EXACT AND PARALLEL INTERSECTION OF 3D TRIANGULAR MESHES

By

Salles Viana Gomes de Magalhães

A Dissertation Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major Subject: COMPUTER SCIENCE

Examining Committee:

_____
Dr. W. Randolph Franklin, Dissertation Adviser

_____
Dr. Christopher D. Carothers, Member

_____
Dr. Barbara M. Cutler, Member

_____
Dr. Richard J. Radke, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2017
(For Graduation December 2017)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I want first to thank my supervisor, Dr. Franklin, for all the support during my Ph.D. studies. His enthusiasm, friendliness and encouragements were fundamental to the development of this work.

Also, I want to thank my lovely family for the support. I would especially like to thank my wife Lígia, my parents Eliezer and Ana, my sisters Maisa and Laura, and my grandmothers Cecília and Odete.

Furthermore, I would like to express my appreciation to all the wonderful professors from Universidade Federal de Viçosa for all the support and encouragement. I especially owe a great deal of gratitude to Drs. Andrade, Bastos and Ferreira.

The professors of Rensselaer Polytechnic Institute were also very important to the development of this work. I would especially like to thank the members of my thesis committee: Drs. Carothers, Cutler and Radke.

Last, but not least, I want to dedicate this thesis to the memory Alcy Barcelos Gomes, my grandfather and teacher. I owe a great deal of appreciation and gratitude to him. His kindness, dedication and intelligence were fundamental for stimulating my curiosity and interest in science since my childhood.

This research was partially supported by CAPES (Ciência sem Fronteiras - grant 9085/13-0) and NSF (grant IIS-1117277).

ABSTRACT

This thesis presents an exact parallel algorithm for computing the intersection between two 3D triangular meshes, as used in CAD/CAM (Computer Aided Design/Computer Aided Manufacturing), CFD (Computational Fluid Dynamics), GIS (Geographical Information Science) and additive manufacturing (also known as 3D Printing). Geometric software packages occasionally fail to compute the correct result because of the algorithm implementation complexity (that usually needs to handle several special cases) and of precision problems caused by floating point arithmetic. A failure in an intersection computation algorithm may propagate to any software using the algorithm as a subroutine. As datasets get bigger (and the chances of failure in an inexact algorithm increase), exact algorithms become even more important.

While other methods for exactly intersecting meshes exist, their performance makes them non-suitable for applications where the fast processing of big geometric models is important (such as interactive CAD systems).

The key to obtain robustness and performance is a combination of 5 separate techniques:

- Multiple precision rational numbers, to exactly represent the coordinates of the objects and completely eliminate roundoff errors during the computations.
- Simulation of Simplicity, a symbolic perturbation technique, to ensure that all geometric degeneracies (special cases) are properly handled.
- Simple data representations and local 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 some of the steps of the algorithm such as testing pairs of triangles for intersection or locating points in the mesh.

- Parallel programming, to accelerate the intersection process and explore better the ubiquitous parallel computing capability of current hardware.

To evaluate our ideas, we have developed and implemented EPUG-Overlay (*Exact Parallel Uniform Grid Overlay*), an exact and efficient algorithm for overlaying 2D maps. EPUG-Overlay applies all the techniques mentioned above and, as a result, it was not only exact but also very efficient. Indeed, it was able to compute the intersection of a map containing 32 million edges with another map having 21 million edges in 264 elapsed seconds using 16 threads with dual 8-core processors. By comparison, GRASS (a Geographic Information System) took $5,300$ seconds to perform the same computation, partly because it is only single-threaded.

We have also developed PinMesh (*Point In Mesh*), an exact and efficient method for locating points in 3D meshes. The point location problem is an important step we intend to use for computing the intersection of 3D meshes. PinMesh was carefully implemented to always handle point location queries correctly. The use of efficient data structures associated with parallel programming made PinMesh very fast. According to our experiments, PinMesh was up to 27 times faster than the RCT (*Relative Closest Triangle*) 3D point location algorithm (that was, to the best of our knowledge, the fastest point location algorithm available).

Finally, we developed 3D-EPUG-Overlay, an algorithm for exactly computing the intersection of pairs of 3D meshes. 3D-EPUG-Overlay employs all the techniques mentioned above to ensure its correctness and efficiency. Experiments showed that it was up to 101 times faster than the algorithm available in LibiGL (the state of art algorithm for exact mesh intersection) and, also, it had a performance that was comparable to a parallel inexact algorithm that was specifically developed to be very fast. Besides being fast, 3D-EPUG-Overlay was more memory efficient than the other evaluated methods. Furthermore, in all test cases the result obtained by 3D-EPUG-Overlay matched the reference solution.

All the software developed for this thesis is freely available for other researchers to use and extend.

CHAPTER 1
Introduction

Computing intersections is a very important operation for CAD systems, computer games, computational geometry, graphic editing, CFD, and GIS. In 2D, given a pair of maps A and B, that are composed of sets of faces or polygons representing partitions of the E^2 plane, the intersection of A with B is a map C where each polygon is the intersection of a polygon of A with a polygon of B. For example, the intersection of a map representing the states of the United States with a map representing American drainage basins is another map where the polygons represent the portion of each basin that is in each state.

These operations also extend to 3D objects. For example, if a set of 3D solids

---

Portions of this chapter previously appeared as:

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li, "Fast exact parallel map overlay using a two-level uniform grid", in *in Proc. 4th ACM SIGSPATIAL Int. Workshop Analytics for Big Geospatial Data*, BigSpatial'15. New York, NY, USA: ACM, 2015.

M. G. Gruppi, S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li, "Using rational numbers and parallel computing to efficiently avoid round-off errors on map simplification", in *Proc. XVI Brazilian Symp. Geoinformatics*, GeoInfo'15, 2015, pp. 162 - 173.

S. V. G. Magalhães, "An efficient algorithm for computing the exact overlay of triangulations", in *Proc. 2nd ACM SIGSPATIAL PhD Workshop*, SIGSPATIAL PhD'15. New York, NY, USA: ACM, 2015, pp. 3:13:4.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li,"Pinmesh - Fast and exact 3D point location queries using a uniform grid", *Comput. & Graph.*, vol. 58, pp. 1 - 11, 2016, Shape Modeling International 2016.

S. V. G. Magalhães, W. R. Franklin, and M. V. A. Andrade, "Fast exact parallel 3D mesh intersection algorithm using only orientation predicates", *Proc. 17th ACM SIGSPATIAL Int. Conf. Advances in Geographic Information Systems*, SIGSPATIAL'17. New York, NY, USA:ACM, 2017.

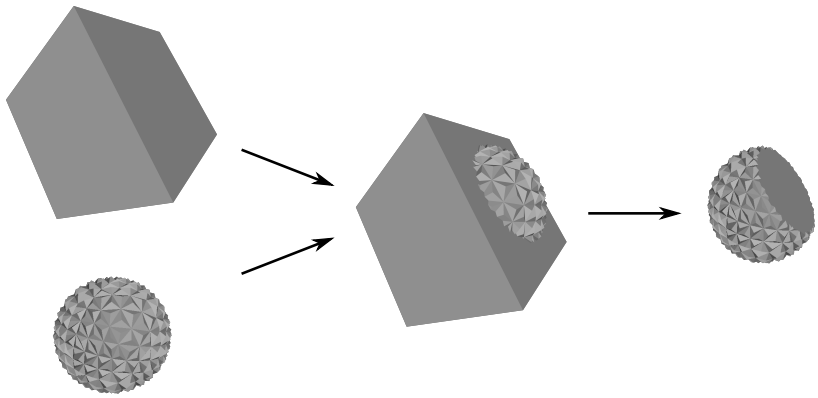


Figure 1.1: Example of mesh intersection.

A represents the layers of soil in a given region and another set B contains one polyhedron representing a section of the soil that will be excavated, the intersection between A and B will represent the different layers of soil that will be extracted during the digging. Figure 1.1 presents an example of the intersection of two polyhedra.

The ability to represent and process 3D data in CAD/CAM and GIS is important in several domains such as architecture, engineering, urban planning, transportation, and military planning [1]. Figure 1.2 illustrates the use of the intersection of two 3D models to create an engineering part in a CAD system.

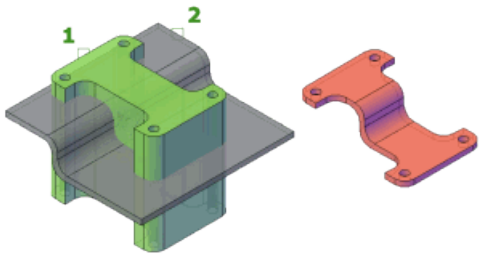


Figure 1.2: Creation of an engineering part (in red) as the intersection of two 3D models in a CAD system. Source: [2].

A popular way to represent 3D objects is the triangular mesh [3], where the boundary of the solids are represented by a set of triangles that are connected in their common edges and vertices.

However, according to Feito et al. [3], although 3D models have been widely

used in computer science, processing them is still a challenge. Due to the algorithm implementation complexity (that usually needs to handle several special cases), the necessity of processing big volumes of data and precision problems caused by floating point arithmetic, software packages occasionally "fail to give a correct result, or they refuse to give a result at all" [3]. The likelihood of failure increases as datasets get bigger. This is a challenge particularly in the problem of intersecting triangulations.

Even though in some situations an algorithm that occasionally fails is acceptable, it is often important to have an algorithm that is both efficient and robust. For example, overlay algorithms are frequently used as subroutines for other algorithms and, therefore, if their outputs are not exact or they are too slow, these problems may propagate to the algorithms using them.

The main goal of this thesis is to develop an efficient and robust algorithm for exactly computing the overlay of 3D triangular meshes. This algorithm employ a combination of 5 separate techniques to achieve both robustness and efficiency. Exact arithmetic is employed to completely avoid errors caused by floating point numbers. Special cases are treated using *Simulation of Simplicity* (SoS) [4]. The computation is performed using simple local information to make the algorithm easily parallelizable and to easily ensure robustness. Since the use of exact arithmetic is expected to add an overhead, efficient indexing techniques and High Performance Computing (HPC) are employed to mitigate this.

As a proof of concept, we initially developed a 2D map intersection algorithm (named EPUG-Overlay– *Exact Parallel Uniform Grid Overlay*), that can efficiently compute the intersection between 2 maps using exact arithmetic. EPUG-Overlay uses the five techniques mentioned above.

Furthermore, we have developed an efficient and exact algorithm for performing point location in triangular meshes. Besides being an important step of the intersection computation, locating points is also an important problem in computational geometry and, thus, part of this thesis is dedicated to present our solutions to this problem.

Finally, the experience acquired during the development of EPUG-Overlay and PinMesh was employed to develop 3D-EPUG-Overlay, our exact algorithm

for computing the intersection of pairs of 3D triangulations.

The main contributions of this thesis are:

- We presented EPUG-Overlay, an efficient and exact 2D map overlay algorithm.
- We developed PinMesh, an algorithm for exactly locating points in 3D meshes.
- We presented 3D-EPUG-Overlay, an exact and efficient algorithm for mesh intersection.
- We designed the three aforementioned algorithms to be parallelizable, to better use the computing capability of current computers (typically containing multicore processors).
- We showed that the symbolic perturbation scheme employed by 3D-EPUG-Overlay eliminates all the special cases.
- We combined exact arithmetic with symbolic perturbation to guarantee that 3D-EPUG-Overlay is exact.
- We combined a series of techniques such as a 3D parallel uniform grid and arithmetic filtering to make 3D-EPUG-Overlay not only exact but also efficient.
- Experiments showed that 3D-EPUG-Overlay was up to 101 times faster than the state of the art algorithm for intersecting meshes.

This work is organized as follows:

- Chapter 2 cites related works and presents a background about the problems solved in this thesis and the challenges associated with them.
- Chapter 3 presents our exact 2D map intersection algorithm.
- Chapter 4 presents the implementation details of the 3D uniform grid employed in the point location and mesh intersection algorithms.

- Chapter 5 describes the algorithm developed for exactly locating points in 3D meshes.
- Chapter 6 presents the algorithm for exactly computing the intersection of 3D meshes.

CHAPTER 2
Background and related works

2.1 Roundoff errors

Usually, non-integer numbers are approximately represented in computers with floating-point values. The difference between the value of a non-integer number and its approximation is often referred as roundoff error. Even though these differences are usually small, arithmetic operations frequently create more errors and, thus, a

---

Portions of this chapter previously appeared as:

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li, "Fast exact parallel map overlay using a two-level uniform grid", in *in Proc. 4th ACM SIGSPATIAL Int. Workshop Analytics for Big Geospatial Data*, BigSpatial'15. New York, NY, USA: ACM, 2015.

M. G. Gruppi, S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li, "Using rational numbers and parallel computing to efficiently avoid round-off errors on map simplification", in *Proc. XVI Brazilian Symp. Geoinformatics*, GeoInfo'15, 2015, pp. 162 - 173.

S. V. G. Magalhães, "An efficient algorithm for computing the exact overlay of triangulations", in *Proc. 2nd ACM SIGSPATIAL PhD Workshop*, SIGSPATIAL PhD'15. New York, NY, USA: ACM, 2015, pp. 3:13:4.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li,"Pinmesh - Fast and exact 3D point location queries using a uniform grid", *Comput. & Graph.*, vol. 58, pp. 1 - 11, 2016, Shape Modeling International 2016.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, M. G. Gruppi and W. Li, "Exact intersection of 3D geometric models", in *Proc. XVII Brazilian Symp. Geoinformatics*, GeoInfo'16, 2016, pp. 44 - 55.

S. V. G. Magalhães, W. R. Franklin, and M. V. A. Andrade, "Fast exact parallel 3D mesh intersection algorithm using only orientation predicates", *Proc. 17th ACM SIGSPATIAL Int. Conf. Advances in Geographic Information Systems*, SIGSPATIAL'17. New York, NY, USA:ACM, 2017.

set of operations performed in sequence usually leads to larger errors.

The presence of floating point errors in computer programs often creates serious consequences in diverse fields such as the failure of the first Ariane V rocket [5], the failure of the Patriot missile defense system [6] and the error in the Vancouver Stock Exchange index [7].

In geometry, roundoff errors can generate topological inconsistencies causing globally impossible results for predicates like point inside polygon. Kettner et al. [8] presented an interesting study of the failures caused by roundoff errors in geometric problems. In this study, they presented examples of how rounding in floating point arithmetic affects the planar orientation predicate and, as consequence, other applications that rely on this predicate such as a planar convex hull algorithm.

The planar orientation predicate is the problem of finding whether three points p, q, r are collinear, make a left turn, or make a right turn. This predicate is computed by verifying the sign of the following determinant containing the points:

$$\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}$$

This determinant will be positive, negative or zero which means that points (p, q, r) form a left turn, right turn or are collinear, respectively. Due to round-off errors in floating point arithmetic the results can be classified incorrectly due to *rounding to zero*, *perturbed zero*, or *sign inversion*. Respectively, it means a non-zero result may be rounded to zero, a zero result may be mis-classified as positive or negative, and a positive result may be mis-classified as negative or vice-versa.

To observe the occurrence of issues caused by floating-point arithmetic, Kettner et al. [8] developed a C++ program to apply the planar orientation predicate ($orientation(p, q, r)$) on a point $p = (p_x + xu, p_y + yu)$ where u is the step between adjacent floating point numbers in the range of p and $0 \leq x, y \leq 255$. This results in a 256×256 matrix containing either 1, -1 or 0 if the point corresponding to the matrix position is to the right, to the left or on the line that passes through q and r. Figure 2.1 shows the geometry of this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$,

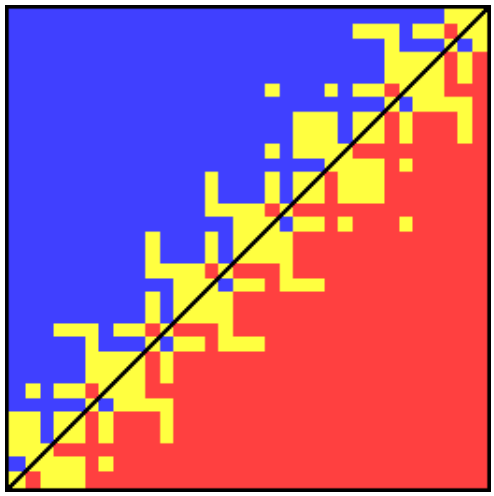


Figure 2.1: Roundoff errors in the planar orientation problem - Geometry of the planar orientation predicate for double precision floating point arithmetic. Yellow, red and blue points represent, respectively, collinear, negative and positive orientations. The diagonal line is an approximation of the segment (q, r). Source: [8].

$q = (12, 12)$ and $r = (24, 24)$. The black diagonal line is an approximation of part of the segment (q, r). Yellow, red and blue cells represent, respectively, collinear, negative and positive orientations with respect to the segment (q, r). According to Kettner et al., even using extended double arithmetic was not enough to overcome this issue.

As shown by [8], these inconsistent results in the $orientation(p, q, r)$ predicate could make algorithms that use this predicate (such as the Incremental Convex Hull algorithm) to fail.

Several techniques have been proposed in order to overcome this problem. The simplest one consists of using an ϵ tolerance to consider two values x and y are equal if $|x - y| \leq \epsilon$. However this is a formal mess because equality is no longer transitive, nor invariant under scaling. Thus, in practice, epsilon-tweaking fails in several situations [8].

Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers [9]. However, Snap rounding can generate inconsistencies and deform the original topology if applied consecutively on a data set. Figure 2.2 illustrates how Snap rounding deform the original geometry of some

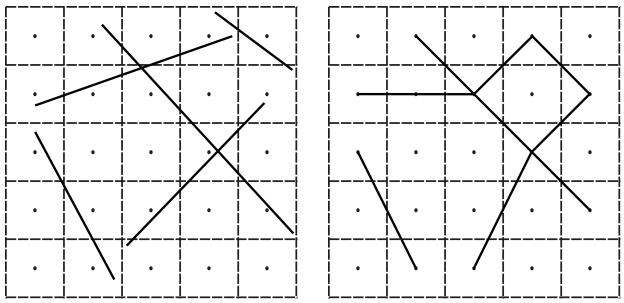


Figure 2.2: Snap rounding - (a) Initial set of segments. (b) Arrangement after the snap rounding execution: each segment endpoint and each intersection point has been snapped to the pixel center.

segments. Some variations of this technique attempt to get around these issues [10]–[12].

Controlled Perturbation (CP), [13] is another approach, based on the use of finite precision approximation techniques. The basic idea is to slightly perturb the input in a controlled manner to remove all degeneracies and such that all the geometric predicates are correctly evaluated even using floating-point arithmetic.

Boissonat [14] describes a robust implementation of the plane sweep approach for detecting segments that intersect using triple of the precision of the input data. The basic idea is to implement the algorithm using predicates that can be reliably evaluated using floating point numbers with more precision than the input numbers.

Shewchuk [15] presents the Adaptive Precision Floating-Point technique, that focus on exactly evaluating predicates (e.g. orientation tests). The idea is to perform this evaluation using the minimum amount of precision necessary to achieve correctness. As a result, it is possible to develop some efficient exact geometric algorithms. Geometric predicates can often be evaluated by computing the sign of a determinant and, thus, the actual value of this determinant does not need to be exactly computed as long as the sign of the approximated result is guaranteed to be correct. To determine if the sign of an approximation can be trusted, the approximation and an error estimate are computed and, if the error is big enough to make the sign possibly incorrect, the values are recomputed using higher precision. As mentioned by Shewchuk [15], this technique focuses on geometric predicates and

it is not suitable to solve all geometric problems. For example, "a program that computes line intersections requires rational arithmetic; an exact numerator and exact denominator must be stored" [15].

Li [16] presented the Exact Geometric Computation (EGC) model, which represents mathematical objects using algebraic numbers to perform computations without errors. By definition, an algebraic number is the root of an univariate polynomial with integer coefficients. For instance, the number $\sqrt{2}$ has no finite representation, but it can be represented exactly as the pair $(x^2 - 2, [1, 2])$, interpreted as the root of the polynomial $x^2 - 2$ that lies in the interval [1, 2]. This model has some interesting features but its main drawback is the performance penalty. Even determining the sign of an expression is nontrivial. E.g.: what is the sign of $\sqrt{\sqrt{\sqrt{5}+1} + \sqrt{\sqrt{5}-1}} - \sqrt[4]{2\sqrt{5}+4}$? (answer: 0)

The formally proper way to effectively eliminate roundoff errors and guarantee algorithm robustness is to use exact computation based on rational number with arbitrary precision [8], [17]–[19]. In this work we intend to develop algorithms that are efficient enough to perform the computations 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., $\frac{214}{433} + \frac{659}{781} = \frac{452481}{338173}$. Casting out common factors helps, but that is rarely possible. E.g., we can cast out a common factor of two when the numerator and denominator are both even — 1/4 of the time. This behavior is acceptable if the depth of the computation tree is small, which is true for the algorithms we will present. Furthermore, as it will be mentioned in section 2.1.1, in some situations the exact computation can be further accelerated by employing techniques such as interval arithmetic.

Besides ensuring exact results in the predicates and arithmetic computations, the use of arbitrary precision rational numbers has other two advantages. First, Simulation of Simplicity [4], the technique we use for treating the special cases, requires exact arithmetic in order to work properly. Second, our algorithm will be

able to support input geometrical data where the coordinates are exactly represented using rational numbers and, thus, we will be able to process meshes that cannot be exactly represented using floating point arithmetic.

2.1.1 Arithmetic filters and interval arithmetic

One technique to accelerate algorithms based on exact arithmetic is to employ arithmetic filters and interval arithmetic [20]. The idea is to use an interval of floating-point numbers containing each exact value. During the evaluation of predicates (which typically consists in the computation of the sign of an arithmetic expression), the arithmetic operations are initially applied to the intervals. After each arithmetic operation the result (an interval) is adjusted to guarantee that it will still contain the exact result of the operation (this is called the containment property). At the end, if the sign of the exact result can be safely inferred based on the sign of the bounds of the interval, its value is returned. Otherwise, the predicate is re-evaluated using exact arithmetic instead of the floating-point intervals. The term *arithmetic filter* derives from the process of filtering the unreliable results and recomputing them with exact arithmetic.

The key to the correct and efficient implementation of operations with interval arithmetic is the fact that the IEEE-754 standard for floating-point numbers explicitly define how the arithmetic operations are approximated: "the result of operations can be seen as if they were performed exactly, but then rounded to one of the nearest floating-point values enclosing the exact value" [20]. IEEE-754 also defines three rounding-modes (that can be selected at runtime): the results of the operations can be rounded to the nearest representable floating-point value, towards $-\infty$ or $+\infty$ (which selects, respectively, the previous or the next nearest representable floating-point numbers).

These rounding modes are employed to adjust the intervals after each arithmetic operation, which guarantees that they always contain the exact value of the expressions. [20] illustrates this process with the addition operation. Suppose $xInterval = [x.lower, x.upper]$ and $yInterval = [y.lower, y.upper]$ are, respectively, floating-point intervals containing the exact values $xExact$ and $yExact$. The

floating-point interval $[x.lower \pm y.lower, x.upper \mp y.upper]$ (where $\pm$ and $\mp$ represent, respectively, rounding towards $-\infty$ or $+\infty$) is guaranteed to contain the exact value of the expression $xExact + yExact$.

Since the intervals are computed in a way that the containment property is always preserved, if both bounds have the same sign then this sign is equal to the exact sign of the expression. Otherwise, the interval cannot be employed to infer the exact sign and thus, the expression will have to be re-evaluated with exact arithmetic. For example, if $xExact$ is in the interval $[0.01, 0.03]$, then $xExact$ is certainly a positive number. However, if $xExact$ is in the interval $[-0.0001, 0.0001]$, then the sign of $xExact$ can be either negative, zero or positive.

Since the roundoff errors accumulate, the width of the intervals increases as arithmetic operations are performed and thus, the deeper the computation tree is, the higher are the chances that computation with exact arithmetic will be necessary, which could slow down the algorithms. However, many practical algorithms do not present this problem [20]. The method proposed in this work, for example, has a shallow computation tree.

While arithmetic filters can accelerate predicates, in some situations the exact computation cannot be avoided. For example, exact arithmetic would be necessary in operations where new coordinates have to be computed (these types of operations are called *geometric constructions*). To illustrate this example, consider the problem of computing pairwise intersections of line segments: arithmetic filters could be employed to accelerate the orientation predicates employed to detect if two line segments intersect, but exact arithmetic is necessary in order to output the coordinates of the vertices generated by the intersection of pairs of edges.

The excellent Computational Geometry Algorithms Library (CGAL) [21] supports exact computation through the use of arbitrary precision rational numbers and arithmetic filters in its algorithms. Furthermore, this library provides a framework that allows programmers to easily develop algorithms with arithmetic filters.

Listing 2.1 illustrates one of the ways to develop an arithmetic filter using C++ and CGAL: variables with the suffix *Exact* were created as GMP (*GNU Multiple Precision Arithmetic Library*) arbitrary precision rationals (which are represented

using the *mpq_class* type) while the ones with suffix *Interval* were defined using the interval arithmetic number type provided by CGAL. Arithmetic and boolean operators are overloaded for both the interval and arbitrary precision arithmetic types. If the comparison (line 4) cannot be evaluated safely, CGAL throws an *unsafe_comparison* exception. Once that exception is caught, the predicate can be re-evaluated using the exact version of the respective variables (line 7).

Listing 2.1: Using CGAL interval arithmetic framework

```
1  //This predicate returns 1 if the sum of xExact with yExact is
2  //positive and 0 otherwise
3  //xInterval and yInterval must contain, respectively, xExact and
4  //yExact
5  bool predicate(mpq_class xExact,CGAL::Interval_nt<true> xInterval,
6                 mpq_class yExact,CGAL::Interval_nt<true> yInterval){
7      try {
8          if(xInterval+yInterval > 0) return 1;
9          else return 0;
10     } catch(CGAL::Interval_nt<true>::unsafe_comparison &ex) {
11         if(xExact+yExact > 0) return 1;
12         else return 0;
13     }
14 }
```

As will be mentioned in Chapter 6, arithmetic filters will be employed to accelerate the mesh intersection algorithm presented in this thesis. Even though arithmetic filters could also be employed to accelerate the 2D map overlay algorithm presented in Chapter 3 and the standalone point location algorithm described in section 5, we have not implemented this optimization in these two algorithms yet (this will be left as future work).

2.2 Representing spatial data

2.2.1 2D maps

For the scope of this work, a *map* (or *polygonal map*) is a partition of the E^2 plane into a finite number of *faces* or polygons. Except for the infinite outside face, all the other faces are finite.

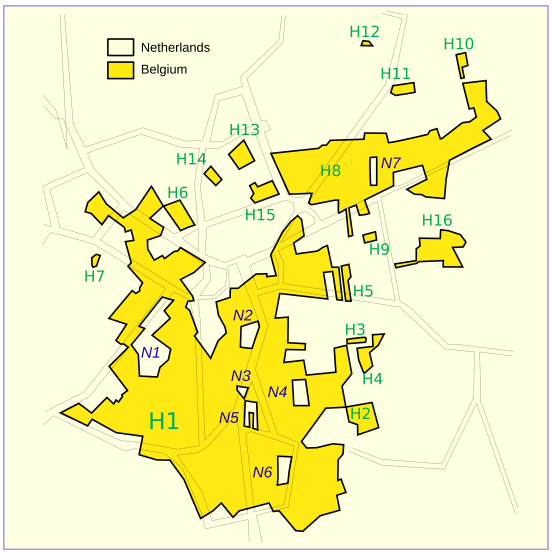


Figure 2.3: Baarle-Nassau and Baarle-Hertog exclaves. Source: [23].

There are several possible ways to represent a map in a computer. We will represent it as a planar graph, with faces bounded by edges and vertices, each face labeled with an identification number (by convention, the map's exterior face is labeled with 0). A face does not need to be a connected set or convex. For example, a face representing Spain would include the exclave (a portion of the territory of a state that is disconnected from the main territory) Llivia. Perhaps the most extreme example are the Baarle-Nassau and Baarle-Hertog exclaves [22]. See Figure 2.3.

We store a *map* in a text file containing a set of chains. A *chain* is a sequence of edges with the same adjacent faces (Grouping edges into chains is done only for efficiency).

Each chain has the following header: $(l, n_e, v_0, v_1, f_l, f_r)$, with l the chain label, n_e the number of edges in the chain, v_0 and v_1 respectively the initial and final vertices, and f_l and f_r the left and right adjacent faces (in other words, f_l and f_r represent, respectively, the faces in the positive and negative sides according to the right hand rule). Each chain header is followed by the $n_e + 1$ coordinates of its vertices.

In a valid map, except for the exterior face, all the faces must be closed (i.e., the map must be *watertight*) and the chain labels must be consistent. Furthermore, two chains can only intersect at their common endpoints (i.e., the map must be free

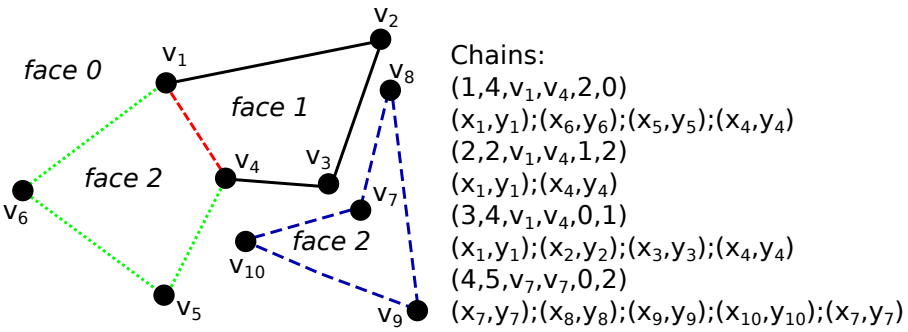


Figure 2.4: Example of map and the chains representing it.

of self-intersections).

Since our algorithm does not need it, faces are not explicitly stored. That is, the topology is represented implicitly. Figure 2.4 presents an example of a map composed of 3 faces (face 2 has two connected components and face 0 is the exterior of the map) and the corresponding chains that represent this map.

Figure 2.5 illustrates two invalid maps (one of them is not watertight and the other one has a self-intersection). In Figure 2.5 (a) point p is apparently in face 1. However, one could trace a curve connecting p to a point in face 0 without crossing any edge and, thus, this would imply that p is in face 0.

In Figure 2.5 (b), if the ray l_1 is traced it will first intersect edge (v_5, v_7), which bounds, respectively, faces 2 and 0 on its positive and negative sides. Thus, this would imply that p is in face 0. On the other hand, if the ray l_2 was traced this would imply that p is in face 2.

Thus, both maps in Figure 2.5 are ambiguous and we consider them to be invalid inputs. As will be mentioned later, the same restrictions are applied to 3D triangular meshes.

2.2.2 3D meshes

In this work, solids will be modeled using a 3D triangular mesh. This representation is similar to the representation of polygonal maps presented in section 2.2.1. I.e., the E^3 space will be partitioned into a finite number of polyhedra bounded by triangular faces.

A mesh M is a set of triangles (a triangle soup) $\{t_i\}$. Each triangle t is a tuple

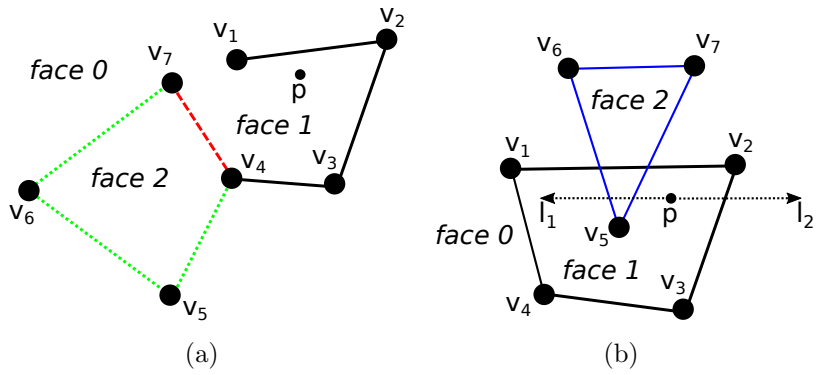


Figure 2.5: Example of problems caused by a non-watertight 2D map (a) and by a self-intersecting map (b): in both situations there is a contradiction related to the face where point p is located.

$(a_0, a_1, a_2, q_+, q_-)$. (a_0, a_1, a_2) is the oriented list of triangle vertices, and (q_+, q_-) are the adjacent polyhedra on the positive and negative sides of t (the order of its vertices in the list gives t a well-defined positive and negative sides). Vertices and polyhedra are labeled with ids.

Similarly to the polygonal map representation, since that information is unnecessary to our algorithms, in 3D we save space by not explicitly storing the polyhedra. Thus, the global topology is represented only implicitly. Not having to worry about global topological properties like face shells and nesting makes many things easier.

Figure 2.6 shows an example of a mesh containing 11 triangles and 3 polyhedra (polyhedron 1 in red, polyhedron 2 in blue, and the exterior polyhedron). Triangle ABC, for example, could be represented as $(A, B, C, 2, 1)$ (polyhedra 2 and 1 are, respectively, on the positive and negative sides of ABC). On the other hand, triangle DCF could be represented as $(D, C, F, 0, 1)$, since it bounds the exterior region (represented, by convention, as polyhedron 0) on its positive side and the polyhedron 1 on its negative side.

Similarly to 2D maps, we require the polyhedra to be closed (*watertight*) and two triangles can only intersect on shared vertices or edges (i.e., they must be free from shelf-intersections). All triangles have a nonzero area and all polyhedra a nonzero volume. (A mesh that violates those conditions can be cleaned up by splitting edges and re-triangulating.)

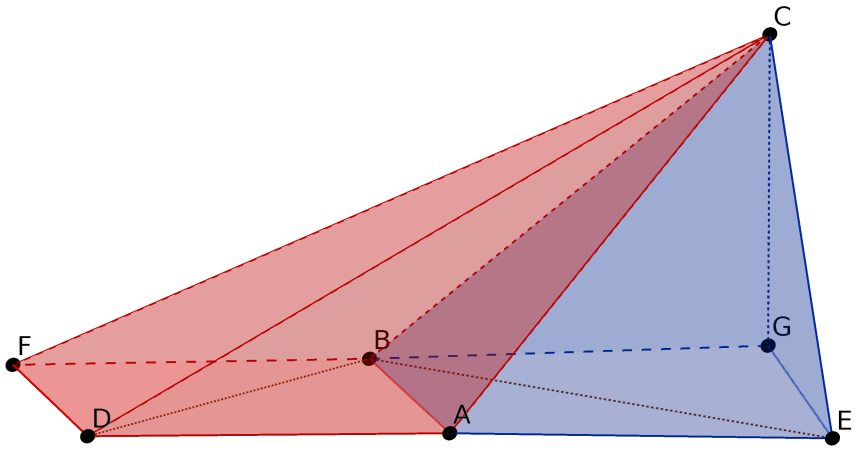


Figure 2.6: Example of mesh representing 2 polyhedra - Triangle ABC bounds the two main polyhedra of the mesh while the other triangles bound one of the two polyhedra and the exterior polyhedron.

2.3 Overlay and point location

2.3.1 The point location problem

Given a set of query points and a polygonal map, the 2D point location problem consists in determining in which polygon each point is. For the simplest case, there is one input polygon and the objective is to determine if the point is inside or outside the polygon. For the more general case, the map contains many polygons and the objective is to determine in which polygon the point is.

This problem extends to 3D meshes: given a mesh representing one or more polyhedra, the objective is to determine in which polyhedron each point is (or if it is in the exterior of all polyhedra).

Both the 2D and 3D versions of this problem have many applications in Computer Graphics, Computer Aided Design, Additive Manufacturing, Computer Games, and Geographic Information Science [24], [25]. Point location is important for solving problems such as computing the intersection of two meshes and detecting collisions. Thus a correct and efficient algorithm is important.

Several existing solutions are presented in Ogayar et al. [24]. The most important ones are Jordan Curve and Feito-Torres. The algorithm based on the Jordan

Curve Theorem [26] is an extension of PNPOLY, the well known ray casting 2D point-in-polygon algorithm [27]. The idea is to cast a semi-infinite ray starting at the query point, and count the number of intersections between this ray and the polygon's edges. The point is considered to be inside the polygon iff the number of intersections is odd. Although this method is simple and efficient for performing single queries, it is a challenge to efficiently implement it to correctly handle all the singularities, which get much worse in 3D. For example, if the ray hits a vertex how many intersections should be counted?

Feito-Torres [28] builds on the ease of determining whether a point is in a tetrahedron by evaluating the sign of a determinant. It partitions the polyhedron into tetrahedra, one between each triangular polyhedron face and the coordinate origin, and then counts how many of them contain the query point q. Iff that is odd, then q is in the polyhedron.

These methods work well for performing single queries against single polyhedra; without preprocessing, a query takes linear time in the data size. A query against a mesh with many polyhedra can be reduced to successive queries against all the polyhedra in turn. However, in the usual cases of many queries on the same dataset, or many polyhedra, this is not optimal. Therefore, Ogayar et al. [24] also extend these two algorithms with common pre-processing techniques (such as the use of an octree to index the mesh) to accelerate the multiple query case.

Recently, Liu et al. presented RCT (*Relative Closest Triangle*) [29], an efficient method for locating points in 3D triangular meshes. RCT can perform point location tests even in models composed of multi-materials, that is, in models where internal boundaries divide the polyhedron into smaller polyhedra and the objective is to determine in which smaller polyhedron the query point is.

For each query point q, RCT uses an octree to efficiently locate a mesh triangle t that is visible from q. t is visible from q if a line segment that does not cross the mesh can be traced between q and t. Once a visible triangle t is found, an orientation operation that evaluates t's normal is used to determine q's position with respect to the polyhedron. To the best of our knowledge, RCT is the fastest algorithm available for performing multiple queries in polyhedra. Indeed, according to the

experiments in Liu et al. [29], RCT was much faster than an efficient ray-casting algorithm based on Axis-Aligned Bounding Boxes, and also than the CGAL [21] point location algorithm.

In Chapter 5 we will present PinMesh, an exact and efficient algorithm for performing point location queries in 3D triangular meshes, which is faster than RCT.

2.3.2 The overlay problem

2.3.2.1 2D map overlay

Given a pair of polygonal maps A and B, the overlay (or intersection) of A with B is a map C where each polygon is the intersection of a polygon of A with a polygon of B. This is a classical problem with many applications in Computational Geometry, Computational Cartography and Geographic Information Science (GIS) [30]–[34].

Figure 2.7 illustrates the overlay of two maps (the black map contains two polygons: polygon 1 and polygon 2, and the blue map contains the polygons 3 and 4). Each map also contains an exterior polygon (by convention, the exterior is labeled with 0). In Figure 2.7 (a) the two maps are superimposed, and in Figure 2.7 (b) the overlay of them is computed.

In this thesis we consider a variation of the overlay where the output polygons intersecting the exterior of one of the input maps are removed from the output after the overlay is computed (this is illustrated in Figure 2.7 (c)). While both problems are equally hard to solve (the polygons are removed in a simple classification step), we decided to choose this variation in order to compare our algorithm with other algorithms that deal with the same variation.

Sometimes, the special case of *triangulation overlay* is considered [35] (an excellent survey of various surface representations and algorithms). Map overlay can also be embedded into higher-level algorithms such as interpolating from known county populations to estimated watershed populations [33], [36].

Franklin [33] presented an efficient algorithm that uses local topological formulae to compute the area of the overlay. This algorithm can be used in situations where the application does not need to explicitly compute the actual overlaid poly-

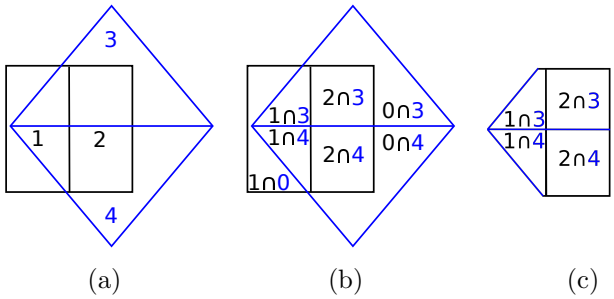


Figure 2.7: Example of map overlay: (a) two superimposed maps, (b) overlay, (c) overlay without the polygons intersecting the exterior of the maps.

gons but only needs to know the area of these polygons. An example of such an application is to estimate the population of some regions, which can be done by computing the area of the overlay of a map containing the polygons of these regions with another map containing polygons representing different population densities. Other similar works using minimal local topology for polygons and polyhedra were presented in [37]–[41].

A common group of algorithms are the ones based on the plane sweep paradigm [32], [42]–[44] but, as stated by Audet et al. [45]: "the plane sweep strategy does not parallelize efficiently, rendering it incapable of benefiting from recent trends of multicore CPUs and general-purpose GPUs".

Other map overlay algorithms use a special data structure, such as an R-Tree [30], [46], QuadTree [47], [48], or Uniform Grid [33]. The basic idea is a spatial sort to reduce the number of pairs of edges to be tested for intersection, since only segments that are in the same cell of a data structure need to be tested against each other.

Geometric Performance Primitives (GPP), the commercial product described in [45] also uses a uniform grid to compute map overlays in parallel . However it computes with floats, and must ameliorate the resulting roundoff errors with snap rounding, which can change the maps' topology.

Other early presentations of using the uniform grid index in computational geometry and GIS include [49]–[51]. Earlier parallel computational geometry research with the uniform grid includes [52]–[56]. Wu[57], [58] used Prolog for overlaying polygons.

2.3.2.2 3D mesh overlay

Similarly to the map overlay problem, the computation of mesh intersections has applications in fields such as Computational Geometry, GIS and CAD systems.

A common technique for overlaying 3D polyhedra is to convert the data to a volumetric representation (this process is known as voxelization), and then perform the overlay using the converted data. This approach has several advantages: first, the volumetric model can be created using any precision, and thus, if the application does not demand a high precision, this algorithm can be used to compute a fast approximation of the overlay. Furthermore, it is trivial to perform a robust overlay of volumetric representations. The drawback of this approach is that the volumetric representation is usually not exact, and thus the overlay results are usually an approximation. Pavić et al. [59] present an efficient algorithm for performing this kind of overlay.

For computing the non-approximate overlay, a common strategy is to use indexing to accelerate spatial operations performed during the overlay process (such as computing the triangle-triangle intersection). For example, Franklin [60] uses a uniform grid to intersect two polyhedra, Feito et al. [3] and Mei et al. [61] use octrees and Jing et al. [62] use Oriented Bounding Boxes trees (OBBs) to intersect triangulations.

While the authors of these algorithms consider them exact in the sense that they do not use approximation techniques such as voxels, robustness cannot be always guaranteed due to floating point errors. For example, the algorithm presented in [3] uses a tolerance to process floating-point numbers, which as was mentioned in Section 2.1 is not guaranteed to always work.

Another example of algorithm that does not guarantee robustness is QuickCSG [63], a method that was specifically designed to be extremely efficient. QuickCSG employs parallel programming and a kd-tree index to accelerate the computation. However, it does not handle special cases (it assumes vertices are in general position), and does not handle the numerical non-robustness associated with floating-point arithmetic [64]. In order to try to reduce errors caused by special cases, QuickCSG allows the user to apply random numerical perturbations to the input (however, this

does not guarantee robustness).

Even if an algorithm using floating-point arithmetic can intersect two specific meshes consistently (i.e., without creating topological impossibilities or crashing), the resulting mesh may not correctly represent the actual intersection since the coordinates of the vertices created after the intersection may not be exactly representable as floating-point numbers. For example, even though the vertices of the segment connecting $(0, -1)$ to $(1, 2)$ can be exactly represented using floating point numbers, the intersection of this segment with the line $y = 0$ (point $(\frac{1}{3}, 0)$) cannot be represented.

Even though in some applications small errors in the position of the resulting vertices may be acceptable, these errors may accumulate if several inexact operations are performed in sequence. The problem of accumulating errors is even worse in CAD and GIS systems because, in these applications, it is common to perform a sequence of many operations to combine and transform the data sets.

Since in many applications it is important to have exact algorithms, Hachenberger [65] presented an algorithm for computing the exact intersection of Nef polyhedra. The basic idea of a Nef polyhedron is to represent the polyhedron as a finite sequence of complement and intersection operations on half-spaces [65]. This kind of representation has some important features. For example, Nef polyhedra can represent open and closed objects, are closed for intersection and other set operations.

Even though Nef polyhedra have been known since the 1970's, only in the 2000's were concrete algorithms and data structures for representing and processing these kind of geometric data presented [65]. Because of their importance, the algorithms proposed by Hachenberger were implemented in the CGAL library[66]. An example of an application of CGAL exact geometry in GIS is the SFCGAL [67] backend of the PostGIS DBMS. SFCGAL wraps the CGAL exact representation for 2D and 3D data, allowing PostGIS to perform exact geometric computations.

According to Leconte et al. [68], even though these algorithms are exact they have some limitations such as poor performance, which limits the applications that can benefit from their exactness. Besides the performance problems, another limitation is that geometric data are more commonly stored using other representations

such as triangular meshes than Nef Polyhedra.

Bernstein et al.[69] also presented an algorithm that tries to achieve robustness in mesh intersection. Their basic idea is to represent the polyhedra using binary space partitioning (*BSP*) trees with fixed-precision coordinates. The algorithm proposed by Bernstein et al. can intersect two polyhedra represented using a *BSP* by only evaluating predicates (which can be done using fixed-precision arithmetic). As the authors mention, the main limitation is that the process to convert BSPs to widely used representations (such as triangular meshes) is slow and, more important, inexact.

Recently Zhou et al.[64] presented an exact and parallel algorithm for performing booleans on meshes. The key of their algorithm 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, and as 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.

This algorithm is freely available and distributed in the LibiGL [70] package (in this thesis we will refer to this algorithm as LibiGL). Its implementation employs exact predicates provided by CGAL. Furthermore, the triangle-triangle intersection computation is accelerated using CGAL's bounding-box-based spatial index.

As shown by the authors [64], LibiGL is not only exact, but also much faster than CGAL algorithm for Nef Polyhedra (however, it is still slower than fast inexact algorithms such as QuickCSG).

As will be shown in Chapter 6, the mesh intersection algorithm presented in this thesis was able to outperform LibiGL (it was up to 101 times faster than LibiGL) and, in general, it also outperformed QuickCSG.

2.4 Simulation of Simplicity

Another common source of error in geometric algorithms is special cases (geometric degeneracies). Algorithms are usually described considering they will process non-degenerate input. However, during the actual implementation, degenerate data have to be considered. Degeneracies increase the number of cases that must be considered. E.g., when comparing point q against line l, there are now three cases: q may be (above / on / below) l instead of only two (above / below). By itself, this would not be bad, except that this predicate may be a component of a larger one. Perhaps q is a vertex of a piecewise straight line m with vertices qpr, and we wish to know how m intersects l. There are now more special cases. Next, consider the intersection of the piecewise straight line with vertices $a_0a_1a_2$ with the piecewise straight line with vertices $b_0b_1b_2$. Perhaps $a_1 = b_1$, or a_0a_1 is collinear with b_1b_2, and so on...

As mentioned by Yap [71], "sometimes, even careful attempts at capturing all degenerate cases leave hard-to-detect gaps". Properly handling special cases is a challenge mainly in geometric problems such as the mesh intersection, that depend on several subproblems, each one with its own special cases.

To correctly handle special cases in the algorithms we developed, we employed Simulation of Simplicity (SoS) [4]. This is a general purpose symbolic perturbation technique designed to treat special (degenerate) cases, or geometric coincidences in the data. 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 brilliant solution that uses a symbolic perturbation by a formal indeterminate infinitesimal value, ϵ, or by ϵ^i, for some natural number i. The mathematical formalization of SoS extends some exactly computable field, such as exact reals or rationals, by adding orders of infinitesimals, ϵ^i. Floating point numbers with roundoff errors cannot be the base. Indeed, floats are not even a field because roundoff errors cause most of the field axioms to be violated. E.g., because $(10^{-20}+1)$ rounds to 1, so $(10^{-20}+1)-1 \neq 10^{-20}+(1-1)$.

The infinitesimal ϵ is an *indeterminate*. It has no meaning apart from the rules for how it combines. E.g., if a is a positive finite number, then two of those rules are that $a + \epsilon$ is equivalent to a and $\epsilon < a$. For a charming take on this, see [72].

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 algebra field. It is attractive to think of ϵ as an actual very small finite number, perhaps 10^{-10} or 10^{-100}. Although it may be useful to develop an initial intuitive understanding, the use of a concrete value for the infinitesimals could lead to wrong conclusions.

The result of SoS is that degeneracies are resolved in a way that is globally consistent. For example, consider Figure 2.8: two identical squares (*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 square 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.

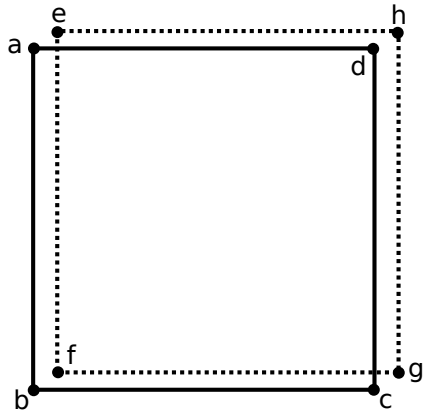


Figure 2.8: Testing edges for intersection after the application of SoS.

To see how hard it would be to explicitly treat degeneracies without a logical framework such as SoS, consider the problem of intersecting two edges or line segments, e_1 and e_2. The relevant degeneracies occur when the end vertex of one lies on the other, or when two vertices coincide. With SoS, the lines are symbolically

perturbed so that this does not happen. Considering the degeneracies explicitly is difficult because there are many more cases to get correct. Consider the problem of intersecting two polylines p_1 and p_2. Without degeneracies, p_1 and p_2 cross if and only if their constituent edges intersect an odd number of times. With degeneracies, every incidence type of a vertex or edge with another vertex or edge must be considered. SoS simplifies the problem considerably.

For another example, consider the 2D point-in-polygon test PNPOLY [27], that checks if a point is a polygon by casting a semi-infinite ray r and counting the number of intersections between r and the polygon's boundary. The simple case of the ray intersecting a vertex is easy; the whole PNPOLY function has only 8 lines of executable C code. But now consider the problem of tracing a vertical ray in a polygonal map to find an edge that bounds the polygon containing a query point. This problem could be solved by locating the lowest intersection point between the ray and the polygons' boundaries and, then, returning the edge containing that point. Consider Figure 2.9: the vertical ray traced from q_1 hits the map's boundary first on a point in edge GC. Thus, it is clear that this edge bounds the polygon containing q_1 (Polygon 2). However, there are special cases that need to be handled: the query point may be on the boundary of a polygon or the lowest intersection point may be on a vertex. If the lowest intersection point is on a vertex v, some of the edges ending on v may not even bound the polygon containing q. For example, the ray traced from q_3 hits a vertex of edge EH (vertex E) and EH does not even bound the polygon containing q_3. Thus, the algorithm solving this problem needs to carefully handle theses special cases. This version of this problem for polyhedral meshes would be even harder to solve correctly.

To use SoS for perturbing the dataset, it is important to present a suitable perturbation scheme. Edelsbrunner and Mücke [4] presented a perturbation scheme to avoid degeneracies in the point orientation problem in E^d. In this problem, the geometric objects are points and the scheme added a different infinitesimal perturbation (more specifically, the j-th coordinate of the i-th point is translated by $\epsilon^{2^{id-j}}$) to each coordinate of each point. By doing that, all the perturbed points are in general position (in E^3, this means that no set of 4 perturbed points can be

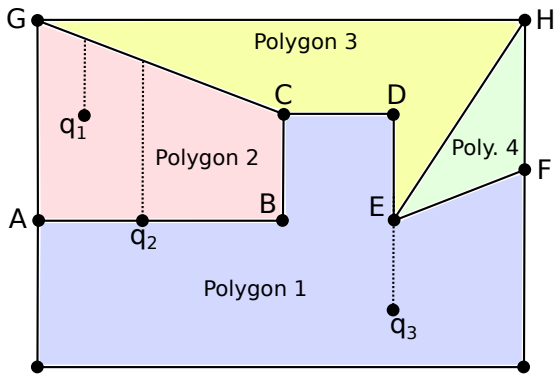


Figure 2.9: Example of special cases during the process of finding an edge bounding a polygon.

co-planar) and, therefore, the input is non-degenerate. As shown by the authors, this perturbation choice satisfies 3 important conditions:

1. The perturbed geometric objects are simple (non-degenerate) if $\epsilon > 0$ is sufficiently small.

2. If an object is non-degenerate, then its perturbed version retains the properties of its original version.

3. The computational overhead of processing the perturbed objects is negligible.

Since these conditions are satisfied, the orientation algorithm implemented to perform the computation using the perturbed points will correctly (and efficiently) handle all the special cases.

Implementing SoS and computing with orders of infinitesimals would seem to be very slow. However, 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 the exact computation with rationals, would have caused a tie in a predicate, e.g., make a determinant to be zero. 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.

CHAPTER 3
Processing 2D maps

In this chapter, the algorithms for processing 2D maps will be presented. Even though the main objective of this thesis is to develop an exact method for processing 3D triangular meshes, we also developed methods for processing polygonal maps to evaluate the techniques we intended to apply to the 3D algorithms.

Initially, the two-level uniform grid, the indexing data structure we used, will be described. Then, EPUG-Overlay, our exact map overlay algorithm will be presented and evaluated. EPUG-Overlay is novel because it combines the techniques for exactness and efficiency mentioned in Chapter 2: exact arithmetic and Simulation of Simplicity are used for ensuring robustness, and a two-level parallel uniform grid and parallel programming are employed for accelerating the computation. Furthermore, the overlay is performed using only simple local information about the individual edges representing the map.

Besides being exact, EPUG-Overlay is also fast. The largest test case for EPUG-Overlay (using big rationals) overlaid two maps: one having 32 million edges and 220 thousand faces with another map having 21 million edges and 519 thousand faces in 264 elapsed seconds using 16 threads with dual 8-core 3.1 GHz Intel Xeon E5-2687 processors. That is a factor of 5 speedup compared to using one thread. By comparison, GRASS (using roundoff-error-prone floats), took $5,300$ seconds, partly because it is only single-threaded. (However, even when only one thread was used EPUG-Overlay was still faster than GRASS.)

Portions of this chapter previously appeared as:

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li, "Fast exact parallel map overlay using a two-level uniform grid", in *in Proc. 4th ACM SIGSPATIAL Int. Workshop Analytics for Big Geospatial Data*, BigSpatial'15. New York, NY, USA: ACM, 2015.

3.1 Two-level uniform grid

Franklin et al [73] proposed a uniform grid to accelerate his algorithm for computing the area of overlaid polygons. The basic idea for creating an index based on a uniform grid is to superimpose a regular grid over the input maps and determine, for each edge, the set of incident grid cells. Then, pointers to the edges are inserted into the corresponding cells. In our implementation we used a C++ vector for each cell to record the edges incident on it. (Another implementation choice would be a ragged array.)

The uniform grid will be employed to accelerate two steps of the map overlay process: the point location and the computation of the edge-edge intersections. Each grid cell c contains the edges from both maps intersecting c. Thus, the edges intersections (for example) can be found by processing each cell and comparing the edges in that cell pair-by-pair (one edge from each map) to determine which pairs intersect and to compute the intersection points.

The uniform grid works well even for uneven data for various reasons [33], [74]–[76]. First, the total time is the sum of one component (inserting edges into cells) that runs slower with a finer grid, plus other components (for example, intersecting edges in cells) that run faster. The sum varies only slowly with changing grid size. Second, an empty grid cell is very inexpensive, so that sizing the grid for the dense part of the data works.

Nevertheless, to process very uneven data, in this project we have incorporated a second level grid into those few cells that are densely populated with edges. 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 in each cell the edges from one map are tested for intersection with edges from the other map, 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.

Figure 3.1 presents an example with two maps superposing this *2-level uniform grid*. In this example, the first level has 4×7 cells and the second level (created in those first level cells having more than 4 edges) was created with 3×3 cells.

This nesting could be recursively repeated until all grid cells have fewer el-

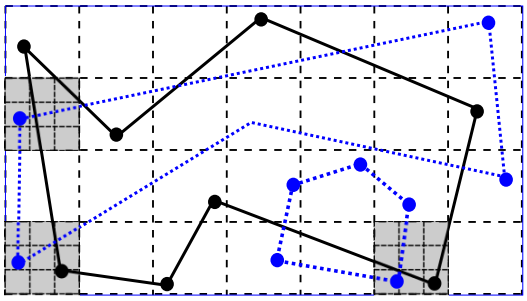


Figure 3.1: Two-level uniform grid - Uniform grid with 4×7 first level and 3×3 second level (created in first level cells containing more than 4 edges).

ements than a given threshold, creating a structure similar to quadtree, although with more branching at each level. Our solution could be considered a special case of that. However, the general solution uses more space for pointers (or is expensive to modify) and is irregular enough that parallelization is difficult. Also, for map overlay, our tests have shown that the best performance is achieved using just a second level. 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.

3.1.1 Implementation details

The uniform grid can be easily constructed. Initially, the edges from both maps are processed to determine in what grid cells they must be inserted. Determining which grid cells contain each edge (i.e., rasterizing the edges) can be parallelized over the edges, although inserting them into the grid structure must be serialized, e.g. with an atomic increment-and-copy operation on the count of edges in the cell.

After the edges are inserted into the corresponding grid cells, those cells that need to be refined (for example, the ones with more than, say, 50 edges) are processed. For each of those cells c, a second level uniform grid is created in c and the edges in c are again rasterized and inserted into the corresponding second-level grid cells. Since the edges in a first-level cell do not interact with other cells, the refinement process can be easily parallelized by refining the cells in parallel.

3.2 Overlaying 2D planar graphs

We represent a map as a planar graph as described in section 2.2.1. The faces are bounded by edges and vertices, and each face is labeled with an identification number.

Since our algorithm does not need it, faces are not explicitly stored. That is, the topology is represented implicitly basing on the adjacency information that is stored in the edges. For performance, edges with the same adjacent faces are grouped in *chains*.

Given two maps $\mathcal{A}$ and $\mathcal{B}$, the objective is to create a resulting map where each face represents the intersection of a face of $\mathcal{A}$ with a face of $\mathcal{B}$.

3.2.1 The algorithm

In EPUG-Overlay all the coordinates are represented using rational numbers managed by the GMP (*GNU Multiple Precision Arithmetic Library*) library [77]. EPUG-Overlay reads the two input maps and the vertices coordinates are converted to rationals using the GMP (overloaded) operand. This operand automatically uses the required precision to represent the number exactly. (Another strategy to convert float to a rational is to use a continued fraction to find the simplest rational within some given tolerance, such as one half the least significant bit.)

Algorithm 3.1 summarizes the overlay process; the next sections give details.

Algorithm 3.1: Computes the overlay of two maps A and B given as input.

1: Create the 2-level uniform grid
2: Compute the intersection points between all edges of maps A and B
3: Locate all vertices of map A in map B and vice-versa
4: Create the resulting map

Our reported times are elapsed times, that is wall clock times. For parallel computation, reporting total CPU time is meaningless because it does not capture the parallelization's effectiveness. That is, does an algorithm taking 1 CPU-minute, run sequentially in 1 elapsed minute, or does it run on 16 parallel threads, taking 4 seconds on each, finishing in 4 elapsed seconds? Elapsed time is not susceptible to

that ambiguity. This also avoids technical difficulties with reliably measuring CPU times for parallel threads.

3.2.1.1 Exploiting local topology

When computing a map $\mathcal{C}$ corresponding to the overlay of maps $\mathcal{A}$ and $\mathcal{B}$, each of whose faces is the intersection of a pair of faces from the two maps, the obvious solution is to intersect each face from map $\mathcal{A}$ with each face from $\mathcal{B}$, and report the non-empty intersections. We do not do that.

Instead, we exploit the fact that a face's boundary is a set of edges, and look for edge intersections. 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.

3.2.1.2 Creating the two-level uniform grid

The uniform grid is constructed in parallel as described in section 3.1. Since in the overlay problem the two input maps need to be indexed, edges from the two maps are inserted simultaneously into the grid.

Of course, an alternative is to create two separate grids to index each map separately. However, having only one grid to index the two maps simultaneously simplifies the overlay process. For example, to detect the intersections between the edges of the two maps, for each grid cell c, the edges from one map in c are tested for intersection with the edges from the other map also in c. If two grids were used, for each cell c of one grid, the edges in c would need to be tested for intersection with edges from all the cells from the other grid that intersect c.

Choosing the resolution of a uniform grid gives the user an opportunity to trade off speed and size. Section 3.3 presents some experiments showing this tradeoff. The exact grid size is not too important for the time because varying it a factor of two in either direction from the optimum often increases the time by much less than 50%. For example, in our largest experiment (presented in Table 3.4) if the resolution of the uniform grid was reduced from 4000^2 cells to 2000^2 (that is, if the total

number of cells was reduced by a factor of 4) the elapsed running time (excluding I/O) increased in only 16% (if we consider the I/O the difference in the time is even smaller since I/O time does not change). Therefore, we used a conservative empirical formula for the grid size that gave a good execution time and a reasonable memory usage.

As it will be explained in section 3.2.1.3, the edge-edge intersection computation (one of the most important steps of the overlay process) is performed by intersecting, for each cell c, all the edges from one of the maps in c with the edges from the second map in c. Therefore, the running time to compute the edge-edge intersection in each cell c is expected to be proportional to the number of pairs of edges (composed of edges from the two maps) in c.

The time spent locating vertices of one map in the other one is also related to the number of pairs of edges in the cells: as it will be explained later, to locate the vertices from the map $\mathcal{A}$ in the map $\mathcal{B}$ (and vice-versa), for each vertex of $\mathcal{A}$ in a grid cell c (the number of vertices in a cell is proportional to the number of edges) a vertical ray is traced and this ray is tested for intersection with the edges of the map $\mathcal{B}$ in c.

Thus, the total number of pairs of edges in the uniform grid is an important factor in the running-time of EPUG-Overlay. Therefore, we decided to refine the cells where this product is greater than or equal to a given threshold. In these cells we add a 40×40 second level uniform grid (the exact size is not critical) — more details are in Section 3.3.

3.2.1.3 Computing the intersection points

The next step is a parallel iteration over all the grid cells. In each cell, we test each edge of map $\mathcal{A}$ in the cell against each edge of $\mathcal{B}$ in that cell. This process is extremely parallelizable since the cells do not influence each other.

Degenerate cases are handled with *Simulation of Simplicity (SoS)* [4]. 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 that is smaller than any positive real number. The second order infinitesimal ϵ^2 is smaller than any positive finite multiple of the first order infinitesimal ϵ. As mentioned in section 2.4, such a number system can be axiomatized and is consistent [72]. 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 $\mathcal{B}$, neither is any point in $\mathcal{A}$ on any edge in $\mathcal{B}$, nor do two any edges coincide.

As mentioned before, using SoS to resolve degeneracies is a solution that generalizes up. If we use it to test whether two edges intersect, we can utilize that function in a test of whether two chains intersect, and get topologically valid results. E.g., if two chains cross at a common vertex, we will get a total of either one or three edge-edge intersections, even though our edge intersection function knows nothing about chains.

3.2.1.4 Locating one map's vertices in the other

The third step is to determine which face of map $\mathcal{A}$ contains each vertex of map $\mathcal{B}$ and vice-versa. When a vertex is on an edge, SoS puts it into exactly one of the two adjacent faces in a way that later will produce a consistent answer.

The idea is to run a semi-infinite ray up from vertex v of $\mathcal{A}$, to find the lowest edge e of $\mathcal{B}$ that it hits. Then, v is in one of the adjacent faces of e, which one depending on whether the ray hits e from the left or the right. Because of SoS, if e is vertical then no ray will ever hit it.

The expected time to locate a point in a map using our grid is *constant*, independent of the map's size.

Assume that the grid is sized so that the expected number of edges per cell is constant. Determining which cell $\mathbb{c}$ contains v takes constant time. Testing the ray against all the edges in $\mathbb{c}$ takes constant time. If the ray hits at least one edge, then we know the face. However there is a probability p that it does not. Because the expected number of edges in $\mathbb{c}$ is constant, so also p is a constant independent

of the map size. Then we continue the ray into the next higher cell and test for an intersecting edge. The expected number of cells to process until we find an intersecting edge is $1/(1-p)$. If we fall off the top of the grid without finding an intersecting edge, then v is inside $\mathcal{B}$'s exterior face, i.e., face 0.

For example, in Figure 3.2, we have to follow the ray up through three cells until it hits edge $e = (u, w)$.

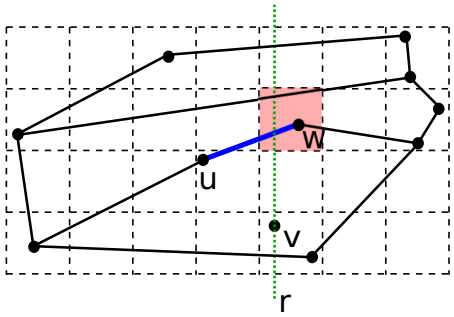


Figure 3.2: Determining the face containing a vertex.

If a grid cell does not contain any edge from one of the maps, then this cell is completely inside a face f of this map (or in the exterior face) and, because of the Jordan Curve Theorem, any vertex in this cell will automatically be in f. Thus, to accelerate even more the point location process, an empty grid cell is labeled with the identification number of the face containing it.

Furthermore, when a point v is queried if the vertical ray hits an empty grid cell c before hitting any edge, a continuous path connecting v to the cell c can be traced along this ray and this path will not intersect any edge. Thus, it is clear that , because of the Jordan Curve Theorem, v will be in the same face where c is. Therefore, c's label can be used to locate v. In the example presented in Figure 3.2, the ray r hits an empty cell just before hitting the detached cell containing $u - w$ and, therefore, the point location algorithm could have returned the label associated with that empty cell and avoided the processing of the cell above it.

To perform the cell labeling process, for each empty cell c the point location algorithm presented in this section is applied to determine the face f where the center point of c is located and, then, c is labeled with f. If a grid cell c' adjacent to c is empty, c' can also be labeled with f since a continuous path connecting c to

c' can be traced without crossing any edge. Thus, whenever an empty grid cell is found its entire connected component of empty cells is labeled with the information about the face where the cell is.

3.2.2 Constructing the resulting map

It is possible that edge e of map $\mathcal{A}$ does not intersect any edge of map $\mathcal{B}$. Then e is completely inside one face (perhaps the exterior face) of $\mathcal{B}$. It is even possible that no edge of $\mathcal{A}$ intersects any edge of $\mathcal{B}$.

If e is inside a face that is not the exterior face (i.e. face 0), e will be an edge of the resulting map. Otherwise, if e is exterior (i.e. inside the face 0), e will not be an edge of the resulting map. For example, in Figure 3.3, the edge e from map $\mathcal{A}$ (represented by dotted lines) is inside face $\mathcal{B}_1$ (the face 1 of map $\mathcal{B}$ represented by solid lines). Thus, e will be an edge of the resulting map having as adjacent faces: one face resulting from the intersection between faces $\mathcal{A}_1$ and $\mathcal{B}_1$ and the exterior face resulting from the intersection of faces $\mathcal{B}_1$ and $\mathcal{A}_0$ (the exterior face of map $\mathcal{A}$). On the other hand, the edge f will not be in the resulting map since it is outside the map $\mathcal{A}$ (it is inside face $\mathcal{A}_0$).

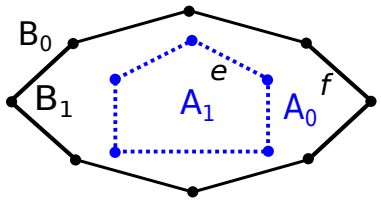


Figure 3.3: Two maps with no edge intersection.

Furthermore, an edge can intersect one (or more) edges of the other map. Let $e = (u, w)$ be an intersecting edge and suppose that e intersects $k \geq 1$ edges of the other map and that the intersection points are $i_1, i_2, ..., i_k$. Then, e is subdivided into $k + 1$ non-intersecting edges $e_1 = (u, i_1), e_2 = (i_1, i_2), e_3 = (i_2, i_3), ...e_k = (i_{k-1}, i_k), e_{k+1} = (i_k, w)$. To determine which of these (new) edges will be included in the resulting map, the midpoint m_j of each edge e_j is analyzed: if m_j is outside the other map, e_j will not be in the resulting map. Otherwise, if m_j is inside a face g of the other map, e_j will be in the resulting map since the faces in both sides of e_j will intersect with g.

Figure 3.4 presents an example of an edge $e = (u, w)$ from map $\mathcal{A}$ (represented by dotted line) that intersects more than one edge from map $\mathcal{B}$ (solid lines). In this case, the new edge $e_3 = (i_2, i_3)$ has midpoint m_3 that is inside face $\mathcal{B}_2$. Since the face in the right and left sides of (original edge) e were, respectively, $\mathcal{A}_1$ and $\mathcal{A}_0$, the face in the right side of e_3 will be the new face obtained from the intersection of face $\mathcal{A}_1$ with $\mathcal{B}_2$ and the face in the left side of e_3 will be the exterior face that results from the intersection $\mathcal{A}_0$ with $\mathcal{B}_2$. The edge $e_5 = (i_4, w)$, on the other hand, will not be in the resulting map since its midpoint m_5 is outside map $\mathcal{B}$.

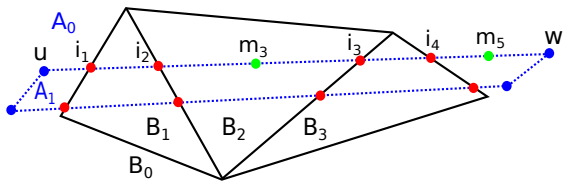


Figure 3.4: Example of an edge of map $\mathcal{A}$ intersecting more than one edge of map $\mathcal{B}$.

3.2.2.1 Parallel implementation

We implemented EPUG-Overlay in parallel using a shared-memory architecture with the OpenMP API. The edges are processed in parallel to determine the cells incident on each edge. Each thread employs a private array to accumulate its results, and at the end a critical section is employed to merge them.

The next step, edge intersection is parallelized over the grid cells as mentioned in section 3.2.1.3.

We also parallelize locating the vertices of each map in the other map's faces. First, to initialize the empty cell labels the uniform grid was divided into smaller blocks and the labeling process was performed in each block separately. Second, when the vertices of one map are located in the other map's faces this process is parallelized over the vertices.

Finally, the output faces are computed by processing the output edges in parallel. Since these three steps were designed to be processed without data dependency, they can be easily parallelized.

3.3 Experimental results

We implemented EPUG-Overlay in C++ and OpenMP using the multiple precision arithmetic package GMP[77]. It was compiled with g++ 4.8.2 and tested on a workstation with dual Intel Xeon E5-2687 processors, each with 8 cores. The workstation has 128 GiB of RAM memory and runs the Linux 3.16 Mint 17 operating system.

The tests were performed using six datasets: two from Brazil, distributed by IBGE (the Brazilian geography agency) and four from the United States. Figure 3.5 illustrates some of these datasets. They were obtained from the ArcGIS, United States Census and National Atlas web-pages:

- *BrSoil*: kinds of soils in Brazil.
- *BrCounty*: Brazilian counties.
- *UsAquifers*: US aquifers.
- *UsCounty*: US counties.
- *UsWaterBodies*: the surface drainage system of the United States.
- *UsBlockBoundaries*: 2010 United States Census block groups.

Table 3.1 gives statistics about the maps. After conversion to the representation described in section 2.2.1 we did these overlay tests:

- *BrSoil* with *BrCounty*,
- *UsAquifers* with *UsCounty*, and
- *UsWaterBodies* with *UsBlockBoundaries*.

Table 3.1: Experimental datasets.

Dataset	# Vertices	# Edges	# Polygons	Size (GB)
BrSoil	248,493	251,011	2,959	0.03
BrCounty	320,621	326,193	5,567	0.04
UsAquifers	351,813	352,924	3,552	0.04
UsCounty	3,633,226	3,636,347	3,110	0.37
UsWaterBodies	21,014,498	21,060,354	518,837	2.25
UsBlockBoundaries	31,875,031	32,103,306	219,831	3.40

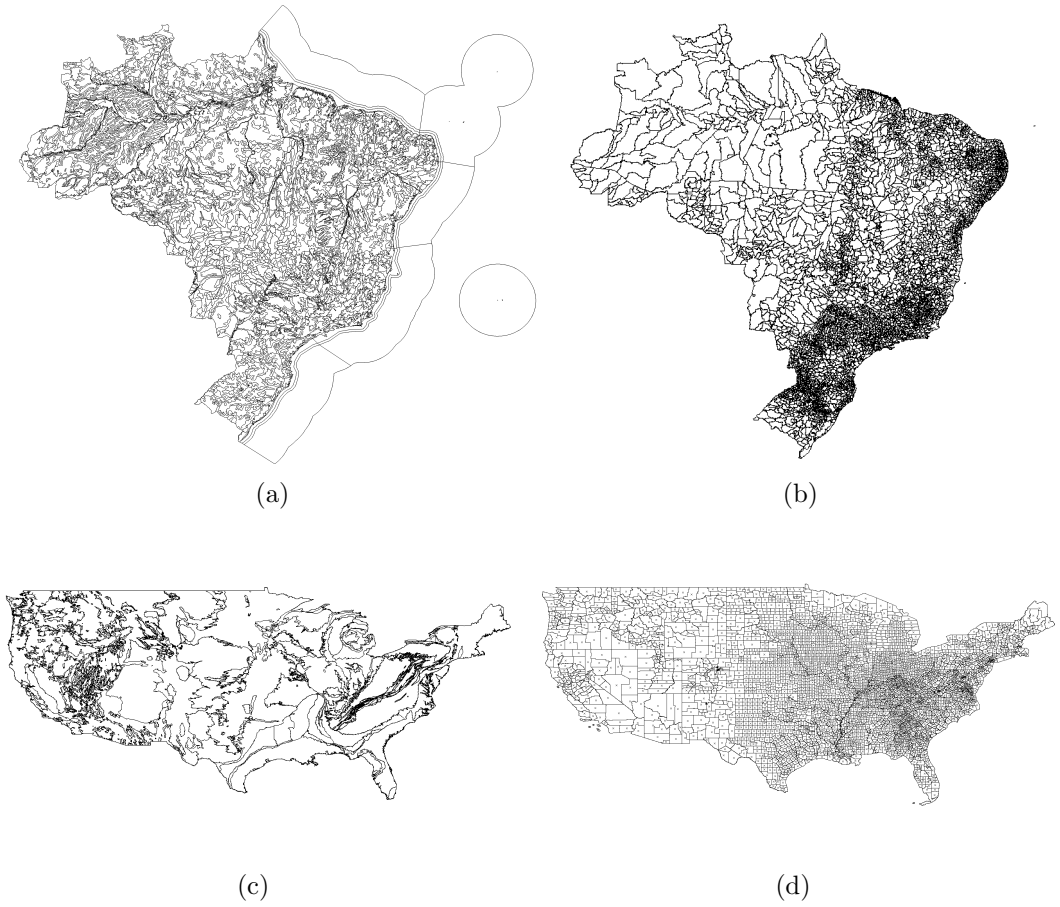


Figure 3.5: Examples of maps used in the overlay experiments - BrSoil (a), BrCounty (b), UsAquifers (c), UsCounty (d) (these figures are not to scale).

3.3.1 Algorithm performance

First we tested the effect of different grid sizes and collected some statistics during this processing. The threshold for refining a first level cell was 50 pairs of edges, that is, if the product of the number of edges from the first map with the number of edges of the second map in a cell c is greater than 50 then c was refined creating a second level grid. In all experiments the times represent the average of 3 executions.

Tables 3.2, 3.3 and 3.4 present the results of this experiments for the three pairs of maps evaluated in the experiments. These tables present the elapsed times (in seconds) spent by each category of processing performed by EPUG-Overlay using 16 threads: column *Grid.* represents the time spent to create and refine

the uniform grid; column *Inte.* presents the time spent intersecting edges; *Loc.* represents the time spent labeling empty grid cells, locating points and creating the output map; *Total* represents the total elapsed time. The more detailed timing data will be presented in another table later.

Table 3.2: Elapsed time (in seconds) for the overlay of the BrSoil dataset with BrCounty.

BrSoil × *BrCounty*									
Grid size		Time (s)				Mem.	# inter.	% cells	% cells
1^{st}	2^{nd}	Grid.[a]	Inte.[b]	Loc.[c]	Total	(GB)	tests[d]$\times 10^6$	refined[e]	labeled[f]
	25^2	0.90	0.13	0.37	1.40	0.3	0.21	28.80	84.52
100^2	40^2	1.02	0.10	0.46	1.58	0.5	0.14	28.80	90.06
	55^2	1.18	0.10	0.60	1.88	0.8	0.11	28.80	92.68
	25^2	1.00	0.10	0.46	1.56	0.5	0.14	20.37	90.18
200^2	40^2	1.22	0.10	0.76	2.08	1.1	0.11	20.37	93.77
	55^2	1.48	0.12	1.17	2.77	1.9	0.10	20.37	95.44
	25^2	1.10	0.12	0.59	1.81	0.7	0.17	13.09	91.69
300^2	40^2	1.44	0.14	1.11	2.69	1.6	0.15	13.09	94.76
	55^2	1.54	0.15	2.02	3.71	2.7	0.15	13.09	96.18
	25^2	1.14	0.18	0.71	2.03	0.8	0.26	7.75	92.19
400^2	40^2	1.35	0.20	1.22	2.77	1.7	0.25	7.75	95.09
	55^2	1.53	0.26	1.93	3.72	2.8	0.25	7.75	96.43
	25^2	1.12	0.27	0.76	2.15	0.7	0.37	4.28	92.25
500^2	40^2	1.32	0.28	1.38	2.98	1.5	0.36	4.28	95.15
	55^2	1.46	0.41	1.82	3.69	2.4	0.36	4.28	96.48

Elapsed time in seconds (excluding I/O) for the main steps of EPUG-Overlay and other statistics using different grid sizes for the first and second level;
[a] time spent creating and refining the uniform grid; [b] time spent intersecting edges;
[c] time spent labeling empty grid cells, locating points and creating the output map;
[d] total number of edge-edge intersection tests performed; [e] percentage of first level grid cells that were refined; [f] percentage of grid cells that were labeled.

As it can be seen, in all the experiments the time to create the uniform grid depends mainly on the size of the input maps. Indeed, in the smallest datasets the total grid creation time ranged from $1.6\mu s$ to $2.7\mu s$ per input edge (with average $2.2\mu s$ per edge) while in the *UsAquifers* × *UsCounty* experiment it ranged from $1.5\mu s$ to $1.6\mu s$ (with average $1.5\mu s$) and in the largest dataset it ranged from $1.4\mu s$ to $1.5\mu s$ (with average $1.5\mu s$).

The time performing intersections and locating points is also dependent on the grid resolution. Indeed, it is expected that higher grid resolutions reduce the intersection times (since each grid cell will have fewer pairs of edges to intersect) and also the point location times, since it is expected that finer grid cells will lead to a higher percentage of empty cells (that are labeled with the id of the polygon where it is and, thus, accelerate the queries) and the point location in non-empty cells will also be faster (since there will be fewer edges per cell).

As it can be seen in Tables 3.2, 3.3 and 3.4, in the largest datasets higher grid resolutions led to fewer intersection tests (column *# inter. tests* $\times 10^6$). However, in the smallest dataset the number of intersections tested increased as the resolution

Table 3.3: Elapsed time (in seconds) for the overlay of the UsAquifers dataset with UsCounty.

UsAquifers × *UsCounty*									
Grid size		Time (s)				Mem.	# inter.	% cells	% cells
1^{st}	2^{nd}	Grid.[a]	Inte.[b]	Loc.[c]	Total	(GB)	tests[d]$\times 10^6$	refined[e]	labeled[f]
100^2	25^2	6.38	2.06	1.48	9.92	0.3	3.88	9.13	82.90
	40^2	6.26	1.19	1.53	8.98	0.5	1.95	9.13	88.53
	55^2	6.06	0.75	1.45	8.26	0.8	1.26	9.13	91.41
200^2	25^2	5.97	0.86	1.32	8.15	0.5	1.43	6.58	88.94
	40^2	6.22	0.52	1.41	8.15	1.1	0.81	6.58	92.80
	55^2	6.09	0.36	1.51	7.96	1.9	0.56	6.58	94.68
300^2	25^2	5.93	0.49	1.18	7.60	0.7	0.86	4.97	91.21
	40^2	6.12	0.33	1.36	7.81	1.6	0.52	4.97	94.32
	55^2	5.94	0.27	1.68	7.89	2.7	0.37	4.97	95.82
400^2	25^2	5.96	0.35	1.13	7.44	0.8	0.62	3.79	92.32
	40^2	5.89	0.27	1.40	7.56	1.7	0.39	3.79	95.04
	55^2	5.99	0.23	1.88	8.10	2.8	0.29	3.79	96.35
500^2	25^2	6.13	0.33	1.13	7.59	0.7	0.50	2.95	92.99
	40^2	6.11	0.26	1.55	7.92	1.5	0.31	2.95	95.47
	55^2	6.39	0.24	2.06	8.69	2.4	0.24	2.95	96.67

Elapsed time in seconds (excluding I/O) for the main steps of EPUG-Overlay and other statistics using different grid sizes for the first and second level;
[a] time spent creating and refining the uniform grid; [b] time spent intersecting edges;
[c] time spent labeling empty grid cells, locating points and creating the output map;
[d] total number of edge-edge intersection tests performed; [e] percentage of first level grid cells that were refined; [f] percentage of grid cells that were labeled.

increased. This happened because in this dataset the smallest grid resolutions were already enough to minimize the number of intersection tests and, as the resolution is increased (and the size of the cells was reduced), the number of grid cells intersected by some edges increased and, thus, these edges were tested for intersection in several cells. Indeed, during the overlay of the smallest datasets each edge intersected, on average, 2.1 cells when the resolution of the first and second level grid sizes were set to, respectively, 100^2 and 25^2 and 5.1 cells when the resolution was set to the largest value (500^2 and 55^2).

In the experiment with the largest datasets (during the overlay of UsWaterBodies with UsBlockBoundaries) each edge intersected, on average, 1.1 cells when

Table 3.4: Elapsed time (in seconds) for the overlay of the UsWaterBodies dataset with UsBlockBoundaries.

UsWaterBodies × *UsBlockBoundaries*									
Grid size		Time (s)				Mem.	# inter.	% cells	% cells
1^{st}	2^{nd}	Grid.[a]	Inte.[b]	Loc.[c]	Total	(GB)	tes.[d]$\times 10^6$	refined[e]	labeled[f]
	25^2	77.13	299.04	53.78	429.95	9.0	497.93	0.64	82.42
1000^2	40^2	78.82	143.07	40.25	262.14	9.6	233.15	0.64	86.32
	55^2	79.75	83.23	36.61	199.59	10.1	143.20	0.64	89.00
	25^2	77.65	92.11	38.83	208.59	10.0	165.51	0.53	88.77
2000^2	40^2	78.90	50.18	36.53	165.61	11.6	83.13	0.53	91.13
	55^2	79.04	32.79	38.92	150.75	13.5	54.16	0.53	92.87
	25^2	80.17	54.63	35.37	170.17	11.2	91.16	0.43	91.18
3000^2	40^2	78.52	28.40	38.45	145.37	14.1	48.46	0.43	92.89
	55^2	79.05	20.50	46.26	145.81	17.5	32.98	0.43	94.25
	25^2	77.36	37.94	36.34	151.64	12.6	61.45	0.36	92.58
4000^2	40^2	76.24	21.87	44.49	142.60	16.5	34.21	0.36	93.89
	55^2	77.85	15.92	56.32	150.09	21.7	24.08	0.36	95.03
	25^2	76.38	29.66	36.50	142.54	14.0	46.06	0.30	93.53
5000^2	40^2	78.50	18.15	49.64	146.29	19.4	26.65	0.30	94.54
	55^2	81.56	14.31	70.11	165.98	25.9	19.31	0.30	95.52

Elapsed time in seconds (excluding I/O) for the main steps of EPUG-Overlay and other statistics using different grid sizes for the first and second level;
[a] time spent creating and refining the uniform grid; [b] time spent intersecting edges;
[c] time spent labeling empty grid cells, locating points and creating the output map;
[d] total number of edge-edge intersection tests performed; [e] percentage of first level grid cells that were refined; [f] percentage of grid cells that were labeled.

the resolution of the grid was the smallest one (1000^2 in the first level and 25 in the second one) and 1.5 cells when the resolution of the grid was the largest one (5000^2 and 55^2).

Thus, too coarse grids will lead to more pairs of edges tested for intersection in each grid cell while too fine grids leads edges to be copied to several grid cells, which also may increase the amount of intersection tests.

It is also interesting to observe that the number of intersection tests performed is usually small if compared with the number of edges in the input maps. If we choose a first level grid with sizes 200^2, 400^2 and 2000^2 (which, as it will be described later, is a reasonable choice) and a second level grid with size 45^2, the number of intersection tests performed was maximum 2.6 times the number of edges in the largest map of the pair being intersected.

As expected, the percentage of cells that are labeled with information about the polygons where they are located increases as the resolution of the uniform grid increases (since higher resolution grids tend to have more empty cells). While, on one hand, a larger amount of empty cells (labeled) accelerates the point location process more, on the other, the larger the amount of cells that need to be labeled the higher the amount of time spent labeling the connected component of empty grid cells.

As it can be seen, some components of EPUG-Overlay benefit from coarser grid while others benefit from finer grids. However, as these experiments suggest, the actual running time of the algorithm changes slowly as the grid resolution changes and, therefore, a reasonable choice for the resolution is enough to obtain a good performance.

We employed a heuristic for choosing a reasonable first level grid resolution. Supposing that the resolution of the first level grid is g_1, the maps A and B have, respectively, n_A and n_B edges, and that an edge from maps A and B will intersect, respectively, k_A and k_B cells on average, the expected number of edges per grid cell in the two maps will be, respectively, $\frac{k_A \times n_A}{g_1^2}$ and $\frac{k_B \times n_B}{g_1^2}$.

Assuming the distribution of edges per grid cell is uniform, the expected number of pairs of edges per cell (*pairs*) will be:

$$pairs = \frac{n_A \times n_B \times k_A \times k_B}{g_1^4} \tag{3.1}$$

Thus, the resolution of the first level grid can be estimated using the following formula:

$$g_1 = \sqrt[4]{\frac{n_A \times n_B \times k_A \times k_B}{pairs}} \tag{3.2}$$

If we fix $\frac{k_A \times k_B}{pairs} = \frac{1}{50}$ and round g_1 to a simple multiple of a power of ten (for readability since the optimum is so broad), the suggested grid resolution for processing the pairs of maps *BrSoil/BrCounty*, *UsAquifers/UsCounty*, and *UsWaterBodies/UsBlockBoundaries* will be, respectively, 200^2, 400^2 and 2000^2, what represents a reasonable balance between memory and performance according to Tables 3.2, 3.3 and 3.4. Based on these experiments, we always use a 40×40 second level grid.

Of course, since the grid resolution and the criteria for refining the first level grid can be easily changed in EPUG-Overlay, the user may use other strategies for fine-tuning these parameters.

Tables 3.5, 3.6 and 3.7 show the quality of EPUG-Overlay's implementation processing 3 pairs of maps and using the design choices described above, that is, the resolution of the first level grid was set using equation 3.2, the expected number of pairs of edges per cell was set to 50 and all cells with more than 50 pairs of edges was refined creating a 40×40 second level grid. The table gives the elapsed times for our three test cases. The times are given for each of the program's eight stages, both when using one thread and when using 16 threads.

Excluding the I/O time, the total parallel speedup ranged from 5 to 7 times. It is less than 16 because operations like memory allocation and writing to a common global array are sequential and because the Xeon processor used in the experiments supports the Intel® Turbo Boost technology [78] to increase their frequency from their 3.1GHz base operating frequency to up to 3.8GHz when fewer cores are active.

In general, the slowest processing steps of EPUG-Overlay scale better. For example, during the processing of the largest pair of maps (Table 3.7), the two slowest steps are computing the intersections and locating the vertices of one map

in the other one. Both these steps achieved a 10 times speedup.

The initialization of the empty cell labels presented a reasonable parallel speedup in the smallest dataset (5 times speedup), but this speedup was reduced as the dataset size increased (in the largest dataset this step achieved 1.4 times

Table 3.5: Elapsed time of the main steps of EPUG-Overlay for the overlay of the BrSoil with BrCounty. Times considering the grid size from Equation (3.2) and using 1 and 16 threads.

Maps:	*BrSoil* × *BrCounty*		
Grid size:	200×200		
	Time (sec.)		**Parallel**
Threads:	1	16	**speedup**
Read maps	0.88	0.86	1.0
Make grid	2.11	0.67	3.1
Refine 2-level grid	6.45	0.55	11.7
Intersect edges	1.20	0.10	12.0
Initialize cell labels	2.57	0.48	5.4
Locate vertices	1.78	0.17	10.5
Comp. output faces	0.37	0.11	3.4
Write output	0.70	0.63	1.1
Total w/o I/O	14.48	2.08	7.0
Total with I/O	16.06	3.57	4.5

Table 3.6: Elapsed time of the main steps of EPUG-Overlay for the overlay of the UsAquifers with UsCounty. Times considering the grid size from Equation (3.2) and using 1 and 16 threads.

Maps:	*UsAquifers* × *UsCounty*		
Grid size:	400×400		
	Time (sec.)		**Parallel**
Threads:	1	16	**speedup**
Read maps	5.47	5.23	1.0
Make grid	15.05	5.22	2.9
Refine 2-level grid	8.75	0.67	13.1
Intersect edges	2.79	0.27	10.3
Initialize cell labels	2.27	0.53	4.3
Locate vertices	9.73	0.77	12.6
Comp. output faces	0.79	0.10	7.9
Write output	3.82	4.71	0.8
Total w/o I/O	39.38	7.56	5.2
Total with I/O	48.67	17.50	2.8

speedup). This can be explained because, for simplicity, the labeling process was implemented using a recursive flood-fill algorithm, what is very memory intensive and does not scale well mainly for larger grids. However, this scalability does not significantly affect EPUG-Overlay since the labeling step is the fastest one in EPUG-Overlay (taking only 0.9% of the total running time when EPUG-Overlay process the largest dataset sequentially). If the performance of this stage become critical, we suggest to implement it using a faster and more memory efficient flood-fill algorithm (such as a scan-line algorithm).

Even the sequential version of EPUG-Overlay is very competitive compared to other overlay programs. E.g., the GRASS GIS [79] overlay (sequential) module that uses floating point (and thus, does not compute the exact overlay) takes 5321 seconds to overlay UsWaterBodies with UsBlockBoundaries while the sequential version of EPUG-Overlay uses only 1240 seconds (including I/O) with a 2000×2000 1^{st}-level grid and 40×40 2^{nd}-level grid. EPUG-Overlay's execution time is reduced to 264 seconds if it is executed in parallel. Of course, its important to notice that the GRASS running time includes some systems overheads associated with the GIS environment.

Thus, the overhead added by using exact arithmetic can be balanced by using

Table 3.7: Elapsed time of the main steps of EPUG-Overlay for the overlay of the UsWaterBodies with UsBlockBoundaries. Times considering the grid size from Equation (3.2) and using 1 and 16 threads.

Maps:	*UsWaterBod.* × *UsBlockBound.*		
Grid size:	2000×2000		
	Time (sec.)		**Parallel**
Threads:	1	16	**speedup**
Read maps	72.39	70.98	1.0
Make grid	184.89	63.51	2.9
Refine 2-level grid	162.62	15.39	10.6
Intersect edges	506.08	50.18	10.1
Initialize cell labels	11.03	7.66	1.4
Locate vertices	251.87	24.75	10.2
Comp. output faces	22.60	4.12	5.5
Write output	29.00	27.44	1.1
Total w/o I/O	1,139.09	165.61	6.9
Total with I/O	1,240.48	264.03	4.7

a simple and efficient data structure and parallel programming.

3.3.2 The two-level uniform grid relevance

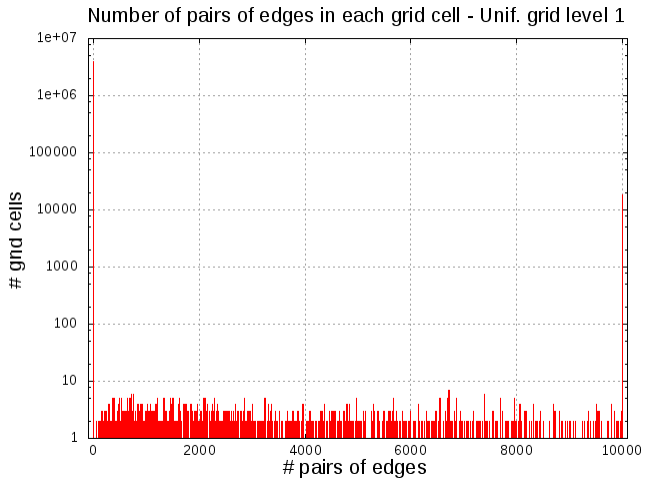
In this section we will present some results to show the relevance of using a two-level uniform grid instead of a conventional uniform grid or a Quadtree.

First, the main purpose for using a data structure in map overlay is to try to reduce the number of segment pairs that need to be checked to verify if they intersect. For example, Figure 3.6 presents some statistics for overlaying maps *UsWaterBodies* with *UsBlockBoundaries* using a 2-level uniform grid with 2000×2000 cells in the first level, 40×40 cells in the second one and 50 as the threshold. After the 1-level uniform grid creation, see Figure 3.6(a), there are 20000 cells with over 10000 pairs of edges to be checked. Then, it would be necessary to check more than 2×10^8 edge pairs. Nevertheless, with a second level, see Figure 3.6(b), there are now about only 100 cells with more than 10,000 pairs of edges to be checked.

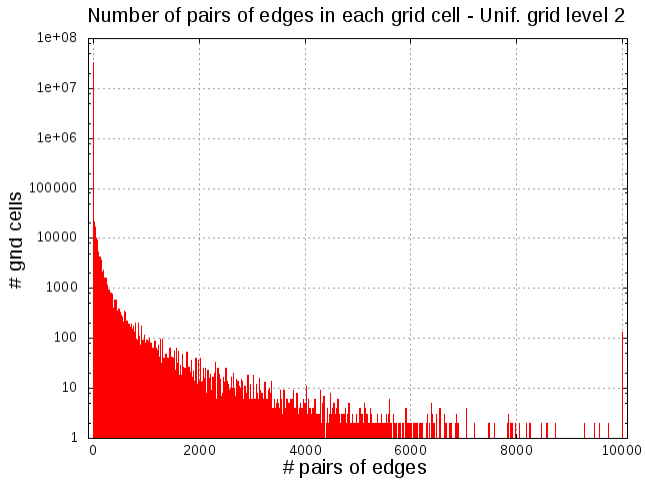
As mentioned before, besides accelerating the edge-edge intersection computation process, the uniform grid also accelerates the point location step for similar reasons.

The next question is, why do not use a more dynamic data structure such as a Quadtree? This is a bad idea because our tests found that just creating a Quadtree requires too much memory and takes more time than the whole process to overlay the two maps using a 2-level uniform grid. Furthermore, it is harder to navigate in the cells of dynamic data structures, and also they are usually harder to parallelize efficiently.

For example, Table 3.8 shows the time and the memory required just to create a Quadtree for the three datasets used in the tests (the threshold for creating another branch in these structures was set to 50 pairs of edges). Also, the maximum depth achieved by the tree and the total number of pairs of edges in the cells are presented. In all cases, just the grid creation spent more time than the EPUG-Overlay processing time presented in Tables 3.5, 3.6 and 3.7.



(a)



(b)

Figure 3.6: Histogram for the number of pairs of edges in the grid cells - The number of grid cells distribution considering the number of pairs of edges to be evaluated when overlaying maps *UsWaterBodies* (containing 21 million edges) with *UsBlockBoundaries* (containing 32 million edges): (a) 1-level uniform grid (with 2000^2 cells); (b) 2-level uniform grid (with 40^2 cells).

Table 3.8: Elapsed time and memory size spent just to create a Quadtree sequentially.

Maps overlaid	Max. Depth	Time (sec.)	Memory (GB)	Pairs of edges ($\times 10^6$)
BrSoil × *BrCounty*	13	64.21	0.16	0.7
UsAquifers × *UsCounty*	18	423.76	0.68	0.8
UsWBodies × *UsBBound.*	21	8,270.07	9.81	30.8

3.4 Summary

We have presented EPUG-Overlay, an efficient algorithm, with implementation, using rational numbers to compute the exact overlay between two maps. Even though EPUG-Overlay performs computation using multiple precision rational arithmetic, which is much slower than hardware-implemented floating point, its performance is at least competitive (being more than 4 times faster when executed sequentially and more than 20 times faster when executed using 16 threads) to the approximate overlay method included in the widely used GRASS GIS.

Furthermore, EPUG-Overlay is eminently parallelizable. With OpenMP, we achieved a speedup (excluding I/O) of a factor of up to 7 compared with the sequential implementation. And, we have ideas about how to make these times even better, if there were a need.

EPUG-Overlay was developed to evaluate the ideas we planned to apply for exactly computing the intersection of triangular meshes in E^3. As it will be shown in Chapter 6, these ideas were extended to E^3 with good results.

CHAPTER 4
2-level 3D uniform grid indexing

In this chapter, we will present the details of an efficient implementation of a parallel 3D uniform grid. A 2D uniform grid was already described in Chapter 3, where it was applied to the 2D map overlay problem. In this chapter we will describe a 3D version of the grid and present details about the optimization techniques we have incorporated into this index.

Since the 3D uniform grid is employed in both the PinMesh and 3D-EPUG-Overlay methods, experiments related to the grid will be performed in the chapters presenting these two algorithms.

As mentioned in Section 3.1, the uniform grid is useful in computational geometry to efficiently cull a combinatorial set of pairs or triples of objects, to find a much smaller subset that are likely to coincide. For data that is uniformly independently and identically distributed, the expected number of pairs or triples of objects processed is linear in the size of the input plus the output [74], [76], [80]. The basic idea is to superimpose a grid over the data, with the grid cell size set so that the expected number of edges per cell remains constant as the total number of edges grows. Then, insert into each cell c the edges of the map intersecting c. As presented in Chapter 3, we have already used a two-level uniform grid to develop an exact parallel 2D map overlay algorithm. Experimentally, three-level grids and trees are slower.

In this project we implemented a two-level 3D uniform grid to accelerate the

---

Portions of this chapter previously appeared as:

S. V. G. Magalhães, "An efficient algorithm for computing the exact overlay of triangulations", in *Proc. 2nd ACM SIGSPATIAL PhD Workshop*, SIGSPATIAL PhD'15. New York, NY, USA: ACM, 2015, pp. 3:13:4.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li,"Pinmesh - Fast and exact 3d point location queries using a uniform grid", *Comput. & Graph.*, vol. 58, pp. 1 - 11, 2016, Shape Modeling International 2016.

point location algorithm. We intend to use this same indexing data structure to also accelerate the other steps of the intersection computation. Thanks to its simplicity and regularity, the uniform grid creation was successfully implemented in parallel using OpenMP.

Given the input mesh M composed of a set of 3D triangles as described in Section 2.2.2, a 3D regular $g_1 \times g_1 \times g_1$ grid G overlapping M is created. Then, each triangle t in M is inserted into the grid cells that it intersects. Each cell actually stores pointers to the triangles whose axis-aligned bounding boxes intersect it. The resolution of the grid (g_1) is a parameter that can be defined manually or dynamically computed based on statistics about the input data.

The grid cells' contents (the pointers to the triangles) are stored in a ragged array. This is more compact than an array of linked lists, and stores each cell's contents contiguously, which might play better with the computer's cache. A ragged array is created by scanning the data twice. In the first scan, the number of triangles that will be inserted into each grid cell is added up. Then a linear array A containing enough slots to store all the pointers to the triangles is allocated. Since the number of triangles in each grid cell is now known, for each cell c we create a pointer pointing to the beginning of a subarray of A containing enough slots to store c's triangles. Finally, a second scan is performed on the triangles and their pointers are inserted into their positions in A. Figure 4.1 illustrates the difference between the memory layout obtained when each cell has a dynamic data structure (such as a C++ vector) and the layout obtained using a ragged array. This technique is also common in, e.g., parallel bucket sorting.

While the creation of the ragged array requires two scans through the data, it replaces the allocation of many small memory blocks with one big allocation of the array A. This strategy has two advantages.

First, by avoiding the use of dynamic data structures such as lists and dynamic arrays, the ragged array reduces the memory overhead: since the uniform grid is 3D, the number of cells grows cubicly with the grid resolution and, thus, the total memory requirement of the grid would be big if each cell stored a dynamic array (which, because of its allocation strategy, usually requires more slots than the

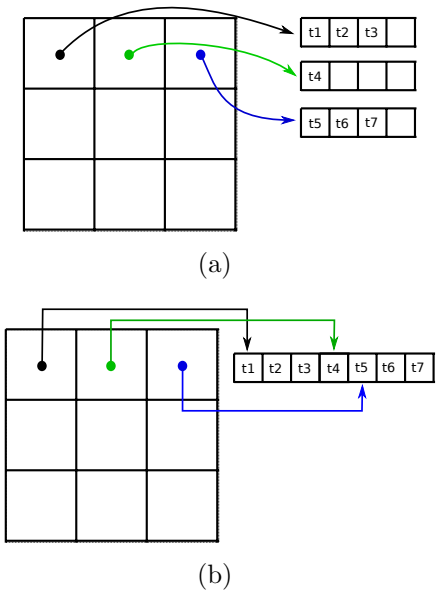


Figure 4.1: Dynamic array versus ragged array - 3×3 uniform grid (for clarity, a 2D grid was used) using dynamic arrays (a) versus ragged array (b). Only the memory related to the first row of the grid is shown.

number of elements stored) or a linked list (which has an overhead caused by the pointers in each node). By using a ragged array, on the other hand, we have exactly one slot for each triangle stored, and each grid cell needs to store only one pointer.

Second, since one big allocation is performed instead of several smaller allocations the memory is not fragmented and there is more data locality. Furthermore, big allocations are usually performed faster than several smaller allocations.

As will be mentioned in Section 5.3, in our point location algorithm the uniform grid was tuned by choosing an specific grid size that is suitable to efficiently perform queries and by augmenting the grid cells with information about the polyhedron where each cell is located.

4.1 Implementation details

Algorithm 4.1 summarizes the process of creating a two-level uniform grid. Initially, the first level uniform grid is created. Then, some of the cells are refined (creating a second level grid).

Algorithm 4.1: Creation of the uniform grid.

```
1: M: mesh represented as a set of triangles
2: g_1: resolution of the first level of the uniform grid
3: g_2: resolution of the second level of the uniform grid
4: maxTrianglePerCell: threshold for refining a cell
5: //Create and refine the uniform grid G
6: G ← g_1^3 3D uniform grid
7: for each triangle t in M do
8:     Insert t into the G's cells intersecting its bounding-box
9: end for
10: for each grid cell c in G do
11:     if |triangles(c)| > maxTrianglePerCell then
12:         Create a g_2^3 uniform grid in c
13:         for each triangle t in triangles(c) do
14:             Insert t into the c's cells intersecting its b.box
15:         end for
16:     end if
17: end for
```

As was mentioned before, because of the ragged array in our actual implementation the insertion of the triangles into the uniform grid is performed in two steps using a ragged array. Algorithm 4.2 illustrates the creation of a uniform grid using a ragged array (this algorithm considers only one of the levels of the grid). For simplicity, Algorithm 4.2 assumes the uniform grid is represented linearly and the cells are labeled from 0 to $g^3 - 1$ (where g is the grid size).

First (lines 5 to 10), the number of triangles in each grid cell is counted (the grid NT contains the counter of each cell). Then (lines 11-14), a prefix-sum is employed to compute $StartPos$, the starting position (in the ragged array of triangles) of each uniform grid cell. Next (lines 15-18), the ragged array ($Triangles$) is allocated with size equal to the total number of triangles in all cells and a temporary array $NInsertedTris$ is created. $NInsertedTris$ will store the current number of triangles already inserted into each grid cell.

Finally, in the second pass through the triangles (lines 21 to 26), they are actually inserted into the ragged array $Triangles$.

Because of its uniformity and simplicity, the uniform grid can be easily constructed in parallel. Initially, the bounding-boxes of the triangles in the input mesh

Algorithm 4.2: Inserting triangles into a ragged array.

```
1: M: mesh represented as a set of triangles
2: g: resolution of the uniform grid
3:
4: //First pass: count number of triangles in each cell
5: NT ← array with g^3 zeros
6: for each triangle t in M do
7:     for each grid cell c intersecting t do
8:         NT[c] ← NT[c] + 1
9:     end for
10: end for
11: StartPos ← array with g^3 zeros
12: for each grid cell c in 1...g^3 − 1 do
13:     StartPos[c] ← StartPos[c − 1] + NT[c − 1]
14: end for
15: TotalNT ← StartPos[g^3 − 1] + NT[g^3 − 1]
16: //Allocates ragged array of triangles
17: Triangles ← array with TotalNT triangles
18: NInsertedTris ← array with g^3 zeros
19:
20: //Second pass: insert triangles into grid
21: for each triangle t in M do
22:     for each grid cell c intersecting t do
23:         TriPosThisCell ← NInsertedTris[c]
24:         NInsertedTris[c] ← NInsertedTris[c] + 1
25:         Triangles[StartPos[c] + TriPosThisCell] ← t
26:     end for
27: end for
```

can be computed in parallel.

Then, the first level grid can be created in parallel. Consider Algorithm 4.2: the for loop in lines 6-10 can iterate through the triangles in parallel (to avoid atomics, in our implementation each thread keeps a private grid NT to count its triangles and these private grids are added at the end of the for loop).

The loop in lines 12-14 could also be processed in parallel using a parallel prefix-sum algorithm. However, for simplicity it was not parallelized since it is not a performance bottleneck: the complexity of this loop is $\Theta(g^3)$ and the grid sizes are typically small in comparison with the number of triangles in the meshes.

Finally, the triangles can be inserted into the grid cells in parallel (loop in

lines 21-27). However, since two triangles may concurrently be inserted into the same cell, the operations performed in lines 23 and 24 have to be done as a single atomic operation (this can be efficiently performed using the *atomic* and *capture* OpenMP directives).

Once the first level of the grid is created, the next step is to refine the grid cells. Since during the refinement of a grid cell c a new (nested) uniform grid is created inside c and c's triangles are inserted only into this new grid, different first level cells can be safely refined in parallel without the use of synchronization and, therefore, the set of grid cells that need to be refined is processed in parallel.

4.2 Summary

This chapter described the 3D two-level uniform grid we intend to use for indexing the triangular meshes, and thus accelerate the computation. Because of its uniformity and simplicity, the uniform grid can be quickly created in parallel.

As will be shown in Chapters 5 and 6, this data structure was successfully applied to accelerate the point location and mesh intersection processes.

CHAPTER 5
The point location problem

This chapter presents PinMesh our solution for the 3D point location problem. Given an input mesh and a set of query points, the objective is to determine in what polyhedron each point is located.

As mentioned in chapter 2, the point location problem is an important step of the overlay computation. However, since this problem itself is an important Computational Geometry subject, we dedicated this chapter to it.

5.1 The problem

To illustrate the point location problem, Figure 5.1 presents an example input mesh containing 11 triangles and 3 regions (region 1 in purple, region 2 in yellow, and the exterior region). Triangle ABC bounds regions 1 (on the negative side) and 2 (on the positive side) while the other triangles bound the exterior region and either region 1 or 2. In this example there is only one query point q and it is in region 2.

5.2 Performing queries

Given a mesh M and a set of query points Q, the objective is to determine what region contains each point $q \in Q$. If a point is not inside a polyhedron in the mesh it is considered to be in the *exterior region*.

Portions of this chapter previously appeared as:

S. V. G. Magalhães, "An efficient algorithm for computing the exact overlay of triangulations", in *Proc. 2nd ACM SIGSPATIAL PhD Workshop*, SIGSPATIAL PhD'15. New York, NY, USA: ACM, 2015, pp. 3:13:4.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li,"Pinmesh - Fast and exact 3D point location queries using a uniform grid", *Comput. & Graph.*, vol. 58, pp. 1 - 11, 2016, Shape Modeling International 2016.

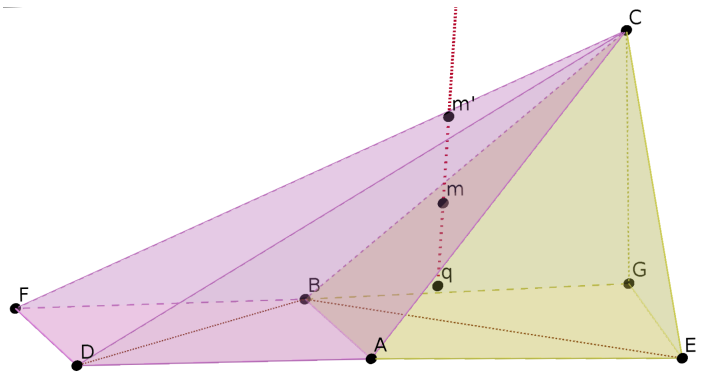


Figure 5.1: Example of an input mesh and a query point.

Given a query point $q \in Q$, its location is determined by casting a vertical ray l oriented from q in the positive z direction (the particular orientation is not important) and, then the mesh triangle t that intersects l in the lowest intersecting point m is computed. Supposing m is on a face of M (but not on an edge or vertex, which are two special cases), then the segment qm will only intersect the mesh at m and q will necessarily be in one of the two regions bounded by t.

Consider, for example, the mesh in Figure 5.1: the vertical ray l starting on q intersects the mesh at points m and m'. Since m is the lowest mesh point intersecting l, we know that q is in one of the two regions adjacent to the triangle containing m (otherwise, because of the Jordan Curve Theorem, l would intersect the mesh again on a point closer to q than m, which is impossible since m is the lowest mesh point above q).

To determine in which of the two regions bounded by a triangle t a query point q is, the sign of the dot product between the vector $(0, 0, -1)$ (a vector parallel to l) and t's normal is computed (this process is similar to the Back-Face Culling technique commonly used in Computer Graphics). Iff the sign is positive, q is on the positive side of t. If the slope of t's normal is 0, then t is a vertical triangle and the ray intersected t on a vertex or edge; we will discuss this later.

Figure 5.2 presents an example where a triangle t is used to determine the region where a query point q is: since the sign of the dot product between the vector $(0, 0, -1)$ and the normal n is negative, q is on the negative side of t. Figure 5.3 presents an example of the 2D version of the point containment problem (the 3D

version is equivalent) that illustrates the challenge of determining the position of a query point q basing on the orientation of a vertical edge (similar to a vertical triangle in 3D): in Figure 5.3 (a) the edge AB is the first edge crossed by the ray traced from q and, since the sign of the dot product between $(0, -1)$ and n is negative, this means that q is on the region that is on the negative side of AB. If the normal was parallel to the x-axis (Figures 5.3 (b) and (c)), on the other hand, the first point hit by the ray would be on a vertex of AB and, thus, the process of determining in what region q is would need to treat an ambiguity: even though the segment AB is the same on Figure 5.3 (b) and (c) and q is on the same region in these two figures, in (b) q is on the region on the negative side of AB while in (c) it is on the positive side.

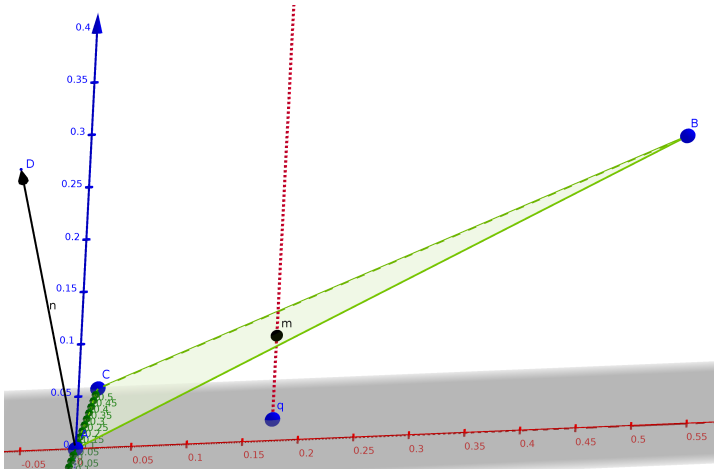


Figure 5.2: Example of a query point q directly below a triangle ABC with normal n.

To summarize, PinMesh needs three basic predicates in order to determine in what region a query point q is:

- $isOnProj(t, q)$: given a triangle t and a query point q, returns true iff the vertical projection of q onto the plane passing through t is in the interior of t.
- $isAbove(t, q)$: given a query point q and a triangle t such that $isOnProj(t, q)$ is true, returns true iff the projection of q onto t is above q. That is, this operation verifies if a triangle is above a point.

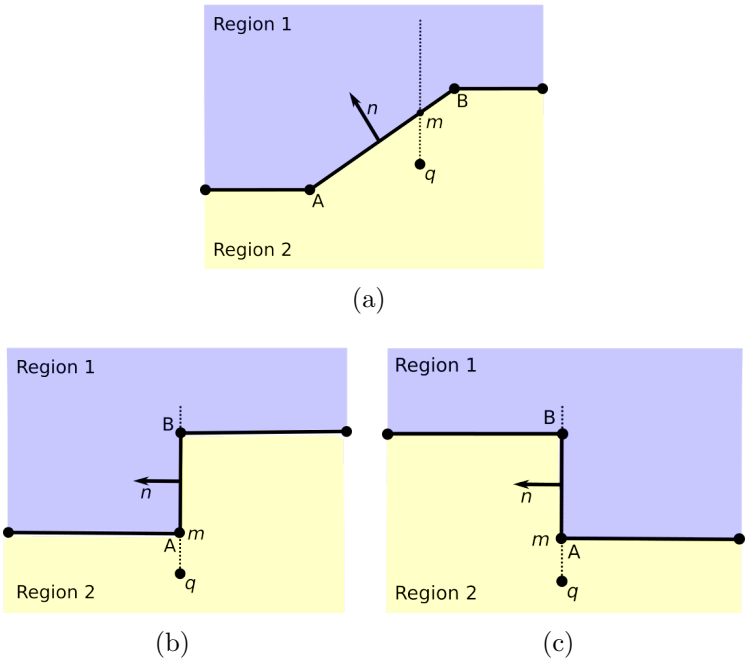


Figure 5.3: Challenge to the 2D version of the point location problem caused by vertical edges - In (a) q's location can be easily computed since the ray starting on q hits the interior of the edge AB. In (b) and (c), on the other hand, the orientation of AB cannot be easily used to locate q since AB is a vertical line segment.

- $isBelow(t, t', q)$: given two triangles t and t' directly above a query point q, returns true iff the z component of the projection of q onto t is smaller than the z component of the projection of q onto t'.

Thus, to locate q, the first step constructs a subset T of the triangles in the mesh M that are directly above q, that is, $T = \{t \in M \mid isOnProj(q, t) \text{ and } isAbove(q, t)\}$. Then the lowest triangle u directly above q is selected from T using the operation $isBelow(q, t, t')$ to compare pairs of triangles. Finally, the location of q is determined by verifying the sign of the dot product between the vector $(0, 0, -1)$ and $u's$ normal.

The predicate $isOnProj(t, q)$ was implemented using the barycentric coordinates of the projection of t onto the plane $z = 0$. Similarly, $isBelow(t, t', q)$ was implementing by computing the plane equations for the triangles t and t' and, then, using these equations to project q onto the triangles and compute the z-coordinates

of the two projections. More details about this implementation will be presented in Section 5.4.3.

5.3 Using a two-level uniform grid to accelerate the queries

As mentioned in Section 4, we use a two-level 3D uniform grid quickly created in parallel to accelerate the point location algorithm. In this section we will explain why the uniform grid is an efficient indexing data structure for performing point location queries and present how we augmented the grid cells with labels that accelerate even more the queries.

If the grid resolution is chosen such that the expected average number of triangles per cell is constant, the expected time to locate each point q will be constant: determining in what grid cell c the point q is takes constant time. Testing the intersection between the ray and the triangles in c also takes constant time (since the number of triangles in the cells is constant). If the ray hits at least one triangle, the query result is found in constant time. However, there is a probability p that it does not. Because the expected number of triangles in c is constant, p is also expected to be a constant independent of the map size. Then we continue the ray into the next higher cell and test for an intersecting triangle. The expected number of cells to process until to find an intersecting triangle is $1/(1-p)$.

To further accelerate the queries, we augmented each grid cell c with a label indicating in what region c is. If there is no triangle in c, this means that c is completely inside a region R and, therefore, c's label is initialized with R. Because of the Jordan Curve Theorem, any query point q in c will automatically be in R and, therefore, queries performed using points in empty cells can be quickly performed by just returning the label associated with the cell. If c contains at least one triangle, on the other hand, c may cover more than one region and, therefore, its label is not initialized.

Given an empty cell c, its label is initialized by sampling a point in c (for example, the center of the cell), and then querying this point using the proposed point location algorithm. If c has an empty neighbor cell c' it is clear that, because a continuous path connecting these two cells may be traced without crossing any

triangle, c' is in the same region where c is, and therefore c' could be tagged with the same label. This process could be performed recursively labeling entire connected components of empty cells. For performance purposes, it is performed using a scan-line flood fill algorithm.

5.4 Implementation details

PinMesh is composed of two steps: in the first one (the pre-processing step), the uniform grid is created, refined and, finally, empty grid cells are labeled to identify the region where they are. After that, in the second step the queries are performed.

Algorithm 5.1 presents the pseudo-code of the pre-processing step discussed in the previous Section. Initially, the uniform grid is initialized as described in Section 4. After the creation and refinement of the uniform grid, the next step is the initialization of the labels of the empty grid cells with the ids of the regions they are located. As mentioned in Section 5.3, these labels are initialized using a scan-line flood-fill algorithm to label the connect-components of empty grid cells. To avoid the implementation of a parallel scan-line algorithm (which would require synchronizations), we divided the first level uniform grid into cubic blocks (for example, a 64^3 grid could be divided into 512 8^3 blocks) and used the flood-fill algorithm to label the connected components in each block independently. Since the blocks are processed independently, this processing was performed in parallel.

Algorithm 5.2 presents the high level pseudo-code of the point location function. For simplicity, we assume the grid has only one level (the algorithm for a grid with two levels is similar).

Since the objective is to find the lowest intersection point m between a triangle (the *lowest triangle*) and a vertical ray l starting on q, the algorithm iterates through all the grid cells that would intersect l (since m will be necessarily on a triangle in one of these cells).

Therefore, the loop on line 7 iterates through the cells c from q_c (q's cell) to the highest cell above q_c. Since the cells are processed in order from the lowest to the highest one, if a labeled cell is hit (that is, if l hits an empty cell) before any

Algorithm 5.1: PinMesh's pre-processing step.

```
1: M: mesh represented as a set of triangles
2: G ← 3D uniform grid to index M
3: //Label G's cells
4: Initialize G's cells labels with NULL
5: for each empty grid cell c in G do
6:     if c.label = NULL then
7:         q ← point in the center of c
8:         label ← locatePointInMesh(q,M,G)
9:         //Use a scanline algorithm to label c's connected
10:        //component of empty cells
11:        scanlineFloodFill(c,label)
12:    end if
13: end for
```

triangle the label of this cell is returned. Otherwise, for each triangle t in c (loop on line 12) that is directly above q, if the projection of q on t is lower than the projection of q on the *lowest triangle* seen so far, the *lowest triangle* is updated with t.

For performance, every time a lower triangle is found, the highest cell that needs to be processed is updated with the cell c containing the projection of q on $lowestTriangle$ (line 15) since no cell above c can have a lower triangle. This strategy reduces the number of grid cells that need to be processed.

Figure 5.4 illustrates this process in 2D: the algorithm starts the search on c_0 (q's cell) and, among the 4 edges in c_0, only edge xy intersects the ray l (on point m_2). Notice that, even though xy is in c_0, m_2 is not in c_0 since xy intersects several grid cells. In the next iteration, cell c_1 is processed and the edge uv containing the intersection point m_1 (that is lower than m_2) is found. Since the lowest intersection seen so far (m_1) is on an edge in c_1, no grid cell above c_1 needs to be processed.

After the loop on line 7 ends, the algorithm tests if the lowest triangle was initialized. If it was not, this means that all cells above q_c were processed and no triangle directly above q was found. Thus, q must be in the exterior region. If the lowest triangle was initialized, we know that the face of this triangle is directly above q (and the projection of q on this face represents the lowest point on the mesh that is directly above q) and, thus, the sign of the dot product between $(0, 0, -1)$

and the normal of the lowest triangle is used to determine in what region q is.

Algorithm 5.2: Function locatePointInMesh(q,M,G).

```
1: q (argument): query point
2: M (argument): mesh represented as a set of triangles
3: G (argument): 3D uniform grid created on M
4: (q_cx, q_cy, q_cz) ← coordinates of the grid cell containing q
5: highestCzToProcess ← G.gridResolution
6: lowestTriangle ← NULL
7: for c_z in q_cz...highestCzToProcess do
8:     c ← grid cell of G in coordinate (q_cx, q_cy, c_z)
9:     if lowestTriangle = NULL AND c.label ≠ NULL then
10:        return c.label
11:    end if
12:    for each triangle t in triangles(c) | isOnProj(t, q) AND isAbove(t, q) do
13:        if lowestTriangle = NULL OR isBelow(t, lowestTriangle, q) then
14:            lowestTriangle ← t
15:            highestCzToProcess ← getGridZProjection(q, t)
16:        end if
17:    end for
18: end for
19: if lowestTriangle ≠ NULL then
20:    if lowestTriangle.normal_z < 0 then
21:        return lowestTriangle.positiveSide
22:    else
23:        return lowestTriangle.negativeSide
24:    end if
25: else
26:    return EXTERIOR //q is in the exterior region
27: end if
```

Besides the special cases (that will be treated later), PinMesh could also fail because of floating point roundoff errors. To completely avoid these errors, PinMesh was implemented in C++ using GMP[77] to store and process all coordinates using rational numbers. Another advantage of using exact computation is that the technique we use to treat the special cases (Simulation of Simplicity [4]) requires exact arithmetic to correctly handle these cases. Although, in principle, adapting code to use rational numbers is a straightforward task, simply replacing the floating-point variables with multiple precision ones does not lead to good performance. Thus, PinMesh was carefully developed to use these numbers in an

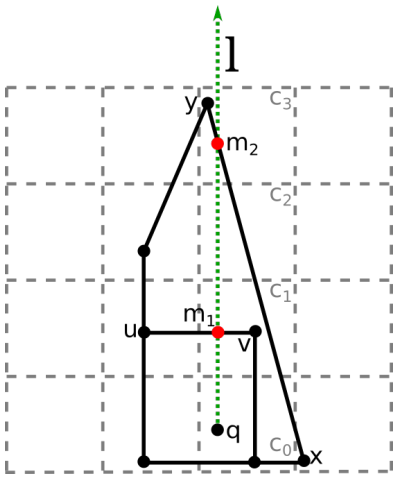


Figure 5.4: Computing the point locations using a uniform grid.

efficient way.

For example, the temporary multiple precision variables created in functions that are called several times (for example, the $isOnProj(q, t)$ function) usually require memory allocations on the heap. These allocations especially slow down the algorithm when the function is called by several *threads* in parallel. So, we avoided the creation of temporary rational numbers by carefully implementing the functions to use pre-allocated variables. Also, we avoided mathematical expressions that lead to the creation of temporary rationals (the particular kind of expressions which creates temporary allocations is described in the GMP manual [77]).

Furthermore, we also performed a space-time tradeoff by pre-computing values that involve rational numbers and that would need to be computed often. For example, given a vertex v of the mesh the operation that determines the grid cell containing v is usually performed several times through the algorithm. Thus, in the pre-processing step we compute in what cell each vertex is and store this information associated with the vertex.

5.4.1 Parallel implementation

PinMesh was developed to be easily parallelizable. Our current implementation was parallelized using OpenMP. In the pre-processing step, the axis-aligned bounding boxes of the triangles and the location of the vertices in the grid cells are computed in parallel. Because of its uniformity and simplicity, the uniform grid can

also be easily constructed in parallel.

After the bounding-boxes of the triangles are computed, the next step is to allocate the grid and insert the triangles into the corresponding grid cells. This step could be performed in parallel as described in Chapter 4, however, preliminary experiments performed during the implementation of PinMesh showed that this process is very fast if compared with other steps of PinMesh, and thus it was not worth the parallelization.

Once the first level of the grid is created, the next step is to refine the grid cells. Since during the refinement of a grid cell c a new (nested) uniform grid is created inside c and c's triangles are inserted only into this new grid, different first level cells can be safely refined in parallel without the use of synchronization and, therefore, the set of grid cells that need to be refined is processed in parallel. Because in the creation of the nested uniform grids no synchronization is performed and no memory is allocated for the rational numbers (since the triangles are inserted basing on the pre-computed grid coordinates of their bounding boxes, no operation with rational numbers is performed during this step), the grid refinement step presents a good parallel scalability (as it will be shown in Section 5.5).

After the creation and refinement of the uniform grid, the next step is the initialization of the labels of the empty grid cells with the ids of the regions they are located. As mentioned in Section 5.3, these labels are initialized using a scan-line flood-fill algorithm to label the connect-components of empty grid cells. To avoid the implementation of a parallel scan-line algorithm (which would require synchronizations), we divided the first level uniform grid into cubic blocks (for example, a 64^3 grid could be divided into 512 8^3 blocks) and used the flood-fill algorithm to label the connected components in each block independently. Since the blocks are processed independently, this processing was performed in parallel.

Finally, since the queries do not change the index, they can all be performed in parallel.

5.4.2 Special cases

Special cases, also called geometric degeneracies, are a nasty part of the life of a geometric algorithm designer. This section presents our solutions to the special cases arising in point location.

Given a query point q, its location is computed by tracing a vertical ray l starting on q and pointing to the positive z direction, and then determining what is the lowest point m of intersection between l and the surface of the mesh. If q is directly below the interior of a triangle t and m is in the interior of this triangle, q is considered to be in the region R below this face. No special case can happen in this situation since a continuous path connecting q to t's interior can be traced along l and this curve would not intersect any other triangle.

If two or more triangles intersect l at the point m (that is, if there is a tie in the process of determining the lowest triangle intersecting l), m will be, necessarily, on an edge or vertex (if m was on a face, one or more triangles would intersect on that face, what is an invalid input since triangles can intersect only on common edges or vertices).

Thus, the only possibilities of degenerate cases may happen if the query point q is on the surface of the mesh or if m is on the edge or vertex of a triangle.

First, let us assume q is not on the surface of the mesh. If m is on a vertex or edge of a triangle, we cannot arbitrarily choose one of the triangles to determine in what region q is since some of these triangles may not even be on the surface of the region containing q. See the example in Figure 5.5: in this figure we have two pyramids and the query point is exactly below the edge AB. This edge is shared among 3 triangles: ABC, ABD and ABE and, therefore, the lowest point directly above q (point m) is shared by the three triangles. However, only the triangles ABD and ABE bounds $q's$ region (the *exterior region*).

If q is on the surface of the mesh, during the processing of the triangles to determine the lowest mesh point above q, PinMesh would eventually find a point with height equal to q's height and, therefore, PinMesh could be easily adapted to treat this special case by returning a flag to indicate that q is on the surface. However, as it will be explained later, we symbolically perturb the points and, in

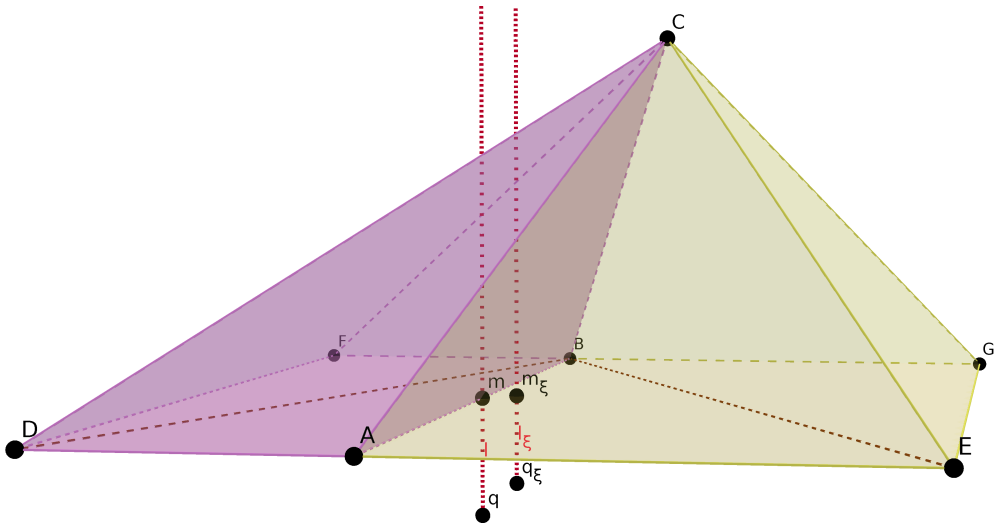


Figure 5.5: Query point exactly below a mesh edge.

order to be consistent with this perturbation, a point will never be considered to be on the surface.

To correctly handle these special cases, we used Simulation of Simplicity (SoS) [4]. Since the special cases only happen when the query point is directly below a vertex or edge of the mesh, we perform a symbolic perturbation where each query point is translated using the vector $(\epsilon,\epsilon^2,\epsilon^3)$, where ϵ is a positive infinitesimal constant, that is, each query point $q = (q_x, q_y, q_z)$ is replaced with $q_\epsilon = (q_x + \epsilon, q_y + \epsilon^2, q_z + \epsilon^3)$. As it will be shown later, this specific choice of perturbation correctly handles all these special cases.

Figure 5.6 illustrates some special cases of the 2D version of the point location problem and the effect caused by SoS, when each point q_i is slightly translated to $q_{i\epsilon}$.

As mentioned in Section 2.4, Edelsbrunner and Mücke [4] presented three requirements to guide the choice of the polynomials used to represent the perturbations in an algorithm using SoS:

1. The perturbed geometric objects must be simple (non-degenerate) if $\epsilon > 0$ is sufficiently small.

2. If an object is non-degenerate, then its perturbed version must retain the properties of its original version.

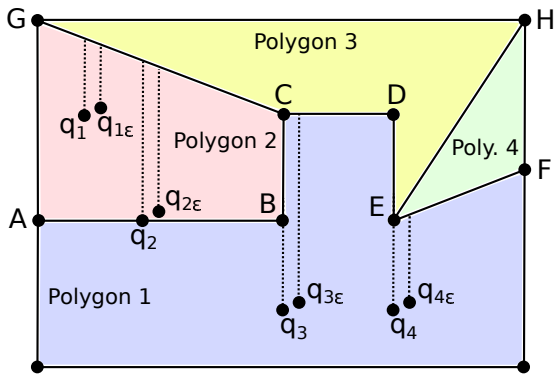


Figure 5.6: Using SoS for avoiding special cases in the 2D version of the point location problem.

3. The computational overhead of processing the perturbed objects should be negligible.

We claim that the perturbation scheme used in our work is suitable for the point location problem and satisfies the three conditions presented above. Condition 2 is automatically satisfied for a sufficiently small ϵ, that is, if the lowest mesh point above a query point q is in the interior of a triangle, the lowest point on the mesh directly above q_ϵ will also in the interior of the same triangle. Similarly, if q is not inside the mesh, q_ϵ will also not be in the mesh. As it will be shown later, the computational overhead of processing q_ϵ instead of q is negligible and, thus, condition 3 is also met.

To show that condition 1 is also satisfied we need to show that q_ϵ will be a non-degenerate input, that is, q_ϵ will never be exactly below a vertex or edge of the triangles and, also, q_ϵ will never be on the mesh surface.

First, let us show that q_ϵ will never be on the mesh surface. If q is on a mesh triangle t, q_ϵ cannot be on the plane Π passing through t for the following reason: suppose Π has a plane equation: $ax + by + cz + d = 0$ and both q and q_ϵ are on Π. Since q is on Π, we have $aq_x + bq_y + cq_z + d = 0$ and, since q_ϵ is also on Π we have $a(q_x + \epsilon) + b(q_y + \epsilon^2) + c(q_z + \epsilon^3) + d = 0$. Thus, $a\epsilon + b\epsilon^2 + c\epsilon^3 = 0$ and, since $\epsilon > 0$, $a + b\epsilon + c\epsilon^2 = 0 \Rightarrow a = -b\epsilon - c\epsilon^2$. Because a, b and c cannot simultaneously be 0, b and c are not simultaneously 0 and, thus, a is infinitesimal (what is impossible since, by definition, an infinitesimal is smaller than any measurable value).

Also, q_ϵ cannot be on the surface of another triangle t' not intersecting q because a ball with infinitesimal radius centered on q cannot intersect t'.

Now, lets show that q_ϵ will never be exactly below a vertex or edge of the triangles. Consider the projections q' and q'_ϵ of, respectively, q and q_ϵ onto the plane $z = 0$. To show that q_ϵ will never be directly below a mesh vertex or edge we need to show that q'_ϵ will never be on the projection of any vertex or edge onto $z = 0$.

Lemma 5.4.1. *If q' is exactly on the projection v' of a mesh vertex v onto the plane $z = 0$, q'_ϵ cannot be on the projection of any vertex or edge.*

Proof. Since v' is a point and q' is translated by a positive distance, q'_ϵ cannot be on v'. Also, q'_ϵ cannot be on the projection u' of any other vertex u onto $z = 0$ since, otherwise, the distance between v' and u' would be proportional to $\sqrt{\epsilon^2 + \epsilon^4} = \epsilon\sqrt{1 + \epsilon^2}$, what is infinitesimal.

Furthermore, if q' is on v' then q'_ϵ cannot be on the projection e' of an edge e onto $z = 0$ for two reasons. First, if v' does not intersect e', the shortest distance between e' and v' should be a non-infinitesimal positive value, but the distance between q' and q'_ϵ is infinitesimal. Second, if v' intersects e' and q'_ϵ is on e', this means that q' and q'_ϵ should be on the same edge e'. However, q' and q'_ϵ could not be on the same edge because $q'_\epsilon = (q'_x + \epsilon, q'_y + \epsilon^2)$ and, thus, the slope of this edge would be infinitesimal. □

It is also straightforward from this previous Lemma that if q' is on the projection of an edge then q'_ϵ will not on the projection of a vertex.

It is worth mentioning that the property that q' and q'_ϵ could not be simultaneously on the same segment would not be true for any edge if q_ϵ was equal to $(q_x + \epsilon, q_y + \epsilon, q_z + \epsilon)$ since, in this case, q' and q'_ϵ could be simultaneously on a segment with slope 1. This shows that a careful choice of the infinitesimals is an important task for correctly developing a SoS strategy.

Lemma 5.4.2. *If q' is on the projection e' of a mesh edge e onto the plane $z = 0$, q'_ϵ will not be on the projection of any edge.*

Proof. As mentioned above, q' and q'_ϵ cannot be simultaneously on the same edge. Thus, we only need to show that if q' is on e', q'_ϵ will not be on the projection f' of another edge f onto $z = 0$.

If f' does not intersect e', the shortest distance between f' and e' should be a non-infinitesimal positive value and, thus, q'_ϵ could not be on e' (otherwise the shortest distance between e' and f' would be infinitesimal).

If f' intersects e' on a vertex v', q' could be either on v' or not. As mentioned in the previous lemma, if q' is on v', then q'_ϵ cannot be on an edge. If the smallest angle between e' and f' is not zero, it should be a positive non-infinitesimal and since the distance between q' and v' is also a positive non-infinitesimal value, then an infinitesimal disc centered on q' could not intersect the projection of any edge other than e'. Because the distance between q' and q'_ϵ is infinitesimal, it follows that q'_ϵ cannot be on f'.

If the smallest angle between e' and f' is zero and the two edges intersect, it is clear that if q' is on e', then q'_ϵ could not be on f' (otherwise the slope of these edges would be infinitesimal). □

To conclude, as explained above, all query points q will be translated to a position q_ϵ that is not on the mesh and either below the interior of a triangle or not below any triangle. Notice that, since a perturbed point q_ϵ will never be directly below a vertex or edge, q_ϵ will also never directly below a vertical triangle (that is, a triangle whose normal is parallel to the plane $z = 0$).

5.4.3 Implementing the symbolic perturbations

The only parts of PinMesh that directly deal with the point coordinates, and thus need to be adapted to use SoS are the three main operations described in Section 5.2, that is, the functions $isOnProj(q, t)$, $isAbove(q, t)$ and $isBelow(q, t, t')$. In this section we will present details of how these functions were implemented and how they where adapted to treat the special cases using SoS.

Given a triangle t and a point q, the function $isOnProj(q, t)$ uses the q's barycentric coordinates to determine whether or not the projection of q onto the plane passing through t is on t. More specifically, we project both q and t onto

the plane $z = 0$, creating the point q' and the triangle t', compute the barycentric coordinates of q' with respect to t' and, then, use these coordinates to check whether or not q' is in the interior of t'.

Consider the three vertices t'_0, t'_1 and t'_2 of t' and the point q'. The barycentric coordinates λ_0,λ_1,λ_2 of q' with respect to t' can be computed using the following equations:

$$\lambda_0 = \frac{(t'_{1y} - t'_{2y}) \times (q'_x - t'_{2x}) + (t'_{2x} - t'_{1x}) \times (q'_y - t'_{2y})}{det} \tag{5.1}$$

$$\lambda_1 = \frac{(t'_{2y} - t'_{0y}) \times (q'_x - t'_{2x}) + (t'_{0x} - t'_{2x}) \times (q'_y - t'_{2y})}{det} \tag{5.2}$$

$$\lambda_2 = 1 - \lambda_0 - \lambda_1 \tag{5.3}$$

$$det = (t'_{1y} - t'_{2y}) \times (t'_{0x} - t'_{2x}) + (t'_{2x} - t'_{1x}) \times (t'_{0y} - t'_{2y}) \tag{5.4}$$

In order to q' be in the interior of t', the following condition must be met: $0 < \lambda_i < 1$, for $i = 0, 1, 2$. The degenerate cases for this test happens when $det = 0$ (this means the normal of the original triangle t is parallel with respect to the plane $z = 0$ and, therefore, t is a vertical triangle), when one of the λ_i is 1 (this means q' is on one of the vertices of t') or when one of the λ_i is 0 and the others are not 1 (this means q' is on one of the edges of t').

As mentioned in Section 5.4.2, a perturbed vertex will never be exactly below a vertical triangle and, therefore, if $det = 0$ the function $isOnProj(t, q)$ should return false. Therefore, the only special cases that need to be considered are the ones that happen when at least one of the λ_i is either 0 or 1.

If q is perturbed creating the new point $q_\epsilon = (q_x + \epsilon, q_y + \epsilon^2, q_z + \epsilon^3)$, the projection q'_ϵ of q_ϵ onto the plane $z = 0$ will be equal to $(q_x + \epsilon, q_y + \epsilon^2)$. Replacing q' with q'_ϵ in the equations 5.1,5.2 and 5.3 we have the following barycentric coordinates of the perturbed points:

$$\lambda_{\epsilon_0} = \lambda_0 + \frac{(t'_{1y} - t'_{2y}) \times \epsilon + (t'_{2x} - t'_{1x}) \times \epsilon^2}{det} \tag{5.5}$$

$$\lambda_{\epsilon_1} = \lambda_1 + \frac{(t'_{2y} - t'_{0y}) \times \epsilon + (t'_{0x} - t'_{2x}) \times \epsilon^2}{det} \tag{5.6}$$

$$\lambda_{\epsilon_2} = 1 - \lambda_{\epsilon_0} - \lambda_{\epsilon_1} \tag{5.7}$$

As expected, because of SoS, λ_{ϵ_0} will never be 0 or 1: λ_{ϵ_0} is equal to λ_0 plus an expression containing an infinitesimal and this expression can never be zero since, otherwise, we would have $t'_{1y} = t'_{2y}$ and $t'_{2x} = t'_{1x}$ (what would imply in $det = 0$). This same observation is also valid for λ_{ϵ_1} and λ_{ϵ_2}.

Therefore, if the function $isOnProj(t, q)$ is implemented such that it returns true iff $0 < \lambda_{\epsilon_i} < 1$, for $i = 1, 2, 3$, no degeneracy will happen. As it can be seen below, this implementation will be as efficient as an implementation that does not consider the special cases (and, thus, does not deal with the infinitesimals), satisfying the requirement 3 of SoS mentioned in Section 5.4.2.

For example, consider the problem of verifying the following predicate $0 < \lambda_{\epsilon_0} < 1$:

- if $\lambda_0 \neq 0$ *or* 1 (what is expected to happen most of the time): it is clear that $0 < \lambda_{\epsilon_0} < 1 \iff 0 < \lambda_0 < 1$.

- if $\lambda_0 = 0$: $\lambda_{\epsilon_0} < 1$ and, thus, we only need to check if $0 < \lambda_{\epsilon_0}$. Considering that $det > 0$ (if $det < 0$ the value of this predicate needs to be negated), $0 < \lambda_{\epsilon_0} \iff (t'_{1y} - t'_{2y}) \times \epsilon + (t'_{2x} - t'_{1x}) \times \epsilon^2 > 0$, what happens if $t'_{1y} > t'_{2y}$ or if $(t'_{1y} = t'_{2y})$ *and* $(t'_{2x} > t'_{1x})$ (since $\epsilon^2 << \epsilon$).

- if $\lambda_0 = 1$: $\lambda_{\epsilon_0} > 0$ and, thus, we only need to check if $\lambda_{\epsilon_0} < 1$. Considering that $det > 0$ (again, if $det < 0$ the value of this predicate needs to be negated), $\lambda_{\epsilon_0} < 1 \iff (t'_{1y} - t'_{2y}) \times \epsilon + (t'_{2x} - t'_{1x}) \times \epsilon^2 < 0$, what happens if $t'_{1y} < t'_{2y}$ or if $(t'_{1y} = t'_{2y})$ *and* $(t'_{2x} < t'_{1x})$.

This same strategy can be used to implement the functions $isAbove(t, q)$ and $isBelow(t, t', q)$. Besides the three functions mentioned in this section, the only

other step of PinMesh that deals with the coordinates of the query points is the operation of determining in what uniform grid cell c a query point q is. The only possibility of degeneracy in this operation happens when q is exactly on the border of a cell. This case is treated by considering that the point is in the cell with greatest index (that is, if a point is on the border between a cell with x index 8 and a cell with x index 9, we consider it is on the cell with index 9), what is consistent with the perturbation presented in this section (that adds positive infinitesimals to all coordinates).

5.5 Experimental evaluation

We implemented PinMesh in C++ using the packages GMP [77] to provide multiple precision rational numbers and OpenMP 4.0 to provide shared memory parallel programming constructors. Our implementation was compiled using g++ 4.9.3 (with the -O3 optimization flag) and tested on a workstation with a dual Xeon 3.1GHz E5-2687 processor (each one with 8 physical cores), 128 GiB of RAM and running the Linux Mint 17 operating system. Since PinMesh is parallel, unless otherwise noted it was configured to use 16 threads in the performance experiments.

PinMesh was compared with RCT, a sequential point location algorithm proposed by Liu et al.[29]. Differently from PinMesh, RCT is based on floating-point arithmetic and, thus, it is not guaranteed to always locate the points exactly. As far as we know, RCT is the most efficient available algorithm for the point location problem. According to the experiments performed by Liu et al.[29], their algorithm was much faster than other algorithms such as the one implemented in CGAL and the AABB-tree-based algorithm proposed by Baerentzen et al.[81]. RCT's C++ source code was made available by the authors [29] and, thus, we were able to compile and run RCT using the same platform we used to evaluate PinMesh.

Experiments were performed on 13 datasets, with sizes ranging from one hundred thousand triangles to fifty million triangles, as described in Table 5.1. Some of them were downloaded from the Stanford Scanning Repository [82], while others were downloaded from the Large Geometric Model Archive [83] and from the AIM@SHAPE-VISIONAIR Shape Repository [84]. Table 5.1 also includes the cre-

ator of each model downloaded from the AIM@SHAPE-VISIONAIR Shape Repository. Figure 5.7 illustrates some of these meshes.

The ten smallest meshes are single-material meshes, that is, meshes containing only one object while the three largest ones are multi-material. The *6 Materials* dataset was created by joining the six largest single-material meshes used in the experiments side-by-side (creating a grid with $3x2$ meshes, where each grid cell has size equal to the bounding-box of the largest mesh and there is no intersection between the triangles bounding different materials). The datasets with 12 and 24 materials, on the other hand, were generated by joining, respectively, 2 and 4 copies of each object used in the *6 Materials* dataset.

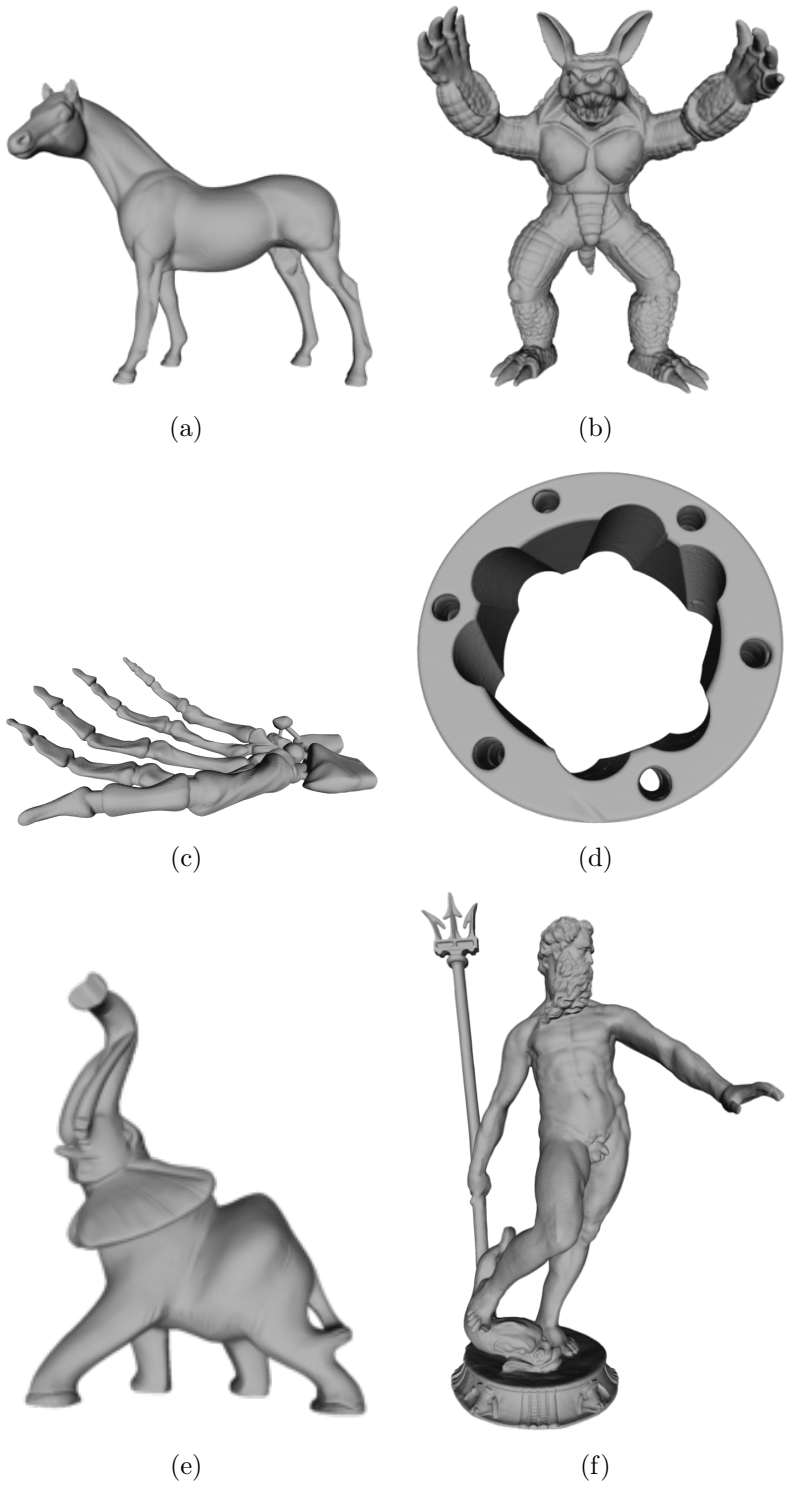
Table 5.1: Datasets used in the experiments.

Dataset	Source	Creator	Vertices	Triangles
Horse	GIT	-	48,485	96,966
Armadillo	Stanford	-	172,974	345,944
Hand	GIT	-	327,323	654,666
Pierrot	AIM@SHAPE	Frank_terHaar	443,805	887,606
Chinese dragon	AIM@SHAPE	Laurent_Saboret	655,980	1,311,956
Rolling stage	AIM@SHAPE	INRIA	660,267	1,320,558
Buddha	AIM@SHAPE	VCG-ISTI	719,553	1,439,102
Ramesses	AIM@SHAPE	Marco_Attene	826,266	1,652,528
Elephant	AIM@SHAPE	ISTI	1,512,290	3,024,588
Neptune	AIM@SHAPE	Laurent_Saboret	2,003,932	4,007,872
6 Materials	-	-	6,378,288	12,756,604
12 Materials	-	-	12,756,576	25,513,208
24 Materials	-	-	25,513,152	51,026,416

5.5.1 Correctness evaluation

PinMesh is simple enough that its correctness is more obvious than would be the case with a more complicated algorithm, such as a topological sweep line. In addition, two strategies were used to verify the correctness of our implementation.

1. Points randomly positioned in the bounding-box of the objects were queried using PinMesh and RCT. Since it is improbable that point location queries with random points are incorrectly computed (because of floating point er-



(a) (b)

(c) (d)

(e) (f)

Figure 5.7: Illustration of some datasets used in the experiments - Horse (a), Armadillo (b), Hand (c), Rolling Stage (d), Elephant (e) and Neptune (f). These figures were renderized using MeshLab.

rors or special cases not treated), it is expected that the results obtained by PinMesh are equal to the ones obtained by RCT.

2. Experiments with query points positioned to represent special cases were used to test if PinMesh correctly located the points.

In our experiments we evaluated millions of queries and PinMesh correctly handled all of them. Since we are carefully treating all the singularities using SoS and, because of the exact arithmetic, PinMesh does not have any roundoff error, we believe that it can correctly handle any valid input.

We also generated a mesh that represents a challenge to the point locations algorithms. This mesh, illustrated in Figure 5.8, consists of two pyramids with height 10, each one representing different regions. The top one has id 2 while the bottom one has id 1 and the two pyramids intersect on the vertex $v = (0, 0, 10)$ (this point is shared among 4 triangles of the bottom pyramid and 4 triangles of the top pyramid).

If a query point q is positioned directly below v (for example, in point $(0, 0, 9)$), the vertical ray used by PinMesh will intersect the mesh in v and, thus, it will need to correctly choose a triangle that actually bounds the region where v is (and not the other region). In our experiments PinMesh successfully handled this special case. The RCT algorithm, on the other hand, was not able to correctly compute the region containing q for the following reason: in this mesh, the RCT will try to find the closest triangle to q. Since all the 8 triangles containing v are at the same distance from q (that is, they are the closest ones to q) and since the RCT does not perform any treatment to disambiguate this computation, the first of these 8 triangles found by the algorithm is used to compute q's position. If the triangles bounding the region 2 are stored before the triangles bounding the region 1, the RCT will choose a triangle in region 2's boundary to locate q and, thus, return an incorrect output.

5.5.2 Performance evaluation

For each test dataset, we created a set of query points containing 500,000 points randomly and uniformly distributed inside the mesh and 500,000 points randomly

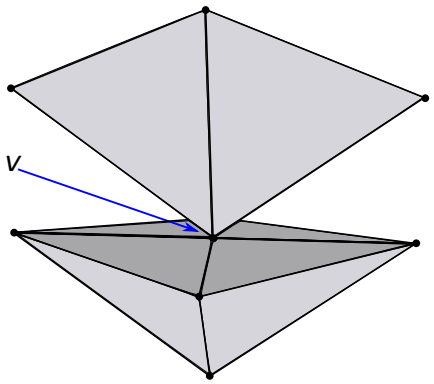


Figure 5.8: Example of dataset representing a special case.

distributed outside the mesh, but inside its bounding-box. These same points were used to evaluate the performance of RCT and PinMesh. Our reported running times represent the average wall-clock time of 10 runs.

The uniform grid used by PinMesh was created having a resolution of 64^3 cells in the first level and each cell containing more than 1 triangle was refined. The resolution of the second level grid was chosen such that the expected number of triangles per cell is 0.0005. More specifically, the resolution of each second level grid is $(\sqrt[3]{t/(64^3 \times 0.0005)})^3$, where t is the number of triangles in the mesh.

Table 5.2 compares the times spent by PinMesh with the time spent by the RCT algorithm. If we consider the total running-time (that is, the pre-processing and time to perform one million queries), PinMesh was up to 27 times faster than RCT. As it can be seen in this table, the speedup of PinMesh improves as the size of the mesh increases, indicating that it scales better than RCT.

The main performance advantage of PinMesh over RCT is in the pre-processing time, that is the bottleneck of the two algorithms even when 1 million queries are performed. Because of a careful implementation of the uniform grid and of the use of parallel programming, PinMesh constructs the index much faster than RCT, being up to 28 times faster. If we consider the query times, the difference between RCT and PinMesh is smaller and PinMesh is up to 8 times faster than RCT.

Figure 5.9 illustrates how the processing time of PinMesh and RCT behaves as a function of the number of triangles in the mesh. As it can be seen in Figure 5.9 (a), the pre-processing step of both algorithms scales linearly, but PinMesh is much

faster than RCT. Considering the query time (Figure 5.9 (b)), it is easy to see that the time spent by PinMesh changes very slowly as the number of triangles increase (being almost constant).

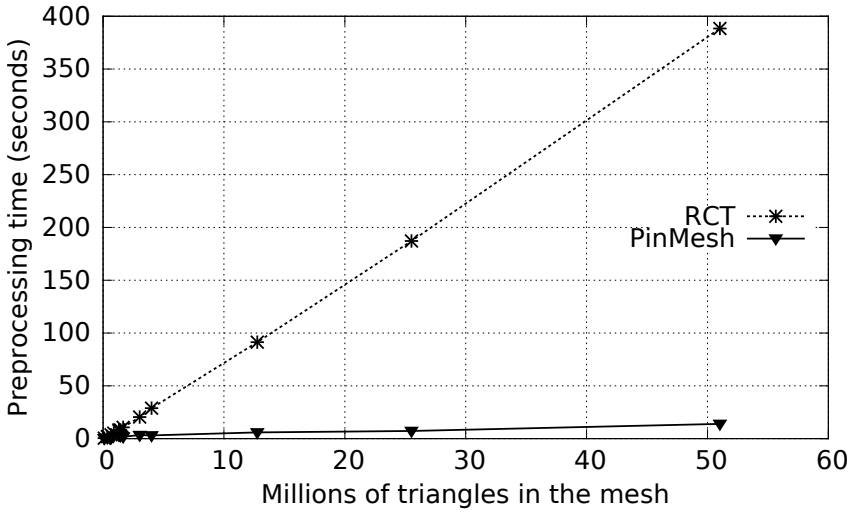
To evaluate PinMesh's parallel performance, we performed experiments considering a varying amount of threads. Table 5.3 presents the times (in seconds) spent by the main steps of PinMesh during the processing of the *24 Materials* dataset, the largest mesh used in the experiments.

Even when only 1 thread is used, PinMesh was able to pre-process the dataset faster than RCT. Indeed, while RCT (that is sequential) spent 388 seconds to pre-process the *24 Materials* dataset and 2.55μs to perform each query, PinMesh processed it in 64 seconds and spent 6.61μs per query when only one thread was used. Since the pre-processing time is usually big compared with the query time, the total time spent by the RCT algorithm is only smaller than the total time spent by PinMesh running with 1 thread when more than 80 million points are queried in this dataset. When 4 or more threads were used, PinMesh was faster than RCT in both the pre-processing step and during the queries.

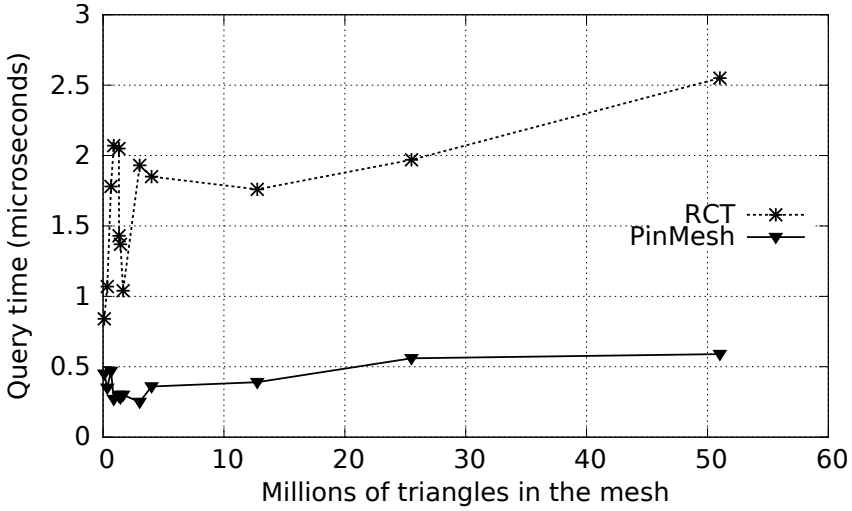
Table 5.2: Experimental pre-processing and query times for PinMesh and RCT.

	N_t[a]		PinMesh		RCT	
Mesh	$\times 10^3$	G_2[b]	T_p(s)[c]	$T_q(\mu s)$[d]	T_p(s)[c]	$T_q(\mu s)$[d]
Horse	97	9^3	0.30	0.45	0.42	0.84
Armadillo	346	14^3	0.86	0.35	1.88	1.07
Hand	655	17^3	0.78	0.47	3.64	1.78
Pierrot	888	19^3	2.83	0.27	5.24	2.07
Rolling Stage	1,312	22^3	3.73	0.30	8.27	2.05
Chinese Dragon	1,321	22^3	3.34	0.29	7.31	1.43
Buddha	1,439	22^3	2.89	0.28	8.20	1.37
Ramesses	1,653	23^3	1.84	0.30	10.72	1.04
Elephant	3,025	28^3	3.38	0.25	20.50	1.93
Neptune	4,008	31^3	3.07	0.36	28.88	1.85
6 Materials	12,757	46^3	5.86	0.39	91.34	1.76
12 Materials	25,513	58^3	7.31	0.56	187.18	1.97
24 Materials	51,026	73^3	14.00	0.59	388.38	2.55

[a] number of triangles in each dataset;
[b] size of the second level uniform grid; [c] data set pre-processing time, in seconds; [d] query time, per point, in microseconds.



(a)



(b)

Figure 5.9: Comparing the preprocessing (a) and average query (b) times spent by PinMesh and RCT.

The slowest pre-processing steps of PinMesh when only one thread is used are the initial computation of the grid cells where each mesh vertex is and the refinement of the grid. As mentioned in Section 5.4.1, these steps are easily parallelizable and, as it can be seen in Table 5.3, they are the pre-processing steps that better scale as the number of threads increases.

The creation of the first level uniform grid, on the other hand, is the fastest

step when PinMesh runs using only one thread. This step was not parallelized since it consists basically in inserting the ids of the triangles into the relevant positions of the ragged array representing the uniform grid, what is a very memory intensive process that would require synchronizations to be implemented in parallel.

PinMesh presents a good scalability mainly when the number of threads ranges from 1 to 8. For example, when the number of threads is increased from 1 to 2, the total running time is reduced by 41%. This scalability happens because the slowest steps are the ones that scale better. It is worth mentioning that the Xeon processor used in the experiments supports the Intel® Turbo Boost technology [78] to increase its frequency from their 3.1GHz base operating frequency to up to 3.8GHz. However, to keep from overheating the CPU, when more cores are active, less Turbo Boosting is allowed. Therefore, one cannot expect perfect scalability even in a completely parallelizable function.

When the number of threads increases from 8 to 16, on the other hand, the total running time is reduced by 20%. This smaller reduction happens mainly because of two reasons. First, because of Amdahl's law, the percentage of time spent performing operations that do not scale (such as memory allocations) and running the steps that were not implemented in parallel increases as the number of threads increase. Second, since some of the steps are very memory intensive we believe that they saturate the processor's memory when 16 threads are used.

Table 5.3: Time spent by the main steps of PinMesh. Experiments performed on the *24 Materials* dataset using a varying number of threads. The query time is the average time per query (in μs).

	Preprocessing (s)				Queries (μs)	
Threads	Locate gr. cells	Create 1^{st} level grid	Refine grid	Label empty gr. cells	Comp. query points grids	Locate points
1	31.92	4.04	15.08	12.57	1.30	5.31
2	16.43	4.34	8.52	8.44	0.74	2.85
4	8.84	4.41	4.85	5.88	0.39	1.54
8	4.60	4.13	2.86	5.54	0.21	0.80
16	2.49	4.04	2.00	5.46	0.13	0.46

5.6 Summary

In this section we presented PinMesh, an exact and efficient algorithm for locating points in 3D meshes. PinMesh was carefully implemented to always handle point location queries correctly. The use of rational numbers to store and process the 3D coordinates completely avoids problems caused because of roundoff errors typically present in algorithms implemented using floating-point arithmetic. Another typical source of errors in 3D algorithms, the treatment of special cases, was also carefully handled in PinMesh using Simulation of Simplicity.

PinMesh is not only exact, but also very efficient. The use of efficient data structures associated with parallel programming made PinMesh very fast. According to our experiments, PinMesh was up to 27 times faster than the RCT algorithm (that was, to the best of our knowledge, the fastest 3D point location algorithm available).

CHAPTER 6
3D mesh intersection

This chapter presents 3D-EPUG-Overlay, our algorithm for exactly intersecting 3D meshes. Its input is composed of two triangular meshes M_0 and M_1. Each mesh contains a set of 3D triangles as described in Section 2.2.2 and represents 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.

We incorporated in 3D-EPUG-Overlay all the optimization techniques we previously tested on EPUG-Overlay and on PinMesh. Furthermore, two techniques (that have not been applied to 3D-EPUG-Overlay and to EPUG-Overlay yet) were employed to accelerate 3D-EPUG-Overlay: arithmetic filters to accelerate the exact computation and a better parallelization of the uniform grid creation.

Portions of this chapter previously appeared as:

S. V. G. Magalhães, "An efficient algorithm for computing the exact overlay of triangulations", in *Proc. 2nd ACM SIGSPATIAL PhD Workshop*, SIGSPATIAL PhD'15. New York, NY, USA: ACM, 2015, pp. 3:13:4.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, and W. Li,"Pinmesh - Fast and exact 3D point location queries using a uniform grid", *Comput. & Graph.*, vol. 58, pp. 1 - 11, 2016, Shape Modeling International 2016.

S. V. G. Magalhães, M. V. A. Andrade, W. R. Franklin, M. G. Gruppi and W. Li, "Exact intersection of 3D geometric models", in *Proc. XVII Brazilian Symp. Geoinformatics*, GeoInfo'16, 2016, pp. 44 - 55.

S. V. G. Magalhães, W. R. Franklin, and M. V. A. Andrade, "Fast exact parallel 3D mesh intersection algorithm using only orientation predicates", *Proc. 17th ACM SIGSPATIAL Int. Conf. Advances in Geographic Information Systems*, SIGSPATIAL'17. New York, NY, USA:ACM, 2017.

6.1 Data representation

The program input is a pair of triangular meshes in 3D (E^3) as described in Section 2.2.2 (both meshes must be watertight and free from self-intersections). The polyhedra may have complex topologies, with holes and disjoint components. The two meshes have a nonempty intersection (else the output is trivial) and often are identical (i.e., represent the same polyhedra). Indeed, intersecting two identical meshes is an excellent stress test, because of all the degeneracies.

Two types of vertices are processed by the algorithm: (1) input vertices occurring in the input meshes, and (2) intersection vertices resulting from intersections between an edge of one mesh and a triangle of the other.

Each input vertex stores two ids: the vertex id and the id of the mesh to where it belongs (since meshes are processed in pairs, we assume there are two possible mesh ids: 0 or 1). Input vertices in the same mesh sharing the same coordinates are considered identical (i.e., they share the same ids).

Similarly to the vertices, there are two types of triangles: input triangles and triangles from retesselation. The first one contains only input vertices while the second one may contain vertices generated from intersections, which are created during the retesselation of input triangles.

A vertex from intersection is represented by the edge (composed of two input vertices) and the input triangle whose intersection generated it. Even though it is possible to compute the coordinates of these vertices using the information about how they were generated, we have an additional data structure that caches these coordinates to avoid recomputation.

As will be mentioned later, distinguishing input vertices from vertices generated from intersections is important for the implementation of the symbolic perturbation.

6.1.1 The symbolic perturbation

Edelsbrunner [4] uses the convex hull problem as example of application of Simulation of Simplicity. In the 2D version of the problem, the j-th coordinate of the i-th vertex was perturbed by translating it by $\epsilon^{2^{2i-j}}$. As shown by the authors,

even if all the points were originally coincident, after the perturbation no three points would be collinear, which completely eliminates degeneracies for the convex hull problem.

In contrast, for the mesh intersection problem, we assume that the individual meshes are not degenerate, and coincidences happen only when they are processed together. For example, two triangles from the same mesh only intersect at common edges or vertices but the interior of a triangle from one mesh can intersect a co-planar triangle from the other mesh, which represents a special case of the intersection algorithm.

Allowing an individual mesh M to be degenerate is challenging because it is hard to develop a perturbation scheme that ensures M will be consistent after the perturbation. For example, M could contain two triangles t_1 and t_2 where all six vertices coincide. Even though a perturbation scheme such as the one employed by Edelsbrunner [4] for the 3D orientation would ensure no two vertices will coincide anymore, there is no guarantee that the information about the tetrahedra on each side of the triangles will be consistent after the perturbation.

For the rest of this chapter the following notation will be employed. The two input meshes will be represented by M_0 and M_1 (or simply by mesh 0 or mesh 1) and the corresponding perturbed meshes will be, respectively, $M_{0\epsilon}$ and $M_{1\epsilon}$. All the perturbed geometric objects (vertices, edges and triangles) will be followed by an ϵ subscript (for example, vertex v_ϵ is the perturbed version of vertex v).

We propose the following perturbation scheme for the 3D mesh intersection problem:

1. Do not modify mesh 0.

2. Translate each vertex of mesh 1 equally by the vector $(\epsilon, \epsilon^2, \epsilon^3)$, i.e., $v_{j\epsilon} = (v_{jx} + \epsilon, v_{jy} + \epsilon^2, v_{jz} + \epsilon^3)$.

Now, no vertex from $M_{0\epsilon}$ can coincide with any vertex from $M_{1\epsilon}$ (since vertices from $M_{0\epsilon}$ do not have infinitesimal components).

The rest of this section will present lemmas describing properties of the perturbed meshes. As will be shown later, a consequence of these lemmas is that there

will be no special case during the intersection of $M_{0\epsilon}$ with $M_{1\epsilon}$.

Lemma 6.1.1. *If a_ϵ is a vertex from mesh i (i is 0 or 1) and t_ϵ is a triangle from mesh $1-i$, then a_ϵ and t_ϵ are not coplanar.*

Proof. Since a_ϵ is in mesh i and t_ϵ is in mesh $(1-i)$ (for $i = 0$ or 1), then, $a_\epsilon = (a_x, a_y, a_y) + (i\epsilon, i\epsilon^2, i\epsilon^3)$ and each vertex of t_ϵ is represented by $t_{j\epsilon} = (t_{jx}, t_{jy}, t_{jz}) + ((1-i)\epsilon, (1-i)\epsilon^2, (1-i)\epsilon^3)$ $(j = 0...2)$.

Let $t_{0\epsilon} = (t_{0x} + (1-i)\epsilon, t_{0y} + (1-i)\epsilon^2, t_{0z} + (1-i)\epsilon^3)$, $e = (e_x, e_y, e_z) = t_{1\epsilon} - t_{0\epsilon}$ and $f = (f_x, f_y, f_z) = t_{2\epsilon} - t_{0\epsilon}$. Observe that, because of the subtraction, neither e nor f do not contain infinitesimal terms.

The plane containing t can be represented by the parametric equation:

$$t_{0\epsilon} + se + uf = X \tag{6.1}$$

where s and u are parameters.

Suppose, that a_ϵ and t_ϵ were coplanar. Then, there would exist two scalars s and u such that:

$$t_{0\epsilon} + se + uf = (a_x + i\epsilon, a_y + i\epsilon^2, a_y + i\epsilon^3) \tag{6.2}$$

Since all the coordinates of f and e are finite numbers and either $t_{0\epsilon}$ or a_ϵ contain infinitesimal terms (since they are from different meshes and only one of the meshes is translated by infinitesimals), then s and u are ϵ-polynomials. Let $s = s_0 + s_\epsilon$ and $u = u_0 + u_\epsilon$, where s_0 and u_0 represent, respectively, the rational monomials (i.e., the ϵ-monomials of degree 0) of s and u. Thus, s_ϵ and u_ϵ represent the higher order ϵ-terms of s and u. Then:

$$t_0 + (1-i)(\epsilon, \epsilon^2, \epsilon^3) + (s_0 + s_\epsilon)e + (u_0 + u_\epsilon)f = a + (i)(\epsilon, \epsilon^2, \epsilon^3) \tag{6.3}$$

Let $r = a - t_0 - s_0 e - u_0 f$, then:

$$s_\epsilon e + u_\epsilon f = (2i-1)(\epsilon, \epsilon^2, \epsilon^3) + r \tag{6.4}$$

Since r, e and f contain only rational coordinates and s_ϵ and u_ϵ are ϵ-polynomials where all the monomials contain degree larger than 0, then $r = (0, 0, 0)$ and Equation 6.4 can be rewritten as:

$$s_\epsilon(e_x, e_y, e_z) + u_\epsilon(f_x, f_y, f_z) = (2i-1)(\epsilon, \epsilon^2, \epsilon^3) \tag{6.5}$$

We have to show that Equation 6.5 cannot be satisfied, which will make the assumption that a_ϵ and t_ϵ are coplanar contradictory.

If either s_ϵ or u_ϵ are 0, then it is clear that Equation 6.5 cannot be satisfied since the product of an ϵ-polynomial and a rational vector cannot be a vector where each term has a different degree.

Furthermore, no term of e or f can be 0. For example, if $e_x = 0$, then $u_\epsilon = \frac{\epsilon}{f_x}$ (f_x and e_x cannot simultaneously be 0 since $s_\epsilon e_x + u_\epsilon f_x = (2i-1)\epsilon$) and $s_\epsilon e_y + \epsilon\frac{f_y}{f_x} = (2i-1)\epsilon^2$. Since e_y and $\frac{f_y}{f_x}$ are rational numbers, then the degree of s_ϵ is 2. Because the degree of u_ϵ is 1, the degree of s_ϵ is 2 and the right side of Equation 6.5 has an ϵ-coefficient of degree 3, then the equation cannot be satisfied.

Now, we have to show that Equation 6.5 cannot be satisfied when $s_\epsilon \neq 0$, $u_\epsilon \neq 0$ and no term of e or f is 0.

From Equation 6.5, it follows that

$$u_\epsilon = \frac{(2i-1)\epsilon - s_\epsilon e_x}{f_x} = \frac{(2i-1)\epsilon^2 - s_\epsilon e_y}{f_y} \tag{6.6}$$

$$u_\epsilon = \frac{(2i-1)\epsilon^2 - s_\epsilon e_y}{f_y} = \frac{(2i-1)\epsilon^3 - s_\epsilon e_z}{f_z} \tag{6.7}$$

Since the coordinates of e and f are rational numbers, Equation 6.6 above implies that s_ϵ is a polynomial of degree 2 while Equation 6.7 implies that s_ϵ has degree 3, which is a contradiction.

Therefore, the assumption that a vertex a_ϵ from mesh i (for $i = 0$ or 1) and a triangle t_ϵ from mesh $(1 - i)$ can be coplanar is an absurdity.

□

Lemma 6.1.2. *Given an edge $e_{1\epsilon} = a_\epsilon b_\epsilon$ ($a_\epsilon \neq b_\epsilon$), i.e., the endpoints are a_ϵ and b_ϵ, from mesh i and another edge $e_{2\epsilon} = c_\epsilon d_\epsilon$ ($c_\epsilon \neq d_\epsilon$) from mesh $1-i$ such that $e_{1\epsilon}$ and $e_{2\epsilon}$ are not parallel, then $e_{1\epsilon}$ and $e_{2\epsilon}$ do not intersect.*

Proof. W.l.o.g., assume $e_{1\epsilon}$ is in mesh 0. Then, $a_\epsilon = (a_x, a_y, a_z)$, $b_\epsilon = (b_x, b_y, b_z)$, $c_\epsilon = (c_x + \epsilon, c_y + \epsilon^2, c_z + \epsilon^3)$ and $d_\epsilon = (d_x + \epsilon, d_y + \epsilon^2, d_z + \epsilon^3)$.

A necessary condition for their intersection of $e_{1\epsilon}$ and $e_{2\epsilon}$ is the coplanarity of a_ϵ, b_ϵ, c_ϵ and d_ϵ.

Let Π be the plane containing a_ϵ, b_ϵ and c_ϵ, a point (x, y, z) will be on Π iff:

$$\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x + \epsilon & c_y + \epsilon^2 & c_z + \epsilon^3 & 1 \\ x & y & z & 1 \end{vmatrix} = 0 \tag{6.8}$$

Assuming d_ϵ is on Π, if (x, y, z) is replaced in the determinant with d_ϵ, the resulting equation obtained after the determinant is expanded will have the following format:

$$D_0 + D_1\epsilon + D_2\epsilon^2 + D_3\epsilon^3 = 0 \tag{6.9}$$

, where:

$$\begin{aligned} D_0 = {} & b_z(c_y d_x - c_x d_y) + a_z(c_x d_y - c_y d_x + b_y(d_x - c_x) + b_x(c_y - d_y)) + \\ & b_y(c_x d_z - c_z d_x) + b_x(c_z d_y - c_y d_z) + a_x(c_y d_z - c_z d_y + b_z(d_y - c_y) + \\ & b_y(c_z - d_z)) + a_y(b_z(c_x - d_x) + c_z d_x - c_x d_z + b_x(d_z - c_z)), \\ D_1 = {} & (a_y - b_y)(c_z - d_z) - (a_z - b_z)(c_y - d_y), \\ D_2 = {} & (a_z - b_z)(c_x - d_x) - (a_x - b_x)(c_z - d_z), \\ D_3 = {} & (a_x - b_x)(c_y - d_y) - (a_y - b_y)(c_x - d_x) \end{aligned} \tag{6.10}$$

Equation 6.9 has the following four solutions:

$$\left\{ d_y = \frac{(a_x - b_x)c_y - (a_y - b_y)(c_x - d_x)}{a_x - b_x}, d_z = \frac{(a_x - b_x)c_z - (a_z - b_z)(c_x - d_x)}{a_x - b_x} \right\} \tag{6.11}$$

$$\left\{ b_x = a_x, d_x = c_x, d_z = \frac{(a_y - b_y)c_z - (a_z - b_z)(c_y - d_y)}{a_y - b_y} \right\} \tag{6.12}$$

$$\{b_x = a_x, b_y = a_y, d_x = c_x, d_y = c_y\} \tag{6.13}$$

$$\{b_x = a_x, b_y = a_y, b_z = a_z\} \tag{6.14}$$

However, the line segments $e_{1\epsilon} = a_\epsilon b_\epsilon$ and $e_{1\epsilon} = c_\epsilon d_\epsilon$ satisfying any of these four solutions are either parallel or have coincident endpoints, what contradicts the original assumptions of the lemma.

□

Lemma 6.1.3. *Given two distinct vertices a_ϵ and b_ϵ from mesh i and another vertex c_ϵ from mesh $1 - i$, then a_ϵ, b_ϵ and c_ϵ are not collinear.*

Proof. Assume the three distinct vertices $a_\epsilon = (a_x + i\epsilon, a_y + i\epsilon^2, a_z + i\epsilon^3)$, $b_\epsilon = (b_x + i\epsilon, b_y + i\epsilon^2, b_z + i\epsilon^3)$ and $c_\epsilon = (c_x + (1-i)\epsilon, c_y + (1-i)\epsilon^2, c_z + (1-i)\epsilon^3)$ were collinear. Then, $(b_\epsilon - a_\epsilon) \times (c_\epsilon - a_\epsilon) = (0, 0, 0)$.

If a_ϵ and b_ϵ are in mesh 0 (and, consequently, c_ϵ is in mesh 1), then $(b_\epsilon - a_\epsilon) \times (c_\epsilon - a_\epsilon)$ will be equal to the following vector:

$$\begin{bmatrix} (a_y(b_z - c_z) + a_z(c_y - b_y) - b_z c_y + b_y c_z) + (a_z - b_z)\epsilon^2 + (b_y - a_y)\epsilon^3 \\ (a_z(b_x - c_x) + a_x(c_z - b_z) - b_x c_z + b_z c_x) + (b_z - a_z)\epsilon + (a_x - b_x)\epsilon^3 \\ (a_x(b_y - c_y) + a_y(c_x - b_x) - b_y c_x + b_x c_y) + (a_y - b_y)\epsilon + (b_x - a_x)\epsilon^2 \end{bmatrix} \tag{6.15}$$

Similarly, if a_ϵ and b_ϵ are in mesh 1 (and, consequently, c_ϵ is in mesh 0), then $(b_\epsilon - a_\epsilon) \times (c_\epsilon - a_\epsilon)$ will be equal to the following vector:

$$\begin{bmatrix} (a_y(b_z - c_z) + a_z(c_y - b_y) - b_z c_y + b_y c_z) + (b_z - a_z)\epsilon^2 + (a_y - b_y)\epsilon^3 \\ (a_z(b_x - c_x) + a_x(c_z - b_z) - b_x c_z + b_z c_x) + (a_z - b_z)\epsilon + (b_x - a_x)\epsilon^3 \\ (a_x(b_y - c_y) + a_y(c_x - b_x) - b_y c_x + b_x c_y) + (b_y - a_y)\epsilon + (a_x - b_x)\epsilon^2 \end{bmatrix} \tag{6.16}$$

The vectors 6.15 and 6.16 will be equal to $\vec{0}$ when all the epsilon-coefficient vanishes. A necessary condition for having the epsilon-coefficients equal to 0 is that $a = b$ (and, consequently, $a_\epsilon = b_\epsilon$ since both points are in the same mesh), what contradicts the assumption of this lemma that a_ϵ and b_ϵ are distinct points.

Therefore, a triple of vertex containing at least one vertex from each perturbed mesh cannot be collinear. □

Corollary 6.1.4. *An edge e_ϵ from mesh i cannot intersect a parallel edge f_ϵ from mesh $1 - i$.*

Proof. A necessary condition for the intersection is that one of the endpoints of e_ϵ are collinear w.r.t. f_ϵ, what contradicts Lemma 6.1.3. □

Lemma 6.1.5. *If an edge e_ϵ from a mesh i intersects a triangle t_ϵ from mesh $(1-i)$, then this intersection happens in the interior of t_ϵ.*

Proof. If e_ϵ and t_ϵ intersect on an edge $e_{t\epsilon}$ of t_ϵ, then either e_ϵ and an $e_{t\epsilon}$ are collinear or they are coplanar and nonparallel. Because of Corollary 6.1.4, if they are parallel then they cannot intersect. Furthermore, if e_ϵ and $e_{t\epsilon}$ are not parallel, because of Lemma 6.1.2 they cannot intersect. □

Lemma 6.1.6. *If e_ϵ is an edge from mesh i and t_ϵ is a triangle from mesh $1 - i$, then e_ϵ and t_ϵ are not coplanar.*

Proof. If e_ϵ and t_ϵ were coplanar, then t_ϵ and one of the vertices v_ϵ of e_ϵ would be coplanar, which contradicts Lemma 6.1.1. □

Lemma 6.1.7. *If $t_{i\epsilon}$ and $t_{(1-i)\epsilon}$ are triangles belonging to, respectively, meshes i and $1 - i$, then $t_{i\epsilon}$ and $t_{(1-i)\epsilon}$ are not co-planar.*

Proof. If $t_{i\epsilon}$ and $t_{(1-i)\epsilon}$ were coplanar, then $t_{i\epsilon}$ and an edge of $t_{(1-i)\epsilon}$ would be coplanar, which contradicts Lemma 6.1.6. □

While these lemmas could be still valid for some other perturbation schemes, they are not valid for all perturbations. For example, if mesh M_1 was translated by $(\epsilon, \epsilon, \epsilon)$ instead of $(\epsilon, \epsilon^2, \epsilon^3)$, then the following three points would be collinear:

$a_\epsilon = (0, 0, 0)$, $b_\epsilon = (1, 1, 1)$ and $c_\epsilon = (0 + \epsilon, 0 + \epsilon, 0 + \epsilon)$ (where a_ϵ and b_ϵ are in mesh $M_{0\epsilon}$ and c_ϵ is in mesh $M_{1\epsilon}$).

When the mesh intersection algorithm is described later, we will assume that it will process meshes perturbed as described in this section. Thus, the meshes will have all the properties mentioned in these lemmas.

6.2 Implementing intersection with simple geometric predicates

To simplify the implementation of the symbolic perturbation, we developed two versions of each geometric function employed in the mesh intersection algorithm. The first one focused on efficiency, and was implemented based on efficient algorithms available in the literature. The second one focused on simplicity, and was implemented using as few geometric predicates as possible.

The idea is that, during the computation, the first version of each function is called. If a special case is detected, then the second version is called. In order to make sure the special cases are properly handled we only need to implement the perturbation scheme on these predicates.

As will be seen, the second version of most functions was implemented with only orientation predicates. The only exceptions are the ones related to indexing the data; they are necessary only for performance, not for correct execution.

The main advantage of having the orientation predicates as a base for other functions is the simplicity of the implementation. It is usually much easier to implement symbolic perturbation with orientation predicates than with more complicated functions, such as using barycentric coordinates to detect if a point is in a triangle. Furthermore, the number of predicates that have to be adapted to handle the perturbation is smaller. Indeed, once the lower level predicates are adapted, then all the higher level functions will be automatically consistent with the perturbation scheme.

Since the functions we employed are traditional computational geometry functions (such as point in triangle tests or triangle-triangle intersection detection), details about the first version of the geometric functions will not be presented. Fur-

thermore, as mentioned before, the first versions of the functions are used only for speed. Our algorithm can be correctly implemented without them.

6.2.1 Orientation predicates

From Edelsbrunner [4], the orientation of a sequence of $d+1$ points in E^d is "either negative or positive - unless the $d+1$ points lie on a common hyperplane, in which it is undefined". The orientation is

$$orientation(p_0, p_1, .., p_d) = sgn\left(\begin{vmatrix} p_{00} & p_{01} & \cdots & p_{0(d-1)} & 1 \\ p_{10} & p_{11} & \cdots & p_{1(d-1)} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ p_{d0} & p_{d1} & \cdots & p_{d(d-1)} & 1 \end{vmatrix}\right) \tag{6.17}$$

where p_{ij} is the j-th coordinate of point i and sgn is the sign function that returns -1 if the argument is negative, 1 if it is positive and 0 if it is 0. Thus, the number 0 will indicate a coincidence in the orientation predicate.

As will be shown later, the predicates employed by the 3D mesh intersection algorithm will be the 1D, 2D and 3D orientation. In 1D, $orientation(p_0, p_1) = sign(p_{00} - p_{10})$ will be positive if $p_0 > p_1$ and negative if $p_0 < p_1$, i.e., if p_1 is on the "left" side of p_0. In 2D, the $orientation(p_0, p_1, p_2)$ will be positive iff p_2 is on the left side of the directed line passing from p_0 to p_1. Finally, the 3D $orientation(p_0, p_1, p_2, p_3)$ is positive iff p_3 is on the positive side of an oriented triangle $p_0p_1p_2$.

These predicates will be adapted to handle the perturbed points, remaining consistent with the perturbation. For example, because of Lemma 6.1.1, the predicate $orientation(a_\epsilon, b_\epsilon, c_\epsilon, d_\epsilon)$ will never be 0 if $a_\epsilon b_\epsilon c_\epsilon$ is a triangle from one mesh and d_ϵ is a vertex of the other mesh.

Not all coincidences are eliminated by the perturbations. For example, because all vertices of the same mesh are translated by the same infinitesimals, then $orientation(a_\epsilon, b_\epsilon, c_\epsilon, d_\epsilon)$ may be 0 if all the points belong to the same input mesh. However, as will be shown later, in the few functions where these coincidences may

happen, the behavior of the algorithm is well defined and the coincidence does not propagate to other steps of the algorithm.

6.3 The mesh intersection algorithm

The computation is performed using only local information stored in the individual triangles. The algorithm has 3 basic steps and a uniform grid is employed to accelerate the computation:

1. First, the 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. Then, a new 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. Finally, a classification step 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 two original ones.

6.3.1 Uniform grid

Similarly to PinMesh, a two-level 3D uniform (as described in Section 4) was employed in 3D-EPUG-Overlay. The grid is created at the beginning of 3D-EPUG-Overlay and employed to accelerate two important steps of the algorithm: the detection of intersections between pairs of triangles from the input meshes and the point location algorithm employed in the triangle classification step.

The uniform grid operations are the only ones not implemented using orientation predicates. However, this will not make the implementation of the symbolic perturbations harder for two reasons.

First, the uniform grid is used only for speed; the algorithm could be implemented without indexing the data, but would be much slower.

Second, the only geometric operation related to the uniform grid is the operation that, given a vertex, returns the coordinates (indexes) of the grid cell containing that vertex. If we assume the lower extreme of the bounding-box of the data is at coordinate $(0, 0, 0)$ (otherwise, translate the data), the indexes of the grid cell containing a vertex v are obtained by dividing the corresponding coordinates of v by the size of the cells and rounding the result down.

Since the indexes are always rounded down, non-negative infinitesimals added to the coordinates of the cells will not change the result of these computations and, thus, no special treatment need to be performed to locate a perturbed vertex.

6.3.2 Intersecting triangles

For speed, the uniform grid is employed to cull the number of pairs of triangles that need to be tested for intersection. For each uniform grid cell, the intersections between pairs of triangles from the two triangulations are computed. Triangles that do not occur in the same cell are not tested. Therefore, although there is a quadratic number of triangle pairs, only a much smaller number, often linear, is tested for intersection.

More specifically, a two-level 3D uniform grid is employed to accelerate the computation. That is, the grid will be created by inserting in its cells triangles from both meshes $M_{0\epsilon}$ and $M_{1\epsilon}$. 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 intersection tests performed will probably be linear in the size of the input maps. The exception would be if there were a superlinear number of triangle pairs very close to each other. This does not occur in real datasets, which satisfy a form of Lipschitz condition bounding the maximum density of elements. Also, as mentioned before, experiments with uniform grids in various applications show excellent performance on real data.

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.

6.3.2.1 Implementation with orientation predicates

Let t_0 and t_1 be two triangles from, respectively, meshes M_0 and M_1. Assume $t_{0\epsilon}$ and $t_{1\epsilon}$ intersect and, w.l.o.g, let e_ϵ be an edge of one of the triangles $t_{i\epsilon}$ that intersects $t_{(1-i)\epsilon}$. Since, because of Lemma 6.1.1, no vertex of e_ϵ can be on the plane of $t_{(1-i)\epsilon}$, it is clear that the intersection will necessarily happen in the interior of e_ϵ. Furthermore, e_ϵ will intersect the interior of $t_{(1-i)\epsilon}$ (Lemma 6.1.5)

Since $t_{0\epsilon}$ and $t_{1\epsilon}$ cannot be co-planar (Lemma 6.1.7), input triangles are non-degenerate and edges intersect only the interior of triangles, the intersection of $t_{0\epsilon}$ and $t_{1\epsilon}$ will always be an edge $u_\epsilon v_\epsilon$ (with $u_\epsilon \neq v_\epsilon$), where u_ϵ and v_ϵ are vertices generated from the intersection of the interior of one of the triangles and the interior of one edge of the other triangle.

Vertices u_ϵ and v_ϵ can be computed by testing the intersection of the six combinations of edges from one triangle against the other triangle. Since intersections happen only between the interior of an edge and the interior of a triangle, the number of intersections detected will be either zero (when the triangles do not intersect) or two (when they do intersect).

As mentioned by [85] and illustrated in Figure 6.1, the intersection between an edge e_ϵ and a triangle can be detected by computing five 3D orientations. For example, if both vertices of e_ϵ are on the same side of the triangle, then they do not intersect. Therefore, the intersection computation can be implemented employing only 3D orientation predicates.

Since intersections are computed only between input triangles, in this step the only geometric predicate that is necessary is the 3D orientation of input vertices. The vertices from intersection are created in this step of the algorithm, and since their coordinates are stored implicitly as the pair (edge, triangle) that generated them, no further computation is necessary.

Furthermore, since there will be no coincidence between edges of one perturbed mesh and triangles of the other perturbed mesh, the 3D orientation predicates will never return 0.

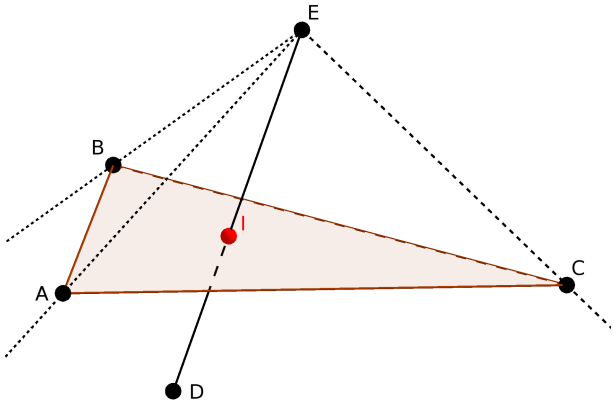


Figure 6.1: Detecting the intersection between an edge and a triangle using 5 orientation predicates - The interior of edge ED will intersect the interior of triangle ABC iff E and D are on opposing sides of the plane of ABC and the sideness of D w.r.t. the triangles EBC, ABE and AEC are the same.

6.3.3 Retesselating the triangles

After computing the intersections between each pair of triangles, the next step is to split the triangles where they intersect, so that after this process 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.

Figure 6.2 presents an example of intersection computation. In Figure 6.2(a), we have two meshes representing two tetrahedra with one region in each one: 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 triangles 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 6.2(b) and (c), respectively). The only triangle from mesh M_0 that intersects mesh M_1 is the triangle 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.

Before retesselating each triangle t_ϵ, the original edges of t_ϵ that intersect other triangles are split at the intersection points. This process is performed by sorting

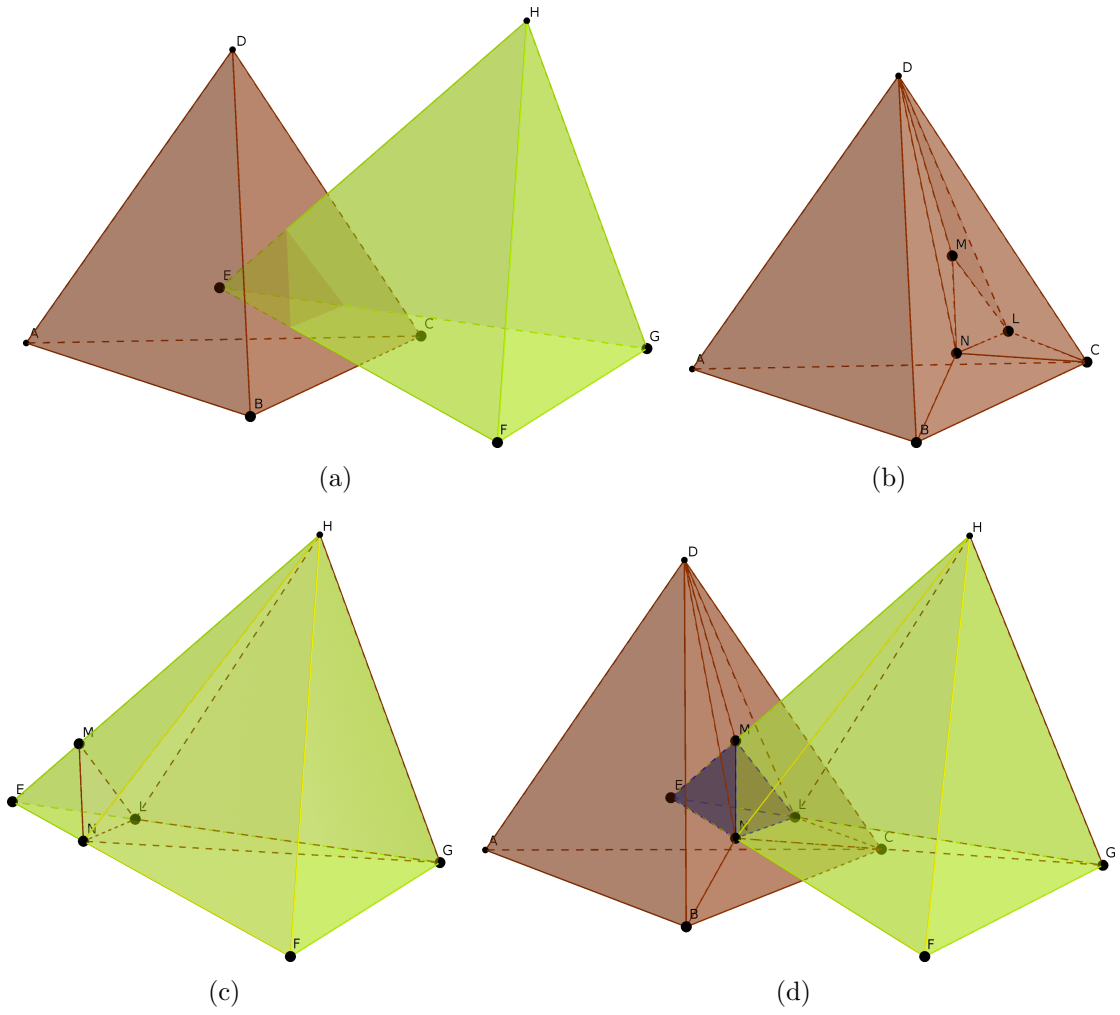


Figure 6.2: Computing the intersection of two tetrahedra - (a): input meshes, (b) and (c): retesselated meshes, (d): classifying the triangles to generate the output.

the intersection points along the edge based on their distance from one of the end vertices of the edge. Then, a planar graph G is created to represent the original non-intersecting edges of t_ϵ, the intersecting edges of t_ϵ split at the intersection points, and the edges generated by intersecting t_ϵ with the other mesh. Since this process is performed on the plane, t_ϵ is first projected onto a plane ($x = 0$, $y = 0$ or $z = 0$) that it is not nearly perpendicular to.

Figure 6.3 (a) illustrates this process: triangle abc (that represents a projection of one of the input triangles onto a plane) intersects three other triangles (generating the edges from intersection xy, yz and zw). To split the edge ca (creating edges cw,

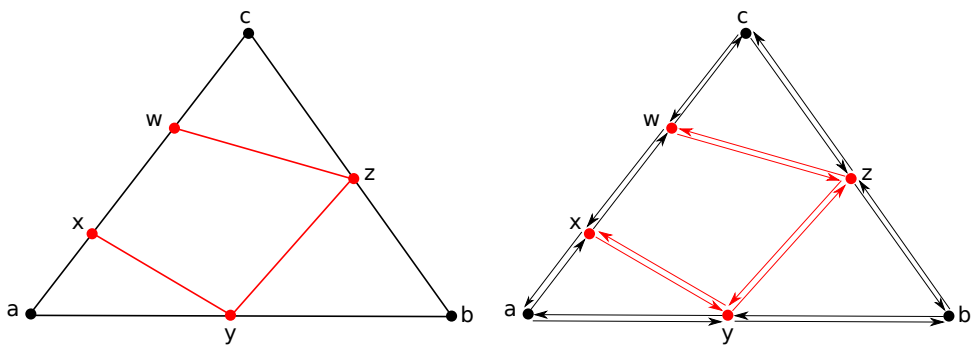


Figure 6.3: Retesselating a triangle that intersects other triangles - (a): triangle abc intersects three other triangles (the edges from the intersection are in red), (b) the edges are split at the intersection points, duplicated and a search procedure is employed to extract the faces generated from the retesselation.

wx and xa) the vertices w and x are sorted along ca based on their distance to c.

The retesselation of t_ϵ is performed using two strategies. First, if the graph G contains only one connected component, then the algorithm presented in [86] is employed to extract the faces of G. This algorithm is based on the idea of replacing each edge with two directed edges (with opposing directions), sorting the edges around each vertex by their polar angles and, then extracting the wedges around each vertex. After that, a search procedure is employed to connect the wedges and generate the list of the faces in G.

Figure 6.3 (b) illustrates this process: after all edges are duplicated, for each vertex v in the graph, the edges starting at v are sorted based on their polar angles. For example, the sorted list of directed edges starting at y is: (yb, yz, yx, ya). Then, the wedges around each vertex are extracted by traversing consecutive items from the previously generated lists: the wedges around y are: (byz, zyx, xya, ayb). Finally, all the wedges from the graph are sorted and a search procedure creates the faces. For example, wedges zyx, yxw, xwz, wzy generates the face $zyxwz$.

The time complexity of this polygon extraction is $\theta(m \log m)$, where m is the number of edges in G. In the set of faces from G, each face is triangulated using the ear-clipping algorithm[87], which has a time complexity quadratic in the number of vertices in the face (the only face in the triangle in Figure 6.3 that needs to be triangulated is the face $zyxwz$, which will be split into two triangles). That time

is acceptable because the expected size of a face is small and constant. If it were a problem, more efficient polygon triangulation algorithms exist, using as little as linear time in the face size, at the cost of considerable complexity.

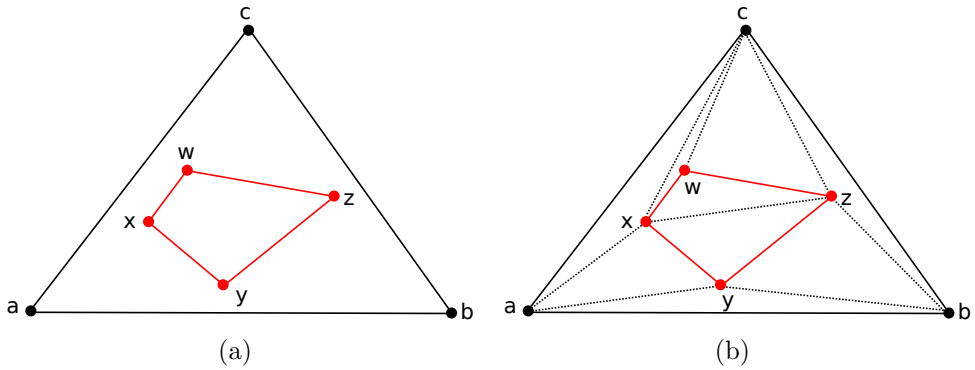


Figure 6.4: Retesselating a triangle where the corresponding graph is disconnected - (a): the original edges from *abc* and the edges generated by the intersection of *abc* with other triangles are disconnected, (b) after greedily trying to insert all the edges (generated by each pair of vertices) that do not intersect the interior of a previously inserted edges, *abc* is completely retriangulated.

Second, if G contains multiple connected components, a trivial algorithm is employed to triangulate G: the resulting set of edges is initialized with the edges of G and for each pair of vertices (u, v) from G, the edge uv is inserted into the solution iff it does not intersect previously inserted edges (except at the endpoints). This process is illustrated in Figure 6.4. A trivial implementation has $\theta(n^4)$ complexity, where n is the number of vertices in G. The reasons why we decided to employ this simpler algorithm are two: the graphs should be relatively small in practice and, according to preliminary experiments, disconnected graphs happen rarely if compared to connected ones. Again, this decision was taken for simplicity and if necessary this algorithm can be replaced with a faster one for constrained triangulation.

6.3.3.1 Implementation with orientation predicates

Assume t_ϵ is a triangle from mesh $M_{i\epsilon}$ and T is the set of triangles from mesh $M_{(1-i)\epsilon}$ intersecting t_ϵ. We will show how to retesselate t_ϵ employing only orientation predicates.

As mentioned before, during the retesselation the vertices are projected onto a plane ($x = 0$, $y = 0$ or $z = 0$) with which t_ϵ is non-perpendicular. For simplicity,

unless otherwise noted all orientation operations will be performed using the projected vertices. For example, if the 2D orientation is applied to three 3D vertices, this operation will assume the three vertices are projected onto one of the planes $x = 0$, $y = 0$ or $z = 0$.

Splitting edges at intersection points Let $e_\epsilon = a_\epsilon b_\epsilon$ be an edge of t_ϵ. In this step, the vertices generated from the intersection of e_ϵ with triangles from $M_{(1-i)\epsilon}$ are sorted based on their distance to a_ϵ. The required geometric predicate to perform this operation is a comparison predicate that verifies if a vertex is closer to a_ϵ than another vertex is.

Let $v_{1\epsilon}$ and $v_{2\epsilon}$ be two vertices generated from the intersection of e_ϵ, with respectively, $t'_{1\epsilon}$ and $t'_{2\epsilon}$ of mesh $M_{(1-i)\epsilon}$. Since both $v_{1\epsilon}$ and $v_{2\epsilon}$ are on e_ϵ, there would be a coincidence on the distance of these points to a_ϵ iff $v_{1\epsilon}$ and $v_{2\epsilon}$ coincided. But, they cannot coincide because otherwise the interior of $t'_{1\epsilon}$ and $t'_{2\epsilon}$ would intersect (which cannot happen since $t'_{1\epsilon}$ and $t'_{2\epsilon}$ are from the same mesh and self-intersecting meshes are considered to be invalid input).

Figure 6.5 illustrates this. Since both a_ϵ and $v_{1\epsilon}$ are on the positive side of $t'_{2\epsilon} = d_\epsilon e_\epsilon f_\epsilon$, then $v_{1\epsilon}$ is closer to a_ϵ than $v_{2\epsilon}$ is.

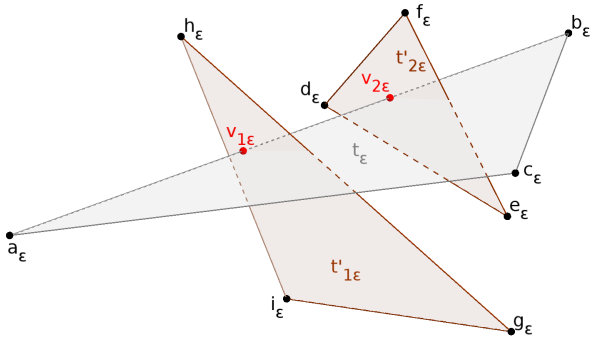


Figure 6.5: Sorting the vertices along an edge - A 3D orientation is employed to determine which vertex generated by an intersection ($v_{1\epsilon}$ or $v_{2\epsilon}$) is closer to an input vertex (a_ϵ).

The predicate to decide which vertex is closer to a_ϵ can be easily implemented by applying a 3D orientation to the non-projected (3D) vertices: $v_{1\epsilon}$ is closer to a_ϵ

than $v_{2\epsilon}$ is iff $v_{1\epsilon}$ and a_ϵ are on the same side of $t'_{2\epsilon}$. Since, as shown above, no coincidence can happen the orientation predicate will never return 0.

Face extraction The only geometric predicate required by the polygon extraction algorithm is the one that sorts pairs of edges by their polar angle around a shared vertex. Given two edges $e_{1\epsilon} = u_\epsilon v_\epsilon$ and $e_{2\epsilon} = u_\epsilon w_\epsilon$, if v_ϵ and w_ϵ are in the same quadrant (assuming that u_ϵ is the origin) then the polar angle of $e_{1\epsilon}$ is smaller than the polar angle of $e_{2\epsilon}$ iff the 2D orientation of u_ϵ, v_ϵ and w_ϵ is positive. If they are in different quadrants, then comparing their polar angle is trivial.

Since vertices on the planar graph may be either input vertices or vertices from the intersection (the planar graph contains the 3 input vertices of the triangle t_ϵ being retesselated and the vertices generated by the intersection of t_ϵ with other triangles), the 2D orientation predicate has to handle all eight combinations of these two types of vertices.

To determine in which quadrant a vertex v_ϵ is considering w_ϵ is the origin, it is necessary to evaluate the sign of each coordinate of the vector $w_\epsilon v_\epsilon$, which is equivalent to computing the 1D orientation of w_ϵ and v_ϵ for the corresponding coordinate.

During the retesselation of $t_\epsilon \in M_{i\epsilon}$ the edges $e_{1\epsilon} = u_\epsilon v_\epsilon$ or $e_{2\epsilon} = u_\epsilon w_\epsilon$ can be either one of the original edges of t_ϵ (possibly split) or one edge generated by the intersection of t_ϵ with a triangle t'_ϵ from $M_{(1-i)\epsilon}$.

If, say, $e_{1\epsilon}$ is one of the original edges of t_ϵ, then its polar angle cannot be equal to $e_{2\epsilon}$'s polar angle otherwise t_ϵ would be degenerate (input meshes with degenerate triangles are invalid), or $e_{1\epsilon}$ and t'_ϵ would be co-planar (this contradicts Lemma 6.1.6). The first and second situations would happen if, respectively, $e_{2\epsilon}$ was an edge of t_ϵ or an edge generated from an intersection.

Now, suppose $e_{1\epsilon}$ and $e_{2\epsilon}$ are edges generated from intersections. Since both edges are on t_ϵ, they are generated from the intersection of t_ϵ with other triangles $t'_{1\epsilon}$ and $t'_{2\epsilon}$ of mesh $M_{(1-i)\epsilon}$. Since the interior of edges generated from intersections is always in the interior of the triangles that generated them, $e_{1\epsilon}$ and $e_{2\epsilon}$ cannot have the same polar angle otherwise the intersection of these two edges would have a

common interior point and, thus, $t'_{1\epsilon}$ and $t'_{2\epsilon}$ would intersect in their interior (which cannot happen since both triangles are in the same mesh). Therefore, there will be no coincidence in the predicate to compare pairs of edges by their polar angle when the perturbed input is processed.

An attentive reader may notice that, even though there will be no coincidence in the sorting, the symbolic perturbation may modify the result of the comparison predicate for a non-degenerate input (which violates one of the three requirements of a SoS perturbation). If before the perturbation an edge has polar angle exactly 0, then after the perturbation this angle can continue to be 0, be greater than 0 or slightly smaller than 2π, which may modify the order of the edges. However, this modification does not affect the face extraction algorithm since the objective of sorting the edges is to extract the wedges and the algorithm works properly as long as the list is sorted in a cyclic order.

Ear-clipping The ear-clipping algorithm employs only two geometric predicates [87]. The first one verifies if a vertex is an ear (convex) while the second one verifies if a vertex is inside (or on the boundary) of a triangle.

Again, these two operations can be trivially performed using 2D orientations. Given two oriented edges $u_\epsilon v_\epsilon$ and $v_\epsilon w_\epsilon$ of a face, v_ϵ is convex iff $orientation(u_\epsilon, v_\epsilon, w_\epsilon)$ is positive.

Also, given a triangle $t_\epsilon = a_\epsilon b_\epsilon c_\epsilon$, we can determine if v_ϵ is inside t_ϵ by evaluating the orientation of v_ϵ w.r.t. the edges $a_\epsilon b_\epsilon$, $b_\epsilon c_\epsilon$ and $c_\epsilon a_\epsilon$ (v_ϵ is inside t_ϵ iff all three orientations are equal).

During the ear-clipping some coincidences may happen even in the perturbed input. Since all vertices of the same mesh are translated using the same perturbation, three vertices from the intersection u_ϵ, v_ϵ and w_ϵ may be collinear (they can be generated, for example, from the intersection of two coplanar triangles from one mesh with a triangle from the other mesh). As a consequence, the 2D orientations employed to verify if a vertex is convex or if a vertex is in a triangle may return 0.

Even though these coincidences may happen, they do not affect the ear-clipping algorithm since it does not need to perform any kind of special treatment

when they happen: if three consecutive vertices u_ϵ, v_ϵ and w_ϵ are collinear then v_ϵ is not considered to be convex (ear). Also, the algorithm assumes the point in triangle algorithm always returns true if a vertex is exactly on the boundary of the triangle.

One could argue that even these simple coincidences should be completely eliminated. However, we believe they do not negatively affect the algorithm since they happen only in a lower level predicate, and do not propagate to higher level functions (even though the orientation of three vertices may be 0, the predicate that detects if a vertex is convex will never return 0). Furthermore, they cannot be completely removed without violating some mathematical properties: for example, the point of intersection of an edge with a triangle will always be collinear with the endpoints of the edge.

Triangulating disconnected subdivisions Disconnected subdivisions are triangulated using only one kind of geometric predicate, which verifies if any two edges $e_{1\epsilon} = u_\epsilon v_\epsilon$ and $e_{2\epsilon} = k_\epsilon w_\epsilon$ intersect other than at their endpoints.

Since all the vertices have unique coordinates, two edges will intersect at their endpoints iff they share a vertex. 2D orientation predicates can be employed to detect if $e_{1\epsilon}$ and $e_{2\epsilon}$ intersect at their interior. The only coincidence that cannot be straightforwardly detected using the signs of the orientations happens when $e_{1\epsilon}$ and $e_{2\epsilon}$ are collinear (which, as mentioned above in the Section on ear-clipping, can occur even in the perturbed dataset).

If $e_{1\epsilon}$ and $e_{2\epsilon}$ are collinear, an intersection in the interior of these edges can be detected by projecting them onto one of the Cartesian axes and verifying if the intervals defined by this projection intersect at their interior. A 1D orientation predicate can be employed to perform this verification.

6.3.4 Classifying triangles

As illustrated in Figure 6.2, after the intersections are detected and all the triangles that intersect other triangles are split at the intersection points, two new meshes $M'_{0\epsilon}$ and $M'_{1\epsilon}$ are created such that each new mesh $M'_{i\epsilon}$ will have the following two kinds of triangles:

- Triangles from the original mesh: if triangle t_ϵ from $M_{i\epsilon}$ did not intersect any triangle from the other mesh (or if this intersection was located on a vertex or edge), then t_ϵ will be in $M'_{i\epsilon}$.

- New triangles: if triangle t_ϵ from $M_{i\epsilon}$ intersects one or more triangles from the other mesh (and this intersection is not located on a common vertex or edge), then t_ϵ will be partitioned into smaller triangles that will be inserted into $M'_{i\epsilon}$.

It is clear that each mesh $M'_{i\epsilon}$ will exactly represent the same regions that $M_{i\epsilon}$ represents. In fact, if no triangle from $M_{i\epsilon}$ intersects the mesh $M_{(1-i)\epsilon}$, then $M'_{i\epsilon}$ will be equal to $M_{i\epsilon}$. Otherwise, each triangle t_ϵ from $M_{i\epsilon}$ that intersects $M_{(1-i)\epsilon}$ will be split in n triangles $t_{0\epsilon}, t_{1\epsilon}, ..., t_{(n-1)\epsilon}$ and these new triangles will be inserted into $M'_{i\epsilon}$ instead of t_ϵ. Since the union of the triangles $t_{0\epsilon}, t_{1\epsilon}, ..., t_{(n-1)\epsilon}$ is t_ϵ and these split triangles contain the same attributes as t_ϵ, then $M'_{i\epsilon}$ represents the same regions $M_{i\epsilon}$ represents.

Thus, computing the intersection between $M'_{i\epsilon}$ and $M'_{(1-i)\epsilon}$ is equivalent to computing the intersection of $M_{i\epsilon}$ with $M_{(1-i)\epsilon}$. However, $M'_{i\epsilon}$ and $M'_{(1-i)\epsilon}$ 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_ϵ from $M'_{i\epsilon}$ will be completely inside a region from $M'_{(1-i)\epsilon}$. Suppose a triangle t_ϵ from $M'_{i\epsilon}$ bounds regions R_a and R_b and is completely inside region R_c from mesh $M'_{(1-i)\epsilon}$. When $M'_{i\epsilon}$ is intersected with $M'_{(1-i)\epsilon}$, t_ϵ will be in the resulting mesh and it will bound regions $R_a \cap R_c$ and $R_b \cap R_c$.

Therefore, the process of classifying the triangles to create the output mesh consists in processing each triangle t_ϵ from the meshes $M'_{i\epsilon}$ $(i = 0, 1)$, determining in what region of $M'_{(1-i)\epsilon}$ t_ϵ is and, then, updating the information about the regions t_ϵ bounds such that we will have a consistent mesh.

If a triangle t_ϵ is in the exterior of the other mesh, in the resulting mesh the two regions t_ϵ 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 6.2(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 $M'_{i\epsilon}$ are being located. If two adjacent triangles t_ϵ and t'_ϵ share an edge that was not generated by an intersection with $M'_{(1-i)\epsilon}$, then these triangles are in the same region of $M'_{(1-i)\epsilon}$. If t_ϵ and t'_ϵ share an edge that was generated by an intersection with triangle t''_ϵ of mesh $M'_{(1-i)\epsilon}$, then t_ϵ and t'_ϵ are in different regions of the other mesh. Since the regions t''_ϵ bound are known, it is possible to determine the location of t_ϵ and t'_ϵ once the location of at least one of these two triangles is known.

Figure 6.6 (a) illustrates this process. Suppose the triangle $t = abc$ (that was split into triangles t_i ($i = 1...7$) during the retesselation) intersects four other triangles from the other mesh and that these four triangles bound regions 1 and 2 (of the other mesh). Since t_1 and t_5 share an edge that was generated by an intersection, then they are in different regions of the other mesh (thus, either t_1 is in region 1 and t_5 is in region 2 or t_1 is in region 2 and t_5 is in region 1). The triangles in each set $\{t_1, t_2\}$, $\{t_3, t_4, t_5\}$, $\{t_6\}$,$\{t_7\}$, on the other hand, share edges that were not generated by an intersection an, therefore, the components of each set belong to the same region of the other mesh. If it is known that t_1 is in region 2 of the other mesh (Figure 6.6 (b)), this implies that t_2 is also in region 2, t_3, t_4 and t_5 are in region 1 and t_6 and t_7 are in region 2.

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, then, 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. This process is performed

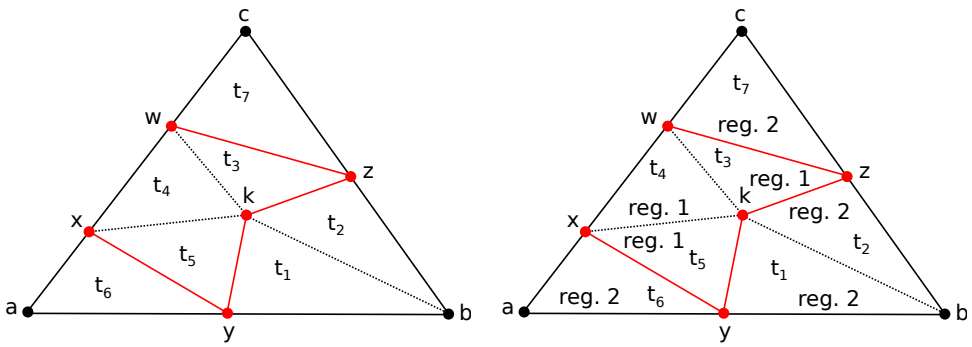


Figure 6.6: Labeling the triangles once the location of at least one triangle is known: (a) before the labeling process, (b) after the regions (abbreviated as reg.) are labeled.

using the PinMesh [88] point location algorithm that, besides being able to perform queries in expected constant time, uses the same index we employ for indexing the triangles.

6.3.4.1 Implementation with orientation predicates

The triangles from one mesh are located in the other one by employing PinMesh to locate triangles with input vertices and, then, using a flood-fill algorithm to assign the location of the other triangles. Thus, we only have to show that PinMesh can be implemented using only orientation predicates.

As mentioned in [88], PinMesh performs only 3 geometric operations:

- $isOnProj(t_\epsilon, q_\epsilon)$: given a triangle t_ϵ and a query point q_ϵ, decide if the projection of q_ϵ onto the plane passing through t_ϵ is on the interior of t_ϵ.

- $isAbove(t_\epsilon, q_\epsilon)$: given a query point q_ϵ and a triangle t_ϵ such that $isOnProj(t_\epsilon, q_\epsilon)$ is true, decide if the projection of q_ϵ onto t_ϵ is above q_ϵ, i.e., the triangle is above the point.

- $isBelow(t_\epsilon, t'_\epsilon, q_\epsilon)$: given two triangles t_ϵ and t'_ϵ directly above a query point q_ϵ, decide if the z component of the projection of q_ϵ onto t_ϵ is smaller than the z component of the projection of q_ϵ onto t'_ϵ.

The first operation is exactly the point in triangle test and, as mentioned earlier

in the Section on ear-clipping, it can be implemented using three 2D orientation predicates. The operation $isAbove(t_\epsilon, q_\epsilon)$ can be implemented by deciding on which side of t_ϵ q_ϵ lies and, then, verifying if t_ϵ's normal has a positive or negative z-component (which can be computed by verifying if the 2D orientation of the 3 vertices of t_ϵ is positive considering t_ϵ is projected onto $z = 0$).

Finally, $isBelow(t_\epsilon, t'_\epsilon, q_\epsilon)$ can also be implemented using orientation: let v_ϵ be the vertex generated from the intersection of t_ϵ with a vertical edge passing through q_ϵ. $isBelow(t_\epsilon, t'_\epsilon, q_\epsilon)$ will be true iff the (3D) orientation of q_ϵ with respect to t'_ϵ is equal to the orientation of v_ϵ with respect to t'_ϵ (i.e., if both q_ϵ and v_ϵ are on the same side of t'_ϵ). This predicate can be implemented by creating the vertex v_ϵ as a dummy vertex from the intersection and applying the 3D orientation predicate.

Figure 6.7 illustrates an analogous process in 2D: to determine if the segment $a_\epsilon b_\epsilon$ intersects a vertical line passing through q_ϵ at a lower point than $a_\epsilon c_\epsilon$ intersects, a dummy vertex $v_{1\epsilon}$ is created as the intersection of $a_\epsilon b_\epsilon$ with $q_\epsilon u_\epsilon$, where u_ϵ is an arbitrary point above $q_\epsilon u_\epsilon$. Since $v_{1\epsilon}$ and q_ϵ are on the negative side of $a_\epsilon c_\epsilon$, then $v_{1\epsilon}$ is lower than $v_{2\epsilon}$.

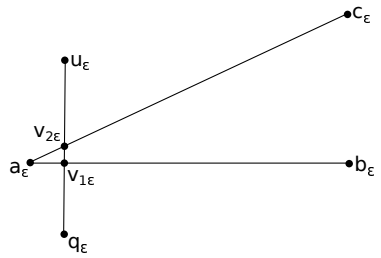


Figure 6.7: Sorting the vertices along an edge - Since $v_{1\epsilon}$ and q_ϵ are on the same side of $a_\epsilon c_\epsilon$, then $v_{1\epsilon}$ is lower than $v_{2\epsilon}$.

PinMesh is employed to query an input vertex of one mesh against the other original mesh and, thus, only input vertices are processed. Also, PinMesh employs the same perturbation scheme employed by 3D-EPUG-Overlay and, thus, there will be no coincidence during the point location queries.

6.4 Implementing the symbolic perturbation

Since all geometric operations can be implemented using only orientation predicates, the symbolic perturbation needs to be implemented only for these predicates.

Because of the determinants' regularity, orientation predicates can be easily adapted to process perturbed points [4].

However, there is a challenge in the mesh intersection problem: the predicates will have not only to handle input vertices (with rational coordinates), but also vertices generated from intersections. Since the coordinates of a vertex generated from an intersection is a function of five input points (two points defining an edge of one mesh and three points defining a triangle of the other mesh) and these points are perturbed, then the orientation has to be modified to handle these points.

As shown above in the Section describing the mesh intersection algorithm, the 3D orientation will only be computed using, as arguments, the three vertices of an input triangle and another vertex that may be either an input vertex or a vertex from the intersection. Thus, at least two versions of the 3D orientation predicate will have to be implemented.

For the 2D orientation, on the other hand, any of the three parameters may be either an input vertex or a vertex generated by an intersection. Thus, eight versions of the orientation predicate will be required. However, since the orientation is computed using a determinant, the order of the parameters may be modified as long as the sign of the result is negated for each parameter swap. For example, $orientation(p_0, p_1, p_2) = -orientation(p_0, p_2, p_1)$. Then the number of functions actually written can be reduced by sorting the parameters by their type (input vertex or vertex from intersection).

During the evaluation of a predicate where at least one of the parameters p_ϵ is a vertex from the intersection, the coordinates of this vertex need to be computed. By definition, p_ϵ is represented by the points $t_{0\epsilon} = t_0 + (i\epsilon, i\epsilon^2, i\epsilon^3)$, $t_{1\epsilon} = t_1 + (i\epsilon, i\epsilon^2, i\epsilon^3)$ and $t_{2\epsilon} = t_2 + (i\epsilon, i\epsilon^2, i\epsilon^3)$ of the triangle t_ϵ and $e_{0\epsilon} = e_0 + ((1-i)\epsilon, (1-i)\epsilon^2, (1-i)\epsilon^3)$ and $e_{1\epsilon} = e_1 + ((1-i)\epsilon, (1-i)\epsilon^2, (1-i)\epsilon^3)$ of the edge e_ϵ, where i is the id (that may be either 0 or 1) of the mesh containing t_ϵ (consequently, $(1-i)$ is the id of the mesh containing e_ϵ) and p_ϵ is the intersection of t_ϵ with e_ϵ.

Each coordinate of p_ϵ will be an ϵ-expression with degree 3, where each coefficient is a function of the coordinates of the input vertices and of id. The coefficient of degree 0 represents the corresponding coordinate of the intersection point if t_ϵ

and e_ϵ were not perturbed.

Once the coordinates are employed in the determinant to compute the sign of the orientation, the resulting ϵ-expression will have maximum degree 6, where the coefficient of degree 0 represents the result that would be obtained in the corresponding predicate if the meshes were not perturbed. Since ϵ is an infinitesimal, the sign of the determinant will be the sign of the non-vanishing coefficient of smaller degree, or 0 if all coefficients vanish.

Given an orientation predicate $orientation(p_{0\epsilon}, p_{1\epsilon}, ..., p_{d\epsilon})$, there are only two possibilities for the perturbation of each vertex $p_{j\epsilon}$. If $p_{j\epsilon}$ is an input vertex, then $p_{j\epsilon}$ will be either translated by $(\epsilon, \epsilon^2, \epsilon^3)$ if it belongs to mesh 1 or it will not be translated if it belongs to mesh 0. If $p_{j\epsilon}$ is a vertex from the intersection, then either the edge that generated $p_{j\epsilon}$ or the triangle that generated $p_{j\epsilon}$ will be translated by $(\epsilon, \epsilon^2, \epsilon^3)$. Since the number of combinations is relatively small, we decided to generate a different predicate to handle each combination instead of evaluating the perturbations at runtime.

For the 2D orientation, for example, there will be eight possible combinations of perturbation schemes used in the three parameters. Furthermore, each of the three parameters may be either input vertices or vertices from the intersection, which results in eight combinations for the parameter types (since the parameters may be sorted, the actual number of combinations that have to be implemented is four). Finally, since the orientation 2D deals with 3D points projected onto one of the planes $x = 0$, $y = 0$ or $z = 0$, one version of the predicate will have to be implemented for each plane. Thus the total number of functions will be 96. For the 3D orientation, at least three of the parameters will be input vertices (because the 3D orientations employed by 3D-EPUG-Overlay always consider the orientation of one vertex with respect to a triangle formed by three *input* vertices) and, thus, the number of predicates that will be implemented is 32.

Similarly, in the 1D orientation the number of predicates will be 36: the predicate has 2 parameters, each one may be either an input or a vertex from the intersection (thus, there are 4 combinations of parameters – however, since they may be sorted only 3 combinations actually have to be implemented). Furthermore,

each parameter may be from either mesh 0 or 1 and, also, the predicate is evaluated considering the points are projected onto the x, y or z axis. Therefore, the total number of combinations that have to be implemented is $3x4x3 = 36$.

The advantage of having different versions of the predicates for each combination of the parameters is that the degrees of the ϵ coefficients are known at compile time, and the implementation may compute the coefficients by processing first the ones with smaller degrees, and returning as soon as the first coefficient processed is non-zero (as mentioned before, the sign of the epsilon-expressions is the signum of the lowest degree non-vanishing coefficient). Also, since the expressions defining the coefficients are known at compile time, they may be easily simplified.

Let us present an example of the implementation of a 2D orientation predicate: consider the function *orientation_z*0_001, the predicate to evaluate the orientation of three vertices $p_{0\epsilon}$, $p_{1\epsilon}$ and $p_{2\epsilon}$ when they are projected onto $z = 0$ and vertices $p_{0\epsilon}$ and $p_{1\epsilon}$ belong to mesh 0 while vertex $p_{2\epsilon}$ belongs to mesh 1 (the suffix 001 in the name of the predicate represents the id of the mesh of each parameter).

Since $p_{0\epsilon}$ and $p_{1\epsilon}$ are in mesh 0, then $p_{0\epsilon} = p_0$ and $p_{1\epsilon} = p_1$. Because $p_{2\epsilon}$ is in mesh 1, $p_{2\epsilon} = p_2 + (\epsilon, \epsilon^2, \epsilon^3)$. As mentioned in Section 6.2.1, the 2D predicate to evaluate the orientation of these three points when they are projected onto $z = 0$ will be:

$$orientation\_z0\_001(p_0, p_1, p_2) = sgn\left(\begin{vmatrix} p_{0x} & p_{0y} & 1 \\ p_{1x} & p_{1y} & 1 \\ p_{2x}+\epsilon & p_{2y}+\epsilon^2 & 1 \end{vmatrix}\right)$$

$$= sgn(p_{0x}(p_{1y}-p_{2y}) + p_{1x}(p_{2y}-p_{0y}) + p_{2x}(p_{0y}-p_{1y}) + \epsilon(p_{0y}-p_{1y}) + \epsilon^2(p_{1x}-p_{0x})) \tag{6.18}$$

Observe that the epsilon monomials of degree 0 in Equation 6.18 represent the determinant of the unperturbed points. Because of the magnitude of the infinitesimals, the return value of the *orientation_z*0_001 predicate will be:

- $sgn(p_{0x}(p_{1y}-p_{2y}) + p_{1x}(p_{2y}-p_{0y}) + p_{2x}(p_{0y}-p_{1y}))$ (the sign of the determinant of the unperturbed points), if the value of this expression is not zero.

- $sgn(p_{0y} - p_{1y})$, if the value of the previous expression is zero and if this one is not.
- $sgn(p_{1x} - p_{0x})$, otherwise.

One may argue that implementing 164 functions is a hard (and error-prone) task. However, all the functions are very regular and a program to generate them automatically can be implemented. Indeed, during the implementation of this mesh intersection algorithm a Wolfram Mathematica script was developed and the code for all the predicates was created automatically by the script.

6.5 Experiments

3D-EPUG-Overlay was implemented in C++ and compiled using g++ 5.4.1. For a better parallel scalability, the Tcmalloc memory allocator provided by gperftools [89] 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. All the experiments were performed on a workstation with dual Intel Xeon E5-2687 processors, each with 8 physical cores, each core able to run 2 threads using the Intel Hyper-threading technology. The workstation has 128 GiB of RAM and runs Ubuntu Linux 16.04.

We evaluated 3D-EPUG-Overlay, by comparing it against three state of the art algorithms: the exact and parallel algorithm developed by [64] and distributed in the LibiGL library, the exact algorithm for intersecting Nef Polyhedra available in the CGAL library, and the fast and parallel intersection algorithm available in QuickCSG [63]. Even though QuickCSG is not an exact algorithm and does not handle special cases [63], we compared 3D-EPUG-Overlay against it to verify how our exact algorithm compared with a fast approximate algorithm.

6.5.1 Datasets

Experiments were performed with a variety of meshes downloaded from 3 datasets. All these meshes are non self-intersecting and watertight. Table 6.1

presents the names of the meshes, source, creator (only for the meshes downloaded from the AIM@SHAPE-VISIONAIR repository), number of vertices, triangles and polyhedra (for meshes with internal structure).

Meshes whose names are numbers were downloaded from the Thingi10k repository [90] (the number represents the *id* employed by the repository). The ones with the suffix *kf* were obtained from the dataset provided by Barki [91]. All the other meshes were downloaded from the AIM@SHAPE-VISIONAIR Shape Repository [84]. Some models are available in different datasets with different resolutions (e.g., there is a version of *Armadillo* with 331 thousand triangles and another one with 52 thousand).

Also, some of the meshes (the ones with suffix *tetra*) were tetrahedralized (i.e., tetrahedral meshes were generated for the 3D domain defined by the original meshes) using the GMSH tool [92]. For example, *ArmadilloTetra* is the tetrahedralization of the *Armadillo* mesh.

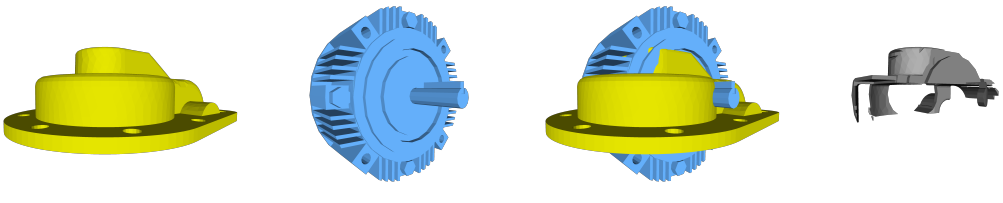
Table 6.2 presents the pairs of meshes used in the intersection experiments, the number of input triangles, the number of triangles in the resulting meshes and the default configuration of the uniform grid employed in the experiments (details about how these configurations were chosen will be presented in Section 6.5.4).

Figure 6.8 illustrates some of these pairs of meshes (models without internal structure) and Figure 6.9 illustrates mesh *914686Tetra*, the smallest tetra mesh used in the experiments.

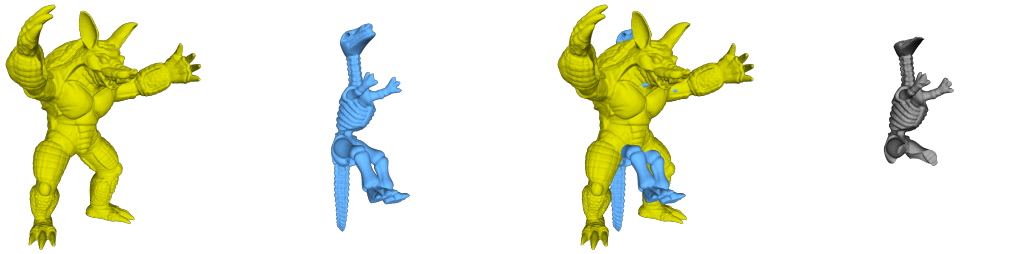
6.5.2 The effect of the use of arithmetic filters and other optimizations

In order to evaluate the effect of different optimization techniques employed in 3D-EPUG-Overlay we profiled two key steps of the algorithm that directly benefit from these optimizations: the creation of the uniform grid and the detection of intersections between pairs of triangles.

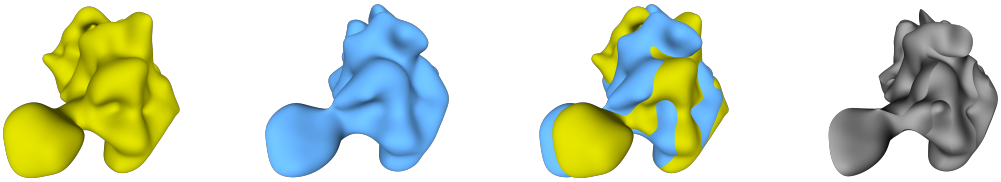
These experiments were performed with the Neptune and Neptune translated meshes using a uniform grid with resolution 64^3 in the first level and 16^3 in the second one.



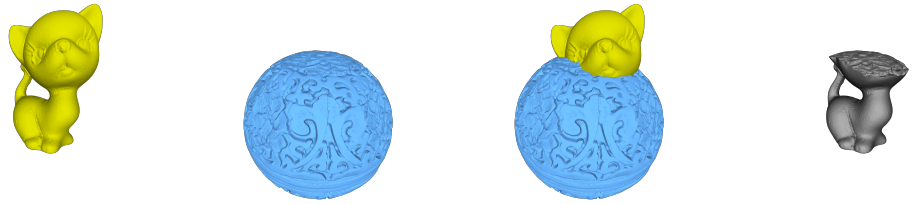
(a) Casting10kf (b) Clutch2kf (c) Same layer (d) Intersection



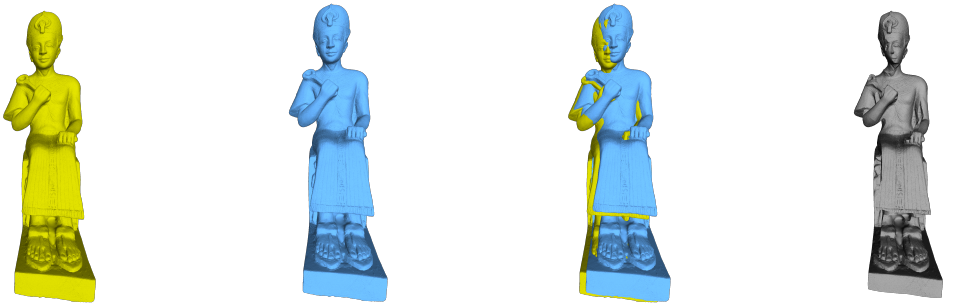
(e) Armadillo52kf (f) Dinausor40kf (g) Same layer (h) Intersection



(i) 461112 (j) 461115 (k) Same layer (l) Intersection



(m) Kitten (n) RedCircBox (o) Same layer (p) Intersection



(q) Ramesses (r) Ram.Transl. (s) Same layer (t) Intersection

Figure 6.8: Some of the pairs of meshes employed in the experiments - Each row presents, respectively, the pair of meshes, the two meshes in the same image layer and the computed intersection.

Table 6.1: Datasets used in the 3D intersection experiments.

Mesh	Source	Creator	Vertices ($\times 10^3$)	Triangles ($\times 10^3$)	Polyh. ($\times 10^3$)
Clutch2kf	Barki	-	1	2	-
Casting10kf	Barki	-	5	10	-
Horse40kf	Barki	-	20	40	-
Dinausor40kf	Barki	-	20	40	-
Armadillo52kf	Barki	-	26	52	-
Camel	AIM@SHAPE	-	35	69	-
Camel69kf	Barki	-	35	69	-
Cow76kf	Barki	-	38	76	-
Bimba	AIM@SHAPE	Marco Attene	75	150	-
Kitten	AIM@SHAPE	Frank_terHaar	137	274	-
Armadillo	Stanford	-	173	346	-
461112	Thingi10k	-	403	805	-
461115	Thingi10k	-	411	822	-
RedCircBox[a]	AIM@SHAPE	Marco Attene	701	1,403	-
Ramesses	AIM@SHAPE	Marco Attene	826	1,653	-
Ramesses Rot.	AIM@SHAPE	Marco Attene	826	1,653	-
Ramesses Transl.	AIM@SHAPE	Marco Attene	826	1,653	-
Vase	AIM@SHAPE	Pierre Alliez	896	1,793	-
226633	Thingi10k	-	1,226	2,452	-
Neptune	AIM@SHAPE	Laurent Sab.	2,004	4,008	-
Neptune Transl.	AIM@SHAPE	Laurent Sab.	2,004	4,008	-
914686Tetra	Thingi10k+GMSH	-	66	605	281
68380Tetra	Thingi10k+GMSH	-	107	1,067	506
Armad.Tetra[b]	Stanford+GMSH	-	340	3,377	1,602
Arm.Tet.[b]Transl.	Stanford+GMSH	-	340	3,377	1,602
518092Tetra	Thingi10k+GMSH	-	603	5,938	2,814
461112Tetra	Thingi10k+GMSH	-	842	8,495	4,046

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

The following versions of the algorithm were evaluated in the experiments:

- Vector: each level of the uniform grid is created by performing a single-pass through the triangles, what requires the use of dynamic STL vectors to push the triangles into the grid cells.

- Ragged: a ragged array is employed to reduce the number of memory allocations. Each level of the uniform grid has to be created by performing two

passes through the data (one to count the amount of triangles to be inserted into each cell and another one to effectively insert the triangles).

- NoAlloc: temporary memory allocation is avoided by reusing temporary rational numbers and by rewriting the arithmetic expressions in order to avoid the creation of temporary rationals (this version includes the optimizations from the *Ragged* version of the algorithm).

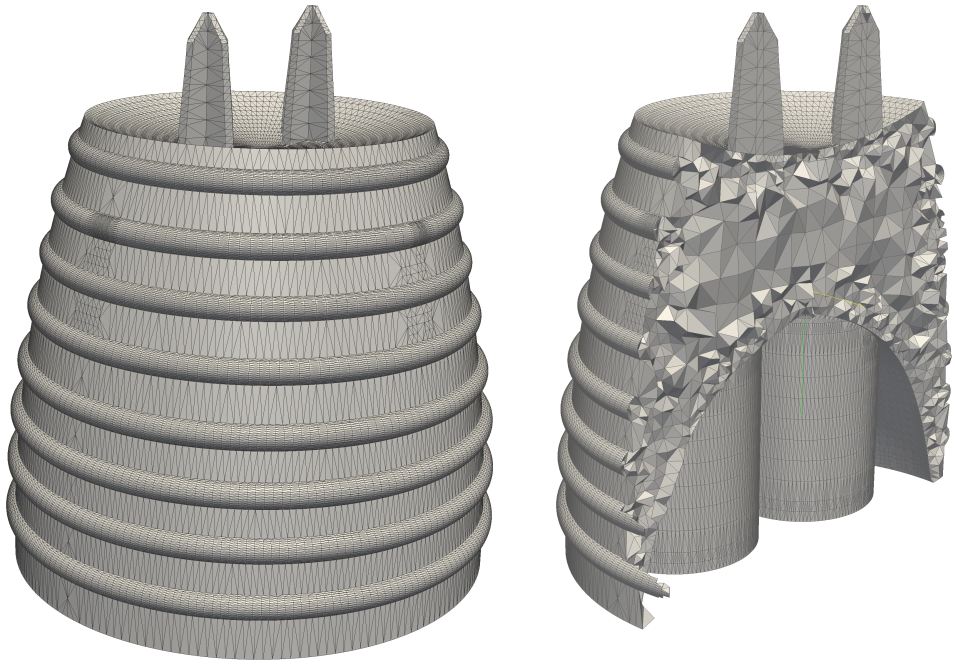
- Interval: interval arithmetic is employed to avoid the use of operations with rationals in the predicates (this version includes the optimizations from the *NoAlloc* version of the algorithm)

Furthermore, some experiments were performed using the default *glibc* memory allocator (using the default g++ compilation flags), while others were performed using the gperftools Tcmalloc allocator (abbreviated to *Tcm.* in the table).

Table 6.2: Pairs of meshes evaluated, number of triangles (in the input and output meshes) and default grid configuration employed in the experiments.

		Number of triangles ($\times 10^3$)			Grid size	
Mesh 0	Mesh 1	Mesh 0	Mesh 1	Output	G_1 [a]	G_2 [b]
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	1,402	246	64	8
Bimba	Vase	150	1,792	724	64	8
226633	461112	2,452	805	1,437	64	8
Ramesses	RamessesTranslated	1,653	1,653	1,571	64	16
Ramesses	RamessesRotated	1,653	1,653	1,691	64	16
Neptune	Ramesses	4,008	1,653	1,112	64	16
Neptune	NeptuneTranslated	4,008	4,008	3,303	64	16
68380Tetra	914686Tetra	1,067	605	9,393	64	2
ArmadilloTetra	ArmadilloTetraTransl.	3,377	3,377	61,325	64	4
518092Tetra	461112Tetra	5,938	8,495	23,181	64	4

[a] resolution of the first level grid; [b] resolution of the second level grid.



(a) (b)

Figure 6.9: (a) Exterior of the *914686Tetra* mesh. (b) clipped version of mesh *914686Tetra*, showing its internal tetrahedra.

Table 6.3 presents the times (in seconds) spent by the different versions of the algorithm both when it was run sequentially and when it was run using 32 threads.

As mentioned in Chapter 4, the creation of the 3D uniform grid has 4 important steps. First, the uniform grid cell where each vertex is located is computed (row *Comp. grid cell*), what is a computationally intensive step mainly when rationals are employed. Then, a first pass (row *First pass*) through the triangles is performed: if the uniform grid stores the triangles in each cell in vectors, these triangles are stored in this step. Otherwise, if a ragged array is employed, the number of triangles in each cell is counted and the triangles are effectively inserted into the cells during a second pass (row *Second pass*). Finally, a second level grid is created (row *Refinement*). These three last steps are more memory intensive.

The intersection detection is performed in two steps: first, the uniform grid is traversed and a list of pairs of triangles to be tested for intersection is created (row *List pairs of tri.*). In the second step, pairs of triangles in the list are tested for intersection (row *Detect inters.*).

Table 6.3 also presents the total times spent creating the grid (row *total time grid*) and detecting intersections (row *total time inter.*).

Table 6.3: Times, in seconds, spent by key steps of 6 versions of 3D-EPUG-Overlay using either 1 or 32 threads.

Version Allocator	Vector Tcm. [a]	Ragged Tcm. [a]	Ragged Glibc	NoAlloc Tcm. [a]	NoAlloc Glibc	Interval Tcm. [a]
Threads			1			
Comp. grid cell [b]	8.29	8.66	8.82	7.00	6.93	0.34
First pass [c]	0.39	0.35	0.31	0.30	0.30	0.31
Second pass [d]		0.41	0.40	0.40	0.39	0.41
Refinement [e]	0.92	0.70	0.65	0.67	0.65	0.69
List pairs of tri. [f]	0.82	0.77	0.78	0.76	0.77	0.76
Detect inters. [g]	1,421.46	1,425.21	1,442.67	1,199.36	1,221.12	2.19
Total time grid [h]	9.60	10.12	10.18	8.37	8.27	1.74
Total time inter. [i]	1,422.28	1,425.97	1,443.45	1,200.12	1,221.89	2.95

Version Allocator	Vector Tcm. [a]	Ragged Tcm. [a]	Ragged Glibc	NoAlloc Tcm. [a]	NoAlloc Glibc	Interval Tcm. [a]
Threads			32			
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] abbreviation of the *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.

Considering the time for the uniform grid creation, while the algorithm using a ragged array is slightly slower than the version using dynamic vectors in single-thread mode, when the algorithm is run in parallel the version with a ragged array is 32% faster than the *Vector* version. This difference happens for three reasons.

First, the version using vectors is less parallelizable: during the first pass the triangles are distributed to the cells and the insertion into the vectors must be done sequentially (since insertion into vectors is not thread-safe). The ragged array, on the other hand, inserts the triangles using two passes that are parallelizable: in the first pass the number of triangles that will be inserted into each cell is counted (for performance, the triangles are processed in parallel and each thread keeps private counters for each cell – after processing all triangles the counters in each thread are added). In the second pass, the ragged array is allocated as one single array and the triangles are effectively inserted into the corresponding positions. The insertion can be performed in parallel because the number of triangles in each cell is already known and, thus, OpenMP atomic capture and increment operations can be employed to perform the insertion safely.

Second, the ragged array has better locality of reference in comparison with a matrix of vectors: while the ragged array is stored as one big sequential memory block, the different vectors associated with each cell may be stored sparsely in the heap.

Third, the use of a ragged array leads to fewer memory allocations on the heap in comparison with dynamic vectors, which parallelizes better. For example, during the refinement of the grid the first level cells are processed independently in parallel. During this processing, a second level grid is created in each refined cell. When the ragged array is employed, each refined first level cell allocates one ragged array. On the other hand, when vectors are employed each refined first level cell allocates G_2^3 vectors (where G_2 is the resolution of the second level grid) and each vector may lead to further memory allocations as they grow when the triangles are inserted. As result, the grid refinement in the experiments with the parallel version of the algorithm took 0.06 seconds when a ragged array was employed and 0.13 seconds when vectors were used.

Observe also that, while there are fewer parallel memory allocations in the grid refinement when the ragged array is employed, this kind of allocation is not completely avoided (since the cells are refined in parallel, and thus the ragged arrays in each cell have to be allocated in parallel). The effect of this behavior can

be observed by comparing the time spent by the *Ragged* version of the algorithm (running in parallel) compiled with and without the *Tcmalloc* allocator: the version using *Tcmalloc* was 6 times faster (0.06 seconds versus 0.36 seconds) than the version using the standard *Glibc* memory allocator. This difference in the performance can be explained because *Tcmalloc* was specifically designed to reduce the overhead of parallel memory allocations [89], which dominates this step of the algorithm.

The use of *Tcmalloc* also improves the performance of computationally intensive steps of the algorithm (mainly when it is run in parallel). Consider the *Ragged* version: the time spent computing the grid cell where each vertex is was 0.58 seconds and 0.77 seconds when, respectively, the *Tcmalloc* and the *Glibc* allocators were employed. A similar behavior can be observed in the time spent testing pairs of triangles for intersection. These improvements happen because these computations lead to parallel memory allocation for creating the temporary rational numbers used in the arithmetic expressions.

Besides employing better memory allocators, avoiding memory allocations (whenever possible) can also improve the performance of the algorithm. For example, in the *NoAlloc* version of the algorithm the arithmetic expressions with rationals were rewritten to avoid the creation of temporary rationals and, as result, the two computationally intensive steps mentioned above performs better in this version than in the *Ragged* version.

Since computing the grid cell where each vertex is 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.

Table 6.4 presents the parallel speedup obtained by the 6 different versions of the algorithm in each step. Observe that the best speedups were obtained when memory allocations were avoided and when the *Tcmalloc* allocator was employed. For a given algorithm, the best speedups were obtained in the most computationally intensive steps.

While the *Interval* version of the algorithm was the fastest one, it presented

the worst total speedup (6.4 times during the creation of the uniform grid and 8.3 times during the intersection detection). This can be explained because in the previous versions of the algorithm detecting intersections and computing the grid cells of each vertex were important bottlenecks and, since these two steps presented the best speedups, the total time of the algorithm also scaled well. In the *Interval* version these two steps still present good speedups, however, since their performance was significantly improved, the speedup is now more dependent on the times spent by less scalable steps of the algorithm.

Notice also that, while improving the times for creating the uniform grid makes little difference in the total running-time of the *Vector* version of the algorithm (since the grid creation accounts for only 1% of the total running time), once interval arithmetic is employed the time spent in the grid creation becomes a very important bottleneck (accounting for 44% of the total running time) and, thus, the improvements in the grid creation obtained thanks to the better memory allocation strategies mentioned above are crucial for the good performance of the *Interval* version.

Table 6.4: Parallel speedup (ratio between the time spent using 1 thread and the time spent using 32 threads) obtained in the key steps of 6 versions of 3D-EPUG-Overlay.

Version Allocator	Vector Tcm. [a]	Ragged Tcm. [a]	Ragged Glibc	NoAlloc Tcm. [a]	NoAlloc Glibc	Interval Tcm. [a]
Comp. grid cell[b]	14.3	14.9	11.4	16.1	16.6	3.4
First pass[c]	1.0	5.2	4.9	4.7	4.6	4.6
Second pass[d]		9.7	9.0	9.2	8.3	9.6
Refinement[e]	6.9	11.4	1.8	11.1	1.8	11.1
List pairs of tri.[f]	4.7	4.7	3.6	4.9	3.6	4.6
Detect inters.[g]	17.4	17.1	11.4	18.0	18.3	11.7
Total time grid[h]	8.7	13.5	8.2	13.9	9.3	6.4
Total time inter.[i]	17.4	17.0	11.4	17.9	18.3	8.3

[a] abbreviation of the *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 spent creating the list of pairs of triangles that may intersect; [g] time spent testing triangles for intersection; [h] total time spent creating the uniform grid;
[i] total time detecting the intersections once the grid has been created.

In the next sections of this thesis we will always perform experiments with the most optimized version of 3D-EPUG-Overlay, i.e., the version that avoids temporary memory allocations, use the Tcmalloc allocator, ragged arrays and arithmetic filtering implemented with interval arithmetic.

6.5.3 The importance of the uniform grid

As mentioned before, in this work the uniform grid is mainly employed to accelerate the detection of pairs of triangles that intersect. To evaluate the efficiency of this step of the algorithm, we compared it against an implementation using the CGAL method for intersecting dD Iso-oriented Boxes. Both evaluated algorithms are exact and employ arithmetic filters with interval arithmetic. Indeed, this CGAL algorithm is employed by LibiGL to accelerate the triangle-triangle intersection detection step of its mesh intersection method.

The CGAL method is sequential and employs a hybrid approach composed of a sweep-line and a streaming algorithm to detect intersection between pairs of Axis Aligned Bounding Boxes. Thus, pairwise intersection 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 is not thread-safe yet, even the triangle-triangle intersection tests were performed sequentially.

Since the uniform grid employed in this work was specifically designed to be parallel, we evaluated it using 32 threads.

Table 6.5 presents the experiments performed in 6 pairs of meshes. Column *# int.* represents the number of intersections detected, column *Int.tests* presents the number of pairwise triangle intersection tests performed by the algorithms, column *Pre.proc.* represents the time spent either creating the uniform grid and extracting pairs of triangles for intersection or using CGAL to extract pairs of intersecting bounding-boxes. Finally, column *Inter.* presents the times spent by the algorithms performing intersection tests.

Observe that the number of intersections detected is not necessarily the same in the two algorithms. This is justified because our algorithm implements Simulation of Simplicity, and thus co-planar triangles, for example, never intersect.

In general, the CGAL algorithm is better at culling pairs of non-intersecting bounding-boxes and, thus, the number of intersection tests performed by it is smaller than the number of tests performed by the algorithm based on the uniform grid. The biggest difference is observed in the last dataset, where CGAL performed 8 times fewer intersection tests than the uniform grid method. However, since the uniform grid is a lightweight structure and since it parallelizes well, its pre-processing time is much smaller (it was up to 134 times faster) than the pre-processing time of CGAL and this difference is never recaptured by CGAL. Indeed, except for the intersection of the Armadillo with itself, even if CGAL took 0 seconds to detect the intersections the total time spent by uniform grid method would still be smaller.

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 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 intersection computation done by the uniform grid method is performed in parallel.

Considering the number of intersection tests performed per second, the worst performance for both methods happened during the intersection of the Armadillo mesh with itself. This can be explained because in this situation there are many coincidences (co-planar triangles being tested for intersection, triangles intersecting other triangles on the edges, etc). These coincidences lead to arithmetic filter failures (because the result of some of the orientation predicates is 0 and, thus, the intervals representing these results are likely to have different signs for their bounds), which lead to exact computations with rationals. Furthermore, coincidences lead to the use of SoS predicates (which have not been optimized yet) in the method based on uniform grid.

6.5.4 The effect of different choices for the grid sizes

To evaluate the impact of different configurations for the uniform grid sizes, we evaluated the running times of 3D-EPUG-Overlay using 6 different meshes and a variety of grid sizes.

Tables 6.6 and 6.7 presents the results of the experiments for 4 meshes without

internal structure while Table 6.8 presents the results for 3 tetrahedral meshes. Column *Grid Size* describes the uniform grid configuration, that is composed of two numbers: the first one is the size of the first level grid and the second one is the size of the second level. For example, the configuration $64, 8$ represents a grid with resolution $64x64x64$ in the first level and $8x8x8$ in the second one. The rows are sorted by the product of the sizes of the two levels of the grid. Column *Pairs of triangles* represents the number of pairs of triangles processed: *Grid* is the number of pairs in all grid cells, *Unique* is the number of unique pairs of triangles (since a triangle may be in different cells, some of the pairs may be duplicate) and *Inter.* represents the number of pairs effectively intersecting.

We present the separate times for the steps that mainly depend on the uni-

Table 6.5: 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)	
Mesh 0	Mesh 1	Mesh 0	Mesh 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	4,611	5,043	1.27	259.23
Kitten	RedC.Box[e]	274	1,402	3	13	2.33	0.01
226633	461112	2,452	805	23	128	7.18	0.08
Ramesses	Ram.Tran.[f]	1,653	1,653	36	237	12.38	0.17
Neptune	Nept.Tran.[g]	4,008	4,008	78	647	36.24	0.47
Uniform grid							
		# faces ($\times 10^3$)		# int.[a]	Int.tests[b]	Time (s)	
Mesh 0	Mesh 1	Mesh 0	Mesh 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	5,351	0.25	63.80
Kitten	RedC.Box[e]	274	1,402	3	27	0.08	0.02
226633	461112	2,452	805	23	307	0.16	0.05
Ramesses	Ram.Tran.[f]	1,653	1,653	36	866	0.16	0.10
Neptune	Nept.Tran.[g]	4,008	4,008	78	5,087	0.27	0.35

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

form grid size: the creation of the grid (column *Grid*), the intersection of pairs of triangles (column *Inter.*) and the triangle classification (column *Class.*). Total time represents the total time of the program (excluding I/O) and, thus, it also includes

Table 6.6: Times spent intersecting meshes using different configurations for the uniform grid sizes. The rows detached in boldface represent the configurations chosen using the strategy described in Section 6.5.4.

Mesh 0: 226633 (2M triangles), Mesh 1: 461112 (805K triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size[a]	Grid[b]	Unique[c]	Inter.[d]	(GB)	Grid[e]	Inter.[f]	Class.[g]	Total
16,8	6,631	6,184	23	1.68	0.09	0.48	0.40	1.12
16,16	2,023	1,633	23	1.67	0.11	0.24	0.49	1.00
32,8	2,023	1,633	23	1.66	0.09	0.25	0.48	0.97
16,32	1,066	603	23	1.85	0.11	0.20	0.46	0.91
32,16	1,066	603	23	1.72	0.09	0.19	0.46	0.88
64,8	**1,066**	**603**	**23**	**1.67**	**0.12**	**0.19**	**0.44**	**0.90**
16,64	1,057	307	23	3.09	0.26	0.26	0.48	1.15
32,32	1,057	307	23	2.12	0.14	0.22	0.48	0.99
64,16	1,057	307	23	1.81	0.14	0.20	0.48	0.97
32,64	2,096	204	23	5.15	0.50	0.34	0.50	1.50
64,32	2,096	204	23	2.68	0.27	0.27	0.50	1.19
64,64	7,061	163	23	9.36	1.06	0.60	0.47	2.27
Mesh 0: Armadillo (346K triangles), Mesh 1: Armadillo (346K triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size[a]	Grid[b]	Unique[c]	Inter.[d]	(GB)	Grid[e]	Inter.[f]	Class.[g]	Total
16,8	17,727	14,326	50	1.15	0.03	63.45	0.49	67.48
16,16	13,670	7,855	50	0.81	0.03	61.36	0.50	65.45
32,8	13,670	7,855	50	0.85	0.05	60.83	0.50	64.89
16,32	18,758	5,669	50	1.18	0.06	62.70	0.47	66.72
32,16	18,758	5,669	50	1.04	0.07	61.51	0.50	65.60
64,8	**18,758**	**5,669**	**50**	**0.98**	**0.08**	**61.52**	**0.50**	**65.62**
16,64	45,935	5,351	50	3.94	0.33	61.72	0.55	66.13
32,32	45,935	5,351	50	2.98	0.22	61.10	0.57	65.41
64,16	45,935	5,351	50	2.56	0.25	62.49	0.56	66.82
32,64	170,098	5,198	50	15.31	1.34	63.14	0.67	68.69
64,32	170,098	5,198	50	10.82	0.95	65.42	0.69	70.60
64,64	871,209	5,119	50	69.40	6.34	70.70	0.83	81.40

[a] resolution of the first and second level grids;
[b] total number of pairs of triangles in the grid cells; [c] number of unique pairs of triangles; [d] number of intersecting pairs of triangles; [e] create the uniform grid; [f] detect the intersections; [g] classify the output triangles.

the time of the retesselation step (this step does not use the grid). Furthermore, column *Memory* presents the peak memory used by 3D-EPUG-Overlay.

In general, small uniform grids can be constructed more quickly but, on the other hand, they lead to more pairs of triangles being tested for intersection. Grids with too high resolutions, on the other hand, are slower to be constructed and, also,

Table 6.7: Times spent intersecting meshes using different configurations for the uniform grid sizes (continued). The rows detached in boldface represent the configurations chosen using the strategy described in Section 6.5.4.

Mesh 0: Ramesses (2M triangles), Mesh 1: Ramesses.rot.[h](2M triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size[a]	Grid[b]	Unique[c]	Inter.[d]	(GB)	Grid[e]	Inter.[f]	Class.[g]	Total
16,8	94,302	90,616	60	4.59	0.11	6.20	0.72	7.72
16,16	22,585	19,852	60	2.31	0.11	1.52	0.59	2.88
32,8	22,585	19,852	60	2.27	0.12	1.41	0.58	2.79
16,32	8,287	5,748	60	1.92	0.13	0.76	0.53	2.13
32,16	8,287	5,748	60	1.84	0.13	0.77	0.61	2.18
64,8	8,287	5,748	60	1.79	0.13	0.74	0.57	2.10
16,64	5,486	2,275	60	3.08	0.27	0.67	0.57	2.21
32,32	5,486	2,275	60	2.44	0.19	0.59	0.57	1.98
64,16	**5,486**	**2,275**	**60**	**2.08**	**0.18**	**0.61**	**0.60**	**2.11**
32,64	7,365	1,240	60	7.00	0.74	0.90	0.55	2.90
64,32	7,365	1,240	60	4.08	0.42	0.70	0.60	2.42
64,64	18,899	865	60	19.26	2.31	1.80	0.52	5.29
Mesh 0: Neptune (4M triangles), Mesh 1: Neptune.transl.[i](4M triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size[a]	Grid[b]	Unique[c]	Inter.[d]	(GB)	Grid[e]	Inter.[f]	Class.[g]	Total
32,8	84,108	81,174	78	6.81	0.26	5.12	1.11	6.90
32,16	19,592	17,286	78	4.22	0.31	1.57	1.18	3.48
64,8	19,592	17,286	78	4.06	0.26	1.45	1.13	3.25
32,32	7,503	5,069	78	4.32	0.31	0.85	1.12	2.73
64,16	**7,503**	**5,069**	**78**	**4.05**	**0.30**	**0.89**	**1.13**	**2.78**
32,64	5,929	2,123	78	7.36	0.67	0.86	1.11	3.07
64,32	5,929	2,123	78	5.37	0.48	0.79	1.08	2.79
64,64	10,599	1,234	78	15.16	1.70	1.45	1.08	4.65

[a] resolution of the first and second level grids;
[b] total number of pairs of triangles in the grid cells; [c] number of unique pairs of triangles; [d] number of intersecting pairs of triangles; [e] create the uniform grid; [f] detect the intersections; [g] classify the output triangles;
[h] abbreviation of Ramesses Rotated; [i] abbreviation of Neptune Translated.

Table 6.8: Times spent intersecting tetrahedral meshes using different configurations for the uniform grid sizes. The rows detached in boldface represent the configurations chosen using the strategy described in Section 6.5.4.

Mesh 0: 68380Tetra (1M triangles), Mesh 1: 914686Tetra (605K triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size[a]	Grid[b]	Unique[c]	Inter.[d]	(GB)	Grid[e]	Inter.[f]	Class.[g]	Total
16,2	276,689	222,000	3,222	9.65	0.04	32.89	3.44	64.59
16,4	147,820	80,877	3,222	6.13	0.05	22.59	2.51	53.39
32,2	147,820	80,877	3,222	6.36	0.05	22.86	2.48	54.38
16,8	161,778	41,707	3,222	6.10	0.06	20.86	2.34	51.03
32,4	161,778	41,707	3,222	6.12	0.06	20.36	2.31	50.41
64,2	**161,778**	**41,707**	**3,222**	**6.08**	**0.11**	**20.28**	**2.24**	**50.83**
16,16	359,051	28,118	3,222	11.45	0.11	22.49	2.34	52.10
32,8	359,051	28,118	3,222	10.69	0.11	20.96	2.20	51.93
64,4	359,051	28,118	3,222	10.43	0.15	20.74	2.22	51.45
32,16	1,354,640	22,475	3,222	38.93	0.32	31.34	2.43	61.66
64,8	1,354,640	22,475	3,222	38.86	0.41	30.77	2.24	61.61
Mesh 0: 518092Tetra (6M triangles), Mesh 1: 461112Tetra (8M triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size[a]	Grid[b]	Unique[c]	Inter.[d]	(GB)	Grid[e]	Inter.[f]	Class.[g]	Total
16,2	710,086	573,863	5,345	22.62	0.27	69.91	6.22	144.70
16,4	559,230	337,272	5,345	18.57	0.27	53.98	5.75	128.59
32,2	559,230	337,272	5,345	19.03	0.26	54.13	5.50	129.11
16,8	729,436	255,703	5,345	21.58	0.28	52.27	5.60	125.83
32,4	729,436	255,703	5,345	23.09	0.26	52.49	5.43	128.15
64,2	729,436	255,703	5,345	22.12	0.37	52.35	5.49	127.23
16,16	1,524,993	220,650	5,345	41.00	0.38	62.82	5.66	138.99
32,8	1,524,993	220,650	5,345	39.26	0.34	60.90	5.46	136.47
64,4	**1,524,993**	**220,650**	**5,345**	**41.41**	**0.44**	**61.52**	**5.53**	**136.71**
64,8	5,020,036	205,020	5,345	103.10	0.85	122.76	5.20	198.22
Mesh 0: ArmadilloTet. (3M triangles), Mesh 1: Arm.Tetr.Transl. (3M triangles)								
Grid	Pairs of triangles ($\times 10^3$)			Memory	Time (s)			
Size	Grid	Unique	Inter.	(GB)	Grid	Inter.	Class.	Total
64,2	895,394	520,640	21,001	39.09	0.22	155.97	16.99	279.09
64,4	**902,735**	**259,436**	**21,001**	**33.23**	**0.26**	**142.7**	**14.55**	**263.11**
64,8	1,819,576	168,122	21,001	51.00	0.46	152.19	13.71	271.44

[a] resolution of the first and second level grids; [b] total number of pairs of triangles in the grid cells; [c] number of unique pairs of triangles;
[d] number of intersecting pairs of triangles; [e] create the uniform grid;
[f] detect the intersections; [g] classify the output triangles.

have more pairs of triangles. The reason why too-high resolution grids can have more pairs of triangles in their cells is because when the cells become smaller than the triangles the higher the resolution the more cells the bounding-box of each triangle intersect on average. As a result, the same pair of triangles may be in different cells. To reduce the number of pairs tested for intersection, before the tests we remove the duplicate pairs from the list of pairs of triangles to evaluate.

For example, for the first set of experiments (intersection of mesh 226633 with 461112) if the first level grid is set to 16^3 cells and the second level is set to 8^3 cells, 6 million pairs of triangles are tested for intersection. This number is reduced to 1.6 million when the second level grid is increased to 16^8 cells (i.e., a 8 times larger grid) and, as result, the time processing intersections is reduced by 50%.

In general, the time spent processing intersections did not change in the same proportion that the number of pairs of unique triangles tested for intersection changed. The reason is because the intersection time includes some steps that run slower and others that run faster for finer grids. Example: traversing the grid generating the list of pairs of triangles to test is slower for finer grids (since more cells have to be traversed), processing a cell tends to be faster for finer grids (since the number of triangles per cell is reduced) and testing pairs of triangles for intersection is faster the smaller the number of pairs is. Furthermore, the intersection phase of the algorithm has a step whose time does not depend on the grid resolution: computing the coordinates of the vertices generated by the intersection (the time of this step depends on the number of intersections found). Finally, testing a pair of triangles for intersection is, on average, faster when the pair does not intersect (since the algorithm can terminate earlier) and, thus, the proportion of intersecting pairs in the list of pairs to test influentiates the running time.

As result, in general the total time spent by 3D-EPUG-Overlay varies slowly for different grid configurations. For example, considering the intersection of meshes 226633 and 461112 (first set of experiments on Table 6.6), the maximum total time observed was 2.6 times greater than the minimum one even though we performed experiments with several grid configurations such that the size of the second level grid cells in the largest resolution $(64, 64)$ was $32,768$ times smaller

than the size of the cells in the experiments using the smallest resolution $(16, 8)$. Similar results were observed in all other experiments.

If the grid is too coarse for a mesh, on the other hand, the memory usage (due to a high number of pairs of triangles being tested for intersection) and the processing time may become too high. For example, during the intersection of *Neptune* and *Neptune translated* when a first level grid with 16^3 cells and a second level grid with 8^3 cells were employed the memory usage grew to more than the available memory in the testing machine and we aborted the execution of 3D-EPUG-Overlay. This happened because the total number of pairs of triangles in all uniform grid cells was 88 billion pairs. This suggests that, even though the range of configurations with reasonable performance is broad, at some point a wrong choice for the grid configuration will lead to a poor performance.

We propose sizing the uniform grid such that the expected number of pairs of triangles tested for intersection in each cell is a small constant. Since the number of expected pairs per cell (np) in a uniformly independently and identically distributed dataset (and assuming each triangle of meshes M_0 and M_1 is inserted in, respectively, k_0 and k_1 cells on average) would be $np = \frac{k_0 \times n_0}{G_1^3 \times G_2^3} \times \frac{k_1 \times n_1}{G_1^3 \times G_2^3}$ (where n_0 and n_1 are the number of triangles in the each mesh and G_1 and G_2 are, respectively, the resolution of the first and second level grids), then $G_1 \times G_2 = \sqrt[6]{\frac{k_0 \times k_1 \times n_0 \times n_1}{np}}$.

Experiments suggest that using $\frac{k_0 \times k_1}{np} = 100000$ leads to reasonable performance. For example, for intersecting the meshes 226633 and 461112, if $\frac{k_0 \times k_1}{np} = 100000$, then $G_1 \times G_2 = 763$. If we round 763 to the next power of 2 (for simplicity we decided to perform all experiments using powers of 2 for the grid resolutions, but this is not a restriction of 3D-EPUG-Overlay) and fix the first level grid at 64, then the size of the second level grid should be 8. Observe that this configuration is close to the best configuration in Table 6.6.

For tetra meshes, we observed that the best results are obtained with a smaller target number of expected triangles per cell. This can be explained because, differently from meshes without internal structure (where all triangles are concentrated in the boundary), tetrahedral meshes typically have triangles in their interior, which makes them more evenly distributed. Thus, we set $\frac{k_0 \times k_1}{np}$ to 10.

We employed this strategy for choosing the grid configuration in all experiments described in this thesis. The rows detached in bold in Tables 6.6, 6.7 and 6.8 represent test cases using these chosen configurations.

While this strategy may not work in all situations, in all experiments we performed the results were reasonable (even in meshes without internal structure, which concentrate all their triangles on their boundaries). The usage of empirical configurations for algorithms with tuning parameters is common in Computer Science. For example, octrees typically have threshold values to determine when they should stop being refined. The algorithm for intersecting pairs of bounding-boxes present in CGAL (and evaluated in Section 6.5.3) also has a cutoff threshold: this algorithm is a hybrid of two other methods and one of its parameters determines in which situation each of the two algorithms will be employed for the computation.

Since the grid creation in 3D-EPUG-Overlay is very fast in comparison with other steps of the algorithm, a possible idea for future work consists in creating an heuristic to automatically choose a reasonable grid resolution. The choice could be tested by trying to create the grid and, verifying, for example, the total number of pairs of triangles in all grid cells. If this choice is considered poor, the heuristic could select a different configuration and try to recreate the grid.

6.5.5 Comparing the performance of 3D-EPUG-Overlay against other methods

We compared 3D-EPUG-Overlay against other three algorithms using the pairs of meshes presented in Table 6.2. The resulting running times (in seconds, excluding I/O) are presented in Table 6.9. 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).

Table 6.10 presents the speedup of 3D-EPUG-Overlay when compared to LibiGL, CGAL and QuickCSG. 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). This can be explained because 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 were not optimized yet) in order to evaluate them considering the perturbed meshes. As future work we intend to optimize these functions in order to be faster even in unusual situations where the special cases happen frequently.

Table 6.9: Times, in seconds, spent by different methods for intersecting pairs of meshes. QuickCSG reported errors during the intersections whose times are flagged with *. Only 3D-EPUG-Overlay was employed in the experiments with tetrahedral meshes (last three rows).

		Time (s)				
				CGAL		
Mesh 0	Mesh 1	3D-Epug	LibiGL	Convert[a]	Intersect[b]	QuickCSG
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.Transl.[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.

It is worth noting that, as mentioned in Section 2.3.2.2, differently from other

algorithms, LibiGL also repairs meshes (by resolving self-intersections) during the intersection computation. Since in the representation employed by 3D-EPUG-Overlay (where each triangle stores the ids of the two polyhedra it bounds) self-intersecting meshes are ambiguous, 3D-EPUG-Overlay does not attempt to perform mesh repair (as mentioned before, experiments were performed only with non self-intersecting meshes).

Because of the overhead associated with the 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.

Table 6.10: Speedup of 3D-EPUG-Overlay when compared against different methods. QuickCSG reported errors during the intersections whose speedups are flagged with *.

			CGAL		
Mesh 0	Mesh 1	LibiGL	Intersect[a]	Total[b]	QuickCSG
Casting10kf	Clutch2kf	8.2	6.9	32.7	**0.4***
Armadillo52kf	Dinausor40kf	22.2	160.1	442.7	0.8
Horse40kf	Cow76kf	23.0	175.9	547.5	0.8
Camel69kf	Armadillo52kf	22.3	180.9	563.3	0.8
Camel	Camel	1.3	16.5	21.0	**0.1***
Camel	Armadillo	63.0	431.5	1,455.8	1.6
Armadillo	Armadillo	1.3	17.9	23.0	**0.1***
461112	461115	71.4	573.5	1,486.3	1.3
Kitten	RedCircBox	82.0	943.4	3,289.8	3.0
Bimba	Vase	101.1	795.1	2,490.5	1.9
226633	461112	101.1	953.5	2,768.8	**2.4***
Ramesses	Ram.Transl.[c]	73.7	750.1	1,986.0	**1.9***
Ramesses	Ram.Rot.[d]	58.0	470.4	1,219.9	1.2
Neptune	Ramesses	98.8	1,284.3	4,241.2	3.4
Neptune	Nept.Transl.[e]	81.5	1,008.4	3,002.3	2.2

[a] Speedup when the time spent by 3D-EPUG-Overlay is compared only against the CGAL intersection time; [b] Speedup when the time spent by 3D-EPUG-Overlay is compared against the total CGAL time; [c] Ramesses Translated; [d] Ramesses Rotated; [e] Neptune Translated;

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. Besides reporting errors during the experiments detached with a * in Table 6.9, QuickCSG also failed in situations where errors were not reported (this will be detailed later).

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. For example, during the intersection of the *Neptune* with the *Neptune translated* datasets (two meshes without internal structure), there are 78 thousand pairs of intersecting triangles and the resulting mesh contains 3 million triangles. On the other hand, in the intersection of *518092_tetra* (a mesh with 6 million triangles and 3 million tetrahedra) with *461112_tetra* (a mesh with 8 million triangles and 4 million tetrahedra) there are 5 million pairs of intersecting triangles and the output contains 23 million triangles.

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. The results are presented in Table 6.11. Except for the smallest mesh and self-intersections (where 3D-EPUG-Overlay used 13% more memory than LibiGL), in almost all test cases 3D-EPUG-Overlay used less memory than LibiGL (in the intersection of Neptune with Ramesses, for example, LibiGL used 2.6 times more memory than 3D-EPUG-Overlay). Because of its heavy data structures employed to represent Nef Polyhedra, CGAL used much more memory than 3D-EPUG-Overlay (up to 40 times more memory). Even QuickCSG (an inexact algorithm) was less memory

efficient than 3D-EPUG-Overlay: the only situations where QuickCSG used less memory than 3D-EPUG-Overlay were in two test cases where it failed.

Table 6.11: Memory usage (in GB) of the different algorithms. QuickCSG reported errors during the intersections whose entries are flagged with *.

Mesh 0	Mesh 1	3D-Epug	LibiGL	CGAL	QuickCSG
Casting10kf	Clutch2kf	0.05	0.03	0.21	**0.08***
Armadillo52kf	Dinausor40kf	0.08	0.13	1.29	0.13
Horse40kf	Cow76kf	0.09	0.15	1.97	0.15
Camel69kf	Armadillo52kf	0.09	0.16	1.67	0.15
Camel	Camel	0.21	0.20	3.32	**0.17***
Camel	Armadillo	0.21	0.53	7.88	0.34
Armadillo	Armadillo	1.02	0.90	20.79	**0.53***
461112	461115	0.96	2.18	24.12	1.06
Kitten	RedCircBox	0.78	1.92	31.41	1.09
Bimba	Vase	1.00	2.43	37.57	1.27
226633	461112	1.68	3.92	51.32	**1.99***
Ramesses	Ram.Transl.[a]	1.77	4.17	48.24	**2.02***
Ramesses	Ram.Rot.[b]	2.10	4.44	48.78	2.06
Neptune	Ramesses	2.58	6.66	83.52	3.32
Neptune	Nept.Transl.[c]	4.06	9.95	115.39	4.66
68380Tet.[d]	914686Tet.[e]	6.13	-	-	-
Armad.Tet.[f]	Arm.Tet.Transl.[g]	33.00	-	-	-
518092Tetra	461112Tetra	42.84	-	-	-

[a] Ramesses Translated; [b] Ramesses Rotated; [c] Neptune Translated;
[d] 68380Tetra; [e] 914686Tetra; [f] ArmadilloTetra;
[g] ArmadilloTetra Translated.

6.5.6 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, a correct algorithm does not ensure that its implementation is bug-free. In order to have evidence that the implementation is correct, we performed experiments comparing it against LibiGL (as reference solution).

We employed the Metro tool [93] 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.

Let $e(p, S)$ be the minimum Euclidean distance between the point p and the surface S. [93] defines the one sided distance $E(S_1, S_2)$ between two surfaces S_1 and S_2 as: $E(S_1, S_2) = \max_{p \in S_1} e(p, S_2)$. The Hausdorff distance between two surfaces S_1 and S_2 is the maximum between $E(S_1, S_2)$ and $E(S_2, S_1)$. The Metro implementation employs an approximation strategy that samples points on the surface of the meshes in order to compute the Hausdorff distance. In all experiments we employed the default parameters (where 10 points are sampled per face).

As mentioned by the authors [93], Metro is not exact (all the computation is performed using double variables), and thus we intend to use the distance between meshes only as evidence that our implementation is correct.

Table 6.12 reports the distances between the meshes. Since we are evaluating several models (with different scales), we configured Metro to report the distance as a percentage of the diagonal of the bounding-box of the meshes instead of the absolute error values.

Table 6.12: Hausdorff distances (normalized as a percentage of the bounding-box) between the output meshes generated by the reference method (LibiGL) and 3 other algorithms.

		Difference (%)		
Mesh 0	Mesh 1	3D-Epug	CGAL	QuickCSG
Casting10kf	Clutch2kf	0.0000	0.0000	-
Armadillo52kf	Dinausor40kf	0.0000	0.0001	0.1181
Horse40kf	Cow76kf	0.0000	0.0001	0.0490
Camel69kf	Armadillo52kf	0.0000	0.0001	0.1254
Camel	Camel	0.0000	0.0000	-
Camel	Armadillo	0.0000	0.0001	0.1121
Armadillo	Armadillo	0.0000	0.0000	-
461112	461115	0.0000	0.0002	0.0119
Kitten	RedCircBox	0.0000	0.0005	0.1020
Bimba	Vase	0.0000	0.0001	0.0847
226633	461112	0.0000	0.0003	-
Ramesses	Ram.Transl. [a]	0.0000	0.0007	-
Ramesses	Ram.Rot. [b]	0.0000	0.0007	0.0465
Neptune	Ramesses	0.0000	0.0007	0.0386
Neptune	Nept.Transl. [c]	0.0000	0.0004	0.0149

[a] Ramesses Translated; [b] Ramesses Rotated;
[c] Neptune Translated.

As it can be seen, in all test cases 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): this can be explained because, even though CGAL is exact, the results are stored using floating-point variables and different strategies to round the vertices coordinates to floating-point numbers and write the values to the text file representing the meshes may lead to slightly different results.

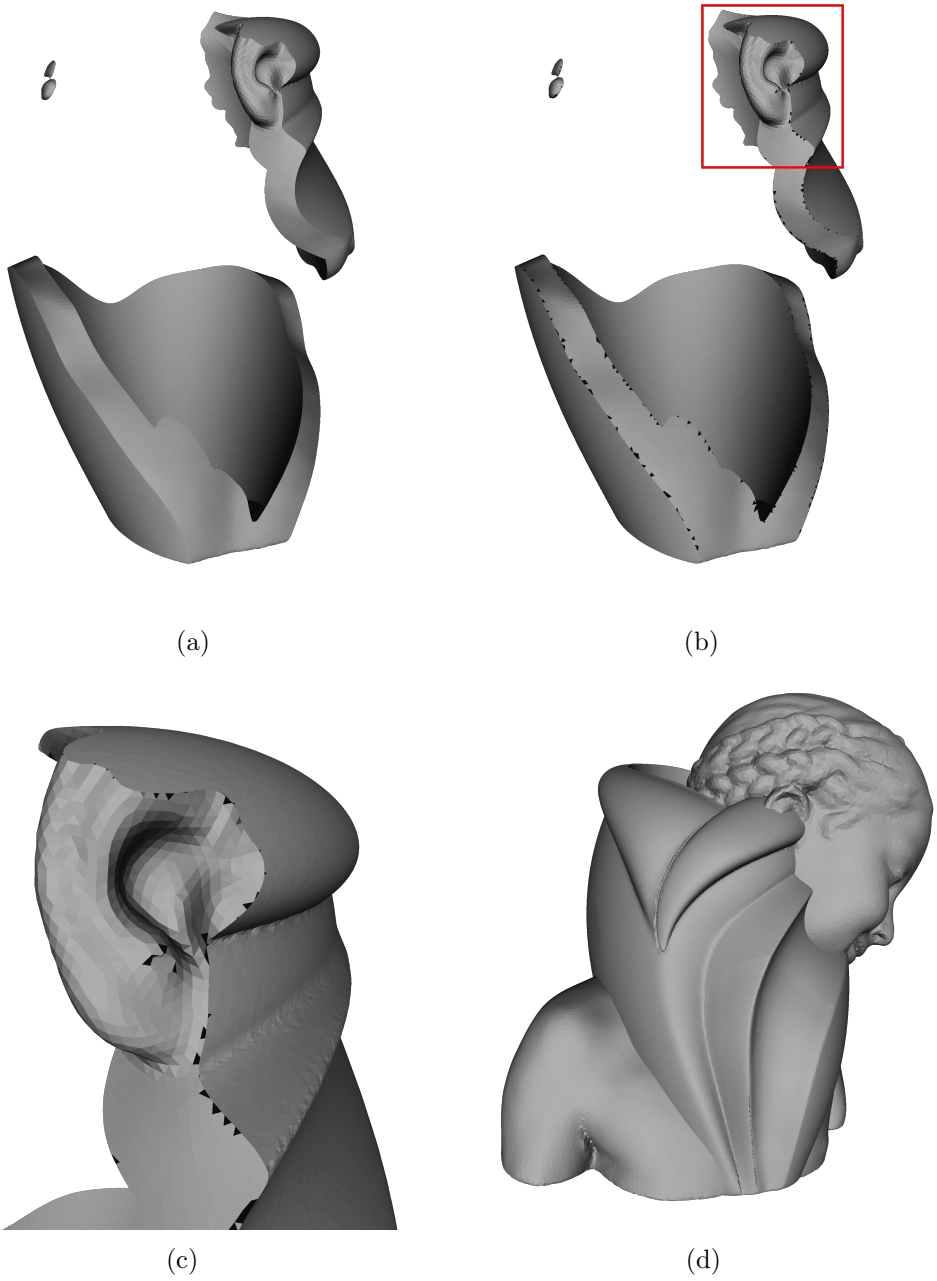
QuickCSG, on the other hand, generated errors much larger than the ones generated by CGAL: in the worst case, the difference between QuickCSG output and LibiGL was 0.13% of the diagonal of the bounding-box). Also, as said before, in some situations QuickCSG reported failure to intersect the meshes (indicated by a dash in Table 6.12).

6.5.6.1 Visual inspection

We also performed visual inspection (using the MeshLab tool) in order to verify the results. 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) can often be identified visually mainly in small meshes.

As mentioned before, QuickCSG reported failures during the intersection of several meshes. Furthermore, even during some intersections where errors have not been reported the output results were frequently inconsistent, presenting problems such as open meshes, spurious triangles or inconsistent orientations.

Figure 6.10 illustrates one test case where QuickCSG did not report any error during the processing, but visual inspection found several triangles with inconsistent orientations. As can be seen, some triangles are oriented incorrectly (since the camera is in the exterior of the meshes, the visible sides of the triangles should be colored the same way MeshLab colors the external sides of the triangles, i.e., with a light color). 3D-EPUG-Overlay, on the other hand, generated consistent results (Figure 6.10 (a)).



(a) (b)

(c) (d)

Figure 6.10: Intersection of the Bimba and Vase meshes computed by 3D-EPUG-Overlay (a), QuickCSG (b), zoom (of the red square from (b)) presenting the details of some errors in the QuickCSG result (c) and the two input meshes presented on together (d) (figures not to scale).

Figure 6.11 (a) presents a zoom in the output of QuickCSG for the intersection of the Ramesses dataset with Ramesses Translated: some triangles are oriented incorrectly. These errors may be created either by floating-point errors or because

QuickCSG doesn't handle the coincidences.

To mitigate this later problem, QuickCSG provides options where the user can apply a random perturbation in the input dataset. In contrast to the symbolic perturbations employed by 3D-EPUG-Overlay (that are conceptual and use indeterminate infinitesimals), these numerical perturbations are not guaranteed to work and the user has to choose the maximum range for them. A too small range may not eliminate all errors while a too big range may modify the mesh too much. Figures 6.11 (b) and (c) and (d) display the results obtained when perturbations with maximum range 10^{-1}, 10^{-3} and 10^{-6} were employed. As it can be seen, none of these perturbations removed all errors and the bigger perturbation (10^{-1}) even added undesirable artifacts to the output. Similar problems in QuickCSG have been reported by Zhou et al. [64].

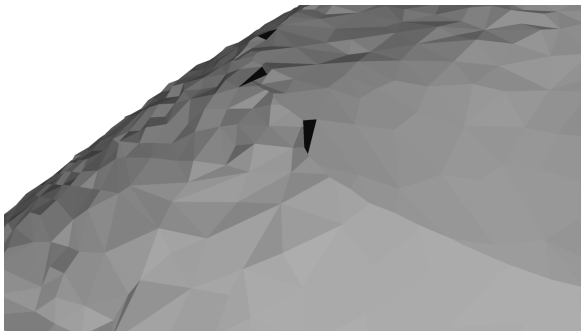
6.5.6.2 Rotation invariance

We also performed experiments to validate 3D-EPUG-Overlay by verifying that its result does not change when the input meshes are rotated. I.e., given a pair of meshes, they were rotated around the same point, intersected, and then the resulting mesh was rotated back. To ensure exactness, we chose a rotation angle whose sine and cosine are rational numbers.

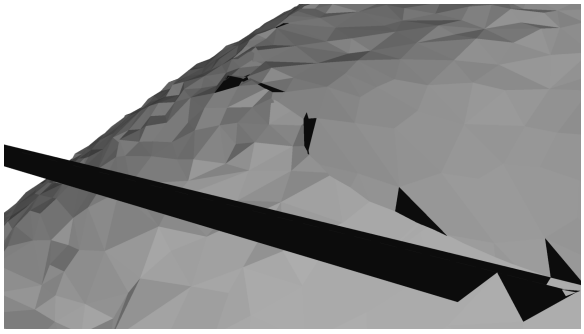
We evaluated all pairs of meshes presented in Table 6.2. For each pair, we performed a rotation around the x axis and, then, a rotation around the y axis (the origin was defined as the center of the joint bounding-box of the two meshes). We chose a rotation angle whose sine and cosine are, respectively, $\frac{400}{10{,}004}$ and $\frac{9{,}996}{10{,}004}$ (this angle measures approximately 2.29 degrees).

In all the experiments Metro reported that the resulting meshes were equal (i.e., the Hausdorff distance was 0.000000) to the corresponding ones obtained without rotation.

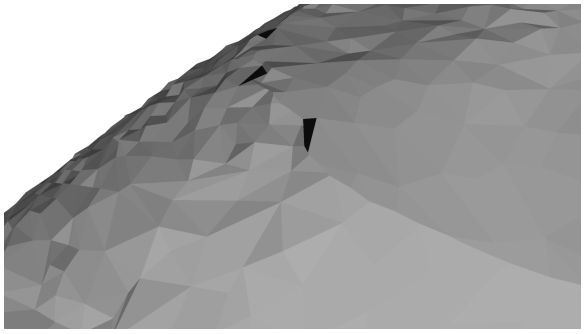
Furthermore, experiments where each mesh from Table 6.2 was intersected with a rotated version of itself were performed. The reasoning of this experiment is that intersecting a mesh with a slightly rotated version of itself could generate a large number of intersections and small triangles (thus it is challenging to the



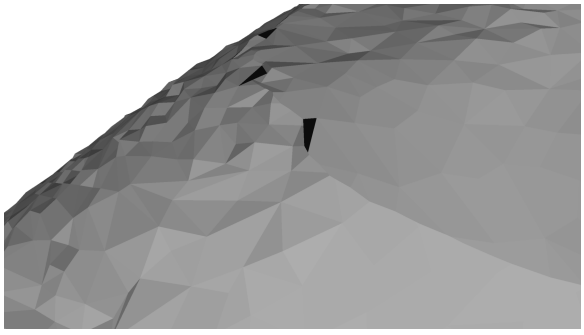
(a)



(b)



(c)



(d)

Figure 6.11: Detail of the intersection of Ramesses with Ramesses Translated generated by QuickCSG using different ranges for the numerical perturbation: no perturbation (a), 10^{-1} (b), 10^{-3} (c) and 10^{-6} (d).

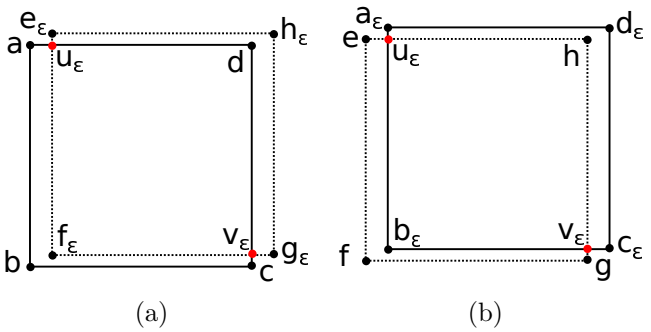
algorithms). Each mesh M was rotated (generating another mesh M_r) around the center of its bounding-box using the angle with rational sine and cosine presented above (approximately 2.29 degrees), and then the intersection of M with M_r was computed using LibiGL and 3D-EPUG-Overlay. In all experiments the results obtained by the two algorithms were considered to be equal by Metro (i.e., the Hausdorff distance between the two outputs was 0.000000).

6.5.7 Limitations

Even though all the computation is performed exactly, common file formats for 3D objects such as OFF represent data using floating-point numbers. The process of converting the rational output into floating-point numbers may introduce errors since not all rationals can be represented exactly as floating-point numbers. While the input coordinates are always read exactly, in this work the output files were created by simply converting the rationals to floating-point numbers, and thus they are not guaranteed to be always correct. Approaches to solve this problem include avoiding the conversion (i.e., always employing multiple-precision rationals in the representations) or using heuristics such as the one presented by Zhou et. al. [64] in order to try to choose floating-point numbers for each coordinate such that the approximate output will not only be similar to the exact one, but also it will not present topological errors.

A limitation of the symbolic perturbation is that the results are consistent considering the perturbed dataset, not necessarily considering the original one [4]. 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).

To illustrate this problem, let us first consider an example (in 2D) where removing the perturbation of the output generates a mesh without degenerate features. Figure 6.12 illustrates the intersection of two coincident squares $abcd$ (square 0) and $efgh$ (square 1): Figure 6.12 (a) assumes that the square 0 is not perturbed and square 1 is translated by infinitesimals before the intersection is computed (gener-



(a) (b)

Figure 6.12: Effect of two different perturbations during the self intersection of squares - (a) Two squares intersect at a common edge (ad and fg), (b) The top rectangle is perturbed (translated by positive infinitesimals), (c) The bottom rectangle is perturbed.

ating square $e_\epsilon f_\epsilon g_\epsilon h_\epsilon$). The resulting intersection of the $abcd$ with $e_\epsilon f_\epsilon g_\epsilon h_\epsilon$ is the square $u_\epsilon f_\epsilon v_\epsilon d_\epsilon$, where u_ϵ, for example, represents the intersection of edges $e_\epsilon f_\epsilon$ and ad. If the perturbation in the output is ignored (i.e., if the infinitesimals in the coordinates of the output vertices are dropped, generating $ufvd$ as output instead of $u_\epsilon f_\epsilon v_\epsilon d_\epsilon$), the resulting square $ufvd$ will be identical to the two unperturbed input squares $abcd$ and $efgh$.

Figure 6.12 (b) assumes that square 1 is not perturbed and square 0 is translated by infinitesimals before the intersection computation. Similarly to example in Figure 6.12 (a), $ubvh$ (the unperturbed version of the output $u_\epsilon b_\epsilon v_\epsilon h_\epsilon$) will be identical to the input squares $abcd$ and $efgh$.

Therefore, in both examples illustrated above the perturbations can be safely ignored when the output is generated and the resulting polygons are the expected result for the intersection.

Figure 6.13 illustrates (in 2D) a situation where extracting a valid unperturbed mesh is a challenge: if two squares $abcd$ and $efgh$ intersect exactly at edges ad and fg (this is illustrated in Figure 6.13 (a)), then an infinitesimal perturbation may lead to two possible results: if $efgh$ is translated by positive infinitesimals (and $abcd$ is not modified), then the intersection of $abcd$ with $e_\epsilon f_\epsilon g_\epsilon h_\epsilon$ will be empty (Figure 6.13 (b)). On the other hand, if $efgh$ is not translated and $abcd$ is translated (generating $a_\epsilon b_\epsilon c_\epsilon d_\epsilon$), then the intersection will be the rectangle $a_\epsilon u_\epsilon g_\epsilon v_\epsilon$, that will have infinitesimal area (since $u = a$ and $g = v$).

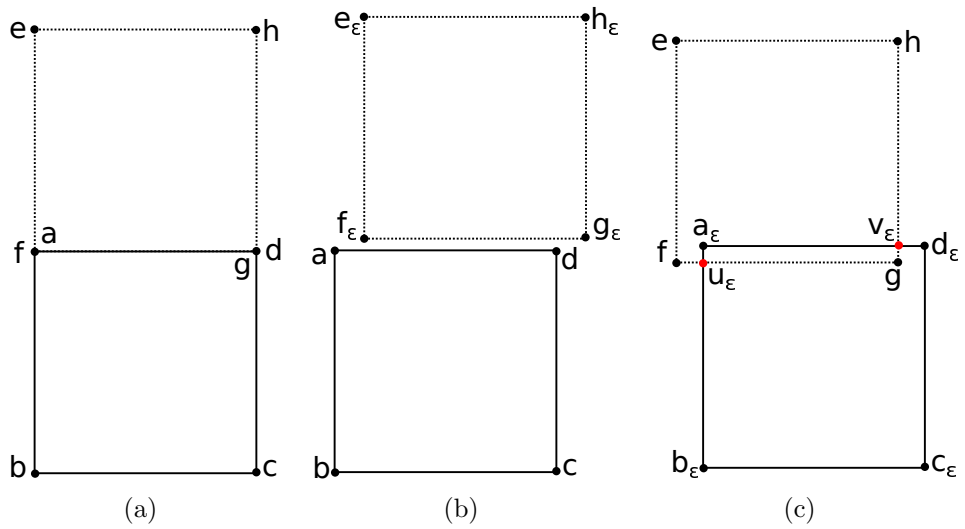


Figure 6.13: Challenge caused by perturbations during the intersection of two squares that intersect at an edge - (a) the two squares intersect at an edge. (b) square $efgh$ is perturbed by positive infinitesimals (intersection is empty). (c) square $abcd$ is perturbed by positive infinitesimals (intersection is an infinitesimal rectangle $a_\epsilon u_\epsilon g_\epsilon v_\epsilon$).

Thus, even though there is no degeneracy in the perturbed output, if the infinitesimal components of $a_\epsilon u_\epsilon g_\epsilon v_\epsilon$ are dropped (for example, when the output is stored in a file) the resulting output will have a degenerate rectangle.

In the experiments performed in this thesis the output files were generated by ignoring the symbolic perturbation and outputting the rational coordinates of the vertices (converted to floating-point numbers). Therefore, even though the resulting perturbed meshes are guaranteed to be correct and free of degenerate features, the meshes stored in the output files are not guaranteed to always retain these properties.

An interesting direction for future work is to develop an algorithm for cleaning the perturbed output dataset when the user does not want a perturbed output (for example, a post-processing could regularize the mesh removing polyhedra with volume 0), generating an unperturbed mesh that is consistent with the non-degenerate features of the perturbed one. Another alternative is to process the output mesh using only algorithms that are aware of the symbolic perturbation (i.e., the pipeline of algorithms processing a dataset should not ignore the infinitesimal factors of the coordinates).

6.6 Summary

We have presented 3D-EPUG-Overlay, an algorithm to intersect a pair of 3D triangular meshes. Except for the indexing (which is necessary only for performance), we showed that all the geometric functions employed by the algorithm can be expressed using 1D, 2D and 3D orientation predicates.

Implementing the symbolic perturbation is simplified since fewer lower level predicates are used, and only these predicates have to directly deal with the perturbation.

Since the perturbations are not expected to modify the results of the functions when there is no coincidence, one can implement the higher level functions using possibly faster strategies and only employ the implementation using orientations when a coincidence is detected.

We showed that the symbolic perturbation eliminates the special cases and, thus, the algorithm can correctly handle all valid inputs. Since all the special cases are properly handled by the perturbation and all computation is exact, the algorithm can exactly compute the intersection of any valid input.

3D-EPUG-Overlay was designed to parallelize very well. Since no global topology needs to be maintained, the individual triangles of the two input meshes can be processed individually in parallel. The process is mostly a series of map-reduce operations. Therefore our implementation can build upon any of many existing, well constructed, parallel tools.

3D-EPUG-Overlay was implemented in C++ using OpenMP to parallelize the computation and GMPXX to provide exact arithmetic. Furthermore, arithmetic filtering (implemented using the CGAL interval arithmetic number type) was employed to accelerate the exact computation, employing multiple-precision rationals in the predicates only when really necessary. 3D-EPUG-Overlay was up to 101 times faster than the parallel and exact algorithm available in LibiGL and up to $1,284$ times faster than the exact algorithm available in CGAL (or up to $4,241$ times faster than CGAL if the time CGAL spends converting the meshes to its Nef Polyhedra representation is taken into consideration). Also, it was faster than QuickCSG (a very fast, but inexact, parallel algorithm) in most of the experiments.

The excellent performance of 3D-EPUG-Overlay allied with its robustness makes it suitable to be used as a subroutine in larger systems such as 3D GIS or CAD systems.

While we decided to focus on the problem of computing intersections, 3D-EPUG-Overlay can be trivially adapted to compute other kinds of overlays (such as union, difference, exclusive-or, etc). Indeed, the only required modification is in the classification step.

CHAPTER 7
Conclusion and future work

We have showed that, by employing a combination of techniques, it is possible to develop efficient exact algorithms for handling geometric data.

3D-EPUG-Overlay, the main result of this thesis, is a parallel algorithm for exactly intersecting pairs of 3D triangular meshes. It employs a combination of five main techniques to achieve correctness and performance. Exact arithmetic is employed to completely avoid errors caused by floating point numbers. Special cases are treated using *Simulation of Simplicity* (SoS). The computation is performed using simple local information to make the algorithm easily parallelizable and to easily ensure robustness. An efficient index and parallel programming are employed for performance.

Before developing 3D-EPUG-Overlay, two simpler algorithms were developed using the five aforementioned techniques. The objective was to evaluate these strategies and iteratively improve them before completing the implementation of 3D-EPUG-Overlay.

The first algorithm we developed was EPUG-Overlay, an efficient algorithm for exactly intersecting 2D maps. EPUG-Overlay applies all the five techniques mentioned above to efficiently compute the exact overlay of 2D polygonal maps. Even though EPUG-Overlay is parallel, even if executed using only one thread it is faster than the inexact overlay algorithm available in GRASS GIS.

Then, we extended some of the ideas employed in EPUG-Overlay to 3D. More specifically, we showed how a 3D two-level uniform grid for indexing triangular meshes could efficiently be constructed in parallel. Then, PinMesh, an exact and efficient algorithm for locating points in 3D meshes, was presented. PinMesh can efficiently locate points in meshes containing millions of triangles. Indeed, according to our experiments it was up to 27 times faster than the RCT algorithm (that, to the best of our knowledge, is the state of the art inexact algorithm for point location).

Finally, the experience acquired during the development of EPUG-Overlay

and PinMesh was employed to develop 3D-EPUG-Overlay. Besides applying all the techniques previously employed in EPUG-Overlay and PinMesh, we further improved 3D-EPUG-Overlay by using a better strategy for creating the uniform grid in parallel and by employing arithmetic filtering to accelerate the evaluation of the exact predicates, what significantly reduced the overhead associated with the operations on multiple precision rationals.

We showed that the symbolic perturbation scheme developed for 3D-EPUG-Overlay eliminates the special cases, and thus the algorithm can correctly handle any valid input. Except for the indexing, 3D-EPUG-Overlay can be implemented using only orientation predicates. As result, only these predicates have to deal with the symbolic perturbation used by SoS, which simplifies the implementation of SoS.

As shown in the experiments, 3D-EPUG-Overlay was not only able to outperform other exact algorithms (being up to 101 times faster than LibiGL and up to $1,284$ faster than CGAL), but also to present a performance similar to an inexact parallel algorithm that was specifically designed to be very efficient. Furthermore, 3D-EPUG-Overlay was more memory efficient than the other methods evaluated.

This excellent performance associated with its exactness makes 3D-EPUG-Overlay suitable for applications where these two characteristics are important. Examples of such applications are CAD/GIS systems, where the user typically deals with big 3D models and needs fast responses when editing these models. Furthermore, since 3D-EPUG-Overlay is parallel it uses better the computing capability of current computers, that usually have multi-core processors.

7.1 Future work

Since the three algorithms presented in this thesis were developed in sequence, some of the techniques employed by 3D-EPUG-Overlay, the method developed lastly, have not been applied to the two previous algorithms. Thus, as future work we intend to use the knowledge acquired during the development of 3D-EPUG-Overlay to improve these other two algorithms. We particularly believe that these two algorithms could benefit from the use of arithmetic filters that, as shown in section 6.5.2, significantly improved the performance of the 3D predicates employed

in our mesh intersection algorithm.

Also, another direction of future work is to adapt 3D-EPUG-Overlay to compute other kinds of overlays (such as union, difference and exclusive-or). Indeed, this adaptation can be easily performed and the only required modification is in the classification step of the algorithm.

We also intend to develop a CGAL kernel containing the predicates employed by 3D-EPUG-Overlay (i.e., predicates adapted to use Simulation of Simplicity). The advantage of having this kernel is that when there is a coincidence the CGAL algorithms will compute results that are consistent with the perturbation scheme employed by 3D-EPUG-Overlay and, as consequence, we will be able to trivially reuse these algorithms in 3D-EPUG-Overlay (for example, the CGAL Delaunay triangulation method could replace the triangulation algorithm implemented in 3D-EPUG-Overlay).

Another future work is to improve the performance of the SoS predicates that are called when coincidences happen. For simplicity, since special cases should happen much less often than the general cases, these predicates have not been optimized yet.

Furthermore, developing new strategies for choosing reasonable configurations for the uniform grid could also be an interesting research topic. While in all the experiments evaluated in this paper the uniform grid performed well, we believe that having a more dynamic way to choose the grid configuration could improve the performance of the algorithm even more. For example, as mentioned in Section 6.5.4, since the grid can be quickly constructed in parallel a possible strategy for choosing the grid size could be to have a set of possible grid configurations, quickly construct the grid using each configuration and choose the one that could lead to the best performance (for example, the configuration that generates the smallest number of pairs of triangles in all grid cells).

Finally, as mentioned in Section 6.5.7, removing the symbolic perturbation of the results and converting the rational coordinates into floating-point numbers may introduce topological errors to the output. Thus, another future work is to study strategies for performing this kind of conversion or for avoiding it.

REFERENCES

[1] S. Zlatanova, A. A. Rahman, and M. Pilouk, "Trends in 3D GIS development," *J. Geospatial Eng.*, vol. 4, no. 2, pp. 71–80, Dec. 2002.

[2] A. knowledge network. (2017) "Autocad INTERSECT (command)". [Online]. Available: http://knowledge.autodesk.com (Retrieved on 10/19/2017).

[3] F. Feito, C. Ogayar, R. Segura, and M. Rivero, "Fast and accurate evaluation of regularized boolean operations on triangulated solids," *Comput. Des.*, vol. 45, no. 3, pp. 705 – 716, Mar. 2013.

[4] H. Edelsbrunner and E. P. Mücke, "Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms," *ACM Trans. Graph.*, vol. 9, no. 1, pp. 66–104, Jan. 1990.

[5] E. S. Agency. (2015) "Ariane 501 inquiry board report". Paris. [Online]. Available: http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf (Retrieved on 06/15/2015).

[6] R. Skeel, "Roundoff error and the patriot missile," *SIAM News*, vol. 25, no. 4, p. 11, July 1992.

[7] M. AKANBI, "Propagation of errors in numerical computations," in *Decision Making Systems Business Administration: Proc. MS'12 Int. Conf.*, vol. 8. World Scientific, 2013, p. 475.

[8] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. K. Yap, "Classroom examples of robustness problems in geometric computations," *Comput. Geom.*, vol. 40, no. 1, pp. 61–78, May 2008.

[9] J. D. Hobby, "Practical segment intersection with finite precision output," *Comput. Geom.*, vol. 13, no. 4, pp. 199–214, Oct. 1999.

[10] M. de Berg, D. Halperin, and M. Overmars, "An intersection-sensitive algorithm for snap rounding," *Computational Geometry*, vol. 36, no. 3, pp. 159–165, Apr. 2007.

[11] J. Hershberger, "Stable snap rounding," *Comput. Geom.*, vol. 46, no. 4, pp. 403–416, May 2013.

[12] A. Belussi, S. Migliorini, M. Negri, and G. Pelagatti, "Snap rounding with restore: An algorithm for producing robust geometric datasets," *ACM Trans. Spatial Algorithms and Syst.*, vol. 2, no. 1, pp. 1:1–1:36, Mar. 2016.

[13] K. Mehlhorn, R. Osbild, and M. Sagraloff, "Reliable and efficient computational geometry via controlled perturbation," in *Proc. 33rd Int. Conf. Automata, Languages and Programming - Volume Part I*, ser. ICALP'06. Berlin, Heidelberg: Springer-Verlag, July 2006, pp. 299–310.

[14] J.-D. Boissonnat and F. P. Preparata, "Robust plane sweep for intersecting segments," *SIAM J. on Comput.*, vol. 29, no. 5, pp. 1401–1421, Mar. 2000.

[15] J. R. Shewchuk, "Adaptive precision floatingpoint arithmetic and fast robust geometric predicates," *Discret. & Comput. Geom.*, vol. 18, no. 3, pp. 305–363, Oct. 1997.

[16] C. Li, "Exact geometric computation: theory and applications," Ph.D. dissertation, Dept. Comp. Sci., Courant Institute - New York Univ., Jan. 2001.

[17] C. Li, S. Pion, and C. K. Yap, "Recent progress in exact geometric computation," *The J. Log. Algebr. Program.*, vol. 64, no. 1, pp. 85–111, July 2005.

[18] C. M. Hoffman, "The problems of accuracy and robustness in geometric computation," *Comput.*, vol. 22, no. 3, pp. 31–40, Mar. 1989.

[19] C. K. Yap, "Towards exact geometric computation," *Comput. Geom.*, vol. 7, no. 12, pp. 3 – 23, Jan. 1997.

[20] S. Pion and A. Fabri, "A generic lazy evaluation scheme for exact geometric computations," *Sci. Comput. Program.*, vol. 76, no. 4, pp. 307 – 323, Apr. 2011.

[21] The CGAL Project, *CGAL User and Reference Manual*, 4.8 ed., 2016, http://doc.cgal.org/4.8/Manual/packages.html (Retrieved on 10/19/2017).

[22] B. Smith. (2010) "Baarle-Nassau / Baarle-Hertog". [Online]. Available: http://ontology.buffalo.edu/smith/baarle.htm (Retrieved on 10/19/2017).

[23] Wikipedia. (2017) "Enclave and exclave — Wikipedia, the free encyclopedia". [Online]. Available: https://en.wikipedia.org/wiki/Enclave_and_exclave (Retrieved on 10/16/2017).

[24] C. J. Ogayar, R. J. Segura, and F. R. Feito, "Point in solid strategies," *Comput. & Graph.*, vol. 29, no. 4, pp. 616 – 624, Aug. 2005.

[25] W. Wang, J. Li, H. Sun, and E. Wu, "Layer-based representation of polyhedrons for point containment tests," *IEEE Trans. Vis. Comput. Graphics*, vol. 14, no. 1, pp. 73–83, Jan 2008.

[26] J. O'Rourke, *Computational Geometry in C*, 2nd ed. New York, NY, USA: Cambridge Univ. Press, 1998.

[27] W. R. Franklin. (2006) "Pnpoly-point inclusion in polygon test". [Online]. Available: http: //www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html (Retrieved on 10/19/2017).

[28] F. R. Feito and J. C. Torres, "Inclusion test for general polyhedra," *Comput. & Graph.*, vol. 21, no. 1, pp. 23–30, Jan. 1997.

[29] J. Liu, Y. Q. Chen, J. M. Maisog, and G. Luta, "A new point containment test algorithm based on preprocessing and determining triangles," *Comput. Aided Des.*, vol. 42, no. 12, pp. 1143–1150, Dec. 2010.

[30] P. van Oosterom, "An R-tree based map-overlay algorithm," in *Proc. EGIS/MARI*, Paris, 1994, pp. 318–327.

[31] A. U. Frank, "Overlay processing in spatial information systems," in *Proc. Autocarto 8*, 1987, pp. 12–31.

[32] J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Commun. ACM*, vol. 25, no. 10, pp. 739–747, Oct. 1982.

[33] W. R. Franklin, V. Sivaswami, D. Sun, M. Kankanhalli, and C. Narayanaswami, "Calculating the area of overlaid polygons without constructing the overlay," *Cartogr. and Geogr. Inf. Syst.*, pp. 81–89, Apr. 1994.

[34] M. de Berg, H. Haverkort, S. Thite, and L. Toma, "I/o-efficient map overlay and point location in low-density subdivisions," in *Algorithms and Computation*, ser. Lecture Notes in Computer Science, T. Tokuyama, Ed. Berlin, Germany: Springer Berlin Heidelberg, Dec. 2007, vol. 4835, pp. 500–511.

[35] M. van Kreveld, "Digital elevation models: overview and selected TIN algorithms," in *Course Notes for the CISM Advanced School on Algorithmic foundations of Geographical Information Systems.* Dept. Computer Science Utrecht University, the Netherlands, Aug. 1996, http://www.cs.uu.nl/docs/vakken/gis/TINalg.pdf (Retrieved on 06/23/2015).

[36] W. R. Franklin, "Calculating map overlay polygon' areas without explicitly calculating the polygons — implementation," in *4th Int. Symp. Spatial Data Handling*, Zürich, 23-27 July 1990, pp. 151–160.

[37] W. R. Franklin, N. Chandrasekhar, M. Kankanhalli, V. Akman, and P. Y. Wu, "Efficient geometric operations for CAD," in *Geometric Modeling for Product Engineering*, M. J. Wozny, J. U. Turner, and K. Preiss, Eds. Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 485–498, selected

and expanded papers from the IFIP WG 5.2/NSF Working Conference on Geometric Modeling, Rensselaerville, USA, 18-22 September 1988.

[38] W. R. Franklin, M. Kankanhalli, C. Narayanaswami, and V. Akman, "Efficient intersection calculations in large databases," in *Int. Cartographic Association 14th World Conf.*, Budapest, Aug. 1989, pp. A–62 – A–63.

[39] W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, "Efficient primitive geometric operations on large databases," in *Proc. Nat. Conf. Challenge 1990s GIS Geogr. Inf. Syst.* Ottawa: Canadian Institute of Surveying and Mapping, Feb. 1989, pp. 1247–1256.

[40] W. R. Franklin, "Computer systems and low level data structures for GIS," in *GIS: Principles and Practice*, D. Maguire, D. Rhind, and M. Goodchild, Eds. London UK: Longman Higher Education and Reference, 1991, vol. 1, pp. 215–225.

[41] W. R. Franklin and V. Sivaswami, "OVERPROP — calculating areas of map overlay polygons without calculating the overlay," in *Second Nat. Conf. Geogr. Inform. Syst.*, Ottawa, Mar. 1990, pp. 1646–1654.

[42] J. W. van Roessel, "A new approach to plane-sweep overlay: Topological structuring and line-segment classification," *Cartogr. Geogr. Inf. Syst.*, vol. 18, pp. 49–67, 1991.

[43] U. Finke and K. Hinrichs, "A spatial data model and a topological sweep algorithm for map overlay," in *Advances in Spatial Databases*, ser. Lecture Notes in Computer Science, D. Abel and B. Chin Ooi, Eds. Berlin, Germany: Springer Berlin Heidelberg, Jun. 1993, vol. 692, pp. 162–177.

[44] H. P. Kriegel, T. Brinkhoff, and R. Schneider, *The combination of spatial access methods and computational geometry in geographic database systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, Aug. 1991, pp. 5–21.

[45] S. Audet, C. Albertsson, M. Murase, and A. Asahara, "Robust and efficient polygon overlay on parallel stream processors," in *Proc. 21st ACM*

SIGSPATIAL Int. Conf. Advances Geographic Information Systems, ser. SIGSPATIAL'13. New York, NY, USA: ACM, Nov. 2013, pp. 304–313.

[46] T. Brinkhoff, H. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *Proc. 1993 ACM SIGMOD Int. Conf. Management Data*, ser. SIGMOD '93. New York, NY, USA: ACM, 1993, pp. 237–246.

[47] H. Samet and R. E. Webber, "Storing a collection of polygons using quadtrees," *ACM Trans. Graph.*, vol. 4, no. 3, pp. 182–222, July 1985.

[48] F. W. Burton, V. J. Kollias, and J. G. Kollias, "A general pascal program for map overlay of quadtrees and related problems," *The Comput. J.*, vol. 30, no. 4, pp. 355–361, Jan. 1987.

[49] W. R. Franklin, "A linear time exact hidden surface algorithm," in *Tutorial: Computer Graphics: Image Synthesis*, K. I. Joy *et al.*, Eds., 1988, pp. 218–224.

[50] W. R. Franklin and V. Akman, "Adaptive grid for polyhedral visibility in object space, an implementation," *Comput. J.*, vol. 31, no. 1, pp. 56–60, Feb. 1988.

[51] W. R. Franklin, "A linear time exact hidden surface algorithm," *Comput. Graph.*, vol. 14, no. 3, pp. 117–123, 1980.

[52] M. Kankanhalli, "Techniques for parallel geometric computations," Ph.D. dissertation, Elect., Comput., and Syst. Eng. Dept., Rensselaer Polytech. Inst., Troy, NY, Oct. 1990.

[53] C. Narayanaswami, "Parallel processing for geometric applications," Ph.D. dissertation, Elect., Comput., and Syst. Eng. Dept., Rensselaer Polytech. Inst., Troy, NY, 1991.

[54] C. Narayanaswami and W. R. Franklin, "Determination of mass properties of polygonal CSG objects in parallel," in *Proc. Symp. Solid Modeling Found. CAD/CAM Applicat.*, J. Turner, Ed. ACM/SIGGRAPH, 5–7 June 1991, pp. 279–288.

[55] ——, "Determination of mass properties of polygonal CSG objects in parallel," *Internat. J. Comput. Geom. Appl.*, vol. 1, no. 4, pp. 381–403, 1991.

[56] W. R. Franklin and M. Kankanhalli, "Parallel object-space hidden surface removal," in *Proc. SIGGRAPH'90*, vol. 24, Aug. 1990, pp. 87–94.

[57] P. Y. F. Wu, "Polygon overlay in prolog," Ph.D. dissertation, Elect., Comput., and Syst. Eng. Dept., Rensselaer Polytech. Inst., Troy, NY, 1987.

[58] P. Y. Wu and W. R. Franklin, "A logic programming approach to cartographic map overlay," *J. Comput. Intelligence*, vol. 6, no. 2, pp. 61–70, May 1990.

[59] D. Pavić, M. Campen, and L. Kobbelt, "Hybrid booleans," *Comput. Graph. Forum*, vol. 29, no. 1, pp. 75–87, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2009.01545.x

[60] W. R. Franklin, "Efficient polyhedron intersection and union," in *Proc. Graphics Interface '82*, ser. GI '82. Toronto, Ontario, Canada: National Research Council of Canada, May 1982, pp. 73–80.

[61] G. Mei and J. C. Tipper, "Simple and robust boolean operations for triangulated surfaces," *The Comput. Res. Repos.*, vol. abs/1308.4434, 2013. [Online]. Available: http://arxiv.org/abs/1308.4434 (Retrieved on 10/19/2017).

[62] J. Yongbin, W. Liguan, B. Lin, and C. Jianhong, "Boolean operations on polygonal meshes using obb trees," in *Proc. Int. Conf. Environmental Science and Information Application Technology 2009*, vol. 1. IEEE, 2009, pp. 619–622.

[63] M. Douze, J.-S. Franco, and B. Raffin, "Quickcsg: Arbitrary and faster boolean combinations of n solids," Ph.D. dissertation, Inria-Research Centre, Grenoble–Rhône-Alpes, France, 2015.

[64] Q. Zhou, E. Grinspun, D. Zorin, and A. Jacobson, "Mesh arrangements for solid geometry," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 39:1–39:15, July 2016.

[65] P. Hachenberger, L. Kettner, and K. Mehlhorn, "Boolean operations on 3d selective nef complexes: Data structure, algorithms, optimized implementation and experiments," *Comupt. Geom.*, vol. 38, no. 1, pp. 64–99, Sept. 2007.

[66] The CGAL Project, CGAL*, Computational Geometry Algorithms Library*, 2015, http://www.cgal.org (Retrieved on 10/19/2017).

[67] Oslandia and IGN, *SFCGAL*, 2017, http://www.sfcgal.org/ (Retrieved on 10/19/2017).

[68] C. Leconte, H. Barki, and F. Dupont, "Exact and Efficient Booleans for Polyhedra," LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon, Tech. Rep. RR-LIRIS-2010-018, Oct. 2010. [Online]. Available: http://liris.cnrs.fr/publis/?id=4883 (Retrieved on 10/19/2017).

[69] G. Bernstein and D. Fussell, "Fast, exact, linear booleans," *Eurographics Symp. on Geom. Process.*, vol. 28, no. 5, pp. 1269–1278, 2009.

[70] A. Jacobson, D. Panozzo *et al.*, *libigl: A Simple C++ Geometry Processing Library*, 2016, http://libigl.github.io/libigl/ (Retrieved on 10/18/2017).

[71] C. K. Yap, "Symbolic treatment of geometric degeneracies," in *System Modelling and Optimization: Proc. 13th IFIP Conference*, M. Iri and K. Yajima, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 348–358.

[72] D. E. Knuth, *Surreal Numbers: How Two Ex-students Turned on to Pure Mathematics and Found Total Happiness: A Mathematical Novelette*. Reading, MA, USA: Addison-Wesley, 1974.

[73] W. R. Franklin, D. Sun, M.-C. Zhou, and P. Y. Wu, "Uniform grids: A technique for intersection detection on serial and parallel machines," in *Proc. Auto Carto 9: Ninth Int. Symp. Computer-Assisted Cartography*, Baltimore, Maryland, Apr. 1989, pp. 100–109.

[74] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, "Geometric computing and the uniform grid data technique," *Comput. Aided Des.*, vol. 21, no. 7, pp. 410–420, Sept. 1989.

[75] L. Cucu, M. Dragan, V. Negru, and D. Mangu, "Three dimensional Delaunay triangulation using an uniform grid," in *11th European Workshop Computational Geometry*. Linz, Austria: Universität Linz, 1995, pp. 21–23.

[76] W. R. Franklin, N. Chandrasekhar, M. Kankanhalli, M. Seshan, and V. Akman, "Efficiency of uniform grids for intersection detection on serial and parallel machines," in *New Trends in Computer Graphics (Proc. Computer Graphics Int.'88)*, N. Magnenat-Thalmann and D. Thalmann, Eds. Berlin, Germany: Springer-Verlag, 1988, pp. 288–297.

[77] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6th ed., 2014, http://gmplib.org/ (Retrieved on 10/19/2017).

[78] Intel. (2016) "Intel Turbo Boost technology 2.0". [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html (Retrieved on 10/19/2017).

[79] GRASS Development Team, *Geographic Resources Analysis Support System (GRASS GIS) Software*, Open Source Geospatial Foundation, 2012, http://grass.osgeo.org (Retrieved on 10/19/2017).

[80] S. Hopkins and R. G. Healey, "A parallel implementation of Franklin's uniform grid technique for line intersection detection on a large transputer array," in *4th Int. Symp. Spatial Data Handling*, K. Brassel and H. Kishimoto, Eds., Zürich, 23-27 July 1990, pp. 95–104.

[81] J. A. Baerentzen and H. Aanaes, "Signed distance computation using the angle weighted pseudonormal," *IEEE Trans. Vis. Comput. Graphics*, vol. 11, no. 3, pp. 243–253, May 2005.

[82] The Stanford 3D Scanning Repository. (2016) "The stanford 3D scanning repository". [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/ (Retrieved on 10/19/2017).

[83] GIT. (2016) "GIT large geometric model archive". [Online]. Available: http://www.cc.gatech.edu/projects/large_models/ (Retrieved on 10/19/2017).

[84] AIM@SHAPE-VISIONAIR. (2016) "AIM@SHAPE-VISIONAIR Shape Repository". [Online]. Available: http://visionair.ge.imati.cnr.it (Retrieved on 10/19/2017).

[85] R. J. Segura and F. R. Feito, "Algorithms to test ray-triangle intersection. comparative study," in *The 9-th Int. Conf. Central Europe Comput. Graph., Visualization Comput. Vision'2001, WSCG 2001*, 2001, pp. 76–81.

[86] X. Jiang and H. Bunke, "An optimal algorithm for extracting the regions of a plane graph," *Pattern Recognit. Lett.*, vol. 14, no. 7, pp. 553 – 558, July 1993.

[87] D. Eberly. (2002) "Triangulation by ear clipping". [Online]. Available: https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf (Retrieved on 11/29/2017).

[88] S. V. G. Magalhes, M. V. A. Andrade, W. R. Franklin, and W. Li, "Pinmesh - fast and exact 3d point location queries using a uniform grid," *Comput. & Graph.*, vol. 58, pp. 1 – 11, Aug. 2016.

[89] S. Ghemawat and P. Menage, *TCMalloc: Thread-caching Malloc*, 2017, http://goog-perftools.sourceforge.net/doc/tcmalloc.html (Retrieved on 10/19/2017).

[90] Q. Zhou and A. Jacobson, "Thingi10k: A dataset of 10,000 3d-printing models," *The Comput. Res. Repos.*, vol. abs/1605.04797, May 2016, http://arxiv.org/abs/1605.04797 (Retrieved on 10/19/2017).

[91] H. Barki, G. Guennebaud, and S. Foufou, "Exact, robust, and efficient regularized booleans on general 3d meshes," *Comput. & Mathematics with Applications*, vol. 70, no. 6, pp. 1235–1254, Sept. 2015.

[92] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities," *Int. J. for Numerical Methods in Eng.*, vol. 79, no. 11, pp. 1309–1331, May 2009.

[93] P. Cignoni, C. Rocchini, and R. Scopigno, "Metro: Measuring error on simplified surfaces," *Comput. Graph. Forum*, vol. 17, no. 2, pp. 167–174, June 1998. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.00236