Efficient Non-incremental Constructive Solid Geometry Evaluation for Triangular Meshes

Rui Wang, Xudong Jiang, Hongbo Fu, Bin Sheng, and Enhua Wu

Abstract—In this paper, we propose an efficient non-incremental approach to evaluate the boundary of constructive solid geometry (CSG). The face membership classification step is a bottleneck in many existing CSG evaluation approaches. The key idea of our approach is to use local coherence of space labels to accelerate this step. We employ a two-level grouping scheme to group faces that share the same space labels. Therefore, these common space labels can be shared within each group to avoid repetitive computation. Our method is optimized for a non-incremental evaluation. It generates the entire result model at once, rather than performing a step-by-step evaluation of Boolean operations. To achieve a robust evaluation, our approach introduces plane-based geometry into the intersection computation step. Comparison experiments with state-of-the-art techniques show that our method can more efficiently perform boundary evaluation of both trivial and complicated CSG with massive faces while maintaining robustness.

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]: union, intersection and difference. CSG is often represented by a binary tree, called a CSG tree, whose leaves are primitives and whose internal nodes are Boolean operations..

CSG can be converted into the widely-used boundary representation (i.e., triangle mesh) through boundary evaluation. Most boundary evaluation methods are based first on intersection computation and then on face membership classification. Robustness and efficiency are their major issues. During the last three decades, many techniques have been developed to solve these problems. However, the efficiency of face membership classification still needs improvement.

Computing face space labels is the basis of face classification with respect to CSG. For each face, the number of space labels to be computed is equal to the number of primitives. For large CSG with massive faces and primitives, computing these space labels can be very time-intensive. A common acceleration method is to use localized schemes. This method is based on the local coherence of space labels: if a face is inside (or outside) a specific primitive, the face's neighborhoods are likewise often inside (or outside) the primitive. Determining whether two adjacent faces share the same space labels is relatively simple (see also Section 5.2); therefore, we can locally group together faces of the same labels and reuse space labels to save computation.

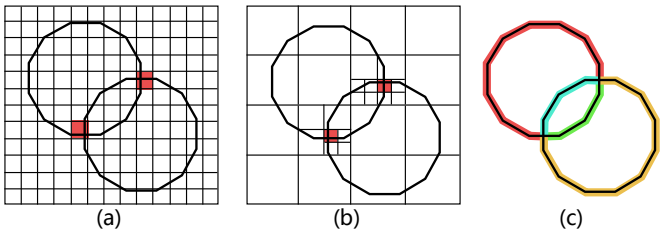


Fig. 1. 2D illustration of grouping schemes. The boundaries of polygons are edges. Suppose each input mesh consists of hundreds of edges (not shown in the figure). Edges are grouped using (a) voxels, (b) octree cells, and (c) edges as the grouping units. Different groups in (c) are marked by different colors.

Several recent studies [2], [3], [4] have used such localized schemes. These approaches often use the cube (e.g., voxels, octree cells) as the basic grouping unit to reuse the space division data structures constructed during the intersection computation. However, using the cube as a basic grouping unit has several disadvantages. In general, connected faces with the same space labels should be grouped together. Thus, an ideal face group is a union of connected faces and can be of any shape. The cube-based grouping scheme can only provide a rough approximation of that shape. Therefore, extra computation, which is usually time-consuming, is needed to classify those ungrouped faces. For example, in Figs. 1(a) and (b), the red cubes contain the intersection of two primitives. Thus, the edges (faces in 3D) in these red cubes should belong to different groups (because they have different labels). However, the size of cubes cannot be illimitably small to distinguish these different groups. Therefore, these edges in the red cubes are left as ungrouped edges. In contrast, a face-based grouping scheme can easily handle groups of arbitrary shapes and is potentially more efficient for face classification (Fig. 1(c)).

Another limitation of prior exact evaluation algorithms

- *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*
- *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*

is that most of them are incremental ones [2], [5], [6]. This means that they can evaluate only one Boolean operation at a time. For a large CSG tree with more than two primitives, incremental algorithms have to decompose the CSG into a series of Boolean operations and then incrementally evaluate each one. This strategy is highly inefficient because it unnecessarily computes the CSG intermediate results. Incremental algorithms have been more popular for a considerable time because human designers typically construct CSG models by progressively adding primitives. However, recent techniques—such as GPU-based approximate evaluation algorithms [3], [4] and CSG visualization algorithms [7], [8]—are more suitable for such a preview computation. Therefore, exact evaluation algorithms are now more likely to be used for final mesh generation (after all the primitives are added), where a non-incremental algorithm has more advantages.

In this paper, we propose a method to perform CSG boundary evaluation with triangular mesh primitives. The proposed method uses the face (i.e., a triangle in our implementation) as the basic unit of the grouping and applies a flood-filling algorithm to group faces. Our method utilizes the local coherence of space labels. It propagates face space labels between adjacent faces and adjacent groups to save computation of space labels. The most prominent characteristics of our method are outlined below.

- **Face classification using two-level grouping** To efficiently classify faces, we use a two-level grouping scheme to reuse space labels. Input faces are firstly grouped into first-level groups at the primitive level; then, the first-level groups are further divided into second-level groups according to their labels (Fig. 2(d)). A flood-filling algorithm is applied to enable efficient grouping and label sharing among adjacent faces. This framework makes a balance between the benefit of face label sharing and the cost of grouping faces for the best performance.
- **Efficient non-incremental evaluation** Unlike many existing Boolean methods [2], [5], [6], [9], the proposed method can perform non-incremental evaluation of CSG, which computes the entire final model directly. Our approach is more efficient at evaluating large CSGs because it computes no intermediate results in the internal nodes of CSG trees. A non-incremental evaluation is more difficult to perform because the conditions of the intersection and face classification are more complicated. We developed a set of techniques, including face-nested binary space partitioning (BSP) and multi-level CSG trimming, to make our method highly efficient for non-incremental evaluation.
- **Robustness and exactness** In the field of solid modeling, efficiency without robustness is impractical. We therefore adopt a series of schemes to make our method robust. To avoid the introduction of errors during intersection computation, we combine a triangle-triangle intersection algorithm [10] with a plane-based representation [11]. In addition, an exact BSP construction technique [2], [6] is adopted to enhance the robustness of face classification.

Experiments performed confirm that our method is more efficient than state-of-the-art techniques [2], [5], [12], [13]. It can quickly and robustly perform CSG evaluations not only for trivial CSGs, such as one with single Boolean operation, but also for large CSGs with hundreds of primitives.

2 Related Work

CSG evaluation has had notorious problems with robustness since its inception in the 1980s [14], [15]. The non-robustness is inherited from the building blocks of CSG: Boolean operations on solids. Many researchers have attempted to solve such an issue using arbitrary precision arithmetic [9], [16], [17], [18], [19] and exact interval computation [20], [21], [22]. However, these methods are often too expensive in computation time and memory to be practical for evaluation of CSG with massive faces. For example, in the Computational Geometry Algorithms Library (CGAL) [12], the state-of-the-art robust Boolean operation algorithm [19] (implemented with arbitrary precision arithmetic) is more than 20 times slower than its non-robust version.

Sugihara and Iri [11] introduced a plane-based representation of polyhedra. They found that Boolean operations are fundamentally robust using the plane equation as the primary geometry representation. In this condition, the evaluation of Boolean operations can be performed with no 'constructions' [23], thereby avoiding the introduction of any numerical errors.

Berstein and Fussell [6] noted that a binary space partitioning (BSP) merging algorithm in Boolean operations [24], [25] is actually a plane-based technique. Therefore, they combined the two conceptions—plane-based geometry and BSP merging—to develop an unconditionally robust method for Boolean operations of polyhedra. Using Shewchuk's adaptive geometry predicate technique [26], this method can be only twice slower than non-robust methods. Our approach adopts this technique in the stage of face classification to ensure the robustness.

Berstein and Fussell's method does not alleviate the problem of high memory consumption of BSP-based algorithms. Thus, it remains unprepared for polyhedra with massive faces. However, it inspired Campen and Kobbelt to develop a more delicate approach [2]. This approach solves the problem of exact and efficient conversion of polyhedra between vertex-based and plane-based representations. Moreover, by leveraging the adaptive octree, BSP structures are nested into octree cells where the intersection between primitives occurs. For efficiency, face classification is performed using cells as the basic classification units. This localized scheme saves considerable computing resources and retains the topology of non-intersected areas.

Such localized scheme is widely used in different methods of CSG boundary evaluation [2], [3], [4], [5]. These methods are usually based on intersection computation and face membership classification. The success of such localized scheme is enabled by two facts. First, the intersections between primitives are locally distributed. Second, space labels of a face are often coherent with those of its neighborhood. The former makes it possible to reduce the space that requires an intersection test. The latter enables sharing of space labels between adjacent faces to accelerate face classification.

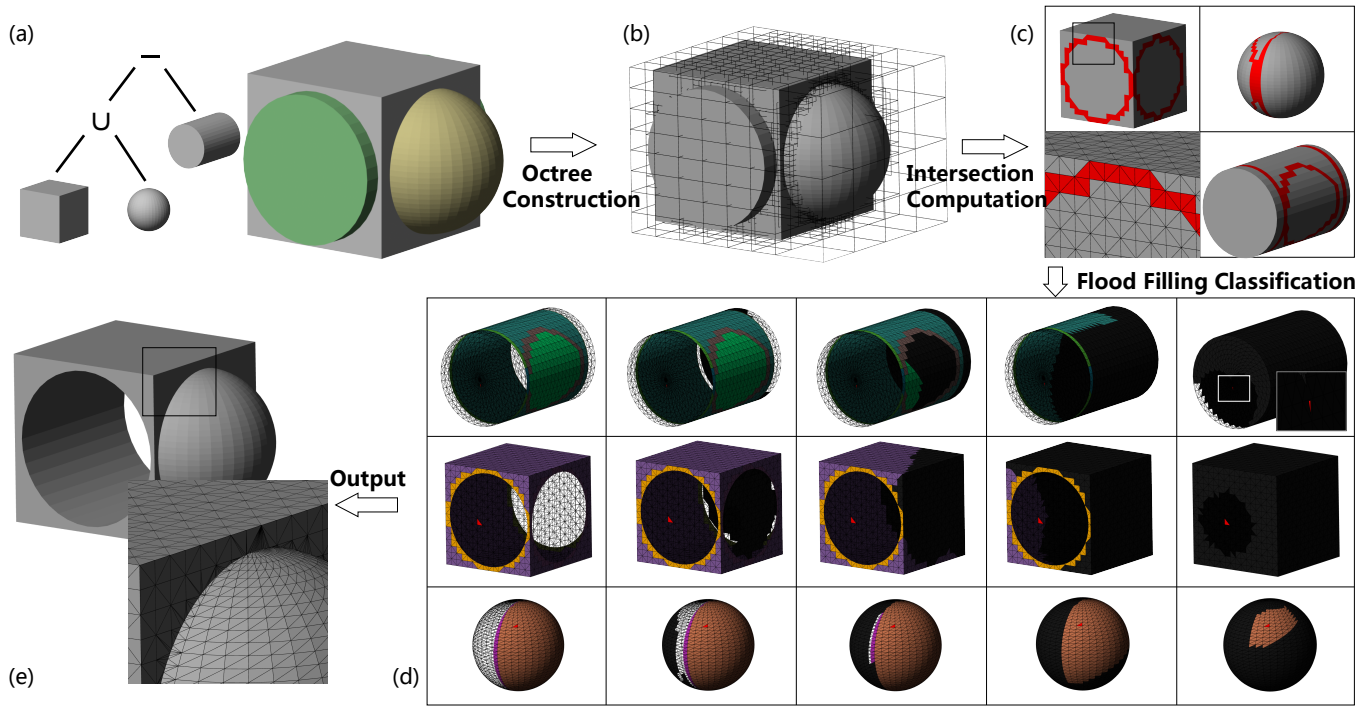


Fig. 2. An overview of the proposed method for CSG evaluation. In this toy example, the CSG tree represents a solid constructed by (Cube ∪ Sphere - Cylinder). (d) shows how to classify faces step by step using a flood filling algorithm. Faces of the same primitive are grouped as the inter-primitive group. Then faces of the same inter-primitive group are further divided into intra-primitive groups according to their conditions of intersection (c). The unvisited faces are black. Different colors of other faces indicate that they are accepted and belong to different intra-primitive groups. Rejected faces are rendered with wire-frame. The seed face of flood filling in each primitive is highlighted in red.

However, most of these methods use the cube (e.g., voxels, octree cells) as the unit of the face group, which is often the natural result of reusing space-division data structures constructed for intersection computation. In this way, connected faces that share the same labels might be distributed in different cubes. Thus, repetitive classifications may be required for each cube. Furthermore, faces of different classifications may coexist within a small space, forming a mixed area (e.g., the red areas in Figs. 1(a) and (b)). Because the size of the cubes cannot be illimitably small, special cubes that contain faces with different space labels typically must be constructed in these mixed areas. Processing these special cubes is often complex and time-consuming. To more efficiently share space labels, we therefore use faces, rather than cubes, as the basic grouping units. Grouping is efficiently performed by utilizing geometry connectivity with a flood-filling algorithm.

Recently, Feito et al. [5] proposed a method for Boolean evaluation. This method also uses a face-based grouping scheme. To share classification results, faces that do not intersect with other primitives are grouped together according to geometry connectivity. This method and the proposed method have similar stages for single Boolean operation evaluation. One of the major differences is that our method supports non-incremental evaluation of CSG. For this purpose, we designed a series of dedicated stages. Additionally, our face grouping scheme is intended for both non-intersected faces and faces that intersect with other primitives. It thus provides a uniform treatment for all faces and further accelerates the face classification stage. Moreover, the similarity between adjacent groups is omitted by Feito et al.'s method. Although different groups have different space labels, we observe that they may still share some of these labels if they are neighboring. We utilize this kind of space label coherence between groups for better performance.

On the other hand, with the development of general-purpose computing on graphics processing units (GPGPUs), many researchers [4], [27], [28], [29], [30] have tried to utilize the grand computation power of graphics hardware for Boolean operations. These methods often have good performance and are suitable for interactive applications, such as digital sculpting. However, owing to the paralleled features of graphics hardware, these methods are usually voxel-based and support only approximate evaluation that inevitably suffers from geometric detail loss, especially at the intersection areas of primitives.

3 Overview

As in most previous works [5], [6], we require the input primitives of the CSG tree to be watertight Nef polyhedra with manifold surfaces. We assume there is no self-intersection. Because we adopt a flood-filling strategy, we additionally assume that the connectivity of adjacent faces is available. Our current implementation is dedicated to primitives of triangular meshes for simplicity. Our approach efficiently and robustly computes the boundary of CSG solids in three steps, as illustrated in Fig. 2. Our main contributions exist in Steps 2 and 3.

3.1 Adaptive Octree Construction

As a typical routine of localization, our method uses the adaptive octree to frequently accelerate the used spatial query afterwards. Our octree construction implementation is akin to the implementation in Feito et al.'s method [5]. The major difference is that, because our method is for non-incremental CSG evaluation, the octree is constructed on the bounding box that contains all input primitives (Fig. 2(b)). To avoid presentation ambiguity between a CSG leaf and an octree leaf, the latter is called a cell. Intersection detection between triangle faces and cells can be efficiently performed by using the separating axis theorem [31]. Cells are classified into two types: if all triangles that intersect a cell belong to the same primitive, we call it a *non-critical* cell. Otherwise, it is a *critical* cell. This classification is needed for intersection computation in the next step.

3.2 Plane-based Intersection Computation

In this step, we compute all intersections between primitives and record them in intersected faces (e.g., the red triangles in Fig. 2(c)) for further computation. Because we have already constructed the octree and obtained its cell classification, it is instantly known that intersection between faces occurs only within critical cells. Thus, we limit the intersection test within these critical cells. The intersection test is performed for every pair of faces in each critical cell. We assume that primitives are not self-intersected; therefore, tests for faces from the same primitives are unnecessary.

The triangle-triangle intersection test always incurs robustness issues. To avoid an unexpected failure caused by numeric errors, we integrate a plane-based representation of polyhedra [11] into the triangle-triangle intersection test. Adaptive precision predicate technique [26] is applied for efficiency of plane-based geometry computation. Details are provided in Section 4.

3.3 Face Classification Using Two-level Grouping

Face classification using two-level grouping is the most novel step of our algorithm. For each primitive face, we must determine whether it, or its part (when the face intersects with other faces), belongs to the final model. A breadth-first flood-filling algorithm is used to traverse faces of each primitive. Starting from a random seed whose space labels are all computed, these labels propagate to neighboring faces and vary according to intersection conditions (Fig. 2(d)). Each face is instantly classified once all of its space labels are computed. We use a two-level face grouping scheme to enable the maximum information reuse. The first level is the *inter-primitive* level; the second is the *intra-primitive* level. When visiting an intersected face, constrained Delaunay triangulation [32] is applied to subdivide the face into smaller triangles, followed by individual classification of the subdivided triangles using BSP techniques. The robustness of this step is ensured by the exact BSP computation technique [6]. Details are shown in Section 5.

4 Plane-based Intersection Computation

In this step, intersections between faces are computed through the triangle-triangle intersection test within each

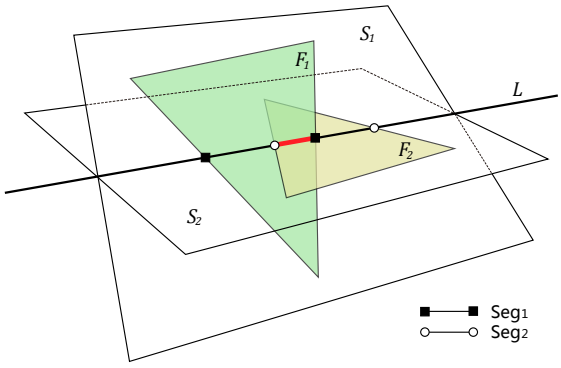


Fig. 3. Seg_1 is the intersection between S_2 and F_1. Seg_2 is the intersection between S_1 and F_2. The intersection between F_1 and F_2, which is the line segment in red, is the overlap of Seg_1 and Seg_2.

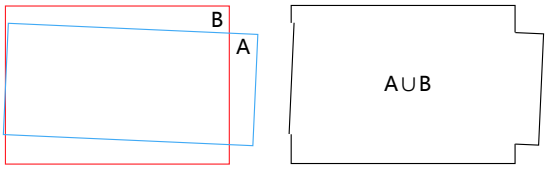


Fig. 4. The left edges of A and B (*left*) are nearly but not exactly collinear. However, under usual float point arithmetic, they might be judged as collinear causing discontinuous edges in the final result (*right*).

critical cell. We adopt Möller's intersection detection algorithm [10] for this test on account of the algorithm's efficiency and simplicity. However, because collision detection algorithms are often accompanied by non-robustness problems, a conventional implementation of the algorithm may cause unpredictable results. Therefore, we integrate plane-based geometry representation [11] into Möller's algorithm to make it unconditionally precise and robust.

4.1 Intersection Detection

Möller's algorithm computes the intersection between two triangles F_1 and F_2 in three steps, as illustrated in Fig. 3. First, an early rejection is performed by testing whether F_1 intersects the plane where F_2 lies, which is denoted as S_2. This is a necessary condition of intersection between F_1 and F_2. If this test is passed, then the same is performed for F_2 and S_1, where S_1 is the plane of F_1. Second, the intersection between F_1 and S_2, denoted as Seg_1, and the intersection between F_2 and S_1, denoted as Seg_2, are separately computed . At last, the intersection between F_1 and F_2 is determined by computing the overlap area of Seg_1 and Seg_2 .

Conventional implementations of Möller's algorithm use vertex-based representation and usual float point arithmetic. However, this is neither exact nor robust. In Fig. 4, we illustrate a non-robust 2D case of the Boolean operation. Although implementing Möller's algorithm with arbitrary precision arithmetic could make it robust, it is too costly for a large CSG evaluation. Actually, the non-robustness of this algorithm is from *constructions* [6], which compute new coordinates of geometry objects based on the known coordinates of existing ones. In Möller's algorithm, the

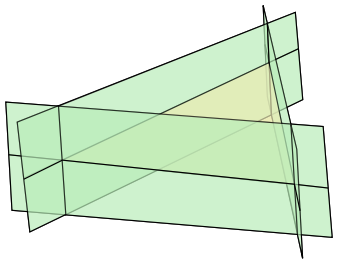


Fig. 5. Plane-based representation of triangles. The yellow triangle is represented by three bounding planes (green) and the supporting plane where the triangle lies.

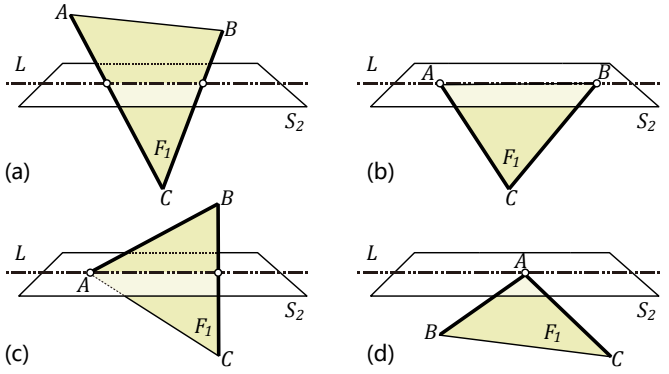


Fig. 6. We denote the signed distance from point X to plane S_2 as d_X. All the four conditions of intersection between F_1 and S_2 (denoted as Seg_1) are: (a) $d_A \cdot d_C < 0$, $d_B \cdot d_C < 0$; (b) $d_A = 0$, $d_B = 0$, $d_C \neq 0$; (c) $d_A = 0$, $d_B \cdot d_C < 0$; (d) $d_A = 0$, $d_B \cdot d_C > 0$. End points of Seg_1 are intersections between S_2 and related edge lines of F_1 (bold edges).

coordinates of Seg_1 and Seg_2 (Fig. 3) are computed as intermediate results. On the other hand, if the computation can be restricted to *predicates*, which make a two or three-way decision based on known coordinates, we could quickly and robustly implement this algorithm.

Fortunately, according to Sugihara and Iri [11], geometry computation of Boolean operations can be restricted to predicates using plane-based representation of polyhedra. This motivated us to integrate plane-based representation into Möller's algorithm. Exact and efficient predicate computation is performed using precise geometry predicate techniques [26]. Details of our implementation are shown below.

We first convert F_1 and F_2 into their respective plane-based representations: a supporting plane surrounded by three bounding planes (Fig. 5). We implement this conversion using the exact conversion method by Campen and Kobbelt [2]. Then, in our implementation, the relative position of a point and a plane can be determined by computing the sign of the multiplication of determinants [11]. To avoid introducing errors, we represent the end points of Seg_1 and Seg_2 by intersections of plane triples, rather than by directly computing their coordinates. As illustrated in Fig. 6(a), these points are the intersection between an edge line of a face and the supporting plane of the other face. Because the edge line of F_1 can be represented as $[S_1, S_X]$ (or $[S_2, S_X]$ for an edge line of F_2), where S_X is the corresponding bounding plane of the edge, the end points of Seg_1 or Seg_2 can be represented in the form of $[S_1, S_2, S_X]$. In Fig. 6, we list all intersection situations between a triangle and a plane, and the S_X in each situation.

The intersection between F_1 and F_2 is the overlap area of Seg_1 and Seg_2. It can be easily computed by comparing the end points of Seg_1 and Seg_2 along L (Fig. 3), where L is the intersection between S_1 and S_2. Unlike Möller's implementation, our approach uses no projection, which requires the exact point coordinates. We call the direction of vector $\vec{N}_1 \times \vec{N}_2$ the positive direction, where $\vec{N}_1$ and $\vec{N}_2$ are normals of S_1 and S_2 respectively. Given two points A and B on L, we compute the sign of the multiplication of determinants:

$$K(A,B) = \begin{vmatrix} \vec{S}_1 \\ \vec{S}_2 \\ \vec{S}_A \\ \vec{S}_B \end{vmatrix} \cdot \begin{vmatrix} \vec{N}_1 \\ \vec{N}_2 \\ \vec{N}_A \end{vmatrix} \cdot \begin{vmatrix} \vec{N}_1 \\ \vec{N}_2 \\ \vec{N}_B \end{vmatrix}, \quad (1)$$

where $\vec{S}_X$ is the coefficient vector of plane equation $[a_X, b_X, c_X, d_X]$, and $\vec{N}_X$ is the normal vector $[a_X, b_X, c_X]$. If $K(A,B)$ is positive (negative), this means that A lies in the positive (negative) direction of B. Otherwise, A and B are concident.

If F_1 and F_2 are coplanar, we check whether they overlap within the common plane using the algorithm by Möller [10]. The substrate of this algorithm is to verify on which side of an edge E a point V lies. We implement it using plane-based geometry by substituting E with its corresponding bounding plane S_E and then computing the sign of distance from V to S_E.

4.2 Intersection Recording

The information of intersection is stored in the intersected faces where the intersection occurs. Because the intersection is generated by two (or more) faces, both (all) faces are intersected faces and store the same copy of intersection information.

Each intersected face maintains two important lists for further processing: *cross list* and *coplanar list*. The cross list stores the end points of intersections, the associated faces of the intersections, and the primitives to which these associated faces belong. The coplanar list is relatively simple. It only stores coplanar faces and the primitives to which these coplanar faces belong.

Moreover, in order to reconstruct geometry connectivity and to avoid coincident vertices in the final model, coincident points detected during intersection computation are merged. A *point list* for each intersected face F is used for this purpose. The list stores all the points inside F or on the boundary of F. When introducing new vertices into the face (e.g., recording the new intersection line into the cross list, the two end points of the intersection line segment are newly introduced vertices), we determine if there exists a coincident point in the point list. Coincident points are merged. In addition, because the points on the edges of F are shared by adjacent faces, these points should be added to the corresponding point lists of both two adjacent faces.

There is a degenerate condition for intersection recording: F_1 and F_2 may intersect on a single point. We treat this

condition as F_1 and F_2 being non-intersecting but add the intersected point into the point lists of both faces. Another point to notice is that if F_1 and F_2 are coplanar and overlap, F_1 is added to the coplanar list of F_2 and vice versa. No new point is introduced under this condition.

5 Face Classification

In this step, we classify faces one by one to generate the final mesh. The intersection information computed in the previous section is used for re-tessellation and space label computation. To better explain our approach, we firstly introduce the background knowledge. Then the framework of this step is outlined. Afterwards, several important techniques in this step are detailed.

5.1 Background

We firstly introduce the space labels, which comprise the basis of face membership classification. The space label of face F with respect to primitive M, denoted as $L_F(M)$, is the relative location of F with respect to M. Also, we use $L(M)$ to denote the space label with respect to M which does not specify a face. In general, space label has four conditions: completely inside (*In*), completely outside (*Out*), on the boundary (*On*) or not available (*N/A*). In the last condition, F crosses the boundary of M. Accordingly, it does not have a uniform label. In addition, if the normal direction of F is considered, there are two derived conditions of *On*: the normal points to the outside (*Same*) and those to the inside (*Oppo*) of M.

The evaluation of Boolean operations between B-reps primitives A and B can be converted to the problem of surface selection according to the labels [33]:

$$\begin{aligned} &A \cup^* B : \{F_A\ Out\ B\} \cup \{F_B\ Out\ A\} \cup \{F_A\ Same\ B\}, \\ &A \cap^* B : \{F_A\ In\ B\} \cup \{F_B\ In\ A\} \cup \{F_A\ Same\ B\}, \\ &A -^* B : \{F_A\ Out\ B\} \cup \{(F_B\ In\ A)'\} \cup \{F_A\ Oppo\ B\}, \end{aligned} \tag{2}$$

where F_X is the faces of X, and F' means F with an inverted normal. The stars (*) after the operation notations ($\cup, \cap, -$) mean that the Boolean operations are regularized.

Now, consider computing $L_F(T)$, where T is a CSG solid T with n primitives $\{M_i \mid i = 1, 2, 3, ..., n\}$. Then there are a total of n space labels for F: $\{L_F(M_i) \mid i = 1, 2, 3, ..., n\}$ (or in vector form $\vec{L}_F$). If all elements in $\vec{L}_F$ are known and none of them is *N/A*, $L_F(T)$ can be easily computed by traversing the whole CSG tree from bottom to top and progressively combining the space labels of the CSG nodes according to combination rules [14]:

$$\begin{aligned} X \cup Out \Rightarrow X, &\quad X \cap Out \Rightarrow Out, \\ X \cap In \ \Rightarrow X, &\quad X \cup In \Rightarrow In, \\ X \cup X \Rightarrow X, &\quad X \cap X \Rightarrow X, \\ Same \cup Oppo \Rightarrow In, &\quad Same \cap Oppo \Rightarrow Out, \end{aligned} \tag{3}$$

where X is an arbitrary label value. Note that the combination rules for difference operation are not shown here, because the difference operation can be converted into the intersection operation using De Morgan's transformations:

$$\begin{aligned} &A - B \Rightarrow A \cap B^c, \qquad (A^c)^c \Rightarrow A, \\ &(A \cap B)^c \Rightarrow A^c \cup B^c, \quad (A \cup B)^c \Rightarrow A^c \cap B^c, \end{aligned} \tag{4}$$

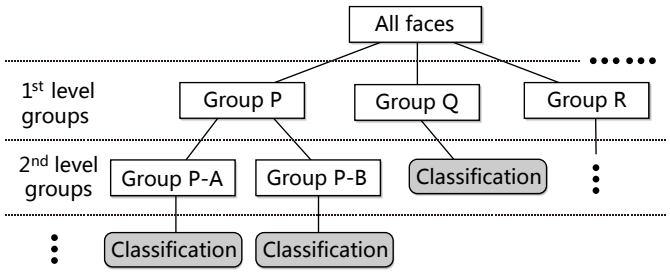


Fig. 7. Illustration of multi-level face grouping framework. Faces are grouped into smaller groups in each level. The trimmed CSG tree of a specific group is obtained by trimming the tree of its parent according to common labels of this group. When the group is not further subdivided, faces in this group are respectively classified based on the trimmed CSG tree of this group.

where X^c denotes the complement of X. If and only if $L_F(T) = Same$, F lies on the boundary of the final model and has the correct normal; accordingly, F belongs to the final result.

If some elements in $\vec{L}_F$ are *N/A* or cannot be computed, we may be unable to compute $L_F(T)$. However, the CSG tree T can be trimmed into a smaller one with fewer primitives to reduce the complexity of further evaluation. For example, assume we have a CSG with its Boolean expression $M_1 \cup (M_2 \cap M_3 - M_4)$. Given the values of two labels $L(M_1) = Out$, $L(M_2) = In$, the expression can be rewritten as $Out \cup (In \cap M_3 - M_4)$. Using the combination rules we can simplify the expression as $M_3 - M_4$. Thus, any face that shares the same two labels $L(M_1)$ and $L(M_2)$ can be classified based on the trimmed CSG tree $M_3 - M_4$, which has only two primitives.

5.2 Face Classification Framework

Space labels of neighboring faces are often coherent. That is, given two neighboring faces F_1 and F_2, $L_{F_1}(M)$ and $L_{F_2}(M)$ are often the same. Therefore, we can group together neighboring faces and evaluate them based on the smaller CSG tree, which is trimmed by the common labels. The trimmed tree contains fewer leaves, thus enabling faster computation. If face grouping can be efficiently performed, this strategy will greatly accelerate the process of face membership classification. Moreover, a multi-level grouping scheme (Fig. 7) may provide even greater capability to shared labels. By using such a scheme, faces are first grouped into first-level groups; then, faces of the same first-level groups are further divided into second-level groups, and so on. Labels are shared in each level and the original CSG tree is trimmed step by step according to common labels in each level. We developed this idea into a novel classification algorithm that uses a two-level grouping scheme. The first level is the *inter-primitive* level; the second is the *intra-primitive* level.

Before we begin face classification, pre-computation is conducted to make the subsequent analysis easier. The original CSG tree is converted into a *positive* tree using the method by Rossignac and Voelcker [34]. 'Positive' means that the CSG tree contains no difference (-) operation. According to Equation 4, there may be some primitives with complement notation. The benefit is that we are not required

to consider the difference operations in the following computation.

The classification starts from face grouping in the inter-primitive level. In this level, faces of the same primitives are simply grouped together. Then, we give a rough estimation of the common labels of the group. Suppose the group is primitive M. Its face labels with respect to primitives whose bounding boxes are totally outside of M's bounding box must be Out (for primitives with complement notations, In). The original CSG tree is trimmed into the first trimmed CSG tree (as short as the *the first trimmed tree*) according to these common labels plus an extra label $L(M)$ whose value is obviously $Same$.

After that, the inter-primitive groups are further divided into smaller intra-primitive groups. To explain how to perform intra-primitive grouping, we firstly introduce an essential concept—intersection primitive (IP). The IPs of a face F is defined as the primitives that F intersects, excluding the primitive to which F belongs (denoted as M). Primitives (excluding M) that do not intersect F are called non-IPs. The set that contains all IPs of F is the IP-set of F. That set can be computed by enumerating all the primitives in the cross list and coplanar list of F.

Now, we require that the faces of the same intra-primitive group meet two requirements: they should be connected together and have the same IP-set. These two requirements give these faces very good properties, as discussed in Proposition 1. The labels referenced in Proposition 1 are used to trim the first trimmed tree into the second trimmed CSG tree (as short as *the second trimmed tree*). For a specific intra-primitive group, the primitives of its second trimmed tree is a subset of the common IP-set of that group. This is because all space labels with respect to non-IPs are shared within the intra-primitive group and all these labels are either In or Out. Therefore, all non-IPs must have been trimmed from the CSG tree (to be explained in Section 5.3).

Proposition 1. If a group of faces are connected together and they have the same IP-set, they must share the same labels with respect to all non-IPs. These labels are either *In* or *Out*.

Proof. We only prove that the conclusion stands for adjacent faces, which is a special case of connected faces. Promoting the conclusion to the general situation is obvious. Assume M is a non-IP and label $L(M)$ is different for the two adjacent faces A and B . Then there must be faces from M between A and B to cause the different space labels. Additionally, because A and B are adjacent, at least one of A and B intersects the faces from primitive M. This contradicts the assumption that M is non-IP. Therefore, $L(M)$ must be the same for A and B. Moreover, because M is non-IP, $L(M)$ is either *In* or *Out*.

Because faces of the same intra-primitive group are connected together, we can efficiently group them using a flood filling algorithm. When visiting a face, its space label vector is instantly computed and its classification is performed. Our intra-level grouping is performed as follows:

1) **First seed generation.** A triangle face F_{S0} is randomly chosen as the first seed (e.g., the red triangle in Fig. 8). Labels of F_{S0} with respect to non-IPs are determined by using F_{S0}'s barycenter as the sample point, through a ray-shooting algorithm [35] and by employing our octree as a spatial search structure [36] to speed up. Note that because the original CSG tree is trimmed into the first trimmed tree, we only compute labels with respect to primitives within that first trimmed tree.
2) **Label propagation.** Once there is a seed F_S with L_{F_S} with respect to all non-IPs, we define the intra-primitive group G_S as follows. All faces that belong to G_S should have the same IP-set as F_S. The common labels of G_S are L_{F_S} with respect to all non-IPs. The first trimmed tree is trimmed into the second trimmed tree according to these common labels. Breadth-first flood filling is used to find all faces that belong to G_S. When visiting a face, we first check if it has the same IP-set as the seed. If so, set membership classification is performed based on the second trimmed tree of G_S (detailed in Section 5.4). Otherwise, it is pushed into a candidate list for Step 3 (e.g., the black faces in Fig. 8 (left)).
3) **Secondary seed generation.** When the flood filling ends, which means no more faces belong to the current face group, a new seed is picked from the candidate list (e.g., the stippled faces in Fig. 8). Then a new group is defined by the new seed and the labels with respect to non-IPs of the new seed are computed (details in Section 5.5). After that, Step 2 is repeated according to the newly defined group, until that candidate list is empty, indicating all faces of M are classified.

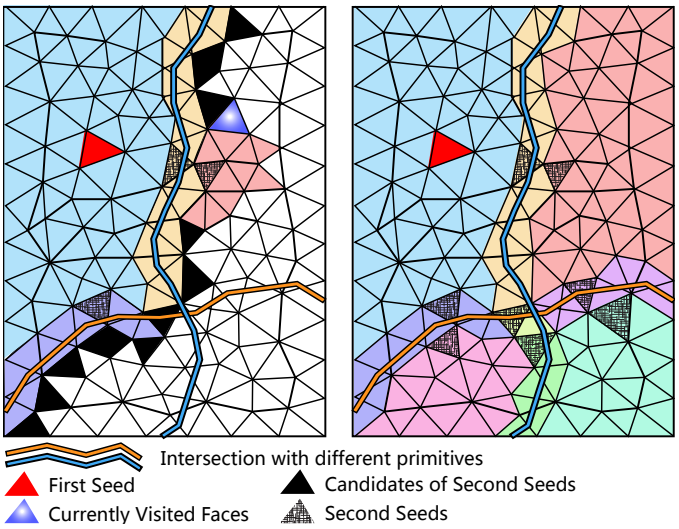


Fig. 8. A snapshot during flood filling (*left*), and the final grouping result (*right*). Different intra-primitive groups are distinguished by different colors. From the final grouping result, the whole mesh is divided into 9 intra-primitive groups.

The face membership classification of the whole CSG is accomplished by performing the above process for each primitive. Procedure 1 shows the framework as pseudo-code. To avoid repetitive visits during flood filling, we use an extra flag for each face to record its visiting status.

5.3 Trimming CSG Tree

Trimming a CSG (binary) tree with the In or Out label is relatively simple. According to the first two lines of

Procedure 1 FaceClassification(PrimitiveList, PosTree)

```
for each M_i in PrimitiveList do
  Get common labels CL^1 of M_i;
  Trim PosTree into TrimTree1 by CL^1;
  Select the first seed F_0 from M_i;
  Put F_0 into CandidateList;
  while CandidateList is not empty do
    Pick a seed F_S;
    Define group G_S, where F_S ∈ G_S;
    Compute the common labels of F_S, denoted as CL^2;
    Trim TrimTree1 into TrimTree2 according to CL^2;
    Put F_S into FloodFillQueue;
    while FloodFillQueue is not empty do
      Get the next face F_i from FloodFillQueue;
      Do set membership classification of F_i based on TrimTree2;
      for each adjacent face F_Neighbor do
        if F_Neighbor belongs to G_s then
          Put F_Neighbor into FloodFillQueue;
        else
          Put F_Neighbor into CandidateList;
        end if
      end for
    end while
  end while
end for
```

Equation 3, given a primitive M, if $L(M)$ is In (Out), the label of M's parent node is either In (Out) or the value of its sibling. In either condition, the node representing M can be trimmed from the tree.

However, when trimming the original tree into the first trimmed tree for a specific inter-primitive group, a special label, $Same$, exists. This label means that the faces of the group are on the boundary of the primitive to which they belong, which is absolutely correct. This label is not easy to process because we cannot determine the label of its parent node unless we know the label of its sibling. Unfortunately, during tree trimming, the labels we have are incomplete. We may have to leave the $Same$ label unprocessed and wait for further information. This incurs considerable computation burden to check this special condition and makes the trimming algorithm inefficient. To solve the problems, we developed a new representation of the CSG tree, called CSGlist, to minimize the side effect of the special $Same$ label and enable early rejection of face groups that do not belong to the final model of the CSG.

Given a face group from primitive M, we construct the CSGlist as follows. We cut the original CSG tree into a set of subtrees along the *critical path*, which is the path from M to the root (e.g., when $M = D$, the construction of CSGlist is shown in Fig. 9). Obviously, none of the subtrees contains the primitive M. Each subtree of CSGlist is associated with 'desired' label values. That is, if the labels with respect to all subtrees meet their corresponding desired values respectively, the labels with respect to the original CSG tree is $Same$. On the contrary, if the labels with respect to any subtree are not desired, the labels with respect to the whole CSG tree is not $Same$; thus the related face group should be dropped. The desired value of a subtree is determined based on how the subtree connects to the critical path. According to the face selection rules (Equation 2), if the subtree connects to a Union ($\cup$) node, then the desired value is In; otherwise, the desired value is Out. In addition, suppose the primitive A in Equation 2 is the left child. Then if the subtree is in the left child of its parent node, $Same$ is also desired.

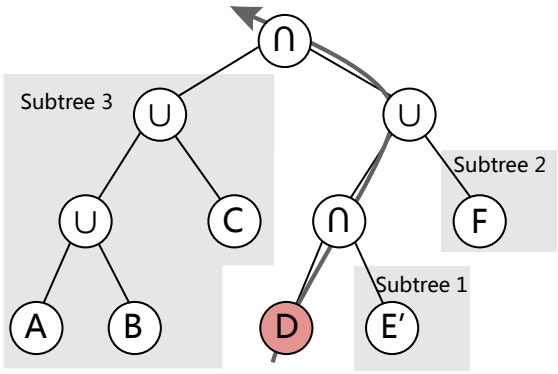


Fig. 9. Convert CSG tree into a CSGlist along the critical path (marked by the arrow) from primitive D to the root. The CSGlist contain three subtrees. Each subtree has its desired label value. Subtree 1: Out or $Same$. Subtree 2: In or $Same$. Subtree 3: Out.

Conversion to CSGlist is performed before the first trimming. After that, all the labels we use for CSG tree trimming are either In or Out. Both the first and the second trimmed trees are represented by CSGlists. The subtrees of the CSGlist can be easily trimmed (since there is only In and Out labels) and their desired values are used to enable early rejection.

5.4 Classifying a Single Face

If a face is non-intersected, the second trimmed tree will have single value, which can be used to determine whether the face is accepted. However, if it is an intersected face, further processing is needed.

Intersected faces usually cannot be classified as a whole. They have to be tessellated into smaller non-intersected triangles, each of which is classified respectively. Robustness and efficiency are our main consideration. For these purposes, we nest the exact BSP structure [2], [6] into the intersected faces. We classify intersected face F as follows.

Valid-IPs & Pseudo-IPs IPs of an intersected face are firstly divided into two types according to whether the IPs are in the second trimmed tree. The IP that is the primitive of the second trimmed tree is called the valid intersection primitive (valid-IP). Otherwise, it is called the pseudo-intersection primitive (pseudo-IP). Pseudo-IPs intersect the face; however, the labels with respect to pseudo-IPs do not affect the classification of F. In other words, the intersections of pseudo-IPs are not edges of the final models, as illustrated in Fig. 10. Thus, we simply omit these pseudo-intersections during classification. This filtering stage saves computation time by avoiding unnecessary splitting of faces. An extreme instance is that there is no valid-IP. For this condition, the face is processed in the same way as a non-intersected face.

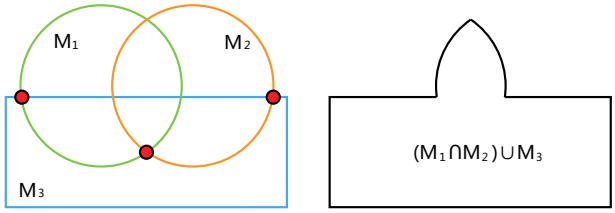


Fig. 10. 2D illustration of pseudo-intersection. The vertices in red do not appear in the final model. They are so called pseudo-intersection vertices (line segments in the 3D case).

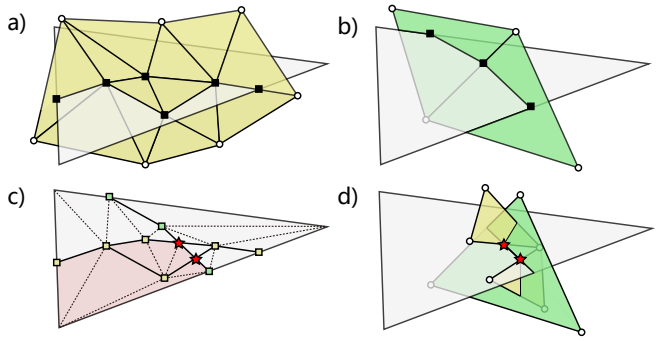


Fig. 11. We classify the face F from primitive M: (a) faces of M_1 (yellow) that intersect F. BSP of M_1 is constructed according to these. (b) Faces of M_2 (green) that intersect F. BSP of M_2 is constructed according to them. (c) If M_1 and M_2 are the only valid-IPs of F, then triangulation can be performed according to intersection lines recorded in the cross list. The red area shows a possible result of sub-triangle classification. Note that there are intersections between lines from different primitives (marked as red stars), which are (d) intersections of three faces each from different primitives.

Tessellation In this step, F is divided into a set of sub-triangles using the intersections recorded in Section 4. Because constrained Delaunay triangulation (CDT) [32] has good topology characteristics, we convert the tessellation into a problem of CDT. The zone of CDT is F, and the constraints are all the intersections by valid-IPs. The constraints ensure that no sub-triangles will cross these intersection lines. Because CDT in our method is a vertex-based 2D problem, the 3D coordinates computed by their plane-based representation are projected on a 2D space, which is the axis-aligned plane where the area of F is maximized.

It should be noted that intersections from different valid-IPs may intersect (Fig. 11(c)). Omitting these intersection points may cause false results of CDT. To obtain the correct constraints, intersections from different valid-IPs are tested to determine whether they intersect. We implement this test again using plane-based geometry for robustness. As we know this kind of special intersection is actually the cross point of the three faces: F, and the two associated intersected faces from the valid-IPs (Fig. 11(d)). Therefore, such a point can be represented by the three supporting planes of the three faces. We determine whether the cross point exists by testing if the point is inside all three of these faces. By adopting the routine of the relative position of a point to a plane [11], this test is robust and efficient. If the cross point of different intersection lines is detected, these lines are split by the cross point and the constrains of CDT are updated accordingly. Moreover, because the cross point is actually shared by the three faces, it is added to the point lists (Section 4.2) of all three of these faces.

Classification of sub-triangles An intersected face F is divided by faces from each valid-IPs into In cells, Out cells and On cells (Figs. 6(a) and (b)). The BSP tree [25] is very suitable for describing such partitions. The labels of sub-triangles can be efficiently computed according to a BSP representation using the BSP-based point-in-polyhedra test [37], [38], [39]. Details are discussed as follows.

Given a valid-IP, denoted as M, we firstly build a BSP tree nested within F according to the intersections between F and M. This is a 2D partition problem. Nevertheless, constructing BSP in 2D space of F will introduce errors on account of the involved conversion from plane triples (plane-based representation) into 2D coordinates. Therefore, it is better to construct the BSP tree in 3D space. If we limit the space of the BSP construction zone within the face F with tiny thickness, denoted as P_F, it is obvious that all faces from M that intersect P_F are recorded in the cross list and the coplanar list of F. These faces give the correct space partition on P_F. We construct the 3D BSP tree according to the intersected faces. The construction of the BSP tree is implemented using an exact plane-based algorithm proposed by Berstein and Fussell [6]. Now, given a sub-triangle SF, the label $L_{SF}(M)$ is then computed based on the classification results of the three corners of SF according to the constructed BSP. If all three classification results are On, then $L_{SF}(M)$ is either $Same$ or $Oppo$. If any classification of the corners is not On, $L_{SF}(M)$ is the same as the non-On classification .

After all space labels of a sub-triangle are computed, we judge whether the sub-triangle belongs to the final model based on the second trimmed tree represented as CSGlist. The subtrees in CSGlist are evaluated using the combination rules of labels. The sub-triangle is accepted if and only if the evaluation results of all subtrees are desired. We might accelerate this step if the coplanar list of $F$is empty or the faces in the coplanar list are all from pseudo-IPs. In this case, the labels with respect to valid-IPs are either In or Out, and each subtree of CSGlist can be efficiently evaluated using Blist, which was proposed by Hable and Rossignac [40]. For CSG models in which the coplanar cases are rare, this acceleration saves considerable computation time.

5.5 Secondary Seeds Generation

As discussed in Section 5.2, during flood filling, faces belonging to the group of the current seed are classified according to the second trimmed tree of this group. The rest are added to a candidate list for generating secondary seeds when no more faces belong to the group of the current seed. The main task of secondary seed generation is to compute the space labels with respect to non-IPs of the new seeds.

A straightforward scheme is to use the same approach as the first seed generation in Section 5.2. However, this scheme requires recalculating all the space labels with respect to non-IPs using the time-consuming ray-shooting method. For complex CSG evaluation, in which secondary seed generation is frequently requested , we need a more

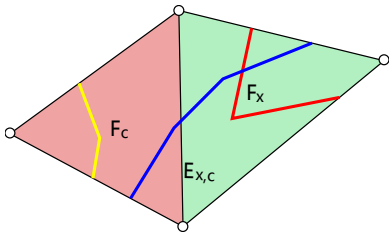


Fig. 12. F_x is the secondary seed, and F_c is its associated face. The labels of F_x with respect to primitives that intersect F_c but do not intersect F_x (e.g., the yellow intersection) can be computed using any point in $E_{x,c}$ as the sample. The labels of F_x with respect to primitives that intersect neither F_x nor F_c are the same for both faces. Line segments of different colors are intersection lines by different primitives.

efficient solution. Again, we use the information of neighboring faces to accelerate this process. When a face F_x is added to the candidate list, we associate the following data with F_x: the face F_c that is currently being visited (therefore F_x's neighbor; see Fig. 12) and the labels of F_c with respect to F_c's non-IPs. Note that the non-IP set of F_x, denoted as C_x, is different from that of F_c, denoted as C_c.

Now, if F_x is selected as a new secondary seed, we can compute the labels with respect to non-IPs of F_x in two complementary ways: we can directly inherit from the labels of F_c, or represent F_x with a point within F_c and compute the label of this point using the BSP technique presented in Section 5.4. The former is used for the computation of the labels with respect to the primitives that are in $C_c \cap C_x$. The labels with respect to these primitives are the same for both F_x and F_c. The latter is used for computing the labels with respect to the remaining of primitives (e.g., those in $C_c \setminus C_x$). According to Proposition 2, the sample point of F_x can be chosen as any point in the common edge of F_x and F_c, denoted as $E_{x,c}$. For robustness, we use one of the end points of $E_{x,c}$ as the sample point because we have its exact plane-based representation.

Proposition 2. Given two adjacent faces F_c and F_x, the labels with respect to primitive M for F_x is the same as the classification result of any point on the common edge of F_x and F_c in the BSP tree of M for F_c (constructed in Section 5.4), if M is in the IP-set of F_c but not in the IP-set of F_x.

Proof. We know M is not in the IP-set of F_x; therefore, it is obvious that all points on F_x have the same labels with respect to M. Because the points on the common edge of F_x and F_c are in F_x, they have the same labels.

6 Experiments and Discussion

We implemented the proposed method in C++ and tested a series of examples with numerous primitives and/or faces on a consumer laptop with an Intel i5-4200D processor at 1.5GHz and with 8 GB of RAM. The OpenMesh library [41] is used for storing triangle meshes and supporting the query of neighboring faces. Tessellation of critical triangles is performed by constrained Delaunay triangulation using the Fade2D library [42].

We compared four state-of-the-art Boolean evaluation techniques, including CGAL [12], Maya 2015 [13], the method proposed by Campen and Kobbelt [2], and the approach proposed by Feito et al. [5]. The latter two methods used localized schemes. Table 1 shows the comparison in terms of computing time. It can be easily to seen that our approach was efficient in all these examples. Feito et al.'s method had performance relatively similar to ours for the examples with a single Boolean operation (Examples 6, 7). This is because both the approaches use a face-based grouping scheme to accelerate face membership classification; thus have similar pipelines for single Boolean operations. However, when the number of primitives grows, the advantage of our non-incremental CSG evaluation approach becomes more obvious. In the examples of more than two primitives, our method is much faster than any other methods. In particular, our method worked out example 2 which consisted of 801 primitives, in 15 s, which was more than 100 times faster than the Boolean evaluation in Maya.

Among the three main steps of our approach, the first step of the octree construction is a typical $O(n\ log\ n)$ algorithm, where n is the total number of faces in the CSG. The second step of intersection computation is only performed within critical octree cells. The number of critical cells is often linear to the number of intersected

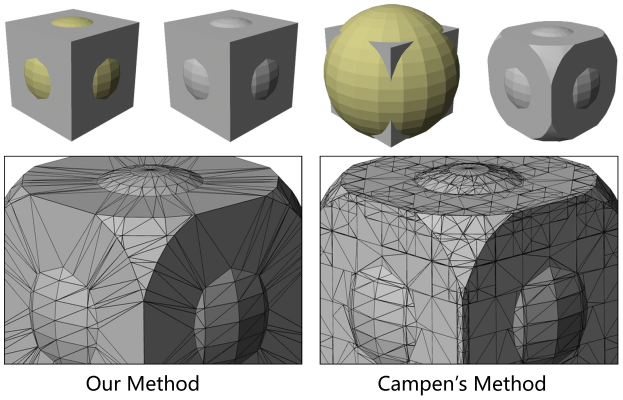


Fig. 13. *Cube* ∪ *Sphere1* ∩ *Sphere2*: our method resulted in simpler mesh connectivity than the method by Campen and Kobbelt [2].

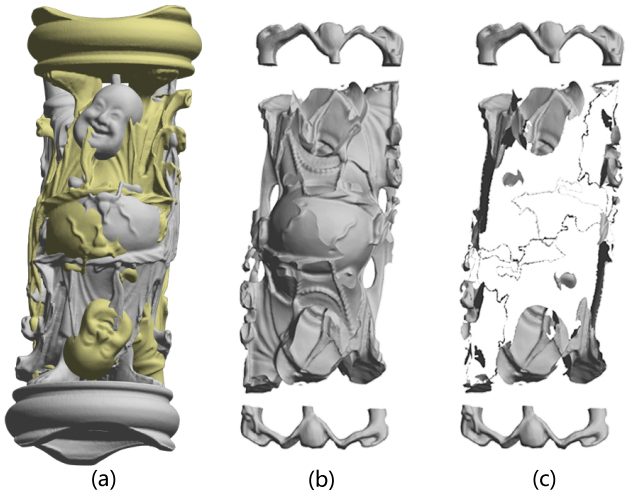


Fig. 14. Boolean operations on model *Budda*: (a) two *Buddas*, (b) results of *Budda* ∩ *Budda*, and (c) the incorrect result of *Budda* ∩ *Budda* generated by non-robust implementation of Feito et al.'s method [5].

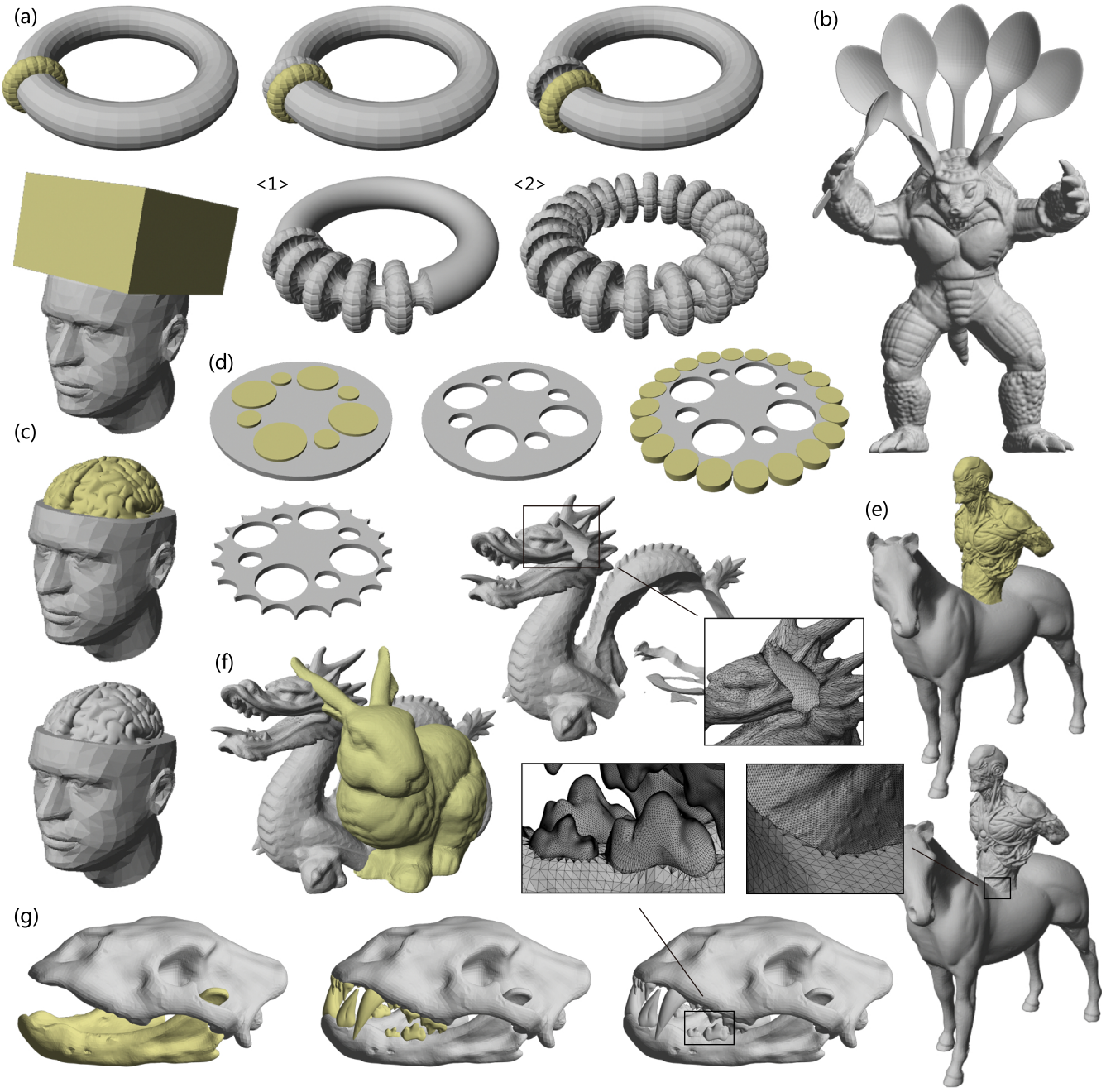


Fig. 15. (a) Boolean operations between a *Ring* and a set of *Spheres*. ⟨1⟩ after 200 operations. ⟨2⟩ after 800 operations. (b) Union of *Armadillo* and six *Spoons*. (c) *Head* - *Cuboid* + *LeftBrain* + *RightBrain*. (d) Difference a *BigCylinder* and 28 *SmallCylinders*. (e) *Man* ∪ *Horse*. (f) *Dragon* - *Bunny*. (g) *Palate* ∪ *Mandible* ∪ (20 *Teeth*).

faces k (often an $O(\sqrt{n})$ number) in the CSG. Therefore, the computation complexity of the second step is almost $O(k)$. The condition of the third step of face membership classification is more complex. Although the flood filling algorithm used for intra-primitive grouping is $O(n)$ complex, classifying an intersected faces could consume much more time than classifying a non-intersected faces. Therefore, if we assume classifying an intersected faces spends θ times of computation time to classify a non-intersected face, we may generalize the time complexity of this step as $O(n + \theta \cdot k)$. When k/n is constant, the third step is $O(n)$ complex.

To further validate that our approach is efficient for non-incremental CSG evaluation, we analyzed how the computation time changes with respect to the number of primitives. Table 2 shows the evaluation time for Fig. 15 (a), which was a CSG consisting of a ring and hundreds of identical spheres. We changed the number of spheres from 100 to 800 and recorded the evaluation times. In this condition, the k/n was relatively constant, and as there are many intersected faces, the bottleneck lies in Step 2 and Step 3. The result shows that the computation time of our algorithm had a linear behavior, which conforms to our previous discussion. The computation time of other incremental approaches rapidly increased when the num-

TABLE 1
Computation Time Statistics (Seconds)

No.	Figure	Face Num.$^{\nabla}$			Primitive	CGAL	Maya	Campen	Feito	Our Approach			
		Total	Min	Max	Num.	[12]	[13]	[2]	[5]	Total$^{\circ}$	Step 1	Step 2	Step 3
1	15(a)	37.6K	180	1.6K	201	1350	49.9	TLE†	15.9	**3.13**	0.328	1.69	0.922
2	15(a)	146K	180	1.6K	801	TLE	1400	TLE	Fail*	**12.8**	1.36	6.41	4.00
3	15(b)	377K	2.56K	346K	7	TLE	224.5	45.7	11.7	**1.44**	0.438	0.062	0.563
4	15(c)	79.2K	12	38.2K	4	97.6	Fail	3.58	1.69	**0.563**	0.187	0.141	0.172
5	15(d)	36K	800	2.4K	29	34.1	7.03	18.7	4.05	**0.313**	0.094	0.094	0.081
6	15(e)	597K	96.9K	500K	2	TLE	38.3	26.4	5.63	**2.13**	0.563	0.031	0.984
7	15(f)	170K	69.7K	100K	2	218	7.45	13.8	2.39	**0.891**	0.328	0.125	0.266
8	15(g)	362K	6.91K	276K	22	TLE	344	256	19.2	**1.80**	0.766	0.172	0.593
9	14	2.16M	1.08M	1.08M	2	Fail	Fail	MLE$^{\bullet}$	Fail	**11.3**	5.55	1.36	3.01

$^{\nabla}$ *Min* (*Max*) means the minimum (maximum) number of faces of a single primitive.
$^{\circ}$ The total computation time of our method includes the construction of half-edge structure and the three steps in Section 3.
* *Fail* means nothing was returned, or we received the wrong evaluation results from the programs.
† *TLE* means the processing time was more than 2,000 s.
$^{\bullet}$ *MLE* denotes that the program was out of memory. The memory limit for our implementation of Campen et al.'s approach was around 1.2GB.

TABLE 2
Number of Primitives and Computation Time (Seconds)

Number of Spheres	100	200	300	400
Our Approach	**1.38**	**2.89**	**4.39**	**5.97**
Maya [13]	17.4	53.4	124	241
CGAL [12]	439	1350	-	-
Feito [5]	5.20	15.9	30.4	-
Number of Spheres	500	600	700	800
Our Approach	**7.90**	**9.49**	**11.0**	**12.8**
Maya [13]	407	628	966	1400
CGAL [12]	-	-	-	-
Feito [5]	-	-	-	-

* This table was the computation time of the evaluation for Fig. 15 (a).

ber of primitives increased, sometimes showing an $O(n^2)$ behavior. This is because for those approaches, as more spheres were added to or subtracted from the ring, the mesh of the ring became more complex, which made it more difficult to perform subsequent Boolean operations.

One possible problem of a non-incremental CSG evaluation approach is memory inefficiency. Unlike incremental ones, non-incremental approaches (including ours) often require loading all the primitives into the memory before the evaluation. Fortunately, our approach does not significantly suffer from such a problem. Our implementation only constructs an octree, which needs typically an $O(n\ log\ n)$ size data structure, and some auxiliary information of intersected faces, which is usually $O(k)$ size. According to our experiment, our approach performed full evaluations in Table 1 with less than 600MB. On the contrary, some incremental approaches can be very memory inefficient. For instance, in the evaluation of Example 6, CGAL consumed more than 6 GB of memory. Furthermore, in the evaluation of Example 2, which contained only 146 K faces, Maya 2015 used at least 5 GB of memory.

Our method is robust and could correctly evaluate all the examples in Table 1. On the other hand, it is evident that the commercial software Maya could not even correctly evaluate Example 4, which contained only 38.6 k faces and four primitives. CGAL gave warnings when evaluating Example 9 and finally terminated without offering a result. Our implementation of Feito et al's method gave an incorrect result for Example 9 (Fig. 14(c)) and a result rife with small holes for Example 2.

Topology simplicity is another significant characteristic of the quality of CSG evaluation. As shown in Fig. 13, our method led to less triangles to represent the final model than Campen et al.'s method without stages for surface merging. A simpler topology is preferred because it is easier to process. In fact, we found that it was owing to such inefficient tessellation that Campen et al.'s method took so long time to evaluate Examples 1 and 2. After hundreds of Boolean operations, the resulting surface became so fragmented that it could take tens of seconds to perform a single Boolean operation.

Limitations & Future work The major limitation of our method is that it is not unconditionally robust. Because the tessellation of intersected faces is performed using vertex-based representation, the computed vertex coordinates may shift from their real location, which may lead to incorrect tessellation in theory. From our tests, we have not found the visible failures caused by this problem, because that the coordinate errors in most cases are small without visible incorrect tessellation, we will investigate to integrate plane-based geometry computation into intersected face tessellation in the future work.

7 Conclusion

In this paper, we proposed a novel method to evaluate CSG with triangular mesh primitives. It is able to efficiently perform non-incremental evaluation of large CSG with massive faces. The key idea of our approach is to use the local coherence of face space labels to accelerate face membership classification. A two-level grouping framework is developed to group neighboring faces together and thus space labels

can be shared within each group. This scheme saves much time for space label computation, which is often very time-consuming for conventional Boolean evaluation algorithms. Additionally, in order to be robust, plane-based geometry computation is introduced into the intersection computation step of our approach. Experiments have verified the proposed approach is more efficient than the state-of-the-art techniques while retaining robustness and stability. We will further investigate the robust tessellation in future.

REFERENCES

[1] A. Requicha, "Mathematical models of rigid solid objects," 1977.

[2] 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.

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

[4] 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.

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

[7] J. Hable and J. Rossignac, "Cst: Constructive solid trimming for rendering breps and csg," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 5, pp. 1004–1014, 2007.

[8] J. Rossignac, "Ordered boolean list (obl): Reducing the footprint for evaluating boolean expressions," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 9, pp. 1337–1351, 2011.

[9] 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.

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

[11] 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.

[12] P. Hachenberger and L. Kettner, "3D boolean operations on nef polyhedra," in *CGAL User and Reference Manual*, 4.6 ed. CGAL Editorial Board, 2015. [Online]. Available: http://doc.cgal.org/4.6/Manual/packages.html#PkgNef3Summary

[13] Autodesk. (2014) Autodesk maya 2015. [Online]. Available: http://www.autodesk.com.hk/products/maya/overview

[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] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] J. R. Shewchuk, "Lecture notes on geometric robustness," in *Interpola-tion, Conditioning, and Quality Measures. In Eleventh International Meshing Roundtable*, 1999.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] H. Chen and S. Fang, "A volumetic approach to interactive csg modeling and rendering," in *Proceedings of the fifth ACM symposium on Solid modeling and applications*. ACM, 1999, pp. 318–319.

[30] E. Eisemann and X. Décoret, "Single-pass gpu solid voxelization for real-time applications," in *Proceedings of graphics interface 2008*. Canadian Information Processing Society, 2008, pp. 73–80.

[31] 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.

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

[33] K. Kuratowski and A. Mostowski, *Set theory*. Elsevier, Academic Press, 1968.

[34] J. R. Rossignac and H. B. Voelcker, "Active zones in csg for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms," *ACM Transactions on Graphics (TOG)*, vol. 8, no. 1, pp. 51–87, 1988.

[35] V. Havran, "A summary of octree ray traversal algorithms," *Ray Tracing News*, vol. 12, no. 2, pp. 11–23, 1999.

[36] 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.

[37] D. Gordon and S. Chen, "Front-to-back display of bsp trees," *IEEE Computer Graphics and Applications*, vol. 11, no. 5, pp. 79–85, 1991.

[38] H. Fuchs, Z. Kedem, and B. Naylor, "On visible surface generation by a priori tree structures, conf," in *Proc. of SIGGRAPH*, vol. 80, no. 14, 1980, p. 3.

[39] A. James, "Binary space partitioning for accelerated hidden surface removal and rendering of static environments," Ph.D. dissertation, University of East Anglia, 1999.

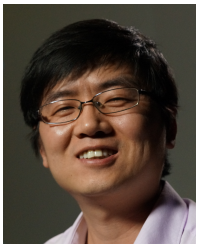
[40] J. Hable and J. Rossignac, "Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 1024–1031.

[41] RWTH Aachen University. (2015) Openmesh 3.3. [Online]. Available: http://www.openmesh.org/

[42] Geom Software. (2015) Fade2d 1.24. [Online]. Available: http://www.geom.at/products/fade2d/

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.