Fast exact parallel map overlay using a two-level uniform grid

Salles V. G. Magalhães
Rensselaer Polytechnic Inst.
Troy, NY, USA
salles@ufv.br

Marcus V. A. Andrade
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
marcus@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Inst.
Troy, NY, USA
mail@wrfranklin.org

Wenli Li
Rensselaer Polytechnic Inst.
Troy, NY, USA
liw9@rpi.edu

ABSTRACT

We present EPUG-Overlay (*Exact Parallel Uniform Grid Overlay*), an algorithm to overlay two maps that is fast and parallel, has no roundoff errors, and is freely available. EPUG-Overlay combines several novel aspects. It represents coordinates with *rational numbers*, thereby ensuring exact computations with no roundoff errors and the ensuing sliver problems and topological impossibilities. For efficiency, EPUG-Overlay performs the map overlay in parallel, thereby utilizing the ubiquitous multicore architecture. Our application goes beyond merely using existing packages, which are inefficient when used in parallel on large problems. Indeed, overlaying two maps with 53,000,000 edges and 730,000 faces took only 322 elapsed seconds (plus 116 seconds for I/O) on a dual 8-core 3.1 GHz Intel Xeon E5-2687 workstation. In contrast, GRASS, executing sequentially and generating roundoff errors, takes 5300 seconds.

The overlay operation combines two input maps (planar graphs) containing faces (polygons) separated by polyline edges (chains), into a new map, each of whose faces is the intersection of one face from each input map. Floating point roundoff errors can cause an edge intersection to be missed or the computed intersection point be in a wrong face, leading to a topological inconsistency. Thus, a program might fail to compute a valid output map at all, using any amount of time. This gets worse when the inputs are bigger or have slivers. Heuristics can ameliorate this problem, but only to an extent.

By representing each coordinate as a vulgar fraction, with multiprecision numerator and denominator, the computation is exact. EPUG-Overlay also executes various useful subproblems very quickly, such as locating a set of points in a planar graph and finding all the intersections among a large set of small edges. EPUG-Overlay is built on our earlier sequential floating-point algorithm that found the areas of the overlay polygons, without finding the polygons themselves.

23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL) 2015 Seattle, WA, USA
 ISBN 978-1-4503-3132-6 ...$15.00

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms

Algorithms, Experimentation, Performance

Keywords

Map Overlay, Exact Computation, Parallel algorithm, Big Data

1. INTRODUCTION

A *map* is a partition of the E^2 plane into a finite number of *faces* or polygons. Let the faces of map $\mathcal{A}$ be called a_i. Excepting the exterior face, each face has finite extent. The *overlay* of two maps $\mathcal{A}$ and $\mathcal{B}$ is a new map $\mathcal{C}$. Each face of $\mathcal{C}$ is the intersection of one face of $\mathcal{A}$ with one face of $\mathcal{B}$. For example, let $\mathcal{A}$ be the map of coterminous states of the USA and $\mathcal{B}$ be a map of river watersheds. Face c_0 might be the part of the Hudson River watershed in New York State, and c_1 might be the part of the Connecticut River watershed in Vermont. Figure 1 shows a simple example.

Map overlay is a classical problem with many applications in Computational Geometry, Computational Cartography and Geographic Information Science (GIS) [9, 12, 15, 31, 38]. Sometimes, the special case of *triangulation overlay* is considered [37] (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 [13, 16].

A common algorithm is the plane sweep paradigm [11, 26, 31, 39] but, as stated by Audet et al. [3]: "the plane sweep strategy does not parallelize efficiently, rendering it incapable of benefiting from recent trends of multicore CPUs and general-purpose GPUs".

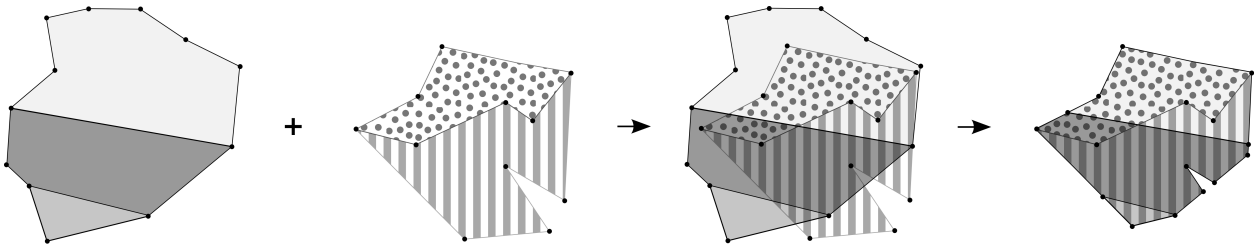


Figure 1: Map overlay example.

Other map overlay algorithms use a special data structure, such as an R-Tree [5,38], QuadTree [6,34], or Uniform Grid [16]. 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.

Tree-based data structures are suboptimal in various respects. They usually use pointers, which may require more space than the data. Chasing pointers slows down data retrieval. The code to walk down a tree is not straight line and so is not optimal when executed on a pipeline processor, which all modern processors are. Trees that branch out a factor of 4 at each level need too many levels to store hundreds of millions of edges. Trees are suboptimal for both data-level parallelism and instruction-level parallelism.

More complex trees can ameliorate some of these problems, at a cost of more code. However, we prefer the uniform grid as the simplest and most parallelizable. The uniform grid is usually overlooked because of the assumption that it works well only with uniformly distributed data, even though the converse has been shown, both theoretically and experimentally.

An important challenge is avoiding topological inconsistencies caused by floating-point arithmetic's roundoff errors. Heuristical solutions include using an ϵ tolerance or snap rounding. These work up to a point. However, as datasets become larger, the chance of these heuristics failing increases. Any chance of a failure prevents map overlay from being used as a guaranteed subroutine in a larger system.

In this paper, we prevent roundoff errors by using exact computation in the algebraic field of rational numbers with arbitrary precision [24, 28, 40]. Instead of using floating point numbers, we represent each coordinate as the ratio of two integers, each of which is stored with as many digits as necessary. An example 2-D point is $\left(\frac{123}{4567}, \frac{8901}{2345}\right)$. Since rational numbers are not a default data class in programming languages like C++, we use a library such as *GMP, the GNU Multi-Precision library* [18]. GMP overloads the operators $+, -, \times, \div$ so that mathematical expressions can be coded as usual. The cost is that computation is slower and variables take more space. However, this is quite tolerable.

A second problem with map overlay is the increasing volume of data to be processed. For example, NASA's *Earth Observatory System (EOS)* satellites collect about 1 TB of spatial data every day. This data is stored by the *Earth Observing Systems Data and Information System (EOSDIS)*, whose size is more than 3 petabytes [33]. More recent is the huge volume of data collected by GPS-enabled mobile devices. Therefore, map overlay can be thought of as a Big Data application, and optimizing execution time and space becomes critical.

As databases get larger, the probability of a *geometric degeneracy*, such as a vertex of one map incident on an edge on the other map, increases. Degeneracies increase the number of cases to handle for any predicate. The low level predicates must be resolved in a way that keeps higher level predicates consistent. We solve this with Simulation of Simplicity (SoS) [10].

Properly optimizing execution time and space requires exploiting the capabilities of current hardware. The first capability is the universality of multi-core processors. Single-core processors scarcely exist today; even smart phones have several cores. A good Intel Xeon processor with hyperthreading can simultaneously execute 16 threads on its 8 cores. Today, algorithms that are not parallelizable risk becoming quaint curiosities. Going further, a large fraction of computers have programmable GPUs, such as from Nvidia [32], or the *MIC - Many Integrated Core* by Intel [22]. Those provide massive parallelism with up to thousands of cores, e.g., 4992 CUDA cores for the best Nvidia GPU as of June 2015, the Tesla K80. The drawback is that each Nvidia CUDA core is perhaps only 5% as powerful as an Intel core, and the programming is much harder.

The second capability of current hardware is the massive amount of main memory. Workstations with a terabyte of main memory are becoming common. Even a high-end GPU, such as the K80, contains 24GB of memory. So, why artificially restrict an algorithm to use only a few GB of internal memory?

Nevertheless, these powerful capabilities come with the cost of a preference for simple regular data structures with few pointers. Data movement often takes more time than computation. The processor's execution pipeline has many steps, and anything unpredictable that causes it to flush, such as an unusual memory access, increases the time.

This paper combines the above two themes (prevent roundoff errors and exploit current hardware) as follows. It presents EPUG-Overlay (*Exact Parallel Uniform Grid Overlay*), a very fast, parallel map overlay algorithm using rational numbers. When processing the largest available datasets with millions of edges (the sequential version of) EPUG-Overlay uses 50% less elapsed time than GRASS does using floats. The source code of EPUG-Overlay is freely available for

nonprofit research and education; we welcome mashups with other software.

EPUG-Overlay is fast because of its use of implied global topology and its novel *2-level uniform grid*. It is implemented in C++, whose user-defined classes and overloadable operators make using rational numbers feasible. Parallelization on a shared memory multicore workstation is achieved with *OpenMP*. OpenMP is a set of *pragmas* added to a C++ program to mark how *for*-loops are to be parallelized, and what code blocks must be serialized. There is also an efficient parallel reduction facility, to total an addend from each thread. The hard part is designing the algorithm so that a *for*-loop's iterations are independent of each other, and that *writes* to global memory are minimized, since they must often be serialized.

The largest test case for EPUG-Overlay (using big rationals) overlaid two maps: one having 32 million edges and 220 thousands faces with another map having 21 million edges and 519 thousands faces in 438 elapsed seconds using 32 threads with dual 8-core 3.1 GHz Intel Xeon E5-2687 processors. That is a factor of 6 speedup compared to using one thread. By comparison, GRASS (using roundoff-error-prone floats), took 5300 seconds, partly because it is only single-threaded. (However, even when using only one thread, EPUG-Overlay is still faster than GRASS.) Geometric Performance Primitives (GPP), the commercial product described in [3] 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.

2. BACKGROUND

2.1 Map representation

This paper represents a map as a planar graph, with faces bounded by edges and vertices, each face labeled with an identification number. A face need not be a connected set. For example, a face representing Spain would include the exclave Llivia. Perhaps the most extreme example is Baarle-Nassau and Baarle-Hertog [36]. By convention, a map's exterior face is 0.

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. Each chain header is followed by the $n_e + 1$ coordinates of its vertices.

Since our algorithm does not need it, faces are not explicitly stored. That is, the topology is represented implicitly. Figure 2 presents an example of a map composed of 2 faces (one with two connected components) and the corresponding chains that are used to represent this map.

2.2 Two-level uniform grid

Franklin et al [17] proposed a uniform grid to accelerate map overlay. The basic idea is to superimpose a grid over the input maps. For each edge, the set of incident grid cells is determined. This implementation uses a C++ vector for each

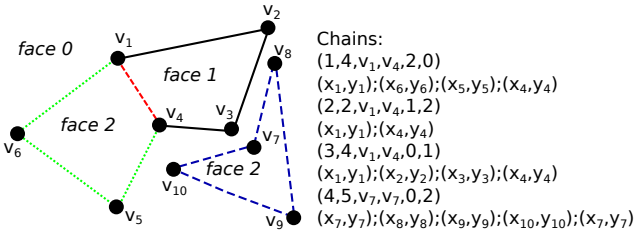


Figure 2: A map's faces and chains.

cell to record the edges incident on it. (Another implementation choice would be a ragged array.) Then, for each cell, the edges in that cell are compared pair-by-pair (one edge from each map) to find which pairs intersect.

If the edges are uniformly independently and identically distributed, then the number of edges in a cell is a random Poisson variable. The time to process a cell is the square of the number of edges. With the Poisson distribution, the mean of the square is equal to the square of the mean. If the grid size is chosen so that the number of edges per cell remains constant as the total number of edges grows, then the expected time to find all the intersections is linear in the number of edges plus the number of intersections, i.e., linear in the size of the input plus the output.

When overlaying two maps, we need only the intersections between an edge of one map with an edge of the other. So, if a cell contains many edges from one map but few (or none) from the other, the process is even more efficient. In contrast, the plane sweep edge intersection algorithm must process all the intersections within at least one map (the *red-blue edge intersection* problem), which is much slower.

The uniform grid works well even for uneven data for various reasons [2, 8, 14, 15]. First, the total time is the sum of one component (inserting edges into cells) that runs slower with a finer grid, plus another component (intersecting edges in cells) that runs 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 with over 50 pairs of edges. (The exact value is not important.)

Figure 3 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 2 pairs of edges that need to be checked) was created with 3×3 cells.

This nesting could be recursively repeated until all grid cells have less elements 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, as mentioned earlier, 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

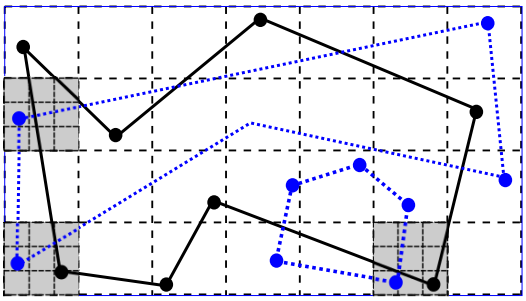


Figure 3: Two-level uniform grid with 4×7 first level and 3×3 second level (created in first level cells containing more than 2 edges).

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.

2.3 Roundoff error problems in map operations

Most computer programs represent non-integer numbers using floating point numbers with a fixed number of bits. In general, non-integer numbers can not be represented exactly, and the difference between the actual value of a non-integer number and its approximate floating point representation is referred to as *roundoff error* or *rounding error*. Worse, arithmetic operations generate further roundoff errors. Commonly, these errors are small, but in a long sequence of operations, the accumulated error can be large and can generate serious problems for the final result.

More formally, the floating point representation is an approximation to an algebraic field of real numbers, which is defined by 11 axioms, such as associativity of addition, $a+(b+c) = (a+b)+c$. Unfortunately, floating point numbers satisfy almost none of the real number axioms. Associativity of addition is not true when $a = 10^{30}, b = -10^{30}, c = 1$ because $-10^{30} + 1$ rounds to -10^{30}, so that $a + (b + c) = 0$ but $(a + b) + c = 1$. An excellent source for information about such problems of numerical computation, by one of the authors of the IEEE 754 floating point standard, is Wm Kahan's web site [23].

Roundoff errors have had serious consequences in many fields, such as: the failure of the first Ariane V rocket [1], the failure of the Patriot missile defense system resulting in 28 people killed [35], and spurious declining values in the Vancouver stock exchange index [29]. In geometry, roundoff errors can generate topological inconsistencies causing globally impossible results for predicates like point inside polygon and 3-point orientation.

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. Simply using double precision works for awhile. GMPXX [18] also has a multiple precision floating point class, albeit much slower than

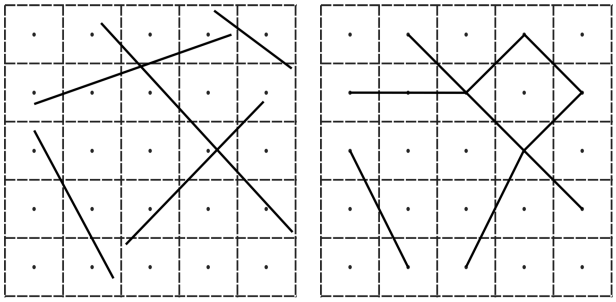


Figure 4: Snap rounding: (a) Initial set of segments. (b) Arrangement after the snap rounding execution: each segment endpoint and each intersection point has been snapped to the pixel center.

builtin double precision. Boissonat [4] describes a robust implementation of the plane sweep approach for intersecting segments using triple of the precision of the input data. Li [27] presents 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.

Controlled Perturbation (CP), [30] 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.

Another technique that also is based on finite precision approximation is snap rounding (SP) [20], whose basic idea is to use some rounding method to convert an arbitrary precision arrangement of segments into a fixed-precision representation. Snap rounding has been used in GIS packages such as [3, 7, 19], but it can generate some inconsistencies by changing the topology - for example, see Figure 4.

2.4 Exact computation using rational numbers

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 [21, 24, 28].

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 paper ameliorates the slowness of rational numbers with an algorithm that is efficient enough to compute with rationals in less time than the widely used GRASS routine can with floats.

3. THE ALGORITHM

EPUG-Overlay reads input two maps and if necessary the vertices coordinates are converted to rational 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 1 summarizes the overlay process; the next sections give details.

Algorithm 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 30 parallel threads, taking 2 seconds on each, finishing in 2 elapsed seconds? Elapsed time is not susceptible to that ambiguity. This also avoids technical difficulties with reliably measuring CPU times for parallel threads.

3.1 Exploiting global 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, one each from map $\mathcal{A}$ and map $\mathcal{B}$, 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's 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 Creating the two-level uniform grid

Choosing the size of a uniform grid, that is the number of cells on each side gives the user an opportunity to trade off speed and size. Section 4 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%. Therefore, we used a conservative empirical formula for the grid size that gave a good execution time and a feasible memory size.

Determining which grid cells contain each edge 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.

Next, for the few cells where more than, say, 50 pairs of edges will need to be tested, we add a 40×40 second level grid (the exact size is not critical) — more details are in Section 4.

This is also completely parallelizable.

3.3 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)* [10]. 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. Such a number system can be axiomatized and is consistent [25]. 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.

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.

3.4 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 rays will ever hit it.

The process is very fast with our grid. Assume that it 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 5, we have to follow the ray up through three cells until it hits edge $e = (u, w)$.

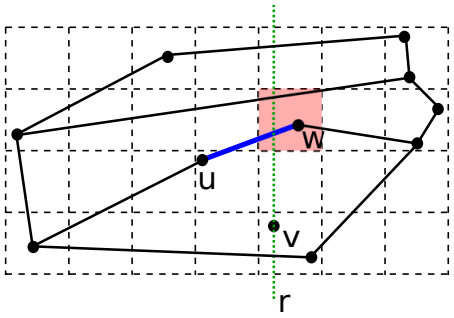


Figure 5: Determining the face containing a vertex.

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

3.5 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 face 0) 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 outside face (i.e. face 0), e will be an edge of the resulting map. Otherwise, if e is outside (i.e. inside the face 0), e will not be an edge of the resulting map. For example, in Figure 6, 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 outside face resulting from the intersection of faces $\mathcal{B}_1$ and $\mathcal{A}_0$ (the outside 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 $\mathcal{A}_0$).

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

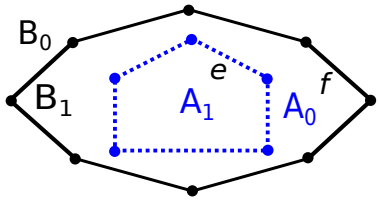


Figure 6: Two maps with no edge intersection.

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 7 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 outside 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 $\mathcal{B}$.

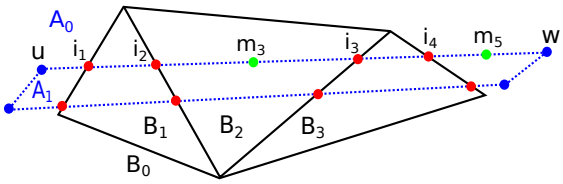


Figure 7: Example of an edge of map $\mathcal{A}$ intersecting more than one edge of map $\mathcal{B}$.

3.6 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. We use an atomic lock to serialize the edge insertion into the uniform grid. Other, more complicated, solutions are possible, such as having each thread accumulate results into a private array and then merging them. That is not necessary here since this step's cost is small enough.

The next step, edge intersection is parallelized over the grid cells as mentioned in section 3.3. We also parallelize locating the vertices of each map in the other map's faces. 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 parallelizable.

4. EXPERIMENTAL RESULTS

We implemented EPUG-Overlay in C++ and OpenMP using the multiple precision arithmetic package GMPXX [18]. It was compiled with g++ 4.8.2 and tested 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 Hyperthreading technology. 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, obtained from the ArcGIS, United States Census and National Atlas webpages:

- *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 1 gives statistics. After conversion to the representation described in section 2.1, we did these overlay tests:

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

4.1 Algorithm performance

First we tested the effect of different grid sizes; see Table 2. The threshold for refining a first level cell was 50 pairs of edges.

The performance depends on the grid size, mainly the first level size. We propose the following expression to define a conservative value for the first level grid size to balance the elapsed time and the memory usage:

$$d = \sqrt[4]{\frac{n_A \times n_B}{m}} \quad (1)$$

where n_A and n_B are respectively the number of edges in map $\mathcal{A}$ and $\mathcal{B}$, and m is the number of pairs of edges that is intended to be inside of each grid cell (in all tests, we selected $m = 50$). Thus, the grid size d is rounded to a simple multiple of a power of ten (for readability since the optimum is so broad). Based on other experiments, we always use a 40×40 second level.

Table 3 shows the quality of EPUG-Overlay's implementation with the design choices described above. The table gives the elapsed times for our three large test cases. The times are given for each of the program's seven stages, both when using one thread and when using 32 threads (on the 16 core hyperthreaded workstation).

Excluding the I/O time, the total parallel speedup on the largest case is a factor of 11. (It is less than 32 because operations like memory allocation and writing to a common global array are sequential, and because hyperthreading can run two simultaneous threads on one core only when they are not competing for a scarce resource.)

Even the sequential version of EPUG-Overlay is very competitive compared to other overlay programs. E.g., the GRASS GIS [19] 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 2291 seconds (including I/O) with a 5000×5000 1^{st}-level grid and 40×40 2^{nd}-level grid, or 3512 seconds (including I/O) using a more conservative 1^{st}-level grid size of 2000×2000 cells. 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 a simple data structure.

4.2 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.

Firstly, the main purpose for using a data structure in map overlay is to try to reduce the number of segment pairs that needs to be checked to verify if they intersect. For example, Figure 8 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 8(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 8(b), there are now about only 100 cells with more than 10,000 pairs of edges to be checked.

The next question is, why do not create a third level or do not use a Quadtree? This is a bad idea because our tests found that just creating a three-level uniform grid or 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. For example, Table 4 shows the time (in seconds) and the memory required just to create a three-level uniform grid and 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). In all cases, just the grid creation spent more time than the EPUG-Overlay processing time presented in Table 2.

5. CONCLUSION AND FUTURE WORK

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 competitive (more than 2 times faster) to the approximate overlay method included in the widely used GRASS GIS.

Furthermore, EPUG-Overlay is eminently parallelizable. On the largest test case, with OpenMP, we achieved a speedup

Table 1: Experiment datasets

(a) Input maps

Dataset	# Vertices	# Edges	# Faces	Size (GB)
BrSoil	258,961	251,011	5,567	0.03
BrCounty	342,738	326,193	2,959	0.04
UsAquifers	358,551	352,924	3,235	0.04
UsCounty	3,645,559	3,636,347	3,552	0.37
UsWaterBodies	21,652,410	21,060,354	219,831	2.25
UsBlockBoundaries	32,762,740	32,103,306	518,837	3.40

(b) Resulting maps.

Dataset	# Vertices	# Edges	# Faces	Size (GB)
BrSoil × *BrCounty*	581,554	608,912	19,400	0.07
UsAquifers × *UsCounty*	3,893,336	3,905,784	10,107	0.45
UsWaterBodies × *UsBlockBoundaries*	24,078,779	24,287,841	580,943	2.45

Table 2: Elapsed time in seconds (excluding I/O) and memory usage of EPUG-Overlay using different grid sizes for the first and second level. Columns *1* and *32* present the times using 1 and 32 threads respectively.

BrSoil × *BrCounty*				
Grid Size		Time (s)		Mem
1st	2nd	1	32	(GB)
	25^2	21	2	0.3
100^2	40^2	23	2	0.5
	55^2	24	2	0.8
	25^2	19	2	0.5
200^2	40^2	20	2	1.1
	55^2	24	2	1.9
	25^2	19	2	0.7
300^2	40^2	20	2	1.6
	55^2	25	3	2.7
	25^2	17	2	0.8
400^2	40^2	19	2	1.7
	55^2	24	3	2.8
	25^2	16	2	0.7
500^2	40^2	18	2	1.5
	55^2	22	2	2.4

UsAq. × *UsCounty*				
Grid Size		Time (s)		Mem
1st	2nd	1	32	(GB)
	25^2	184	20	0.8
100^2	40^2	172	19	0.8
	55^2	167	18	0.9
	25^2	119	14	0.9
200^2	40^2	117	14	1.1
	55^2	117	14	1.3
	25^2	102	13	0.9
300^2	40^2	100	12	1.2
	55^2	101	13	1.6
	25^2	88	12	1.0
400^2	40^2	88	12	1.4
	55^2	92	12	1.9
	25^2	82	11	1.1
500^2	40^2	84	11	1.6
	55^2	85	11	2.2

UsWBodies × *UsBBound.*				
Grid Size		Time (s)		Mem
1st	2nd	1	32	(GB)
	25^2	8260	640	9.0
1000^2	40^2	6183	491	9.6
	55^2	5298	443	10.1
	25^2	4060	357	10.0
2000^2	40^2	3398	322	11.6
	55^2	3154	307	13.5
	25^2	3045	307	11.2
3000^2	40^2	2735	282	14.1
	55^2	2620	276	17.5
	25^2	2520	284	12.6
4000^2	40^2	2424	273	16.5
	55^2	2282	268	21.7
	25^2	2255	282	14.0
5000^2	40^2	2127	273	19.4
	55^2	2156	275	25.9

Table 3: Elapsed time of EPUG-Overlay with the grid size from Eqn. (1) using 1 and 32 threads.

Maps:	*BrSoil* × *BrCounty*			*UsAq.* × *UsCounty*			*UsWBodies* × *UsBBound.*		
Grid size:	200×200			400×400			2000×2000		
	Time (sec.)		**Parallel**	**Time (sec.)**		**Parallel**	**Time (sec.)**		**Parallel**
Threads:	1	32	**speedup**	1	32	**speedup**	1	32	**speedup**
Read maps	1.0	1.0	1	5.3	5.5	1	73.1	74.5	1
Make grid	2.0	0.6	3	14.2	4.4	3	185.9	58.0	3
Refine 2-level grid	6.3	0.4	15	8.4	0.5	16	161.6	9.9	16
Intersect edges	1.0	0.1	8	2.6	0.3	8	505.5	30.9	16
Locate vertices	9.8	0.9	11	61.7	6.1	10	2434.7	211.6	12
Comp. output faces	0.5	0.1	4	0.9	0.2	5	110.4	11.8	9
Write output	1.0	0.6	2	4.5	4.6	1	40.4	41.6	1
Total w/o I/O	19.6	2.2	9	87.9	11.6	8	3398.1	322.2	11
Total with I/O	21.6	3.9	6	97.7	21.6	5	3511.6	438.3	8

Table 4: Elapsed time and memory size spent just to create a three-level uniform grid and a Quadtree using 32 threads.

	3-level grid				**Quadtree**	
Maps overlaid	**1st**	**2nd & 3rd**	**Time (sec.)**	**Size (GB)**	**Time (sec.)**	**Size (GB)**
BrSoil × *BrCounty*	200^2	40^2	54	1.1	70	1.7
UsAquifers × *UsCounty*	400^2	40^2	472	1.5	440	2.5
UsWBodies × *UsBBound.*	2000^2	40^2	290	43.7	8312	15.5

(excluding I/O), of a factor of 11 compared with the sequential implementation. And, we have ideas about how to make these times even better, if there were a need.

We expect these techniques to extend to efficient and robust parallel overlays of triangulations in E^3 in the Computer Aided Design and Computational Fluid Dynamics domains.

Acknowledgment

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

6. REFERENCES

[1] E. S. Agency. Ariane 501 inquiry board report. http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf (accessed on Jun-2015).

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

[3] S. Audet, C. Albertsson, M. Murase, and A. Asahara. Robust and efficient polygon overlay on parallel stream processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 304–313, New York, NY, USA, 2013. ACM.

[4] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comput.*, 29(5):1401–1421, 2000.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD Rec.*, 22(2):237–246, June 1993.

[6] F. W. Burton, V. J. Kollias, and J. G. Kollias. A general pascal program for map overlay of quadtrees and related problems. *Comput. J.*, 30(4):355–361, Aug. 1987.

[7] Cgal, Computational Geometry Algorithms Library. http://www.cgal.org (accessed on Jun-2015).

[8] L. Cucu, M. Dragan, V. Negru, and D. Mangu. Three dimensional Delaunay triangulation using an uniform grid. In *11th European Workshop Comput. Geom.*, pages 21–23. Universität Linz, 1995.

[9] M. de Berg, H. Haverkort, S. Thite, and L. Toma. I/o-efficient map overlay and point location in low-density subdivisions. In T. Tokuyama, editor, *Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 500–511. Springer Berlin Heidelberg, 2007.

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

[11] U. Finke and K. Hinrichs. A spatial data model and a topological sweep algorithm for map overlay. In D. Abel and B. Chin Ooi, editors, *Advances in Spatial Databases*, volume 692 of *Lecture Notes in Computer Science*, pages 162–177. Springer Berlin Heidelberg, 1993.

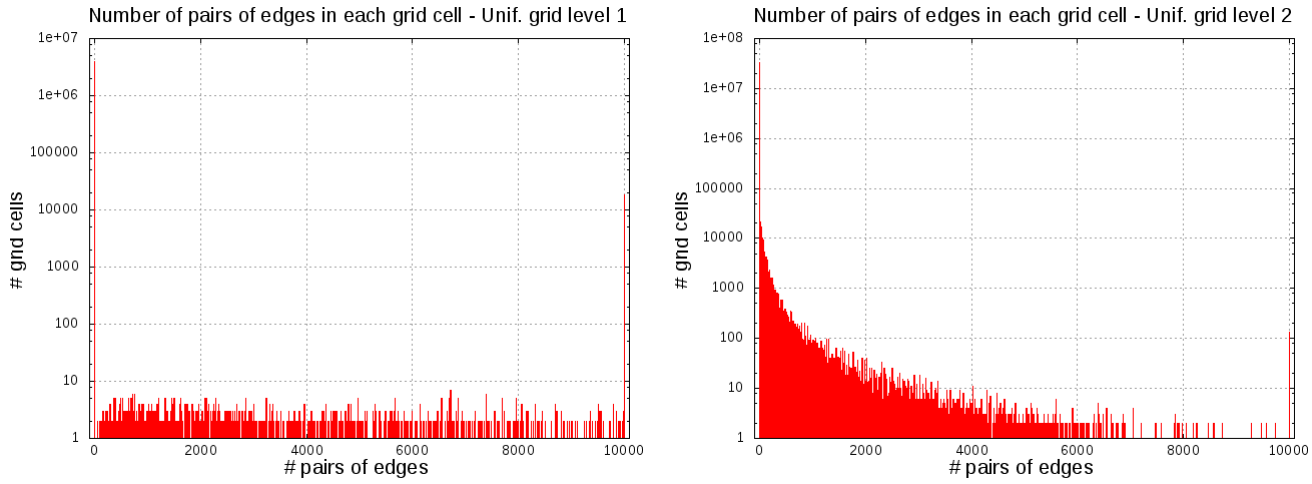
[12] A. U. Frank. Overlay processing in spatial information systems. In *Proceedings of Autocarto 8*, pages 12–31, 1987.

[13] W. R. Franklin. Calculating map overlay polygon' areas without explicitly calculating the polygons — implementation. In *4th International Symposium on Spatial Data Handling*, pages 151–160, Zürich, 23-27 July 1990.

[14] 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 N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, pages 288–297. Springer-Verlag, 1988.

[15] W. R. Franklin, V. Sivaswami, D. Sun, M. Kankanhalli, and C. Narayanaswami. Calculating the area of overlaid polygons without constructing the overlay. *Cartography and Geographic Information Systems*, 21(2):81–89, 1994.

[16] W. R. Franklin, V. Sivaswami, D. Sun, M. Kankanhalli, and C. Narayanaswami. Calculating the area of overlaid polygons



(a) (b)

Figure 8: The number of grid cells distribution considering the number of pairs of edges to be evaluated when overlaying maps *UsWaterBodies* with *UsBlockBoundaries*: (a) 1-level uniform grid; (b) 2-level uniform grid.

without constructing the overlay. *Cartography and Geographic Information Systems*, pages 81–89, Apr. 1994.

[17] 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 *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, pages 100–109, Baltimore, Maryland, April 1989.

[18] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.0.0 edition, 2014. http://gmplib.org/ (accessed on Jun-2015).

[19] GRASS Development Team. *Geographic Resources Analysis Support System (GRASS GIS) Software*. Open Source Geospatial Foundation, 2012. http://grass.osgeo.org (accessed on Jun-2015).

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

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

[22] Intel. Many integrated core architecture (MIC). http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html (accessed on Jun-2015).

[23] W. Kahan. `http://www.cs.berkeley.edu/~wkahan/`, 2014. online, retrieved 2015-06-22.

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

[25] D. E. Knuth. *Surreal Numbers: How Two Ex-students Turned on to Pure Mathematics and Found Total Happiness: A Mathematical Novelette*. Addison-Wesley, 1974.

[26] H. P. Kriegel, T. Brinkhoff, and R. Schneider. The combination of spatial access methods and computational geometry in geographic database systems. In *In Proc 2nd Symposium Spatial Database Systems*, pages 5–21. Springer, 1991.

[27] C. Li. *Exact geometric computation: theory and applications*,. PhD thesis, Department of Computer Science, Courant Institute - New York University, January 2001.

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

[29] B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999.

[30] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2006.

[31] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM*, 25(10):739–747, Oct. 1982.

[32] Nvidia. GPGPU - general-purpose computing on graphics processing unit. http://www.nvidia.com/object/what-is-gpu-computing.html (accessed on Jun–2015).

[33] T. O. D. project. http://data.nasa.gov/about, (accessed on Jun–2015).

[34] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. Graph.*, 4(3):182–222, July 1985.

[35] R. Skeel. Roundoff error and the patriot missile. *SIAM News*, 25, Jul. 1992.

[36] B. Smith. Baarle-Nassau / Baarle-Hertog. `http://ontology.buffalo.edu/smith/baarle.htm`, (retrieved 29 Jan 2010), 2008.

[37] 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*. Department of Computer Science Utrecht University, the Netherlands, aug 1996. `http://www.cs.uu.nl/docs/vakken/gis/TINalg.pdf`, online, retrieved 2015-06-23.

[38] P. van Oosterom. An R-tree based map-overlay algorithm. In *Proceedings of EGIS/MARI*, pages 318–327, Paris, 1994.

[39] J. W. van Roessel. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Systems*, 18:49–67, 1991.

[40] B. E. Weinrich and M. Schneider. Use of rational numbers in the design of robust geometric primitives for three-dimensional spatial database systems. In *Proceedings of the 13th Annual ACM International Workshop on Geographic Information Systems*, GIS '05, pages 163–172, New York, NY, USA, 2005. ACM.