Interactive and Robust Mesh Booleans

GIANMARCO CHERCHI, University of Cagliari, Italy
FABIO PELLACINI, Sapienza University of Rome, Italy
MARCO ATTENE, CNR IMATI, Italy
MARCO LIVESU, CNR IMATI, Italy
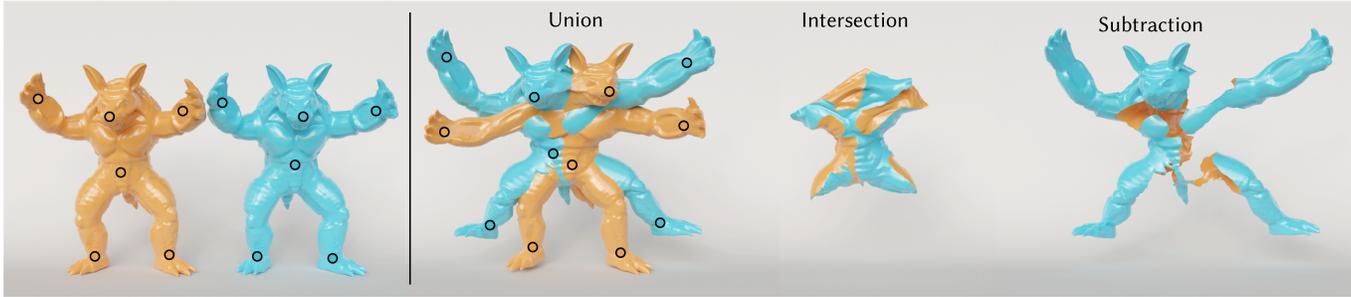
Fig. 1. We allow users to perform robust Boolean operations in real time on non trivial meshes containing thousands of triangles. In this example, the user first selects an arbitrary number of deformation handles (left). During the interactive session the handles can be freely moved in space, and the system applies both As-Rigid-As-Possible deformation [Sorkine and Alexa 2007] and our robust Booleans in real time. The two meshes of the armadillo contain 50K triangles each.

Boolean operations are among the most used paradigms to create and edit digital shapes. Despite being conceptually simple, the computation of mesh Booleans is notoriously challenging. Main issues come from numerical approximations that make the detection and processing of intersection points inconsistent and unreliable, exposing implementations based on floating point arithmetic to many kinds of degeneracy and failure. Numerical methods based on rational numbers or exact geometric predicates have the needed robustness guarantees, that are achieved at the cost of increased computation times that, as of today, has always restricted the use of robust mesh Booleans to offline applications. We introduce an algorithm for Boolean operations with robustness guarantees that is capable of operating at interactive frame rates on meshes with up to 200K triangles. We evaluate our tool thoroughly, considering not only interactive applications but also batch processing of large collections of meshes, processing of huge meshes containing millions of elements and variadic Booleans of hundreds of shapes altogether. In all these experiments, we consistently outperform prior robust floating point methods by at least one order of magnitude.

1 INTRODUCTION

Combining 3D meshes through Boolean operations is a fundamental functionality to define complex shapes via Constructive Solid Geometry (CSG). Despite being intuitive to the user, mesh Booleans are complex to implement correctly since small numerical errors may lead to unpredictable topological errors. Interactive modeling software is mostly based on non-robust methods to achieve interactivity during modeling, but this leads to significant workflow problems for users, as anecdotally shown by the galleries of failure cases that populate user forums. Professional software such as Autodesk Maya and Blender openly report instability issues in presence of challenging configurations such as coplanarity [Blender Doc 2022; Maya Doc 2022].

Academic research has been studying the problem for a long period, developing robust algorithms and sometimes providing free-to-use implementations, such as [Jacobson et al. 2018]. To this day though, robust methods either demand the use of integer coordinates everywhere [Trettner et al. 2022], making Booleans not natively compatible with alternative floating point geometry processing tasks, or are still too slow to secure interactive frame rates, relegating their use to offline modeling applications, e.g. for engineering and fabrication [Alderighi et al. 2019, 2018; Attene 2018; Dai et al. 2018; Fanni et al. 2018; Garg et al. 2016; Jacobson 2017; Muntoni et al. 2018; Nuvoli et al. 2019; Ureta et al. 2016; Yao et al. 2017].

In many existing methods, the calculation of a mesh Boolean is framed as a two step process. In the first step, conflicts between mesh elements are resolved, splitting triangles in order to incorporate intersection lines in the connectivity. In the second step, each mesh element is deemed as being inside or outside each input object. The result of a Boolean is eventually computed as a subset of the mesh
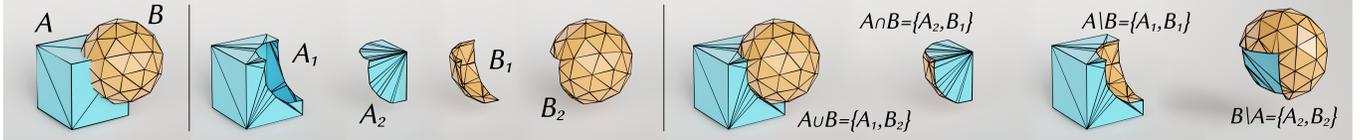
Fig. 2. Computing a Boolean between two shapes amounts to: (i) resolve mesh intersections, creating a set of conforming surface patches; (ii) merge the patches to form the output depending on the Boolean operator of choice.
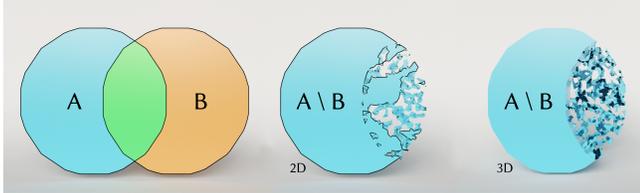
Fig. 3. Two failure examples of Cork [Bernstein 2013].The Euler characteristic of the 2D version (middle) of $A \setminus B$ is -50. The one of the 3D version (right) is 16. The Boolean with thickened disks was also tested on the the experimental tool available in Geogram/Graphite [Levy 2022] – which is restricted to solid objects – leading the program to a crash.

elements generated in the first step, filtered according to the inside/outside labeling computed in the second step. For example, the union $A \cup B$ of two triangle meshes A and B is the set of triangles in A that are outside B plus the triangles in B that are outside A (Figure 2).

The major difficulty in implementing a Boolean pipeline comes from the use of finite precision arithmetic, which does not allow to exactly represent and test intersection points. In practice, this translates into a variety of artifacts that span from the definition of incomplete intersection lines to topologically inconsistent partitions that make the inside/outside relations ill-defined. A typical failure is shown in Figure 3, where the subtraction between two largely overlapping discs has significant issues. Robust methods for inside/outside partitioning exist, but they are either computationally inefficient [Jacobson et al. 2013] or are efficient at query time but are approximate and require initialization [Barill et al. 2018]. The alternative is to eschew the use of floating point arithmetic and represent point coordinates with rational numbers [Zhou et al. 2016] or implicitly [Diazzi and Attene 2021]. These techniques ensure a topologically correct result at the cost of 1 to 2 orders of magnitude slow down w.r.t. floating point arithmetic, and are therefore not suitable for interactive use either.

In this paper we bring the robustness of exact floating point methods into the world of interactive applications for general meshes, by improving the efficiency of exact algorithms by at least one order of magnitude compared to the state of the art, and without any pre-processing. As demonstrated in Section 6, this makes robust mesh Booleans available when interactively editing meshes with up to 200K triangles on commodity laptops. Besides interactive applications, our algorithm performs significantly better than the state of the art in many practically relevant offline applications, spanning from batch processing of large collections of data, Booleans

between production-level high resolution meshes with millions of triangles, and variadic Booleans involving hundreds of input shapes.

Our improvements are made possible by significant contributions to both steps of the Boolean pipeline. For the first part, our method is based on a derivation of the mesh arrangements described in [Cherchi et al. 2020], which we improved as detailed in Section 4, obtaining an average speedup of more than 5×. For the second part, we exploit the guaranteed topological correctness of the arrangement, coupling it with a robust ray casting approach that allows to reliably compute the inside/outside labels by throwing a single ray per patch. Taking inspiration from robust predicates [Attene 2020; Lévy 2016; Shewchuk 1997], we formulate the inside/outside tests as a cascaded sequence of ray casting methods, sorted from faster but non-robust to slower but robust. We eventually resort to fully exact, thus fully robust, ray casting only when strictly necessary. As detailed in Section 5 this solution is up to 100× faster than existing approaches based on topological flooding [Attene 2014] or patch graph processing [Zhou et al. 2016] used by previous exact methods, and also scales optimally to big meshes composed of millions of triangles and variadic Booleans involving hundreds of meshes (Section 6).

All in all, we believe this work considerably improves upon the state of the art in terms of numerical speedup and, perhaps more importantly, it brings robustness to interactive applications. Our experiments begin to explore ideas on how robust Booleans can be used in real-time in combination with other geometry processing tasks, though this is just scratching the surface of what future interactive applications may be able to do. To support these experiments together with future research, we make our prototype implementation available as open source at the following *link*.

2 STATE OF THE ART

Boolean compositions can be displayed without computing an explicit 3D model of the result. This is sufficient for visualizing the result, and for a few other specific applications [Baxter and Wrigh 2019; Chen et al. 2022; Zanni et al. 2018]. In general though, modeling systems require an explicit representation of the Boolean composition and, depending on the target application, its calculation may need to be robust and exact. Existing algorithms can be classified based on the numerical model employed, e.g. floating point vs exact arithmetic, the geometric approach, e.g. surface vs volume-based, or the type of result they produce, e.g. exact vs approximated.

2.1 Numerical models

The easiest approach to implement Boolean operations is to rely on floating point arithmetic. Due to the hardware support, floating point numbers are by far the fastest approach and are indeed used widely in both academic [Levy 2022] and professional implementations, such as AutoDesk Maya and Blender. As mentioned in the introduction, round-off errors make this approach quite *fragile*, which in turn may lead to significant topological errors, shown for example in Figure 3. On the other extreme, unconditional robustness may be obtained by replacing floating point numbers with exact number types, for example with rational numbers [Schifko et al. 2010]. The use of exact arithmetic leads to unacceptable slowdown, by one or more orders of magnitude compared to floating point.

For some geometry processing tasks, robustness can be obtained by just guaranteeing that the program flow is exact independently of round-off errors. This is done by evaluating *geometric predicates* exactly and quickly through arithmetic filtering [Lévy 2016; Shewchuk 1997]. A typical predicate calculates the sign of a polynomial and, as long as the sign is correct, the program flow is guaranteed to be consistent. Arithmetic filtering [Devillers and Preparata 1998] makes it possible to evaluate a polynomial using floating point arithmetic but, along with it, an upper bound for the rounding error is computed. If the magnitude of the evaluated expression is larger than the error bound, its sign is guaranteed correct. If not, the filter *fails*, and the predicate is re-evaluated using arbitrary precision. If the failure rate is low enough, absolute precision rarely comes into play and the slowdown is acceptable [Magalhães et al. 2017], in particular when parallel architectures are employed [de Magalhães et al. 2020].

Arithmetic filtering can only be employed if the predicate input is guaranteed to be globally consistent. It is therefore used only for tasks where the input comes from the ground truth, such as in mesh generation, where the predicate works directly on the coordinates of the input points [Hang 2015; Shewchuk 1996]. For geometric tasks that rely on *intermediate constructions*, the state of the art solution is lazy exact evaluation [Pion and Fabri 2011], which is still too slow for interactive applications. As observed multiple times, for the case of mesh Booleans the only intermediate constructions are the intersection points [Bernstein and Fussell 2009; Campen and Kobbelt 2010a,b; Sugihara et al. 1989]. Such points can be implicitly represented as the intersection of input primitives and the point's expression can be composed with the predicate's expression, enabling the use of arithmetic filtering [Attene 2020]. Our method exploits this latter technique to perform fast and exact geometric queries involving any combination of input and intersection points.

2.2 Surface and volume-based methods

Methods that implement one or both of the steps of the Boolean pipeline may work either with an explicit volumetric mesh or with a surface mesh enclosing the volumes of interest. In the general case, explicit volumetric representations make the algorithm easier (e.g. for in/out labeling) but are less efficient due to the increased dimensionality. Surface based methods are a bit more convoluted but in general more parformant. Our method belongs to this latter category.

Volume-based. Pioneering works supporting Boolean operations were mainly based on simple volumetric primitives (e.g., spheres, cylinders, half-spaces) or on implicit representations for more generic inputs [Pasko et al. 1999; Wyvill et al. 1999]. Besides some notable exception [Sellán et al. 2021], the growing shape complexity in modern geometric modeling has gradually pushed the use of explicit mesh-based representations. Based on CGAL's exact kernel, in [Hu et al. 2018] input meshes are used to partition the space into conforming volumetric cells, obtaining a mesh arrangement. Its faster version [Hu et al. 2020] uses floating point calculations to create a conforming tetrahedral mesh, though with no formal guarantees. The creation of a volumetric mesh which conforms to the input was also used in [Diazzi and Attene 2021], where exact arithmetic was replaced by indirect predicates and Boolean operations could be extracted at a much higher speed while maintaining the correctness guarantees. The problem of partitioning the space based on input facets was also tackled in [Paoluzzi et al. 2020, 2019] based on floating point computation, using the language of geometric and algebraic topology. In [Tao et al. 2019], the fragility of floating point computation was reduced by using axis-aligned planes to cut the input triangles and define the volumetric cells, but being purely based on floats this method does not provide guarantees of correctness.

Surface-based. Among surface-based methods, exact constructions are used in [Schifko et al. 2010] while walking on the input surfaces and splitting triangles when intersections are encountered. Since using exact constructions is expensive, [Attene 2014] proposes a hybrid approach that is still based on walking on the outer surface, but uses floating point constructions whenever the rounding is harmless, while still switching to exact arithmetic when necessary. In [Xu and Keyser 2013], the topology of the resulting surface is guaranteed correct thanks to a clever use of orientation predicates with no need of exact constructions. Instead of walking on the surface, [Mei and Tipper 2013] uses a temporary octree to quickly find the candidate pairs of intersecting triangles. [Zhou et al. 2016] exploit CGAL's exact kernel to partition the space into cells, each labeled with winding numbers w.r.t. the input. Boolean operations are performed by simply selecting a subset of the cells according to their labels. The algebraic composition of intersection points and predicates introduced in [Attene 2020] was exploited in [Cherchi et al. 2020] to quickly transform an arbitrarily self-intersecting soup of triangles into a well-formed simplicial complex. Despite being the fastest arrangement algorithm available, the original implementation of [Cherchi et al. 2020] is still not sufficiently fast for interactive applications. The first step of our Boolean pipeline is heavily based on [Cherchi et al. 2020], but we substituted some of its modules and redesigned the data structures and algorithm flow paths to make it more amenable to parallel execution (Section 4), obtaining an average speedup of 5× (Section 6).

2.3 Exact and approximated methods

When applications tolerate an approximation, input vertex coordinates can be converted to triangle plane coefficients that can be easily combined to implement Boolean operations [Bernstein and Fussell 2009]. Since the conversion is not exact, the result typically

needs to be repaired. Other noticeable options to produce approximated Booleans are [Barill et al. 2018] and [Hu et al. 2020] where robustness and speed are combined. In [Pavić et al. 2010], regions close to the intersection lines are finely remeshed instead of being exactly cut.

In some cases, however, approximations cannot be tolerated. As an example, consider an interactive modeling system where a designer can perform numerous, subsequent and unpredictable operations: approximation would quickly accumulate and lead to poor results. In these cases, exact results can be obtained based on slow arbitrary precision [Schifko et al. 2010; Zhou et al. 2016], or by cleverly considering the floating point rounding. In [Campen and Kobbelt 2010a] input edges are split so that they become short enough to guarantee that the coordinate-plane conversion in [Bernstein and Fussell 2009] is lossless and can then be used to produce Boolean composition and other interesting modeling operations [Campen and Kobbelt 2010b] with no repairing. The problem with these methods is that the plane-based result becomes exceptionally complex when operations are cascaded. In an attempt to reduce this effect, in [Sheng et al. 2018] both the coordinates and the planes are used to reduce the need for conversions. Alternatively, an effective approach to simplify intermediate plane-based representations has been recently reported in [Nehring-Wirxel et al. 2021]. In a recent trend of works, indirect predicates are used as a replacement for intermediate constructions when calculating mesh arrangements [Cherchi et al. 2020] or polyhedral space subdivisions [Diazzi and Attene 2021]. When used to calculate mesh Booleans, this approach guarantees robustness without sacrificing speed. Nonetheless, if the explicit volume is not necessary for the final application, its calculation represents a useless overhead.

Snap rounding. Though exact methods are clearly useful, they have a common problem when the result needs to be saved to a file or passed to other, non exact, algorithms. In these cases exact or implicit coordinates must be converted to inexact floating point values and the necessary approximation may invalidate the model by introducing degenerate or intersecting elements. A provably correct and efficient solution to this problem is still elusive and existing algorithms are either impractical [Devillers et al. 2018; Fortune 1999] or do not guarantee to produce a correct result in all the cases [Milenkovic and Sacks 2019]. However, existing heuristics proved to fail only in an extremely small percentage of practical cases [Zhou et al. 2016]. It is worth noticing that meshes created with exact methods endow topological properties that cannot be obtained by approximate methods. For example, the result of a Boolean operation between two watertight manifold meshes that do not touch tangentially is guaranteed to be manifold watertight. Properties of this kind may be relevant also for downstream applications, regardless of the geometric degeneracies that snap rounding may introduce in the output.

2.4 In/out classification

A common problem in most mesh Boolean algorithms is determining whether elements are part of the result or not. In surface-based methods, these elements are triangles and one needs to know whether they bound the result or not. In volume-based methods, elements

are cells that might or might not be part of the (volumetric) result. A widely used approach for surface-based methods is to *walk* on the surface and track the portions that belong to the result based on geometric reasoning [Attene 2014; Schifko et al. 2010]. Exact methods guarantee that the input arrangement is well-formed, hence cells in a volumetric decomposition can be easily classified by starting from the *infinite* external cell and possibly switching from exterior to interior (and vice versa) when portions of the input surface are crossed. These approaches are clearly incompatible with naive floating point implementations, because of the lack of topological guarantees. When the input has surface holes or self-intersects in an ambiguous way, the concept of generalized winding number proved to be effective [Jacobson et al. 2013], though in some cases algorithms based on graph cuts provide better solutions [Diazzi and Attene 2021]. Since a naive computation of winding numbers is too slow in practice, a faster though approximate algorithm was proposed in [Barill et al. 2018]. These methods are in general slower than topological walking, but constitute an unavoidable cost to pay for inexact methods based on naive floating points. The inside/outside classification system we use in this paper is based on exact ray casting (Section 5) and proved to be much faster than both types of approaches (Section 6), also exhibiting a much better scalability on complex variadic Booleans containing hundreds of input shapes (Section 6.4).

Very recently, a robust and interactive Boolean method called EMBER was presented [Trettner et al. 2022]. This algorithm is based on the use of homogeneous integer numbers to represent point coordinates exactly. Thanks to aggressive parallelization, EMBER is the fastest existing method for mesh Booleans and is capable of operating interactively on meshes containing millions of elements. On the negative sides, the choice of integer coordinates makes EMBER not natively compatible with existing geometry processing tasks, demanding a (lossy) conversion of input point coordinates that must occur at any frame for interactive applications (in case Booleans are chained with some float-based task). Furthermore, their parallelization model requires splitting the input meshes into smaller chunks, introducing unnecessary seams in addition to the intersection lines, possibly spoiling the output of algorithms that exploit mesh conformity, such as [Nuvoli et al. 2019]. All in all, we believe that EMBER and our system are orthogonal takes on fast and robust Booleans. EMBER favors speed at the price of requiring changes to the input meshes. We favor compatibility with existing geometry processing algorithms by keeping mesh inputs intact, at the price of possibly slower execution speeds.

3 OVERVIEW

Our method takes as input a set of input meshes $M_1, M_2, \ldots, M_n$, and a Boolean operator, namely union, intersection, subtraction. Input meshes are always assumed to unambiguously enclose a volume, that is, they are manifold, watertight and with no self-intersections. The output is a mesh B that contains the result of applying the Boolean operator to the input meshes. By definition of Boolean, B is a sub-volume of the input, hence it is bounded by portions of input triangles. Mapping the result of a Boolean to the input shapes is
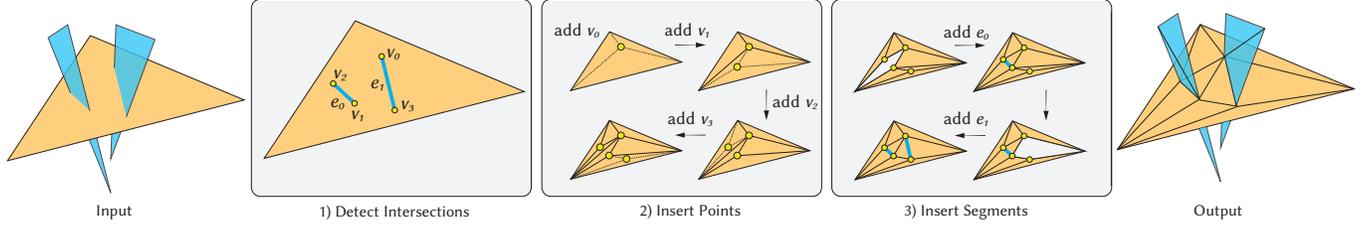
Fig. 4. First step of the Boolean pipeline. We start from an generic triangle soup (left) and detect intersection points and lines (1). We then split triangles to incorporate new points first (2), and then segments (3). The output is a well formed simplicial complex (right).

useful in many applications, therefore for each output triangle we propagate information on its origin. Note that an output triangle can belong to *many* input meshes, for example in the case of meshes that overlap at a coplanar region.

As anticipated in Section 1, the Boolean algorithm may be regarded as a two-steps pipeline. In the first step, intersecting mesh elements are split and intersection lines are incorporated in the elements connectivity. When exact methods are used, splitting intersecting elements is guaranteed to yield a well formed simplicial complex, where surface patches are bounded by closed loops of non-manifold edges, namely the intersection lines. We take advantage of this property in the second phase of our algorithm, that takes the surface patches as input and processes each of them to determine whether it is positioned inside or outside with respect to each of the input meshes $M_1, M_2, \ldots, M_n$. The output of the algorithm is eventually obtained by filtering the patches according to this information. For example, the union of two triangle meshes M_1 and M_2 ($M_1 \cup M_2$) is the set of patches of M_1 that are outside M_2 plus the patches of M_2 that are outside M_1. In the following two sections, we detail our technical contributions to each step of the pipeline. Visual examples for all Boolean operators we support are shown in Figure 2.

4 INTERSECTION RESOLUTION

From the perspective of this module, the input meshes $M_1, M_2, \ldots, M_n$ can be seen as a soup of possibly intersecting triangles. We therefore flatten all input triangles into a single array, associating to each triangle a tag that maps it to the input mesh it belongs to.

At the highest level, all existing algorithms for intersection resolution operate in a similar fashion, detecting intersections between triangles first, and eventually proceeding with mesh refinement, inserting intersection points first, and then segments [Attene 2014; Cherchi et al. 2020; Zhou et al. 2016] (Figure 4). Differences between the various methods are in the fine technical details, such as how intersection points are represented and processed, or how segment insertion is performed. These choices are fundamental to ensure that the algorithm is fast, memory efficient, amenable to parallelization and able to scale well on large datasets. We based our implementation of the splitting step on the method described in [Cherchi et al. 2020]. Even though this is the fastest existing method in its class, it is not fast enough for interactive use on our target mesh size. We therefore introduced a few important improvements to the original pipeline, enhancing the detection of intersections and the insertion of intersection lines (steps 1 and 3 in Figure 4). Overall, we obtained

a speed up factor of $3 - 8\times$ w.r.t. the original algorithm of [Cherchi et al. 2020]. In the remainder of this section we detail all the major improvements that we introduced. For a broader discussion of the whole algorithm we refer the reader to the original article.

Cached Predicates. To obtain unconditional numerical robustness all operations involving the detection of intersections, point in triangle location for vertex insertion and re-triangulation for segment insertion (steps 1,2,3 in Figure 4) must be based on exact orientation predicates [Shewchuk 1997], which therefore constitute a computational bottleneck for the splitting algorithm. The most frequent operation is the so called `orient3D`, which locates a point in space with respect to a given plane. Given a point p and a plane passing through points a, b, c, the orientation amounts to computing the sign of the determinant

$$orient3D(a, b, c, p) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ p_x & p_y & p_z & 1 \end{vmatrix}$$

In its classical form, this determinant is evaluated from scratch at each predicate call. We observe that in our algorithm planes are known a priori (they are the supporting planes of the input triangles) and are tested multiple times against several different points. It is therefore convenient to compute the plane based portion of the determinant once and to use it each time the same plane is tested against a new point. Starting from this intuition, we rewrite the 4×4 determinant above as

$$orient3D(a, b, c, p) = -p_x \begin{vmatrix} a_y & a_z & 1 \\ b_y & b_z & 1 \\ c_y & c_z & 1 \end{vmatrix} + p_y \begin{vmatrix} a_x & a_z & 1 \\ b_x & b_z & 1 \\ c_x & c_z & 1 \end{vmatrix}$$
$$- p_z \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} + \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix}$$

thus obtaining a perfect separation between plane coefficients and point coordinates. We exploit this latter equation to cache, for each input triangle, the four 3×3 determinants, thus reducing each call to `orient3D` to a simple scalar product in 4D. Similar caching techniques were recently exploited for the tetrahedralizations of huge point clouds composed of billions of points [Marot et al. 2019] and are also used in Boolean pipelines based on plane representations [Nehring-Wirxel et al. 2021].

Segment Insertion. To make sure that intersection lines are correctly incorporated in the output mesh, not only intersection vertices but also intersection segments must be inserted (step 3 in Figure 4). Inserting a segment amounts to eliminating, from the current tessellation, all triangles that conflict with it, and then re-triangulate the so-generated polygonal pocket, while making sure that the wanted segment is part of the new tessellation. This is a classical yet widely studied problem in computational geometry [Shewchuk and Brown 2015]. In [Cherchi et al. 2020] segment insertion was performed using the earcut algorithm, which in its best implementation achieves $O(n^2)$ worst case complexity [Eberly 2008], with n being the number of polygon segments. We substituted earcut with a method recently introduced in [Livesu et al. 2021], which ensures optimal deterministic $O(n)$ complexity in all cases and is two orders of magnitude faster than the previously best performing existing method [Shewchuk and Brown 2015].

Low-level Implementation. To further improve execution speed, we redesigned the underlying data structure to increase the possibilities for parallel execution. Overall, we observe that the most costly operations of the splitting algorithm are the removal of duplicate and degenerate elements, the construction of adjacencies, the intersection computations and the final triangulation. The main costs of these operations can be reduced by executing their core components in a data-parallel manner, giving us high speed up while maintaining the same algorithm pipeline. We also optimize octree construction by using nested parallel constructs. Parallel constructs are executed by a work-stealing scheduler that ensures good balanced workloads. Note that we perform degeneracy removal and rebuilding of adjacencies each frame to ensure robustness while allowing any modeling operation to be performed by users. We considered additional parallelization opportunities in patch construction and final mesh extraction. In these case though, parallelism is harder to extract since it requires fine-grained locking. In fact, we tested these approaches but without gaining speed up, suggesting that a meshlet-based approach is likely needed to extract further parallelism from the pipeline [Mahmoud et al. 2021].

The final improvement we implemented was the use of specialized data structures to improve cache coherency, reduce memory fragmentation due to deletions, and reduce the overall pressure on the system memory allocator. Regardless of the data structure used to store the mesh, and the size of the mesh, many small memory operations, including deletions, are required to update the data structure, and these updates have significant performance implications. In the splitting step, the most expensive low-level data structures are sets, dictionaries and sparse graphs stored as adjacency lists. We optimize these data structures with three techniques. First, we use hash tables based on the swiss table design for sets and dictionaries, to both save memory and improve cache coherency. Second, we use arena allocators to reduce the pressure on the memory allocator and reduce overall fragmentation. Third, we use dynamic arrays with small-array optimization for adjacency lists. Overall these techniques provide a relevant speedup throughout the pipeline.

ALGORITHM 1: Inside/outside classification

Input: input meshes $M_1, \ldots, M_n$ and their split patches $P_1, \ldots, P_m$
Output: relative position of patches $P_1, \ldots, P_m$ w.r.t. the input meshes $M_1, \ldots, M_n$

for *each patch P* **do**
  Initialize P as being outside of $M_1, \ldots, M_n$;
  Define a ray r starting at point $p \in P$ towards point at infinite p_∞;   (Sec. 5.1)
  for *each input mesh M* **do**
    compute and sort intersections between r and M;   (Sec. 5.2)
    if *r and M intersect* **then**
      find intersecting triangle $t \in M$;   (Sec. 5.3)
      compute volume of tetrahedron (t, p_∞);   (Fig. 5)
      if *volume is negative* **then**
        set P as being inside M;
      end
    end
  end
end
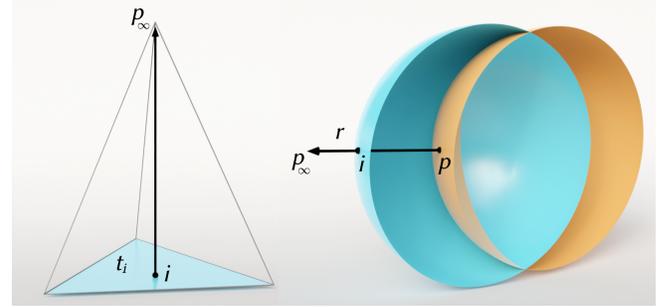
Fig. 5. For each manifold patch in the simplicial complex we robustly devise inside/outside relationships with input shapes with exact ray casting. We shoot a ray towards infinite and analyze its first intersection with all the input meshes. Given an intersection point i and the triangle t_i containing it, the ray traverses the mesh from inside to outside if the volume of the tetrahedron (t_i, p_∞) is negative, from outside to inside otherwise. This check can be performed exactly with arithmetic orientation predicates.

5 INSIDE/OUTSIDE CLASSIFICATION

In this second phase, we consider the simplicial complex computed at the previous step and determine, for each of its manifold surface patches, the relative position with respect to the input meshes $M_1, \ldots, M_n$. Differently from the previous step, which is an amelioration of an existing technique, the computation of the inside/outside classification is entirely different from the topological approaches used in prior art [Attene 2014; Zhou et al. 2016]. Our key insight is that the inside/outside relationship between a patch and an input mesh M can be determined by casting a ray from any patch point along an arbitrary direction and then analyzing its intersection with M (Figure 5, right). An important aspect of such an approach is that the algorithm scales with the number of manifold patches and not with the number of triangles in the mesh. Since the former is typically orders of magnitude lower than the latter and the cost of casting a single ray is almost negligible, the algorithm becomes remarkably fast. In our tests, we achieved up to 100× speedup compared to prior art, while also showing better scalability on variadic Booleans involving numerous input shapes (Section 6).

Algorithm 1 summarizes the main steps of our ray casting approach. For each input patch P, we construct a ray r that emanates

from it and points towards a point at infinity p_∞ that is guaranteed to stay outside of all input meshes. We then test intersections between r and each input mesh M. If at least an intersection occurs, we select the *first* of them, that is, the one that is closest to the point of P from where r emanates. To decide whether r traverses M from the inside to the outside or vice-versa, we compare the ray direction with the outfacing local surface normal. As shown in Figure 5 (left), such a check translates into the evaluation of the signed volume of a tetrahedron, which can be computed exactly using orientation predicates.

This ray casting approach poses several technical challenges. First, it can only be applied to *exact* arrangements computed with robust predicates or rational numbers, because alternative non-robust techniques cannot guarantee the absence of gaps or tiny topological channels connecting different patches. Second, intersection detection must be *exact* as well, because approximations in the computation may introduce artificial intersections or miss existing ones. Third, ambiguities that arise when the ray and the surface are tangent must be properly handled to ensure the correctness of the result. To make things even worse, we recall that intersection points inserted during the construction of the simplicial complex do not have known explicit coordinates, therefore computations must be fully compatible with implicit point representations. Note that this process is intrinsically unstable and demands absolute precision. A ray that misses an intersection because of a tiny geometric or topological imperfection may produce a wrong classification for an arbitrarily big patch. In the remainder of the section, we detail the main algorithmic steps, addressing all these aspects.

5.1 Ray definition

Given a patch P, we need to define a ray that starts at a point $p \in P$ and passes through an infinite point p_∞. If p is known, the infinite point p_∞ can be easily defined by translating p along one of the major axes by a quantity that is bigger than the extent of the bounding box of the input scene along the same axis (this guarantees that p_∞ is outside all input meshes). Moreover, having an axis-aligned ray considerably simplifies the next ray-triangle intersection analysis as it always allows to drop one coordinate and operate on a 2D plane instead of the 3D space.

The main difficulty in this phase is the definition of the emanating point p. In the simplest case, the patch P contains at least one input vertex with known ground-truth floating point coordinates in its interior, which can be used as a starting point for r. However, the simplicial complex may also contain patches that do not include input vertices or do not contain interior vertices at all. In all these cases, defining explicit floating point coordinates for p may be intrinsically impossible as any rounding risks to detach the ray r from the patch P and possibly trigger errors in the classification. As mentioned in previous sections, numerical issues may be fully avoided by switching to costly rational numbers to represent point coordinates, at the cost of a major slowdown. Instead, inspired by the cascaded approaches used by filtered predicates [Attene 2020; Lévy 2016; Shewchuk 1997], we define the ray r by first attempting to find a satisfactory approximate floating point solution, while we resort to guaranteed exact rational numbers only as backup strategy.

Our main idea is that if we can define a ray that starts from *beneath* P and is guaranteed to traverse the patch at some internal point $p \in P$, we can simply sort all the intersections we find, skip all intersections that occur up to p, and perform the inside/outside classification by considering the first intersection that occurs *after* p.

Our strategy is as follows: we pick a random triangle $t \in P$ and convert the coordinates of its implicit vertices into explicit floats, computing the approximate triangle barycenter b_t. To make sure that b_t stays beneath triangle t we push the point backwards along the same axis used to define p_∞. If the snap rounding of the triangle vertices succeeds, the ray starting from b_t and passing through p_∞ intersects triangle t, hence P, giving us a ray with which to perform the in/out classification. Unfortunately, this intersection is not guaranteed to exist because the conversion to floating point coordinates may move the ray away. Thanks to the axis alignment we can efficiently and reliably test that the wanted intersection exists with a simple 2D point in triangle test, performed considering the orthogonal projection of both t and b_t along the ray direction. If the test fails we attempt to produce the same construction with another triangle of P, until we find a valid one. If no valid triangle can be found, we compute the exact barycenter with rational numbers and perform an exact ray casting. In practice, patches that do not contain input vertices are rare, and failures of our approximate strategy are even more so. In our large scale benchmark (Section 6.2) we tested 3.8K Booleans, shooting more than 80K rays overall. Only the 2% of these rays necessitated to perform snap rounding. In the 97% of these cases we successfully defined a valid ray at the first triangle we tried. In 2% of the cases we tested two triangles, and in 0.56% of the cases we tested three triangles. In the worst case, we tested five triangles per patch, and we never hit the last step of our cascaded approach based on rational numbers.

5.2 Intersection detection

In a typical ray casting implementation, ray-triangle intersections are computed using the efficient Möller–Trumbore algorithm [1997]. Unfortunately, our exactness requirements make it impossible to rely on such an algorithm, which involves floating point operations that accumulate error and it is also non stable in case of coplanarity. For the same reason, acceleration data structures that rely on non axis aligned planes or spatial hashing cannot be used, because ray intersection queries would require arithmetic operations that introduce unwanted approximation errors. In our implementation, we use a plain octree as acceleration structure and perform ray casting by testing intersections between the ray bounding box and each octant. Note that, for efficiency, this is the same acceleration structure used in the splitting part to detect triangle intersections. Since both the octree and the ray are axis aligned we have two nice properties: the bounding box of the ray is tight (it's the ray itself), and the intersection with the octant reduces to a 2D check which involves only four comparisons between floats. For each leaf octant intersected by the ray we test all the triangles it contains. Once again, we exploit the axis aligned nature of our problem to recast the ray-triangle intersection as a point in triangle query in 2D, as previously described in Section 5.1. Thanks to this simplified
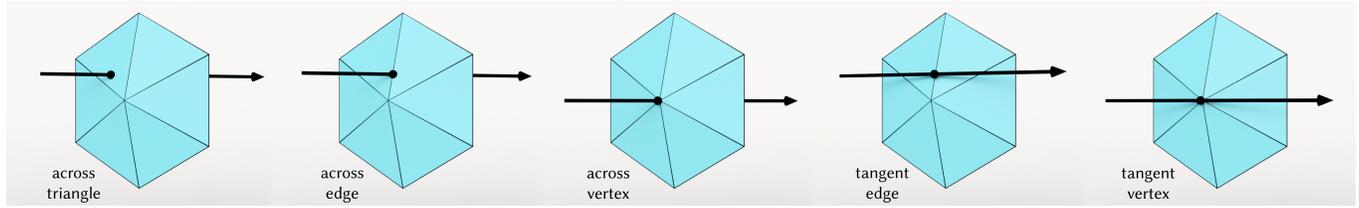
Fig. 6. Five alternative cases of intersection between a ray and a triangle mesh. When the ray crosses the surface at a point that is inside a triangle (left) the test depicted in Figure 5 can reliably determine the inside/outside relation between the patch being tested and the surface crossed. When the ray hits a vertex or an edge the check becomes ambiguous. Our method always reconducts to the leftmost case via numerical perturbation of the ray. Note that the pathological cases depicted in the figure are not exhaustive. In fact, rays may also be tangent at a (coplanar) triangle.

formulation, our exact ray-triangle intersection routine becomes extremely fast.

Since our classification is entirely based on the analysis of the first intersection between the ray and each input mesh, it is necessary to *sort* intersection points. Firstly, we represent intersections implicitly, using the LPI (Line-Plane Intersection) points described in [Attene 2020]. Then, we use the exact comparator introduced in [Cherchi et al. 2020] to sort them from the closest to the ray emanating point to the furthest.

5.3 Classification

The first intersection between a ray r and a mesh M may take place in different ways. The simplest configuration is when r crosses M at a point that is interior to a triangle (Figure 6, left). In this case determining if the ray is passing from inside to outside or vice-versa consists in simply analyzing the triangle orientation, which is encoded in the local winding (triangle vertex order) and can be tested exactly as shown in Figure 5 (left). The cases when the ray intersects a mesh edge or a vertex are more difficult, because the ray may traverse them tangentially without crossing the surface, and because even if the surface is crossed, the choice of the triangle to test is ambiguous. Four of these cases are depicted in Figure 6, but there are also others (e.g. when a tangent ray is also coplanar to a triangle). While exact geometric tests could be performed to distinguish between all these cases, the computational overhead for detection and handling of pathological cases can be substituted with a simpler yet effective strategy. Each time a ray does not intersect a patch triangle at an inner point, we perturb the coordinates of p_∞ by ϵ (without moving its starting point) until the crossing happens at a point that is interior to a mesh triangle. Perturbation of point coordinates is performed using the next floating-point number representable starting from a given number (using `std::nextafter`). The perturbed ray is then tested again for intersection, and the operation is repeated until a valid intersection point is found. Note that perturbed rays are no longer axis aligned, therefore we cannot reduce the ray-triangle intersection to a 2D problem, and we need to perform this test in 3D. The full test consists in checking the sign of three tetrahedra, formed considering the two ray endpoints and the endpoints of each triangle edge. If all signs are strictly positive, or negative, there is an intersection inside the triangle. The computational overhead of this check is minimal (three `orient2D` for the 2D case and three `orient3D` for the 3D case).

Intersections at mesh vertices and edges are extremely unlikely to happen in real shapes. In our large scale benchmark (Section 6.2), we tested 3.8K Booleans, shooting more than 80K rays overall. In no case we performed numerical perturbation because all intersections occurred inside mesh triangles. We were only able to validate this code on an artificial example, obtained by performing a Boolean operation between a mesh and a copy of it translated along the X axis. In this case all rays emanating from the patches of a mesh traverse the vertices of the other mesh. Indeed, a single perturbation was always sufficient to break ties and move the intersection point inside a triangle.

6 DISCUSSION

We implemented our Boolean pipeline in C++, using CinoLib [Livesu 2019] data structures for exact ray casting and detection of intersections and Indirect Predicates [Attene 2020] to robustly query intersection points. For maximum efficiency and parallel performance our code also relies on Google's Abseil fast containers and Intel's TBB.

We tested our algorithm thoroughly, considering interactive applications, batch processing of large collections of data, Booleans between huge meshes composed of millions of triangles, and variadic Booleans of hundreds of input meshes altogether.

Our baseline for comparative analysis is the method of [Zhou et al. 2016], which is the latest released fully fledged exact Boolean pipeline. Since the original authors' implementation was released, the codebase underwent various improvements, also very recently. Unless specified differently, all numbers we report refer to the most recent implementation available in libigl [Jacobson et al. 2018]. We also compare to [Cherchi et al. 2020], but since this method employed a costly intermediate tetrahedralization step to perform the in/out filtering, we restricted the comparison to the first step of the Boolean pipeline. Non-robust alternatives such as [Bernstein 2013; Levy 2022] are not considered because they do not guarantee the topological correctness of the result and are prone to failures (Figure 3).

As detailed in the remainder of this section, our Boolean algorithm proved to be superior than the state of the art by at least one order of magnitude in all experiments and is the only existing exact method capable of sustaining interactive frame rates for real-time applications.

Fig. 7. Two examples of our interactive rotation demo: one mesh rotates on top of the other while the system executes a Boolean operator in real time.

6.1 Interactive Applications

We considered both simple tasks where a scripted animation plays over time and fully dynamic tasks where all objects in the scene evolve over time in response of a user action. All interactive experiments were executed on a commodity laptop, a MacBook with M1 Pro with 8 performance cores and 32GB of RAM. Screen captures have been attached to the submission and are available to the reader to better judge the smoothness of the animation. Booleans are applied *naively*, meaning that each frame is computed separately, without propagating cached data from one frame to the subsequent. We leave this improvement for future work to obtain additional speedups.

Rotation demo. In this first test, two objects are rotated with respect to one another, while our algorithm computes the Boolean between the two, as shown in Figure 7. The user can interact with the system using the keyboard, selecting the type of Boolean operator between union, intersection and difference. This is the easiest interactive scenario: both objects are static up to a rigid movement of one of them. For maximum efficiency rendering and Booleans run in separate threads and are synchronized with double buffering. We considered scenes of growing sizes, in the range from 25K to 200K triangles. Based on our experience the way shapes intersect to each other has a non negligible impact on running times. For example, Booleans become increasingly complex when the two shapes are almost perfectly aligned, and progressively simpler otherwise. To reduce any possible bias we always used the same two models, remeshed at various resolutions using Graphite [2022]. Timings for each scene are reported in Table 1. As can be noticed, existing exact pipelines such as [Zhou et al. 2016] have computation times that are incompatible with interactive use already at the coarsest resolutions, running at roughly 1–2 fps for just 50K triangles. To give a comparative reference, we run at similar fps for 1M triangles. Using [Cherchi et al. 2020] for the first part of the pipeline yields running times that are compatible with interactive use for the coarsest resolutions, but the software is still not fast enough to scale on meshes containing more than 25-30K triangles. Our software runs

Size	libigl (t)	FA(t)	Ours (t)
25K	0.22/0.78/0.39	0.03/0.12/0.07	0.01/0.04/0.02
50K	0.36/1.58/0.50	0.12/0.20/0.13	0.02/0.06/0.04
100K	0.69/3.21/0.77	0.23/0.33/0.25	0.03/0.11/0.07
150K	1.03/4.86/1.13	0.34/0.45/0.36	0.05/0.14/0.10
200K	1.37/6.48/1.50	0.46/0.62/0.50	0.06/0.19/0.14

Table 1. Performances of the interactive rotation demo. For each scene we measure its size as the cumulative number of triangles and report minimum, maximum and average time per frame over a continuous run of 3 minutes. With libigl we denote the most recent implementation of [Zhou et al. 2016]. FA is a hybrid pipeline that uses the original implementation of the arrangement in [Cherchi et al. 2020] with our in/out classification based on ray casting. All timings are in seconds.

Size	min	max	avg
25K	0.028	0.055	0.032
50K	0.043	0.079	0.054
100K	0.085	0.119	0.095
150K	0.133	0.164	0.140
200K	0.188	0.215	0.195

Table 2. Performances of the interactive deformation demo. Timings (in seconds) report on the cumulative time to perform four ARAP iterations and our Boolean algorithm.

interactively for scenes containing up to roughly 100-120K triangles, and starts to lag a bit for larger scene sizes. We point the reader to the attached screen captures to get a better sense of the smoothness of the animation.

ARAP deformation. In this demo, two triangle meshes are interactively deformed using ARAP [Sorkine and Alexa 2007]. At first, the user prescribes an arbitrary number of handles on both shapes with mouse clicks. Then, matrices are factorized and the interactive session starts. During interaction the user can select any handle from both shapes and freely move it in space. The program updates the input meshes with ARAP and immediately performs a Boolean between them (Figure 1). Users can also interactively change the Boolean operator in real time. This demo is more challenging than

dragon1 - 1.3M raptor - 2.0M dragon2 - 2.2M horse - 2.2M neptune - 4.0M ganesha - 4.3M dragon3 - 7.2M lucy - 14.4M

Fig. 8. Dataset of big meshes we considered for the experiments in Table 3.

Input			libigl (t)			FA (t)	Ours (t)			libigl ($\times$)			FA ($\times$)
obj 1	obj 2	size	inters	bool	tot	inters	inters	bool	tot	inters	bool	tot	inters
dragon3	lucy	21.6M	142.81	46.86	189.67	97.49	12.90	6.61	19.50	11.07×	7.09×	9.73×	7.56×
lucy	neptune	18.4M	131.33	42.19	173.52	79.87	12.00	6.13	18.13	10.95×	6.88×	9.57×	6.66×
ganesha	lucy	18.7M	118.19	427.02	545.21	84.02	11.30	6.04	17.34	10.46×	70.76×	31.45×	7.43×
lucy	dragon2	16.6M	117.48	412.19	529.68	80.10	10.63	5.38	16.01	11.05×	76.67×	33.08×	7.53×
horse	lucy	16.6M	116.03	35.83	151.87	74.20	10.41	5.27	15.68	11.14×	6.80×	9.68×	7.12×
lucy	raptor	16.4M	110.12	384.48	494.60	68.10	10.01	5.27	15.28	11.00×	72.94×	32.37×	6.80×
dragon1	lucy	15.7M	110.27	30.12	140.40	69.26	9.60	4.90	14.49	11.49×	**6.15×**	9.69×	7.22×
dragon3	ganesha	11.5M	65.28	242.44	307.72	31.69	6.25	3.09	9.34	10.44×	78.38×	32.93×	5.07×
dragon3	neptune	11.2M	66.66	20.78	87.44	38.56	6.15	2.98	9.12	10.84×	6.98×	9.58×	6.27×
dragon3	horse	9.4M	56.25	15.03	71.28	26.62	5.19	2.31	7.49	10.85×	6.51×	**9.51×**	5.13×
dragon3	dragon2	9.4M	54.84	196.39	251.23	30.46	5.00	2.38	7.38	10.98×	82.38×	34.04×	6.10×
dragon3	raptor	9.2M	50.31	193.76	244.07	21.82	4.70	2.27	6.97	10.71×	85.28×	35.02×	4.65×
ganesha	neptune	8.3M	47.39	174.23	221.62	25.14	4.49	2.27	6.75	10.57×	76.82×	32.82×	5.61×
dragon1	dragon3	8.5M	51.05	12.95	64.00	27.82	4.67	2.06	6.72	10.94×	6.29×	9.52×	5.96×
neptune	dragon2	6.2M	44.53	139.26	183.78	32.65	4.02	2.04	6.06	11.08×	68.26×	30.33×	**8.12×**
ganesha	dragon2	6.5M	35.44	134.41	169.85	14.90	3.62	1.72	5.34	9.80×	78.15×	31.83×	4.12×
ganesha	horse	6.5M	35.22	130.04	165.26	13.46	3.48	1.67	5.15	10.12×	77.82×	32.08×	3.87×
neptune	raptor	6.0M	37.20	138.95	176.15	20.62	3.37	1.70	5.08	11.03×	81.64×	34.71×	6.11×
horse	neptune	6.2M	38.65	11.38	50.03	22.07	3.43	1.63	5.06	11.28×	6.99×	9.90×	6.44×
dragon1	ganesha	5.6M	32.93	119.43	152.35	11.59	3.47	1.39	4.86	9.50×	**85.79×**	31.37×	**3.35×**
ganesha	raptor	6.3M	29.68	125.36	155.04	11.87	3.18	1.61	4.79	**9.35×**	77.82×	32.39×	3.74×
dragon1	neptune	5.3M	32.91	8.99	41.90	18.55	2.76	1.40	4.15	**11.93×**	6.44×	10.09×	6.72×
horse	dragon2	9.4M	27.28	88.21	115.49	10.44	2.36	1.11	3.47	11.56×	79.40×	33.26×	4.42×
raptor	dragon2	4.2M	25.96	87.85	113.80	10.54	2.24	1.05	3.29	11.57×	84.06×	34.60×	4.70×
horse	raptor	4.2M	24.25	80.78	105.03	8.34	2.08	0.98	3.07	11.63×	82.18×	34.24×	4.00×
dragon1	dragon2	3.5M	22.22	69.96	92.17	8.33	2.01	0.86	2.88	11.04×	80.97×	32.04×	4.14×
dragon1	horse	3.5M	22.06	5.27	27.33	7.75	1.90	0.83	2.73	11.61×	6.35×	10.01×	4.08×
dragon1	raptor	3.3M	16.92	63.55	80.47	6.37	1.54	0.75	2.29	10.97×	84.96×	**35.13×**	4.13×

Table 3. Comparative analysis of our Boolean algorithm w.r.t. any combination of the huge models in Figure 8. For each test we report the name of the two objects and their cumulative size, in millions of triangles. With libigl we denote the best performing implementation of [Zhou et al. 2016], which endows a well crafted parallelization that fully exploits the recently improved thread safety of the lazy rational kernel in CGAL 5.4. FA is the author's reference implementation of the Fast Arrangement algorithm [Cherchi et al. 2020]. Running times are in seconds. All tested methods perform equally well on any Boolean operator, we therefore restricted experiments to Boolean unions only. The rightmost sections of the table, denoted with ×, report our speedups. On average, we are almost 25× faster than libigl and our intersection resolution step is more than 5× faster than [Cherchi et al. 2020]. Best and worst speedups for each algorithmic step are highlighted in bold blue and red, respectively. All tests were performed on a Mac Book M1 Pro with 8 performance cores and 32GB of RAM.

the previous one, since all inputs to the Boolean algorithm undergo non trivial dynamic changes at interaction time, and since the cost of ARAP is roughly equivalent to the cost of Booleans. To obtain maximum efficiency we execute the ARAP and Boolean steps in parallel on the same work stealing scheduler, using double buffering to batch operations within the stages. This allows us to hide latency as much as possible. In all our tests we always performed four iterations of ARAP, which are typically sufficient to obtain visually pleasant deformations. As can be noticed from the attached

video, deformations appear quite smooth up to scenes containing 100K triangles, and progressively lag for bigger scenes containing 150K and 200K triangles. Detailed numbers on our running times are reported in Table 2.

It should be noted that ARAP deformation offers no guarantees, and under extreme handle displacements may occasionally introduce self intersections that violate the input requirements of our method and may possibly spoil the in/out classification system. In our experiments we observed that artifacts of this kind occasionally

Fig. 9. Our method scales optimally to variadic Booleans involving hundreds of shapes. We executed the operations in the figure in two different ways: passing in input each decorative element separately, and merging all decorative elements into a single input mesh. In both cases the running time was the same, thus no computational overhead was introduced for the increased number of input shapes.

arose when rotation matrices where computed with [Zhang et al. 2021]. Using Eigen SVD [2010] for the computation of rotation matrices solved all our issues, although the algorithm becomes a bit slower (the local step was 1.5× faster using [Zhang et al. 2021]).

6.2 Large Scale Benchmark

We considered the popular Thingi10K [Zhou and Jacobson 2016] dataset to perform a large scale benchmark, comparing our performances with [Zhou et al. 2016]. We compared to the recently released implementation by the authors that includes significant parallelization efforts. Both methods require the input meshes to be manifold watertight, therefore we extracted 7628 *clean* meshes from the version of the database released by the authors of [Hu et al. 2018]. We halved these meshes in two groups of 3814 elements each and randomly combined them together to perform a Boolean operation. To make sure that the shapes actually intersect with each other we normalized their bounding box and centered them in the origin. Since all methods are exact, they are also guaranteed to produce exactly the same output topology. We exploited this property to validate our algorithm, verifying that indeed the number of connected components and Euler characteristic was the same for each output mesh. We used a machine equipped with an Intel i9 processor at 1.2 GHz with 12 cores and 128 GB of RAM as testing hardware. On this machine, [Zhou et al. 2016] can process all 3814 Booleans in 28.3 minutes, whereas our software terminated the same task in 4.5 minutes, also being faster in the 100% of the cases. Note that the recent update of libigl's implementation is a considerable speed up compared to the one used in the original paper, bringing the times from 80 minutes of the original publication to 28.3 minutes for their current version. Overall, both methods spent most of the computation in the splitting part: 22.5 minutes [Zhou et al. 2016] and 4 minutes ours (5.5×). For the Boolean part, [Zhou et al. 2016] spent 5.8 minutes while our tool completed in 0.47 minutes (12.2×). Remarkably, for the inside/outside classification our method based on ray casting was up to 101× faster than the one of [Zhou et al. 2016] in the best case (2.38× in the worst case, 11.34× on average).

6.3 Processing of Huge Meshes

Thingi10K is mostly populated by medium, small and very small meshes. We complement the large scale benchmark with a smaller test that focuses on high resolution meshes containing millions of triangles. While such big meshes are far from being suitable for interactive usage, this is the common size that can be found

in production in many industries and is therefore practically relevant. We considered the eight high resolution meshes shown in Figure 8, whose polygon count is in between 1.3 and 14.4 millions of triangles. We performed a Boolean operation for any possible pair of meshes, measuring running times of our tool, with [Zhou et al. 2016] and with [Cherchi et al. 2020], the latter only for the splitting part. As shown in Table 3, all methods scale well on very large datasets, mostly maintaining a stable ratio between their running times. Our method consistently operates one order of magnitude faster than [Zhou et al. 2016] for the intersection resolution part, where it is also 5× faster than [Cherchi et al. 2020] on average. The highest variability occurs in the Boolean part, where the speedup compared to [Zhou et al. 2016] oscillates around 80× (in 70% of the cases) and around 6× (in the remaining cases). All in all, our average speedup w.r.t. to the whole Boolean pipeline of [Zhou et al. 2016] is approximately 25×. It is interesting to notice that for most of these tests the bottleneck for [Zhou et al. 2016] was the second step of the Boolean pipeline, which took most of the running time. This never happened for the small models in Thingi10K, where the Boolean part was almost negligible compared to the intersection resolution one. We conjecture that the topological propagation used in [Zhou et al. 2016] does not scale well on very big meshes, while our novel approach based on ray casting remains efficient at all resolutions (53× faster on average) and its computation time never exceeded the cost of the intersection resolution in any of our experiments.

6.4 Variadic Booleans

Not only the mesh size but also the number of input objects affects the performances of a Boolean algorithm, especially for the second part of the pipeline, where the inside/outside relationships must be devised for each input shape. We evaluated the scalability of our method with respect to the number of input meshes involved in a Boolean operation, considering the subtraction between a base mesh A and a large number of non intersecting small decorative elements $B_1 \cup B_2 \cup \cdots \cup B_n$, positioned on its surface with Poisson sampling [Corsini et al. 2012]. To isolate the impact of the number of inputs, we performed this experiment twice. The first time we consider the case of $A \setminus \{B_1 \cup B_2 \cup \cdots \cup B_n\}$ as a variadic Boolean, that is, providing in input $n + 1$ separate meshes. The second time, we merge $B_1 \cup B_2 \cup \cdots \cup B_n$ in a single mesh $\dot{B}$ and then perform a classical pairwise operation $A \setminus \dot{B}$. Since decorations do not interfere with each other, the arrangement and the output result of these operations is identical. Figure 9 shows two results obtained

with our tool. In the first one, we carved the Fertility statue (60K triangles) with 700 little spheres (each one counting 320 triangles, for 224K triangles overall). Our algorithm was able to correctly compute the same result with both approaches in 1.1 seconds each time, thus introducing no measurable overhead for the increased number of input meshes. In the second experiment we subtracted from a vase (700K triangles) an assembly of 500 little stones of 10 different types (from 5 to 16K triangles each, 4.7M triangles overall) randomly oriented and positioned on the surface of the vase. Once again, our algorithm computed both the variadic and the pairwise Boolean in almost the same time (7.49 and 7.59 seconds, respectively). This is possible thanks to our inside/outside labeling based on ray casting, which shoots a ray for each surface patch regardless of the number of input shapes. Prior methods based on topological flooding do not exhibit the same desirable property and tend to introduce unnecessary overhead when the number of inputs grows. For example, on the first experiment the method of [Zhou et al. 2016] had a slowdown factor of more than 5× between the two runs, completing the Booleans in 6.8 and 38 seconds, respectively. On the second experiment their running times were 61.01 and 170.93 seconds (2.8×).

7 CONCLUSION

We have presented a novel pipeline for the computation of topologically exact mesh Booleans. Our main technical contributions amount to an amelioration of the arrangement algorithm in [Cherchi et al. 2020], and to a novel inside/outside classification system based on exact ray casting. As shown in our experiments both contributions are significantly faster than prior art, by at least one order of magnitude overall. Thanks to this speedup we could implement interactive applications that couple basic geometry processing tasks with real-time Booleans. This is the first time robust Booleans and interactive tools are coupled together for real sized meshes. To this end, we expected the community of digital artists and context creators to readily adopt our tools, and we are curious to see what they will be able to create with it.

7.1 Limitations and Future Works

Our system is currently limited in two aspects: inability to achieve interactive frame rates on very high resolution meshes (e.g. more than 200K triangles) and inability to robustly perform cascaded Boolean operations.

Scalability. The lowest hanging fruit to improve on the scalability of our method in interactive mode is to cache partially evaluated computations between frames. As discussed in Section 6.1, the algorithm is at the moment not designed to exploit temporal coherency and naively computes each Boolean from scratch, which is of course not optimal. Things like acceleration data structures used to detect intersection and perform the ray casting could be created once and minimally updated at each frame, greatly reducing the computational cost. Also adjacency data is now recomputed from scratch each frame, while they can be cache in most cases.

Cascaded Booleans. Our interactive tools are currently limited to CSG-like applications and other applications where each frame

can be generated independently from the previous ones. We currently do not support *cascaded* Booleans, where the inputs are the result of a previous Boolean operation. While in our tool it is technically possible to cascade Booleans, no guarantees on the result can be given because after each frame we snap exact coordinates to floating points, possibly introducing small mesh defects that may spoil subsequent operations. As mentioned at the end of Section 6.1 similar problems may also occur when Booleans are coupled with geometry processing tasks that do not guarantee the absence of self-compenetration, like ARAP. This issue was solved in [Diazzi and Attene 2021] by implicitly repairing the input during the process, though the need to construct an intermediate volume necessarily introduces a slowdown. Solving the snap rounding problem is the ultimate solution for all these issues, but this is a remarkably difficult problem as previously discussed. Avoiding the snap rounding step and propagating the implicit points naively is not possible either, because the repeated composition of implicit points would soon lead to complex polynomial expressions for which the arithmetic filtering fails in virtually all cases, introducing major slowdowns, and eventually running out of memory. On the one hand, we argue that such a repeated composition is not strictly necessary for cascading, because the output of a Boolean operation is always a subset of the input (ground truth) primitives, but we lack a proper mechanism to *update* the definition of implicit intersection points so as to avoid second level constructions. In the context of plane-based representations a similar idea was recently explored in [Nehring-Wirxel et al. 2021]. Attempting to realize cascading for the Indirect Predicates [Attene 2020] is an interesting direction for future research.

REFERENCES

Thomas Alderighi, Luigi Malomo, Daniela Giorgi, Bernd Bickel, Paolo Cignoni, and Nico Pietroni. 2019. Volume-aware design of composite molds. *ACM Transactions on Graphics* (2019).

Thomas Alderighi, Luigi Malomo, Daniela Giorgi, Nico Pietroni, Bernd Bickel, and Paolo Cignoni. 2018. Metamolds: computational design of silicone molds. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–13.

Marco Attene. 2014. Direct repair of self-intersecting meshes. *Graphical Models* 76, 6 (2014), 658–668.

Marco Attene. 2018. As-exact-as-possible repair of unprintable STL files. *Rapid Prototyping Journal* (2018).

Marco Attene. 2020. Indirect predicates for geometric constructions. *Computer-Aided Design* 126 (2020), 102856.

Gavin Barill, Neil G Dickson, Ryan Schmidt, David IW Levin, and Alec Jacobson. 2018. Fast winding numbers for soups and clouds. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–12.

Audrey Baxter and Lorelei Wrigh. 2019. Ray-Traced Constructive Solid Geometry. (2019).

Gilbert Bernstein. 2013. Cork Boolean Library. https://github.com/gilbo/cork.

Gilbert Bernstein and Don Fussell. 2009. Fast, exact, linear booleans. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1269–1278.

Blender Doc. 2022. Blender. https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/booleans.html

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

Marcel Campen and Leif Kobbelt. 2010b. Polygonal boundary evaluation of minkowski sums and swept volumes. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 1613–1622.

Shouxin Chen, Ming Chen, and Shengliu Lu. 2022. A Real Time Visual Boolean Operation on Triangular Mesh Models. (2022).

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. https://doi.org/10.1145/3414685.3417818

Massimiliano Corsini, Paolo Cignoni, and Roberto Scopigno. 2012. Efficient and flexible sampling with blue noise properties of triangular meshes. *IEEE transactions on visualization and computer graphics* 18, 6 (2012), 914–924.

Chengkai Dai, Charlie CL Wang, Chenming Wu, Sylvain Lefebvre, Guoxin Fang, and Yong-Jin Liu. 2018. Support-free volume printing by multi-axis motion. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–14.

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.

Olivier Devillers, Sylvain Lazard, and William J. Lenhart. 2018. 3D Snap Rounding. In *34th International Symposium on Computational Geometry (SoCG 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 99)*, Bettina Speckmann and Csaba D. Tóth (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:14. https://doi.org/10.4230/LIPIcs.SoCG.2018.30

O. Devillers and F. P. Preparata. 1998. A Probabilistic Analysis of the Power of Arithmetic Filters. *Discrete & Computational Geometry* 20, 4 (01 Dec 1998), 523–547. https://doi.org/10.1007/PL00009400

Lorenzo Diazzi and Marco Attene. 2021. Convex polyhedral meshing for robust solid modeling. *ACM Transactions on Graphics (TOG)* 40, 6 (2021), 1–16.

David Eberly. 2008. Triangulation by ear clipping. *Geometric Tools* (2008), 2002–2005.

Filippo A Fanni, Gianmarco Cherchi, Alessandro Muntoni, Alessandro Tola, and Riccardo Scateni. 2018. Fabrication oriented shape decomposition using polycube mapping. *Computers & Graphics* 77 (2018), 183–193.

Steven Fortune. 1999. Vertex-rounding a three-dimensional polyhedral subdivision. *Discrete & Computational Geometry* 22, 4 (1999), 593–618.

Akash Garg, Alec Jacobson, and Eitan Grinspun. 2016. Computational design of reconfigurables. *ACM Trans. Graph.* 35, 4 (2016), 90–1.

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Si Hang. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw* 41, 2 (2015), 11.

Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast tetrahedral meshing in the wild. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 117–1.

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (jul 2018), 14 pages. https://doi.org/10.1145/3197517.3201353

Alec Jacobson. 2017. Generalized matryoshka: Computational design of nesting objects. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 27–35.

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–12.

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Bruno Lévy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Computer-Aided Design* 72 (2016), 3–12.

Bruno Levy. 2022. Graphite. https://github.com/BrunoLevy/GraphiteThree.

Marco Livesu. 2019. cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes. *Transactions on Computational Science XXXIV* (2019). https://doi.org/10.1007/978-3-662-59958-7_4 https://github.com/mlivesu/cinolib/.

Marco Livesu, Gianmarco Cherchi, Riccardo Scateni, and Marco Attene. 2021. Deterministic Linear Time Constrained Triangulation using Simplified Earcut. *IEEE Transactions on Visualization and Computer Graphics* (2021).

Salles VG Magalhães, W Randolph Franklin, and Marcus VA Andrade. 2017. Fast exact parallel 3D mesh intersection algorithm using only orientation predicates. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 1–10.

Ahmed H Mahmoud, Serban D Porumbescu, and John D Owens. 2021. RXMesh: a GPU mesh data structure. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16.

Célestin Marot, Jeanne Pellerin, and Jean-François Remacle. 2019. One machine, one minute, three billion tetrahedra. *Internat. J. Numer. Methods Engrg.* 117, 9 (2019), 967–990.

Maya Doc. 2022. AutoDesk. https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Maya-Modeling/files/GUID-302821C5-343C-4F2B-8228-C5333896B207-htm.html

Gang Mei and John C Tipper. 2013. Simple and robust boolean operations for triangulated surfaces. *arXiv preprint arXiv:1308.4434* (2013).

Victor Milenkovic and Elisha Sacks. 2019. Geometric rounding and feature separation in meshes. *Computer-Aided Design* 108 (2019), 12–18.

Tomas Möller and Ben Trumbore. 1997. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools* 2, 1 (1997), 21–28.

Alessandro Muntoni, Marco Livesu, Riccardo Scateni, Alla Sheffer, and Daniele Panozzo. 2018. Axis-aligned height-field block decomposition of 3D shapes. *ACM Transactions on Graphics (TOG)* 37, 5 (2018), 1–15.

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.

Stefano Nuvoli, Alex Hernandez, Claudio Esperança, Riccardo Scateni, Paolo Cignoni, and Nico Pietroni. 2019. QuadMixer: layout preserving blending of quadrilateral meshes. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13.

Alberto Paoluzzi, Vadim Shapiro, Antonio DiCarlo, Francesco Furiani, Giulio Martella, and Giorgio Scorzelli. 2020. Topological computing of arrangements with (co) chains. *ACM Transactions on Spatial Algorithms and Systems (TSAS)* 7, 1 (2020), 1–29.

Alberto Paoluzzi, Vadim Shapiro, Antonio DiCarlo, Giorgio Scorzelli, and Elia Onofri. 2019. Finite Boolean Algebras for Solid Geometry using Julia's Sparse Arrays. *arXiv preprint arXiv:1910.11848* (2019).

Alexander Pasko, V Adzhiev, R Cartwright, E Fausett, A Ossipov, and V Savchenko. 1999. HyperFun project: A framework for collaborative multidimensional F-rep modeling. In *Eurographics/ACM SIGGRAPH Workshop Implicit Surfaces' 99*. 59–69.

Darko Pavić, Marcel Campen, and Leif Kobbelt. 2010. Hybrid booleans. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 75–87.

Sylvain Pion and Andreas Fabri. 2011. A generic lazy evaluation scheme for exact geometric computations. *Science of Computer Programming* 76, 4 (2011), 307 – 323. https://doi.org/10.1016/j.scico.2010.09.003 Special issue on library-centric software design (LCSD 2006).

Martin Schifko, Bert Jüttler, and Bernhard Kornberger. 2010. Industrial application of exact boolean operations for meshes. In *Proceedings of the 26th Spring Conference on Computer Graphics*. 165–172.

Silvia Sellán, Noam Aigerman, and Alec Jacobson. 2021. Swept volumes via spacetime numerical continuation. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–11.

Bin Sheng, Bowen Liu, Ping Li, Hongbo Fu, Lizhuang Ma, and Enhua Wu. 2018. Accelerated robust Boolean operations based on hybrid representations. *Computer Aided Geometric Design* 62 (2018), 133–153.

Jonathan Richard Shewchuk. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Workshop on applied computational geometry*. Springer, 203–222.

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

Jonathan Richard Shewchuk and Brielin C Brown. 2015. Fast segment insertion and incremental construction of constrained Delaunay triangulations. *Computational Geometry* 48, 8 (2015), 554–574.

Olga Sorkine and Marc Alexa. 2007. As-rigid-as-possible surface modeling. In *Symposium on Geometry processing*, Vol. 4. 109–116.

Kokichi Sugihara, Masao Iri, et al. 1989. A solid modelling system free from topological inconsistency. *Journal of Information Processing* 12, 4 (1989), 380–393.

Michael Tao, Christopher Batty, Eugene Fiume, and David IW Levin. 2019. Mandoline: robust cut-cell generation for arbitrary triangle meshes. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–17.

Philip Trettner, Julius Nehring-Wirxel, and Leif Kobbelt. 2022. EMBER: exact mesh booleans via efficient & robust local arrangements. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–15.

Francisca Gil Ureta, Chelsea Tymms, and Denis Zorin. 2016. Interactive modeling of mechanical objects. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 145–155.

Brian Wyvill, Andrew Guy, and Eric Galin. 1999. Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. In *Computer Graphics Forum*, Vol. 18. Wiley Online Library, 149–158.

Songgang Xu and John Keyser. 2013. Technical Note: Fast and Robust Booleans on Polyhedra. 45, 2 (feb 2013), 529–534. https://doi.org/10.1016/j.cad.2012.10.036

Jiaxian Yao, Danny M Kaufman, Yotam Gingold, and Maneesh Agrawala. 2017. Interactive design and stability analysis of decorative joinery for furniture. *ACM Transactions on Graphics (TOG)* 36, 2 (2017), 1–16.

Cédric Zanni, Frédéric Claux, and Sylvain Lefebvre. 2018. HCSG: Hashing for real-time CSG modeling. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 1–19.

Jiayi Eris Zhang, Alec Jacobson, and Marc Alexa. 2021. Fast Updates for Least-Squares Rotational Alignment. *Computer Graphics Forum* (2021).

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–15.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. https://doi.org/10.48550/ARXIV.1605.04797