymbol{e_s}$. Unfortunately, we cannot guarantee to find such a point with precise coordinates.

We find that the BSP constructed from edge neighborhood is so simple that indicators λ_k of all points on the half plane $\boldsymbol{p}^+_{\boldsymbol{s},sp}$ are the same. Here $\boldsymbol{p}^+_{\boldsymbol{s},sp}$ is half of the supporting plane of $\boldsymbol{s}$ that on the positive side of 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 figuring out $\lambda_k(\boldsymbol{v}_x)$, where $\boldsymbol{v}_x \in \boldsymbol{p}^+_{\boldsymbol{s},sp}$. For polygons, there is at least one vertex on $\boldsymbol{p}^+_{\boldsymbol{s},sp}$, whose exact coordinates are known. Therefore that vertex is used as the v_x. Also, if we assign each BSP splitting plane a normal by the normal of triangle it contains, we can use it to decide the orientation ($same$ or $oppo$) when $\boldsymbol{v}_x$ is on a splitting plane, which means $\lambda_k(\boldsymbol{s}) = on$.

$\boldsymbol{v_e} \rightarrow \boldsymbol{e}$ While this is yet another propagation from low dimension to high dimension, there is no neighborhood information stored on vertex. If $\lambda_k(\boldsymbol{v_e}) = on$, we first check whether $\lambda_k(\boldsymbol{e}) = on$ by checking the neighborhood component of the intersection on $\boldsymbol{e}$. If not, we need to find where $\boldsymbol{v_e}$ lies on the surface of P_k and determine the neighborhood of $\boldsymbol{v_e}$. In most cases, it is straightforward since coincident vertices among different intersection-free meshes are shared. Exceptions are discussion is in §8. The neighborhood may be a vertex, an edge or a face. After computing the neighborhood, the same BSP-based classification method is used as in $\boldsymbol{e_s} \rightarrow \boldsymbol{s}$ to determine the high dimension indicator $\lambda_k(\boldsymbol{e})$. We use the other end point of $\boldsymbol{e}$ to be the sample point for the BSP classification.



Vertex Neighborhood

7.3 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. Thanks to the indicator space coherence, we can save the computation time by caching the evaluation result $f(\boldsymbol{\Lambda}(\boldsymbol{s}_i))$ during flood-filling.

The basic cache strategy is to cache final result, that is, the value of $\lambda_f(\boldsymbol{s}_i)$. We know indicator vector will not change if there is no intersections on the edge during propagation. That means the final classification $\lambda_f(\boldsymbol{s}_i)$ will not change, too. Thus, those faces sharing the same indicator vector can be classified as a whole.

Also, we can perform intermediate results cache. We noticed that the boolean expression can be simplified if some components of $\boldsymbol{\Lambda}(\boldsymbol{s}_i)$ is fixed. For example, assume we have a boolean expression $f = P_1 \cup (P_2 \cap P_3 - P_4)$. Given the values of two indicators $\lambda_1(\boldsymbol{s}_i) = out$, $\lambda_2(\boldsymbol{s}_i) = in$, the expression can be rewritten as $f(\lambda_1 = out, \lambda_2 = in) = out \cup (in \cap P_3 - P_4)$. Using the combination rules we can simplify the expression as $f(\lambda_1 = out, \lambda_2 = in) = P_3 - P_4$. This fact is important because for a large CSG, a certain primitive (denoted as P_k) often intersects with only a few other meshes $\Theta(P_k) = \{P_{n_1}, P_{n_2}, \cdots, P_{n_x}\}$. That means all the faces in P_k has the same indicators $\lambda_i, P_i \notin \Theta(P_k)$. Therefore, we can first determine these fixed indicators and simplify the boolean function, and then use simplified one to compute the final indicator for each faces in P_k.

7.4 Tessellating on Classification

We find for a CSG with many meshes, a large percentage of intersections are invalid (see §5.4 for 'invalid'). Tessellation according to these invalid intersections is not necessary. To avoid unnecessary tessellation, we can give an early prediction of which intersections are invalid by intersections information of faces, but have to embed the tessellation stage into the classification stage, which we call it 'tessellating on classification'.

In most situations, a certain faces $\boldsymbol{s}$ only intersect with a few other meshes $\Theta(\boldsymbol{s})$. We can use the same cache strategy in §7.3 to shrink the original CSG into a smaller one by the common indicators of $\{P_i\}\backslash\Theta(\boldsymbol{s})$, where $\{P_i\}$ means all the primitive meshes. After the shrinking, the primitive meshes of the CSG must be from $\Theta(\boldsymbol{s})$. If any mesh in $\Theta(\boldsymbol{s})$ does not appear in the shrinked CSG, it means the indicator of that mesh does not change the classification result, which in another word, intersections on $\boldsymbol{s}$ from that mesh are invalid and do not affect the classification of any subfaces of $\boldsymbol{s}$. If the shrinked CSG is trivial which does not contain any primitives, it means $\boldsymbol{s}$ does not need tessellation. In this way, we save the computation time, and simplify the topology of the final result mesh. Because we can only shrink the CSG for $\boldsymbol{s}$ when we classify it—before that the indicators of meshes from $\{P_i\}\backslash\Theta(\boldsymbol{s})$ are not known—we do not perform tessellation until we do the classification.

8 Discussion

Searching valid loops In §6.2, we claim a valid loops on tess-graph has to be correct with its orientation. However, since tess-graph is undirectional, it is not easy to check whether a loop orientation agrees with the face's (Fig. 13a). We observe that the graph connections lie on face edges have their unambiguous directions. Starting from these edges, we can guarantee that the loops have the right orientations. After we find a valid loop, more graph connections will have unambiguous directions because one connection can participate two valid loops at most. In such way, the correct orientation will propagate in a flood-filling way.

There is another problem that the tess-graph may not be a connected graph. We avoid this problem by inserting auxiliary intersections into the tess-graph to make it connected. An auxiliary intersection links the vertex of face (denoted as $\boldsymbol{v}_i$) on the outer connected component with the vertex on the inner connected component. The intersection refinement has to be performed again if the auxiliary intersections cross any other intersections. To guarantee the auxiliary intersection has a valid PBI-rep, we require the vertex from inner connected component generated by intersection between the face and an edge (denoted as $\boldsymbol{e}_j$) from other meshes—all newly introduced vertices during triangle-triangle intersection are this type. In this way, we get three vertices with exact coordinates ($\boldsymbol{v}_i$ and the two end points of $\boldsymbol{e}_j$) and can construct an exact plane where the auxiliary intersection lies.

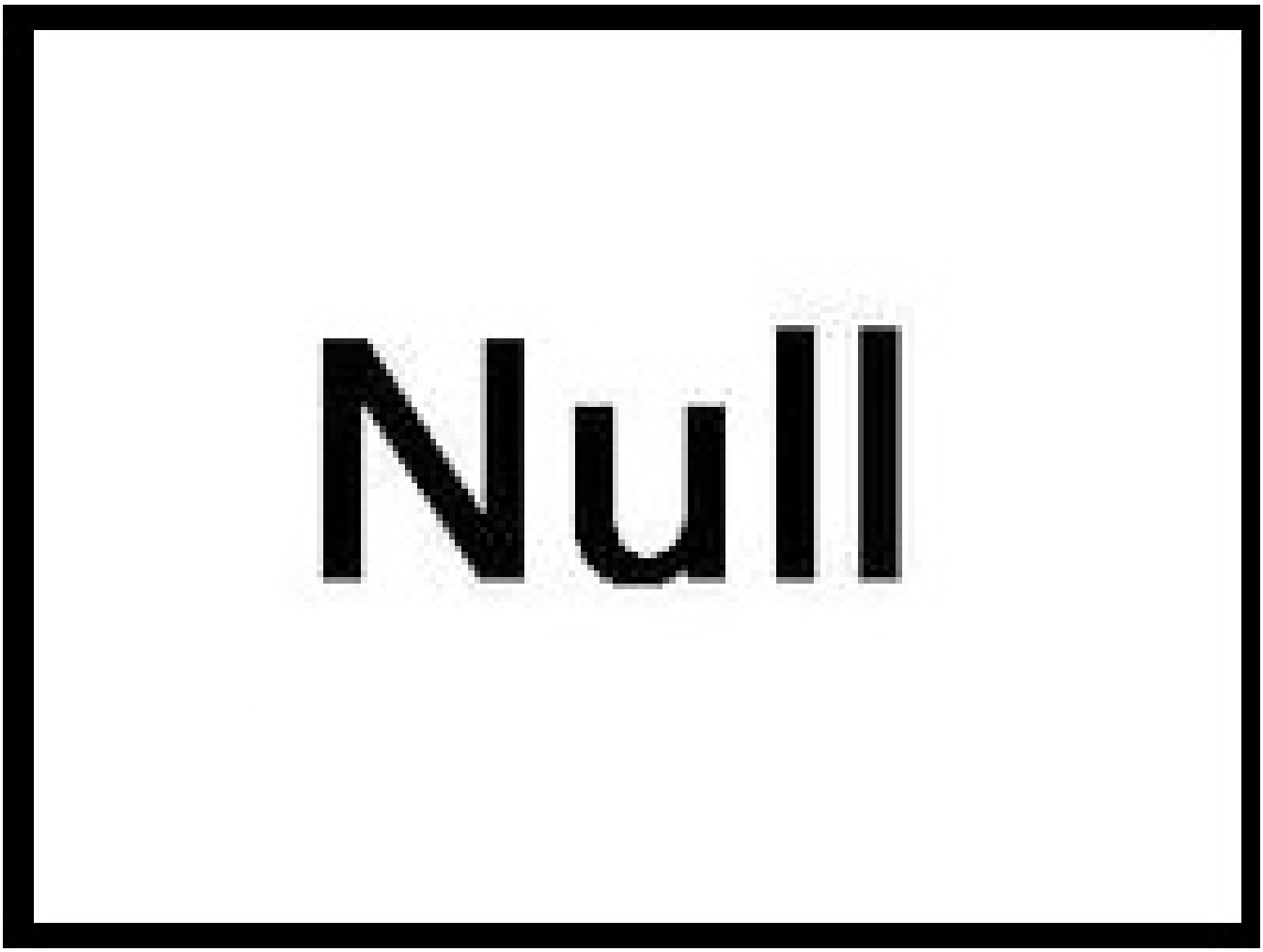


Fig. 12. different models.....

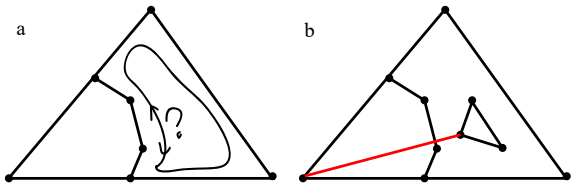


Fig. 13. a) We cannot easily figure out the correct orientation of the loop. b) When there are more than one connected component, we use auxiliary intersection (red line) to connect the two component.

Seed indicator generation In §7.2, we claim that our flood-filling starts from a seed vertex $\boldsymbol{v}_0$. While the indicator vector of the seed $\boldsymbol{\Lambda}(\boldsymbol{v}_0)$ can be generated by point-in-polyhedron test [41] with the octree as acceleration structure [35], we have simpler strategies by using vertex with known indicators as the seed. We choose the vertex with the max x-coordinate as the seed. In this way, $\lambda_k(\boldsymbol{v}_0)$ is either out (in, if complement is applied on mesh) or on. The on indicators are easy to determine in most time. Though sometimes exception occur because of multi-coplanar situation, this will not happen if we carefully choose the right vertex whose neighbor faces are not coplanar.

The indicator vectors can propagate not only among single meshes, but can also between different meshes by the vertices and edges they share. Therefore in most time, we only need one single seed vertex for classification. However, if there are more than two connected components among meshes, we need to new seeds to propagate each component. The indicators of the second and later seeds can be computed by point-in-polyhedron tests.

9 Experiments

We implemented the proposed method in C++ and tested a series of models on a laptop with Intel Core i5 1.5GHz CPU and 8GB RAM. To prove both the efficiency and robustness, we perform boolean evaluation on various CSGs with different complexity. Extreme degenerate cases such as self-union are also tested.

We compared our method with several previous works with available implementations, including CGAL [25], Maya [3, 42],"Cork" [44], "QuickCSG" [7], "Carve" [45], and online service of Campen and Kobbelt's plane-based method [12, 43].

Self-union Our method is unconditionally robust for regular set meshes. Even extreme degenerate cases will not cause invalid output. To prove that, we tested the self-union

TABLE 1
Computation Time Statistics of Binary Boolean Operation (Seconds)

No.	Model	Triangle Num. 1	Triangle Num. 2	CGAL	Maya	Cork	Carve	QuickCSG	Our Method
1	Budda ∪ Lion	1.08M	0.4M	-	-	-	-	3.44	6.88
2	Dragon ∪ Bunny	100k	70k	-	-	-	-	0.334	0.636
3	Armadillo ∪ Armadillo2	150k	150k	-	-	-	-	0.746	1.62
4	Horse ∪ Corpse	-	-	-	-	-	-	-	-
5	Dinasour ∪ Ironman	-	-	-	-	-	-	-	-

TABLE 2
Computation Time Statistics of Large CSG Evaluation (Seconds)

No.	Model	Triangle Num.	Primitive Num.	CGAL*	Maya*	Cork	Carve	QuickCSG	Our Method
1	Sprocket	-	52	-	-	-	-	0.059	-
2	ring and ball	-	201	-	-	-	-	-	-
3	T1	-	50	-	-	-	-	0.433	10.1
4	T2	-	50	-	-	-	-	0.804	-
5	H	-	42	-	-	-	-	2.22	-
6	Organic	-	6	-	-	-	-	0.590	-
7	6 Budda	-	6	-	-	-	-	-	-
8	Serpent	-	-	-	-	-	-	-	-
9	Dithering	-	-	-	-	-	-	-	-

TABLE 3
Face Number of the Results of Self-Union Evaluation

No.	Model	Orig. Num.	CGAL	Maya	Cork	Carve	QuickCSG	QuickCSG2*	Our Method
1	Ball	360	-	360	-	-	1	360	360
2	head	2,716	-	2,716	-	-	5	2,716	2,716
3	Bunny	69,666	-	69,664	-	-	0	69,668	69,669
4	Dragon	276,972	-	0	-	-	1	276,702	277,987

* We perturb the model by 1e-6 to give QuickCSG the ability to partly solve coplanar cases.

of several models and some CSG with coplanar faces. Table 3 compares the difference between input meshes and result meshes by their face number. We find that our method always produces correct or at least near correct results. The slightly deviation of our result from the original one is caused by self-intersection in the original models. QuickCSG failed to produce the result in all coplanar cases, because they cannot deal with coplanar faces. After perturbation, results of QuickCSG are visually OK. However, the topology of their results is messy. We can see hundreds of boundary edges on their results which do not exist in the original model (Fig. 14).

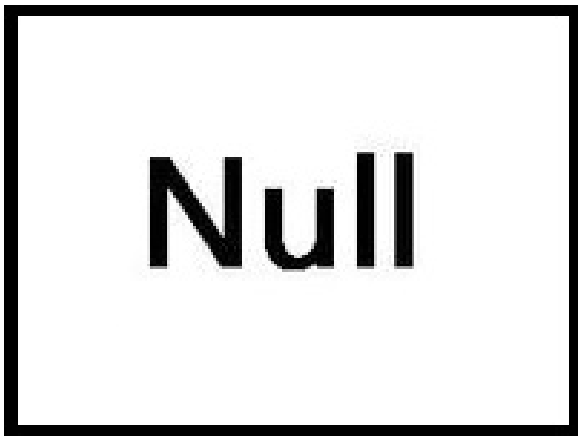


Fig. 14. boundary edges on the models.....

Binary boolean operations The most common situation of CSG evaluation is to perform boolean operations one by one, since many designers are used to modify models progressively. While our method can evaluate multiple boolean operations once for all, we also compared the performance of binary boolean operations to prove the efficiency. Table 1 shows the evaluation time of different methods. We can clearly see our method is very fast that is only slightly slower than the fastest non-robust QuickCSG. Other robust methods such as Maya and CGAL are much slower because both of them use the arbitrary precision arithmetic. It should be noted that in (some) very large CSGs, most time is spent on octree construction in our method. It means other stages which involve plane-based geometry take only a small percentage of time, proving the efficiency of our plane-based

geometry.

CSGs with large number of meshes To identify the ability of evaluating large CSG, we also test some CSG with tens or hundreds of meshes. Maya and CGAL cannot perform multiple boolean operations, so we can only give the comparison of incremental boolean evaluation for these methods. We see that the computation time of robust methods like CGAL and Maya grows rapidly when the number of meshes grows. On the other hand, our method keeps good performance while stay robust. During our experiments, we noticed Maya gives the correct results in the first few boolean operations, but failed in the later ones. It proves another disadvantage of incremental boolean operation methods: if it cannot guarantee exactness, it will accumulate numerical errors which affects the algorithm stability.

10 Extensions

Converting to float-point number The vertices of final mesh in our method are represented in both P-reps and plain coordinates. While the vertices from the input meshes have there exact coordinates, the vertices newly introduced by intersection between meshes require round-off when converting to coordinates representation. Though we guarantee the correct topology in the result mesh, such round-off can still cause topology deficiencies such as self-intersections. Here we adopt Zhou et al.'s method [4] to solve it iteratively.

Incremental evaluation While we claim that in many CSGs, a non-incremental evaluation is faster by avoiding repetitive computation, one can easily find counter-examples that non-incremental evaluation contains useless intersection computation. For example, if 100 spheres densely locate in a small region, each sphere will need to intersect with other 99 spheres. However, if evaluating incrementally, we might not need to compute so many intersections. Fortunately, our method can also perform incremental evaluation. For exactness and robustness, we have to deal with non-triangle input meshes in incremental evaluations and intersection vertices should not be converted to float-point coordinate representation.

Extending to polygon meshes In our method, we claim the inputs are triangle meshes mainly to avoid plane fitting when converting to P-reps. However, if we assume the P-reps of non-triangle faces are exactly known, our method can be easily extend to polygon meshes.

The only different part is intersection computation. The first step of Möller's triangle-triangle intersection method is to test each edge-plane pair respectively. This idea can be extended to polygon intersections. Then we can get two plane-face intersections $Seg1$ and $Seg2$, except that in polygon cases, $Seg1$ and $Seg2$ are two sets of line segments, instead of single segments. Later processes are the same.

Extend to piece-wise wind number meshes Recent works claim that the piece-wise wind number (PWN) are more powerful to identify the inside and outside of meshes [4]. Our method is straightforward to apply PWN because we can use the neighborhoods of intersections to compute the PWN of each point easily. By using PWN, we can extend our input from regular set mesh without self-intersection to so-called PWN mesh that allows deficiencies such as self-intersection and multi-component. However, the cost of this extension is that we have to check every triangle pairs for intersections, even if they are from the same input mesh. Considering this cost, we prefer a pre-computation to resolve topology deficiencies only on those input meshes that we think that might be 'unclean'.

11 Summary

In this paper, we proposed a novel method to evaluate CSG with triangular mesh primitives. It is able to efficiently perform unconditionally robust non-incremental boolean operations of CSG with massive faces. The key idea of our approach is to strictly follow the principle of plane-based geometry, avoiding geometry construction all along the method. Also, we use the local coherence of face space labels to accelerate face membership classification. Experiments have verified the proposed approach is more efficient than the state-of-the-art techniques while retaining robustness and stability.

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] 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.

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

[5] 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.

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

[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] 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.

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

[10] 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.

[11] 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.

[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] M. McGuire, "The half-edge data structure," *Website: http://www. flipcode. com/articles/article halfedgepf. shtml*, 2000.

[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] K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr, "Level set surface editing operators," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 330–338.

[28] 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.

[29] D. Pavić, M. Campen, and L. Kobbelt, "Hybrid booleans," in *Computer Graphics Forum*, vol. 29, no. 1. Wiley Online Library, 2010, pp. 75–87.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

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

[41] C. J. Ogayar, R. J. Segura, and F. R. Feito, "Point in solid strategies," *Computers & Graphics*, vol. 29, no. 4, pp. 616–624, 2005.

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

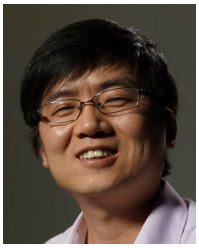
[43] L. Kobbelt. (2010) Webbsp 0.3 beta. [Online]. Available: http://www.graphics.rwth-aachen.de/webbsp/

[44] G. Bernstein. (2013) Cork boolean library. [Online]. Available: https://github.com/gilbo/cork/

[45] T. Sargeant. (2011) Carve csg boolean library. [Online]. Available: https://github.com/VTREEM/Carve

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.