EMBER: Exact Mesh Booleans via Efficient & Robust Local Arrangements

PHILIP TRETTNER, Shaped Code GmbH, Germany
JULIUS NEHRING-WIRXEL, Visual Computing Institute, RWTH Aachen University, Germany
LEIF KOBBELT, Visual Computing Institute, RWTH Aachen University, Germany

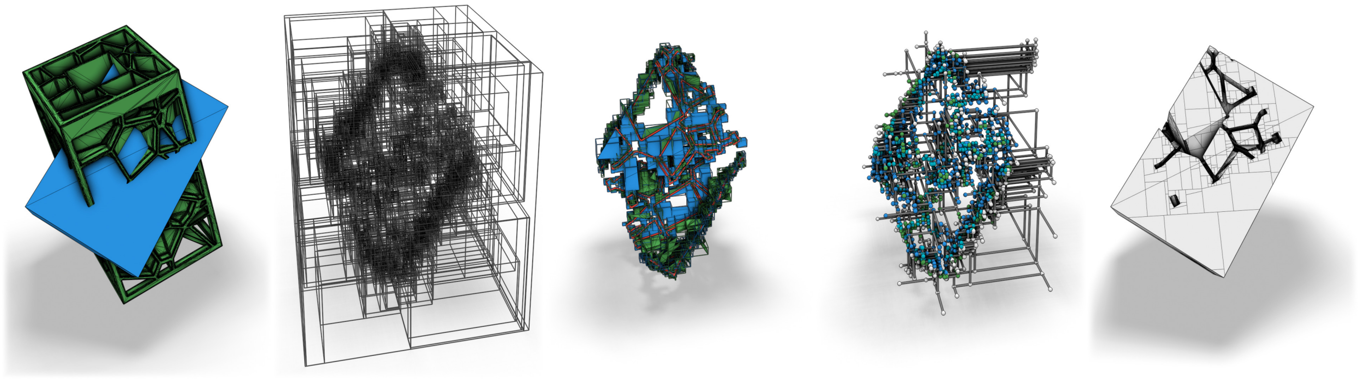


Fig. 1. High-level overview of our approach to mesh Booleans. Our method, "EMBER", performs a single pass of adaptive recursive kd-tree type spatial subdivision while exploiting various early-out pruning criteria. In the leaf nodes, all faces are split into disjoint polygons by pairwise intersection using local BSP trees. These polygons are classified according to their winding numbers via segment traces. Our key contribution towards maximum efficiency is that these winding numbers can be computed locally for each leaf node since we propagate reference points with known winding numbers through the recursive subdivision. All computations are exact due to the use of a plane-based mesh representation with fixed-width homogeneous integer coordinates. This example consists of 1.2 million input triangles and our multi-threaded implementation takes only 34 ms on an 8-core consumer CPU. For comparison, QuickCSG (inexact, [Douze et al. 2017]) takes 1010 ms and Mesh Arrangements (exact, [Zhou et al. 2016]) takes 141 s.

Boolean operators are an essential tool in a wide range of geometry processing and CAD/CAM tasks. We present a novel method, EMBER, to compute Boolean operations on polygon meshes which is exact, reliable, and highly performant at the same time. Exactness is guaranteed by using a plane-based representation for the input meshes along with recently introduced homogeneous integer coordinates. Reliability and robustness emerge from a formulation of the algorithm via generalized winding numbers and mesh arrangements. High performance is achieved by avoiding the (pre-)construction of a global acceleration structure. Instead, our algorithm performs an adaptive recursive subdivision of the scene's bounding box while generating and tracking all required data on the fly. By leveraging a number of early-out termination criteria, we can avoid the generation and inspection of regions that do not contribute to the output. With a careful implementation and a work-stealing multi-threading architecture, we are able to compute Boolean operations between meshes with millions of triangles at interactive rates. We run an extensive evaluation on the Thingi10K dataset to demonstrate that our method outperforms state-of-the-art algorithms, even inexact ones like QuickCSG, by orders of magnitude.

Authors' addresses: Philip Trettner, trettner@shapedcode.com, Shaped Code GmbH, Germany; Julius Nehring-Wirxel, nehring-wirxel@cs.rwth-aachen.de, Visual Computing Institute, RWTH Aachen University, Germany; Leif Kobbelt, kobbelt@cs.rwth-aachen.de, Visual Computing Institute, RWTH Aachen University, Germany.

0730-0301/2022/7-ART39 $15.00
https://doi.org/10.1145/3528223.3530181

CCS Concepts: • **Computing methodologies → Mesh geometry models**; **Mesh models**.

Additional Key Words and Phrases: mesh Booleans, solid modeling, CSG, plane-based geometry, BSP, integer arithmetic, winding number vectors

ACM Reference Format:
Philip Trettner, Julius Nehring-Wirxel, and Leif Kobbelt. 2022. EMBER: Exact Mesh Booleans via Efficient & Robust Local Arrangements. *ACM Trans. Graph.* 41, 4, Article 39 (July 2022), 15 pages. https://doi.org/10.1145/3528223.3530181

1 INTRODUCTION

In solid modeling, a natural basic operation is to compute the union, intersection, or difference of objects. These are called Boolean operators and are required in all flavors of geometry processing and CAD/CAM tasks. As a modeling technique, they form the basis of constructive solid geometry (CSG). Countless applications rely on solid Booleans in one form or another. Milling simulations subtract tool meshes from an initial workpiece, while collision tests implicitly check if intersections of objects are non-empty. In simulations and games, destructible environments often rely on CSG concepts. Virtually all 3D modeling and CAD tools include functionality to build or modify objects using Boolean operators.

In this paper, we focus on arguably the most popular representation of geometry: triangle and polygonal meshes. Even if a mesh processing algorithm does not use Booleans directly, they often have strict input requirements, such as meshes being 2-manifold and watertight. Many pipelines require sophisticated pre- and post-processing to satisfy or restore such requirements. Such mesh repair algorithms have strong similarities to Boolean operators. In fact, a way to fix e.g. self-intersections is to apply a self-union.

Boolean operators on meshes have received much attention and feature a rich history. Achieving solutions both robust and efficient has proven notoriously difficult. Computing intersections, a component of almost any approach to mesh Booleans, is numerically challenging when working in floating-point arithmetic. Small or thin triangles, coplanar or nearly coplanar faces, rays that hit too close to edges or vertices all pose a significant challenge. Round-off errors in these situations easily lead to misclassifications, inconsistent topology, and ultimately incorrect results. In terms of Hausdorff distances, these errors are usually not bounded.

These issues can have a disproportionate impact in automated or unsupervised settings. This happens mainly due to the sheer volume of performed operations and the difficulty or outright inability to recover from such errors. Secondarily, automatically generated CSG operations tend to exhibit patterns, such as regularly spaced translations, that increasingly increase the risk of degenerate configurations. These problems exponentiate if the output is used as input again (an iterated CSG setting). To maintain strong guarantees, numerically exact mesh Booleans are required. The actual methodology varies, but virtually all of these methods use an exact numerical foundation, like arbitrary-precision numbers or exact floating-point predicates. The trade-off is typically a runtime performance impact of multiple orders of magnitude.

Real-world applications often demand exactness and robustness in order to guarantee reliability. At the same time, however, high performance is required, too, which is a conflicting goal. Consider, e.g. the simulation of a CNC milling process where hundreds of thousands of Boolean subtractions have to be computed. Without (faster than) realtime performance, this simulation (required for collision prediction) would significantly slow down the design and fabrication process. While even the most efficient state-of-the-art *inexact* methods do not achieve this performance, our method does and even guarantees exactness of the result.

1.1 Contribution

We introduce EMBER, a method for exact mesh Booleans designed to satisfy these strong requirements of exactness, robustness, and efficiency. From a bird's-eye view, the procedure is conceptually simple (cf. Fig. 1). We recursively subdivide a given set of polygons by splitting across axis-aligned planes. Once the number of polygons is small enough, we compute pairwise intersection segments and integrate those into local, per-polygon BSPs. The resulting set of polygons is classified by tracing winding numbers along a chain of line segments to a local reference position.

Classification itself is based on generalized winding number vectors (WNV) [Jacobson et al. 2013], which we extend to enable our local segment tracing. This allows us to track a local reference position with a known WNV during subdivision. As a result, all leaves of the subdivision can be computed locally and require no global acceleration structure, which greatly boosts our performance. It also allows us to reason about what WNVs can occur during a particular subdivision, leading to efficient early termination criteria.

The mathematical foundation relies on plane-based geometry and the recently introduced homogeneous integer coordinates [Nehring-Wirxel et al. 2021]. Originally developed for 3D BSPs, this paradigm can be implemented extremely efficiently on modern CPUs. We extend their formulation to account for polygons, segments, and intersections of these.

In summary, our method for computing Booleans on polygonal meshes is exact, robust, and extremely performant. More concretely, we contribute:

(1) A subdivision-based approach to mesh Booleans that stays completely local and requires no precomputation.
(2) Polygon classification based on winding number vectors computed by segment traces.
(3) Resolution of pairwise intersections via polygon-local BSPs.
(4) A fast and exact formulation of plane-based geometry for polygonal meshes and segment tracing using integer homogeneous coordinates.

We achieve maximum performance by furthermore exploiting opportunities for early termination and an optimized, multithreaded implementation. Our evaluation on the Thingi10K data set shows that our method is faster than the state-of-the-art by orders of magnitude. To encourage use of our method in further research and applications, an implementation can be acquired on the project page at graphics.rwth-aachen.de/ember-exact-mesh-booleans.

2 RELATED WORK

Mesh Booleans usually fall into vertex- or plane-based approaches and can be exact or inexact. Exact approaches often use arbitrary-precision arithmetic such as GMP [Granlund and the GMP development team 2020] or floating-point predicates such as [Attene 2020; Shewchuk 1997]. [Nehring-Wirxel et al. 2021] recently introduced another alternative based on fixed-width integer homogeneous coordinates allowing high-performance exact predicates and construction. Inexact approaches rely on classical floating-point arithmetic and usually suffer from stability issues with weak guarantees for topological and geometrical correctness. However, they are typically significantly faster to compute.

2.1 Vertex-Based

The common approach for vertex-based methods is to first compute all intersections of the input meshes and then determine which resulting polygons should be part of the output [Barki et al. 2015; Cherchi et al. 2020; Douze et al. 2017; Zhou et al. 2016].

There is a range of approaches that sacrifice algorithmic stability for performance: Most notably, QuickCSG [Douze et al. 2017] uses winding number vectors in combination with an implicit acceleration structure that is built on the fly, in combination with a smart pruning strategy, to evaluate Booleans at high speed. Unfortunately they cannot guarantee correct outputs when dealing with coplanar configurations. Vertex perturbation can be applied to decrease the number of failure cases, but cannot fully eliminate them.

To avoid algorithmic stability issues, usually caused by inconsistencies due to numerical rounding when using floating point numbers, [Zhou et al. 2016] use arbitrary precision arithmetics at the cost of performance to compute all triangle-triangle intersections. Triangles are determined as part of the output depending on the winding number of their corresponding volumetric cell. [Cherchi et al. 2020] replace the exact arithmetic with floating point predicates [Attene 2020; Shewchuk 1997], which results in a faster mesh arrangement computation. Similarly, [de Magalhães et al. 2020] use exact arithmetics and performance comparable to [Douze et al. 2017] but cannot handle self-intersections.

2.2 Plane-Based

Apart from the purely vertex-based approaches there is also a long history in plane-based Boolean approaches. [Naylor et al. 1990] introduced the usage of binary space partitions (BSPs) to the field of mesh Booleans. The input solids are represented by implicit BSP trees and a *merge* function combines two input solids into a single BSP that represents the output solid. However, their approach uses a mesh-cutting subroutine which suffers from stability issues due to inconsistencies caused by floating point rounding. To circumvent this problem, [Bernstein and Fussell 2009] extended BSP-based Booleans with filtered floating point predicates [Shewchuk 1997]. To avoid intermediate results requiring an ever-increasing vertex position resolution they also introduce a purely plane-based polygon definition. New vertices are only ever represented by the intersection of three input planes. Unfortunately, their solution scales poorly for large inputs which is inherited from the original BSP-merging algorithm introduced by [Naylor et al. 1990]. [Campen and Kobbelt 2010] avoid the scaling issues by employing a global octree that is subdivided until only a few triangles are contained in each leaf-node. They then use the BSP-based method from [Bernstein and Fussell 2009] locally in leaf-nodes that contain triangles from both inputs. Recently, [Nehring-Wirxel et al. 2021] presented a BSP-based method that replaces floating point with fixed-width integer based predicates. They employ exact mesh-cutting to extract convex BSP cells. Three-dimensional vertex positions at the intersection of three input BSP planes are stored as four-dimensional homogeneous coordinates with integer coefficients.

While not BSP-based, [Hachenberger et al. 2007] compute mesh Booleans via a combination of exact arithmetics and plane-based Nef-polyhedra [Nef 1978], a plane-based polyhedron representation. Though quite reliable, their method suffers from performance and memory consumption issues.

Our method connects to various previous approaches. We use the integer homogeneous coordinates of [Nehring-Wirxel et al. 2021] and extend their formulation to support segment-polygon intersections. Classification is based on generalized winding numbers [Jacobson et al. 2013; Zhou et al. 2016], though we track them during segment tracing instead of growing volumetric cells. The overall subdivision is similar to [Douze et al. 2017], who also avoid the construction of a global acceleration structure.

3 MATHEMATICAL FOUNDATION

Our method uses the fixed size integer construction of [Nehring-Wirxel et al. 2021]. Their method builds upon the plane-based approach and predicates of [Bernstein and Fussell 2009], but instead of using floating point predicates, they formulate their core arithmetics with integer coefficients. If the input values are bounded, all intermediate results are bounded as well, which leads to a very efficient implementation using fixed size integers. In their formulation, intersections of planes have an exact construction that results in 4D integer positions in homogeneous coordinates.

More concretely, we accept input polygons that have 26-bit integers per 3D coordinate or that our method previously produced. Triangles and polygons are represented by a supporting plane and one plane per edge. Planes are stored with integer coefficients (a, b, c, d) and represent all points that satisfy

$$ax + by + cz + d = 0. \tag{1}$$

The (unnormalized) plane normal n is (a, b, c). The integer bit widths are chosen such that we can construct plane-based triangles for any input mesh and no intermediate result later on exceeds 256 bit. When the input and output precision of each operation is known a priori, extremely efficient code for addition and multiplication can be used. With this, useful predicates and constructions can be implemented. For example, two planes p and q are parallel if

$$n_p \times n_q = \vec{0}. \tag{2}$$

Maybe the most important insight of [Nehring-Wirxel et al. 2021] is that the intersection $x = \texttt{intersect}(p, q, r)$ of three planes p, q, r has an exact construction if the result is represented as a position in homogeneous coordinates:

$$x = \left(\begin{vmatrix} d_p & b_p & c_p \\ d_q & b_q & c_q \\ d_r & b_r & c_r \end{vmatrix}, \begin{vmatrix} a_p & d_p & c_p \\ a_q & d_q & c_q \\ a_r & d_r & c_r \end{vmatrix}, \begin{vmatrix} a_p & b_p & d_p \\ a_q & b_q & d_q \\ a_r & b_r & d_r \end{vmatrix}, \begin{vmatrix} a_p & b_p & c_p \\ a_q & b_q & c_q \\ a_r & b_r & c_r \end{vmatrix} \right)^T \tag{3}$$

The intersection is not unique if the last coordinate, x_4, is zero. Given an additional plane s, we can *classify* x (relative to s) by computing

$$\begin{aligned} \texttt{classify}(x, s) &= \operatorname{sign} \langle x, s \rangle \cdot \operatorname{sign} x_4 \\ &= \begin{cases} 1, & \text{if } x \text{ is on the positive side of } s \\ -1, & \text{if } x \text{ is on the negative side of } s \\ 0, & \text{if } s \text{ contains } x \end{cases} \end{aligned} \tag{4}$$

For 3D integer positions p (e.g. input vertices or octree corners), this simplifies to $\langle n_s, p \rangle + d_s$. Using these operations, especially `intersect` and `classify`, efficient polygon clipping can be implemented (cf. Section 4.2.1).

3.1 Additional Operations

For working directly with polygons, we build additional operations on top of those presented by [Nehring-Wirxel et al. 2021]. The plane-based paradigm yields a set of simple and elegant constructions (cf. Fig. 2). The elementary entity is a plane. Two non-parallel planes p, q define a line. Geometrically, this line is the intersection of p and q. Forming the line, however, does not require any computation. We simply identify the pair (p, q) with the line. Similarly, three planes generally intersect in a single point. Other geometric primitives are also easily defined. A ray is represented by three planes (l_0, l_1, r), where (l_0, l_1) forms the line that the ray lies on and r "cuts away" the half of the line not belonging to the ray. Consequently, a segment has four planes (l_0, l_1, r_0, r_1), where (l_0, l_1) is the "supporting line"

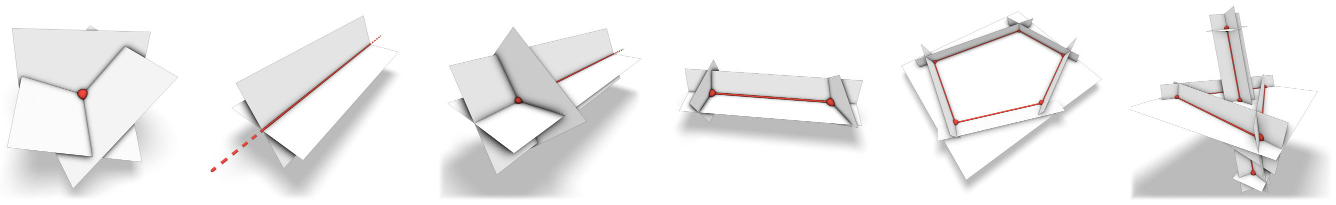


Fig. 2. In plane-based geometry, we can define various geometric primitives as tuples of planes with varying interpretations. Points are the intersection of three planes, lines require two planes. A ray is a line with a starting plane. A segment is a line with two bounding planes. Convex polygons lie on a supporting plane and are bounded by a sequence of edge planes. Intersections of various geometric primitives can be formulated in terms of plane constructions and classifications. For example, the segment-polygon intersection is a point formed by the segment's line and the polygon's supporting plane. It is valid if it lies on the inner side of the segment's bounding planes and the polygon's edge planes.

and r_0 and r_1 delimit the segment on this line. Convex polygons consist of a supporting plane s and a set of edge planes $e_1, \dots, e_n$. A point x belongs to a polygon exactly if $\texttt{classify}(s, x) = 0$ and $\texttt{classify}(e_i, x) \leq 0$ for all edge planes. x is an interior point iff all inequalities are strict. A bounding box can be seen as the space delimited by six particular planes, though we usually choose a less redundant representation for storage.

Intersecting lines, rays, and segments with polygons is surprisingly simple. The intersection point x, should it exist, is given by $\texttt{intersect}(l_0, l_1, s)$, the intersection of the supporting line with the polygon's supporting plane. If $\texttt{classify}(e_i, x) > 0$ for any edge plane, x is outside the polygon. If x classifies positively for r (in case of a ray), r_0, or r_1 (in case of a segment), the intersection is similarly invalid. Corner cases can also easily be detected and handled: x does not exist if (l_0, l_1) is parallel to s. However, there might be an intersection segment if (l_0, l_1) lies within s. This segment can be computed by clipping the query line, ray, or segment against all edge planes e_i.

3.2 Problematic Operations

The fixed-precision planes and homogeneous coordinates are not closed under some common operations. Most notably, while we have chosen the precision bounds such that every polygon with integer coordinates can be converted to its plane-based representation, the same does not hold true (in general) if the polygon coordinates are homogeneous coordinates.

More fundamentally, given two points x and x' in integer homogeneous coordinates, their difference must be computed as

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} - \begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{pmatrix} = \begin{pmatrix} x_1 \cdot x'_4 - x'_1 \cdot x_4 \\ x_2 \cdot x'_4 - x'_2 \cdot x_4 \\ x_3 \cdot x'_4 - x'_3 \cdot x_4 \\ x_4 \cdot x'_4 \end{pmatrix}$$

if $\gcd(x_4, x'_4) = 1$. Thus, even if x and x' lie within the precision bounds, $x - x'$ might not. $x_4 \cdot x'_4$ requires twice as many bits as x_4 and x'_4 individually, which contradicts the fixed-width integer constraints. The immediate consequence is that while points at fractional positions can be constructed via intersection of three planes, our ability to manipulate and perform regular vector arithmetic on them is severely limited. Even something as seemingly harmless as the midpoint of two positions cannot, generally, be computed within the fixed precision limits.

This further implies that new polygons cannot be defined from such positions alone. Supporting and edge planes must be acquired through other means. In particular, triangulation of polygons not possible in general.

Similarly, we cannot directly define the segment between two positions. This becomes a minor obstacle in Section 4.4, where classification necessitates a segment-trace from a reference position to an interior point of a face. While a direct segment is out of reach, we show that a path construction with at most three segments between any two positions is always possible. In particular, we use subsets of the planes defining each point to construct two intermediate representable points.

3.3 Accuracy

When working with 3D or 4D integer coordinates, all presented operations are exact. Inaccuracies can only appear at the interface of our system, i.e. when importing or exporting a mesh. During import, we scale and round the input coordinates to fully utilize the 26 bit integer range. For a 1 m^3 scene, this corresponds to roughly 15 nm accuracy (or, equivalently, almost 8 decimal digits). Mesh formats usually require floating points, which necessitates a rounding step during export, which can be done with full `float` or `double` precision.

3.4 Winding Numbers and Booleans

Same as mesh arrangements [Zhou et al. 2016], we require that all input meshes induce a *piecewise-constant integer generalized winding number* (PWN) field as defined by [Jacobson et al. 2013]. PWN meshes are one of the least restrictive class of meshes that still yields well-defined Boolean operations. Notably, many self-intersecting, non-manifold, or degenerate configurations are supported and still yield unambiguous results.

A winding number vector (WNV) $\mathbf{w} = (w_1, \dots, w_n) \in \mathbb{Z}^n$ is an n-tuple defined for each point in space that does *not* lie on any input surface. Traditionally, n is the number of input meshes and each w_i counts how often mesh i was "entered". w_i can be negative depending on normal orientation. $w_i = 0$ corresponds to "outside of

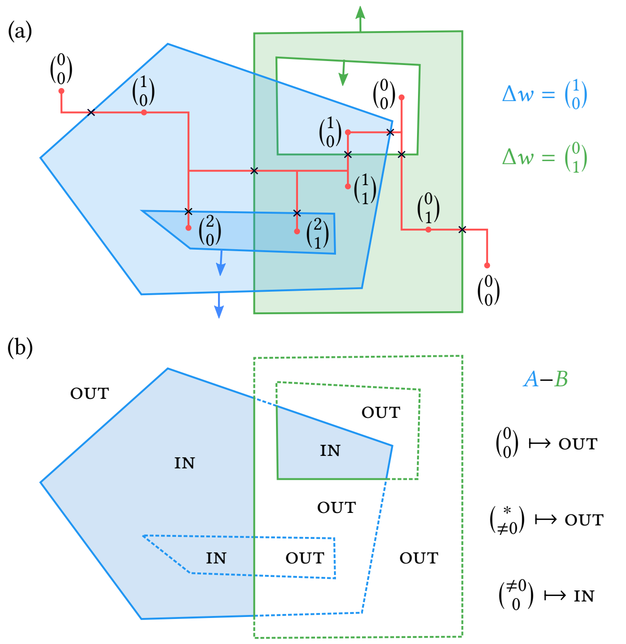

(a)
$\Delta w = \binom{1}{0}$
$\Delta w = \binom{0}{1}$
(b)
OUT
IN
OUT
IN
OUT
IN
OUT
OUT
$A - B$
$\binom{0}{0} \mapsto$ OUT
$\binom{*}{\neq 0} \mapsto$ OUT
$\binom{\neq 0}{0} \mapsto$ IN

Fig. 3. (a) Winding number vectors (WNV) are defined for non-surface points and count how often each mesh is entered or left. They gracefully handle holes and nested components. WNVs can be computed by tracing a path from any point with a known WNV. Each time the path intersects a mesh, the WNV is incremented or decremented depending on normal direction.
(b) Boolean operations can be realized by mapping winding numbers to IN or OUT, i.e. by providing an indicator function on WNV. The result comprises all surfaces that transition IN-OUT or OUT-IN.

mesh i" and $w_i \neq 0$ to "inside of mesh i". Other interpretations can be useful depending on the scenario.

For our method, we introduce the notion of a winding number *transition* vector (WNTV) $\Delta\mathbf{w}$. Any point *on* an input surface (but not on an intersection of surfaces), has a unique WNV *in front* $\mathbf{w_F}$ and *behind* $\mathbf{w_B}$ based on normal direction. Now, $\Delta\mathbf{w}$ is simply defined as $\mathbf{w_B} - \mathbf{w_F}$, i.e. the WNV increments when transitioning through this surface front-to-back (cf. Fig. 3 (a)).

Just as [Zhou et al. 2016], we define Boolean operations as an indicator function on WNVs. For example,

$$f_{\text{union}}(\mathbf{w}) = \begin{cases} \text{IN}, & \text{if any } w_i \neq 0 \\ \text{OUT}, & \text{otherwise} \end{cases} \tag{5}$$

corresponds to the union of n meshes. The indicator function is evaluated on $\mathbf{w_F}$ and $\mathbf{w_B}$. Points where the indicator function changes belong to the output surface. In the actual method, we cut up all input polygons into smaller polygons that have a unique, well-defined $\mathbf{w_F}$ and $\mathbf{w_B}$. Polygons that transition (OUT, IN) are emitted as-is, while (IN, OUT) are emitted with inverted order to preserve proper normal orientation. (OUT, OUT) lies fully outside the result, while (IN, IN) is fully inside (cf. Fig. 3 (b)) and both can thus be skipped.

WNVs represent global information: How often is a given position inside each input mesh? In contrast, WNTVs are local, "derivative" information: How does the WNV change when crossing a certain surface? While WNVs make it easy to evaluate Boolean operations, WNTVs provide a natural way to compute $\mathbf{w_F}$ and $\mathbf{w_B}$ from the inputs.

Each input polygon t has an associated WNTV $\Delta\mathbf{w^t} \in \mathbb{Z}^n$. The usual choice is $\Delta\mathbf{w^t} = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ where the 1 is at the j-th component if t belongs to the j-th input mesh. Intuitively, every time we pass a surface from mesh j outside-in, we increment the j-th component of our WNV. Passing inside-out decrements the component. Formally, given non-surface points x and y as well as the starting WNV $\mathbf{w^x}$ at x, we collect all polygons T that intersect the segment (x, y) and compute

$$\mathbf{w^y} = \mathbf{w^x} + \sum_{t \in T} \operatorname{sign} \langle n_t, x - y \rangle \cdot \Delta\mathbf{w^t},$$

where n_t is the out-facing normal of t and $\Delta\mathbf{w^t}$ the WNTV associated with t. Note that this formula is only valid if x and y do not lie on any surface and all polygons in T have a unique intersection point with (x, y) in their interior. If y lies on a surface with normal n_{ref}, we can compute $\Delta\mathbf{w^y}$, $\mathbf{w^y_F}$, and $\mathbf{w^y_B}$ with a slight tweak as long as y does not lie on any edge:

- Compute $T' = \{t \in T \mid y \in t\}$, i.e. polygons containing y.
- Compute $\mathbf{w^y}$ using $T \setminus T'$.
- Compute $\Delta\mathbf{w^y} = \sum_{t \in T'} \operatorname{sign} \langle n_t, x - y \rangle \cdot \Delta\mathbf{w^t}$.
- Build WNTV depending on $\operatorname{sign} \langle n_{\text{ref}}, x - y \rangle$:
  $1 \mapsto \mathbf{w^y_F} = \mathbf{w^y}$ and $\mathbf{w^y_B} = \mathbf{w^y} + \Delta\mathbf{w^y}$
  $-1 \mapsto \mathbf{w^y_F} = \mathbf{w^y} + \Delta\mathbf{w^y}$ and $\mathbf{w^y_B} = \mathbf{w^y}$

The last case analysis accounts for the fact that $y - x$ can hit the reference surface "from the front" or "from the back".

Often, these winding numbers are either computed via ray tracing or via propagation on 3D cells. What we use is similar to ray tracing, but in a more general segment tracing form due to the constraints described in Section 3.2. This is the basis of our efficient classification scheme of Section 4.4. When using ray tracing, classification is a global problem and challenging to compute efficiently. With our segment tracing, we can keep the classification problem local if a reference point x with known WNV is provided. During the subdivision procedure, we ensure that such a local reference WNV is always available.

4 EXACT MESH BOOLEANS

The input to our method, which we call EMBER, is a polygon soup S with no topology information required. Each polygon t is annotated with a WNTV $\Delta\mathbf{w^t}$. Polygons are expected with homogeneous integer coordinates subject to the limits of Section 3. The input must represent a PWN mesh.

Our method produces a set of convex polygons S'. Each resulting polygon $t' \in S'$ has a WNTV in the form of $(\mathbf{w^{t'}_F}, \mathbf{w^{t'}_B})$. These polygons satisfy strong guarantees:

(P1) Polygons of S' are disjoint and represent the same surface as S
(P2) The uniquely defined WNTV $(\mathbf{w^{t'}_F}, \mathbf{w^{t'}_B})$ of each resulting polygon t' is valid for all points inside t'

(P2) requires that we cut up all polygons with non-constant WNTV. This ensures that Boolean operations can be performed per-polygon instead of per-point.

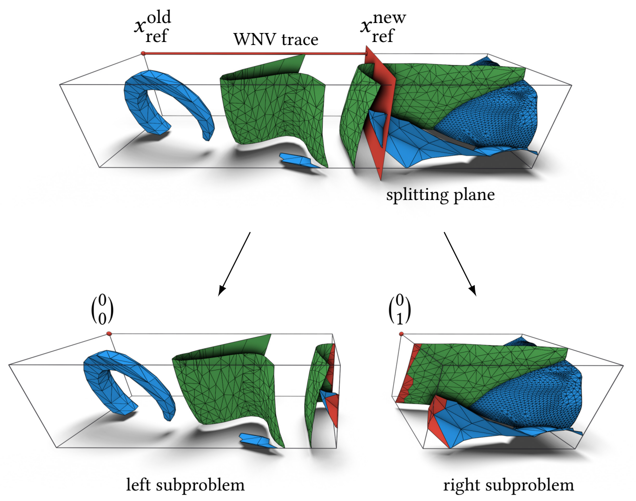


Fig. 4. As part of the overall subdivision structure, subproblems are split at an axis-aligned plane. The plane is located at the center-of-gravity and oriented towards the axis of largest variance. Most polygons are unchanged by this split (affected ones are shown in red). To keep the leaf computation local, we require a local reference point and its WNV. In this example, the previous reference point is in the left half. For the right subproblem, we project $x_{\text{ref}}^{\text{old}}$ onto the right AABB and compute its WNV via segment tracing.

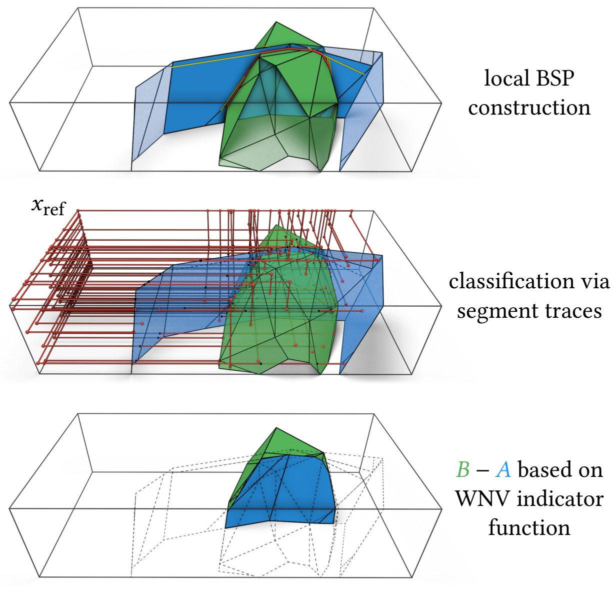

local BSP construction
x_{ref}
classification via segment traces
$B - A$ based on WNV indicator function

Fig. 5. The computation at the leaves of our subdivision is completely local. We start by computing pairwise intersections and build per-polygon BSPs (new edges in yellow, only intersecting polygons are opaque). Each leaf BSP polygon is then classified by tracing a segment path towards the local reference point x_{ref} that has a known WNV. The result of the classification is a pair of WNVs that is then passed through an indicator function to emit exactly those polygons with an IN-OUT or OUT-IN transition.

Usually, an operator indicator function $f_{\text{op}} : \text{WNV} \to \{\text{OUT}, \text{IN}\}$ is also provided. For each polygon, $(f_{\text{op}}(\mathbf{w}_{\text{F}}^{t'}), f_{\text{op}}(\mathbf{w}_{\text{B}}^{t'}))$ is evaluated. If the result is (OUT, IN) we keep the polygon, if it is (IN, OUT) we invert its order. (IN, IN) and (OUT, OUT) are discarded. This can be done as a post-process, but it is significantly more efficient if knowledge of f_{op} is used during creation of S' to discard entire regions when we can prove that those regions will only produce (OUT, OUT) or (IN, IN) polygons (cf. Section 4.5).

4.1 Overview

Given a soup of polygons $t \in S$ annotated with WNTVs $\Delta\mathbf{w}^{\mathbf{t}}$, our algorithm recursively performs subdivision and leaf computations. The adaptive subdivision of the scene's bounding box recurses until the local problem is small enough to be solved directly via a BSP and winding number tracing. During this subdivision phase, we propagate not only the set of polygons lying within the respective sub-box but also a reference point with known WNV. This way we can locally determine the WNV for each compartment by segment tracing using the WNTVs of the intersected polygons. The reference propagation avoids the need to construct a global acceleration structure. Since the recursive subdivision provides many criteria for early-out termination, significant portions of the hierarchy never have to be generated.

More concretely, the subdivision task (cf. Fig. 4) performs the following steps:

- split AABB into two sub-AABBs
- for each sub-AABB B:
  - clip each $t \in S$ against B
  - if $x_{\text{ref}} \notin B$, compute new reference point and its WNV via segment-tracing from x_{ref}
  - recursively call algorithm with clipped S, sub-AABB, and potentially updated x_{ref}

Leaf tasks (cf. Fig. 5) do the following for every polygon $t \in S$:

- build local binary space partitioning (BSP)
  - intersect t with all other polygons in S
  - add each intersection segment to the BSP
- for each leaf polygon f in the local BSP:
  - trace segments to x_{ref} to compute $(\mathbf{w}_{\text{F}}^{\mathbf{t}}, \mathbf{w}_{\text{B}}^{\mathbf{t}})$
  - emit $(t, \mathbf{w}_{\text{F}}^{\mathbf{t}}, \mathbf{w}_{\text{B}}^{\mathbf{t}})$

The core insight is that by keeping track of x_{ref} and $\mathbf{w}_{\text{ref}}$, we can always compute the classification locally using segment traces that stay inside the sub-AABB. Ideally, the problem size is halved with each subdivision, leading to logarithmic recursion depth and only linear overall memory requirements. The details of each step are explained in the following sections.

4.2 Subdivision and Clipping

If the current subproblem has too many polygons, exhaustive pairwise intersection tests are prohibitively expensive. Thus, we try to reduce the problem size until these pairwise tests become feasible. We keep the bounding volume of each subproblem as a simple AABB with integer coordinates. A subdivision is therefore a split along one axis at an integer position. While simply choosing the midpoint

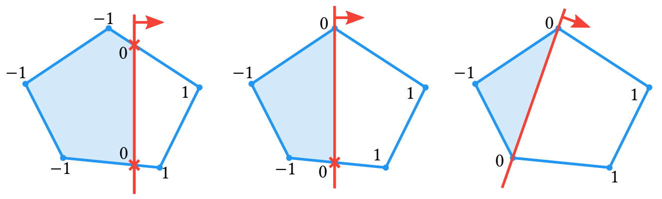


Fig. 6. Clipping a convex polygon against a plane using `classify` is straightforward. Each vertex is classified as −1, 0, or 1. Edges from −1 to 1 are split. The result can be assembled from all non-positive vertices on the one side and all non-negative ones on the other side. Clipping requires creating 0, 1, or 2 new vertices via `intersect`. An example of each case is shown.

of the longest AABB axis leads to an even and reasonably efficient subdivision, we discuss a more elaborate strategy in Section 4.5.

4.2.1 Polygon Clipping. Given a splitting plane s, the subdivision procedure creates the "left" and "right" subproblem and recursively calls the whole method on those. Each polygon in each annotated polygon soup is clipped against the splitting plane and thus either added to a single subproblem or split into two parts and added to both. We assign polygons that lie exactly on the splitting plane to the "left" subproblem.

Clipping a (convex) polygon with vertices $v_1, \ldots, v_n$ is simple (cf. Fig. 6). First, we use Equation 4 to compute the classification $c_i \in \{-1, 0, 1\}$ of each vertex v_i relative to the splitting plane. If there are no positive values, the polygon is "left". Otherwise, if there are no negative values, we assign it "right". All-zero is assigned "left".

We only have to actually split if there are negative and positive values. For each edge between −1 and 1, a new vertex is constructed using the supporting plane, edge plane, and splitting plane. Afterwards, there are exactly two vertices with classification 0, even for cases where the splitting plane exactly hits one or two polygon vertices. All non-positive vertices form the polygon for "left" and all non-negative for "right". Since each vertex is the result of intersecting original (and thus properly quantized) polygons and AABB faces (which are axis-aligned), they can always be represented by our homogeneous integer coordinates.

4.2.2 Updated Reference Point. To retain the ability to classify each polygon locally, we require a known WNV at some position in each subproblem. In general, we already have such a position for one half, but not for the other. Thus, we need to choose a non-surface position in the other half and compute its WNV by tracing segments from the given reference position to the new one as described in Section 3.4. Similar to the classification in Section 4.4, this requires tracing up to three segments. If an intermediate point lies on a surface or any traced segment lies inside a polygon or hits an edge, then this particular path cannot be used for classification. However, as long as the new reference position is in the second subproblem and the path does not leave the current AABB, we have the freedom to choose. As paths are only invalid if they hit degenerate configurations, in the overwhelming majority of cases the first path is already valid. If not, valid paths are found within very few iterations.

For performance reasons, we prefer paths with fewer segments and segments that are axis aligned. Those require less precision and fewer intersection tests to compute. Thus, our first guess is always to just project the previous reference point onto the new subproblem AABB. For the vast majority of practical cases, this yields a valid path ending on an integer position, reachable by a single axis-aligned segment trace. An example is shown in Fig. 4.

4.3 Face-Face Intersections via Locally Constructed BSP

Once a subproblem becomes small enough that quadratic complexity in the number of polygons is acceptable, we can intersect and classify. This section describes the intersection resolution while the next covers classification. The exact criterion when a subproblem is small enough is subject of Section 4.5.

In a leaf task, we are given a set of convex polygons with self-intersections. Our goal is to compute a set of convex polygons *without* self-intersections covering the same surface.

The usual strategy is to compute all pairwise intersection segments and embed them into their respective polygons. This result is then re-triangulated, e.g. via Delaunay triangulation as used by [Zhou et al. 2016]. Special care must be taken for coplanar faces.

We developed a different strategy for several reasons. The most important one is that, in our formulation using fixed-precision homogeneous coordinates, re-triangulation is in general not possible (cf. Section 3.2). Even if this were possible, a Delaunay triangulation itself is already quite costly. Furthermore, in a CSG setting, such a triangulation often leads to vertices of extremely high valence.

Our plane-based formulation places a strong emphasis on edges and the planes defining a polygon, favoring cutting operations, and discouraging operations directly on vertex coordinates. Therefore, we frame the whole intersection step as a BSP construction: For each polygon t, we build a temporary, local BSP. Non-leaf nodes are annotated with a splitting plane and leaves with a polygon representing the leaf geometry. It is important to note that we are not using the BSP as an acceleration structure *containing* intersection segments, but rather as a way to *represent* geometry: The resulting BSP is a disjoint partition of the polygon t and no BSP leaf has an intersection in its interior. All intersections lie on leaf boundaries, with the exception of the overlap case, which is handled later.

The initial configuration is simply a leaf with the geometry of t. For the construction, we only support a single operation: "add" a segment (v_0, v_1, s) to the BSP. Here, v_0 and v_1 are the segment vertices and s is a plane that, together with the supporting plane s_t of the polygon t, defines the line that the segment lies on. As most procedures on recursive data structures, the operation itself is defined recursively as "adding" the segment to a BSP node n:

```
• if $n$ is an inner node split at a plane $q$:
  – compute $c_0$ := classify($v_0, q$) and $c_1$ := classify($v_1, q$).
  – stop if $c_0 = c_1 = 0$ (segment lies on splitting plane).
  – if $c_0 \le 0$ and $c_1 \le 0$, add segment only to the left node.
  – if $c_0 \ge 0$ and $c_1 \ge 0$, add segment only to the right node.
  – if $c_0 < 0$ and $c_1 > 0$:
    ∗ compute new vertex $v'$ := intersect($s, q, s_t$).
    ∗ add segment $(v_0, v', s)$ to the left node.
    ∗ add segment $(v', v_1, s)$ to the right node.
  – (if $c_0 > 0$ and $c_1 < 0$ proceeds analogously)
• if $n$ is a leaf node:
```

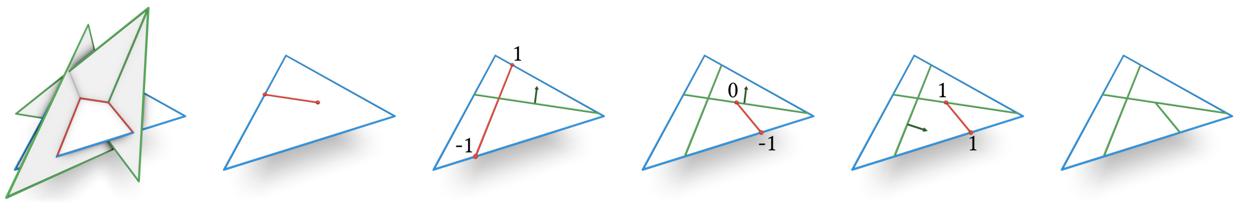


Fig. 7. Construction of the local BSPs is done via adding intersection segments. Each leaf node that contains a non-trivial part of the segment is split. The image shows a triangle with three intersection segments (red) that are added to the BSP (with existing splitting planes in green). Numbers indicate the result of classifying the segment vertices against the current BSP node plane. Note that while each intersection segment eventually lies on a green BSP splitting plane, the converse is not true: In this example, the first intersection segment causes a split that is longer than strictly necessary. This conservative splitting keeps each cell convex, leading to a simple and fast method.

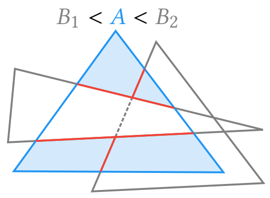


Fig. 8. Top-down view of three overlapping triangles. Overlapping polygons are gracefully handled during BSP construction. Given a total order of input polygons (e.g. indices), BSP leaves are disabled if they are overlapped by a "lower" polygon. This guarantees that overlap regions contribute to the result at most once. In this example, the current triangle A is cut up by two overlapping ones. However, only the light blue parts of A are further classified and might contribute to the result as $B_1 < A$ but $A < B_2$.

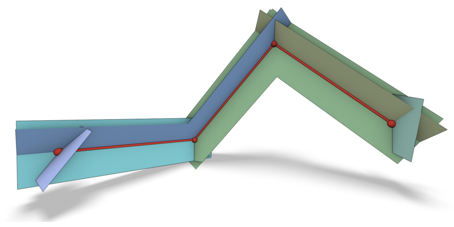


Fig. 9. Given two points in homogeneous coordinates, we cannot, in general, form a segment directly between the points while staying within our precision bounds. However, for classification, we only need a path between them. Each point is defined by the intersection of three planes and by changing one plane at a time, it is always possible to construct a path between two such points with at most three segments.

- use s to split leaf polygon (cf. Section 4.2.1).
- create an inner node with s and two new leaf nodes with the split result.

Now consider all other polygons t' (with supporting plane $s_{t'}$). The intersection of t and t' can be computed solely from the `intersect` and `classify` operations and is one of the following cases:

(C1) no intersection
(C2) a single point
(C3) a non-degenerate segment $(v_0, v_1, s_{t'})$
(C4) a non-empty overlap polygon

(C1) and (C2) can be safely ignored. (C3) is simply added to the BSP, though it might not actually lead to new leaves if the segment lies on an edge or a previous split. An example is shown in Fig. 7.

The overlap case (C4) is slightly more complex: First, all edges of t' are added to the BSP. As t' performs a symmetrical operation, we end up with leaves in t and t' that represent the same surface geometry and thus also have the same WNTV, i.e. the same classification. To satisfy our disjointness guarantee in overlap regions, we "disable" the leaves of all polygons but one. Given a total order on polygons, e.g. input indices, we simply mark the overlap region in t's BSP as "disabled", if $t > t'$. Consequently, in any overlap region, only the polygon with the minimal index emits output polygons (cf. Fig. 8).

At the end of this step, we have a local BSP for polygon t, where all intersection segments are "added". By construction, no leaf polygon has an interior intersection with any other polygon. Each BSP is, as the name implies, a partition of t. Together with the overlap handling, the set of enabled leaf polygons represents the same geometry as the input and is free of self-intersections, thus satisfying our initial guarantees (P1) and (P2).

4.4 Face Classification via Segment Tracing

With the guarantee of (P1) and (P2), the interior of each enabled leaf polygon t has a well-defined WNTV $(\mathbf{w}^{\mathbf{t}}_{\mathbf{F}}, \mathbf{w}^{\mathbf{t}}_{\mathbf{B}})$. We are still in the computation of a subproblem, i.e. are given a soup S of polygons t with annotated $\Delta\mathbf{w}^{\mathbf{t}}$, an AABB, and a reference point x_{ref} with WNV $\mathbf{w}_{\text{ref}}$. The mathematical foundation was already presented in Section 3.4. However, that description presumes that a target position x for classification is known and that the segment from x_{ref} to x is definable. In general, we cannot define a segment from two non-integer positions (cf. Section 3.2). Furthermore, a classification can "fail" in the sense that a path is not suitable for WNV propagation, e.g. because the segment touches the edge of an input polygon.

We start with a simple heuristic that works in most situations and is cheap to compute: Using regular floating point numbers, we compute the center of gravity of t and round that to the nearest integer coordinates c. Then, we define an axis-aligned line starting at c, pointing towards the axis closest aligned to t's normal. This is the "least parallel" axis-aligned choice. Now, we intersect that line with t using the exact arithmetic. If there is an intersection with the interior of t, we construct a simple axis-aligned path of up to 3 segments (cf. Fig. 5).

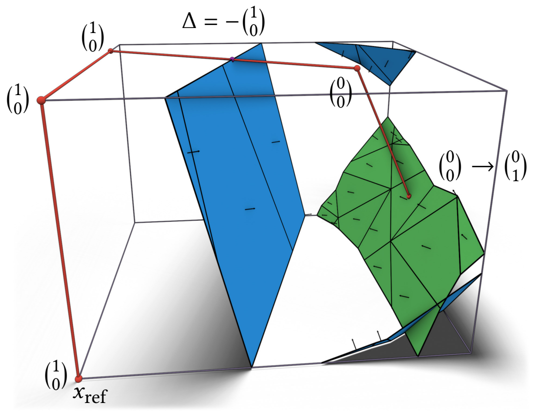


Fig. 10. When classifying a polygon of a local subproblem, we trace a path from the reference point (with known WNV) to an arbitrary point in the interior of the polygon. The 3-segment path construction between two points might leave the subproblem AABB, thus the path is additionally clipped to the AABB.

Should this construction fail, either because the line does not intersect t (can happen with very thin polygons) or because the classification path is invalid, we use a more complex construction consisting of two sub-steps. First, we compute a point in the interior of t as follows:

(1) Take any polygon vertex defined by supporting plane s_t and edge planes $s_{e_0} = (a_0, b_0, c_0, d_0)$ and $s_{e_1} = (a_1, b_1, c_1, d_1)$.
(2) Define $s'_{e_0} = (a_0, b_0, c_0, d_0 + i_0)$ and $s'_{e_1} = (a_1, b_1, c_1, d_1 + i_1)$ where $i_0, i_1 \in \mathbb{N}_{\geq 1}$ are (randomized) offsets.
(3) Compute $x = \text{intersect}(s_t, s'_{e_0}, s'_{e_1})$.
(4) Go to (1) if x is not in the interior of t.

x always lies on the supporting plane of t. The offsets i_0 and i_1 are always positive and thus "move" x into the interior of t. The iterative nature of (4) ensures that x is not moved "too far". The plane coefficients of s_{e_0} and s_{e_1} can be scaled up, making the steps of i_0 and i_1 more granular. In practice, we scale until the largest coefficient is close to our precision limit and start with $i_0 = i_1 = 1$. This is generally sufficient. Only specifically constructed corner cases require more than one iteration.

The second step connects x to x_{ref}. Let's assume x and x_{ref} are defined as the intersection of three planes (s_0, s_1, s_2) and (r_0, r_1, r_2), respectively. Consider the point x_1 defined by (s_0, s_1, r_2). x and x_1 both lie on the planes s_0 and s_1 and thus on the line (s_0, s_1). Thus, (s_0, s_1, s_2, r_2) is a segment starting at x and ending at x_1. Similarly, we can define x_2 by (s_0, r_1, r_2). This gives us a path $(x, x_1, x_2, x_{\text{ref}})$ from x to x_{ref} where each intermediate segment and point is definable within our precision bounds.

The second construction works by iteratively replacing a plane defining x with a plane defining x_{ref}. The actual order does not matter and leaves us with another freedom to choose differently should the path be invalid (e.g. because it hits a polygon edge during tracing). The path $(x, x_1, x_2, x_{\text{ref}})$ might leave the subproblem AABB, in which case we need to clip it against the AABB planes. An example is shown in Fig. 9 and Fig. 10.

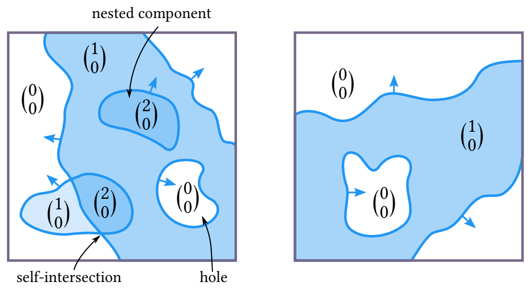


Fig. 11. Often, additional input guarantees are available. These can be leveraged when a subproblem consists of only one WNTV type. If each input is individually self-intersection free, the local BSP construction step can be skipped. If there are no nested components, a single classification is valid for all contained surfaces. On the left is an example without these guarantees and on the right with them. Note that surfaces on the right always transition $\binom{0}{0}$ to $\binom{1}{0}$.

The essence of these constructions is that we have a simple default case that succeeds in almost all cases. Due to our exact arithmetic, we can accurately decide when a construction was insufficient and change to the more complex one.

The result is a $(\mathbf{w}_{\mathbf{F}}^{\mathbf{f}}, \mathbf{w}_{\mathbf{B}}^{\mathbf{f}})$-pair for each enabled leaf polygon f. Together, these polygons satisfy the stringent guarantees laid out at the beginning of Section 4. If an operator indicator function is given, the resulting geometry can be emitted directly.

4.5 Optimizations

The previous sections present all steps that are required to develop an intuition and understanding of why the method computes the exact, correct result and its expected time and space complexity. However, we also claim that our method is exceedingly fast on real-world data. To achieve this, we present a few algorithmic and implementation optimizations that neither affect correctness nor asymptotic complexity but still improve performance by orders of magnitude under the right circumstances.

4.5.1 Leveraging Additional Input Assumptions. The input to our method is a soup S of polygons t with annotated WNTV $\Delta\mathbf{w}^{\mathbf{t}}$. Intuitively, all polygons with the same WNTV correspond to a single input mesh. While different meshes usually intersect each other, we often know that every individual mesh S_i is self-intersection free and contains no nested components. In particular, any mesh that results from our method satisfies these assumptions. We thus allow each "WNTV class" to be flagged as containing no self-intersections `NSI` and/or containing no nested components `NNC`. These will be used in later optimizations. For example, a leaf subproblem containing only a single WNTV type in general still requires local BSP construction and classification. However, if this WNTV is marked `NSI`, the BSP construction can be skipped. If it is furthermore `NNC`, only a single polygon needs to be classified as all others will result in the same $(\mathbf{w}_{\mathbf{F}}^{\mathbf{t}}, \mathbf{w}_{\mathbf{B}}^{\mathbf{t}})$. An example is shown in Fig. 11.

4.5.2 Early Subdivision Termination. Even without additional assumptions, the operator indicator function itself provides ample opportunities for early-outs. For example, consider the computation of $A - B$ given two meshes A and B with WNV $(1, 0)$ and $(0, 1)$,

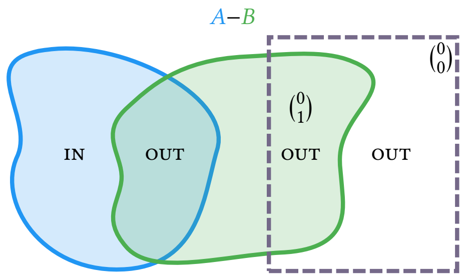


Fig. 12. Even without additional input assumptions, certain configurations admit optimization. The highlighted subproblem in this difference operation has a reference WNV of $(0,0)$ and only contains faces of the $(0,1)$ mesh. For Boolean difference, only WNVs of the form $(z,0)$ with $z \neq 0$ would classify IN. Any WNV trace in the subproblem results in $(0,0) + k \cdot (0,1)$ and can thus never be IN. This particular subproblem will therefore never contribute to the result.

respectively. Generally, a subproblem consisting only of polygons from B requires full subdivision and classification if B is neither NSI nor NNC. However, if the current reference WNV has the form $(0,b)$, the subproblem can be immediately discarded: The indicator function for $A-B$ is only IN if the WNV has the form $(a,0)$ with $a \neq 0$. Classification in the current subproblem starts at the reference WNV of $(0,b)$. Only polygons from B are present, so any WNV used for classification has the form $(0,b+z)$ for some $z \in \mathbb{Z}$. Thus, without any further subdivision and classification, we already know that all polygons will classify (OUT, OUT) and are therefore discarded (cf. Fig. 12).

Such rules exist for all Boolean operations. While the actual rules differ for each operation, they can be mechanically computed, quite similar to a reachability analysis on a finite automaton. This optimization is especially potent for variadic operations. In a milling simulation, many instances T_i of a "tool" mesh are subtracted from an initial workpiece W. Now consider the computation of $W - T_1 - \ldots - T_n$. Again, the indicator function is only IN if inside W and outside all T_i. The computed optimization rules are:

- Discard if outside W but subproblem does not contain W.
- Discard if inside a T_i that is not part of the subproblem.

4.5.3 Splitting Strategy. An important aspect of any space subdivision scheme is where to split and when to stop. Our baseline strategy is splitting the largest AABB axis in half and stop when the total number of polygons is below a threshold. Profiling showed two main performance hotspots (cf. Fig. 16):

(H1) Testing and splitting polygons during early subdivisions
(H2) BSP construction and polygon classification in leaves

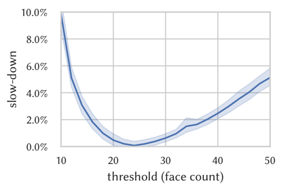
(H1) is expensive due to the large amount of polygons that must be tested. With the previously described early-out strategies, (H2) naturally concentrates where different input meshes intersect.

A good subdivision strategy must therefore focus on reducing the overall polygon count in the early levels as efficiently as possible. For mostly uniform and isotropic tessellations, the baseline strategy is already quite effective. Another common strategy that we tested is to split the set of polygons at their center-of-gravity along the axis of largest variance. This leads to a more efficient subdivision when tessellations or polygon distributions are less uniform. Practically, only a subset of polygons should be used to determine the splitting plane as otherwise the cost of finding the splitting plane exceeds the actual splitting. Many more complicated strategies suffer the same pitfall of costing more to compute than they save. The strategy that we settled with for our implementation is simple but tries to exploit the early-out behavior. The polygon soup S is stored as a set of soups S_i with uniform WNTV.

- Compute center-of-gravity c_i for each sub-soup S_i.
- Consider the AABB planes p of each other S_j:
- Add p to splitting plane candidates if it separates c_i from S_j.
- Return candidate with the farthest separation distance.
- Fall back to largest-variance-split if no candidate found.

This strategy largely keeps the "median split" spirit but all candidates guarantee that one subproblem will not contain S_j, thus increasing the chances for early-out policies to act.

Towards the end of the subdivision, subproblems contain fewer but larger polygons. This inevitably reduces subdivision efficiency, as any polygon that intersects the splitting plane ends up in both recursion branches. The result are two opposed trends: More subdivisions lead to smaller leaf subproblems and thus cheaper per-leaf runtime. On the other hand, the total number of polygons of all leaf subproblems increases, thus driving up the runtime. The optimal tradeoff point is hard to predict and depends on many factors. The inset shows our experiments to determine the threshold based on our benchmark data set. There is a optimum at 25 polygons, though the slowdown is small enough that the exact value is not too important.



4.5.4 Parallel Implementation. Our method was carefully designed so that each subdivision step is basically a pure function: The resulting polygons of the current subproblem only depend on local data. Further subdivision, BSP construction, and classification only require the given polygon soups and the reference point with its WNV. The overall recursive procedure is thus perfectly suited for a work-stealing approach: A pool of threads holds a centrally synchronized queue Q of subproblems. In each subdivision step, the algorithm only recursively continues with one subproblem. The other is added to Q. When idle, threads take and process a new subproblem from Q. Once the first few subdivisions took place, Q contains enough subproblems to saturate the cores of any typical workstation. This simple strategy already results in an effective parallelization (cf. Section 5.3).

4.5.5 Important Implementation Details. As the final part of this section, we mention a few implementation details that offer little insight into the actual method but are nevertheless crucial to achieving maximal performance.

Most high-performance computations end up being memory-bound, either by latency or bandwidth. We made sure to minimize allocations and re-use memory whenever possible. The subdivision structure and purely local computation aided enormously: Once one branch of the recursion returns, all allocated memory in that branch can be re-used.

Using the full 256 bit arithmetic everywhere is needlessly conservative. Various coefficients of axis-aligned planes and lines are -1, 0, or 1. Many intermediate results can be safely computed with reduced bit sizes. In contrast to arbitrary-precision integers, all these constraints are known statically and are implemented using individual types.

Naturally, we also exploit the common technique of doing conservative AABB checks wherever possible. In particular, segment tracing always performs an AABB test before actually computing any intersections.

5 EVALUATION

All benchmarks were performed on an Intel i9-9900K processor at 3.60 GHz (5.00 GHz with boost). As the state-of-the-art methods are multithreaded, we use our parallel implementation for comparison. The code was written in C++ compiled with `clang-12`, with optimizations enabled at `-O3`.

5.1 Thingi10K Benchmark and Comparison

Representative performance benchmarks of CSG algorithms are notoriously difficult to perform. To approximate real-world applications, we use the following setup: We take the Thingi10K [Zhou and Jacobson 2016] data set as a set of real-world models with a broad distribution of face counts, triangle anisotropy, and tessellation quality. A common benchmark choice is to compute a self-intersection of each model with a slightly rotated version of itself. However, this seems far removed from real-world CSG operations. As a better alternative, we propose and use the following operation:

- Draw two random meshes from the data set.
- Apply a randomized transformation on both that results in largely overlapping bounding boxes.
- Perform and measure a basic Boolean operation.

An example of this is shown in Fig. 15. This methodology is, in our opinion, closer to real-world CSG usage. Each input pair consists of meshes with different complexity, anisotropy, and tessellation. The transformation ensures a healthy mix of intersecting and non-intersecting mesh regions. A slightly rotated self-intersection might stress-test the intersection handling of an CSG algorithm but efficient handling of non-intersecting parts of a mesh is no less important in a real-world setting.

While some methods, such as [Zhou et al. 2016] or ours, can handle any PWN mesh, most others require stronger input assumptions. Thus, for the actual comparison, we restrict the data set to 1000 solid, manifold meshes without self-intersections with 1000 to 100 000 faces. The lower bound discards trivial meshes. Without the upper bound, we ran into time and memory limits on our test machine for some methods. In the supplemental material, we provide the complete benchmark data consisting of mesh ID pairs and transformations.

We compare against *QuickCSG* [Douze et al. 2017], *Cork* [Bernstein 2013], *CGAL* [Hachenberger et al. 2007] (Nef, lazy exact), *Mesh Arrangements* [Zhou et al. 2016], and *Floating-point Mesh Arrangements* [Cherchi et al. 2020]. Note that the available implementation

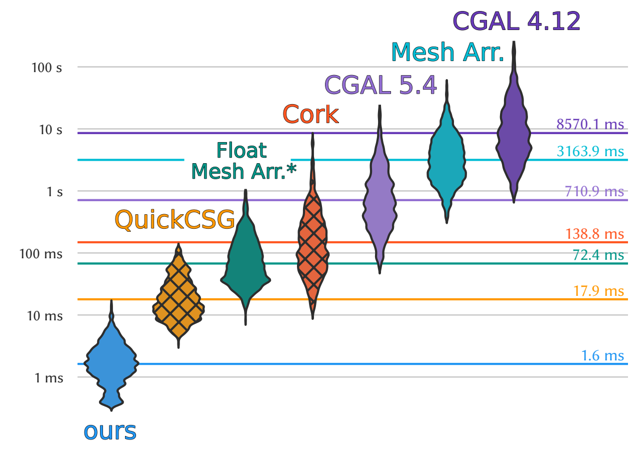


Fig. 13. Timing histograms of various CSG methods on 1000 pairs of meshes from Thingi10K with 1000 to 100 000 faces. Non-exact methods are cross-hatched. The colored horizontal lines show geometric mean performance. Additional discussion is in Section 5.1. In particular, the reference implementation for floating-point mesh arrangements only includes self-intersection resolution. Note the log scale on the vertical axis. Lower is better.

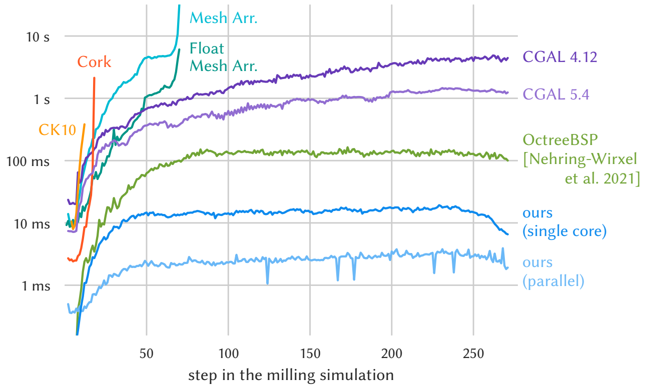
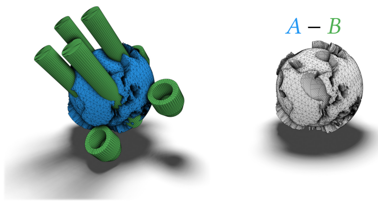


Fig. 14. Benchmark timings for an iterated CSG scenario where a drill bit is repeatedly subtracted from a workpiece. Original benchmark and mesh data provided by [Nehring-Wirxel et al. 2021] (cf. their Figure 13, Figure 14, and corresponding discussion). The dip at the end (most notable in our single core version) is due to the drill bit leaving the workpiece. Note the log scale on the vertical axis. Lower is better.

for [Cherchi et al. 2020] only includes self-intersection resolution, not the actual classification and thus CSG operation. Resolving self-intersections was the bottleneck for the original mesh arrangements, though this might not be the case anymore considering the performance difference. Also, for the benchmark of this method only, the code was compiled with `gcc-10` as required by the reference implementation.



Mesh A:	29 472 faces
Mesh B:	2568 faces
Result:	48 750 faces
ours:	3.86 ms
QuickCSG:	36.46 ms
Mesh Arr.:	9578.03 ms

Fig. 15. Example from our test data set based on Thingi10K. Here, we only compare against QuickCSG [Douze et al. 2017] and Mesh Arrangements [Zhou et al. 2016]. All implementations are multithreaded and the tests were performed on the same machine. Note that only Mesh Arrangements and our method produce exact results.

Of all compared methods, only ours, the two mesh arrangements, and CGAL are exact. QuickCSG and Cork regularly produce topologically and geometrically incorrect results, sometimes catastrophically so. Especially degenerate or close-to-degenerate configurations tend to fail.

Traditionally, exact methods are at least one order of magnitude slower compared to inexact CSG methods. The benchmark in Fig. 13 shows that we have reversed the situation: Our method is roughly one order of magnitude faster than the fastest inexact CSG method we compared against. The geometric mean is 1.6 ms per mesh Boolean. The corresponding input size is roughly 20 000 faces. Over the whole benchmark, we process about 15 000 000 input triangles per second.

Fig. 14 shows per-step timings for the milling simulation of [Nehring-Wirxel et al. 2021] (their Figure 13 and 14). This iterated CSG scenario is particularly challenging because the inputs have asymmetric complexity. The workpiece becomes extremely complicated while the drill bit is small and low-poly. The method of [Nehring-Wirxel et al. 2021] deals with this gracefully by keeping BSPs with bounded complexity embedded in a persistent octree data structure. However, Boolean operations on BSPs are significantly more expensive than our approach, which allows us to outperform their method even if the subdivision structure has to be recomputed in each step. For future work, we anticipate that keeping our implicit *k*d-tree persistently and not rebuilding it each step will provide another substantial boost of performance.

5.2 Performance Breakdown

Fig. 16 shows the contribution of various steps of our algorithm to the total runtime as distributions over the benchmark data set. At a high level, our method consists of subdivision and leaf steps. Typically, the leaf computation is twice as expensive as the subdivision, though this can vary significantly depending on the actual task: Meshes with many non-intersecting parts profit immensely from subdivision early-outs and require less leaf computation.

The subdivision itself mainly consists of splitting polygons against the splitting plane. While not expensive per polygon, the extreme number of times this has to be performed adds up. Computing the splitting plane can be expensive if strategies iterate over all polygons. However, this can often be limited by considering a bounded subset. Updating the reference WNV requires a segment trace over

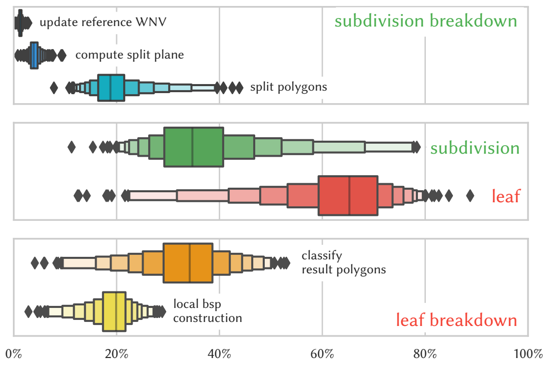


Fig. 16. Relative distribution of runtime performance over our benchmark data set. In most configurations, twice as much time is spent in leaf nodes as in the subdivision. Subdivision itself is dominated by polygon splitting. In the leaf nodes, tracing classification segments is almost twice as expensive as computing and resolving pairwise intersections.

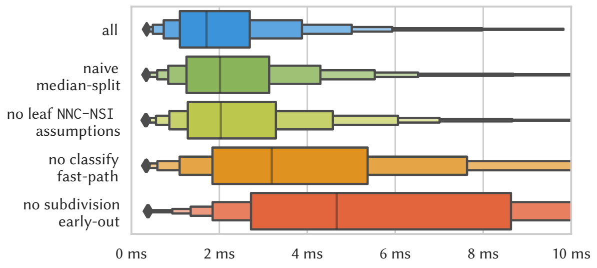


Fig. 17. A small ablation study to show the runtime impact of disabling various optimizations individually. Timing distribution is shown over our benchmark data set.

all subproblem polygons. Perhaps surprisingly, this barely shows up on the profile. These traces tend to be axis-aligned and lie at the AABB border, which makes them extremely efficient.

The biggest contributor to the leaf computation is the polygon classification, i.e. tracing segments towards the reference WNV. Resolving pairwise intersections and building per-face BSPs takes a strong second place.

Fig. 17 complements this breakdown with an ablation study. Various configurable optimizations are individually disabled to show their effect on the total runtime. The two biggest gains are classify fast-paths (cf. Fig. 5 vs. Fig. 10) and early termination during subdivision (cf. Fig. 11 and Fig. 12).

5.3 Parallelization

The parallelization using work-stealing as described in Section 4.5.4 scales reasonably well. Fig. 18 (a) shows the speedup factor depending on the number of threads. For a CPU with 8 physical cores, scaling beyond 8 threads has diminishing returns and is mostly limited to hyperthreading. A fully linear scaling is difficult to achieve in practice as many parts of our method bottleneck on memory bandwidth, not computational power. Furthermore, Fig. 18 (b) shows

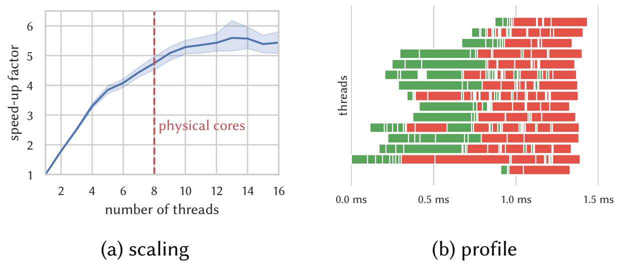


Fig. 18. (a) Scaling of our method depending on the number of threads on a CPU with 8 physical cores. The speedup is substantial but not fully linear as memory bandwidth becomes saturated and the beginning of our method has limited scaling. Variation across test data set is shown in light blue. (b) Representative performance profile of our parallelization using work-stealing with subdivision tasks (green) and leaf tasks (red). Each subdivision task results in two more tasks. As there is only one initial task, full parallelization is only achieved after a few subdivisions.

an example profile of the work-stealing approach. We start with a single subdivision task that spawns two new tasks, which spawn another two tasks each. Thus, the start of our method has a certain "burn-in" phase until it reaches full parallelism. Still, a speed-up of 5.5× on an 8-core CPU is an adequate result.

5.4 Examples

We conclude our evaluation with several examples and applications. Fig. 19 shows Booleans between a few larger models with timings. As each result polygon is a subset of an input polygon, barycentric weights can be used to transfer additional input attributes to the output as shown in Fig. 20 with texture coordinates. The WNV dimension does not have to correspond to the number of input meshes. In Fig. 21, we use $z \in \mathbb{Z}$ as WNVs with a WNTV of 1. The z thus corresponds to how many input surfaces a position is inside of. $0 \longrightarrow 1$ transitions are the outermost "layer", $1 \longrightarrow 2$ the second layer, and so on. Fig. 22 shows the intersection of an organic model with a dense grid of cubes. In Fig. 23, we demonstrate the robustness and correct handling of various special cases. An example where a linear amount of input faces can lead to quadratically many output faces is shown in Fig. 24. Similarly, challenging is Fig. 25 where a configuration of cubes has up to 20 coplanar faces at the same location. In Fig. 26, we show the different tessellations produced by various methods for mesh Booleans. Many methods create triangulations with many thin triangles and high-valence vertices. As an interesting trade-off, our method generally produces more output faces and low-valence vertices. In particular, the subdivision structure ensures that intersections only have local influence on the tessellation.

6 LIMITATIONS AND FUTURE WORK

Technically, our method can only be considered exact if input and output are in integer homogeneous coordinates subject to the fixed-width precision limits. Conversions to and from this format typically incur quantization steps that might re-introduce tiny self-intersections, though this is basically a problem of every algorithm interfacing with a floating-point world. However, as the output of our method can be used as its input again, iterated CSG operations can be performed without intermediate loss of precision.

We require no input topological information and process a soup of convex polygons. In particular, the output is not triangulated and contains T-junctions. As our method is exact, topological information can be recovered as a post-process. However, it should also be possible to track all required information during subdivision and reconstruct the topology on-the-fly.

While our method is already extremely fast, there are still several improvements we would like to investigate in the future. The parallelization has a certain burn-in phase that could be removed by starting multiple statically pre-clipped tasks. Additionally, there is still untapped potential in reducing the number of segment traces needed in leaf tasks. We already exploit various early-out opportunities. However, for large CSG trees and special use cases such as milling simulations, even more high-level knowledge could be used to speed up the operation. At a certain point, the WNV dimension might become so large that it warrants further optimization.

7 CONCLUSION

We have introduced a novel method, called EMBER, for computing mesh Booleans in a way that is exact, reliable, and extremely performant. Exactness follows from plane-based geometry using the integer homogeneous coordinates of [Nehring-Wirxel et al. 2021]. We extend their formulation to account for segments and polygons. Our classification is based on generalized winding number vectors (WNVs), which robustly handle any Boolean operation, including variadic ones and various special cases. Here, we introduce winding number transition vectors (WNTVs) that enable a localized segment tracing of WNVs. During subdivision, we always keep and trace a local reference position with known WNV. In the leaf computation, we can always trace segments to this local reference to classify results. Intersections are resolved by constructing face-local BSPs. The high performance of our method is owed to the fast fixed-width integer homogeneous coordinates, the localized handling of intersection resolution and classification, exploiting various early-out opportunities, and a well optimized multithreaded implementation.

Our evaluation on the Thingi10K data set shows that our method is orders of magnitude faster than the state-of-the-art, even inexact ones. A Boolean operation with 20 000 input faces takes only 1.6 ms on average.

We believe that our method enables previously intractable applications. At a speed of several million triangles per second and guarantees of exactness and robustness, mesh Booleans can now be used universally as a building block without reserve.

ACKNOWLEDGMENTS

We use the Thingi10K data set [Zhou and Jacobson 2016] for evaluation and illustrations. The Thingiverse models in our figures are: 11660 by m6mafia, 11782 by colah, 16673 by addy, 17336 by bld, 18284 by fong182, 21103 by chanso1, 43969 by CentaurDesign, 49865 by MakerBot, 51501 by CosmoWenman, 163032 by bourbon_and_cigars, 259005 by MakerBot, 311002 by LeFabShop, 584714 by Markellov,

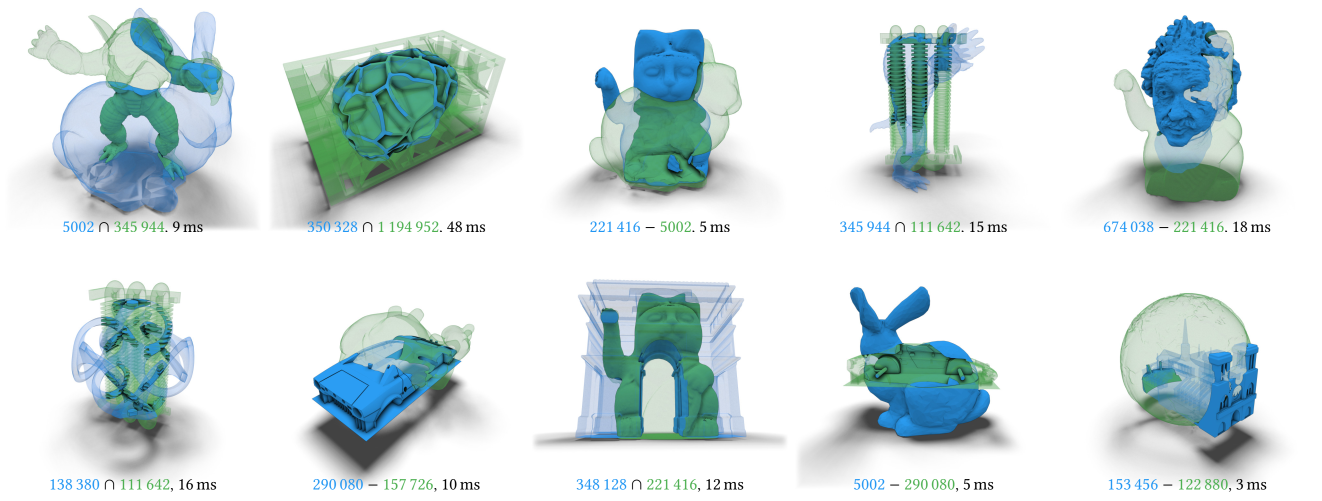


Fig. 19. Example outputs of our method with number of faces per transparent input and timings. Models from Thingi10K and the Stanford Scanning Repository.

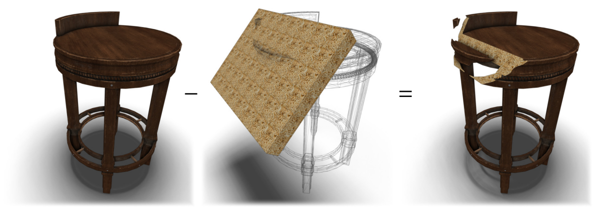


Fig. 20. Because we can identify the corresponding input polygon for each output polygon, properties like texture coordinates can easily be transferred from the input to the output.

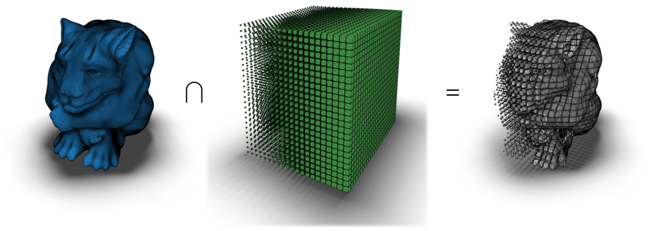


Fig. 22. Even with dense interactions, our method retains high performance. This example has roughly 350 000 input triangles and takes 43 ms.

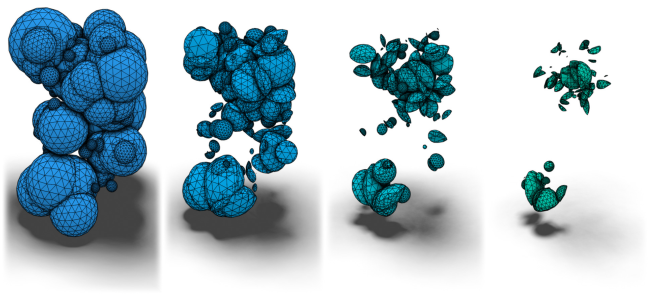


Fig. 21. WNV can be used to implement various variadic operations efficiently. In this example, we have a collection of spheres and extract the solid contained in at least 1, 2, 3, or 4 spheres (left-to-right). As we're computing the WNTV of each polygon, we can extract all layers at the same time. This example of 32 000 input triangles takes 15 ms.

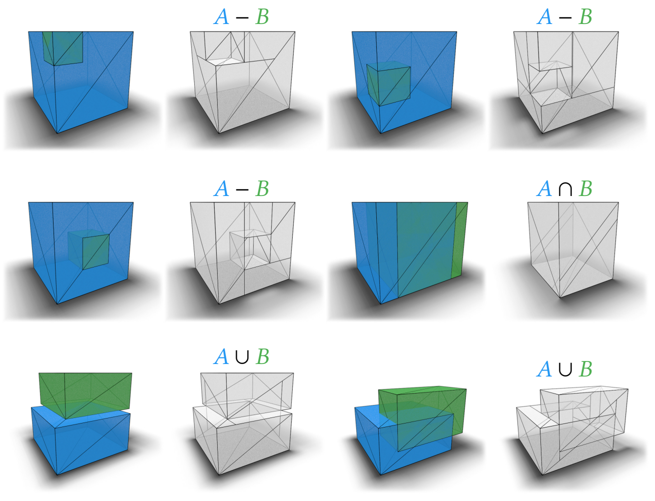


Fig. 23. Small collection of various corner and overlap cases. Includes challenges like coplanar faces and exact edge hits. The results are shown with transparency to demonstrate that no surface is emitted twice and the interior is empty.

787623 by dungbeetle24, 807591 by adafruit, 1081535 by ye3d. Model and texture of Fig. 20 are from [Sanchez 2021].

REFERENCES

Marco Attene. 2020. Indirect Predicates for Geometric Constructions. *Computer-Aided Design* 126 (Sep 2020), 102856. https://doi.org/10.1016/j.cad.2020.102856

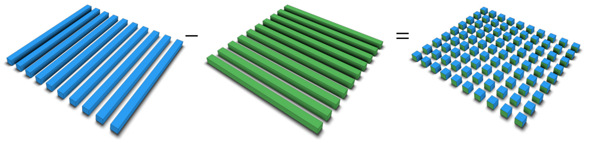


Fig. 24. 10-by-10 beams whose intersection produce 100 cubes. This shows a worst-case configuration: Linear input complexity produces quadratic output. For 20-by-20 beams, our method takes 4.5 ms, while mesh arrangements takes 18 018 ms, floating-point mesh arrangements 1556 ms, CGAL 4.12 6465 ms, and CGAL 5.4 941 ms. The non-exact methods produce severely corrupted results.

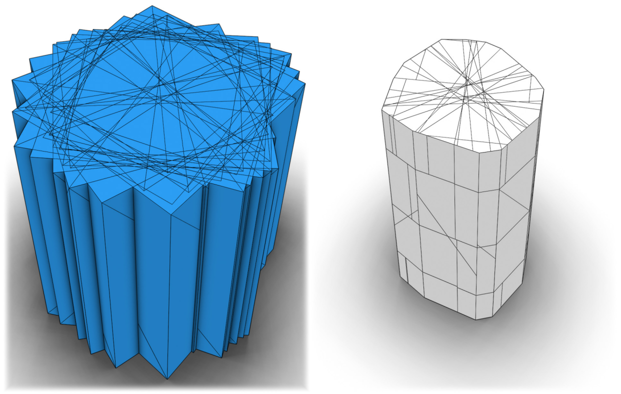


Fig. 25. Intersection of 20 cubes that share the same top and bottom plane, thus creating regions with up to 20 overlapping polygons at the same position. This case is extremely challenging and many optimizations are ineffective. Though consisting of only 240 triangles, this example takes 5.9 ms to compute with our method. Mesh arrangements takes 58 280 ms and floating-point mesh arrangements 5620 ms.

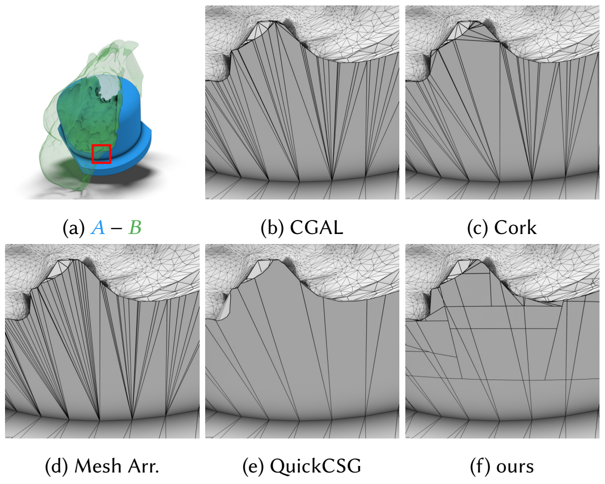


Fig. 26. Closeup of a mesh difference and the tessellations produced by various methods. CGAL, Cork, and Mesh Arrangements create triangulations, we emit a soup of convex polygons, and QuickCSG even concave ones.

Hichem Barki, Gael Guennebaud, and Sebti Foufou. 2015. Exact, robust, and efficient regularized Booleans on general 3D meshes. *Computers & Mathematics with Applications* 70, 6 (2015), 1235–1254.

Gilbert Bernstein. 2013. Cork Boolean Library. https://github.com/gilbo/cork Accessed January 19, 2022.

Gilbert Bernstein and Don Fussell. 2009. Fast, Exact, Linear Booleans. In *Proceedings of the Symposium on Geometry Processing* (Berlin, Germany) *(SGP '09)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1269–1278.

Marcel Campen and Leif Kobbelt. 2010. Exact and robust (self-) intersections for polygonal meshes. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 397–406.

Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and Robust Mesh Arrangements Using Floating-Point Arithmetic. *ACM Trans. Graph.* 39, 6, Article 250 (Nov. 2020), 16 pages.

Salles Viana Gomes de Magalhães, W Randolph Franklin, and Marcus Vinícius Alvim Andrade. 2020. An Efficient and Exact Parallel Algorithm for Intersecting Large 3-D Triangular Meshes Using Arithmetic Filters. *Computer-Aided Design* 120 (2020), 102801.

Matthijs Douze, Jean-Sébastien Franco, and Bruno Raffin. 2017. QuickCSG: Fast Arbitrary Boolean Combinations of N Solids. *arXiv preprint arXiv:1706.01558* (2017).

Torbjörn Granlund and the GMP development team. 2020. GNU MP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/.

Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn. 2007. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. *Computational Geometry* 38, 1-2 (2007), 64–99.

Alec Jacobson, Ladislav Kavan, and Olga Sorkine. 2013. Robust Inside-Outside Segmentation using Generalized Winding Numbers. *ACM Trans. Graph.* 32, 4 (2013).

Bruce Naylor, John Amanatides, and William Thibault. 1990. Merging BSP trees yields polyhedral set operations. *ACM Siggraph Computer Graphics* 24, 4 (1990), 115–124.

Walter Nef. 1978. Beiträge zur Theorie der Polyeder: Mit Anwendungen in der Computergraphik. (1978).

Julius Nehring-Wirxel, Philip Trettner, and Leif Kobbelt. 2021. Fast Exact Booleans for Iterated CSG using Octree-Embedded BSPs. *Computer-Aided Design* 135 (2021), 103015.

Dairon Sanchez. 2021. Bar Chair Round 01. https://polyhaven.com/a/bar_chair_round_01 Accessed January 19, 2022.

Jonathan Richard Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–363.

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Trans. Graph.* 35, 4, Article 39 (July 2016), 15 pages.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).