

MSc thesis in Geomatics

Validation and automatic repair of planar partitions using a constrained triangulation

Ken Arroyo Ochori

August 2010

ISBN: 978-94-6186-034-7

On cover:

INEGI land cover data for Central Mexico. 1:1,000,000 scale.
Every shade represents a different land cover type.

VALIDATION AND AUTOMATIC REPAIR OF PLANAR PARTITIONS USING A CONSTRAINED TRIANGULATION

A thesis submitted to the Delft University of Technology in partial fulfilment
of the requirements for the degree of

Master of Science in Geomatics

by

Gustavo Adolfo Ken Arroyo Ohori

August 2010

Supervisors: Prof. dr. Peter van Oosterom
Dr. Hugo Ledoux
Martijn Meijers
Co-reader: Dr. Ben Gorte



Abstract

Planar partitions (subdivisions of the plane into polygonal areas) constitute one of the most important data representations in GIS. They are used to model concepts as varied as land use, administrative units, natural features and cadastral parcels, among many others.

However, since polygons are often stored separately, different errors and inconsistencies are introduced during their creation, manipulation (both manual and automatic) and exchange. These come in the form of invalid polygons, gaps, overlaps and disconnected polygons, which severely hampers their use in other software. Existing approaches to solve this problem usually involve polygon repair using a list of constraints, and complex planar partition repair operations performed on a planar graph. However, these have many shortcomings in terms of complexity, numerical robustness and difficulty of implementation. Moreover, they leave many invalid cases untouched.

To solve this problem, a novel method to validate and automatically repair planar partitions has been developed. It uses a constrained triangulation of the polygons as a base, which being by definition a planar partition, means that only relatively simple operations are needed to ensure that the output becomes valid. Point locations are maintained throughout the process, while fully automatic repair is possible using customisable criteria. This approach is also extensible to individual polygons, is capable of handling a larger variety of cases and has good performance compared to existing alternatives; all of this with numerical robustness and maintaining topological consistency throughout.

In order to analyse, test and improve the developed algorithms, and encourage further development, a fast and efficient implementation has been written in C++, which has been tested with several large data sets and compared with other available software, regarding both performance and functionality. This prototype is able to successfully repair planar partitions of more than 100,000 polygons. It is also open source and freely available on the GDMC website (<http://www.gdmc.nl/>).



Acknowledgements

I would like to express my utmost gratitude to the people who helped me complete this thesis, either directly or indirectly. First of all, I offer thanks to my supervisors Peter van Oosterom, Hugo Ledoux and Martijn Meijers who were very much involved in this work and provided their guidance, together with plenty of their time and effort.

Additionally, I would like to thank the other people from the Department of GIS Technology for providing me with helpful comments and support whenever I needed them, especially to Theo Tijssen for the tech support and information about Radius Topology, and Elfriede Fendel for her assistance with all the graduation procedures.

I also wish to thank the people involved in the Geomatics programme: my office mate Filip Biljecki for the good company, Dinesh Kalpoe for the extensive thesis discussions, and all my fellow students and the faculty members, particularly to Ben Gorte for agreeing to read and evaluate this thesis. I have learned a great deal in the past two years, but it has also been a very enjoyable time for me.

Last, but not least, I thank my mother, Kimiyo Ohori Mase, for her always untiring support, and my friends in Mexico, for striving to keep in touch despite the long distance.

Thank you all.



Contents

1	Introduction	1
1.1	Motivation	2
1.2	Objectives	6
1.3	Scope	8
1.4	Structure Overview	9
2	Background Information and Theory	11
2.1	Computer Representation of Geometric Features	12
2.2	Robustness in Geometric Operations	15
2.3	Valid Polygons	16
2.4	The Conflation Problem	18
2.5	Planar Partitions	21
2.6	The Constrained Delaunay Triangulation	22
3	State of the Art in Planar Partition Validation and Repair	27
3.1	Validation and Repair of Individual Polygons	28
3.2	Topological Validation of Planar Partitions	31
3.3	Planar Partition Repair Using Vertex, Edge and Face Snapping and Splitting	33
3.4	Planar Partition Repair Using Topological Information	39
3.5	Topological Planar Partition Repair Using A Database	42
4	Using a Constrained Triangulation for Planar Partition Validation and Repair	45
4.1	Repairing a Single Polygon	47
4.2	Triangulation and Tagging of a Planar Partition	50
4.3	Schemata Matching	52

4.4	Repair Operations	55
4.5	Extraction of Polygons From a Triangulation	57
5	Implementation, Experiments and Discussion	63
5.1	The Developed Prototype	63
5.2	Invalid Polygons Repair Comparison	64
5.3	Large Planar Partition Repair Comparison	70
6	Conclusions, Recommendations and Future Work	73
6.1	Conclusions	74
6.2	Main Contributions	75
6.3	Recommendations and Future Work	75
A	Design Considerations and Implementation Checks	77
A.1	Data structures designed	77
A.2	Triangulation data structures	77
A.3	Deciding whether to triangulate and reconstruct a ring	80
A.4	Removing duplicate (constrained) edges	81
A.5	Algorithm to recursively tag exterior and interior of a ring	82
A.6	Algorithm to generate a correctly oriented polyline with all boundaries of a polygon	83
A.7	Algorithm to create valid rings from a polyline with all boundaries of a polygon	88
A.8	Finding the nesting of inner and outer boundaries	90
A.9	Schemata Matching Criteria	92
B	Completeness and Correctness of Planar Partitions Repair	95
B.1	Points	95
B.2	Edges	95
B.3	Rings	95
B.4	Polygons	96
C	Data Sets Used	97
C.1	Unit Test Polygons	97
C.2	Large Data Sets	103
	Bibliography	105



List of Figures

1.1	Old analog and digital choropleth maps.	3
1.2	Examples of the use of planar partitions in GIS.	4
1.3	A selection of the largest polygon in the CORINE 2000 data set.	6
1.4	Different errors present in the CORINE 2000 data set.	7
2.1	A polygon.	13
2.2	The spaghetti data structure.	13
2.3	The Winged Edge data structure and the spatial relationships associated with it.	14
2.4	Intersections in a floating point grid have a low likelihood of having an exact representable value.	15
2.5	The results of the orientation predicate in a very small area.	17
2.6	The bow-tie polygon, modelled according to different criteria	18
2.7	The ESRI Shapefile and ISO standards differ in the modelling of polygons with holes.	19
2.8	A region of the polygons for the Arribes del Duero Natural Park in Spain and the International Douro Natural Park in Portugal.	20
2.9	Two contiguous tiles from the CORINE 2000 data set, separated by a small distance.	21
2.10	Two types of triangulations.	23
2.11	A point set triangulated in two different ways.	23
2.12	A polygon triangulated in two different ways.	24
2.13	A region of the triangulation of the polygons in CORINE tiles E39N32 and E40N32.	25
3.1	Zero area polygon in ArcGIS.	31
3.2	Overlapping polygons in GRASS	33

LIST OF FIGURES

3.3	Defining a threshold for vertex, edge and face snapping.	34
3.4	Spikes and punctures created by snapping.	35
3.5	Region split by snapping.	35
3.6	Four adjacent tiles from CORINE joined and repaired in FME	36
3.7	Joining and repairing four adjacent tiles from CORINE in FME	37
3.8	Snapping to the closest line can cause topologically invalid configurations. .	39
3.9	Entering topology rules in ArcGIS.	40
3.10	Overshoots and undershoots.	41
3.11	Topology errors found in ArcGIS.	41
3.12	Planar partition repair in ArcGIS.	42
3.13	Topology errors found in a small region from CORINE tile E41N27.	43
4.1	The process of validation and repair of a planar partition from a user's perspective.	46
4.2	The different types of zero area features that can occur in a ring.	48
4.3	The polyline generated from a seeding triangle in the interior of the ring joins all holes with the external boundary.	49
4.4	The previously generated polylines are first cut and then joined in the correct order to form separate rings.	50
4.5	Once orientation criteria are met, it is known that the ring interior lies to the same side of each directed edge on the boundary.	51
4.6	Triangles adjacent to the outer boundary of a polygon are tagged first. . . .	52
4.7	Different undesirable results when selecting different tagging criteria. . . .	53
4.8	A tagged triangulation of the convex hull of two polygons for the Arribes del Duero Natural Park in Spain and the International Douro Natural Park in Portugal.	54
4.9	Regions are defined as adjacent triangles with equivalent sets of tags.	56
4.10	Different repair operations used in the two polygons for the Arribes del Duero Natural Park in Spain and the International Douro Natural Park in Portugal.	56
4.11	A formerly unified polygon is separated into pieces during the reconstruction process.	58
4.12	Finding nodes for the Winged Edge data structure	60
4.13	CORINE tile E37N28, after finding nodes to be used in the Winged Edge data structure.	60
4.14	The topology generation process in CORINE tile E37N28.	61
5.1	Example of a program to repair four tiles of the CORINE 2000 data set. . . .	65
5.2	The output from the input file from Figure 5.1.	66
5.3	Different interpretations of the "backforth" polygon (Appendix Figure C.1d). .	67
5.4	Different interpretations of the "bighole" polygon (Appendix Figure C.1g). .	67
5.5	Different interpretations of the "partially" polygon.	68
A.1	UML diagram of the classes used in the software prototype.	78

A.2	UML diagram of the data structures used to generate closed rings.	79
A.3	UML diagram of the data structures used to generate topology.	80
A.4	Edge splitting when one edge lines in the interior of another one.	81
A.5	Passing once through a constrained edge is equivalent to moving from the exterior to the interior of a ring, or vice versa.	83
A.6	Seeding the polyline generation algorithm.	84
A.7	Order of the recursive algorithm to obtain the polyline containing all boundaries of a polygon.	84
A.8	Traversing order when navigating through triangles in a clockwise manner.	85
A.9	Step by step demonstration of the polyline generation algorithm.	86
A.10	The polyline generated by the line creation algorithm in polygon 67 of CORINE tile E39N32.	87
A.11	“Bridges” are generated connecting interior triangles and holes to the outer boundary of a polygon.	89
A.12	Removal of a “bridge” connecting the outer boundary to a triangle in the interior of a polygon.	89
A.13	An intermediate state in the ring reconstruction algorithm.	91
A.14	Using the point in polygon test, some holes might not be correctly detected as being inside the outer boundary.	92
C.1	Zero area unit test polygons.	98
C.2	Unit test polygons with a single interior connected region.	99
C.3	Unit test polygons with two interior connected regions.	100
C.4	Unit test polygons with three interior connected regions.	101
C.5	Unit test polygons with degenerate holes.	102
C.6	Unit test polygons with an unknown number of interior connected regions, depending on the interpretation of the polygon.	102



List of Tables

3.1	Relevant constraints for polygon validation and repair in ArcGIS [ESRI, 2009a].	29
3.2	Relevant constraints for geometry cleaning in JTS and GEOS [GEOS, 2010].	29
3.3	Relevant constraints for geometry validation in Oracle Spatial [Oracle, 2009a,b, 2010].	30
3.4	Map axioms for planar partition validation.	32
3.5	Relevant constraints for planar partition repair in GRASS [GRASS, 2006]. .	38
3.6	Relevant constraints for planar partition repair in Radius Topology [Baars, 2003; Louwsma, 2003].	38
4.1	The repair operations implemented in the prototype.	55
5.1	Validation of the unit test polygons.	69
5.2	Planar partition repair comparison using large data sets.	72
A.1	CGAL Type definitions used in the prototype.	79
A.2	Properties of the polyline passing through all boundaries of a polygon. . . .	85



Acronyms

CGAL	Computational Geometry Algorithms Library
CGIS	Canada Geographic Information System
CORINE	Coordination of Information on the Environment
GDMC	Geo-Database Management Center
GEOS	Geometry Engine Open Source
GIS	Geographic Information Systems
GRASS	Geographic Resources Analysis Support System
GTS	GNU Triangulated Surface
IEEE	Institute of Electrical and Electronics Engineers
INEGI	National Institute of Statistics and Geography (<i>Instituto Nacional de Estadística y Geografía</i>)
INSPIRE	Infrastructure for Spatial Information in the European Community
ISO	International Organization for Standardisation
JTS	Java Topology Suite
NAA	Node-Arc-Area
OGC	Open Geospatial Consortium
QGIS	Quantum GIS

LIST OF TABLES

SYMAP	Synagraphic Mapping Technique
UML	Unified Modelling Language

Introduction

Planar partitions constitute one of the most important data representations in GIS, and are extensively used as a base for other operations. However, since polygons are often stored independently, various errors are introduced during their creation, manipulation and exchange. This creates many problems for algorithms that rely on valid planar partitions, and can cause them to fail or give erroneous results, often without any warning to the user.

Existing approaches to solve this usually involve polygon repair using a list of constraints, and complex planar partition repair operations performed on a planar graph. However, these have many shortcomings in terms of complexity, numerical robustness and difficulty of implementation. Moreover, they leave many invalid cases untouched.

To solve this problem, a novel method to validate and automatically repair planar partitions has been developed. It uses a constrained triangulation of the polygons as a base, which being by definition a planar partition, means that only relatively simple operations are needed to ensure that the output becomes valid. Point locations are maintained throughout the process, while fully automatic repair is possible using customisable criteria. This approach is also extensible to individual polygons, is capable of handling a larger variety of cases and has good performance compared to existing alternatives; all of this with numerical robustness and maintaining topological consistency throughout.

This chapter starts by giving a brief history and motivation behind the problem in Section 1.1. Afterwards, the objectives of this thesis research are discussed in Section 1.2. The scope of the thesis is then summarised in Section 1.3. Finally, an outline of the organisation of this document is covered in Section 1.4.

1.1 MOTIVATION

Geographic Information Systems (GIS) have gone a long way since their inception in the late 1960s [Coppock and Rhind, 1991]. From their origins in map making and analysis, modern systems dramatically increase the amount of information that can be handled and the variety of operations that may be performed on a map.

However, despite ever increasing functionality and sophistication, many of the representations and techniques that originated in cartography, and were back then introduced in digital form, still conform the basis of modern GIS. One prime example is the use of thematic maps where areas are patterned according to a set of criteria, so that each area with the same pattern is considered to have certain characteristics¹ (see Figure 1.1). These types of maps were already supported in the earliest systems, like the Synagraphic Mapping Technique (SYMAP) and the Canada Geographic Information System (CGIS) [Tomlinson, 1988].

In analog (hard copy) versions of these maps, the boundaries between different areas were simply the hand drawn lines made by a cartographer. Later on, when digital versions were implemented in computer programs, polygons became the base representation to delimit these areas [Peucker and Chrisman, 1975], since they have the advantage of being well suited for computer based storage, processing and display [Mortenson, 1999]. These maps, known as *planar partitions* or *polygonal coverages*, are thus a central concept in GIS.

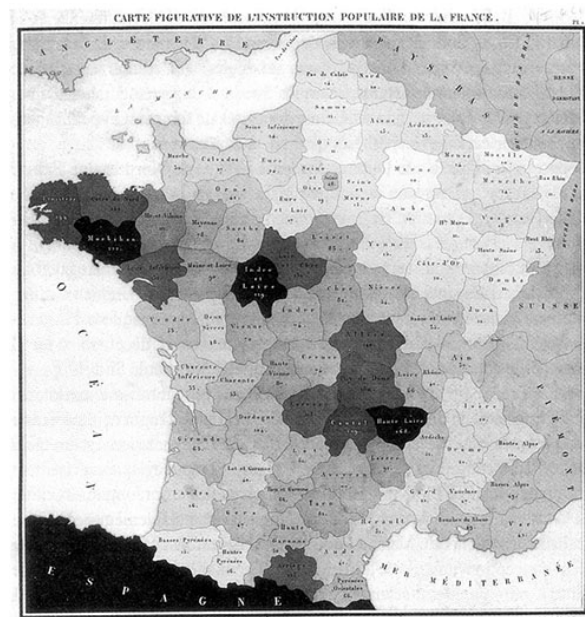
A planar partition, in a more formal definition, is a tessellation (subdivision) of a region of the plane² into a set of areas, without any gaps or overlaps; and is a structure that is not only used in GIS, but in computer graphics and finite element analysis as well, among other fields. Figure 1.2 shows some examples of planar partitions used in GIS.

In a strict sense, a planar partition comprises a single joint region on the plane, without any gaps or overlaps. However, in real-life data sets, there might be several disjoint regions, while some holes could be allowed. For instance, in CORINE 2000 (a European land cover data set which is defined as a planar partition), islands are valid disjoint regions, while some areas without coverage (e.g. parts of Andorra, Switzerland and Kosovo) are surrounded by areas with coverage. The regions that should be covered in a certain data set are known as its *domain*.

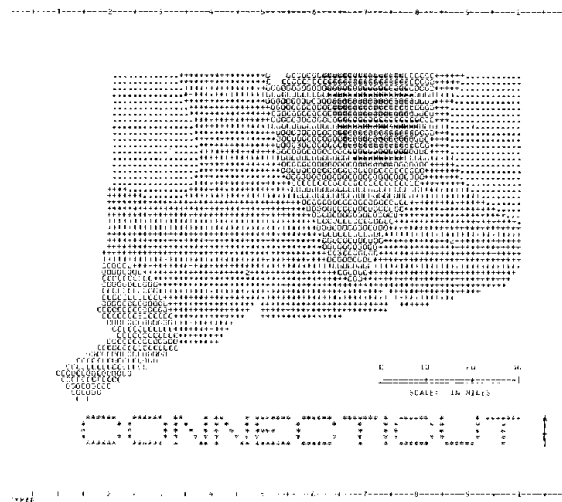
Planar partitions are not only used independently, but are also very common as a base for other operations, since their properties simplify many calculations and diminish the number of conditions that have to be checked for in an algorithm. For instance, since we know that there are no gaps or overlaps, the total area of a certain type of features is simply the sum of the area of all polygons of that type. However, if these features are overlapping, the area would be overestimated, since some regions would be counted multiple times.

¹Such as choropleth (numerical data mapped to a colour scale), isarithmic (contour maps) and dasy-metric (homogenous aggregated regions) maps.

²This concept is extensible to higher dimensions, in which planes (or hyperplanes) divide space into regions. However, this thesis deals with the 2D case only.

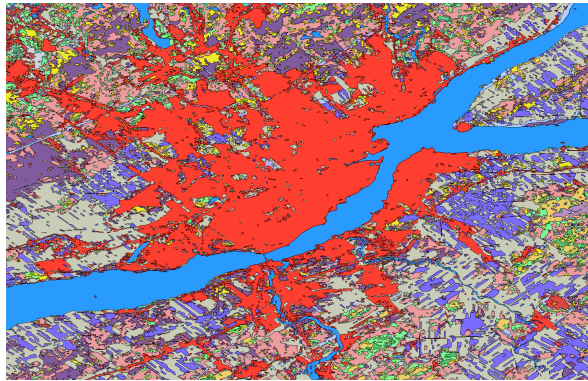


(a) Pierre Charles Dupin created the first choropleth map already in 1819, representing the distribution of illiteracy in France at the time. From Dupin [1826] through Friendly and Dennis [2001].



(b) In 1964, SYMAP was already able to generate digital choropleth maps from spatial statistical data, which were displayed with a line printer [Chrisman, 1988]. Through Friendly and Dennis [2001].

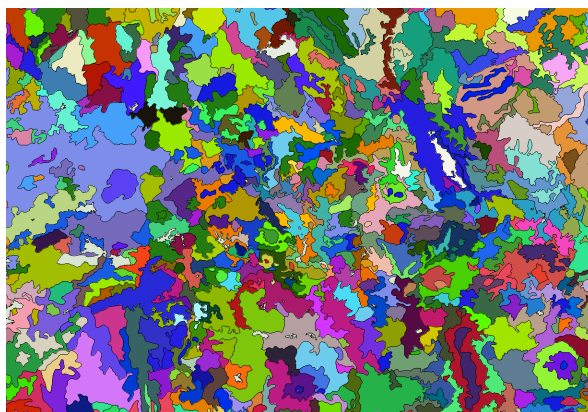
Figure 1.1: Choropleth maps were already commonplace for the display of thematic information long before the advent of computers, and a digital form of them appeared early in GIS history.



(a) A choropleth map from the area surrounding Quebec City in Canada, where every subdivision represents an area with a certain land cover type. Made from GeoBase data.



(b) A topographic map from Delft. Polygons show land use, infrastructure and relief. From Top10NL.



(c) A soil map from central Mexico. Each polygon is a region with approximately homogenous soil composition. Generated from INEGI data.

Figure 1.2: Different examples of the use of planar partitions in GIS.

Algorithms based on planar partitions assume that these are valid, and when an invalid planar partition is given as input, software will likely fail or give an incorrect output, often without any warning to the user. For example, if we need to know the length of the border between two countries, an algorithm will fail if the two polygons representing them are separated by a small distance. This distance might be too small to be noticed by a user working with the data when viewing it at a normal zoom level, and therefore pass undetected [Laurini and Milleret-Raffort, 1994].

For a planar partition to be considered valid, its component polygons should be valid to begin with. Having this in mind, several standards to define what exactly is a valid polygon have been created, with the ISO 19107 [ISO, 2003] and the OGC Simple Features Specification [OGC, 2006] being heavily promoted by standardisation bodies, and together with the ESRI Shapefile specification, they have become predominant. Still, these are not only incompatible with each other, but individual implementations for a single standard yield significantly different results [van Oosterom et al., 2003; van Oosterom et al., 2004], creating a need to agree on a polygon definition, interpretation and a few more rules beforehand (e.g. tolerance values) before being able to establish what is valid in a certain context. Based on such rules, some appropriate validation and repair operations on a single polygon can be defined.

Once polygons themselves are deemed to be valid, it is necessary to consider some additional requirements for a valid planar partition. Namely, that its component polygons should be non-overlapping and cover the entire domain, which implies that they should be connected (unless they were originally disjoint) and that their union should be equal to the original undivided polygon (that defined the complete region in the first place). Importantly, several of the polygons have holes, which should be filled with other polygons. These correspond to physical features, such as enclaves and exclaves, islands or lakes. Recursively, it is possible to have polygons with holes within polygons with holes and so on iteratively, which can be very complex topologically and hard to repair (see Figure 1.3), since errors create much more ambiguous situations where the original intent behind the data is harder to discern. For instance, consider if a hole of a polygon is found outside its outer boundary, in the interior of a different polygon. It could be that the hole is in the wrong place, that it belongs to the other polygon, or that it was the outer boundary of another piece of a multi polygon, among other feasible options.

In practice, the geometry of the polygons in a planar partition is often stored independently, so that the shared boundaries of adjacent polygons are stored in each of them, and also have no explicit topological information, such as telling us the neighbours of a given polygon. This is true for many land cover data sets, like CORINE and the Mexican and Canadian land cover data. There is always a trade-off in the storage of more or less topological information, as discussed in van Oosterom et al. [2002].

Meanwhile, while other data sets do have topological information, it is sometimes wrong or not enforced in their geometry. This leads to consistency problems being introduced, both from human error [Gold et al., 1996] (e.g. shifting a polygon by hand), or from computer inherent problems coming from floating-point arithmetic or limited precision [Schirra, 1997]. Two related examples from the CORINE data set are shown

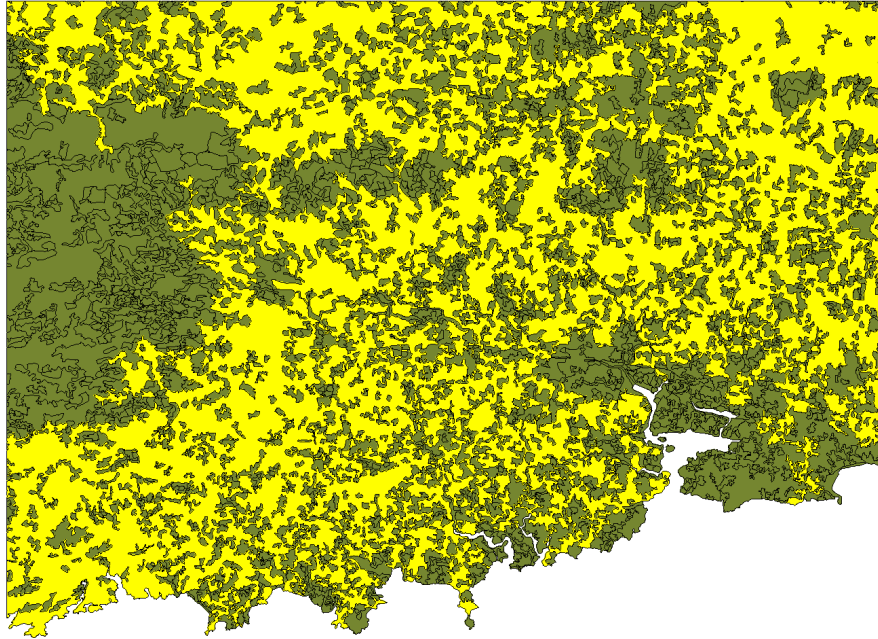


Figure 1.3: A selection of the largest polygon in the CORINE 2000 data set (in yellow). It consists of over 77000 vertices and has more than 500 holes.

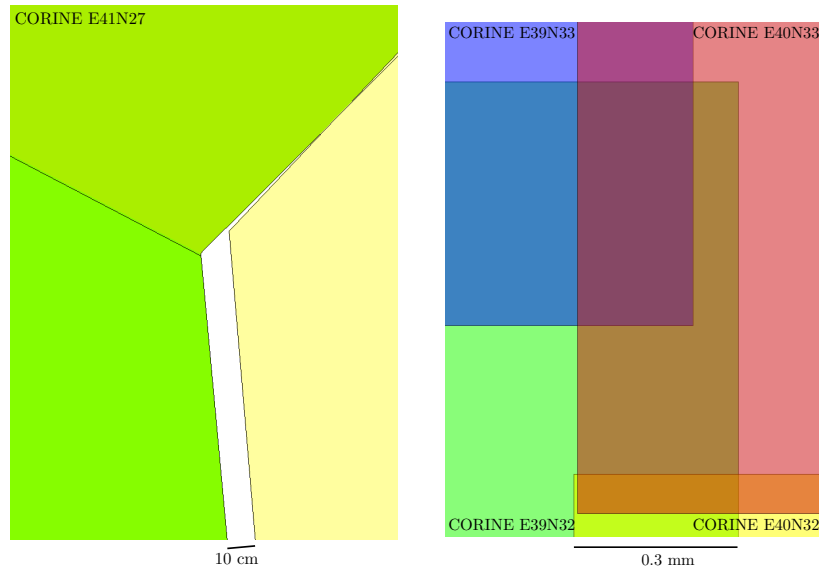
in Figure 1.4.

This situation is the opposite to what is usually desired for computational geometry algorithms, which are designed for a machine model with exact real arithmetic [Kettner et al., 2007], and assume valid and consistent input data, which is of particular relevance when generating topological information or incorporating data sets from different sources.

1.2 OBJECTIVES

Considering that planar partitions are extensively used, and that the algorithms that use them need to start with valid planar partitions, there is a need for good tools and algorithms to validate and repair them. Such tools are a crucial part in the workflow of GIS users, since as ESRI states: “the onus is on the data’s user to verify that it contains features with valid geometries before using it” [ESRI, 2009a]. In other words, users should not readily assume that the data given to them is valid. After all, their results depend on it.

Additionally, since planar partition data sets can be very large, sometimes consisting of hundreds of thousands of polygons, each of them with hundreds of vertices, this validation and repair process should be done in as much an automated manner as possible. However, existing solutions for this problem usually work only in a semi-automatic way, and are based on enforcing geometric and topological correctness in separate and



(a) A polygon appears to have been manually shifted, creating a gap with its neighbours.

(b) In a data set split into rectangular tiles, the area where four tiles join shows that they do not match properly.

Figure 1.4: Different errors present in the CORINE 2000 data set.

weakly linked processes. These require the construction of a graph-based representation of the polygons³, upon which a set of axiomatised validation rules are applied [Plümer and Gröger, 1996, 1997], which must combine the geometric and topological aspects. This introduces several difficulties and limitations, which are discussed in Chapter 3.

However, based on the preliminary work of Ledoux and Meijers [2010], there are advantages in using a constrained triangulation. This benefits from simpler algorithms and data structures and simpler handling of holes/islands. Also, degenerate cases should be easier to deal with, and there may be additional advantages in using a constrained triangulation for repair as well. This will be discussed in Section 2.6.

The main objective of this thesis is therefore to answer the following research question:

“What benefits does using a constrained triangulation bring for planar partition validation and fully automated repair?”

To accomplish this, new algorithms to validate and automatically repair planar partitions conforming to the Simple Features paradigm will be developed. These will use a constrained triangulation of a set of polygons as a base (which is by definition a planar partition), together with relatively simple operations to ensure its validity throughout

³Something that is not easy to do robustly, since there are many degenerate cases to take into account.

the process. The beneficial properties of planar partitions will be used to achieve a good solution.

To analyse, test and improve the algorithms, and encourage further development, a fast implementation is to be written in C++, using external libraries for some functionality⁴. This should be extensively tested, which is possible with the excellent availability of freely available data sets, such as the Coordination of Information on the Environment (CORINE) Land Cover from the European Environmental Agency.

Additionally, the capabilities and performance of this approach should be compared with existing solutions, as described in the following section, both using purpose-made test data showing specific problems to be tackled, and with large general purpose data sets. The former are meant to see if the software is able to distinguish and handle each case appropriately, while the latter is a stress test to assess performance characteristics.

As mentioned previously, there is a need for validation and automatic repair tools for planar partitions. In this context, the development and analysis of such a tool in GIS research can be seen as *toolmaking*, where end users can only benefit from the engineering of better tools [Wright et al., 1997], especially when it concerns a representation as prevalent as planar partitions.

1.3 SCOPE

Because of time constraints and design considerations⁵ that have to be made, the scope for the algorithms and software can be summarised as follows:

Input polygons Accept the input of individual (multi) polygons in the same coordinate system, without making any assumptions about compliance of the data to any particular standard (e.g. modelling of holes, polygon winding rules). This had to be decided based on the fact that most data sets tested have invalid polygons.

Integrity of Data Points should not be moved during processing, and unnecessary points should not be created. In this manner, points can be exactly the same as in the input (see Section 2.1).

Numerical Robustness Care should be taken not to use operations whose result can change due to numerical inaccuracies in floating point arithmetic. This is discussed further in Section 2.2.

Repair Algorithms Simple and generic repair functions. At the very least, repair operations based on boundary length, number of neighbours and priority classes should be implemented. Also, there should be a function able to always repair the planar partition, no matter the circumstances.

⁴OGR for input and output, and CGAL for geometric operations.

⁵e.g. increased processing time vs. increased memory usage

Output polygons Two types of output are expected: valid polygons according to the Simple Features specification (for geometry only) and a Winged Edge data structure (for geometry and topology) [Baumgart, 1974].

Meanwhile, testing and comparisons have been done with the main software tools available to me: FME, Oracle and Radius Topology⁶, ArcGIS, GRASS, QGIS and JTS / GEOS⁷. However, since these range from general purpose GIS to very specific tools for spatial data consistency, their use and approaches are sometimes not directly comparable, and their specific strengths with regards to planar partition validation and repair vary. Due to this, representative examples are emphasised throughout this thesis, while the time spent on each tool reflects the areas in which it exhibits the most potential within the topic. Additional information regarding the approaches of these tools is found in Chapter 3, while testing is in Chapter 5.

1.4 STRUCTURE OVERVIEW

The structure of the thesis is as follows:

- Chapter 2 introduces the reader to the *Background Information and Theory* behind the topic, both to put the work in context and to give enough information to understand the decisions made. It covers various computational geometry subjects, while pointing to specific problems which are relevant to this thesis.
- Chapter 3 covers the *State of the Art in Planar Partition Validation and Repair*, grouping the most commonly used approaches of currently used software by their approach. Benefits and drawbacks with respect to geometry and topology are pointed out as well.
- Chapter 4 deals with *Using a Constrained Triangulation for Planar Partition Validation and Repair*, the main topic of this thesis. There, the entire process of validation and repair which was developed is explained, together with the most important issues that arise, both in theory and implementation. Examples from the developed prototype are used to clarify the method, where necessary.
- Chapter 5 covers *Implementation, Experiments and Discussion*, describing the implementation of the developed prototype, including the main invalid polygon and planar partition configurations and how they are handled. Later, there are comparisons to analyse how the developed algorithms compare to existing approaches.

⁶Radius Topology builds on the Oracle database, and their functionalities should therefore be compared together. Oracle provides tools for individual polygon validation and repair, while Radius Topology introduces topological data, which is useful for planar partitions. However, Oracle also has its own topology solution with Oracle Topology.

⁷Which is used in GRASS, FME, PostGIS, and MapServer, among others. GRASS and FME are analysed separately, since they provide additional functionality than that provided by JTS and GEOS.

- Chapter 6 includes *Conclusions, Recommendations and Future Work*.

Additionally, four appendices supplement the body of the thesis. These are:

- Appendix A explains the reason behind some *Design Considerations and Implementation Checks*, including algorithms which were deemed too low level to be included in the main body of the thesis. All important data structures used are also discussed.
- Appendix B shows *Completeness and Correctness of Planar Partitions Repair*, summarising how each possible invalid configuration is identified and dealt with. It also includes the handling of degenerate cases, which is crucial in the implementation, but do not contribute to the understanding of the algorithms.
- Appendix C gives a brief description of the *Data Sets Used*, which should help with a better understanding of the examples presented.

Background Information and Theory

Whether spatial information is a special kind of data is a very contested and long standing point of debate for people in computer graphics, GIS, robotics, computer vision, and other fields where computational geometry is a main concern [Anselin, 1989]. While all fields have their own very specific needs and concerns; specially designed data structures and algorithms, indexing schemes and the duality between geometric and topological data all make a strong point that it just might be.

Nevertheless, it is a fact that computers are one-dimensional processing machines to a large degree¹, requiring many provisions in order to deal with spatial information efficiently. This, together with the large volumes of data to be processed makes good algorithms in spatial data processing crucial to be able to manage even very simple spatial operations.

For the understanding of the issues involved in planar partition validation and repair, several computational geometry related topics are explained within this chapter. These are treated in increasing abstraction, from low level to high level, which should help to emphasise how much low level details still influence the general process in geometric algorithms.

Section 2.1 therefore starts with an explanation of how geometric features (e.g. polygons) are represented in a computer, which introduces several practical considerations, such as storage, computing time and memory consumption. Afterwards, Section 2.2 discusses how robustness issues affect geometric operations due to the finite precision inherent in computers. Section 2.3 then deals with the similarities and differences in polygon definitions, and how using different definitions can bring certain difficulties. Later on, Section 2.4 introduces the related problem of data conflation, where separate data sets are integrated. After this, and expanding on the already covered topics, planar partitions are covered in Section 2.5, including their relevant properties

¹For instance, with regards to storage, memory addressing and instruction execution.

to this thesis. Finally, Section 2.6 progressively introduces the main types of triangulations and the triangulation related concepts necessary to the understanding of the algorithms described in this document.

2.1 COMPUTER REPRESENTATION OF GEOMETRIC FEATURES

The notion of geometry is central to the understanding of the properties of space, including the size, shape and placement of objects within it. Additionally, together with the use of coordinates, objects are able to be represented analytically².

In this manner, points in Euclidean space³ are able to be represented uniquely as an ordered tuple of numbers (a_0, a_1, \dots, a_n) , where each term refers to the distance, from the origin of a predefined reference frame to the point, in the direction of a certain axis. For the context of this thesis these are to be restricted to two-dimensions and be stored with limited precision (i.e. assigned only pre-defined discrete values), as is most common in digital environments, which has important consequences in terms of the operations that are allowed (see Section 2.2).

Similarly, lines and polygons are to be stored as a sequence of points, where each point represents a vertex, which are joined by straight line segments⁴ [Preparata and Shamos, 1985]. Note that this entails a certain ordering for the vertices of a polygon, as seen on Figure 2.1. In conjunction with their coordinates, points, lines and polygons also contain ancillary information related to them, which together are known as *features*. One example use of this paradigm is the so-called spaghetti data structure [Egenhofer and Herring, 1991], which is shown in Figure 2.2.

Expanding on this representation, the Simple Features specification defines polygons with holes by one exterior boundary and zero or more interior boundaries, together with some geometric and topological constraints [OGC, 2006] (further discussed in Section 2.3).

However, one might consider what happens when a planar partition has to be stored in this manner. The coordinates of all points (except for possibly the ones adjacent to the exterior) have to be stored several times, one for each polygon passing through that point, together with all accompanying information related to them. This is not only inefficient storage-wise, but also having several unrelated points at the same location means that changes might be applied inconsistently. A shifted polygon, as the one shown previously in Figure 1.4a, is a possible result.

A different problem caused by storing each polygon independently is the lack of information showing relations between polygons (topology). For instance, to know all polygons that are adjacent to a certain one, it might even be necessary to check all the coordinates of every polygon that is stored (without any spatial indexing).

²Referring to classical analytic geometry, not algebraic geometry.

³Other types of space (e.g. set-based) are also possible, but are not particularly relevant within the scope of this thesis [see Worboys and Duckham, 2004, chap. 3].

⁴More generalised versions of a polygon are also possible, allowing for curves instead of only straight line segments. However, these are usually known as geometric figures or shapes, not polygons.

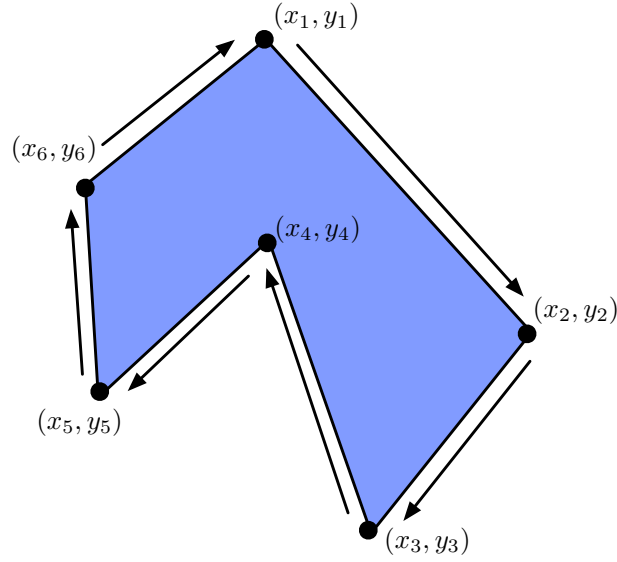


Figure 2.1: A polygon is represented by an ordered sequence of points, each of them defined by their coordinates.

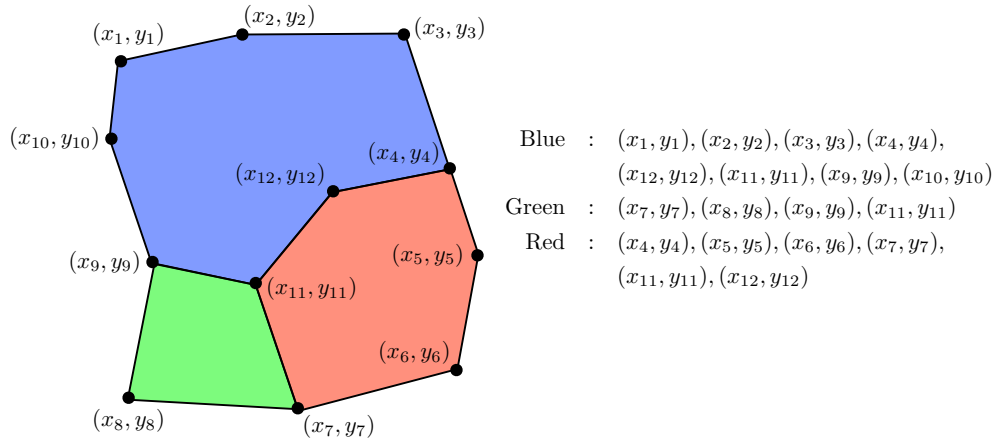


Figure 2.2: In the spaghetti data structure, polygons are represented as a closed loops, each of them being a list of points.

To have fast access to these types of relations, topological data structures might be used. In these, additional information, such as adjacencies, ordering and connectivity are stored. Examples include the Node-Arc-Area (NAA) (also known as Node-Arc-Polygon) [Laurini and Thompson, 1992] and Winged Edge [Baumgart, 1974] data structures. The latter is shown in Figure 2.3.

Topology has different meanings in different fields of study. However, in this context, it is the branch of geometry concerned with properties that remain invariant under *topological transformations* or *homeomorphisms*, which are continuous functions with a continuous inverse function as well. These can be equivalent to deformations of a rubber sheet, in which stretching and contracting space is valid, but tearing or folding is not.

Similarly, topological information in computational geometry refers to the relations that are kept under these conditions. Some examples, which were used during this thesis include: adjacencies between faces, ordering of edges incident to a vertex, orientation of rings and polygons, sets of triangles inside and outside a set of polygons, among others. These are specially useful, since they mostly avoid numerical robustness problems⁵, which are discussed in the next section.

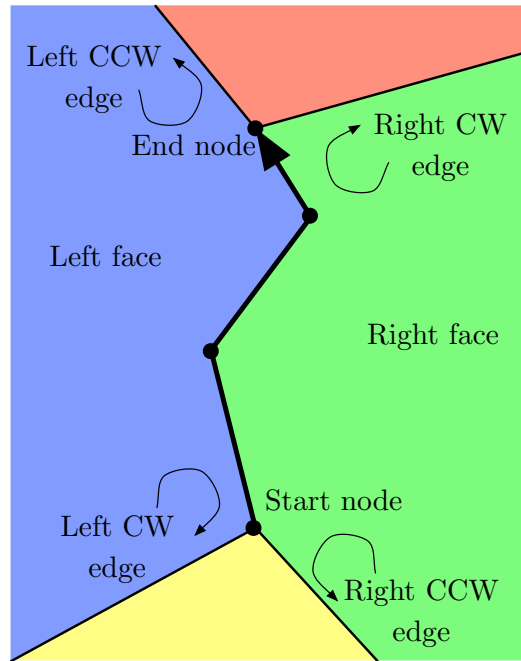


Figure 2.3: The Winged Edge data structure and the spatial relationships associated with it. In the figure, they are defined with respect to the thick edge.

⁵Not completely, since points can be moved in a way that breaks these conditions (e.g. a point that moves from one side of a line to the opposite one). However, numerical robustness problems are much easier to avoid.

2.2 ROBUSTNESS IN GEOMETRIC OPERATIONS

In the previous section, it was mentioned that points are stored with limited precision. This is a necessary consequence of having a digital representation of numbers, which allows points to be only at intersections of a grid, which is evenly spaced in the case of integers, but unevenly in the case of floating-point numbers⁶.

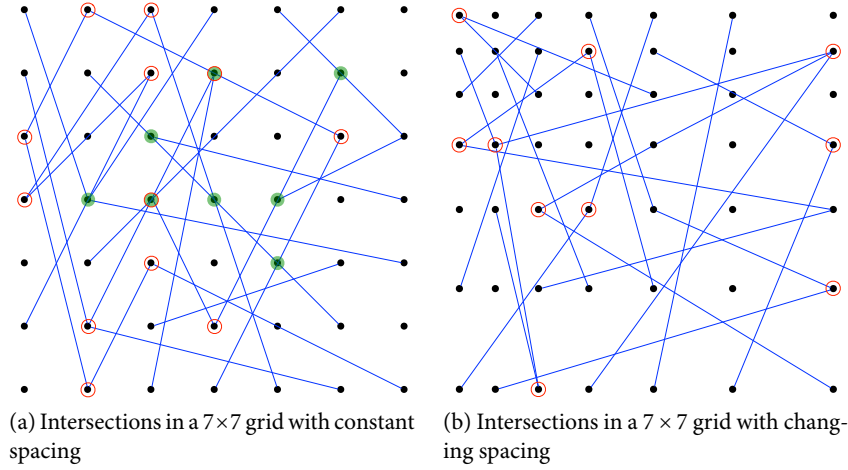


Figure 2.4: When computing intersections of randomly generated line segments, intersections in a grid with growing spacing have a much lower likelihood of having an exact representable value (green discs), except for when they are at end-points of both lines (red circles). This means that if segments have to be split, their component parts most likely become non collinear.

Floating-point numbers were devised in order to represent a much larger range of numbers than it would be possible otherwise. However, in order to achieve this, it uses a flexible representation of the form:

$$\pm d_0.d_1d_2 \dots d_{p-1} \times \beta^e$$

where d is called the *significand*, represented by p digits; β is a fixed pre-defined number (generally 2 or 10); and e is the exponent. For instance, for double precision numbers in the widely used IEEE 754 standard, one bit is used to store the sign, 52 for the significand and 11 for the exponent. This is the representation used in ESRI Shapefiles.

There are a few important consequences of this representation. First of all, the density of numbers that are able to be represented is highest close to zero, and declines as one gets away from it [see Goldberg, 1991]⁷. This means that moving a polygon

⁶It is possible to use exact arithmetic instead, but it has a severe impact on performance and memory consumption [see Schirra, 1997].

⁷Actually, the distribution of floating-point numbers is generally considered to be logarithmic [Scheidt and Schelin, 1987].

slightly distorts its shape, and is therefore a destructive operation (i.e. even if it is moved back to its original position, it will likely be deformed); while the same holds for changes in the representation, like moving from a 64-bit to a 32-bit one. For the same reason, the intersection of two line segments will almost certainly have to be stored at a point which is on neither line, as shown in Figure 2.4.

A second consequence is that there are many numbers in common use that are not able to be represented. A famous example is 0.1, which when using $\beta = 2$ (binary representation) can only be approximated as $1.1001100110011001 \dots \times 2^{-4}$. In the context of this thesis, this entails that points will be most likely be stored with a position slightly different than that that was intended, which is only exacerbated when dealing with projections and internal accuracy models in software, since it would imply resampling points to the internal model (i.e. a different grid) and then again when exporting the points back.

However, the most important consequence is that some geometric operations using floating-point arithmetic might give incorrect results due to the loss of precision involved⁸. This problem can occur in almost any geometric operation, including but not limited to, the point-in-polygon, side and convex hull operations [see Kettner et al., 2007], with one such example shown in Figure 2.5. Because of this, to ensure that the algorithms developed for this thesis are robust, all algorithms developed work strictly either with topology or geometric operations that are known to be safe, and new points are generated only when necessary.

2.3 VALID POLYGONS

There are several conditions which can be intuitively expected as part of any valid polygon with holes definition (i.e. those which it cannot be considered a polygon without). For instance, most polygon definitions [ESRI, 1998; ISO, 2003; OGC, 2006] agree that the rings that define the outer boundary and inner boundaries (for holes) of each polygon should be closed, that rings that define holes are properly nested inside the outer boundary and not crossing each other, and that segments that have zero-area should not be allowed (e.g. zero area rings or polygons, spikes, cut-lines, punctures, polygons with a hole covering it completely, etc.).

Additionally, we find other criteria meant so that each polygon is uniquely defined, since it is desirable to have a unique representation of each polygon, both for standardisation and to simplify geometric algorithms. This is a distinct set of requirements, since different standards adopt different decisions. The Simple Features specification thus adds that the interior of a polygon should be connected, implying that each hole should touch the boundary only at one point, among other things; while the ISO one specifies that the outer boundary should be specified in counterclockwise order and the inner boundaries in clockwise order. Figure 2.6 shows how a bow tie polygon is modelled with increasingly strict requirements, while Figure 2.7 exemplifies how the ESRI Shapefile and the OGC standards differ in modelling of holes.

⁸Such as when subtracting two very similar numbers. It is known in this context as loss of significance.

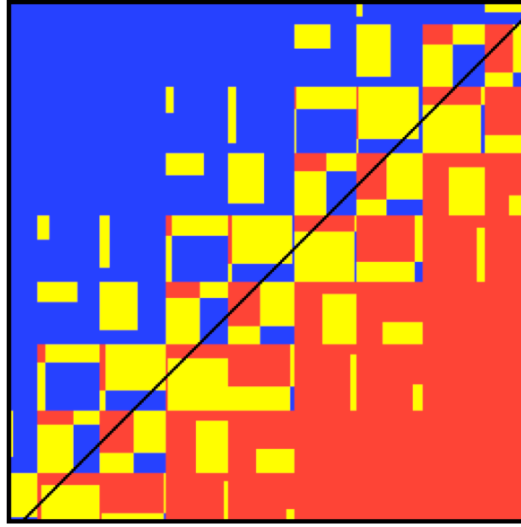


Figure 2.5: The results of the floating point orientation predicate using extended double precision arithmetic in a small area around the midpoint of the line segment between $(17.300000000000001, 17.300000000000001)$ and $(24.000000000000005, 24.00000000000000517765)$. Pixel colours represent whether the result is that a pixel is collinear (yellow), negative (red) or positive (blue), while the line is represented in black. Note that there are sign reversals (blue areas below the line and red areas above it) and the uneven fractured nature of the collinear areas. From Kettner et al. [2007].

Finally, there is an additional set of conditions that are application (or implementation) dependent, meant to avoid specific problems (e.g. inaccuracy). To this end, restrictions on a minimum separation between vertices and/or edges, minimum angles between edges and minimum area of rings and/or polygons may be defined [Milenkovic, 1993]. In this respect, the concept of a tolerance value is best defined in van Oosterom et al. [2004], where it is able to provide a measure of the robustness of a given polygon.

For the purpose of this thesis, the rules for a valid polygon are based on van Oosterom et al. [2004] as well, with polygons formed by rings composed of straight line segments. However, in order to promote having a unique representation for a given polygon, these rings should form only one outer boundary, which is oriented counter-clockwise; and zero or more inner boundaries, which are oriented clockwise. Rings are allowed to touch (but not cross) themselves or other rings, as long as any point inside or on the boundary of the polygon can be reached through the interior of the polygon from any other point inside the polygon. The use of a tolerance value is not enforced, since the algorithms developed for this thesis are robust and do not move points. However, it might be advisable to use this concept in other software, once validation and repair have been completed.

The use of these rules implies that the polygons that are considered valid also conform to the Simple Features definition, and have the ISO orientation rules. This makes

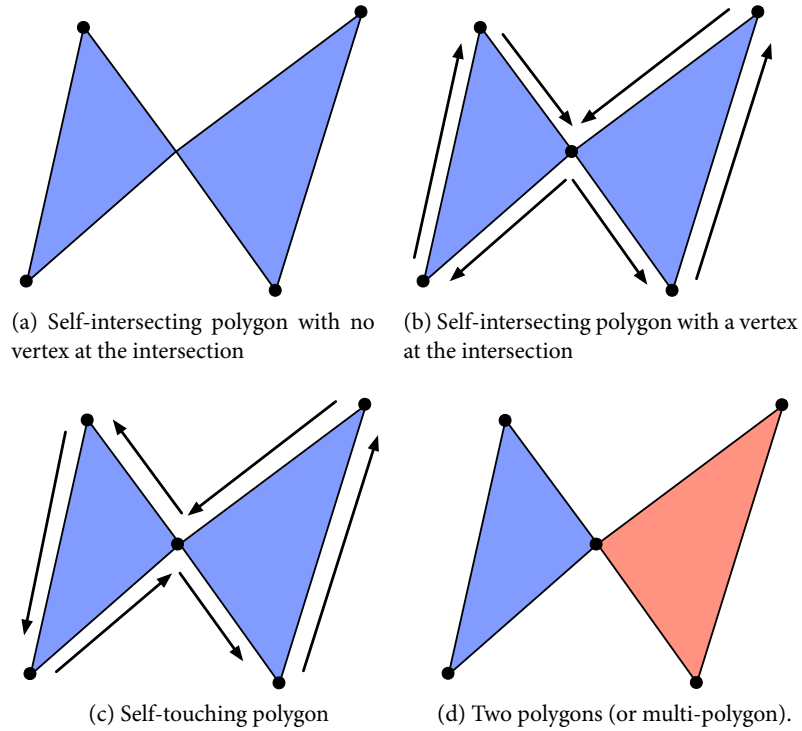


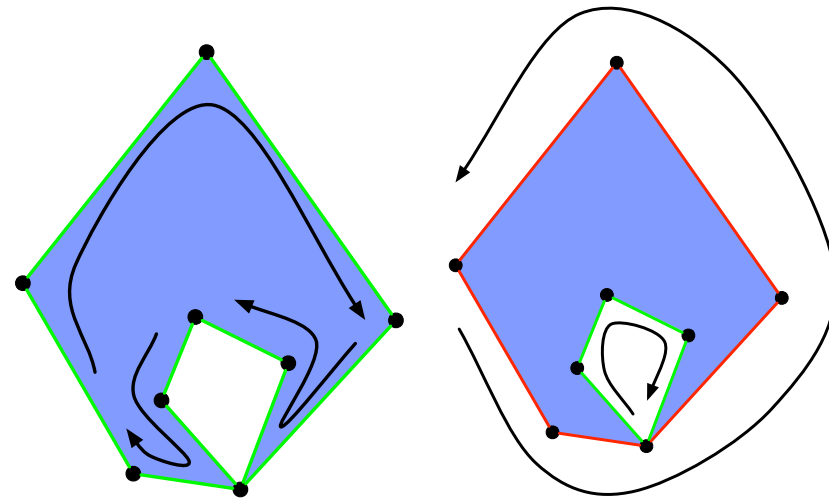
Figure 2.6: The bow-tie polygon, modelled according to different criteria

them compatible with a wide range of applications based on both standards.

Since there are several differing polygon definitions commonly used; and even when the standards for what is considered as a valid polygon are well defined, invalid polygons continue to be prevalent in practice, there is clearly a need for tools to correct these problems. Moreover, since there are data set dependent restrictions in place which should not be disregarded, it is important to avoid moving points during processing, and setting uniform rules for polygon representation brings easier post processing as an added benefit.

2.4 THE CONFLATION PROBLEM

Conflation involves combining multiple data sets in order to make a new one, usually to improve either the spatial extent or the accuracy of the data [Lynch and Saalfeld, 1985]. In Yuan and Tao [1999], a distinction is made between horizontal and vertical conflation. Horizontal conflation refers to edge-matching of neighbouring data sets to eliminate discrepancies at the border region, while vertical conflation involves combining data sets covering the same area. Both topics are relevant to planar partitions, since the former is used to join tiled and cross-border data, and the latter to create overlaid maps using data from different sources.



(a) Shapefiles can include holes touching the outer boundary as part of it, all in clockwise order. This is equivalent to saying that the interior of the polygon should always be to the right of an observer walking along the boundary [ESRI, 1998]. Note that there is a single boundary describing this entire polygon.

(b) A valid polygon according to the ISO and OGC standards should have the outer boundary in counterclockwise order, while the inner boundaries should be in clockwise order. This means that the interior of the polygon should always be to the left of an observer walking along the boundary [ISO, 2003; OGC, 2006]. Note that the polygon is described by two separate boundaries.

Figure 2.7: The ESRI Shapefile and ISO standards differ in the modelling of polygons with holes.

The issue of horizontal conflation is becoming increasingly relevant, since the need to harmonise different data sets across borders is also becoming more and more apparent, such as within the framework of the INSPIRE project⁹ [Wiemann and Bernard, 2010]. Country borders defined based on natural features of the terrain are a good example, since their continuous nature basically ensures that independently produced data will not match at the border [see Burrough, 1992]. Figure 2.8 shows an area along the Spanish-Portuguese border with this problem. However, these sort of problems are not only present between different data sets, but within a single (tiled or otherwise subdivided) data set as well [Chrisman, 1990], as shown previously in Figure 1.4. Digitised data sets suffer most from this issue [Beard and Chrisman, 1988].

An entirely different issue is that of vertical conflation, in which two very different types of behaviour might be expected. One would be to establish relations between features that are supposed to match (e.g. to obtain features that contain data from multiple data sources), which is not discussed within this thesis; but the second would involve taking certain features from different sources to create an overlaid map (e.g.

⁹Semantic matching might be involved as well, as it is a crucial part of data conflation [Kiehle et al., 2007], but it is not relevant within the scope of this thesis.

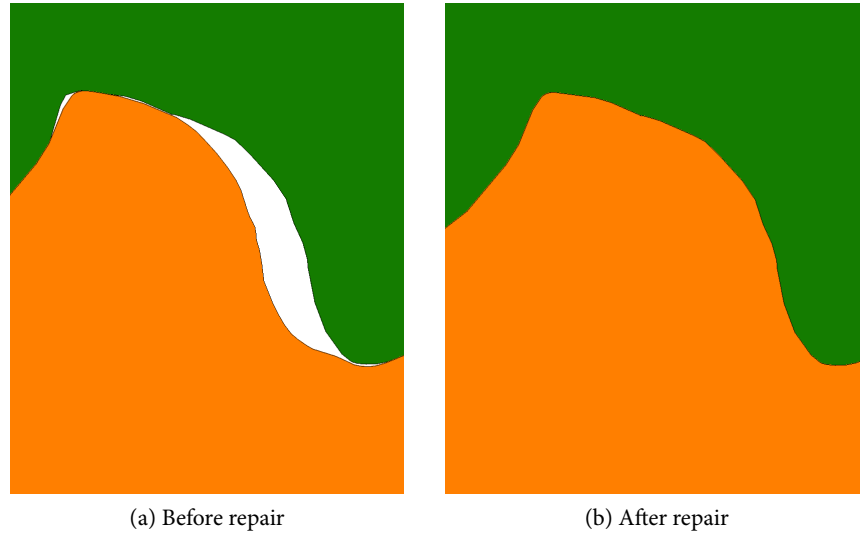


Figure 2.8: A region of the polygons for the Arribes del Duero Natural Park in Spain (orange) and the International Douro Natural Park in Portugal (green). Since the border is traced along a river, data does not match perfectly.

using a pre-defined list of priorities among feature classes), for which the algorithms developed are very well suited, since snapping thresholds might be too large in other approaches (see Section 3.3).

The most common solution presented to have harmonised data sets is based on the dubious notion of *reference data*, which assumes that there is a data set which is the best according to some metric. Other data sets are then snapped to it, using the techniques which will be discussed in Section 3.3. However, even assuming that there is a certain data set which is best, it is undesirable to harmonise data in this manner due to several reasons:

- As new better reference data is created, all other data sets have to be re-processed. This is due to the fact that the main characteristic of reference data is its high accuracy, and higher accuracy data is almost always expected to be available in the future.
- Order dependency might be introduced, and errors accumulate as data is re-snapped to a new reference data set (i.e. data points drifting).
- When using data sets that do not include the reference data, an additional and unnecessary source of error is introduced.
- Threshold values involve a clear-cut decision for snapping, which is usually not the case. Some mismatching features might be erroneously snapped, while others representing the same points might not be.

Nevertheless, this solution is still very common due to the notion that it is very computationally expensive to harmonise data sets, often requiring an iterative process [Lupien, 1987]. However, using a planar partition for this purpose is fast enough to be considered for doing this on-the-fly, since only the data sets being currently used would need to be processed. This would be preferable in most situations, and fits better with data interoperability policies such as INSPIRE.

2.5 PLANAR PARTITIONS

Planar partitions are a natural representation for many types of data, generally being top-down views of features which their only practical requirement is to be able to be clearly divided into different classes. These classes should be mutually exclusive, although each of them can represent a set of conditions (i.e. a class that recursively implies membership to a set of other classes).

In the strict sense, a planar partition comprises a single joint region on the plane, without any gaps or overlaps. However, in real-life data sets, there might be several disjoint regions, while some holes could be allowed. For instance, in the CORINE 2000 data set, islands are valid disjoint regions, while some areas without coverage (i.e. parts of Andorra, Switzerland and Kosovo) are surrounded by areas with coverage, creating holes. At the same time, there are gaps between some tiles that cause undesirable disjoint regions, which should be fixed. An example of this is shown in Figure 2.9.



Figure 2.9: Two contiguous tiles from the CORINE 2000 data set, separated by a small distance. Notice how a dividing line between two features matches well on both sides.

Since many of these situations are topologically equivalent, it becomes impossible to distinguish them without some external information. However, it is possible to make some generalised assumptions:

- Overlaps are never allowed and should be repaired.
- Holes are generally not allowed, since they are commonly errors in the data. However, it should be possible to insert allowed holes that should not be filled by a repair algorithm.

- Disjoint regions are allowed, since there are many cases where they can occur (e.g. islands and exclaves). However, it should be possible to know the amount of regions and get their boundaries.

It is important to note that this is only a subset of the constraints that a planar partition might be expected to fulfil. However, this does not mean that they are incomplete. Other constraints are dealt with at the polygon level, as discussed in Section 2.3, while the rest are a result of the triangulation, which will be discussed in the following section. For a complete axiomatisation of these conditions, see Plümer and Gröger [1997].

2.6 THE CONSTRAINED DELAUNAY TRIANGULATION

A triangulation is, in its simplest form, the subdivision of a geometric object into triangles. Throughout this thesis, two basic types of triangulations are extensively used, shown in Figure 2.10, and described as follows:

Point Set Triangulation A set of points on the plane are triangulated by joining them with a maximal set of straight line segments which intersect only at their end points [Lloyd, 1977]. This results in the triangulation of the *convex hull*¹⁰ of the point set [Deza and Rosenberg, 1980]. Outside the convex hull there is a single face, which is the only one that is not a triangle and is unbounded. This is known as the *infinite face* of the triangulation.

Polygon Triangulation A polygon is triangulated by creating line segments between the vertices of the polygon without passing through its exterior (diagonals), which subdivide the polygon into triangles [O'Rourke, 1998, chap. 1]. Usually the convex hull of the polygon is triangulated instead¹¹, since its better for efficiency reasons [Fournier and Montuno, 1984]. Outside the convex hull also lies the infinite face of the triangulation.

It is important to note that these two cases can be made completely equivalent by assuming certain triangulation rules (i.e. using criteria to decide which edges are to be generated and triangulating always the entire convex hull of a polygon.). Therefore, it is possible to add both vertices and edges interchangeably to a triangulated point set, and later on go back to a polygon, assuming that the information regarding which triangles are part of the interior of the polygon is kept.

Meanwhile, the inclusion of the infinite face concept brings many implementation advantages, since it is possible to traverse the triangulation from a face known to be in the exterior of a polygon. This will prove very useful for the algorithms developed, as discussed in Sections 4.1 and 4.2. The creation of this infinite face is usually done through the concept of a “big triangle” or “far-away point” [Facello, 1995; Liu and Snoeyink, 2006].

¹⁰Defined as the smallest convex set that contains all points in the point set [de Berg et al., 2008, chap. 1].

¹¹As opposed to only the interior of the polygon.

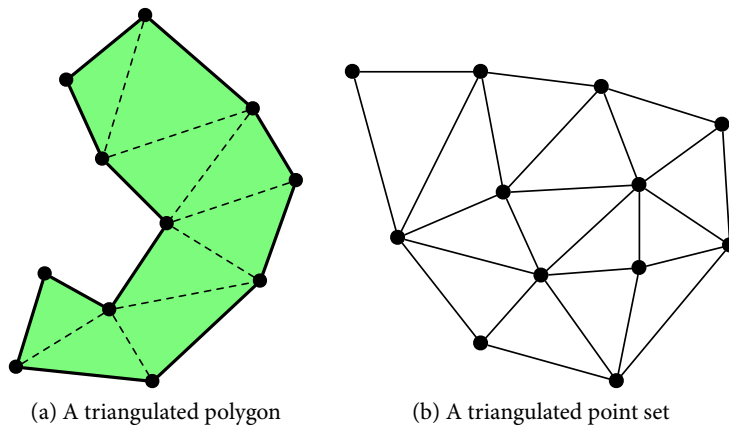


Figure 2.10: Two types of triangulations.

A polygon can always be triangulated [de Berg et al., 2008, chap. 3]. However, a triangulation of a point set or polygon is not unique, as illustrated in Figures 2.11 and 2.12. This is an already undesirable property, and it is important to note that there are triangulations that are better suited than others for most applications. In general, the length of their edges should be as even as possible, since thin and elongated triangles are a problem for many algorithms. Because of this, it is preferable to have a triangulation which is unique and maximises the minimum angles of a triangulation, which is the *Delaunay triangulation* of a point set¹², in which the circumscribing circle of any three vertices forming a triangle has no other points in its interior (i.e. the empty circle or Delaunay property).

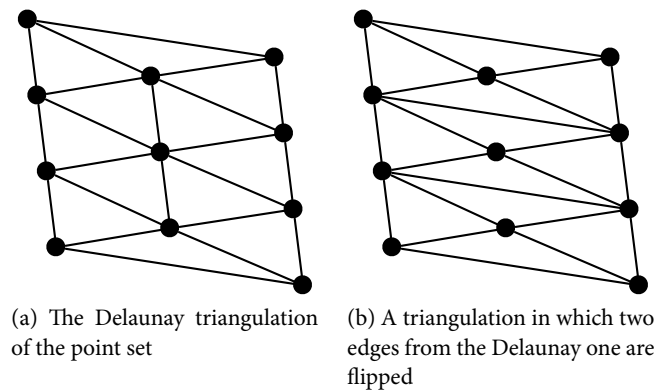


Figure 2.11: A point set triangulated in two different ways.

¹²The Delaunay triangulation is not unique when there are four or more co-circular points in the set, although it can be made unique by assuming certain triangulation rules.

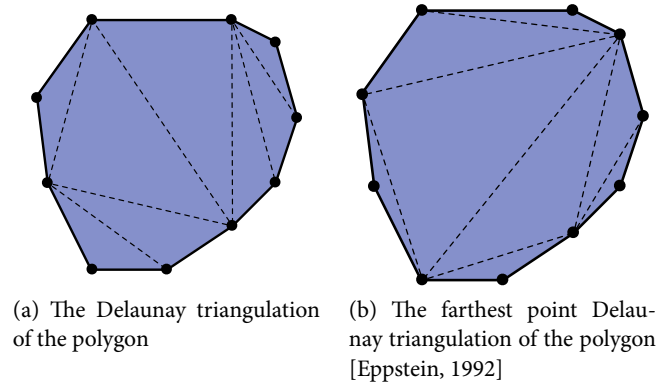


Figure 2.12: A polygon triangulated in two different ways.

Another type of triangulations which are relevant to this thesis are *constrained triangulations*. These have the added requirement that they must contain certain input edges in the triangulation, which are known as *constrained edges*. While a constrained triangulation is generally not unique, and most of them are not Delaunay, it is possible to have a so-called constrained Delaunay triangulation. This is also generally unique¹³ and fulfils a weaker property, the constrained empty circle [see CGAL, 2010]¹⁴.

Using a triangulation as a base for validation and repair has many good properties, including:

- It is, by definition, a planar partition. Therefore, as long as the information relating to which polygon each triangle belongs to is kept, the polygons reconstructed from it will be a valid planar partition.
- Point location queries are very fast [Devillers et al., 2002], especially when using a triangulation hierarchy [Devillers, 2002]. This is important, since triangulations are built through the on-line insertion of vertices [Boissonnat et al., 2002].
- Changes to the triangulation (e.g. adding a new constrained edge) are local, and therefore fast.
- Constrained edges can usually be added in constant time, being only significantly slower when there is an intersection with an existing constrained edge.
- Implementation-wise, several stable and fast triangulation libraries exist, including CGAL [CGAL, 2010], Triangle [Shewchuck, 1996] and GTS [GTS, 2006].

The combination of these factors, allows the algorithms developed to retain good performance characteristics with the very large triangulations that are created when

¹³The constrained Delaunay triangulation may not be unique when there are four co-circular points.

¹⁴The empty circle property can however be maintained by using a *conforming* triangulation instead [Hansen and Levin, 1991], at the expense of adding more points to the triangulation.

triangulating large planar partitions, of which an example from two adjacent tiles of CORINE is shown in Figure 2.13.

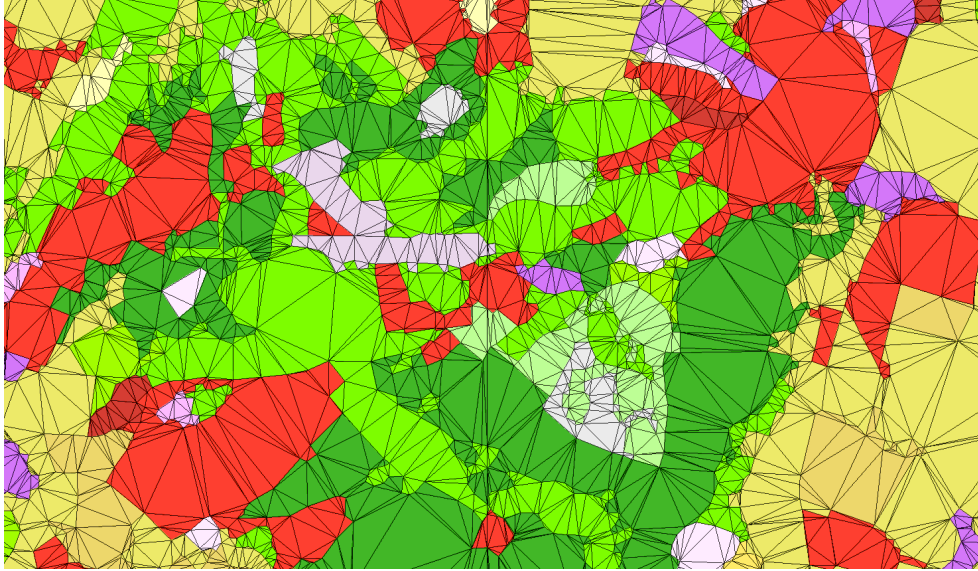


Figure 2.13: A region of the triangulation of the polygons in CORINE tiles E39N32 and E40N32, which is now, by definition, a planar partition. The triangles that are in the small region with an overlap between the two tiles are now not repeated, and are marked instead with *a single tag comprising a set of regions*; as opposed to having two sets of triangles, each of them with different tags.

State of the Art in Planar Partition Validation and Repair

Most current approaches to planar partition validation and repair deal with geometry and topology in a few separate semi-automated processes that are almost completely independent from each other. This has important consequences, which will be discussed in each relevant section.

When the tools allow for it, individual polygons are checked first (either while the input is being read or afterwards), and it is assumed that these are valid before attempting to validate a planar partition. To validate (and possibly repair) these polygons, a list of constraints is used, as is discussed in Section 3.1. Such a list may also contain the actions that should be taken when invalid cases are found.

Afterwards, an entire planar partition is validated and repaired. While validation is (loosely) based on the same concepts in all the tools studied, and therefore discussed together in Section 3.2; repair is quite different, and three main approaches have been identified. These are:

Vertex, edge and face snapping and splitting Using tolerance values, points are snapped to points, points with edges, and edges with other edges (e.g. when the angle between them is too small), among other variations. These primitives are then split when a new vertex is generated along its boundary. This method, which is available in most of the tools reviewed, is discussed in Section 3.3.

Using topological information Based on the creation of a graph-based representation of the planar partition, which is able to identify gaps and overlaps, these areas are assigned based on certain criteria. It is somewhat similar to the approach developed in this thesis. This method, only fully available in ArcGIS, is presented in Section 3.4.

Topologically through database operations Some tools (i.e. Oracle and Radius Topology, and GRASS) are based on a database storing topological information, which

can be used to repair a planar partition by either of the two methods described above. Among these, snapping is already implemented and works no different than in other software; this being the case for both Radius Topology and GRASS. However, while the main data structures for automated topological repair already exist in both as well, other required operations do not, and would have to be implemented by the user. This would involve a significant amount of work from a user, but it is certainly possible, so it is only discussed briefly in Section 3.5.

3.1 VALIDATION AND REPAIR OF INDIVIDUAL POLYGONS

The geometry of individual polygons is generally validated by using a list of constraints, which are successively checked for every polygon. In some cases these constraints are disjoint¹ and independent from each other, while in others they are interrelated or build on the previous ones' requirements.

While the constraints checked in each solution are notably different, and in some cases, not completely unambiguous in their requirements; it is possible to get a good overview of the types of checks involved by examining two representative examples: ArcGIS, which checks for mainly geometric characteristics which are tightly connected to the Shapefile specification and their internal geometry model; and the one from JTS/GEOS, which follows more of a topological approach. These are respectively summarised in Tables 3.1 and 3.2, with their non-applicable (e.g. semantic or formatting related) constraints removed. For comparison and later reference, the constraints used in the Oracle database are presented in Table 3.3, which provides validation, but only basic repair tools, since many errors cannot be repaired without extensive user intervention to modify the data.

Meanwhile, GRASS, FME and Radius Topology do not provide additional validation or repair operations *specifically* meant for individual polygons, so they are not further covered in this section². However, the operations that they provide for groups of polygons can be applied to individual polygons as well, with mixed results. These will be discussed in Section 3.2, together with their application for planar partitions.

In the aforementioned tables, it is possible to see that some of the checks involved are equivalent or similar (e.g. intersection tests), but there is no consensus on the operations that need to be done to ensure valid polygons. ArcGIS checks and corrects for closed polygons, unlike JTS/GEOS. However, JTS/GEOS is the only one that makes an effort to keep polygons interior connected³. This is especially true regarding the operations meant to avoid precision and robustness problems, as defined in Section 2.3.

¹Meaning that they check for errors which are not interdependent (i.e. each constraint can be triggered individually by a certain invalid configuration).

²Other than that already provided by JTS/GEOS, in the case of GRASS and FME; and that of the Oracle database, in the case of Radius Topology.

³Although in ArcGIS this might be less necessary, since according to the Shapefile specification touching rings could be modelled as a single ring [ESRI, 1998].

Table 3.1: Relevant constraints for polygon validation and repair in ArcGIS [ESRI, 2009a].

Constraint	Description
Short segments	At least one line segment is shorter than the unit in the spatial reference associated. These are deleted during repair.
Unclosed rings	The last line segment of a ring does not end at its starting point. These are closed by connecting their end points during repair.
Self-intersections	Rings touch or intersect with themselves or other rings in the polygon (not simple). These are split at their intersections during repair.
Incorrect ring ordering	A polygon is simple but at least one of its rings are incorrectly oriented. Their orientation is corrected during repair.

Table 3.2: Relevant constraints for geometry cleaning in JTS and GEOS [GEOS, 2010].

Constraint	Description
Self-touching rings	Polygon rings must not self-touch. Disconnected components are separated and those with too small areas are removed.
Zero area rings	Rings should not have zero area. These should be removed.
Zero area polygons	Polygons should not have zero area. These should be removed.
Properly nested rings	Rings should not intersect and only touch once. Zero width dangling edges are removed.
Duplicate vertices	Rings should not have duplicate vertices or vertices within a tolerance of each other.
Spikes and gores	Rings should not have very close and almost parallel line segments.
Touching parts	Multi polygons should not have parts that touch along an edge. The edges between them are dissolved.
Crossing rings	Polygons should not have crossing rings.

Table 3.3: Relevant constraints for geometry validation in Oracle Spatial [Oracle, 2009a,b, 2010].

Constraint	Description
ORA 13343	Polygon with fewer than four coordinates.
ORA 13348	Ring not closed.
ORA 13349	Self-intersection. This can be fixed with the routine SDO_UTIL.RECTIFY_GEOMETRY.
ORA 13350	Touching rings.
ORA 13351	Overlapping rings.
ORA 13356	Adjacent points are closer than a tolerance value. The routine SDO_UTIL.REMOVE_DUPLICATE_VERTICES may be used to remove them.
ORA 13367	Wrong orientation. This can be fixed with the routine SDO_UTIL.RECTIFY_GEOMETRY.

ArcGIS has tests to ensure that consecutive vertices on a polygon are not too close to each other, while JTS/GEOS extends these to non consecutive ones as well.

Moreover, while these checks cover most of the commonly found cases, it is worth noting that no implementation among the ones examined clearly states that they are able to detect *every* incorrect configuration. For example, ArcGIS has no capabilities to remove duplicate vertices.

Also, it is very hard to guarantee that every degenerate case (of which there are many) has been taken into account. For instance, ArcGIS is unable to detect any error in a polygon without an interior, due to it having a hole of its exact size and shape, unless this hole is subdivided into two or more smaller holes (Figure 3.1), despite the fact that this is a self-intersection, which ArcGIS tests for, and it is also a zero area feature, and thus disallowed in the Shapefile specification [ESRI, 1998].

In fact, since the classification of incorrect configurations is very dependent on the implementation (as it can be seen from the previous examples), the cases that are degenerate and may not be correctly handled also vary, which complicates matters further. Therefore for this thesis a comprehensive set of test polygons has been created for testing purposes, which are presented together with their interpretations in different implementations in Section 5.2.

Now, when considering repair as well, there are a couple other aspects to consider: the introduction of order dependency into the repair operations, and the possibility of breaking edge matching (e.g. by creating features that match only approximately, not exactly) and topological constraints when the repair operations themselves are used.

Order dependency stems from the fact that the lists of constraints used for polygon validation and repair have an inherent order in them, causing unpredictability in the

3.2. Topological Validation of Planar Partitions

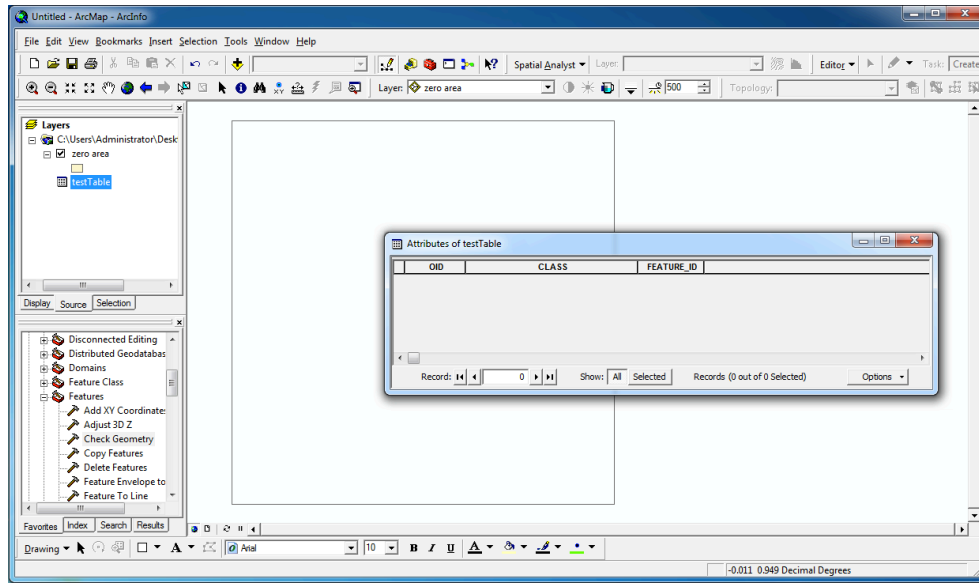


Figure 3.1: ArcGIS is unable to detect any error in a zero area polygon, due to it having a hole that is identical to its outer boundary (but with opposite orientation). The main window shows this polygon (a square), while the smaller window shows a table with the errors found in it (none).

repair operations⁴.

Meanwhile, breaking edge-matching is more complex and can occur due to several reasons. Some examples include: moving vertices beyond a certain matching threshold (discussed in Section 3.3), removal of features which would match others (e.g. removing a hole that would have provided the space for a different polygon), and splitting features that should have been kept together, among others. JTS/GEOS solves this by snapping edges together [GEOS, 2010], a common approach in many of the tools studied. This technique will be examined in Section 3.3.

3.2 TOPOLOGICAL VALIDATION OF PLANAR PARTITIONS

Assuming that individual polygons have been deemed to be valid, it is possible to test the validity of a planar partition by identifying the two types of invalid configurations: gaps and overlaps. However, unlike for individual polygons, using stored topological relationships between different polygons is necessary for the efficient detection of both.

Otherwise, in order to find overlaps without additional topological information, it is necessary to check whether any possible pair of polygons are disjoint (i.e. when their interiors do not overlap) or not. This is already a computationally expensive operation

⁴Or in the worst case, recursive generation of errors (i.e. error a is fixed in a procedure that creates a new error b , which is fixed in a procedure which creates a again), thus creating an infinite loop.

to make, but especially so since many possible pairs have to be checked⁵, even when using heuristics to speed up the process [Badawy and Aref, 1999; Kirkpatrick et al., 2000]. Additionally, robustness issues are an important issue in polygon intersection tests [Hoffmann et al., 1988].

Even more problematic is finding holes without topological information. For this, the computation of the union of the entire set of polygons is required; which is also computationally expensive for the large sets of polygons with holes that are dealt with in planar partitions [Margalit and Knott, 1989; Rivero and Feito, 2000]⁶.

Due to these difficulties, the addition of topology is highly desirable in order to do planar partition validation. Namely, adjacency relationships for each polygon are required, which take the form of a planar graph⁷. Based on this graph, Plümer and Gröger [1996, 1997] specify a set of mathematical axioms that can be used to check the validity of a map. In all of the tools studied where planar partition validation is possible, with the exception of GRASS, a procedure similar to this would be followed in order to do planar partition validation.

An adapted set of the aforementioned axioms is presented in Table 3.4, based on the assumption that in the transformation from a geometric model to a topological one, points with the (exact or approximate) same coordinates were united in a single vertex. Such a list of axioms can be individually checked in the graph, giving the information of whether the planar partition is valid or not.

Table 3.4: Minimal map axioms for planar partition validation, adapted from Plümer and Gröger [1996].

Axiom	Description
No dangling edges	Each vertex has at least two incident edges
No zero-length edges	Each edge has two distinct vertices as end points
Planarity	Edges do not intersect, except at their end points
No holes	Each edge has exactly two incident faces
No self-intersections	Each face has exactly one simple circuit as window
No overlaps	No part of the interior of an edge lies in the interior of a face
Connectivity	The graph is connected

In GRASS, the situation is different since it is a topological GIS, and therefore topology is directly generated as features are read. Theoretically, this makes detecting gaps and overlaps much easier. This is definitely true for overlaps, as shown in Figure 3.2.

⁵In the worst case, a brute force $O(n^2)$ operation, where n is the number of polygons to be checked.

⁶However, optimisations exist for some special case polygons [van Kreveld, 1998], and many algorithms work by first decomposing the polygon into convex parts [Chazelle and Dobkin, 1979].

⁷A valid planar partitions represents, by consequence, a planar graph. However, the opposite does not hold true, e.g. due to dangling edges.

3.3. Planar Partition Repair Using Vertex, Edge and Face Snapping and Splitting

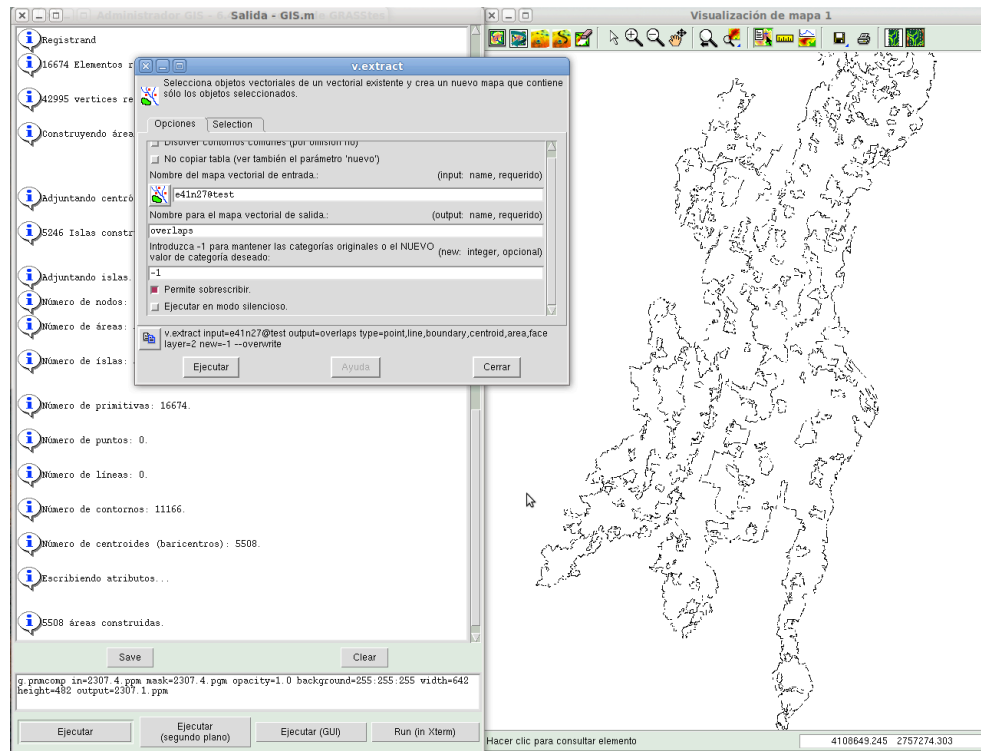


Figure 3.2: Since GRASS automatically detects overlapping polygons as they are read, they are put into different layers. Shown is layer 2 from CORINE tile E41N27, with a polygon that overlaps others.

3.3 PLANAR PARTITION REPAIR USING VERTEX, EDGE AND FACE SNAPPING AND SPLITTING

The most common method for planar partition repair is based on the concept that polygons *approximately* match each other at their common boundaries. This implies that they should always be within a certain distance of each other along those edges. If, additionally, all parts farther apart than this value are known not to be common boundaries, it is possible to “snap” together polygons that are closer to each other than this threshold, while keeping the rest untouched. This method of planar partition repair is available in many tools, including ArcGIS, FME, GRASS and Radius Topology.

Since thresholds are central to this method, it is of utmost importance to select a good threshold value, something that is completely different in each data set. For planar partition repair to be successful using this method, such a threshold should be chosen in a careful manner, and always comply with a few conditions. These have been summarised as follows:

- Adjacent polygons should not be farther apart than this threshold along any part of their common boundaries (shown as the maximum threshold in Figure 3.3a).

Otherwise, gaps are not able to be fixed.

- Adjacent polygons should not overlap each other in areas which are further inwards than this threshold from their common boundaries (shown as the maximum threshold in Figure 3.3b). Otherwise, overlaps are not able to be fixed.
- No vertices of a polygon should be closer to each other than this threshold, including non consecutive vertices (shown as the minimum thresholds in Figure 3.3). Otherwise, they might be snapped together, creating repeated vertices, disjoint regions, or various topological problems.
- No vertices of a polygon should be closer than this threshold to any non incident edge. Otherwise, they might be snapped together, creating disjoint regions or various topological problems.

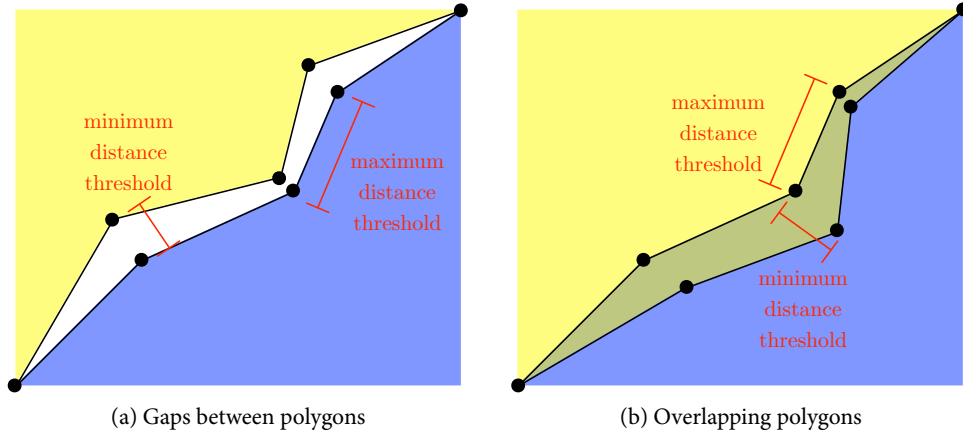


Figure 3.3: Defining a threshold for vertex, edge and face snapping. The threshold to use should be larger than the largest minimum distance between the matching boundaries, and smaller than the minimum distance between vertices.

This threshold value is usually manually determined, either by trial and error, or by analysing certain properties of the data set(s) involved (e.g. point spacing, point precision, sampling rates or internal geometry models). However, it is often hard to find an optimal threshold for a certain data set, since the sheer size of the data sets used makes ensuring that it works well for every part of the data set unrealistic. Moreover, sometimes such a threshold does not even exist (e.g. because point spacing might be in some places smaller than the width of the gaps and overlaps present).

Since the aforementioned conditions are frequently not met, or are not checked beforehand, and it is still necessary to perform repair of a data set, snapping might be performed nevertheless, possibly creating invalid polygons or planar partitions, or significantly changing the topology of the existing features. Two examples of this phenomenon are shown in Figures 3.4 and 3.5.

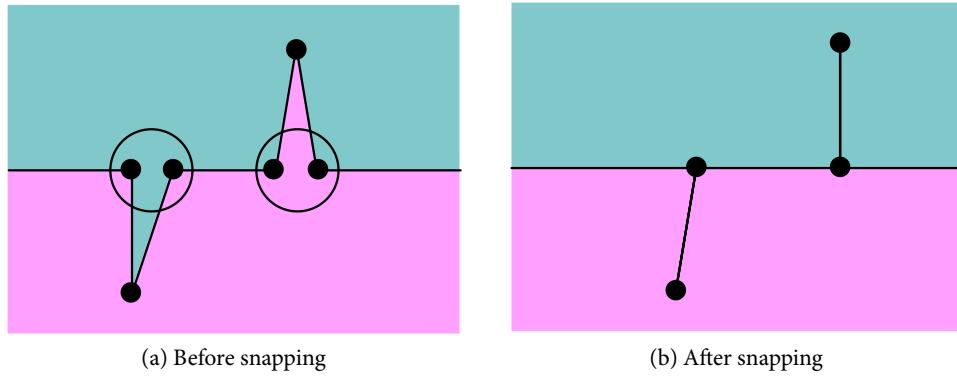


Figure 3.4: Spikes and punctures can be created by snapping, since the bases of these elongated forms (encircled) might be narrower than the threshold, but its length not.

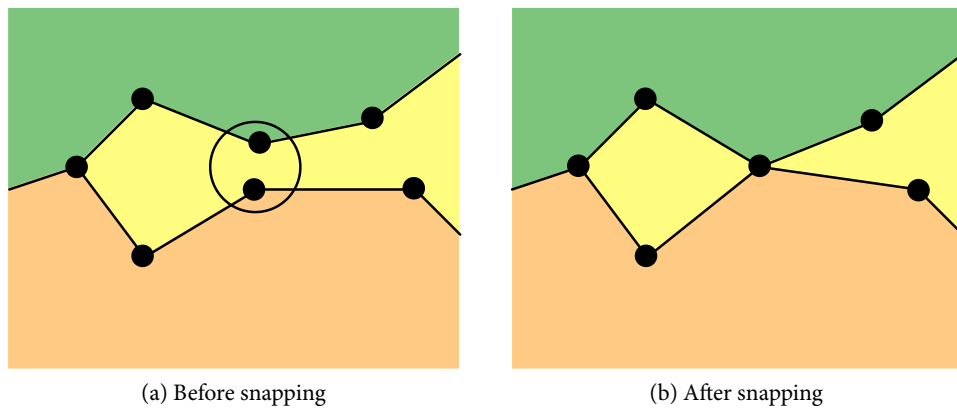


Figure 3.5: Polygons can be split by snapping, since some parts of them might be narrower than the threshold (encircled). While this result does not create an invalid planar partition, it can change the number of polygons present and their topological relations, and can therefore be undesirable.

However, if a threshold values exists and snapping is done carefully, it is possible to generate perfectly valid planar partitions in a simple manner. Figure 3.6 shows an example of 4 CORINE tiles put together with the FME transformers shown in Figure 3.7.

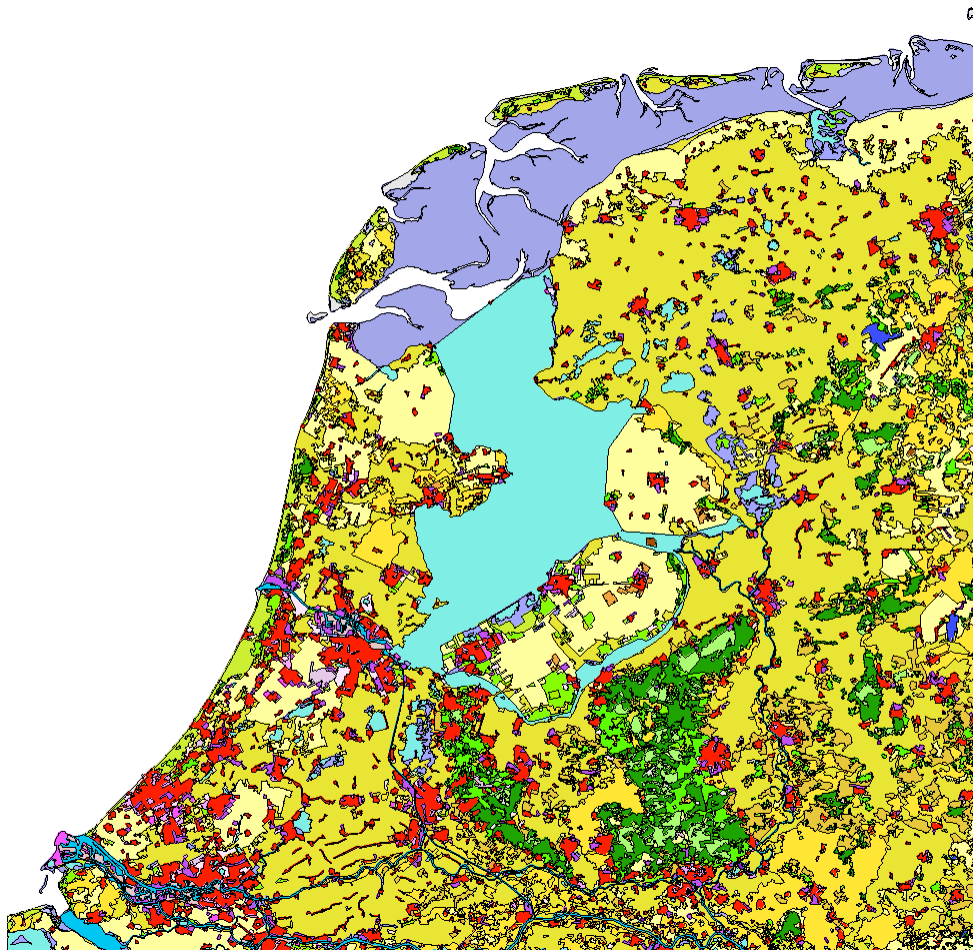


Figure 3.6: Four adjacent tiles (2×2) from CORINE joined and repaired in FME, using the transformers from Figure 3.7. Notice that there are no seams at the former tile boundaries.

While the examples shown previously prove that this repair method is not problem-free, until this point only simplified snapping rules have been considered. In practice, one finds very different snapping criteria and more complex situations. To illustrate this, the list of constraints for planar partition repair in GRASS are summarised in Table 3.5, while some notable complications are listed as follows:

- All intersecting edges are split at their intersection point in some implementations, adding new vertices at those places, which in turn reduces vertex spacing

3.3. Planar Partition Repair Using Vertex, Edge and Face Snapping and Splitting

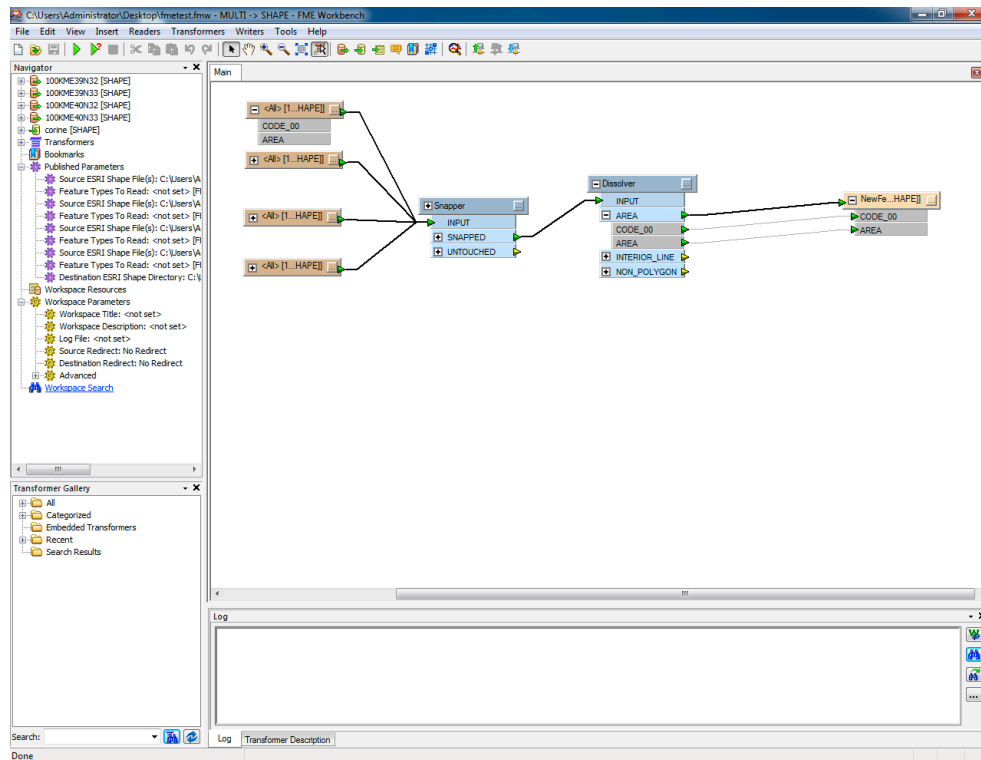


Figure 3.7: Joining and repairing 4 adjacent tiles from CORINE in FME, using the transformers shown above. The process consists of reading the 4 input files, snapping them together, dissolving the boundaries that have the same feature classification on both sides, and creating a unified output. Note that this involves checking boundaries within each CORINE tile as well, which might be unnecessary.

and might cause these vertices to be snapped together.

- Different types of snapping exist (e.g. point to point, point to edge and edge to edge; simplex and complex features; or using reference data), which can have individually set threshold values [MacDonald, 2001a].
- Post-processing operations to clean a planar partition might be required, which can introduce substantial changes to it. Examples include: disposing of polygons with small areas, removing redundant lines, thresholds for minimum angles (for robustness, removing spikes), etc. A good example for this is GRASS (Table 3.5), in contrast with Radius Topology, which has simpler constraints (Table 3.6). These might create new gaps and overlaps themselves, requiring an iterative cleaning process.
- The geometry of the new polygons may have to be reconstructed in a topologically correct manner [Barker, 1995]. For instance, the polygons in Figure 3.4

should have the spikes/punctures removed, and the yellow polygon in Figure 3.5 should be split into two.

- Snapping is an intricate problem in itself, since there are many possible criteria that can be followed, for both points and edges (e.g. points to the closest line, points to the closest point, points orthogonally to the closest line). All of these can have different problematic consequences, as the example in Figure 3.8 shows.

Table 3.5: Relevant constraints for planar partition repair in GRASS [GRASS, 2006].

Constraint	Description
break	Line segments are broken at their intersections.
rmdupl	Removes duplicate line segments.
rmdangle	Removes dangling edges.
rmbridge	Removes bridges connecting holes or islands.
snap	Line segments are snapped to vertices within a threshold.
prune	Removes vertices within a threshold of line segments.
rmarea	Removes areas smaller than a threshold.
rmsa	Removes small angles between lines at vertices.

Table 3.6: Relevant constraints for planar partition repair in Radius Topology [Baars, 2003; Louwsma, 2003].

Constraint	Description
SHARE_NODE	Nodes within a threshold are snapped together.
NODE_SPLIT_EDGE	Edges within a distance from a node are snapped to it, creating an intermediate node.
EDGE_SPLIT_EDGE	Intersecting edges or edges within a distance from other edges are snapped together.

Finally, it is worth mentioning that although all planar partitions *could* be repaired by snapping and splitting polygons; it might require the use of thresholds so large so as to have no physical basis, and create planar partitions that are substantially different from the original data.

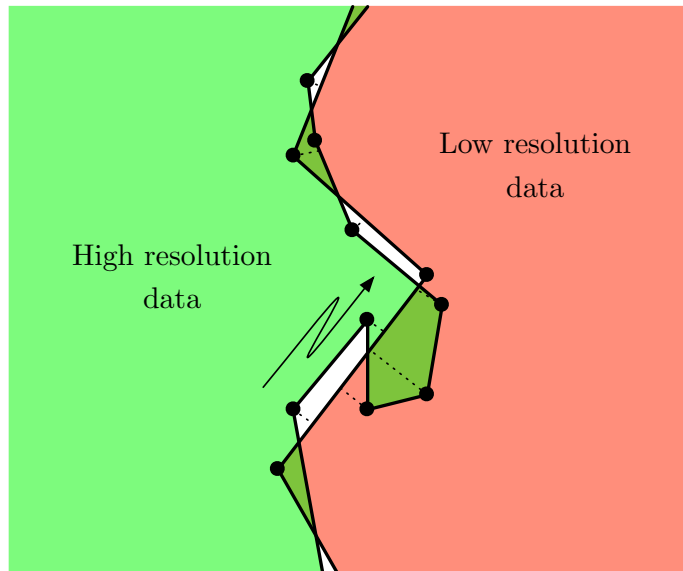


Figure 3.8: Snapping to the closest line can cause topologically invalid configurations. When two data sets of differing levels of detail are joined together by snapping the vertices of the high resolution data set to the edges of the low resolution one, a situation where the line reverses on itself is created.

3.4 PLANAR PARTITION REPAIR USING TOPOLOGICAL INFORMATION

A different approach for planar partition repair, mostly based on (internally used) topological information, is available in ArcGIS⁸. This method is therefore similar in some ways to the one developed during this thesis, although ArcGIS requires user intervention to make a decision to fix each invalid region individually and is based on polygonal regions, rather than triangles.

For this, it is necessary to import all the polygons into the geodatabase feature of the software, which allows for topology based validation rules to be entered as well [ESRI, 2002], as shown in Figure 3.9. The most relevant rules for planar partition repair, which all work on any number of polygons, are summarised as follows⁹:

Must Not Overlap The interior of any of the polygons entered must not overlap. Vertices and edges can still be shared.

Must Not Have Gaps The polygons must not have voids within themselves or between adjacent polygons. Polygons can still share vertices, edges or interior areas.

⁸The supporting data structures are available in GRASS as well, and possibly in FME, but the operations required are not.

⁹Some other rules might be useful as well (e.g. Area boundary must be covered by boundary of, boundary must be covered by, tessellate, etc.). All the rules are available in MacDonald [2001b].

3. STATE OF THE ART IN PLANAR PARTITION VALIDATION AND REPAIR

Must Not Overlap With The interior of the polygons in one layer must not overlap with interiors of the polygons in another.

Must Not Have Dangles Lines must touch other lines at their end points. This can take care of overshoots and undershoots (Figure 3.10).

Must Not Have Pseudonodes Lines must connect to at least two other lines at each of their end points. This can be used to remove unnecessary nodes, such as one situated at the begin and end point of a looping edge.

Must Not Self Intersect Line features must not cross themselves. This can be used to remove spikes, as long as a suitable precision value is used first.

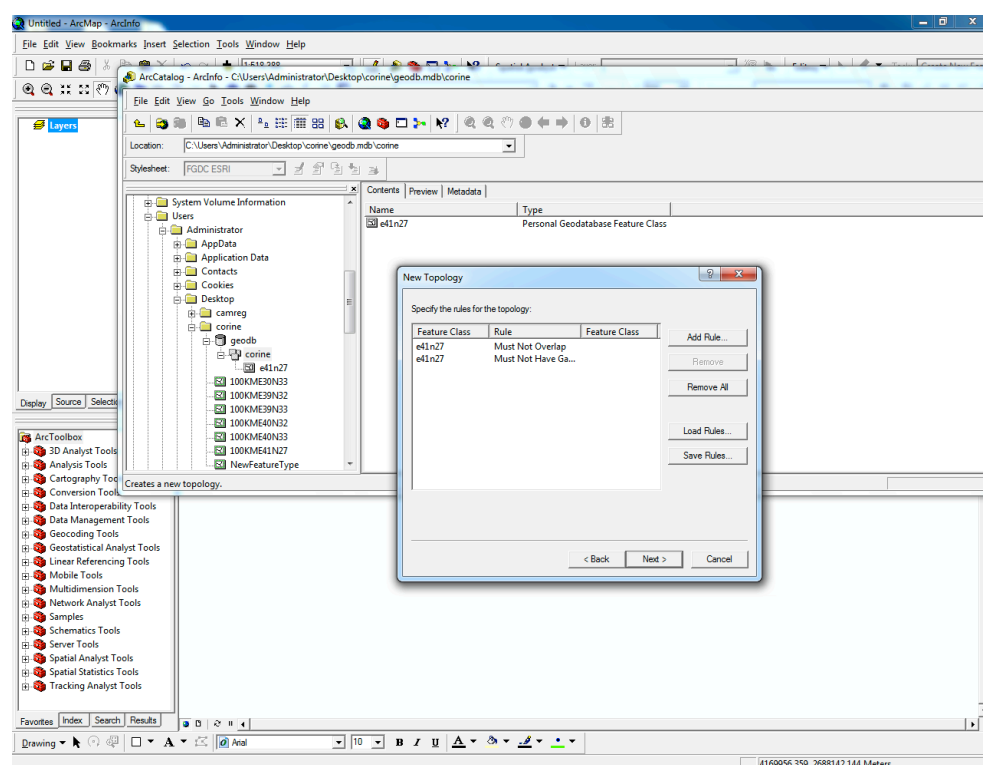


Figure 3.9: Entering topology rules in ArcGIS. For this example, a CORINE tile with known problems (E41N27) is put into the Geodatabase, together with two simple validation rules: must not overlap and must not have gaps.

Regions on a map that do not fulfil the properties of a valid planar partition can be identified by a combination of the rules above, as in [Stanton et al., 2005] or [Wahl, 2004]. The simplest case would be to only use the first two rules. Keeping to the previous CORINE example, Figure 3.11 shows the regions that fail to comply with the validation rules imposed.

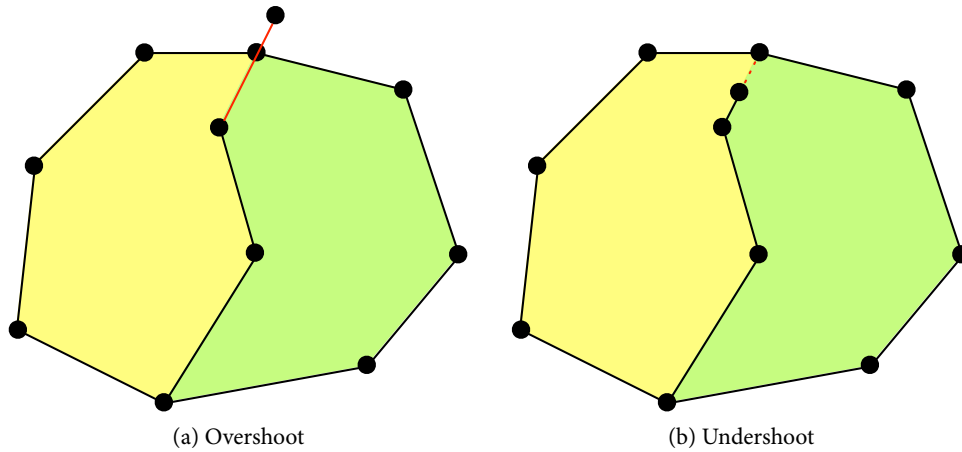


Figure 3.10: Overshoots and undershoots are invalid configurations that break polygon topology. These are usually solved by snapping (Section 3.3).

The screenshot shows the 'Topology Properties' dialog box with the 'Errors' tab selected. It displays a summary of topology errors for CORINE tile E41N27. The table lists three rules: 'Must Be Larger Than Cluster Tolerance', 'Must Not Have Gaps', and 'Must Not Overlap'. The 'Total' row shows 10892 errors and 0 exceptions.

Rule	Errors	Exceptions
Must Be Larger Than Cluster Tolerance	0	0
Must Not Have Gaps e41n27	5384	0
Must Not Overlap e41n27	5508	0
Total	10892	0

Figure 3.11: Topology errors found in CORINE tile E41N27 with ArcGIS.

3. STATE OF THE ART IN PLANAR PARTITION VALIDATION AND REPAIR

Validating planar partitions with this method can be therefore done efficiently in ArcGIS. Also, it is quite a robust implementation, since points are initially snapped to a given tolerance [ESRI, 2009b]. This however comes with the disadvantage that points will be moved during processing.

Meanwhile, the repair tools in ArcGIS require extensive user interaction. When acting with multiple invalid polygonal regions, the only options available that can be applied to all simultaneously are: deleting everything in the region (in the case of overlaps), or creating new polygons from those regions. None of these makes sense in the context of planar partition repair. Deleting everything is especially troublesome, since it creates new holes, which are again invalid regions.

However, it is possible to manually repair a planar partition using ArcGIS. The procedure would then be to zoom in to every error present using the Error Inspector (Figure 3.12a), manually decide on a choice to be made, and assign the region to a certain polygon (Figure 3.12b). This is an unfeasible solution for the very large data sets common in planar partitions. Figure 3.13 shows the large number of different errors present in a small region in the CORINE tile from previous examples. There are almost 11,000 total errors present in this tile¹⁰.

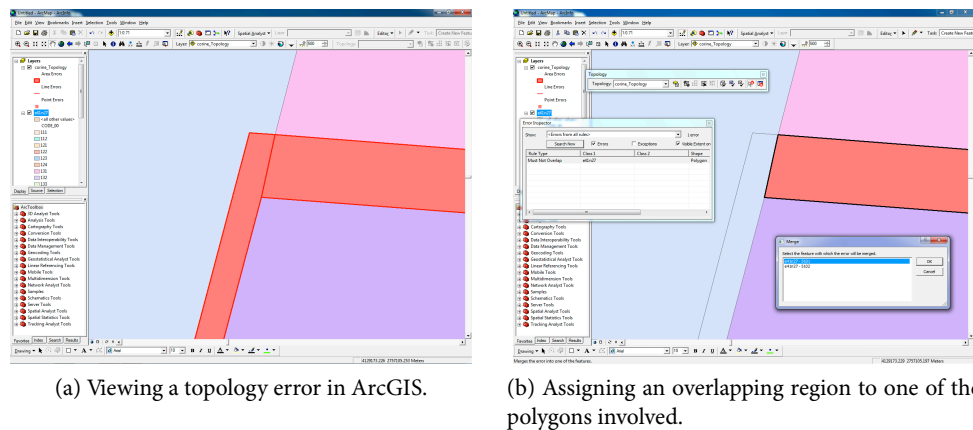


Figure 3.12: Planar partition repair in ArcGIS. The user is expected to zoom in to a particular error, analyse the situation (e.g. by looking at the properties from the surrounding polygons), and make a decision to assign the problematic region to a certain polygon.

3.5 TOPOLOGICAL PLANAR PARTITION REPAIR USING A DATABASE

Spatial databases with topological information are an interesting case, since they are able to do planar partition repair using both of the aforementioned methods. However,

¹⁰Using the precision threshold specified in the original Shapefiles.

3.5. Topological Planar Partition Repair Using A Database

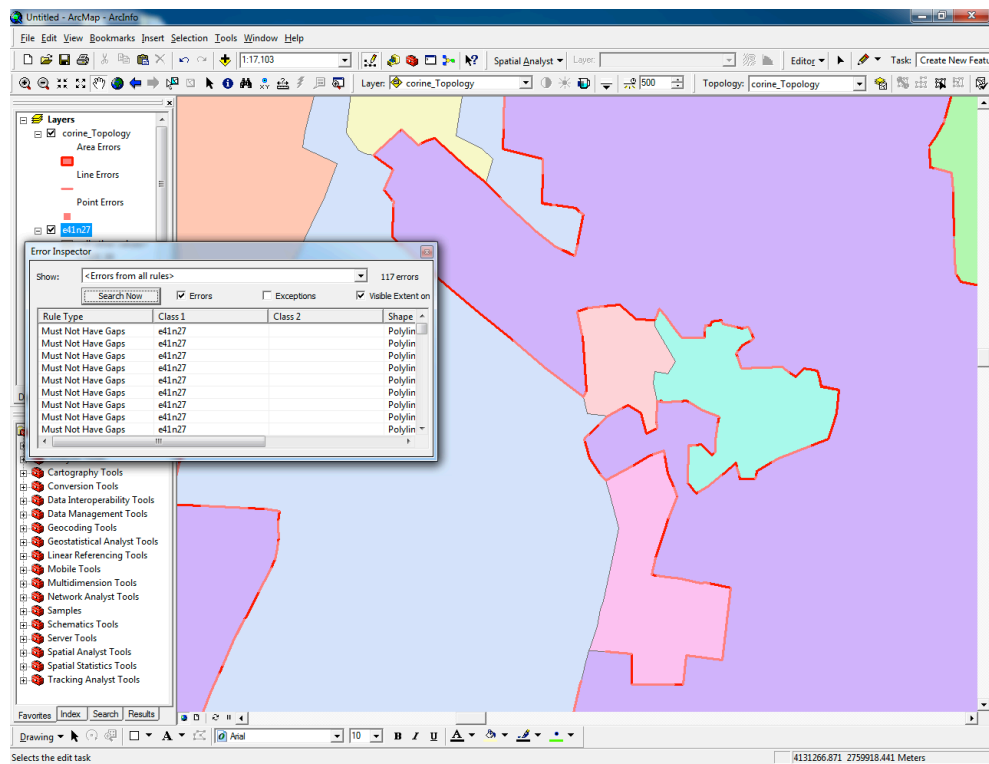


Figure 3.13: Topology errors found in only a small region from CORINE tile E41N27. Gaps are shown in pink, while overlaps in red. There are 117 errors in this region, 1% of the total of this tile.

while snapping based repair is already implemented and working in both of the solutions examined (Oracle with Radius Topology and GRASS when used together with a database), topological repair is not.

Nevertheless, the main required data structures and basic operators for topology based repair already exist, including checking the number of polygons at a certain point (0 being a hole or in the exterior, 2 or more being an overlap), adjacency queries, length of boundaries, and so on.

Still, while the underlying data structures are solid and have all the required functionality, it would involve a significant amount of effort to develop the operations required efficiently and in a robust manner. These would, at the very least, include:

Detecting gaps and overlaps For instance, by analysing faces situated left and right from every edge present. Overlaps are not a problem, since this information is yielded directly with the topological data structures present. However, it is hard to distinguish a gap from a region outside the domain of the planar partition without applying first operations to join multiple polygons.

Repairing gaps and overlaps To repair gaps and overlaps, polygons would have to be

assigned to a certain feature class, which could be determined based on the configuration of its neighbours. Afterwards, the polygon that a hole would be assigned to would be reassigned to become a union of the hole and its former self. For overlaps, the overlapping region would be subtracted from all polygons that this overlapping region would *not* be assigned to.

Detecting the number of disjoint regions Asserting the number of disjoint regions is also a difficult operation without performing an expensive union of all polygons.

Other operations (e.g. to repair the generated topology) might be required as well, depending on the implementation, which could further complicate matters.

Using a Constrained Triangulation for Planar Partition Validation and Repair

The process of planar partition validation and repair is defined by the developed algorithms, and consists of several steps which build on the successive validation and repair of simpler elements: rings to form polygons and polygons to form planar partitions. This gives both the power to remove degenerate cases at any point of the process, which are sometimes allowed in a certain specification¹, and the flexibility to allow the removal of some of these steps when these elements are known to be valid.

Computationally, there is commonly a tradeoff between speed and memory usage, which is especially applicable to geometric algorithms, since they are subject to both complex operations and large data sets. Therefore, choices have to be made while trying to keep a balance between the two with the available resources. For this thesis, this implies that algorithms developed can be sometimes made *either* faster or more memory efficient. However, much care has been taken to keep a balanced approach.

The method developed, as seen from a would-be user's perspective is shown in Figure 4.1. Following approximately the same order, the main five steps in this process are explained in the following sections (excluding the final output, which is trivial), with the output from each step being the input to the next one.

First of all, an input file containing a set of polygons is opened, from which polygons are individually repaired and added to the triangulation, which is discussed in Section 4.1. In order to do so, each of their rings are tested for self-intersections, and if any of them do self-intersect, they are reconstructed using a tagged triangulation as well. This is repeated until all edges from all polygons have been added to the triangulation as constrained edges.

Next, using the edge information from the previous step, the triangles in the triangulation are tagged with the polygons that they belong to, with a special tag used for the exterior of the planar partition (i.e. the regions outside its domain). This means

¹e.g. repeated points within a polygon in ESRI Shapefiles.

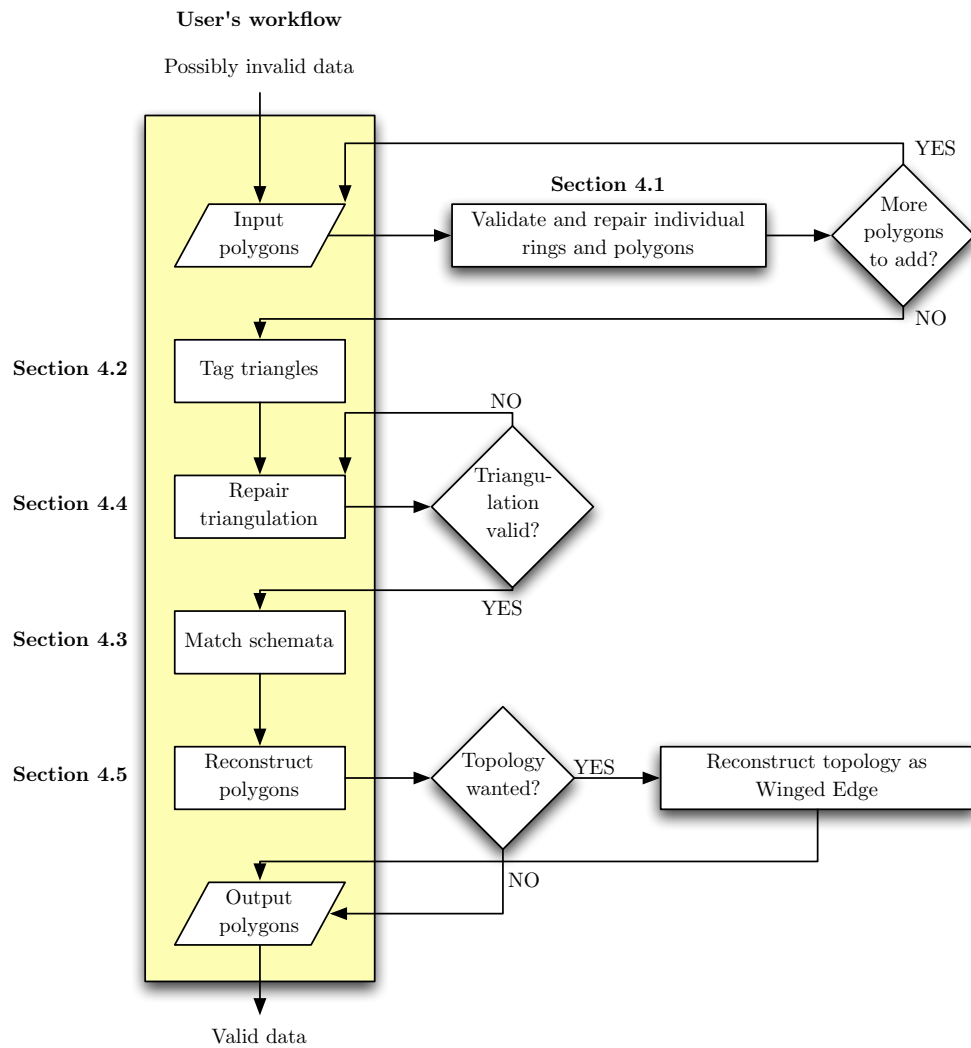


Figure 4.1: The process of validation and repair of a planar partition from a user's perspective. The workflow of a user is represented on the left, with the operations that are performed in the repair tool in the yellow box.

that each triangle can have a different number of tags: zero (gap), one (correct), or two or more (overlap). This procedure is discussed in Section 4.2.

Afterwards, the polygons in the triangulation can be repaired by a number of different operations, as it is explained in Section 4.4. These should ensure that, at the end, each triangle has exactly one tag, which allows the polygon reconstruction phase to create a valid planar partition as output.

Subsequently, adjacent polygons that should be joined are marked as such in the schemata matching process (Section 4.3). The purpose is akin to the dissolve operation common in GIS. However, unlike this operation, it is meant so that different criteria can be used to decide which polygons to merge (e.g. using a map of feature classifications between two different data sets).

Finally, the polygons are extracted from the triangulation, using first an algorithm that generates correctly oriented polylines that connect all boundaries of a polygon, which are then cut and assembled into individual rings. If topological data is also required, it is also extracted from the triangulation to a Winged Edge representation.

4.1 REPAIRING A SINGLE POLYGON

Since individual polygons must be valid for the entire procedure to work, the first step that must be performed consists of ensuring that these are repaired if necessary (see Appendix A.3). In order to achieve this, a very similar procedure to the one used for planar partitions is used, using also a constrained triangulation to get a valid polygon in each case. While other techniques for polygon validation and repair might work just as well (e.g. those mentioned in Section 3.1), it serves as a proof to the capabilities and extensibility of this approach.

In the same manner that a valid planar partition is composed of valid polygons, a fundamental building block in this hierarchy is that valid polygons (possibly with holes) should themselves be composed of valid rings (equivalent to polygons without holes). Based on the definition set for this purpose (see Section 2.3), these rings should be closed and not self-touching, while their orientation should be counterclockwise for rings that define an outer boundary and clockwise for inner boundaries. Additionally, zero area features (cut-lines, punctures and spikes and rings themselves) should be removed². What is meant by these terms is shown in Figure 4.2.

To fulfil the aforementioned conditions, a few rules regarding the interpretation of ambiguous polygons were used:

- The last vertex of a ring is always joined to the first one, yielding a closed loop even in representations where this is stored only implicitly.
- Successive identical vertices are removed, which solves problems with repeated vertices possibly generated by the previous step and in the representations where this condition is allowed (e.g. Shapefiles).

²Known commonly as the regularisation of a polygon, which is equivalent to the closure of the interior of it [Worboys and Duckham, 2004, chap. 3].

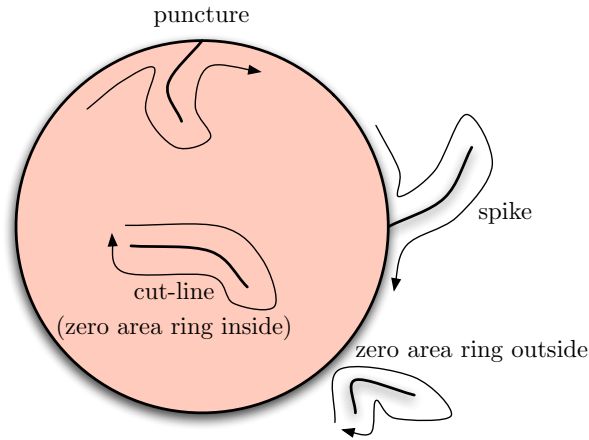


Figure 4.2: The different types of zero area features that can occur in a ring.

- Based on the specifications of the input format, it is assumed that whether a ring constitutes an internal or external boundary is known, but its orientation could be wrong³. This are mutually exclusive conditions for repair, since some representations identify holes solely by their orientation (e.g. Shapefiles). Alternatives could be: assuming that orientation defines whether a boundary is internal or external, disregarding known information in this respect; or examining which ring is outermost, to select it as an outer boundary.
- A ring is always bounded (i.e. it has a finite interior)⁴.

Based on these, it is known that rings are closed and bounded. If it is necessary to further repair the ring (see Appendix A.3), a constrained triangulation is incrementally built, containing all edges pertaining to it. When there are intersections at an internal point of an edge, it is split into two edges in the process. If the same edge is defined an even number of times, it is removed, which removes most of the degenerate cases (see Appendix A.4).

Because a ring is bounded, the *infinite face* of a triangulation is known to be in the exterior of the ring [see Yvinec, 2010], and it is therefore tagged as such. From this face, the same tag can be expanded to all adjacent faces reachable from it without passing through a constrained edge. When these are exhausted, all remaining faces reachable by passing once through a constrained edge are known to be in its interior. From the remaining faces, those that can be reached by passing through two constrained edges are in its exterior, and so on iteratively. The reasons and details of this algorithm are included in Appendix A.5.

When all triangles have been tagged⁵, the former ring is now reconstructed, in a procedure whose output may be any number of separate rings, some of them possibly

³Implementation-wise, this is done with the OGR library.

⁴The interior of a hole being the void area itself.

⁵Which in the case of CGAL are the triangles in the convex hull of the polygon, plus the infinite face.

with new holes inside them⁶. To achieve this, correctly oriented long polylines that include all boundaries of a given former ring are generated by the algorithm fully described in Appendix A.6, which involves rotating around a seeding triangle known to be in the interior of the ring, creating a long chain of edges in a manner similar to walking along the edges of a ring, but ensuring the correct orientation and creating “bridges” that connect a hole inside a ring to its outer boundary, if there is not one already. These are used to preserve the connectivity between different rings while the polygon is being reconstructed. One such polyline is generated for each separate component of the former ring. The results after this step are similar to the example shown in Figure 4.3.

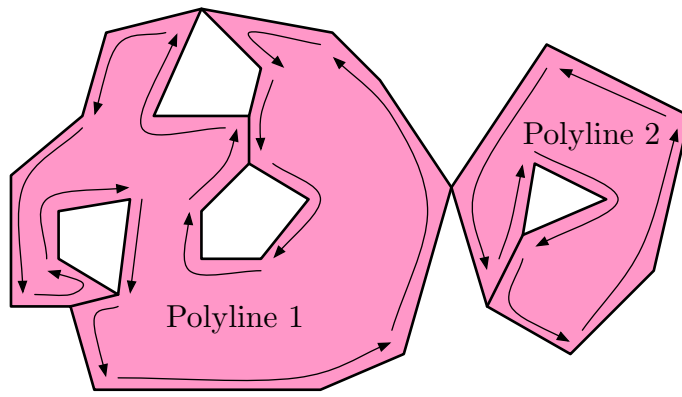


Figure 4.3: The polyline generated from a seeding triangle in the interior of the ring joins all holes with the external boundary, always while keeping the interior connected and on the same side of the line (left in this case). A separate polyline is always generated for each different interior connected component. Note that the “bridges” generated involve passing through them twice in the polyline.

From this polyline, the *degree* (number of incident edges) for each vertex is computed, only taking into account the edges coming from the polyline, yielding “cutting points” where the degree is larger than two. At these points the polyline is cut into smaller pieces, since they represent the places where pieces might be joined in a different order later on. From these pieces, some will be joined with others with the same end-points but opposite direction and be eliminated, while the remainder will be joined in the correct order to create new closed rings, as shown in Figure 4.4. This process is described in more in the full description of the algorithm in Appendix A.7. This procedure is repeated for all the input rings where repairing might be required, yielding a set of rings for the entire polygon, where their individual orientation specifies whether they are outer or inner boundaries.

Based on this set, the correct nesting of each inner boundary is identified. For this, it is necessary to find out in which outer boundaries they are located, which can be zero (a hole outside the boundary of any ring), one (correctly nested), or several (a hole

⁶It is possible to have no rings as output, since it could very possibly be a zero area feature.

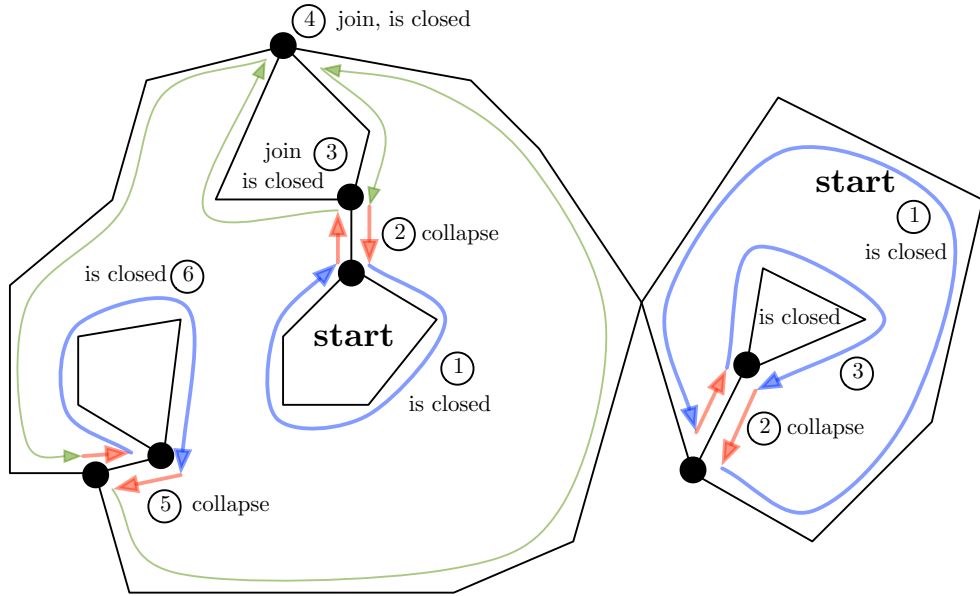


Figure 4.4: The vertices with a degree larger than two serve as cutting points (marked with black discs), from which separate polylines are generated. For each of these, a closed ring is first found, which serves as a starting point. From here, closed rings are removed and put into the list of completed rings (blue), duplicate polylines are removed (red), and all others are joined until deemed to be duplicate or closed.

which fits in multiple outer boundaries). To determine the nesting efficiently⁷, point-in-polygon operations are made, finding the location of each vertex of every inner ring in relation to the outer ones. This operation works on all cases that might be commonly expected, without causing performance problems. However, the full consequences of this are discussed in Appendix A.8.

At this point, the validity of rings is ensured, which allows for validation and repair of polygons to be done at the same time as for an entire planar partition. Again, in a similar manner as with the individual rings, all edges belonging to a polygon will be added into a triangulation incrementally. However, this time it is not necessary to check for overlapping edges, and the orientation of all rings is already well defined (i.e. rings are orientable).

4.2 TRIANGULATION AND TAGGING OF A PLANAR PARTITION

After validating and repairing all individual input polygons, or if some validity rules are known beforehand, it is possible to create a tagged triangulation of an entire planar partition at once. In this context, tags specify which polygon(s) does a triangle belong

⁷Without performing a brute force intersection test between every possible pair of inner and outer boundaries.

to. Namely, it is necessary to have polygons constituted from relatively simple rings⁸ with a predefined orientation, and that holes that are outside the polygon which should contain them are removed.

Based on this, a triangulation that contains all edges of all rings of all polygons is created incrementally, edge by edge. When edges are found to intersect, they should be split with a new point created at the intersection. This is the only condition where the generation of new points is required.

Since it is known that rings are closed and with known orientation⁹, it is also known on which side of a certain line segment the interior of the polygon lies, as shown in Figure 4.5.

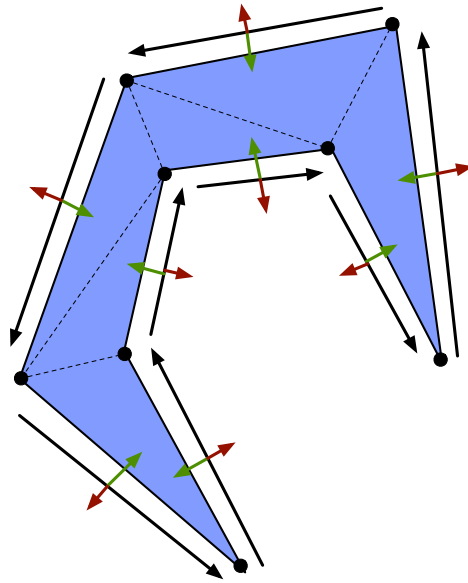


Figure 4.5: Once orientation criteria are met, it is known that the ring interior lies to the same side of each directed edge on the boundary. In this case, the interior of the polygon always lies to the left of each directed edge (in green), while the exterior is to the right (in red).

This property is now used for robust tagging of each polygon. Faces adjacent to the outer boundary of the polygons are first tagged, and later this is expanded to triangles further in the interior of the polygon, recursively tagging adjacent untagged faces as long as no constrained edges are traversed (see Figure 4.6). Tagging from the inner boundaries outwards is not done, since there is no guarantee that holes are *entirely*

⁸Not necessarily simple, but still orientable. Being self touching without being self intersecting does not cause any problems.

⁹Which can be quickly checked by looking at the turn direction around the leftmost or rightmost vertices. It is known as the convex corner method.

inside a certain polygon¹⁰. Meanwhile, removing tags from a hole inwards is another possibility, which ensures that holes sharing edges on the boundary of a polygon are preserved, but whether this is a desirable property is debatable, and is therefore not implemented in the prototype¹¹. See Figure 4.7 for the implications of these possibilities.

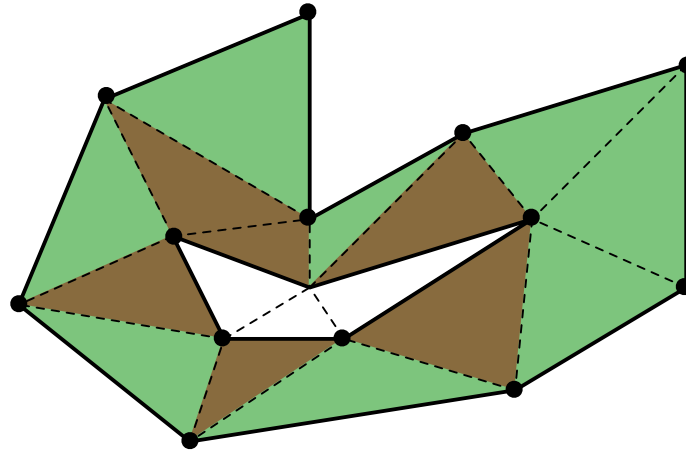


Figure 4.6: Tagging begins with the triangles adjacent to the outer boundary of the polygon (green), recursively advancing inwards until the rest of the triangles are tagged as well (brown), unless they are separated by a constrained edge from the rest (white).

After this operation, all triangles that are part of any polygon are tagged, with overlapping regions having multiply tagged triangles. However, holes are indistinguishable from triangles outside the planar partition, since both have zero tags. Therefore, a special tag is created for all triangles outside the planar partition (i.e. the “universe”), which are tagged by recursively tagging adjacent triangles from the infinite face of the triangulation¹², as long as no constrained edges are traversed. The results at the end of such tagging are shown in Figure 4.8.

4.3 SCHEMATA MATCHING

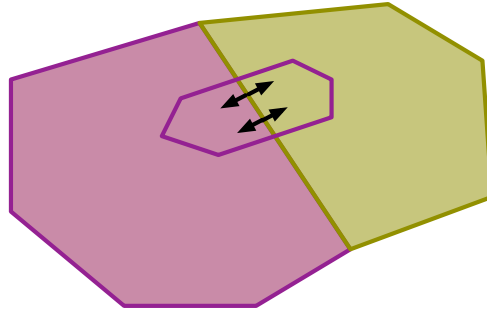
As sets of polygons from different files or data sources are added together, it is necessary to decide which attributes from one are equivalent to those in another. This issue is often not trivial, because while these might look superficially similar, they often differ in the format data is stored (e.g. types¹³, precision, length, etc.) and in the names of their respective attributes.

¹⁰It is possible to check for this using the tests mentioned in Appendix A.8. However, when repairing this situation would still require further operations in the individual polygon repair phase.

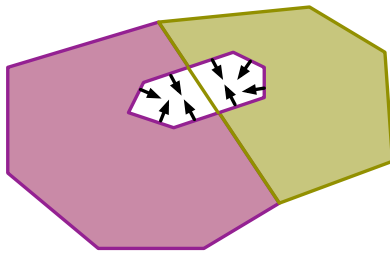
¹¹For instance, because a hole might be carved into a different polygon than the one where the hole should be in.

¹²Or a triangle known to lie in the exterior of the planar partition, in case this is not available in the implementation used.

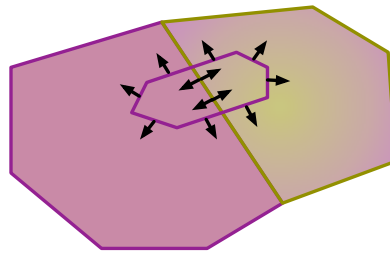
¹³For instance, numerical data is often stored as a string of characters.



(a) Tagging inwards from outer boundary only. Holes touching or intersecting an outer boundary disappear completely as their interiors are filled. This solution was selected for the prototype developed.



(b) Tagging inwards from the outer boundary and removing tags inwards from a hole. Holes will always be empty, but without additional information other tags may disappear as well.



(c) Tagging inwards from the outer boundary and outwards from a hole. Polygons might be double tagged when a hole from a different polygon is situated in their interior or boundary.

Figure 4.7: Different undesirable results may occur when selecting different tagging criteria. However, all three sets of criteria presented can be considered as a valid interpretation of the situation in different examples. Discerning the original intention behind an invalid polygon can be virtually impossible.

Additionally, it might be sometimes incongruent to join data at all, especially when it comes from different sources, since the meaning of their fields could be incompatible. For example, IDs can have only an internal meaning, units can be different and some fields require special treatment (e.g. area, perimeter and average values).

However, the semantic issues of data conflation are an entirely different topic, and fall outside the scope of this thesis. Therefore they are not discussed any further, except to briefly explain how such information is used in the process of planar partition repair. Meanwhile, the implementation details for the prototype regarding this topic are explained in Appendix A.9.

The objective of the process of schemata matching for planar partition repair is generating a set of equivalency relationships that state which polygons should be merged together. These relationships range from very simple (e.g. do not match anything) to quite complex (e.g. regular expressions involving several fields), depending on the data

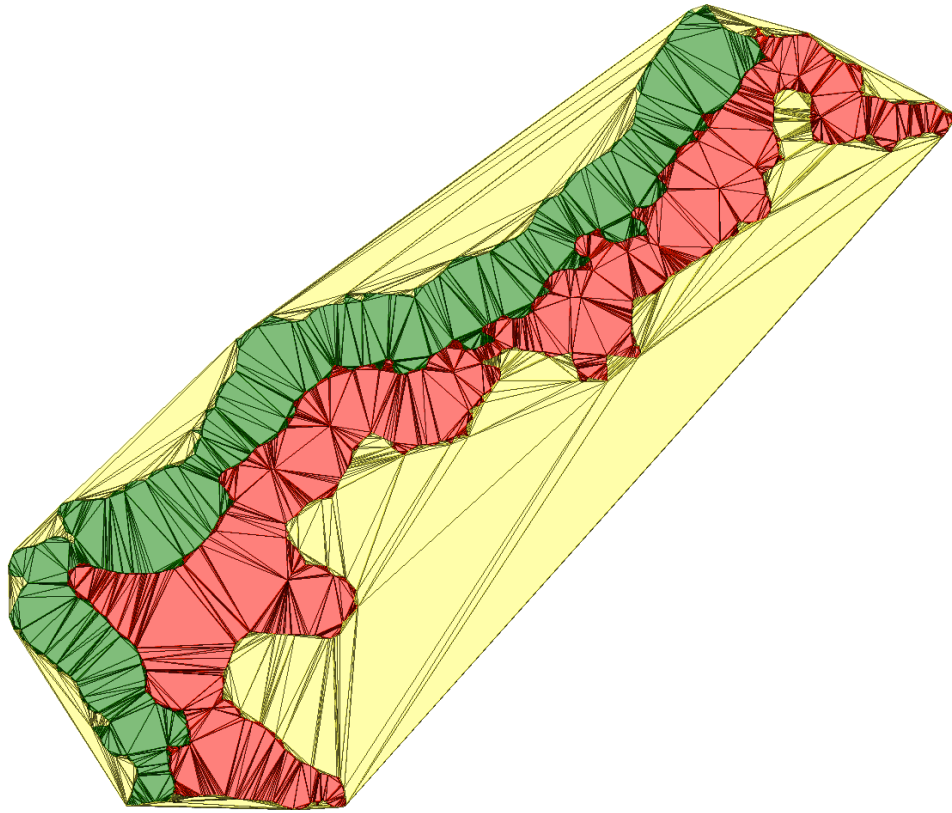


Figure 4.8: A tagged triangulation of the convex hull of two polygons for the Arribes del Duero Natural Park in Spain (red) and the International Douro Natural Park in Portugal (green). The triangles in the exterior of this planar partition are shown in yellow.

sets involved. The schemata matching operation should also be able to accommodate the conflation of both subdivided (e.g. tiled), where simply polygons with the same ID are merged (horizontal conflation); and independent data sets, where a map of feature classification equivalencies might be required (horizontal or vertical conflation).

Once this equivalency relationships are generated, they might be used for different purposes by the planar partition repair program:

- To reconstruct seamless polygons across the boundaries of different data sets.
- To create joint feature classes from formerly separate ones.
- To repair polygons only, using the file of origin information for each polygon to distinguish features sharing the same IDs in different data sets.
- To compute statistics on the newly created planar partition (e.g. number of disconnected components).

4.4 REPAIR OPERATIONS

The greatest benefit of using a tagged triangulation for planar partition repair stems from the fact that while repair operations are performed, the validity of the planar partition is always kept, together with the integrity of the data. This comes as a contrast to other methods, where care needs to be taken to ensure that the (geometric or topologic) validity is not broken. For instance, if a zero width corridor that joins to regions is created, it should be detected and removed. Therefore, as it will be shown in this section, simple and fully automatic repair can be easily and robustly implemented.

Based on the tagged triangulation, it is simple to implement different repair operations based on adding and removing tags from sets of triangles based on certain criteria. More complex operations that change the triangulation itself (e.g. splitting) are also possible¹⁴ [Meijers et al., 2010].

Within the scope of this thesis, six straightforward operations were implemented, which are valid for both gaps and overlaps, and are presented in Table 4.1. Four of them use triangles as a base, which is faster and modifies the area of each polygon the least; while two of them use regions of adjacent triangles with equivalent sets of tags (Figure 4.9), which is slower but has better cartographic properties. A sample of their results is also shown in Figure 4.10. What is meant by the type of operation is subsequently explained.

Table 4.1: The repair operations implemented in the prototype.

Repair operation	Type	Criteria
Triangle by number of neighbours	Focal	The tag present in the largest number of adjacent faces, overlaps included
Triangle by absolute majority	Focal	Tag present in two or more valid adjacent faces
Triangle by longest boundary	Focal	Tag present along the longest portion of the boundary of the adjacent faces
Regions by longest boundary	Focal of zonal	Tag present along the longest portion of the boundary of the adjacent faces
Regions by random neighbour	Focal of zonal	Random tag from the adjacent faces
Triangle by priority list	Varies	Tag with the highest priority

However, the set of criteria used in the functions created are not exhaustive, and based on these examples it is relatively easy to create other specialised repair operations that fit the spatial characteristics of the data best. For instance, in a particular data set, it might be preferable to have a simple function that selects a tag randomly from those

¹⁴But care should be taken to ensure robustness.

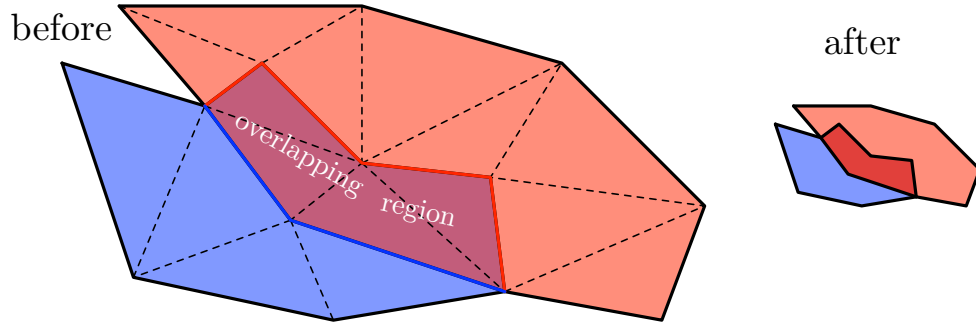
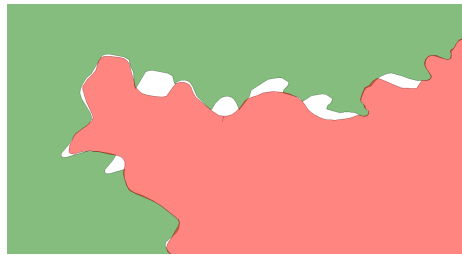


Figure 4.9: Regions are defined as adjacent triangles with equivalent sets of tags. In this example, the overlapping region between the red and blue polygons is repaired by the tag present along the longest part of the boundary surrounding the region (red). The creation of the region is a zonal operation, and the selection of the appropriate tag is a focal one.



(a) The original polygons.



(b) Repaired each triangle using the tag adjacent along the longest boundary from the neighbouring triangles.



(c) Repaired each region using a random tag from the neighbouring triangles.



(d) Repaired each region using the tag adjacent along the longest boundary from the neighbouring triangles.

Figure 4.10: Different repair operations used in the two polygons for the Arribes del Duero Natural Park in Spain (red) and the International Douro Natural Park in Portugal (green). All of them can be considered best by a certain criterion, like preserving the area ratio between the two polygons (b), smoothness of the boundary (d), or a balance between the two (c).

already in a multiply-tagged triangle; or a complex function that assigns a tag from a statistical analysis of the surrounding area. The only practical requirement for such a function is that, at the end, all triangles have exactly one tag assigned to them. This ensures that the result is entirely free of errors.

As shown in Table 4.1, it is possible to distinguish different types of operations based on whether they re-tag a triangle from the tags of the same triangle only (local), the surrounding triangles (focal), a region that the triangle is part of (zonal), or the entire triangulation (global)¹⁵. This is an important distinction, since it affects computing time and introduces specific things to take into account. For instance, the fact that focal operations can introduce order dependency, and that zonal operations might require new constrained edges to identify regions correctly.

However, most repair operations can be generalised to a simple algorithm, which is presented as follows:

```

Input: A tagged triangulation  $T$ .
Output: A (partially) repaired tagged triangulation  $T'$ .
1 Create a list  $C$  of changes to apply later.
2 foreach untagged or multiply tagged triangle  $t$  in  $T$  do
3   | Possibly expand  $t$  to the uniform region  $R$  containing it.
4   | Compute the tag to assign to the region  $R$ , if possible
5   | if a tag is found then enter the pair  $(R, tag)$  into  $C$ .
6 end
7 foreach pair  $(R, tag)$  in  $C$  do
8   | Re-tag each triangle in  $R$  with  $tag$ .
9 end

```

In this manner, order dependency in the operations is eliminated by performing all re-tagging at the end of the algorithm. Also, dubious cases (where a tag to assign is not found) are skipped, allowing for the creation of a more complex repair operation composed of a sequence of repair operations, which is equivalent to the use of a hard hierarchy of repair criteria.

Finally, repair can be accompanied by other pre and post processing operations, which serve to preserve certain properties desired in the polygons generated. For instance, as discussed previously, triangles could be split to subdivide an area with problems. Other options include discarding polygons with too small an area, or reassigning long and thin polygons a different tag.

4.5 EXTRACTION OF POLYGONS FROM A TRIANGULATION

Starting from a tagged triangulation, it is possible to reconstruct the individual polygons of a planar partition in a manner similar to how individual polygons were reconstructed after their repair, discussed in Section 4.1. If this triangulation has exactly

¹⁵The types of operations are defined in map algebra in Tomlin [1994], except for the addition of a global operator, which is also necessary in this context.

one tag per triangle, the triangles will also form a planar partition, without any gaps or overlaps.

This is an important step in incorporating a planar partition repair tool in the workflow of users working with planar partitions, since it ensures returning a now valid set of polygons, in the same format as they were input into the repair process.

In order to do this, a starting triangle inside a polygon is found. Afterwards, a single long well oriented polyline that runs along all the boundaries of the polygon is generated using the algorithm fully described in Appendix A.6, which involves a depth first clockwise search that recursively reaches until the boundary of a polygon, returning a long chain of edges in a procedure similar to following the boundary edge by edge. However, the polyline created with this method has “bridges” connecting all inner boundaries (holes) with the outer one, if there is not one already, while keeping the interior connected. These help to preserve connectivity and the relations between different (outer and inner) boundaries, but will be removed later in the process. Also, its orientation conveys the information of whether a section of it is an part of an inner or outer boundary.

It is important to note that during this procedure, interior disconnected polygons are separated into pieces, since the algorithm will not pass from one part of the polygon to another that is not interior connected to it, as shown in Figure 4.11.

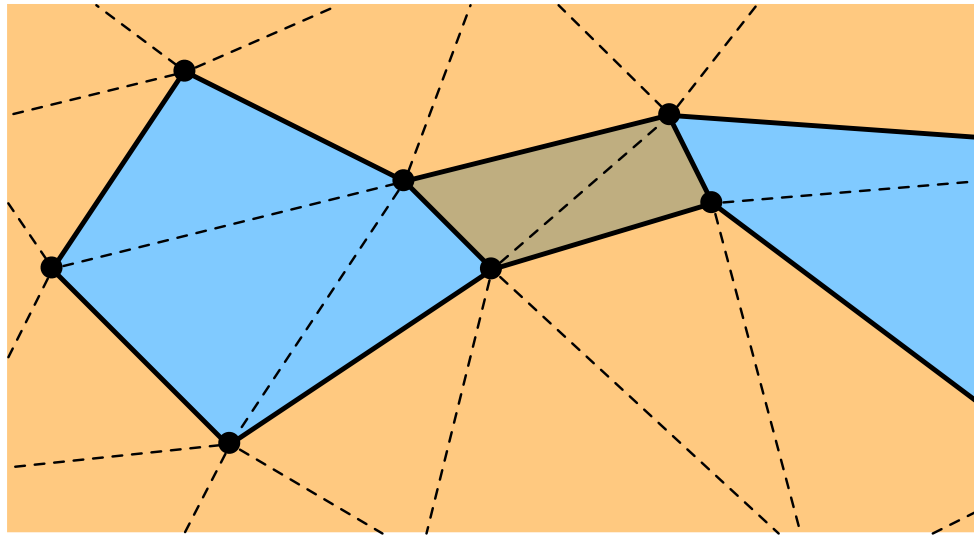


Figure 4.11: Suppose that two input polygons (orange and blue) are overlapping in a certain region (brown), which is assigned to the orange polygon in the repair process, causing the blue polygon to be divided into two pieces. The reconstruction algorithm should generate two separate polygons (blue), each of them with its respective attributes. Meanwhile, the former boundary separating the blue polygon from the orange one becomes a new (inner) boundary.

Afterwards, this polyline is collapsed around a vertex in regions where there are

successive edges with the same end points but opposite orientation, i.e. spikes or “bridges”, the former being generated when the starting triangle in the algorithm is not adjacent to the outer boundary of the polygon, while the latter forms a connection between inner and outer boundaries. This is shown in Figure A.12, along with a more thorough explanation of this algorithm in Appendix A.7. While this procedure was only a possible optimisation when reconstructing individual rings from a polygon, now it is necessary, since this removes some edges that were changed during the repair phase.

After the collapse of these sections of the polyline, a *degree* (number of incident edges) is computed for each vertex in it, only taking into account unique¹⁶ edges that are part of the polyline itself. At the vertices where the degree is larger than two (junction point), the polyline is cut into smaller pieces, since it is at these points where the polyline might be joined in a different order later on.

From these pieces, a successive series of operations is performed: closed rings are identified, pieces are collapsed and matching pieces are joined, yielding new closed rings. These operations continue until all pieces of the polyline are either part of a closed ring or are collapsed.

At this point the geometry of the polygons in a planar partition has been reconstructed, which can be used as a base to reconstruct its topology. It is worth noting that this is not the most efficient technique to get its topology when the geometry will not be used. However, within the context of this thesis, retrieving polygon geometry is always needed, since it is necessary to generate simple features output, while generating polygon topology is not.

A Winged Edge data structure [Baumgart, 1974] (see Figure 2.3 or the green classes in Figure A.1) has been chosen as the output format for the topology generation. For this, a tagged triangulation is required, although with a few differences with respect to the one that was there before the geometry generation. Namely, it is necessary to re-tag all triangles based on the sets of new rings generated previously, giving a new (unique) ID to every interior connected component of the planar partition.

Afterwards, a count of the number of unique IDs in the incident faces of each vertex is calculated, which serves the same function as the degree that was calculated to split the long polyline in the reconstruction of geometry, i.e. cutting the rings at where the count is larger than two, being junction points where several edges come together. These points will become nodes to be added to the Winged Edge data structure.

However, it is also necessary to add additional nodes to accommodate rings without any junction points, like the red node in Figure 4.12. To add these nodes, it is possible to simply scan all rings to ensure that each has at least one node, and add one for those which do not. An example of the result of this process is shown in Figure 4.13.

After all required nodes have been added, all rings of all the polygons generated in the geometry phase are scanned, so as to create the edges for the Winged Edge data structure. In order to do this, any node along a ring is located, from where unprocessed edges are added until the starting node is reached again. To ensure that an edge is only processed once, pairs of points that describe the edge are added into an auxiliary data

¹⁶Unique as defined by their end points, no matter their orientation.

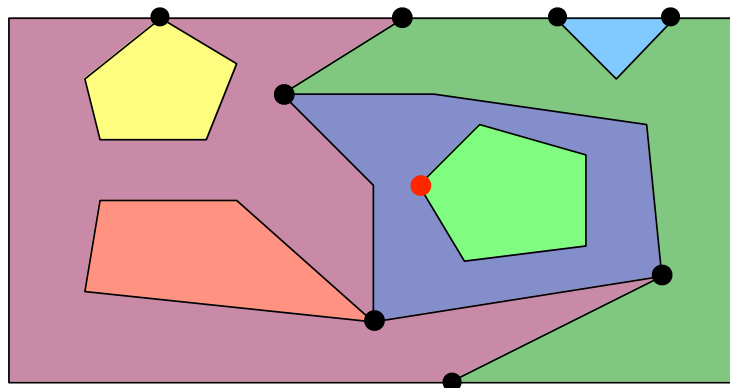


Figure 4.12: Some nodes for the Winged Edge can be chosen as those that have more than two unique IDs in their incident faces (black discs). However, it is also necessary to add nodes for rings without any junction points (red disc), which are those that are completely surrounded by a single polygon, like the light green polygon, but without any node incident to a different polygon, like the red polygon.

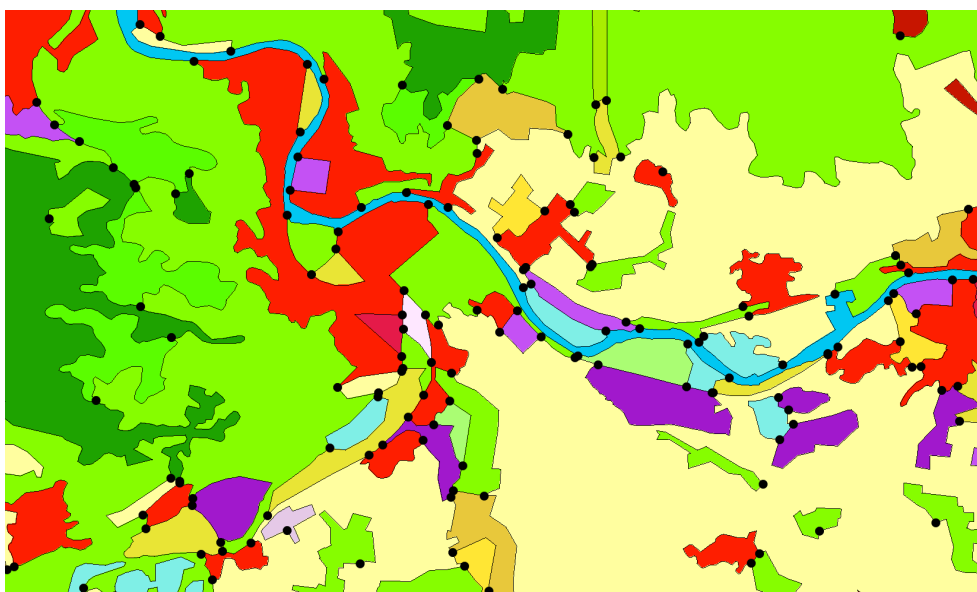


Figure 4.13: In this example from CORINE tile E37N28 (overlaid), the nodes for the Winged Edge data structure (black dots) can be seen at the vertices at the junction of three or more polygons, plus at any point along the boundary of those polygons completely surrounded by a single one.

structure (see Figure A.3 for the details). At the end, all parts of any polygon boundary should be included in the edges created.

Once all edges are known and added to the Winged Edge data structure, faces (together with all their information from the original input) and one edge per ring are stored as well. This is done after edge creation so as to use the same edge IDs created during that process.

Finally, the remaining data for each edge (left clockwise, left counterclockwise, right clockwise and right counterclockwise edges) are added by scanning all the edges created beforehand and rotating around their end points using the triangulation data structure until this data is found. An example final result from the topology generation process is shown in Figure 4.14.

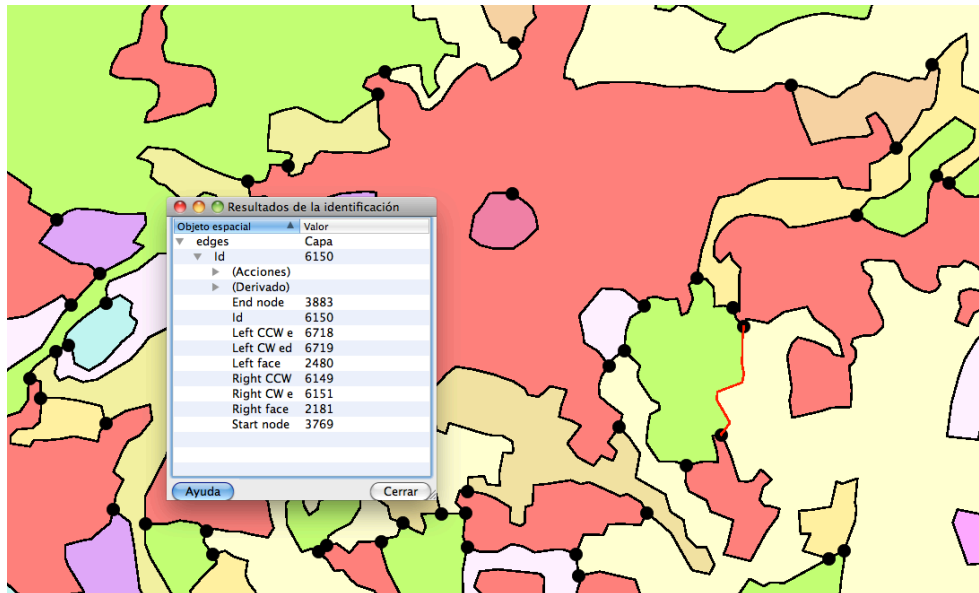


Figure 4.14: The topology generation process in CORINE tile E37N28. The attributes for the generated edge 6150 (in red) are shown.

Implementation, Experiments and Discussion

In order to see how the algorithms designed for this thesis perform with respect to the existing solutions, it is important not only to make comparisons about how they work in theory (Chapters 3 and 4), but also to make some tests that evaluate the capabilities of each and show some important practical considerations (e.g. running times and memory usage).

For this, it is primordial to first describe how the implementation of the prototype has been done, with Section 5.1 being devoted to this topic. Also, it includes basic information on how to get and use this software.

Afterwards, the practical tests follow, which have been divided in two complementary types. First, small scale tests are run using a purpose-made data set showing specific problematic polygons that should be tackled in programs for polygon repair, which is discussed in Section 5.2. Later, in Section 5.3, planar partitions of different characteristics are repaired in each of the available programs, showing their performance in different situations.

5.1 THE DEVELOPED PROTOTYPE

To analyse, test and improve the algorithms, and encourage further development, a fast implementation was written in the C++ programming language, using external libraries for some functionality. C++ was selected in order to have plenty of control with regards to low level details and to achieve good performance, which makes it possible to compare it with existing solutions. Meanwhile, the libraries directly used are: the OGR Simple Features Library, which allows input and output from a large variety of data formats common in GIS; and the Computational Geometry Algorithms Library (CGAL), which has support for many robust spatial data structures and the operations based on them, including polygons and triangulations.

The developed prototype is open source, and freely available on the Geo-Database Management Center (GDMC) (<http://www.gdmc.nl>) website, together with this MSc thesis. Regarding the other required libraries, OGR is also open source, and available under a X/MIT license from <http://www.gdal.org/ogr/>, while CGAL is available with some restrictions, depending on its intended use, from <http://www.cgal.org>.

The end result comprises about 140 functions and 4,500 lines of code, with the main classes that were programmed shown in Appendix A.1. However, it is also easily extensible, which should help to promote future work on the topic (Section 6.3).

It also takes care of the many degenerate cases that occur, as common as they are in computational geometry algorithms, of which the main ones are discussed in Appendix B. Nonetheless, the degenerate cases have simpler solutions than in other approaches.

Since the software is meant as a prototype, there is currently no user interface. Instead, high level functions are provided, which are run from a separate file. A working example of what should such a file contain is shown in Figure 5.1, while an extract of the output generated using this file is in Figure 5.2.

The algorithms developed are covered in Chapter 4, with a more extensive discussion regarding low level details and their implementation in Appendix A. Therefore Appendix A.1 starts with UML diagrams of the main data structures in the prototype implementation, while A.2 describes the data structures and types used to keep the required information in the triangulation, and A.3 to A.9 discuss various low level details of the algorithms in this thesis, including pseudocode of each and its computational complexity.

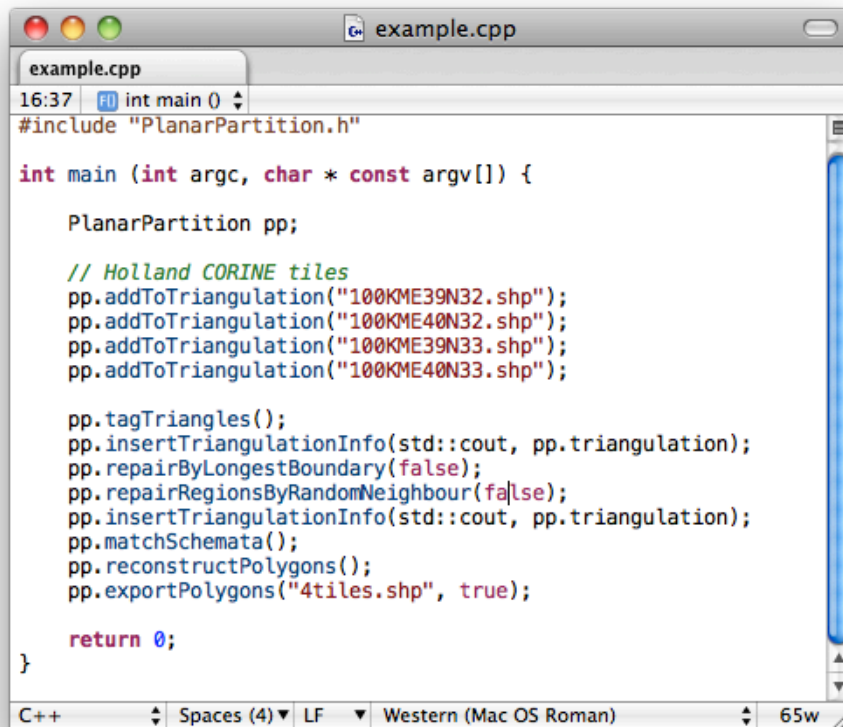
5.2 INVALID POLYGONS REPAIR COMPARISON

To test the capabilities of individual polygon validation and repair in the developed prototype and existing implementations, a set of test polygons was created, which is shown in Appendix C.1. The situations depicted in these purposefully involve many degenerate cases for different steps of validation and repair processes, both with regards to interpretation and implementation. In this manner, they are meant as a sort of *unit testing* polygons to compare how they fare in different tools¹ [Burns, 2001].

All of these polygons were first viewed in ArcGIS, FME, GRASS and Quantum GIS (QGIS) to check for differences in their interpretation. This is an important step in evaluating these tools in polygon validation and repair, since it helps in two main respects:

- Learn how different implementations deal with polygons in reality, since documentation does not cover every case or is sometimes inaccurate.

¹Strictly, it is not true unit testing, since it is not known with certainty whether the polygons reliably trigger all existing checks in every method. This would require far more implementation knowledge of every solution, which works as a black box in many cases, and is therefore hard to get and beyond the scope of this thesis.



```

example.cpp
16:37 int main ()
#include "PlanarPartition.h"

int main (int argc, char * const argv[]) {

    PlanarPartition pp;

    // Holland CORINE tiles
    pp.addToTriangulation("100KME39N32.shp");
    pp.addToTriangulation("100KME40N32.shp");
    pp.addToTriangulation("100KME39N33.shp");
    pp.addToTriangulation("100KME40N33.shp");

    pp.tagTriangles();
    pp.insertTriangulationInfo(std::cout, pp.triangulation);
    pp.repairByLongestBoundary(false);
    pp.repairRegionsByRandomNeighbour(false);
    pp.insertTriangulationInfo(std::cout, pp.triangulation);
    pp.matchSchemata();
    pp.reconstructPolygons();
    pp.exportPolygons("4tiles.shp", true);

    return 0;
}

C++ Spaces (4) LF Western (Mac OS Roman) 65w

```

Figure 5.1: Example of a program to repair four tiles of the CORINE 2000 data set. Basically, it adds the four tiles to the triangulation, tags them, tries to repair them using the longest neighbour criteria first (Section 4.4), otherwise assigns a tag randomly, matches the schemata of the four tiles, extracts the polygons from the triangulation, and writes them to a file.

- Verify that polygon repair functions are consistent with the interpretation of polygons in a certain program, i.e. that the polygons displayed before and after repair are equivalent.

These tests showed that there are very significant differences with regards to how degenerate cases are interpreted, which can certainly cause interoperability problems when using different software. A few representative examples of this situation are presented in Figures 5.3, 5.4 and 5.5.

These situations are not at all unique, with many of the polygons generated having different interpretations in each software tested. This was expected, since polygon definitions specify how a certain polygon should be stored, but not how a certain stored representation of a polygon should be interpreted (i.e. from a polygon to its representation, but not vice versa).

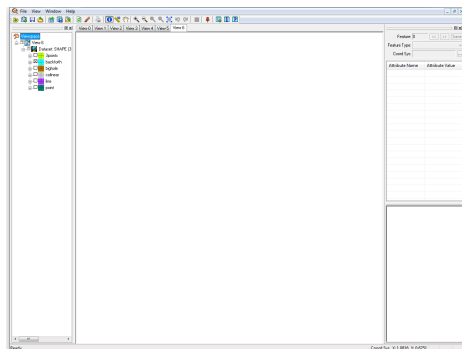
```

Triangles: 82661
Adding a new set of polygons to the triangulation...
File opened.
  Name: /Volumes/Buffalo/corine/100KME40N32.shp
  Type: ESRI Shapefile
  Layers: 1
  Layer[1]: 2081 polygons {
    string    CODE_00
    double    AREA
  }
(213) OB: Self intersecting.
Splitting ring (2169 nodes) at (4090897.81947201,3205131.85136284)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(295) OB: Self intersecting.
Splitting ring (171 nodes) at (4014338.28224871,3205959.57513961)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(560) OB: Self intersecting.
Splitting ring (827 nodes) at (4006889.99296177,3214962.87908400)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(661) OB: Self intersecting.
Splitting ring (137 nodes) at (4052575.12529232,3220418.25137708)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(954) OB: Self intersecting.
Splitting ring (206 nodes) at (4041756.04611424,3231507.73413940)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(983) OB: Self intersecting.
Splitting ring (135 nodes) at (4015515.72557781,3236851.22009638)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(1219) OB: Self intersecting.
Splitting ring (186 nodes) at (4028234.13326965,3246289.96980117)...
Created 2 rings.
Outer rings: 1 inner rings: 1
(1264) OB: Self intersecting.
Splitting ring (952 nodes) at (4041062.88515360,3236397.85638396)...
Created 2 rings.
Debugging terminated.
Succeeded 2

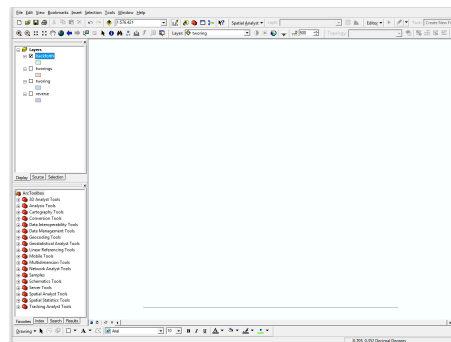
```

Figure 5.2: An excerpt from the output of the prototype, using the input file shown in Figure 5.1. Several rings from tile E40N32 were found to be self-intersecting, which caused them to be split during the individual polygon repair phase of the program.

5.2. Invalid Polygons Repair Comparison

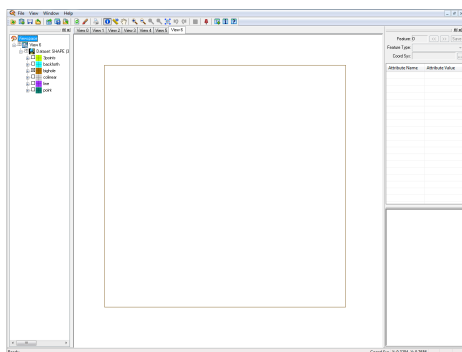


(a) FME deletes the line. QGIS and GRASS do it as well.

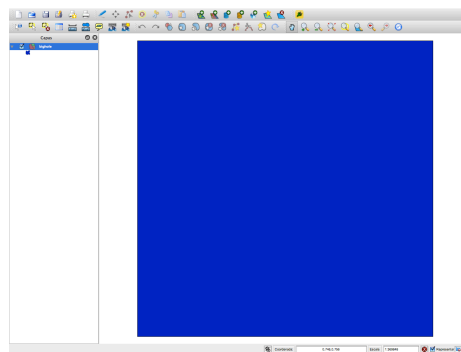


(b) ArcGIS considers that the line exists, although it is deleted during repair.

Figure 5.3: Different interpretations of the “backforth” polygon (Appendix Figure C.1d).



(a) FME presents it as a zero area feature. ArcGIS does this as well. However, during repair the hole is eliminated.



(b) QGIS discards the hole. GRASS does this as well.

Figure 5.4: Different interpretations of the “bighole” polygon (Appendix Figure C.1g).

Therefore, performing a certain process in different GIS tools can give significantly different outputs if a unique unambiguous valid polygon is not created when degenerate polygons are present. This makes a strong case that it is desirable for polygon repair tools to also ensure that polygons with multiple possible valid representations are standardised to a certain one. An added benefit of putting a polygon repair tool at the beginning of a GIS user’s workflow, is that it then also guarantees that different GIS software will be able to give similar results to similar queries.

After checking for the possible differences in interpretation of the test polygons, they were validated and repaired in ArcGIS and the developed prototype. Comparisons with the other available tools are very not meaningful, since they provide only basic

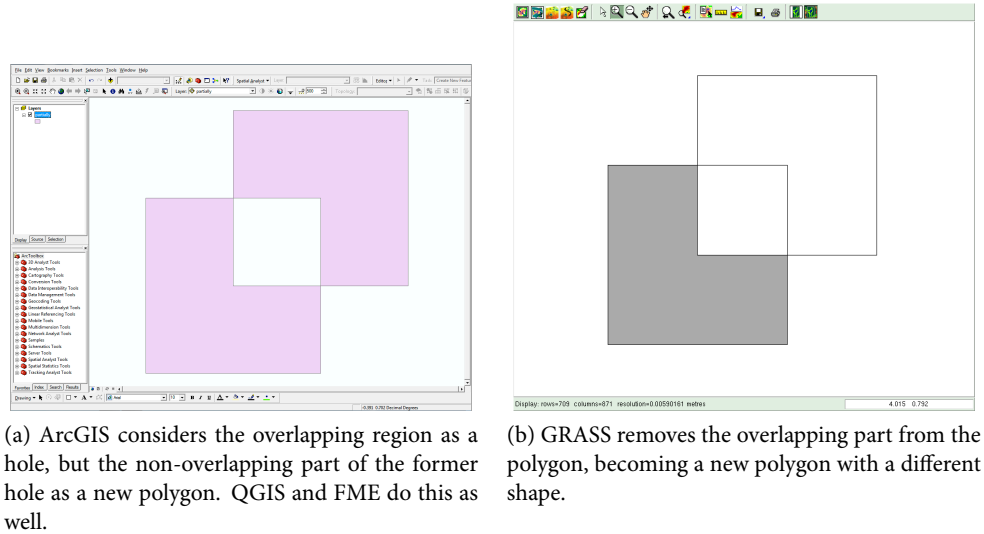


Figure 5.5: Different interpretations of the “partially” polygon.

polygon validation and do not provide any error codes². Moreover, the results of their repair operations heavily depend of their (user defined) process.

The results of this test are summarised in Table 5.1. Note that the results from this test are not meant as a direct evaluation of the capabilities of each, since there is not a clearly defined right or wrong answer. For instance, ArcGIS generates polygons with opposite winding as Simple Features based programs, but this should be expected. However, from these results, there are some observations that deserve further mention:

- Although it might be intuitively expected that only those polygons that are considered invalid should be modified; as stated previously, there are situations where it is better to make changes to them. This helps to have a unique representation for each situation and is done extensively in the developed algorithms (e.g. breaking multi polygons in parts). ArcGIS does this as well, to a limited extent (mainly with respect to polygon winding).
- ArcGIS considers the situations in polygons “bighole” and “tworing” valid, despite the fact that they have both self intersections and zero area, which is disallowed in the Shapefile specification [ESRI, 1998].
- If some holes are expected not to have any vertices in the interior of its containing outer boundary (e.g. polygons “3touch” and “3node”), it is better to have a more computationally expensive approach to decide which outer boundaries they belong to, something that is discussed in Appendix A.8.

²Except for QGIS when using ftools.

Table 5.1: Validation of the unit test polygons from Appendix C.1. The prototype (CT) error codes are: (DV) duplicate vertices, (ZA) zero area, (SI) self intersections, (NOB-FIB) no outer boundary for an inner boundary to fit into. The ArcGIS ones are: (SI) self intersections, (UR) unclosed rings, and (IRO) incorrect ring ordering.

Polygon	CT error	CT action	ArcGIS error	ArcGIS action
point	ZA,SI	deleted	SI	deleted
3point	DV,ZA,SI	deleted	SI	deleted
line	ZA	deleted	UR	deleted
backforth	ZA	deleted	SI	deleted
colinear	SI	deleted	SI	deleted
3backforth	SI	deleted	SI	deleted
bighole	–	deleted	–	–
2bigholes	–	deleted	SI	removed holes
cw	–	reversed winding	–	–
ccw	–	–	–	reversed winding
innerhole	–	–	–	–
2innerholes	–	–	–	–
hole	–	added node	SI	added node
holenode	–	–	–	–
2holenode	–	separated holes	–	–
2separate	–	–	–	–
bridge	SI	removed bridge	SI	removed bridge
bridgeback	SI	removed bridge	SI	removed bridge
edgehole	–	removed polygon section	SI	removed hole
2edgehole	–	removed polygon section	SI	removed hole
spikeout	SI	removed spike	SI	removed spike
spikein	SI	removed spike	SI	removed spike
selfintersect	SI	created multi polygon	SI	created multi polygon
selfnode	SI	created multi polygon	IRO	created multi polygon
selftouch	SI	created multi polygon	–	–
tworings	–	–	–	–
twoareas	SI	created multi polygon	SI	created multi polygon
twosemi	SI	created multi polygon	SI	created multi polygon
2holes	–	–	–	–
3holes	–	–	–	–
3touch	NOBFIB	removed hole	SI	created multi polygon
3node	NOBFIB	removed hole	–	–
3separate	–	–	–	–
3loop	SI	created multi polygon	SI	created multi polygon
out	NOBFIB	removed hole	–	hole to polygon
corner	NOBFIB	removed hole	–	hole to polygon
edge	NOBFIB	removed hole	SI	hole to polygon
partially	–	removed part of polygon	SI	part of hole to polygon
multiple	–	–	–	inner hole to polygon
open	–	closed ring	UR	closed ring
tworing	SI,ZA	deleted	–	empty interior
reverse	SI,ZA	deleted	SI	filled interior

- In ArcGIS, QGIS and FME, having holes very close to a polygon boundary could cause them to be considered outside a polygon, and therefore have a new polygon generated, in case of numerical robustness problems, as discussed in Section 2.2.
- Polygon repair might change the polygons originally visualised entirely, as seen in polygons “2bigholes”, “edgehole” and “2edgehole”, where their holes were eliminated despite covering (part of) these polygons.

5.3 LARGE PLANAR PARTITION REPAIR COMPARISON

While the capabilities for planar partition repair among the software tested vary considerably, with full topological repair only available in ArcGIS (manual only) and the developed prototype³; it is also important to consider how different repair implementations scale to large data sets.

For this, a few performance tests were made in three available planar partition repair tools (ArcGIS, FME and GRASS) that perform this process using snapping and splitting, and the developed prototype. The testing methodology for each tool is as follows:

ArcGIS In ArcCatalog, a multiple feature data set is created in a geodatabase, set with tolerance values equal to the snapping threshold. Features are imported into it and the merge and dissolve operations are used to merge adjacent polygons with the same ID. Topology is then generated to check that the planar partition is valid. Everything is finally exported to a single Shapefile. The individual parts of the process are timed and their total timing is recorded. Memory usage is calculated as the difference between the just loaded ArcCatalog application and its maximum memory usage throughout the process.

FME A reader is created for each input file, which serve as input to a Snapper transformer; features with the same IDs are then dissolved, and finally they are output into a new Shapefile writer. The topology generator is used to be able to tell whether the result is a valid planar partition. Results are timed and the maximum memory of the `fme.exe` process is recorded.

GRASS Input files are imported with `v.in.ogr`, with all polygon cleaning operations performed and snapping set to the correct values. Boundaries between features with the same IDs are then dissolved using `v.dissolve`. Files are then exported with `v.out.ogr`. Times reported by GRASS are added together to give the total, while memory usage by the `v.in.ogr.exe`, `v.dissolve.exe` and `v.out.ogr.exe` are monitored and their maximum is recorded.

Constrained Triangulation Repair Files are read and put into the triangulation, the triangulation is tagged, repair is performed with the `repairByLongestBoundary()` first, with ambiguous cases handled by `repairRegionsByRandomNeigh-`

³And ArcGIS requiring extensive user interaction for it to work, as discussed in Section 3.4.

`bour()`. Polygons are then extracted from the triangulation and output to a single Shapefile. The entire process is timed, and the maximum amount of memory used is recorded.

As stated previously in Section 3.3, the results are not directly comparable, since planar partition repair using a constrained triangulation is able to repair many more cases than other methods, keeps topological consistency, does not require finding out good threshold values, and does repair in a robust manner. Moreover, it is able to directly state whether the result is a planar partition, unlike all the other solutions. In ArcGIS, the topology constraints would have to be checked, in FME the topology would have to be checked externally, and in GRASS, the second layer would be checked to make sure that it does not contain any features.

Despite this limitations, in order to have an idea of the processing time and memory usage involved, these tests have been put into Table 5.2. Only cases that are acceptably solved by snapping and splitting have been considered, since there is no other comparable automated topological repair tool among those studied that would also work for more complex cases, such as those that require a snapping threshold too high to be practical (e.g. horizontal conflation of independently generated data sets).

All tests have been run 5 times in a machine just booted and the results averaged to account for the small variations that occurred⁴. The hardware is a 2.66 Ghz Core 2 Duo MacBook Pro with 4 GB of RAM. ArcGIS 9.3, FME 2010 SP1 and GRASS 6.4 were run in Windows 7, while the developed prototype was run in Mac OS X 10.6.4.

The data sets tested are:

E41N27 CORINE 2000 tile E41N27, which contains a shifted polygon about 10 cm, creating many small gaps and overlaps in the data set. The snapping threshold has been set at 1 m.

4tiles CORINE 2000 tiles E39N32, E39N33, E40N32 and E40N33, which are known to have long and thin overlapping regions (< 1 mm) with each other. The snapping threshold has been set at 1 cm.

16tiles 16 adjacent CORINE 2000 tiles: E39N30, E39N31, E39N32, E39N33, E40N30, E40N31, E40N32, E40N33, E41N30, E41N31, E41N32, E41N33, E42N30, E42N31, E42N32, E42E33. Some have gaps between one another, some overlap, but match within a few centimetres. The snapping threshold has been set at 10 cm.

Mexico 1:1,000,000 scale land cover data set from INEGI consisting of over 26,000 polygons. It is mostly already valid, but contains some very large polygons (with tens of thousands of vertices).

As the aforementioned table shows, the constrained triangulation approach uses more memory than other solutions. However, it is the only fully automated method,

⁴Except in the case where the execution was cancelled after one full day, due to time limitations, or when it causes the program to crash.

Table 5.2: Planar partition repair comparison using large data sets.

Test	CT		ArcGIS		FME		GRASS	
	memory	time	memory	time	memory	time	memory	time
E41N27	124 MB	46s	145 MB	1m3s	158 MB	31s	59 MB	3m09s
4tiles	100 MB	3m25s	113 MB	37s	105 MB	31s	49 MB	53s
16tiles	1.51 GB	1h20m	crashes	–	636 MB	15m48s	crashes	–
Mexico	983 MB	18m53s	216 MB	>1d	264 MB	2m45s	408 MB	11m38s

since others require manually finding a good snapping threshold. Also, it is able to scale well compared to other approaches, especially taking into consideration the fact that it is more capable in solving complex problems.

Conclusions, Recommendations and Future Work

Planar partitions constitute one of the most important data representations in GIS, and are extensively used as a base for other operations. However, since polygons are often stored separately, various errors are introduced during their creation, manipulation and exchange. This creates many problems for algorithms that rely on valid planar partitions, and can cause them to fail or give erroneous results, often without any warning to the user.

Existing approaches to solve this usually involve polygon repair using a list of constraints, and complex planar partition repair operations performed on a planar graph. However, these have many shortcomings in terms of complexity, numerical robustness and difficulty of implementation. Moreover, they leave many invalid cases untouched.

To solve this problem, a novel method to validate and automatically repair planar partitions has been developed. It uses a constrained triangulation of the polygons as a base, which being by definition a planar partition, means that only relatively simple operations are needed to ensure that the output becomes valid. Point locations are maintained throughout the process, while fully automatic repair is possible using customisable criteria. This approach is also extensible to individual polygons, is capable of handling a larger variety of cases and has good performance compared to existing alternatives; all of this with numerical robustness and maintaining topological consistency throughout.

To analyse, test and improve the algorithms, and encourage further development, a fast implementation was written in the C++ programming language, using external libraries for some functionality. The developed prototype is open source, and freely available on the GDMC (<http://www.gdmc.nl>) website, together with this MSc thesis.

For this chapter, the main conclusions of this thesis work are summarised in Section 6.1. Afterwards, the most important original contributions are described in Section 6.2. Finally, some recommendations for future work are included in Section 6.3.

6.1 CONCLUSIONS

This thesis has shown that using a constrained triangulation as a base structure brings several advantages for planar partition validation, but where it really excels compared to existing solutions is in performing automatic repair operations.

The main advantages of using a constrained triangulation for planar partition validation and repair can be summarised as follows:

Individual polygons can be validated and repaired as well Current solutions for planar partition repair either assume that individual polygons are valid, or perform very simple repair operations which work only for some cases. In the worst case, these repair operations change the interpretation of a polygon entirely (Section 5.2). However, using a constrained triangulation naturally extends to the validation and repair of simpler elements (rings and polygons), using the same data structures and similar algorithms.

Snapping is possible, but not required Using snapping for planar partition repair imposes many restrictions on the data, and is also a destructive operation that can potentially break edge matching operations later on (Section 3.3). Using a constrained triangulation allows point locations to be preserved¹, and allows for robust operations that maintain topological consistency throughout the repair process. Because of the same reason, it is not necessary to have a reference data set. However, if snapping is required, it can also be done beforehand, being compatible with the algorithms used here. Additionally, in many cases comparable results can be obtained without it (e.g. by checking for very short triangles).

Fully automated repair Topological repair operations for planar partitions are relatively uncommon in GIS software, with only ArcGIS having them among the tools studied. Others, like GRASS and Radius Topology use topology only as a base, and snapping to solve the problems. However, this implementation requires extensive user interaction, making it unfeasible for large data sets. Having fully automated repair greatly extends the range of applicability of such a solution.

Simple and robust repair operations Since a constrained triangulation is, by definition, a planar partition, the repair operations built on top of it only need to ensure that all triangles have exactly one tag. By meeting this simple condition, topological consistency of the planar partition is guaranteed.

Admittedly, these come at a cost of somewhat higher (although comparable) memory usage than some comparable solutions and the need for efficient triangulation algorithms. However, these disadvantages are easily overcome with the availability of

¹As long as they are stored with the same data type as originally. In the case of the prototype developed, values are stored with double floating point precision, but using the CGAL templates this can be easily changed.

high quality triangulation libraries like CGAL [CGAL, 2010] and Triangle [Shewchuck, 1996], and the research done to reduce the memory footprint of triangulations, as it will be discussed in the next section.

6.2 MAIN CONTRIBUTIONS

The main accomplishment of this thesis has been understanding and describing how constrained triangulations can be best used for the validation and automatic repair of planar partitions, taking advantage of all their assets, which has been laid out in detail throughout this thesis. In order to achieve this, some important contributions have been made, from which three are most notable:

Extension of the algorithms to individual polygons Having valid polygons is a necessary step for the successful validation and repair of planar partitions. Since the triangulation of these is an integral part of the algorithms developed, this same data structure can be used to validate and repair them as well, with adapted versions of the same principles.

Development of robust algorithms As discussed in Section 2.2, numerical robustness is paramount in geometric operations. However, many algorithms that are extensively used are inherently not “safe” when dealing with very low values. This issue is beyond what most users of GIS software are aware of, or want to be concerned with. Special care has been taken to ensure that all algorithms developed for this thesis are robust, including those described in Appendices A.5 to A.7. This has been achieved with the use of CGAL and appropriate measures wherever needed.

Framework for automatic repair operations Repair operations on the tagged triangulation can be implemented in a few simple steps, as established in Section 4.4.

6.3 RECOMMENDATIONS AND FUTURE WORK

This thesis work helps to understand how constrained triangulations can bring robust validation and fully automated repair to planar partitions. However, it also raises a few questions and brings some suggestions with regards to future work on the topic.

One consideration for future work is that planar partition validation is often more computationally expensive than repair, and the two are not necessarily connected. For the prototype developed, in order to give meaningful validation messages, it was often necessary to deviate from a given repair task. Future work might benefit from separating these two operations entirely.

Meanwhile, the following points merit further work on the topic:

Optimisations for simpler polygons The algorithms developed have been designed to perform best with big polygons consisting of many vertices, such as land cover data sets. However, due to the same fact, they perform poorly on data sets which

consist of simple polygons with just a few vertices each, such as cadastral data. Optimisations to the algorithms could be made to better handle these polygons, such as a different indexing scheme for edges in topology generation.

Improved algorithms for extracting polygons from a triangulation The slowest step in the planar partition repair process is the extraction of simple features polygons from the triangulation, taking anywhere from 30% to over 90% of the total running time. It is an intricate problem to generate these in the correct order and orientation, and doing so robustly and with all their corresponding holes. However, this can be improved by generating points for multiple polygons at the same time (e.g. for the polygons laying on both sides of a given edge), by keeping more ancillary information in the triangulation vertices and edges, or by removing edges from the triangulation data structure directly.

Eliminating memory limitations As seen in Chapter 5.3, using this method is quite memory intensive, which is due to the need to store very large triangulations with added tags for each. While the CGAL implementation of constrained triangulations is quite fast, memory usage is not optimal, limiting the size of the data sets that can be processed while using it. Improvements can be done on two fronts: reducing the memory footprint of the triangulation [Blandford et al., 2005], or implementing techniques to keep only a portion of the (constrained) triangulation in main memory at a time (e.g. streaming) [Isenburg et al., 2006].

Improving the ordering of point insertions During the creation of the triangulation, points are now inserted in the same order as they are in the input files, which is not optimal. A randomised, or known order approach would be better [Amenta et al., 2003].

Extension to 3D Most of the algorithms developed for this thesis extend well to 3D, so that tagged tetrahedralisations could be used for validation and repair of 3D volumes. Higher dimensions could also be possible, such as 3D plus time, but would be more complex.

Implementation in a database Because of the large planar partition data data sets that are frequently used and the relatively simple operations required for the algorithms developed, their implementation in a database would be a natural next step. However, the limitations of topological operations in spatial databases should be taken into account [Zlatanova and Stoter, 2006]. At the very least, a database could serve as off memory storage with good spatial indexing, so parts of the data set could be loaded, processed and dumped into the database. This would make it possible to process much larger data sets than currently possible, and to have map overlay and data cleaning operations for data already stored in a database.

Design Considerations and Implementation Checks

This appendix contains relevant information for the understanding of the algorithms, but which was considered to be too low level to be put into the main text. With the information here presented, it should be possible to re-implement the algorithms developed, if necessary.

Therefore, to give an overview, Appendix A.1 starts with the data structures specifically designed for this prototype. Later, Appendix A.2 specifies which CGAL data structures were used, and how were they defined. Afterwards, Appendices A.3 to A.9 contain low level description of the more complex algorithms in the prototype.

A.1 DATA STRUCTURES DESIGNED

Within this appendix, the main data structures generated for the prototype are presented. Figure A.1 shows the general structure of the prototype, Figure A.2 shows all the required information to have quick generation of closed rings, as discussed in Appendix A.7, and Figure A.3 has the data structures used for the generation of topology of planar partitions.

A.2 TRIANGULATION DATA STRUCTURES

CGAL's constrained Delaunay triangulation using the Delaunay hierarchy is used [see Boissonnat et al., 2002], which ensures the fastest point locations readily available in the library and therefore the fastest point and edge insertions as well. The increase in memory footprint from the use of this additional data structure is negligible when compared to the already large triangulations generated [Devillers, 2002].

For the storage of the tags (`PolygonHandle`) assigned to each triangle in the triangulations, a different scheme is used depending on whether there is a single one, multiple ones, or none. When none, it works as a pointer with `NULL` value; when one, it points to

A. DESIGN CONSIDERATIONS AND IMPLEMENTATION CHECKS

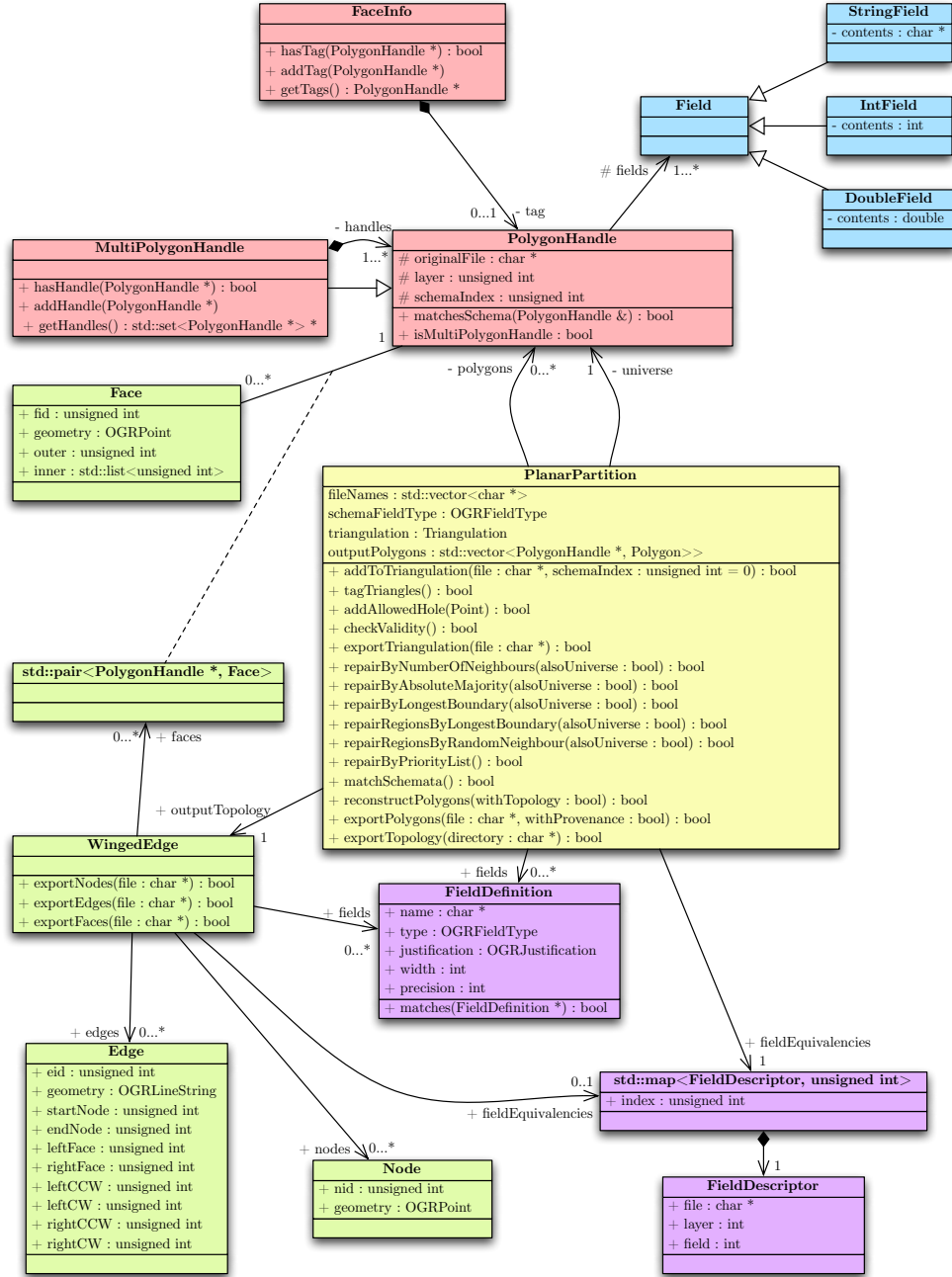


Figure A.1: UML diagram with a simplified overview of the main classes used in the software prototype.

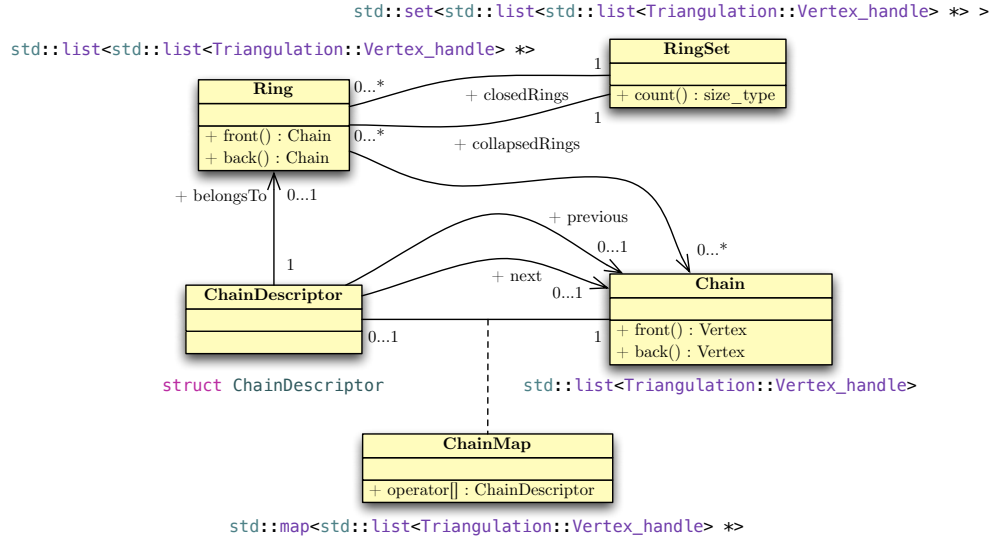


Figure A.2: UML diagram of the data structures used to generate closed rings, with only the main access functions included. The definition of each class in the prototype developed is included as well.

the data structure storing all polygon information from the input; and when multiple tags are required, it points to an STL set which keeps all tags (MultiPolygonHandle), themselves pointers to where the polygon data is stored. In this manner, there is fast access to multiple tags without incurring in the memory overhead associated when there is a single or no tags [see Austern, 2000].

Meanwhile, the other type definitions used are presented in Table A.1.

Table A.1: CGAL Type definitions used in the prototype.

Type	Definition
Kernel	Exact predicates but inexact constructions
Triangulation vertex base	Triangulation vertex base with Delaunay hierarchy
Triangulation face base	Constrained triangulation face base with info (FaceInfo)
Triangulation type	Constrained Delaunay triangulation
Point	from triangulation
Segment	from triangulation
Ring	CGAL::Polygon_2
Polygon	CGAL::Polygon_with_holes_2
Polygon sets	CGAL::Polygon_set_2
Intersection tests	CGAL::Object

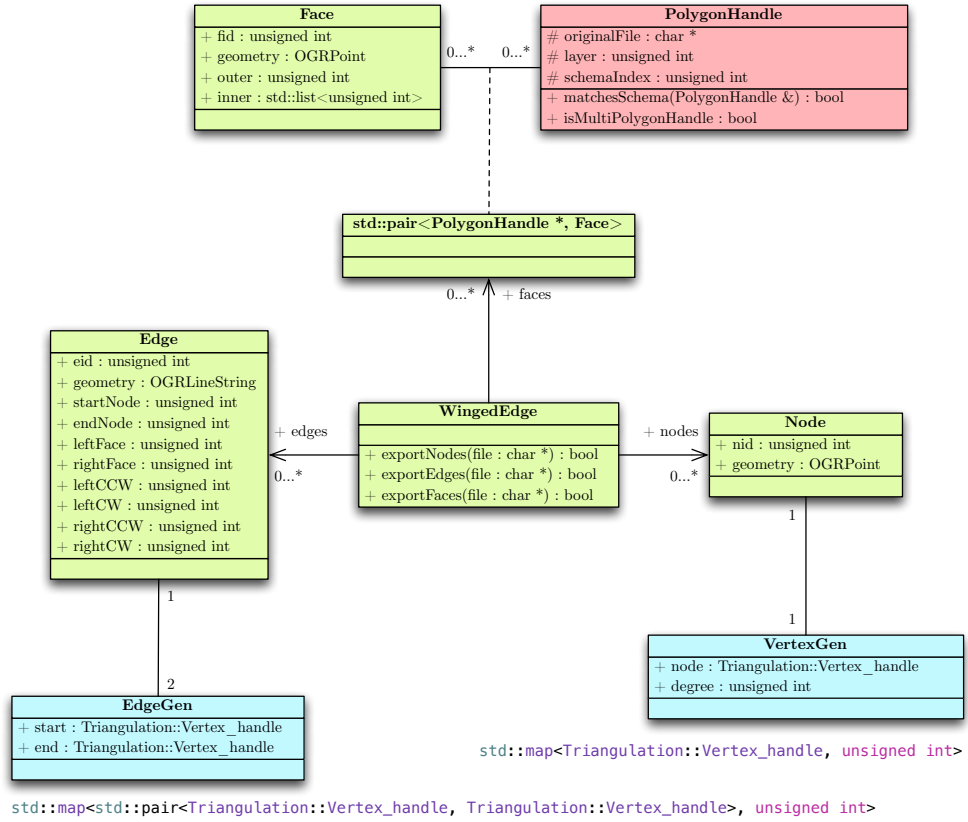


Figure A.3: UML diagram of the data structures used to generate topology. The classes in blue are temporary, meant to find out if a vertex or face has been already put into the data structure.

A.3 DECIDING WHETHER TO TRIANGULATE AND RECONSTRUCT A RING

As an optimisation, it is good to avoid having to triangulate and reconstruct every ring in a polygon. Therefore, a simple method has been determined to decide which rings should be processed in this manner.

All rings are first processed to remove duplicate vertices, with those having less than 3 total vertices being removed completely (since they are always zero-area), since this check is quick and can be done in $O(n)$. However, since triangulating and reconstructing a ring is more computationally expensive, and CGAL has a fast $O(n \log n + I \log n)$ sweep-line algorithm to decide whether a ring is simple, with n being the number of vertices and I the number of intersections, it is used to decide whether to triangulate and reconstruct a ring. Since a simple polygon ring is, by definition not self-touching, there is no need to process these rings, which in a normal planar partition data set should constitute the majority¹.

¹If invalid polygons constituted the majority, it would be better to avoid this check altogether and

A.4 REMOVING DUPLICATE (CONSTRAINED) EDGES

Sometimes, a polygon will be self-intersecting along an edge, or a section of it. This is an important case to check for, since when it happens, the exterior or the interior of the polygon will be on both sides of the edge, breaking the algorithms for triangle tagging later on. Although it is not a common occurrence in a digital environment, the potential for disrupting the algorithms later on makes it necessary to perform this check.

In general, passing through an edge an even number of times is equivalent to creating zero-area features when passing back and forth through the same points, which means that edge that are passed over an even number of times are removed. However, it is important to take into account what happens when the line segments are not identical (see Figure A.4). In this case, edges have to be split at the vertices located in the interior of the other edge, and only the section with an even number of passes should be removed.

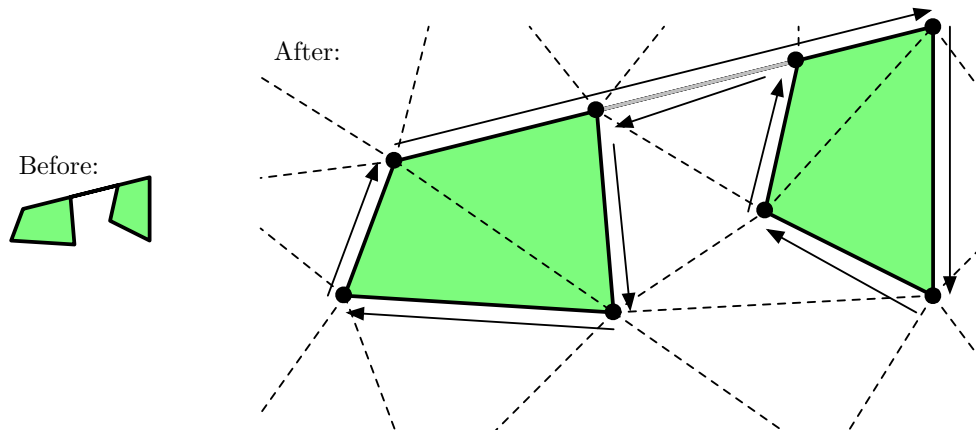


Figure A.4: When one edge lies in the interior of another one, it is split at the endpoints of the shorter line. Note that the long edge at the top goes directly from end to end and that there is a section that has been removed (in grey). The shape of the original polygon is shown on the left.

To ensure numerical robustness in this case, it is important to check whether any of the end points of the two edges are the same first, since computing their intersections directly is not a robust operation. Still, with help of the Delaunay hierarchy in the implementation, performing this check on a point can be done in $O(\log n)$ time or $O(n \log n)$ in total, with n being the total number of points in the triangulation.

A.5 ALGORITHM TO RECURSIVELY TAG EXTERIOR AND INTERIOR OF A RING

Input: A face IF of the triangulation of a ring in its exterior.

Output: A tagged triangulation, where each triangle is marked as being part of either the interior or the exterior of the ring.

```

1 Push  $IF$  into a stack  $ES$ , with another stack  $IS$  still empty
2 while  $IS$  and  $ES$  are not empty do
3   if  $IS$  is not empty then
4     Pop a face  $F$  from  $IS$ .
5     foreach neighbour  $N$  of  $F$  not processed do
6       Mark  $N$  as processed.
7       if there is a constrained edge between  $F$  and  $N$  then push  $N$  into  $ES$ .
8       else push  $N$  into  $IS$ .
9     end
10  else
11    Pop a face  $F$  from  $ES$ .
12    foreach neighbour  $N$  of  $F$  not processed do
13      Mark  $N$  as processed.
14      if there is a constrained edge between  $F$  and  $N$  then push  $N$  into  $IS$ .
15      else push  $N$  into  $ES$ .
16    end
17  end
18 end

```

Based on the information that a ring must be bounded, the infinite face of a triangulation, or a face in its exterior is known to lie outside of a ring. Therefore, and knowing that all edges along the boundary² are the constrained edges, it is also known that stepping over a constrained edge is equivalent to changing from the exterior to the interior of the polygon ring (and vice versa), as it is shown in Figure A.5.

Because of this, it is possible to recursively tag each triangle, starting from the infinite triangle as exterior, using the same tag for adjacent triangles when they do not have a constrained edge in between, and changing tags (exterior to interior or vice versa) when passing through a constrained edge, as long as care is taken to ensure that no constrained edges between exterior only or interior only remain (discussed in Appendix A.4).

Implementation-wise, two stacks of triangles can be used in a recursive traversal of all connected faces, one to keep the interior ones, and one to keep the exterior ones. Other implementations could work with a single stack. The computational complexity of this procedure is therefore $O(n)$, with n being the number of triangles in the triangulation³.

²The boundary denoting the edges adjacent to the interior and exterior.

³As long as it can be checked whether a face has been processed or not in constant time (e.g. by removing the tags in it).

A.6. Algorithm to generate a correctly oriented polyline with all boundaries of a polygon

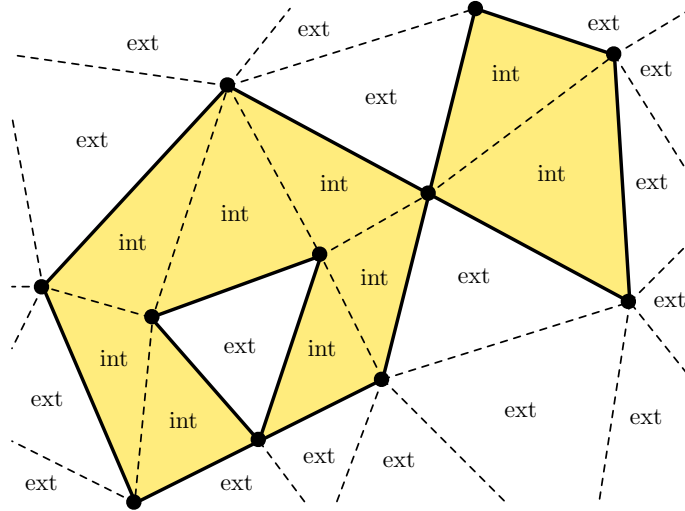


Figure A.5: Once duplicate edges are removed, traversing faces while passing once through a constrained edge is equivalent to moving from the exterior to the interior of a ring, or vice versa.

A.6 ALGORITHM TO GENERATE A CORRECTLY ORIENTED POLYLINE WITH ALL BOUNDARIES OF A POLYGON

Input: A face F of the triangulation in the interior of a polygon.

Output: An ordered list of vertices L representing a polyline, which contains all the boundaries of the polygon and has its interior always to the right, possibly including bridges joining inner and outer boundaries.

```

1 foreach edge  $E$  of  $F$  in clockwise order do
2   if  $E$  is not constrained and the neighbour  $N$  of  $F$  along  $E$  has not been processed
   then
3     Append to  $L$  the results of applying the algorithm recursively with  $N$ .
4     Append to  $L$  the vertex clockwise from  $E$ .
5   end
6 end

```

In order to generate a polyline passing through all the boundaries of a polygon, a face from the triangulation that is known to be in the interior of the polygon is required. From this face, the algorithm is seeded with each of its three incident edges and appended its three defining vertices in a clockwise order, as shown in Figure A.6.

The algorithm works by recursively performing a depth-first search of edges on the boundary of the polygon, going clockwise whenever possible. From a seeding face and edge (see Figure A.7), the algorithm gets the results from applying itself to the clockwise face, then appends the opposite vertex to them, and appends the results from applying itself to the counterclockwise face. The clockwise traversal of the triangles belonging to

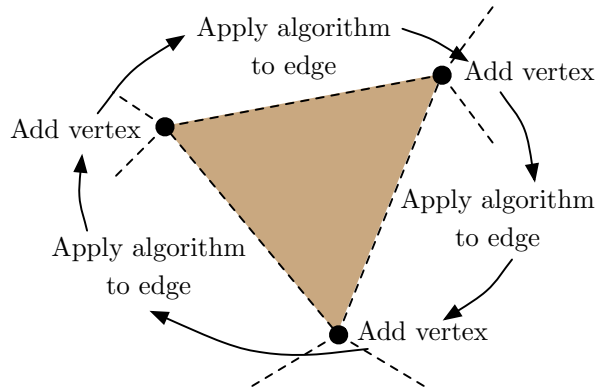


Figure A.6: From a starting face, the algorithm is applied to its three edges in a clockwise order, interleaving the inclusion of the vertex between the edges.

a polygon is best exemplified in Figure A.8, while the step by step polyline generation is in Figure A.9.

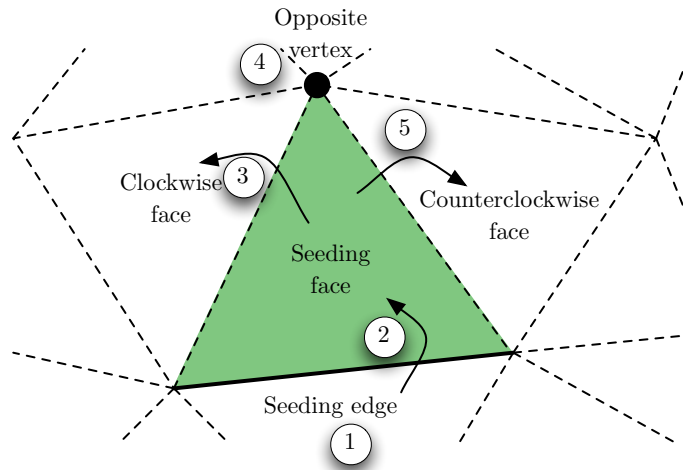


Figure A.7: From a seeding face (2) arrived through a seeding edge (1), the algorithm is recursively applied to the clockwise face (3), then the opposite vertex (4) is added, and then it is also recursively applied to the counterclockwise face (5). This produces a constant clockwise turn, whenever possible, which gives the boundary in clockwise order.

Based on this clockwise traversal, a long polyline containing all boundaries reachable through the interior of the polygon from that triangle is created. Because of the clockwise order, the polyline has the correct orientation for each boundary already, but it also contains “bridges” joining internal and external boundaries, and possibly other

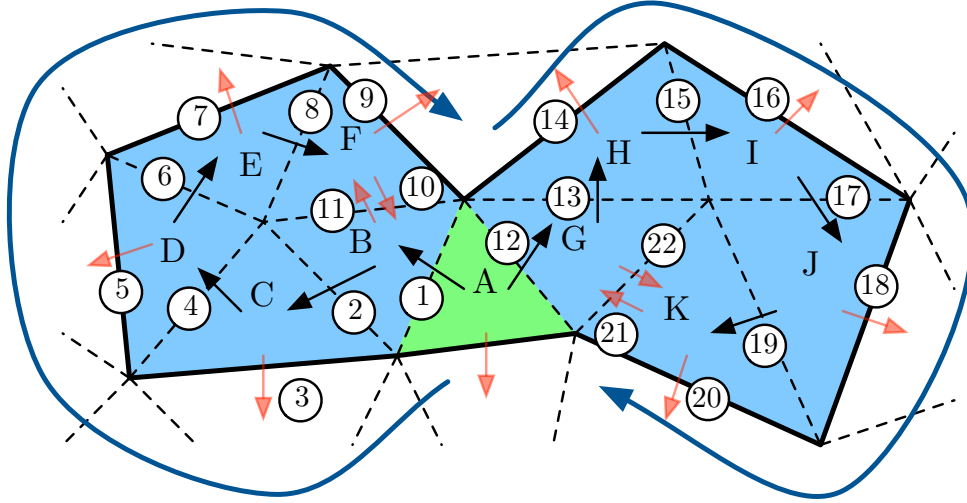


Figure A.8: The traversing order when navigating through triangles in a clockwise manner. Starting at $\triangle A$ and the edge between $\triangle A$ and $\triangle B$, the following occurs: (1) Move to $\triangle B$, (2) Check CW: $\triangle C$, Move to $\triangle C$, ..., (8) Move to $\triangle F$, (9) Check CW: exterior, (10) Check CCW: visited, Move back to $\triangle E$, ..., Move back to $\triangle A$. Similarly, when starting from $\triangle A$ and the edge between $\triangle A$ and $\triangle G$: (12) Move to $\triangle G$, (13) Check CW: $\triangle H$, Move to $\triangle H$, ..., (19) Move to $\triangle K$, (20) Check CW: exterior, (21) Check CCW: visited, Move back to $\triangle J$, Move back to $\triangle I$, ..., Move back to $\triangle A$. Notice how the traversal is clockwise (shown in dark blue arrows) for both cases, despite starting from different sides.

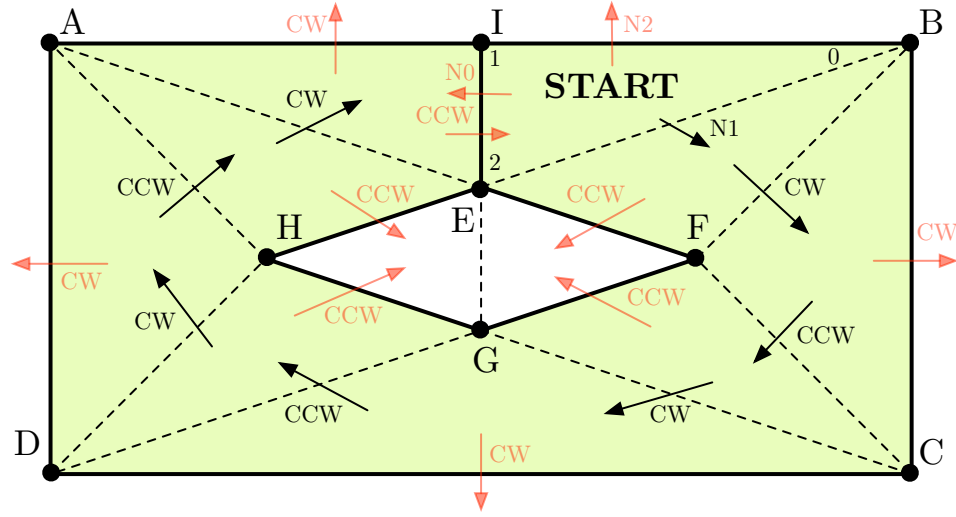
triangles⁴. Specifically, the relevant properties of this polyline are summarised in Table A.2. Meanwhile, Figure A.10 shows an example of such a polyline from the CORINE data set.

Table A.2: Properties of the polyline passing through all boundaries of a polygon.

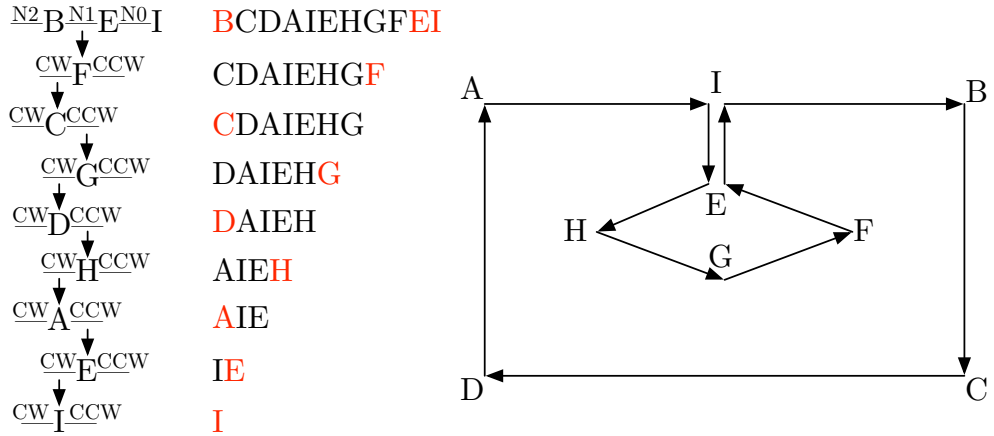
Ready for output	To be corrected
Interior is connected	Possible “bridges” to inner boundaries
Passes through all boundaries	Possible “bridges” to interior triangles
Correct orientation	Multiple polylines for multi polygons

For the implementation, using a stack for the triangle traversal and ensuring that triangles are only visited once (e.g. by removing the tags in the triangle as it is visited), the polyline is computable in $O(n)$, with n being the number of triangles in the inte-

⁴Such as the seeding face, if it is not in the boundary of the polygon, as discussed in Appendix A.7.

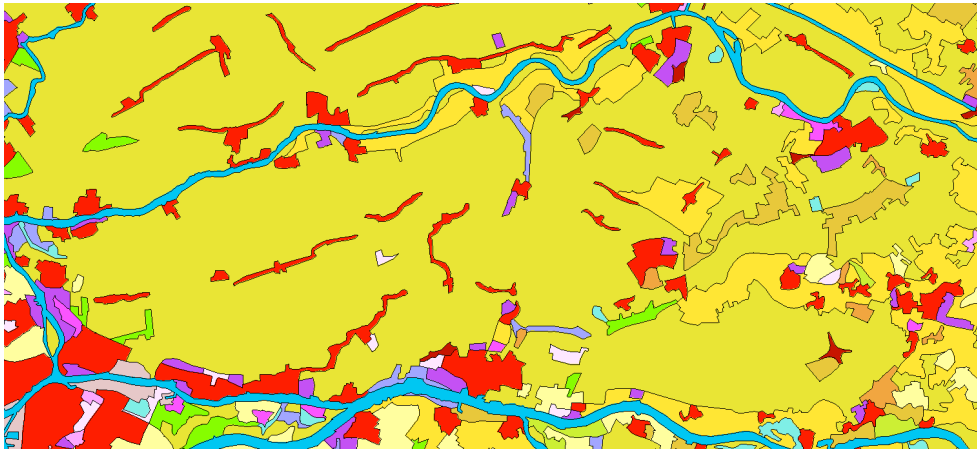


(a) The triangle traversal



(b) The polyline generated

Figure A.9: Step by step demonstration of the polyline generation algorithm. It starts at the seeding triangle, from which the algorithm is applied to all its three incident edges, although all the polyline is generated from N1 and following the traversal order shown in black arrows (above). The nodes describing the polyline at each step of the algorithm are in the lower left (with the vertex added at each point of the recursion in red), while the final result is in the lower right. Notice that the interior of the polygon always lies to the right of the polyline.



(a) The original polygon



(b) The polyline generated from it

Figure A.10: The polyline generated by this algorithm in polygon 67 of CORINE tile E39N32. Note how the interior is connected and all holes are part of the polyline as well.

rior of the triangulation⁵. If we wish to generate a polygon oriented counterclockwise instead, the final list of vertices could simply be read from back to front.

A.7 ALGORITHM TO CREATE VALID RINGS FROM A POLYLINE WITH ALL BOUNDARIES OF A POLYGON

Input: An ordered list of vertices L conforming a polyline, which contains all the boundaries of the polygon and has its interior always to the right.

Output: A set of closed rings S , which preserve the orientation from L and has all “bridges” removed.

- 1 Collapse “bridges” in L connecting triangles in the interior to the outer boundary.
- 2 Compute the *degree* of each vertex in L , only taking into account the edges $\in L$.
- 3 Cut the polyline at the vertices with *degree* > 2 , into a list of shorter polylines P .
- 4 **foreach** polyline $p \in P$ **do**
- 5 | Check if p is a closed ring.
- 6 **end**
- 7 **while** P is not empty **do**
- 8 | Find a closed polyline $p \in P$.
- 9 | Remove p from P and insert it into S .
- 10 | Collapse the neighbouring polylines of p , if possible.
- 11 **if** the degree of the end points of $p \leq 2$ **then**
- 12 | Join the neighbouring polylines of p .
- 13 **end**
- 14 | Check if the joined polylines form a closed ring.
- 15 **end**

Based on the long polyline passing through all boundaries of a given polygon, as generated by the algorithm detailed in Appendix A.6, individual closed rings are generated, all while preserving the correct orientation of the polyline. This allows for the export of polygons represented as Simple Features.

However, as stated previously in Table A.2, some situations have to be dealt with. First of all, undesired “bridges” connecting triangles in the interior to the exterior, such as the one shown in Figure A.11, must be removed. To achieve this, these bridges are collapsed around a central point, working as a pivot from which surrounding pairs of points with the same coordinates cause the removal of the central point and one of the surrounding ones, while the other remaining point becomes the new pivot. This procedure, which is best shown in Figure A.12, is run for each point in the polyline.

Afterwards, the degree⁶ of every vertex that is part of the polyline is computed,

⁵For this to be strictly true, it is also necessary to ensure that the data structure that keeps the list of vertices in the polyline has constant access to the last element, and that two lists are able to be joined in constant time as well (e.g. by using doubly linked lists with a pointer to the last element, such as the one provided in the STL `list`).

⁶The number of incident edges to a certain vertex.

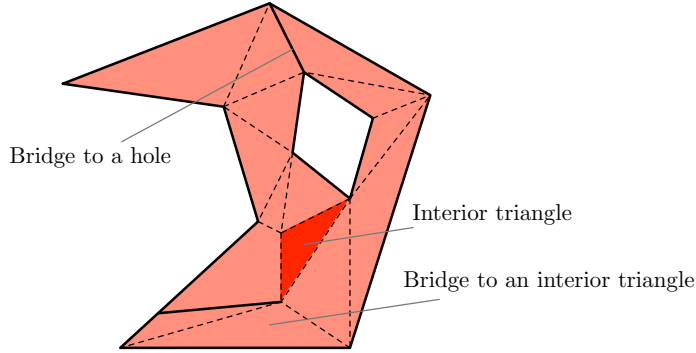


Figure A.11: “Bridges” are generated connecting interior triangles and holes to the outer boundary of a polygon.

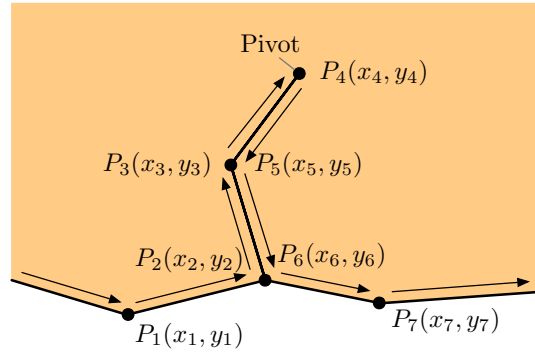


Figure A.12: Whenever the surrounding points of a central point have the same coordinates, it causes the collapse of a “bridge” connecting the outer boundary to a triangle in the interior of a polygon. When the pivot has reached P_4 , P_3 and P_5 are compared, and since they are identical, P_3 and P_4 are removed, while the pivot moves on to P_5 . There, P_2 and P_6 are compared and found to be equal as well, causing the removal of P_2 and P_3 . The final polyline is therefore $(\dots, P_1, P_6, P_7, \dots)$.

only taking into account the edges that are part of the polyline. With this, edges with a degree larger than 2 are identified as junction points for different pieces of the polyline, which is therefore divided at those points.

All pieces of this polyline are then checked to see if they are closed rings already, which happens when the next piece starts with the same vertex as the current piece⁷.

Afterwards, the algorithm consists of continuously removing already closed rings, which are already ready for the output, and performing some operations on the neighbouring polylines from these. First, if they have identical vertices and opposite direction, they are collapsed with the same procedure mentioned above, eliminating the “bridges” connecting them to the outer boundary or to other rings. Then, if the degree

⁷Assuming that the polyline is a loop as well, and therefore the last piece is adjacent to the first one.

of the vertex that was the start and end point of the removed ring is smaller or equal to two (i.e. no longer a junction), the neighbouring polylines are joined together⁸. Otherwise, the degree of the vertex is updated⁹. Finally, it is checked whether these form a closed ring, which would therefore be processed as such. Figure A.13 shows an intermediate state of the ring reconstruction algorithm.

This algorithm continues until all polylines are either collapsed or have been converted into closed rings, which since their order is never modified, keep the correct orientation from the algorithm from Appendix A.6. This orientation can be used to tell whether the newly created rings form outer or inner boundaries.

Implementation-wise, many relationships between data structures need to be stored in order to keep the algorithm efficient (e.g. pieces to rings, pieces to metadata, pieces to previous and next pieces), so a fast indexed data structure is needed to locate rings in memory. Also, it is preferable to have unfinished rings only as virtual data structures and only build them when completely finished, to avoid doing unnecessary operations in this respect. Appendix A.1 includes the data structures used for this in Figure A.2.

The time complexity of this algorithm is hard to assess, since it is dependent on many implementation details. Assuming that we have a list-like pointer data structure with constant time appendage and removal from the back and front, constant time access to the relations described in Figure A.2 and $O(\log n)$ access to the sets of rings and polylines themselves, it is possible to give some generalised complexities. For individual parts of the long polyline processing, bridge collapse can be generally done in $O(n)$, degree computation in $O(n \log n)$ using a binary tree or $O(n)$ using data in the vertex itself¹⁰, cutting the polyline in $O(n)$ if the high degree vertices are not kept, with n being the number of vertices in the polyline. For the rest of the algorithm, it is dependent on the number of polylines generated instead (p), being of complexity $O(p \log p)$. Therefore the dominating complexity can come from different parts of the algorithm. For the prototype, it is $O(n \log n + p \log p)$, although usually $n \gg p$.

A.8 FINDING THE NESTING OF INNER AND OUTER BOUNDARIES

To find out the nesting of every inner boundary within the outer ones, a point-in-polygon test was made for every vertex of each inner boundary within the outer boundaries. To do so efficiently, the same triangulation data structures used previously might be used, although for simplicity of programming and testing, a new triangulation was used in the developed prototype.

⁸This check is necessary, since when the degree is larger than two it means that there is another bridge or ring connecting at this vertex, creating ambiguity as to whether the next polyline is part of the same ring (to be joined), another ring (to be put in the output and removed) or a bridge (to be removed).

⁹Because a ring or bridge was removed at this location, reducing the number of incident edges. It makes no sense to always update it, since the value is not used anymore when the neighbours are joined together.

¹⁰Which has a substantial impact on memory, since instead of keeping only a degree for the vertices of this polygon, it would keep one for all the ones in the entire triangulation.

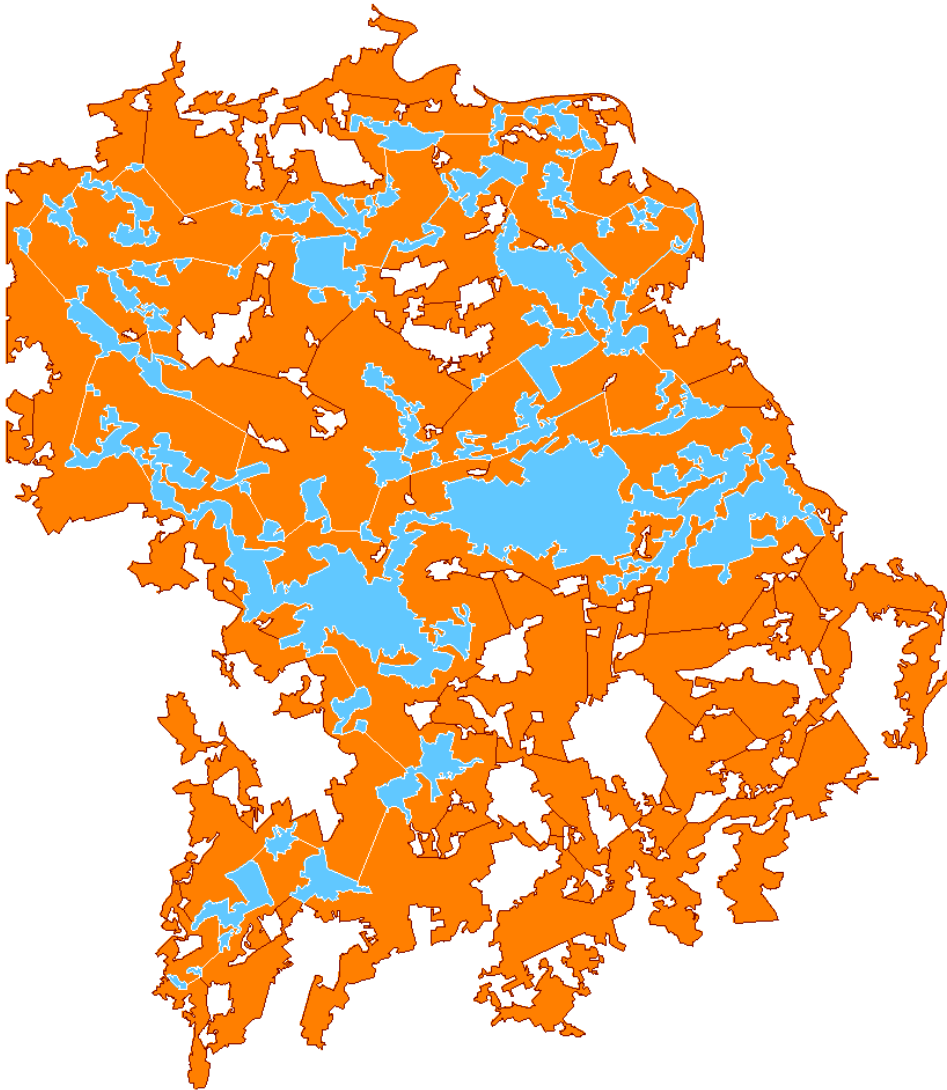


Figure A.13: An intermediate state in the ring reconstruction algorithm. In this example from polygon 1752 from CORINE tile 1752, part of the inner rings (in blue) have been reconstructed, while the outer boundary and other inner rings remain as open pieces of the long polyline.

This is a solution that works in all cases that might be commonly expected, without the computational complexity of having to compute much more difficult intersection tests. However, these are cases where holes would not be detected as being inside a certain outer boundary, as is shown in Figure A.14.

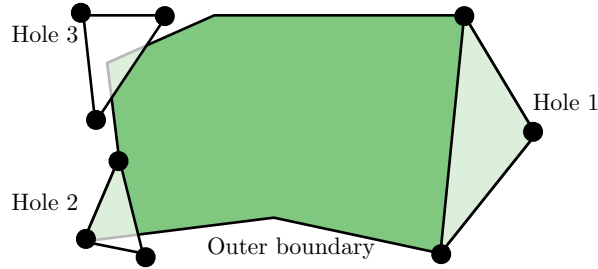


Figure A.14: Using the point in polygon test, some holes might not be correctly detected as being inside the outer boundary. Specifically, those that are comprised of successive vertices of the outer boundary (hole 1), those with some successive vertices of the outer boundary and some outside it (hole 2), and those with all vertices outside the outer boundary, but with an edge crossing the outer boundary an even number of times (hole 3).

These problems can be solved by performing additional point-in-polygon tests for areas inside triangles (e.g. the centroid), although that can cause problems with numerical robustness; or by checking for edge splitting during the construction of the triangulation¹¹. Nevertheless, this sort of holes are uncommon and the original intent behind them is hard to discern, which makes discarding them also a reasonable solution¹².

An average computational complexity of this algorithm is hard to ascertain, since it depends on the probability of a vertex from an inner ring being within an outer ring. However, the absolute worst case is $O(N \log n)$ with N being the total number of points in the inner rings and n the total number of points in the outer rings. While this could become an issue with polygons with an extraordinary number of points, it is never a problem in the tested data sets. If it was, a selective test beforehand could be a good improvement (e.g. a bounding box intersection one). Nevertheless, since in a normal planar partition data set the memory to store such polygons would likely become a limitation much sooner, further possibilities for enhancing this algorithm are not investigated.

A.9 SCHEMATA MATCHING CRITERIA

For the implementation of schemata matching in the developed prototype, all fields from each polygon entered are stored independently, together with additional informa-

¹¹Caused when edges cross.

¹²If rings were known not to intersect each other, it becomes a much simpler problem. See Bajaj and Dey [1990] for a good solution in that case.

tion for each (file, layer, field number, name, type, justification, width and precision). As a new polygon is created, its fields are matched with the previously known ones by name and type. When an exact match (in both name and type) is found, an equivalency record is created and kept until the polygon reconstruction phase. The data structures used for this can be seen in more detail in Figure A.1 in Appendix A.1 (in the `FieldDefinition` and `FieldDescriptor` classes and the map relationship indicating the field index number, shown in purple).

Meanwhile, to solve the problem that polygons with the same ID¹³ should be matched only in some cases, this operation has been split into a separate function `matchSchemata()`, which looks at the equivalency records, creates new polygon metadata which combines all fields from matched polygons, and assigns them this new metadata for reconstruction.

Finally, when the algorithms to reconstruct polygons from the triangulation are called, these equivalency records may be used for different purposes:

- To reconstruct seamless polygons across the boundaries of different data sets.
- To create joint feature classes from formerly separate ones.
- To repair polygons only, using the file of origin information to distinguish features sharing the same IDs in different data sets.
- To compute statistics on the newly created planar partition (e.g. number of disconnected components).

¹³Or any other field used for matching.

B

Completeness and Correctness of Planar Partitions Repair

In this appendix, the main possible invalid situations that occur are listed, with the way that they are handled in the developed algorithms. The situations are divided into those related to points (Appendix B.1), edges (Appendix B.2), rings (Appendix B.3), and polygons (Appendix B.4).

B.1 POINTS

Repeated points Duplicates are eliminated without giving any error, since they are allowed in some representations (e.g. Shapefiles), and commonly present when the first and last point of a polygon are required to be the same.

B.2 EDGES

Vertex on an edge If it is exactly in the interior of an edge, it splits it at that point, creating two edges from it.

Crossing edges A new vertex is generated at their intersection, except when they cross at the start or end point of either of the edges, in which case it is not necessary.

(Partially) overlapping edges The overlapping sections are eliminated when passed over an even number of times. If they are odd, a single instance of the section is kept.

B.3 RINGS

Open rings They are closed by joining their end and start points.

Zero area rings Are eliminated whether they represent outer or inner boundaries. If all outer boundaries are eliminated, the related inner boundaries are eliminated as well.

Spikes and bridges They are eliminated and a self-intersection error is reported.

Self-intersecting As long as it is not only self-touching, multiple rings are formed. A self-intersection error is also reported.

B.4 POLYGONS

Zero area polygons Are eliminated, including both its outer and inner boundaries.

Touching holes Are joined, if possible (i.e. when they are touching along a section of an edge).

Holes (partially) outside an outer boundary If they are partially outside, they are eliminated, since it is dubious which outer boundary they belong to. If they are completely outside but completely inside a different outer boundary, they are assigned as holes of it. Otherwise, an error is given and they are eliminated.

Overlapping holes They are joined together.

Data Sets Used

In this Appendix, the most important data sets referred to in this thesis are described. Appendix C.1 deals with the simple polygons used for testing specific problems, while Appendix C.2 describes the other important data sets used.

C.1 UNIT TEST POLYGONS

In this appendix, the simple polygons used in Section 5.2 are shown. Black lines describe outer boundaries, red lines inner boundaries, and dark red lines when they both overlap (inner and outer). The same applies for vertices. Arrows are used to disambiguate some cases, with the same colour scheme. The interior of each part of a (multi) polygon is shown in a different colour, while holes are white, and dubious cases in grey.

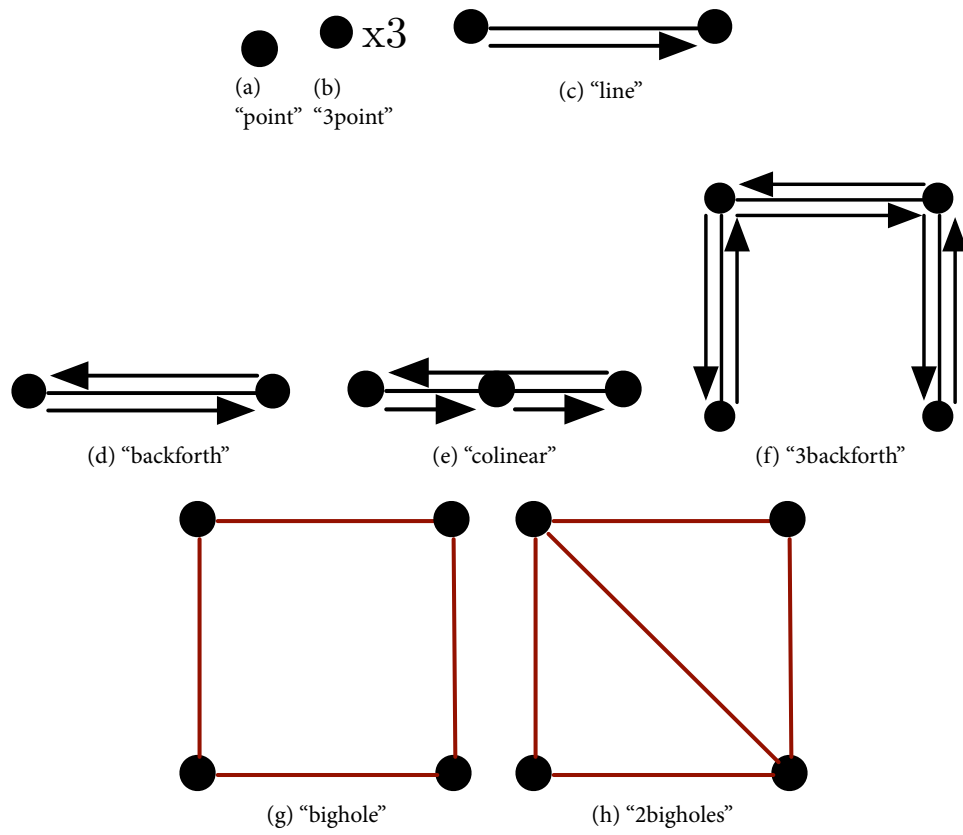


Figure C.1: Zero area unit test polygons.

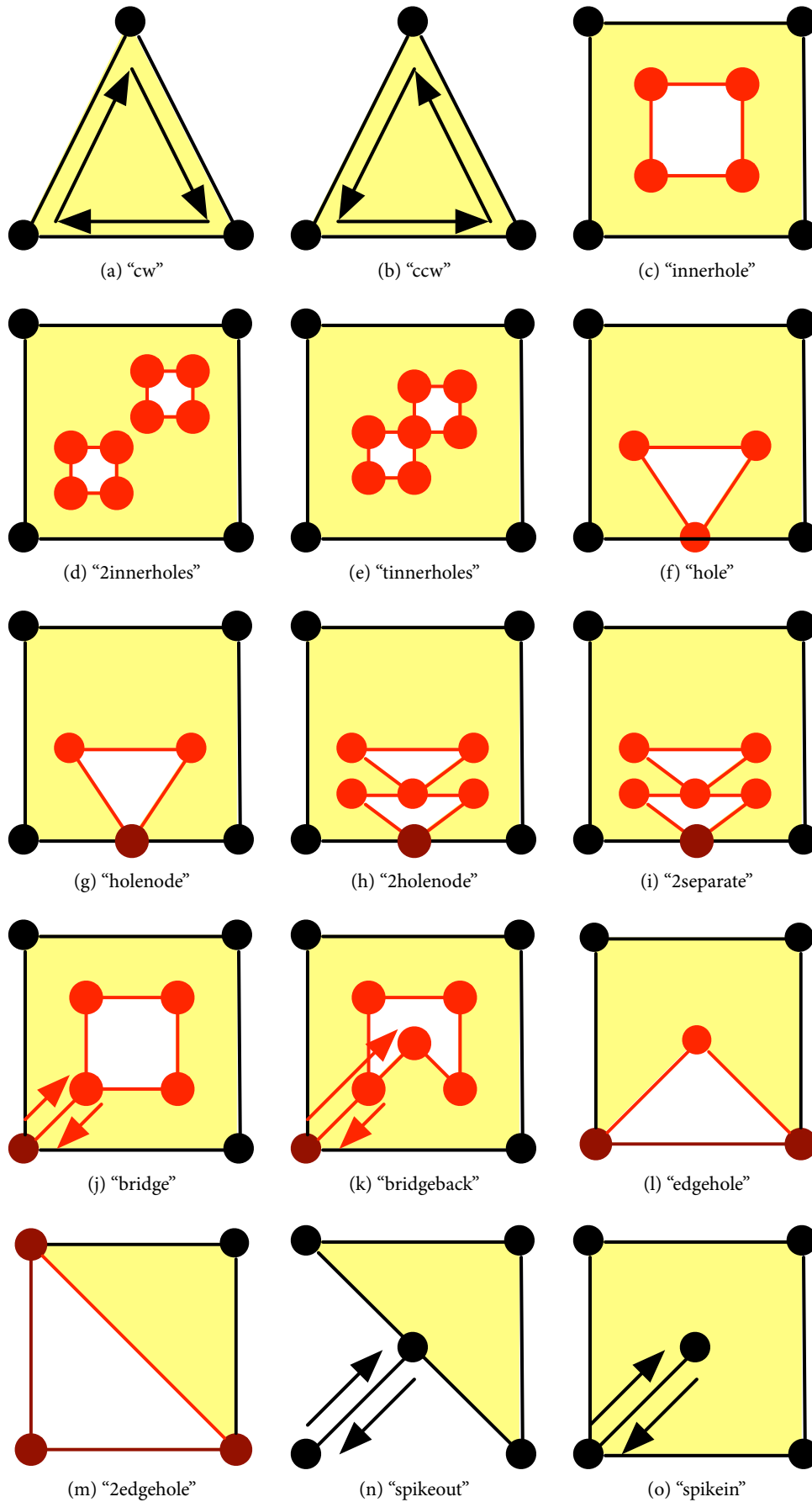


Figure C.2: Unit test polygons with a single interior connected region.

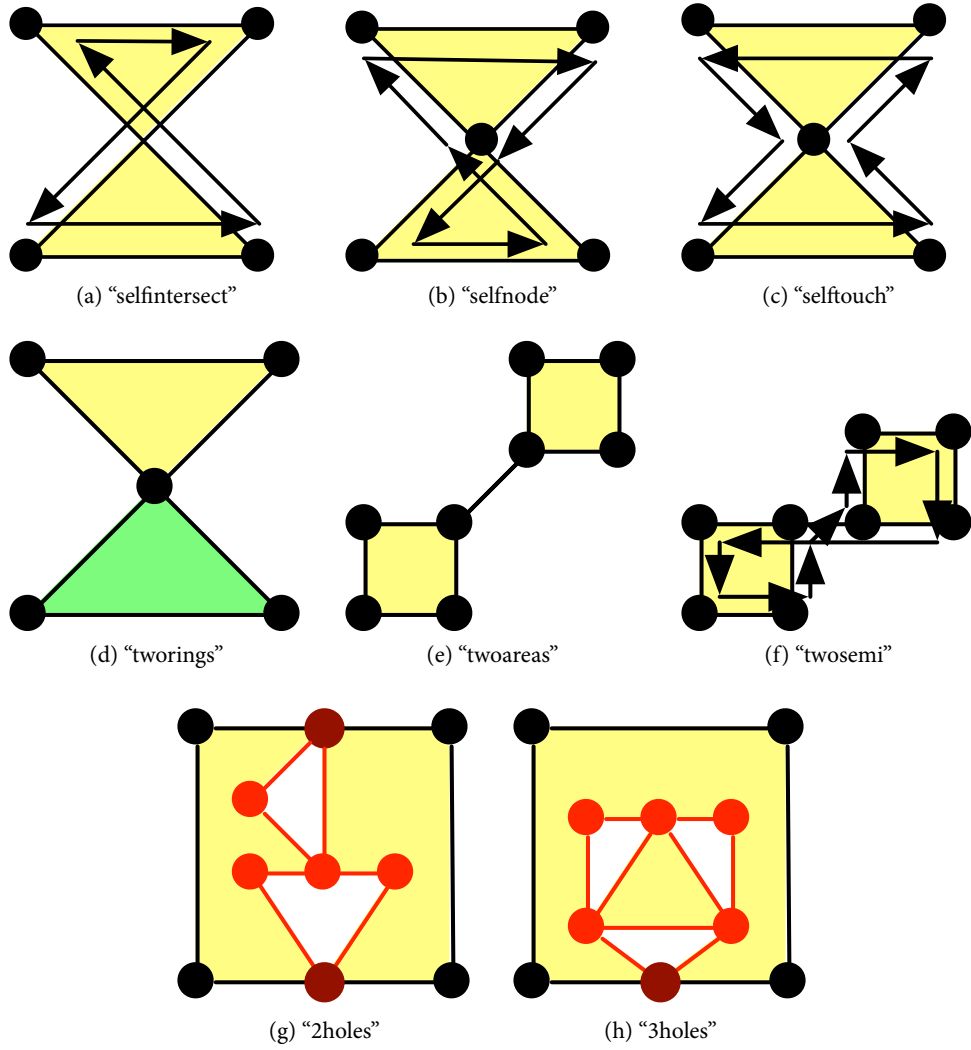


Figure C.3: Unit test polygons with two interior connected regions.

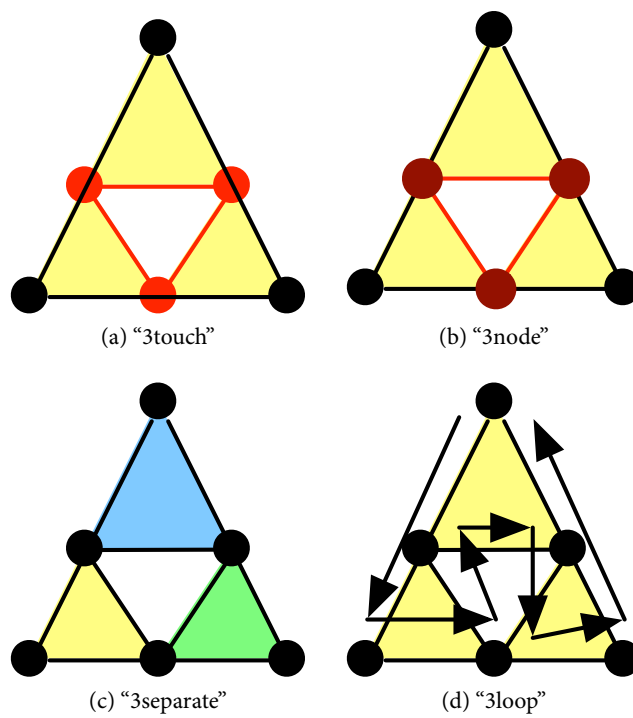


Figure C.4: Unit test polygons with three interior connected regions.

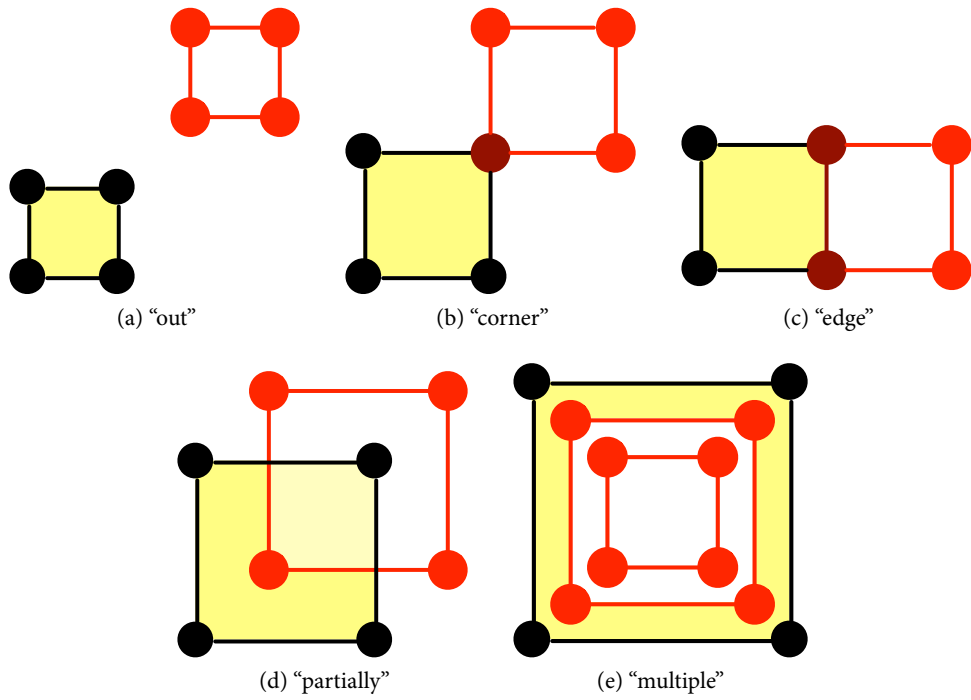


Figure C.5: Unit test polygons with degenerate holes.

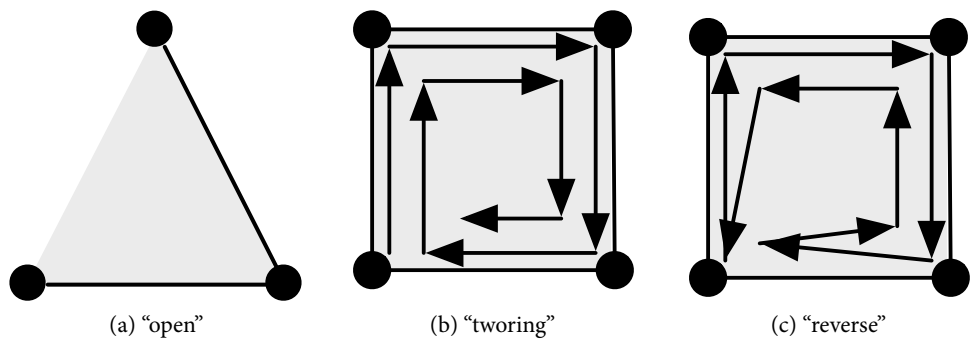


Figure C.6: Unit test polygons with an unknown number of interior connected regions, depending on the interpretation of the polygon.

C.2 LARGE DATA SETS

CORINE

The Coordination of Information on the Environment (CORINE) data set is the result of a programme from the European Commission originally developed from 1985 until 1990. With later revisions in 2000 and 2006, it now contains land cover data for 39 European countries, with a geometric accuracy better than 100 m [EEA, 2007].

For most of the work performed within this thesis, the data set from the year 2000 is used [Bossard et al., 2000]. It comes in two forms: a tiled version in 713 parts and about 2,800 polygons per tile, on average; and a “seamless” version where each land cover type is provided separately. Since the tiled data set allows obtaining all land cover types in a certain region without having to get data from all the files in the data set, it is more useful for the work of this thesis.

While the data set maintains relatively good quality throughout, it does have some problems, which are used to exemplify cases presented throughout this thesis. The most important of these are: overlaps and gaps between tiles, shifted polygons, and modelling inconsistencies (e.g. the use of both multi polygons and separate polygons).

Portugal/Spain Cross-border Data Set

It consists of two polygons for the Arribes del Duero Natural Park in Spain (8643 vertices) and the International Douro Natural Park in Portugal (2428 vertices). Together, they form a binational protected area along the Douro river.

Because of these data sets are independently generated, and border along a natural feature, they form only an approximate match and it provides a very good example of both horizontal conflation and data harmonisation.



Bibliography

- Amenta, N., Choi, S. and Rote, G. [2003], Incremental constructions con BRIO, *in* 'Proceedings 19th Annual Symposium on Computational Geometry', ACM Press, pp. 211–219.
- Anselin, L. [1989], What is special about spatial data? Alternative perspectives on spatial data analysis, *in* 'Proceedings Symposium on Spatial Statistics, Past, Present and Future', pp. 63–77.
- Austern, M. [2000], 'Why you shouldn't use set (and what you should use instead)', *C++ Report* 12(4).
- Baars, M. [2003], A comparison between ESRI geodatabase topology and Laser-Scan radius topology. Case Study.
- Badawy, W. M. and Aref, W. G. [1999], On local heuristics to speed up polygon-polygon intersection tests, *in* 'Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems', ACM, pp. 97–102.
- Bajaj, C. L. and Dey, T. [1990], 'Polygon nesting and robustness', *Information Processing Letters* 35(1), 23–32.
- Barker, S. M. [1995], 'Towards a topology for computational geometry', *Computer-Aided Design* 27(4), 311–318.
- Baumgart, B. G. [1974], *Geometric Modeling for Computer Vision*, Stanford University.
- Beard, K. M. and Chrisman, N. R. [1988], 'Zipper: A localized approach to edgematching', *Cartography and Geographic Information Science* 15(2), 163–172.

- Blandford, D. K., Blueloch, G. E., Cardoze, D. E. and Kadow, C. [2005], 'Compact representations of simplicial meshes in two and three dimensions', *International Journal of Computational Geometry and Applications* 15(1), 3–24.
- Boissonnat, J.-D., Devillers, O., Pion, S., Teillaud, M. and Yvinec, M. [2002], 'Triangulations in CGAL', *Computational Geometry: Theory & Applications* 22, 5–19.
- Bossard, M., Feranec, J. and Otahel, J. [2000], CORINE land cover technical guide – Addendum 2000, Technical report, European Environmental Agency.
- Burns, T. [2001], 'Effective unit testing', *ACM Ubiquity* .
- Burrough, P. A. [1992], *Principles of Geographical Information Systems for Land Resources Assessment*, Oxford University Press.
- CGAL [2010], *CGAL 3.6 User and Reference Manual*, CGAL Editorial Board.
- Chazelle, B. and Dobkin, D. [1979], Decomposing a polygon into its convex parts, in 'Proceedings of the 11th Annual ACM Symposium on Theory of Computing', ACM, pp. 38–48.
- Chrisman, N. R. [1988], 'The risks of software innovation: A case study of the harvard lab', *The American Cartographer* 15(3), 291–300.
- Chrisman, N. R. [1990], 'Deficiencies of sheets and tiles: building sheetless databases', *International Journal of Geographical Information Science* 4(2), 157–167.
- Coppock, J. and Rhind, D. [1991], The history of GIS, in P. A. Longley, M. F. Goodchild, D. J. Maguire and D. W. Rhind, eds, 'Geographic Information Systems and Science', John Wiley & Sons, chapter 2, pp. 21–43.
- de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkopf, O. [2008], *Computational Geometry: Algorithms and Applications*, 3 edn, Springer-Verlag.
- Devillers, O. [2002], 'The Delaunay hierarchy', *International Journal of Foundations of Computer Science* 13(2), 163–180.
- Devillers, O., Pion, S. and Teillaud, M. [2002], 'Walking in a triangulation', *International Journal of Foundations of Computer Science* 13(2), 181–199.
- Deza, M. and Rosenberg, I. G. [1980], *Combinatorics* 79, Vol. 9 of *Annals of Discrete Mathematics*, Elsevier.
- Dupin, C. [1826], 'Carte figurative de l'instruction populaire de la France', *Jobard* .
- EEA [2007], CLC2006 technical guidelines, Technical report, European Environmental Agency.

- Egenhofer, M. and Herring, J. [1991], High-level spatial data structures for GIS, in P. A. Longley, M. F. Goodchild, D. J. Maguire and D. W. Rhind, eds, 'Geographic Information Systems and Science', John Wiley & Sons, chapter 16, pp. 227–237.
- Eppstein, D. [1992], 'The farthest point delaunay triangulation minimises angles', *Computational Geometry* 1(3), 143–148.
- ESRI [1998], Shapefile technical description, White paper, ESRI.
- ESRI [2002], Working with the Geodatabase: Powerful multiuser editing and sophisticated data entry, White paper, ESRI.
- ESRI [2009a], 'ArcGIS Desktop 9.3 help'.
URL: <http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm>
- ESRI [2009b], Topology in ArcGIS, White paper, ESRI.
- Facello, M. A. [1995], 'Implementation of a randomized algorithm for Delaunay and regular triangulations in three dimensions', *Computer Aided Geometric Design* 12(4), 349–370.
- Fournier, A. and Montuno, D. Y. [1984], 'Triangulating simple polygons and equivalent problems', *ACM Transactions on Graphics* 3(2), 153–174.
- Friendly, M. and Dennis, D. [2001], 'Milestones in the history of thematic cartography, statistical graphics, and data visualization'.
URL: <http://www.math.yorku.ca/SCS/Gallery/milestone/>
- GEOS [2010], 'Geometry cleaning'.
URL: <http://trac.osgeo.org/geos/wiki/GeometryCleaning>
- Gold, C. M., Nantel, J. and Yang, W. [1996], 'Outside-in: An alternative approach to forest map digitizing', *International Journal of Geographical Information Science* 10(3), 291–310.
- Goldberg, D. [1991], 'What every computer scientist should know about floating-point arithmetic', *Computing Surveys* 23(1), 5–48.
- GRASS [2006], GRASS GIS 6.5.svn Reference Manual.
URL: http://grass.itc.it/grass65/manuals/html65_user/index.html
- GTS [2006], GTS Library Reference Manual.
URL: <http://gts.sourceforge.net/reference/book1.html>
- Hansen, A. J. and Levin, P. L. [1991], 'On conforming Delaunay mesh generation', *Advances in Engineering Software* 14, 129–135.
- Hoffmann, C. M., Hopcroft, J. E. and Karasick, M. S. [1988], Towards implementing robust geometric computations, in 'Proceedings of the 4th Annual Symposium on Computational Geometry', ACM.

- Isenburg, M., Liu, Y., Shewchuk, J. R. and Snoeyink, J. [2006], 'Streaming computation of Delaunay triangulations', *ACM Transactions on Graphics* 25(3), 1049–1056.
- ISO [2003], '19107: Geographic information – spatial schema'.
- Kettner, L., Mehlhorn, K., Pion, S., Schirra, S. and Yap, C. [2007], 'Classroom examples of robustness problems in geometric computations', *Computational Geometry* 40(1), 61–78.
- Kiehle, C., Heier, C. and Greve, K. [2007], 'Requirements for next generation spatial data infrastructures – Standardized web based geoprocessing and web service orchestration', *Transactions in GIS* 11(6), 819–834.
- Kirkpatrick, D., Snoeyink, J. and Speckmann, B. [2000], Kinetic collision detection for simple polygons, in 'Proceedings of the 16th Annual Symposium on Computational Geometry', ACM.
- Laurini, R. and Milleret-Raffort, F. [1994], 'Topological reorganization of inconsistent geographical databases: A step towards their certification', *Computers & Graphics* 18(6), 803–813.
- Laurini, R. and Thompson, D. [1992], *Fundamentals of Spatial Information Systems*, Apic Studies in Data Processing, Academic Press.
- Ledoux, H. and Meijers, M. [2010], Validation of planar partitions using constrained triangulations, in 'Proceedings Joint International Conference on Theory, Data Handling and Modelling in GeoSpatial Information Science', Hong Kong, pp. 51–55.
- Liu, Y. and Snoeyink, J. [2006], 'Faraway point: A sentinel point for Delaunay computation', *International Journal of Computational Geometry and Applications* 18(4), 343–355.
- Lloyd, E. L. [1977], On triangulations of a set of points in the plane, in 'Proceedings of the 18th Annual Symposium on Foundations of Computer Science', pp. 228–240.
- Louwsma, J. [2003], Topology versus non-topology storage structures: Functional analysis and performance test using LaserScan radius topology. Case Study.
- Lupien, A. E. [1987], A general approach to map conflation, in 'Proceedings of Auto-Carto VIII', pp. 630–639.
- Lynch, M. P. and Saalfeld, A. J. [1985], Conflation: Automated map compilation –a video game approach–, in 'Proceedings of Auto-Carto VII', pp. 343–352.
- MacDonald, A. [2001a], *Building a Geodatabase*, GIS by ESRI, ESRI.
- MacDonald, A. [2001b], *Editing in ArcMap*, GIS by ESRI, ESRI.

- Margalit, A. and Knott, G. D. [1989], 'An algorithm for computing the union, intersection or difference of two polygons', *Computers & Graphics* 13(2), 167–183.
- Meijers, M., Savino, S. and van Oosterom, P. [2010], SplitArea: An algorithm for splitting faces in the context of a hierarchical data structure. Submitted to the ISPRS Journal of Photogrammetry and Remote Sensing.
- Milenkovic, V. [1993], 'Robust polygon modelling', *Computer-Aided Design* 25(9), 546–566.
- Mortenson, M. E. [1999], *Mathematics for Computer Graphics Applications*, Industrial Press.
- OGC [2006], *OpenGIS Implementation Specification for Geographic Information - Simple Feature Access - Part 1: Common Architecture*, 1.2.0 edn.
- Oracle [2009a], *Oracle 11g Release 2 Database Error Messages*, Oracle.
- Oracle [2009b], Oracle locator and Oracle spatial 11g: Best practices, White paper, Oracle.
- Oracle [2010], Oracle spatial 11g release 2 developer's guide, Technical report, Oracle.
- O'Rourke, J. [1998], *Computational Geometry in C*, 2nd edn, Cambridge University Press.
- Peucker, T. K. and Chrisman, N. R. [1975], 'Cartographic data structures', *Cartography and Geographic Information Science* 2(1), 55–69.
- Plümer, L. and Gröger, G. [1996], Nested maps—a formal, provably correct object model for spatial aggregates, in 'Proceedings 4th ACM international workshop on Advances in geographic information systems', ACM, pp. 76–83.
- Plümer, L. and Gröger, G. [1997], 'Achieving integrity in geographic information systems—maps and nested maps', *GeoInformatica* 1(4), 345–367.
- Preparata, F. P. and Shamos, M. I. [1985], *Computational Geometry: An Introduction*, Texts and Monographs in Computer Science, Springer.
- Rivero, M. and Feito, F. [2000], 'Boolean operations on general planar polygons', *Computers & Graphics* 24(6), 881–896.
- Scheidt, J. K. and Schelin, C. W. [1987], Distributions of floating point numbers, in S. Wien, ed., 'Computing', Vol. 38, Springer-Verlag, pp. 315–324.
- Schirra, S. [1997], *Precision and Robustness in Geometric Computations*, Vol. Algorithmic Foundations of Geographic Information Systems of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, chapter 9, pp. 255–287.

- Shewchuck, J. R. [1996], Triangle: Engineering a 2D quality mesh generator and De-launay triangulator, in M. C. Lin and D. Manocha, eds, 'Applied Computational Geometry: Towards Geometric Engineering', Vol. 1148 of *Lecture Notes in Computer Science*, Springer-Verlag, Pittsburgh, Pennsylvania, pp. 203–222.
- Stanton, H. I., O'Meara, S., Chappell, J. R., Poole, A. R., Mack, G. and Hybels, G. [2005], Ensuring data quality using topology and attribute validation in the geodatabase, in 'Digital Mapping Techniques '05–Workshop Proceedings'.
- Tomlin, C. D. [1994], 'Map algebra: One perspective', *Landscape and Urban Planning* 30(1–2), 3–12.
- Tomlinson, R. [1988], 'The impact of the transition from analogue to digital cartographic representation', *The American Cartographer* 15(3), 249–261.
- van Kreveld, M. [1998], 'On fat partitioning, fat covering and the union size of polygons', *Computational Geometry* 9(4), 197–210.
- van Oosterom, P., Quak, W. and Tijssen, T. [2003], Polygons: The unstable foundation of spatial modeling, in 'ISPRS Joint Workshop on Spatial, Temporal and Multi-Dimensional Data Modelling and Analysis'.
- van Oosterom, P., Quak, W. and Tijssen, T. [2004], About invalid, valid and clean polygons, in P. F. Fisher, ed., 'Developments in Spatial Data Handling—11th International Symposium on Spatial Data Handling', Springer, pp. 1–16.
- van Oosterom, P., Stoter, J., Quak, W. and Zlatanova, S. [2002], The balance between geometry and topology, in D. Richardson and P. van Oosterom, eds, 'Advances in Spatial Data Handling—10th International Symposium on Spatial Data Handling', Springer, pp. 209–224.
- Wahl, R. R. [2004], The use of topology on geologic maps, in D. R. Soller, ed., 'Digital Mapping Techniques '04–Workshop Proceedings', pp. 159–164.
- Wiemann, S. and Bernard, L. [2010], Conflation services within spatial data infrastructures, in '13th AGILE International Conference on Geographic Information Science 2010'.
- Worboys, M. and Duckham, M. [2004], *GIS: A Computing Perspective*, CRC Press.
- Wright, D. J., Goodchild, M. F. and Proctor, J. D. [1997], 'Demystifying the persistent ambiguity of GIS as “tool” versus “science”', *The Annals of the Association of American Geographers* 87(2).
- Yuan, S. and Tao, C. [1999], 'Development of conflation components', *Geoinformatics and Socioinformatics* pp. 1–13.
- Yvinec, M. [2010], 2D triangulations, in 'CGAL User and Reference Manual', 3.6 edn, CGAL Editorial Board.

Zlatanova, S. and Stoter, J. [2006], *The role of DBMS in the new generation GIS architecture*, Frontiers of Geographic Information Technology, Springer, chapter 8, pp. 155–180.

Colophon:

This document was typeset using Xe_{La}TeX 0.999.6 on Mac OS X 10.6 using the Minion Pro, Myriad Pro and Inconsolata typefaces, with TeXShop as a front-end and BibTeX for bibliography management.

The document uses Peter Wilson's excellent memoir class, with a modified chapter style based on Danie Els' BlueBox.

Most figures have been hand drawn using OmniGraffle by The Omni Group, except for those displaying data sets, which were exported from QGIS instead.

