Exact intersection of 3D geometric models

Salles V. G. de Magalhães [1,2], Marcus V. A. Andrade [1], W. Randolph Franklin[2], Wenli Li[2], Maurício G. Gruppi [1]

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV)
Viçosa – MG – Brazil

[2]Rensselaer Polytechnic Institute (RPI), Troy – NY – USA

{salles,marcus,mauricio.gruppi}@ufv.br, mail@wrfranklin.org, liw9@rpi.edu

Abstract. *We present* 3D-EPUG-OVERLAY*, an exact algorithm for computing the intersection of two 3D triangulated meshes. This is useful in GIS and CAD.* 3D-EPUG-OVERLAY *has several innovations, including the use of exact rational arithmetic to avoid floating roundoff errors and the ensuing topological impossibilities. It also uses a uniform grid to efficiently index the geometric data.* 3D-EPUG-OVERLAY *was designed to be easily parallelizable. We are now incorporating Simulation of Simplicity, to correctly handle geometric degeneracies (coincidences). Our current implementation can easily process examples with millions of triangles.*

1. Introduction

Computing intersections or overlays is important to CAD, GIS, computer games and computational geometry. E.g., in 2D, consider two maps A and B, each composed of faces or polygons representing a partition of the E^2 plane. The overlay of A with B is a map C where each polygon of C is the intersection of a polygon of A with a polygon of B. For example, if A represents the coterminous states of the United States, and B represents drainage basins, then C is a new map where each polygon represents the part of each basin that is in each state.

While GIS usually deal with 2D geometric data, there are several applications for 3D GIS. For example, while a 2D map could model the street network of a city, the hydrological network of a state, or the different kinds of soil in a region, a 3D model could model more complex features such as layers of soil in a mine, the subway tunnels in a city, the buildings in region, etc Yanbing et al. (2007). Computing intersections is an important operation often required by these systems. An example of application is to intersect polyhedra representing layers of soil with a polyhedron representing a section of the soil to be digged in a mine. The resulting intersection represents the different kinds of soil that will be extracted during the excavation.

According to Feito et al. (2013), although 3D models have been widely used, processing is still a challenge. Due to the algorithm complexity caused by the need to handle special cases, the necessity of processing big volumes of data, and the loss if precision problems caused by floating point arithmetic, they note that software packages occasionally "fail to give a correct result, or they refuse to give a result at all". The likelihood of failure increases as datasets get bigger.

An algorithm that occasionally fails might be acceptable. Nevertheless, an efficient, robust, and even exact, algorithm is especially important when it is a subroutine of another algorithm.

Hachenberger et al. (2007) presented, and CGAL (2016) implemented, 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. However, according to Leconte et al. (2010), these algorithms have some limitations such as poor performance. Another limitation is their use of Nef Polyhedra, which are uncommon. Bernstein and Fussell (2009) also presented an intersection algorithm that tries to achieve robustness. Their basic idea is to represent the polyhedra using binary space partitioning (*BSP*) trees with fixed-precision coordinates. They mention that the main limitation is that the process to convert BSPs to widely used representations (such as meshes) is slow and inexact.

In previous works we have developed exact and efficient algorithms for processing 2D (polygonal maps) and 3D models (triangulated meshes). More specifically, we have developed algorithms for intersecting polygonal maps (Magalhães et al., 2015) and performing point location queries (Magalhães et al., 2016) in both polygonal maps and 3D meshes. These algorithms employ a combination of five separate techniques to achieve both robustness and efficiency. Exact arithmetic is employed to completely avoid errors caused by floating point numbers. Special cases (geometric degeneracies) are treated using *Simulation of Simplicity* (SoS) (Edelsbrunner and Mücke, 1990). The computation is performed using simple local information to make the algorithm easily parallelizable and to easily ensure robustness. Efficient indexing techniques with a uniform grid, and High Performance Computing (HPC) are used to mitigate the overhead of exact arithmetic.

In all these algorithms our spatial data is represented using simple topological formats. The 2D maps are represented using sets of oriented edges where each edge contains the labels of the polygons on its positive and negative sides. In 3D, the meshes are represented using a set of oriented triangles and each triangle has the labels of the polyhedra on its positive and negative sides.

In this paper we will present a brief description of these previous works and present our current research: 3D-EPUG-OVERLAY (3D-Exact Parallel Uniform Grid-Overlay), a parallel algorithm for exactly intersecting 3D triangulated meshes.

2. Roundoff errors

Non-integer numbers are usually approximated with floating-point values. The difference between a non-integer and its approximation is often referred as roundoff error. Even though these differences are usually small, arithmetic operations frequently create more errors, which accumulate, becoming larger.

In geometry, roundoff errors can generate topological inconsistencies causing globally impossible results for predicates like point-inside-polygon. For example, Kettner et al. (2008) presented a study of the failures caused by roundoff errors in geometric problems such as the planar orientation computation.

Several techniques have been proposed to overcome this problem. The simplest one consists of using an ϵ tolerance, and then consider two values x and y as equal if $|x-y| \leq \epsilon$. However this is not a good strategy because equality is no longer transitive, nor

invariant under scaling. In practice, epsilon-tweaking fails in several situations, (Kettner et al., 2008).

Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers (Hobby, 1999). However, snap rounding can generate inconsistencies and deforms the original topology if applied consecutively on a data set. Some variations of this technique attempt to get around these issues (Hershberger, 2013; Belussi et al., 2016).

Shewchuk (1996) presents the Adaptive Precision Floating-Point technique, that focus on exactly evaluating predicates. The idea is to perform this evaluation using the minimum amount of precision necessary to achieve correctness. As mentioned by the author, this technique focus on geometric predicates and it is not suitable to solve all geometric problems. For example, "a program that computes line intersections requires rational arithmetic".

The formally proper way to eliminate roundoff errors and guarantee robustness is to use exact computation based on rational numbers with arbitrary precision (Li et al., 2005; Hoffman, 1989; Kettner et al., 2008). In this work, our algorithms perform computation using arbitrary precision rationals provided by the GMP library. 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 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. This behavior is acceptable if the depth of the computation tree is small, which is true for the algorithms we will present.

Besides ensuring exact results, the use of arbitrary precision rationals has other advantages. First, Simulation of Simplicity, a technique for treating degeneracies, requires exact arithmetic. Second, our algorithms will be able to support input data where the coordinates are represented using rationals and, thus, we will be able to process meshes that cannot be exactly represented using floating point numbers.

3. Previous works

In this section, EPUG-OVERLAY (Magalhães et al., 2015) and PINMESH (Magalhães et al., 2016), two previous algorithms developed for, respectively, intersecting maps and performing point location queries in 3D meshes will be presented.

First, two important techniques applied in these works will be briefly described: the use of a uniform grid for indexing the data and the application of the Simulation of Simplicity technique for handling special cases. Both techniques will be also applied in the intersection algorithm described in this paper.

Understanding EPUG-OVERLAY, PINMESH and Sections 3.1 and 3.2 is important because techniques similar to the ones described in these sections are applied to 3D-EPUG-OVERLAY.

3.1. Indexing data with a uniform grid

Franklin et al. (1989) proposed a uniform grid to accelerate his algorithm for computing the area of overlaid polygons. When a polygonal map (or triangular mesh) is indexed with a uniform grid, a 2D grid (or 3D grid for meshes) is created, superimposed over the input

datasets, and then the edges (or triangles) intersecting each cell c are inserted into c. The efficiency of this idea depends on a careful choice of the concrete data structure. After the grid is created, it can be employed to accelerate the geometric algorithms. For example, given two maps indexed by the grid, the intersection of pairs of edges from the two maps can be found by processing each cell and comparing the edges in that cell pair-by-pair (one edge from each map) to compute the intersection points.

The uniform grid works well even for unevenly distributed data for various reasons (Akman et al., 1989; Franklin et al., 1988). First, the total time is the sum of one component (constructing the grid) that runs slower with a finer grid, plus other components (e.g., intersecting edges) that run faster. The total running-time varies slowly with changing grid resolutions. Second, an empty grid cell is very inexpensive, so that sizing the grid for the densest part of the data works.

Nevertheless, to process very uneven data, in EPUG-OVERLAY and PINMESH we have incorporated a second level grid into those few cells that are densely populated. The exact criteria for determine what cell to refine depends on the algorithm that will use the grid. For example, since in the intersection computation pairs of edges in the cells are tested for intersection, one could refine the grid cells where the number of intersection tests (i.e, the number of pairs of edges from the two maps) is greater than a threshold.

This nesting could be recursively repeated until all grid cells have fewer elements than a given threshold, creating a structure similar to quadtree (or octree), although with more branching. However, the general solution uses more space for pointers (or is expensive to modify) and is irregular enough that parallelization is difficult. Also, experiments have shown that the best performance is achieved using just a second level (Magalhães et al., 2015). This can be explained because the first level grid, in general, has many cells with more elements than the threshold justifying the second level refinement. But, in the second level, only a few number of cells exceed the threshold and the overhead (processing time and memory use) to refine those cells is never recaptured.

3.2. Simulation of Simplicity

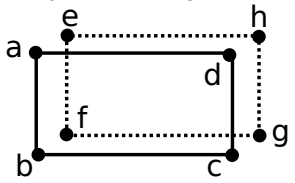
To correctly handle special cases such as coincident edges when intersecting maps, we apply Simulation of Simplicity (SoS) Edelsbrunner and Mücke (1990). This is a general purpose symbolic perturbation technique designed to treat special (degenerate) cases. The inspiration for SoS is that if the coordinates of the points are perturbed, the degeneracies disappear. However, too big a perturbation may create new problems, while a too small one may be ineffective because of the limited precision of floating point numbers.

SoS is a solution that uses a symbolic perturbation by an indeterminate infinitesimal value ϵ^i, for some natural number i. Its mathematical formalization extends some exactly computable field, such as rationals, by adding orders of infinitesimals, ϵ^i. Floating point numbers with roundoff error cannot be the base. The infinitesimal ϵ is an *indeterminate*. It has no meaning apart from the rules for how it combines. All positive first-order infinitesimals are smaller than the smallest positive number. All positive second-order infinitesimals are smaller than the smallest positive first-order infinitesimal, and so on. All this is logically consistent and satisfies the axioms of an abstract algrebra field.

The result of SoS is that degeneracies are resolved in a way that is globally consistent. For example, consider Figure 1. Two identical rectangles ($abcd$ represented using

solid edges and *efgh* represented using dashed edges) are overlaid, but all the vertices of *efgh* are slightly translated using the vector (ϵ, ϵ^2). This translation is globally consistent, i.e., even if the rectangle is stored as separate edges an intersection test with edge *ef* will return true only when this test is performed against the edge *ad* while an intersection test performed with *gf* will return true only when the test is performed against *cd*.

Figure 1. Effect caused by SoS during the intersection computation.



The infinitesimals do not need to be explicitly used in the program since they will be used only to determine signs of expressions. The only time that the infinitesimals change the result is when there is a tie in a predicate. Then, the infinitesimals break the tie. The effect is to make the code harder to write and longer. However, unless a degeneracy occurs, the execution speed is the same. When a degeneracy does occur, the code is slightly slower.

3.3. Point location

PINMESH (Magalhães et al., 2016) is an exact and efficient algorithm for performing point location queries in 3D meshes. It is based on the idea of ray-casting: given a query point q, a semi-infinite vertical ray r is traced from q, and then the triangle t whose intersection with r is the lowest is used to determine q's location. Since t is the lowest triangle to intersect r, because of the Jordan Curve Theorem, q will necessarily be on the polyhedron below t (this polyhedron can be quickly determined since all triangles contain the labels of the two polyhedra it bounds).

A uniform grid is used to reduce the number of ray-triangle intersection computation tests. Also, empty grid cells, which are each necessarily completely inside one polyhedron, are labeled with that containing polyhedron. That accelerates many queries. As a result of a careful implementation and use of parallelization, PINMESH is very efficient, being able to index a dataset and perform 1 million queries on a 16-core processor up to 27 times faster than RCT (Liu et al., 2010), a sequential and inexact algorithm, which was previously the fastest.

To summarize, PINMESH, represents coordinates with rational numbers to completely prevent roundoff errors, and handles special cases with simulation of simplicity.

3.4. Exact 2D map overlay

EPUG-OVERLAY (Magalhães et al., 2015) is an exact and efficient algorithm for overlaying two polygonal maps. Given two maps $\mathcal{A}$ and $\mathcal{B}$ composed of faces represented implicitly as sets of edges, the goal is to create a map where each face represents the intersection of a face of $\mathcal{A}$ with a face of $\mathcal{B}$. Parallel programming plus efficient indexing make EPUG-OVERLAY very efficient. It can process maps with more than 50 million edges faster than GRASS GIS, which is sequential and subject to roundoff error, since it does not use exact arithmetic.

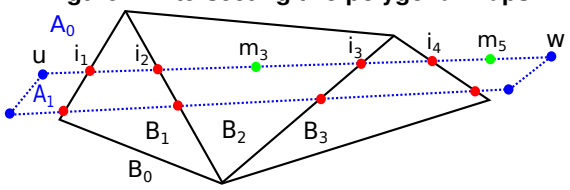
As described in (Magalhães et al., 2015), EPUG-OVERLAY has the following basic steps:

1. **Create a 2-level uniform grid** to index the edges from the two input maps A and B.
2. **Compute all intersection points between an edge of A with an edge of B** using the uniform grid is applied to accelerate the process, by iterating through the grid cells and testing all pairs of edges in each grid cell for intersection. The intersecting edges are split at the intersection point. After that, edges intersections will happen only at vertices.
3. **Label the resulting split edges** with their adjacent polygons.

Figure 2 illustrates this process: map $\mathcal{A}$ (in dotted blue) contains 4 edges and two polygons (polygon A_1 and polygon A_0, representing the exterior of the map) while map $\mathcal{B}$ (solid black lines) contains 7 edges and 4 polygons. After the intersections are detected and the edges are split at the intersection points (in red) the resulting edges are classified. For example, edge (u, w) bounds polygons A_0 (positive side) and A_1 (negative side). Edge (i_2, i_3) (generated after (u, w) was split) is inside polygon B_2 of the other map and, thus, in the output map (i_2, i_3) will bound polygon $A_0 \cap B_2$ (this polygon is equivalent to the exterior of the resulting map) on its positive side and $A_1 \cap B_2$ on the negative side.

Since the edges are split at the intersection points, after this process all edges will be completely inside a polygon of the other map. Thus, one strategy to determine in what polygon an edge e is consists in using a fast 2D point location algorithm to locate a point from e in the other map (for example, the location of m_3 from Figure 2 can determine in what polygon (i_2, i_3) is).

Figure 2. Intersecting two polygonal maps.



This strategy uses only local information to compute intersections. That is, instead of intersecting pairs of faces, the individual edges are intersected and classified; the resulting faces will be represented implicitly by the edges. This has several advantages. First, it is easier to test a pair of edges for possible intersection than to test a pair of faces, which would devolve to testing pairs of edges anyway. Second, knowing an intersection of a pair of edges contributes information about four output faces. Third, as an edge is fixed size but a face is not, parallel operations on edges are more efficient.

Degenerate cases are handled with *Simulation of Simplicity (SoS)*. The idea is to pretend that map $\mathcal{A}$ is slightly below and to the left of map $\mathcal{B}$. Thus no edge from $\mathcal{A}$ will coincide with an edge from $\mathcal{B}$ during the intersection computation. Oversimplified slightly, the process proceeds by translating map $\mathcal{B}$ by (ϵ, ϵ^2), where ϵ is an infinitesimal. As mentioned before, we do not actually compute with infinitesimals, but instead determine the effect that they would have on the predicates in the code, and modify the predicates to have the same effect when evaluated as if the variables could have infinitesimal values. For instance, the test for $(a_0 \leq b_0)\&(b_0 \leq a_1)$ becomes $(a_0 \leq b_0)\&(b_0 < a_1)$. With SoS, no point in $\mathcal{A}$ is identical to any point in $\mathcal{B}$, and neither do two any edges coincide.

4. Exact 3D mesh intersection

Similarly to our 2D intersection algorithm, in 3D the computation is performed using only local information stored in the individual triangles. That is, the triangles from one mesh are intersected with the triangles from the other one. Then a new mesh containing the triangles from the two original meshes is created and the original triangles are split at the intersection points. That is, if a pair of triangles in this new mesh intersect, then this intersection will happen necessarily in a common edge or vertex. Finally, the adjacency information stored in each triangle is updated to ensure that the new mesh will consistently represent the intersection of the original ones.

4.1. Intersecting triangles and remeshing

For performance, a strategy similar to the one used in EPUG-OVERLAY was adopted: for each uniform grid cell, the intersections between pairs of triangles from the two triangulations are computed. The pairs of triangles are intersected using the algorithm presented by Möller (1997), that uses several techniques to avoid unnecessary computation by detecting as soon as possible if the pair of triangles does not intersect.

More specifically, a two-level 3D uniform grid is employed to accelerate the computation using an strategy similar to the one we used in the 2D map intersection algorithm. That is, the grid will be created by inserting in its cells triangles from both meshes M_1 and M_2. Then, for each grid cell c, the pairs of triangles from both meshes in c are intersected. If the resolution of the uniform grid is chosen such that the expected number of triangles per grid cell is a constant K, then it is expected that each triangle will be tested for intersection with the other K triangles in its grid cell. Thus, the expected total number of intersections tests performed will be linear in the size of the input maps.

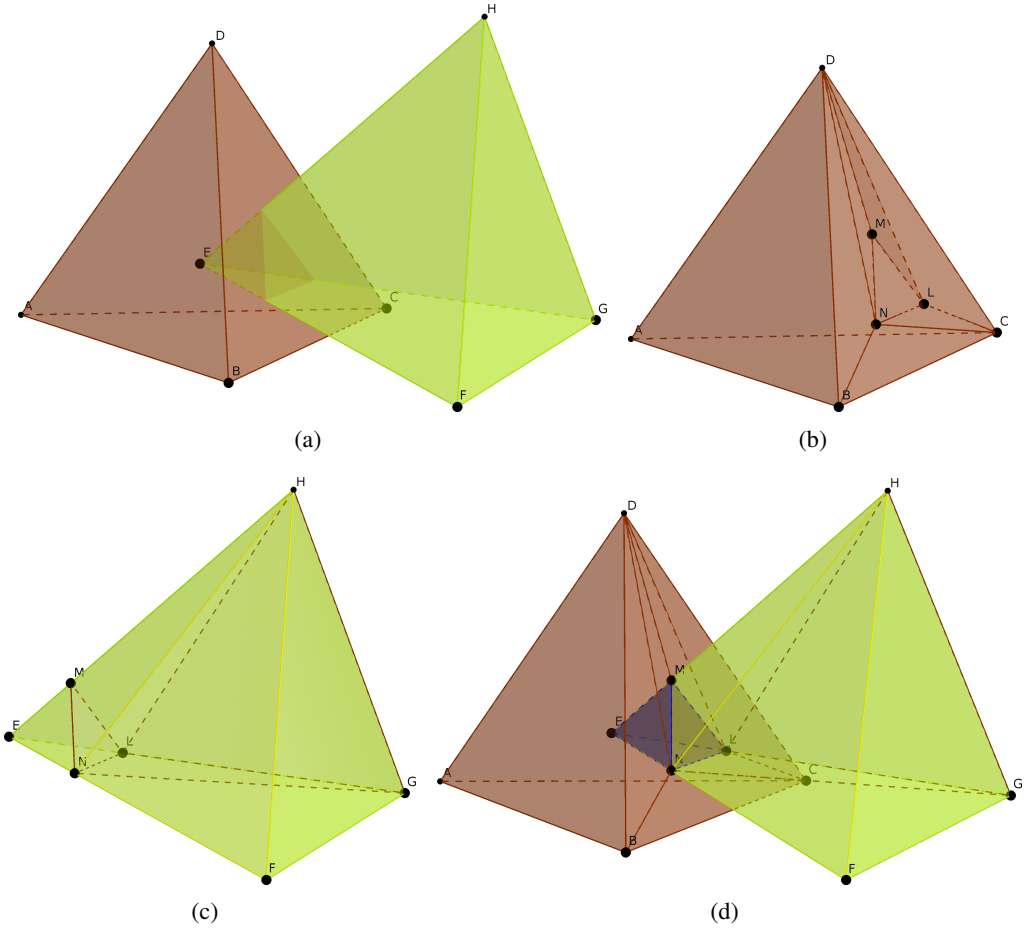
Since the cells do not influence each other, the process of intersecting the triangles can be trivially parallelized: the grid cells can be processed in parallel by different threads using a parallel programming API such as OpenMP.

After computing the intersections between each pair of triangles, the next step is to split the triangles where they intersect to create new ones, so that now all the intersections will happen only on common vertices or edges. When a triangle is split, the labels of its two bounding objects will be copied to the new triangles. This process is similar to the 2D map overlay step where the edges are split at the intersection points to ensure that all intersections happen in vertices.

Figure 3 presents an example of intersection computation. In Figure 3(a), we have two meshes representing two tetrahedra with one region in each one: the brown mesh (mesh M_1) bounds the exterior region and region 1 while the yellow mesh (mesh M_2) bounds the exterior region and region 2.

After the intersections between the triangles are computed, the triangles from one mesh that intersect triangles from the other one are split into several triangles, creating meshes M_1' and M_2' (for clarity, these two meshes are displayed separately in Figures 3(b) and (c), respectively). The only triangle from mesh M_1 that intersects mesh M_2 is the triangle BCD. Since BCD intersects three triangles from M_2, it was split in 7 triangles when M_1' was created (triangles LMN, CLN, CBN, BDN, DMN, DLM and CDL). Similarly, each of the three triangles from M_2 intersecting M_1 was split into 3 smaller triangles.

Figure 3. Computing the intersection of two tetrahedra.



4.2. Classifying triangles

After the intersections are detected and all the triangles that intersect other triangles are split at the intersection points, two new meshes M_1' and M_2' are created such that each new mesh M_i' will have the following two kinds of triangles:

- Triangles from the original mesh: if a 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 a 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 split into several smaller triangles and these smaller triangles will be inserted into M_i'.

It is clear that each mesh M_1' will exactly represent the same regions that M_1 represents. In fact, if no triangle from M_1 intersects the mesh M_2, then M_1' will be equal to M_1. Otherwise, each triangle t from M_i that intersects M_2 will be split in n triangles $t_1, t_2, ..., t_n$ and these new triangles will be inserted into M_i' instead of t. Since the union of the triangles $t_1, t_2, ..., t_n$ is t and these split triangles contain the same attributes as t, then M_1' represents the same regions M_1 represents. This observation is also valid for M_2'.

Thus, computing the intersection between M_1' and M_2' is equivalent to computing the intersection of M_1 with M_2. However, M_1' and M_2' are easier to process: since the triangles from one mesh intersect with the triangles of the other one only in common vertices or edges, then each triangle t from M_1' will be completely inside a region from M_2'. Suppose a triangle t from M_1' bounds regions R_a and R_b and is completely inside region R_c from mesh M_2'. When M_1' is intersected with M_2', t will be in the resulting mesh and it will bound regions $R_a \cap R_c$ and $R_b \cap R_c$. The same process can be performed with the triangles from M_2.

Therefore, the process of classifying the triangles to create the output mesh consists in processing each triangle t from the mesh M_1', determining in what region of M_2' t is and, then, updating the information about the regions t bounds such that we will have a consistent mesh. The same process needs to be performed with triangles from M_2'.

To determine in what region from the other mesh a triangle is, the point location algorithm from Section 3.3 is applied. That is, since point location queries can be quickly performed, an efficient way to locate a triangle that is completely inside a region consists in locating one of its interior points (for example, its centroid).

This classification can also be performed in parallel since updating the regions that a triangle bounds does not influence other triangles.

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 3(d) illustrates the classification step. All the intersections happen at common edges, and the only triangle from M_1' that is completely inside region 2 (of M_2') is triangle LMN. Since LMN bounds region 1 and the exterior region in M_1', in the resulting intersection LMN will bound region $1 \cap 2$ and the exterior region. All the other triangles from M_1' are in the exterior region of M_2' 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_1' the only triangles that are inside region 1 of M_1' are triangles EMN, ELM and ELN. These three triangles will also bound the exterior region and region $1 \cap 2$ in the resulting mesh.

4.3. Handling the special cases

The current version of 3D-EPUG-OVERLAY does not handle special cases (degeneracies) yet. However, the ideas we intend to apply in order to handle these cases have already been successfully implemented for EPUG-OVERLAY and PINMESH, and therefore we believe they will suitable to 3D-EPUG-OVERLAY.

Without SoS, it would be too difficult to guarantee that all degeneracies are considered (this is particularly true in 3D). An adequate perturbation scheme associated with the use of exact arithmetic and a careful implementation will ensure our intersection algorithm is robust.

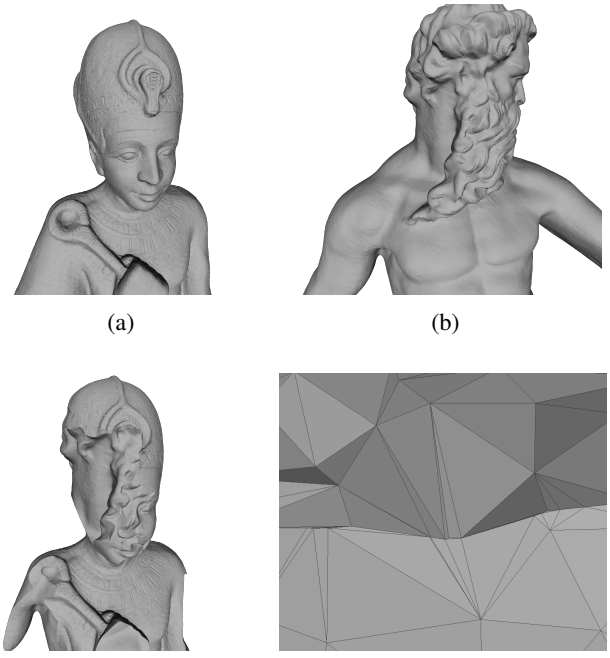
5. Preliminary results

3D-EPUG-OVERLAY was implemented in C++, and several experiments performed. Figure 4 presents an example of intersection computed using 3D-EPUG-OVERLAY: the

model Ramesses (a) and Neptune (b) were intersected. These two models were downloaded from the repository in AIM@SHAPE (2016), and were produced by, respectively, Marco Attene and Laurent Saboret. The Ramesses model contains more than 1 million triangles while the Neptune model contains more than 4 million triangles. Figure 4(c) shows the result of the intersection.

Figure 4(d) presents a zoom that detaches the region of the resulting mesh where the triangles from the two models intersect. We see that the remeshing process generates several thin triangles (displayed in the vertical center of the figure), which are usually hard to process with methods using floating-point arithmetic.

Figure 4. Computing the intersection of two 3D models.



(a) (b)

(c) (d)

Since some features of 3D-EPUG-OVERLAY, such as SoS, are still under implementation, and the main feature of 3D-EPUG-OVERLAY is its exactness, we intend to optimize its performance only after those features are implemented. However, we intend to employ the same strategies successfully used in EPUG-OVERLAY and PINMESH. They include:

1. Trading memory for computation, pre-computing and storing results that will be needed several times.
2. Parallelization of the bottlenecks of the algorithm using OpenMP: similarly to our previous work, 3D-EPUG-OVERLAY was designed specifically for being easily parallelizable.
3. Reduction of memory allocations on the heap since they cannot be efficiently performed in parallel. Our previous experience has showed that this should be avoided especially inside parallelized blocks of code. However, as rationals grow, memory needs to be allocated. Therefore we pre-allocate enough temporary rationals that creating them inside parallelized functions is not necessary.

Since these techniques were so sucessful in our previous works, so that they even outperformed inexact algorithms, we believe they will also make 3D-EPUG-OVERLAY very efficient.

6. Conclusion and future work

We have presented 3D-EPUG-OVERLAY, an exact and parallel algorithm for computing the intersection of 3D models represented by triangulated meshes. 3D-EPUG-OVERLAY uses arbitrary precision rational numbers to store all the geometric coordinates and perform computation, and so is roundoff error free.

Even though the current implementation of 3D-EPUG-OVERLAY does not treat special cases, preliminary experiments have indicated that 3D-EPUG-OVERLAY can successfully intersect some big meshes available in public repositories.

Next, we intend to implement a symbolic-perturbation scheme on 3D-EPUG-OVERLAY to ensure that all the special cases are properly handled. Furthermore, the optimization techniques that have been so successful in our previous works will be also applied to 3D-EPUG-OVERLAY.

7. Acknowledgement

This research was partially supported by CNPq, CAPES (Ciência sem Fronteiras), FAPEMIG and NSF grant IIS-1117277.

References

AIM@SHAPE (2016). AIM@SHAPE-VISIONAIR Shape Repository. `http://visionair.ge.imati.cnr.it//` (accessed on Sep-2016).

Akman, V., Franklin, W. R., Kankanhalli, M., and Narayanaswami, C. (1989). Geometric computing and the uniform grid data technique. *Comput. Aided Design*, 21(7):410–420.

Belussi, A., Migliorini, S., Negri, M., and Pelagatti, G. (2016). Snap rounding with restore: An algorithm for producing robust geometric datasets. *ACM Trans. Spatial Algorithms Syst.*, 2(1):1:1–1:36.

Bernstein, G. and Fussell, D. (2009). Fast, exact, linear booleans. *Eurographics Symposium on Geometry Processing*, 28(5):1269–1278.

CGAL (2016). CGAL, Computational Geometry Algorithms Library. http://www.cgal.org (accessed on Sep-2016).

Edelsbrunner, H. and Mücke, E. P. (1990). Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (TOG)*, 9(1):66–104.

Feito, F., Ogayar, C., Segura, R., and Rivero, M. (2013). Fast and accurate evaluation of regularized boolean operations on triangulated solids. *Computer-Aided Design*, 45(3):705 – 716.

Franklin, W. R., Chandrasekhar, N., Kankanhalli, M., Seshan, M., and Akman, V. (1988). Efficiency of uniform grids for intersection detection on serial and parallel machines. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, pages 288–297. Springer-Verlag.

Franklin, W. R., Sun, D., Zhou, M.-C., and Wu, P. Y. (1989). Uniform grids: A technique for intersection detection on serial and parallel machines. In *Proceedings of Auto Carto 9*, pages 100–109, Baltimore, Maryland.

Hachenberger, P., Kettner, L., and Mehlhorn, K. (2007). Boolean operations on 3d selective nef complexes: Data structure, algorithms, optimized implementation and experiments. *Computational Geometry*, 38(1):64–99.

Hershberger, J. (2013). Stable snap rounding. *Computational Geometry*, 46(4):403–416.

Hobby, J. D. (1999). Practical segment intersection with finite precision output. *Comput. Geom.*, 13(4):199–214.

Hoffman, C. M. (1989). The problems of accuracy and robustness in geometric computation. *Computer*, 22(3):31–40.

Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., and Yap, C. (2008). Classroom examples of robustness problems in geometric computations. *Comput. Geom. Theory Appl.*, 40(1):61–78.

Leconte, C., Barki, H., and Dupont, F. (2010). Exact and efficient booleans for polyhedra. Citeseer.

Li, C., Pion, S., and Yap, C.-K. (2005). Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming*, pages 85–111.

Liu, J., Chen, Y. Q., Maisog, J. M., and Luta, G. (2010). A new point containment test algorithm based on preprocessing and determining triangles. *Comput. Aided Des.*, 42(12):1143–1150.

Magalhães, S. V., Andrade, M. V., Franklin, W. R., and Li, W. (2016). Pinmesh - fast and exact 3d point location queries using a uniform grid. *Computers & Graphics*, 58:1 – 11. Shape Modeling International 2016.

Magalhães, S. V. G., Andrade, M. V. A. A., Franklin, W. R., and Li, W. (2015). Fast exact parallel map overlay using a two-level uniform grid. In *Proc. of the 4th ACM Bigspatial*, BigSpatial '15, New York, NY, USA. ACM.

Möller, T. (1997). A fast triangle-triangle intersection test. *Journal of graphics tools*, 2(2):25–30.

Shewchuk, J. R. (1996). Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363.

Yanbing, W., Lixin, W., Wenzhong, S., and Xiaomeng, L. (2007). On 3d gis spatial modeling. In *Proceedings of the ISPRS Workshop on Updating Geo-spatial Databases with Imagery and the 5th ISPRS Workshop on DMGISs, Urumchi, Xinjiang, China*, pages 237–240. Citeseer.