Multidisciplinary Research on Geographical Information in Europe and Beyond
Proceedings of the AGILE'2012 International Conference on Geographic Information Science, Avignon, April, 24-27, 2012
ISBN: 978-90-816960-0-5
Editors: Jérôme Gensel, Didier Josselin and Danny Vandenbroucke

Automatically repairing invalid polygons with a constrained triangulation

Hugo Ledoux
Delft University of Technology
the Netherlands
h.ledoux@tudelft.nl

Ken Arroyo Ohori
Delft University of Technology
the Netherlands
g.a.k.arroyoohori@tudelft.nl

Martijn Meijers
Delft University of Technology
the Netherlands
b.m.meijers@tudelft.nl

Abstract

Although the validation of single polygons has received considerable attention, the automatic repair of invalid polygons has not. Automated repair methods can be considered as interpreting ambiguous or ill-defined polygons and giving a coherent and clearly defined output. At this moment, automatic tools are not satisfactory and repairing is thus mostly a semi-automatic task. We present in this paper a novel method, based on a constrained triangulation, to automatically repair invalid polygons. We describe our method, highlight some implementation details and describe an experiment with highly degenerate input for which predictable output is obtained. We believe our approach is superior to other tools since it is simple, intuitive and scales to big polygons.
Keywords: validation, automatic repair, constrained triangulation.

1 Introduction

While there are different definitions for a polygon, most GIS packages now use that of the Open Geospatial Consortium (OGC) and the International Organization for Standardization (ISO)[1] [11, 8], and provide validation functions to ensure that a given polygon conforms to the definition. There are small variations between different implementations [14], but we can consider the validation of a two-dimensional polygon a solved problem. Having one definition together with validation tools ensures that practitioners can now exchange datasets and use spatial analysis operations with their data (i.e. valid input is a prerequisite for most operations).

However, if a polygon is *invalid*—that is, it does not comply with the definition—then one has to repair it. Most validation tools give the user a list of errors and locations, but the user has to *manually* fix them (see for instance Figure 1). This is a very tedious and time-consuming task.

We investigate in this paper *automatic* methods for repairing polygons. Surprisingly, it is a topic that so far has received little attention. As we discuss in Section 3, most GIS packages perform some form of cleaning/repairing (e.g. deleting "unwanted parts" for display purposes) when reading invalid input, but how this is done is (often) not documented. An example of this cleaning, and of how the interpretation of the input can differ, is shown in Figure 2 for two packages.

To automatically repair polygons, practitioners often resort to *ad hoc* solutions and tricks, the most known being the "buffer-by-0" operation. A buffered geometry is built, but the original lines are offset by nothing (zero). To construct a buffer, the planar graph of the input is built; in other words the topology is built, which will be structurally identical to the original input. While this trick works fine for solving a few problems, parts of a polygon can disappear for some inputs (see Ramsey [12] for

[1] These are almost identical, see Section 2.

Figure 1: JTS interface that helps users to locate and manually fix errors in invalid polygons.

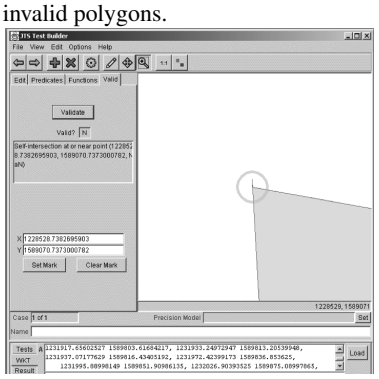
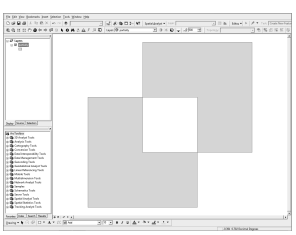
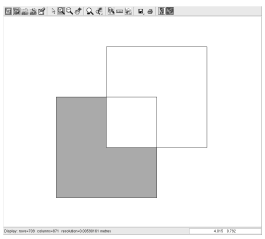


Figure 2: Different interpretations of the polygon p_3, as shown in Figure 3. (a) ArcGIS considers the overlapping region as a hole, but the non-overlapping part of the hole as a new polygon (QGIS and FME do this as well). (b) GRASS removes the overlapping part from the polygon, becoming a new polygon with a different shape.



(a)



(b)

some examples), and perhaps worse, its use is almost not documented, making the output unpredictable. To our knowledge, the only automatic repair tool is `ST_MakeValid`, which will be available in the next version of PostGIS (version 2.0, which is already available as a beta version at the time of writing). As explained in Section 3, the function is not documented (one has to read the code and try with different input) and is based on a series of tricks: buffer-by-0 is used for some inputs, for others another function is used, a given configuration triggers the use of another function, etc.

We present in this paper a novel method to automatically repair invalid polygons. As described in Section 4, it is conceptually simple and is based on the properties of a constrained triangulation of the input polygon. It permits us to formalise and document how we perform the repair (and permits a practitioner to predict the behaviour of our implementation). It should be said that repairing is not an exact science (different persons could repair an invalid polygon in different way), but we clearly define what situations we handle and how we fix them. We have implemented our algorithm (only about 300 lines of code, without the code to triangulate) and we report in Section 5 on some experiments we ran with problematic polygons and some real-world complex polygons. We also compare our implementation to that of `ST_MakeValid`; we show that our approach scales better to big polygons, and that it outperforms `ST_MakeValid` by several orders of magnitude. Finally, in Section 6, we elaborate on the advantages of our method and discuss future work.

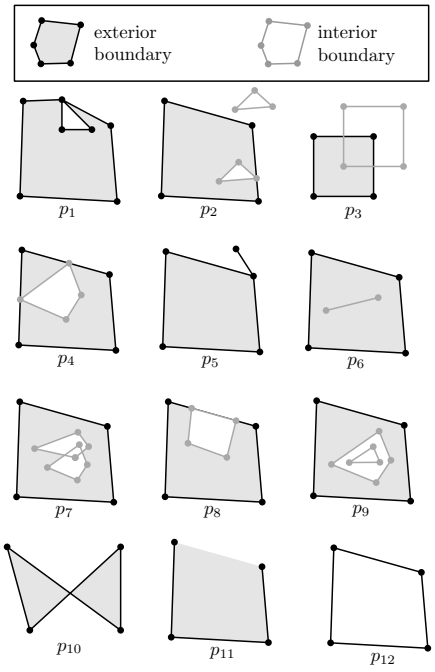
2 What is a polygon?

For the validation and repair of a polygon, we follow the definition of the Simple Feature specifications (SFS): "a planar Surface defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon." [11]. Different rules are provided, the most relevant being the following (examples of invalid polygons are given between brackets, they refer to those in Figure 3):

1. the rings that define the exterior and inner boundaries should be simple and closed (p_{10} and p_{11}). Notice that this prevents the existence of rings with zero-area (p_6). The polygon p_1 is thus not allowed by the SFS (the triangle should be represented with an interior boundary), but ESRI's Shapefile allows it.
2. the rings should not cross (p_3, p_7 and p_8) but may intersect at one tangent point (interior boundary of p_2 is a valid case, although p_2 as a whole is not since the other interior boundary is located outside the interior one).
3. a polygon may not have cut lines, spikes or punctures (p_5 or p_6); removing these is known as the regularisation of a polygon.
4. the interior of every polygon is a connected point set (p_4).
5. each inner ring creates a new area that is disconnected from the universe. Thus, they cannot be outside the outer ring (p_2) or inside other inner rings (p_9).

Furthermore, since this definition does not enforce an orientation for the rings (besides that the exterior and interior boundaries should have opposite orientations), we follow the ISO rules which state that the exterior boundary of a polygon is counterclockwise, and the inner boundaries clockwise.

Figure 3: Several invalid polygons (this is not a complete list of all problematic polygons, but rather an overview of common cases).



3 Related work

Ramsey [12] gives an excellent overview of the tricks used by practitioners to automatically repair their polygons (his examples are PostGIS-related only, but since it uses other open-source libraries such as GEOS we believe this is representative of what practitioners do). It can be seen that different functions of GEOS and PostGIS have to be used to fix different problems, as some functions fail in certain conditions. For instance, buffer-by-0 works for polygon p_1 in Figure 3, but removes half of the bow-tie of p_{10}—repairing correctly p_{10} (i.e. with two output polygons) requires using three functions[2]. All these functions are based on the construction of a planar graph of the input, and on identifying loops in this graph to form rings. Some of them reconstruct all the possible loops, while others stop after one loop has been found.

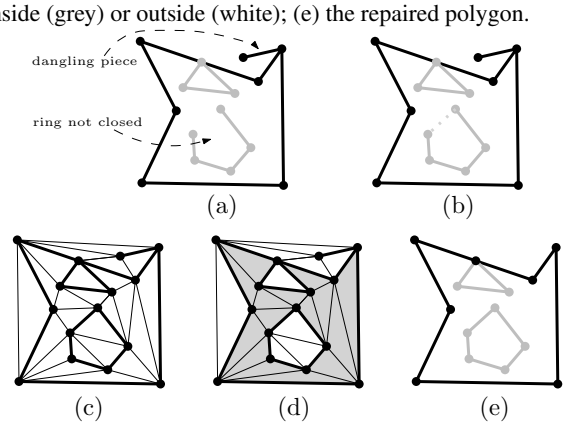
The script `cleanGeometry.sql`[3] was the first attempt to formalise the decision tree based on a given input. Unfortunately, polygons with interior rings are not properly handled.

`ST_MakeValid` is an attempt to build a high-level function in PostGIS to repair any input polygon. It uses the functions of GEOS and PostGIS, and depending on the topological and geometrical configuration of the input rings, different functions are used to repair. Basically, a planar graph of the input is built first,

[2] `ST_ExteriorRing` + `ST_Union` + `ST_BuildArea`

[3] Available at: `trac.osgeo.org/postgis/wiki/UsersWikiCleanPolygons`

Figure 4: Workflow of our approach to repair a polygon. In (a) the input polygon has 2 problems; (b) the interior ring is closed; (c) the CT is constructed; (d) triangles are labelled as inside (grey) or outside (white); (e) the repaired polygon.



and then one loop in the graph is found and a ring is built (at this point it is unknown if it is an exterior or an interior ring). Then, for all the other loops in the graph, the resulting polygon is obtained by the symmetric difference of this ring and the one already found. Each symmetric difference requires building a new independent graph where the topological relations of the rings are extracted (to detect which ring is the exterior and which are the interior). As a consequence, `ST_MakeValid` is highly inefficient for input containing a large number of interior rings, as Section 5 shows. It should be noticed that the function attempts to create a valid representation of a given invalid geometry without losing any of the input vertices, i.e. if a ring collapses to a line segment, this line segment is also returned to the user.

4 Repairing a polygon with a constrained triangulation

Our approach to repairing polygons uses a constrained triangulation (CT) as a supporting structure, and as Figure 4 illustrates, it has three steps:

1. construction of the CT of the input polygon, closing rings if necessary. Notice that at this step new vertices are added at the intersection of line segments.
2. labelling of the triangles (*outside* or *inside*) using an "area-filling" method which starts from the outside, and is performed by using simple graph-based algorithms.
3. reconstruction of the polygon(s) by a depth-first counter-clockwise search of their boundaries.

The CT permits us to embed together both the geometry and the topology of the input polygons in the same structure, which allows us to perform less operations when repairing (for instance `ST_MakeValid` needs to perform extra operations to detect topological relationships between rings, while with a CT this is not necessary). Another advantage is that we can exploit the properties of the CT to perform some cleaning that is otherwise rather cumbersome (e.g. duplicate vertices, collapsing of rings to points or lines). Furthermore, implementation-wise, several stable and fast constrained triangulation libraries exist (including CGAL [2], Triangle [13] and GTS [6]) and we can build over them.

Notice that the valid representation of an invalid output can be either:

- nothing (e.g. the only ring of a polygon is a line segment);
- one polygon (potentially with interior boundaries);
- several polygons (polygons p_2, p_4, p_9 and p_{10} in Figure 3).

4.1 Constrained triangulations

Given a set of points S and (straight-line) segments in the plane (such as in Figure 4b), a constrained triangulation (CT) decomposes the convex hull of S into triangles that are non-overlapping, and every input segment appears as an edge of CT(S). If S contains segments forming a loop (which defines one ring of a polygon in our case), it permits us to triangulate the interior of this loop (i.e. a triangulation of the polygon). Notice here that for the sake of repairing a polygon, we cannot use algorithms to triangulate a single polygon (e.g. Chazelle [3]) as these often assume that the input is simple and forms only one polygon.

In our approach, the triangulation is performed by constructing a CT of all the segments representing the boundaries (outer + inner) of a polygon. Thus, repairing a polygon starts at this step, as the properties of the CT are strict. If two input segments intersect, they are split into sub-segments and thus a new point can be added at their intersection. Also, no two vertices or edges of a CT can be at the same location, which means that if two identical segments are in the input, only one will be kept in the CT. At this step, we also perform another validation operation: we close rings whose first and last points are not the same (Figure 4b). We believe that this is consequent with the intention of the user: if a new ring was defined it is probably a mistake that it is not closed.

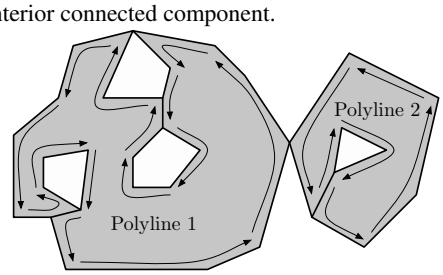
Observe that while the shape of the triangles constructed is important for many applications [13], here it is not relevant and any CT can be used (the constrained *Delaunay* triangulation can be used but is not necessary). A CT can be built efficiently with a variety of approaches [7, 4]. Once the CT is stored, it can be used for solving quickly, and with an extremely light auxiliary data structure, the point-location problem [10], which is useful to identify double vertices and intersections of segments.

4.2 Labelling triangles: outside or inside

To label each triangle as either outside or inside, we start at one triangle located *outside* any input ring, we label it as outside and we expand to all triangles reachable from it without passing through a constrained edge of the CT. When these are exhausted, all remaining triangles reachable by passing once through a constrained edge are known to be in its interior. From the remaining triangles, those that can be reached by passing through two constrained edges are in its exterior, and so on.

The fact that we start from the outside is key to ensuring that we always repair a given polygon in the same way, and it also permits us to predict how a polygon will be repaired. To find a triangle located outside any ring, we exploit the "far-away point" (also called the "big triangle") that is used by several CT implementations [9, 5]. In brief, every edge on the border of the con-

Figure 5: The polyline generated from a seeding triangle in the interior of the ring joins all holes with the external boundary, while always keeping the interior connected and on the same side of the line (left). A separate polyline is generated for each different interior connected component.



vex hull has a triangle incident to it, and this triangle is formed by the edge and a special "infinite" point.

Thus, to label the triangles, the algorithm performs several passes. First the triangles incident to the infinite point and the reachable ones are labelled as outside. Then this operation is expanded to triangles further in the interior of the polygon (labelling them as inside). If all the triangles have been flagged (if there are no interior rings) then the process if finished, otherwise the labelling continues the same way, alternating between outside and inside, until all triangles have been labelled.

4.3 From the CT to a polygon

To reconstruct the polygon from the labelled CT, we construct a path (a polyline) that runs along the boundary segments of the polygon, on the inside of it. In a nutshell, we traverse one area formed by several triangles labelled as interior (or exterior) in a depth-first search order, always going counter-clockwise. In this process, so-called "bridges" are generated. Here the generated polyline goes over an unconstrained triangulation edge twice, once on the left side, once on the right side. These bridges connect the interior rings to each other or to the outer ring. As these bridges obviously do not belong to the geometry of the input polygon, they are discarded when the individual rings of the polygon are obtained. Figure 5 shows an example.

4.4 Example of repaired polygons

Figure 6 shows examples of invalid polygons that were repaired with the method described in this section. The following should be noticed:

Level of repair Figure 6 shows all the polygons that can be constructed. Polygon p_7 is repaired with 4 polygons, since there are 3 levels and intersections between the two interior rings. It is possible to modify the algorithm so that only the first level (exterior ring + 1 level of interior rings) are returned.

Dangling pieces These are ignored because the labels on the left and the right are the same.

Disconnected interior This is handled properly and one new polygon is created per interior-connected part.

Collapsed area These areas are simply ignored in the output (same labels on left and right). However, if they intersected another boundary, then the point(s) added during the construction of the CT is present in the output. It is possible to post-process segments and merge two consecutive collinear ones, but we have not implemented it.

Overlapping boundaries Such boundaries are merged / dissolved together.

Self-intersections Self-intersections, such as p_1 in Figure 3, are repaired as an interior boundary is constructed.

5 Experiments and comparison with other tools

We have implemented the algorithm described in Section 4 in C++; it is open-source and freely available[4]. Two libraries are used: (1) CGAL[5] (we use its constrained triangulation module and its robust geometric operations); (2) the OGR Simple Features Library[6] (which allows read and write from a large variety of GIS data formats). In its current form, the prototype reads one `Polygon` represented with *well-known text* (WKT) and returns one *valid* `MultiPolygon` also represented as a WKT.

We have tested our prototype (which we named `prepair`) with all the polygons shown in Figure 3 and with other similar ones. They are meant as a sort of *unit testing* polygons to compare how they fare in different tools [1]. We are able to repair all of these, with the behaviour explained in Section 4.

To test the efficiency of our method, we have tested it with three invalid real-world polygons from the CORINE land cover dataset (CLC2006), these are shown in Figure 7. These are land use polygons that can become rather complex and big. The errors in all these polygons are self-intersection of the exterior boundary (because the *shapefile* standard does not follow the SFS); one example is shown in Figure 8(c).

To compare our approach and implementation, we have tested `ST_MakeValid` with the same polygons. It repairs the same way as `prepair` and for the unit tests the performances are similar; these contain very few points and that was expected. However, for polygons with more points, `ST_MakeValid` is far less efficient. Indeed, as Table 1 shows, as the number of points grows `ST_MakeValid` becomes far less efficient than `prepair` since several operations have to be repeated for all the pairs of rings for instance.

By comparison, with `prepair`, one global operation needs to be done (construction of the CT + labelling), but afterwards no operations with quadratic behaviour need to be performed. The algorithm's running time is defined by the computational complexity of creating the CT, $O(v \log v)$[7], and of reconstructing the polygon, $O(vr \log r)$, with v the number of vertices and r the number of rings. The other operations (i.e. tagging, creation of the polylines, and splitting) are performed in linear time. Therefore, the total running time is $O(v \log v + vr \log r)$. If v is exponentially larger than r, the algorithm is dominated by the triangulation time, $O(v \log v)$.

[4] `prepair.googlecode.com`
[5] `www.cgal.org`
[6] `www.gdal.org/ogr`
[7] Constructing the CT could conceivably take $O(v^2)$ time, since a quadratic number of edge-edge intersections are possible (e.g. in certain star polygons). However, in practice the number of intersections is much smaller than v.

Figure 6: Some polygons from Figure 3 and how our method repairs them. Polygon p_{13} was added.

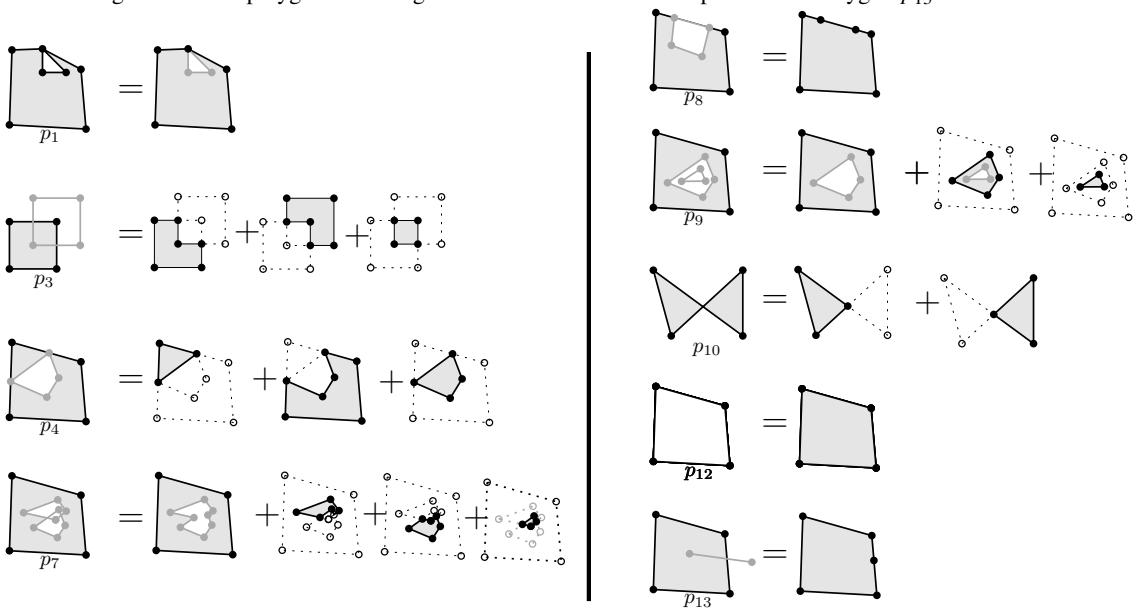
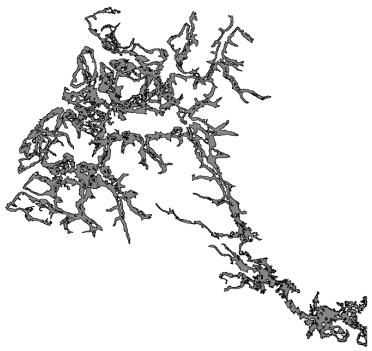
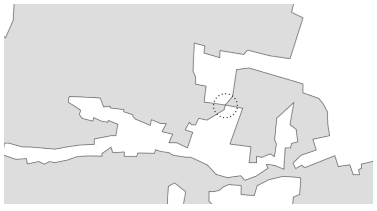


Figure 7: Three polygons from CLC2006: **(a)** EU-47552 (light gray) and EU-47997 (dark gray). **(b)** EU-180927. Their characteristics are shown in Table 1. **(c)** Self-intersection of the exterior boundary of the polygon EU-47552.

(a)



(b)



(c)

Table 1: For the three CLC2006 polygons in Figure 7: total number of points and of rings, and the running times to repair them.

	points	rings	`prepair`	`ST_MakeValid`
EU-47552	2 412	10	0.5s	0.8s
EU-47997	32 473	346	11.4s	314.0s
EU-180927	102 272	299	52.2s	740.2s

6 Discussion

While designing our method we had to make several—often arbitrary—choices for its behaviour. While the way polygons are repaired is perhaps not always consistent with what one might do manually, our method is formalised and permits users to predict easily how their polygons will be repaired. We plan to investigate alternative paradigms to repair, for instance by following a point-set topology approach. That is, where a polygon p is defined as $p = outerring \setminus \{innerrings\}$ (as in Figure 2b). That would be relatively simple to implement with the labelling of a triangulation.

It should be noticed that our method also improves the robustness of polygons even if they are valid since we automatically add a node to two touching rings. This is not mandatory according to the ISO/OGC definitions, but a wished property for representing polygons and manipulating them [14]. Furthermore, we can claim that our implementation is fully robust since we rely on CGAL (which uses robust arithmetic) and our repair operations are expressed solely in terms of labelling of triangles (no geometric computations are involved).

Multidisciplinary Research on Geographical Information in Europe and Beyond
Proceedings of the AGILE'2012 International Conference on Geographic Information Science, Avignon, April, 24-27, 2012
ISBN: 978-90-816960-0-5
Editors: Jérôme Gensel, Didier Josselin and Danny Vandenbroucke

References

[1] Tim Burns. Effective unit testing. *ACM Ubiquity*, January 2001, 2001.

[2] CGAL. *CGAL 3.8 User and Reference Manual*. CGAL Editorial Board, 2011.

[3] Bernard Chazelle. A theorem on polygon cutting with applications. In *Proceedings 23rd Annual Symposium on Foundations of Computer Science*, pages 339–349, Washington, DC, USA, 1982. IEEE Computer Society.

[4] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. In Alain Finkel and Matthias Jantzen, editors, *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 461–474. Springer Berlin / Heidelberg, 1992.

[5] Michael A. Facello. Implementation of a randomized algorithm for Delaunay and regular triangulations in three dimensions. *Computer Aided Geometric Design*, 12:349–370, 1995.

[6] GTS. *GTS Library Reference Manual*, 2006. URL `http://gts.sourceforge.net/reference/book1.html`.

[7] Leonidas J. Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2): 74–123, 1985.

[8] ISO(TC211). ISO 19107:2003: Geographic information—Spatial schema. International Organization for Standardization, 2003.

[9] Yuanxin Liu and Jack Snoeyink. The "far away point" for Delaunay diagram computation in $\mathbb{E}^d$. In *Proceedings 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, pages 236–243, Seoul, Korea, 2005.

[10] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry—Theory and Applications*, 12:63–83, 1999.

[11] OGC. OpenGIS implementation specification for geographic information—simple feature access. Open Geospatial Consortium inc., 2006. Document 06-103r3.

[12] Paul Ramsey. PostGIS: Tips for power users. Presentation at the FOSS4G 2010 Conference, Barcelona, Spain, 2010. `http://s3.opengeo.org/postgis-power.pdf`.

[13] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, USA, 1997.

[14] Peter van Oosterom, Wilko Quak, and Theo Tijssen. About invalid, valid and clean polygons. In Peter F. Fisher, editor, *Developments in Spatial Data Handling—11th International Symposium on Spatial Data Handling*, pages 1–16. Springer, 2004.