An efficient and exact parallel algorithm for intersecting large 3-D triangular meshes using arithmetic filters

Abstract

We present 3D-EPUG-OVERLAY, a fast, exact, parallel, memory-efficient, algorithm for computing the intersection between two large 3-D triangular meshes with geometric degeneracies. Applications include CAD/CAM, CFD, GIS, and additive manufacturing. 3D-EPUG-OVERLAY combines 5 techniques: multiple precision rational numbers to eliminate roundoff errors during the computations; Simulation of Simplicity to properly handle geometric degeneracies; simple data representations and only local topological information to simplify the correct processing of the data and make the algorithm more parallelizable; a uniform grid to efficiently index the data, and accelerate testing pairs of triangles for intersection or locating points in the mesh; and parallel programming to exploit current hardware. 3D-EPUG-OVERLAY is up to 101 times faster than LibiGL, and comparable to QuickCSG, a parallel inexact algorithm. 3D-EPUG-OVERLAY is also more memory efficient. In all test cases, 3D-EPUG-OVERLAY's result matched the reference solution. It is freely available for nonprofit research and education at [CITATION REMOVED].

Keywords: CAD, Boolean operations, Parallel programming, Exact computation, Polyhedron intersection

1. Introduction

The classic problem of intersecting two 3-D meshes has been a foundational component of CAD systems for some decades. However, as data sizes grow, and parallel execution becomes desirable, the classic algorithms and implementions now exhibit some problems.

1. *Roundoff errors*. Floating point numbers violate most of the axioms of an algebraic field, e.g., $(a+b)+c \neq a+(b+c)$. These arithmetic errors cause topological errors, such as causing a point to be seen to fall on the wrong

side of a line. Those inconsistencies propagate, causing, e.g., nonwatertight models. Heuristics exist to ameliorate the problem, and they work, up to a point. Larger datasets mean a higher probability of the heuristics failing.

2. *Special cases (geometric degeneracies).* These include a vertex of one object incident on the face of another object, coincident vertices, coincident edges, etc. In principle, simple cases could be enumerated and handled. However, some widely available software fails, for several reasons.

   First, The number of special cases grows exponentially with the dimension. When intersecting a 2-D infinite line l with a polygon, (at least) the following cases occur with respect to the line's intersection with a finite edge e of the polygon: l crosses e's interior, l is coincident with e, and l is incident on a vertex v of e, and the other edge e' incident on v is either coincident with l, on the same side of l as e, or on the opposite side of l as e. In 3-D, the problem is much worse, so that a complete enumeration may be infeasible.

   Second, one technique is to reduce the number of cases by combining them. E.g., when comparing point p against line l, the three cases of *above*, *on*, and *below* may be combined into two: *above or on* and *below*. The problem is to do this in a way that results in higher level functions that call this as a component executing correctly. E.g., does intersecting two polylines, where a vertex of one is coincident with a vertex of the other, still work?

3. Another problem is that current data structures are often too complex for easy parallelization, Audet [2]. Efficient parallelization prefers simple regular data structures, such as structures of arrays of plain old datatypes. If the platform is an Nvidia GPU, then warps of 32 threads are required to execute the same instruction (or be idle). Ideally, the data used by adjacent threads is adjacent in memory. That disparages pointers, linked lists, sweep lines, and trees.

Some components of 3D-EPUG-OVERLAY have been presented earlier, [CITATIONS REMOVED]. This paper extends [CITATIONS REMOVED]. with more details, such as about symbolic perturbation, and newer experiments.

1.1. Background

Kettner et al [23] studied geometric failures caused by roundoff errors, showing situations where epsilon-tweaking failed. (That uses an ε tolerance to consider two values x and y to be equal if $|x-y| \leq \varepsilon$.) Snap rounding arbitrary precision segments

into fixed-precision numbers, Hobby [19], can also generate inconsistencies and deform the original topology. Variations attempting to get around these issues include de Berg et al [6], Hersberger [18], and Belussi et al [3]. Controlled Perturbation, Melhorn [27], slightly perturbs the input to remove degeneracies such that the geometric predicates are correctly evaluated even using floating-point arithmetic. Adaptive Precision Floating-Point, Shewchuk [34], exactly evaluates predicates using the minimum necessary precision.

Frey and George [15] contains a comprehensive description of spatial data structures. Common algorithms for computing intersections include plane sweep and quad or octrees [6, 32]. According to Audet [2], "the plane sweep strategy does not parallelize efficiently, rendering it incapable of benefiting from recent trends of multicore CPUs and general-purpose GPUs".

The formally proper way to effectively eliminate roundoff errors and guarantee robustness is to use exact computation based on rational numbers with arbitrary precision [20, 23, 25]. We present algorithms that are efficient enough to computate using arbitrary precision rationals.

Computing in the algebraic field of the rational numbers over the integers, with the integers allowed to grow as long as necessary, allows the traditional arithmetic operations, $+, -, \times, \div$, to be computed exactly, with no roundoff error. The cost is that the number of digits in the result of an operation is about equal to the sum of the numbers of digits in the two inputs. E.g., $\frac{214}{433} + \frac{659}{781} = \frac{452481}{338173}$. This behavior is acceptable if the depth of the computation tree is small, which is true for our algorithm.

During the evaluation of predicates, arithmetic operations are applied to the intervals. If the result (typically the sign of a determinant) can be inferred based on the the bounds of the interval, its value is returned. Otherwise, the predicate is re-evaluated using exact arithmetic.

1.2. Current freely available implementations:

One technique for overlaying 3-D polyhedra starts by converting the data to a volumetric representation (voxelization), perhaps using an octree, Meagher [26]. While the intersection is trivial and the representation robust, it is usually inexact. For example, oblique surfaces cannot be represented exactly, affecting fluid flow and visualization. Pavić et al. [30] present an efficient algorithm for performing this kind of overlay.

For exactly computing overlays, a common strategy is to use indexing to accelerate operations such as computing the triangle-triangle intersections. For example, Franklin [12] uses a uniform grid to intersect two polyhedra, Feito et al

[11] and Mei et al [28] use octrees, and Yongbin et al [36] use Oriented Bounding Boxes trees (OBBs) to intersect triangulations. Although those algorithms do not use approximations, robustness cannot be guaranteed because of floating point errors. For example, Feito et al [11] use a tolerance to process floating-point numbers, but this is error-prone.

Another algorithm that does not guarantee robustness is QuickCSG, Douze et al [8], which is designed to be extremely efficient. QuickCSG employs parallel programming and a *k-d*-tree index. It does not handle geometric degeneracies (it assumes vertices are in general position), and does not handle the numerical non-robustness from floating-point arithmetic, Zhou et al [37]. To reduce errors caused by special cases, QuickCSG allows the user to apply random numerical perturbations to the input, but this has no guarantees.

Some small errors might be acceptable, but they accumulate when several inexact operations are performed in sequence, which is common in CAD and GIS. For use when exactness is required, Hachenberger et al [17] presented an algorithm for computing the exact intersection of Nef polyhedra. A Nef polyhedron is a finite sequence of complement and intersection operations on half-spaces. Although dating from the 1970s, only in the 2000s were concrete algorithms developed, and then embodied into CGAL [4]. One application is SFCGAL [29], which uses CGAL to allow the PostGIS DBMS to perform exact computation. Although exact, they are slow, Leconte et al [24]. Also, in most cases, the data must first be converted into the Nef format.

Recently, Zhou [37] presented an exact and parallel algorithm for performing booleans on meshes. The key is to use the concept of winding numbers to disambiguate self-intersections on the mesh. Their algorithm first constructs an arrangement with the two (or more) input meshes, and then resolves the self-intersections in the combined mesh by retesselating the triangles such that intersections happen only on common vertices or edges. The self-intersection resolution eliminates not only the triangle-triangle intersections between triangles of the different input meshes, but also between triangles of the same mesh. As a result, their algorithm can also eliminate self-intersections in the input meshes, repairing them. Finally, a classification step is applied to compute the resulting boolean operations.

That algorithm is freely available and distributed in the LibiGL package, Jacobson et al [21]. Its implementation employs CGAL's exact predicates. The triangle-triangle intersection computation is also accelerated using CGAL's bounding-box-based spatial index. LibiGL is not only exact, but also much faster than Nef Polyhedra. However, it is still slower than fast inexact algorithms such as QuickCSG.

2. Our techniques

Our solution to the above problems combines the following five techniques.

2.1. Big rational numbers

The classic technique of representing a number as the quotient of two integers, each represented as an array of groups of digits has implementation challenges. C++ usually constructs new objects on a global heap, assuming that cost to be negligible. That is false for parallel programs (which serialize modifications to the heap) processing large datasets.

Therefore we carefully construct our code to minimize the number of times that a rational variable needs to be constructed or enlarged, including minimizing the number of necessary temporary variables.

Furthermore, we employ arithmetic filters and interval arithmetic [31] to accelerate the exact computation. This uses an interval of two floats containing each exact value. During a predicate evaluation (typically an expression's sign), the operations are initially applied to the intervals. After each operation the result (an interval) is adjusted to guarantee that it will still contain the exact result of the operation. Then if that interval does not contain 0, its sign is returned. Otherwise, the predicate is re-evaluated using exact arithmetic.

2.2. Simulation of Simplicity

Simulation of Simplicity (SoS), Edelsbrunner et al [10], addresses the problem that, "sometimes, even careful attempts at capturing all degenerate cases leave hard-to-detect gaps", Yap [35].

Figure 1 is a challenging example arising in 2D point location. The lower apparent triangle is really a quadrilateral with the top two edges collinear, and joined at c, which is also a vertex of the upper triangle.

Given a set of polygons and a query point q, the polygon containing q can be found by extending a vertical ray r from q, and finding the first edge e intersecting r. Here, edge dc is the first to intersect r, and thus, q is in the polygon on the negative side of dc. However, locating q' is a challenge since the ray r' intersects edges ac, bc, dc and cf at the same point. Edges bc and ac don't even bound the polygon containing q'. RCT gets a $3D$ version of this problem wrong; PINMESH is correct because of SoS, which ensures a vertical ray from a query point will never intersect a vertex or a vertical triangle from the polyhedral mesh, Magalhães et al [7].

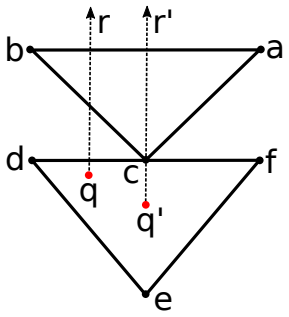


Figure 1: Difficult test case for 2-D point location.

SoS symbolically perturbs coordinates by adding infinitesimals of different orders to destroy any coincidences. E.g., three points are never collinear. A positive infinitesimal, ε, is smaller than any positive real number (but greater than zero). That violates the Archimedean property for the real field, but we don't need this independent axiom. A second order infinitesimal, ε^2, is smaller than any first order infinitesimal, etc. Linear combinations of reals and infinitesimals work. In 1-D, SoS can be realized by indexing all the input variables of both input objects, and then modifying them thus:

$$x_i \rightarrow x_i + \varepsilon^{(2^i)} \tag{1}$$

A coordinate's perturbation depends on its index. The efficient implementation of SoS examines its effects on each predicate and then recodes the predicate to have the same effect, but without using infinitesimals. E.g.,

$$x_i \leq x_j \rightarrow (x_i < x_j) \vee ((x_i = x_i) \wedge (i > j)) \tag{2}$$

To intersect two meshes M_0 and M_1, we perturb M_1 thus:

$$v_j \rightarrow (v_{jx} + \varepsilon, v_{jy} + \varepsilon^2, v_{jz} + \varepsilon^3) \tag{3}$$

Now, no vertex from $M_{0\varepsilon}$ can coincide with any vertex from $M_{1\varepsilon}$. The following lemmas present properties of the perturbed meshes (the proofs have been removed because of space limitations).

Lemma 2.1. *If a_ε is a vertex from mesh $M_{i\varepsilon}$ (i is 0 or 1) and t_ε is a triangle from mesh $M_{1-i\varepsilon}$, then a_ε and t_ε are not coplanar.*

Lemma 2.2. *Given an edge $e_{1\varepsilon} = a_\varepsilon b_\varepsilon$ ($a_\varepsilon \neq b_\varepsilon$), i.e., the endpoints are a_ε and b_ε, from mesh $M_{i\varepsilon}$ and another edge $e_{2\varepsilon} = c_\varepsilon d_\varepsilon$ ($c_\varepsilon \neq d_\varepsilon$) from mesh $M_{1-i\varepsilon}$ such that $e_{1\varepsilon}$ and $e_{2\varepsilon}$ are not parallel, then $e_{1\varepsilon}$ and $e_{2\varepsilon}$ do not intersect.*

Lemma 2.3. *Given two distinct vertices a_ε and b_ε from mesh $M_{i\varepsilon}$ and another vertex c_ε from mesh $M_{1-i\varepsilon}$, then a_ε, b_ε and c_ε are not collinear.*

Corollary 2.4. *An edge e_ε from mesh $M_{i\varepsilon}$ cannot intersect a parallel edge f_ε from $M_{1-i\varepsilon}$.*

Lemma 2.5. *If an edge e_ε from a mesh $M_{i\varepsilon}$ intersects a triangle t_ε from $M_{1-i\varepsilon}$, then this intersection happens in the interior of t_ε.*

Lemma 2.6. *If e_ε is an edge from mesh $M_{i\varepsilon}$ and t_ε is a triangle from mesh $M_{1-i\varepsilon}$, then e_ε and t_ε are not coplanar.*

Lemma 2.7. *If $t_{i\varepsilon}$ and $t_{(1-i)\varepsilon}$ are triangles belonging to, respectively, meshes $M_{i\varepsilon}$ and $M_{1-i\varepsilon}$, then $t_{i\varepsilon}$ and $t_{(1-i)\varepsilon}$ are not co-planar.*

A consequence of the previous lemmas is that there will be no coincidence when triangles from one mesh intersect triangles of the other one. E.g., a triangle from one mesh is never coplanar w.r.t. a triangle of the other one, the intersection of triangles from the two meshes is always either empty or an edge, etc.

We developed two versions of each geometric function, one focused on efficiency, and the second using SoS. We always execute the first version. Only if it detects a coincidence, do we call the second version.

2.3. Minimal topology

The minimal explicit topology required for computing some property of an object depends on the desired property. E.g., testing for point location in a polygon requires only the set of unordered edges. That is still true for multiple and nested components. A sufficient representation of a 3-D mesh comprises the following:

1. the array of vertices, (v_i), where each $v_i = (x_i, y_i, z_i)$.
2. the array of tetrahedra or other polyhedra, t_i, used solely to store properties such as density, and
3. the array of augmented oriented triangular faces (f_i), where $f_i = (v_{i1}, v_{i2}, v_{i3}, t_{i1}, t_{i2})$. The tetrahedron or polyhedron t_{i1} is on the positive side of the face $f_i = (v_{i1}, v_{i2}, v_{i3})$; t_{i2} on the negative.

It is unnecessary to store any further relations, such as from face to adjacent face, from vertex to adjacent face, edge loops, or face shells.

Note that there are no pointers or lists; we need only several structures of arrays. If the tetrahedra have no properties, then the tetrahedron array does not need to exist, so long as the tetrahedra, which we are not storing explicitly, are consistently sequentially numbered. The goal is always to minimize what types of topology need to be stored.

2.4. Uniform grid

The uniform grid, Akman et al [1], Franklin et al [13, 14] is used as an initial cull so that, when two objects are tested for possible intersection, then the probability of intersecting is bounded below by a positive number. Therefore, the number of pairs of objects tested for intersection is linear in the number that intersect. Thus the expected execution time is linear in the output size. The basic algorithm goes as follows.

1. Choose a positive integer g for the grid resolution as a function of the statistics of the input data. Typically, $10 \leq g \leq 1000$. The goal is for each grid cell, as described later, to contain an expected constant number of intersections with input objects.

2. Superimpose a 3-D grid of $g \times g \times g$ cells on the input data.

3. There will be an abstract data structure, the *cell intersection set*, of the set of input objects intersecting each cell.

4. For each input object, determine which cells it intersects, and insert it into each of those cells' intersection sets.

A careful concrete implementation of the *cell intersection set* is critical. We tested several choices; details are in [CITATION REMOVED]. We also tested an octree, but our uniform grid implementation is much faster. We also used a second level grid for some cells. This allowed us to use an approximation to determine which faces intersected each cell: enclosing oblique faces with a box and then marking all the cells intersecting that box, which is more cells than necessary.

2.5. OpenMP

Our simple regular data structures are easily parallelizable with OpenMP.

3. 3-D mesh intersection

3D-EPUG-OVERLAY exactly intersects 3-D meshes. Its input is two triangular meshes M_0 and M_1. Each mesh contains a set of 3-D triangles representing a set of polyhedra. The output is another mesh where each represented polyhedron is the intersection of a polyhedron from M_0 with another one from M_1. The key is the combination of five techniques previously mentioned. Extra details are in [CITATION REMOVED].

3.1. Data representation

The input is a pair of triangular meshes in 3-D (E^3). Both meshes must be watertight and free from self-intersections. The polyhedra may have complex and nonmanifold topologies, with holes and disjoint components.

There are two types of output vertices: input vertices, and intersection vertices resulting from intersections between an edge of one mesh and a triangle of the other. Similarly, there are two types of output triangles: input triangles and triangles from retesselation. The first contains only input vertices while the second may contain vertices generated from intersections created during the retesselation of input triangles.

An intersection vertex v is represented by the pair (e,t), where v is the intersection of the edge e with the triangle t. For speed, its coordinates are cached when first computed. If during the evaluation of a predicate a coincidence is detected, this predicate is re-evaluated using Simulation of Simplicity.

3.2. The mesh intersection algorithm:

This method uses only local information. The algorithm has 3 basic steps and a uniform grid is employed to accelerate the computation:

1. Intersections between triangles of one mesh and triangles of the other mesh are detected and the new edges generated by the intersection of each pair of triangles are computed.

2. A retesselated mesh containing the triangles from the two original meshes is created and the original triangles are split (retesselated) at the intersection edges. I.e., if a pair of triangles in this resulting mesh intersect, then this intersection will happen necessarily on a common edge or vertex.

3. A classification is performed: triangles that shouldn't be in the output are removed and the adjacency information stored in each triangle is updated to ensure that the new mesh will consistently represent the intersection of the input.

Figure 2 illustrates the intersection computation. In Figure 2(a), we have two meshes representing one tetrahedron each: the brown mesh (mesh M_0) bounds the exterior region and region 1 while the yellow mesh (mesh M_1) bounds the exterior region and region 2.

After the intersections between the triangles are computed, the ones from one mesh that intersect triangles from the other one are split into several triangles, creating meshes M_0' and M_1' (for clarity, these two meshes are displayed separately in Figures 2(b) and (c), respectively). The only triangle from mesh M_0 that intersects mesh M_1 is *BCD*. Since *BCD* intersects three triangles from M_1, it was split into seven triangles when M_0' was created (triangles *LMN*, *CLN*, *CBN*, *BDN*, *DMN*, *DLM* and *CDL*). Similarly, each of the three triangles from M_1 intersecting M_0 was split into three smaller triangles. Figure 2(d) illustrates the classification step, where triangles contained in a region of the other mesh are selected to be in the resulting mesh.

Some details about each step are presented in the next subsections.

3.2.1. Detecting the intersections

A two-level 3-D uniform grid is used to accelerate the detection of triangle intersections and point location. The intersections between a pair of triangles is detected using Segura et al.'s algorithm [33], which computes five 3D orientations in order to verify if there is an intersection between each edge of one of the triangles and the other triangle.

3.2.2. Retesselating the triangles

The next step is to split them where they intersect (creating the retesselated meshes M_0' and M_1'), so that after this process all the intersections will happen only on common vertices or edges. This process is performed by creating a graph G in each triangle t (where the edges are the original edges of t and the ones resulting from the intersections between t and other triangles) and using the algorithm presented in [22] to extract the faces from G. These faces are then triangulated using the ear-clipping algorithm [9], which has a time complexity quadratic in the number of vertices in the faces. That time is acceptable because the expected size of a face is small. If it were a problem, more efficient polygon

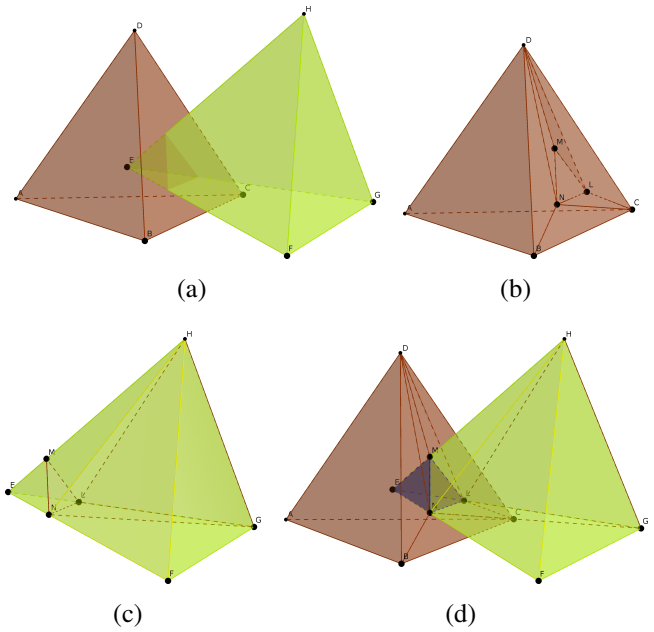


Figure 2: Computing the intersection of two tetrahedra - (a): input meshes, (b) and (c): retesselated meshes, (d): classifying the triangles to generate the output.

triangulation algorithms exist, using as little as linear time in the face size, at the cost of considerable complexity.

Since the retesselation is a 2D process, the triangle t being processed is projected onto one of the planes ($x = 0$, $y = 0$ or $z = 0$) with which t is non-perpendicular and then processed. The predicates employed by the retesselation (for example, to check if a vertex is convex, to sort the vertices along an edge, to check if a triangle contains a point, etc) can all be implemented using orientation predicates.

3.2.3. Triangle classification

As illustrated in Figure 2, after the intersections are detected and all the triangles that intersect other triangles are split at the intersection points, two new meshes M'_0 and M'_1 are created such that each new mesh M'_i will have the following two kinds of triangles:

- Triangles from the original mesh: if triangle t from M_i did not intersect any triangle from the other mesh (or if this intersection was located on a vertex or edge), then t will be in M'_i.

- New triangles: if triangle t from M_i intersects one or more triangles from the other mesh (and this intersection is not located on a common vertex or edge), then t will be partitioned into smaller triangles that will be inserted into M'_i.

It is clear that each mesh M'_i will exactly represent the same regions that M_i represents. Thus, computing the intersection between M'_i and $M'_{(1-i)}$ is equivalent to computing the intersection of M_i with $M_{(1-i)}$.

The process of classifying the triangles to create the output mesh consists in processing each triangle t from the meshes M'_i $(i = 0, 1)$, determining in what region of $M'_{(1-i)}$ t is (each triangle will be entirely contained in a region since the meshes have been retesselated at the intersection edges) and, then, updating the information about the regions t bounds such that we will have a consistent mesh.

If a triangle t is in the exterior of the other mesh, in the resulting mesh the two regions t bounds will be the exterior region. To maintain the mesh consistency, the triangles bounding only the exterior region can be ignored and not stored in the output mesh.

Figure 2(d) illustrates the classification step. All the intersections happen at common edges, and the only triangle from M'_0 that is completely inside region 2 (of M'_1) is triangle LMN. Since LMN bounds region 1 and the exterior region in M'_0, in the resulting intersection LMN will bound region $1 \cap 2$ and the exterior region. All the other triangles from M'_0 are in the exterior region of M'_1 and, thus, they will only bound the exterior region in the resulting intersection (therefore, they will be ignored when the output mesh is computed). Similarly, in M'_0 the only triangles that are inside region 1 of M'_0 are triangles EMN, ELM and ELN. These three triangles will also bound the exterior region and region $1 \cap 2$ in the resulting mesh.

The process of locating triangles of one mesh in the other one can be performed using a point location and a flood-fill algorithm. Suppose triangles of mesh M'_i are being located. If two adjacent triangles t and t' share an edge that was not generated by an intersection with $M'_{(1-i)}$, then these triangles are in the same region of $M'_{(1-i)}$. If t and t' share an edge that was generated by an intersection with triangle t'' of mesh $M'_{(1-i)}$, then t and t' are in different regions of the other mesh. Since the regions t'' bound are known, it is possible to determine the location of t and t' once the location of at least one of these two triangles is known.

Thus, the location of all triangles in each connected component of triangles can be performed by locating one of the triangles as a seed and using a traversal algorithm to locate the other ones. We use as seed a triangle containing an input vertex. Since the location of an input vertex is the same of the triangles containing it, the seed is located by locating one of its input vertices.

The point location process is performed with PinMesh [7], which uses a uniform grid to locate points in expected constant time. Pinmesh also uses SoS.

3.3. Experiments:

3D-EPUG-OVERLAY was implemented in C++ and compiled using g++ 5.4.1. For better parallel scalability, the Tcmalloc memory allocator, was employed. Parallel programming was provided by OpenMP 4.0, multiple precision rational numbers were provided by GNU GMPXX and arithmetic filters were implemented using the Interval_nt number type provided by CGAL for interval arithmetic. The experiments were performed on a workstation with 128 GiB of RAM and dual Intel Xeon E5-2687 processors, each with 8 cores and 16 hyper-threads, running Ubuntu Linux 16.04. Unless otherwise noted, experiments were performed using 32 threads.

We evaluated 3D-EPUG-OVERLAY, by comparing it against three state-of-the-art algorithms:

1. *LibiGL* [37], which is exact and parallel,
2. *Nef Polyhedra* [4], which is exact, and
3. *QuickCSG* [8], which is fast and parallel, but not exact, and does not handle special cases.

Our experiments showed that 3D-EPUG-OVERLAY is fast, parallel, exact, economical of memory, and handles special cases.

3.3.1. Datasets:

Experiments were performed with a variety of non self-intersecting and watertight meshes; see Figure 3 and Table 1. The ones with suffix *tetra* were tetrahedralized with GMSH [16]. The sources of the data are as follows: Barki (Clutch2kf, Casting10kf, Horse40kf, Dinausor40kf, Armadillo52kf, Camel69kf, Cow76kf), AIM@SHAPE (Camel, Bimba, Kitten, RedCircBox, Ramesses, Vase, Neptune), Stanford (Armadillo), Thingi10K (461112, 461115, 226633), Thingi10k+GMSH (914686Tetra, 68380Tetra, 518092Tetra, 461112Tetra), and Stanford+GMSH (Armad.Tetra).

Table 2 presents the pairs of meshes used in the intersection experiments, the number of input triangles, the number of triangles in the resulting meshes and the uniform grid size.

Figure 4 shows one test, which took 0.2 seconds.

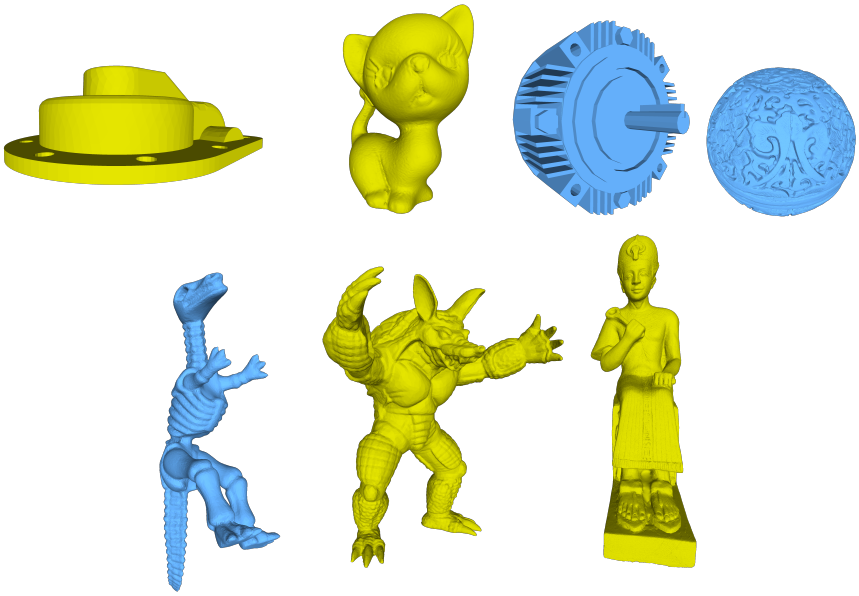
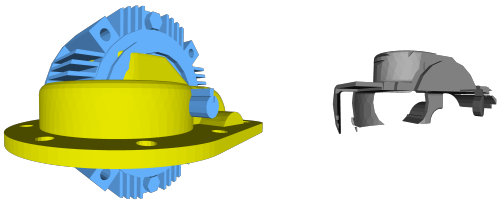


Figure 3: Some test meshes.



(a) Visually overlaid (b) Intersection

Figure 4: Intersecting Casting10kf with Clutch2kf

3.3.2. Arithmetic filters and other optimizations:

To evaluate the effect of different optimizations, we profiled two key steps: the creation of the uniform grid, and the detection of intersections between pairs of triangles. Two important optimizations concern the uniform grid construction and the use of interval arithmetic. We tested the following algorithm versions using a uniform grid with first level resolution 64^3 and second level resolution 16^3. The datasets were Neptune and Neptune-translated.

Vector: Each level of the uniform grid is created by performing a single-pass through the triangles, using dynamic STL vectors to store the triangles in the grid cells.

Ragged: This reduces the number of memory allocations. Each level of the uniform grid is created with two passes through the data: first to count the amount of triangles that will go into each cell in order to allocate a ragged array of just the needed size, followed by a second pass to insert the triangles.

NoAlloc: This is the ragged version, plus: Temporary memory allocation is avoided by reusing rational numbers and by rewriting the arithmetic expressions in order to avoid creating temporary rationals.

Interval: This is the NoAlloc version, plus: Interval arithmetic is employed to minimize operations with rationals.

We also compared the default *glibc* memory allocator to the GPerftools Tcmalloc allocator (abbreviated to *Tcm.* in the table). Table 3 presents the measured times (in seconds).

Creating the 3D uniform grid has several steps. (i) The grid cell containing each vertex is computed (row *Comp. grid cell*), which is computationally intensive step when using rationals. (ii) A first pass (row *First pass*) through the triangles is performed. If the uniform grid uses STL vectors to store the triangles, then that happens now. Otherwise, if using a ragged array, the number of triangles in each cell is counted, and the triangles are inserted during the second pass (row *Second pass*). (iii) A second level grid is created (row *Refinement*). These three last steps are more memory intensive.

Intersection detection goes as follows. (i) The uniform grid is traversed, and a list of pairs of triangles to be tested for intersection is created (row *List pairs of tri.*). (ii) Each pair of triangles is tested for intersection (row *Detect inters.*). Table 3 also presents the total times spent creating the grid (row *total time grid*) and detecting intersections (row *total time inter.*).

The ragged array version constructs the uniform grid 32% faster than the vector version because it is more parallelizable, has better locality of reference, and has fewer memory allocations.

Besides employing better memory allocators *(Tcmalloc)*, minimizing memory allocations can also improve savetime. For example, in the *NoAlloc* version the two computationally intensive steps mentioned above perform better than in the *Ragged* version.

Since computing the grid cell containing each vertex and detecting intersections are the only steps directly dealing with rationals, they were the only steps that benefited from the use of interval arithmetic. The best results were observed in the intersection detection, which improved 357 times when interval arithmetic was employed.

While the first versions of the algorithm lead to only marginal improvements in the total running time of the algorithm (since operations with rationals dominate the time), these optimizations are important because the last version (with interval arithmetic) significantly reduces this domination. Indeed, while in the first version of the algorithm 98% of the time was spent detecting intersections between triangles, this step used 31% of the time in the *Interval* version.

3.3.3. The importance of the uniform grid:

This accelerates the detection of pairs of intersecting triangles. To evaluate this idea, we compared it against an implementation using the CGAL method for intersecting d-D Iso-oriented Boxes. Both algorithms are exact and employ arithmetic filters with interval arithmetic. Indeed, this CGAL algorithm is employed by LibiGL's mesh intersection.

The CGAL method is sequential, and employs a hybrid approach composed of a sweep-line and a streaming algorithm to detect intersections between pairs of Axis Aligned Bounding Boxes. Pairwise intersections of triangles can be detected by filtering the pairs of intersecting bounding-boxes, and then testing the triangles for intersection. Since the CGAL exact kernel was not thread-safe, even the triangle-triangle intersection tests were performed sequentially.

Table 4 presents these comparative experiments, performed on 6 pairs of meshes. The number of intersections detected is not necessarily the same in the two algorithms because our algorithm implements Simulation of Simplicity. E.g., co-planar triangles never intersect.

CGAL is better at culling pairs of non-intersecting bounding-boxes and so performs fewer intersection tests. However, since the uniform grid is lightweight and parallelizes well, its pre-processing time is much smaller (up to 134 times

faster, which is much more than the degree of parallelism), and this difference is never recaptured.

The only situation where the intersection detection time was much larger than the pre-processing time was in the intersection of the Armadillo mesh with itself. In the uniform grid implementation this test case leads to many coincidences, triggering the SoS predicates, which have not been optimized yet. In this situation the uniform grid was still faster than CGAL for two reasons: first, the number of intersection tests performed by the two methods was similar. Second, the uniform grid detects intersections in parallel.

3.3.4. Comparing 3D-EPUG-OVERLAY *to other methods:*

We compared 3D-EPUG-OVERLAY against other three algorithms using the pairs of meshes presented in Table 2. The resulting running times (in seconds, excluding I/O) are presented in Table 5. Since the CGAL exact intersection algorithm deals with Nef Polyhedra, we also included the time it spent converting the triangulating meshes to this representation and to convert the result back to a triangular mesh (it often takes more time to convert the dataset than to compute the intersection). Both times are reported to let the user choose.

3D-EPUG-OVERLAY was up to 101 times faster than LibiGL. The only test cases where the times spent by LibiGL were similar to the times spent by 3D-EPUG-OVERLAY were during the computation of the intersections of a mesh with itself (even in these test cases 3D-EPUG-OVERLAY was still faster than LibiGL). In this situation, the intersecting triangles from the two meshes are never in general position, and thus the computation has to frequently trigger the SoS version of the predicates, which we haven't not optimized yet. In the future, we intend to optimize this.

However, LibiGL also repairs meshes (by resolving self-intersections) during the intersection computation, which 3D-EPUG-OVERLAY does not attempt.

Because of the overhead of Nef Polyhedra and since it is a sequential algorithm, CGAL was always the slowest. When computing the intersections, 3D-EPUG-OVERLAY was up to $1,284$ times faster than CGAL. The difference is much higher if the time CGAL spends converting the triangular mesh to Nef Polyhedra is taken into consideration: intersecting meshes with 3D-EPUG-OVERLAY was up to $4,241$ times faster than using CGAL to convert and intersect the meshes.

While 3D-EPUG-OVERLAY was faster than QuickCSG in most of the test cases (mainly the largest ones), in others QuickCSG was up to 20% faster than 3D-EPUG-OVERLAY. The relatively small performance difference between 3D-EPUG-OVERLAY and an inexact method (that was specifically designed to be very

fast) indicates that 3D-EPUG-OVERLAY presents good performance allied with exact results. The * in some of the test cases in Table 5 indicates that QuickCSG reported errors during the computation.

Finally, we also performed experiments with tetra-meshes. Each tetrahedron in these meshes is considered to be a different object and, thus, the output of 3D-EPUG-OVERLAY is a mesh where each object represents the intersection of two tetrahedra (from the two input meshes). These meshes are particularly hard to process because of their internal structure, which generates many triangle-triangle intersections.

To the best of our knowledge, LibiGL, CGAL and QuickCSG were not designed to handle meshes with multi-material and, thus, we couldn't compare the running time of 3D-EPUG-OVERLAY against them in these test cases.

We also evaluated the peak memory usage of each algorithm. 3D-EPUG-OVERLAY was: almost always smaller than LibiGL, with the difference increasing as the datasets became larger; smaller than QuickCSG in every case where QuickCSG returned the correct answer; and much smaller than CGAL. A typical result was the intersection of Neptune (4M triangles) with Ramesses (1.7M triangles): 3D-EPUG-OVERLAY used 2.6GB, LibiGL used 6.7GB, and CGAL 84GB. The largest example that 3D-EPUG-OVERLAY processed, 518092Tetra (6M triangles) with 461112Tetra (8.5M triangles) used 43GB. [CITATION REMOVED] contains detailed results.

3.3.5. Correctness evaluation:

3D-EPUG-OVERLAY was developed on a solid foundation (i.e., all computation is exact and special cases are properly handled using Simulation of Simplicity) in order to ensure correctness. However, perhaps its implementation has errors? Therefore, we performed extensive experiments comparing it against LibiGL as a reference solution. We employed the Metro tool, Cignoni et al [5], to compute the Hausdorff distances between the meshes being compared. Metro is widely employed, for example, to evaluate mesh simplification algorithms by comparing their results with the original meshes.

Since Metro is not exact (all the computation is performed using double precision floats), we use the distance between meshes only as evidence that our implementation is correct. In every test, the difference between 3D-EPUG-OVERLAY and LibiGL was reported as 0. In some situations the difference between LibiGL and CGAL was a small number (maximum 0.0007% of the diagonal of the bounding-box). We guess this is because the exact results are stored using floating-point variables, and different strategies are used to round the vertices to

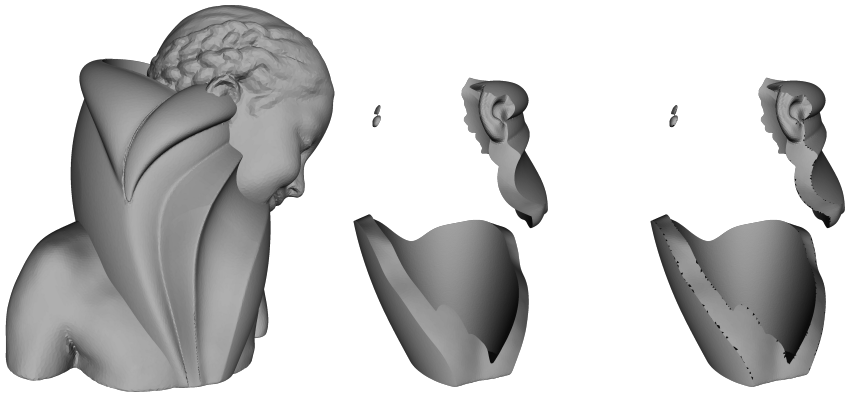


Figure 5: Intersection of the Bimba and Vase meshes computed by 3D-EPUG-OVERLAY and QuickCSG, showing only 3D-EPUG-OVERLAY computing a region correctly.

floats and write them to the text file.

QuickCSG, on the other hand, generated errors much larger than CGAL: in the worst case, the difference between QuickCSG output and LibiGL was 0.13% of the diagonal of the bounding-box). [CITATION REMOVED]

3.3.6. Visual inspection:

We also visually inspected the results using MeshLab. Even though small changes in the coordinates of the vertices cannot be easily identified by visual inspection (and even the program employed for displaying the meshes may have roundoff errors), topological errors (such as triangles with reversed orientation, self-intersections, etc) often stand out.

Even when QuickCSG did not report a failure, results were frequently inconsistent, with open meshes, spurious triangles or inconsistent orientations.

Figure 5 shows the intersection of Bimba and the Vase. The first part is the complete overlay mesh, as computed by 3D-EPUG-OVERLAY. The second is a detail of an error-prone output region, computed correctly by 3D-EPUG-OVERLAY. The third part shows the same region computed by QuickCSG. Note the errors along the edges.

3.3.7. Rotation invariance:

We also validated 3D-EPUG-OVERLAY by verifying that its result does not change when the input meshes are rotated. I.e., a pair of meshes were rotated around the same point, intersected, and the resulting mesh was rotated back. To

ensure exactness, we chose a rotation angle with rational sine and cosine. We evaluated all the pairs in Table 2. For each pair, we performed a rotation around the x axis and, then, a rotation around the y axis (the origin was defined as the center of the joint bounding-box of the two meshes). We chose rotation angle θ such that $\sin\theta = 400/10004$ and $\cos\theta = 9996/10004$. $\theta \approx 2.29$ degrees.

In all the experiments Metro reported that the resulting meshes had distance 0.000000 w.r.t. the corresponding ones obtained without rotation.

In addition, we intersected each mesh from Table 2 with a rotated version of itself. This is a notoriously difficult case for CAD systems because the large number of intersections and small triangles. Each mesh M was rotated around the center of its bounding-box using the above θ, and intersected with its original version, using both LibiGL and 3D-EPUG-OVERLAY. In every experiment the Hausdorff distance between the two outputs was 0.000000. That is, we can quickly process cases that can crash CAD systems.

3.3.8. Limitations:

Even though the computations are performed exactly, common file formats for 3D objects such as OFF represent data using floating-point numbers. Converting 3D-EPUG-OVERLAY's rational output into floats may introduce errors since most rationals cannot be represented exactly. Possible solutions include avoiding the conversion (i.e., always employing multiple-precision rationals in the representations), or using heuristics such [37] to try to choose floats for each coordinate so that the approximate output will not only be similar to the exact one, but also it will not present topological errors.

A limitation of symbolic perturbation is that the results are consistent considering the perturbed dataset, not necessarily considering the original one [10]. Thus, if the perturbation in the mesh resulting from the intersection is ignored, the unperturbed mesh may contain degeneracies such as triangles with area 0 or polyhedra with volume 0 (these polyhedra would have infinitesimal volume if the perturbation was not ignored). More details are in [CITATION REMOVED].

3.4. Summary:

3D-EPUG-OVERLAY is an algorithm and implementation to intersect a pair of 3D triangular meshes. It is simultaneously the fastest, free from roundoff errors, handles geometric degeneracies, parallelizes well, and is economical of memory. The source code, albeit research quality, is freely available for nonprofit research and education at [CITATION REMOVED]. We have extensively tested it for errors; we encourage others to test it. It is a suitable subroutine for larger

systems such as 3D GIS or CAD systems. Computing other kinds of overlays, such as union, difference, and exclusive-or, would require modifying only the classification step. We expect that 3D-EPUG-OVERLAY could easily process datasets that are orders of magnitude larger, with hundreds of millions of triangles. Finally, 3D-EPUG-OVERLAY has not nearly been fully optimized, and could be made much faster. Indeed, we are currently adapting it to employ an NVIDIA GPU to further accelerate the computation.

References

[1] Varol Akman, Wm Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1989. Geometric Computing and the Uniform Grid Data Technique. *Computer Aided Design* 21, 7 (1989), 410–420.

[2] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and Efficient Polygon Overlay on Parallel Stream Processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13)*. ACM, New York, NY, USA, 304–313. https://doi.org/10.1145/2525314.2525352

[3] Alberto Belussi, Sara Migliorini, Mauro Negri, and Giuseppe Pelagatti. 2016. Snap Rounding with Restore: An Algorithm for Producing Robust Geometric Datasets. *ACM Trans. Spatial Algorithms and Syst.* 2, 1, Article 1 (March 2016), 36 pages. https://doi.org/10.1145/2811256

[4] CGAL. 2018. Computational Geometry Algorithms Library. (2018). Retrieved 2018-09-09 from https://www.cgal.org

[5] P. Cignoni, C. Rocchini, and R. Scopigno. 1998. Metro: Measuring Error on Simplified Surfaces. *Comput. Graph. Forum* 17, 2 (June 1998), 167–174. https://doi.org/10.1111/1467-8659.00236

[6] Mark de Berg, Dan Halperin, and Mark Overmars. 2007. An intersection-sensitive algorithm for snap rounding. *Computational Geometry* 36, 3 (Apr. 2007), 159–165.

[7] Salles V. G. de Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, and Wenli Li. 2016. PinMesh – Fast and exact 3D point location queries using

a uniform grid. *Computer & Graphics Journal, special issue on Shape Modeling International 2016* 58 (Aug. 2016), 1–11. https://doi.org/10.1016/j.cag.2016.05.017 (online 17 May). Awarded a reproducibility stamp, http://www.reproducibilitystamp.com/.

[8] Matthijs Douze, Jean-Sébastien Franco, and Bruno Raffin. 2015. *QuickCSG: Arbitrary and faster boolean combinations of n solids*. Ph.D. Dissertation. Inria-Research Centre, Grenoble–Rhône-Alpes, France.

[9] David Eberly. 2002. Triangulation by ear clipping. *Geom. Tools* (2002). https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf, (Retrieved on 11/29/2017).

[10] Herbert Edelsbrunner and Ernst Peter Mücke. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM TOG* 9, 1 (1990), 66–104.

[11] F.R. Feito, C.J. Ogayar, R.J. Segura, and M.L. Rivero. 2013. Fast and accurate evaluation of regularized Boolean operations on triangulated solids. *Computer-Aided Design* 45, 3 (2013), 705 – 716. https://doi.org/10.1016/j.cad.2012.11.004

[12] Wm Randolph Franklin. 1982. Efficient polyhedron intersection and union. In *Proc. Graphics Interface*. Toronto, 73–80.

[13] Wm Randolph Franklin. 1984. Adaptive Grids for geometric operations. *Cartographica* 21, 2–3 (Summer – Autumn 1984), 161–167. monograph 32–33.

[14] Wm Randolph Franklin, Narayanaswami Chandrasekhar., Mohan Kankanhalli, Manoj Seshan, and Varol Akman. 1988. Efficiency of uniform grids for intersection detection on serial and parallel machines. In *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, Nadia Magnenat-Thalmann and D. Thalmann (Eds.). Springer-Verlag, 288–297.

[15] Pascal Jean Frey and PaulLouis George. 2010. *Mesh Generation: Application to Finite Elements, Second Edition*. ISTE Ltd. and Wiley. https://doi.org/10.1002/9780470611166

[16] Christophe Geuzaine and Jean-François Remacle. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *Int.*

J. for Numerical Methods in Eng. 79, 11 (May 2009), 1309–1331. https://doi.org/10.1002/nme.2579

[17] Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn. 2007. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. *Comupt. Geom.* 38, 1 (Sept. 2007), 64–99.

[18] John Hershberger. 2013. Stable snap rounding. *Comput. Geom.* 46, 4 (May 2013), 403–416.

[19] John D. Hobby. 1999. Practical segment intersection with finite precision output. *Comput. Geom.* 13, 4 (1999), 199–214. http://dblp.uni-trier.de/db/journals/comgeo/comgeo13.html\#Hobby99

[20] Christoff M. Hoffman. 1989. The Problems of Accuracy and Robustness in Geometric Computation. *Computer* 22, 3 (1989), 31–40. https://doi.org/10.1109/2.16223

[21] Alec Jacobson, Daniele Panozzo, et al. 2016. libigl: A Simple C++ Geometry Processing Library. (2016). Retrieved 2017-10-18 from http://libigl.github.io/libigl/

[22] X.Y. Jiang and H. Bunke. 1993. An optimal algorithm for extracting the regions of a plane graph. *Pattern Recognit. Lett.* 14, 7 (July 1993), 553 – 558.

[23] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. 2008. Classroom Examples of Robustness Problems in Geometric Computations. *Comput. Geom. Theory Appl.* 40, 1 (May 2008), 61–78. https://doi.org/10.1016/j.comgeo.2007.06.003

[24] Cyril Leconte, Hichem Barki, and Florent Dupont. 2010. *Exact and Efficient Booleans for Polyhedra*. Technical Report RR-LIRIS-2010-018. LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon. http://liris.cnrs.fr/publis/?id=4883 (Retrieved on 19 Oct 2017).

[25] Chen Li, Sylvain Pion, and Chee-Keng Yap. 2005. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming* (2005), 85–111.

[26] Donald J. Meagher. 1982. Geometric Modelling Using Octree Encoding. *Computer Graphics and Image Processing* 19 (June 1982), 129–147.

[27] Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff. 2006. Reliable and Efficient Computational Geometry Via Controlled Perturbation.. In *ICALP (1)* (2006-07-03) *(Lecture Notes in Computer Science)*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.), Vol. 4051. Springer, 299–310. http://dblp.uni-trier.de/db/conf/icalp/icalp2006-1.html\#MehlhornOS06

[28] Gang Mei and John C. Tipper. 2013. Simple and Robust Boolean Operations for Triangulated Surfaces. *CoRR* abs/1308.4434 (2013). http://arxiv.org/abs/1308.4434

[29] Oslandia and IGN. 2017. SFCGAL. (2017). Retrieved 2017-10-19 from http://www.sfcgal.org/

[30] Darko Pavić, Marcel Campen, and Leif Kobbelt. 2010. Hybrid Booleans. *Comput. Graph. Forum* 29, 1 (Jan. 2010), 75–87. https://doi.org/10.1111/j.1467-8659.2009.01545.x

[31] Sylvain Pion and Andreas Fabri. 2011. A generic lazy evaluation scheme for exact geometric computations. *Sci. Comput. Program.* 76, 4 (Apr. 2011), 307 – 323.

[32] Robert Sedgewick. 1988. *Algorithms, 2nd Edition*. Addison-Wesley.

[33] Rafael Jesús Segura and Francisco R. Feito. 2001. Algorithms to Test Ray-Triangle Intersection. Comparative Study. In *The 9-th Int. Conf. Central Europe Comput. Graph., Visualization Comput. Vision'2001, WSCG 2001*. 76–81.

[34] Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discret. & Comput. Geom.* 18, 3 (Oct. 1997), 305–363.

[35] Chee Keng Yap. 1988. Symbolic treatment of geometric degeneracies. In *System Modelling and Optimization: Proc. 13th IFIP Conference*, Masao Iri and Keiji Yajima (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–358. https://doi.org/10.1007/BFb0042803

[36] Jing Yongbin, Wang Liguan, Bi Lin, and Chen Jianhong. 2009. Boolean Operations on Polygonal Meshes Using OBB Trees. In *ESIAT 2009*, Vol. 1. IEEE, 619–622.

[37] 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.

Table 1: Test datasets.

Mesh	Vertices ($\times 10^3$)	Triangles ($\times 10^3$)	Polyhedra ($\times 10^3$)
Clutch2kf	1	2	-
Casting10kf	5	10	-
Horse40kf	20	40	-
Dinausor40kf	20	40	-
Armadillo52kf	26	52	-
Camel	35	69	-
Camel69kf	35	69	-
Cow76kf	38	76	-
Bimba	75	150	-
Kitten	137	274	-
Armadillo	173	346	-
461112	403	805	-
461115	411	822	-
RedCircBox	701	1403	-
Ramesses	826	1653	-
Ramesses Rot.	826	1653	-
Ramesses Tran.	826	1653	-
Vase	896	1793	-
226633	1226	2452	-
Neptune	2004	4008	-
Neptune Tran.	2004	4008	-
914686Tetra	66	605	281
68380Tetra	107	1067	506
ArmadilloTetra	340	3377	1602
ArmadilloTetra Tran.	340	3377	1602
518092Tetra	603	5938	2814
461112Tetra	842	8495	4046

[*] meshes with the suffix Tetra have been tetrahedralized;
[*] Rot. and Tran. mean, respectively, that the mesh has been rotated or translated; [a] Red Circular Box; [b] tetrahedralized version of the Armadillo mesh.

Table 2: Pairs of meshes intersected.

		# triangles ($\times 10^3$)			Grid size	
M_0	M_1	M_0	M_1	Out	G_1	G_2[a]
Casting10kf	Clutch2kf	10	2	6	64	2
Armadillo52kf	Dinausor40kf	52	40	25	64	4
Horse40kf	Cow76kf	40	76	24	64	4
Camel69kf	Armadillo52kf	69	52	16	64	4
Camel	Camel	69	69	81	64	4
Camel	Armadillo	69	331	43	64	4
Armadillo	Armadillo	331	331	441	64	8
461112	461115	805	822	808	64	8
Kitten	RedCircBox	274	1402	246	64	8
Bimba	Vase	150	1792	724	64	8
226633	461112	2452	805	1437	64	8
Ramesses	RamessesTrans	1653	1653	1571	64	16
Ramesses	RamessesRotated	1653	1653	1691	64	16
Neptune	Ramesses	4008	1653	1112	64	16
Neptune	NeptuneTrans	4008	4008	3303	64	16
68380Tetra	914686Tetra	1067	605	9393	64	2
ArmadilloTetra	ArmadilloTetraTran.	3377	3377	61325	64	4
518092Tetra	461112Tetra	5938	8495	23181	64	4

[a] resolution of the first level grid, second level grid.

Table 3: Times, in seconds, spent by key steps of 6 versions of 3D-EPUG-OVERLAY .

Version:	Vector	Ragged	Ragged	NoAlloc	NoAlloc	Interval
Allocator:	Tcm. [a]	Tcm. [a]	Glibc	Tcm. [a]	Glibc	Tcm. [a]
Step						
Comp. grid cell [b]	0.58	0.58	0.77	0.44	0.42	0.10
First pass [c]	0.39	0.07	0.06	0.06	0.07	0.07
Second pass [d]		0.04	0.04	0.04	0.05	0.04
Refinement [e]	0.13	0.06	0.36	0.06	0.36	0.06
List pairs of tri. [f]	0.17	0.16	0.22	0.15	0.21	0.17
Detect inters. [g]	81.77	83.48	126.54	66.77	66.71	0.19
Total time grid [h]	1.10	0.75	1.24	0.60	0.89	0.27
Total time inter. [i]	81.94	83.64	126.76	66.92	66.93	0.35

[a] *Tcmalloc* memory allocator; [b] time spent determining in which grid cell each input vertex is; [c] time spent performing the first pass throught the triangles in order to count the number of triangles in each cell; [d] time spent actually inserting the triangles into the cells; [e] time spent creating the second level grid; [f] time creating the list of pairs of triangles that may intersect; [g] time testing triangles for intersection; [h] total time spent creating the uniform grid; [i] total time detecting the intersections once the grid has been created.

Table 4: Comparing the times (in seconds) for detecting pairwise intersections of triangles using CGAL (sequential) versus using a uniform grid (parallel).

CGAL							
		# faces ($\times 10^3$)		# int.[a]	Int.tests[b]	Time (s)	
M_0	M_1	M_0	M_1	$\times 10^3$	$\times 10^3$	Pre.proc.[c]	Inter.[d]
Camel	Armadillo	69	331	3	14	0.32	0.01
Armadillo	Armadillo	331	331	4611	5043	1.27	259.23
Kitten	RedC.Box[e]	274	1402	3	13	2.33	0.01
226633	461112	2452	805	23	128	7.18	0.08
Ramesses	Ram.Tran.[f]	1653	1653	36	237	12.38	0.17
Neptune	Nept.Tran.[g]	4008	4008	78	647	36.24	0.47
Uniform grid							
		# faces ($\times 10^3$)		# int.[a]	Int.tests[b]	Time (s)	
M_0	M_1	M_0	M_1	$\times 10^3$	$\times 10^3$	Pre.proc.[c]	Inter.[d]
Camel	Armadillo	69	331	3	33	0.06	0.02
Armadillo	Armadillo	331	331	50	5351	0.25	63.80
Kitten	RedC.Box[e]	274	1402	3	27	0.08	0.02
226633	461112	2452	805	23	307	0.16	0.05
Ramesses	Ram.Tran.[f]	1653	1653	36	866	0.16	0.10
Neptune	Nept.Tran.[g]	4008	4008	78	5087	0.27	0.35

[a] number of intersections detected; [b] number of intersection tests performed;
[c] pre-processing time; [d] time spent testing pairs of triangles for intersection;
[e] Red Circular Box; [f] Ramesses Translated; [g] Neptune Translated.

Table 5: Times, in seconds, spent by different methods for intersecting pairs of meshes. QuickCSG reported errors during the intersections whose times are flagged with *. The tetrahedral mesh tests (last three rows) used only 3D-EPUG-OVERLAY.

		Time (s)				
		3D-	Libi-	CGAL		Quick-
M_0	M_1	Epug	GL	Convert[a]	Intersect[b]	CSG
Casting10kf	Clutch2kf	0.2	1.3	4.2	1.1	**0.1***
Armadillo52kf	Dinausor40kf	0.1	3.0	38.0	21.5	0.1
Horse40kf	Cow76kf	0.1	3.2	51.1	24.2	0.1
Camel69kf	Armadillo52kf	0.1	3.2	54.3	25.7	0.1
Camel	Camel	13.9	18.0	62.7	230.6	**0.9***
Camel	Armadillo	0.2	11.7	189.9	80.0	0.3
Armadillo	Armadillo	67.0	88.1	339.7	1,198.2	**4.1***
461112	461115	0.8	58.9	753.2	473.2	1.1
Kitten	RedCircBox	0.3	28.6	819.8	329.6	1.1
Bimba	Vase	0.6	58.0	971.7	455.7	1.1
226633	461112	0.9	96.0	1,723.7	905.5	**2.2***
Ramesses	Ram.Tran.[c]	1.3	93.0	1,558.8	946.1	**2.4***
Ramesses	Ram.Rot.[d]	2.1	122.0	1,577.3	989.8	2.4
Neptune	Ramesses	1.2	118.1	3,535.5	1,535.6	4.1
Neptune	Nept.Tran.[e]	2.7	220.2	5,390.7	2,726.2	6.1
68380Tet.[f]	914686Tet.[g]	51.3	-	-	-	-
Armad.Tet.[h]	Arm.Tet.Tran.[i]	263.3	-	-	-	-
518092Tetra	461112Tetra	136.6	-	-	-	-

[a] time converting the meshes to CGAL Nef Polyhedra; [b] time intersecting the Nef Polyhedra; [c] Ramesses Translated; [d] Ramesses Rotated; [e] Neptune Translated; [f] 68380Tetra; [g] 914686Tetra; [h] ArmadilloTetra; [i] ArmadilloTetra Translated.