Thesis in Geographical Information Technology nr 20

Methodology for creating and modifying distributed topologically structured geographical datasets

Lisette Danebjer

Program in Surveying and Land Management
Faculty of Engineering

Department of Physical Geography and Ecosystem Science
Lund University

Lund University
Faculty of Engineering

Methodology for creating and modifying distributed topologically structured geographical datasets

EXTM05 Master Thesis, 30 ECTS
Program in Surveying and Land Management

Lisette Danebjer

Supervisor: Lars Harrie
Department of Physical Geography and Ecosystem Science

Practical Supervisor: Björn Harrtell
Sweco Position Malmö

Juni 09, 2016

Opponent: Gunnar Rolander
Examiner: Ali Mansourian

Department of Physical Geography and Ecosystem Science
Lund University
Sölvegatan 12
223 62 Lund

Telephone: 046-222 30 30
Fax: 046-222 03 21
Web: http://www.nateko.lu.se

Thesis in Geographical Information Technology nr 20
Printed by E-tryck, E-huset, 2016

Preface

This thesis is the final step towards completing a Master of Science in Engineering, specializing in Surveying and Land Management. The work is performed at the Department of Physical Geography and Ecosystem Science, Lund University in cooperation with Sweco Position in Malmö.

The greatest **thank you** goes to both my supervisor. **Björn Harrtell** at Sweco Position for contributing with the interesting idea proposal that made this project possible. He has been a valuable resource throughout the process, in particular during the implementation phase. With weekly meetings and ever so many additional questions, he has made the practical work proceed nicely. **Lars Harrie** at Lund University for providing the tools to make this into a readable and complete report. He has given regular feedback with comments from a point of view different from my own.

I would also like to thank the employees at Sweco Position in Malmö. First of all for giving me a place to work. But more importantly, for making me feel welcome from day one. It has been a lot of fun to get to know everybody and I will bring with me all the interesting conversations and laughs.

2016-05-19
Lisette Danebjer

Abstract

Spatial relationships are a comprehensive and essential part of geographical information management and GIS. Topological relationships are a subgroup of the spatial relationships. They are defined by the fact that they do not break under topological transformations, meaning they remain when stretching, twisting or crumbling the underlying space. Topological relationships are for instance relationships such as *contains*, *crosses* and *overlaps*.

There are different ways of representing geographical data, many of which do not take the topological relationships into account. For full support the topological relations need to be stored explicitly, which is done according to a topological data structure. Well-established database have implemented this type of structure in a successful way. When looking to manage the data without any database dependency, the solutions are limited.

This study look to find a method for developing a GIS client application for creating and modifying distributed topologically structured geographical datasets. The client application should support being disconnected from any database when worked in. Using an existing topology as a start value, for instance data from a topologically structured database, should be possible. Alternatively, the user should be able to create data from an empty topology.

During this study, a client application as the one described above is developed. The product is near complete when compared to the requirements set up. Some weaknesses occur, however they are due to other factors than the pure logic of the topological calculations. A method for developing a topologically structured client application is therefore considered found. The implemented functionality covers basic operations, leaving room for further developments.

With the method brought by this study, similar client applications can be developed for other devises including mobile phones and tablets. Furthermore, a general topological library could possibly be developed using the same techniques, which would contribute to better conditions for developing topologically structured services.

Sammanfattning

Spatiala relationerna utgör en omfattande och betydande del inom geografisk informationsbehandling och GIS. Topologiska relationer är en undergrupp av spatiala relationer, definierade av det faktum att de inte påverkas under topologiska transformationer. Enklare uttryckt, det är de relationer som består när det underliggande rummet trycks ihop, dras ut eller vrids. Exempel är relationer så som *innehåller*, *korsar* och *överlappar*.

Geografiska data kan representeras på olika sätt. Många av de vedertagna formaten och standarderna är inte tillräckliga när hänsyn ska tas till de topologiska relationerna. För att kunna hantera topologiska relationer måste de lagras explicit. Detta görs i enlighet med en topologisk datastruktur. Konceptet är tillämpat för etablerade databaser och fungerar väl. Önskas emellertid en tjänst där topologiska data kan hanteras utan databaskoppling, är alternativen begränsade.

Denna studie syftar till att ta fram en metoder för att utveckla en GIS-klient applikation där distribuerad topologiskt strukturerad geografiska data kan importeras, skapas och modifieras. Klient applikationen skall kunna köras utan koppling till en databas. Användaren ska kunna skapa en topologi från grunden, alternativt ladda in en topologiskt strukturerad datamängd och använda den som startvärde.

Under detta projekt har en klient applikation, likt den ovan beskriven, utvecklats. Vid jämförelse mot de krav som satts upp är resultatet nära fullständigt. De brister som råder är inte kopplade till den logik som utgör de topologiska beräkningarna. En metod för att utveckla en topologisk strukturerad klient applikation anses därför framtagen. Den implementerade funktionaliteten är begränsar och utgör grundläggande operationer, vilket lämnar rum för vidare utveckling.

Den framtagna metoden kan utnyttjas vid liknande projekt, till exempel för topologiska tjänster till mobiltelefoner och surfplattor. Vidare kan samma tekniker underlätta utvecklandet av ett generellt topologiskt kod-bibliotek. En sådan tillgång hade bidragit till bättre förutsättningar och främjat utvecklingen av topologisk strukturerade tjänster.

Abbreviation list

API	***A**pplication **P**rogramming **I**nterface*: An interface constituting the link between an application and a specific software or library.
CSS	***C**ascading **S**tyle **S**heets*: A language used to style HTML or XML documents.
DOM	***D**ocument **O**bject **M**odel*: A platform based interface giving languages the possibility to dynamically read and update a document.
GIS	***G**eographical **I**nformation **S**ystem*: A system for gathering, storing, analyzing and presenting geographical data.
GML	***G**eography **M**arkup **L**anguage*: An XML bases standard specialized in expressing geographical features.
HTML	***H**yper**T**ext **M**arkup **L**anguage*: A standard language for creating web pages.
ISO	***I**nternational **O**rganization for **S**tandardization*: An organization developing and publishing international standards.
JSON	***J**ava**S**cript **O**bject **N**otation*: A data-interchange format for storing and exchanging data.
OGC	***O**pen **G**eospatial **C**onsortium*: A non-profit organization working with standardizing geographical data.
OSM	***O**pen **S**treet **M**ap*: An open source world map.
SFS	***S**imple **F**eature **S**tandard*: A standard specifying a storage and access model for two dimensional geographical data.
SQL	***S**tructured **Q**uery **L**anguage*: A query language for communicating with relational databases management systems.
W3C	***W**orld **W**ide **W**eb **C**onsortium*: International community for developing web standards.
XML	*E**x**tensible **M**arkup **L**anguage*: A markup language defining rules readable for both humans and machines.

Table of content

1 Introduction

1.1 Background

Spatial relationships are an essential and obvious part of GIS today. They are defined as the relationships that exist between spatial objects in space. These relations are constantly used when analyzing and working with geographical data. Topological relationships are a subgroup of the spatial relationships. Under this category fall relationships that do not change under a topological transformation. Examples are relationships such as *contains*, *touches* and *overlaps*.

The common way of representing geographical data today is through the SFS (Simple Features Standard), where the geometries are mainly stored and accessed as two-dimensional data such as points, lines and polygons. When working with some of the spatial relationships this structure is sufficient. In other cases though, the data would become more manageable if the topological relationships were stored explicitly. For instance, real estate registries and forest plans benefits a great deal from knowing the topological associations when changing borders and adding new properties.

Storing data in a topological oriented way is done through topological data structures. The structures can differ, but normally shares the same basic elements (nodes, edges and faces) to represent the data. At present (2016), the concept of handling topological data is established and well working for spatial databases such as Oracle Spatial and PostGIS. Formats such as TopoJSON can store and visualize topological data. However, there are no standard solutions to handle topological data in a local client that is not connected to a database.

1.2 Problem Statement

Using a topological storage structure is many times favorable. The issue is that there are few GIS clients that can manage topological data without being connected to a database. This is a severe limitation when being absent of database access, also when looking to have a non-centralized storage model. Furthermore, the concept of handling subsets of data is not supported but would be of great value, for example when two users are working in the same database but with different part of the data. Also when loading a subset of data from the database to work on elsewhere, and then loading it back in a correct way. The concept is illustrated more thoroughly in figure 1.1.

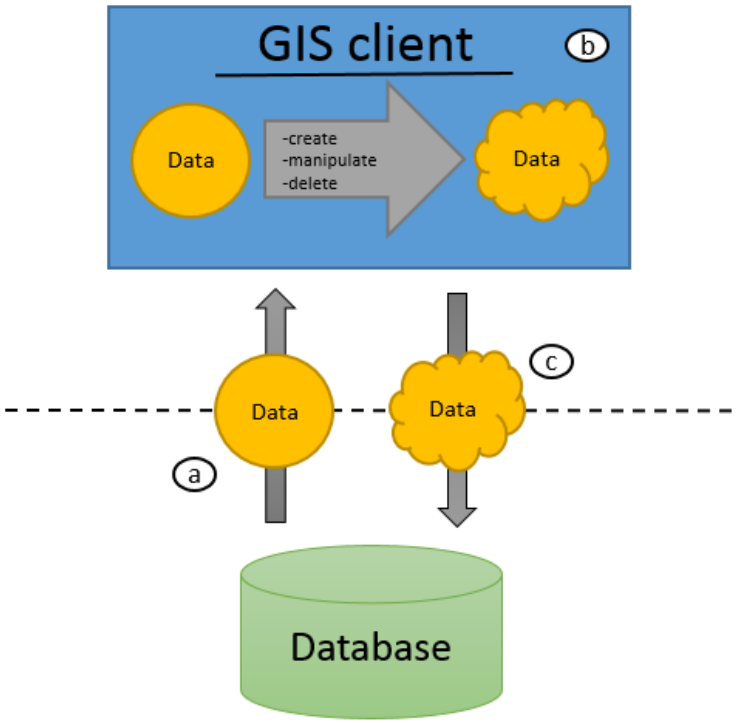

Figure 1.1. The process of (a) loading a data subset from a database, (b) creating, deleting and rendering the data in a GIS client without any connection to the database and (c) loading the manipulated data subset back into the database. The dotted line represents the disconnection between the GIS client and the database.

In figure 1.1 a data subset (yellow circle) is loaded from a database into a GIS client. In the GIS client the data are manipulated when not connected to the database. The data subset (yellow cloud) is then loaded back into the database in a correct way. This process is common within data management in general, not just for geographical data. Applying the concept on topologically represented data is however a complicated procedure. The process can be divided into three parts, corresponding to the ones in figure 1.1; (a) loading data from a topologically structured database into a GIS client application, (b) developing a GIS client application that in a correct way can handle topological data and (c) loading a topological data subset into a topologically structured database in a correct way.

Loading a data subset from a topologically structured database is straight forward. The problem is to find a serialization format that the database can write to, as well as the client application can read and interpret.

Developing a GIS client application for topological data is a complicated process. The topological rules need to apply without the user having to think about them. The user is not expected to know the coordinates or id:s of the objects, as when working in a database. Simply drawing the topology on a background map should be possible, which requires extensive controls when trying to add the data. The calculations need to be performed in a correct, but also efficient way; more or less so due to the size of the dataset and the storage method used. The logical parts need to be connected to the graphical parts, providing the user with an intuitive interface. Still, the logic needs to be independent and not rely on the graphics.

The associations between the objects are what create the topology, meaning that most of the data depend on each other. Due to this, it is difficult to load a data subset back into a database and merge it with the rest of the data in a correct way. When an element is changed in a topology, all the connecting elements must be updated if affected, this could possibly be an element still in the database.

1.3 Aim

The main aim of the project is to developing a client application where topological data can be loaded, created, deleted and rendered without any direct connection to a database. This can be summarized as part a and b in figure 1.1.

The aim of the theoretical part of the project is to studying common formats and standards that support a topological data structure. By recognizing the way the structures are built and what methods are required, the practical work can be performed in a correct and persistent way.

Furthermore, the difficulties that come with storing data as a topology are to be address and evaluated. The idea of loading a subset of topological data from a database, manipulate it and load it back without complications is discussed.

1.4 Method

To reach the goals set up for this report the work is performed in phases. (1) A theoretical study is completed and use cases are described. (2) User requirements are set up, formats and standards are evaluated and an overall system structure is designed. (3) A client application for topological data is implemented and (4) tested. (5) The result is evaluated.

The theoretical part of the report involves studying topological relationships, definitions of algorithms, topological data structures and databases. Based on the theory, the standards and formats are evaluated. Based on use cases, requirements are set up to specify the functionality of the client application. The gathered information is used to select the formats and standards looking most promising for this project. The system structure is designed on an overall level. This work lies as a foundation of the practical part. (1) + (2)

The practical work includes developing a GIS client application that can load, create, delete and render topological data without being connected to a database. The overall idea is to develop a client application that is comparable to the one in figure 1.1. The practical work is focused to process a and b. Putting the data subset back into the database (process 3) is not dealt with on a practical note in this report. The implementation looks to follow the requirements set up in chapter 6.1. However, throughout the implementation it becomes clear if and where demarcation needs to be drawn, possibly resulting in all requirements not being fulfilled. To accomplish the practical work, code samples that can help the development are viewed. The knowledge and experience held by the supervisor, as well as other employees at Sweco Position, is taken advantage of when needed. (3)

The client application, when considered finished within the limitations that are set up, is tested and improved. The result is evaluated and compared to the requirements. (4) + (5)

1.5 Delimitations

To fully complete the process shown in figure 1.1, it must be possible to load the data back into the database (process c). This part of the process is too complex to include in this study. The theoretical part alone would be very comprehensive, and could possibly be another thesis itself.

1.6 Disposition

The report starts with an introducing chapter including aim and problem statement. In chapter 2, use cases are described to provide a background and define user requirements. Chapter 3, 4 and 5 are theoretical chapters, describing spatial relationships, topological relationships, topological data structures, formats and standards. A general structure overview is presented before a specific solution is designed. The case study is given in chapter 7, describing the choices made, the implementation process and the result. This is followed by a discussion in chapter 8 and conclusions in chapter 9.

In the report, *italics* font is used as a tool to indicate when a word is first introduced and later described (within the chapter). Also, when something is being listed or should draw attention.

2 Use Cases

A common issue within GIS is data quality problems. Quality problems often lead to incorrect result when performing calculations and geometrical operations on the data. Some of the problems would disappear if the data were to be represented as a topology. At present (2016), there are a number of databases that offer a topological storage structure. When wishing to use a mobile client together with geographical data stored in files, the solutions are limited. Below, two use cases are described where the advantages of handling the topological data in a client application are presented.

2.1 Use Case 1 - Changing borders of cadaster data.

When working with cadasters, changing borders is a common procedure. It often includes site visits for measuring and gathering of data. The new data are then normally brought back “to the office” for the updates to be made.

Large volumes of real estate data are often stored in databases as polygons. The polygons geometry corresponds with the geometric extent of each real estate. Now this storage model works well as a whole, but contains redundant data. If this redundancy is not handled in a correct way, errors which are hard to detect can occur. More clearly, when two real estate lie next to each other they share a border. The line segment representing the shared border is part of both polygons (representing the real estate), and must correspond. This means that if changing a line segment of one polygon, the line segment of the other polygon must be changed as well (if the borders are still meant to meet each other). One or several new polygons have to be created, where all the line segments of the new polygons again must correspond to its neighboring line segment. If the new polygons are created with the wrong geometries and the borders do not agree, there will be holes in between the polygons, alternatively overlapping surfaces. This will lead to miscalculations when preforming geometrical operations. The scenarios are illustrated in figure 2.1, where the border of the orange real estate is moved to the dotted line. In the left bottom figure (a), the border of the yellow real estate is not moved, creating a hole (grey area). In the right bottom figure (b), the boarder of the yellow real estate is misplaced, creating an overlapping surface (grey area).

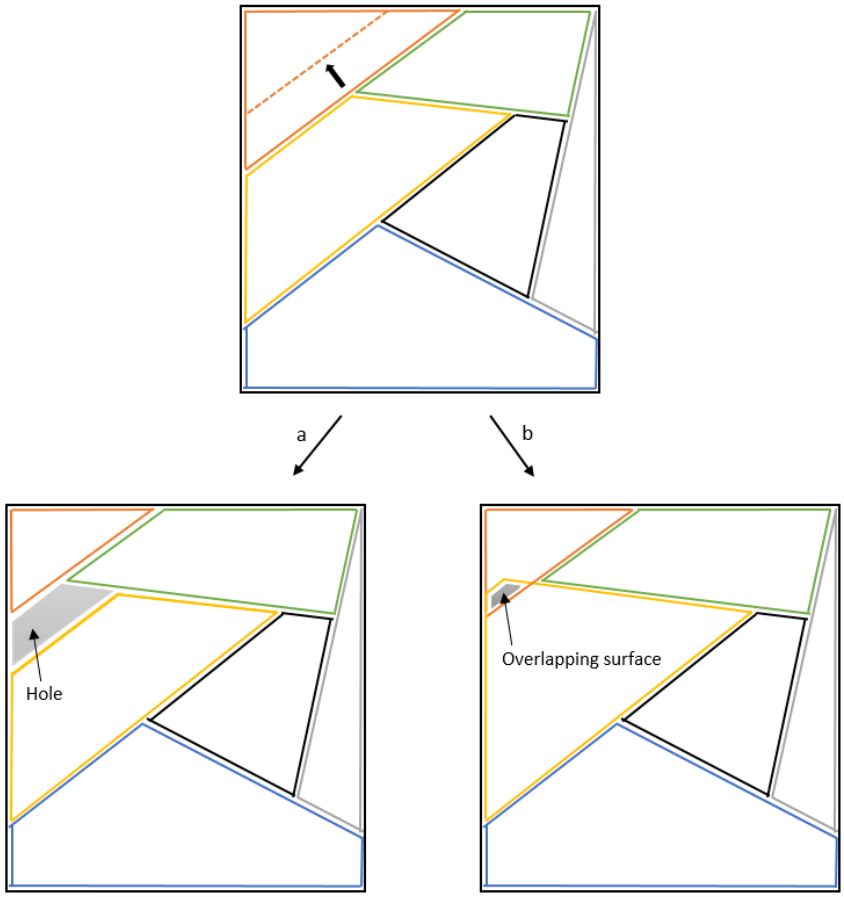

Figure 2.1. The line segment of the orange real estate (polygon) is moved to the dotted line. In (a) the bottom left scenario the line segment of the yellow real estate (polygon) is not corrected creating a hole. In (b) the bottom right scenarios the line segment of the yellow real estate (polygon) is moved incorrectly, creating an overlapping surface.

When using a topological data structure, there are no polygons in the geometry. The data consists of nodes (points) and edges (line strings) that together encloses surfaces building the so called faces (what looks like polygons). There is only one line to represent each border, reducing duplicative data. Figure 1.2 shows how the same real estate data can be stored using polygons (left figure) and as a topology (right figure).

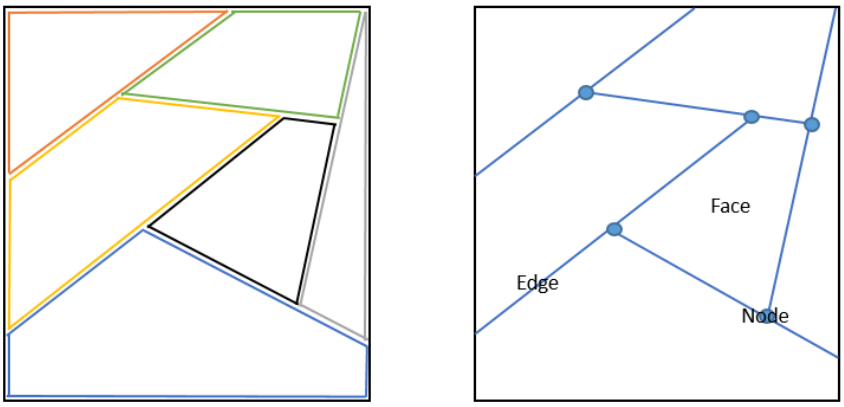

Figure 2.2. The same real estate data represented in SFS as polygons (left figure) and with nodes, edges and faces in a topology (right figure).

When moving a border in a topological data structure, no surfaces are recreated. The actual border (edge) is moved, and the surfaces will be according to the moved edges. In figure 2.3

the green edges and node are added and the crossed out edges and node are removed. This will result in an absolute correct presentation of all the properties.

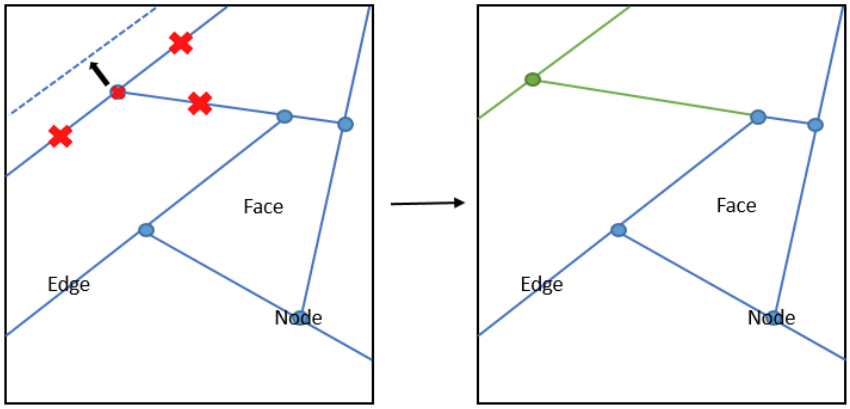

Figure 2.3. Moving an edge in a topology to the dotted line. Showing what nodes and edges to remove with the red crosses (left figure) and what edges and nodes to add in green (right figure).

If using a client application, it would be possible to perform work on the topological data at site. The client application allows the user to bring a subset of the cadaster data, manipulate it at site and thus make sure it agrees with reality as the updates are being made. Adding to this, being able to merge the subset with the remaining data in a correct way would complete the whole process.

2.2 Use Case 2 – Calculating snow cover for slopes and thoroughfares

Ski resorts stores data of their ski slopes and thoroughfares normally represented as polygons. The data are partly used to calculate the amount of snow to distribute over the region. The calculations are based on the area of the polygons. However, when the thoroughfares crosses the slopes, overlapping surfaces appear (see figure 2.4). Simply summarizing the areas will add up to an incorrect result. There are ways to correct this, such as subtracting the intersection or splitting up the polygons when preforming the calculations. The accuracy when doing so can however vary a lot, possibly leading to an insufficient result when dealing with large datasets.

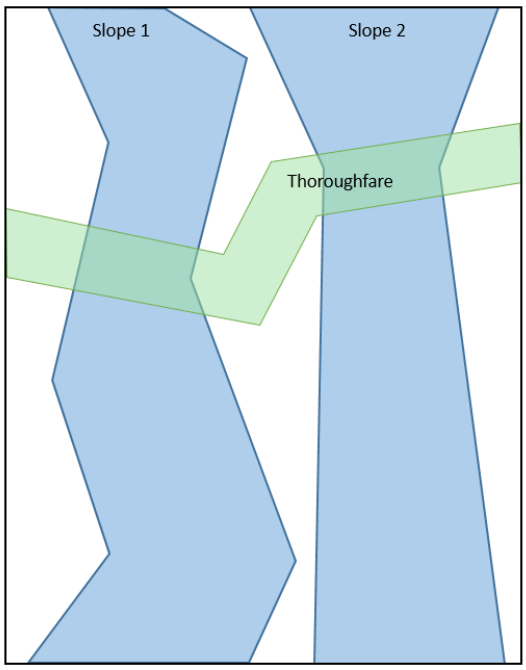

Figure 2.4. Ski slopes and thoroughfares as polygons, creating overlapping surfaces.

By using a topological data structure, the risk of this happening is erased. Figure 2.5 show the slopes and thoroughfares represented as a topology. The faces are as stated the surfaces enclosed by the edges. Summarizing the areas of all the faces constituting a skiable surface will thereby give the correct result.

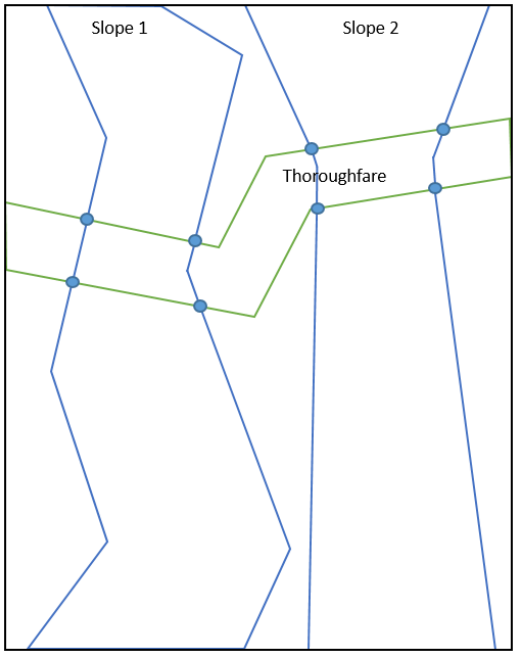

Figure 2.5. Ski slopes and thoroughfares as a topology, creating one face for each enclosed surface.

Again, performing the work in a client application at locally could benefit for the user. It gives opportunity to update the data at site, making sure it all agrees. The client application also offers an intuitive interface where the user does not have to communicate with a database through a query language.

3 Spatial Relationships

Spatial relationships are used to describe how a spatial object is located in space in relation to another object. Blake (2007) states: “A description of the one or more ways in which the location or shape of a feature is related to the location or shape of another feature”. Under this category fall relations such as *distance*, *intersection*, *size* and *time*. Knowing the spatial relationships between features is essential when working in GIS (Esri 2016). Questions like “How close to a school can a windmill be placed?” and “Where does the river cross the country border?” are answered based on these relationships.

In the process of establishing the spatial relationships a series of methods have been developed. The extent of different relationships is large, why a single method compatible for all of them does not exist. However, by grouping the relationships they can within that group be investigated as a whole (Egenhofer et al. 1993).

One way of grouping the relationships is by looking at the correspondence to the fundamental mathematical concepts: topology, order and algebra. Through this, three categories describing relationships can be formed: *topological relationships*, *spatial ordered relationships* and *metric relationships* (Kainz 1989; Egenhofer and Herring 1990). This is a common way of grouping the relationships, yet it does not cover all of them. *Fuzzy relationships* are here not included. Neither the relationships expressed about the motion of one or several objects (Egenhofer and Herring 1990).

Topological relationships are the main subject of this report and are described further in chapter 4. Spatial order relationships follow under the definition of an order or strict order described by prepositions such as *behind, in-front, above* and *below*. The relations normally have a *converse relationship*, e.g. the relation behind have the converse relationship in front of (Freeman 1975; Egenhofer and Herring 1990). Metric relationships are defined by the fact that they use measurements in terms of distance and directions (Peuquet and Ci-Xiang 1987; Egenhofer and Franzosa 1991). Fuzzy relationships are the ones such as *close* and *next to*, whereas *through* and *into* expresses the motion of one or several objects.

4 Topological Relationships

4.1 Introduction

Topological relationships are as mentioned in the previous chapter a subgroup of spatial relationships. The relationships are defined by the fact that they are invariants under *topological transformations*.

A topological transformation is a one-to-one transformation between the points, where the transformation is bicontinuous (continuous in both directions). This includes transformations such as translation, scaling and rotation (Egenhofer and Herring 1990). An example of a continuous transformation is shown in figure 4.1, where a donut is turned into a mug without any tearing.

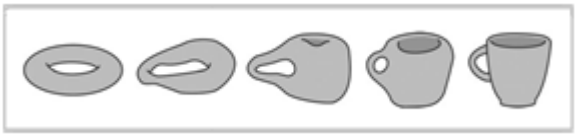

Figure 4.1. A continuous transformation, turning a donut into a mug without any tearing (MathCamp 2016).

For better understanding of topological relationships imagine a map made of rubber; the relationships that do not differ after stretching, twisting or crumbling the map are topological. Examples can be viewed in figure 4.2, where the topological relationship *contains* and *overlaps* is visualized before and after stretching the surfaces. Topological equivalents do not preserve distance, instead investigations are based on continuity described in terms of coincidence and neighborhood (Egenhofer and Herring 1990).

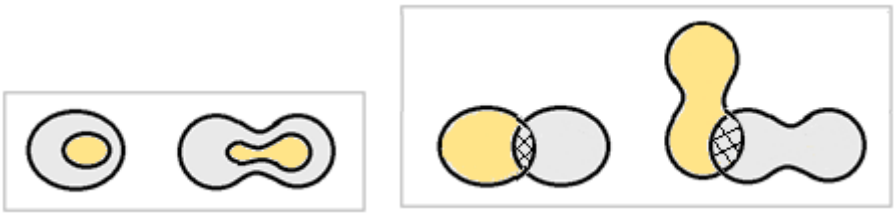

Figure 4.2 Showing how the topological relationship *contains* (the outer surface contains the inner surface) (left image) and *overlaps* (right image) is preserved after stretching the surfaces (a continuous transformation).

4.2 Definition of Topological Relationships

The field topology has many branches including algebraic topology, combinatorial topology and *point-set topology* (Lake et al. 2004). To define the topological relationships a number of models have been developed. Common models are the *4 intersection model*, the *9 intersection model* and the *Dimensionally Extended 9 Intersect Model*. They are described in the following sections.

When defining topologies the point-set topology is often encountered. This means the concept of basing the topological relationships on the point-set notation of the *interior* and *boundary* (Egenhofer and Franzosa 1991). The interior is defined as the union of all open sets that are contained in a point-set, *Y*. The boundary is the intersection of the *closure* of Y and the closure of the complement of *Y*. The closure represents the intersection of all closed sets

that are contained in *Y* (Egenhofer and Herring 1990). Further, the interior consists of the points that are left when the boundary points are removed, and vice versa (Herring 2011). Examples of models using the point-set notation are the 4- and 9-intersection model.

4.3 The 4-Intersection Model (4IM)

The 4-intersection model (4IM) was presented by Egenhofer and Franzosa 1991. It is used to determine topological relationships between geometric objects, and has become the foundation of several other models. The model compares two 2-dimentianal objects (A and B) in $\mathbb{R}^2$, which are defined as the product of their interior (A^o and B^o) and their boundary (∂A and ∂B) (Egenhofer and Franzosa 1991; Egenhofer et al. 1993).

The topological relationship between A and B are established by looking at the intersection of A:s interior and A:s boundary with B:s interior and B:s boundary (Egenhofer and Franzosa 1991; Egenhofer et al. 1993). To make it more readable this is often presented in a 2x2 matrix, see figure 4.3. Now, let the interior and boundaries be considered as empty and non-empty sets. By combining all possible intersections of the interior and boundaries being empty or non-empty sets, 2^4=16 possible outcomes will appear. These are the topological relations between the objects.

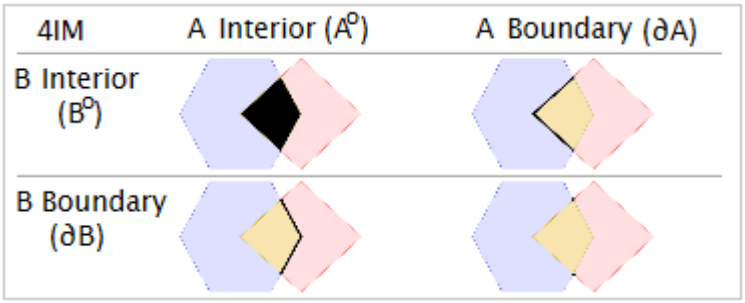

Figure 4.3. The 4-intersection Matrix. The intersection of the interior and boundary of a geometry (A) with the interior and boundary of another geometry (B) (Based on GeoTools 2016).

Eight of the relationships can be acknowledged as homogeneously 2-dimentianal objects with connected boundaries when in $\mathbb{R}^2$. They are illustrated in figure 4.4. Eight of them can be acknowledged between two lines in $\mathbb{R}^1$. (Egenhofer and Herring 1990; Egenhofer et al. 1993).

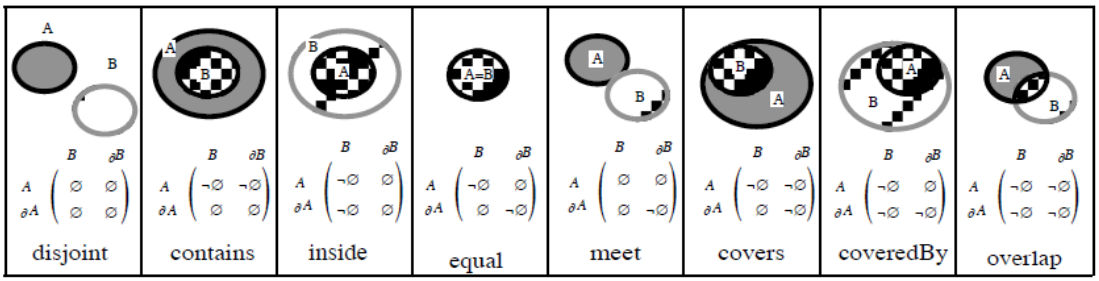

Figure 4.4. Eight topological relationships between two regions in $\mathbb{R}^2$ (Egenhofer et al. 1993).

Empty and non-empty sets are the most common *topological invariants* to use. Other properties preserved under topological transformations can be used, e.g. *content of intersection*, *dimension* and *number of separate boundary intersections* (Franzosa and

Egenhofer 1992; Egenhofer et al. 1993). The advantages of using empty and non-empty sets are that they describe closed sets of relationships with complete coverage, which leads to the identification of more relationships (Egenhofer and Herring 1990).

4.4 The 9-Intersection Model (9IM)

The 9-intersection model is an extension of the 4-intersection model. The principal is the same but there is an additional part defining each object, the *exterior* (A^e). The exterior consists of the points not in the interior or boundary (Herring 2011).Each object is here represented by its interior, boundary and exterior, resulting in the intersection matrix expanding to a size of 3x3 (see figure 4.5). All possible combinations when using empty and non-empty sets will result in 9^2=512 different outcomes. However, only a handful of these topological relationships are realized between two objects in $\mathbb{R}^2$. Finding them requires looking at the properties of the objects themselves and their relationships in space when combining the sets (Egenhofer and Herring 1991). Their existents can be proven by finding the geometric interpretations for the corresponding 9-intersections (Egenhofer et al. 1993).

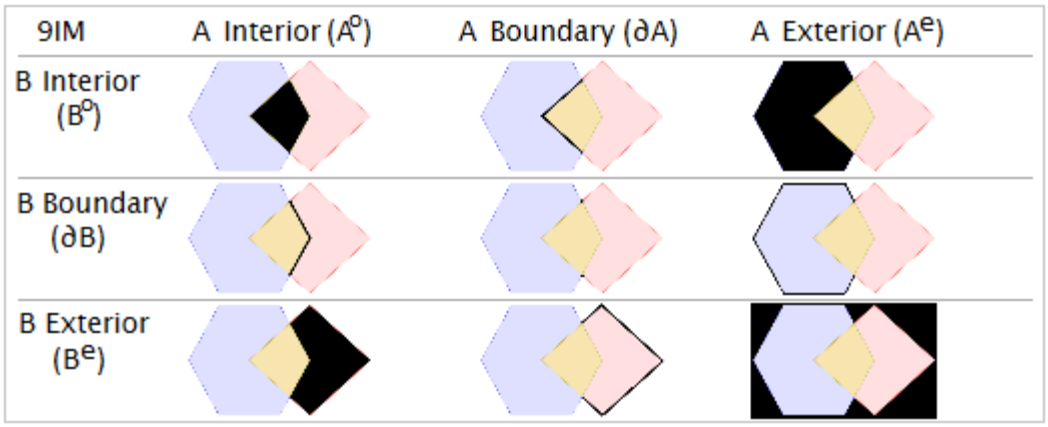

Figure 4.5. The 9-intersection Matrix. The intersection of the interior, boundary and exterior of a geometry (A) with the interior, boundary and exterior of another geometry (B) (GeoTools 2016).

4.5 The Dimensionally Extended 9-Intersect Model (DE-9IM)

The Dimensionally Extended 9-Intersect Model is an extension of the 9-intersection model. When looking at the result of the intersection between two geometric objects interior, boundaries and exterior, the dimension of the result is also considered. This is valuable since the geometries brought by the intersections can adopt different dimensions depending on the location of the objects. For instance, an intersection between two polygons boundaries can consist of either a point or a line string (see figure 4.6) (Esri 2016).

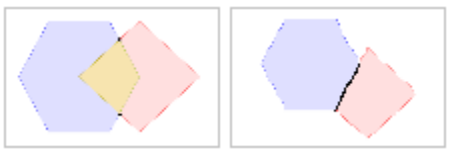

Figure 4.6. The intersection of two polygons boundaries can consist of both 0-dimension geometries (points) (left figure) and 1-dimension geometries (lines) (right figure) (Partly based on GeoTools 2016).

The DE-9IM uses a dim-function to establish the dimensions of the geometric objects. The value -1 is returned when the intersection is the empty set (dim(Ø)). When non-empty, 0, 1 and 2 is returned as the corresponding value to the dimension of the geometry. Points are of dimension 0, line strings of dimension 1 and polygons of dimension 2. The boundary of a geometric object is the set of geometric objects of the next lower dimension. The boundary of a point is thus the empty set. The example above in figure 4.5 will return the *maximum dimension* of 1, since the line string has the dimension 1. Table 4.1 shows the general form of the dimensionally extended 9-intersection matrix between two geometric objects A and B (Herring 2011).

Table 4.1. The general form of the Extended 9-intersection Matrix. A and B are geometric objects (Based on Herring 2011).

	A Interior (A^O)	A Boundary (∂A)	A Exterior (A^e)
B Interior (B^o)	dim((A^O) ∩ (B^o))	dim((∂A) ∩ (B^o))	dim((A^e) ∩ (B^o))
B Boundary (∂B)	dim((A^O) ∩ (∂B))	dim((∂A) ∩ (∂B))	dim((A^e) ∩ (∂B))
B Exterior (B^e)	dim((A^O) ∩ (B^e))	dim((∂A) ∩ (B^e))	dim((A^e) ∩ (B^e))

A *predicate* is a Boolean function that returns the value true or false when determining if a specific relationship exists between two geometries. The functions are commonly used when checking for a specific type of topological relationships and are based on the logic of the intersections. Examples of predicates are *intersects*, *equal*, *disjoint* and *overlap*. To understand and verify the result of the predicates, it is often compared in a *pattern matrix*. A pattern matrix is the compilation of the accepted values for each of the intersection. Each predicates can have more than one pattern matrix that represents its accepted values from the DE-9IM. This simply depends on the combinations of the geometry type (Esri, 2016). The pattern matrixes use the following values:

Table 4.2. The possible values for the Pattern Matrixes (Esri 2016).

T	An intersection must exist; dim = 0, 1, or 2.
F	An intersection must not exist; dim = -1.
*	It does not matter if an intersection exists or not; dim = -1, 0, 1, or 2.
0	An intersection must exist and its maximum dimension must be 0; dim = 0.
1	An intersection must exist and its maximum dimension must be 1; dim = 1.
2	An intersection must exist and its maximum dimension must be 2; dim = 2.

The relationship *equal* does accordingly have the following pattern matrix:

Table 4.3. The Pattern Matrix for the relationship *equal* (Esri 2016).

	A Interior (A^O)	A Boundary (∂A)	A Exterior (A^e)
B Interior (B^o)	T	*	F
B Boundary (∂B)	*	*	F
B Exterior (B^e)	*	*	*

This can also be expressed as “T*F**F***” (going left to right, row by row, in table 4.3). If a spatial relationship between two geometric objects corresponds to one of the accepted values of a pattern matrix, the predicate will return true (Herring 2011).

5 Topological Data Formats

Implementing topologies in GIS in done according to *topological data structures* (Theobald 2001). Several commonly used data formats supports such a data structure and can therefore store topological relationships explicitly. This brings advantages such as reduced data storage due to shared boundaries between polygons. It offers complete coverage without having to deal with overlapping polygons. It also simplifies handling digitizing and editing errors and artifacts. Furthermore, questions that cannot be answered with relational databases and other traditional information management systems can with a topological data structure be analyzed and replied (Theobald 2001). Algebraic topology and combinatorial topology are most applicable to describe geographical objects (Lake et al. 2004).

This chapter starts with a description of one topological data structure. After follows descriptions about some of the most common data formats used for topological data today. It should be noted that common data formats such as Shapefiles, KML (see Harold and Means 2004) and GeoPackage do not store topological data explicitly and they are therefore not described here.

5.1 Topological Data Structure

The topological data structure described below is the ISO/IEC 13249-3:2016 standard, here implemented by Oracle Spatial. It is based on three basic elements: *nodes, edges* and *faces*. The "empty" topology consists of one covering face (*universal face*) with the id 0. The topology is describes through the basic elements (nodes, edges, faces) and the way they reference each other. All elements have a unique id. In figure 5.1, an example of a simplified topology in Oracle Spatial is illustrated. The figure is used as a reference when describing the topological data structure. Further notes on the figure are found in appendix A.

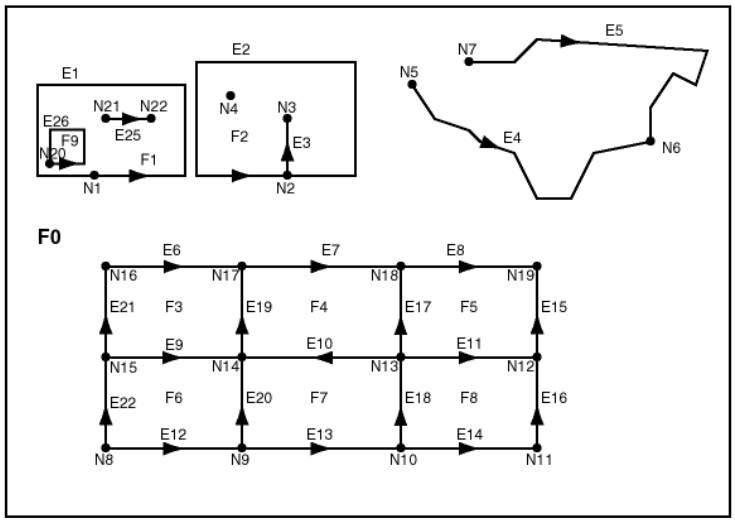

Figure 5.1. An example of a simplified topology in Oracle Spatial database (Oracle 2016).

A *node* is similar to a point, with coordinates directly connected to it. A node can be either isolated (island node) or non-isolated; no edges are bound to it respectively that at least one

edge is bound to it and that two or more edges can meet in the node. In figure 5.1, N4 is an isolated node, N3 has one edge bound to it and N2 has two edges meeting in it. Nodes have four attributes, they are explained in table 5.1.

Table 5.1. The attribute table for Nodes in Oracle Spatial database (Oracle 2016).

Column Name	Data Type	Description
NODE_ID	NUMBER	Unique ID number for this node
EDGE_ID	NUMBER	ID number (signed) of the edge (if any) associated with this node
FACE_ID	NUMBER	ID number of the face (if any) associated with this node
GEOMETRY	SDO_GEOMETRY	Geometry object (point) representing this node

An *edge* is comparable to a line string and is always bound to a node at its start and end point. The coordinates are stored in associative objects describing the spatial representation of the edge. It can consist of several line segments making up a line string, see edge E4 in figure 5.1. The extent of line segments does not fall under any constraint, nor does a line segment have a start and end node. Since the edges themselves must have a start and end node, circular edges (without any nodes) are not supported. In order to keep true to the topological relationships when handling the data, every edge holds a direction. The order of the coordinates describing the edges is used to determine the direction. The orientation of the edge is called a *directed edge*. Each directed edge is the mirror image of its other directed edge (holds the opposite direction). The directed edges hold a reference to the following directed edge, meaning the next one in the contiguous perimeter of the face on its left side. Every edge references to the face it lies within. If an edge lies between two faces, it has references to both of them (like edge E2 in figure 5.4). Edges have ten attributes, see table 5.2. Some of the attributes are self-explanatory, the other are described further.

Table 5.2. The attribute table for Edges in Oracle Spatial database (Oracle 2016).

Column Name	Data Type	Description
EDGE_ID	NUMBER	Unique ID number for this edge
START_NODE_ID	NUMBER	ID number of the start node for this edge
END_NODE_ID	NUMBER	ID number of the end node for this edge
NEXT_LEFT_EDGE_ID	NUMBER	ID number (signed) of the next left edge for this edge
PREV_LEFT_EDGE_ID	NUMBER	ID number (signed) of the previous left edge for this edge
NEXT_RIGHT_EDGE_ID	NUMBER	ID number (signed) of the next right edge for this edge
PREV_RIGHT_EDGE_ID	NUMBER	ID number (signed) of the previous right edge for this edge
LEFT_FACE_ID	NUMBER	ID number of the left face for this edge
RIGHT_FACE_ID	NUMBER	ID number of the right face for this edge
GEOMETRY	SDO_GEOMETRY	Geometry object (line string) representing this edge, listing the coordinates in the natural order for the positive directed edge

The value of *NEXT_LEFT_EDGE_ID* is the id of the closest connecting edge when looking clockwise from the end point of the edge itself (the directed edge is thus in the direction start point to end point). If (1) the next left edge is directed away from the edge, the id is to be positive (start point meeting end point). If (2) the next left edge is directed towards the edge the id is to be negative (end point meeting end point). Furthermore, if (3) there are no edges meeting in the point, NEXT_LEFT_EDGE_ID will be the negated id of the edge itself. The three outcomes are illustrated (in order from left to right) in figure 5.2, where the edge itself has id 1.

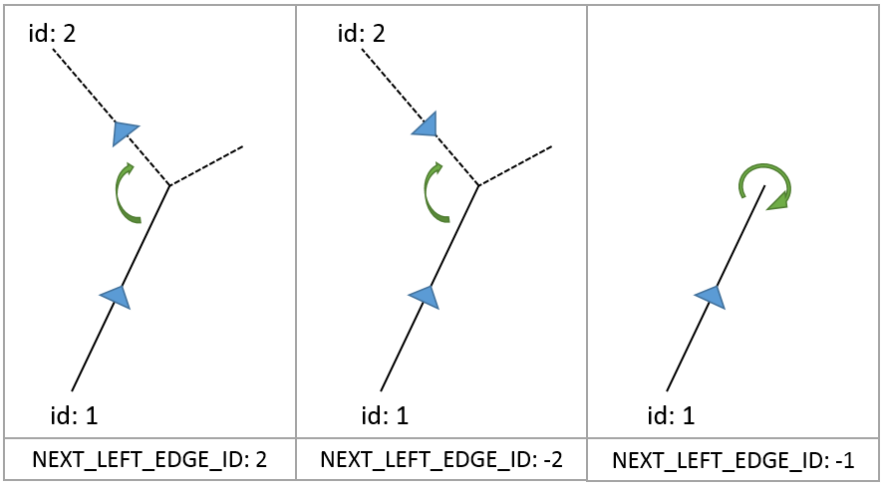

Figure 5.2. The way the NEXT_LEFT_EDGE_ID attribute is set for the edge with id 1. The blue arrows show the direction of the edges (start point to end point).

The value of *NEXT_RIGHT_EDGE_ID* is set in the corresponding way, only looking at the mirror image of the directed edge used for NEXT_LEFT_EDGE_ID instead (the directed edge is here in the direction end point to start point). If (1) the next right edge is directed away from the edge, the id is to be positive (start point meeting start point). If (2) the next right edge is directed towards the edge the id is to be negative (end point meeting start point). If (3) there are no edges meeting in the point, NEXT_RIGHT_EDGE_ID will be the (positive) id of the edge itself. The three outcomes are illustrated (in order from left to right) in figure 5.3, where the edge itself has id 1.

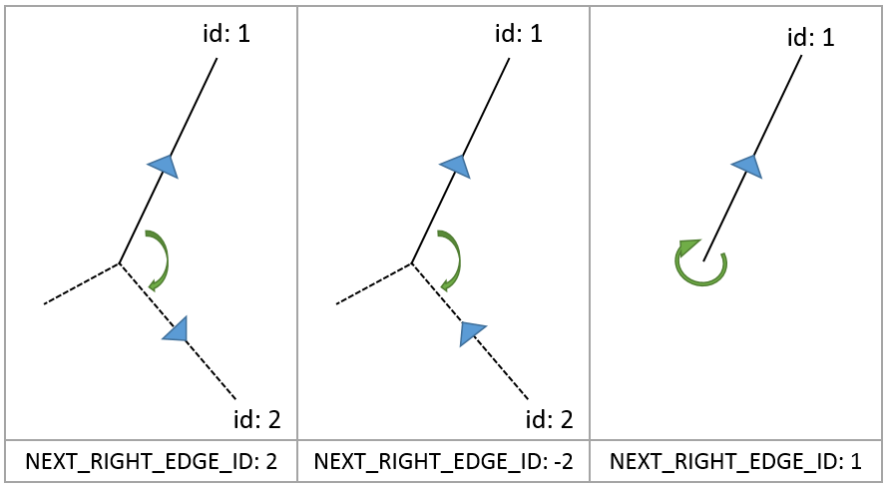

Figure 5.3. The way the NEXT_RIGHT_EDGE_ID attribute is set for the edge with id 1. The blue arrows show the direction of the edges (start point to end point).

The values of *PREV_LEFT_EDGE_ID* and *PREV_RIGHT_EDGE_ID* are the id of the edges when looking at the next edge in the counter-clockwise direction from the start respectively end point of the edge itself. Negated id:s are set the opposite way from NEXT_LEFT_EDGE_ID and NEXT_RIGHT_EDGE_ID; if the next edge is directed away from the edge itself the id is to be negative, if the next edge is directed towards the edge the id is to be positive. For further explanation on PREV_LEFT_EDGE_ID and PREV_RIGHT_EDGE_ID, see figure 5.4 and table 5.3

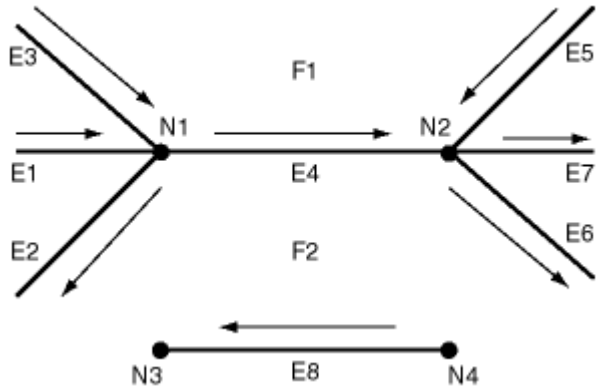

Figure 5.4. A collection of nodes and edges in Oracle Spatial database (Oracle 2016).

Table 5.3. The attribute data for edge E4 and E5 (see figure 5.4) in Oracle Spatial database (Oracle 2016).

EDGE_ID	START_NODE_ID	END_NODE_ID	NEXT_LEFT_EDGE_ID	PREV_LEFT_EDGE_ID	NEXT_RIGHT_EDGE_ID	PREV_RIGHT_EDGE_ID	LEFT_FACE_ID	RIGHT_FACE_ID
E4	N1	N2	-E5	E3	E2	-E6	F1	F2
E8	N4	N3	-E8	-E8	E8	E8	F2	F2

The *LEFT_FACE_ID* and *RIGHT_FACE_ID* attributes are the face id of the faces lying to the left respectively to the right of the edge when looking at the directed edge start point to end point, see figure 5.5. If there is no face to the left and/or to the right of the edge, the value is 0, representing the id of universal face. The GEOMETRY attribute holds the geometry of the edge (the coordinates).

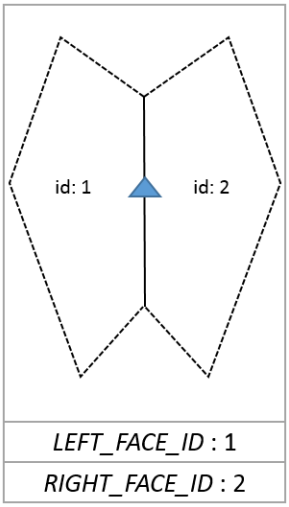

Figure 5.5. The way the *LEFT_FACE_ID* and *RIGHT_FACE_ID* attribute are set for the edge (solid line).

A *face* corresponds to a polygon and is the surface that occurs when one or several edges encloses an area, like face F3 in figure 5.1. A face holds a reference to one of the directed edges of its outer boundary. If the face contains any island nodes or island edges, a reference to one directed edge of each island is required. Faces have five attributes, explained in table 5.4.

Table 5.4. The attribute table for Faces in Oracle Spatial database (Oracle 2016).

Column Name	Data Type	Description
FACE_ID	NUMBER	Unique ID number for this face
BOUNDARY_EDGE_ID	NUMBER	ID number of the boundary edge for this face. The sign of this number (which is ignored for use as a key) indicates which orientation is being used for this boundary component (positive numbers indicate the left of the edge, and negative numbers indicate the right of the edge).
ISLAND_EDGE_ID_LIST	SDO_LIST_TYPE	Island edges (if any) in this face. (The SDO_LIST_TYPE type is described in Section 1.6.6.)
ISLAND_NODE_ID_LIST	SDO_LIST_TYPE	Island nodes (if any) in this face. (The SDO_LIST_TYPE type is described in Section 1.6.6.)
MBR_GEOMETRY	SDO_GEOMETRY	Minimum bounding rectangle (MBR) that encloses this face. (This is required, except for the universe face.) The MBR must be stored as an optimized rectangle (a rectangle in which only the lower-left and the upper-right corners are specified). The SDO_TOPO.INITIALIZE_METADATA procedure creates a spatial index on this column.

Topological geometries, also addressed features, are equivalent to real world objects (e.g. a street or a park). When using a topological data structure, they are represented by a collection of topological elements forming what looks like objects. Figure 5.6 shows the topological elements in figure 5.1 represented as features. Here the linear features are meant to represent roads (R1-R4), the point features traffic signs (S1-S4) and the area features land parcels (P1-P5).

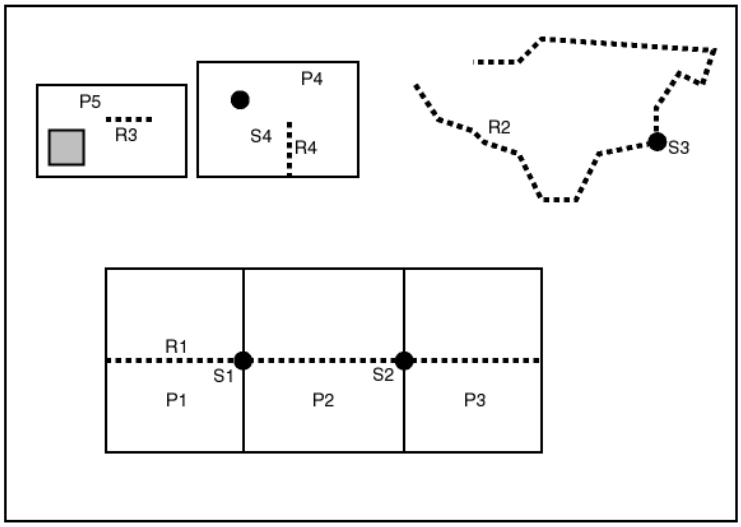

Figure 5.6. The topology in figure 5.4 represented as Features (Oracle 2016).

This looks similar to the way features are commonly represented when using a SFS. The land parcels for example, look like polygons lying next to each other. The difference is that the "polygons" share the lines creating them, there are not two lines on top of each other belonging to one polygon each. The same logic goes for a polygon lying inside another polygon; there are not two surfaces on top of each other.

5.2 GML

Geographical Markup Language (GML) is an OpenGIS standard specialized in expressing geographical features that can be shared on the Internet. The features describe real world objects. (Lake et al. 2004) GML is based on XML (Extensible Markup Language) grammar, consisting of a schema describing the document and an instance document containing the actual data. The schema sets the rules for how the geometries should be presented, allowing the user to create datasets containing point, line and polygon features. GML 3 is the most recent version (at 2016) and supports several entities, including topologies (Portele 2012).

The topological model for GML is according to the ISO 19107. An *abstract topology type* (gml:AbstractTopology) is used as a base element for all topological elements (*topological primitives* and *topological complexes*). A *topological abstract primitive type* (gml:AbstractTopoPrimitive) is in turn used as the base for all topological primitives. There are four topological primitives and a number of properties. The topological primitives are defined as the smallest units of the topology and vary in dimension from 0 to 3. The 0-dimensional primitive and property are the *node type* (gml:Node) and the *directed node property type* (gml:directedNode). The 1-dimensional primitive and property are the *edge type* (gml:Edge) and the *directed edge property type* (gml:directedEdge). The 2-dimensional primitive and property are the *face type* (gml:Face) and the *directed face property type* (gml:directedFace). The 3-dimensional primitive and property are the *topo solid type* (gml:TopoSolid) and the *directed topo solid property type* (gml:directedSolid) (Portele 2012).

The geographical features in GML may have a topological description, a geometric description or both. Each topology primitive type has a corresponding *geometry primitive*, see table 5.5.

Table 5.5 Topological Primitives and their corresponding Geometrical Primitives (Based on Table 1, Chapter 13 in Lake et al. 2004).

Topology Primitive	**Geometry Primitive**
Node	Point
Edge	Curve
Face	Surface
TopoSolid	Solid

There are two ways of connecting the topology model to the geometry model. One way is through *geometric realizations*, see table 5.6.

Table 5.6 Topological Primitives and their Geometric realizations (Based on Table 2, Chapter 13 in Lake et al. 2004).

Topology Primitive	**Geometry-Valued Property**	**Geometry Value**
Node	pointProperty	A Point
Edge	curveProperty	An object substitutable for a _Curve
Face	surfaceProperty	An object substitutable for _Surface

The example below shows how the geometric realizations can be encoded into a Node (Lake et al. 2004 p. 160-161):

```
<gml:Node gml:id=”A”>
  <gml:pointProperty>
    <gml:Point>
      <gml:pos>0 100</gml:pos>
    </gml:Point>
  </gml: pointProperty>
</app:Node>
```

(Taken from Chapter 13, Lake et al. 2004)

The connection can also be described using a *topological collection type* (gml:TopoPoint, gml:topoPointProperty, gml:TopoCurve, gml:topoLineCurve, gml:TopoSurface, gml:topoSurfaceProperty, gml:TopoVolyme , gml:topoVolymeProperty and gml:TopoComplex). The topological collection types are encapsulated within another feature (point, line, surface or solid) and therefore enables expressing the structural and geometric relationships of itself to other feature via shared edge definitions (Portele 2012). The example below show a BusRoute feature, which has a topoCurveProperty, whose value is a TopoCurve (Lake et al. 2004 p. 165):

```
<app: BusRoute gml:id:”rt44”>
  <gml:topoCurveProperty>
    <gml:TopoCurve>
       <gml:directedEdge orientation=”+” xlink:href=”#e5”/>
       <gml:directedEdge orientation=”+” xlink:href=”#e3”/>
       <gml:directedEdge orientation=”-” xlink:href=”#e6”/>
       <gml:directedEdge orientation=”+” xlink:href=”#e2”/>
       <gml:directedEdge orientation=”+” xlink:href=”#e7”/>
    </gml:TopoCurve>
  </gml:topoCurveProperty>
</app:BusRoute>
```

(Taken from Chapter 13, Lake et al. 2004)

The *topology complex type* (gml:TopoComplex) is a collection of topological primitives and other sub-collections (Portele 2012).

The relationships between the primitives of different dimensions are strong, they are bound to the directed primitives of a lower dimension and co-bounded to the directed primitives of

a higher dimension. The characterization of the relationships between objects are done using simple combinatorial or algebraic algorithms (Portele 2012).

5.4 JSON, GeoJSON and TopoJSON

JavaScript Object Notation (JSON) is a lightweight data-interchange format for storing and exchanging data. It is a "self-describing" text format and accordingly easy to read and write (JSON 2016). *GeoJSON* is a JSON format encoding a geographical data structure. The objects in GeoJSON can be represented as a geometry, a feature or a collection of features. GeoJSON supports several geometry features: *Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon* and *GeometryCollection* (GeoJSON 2016). *TopoJSON* is a format representing geometries in a topologically oriented way. It is an extension of the format GeoJSON. TopoJSON encodes geographical data structures into shared topologies. It supports all the geometries that GeoJSON does (TopoJSON 2013).

The structure in TopoJSON is based on *arcs*, which are arrays of arrays. The coordinates of a Point are stored in an array in a specific order: *easting, northing, altitude* alternatively *longitude, latitude, altitude*. Arcs are arrays of Point arrays. For instance, the geometry of a LineString will be represented by an array of two or more Point arrays (start and end point). The coordinates are stored in the arcs rather than for each object individually. The objects geometries are identified through the indexes of the arcs, which are stitched together to form the topology. An example is presented below (TopoJSON 2013):

```
{
  "type": "Topology",
  "transform": {
    "scale": [0.0005000500050005, 0.00010001000100010001],
    "translate": [100, 0]
  },
  "objects": {
    "example": {
      "type": "GeometryCollection",
      "geometries": [
        {
          "type": "Point",
          "properties": {
            "prop0": "value0"
          },
          "coordinates": [4000, 5000]
        },
        {
          "type": "LineString",
          "properties": {
            "prop0": "value0",
            "prop1": 0
          },
          "arcs": [0]
        },
        {
          "type": "Polygon",
          "properties": {
            "prop0": "value0",
            "prop1": {
              "this": "that"
            }
          },
          "arcs": [[1]]
        }
      ]
    }
  },
  "arcs": [
    [[4000, 0], [1999, 9999], [2000, -9999], [2000, 9999]],
    [[0, 0], [0, 9999], [2000, 0], [0, -9999], [-2000, 0]]
  ]
}
```

Point and MultiPoint are exceptions, they are simpler geometries described directly by the coordinates the way GeoJSON objects are (see example above). They are arrays of 1 dimension. LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection are complex geometries and represented through the arcs (the arrays in arrays). The dimension of the arrays depends on the geometry. In the example above the LineString has the geometry of the first element (index 0) in the arcs array, [[4000, 0], [1999, 9999], [2000, -9999], [2000, 9999]], and the Polygon the geometry of the second element (index 1), [[0, 0], [0, 9999], [2000, 0], [0, -9999], [-2000, 0]]. The following rules apply (TopoJSON 2013):

- For type **Point**, the coordinates member must be a single position.
- For type **MultiPoint**, the coordinates member must be an array of positions.
- For type **LineString**, the arcs member must be an array of arc indexes.
- For type **MultiLineString**, the arcs member must be an array of LineString arc indexes.
- For type **Polygon**, the arcs member must be an array of LinearRing arc indexes. For Polygons with multiple rings, the first must be the exterior ring and any others must be interior rings or holes.
- A **LinearRing** is closed LineString with 4 or more positions. The first and last positions are equivalent (they represent equivalent points).
- For type **MultiPolygon**, the arcs member must be an array of Polygon arc indexes.
- A TopoJSON object with type **GeometryCollection** is a geometry object which represents a collection of geometry objects.

The objects are not stored as features, instead they are converted to their respective geometry. Other properties such as identifier (id), bounding box (bbox) and transformers (transform) can be stored directly on the geometry objects. When the geometry is of value NULL, the objects can still exist and retain other properties. The simplicity in the representation enables efficient encoding (TopoJSON 2013).

5.5 PostGIS

PostGIS is an extension of the object-relational database PostgreSQL, focused on managing spatial data. PostGIS support storage of topological data and is an open source software released under the GNU license (General Public License). Just as Oracle Spatial, it is based on the ISO/IEC 13249-3:2016 standard. The structure is equivalent, however the attributes tables differ slightly (PostGIS 2016).

Nodes:

id	containing_face	geom

The *id* attribute is unique for every node. The value is an integers, starting from 0 and increasing by 1 as every new node is created. The *containing_face* attribute has the value of the face id of the face it is located in. If the node lies in the universal face, the value is NULL. The *geom* attribute holds the geometry of the node (the coordinates).

Edges:

id	start_node	end_node	next_left_edge	abs_next_left_edge	next_right_edge	abs_next_right_edge

left_face	right_face	geom

The *id* attribute is unique for every edge. The value is an integers, starting from 0 and increasing by 1 as every new edge is created. The *start_node* and *end_node* attributes are the id of the start and end node. The *next_left_edge* and *next_right_edge* attributes are the id of the closest connecting edge when looking in the clockwise direction from the end point respectively the start point of the edge itself (for further description, see section 5.1). The *abs_next_left_edge* and *abs_next_right_edge* attributes are the absolute values of the next_left_edge and next_right_edge attributes. The *left_face* and *right_face* attributes are the id of the faces lying to the left respectively to the right of the edge when looking at the edge in the direction start point to end point. If there is no face to the left and/or to the right of the edge, the value is 0, representing the id of universal face (for further description, see section 5.1). The *geom* attribute holds the geometry of the edge.

Faces:

id	geom

The *id* attribute is unique for every face. The value is an integers, starting from 0 and increasing by 1 as every new face is created. The *geom* attribute holds the geometry of the face (PostGiS 2016).

5.6 Spatial Query Language

Query languages are used to interact with a *database management system* (DBMS). *Structured database query language* (SQL) is a recognized language for *relational database management system* (RDBMS). A *spatial database management system* (SDBMS) is an extension of a RDBMS dealing with both spatial and non-spatial data. SQL, and formal query languages such as *relational algebra interpreter* (RA), can only provide simple datatypes: integer, date, string etc. In SDBMS, complex data types such as points, lines and polygons must be handled, an extension of the SQL is therefore required (Shekhar and Chawla 2003).

The *Office of Government Information Services consortium* (OGIS) introduced an *OGIS spatial data model* based on the geometry data model in figure 5.7. The base class is the Geometry class, which is non-instantiable. There are four subclasses: Point, Curve, Surface and Geometry Collection. Associated with each subclass is a set of operations that act on instances of classes.

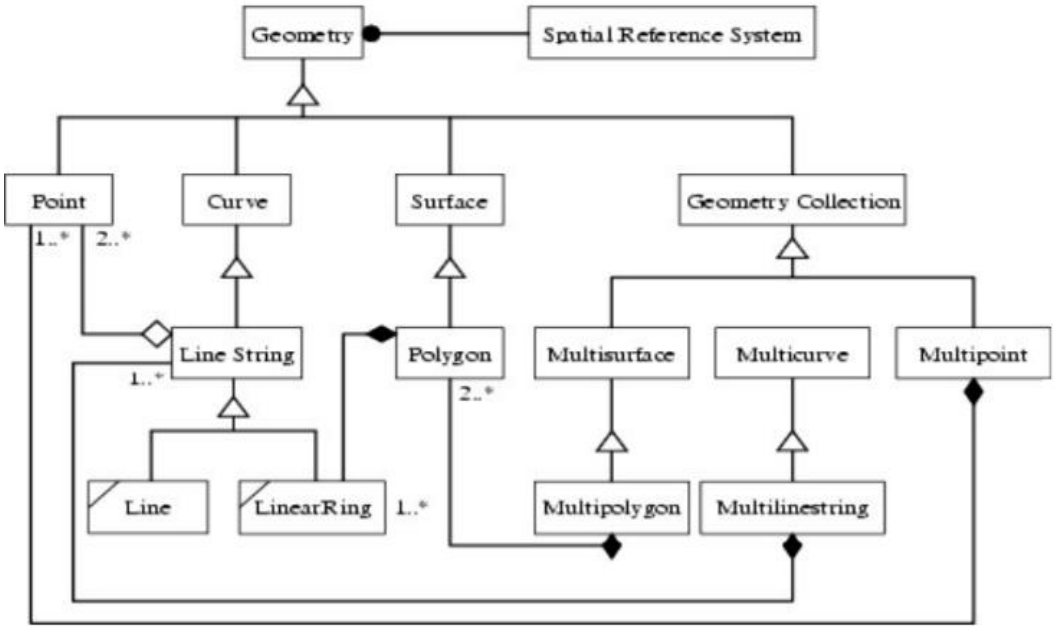

Figure 5.7. An OGIS proposal for building blocks of spatial geometry in UML notation (Figure 2.2, Chapter 2 in Shekhar and Chawla 2003).

An efficient way to expand the capabilities of DBMS to complex objects is through object-oriented systems. Extended relational databases with object-oriented features falls under the general framework *OR-DBMS* (object-oriented database management system). To query the OR-DBMS, *SQL3/SQL 1999* is used. It is the standardization platform for the object-relational extension of SQL. It is not specific for GIS or spatial databases, it covers object-relational extensions of SQL in general. Within GIS, the natural scenario is to implement the OGIS standard recommendations in a subset of SQL3 when developing spatial query languages (Shekhar and Chawla 2003).

When working in a SDBMS, it is valuable to know how the spatial objects interact with each other. The underlying *embedded space* determines the relationship that can exist between the objects; there is e.g*. set-oriented space, topological space, directional space* and *metric space*. Regarding the topological space, there are two common queries types:

- Find all objects that have a topological relationship between two objects.
- What is the topological relationship between object A and B?

In an object-based model the topological relationships are found using the 9-intersection model. Table 5.7 shows some examples of recognized topological operations (Shekhar and Chawla 2003).

Table 5.7. Examples of Topological Operations and Scenarios (Table 2.1, Chapter 2 in Shekhar and Chawla 2003).

Topological Operations	**Scenarios**
endpoint(point, arc)	A point at the end of an arc.
simple-nonself-intersection(arc)	A nonself-intersecting arc does not cross itself.
on-boundary(point, region)	Vancouver is on the boundary of Canada and the United States.

inside(point, region)	Minneapolis is inside of Minnesota city.
outside(point, region)	Madison is outside of Minnesota city.
open(region)	Interior of Canada is an open region.
close(region)	Carleton Country is a closed region.
connected(region)	Switzerland is a connected region, whereas Japan is not (given any two point in the area, it is possible to follow a path between the points entirely within the area.
inside(point, loop)	A point is within the loop.
crosses(arc, region)	A road (arc) passes through a forest (region).
touches(region, region)	Interstate freeway 90 (arc) passes by Lake Michigan (region).
overlap(region)	Land cover (region) overlaps with land use (region).

6 System Architecture

6.1 Requirements

Requirements are set up for the client application, specifying how it is expected to work. The requirements are written based on standard operations in the PostGIS database, the use cases and the problem statement. *User requirements* specify what the user should be able to do while using the client application. *Functional requirements* specify the functionality performed by the client application. These are strived to be fulfilled during the implementations phase.

User requirements

1 Edges

1.1 The user should be able to create isolated edges.

1.2 The user should be able to create edges that connect to one or several other edges by

- connecting its own start and/or end point to the other edge(s) start and/or end point(s).
- connecting its own start and/or end point to an arbitrary point along the other edge(s) geometry.

1.3 The user should be able to remove an isolated edge.

1.4 The user should be able to remove an edge connected to other edges by

- connecting its own start and/or end point to the other edge(s) start and/or end point(s).
- connecting its own start and/or end point to an arbitrary point along the other edge(s) geometry.

1.5 The user should not be able to create an edge intersecting itself (non-simple geometry).

1.6 The user should not be able to create an edge intersecting another edges geometry, if it is not with its own start and/or end point (see functional requirement 3.3).

2 Faces

2.1 The user should be able to create a face by enclosing a surface when creating an edge.

3 Interface and drawing tools

3.1 The user should be able to create an edge by drawing a line string.

3.2 The user should be able to remove an edge by selecting it and confirming the removal.

Functional requirements

1 Nodes

1.1 A node should be created at the start point and at the end point of all new edges. However, if there is already a node in the start point or the end point, a new node should not be created there.

1.2 The start node and end node of an edge should be removed when the edge is removed. However, if another edge is connected to the node, the node should not be removed.

2 Edges

2.1 When creating a valid edge, the edge should be added to the topology and the attributes of the edge should be updated accordingly to existing topology.

2.2 An edge should be split if another edge connects to the edge with its start and/or end point in an arbitrary point that is not the edges own start and/or end point.

2.3 When creating an edge intersecting itself or another edge, the edge should not be added to the topology.

3 Faces

3.1 A face should be created and added to the topology when a surface is enclosed.

3.2 A face should be removed when one of the edges enclosing the surface of the face is removed.

4 Interface and Drawing tools

4.1 The client application should provide an intuitive interface where topological data can be manipulated. Most desirable with a background map.

5 Data Management

5.1 The client application should be able to store the data on local files.

5.2 The client application should be able to load from a file with existing data.

6.2 Overview of Techniques

There is no obvious way to implement a client application that meets the requirements set up in the previous section. A general system architecture can however be presented. Figure 6.1 is based on figure 1.1 in section 1.2, additionally showing suggestions of languages, libraries, formats and standards possibly applicable in the different stages. They are only mentioned in this section, the ones actually used in the implementation are described further in section 6.3 to 6.7.

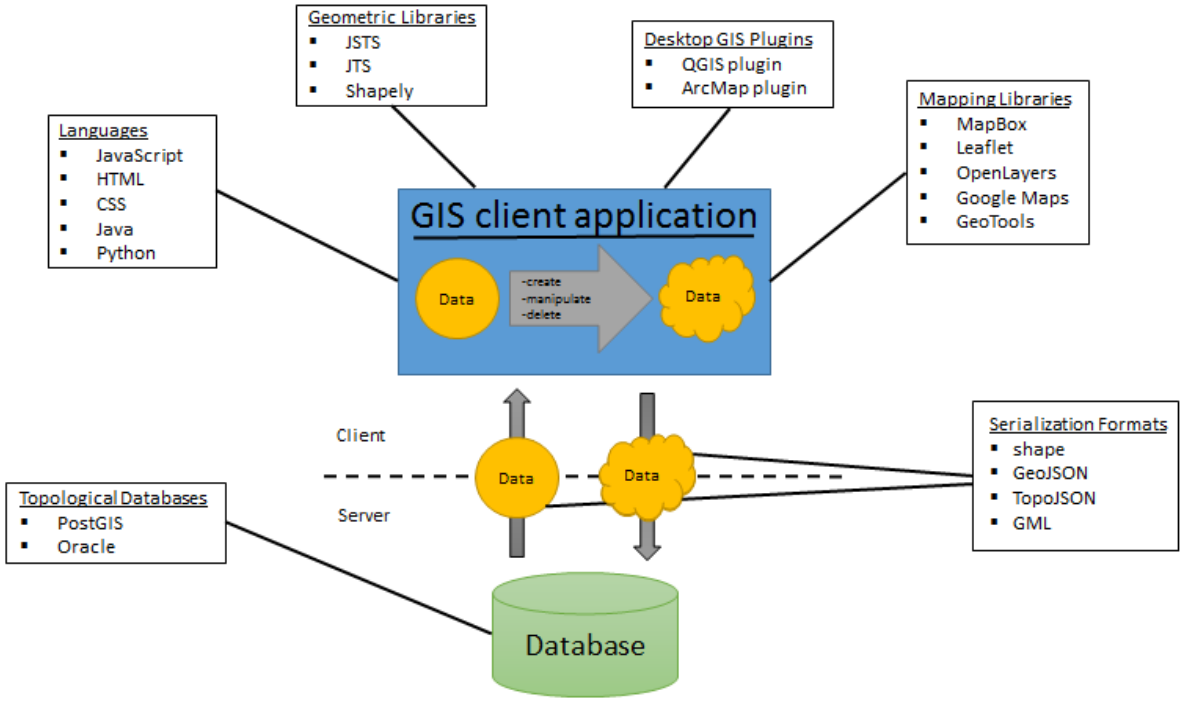

Figure 6.1. Based on figure 1.1, additionally showing candidates for languages, libraries, formats and standards to use when implementing the GIS client application.

The database needs to support a topological data structure. There are both commercial databases (Oracle Spatial) and open source databases (PostGIS) implementing a topological data structure.

Regarding the GIS client application, several aspects need to be taken in to account. Whether the client is a web application or a native application will have an impact on what languages to use. For a web application the predominant choice is JavaScript, it is naturally combined with HTML and CSS to create the web page. For a native application, there are many languages available. Most suitable are possibly the languages with access to a mapping library. GeoTools is for instance a mapping library for Java.

To preserve the topological relationships, geometrical calculations are required. It could be of interest to use libraries offering support for geometrical operations. Available libraries are for instance JTS (JTS Topology Suite) for Java, JSTS (JavaScript Topology Suite) for JavaScript and Shapely for Python.

Mapping oriented libraries simplifies making the graphics of the application. There are a number of established mapping libraries, such as MapBox, Leaflet, OpenLayers, Google Maps and GeoTools. OpenLayers and Leaflet offers the functionality of storing geometries corresponding to the topological element (node, edge and face) as features, which could be a possibly storage method for the data. Leaflet has an extension where topologically structured data (for instance from a database) can be modified using a drawing tool. The user can move the data in ways corresponding to a topological transformation. However data cannot be added and deleted.

Another alternative is to build the client application as a plugin on top of a desktop GIS. The graphical environment and the mapping tools are then easy to access. ArcMap is a commercial GIS platform that offers both server and desktop versions. QGIS is an open source alternative, offering an intuitive plugin development environment.

Transferring the data between the database and the client application requires a serialization format. The topological data consists of geometry data and attribute data. Candidates are accordingly standards and formats managing this kind of information. Common formats and standards available are GeoJSON, TopoJSON, Shape-files and GML. Developing a custom-made format is a possibility.

6.3 HTML

HyperText Markup Language (HTML) is a standard language for creating web pages. There are several established markup languages, although HTML is most accepted and used (Musciano & Kennedy 2002). It is built on tag elements holding all the information. A page usually consists of two basic elements, a *<head>* tag and a *<body>* tag. The rest of the tag elements are places within one of these two (Fulton and Fulton 2011). When the page is rendered the tags themselves are not displayed, instead they tell the browser how to interpret the imbedded content (text, images, other media). The language also allows interaction through specials hypertext links, which connect the document to other documents on the computer or through other internet resources. Musciano & Kennedy states that the "…fundamental purpose is to define the structure and appearance of documents and document families so that they may be delivered quickly and easily to a user over a network for rendering on a variety of display devises." The latest version is at this day (2016) HTML5.

6.4 JavaScript and jQuery

JavaScript is the language of the web browser and one of the most used programming languages in the world. It has a powerful object literal notation. It can create objects simply by listing their components, which moreover was the inspiration for JSON (Crockford 2008). When combining it with the DOM (Document Object Model) defined by the browser, it enables the creation of dynamic HTML content and interactive client-side web applications. The syntax is based on the programming languages C, C++ and Java, which makes it easy for experienced programmers to interpret the language. At the same time, it has proven flexible and offers a forgiving environment for new programmers (Flanagan 2001). Many people use it without ever having learned it, resulting in a wide spread of opinions about it. Crockford (2008) states: "It is built on some very good ideas and a few very bad ones." JavaScript is defined under the standard ECMAScript programming language (Crockford 2008). ECMAScript 6 (released in June 2015) is the latest official version of JavaScript (W3S 2016).

JQuery is a JavaScript library. It has an easy-to-use API (Application Programming Interface) that works with most of the well-established browsers. The library is fast and feature-rich, as well as makes HTML document manipulation, event handling and animations easier (jQuery

2016). Because many programmers have better experience with HTML and CSS than with JavaScript, jQuery is designed to give the unexperienced programmer a quick start (Chaffer and Swedberg 2013).

6.5 CSS

Cascading Style Sheets (CSS) is a stylesheet language for styling HTML or XML documents. It describes how the elements must be rendered on screen, on paper and in other media. External stylesheets can be stored in CSS files and used to style multiple webpages at once. The entire look of a webpage can be renewed by simply changing the CSS file. The language was developed by the World Wide Web Consortium (W3C) to remove style formatting from HTML pages (W3S 2016).

6.6 OpenLayers

OpenLayers is a JavaScript web mapping library. According to the official webpage it is: "A high-performance, feature-packed library for all your mapping needs." (OpenLayers 2016) It was first developed in 2006 as an open source alternative to Google Maps that arrived in 2005. Google Maps was early to popularized geographical information applications in the web, and OpenLayers was fast to follow. OpenLayers is compatible with many well-known standards including HTML5, GML, KML and GeoJSON. The latest version is at this day (2016) OpenLayers3. The library offers all the required components necessary to work with geographic information on the browser side (Santiago 2015).

6.7 JTS and JSTS

JTS Topology Suite (JTS) is an open source Java library with functions for geometries in the 2-dimensional Cartesian plane. It provides a complete, consistent and robust implementation of fundamental algorithms using an intuitive API. JTS conforms to the Simple Features Specification for SQL published by the Open Geospatial Consortium (JTS 2016).

JavaScript Topology Suite (JSTS) is a library corresponding to JTS, only it is intended for JavaScript rather than for Java. The library was developed using automatic translations of the original JTS Java source, preserving the JTS API, as well as by manually porting the io-related classes supporting WKT, GeoJSON and OpenLayers3. JSTS is an ECMAScript 2015 library. Just as JTS, it strives to provide web mapping applications with a library for processing and analyzing simple geometries. It can also be used as a free standing geometry library (JSTS 2016).

7 Case Study

The case study of this report centers on developing a GIS client application for loading, creating, deleting and rendering topological data. In this chapter, the technical choices are initially presented. The process of the implementation is then described, where the difficulties and changes of thought during the implementation are pointed out.

When writing “the client application”, it is referred to *the topologically structured GIS client application* developed in this project.

7.1 Technical Choices

The client application is developed as a web based application running in a web browser from a local file. Figure 7.1 shows the languages, libraries, formats and standards chosen for this project (compare with figure 6.1 in section 6.2).

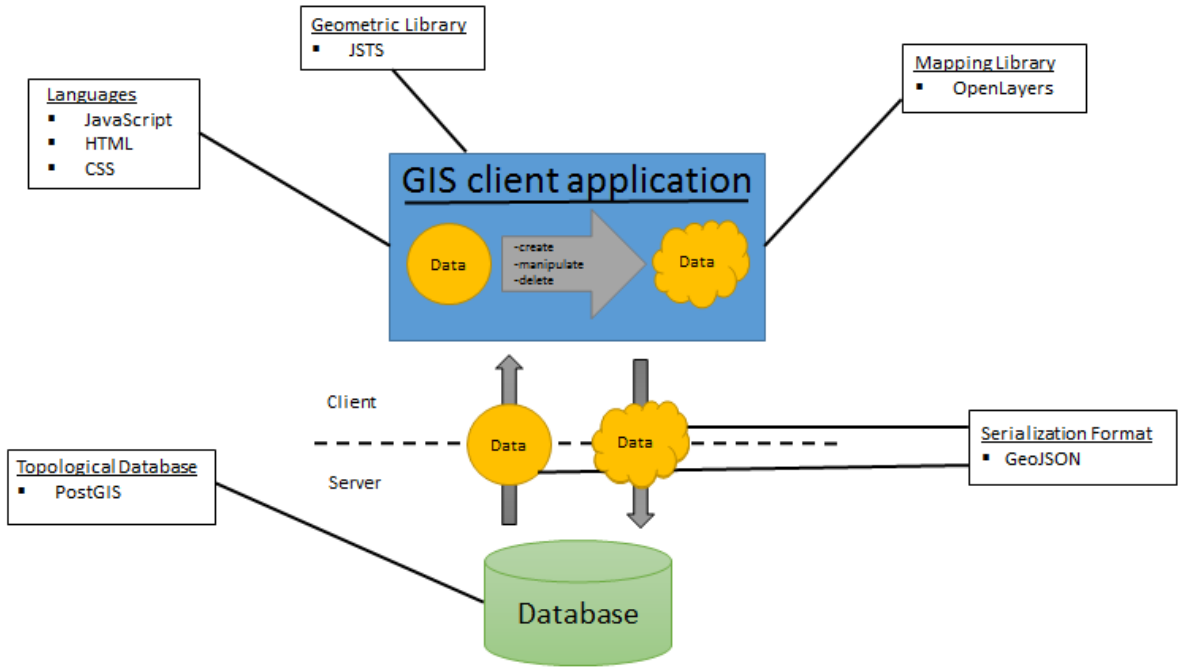

Figure 7.1. The languages, libraries, formats and standards used for the developed client application.

The main reason for making a web based application was the anticipation to succeed if using the tools known available and the previous experience regarding the same type of projects. There is however nothing indicating that another type of solution would be less successful or more complicated to implement. In the wider perspective, it would be of use if the client application was compatible with different types of devices including mobile phones and tablets. The question whether to make a hybrid application (similar to this web bases solution but suitable for apps) or a native application would then become more significant.

HTML, CSS and JavaScript are well-established languages when working within web development. They are commonly used together as they integrate well. They were chosen

for this project because they seemed appropriate. Mapping libraries, code examples and other documentations are available, factors that can help the implementation proceed.

OpenLayers was mainly selected due to recognition in comparison to other mapping libraries. Also, the majority of the functionality is complemented with well-designed examples involving HTML and JavaScript.

Beforehand, TopoJSON was assumed to be an appropriate serialization format as it can represent topological data. While implementing, it became clear that the attribute data constitutes an essential part of preserving and controlling the topological relationships. TopoJSON does not offer the same flexible attribute management as GeoJSON does, why GeoJSON was chosen instead. The fact that the OpenLayers API offers functions that write OpenLayers features to GeoJSON objects, encouraged the choice.

7.2 Technical Environment

The client application runs in an arbitrary web browser, which is all that is required for the topological operations to work. However, to make the interface more user friendly and graphically engaging, a map is rendered in the background. The map currently used is loaded from the OpenLayers library, using internet to do so. Both a web browser and internet access is therefore advised for optimal usage.

7.3 System Architecture

The client application can functionality wise be divided into an *application part* and a *library part*. Figure 7.2 illustrates this while showing the file structure of the client application. The top level of the architecture represents the application part, where the logic is made available to the user through an interface. The bottom level of the architecture represents the library part, or the *topological library*, holding all the functions performing the topological operations.

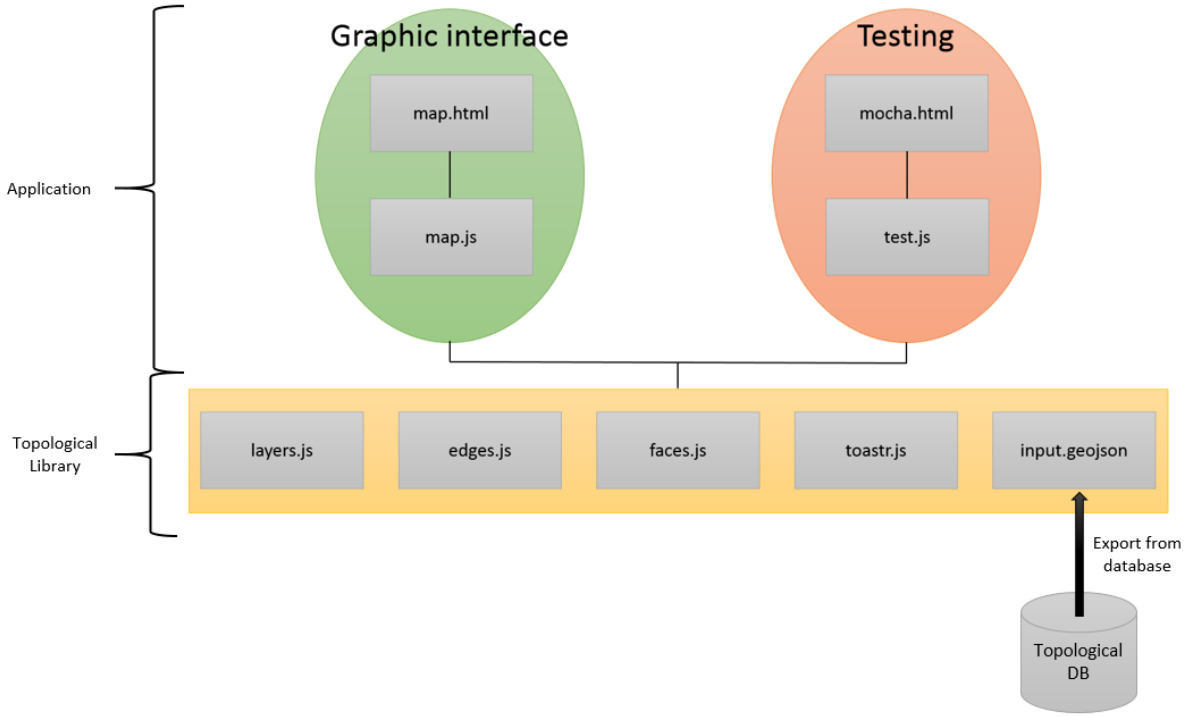

Figure 7.2. The File Structure of the client application. Functionality wise separated into an *Application part* and a *Topological Library part*.

The client application is run through an HTML file in a web browser; in figure 7.2 the file *map.html* is used for the graphical version whilst *mocha.html* is used when testing. The file map.html contains objects forming an interface. It includes objects such as a background map, buttons and information panel (see figure 7.3). The objects are styled using CSS. 0

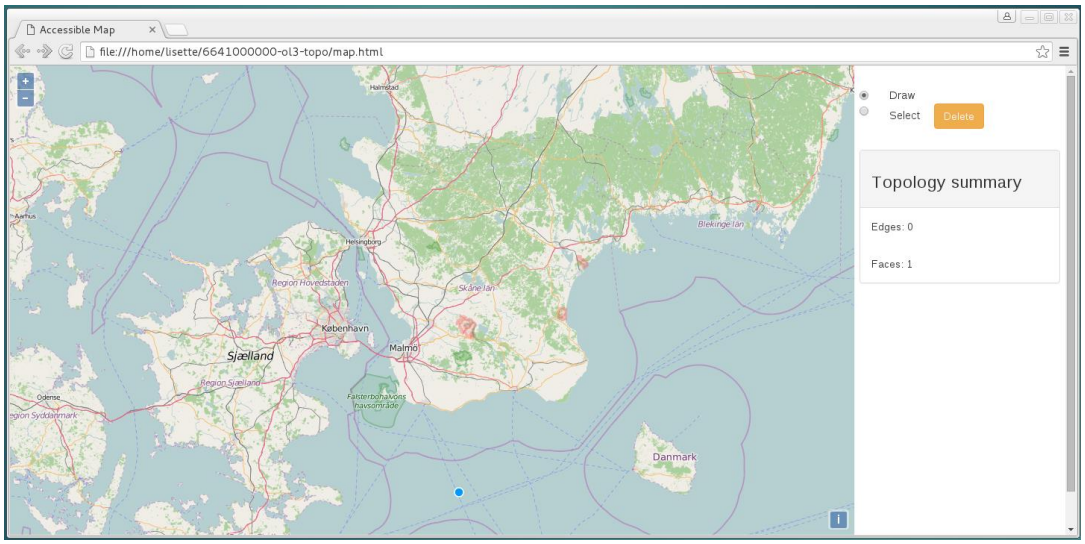

Figure 7.3. A print screen of the Interface of the client application.

When testing purely the logic of the client application mocha.html in used to avoid time consuming processes such as drawing the edges, as well as for testing subparts of the functionality. The JavaScript test framework *Mocha* is utilized for a structured and easy testing process.

The logics are distributed between several JavaScript files. The JavaScript files placed in the application part in figure 7.2 (*map.js* and *test.js*) works as a connection between the HTML file and the files in the topological library (*layers.js*, *edges.js*, *faces.js* and *input.geojson*). In map.js the style of the background map is set using the OpenLayers library.

The functions making sure the topological rules apply are gathered in layers.js, edges.js and faces.js. layers.js holds the OpenLayers Vector Layer (*ol.layer.Vector*) that contains the topological data. The data are stored in OpenLayers Geometries; LineStrings (*ol.geom.LineString)* are used for edges (both for visualizing them and for storing their attribute data) and Polygons (*ol.geom.Polygon*) are used for faces (only to visualizing them). Functions that interact with the Vector Layer are placed here. edges.js and faces.js contain functions that changes the state of the topology. More thoroughly, the calculations taking place when adding, deleting and splitting edges, which in turn affect the faces, are performed here.

toastr.js contains settings for the non-blocking notification that are shown when the user tries to add invalid edges. toastr is a JavaScript library, it requires jQuery.

input.geojson can be loaded into the client application as an initial action. If the file is empty, the user starts with an empty topology containing only the universal face. If the file contains a non-empty topology, the data are read and loaded into the client application and the topology is used as start value.

7.4 Implementation

To fully understand how to implement the rules that apply when handling topological data, PostGIS database is used as a valuable tool throughout the process. Initially, test data are created and studied with regards to attributes and the relations taking place. The standard functions are used on different sets of data to recognize the changes and differences in the result. This is done to get an idea of what is expected as basic functionality when working with topologies and to figure out how to start the implementation process.

7.4.1 Node

In PostGIS the user must create nodes in order to create edges. Every edge is bound to have a start and an end node. This rule is a part of the basic concept and is therefore naturally applied in the GIS client application initially. A node layer is created early on, to store the node data and to make them available for visualization. The function *createNode()* is the first function to add data to the topology. The users is at this point forced to add a node before adding an edge.

The user experience is later discussed, where the fact that the user has to create nodes before creating edges seems unnecessary. This is adjusted in a way where the user can draw an edge straight away, the start and end coordinates are then collected and made into nodes automatically.

In a later state of the implementation process, when most of the major logical operations are working, it is recognized that creating nodes does not bring any advantages regarding the functionality. The calculations are based on the start and end coordinates of the edges, not on their start and end nodes. Besides in visualization purposes, the client is not affected if not creating node objects anymore. This whole concept of creating nodes is thus neglected.

7.4.2 Edge

The first step towards creating edges is the function *createEdge()*. The function creates an OpenLayers Geometry (ol.geom.LineString()), taking the coordinates of the edge as argument. The decision to store an edge in an ol.geom.LineString() is based on the fact that the ol.geom.LineString() is easy to render on the background map. Also, the user can define and add attributes to the OpenLayers Geometry, which are easy to access through a get function.

The edges in the PostGIS database have ten attributes: edge_id, start_node, end_node, next_left_edge, abs_next_left_edge, next_right_edge, abs_next_right_edge, left_face, right_face and geom. To copy this structure the ol.geom.LineString() is given the corresponding attributes: id, startNode, endNode, nextLeftEdge, nextRightEdge, nextLeftAbs, nextRightAbs, leftFace, rightFace and coordinates (the geometry). The collected data for each edge is captured in an OpenLayers Feature (*ol.Feature()*), see figure 7.4.

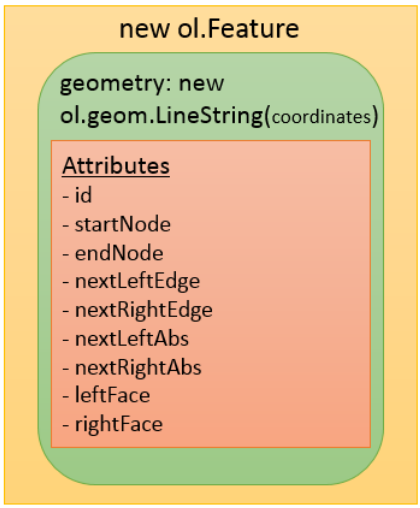

Figure 7.4. The structure of the edges. An OpenLayers Feature (ol.Feature()), holding an OpenLayers Geometry (ol.geom.LineString()), holding the geometry (coordinates) and the rest of the attributes.

The *id* is received from an integer increasing every time an edge is created. The id number of an edge is always reserved to that edge precisely. If removing an edge, the next thereafter created edge will not get the id of the deleted edge, even though that number is now available. The first edge is given the id 1 and is thereafter increased by whole numbers. Id 0 is not used for the edges because the id of the edge can be referenced to both as a positive or negative integer.

The *startNode* and *endNode* attributes point to the id:s of the start and end nodes of the edge in the PostGIS database. As mentioned, the concept of nodes are not implemented for the client application. Instead, the attributes hold the coordinates of their respectively start and end point directly in the attribute.

The *nextLeftEdge* is the positive or negative id of the closest connecting edge when looking clockwise from the endpoint of the edge itself (see section 5.1 for a further description of the attribute). Finding the nextLeftEdge attribute is done using angles. The exception of not having any connecting edges in the end point is first handled. Functions checking if the edge is isolated (*isIsoEdge()*), alternatively isolated in the end point (*isoAtEnd()*), are used to avoid preforming calculations when not needed. When there are one or more connecting edges in the end point, the function *closestEdge()* is called to determine which edge has the smallest angle from the edge itself when going in a clockwise direction. The functions used to perform the calculations are described in section 7.4.7.

The *nextRightEdge* attribute is determined in the same way as the nextLeftEdge attribute. The difference is to find the closest connection edge from the start point of the edge when looking in the clockwise direction. The functions isIsoEdge() and *isoAtStart()* are used to rule out the exception of not having any connecting edges in the start point. When there are one or several connecting edges in the end point, the function closestEdge() is called. Again, see section 7.4.7 for further descriptions.

The *nextLeftAbs* and *nextRightAbs* attributes hold the absolute value of nextLeftEdge respectively nextRightEdge. The value is determined by simply making the values of nextLeftEdge and nextRightEdge positive. The attributes are initially added since they exist in the attribute list of the PostGIS database. At the final state of the implementation it is recognized that they are only used a handful of times. Using the nextLeftEdge and nextRightEdge attributes instead, just making their value positive, is an alternative solution. However, the attributes do not bring any awkwardness to the implementation and could possibly be useful in further developments of the client application.

The *leftFace* and *rightFace* attributes have the default value of 0; if no faces are created when adding the edge they will point to the universal face. The default value is set when the edge is created, and later updated if needed after checking for new faces. This all happen as a part of the process of adding an edge. If the edge is isolated (isIsoEdge()), isolated at the start point (isoAtStart()) or isolated at the end point (isoAtnd()), no new faces could of appeared and the values are not updated. If the edge on the other hand has connecting edges at both ends, the function *walkingOnEdgeSide()* is called. This function is describes further under the section 7.4.8.

7.4.3 Face

A face cannot be added to the topology as an operation of its own, it is rather just a consequence of the way the edges are placed. If an edge is added in such way that a surface in enclosed, a face should be created. Adding an edge can thus lead to none, one or two

(one on each side of the edge) new faces in the topology. The faces are given an *id*, starting at 1 (since the universal face has id 0) and increasing by whole numbers. The id number is, as for the edges, unique for every face and cannot be reused for another face after becoming available. A face does not hold any other information than the id value. At the testing stage of the implementation, the faces are therefore simply integers stored in an array to keep track of what faces exists. When working in the interface, the faces are additionally stored in the ol.geom.Polygon(). This way they can be graphically distinguished together with the edges.

A face is created using the functions *createFace()* and *olPolygon()*. createFace() simply adds to the array storing the id:s of all existing faces. The function olPolygon() uses the return value of the function walkingOnEdgeSide(), which is an array with all edges constituting a face, to collect the geometries of the edges and create the ol.geom.Polygon(). The ol.geom.Polygon() is captured in an OpenLayers Feature with the structure shown in figure 7.5. As mentioned, the OpenLayers Polygon is only used for graphical purposes.

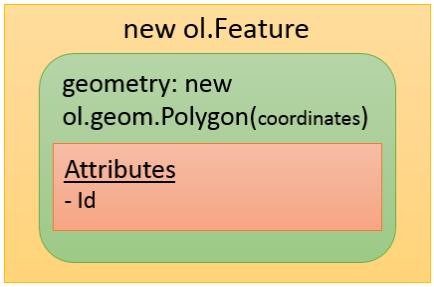

Figure 7.5. The structure of the faces. An OpenLayers Feature (*ol.Feature()*), holding an OpenLayers Geometry (ol.geom.LineString()), holding the geometry (coordinates) and the id attribute.

7.4.4 AddEdgeNewFace

The function *addEdgeNewFace()* is called when trying to add an edge to the topology. It contains a number of controls making sure the edge is only created and added if it agrees with the topological rules, as well as updating the topology when the new data are added. In the databases studied, there are a hand full of operations for adding an edge. When using the standard operations, the user must be aware of what type of edge to add and what the consequences of adding the edge will be. For example, the user can add an isolated edge, an edge creating a new face or an edge slipping an existing face. However, all of these actions require different operations. The user does not have to tell the client application what kind of edge is to be added. Instead, the controls and calculations preforms under this function will make sure the edge it added in a correct way. An activity diagram over the function is shown in figure 7.6 (using the UML Activity Diagram references from Microsoft).

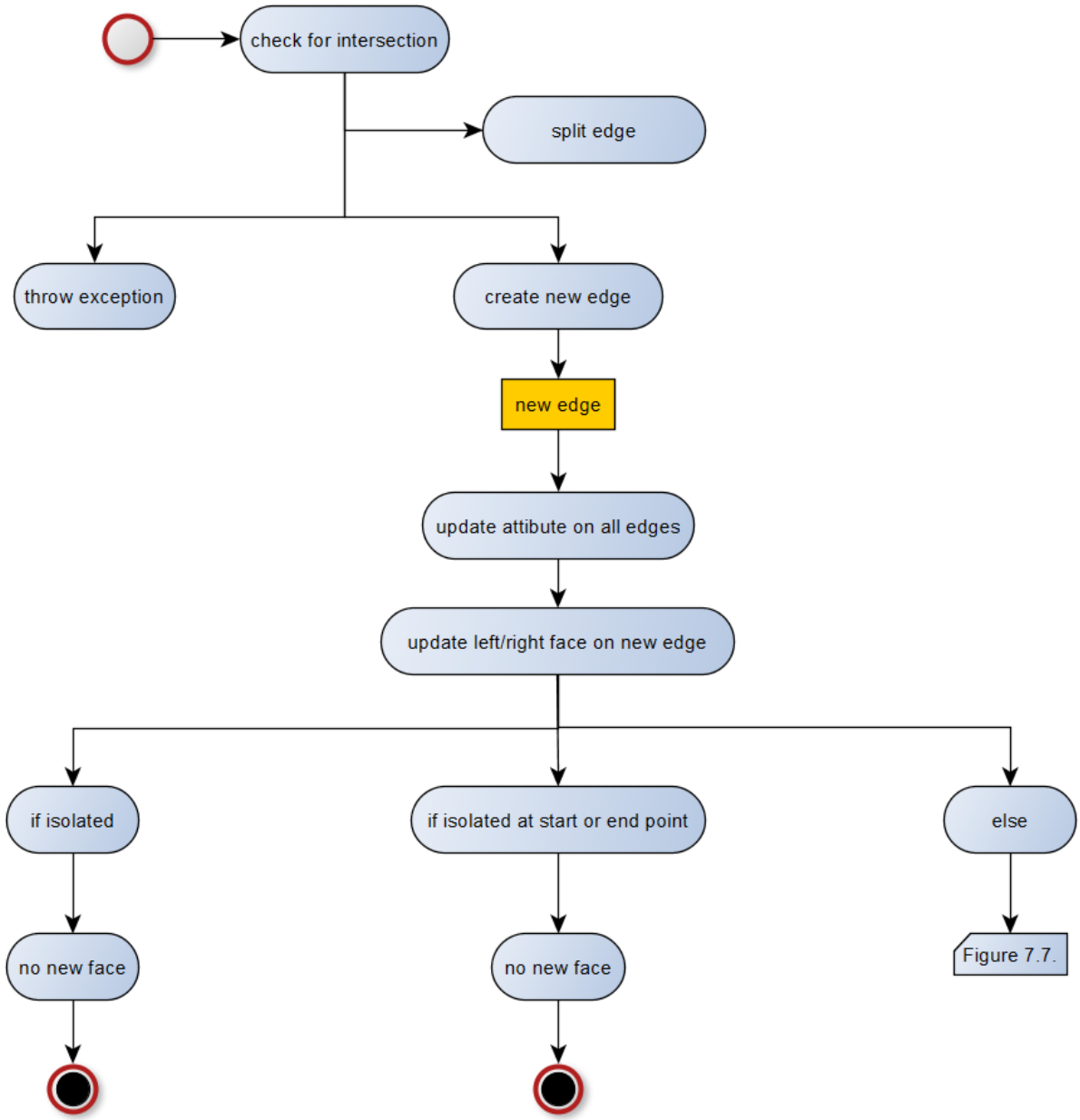

Figure 7.6. An activity diagram over the function addEdgeNewFace(), part 1.

The function takes the coordinates of the new edge as input. The new edge is not allowed to intersect the existing edges. However, if the start or the end point of the new edge intersects an edge, the existing edge is to be split in this point (splitting the edge is described further in section 7.4.6). If the intersection check is valid (there is no intersection or an edge split intersection), the new edge is created and its next left/right edge attributes are set. The next left/right edge attributes of all the connection edges are then updated. The left/right face attributes are set for the new edge accordingly to the connecting edges. The values of leftFace and rightFace must be equal, else the edge is not added in a correct way. When the attributes of the new edge are set, potential new faces are detected. If the new edge is isolated, alternatively isolated in the start or end point, no new faces could have appeared. No further actions are required and the function will return the id of the new edge. If the edge on the other hand have connection edges in both start and end point, one or two new

faces could have been formed. This case is represented by the else action in figure 7.6, which continues in figure 7.7.

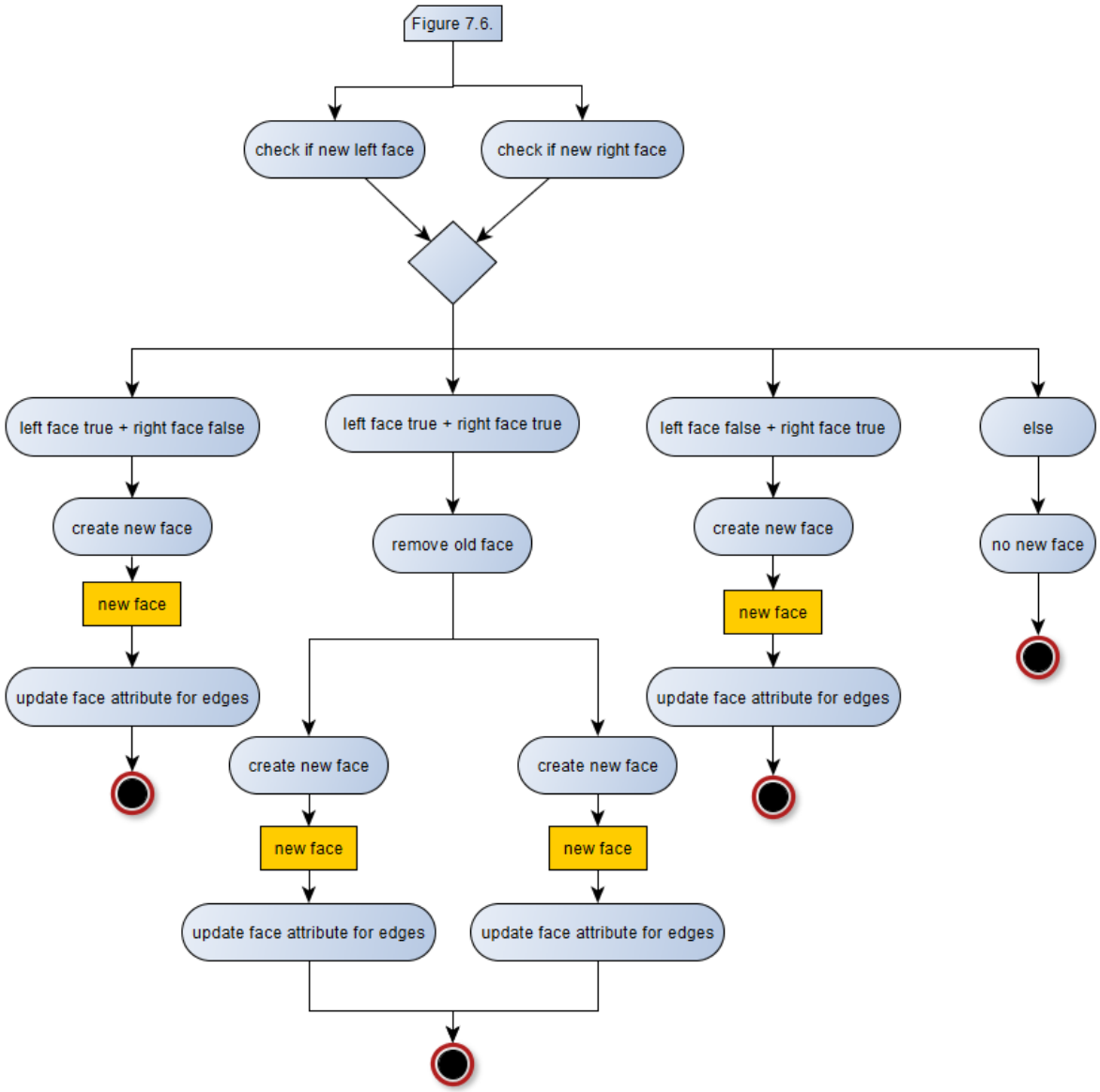

Figure 7.7. An activity diagram over the function addEdgeNewFace(), part 2.

Checking for new faces is done for the left and right faces separately (described further in section 7.4.8). If no new faces are found no further actions are required. If either a left or a right face is found, a new face is created and the left/right face attributes of all the involved edges are updated. If two new faces are detected, it is implied that an existing face is split. The existing face is removed, two new faces are create and all the left/right face attributes of all the involved edges are updated. The function then returns the id of the new edge.

7.4.5 RemEdgeNewFace

Removing an edge in a topology will most often affect other parts of the data. The exception is when the edge is totally isolated. To keep the topology correct the affected edges and faces need to be updated or (for the faces) removed. The function *remEdgeNewFace()* is

implemented in this purpose and will remove the edge as well as update the affected data. An activity diagram over the function is shown in figure 7.8.

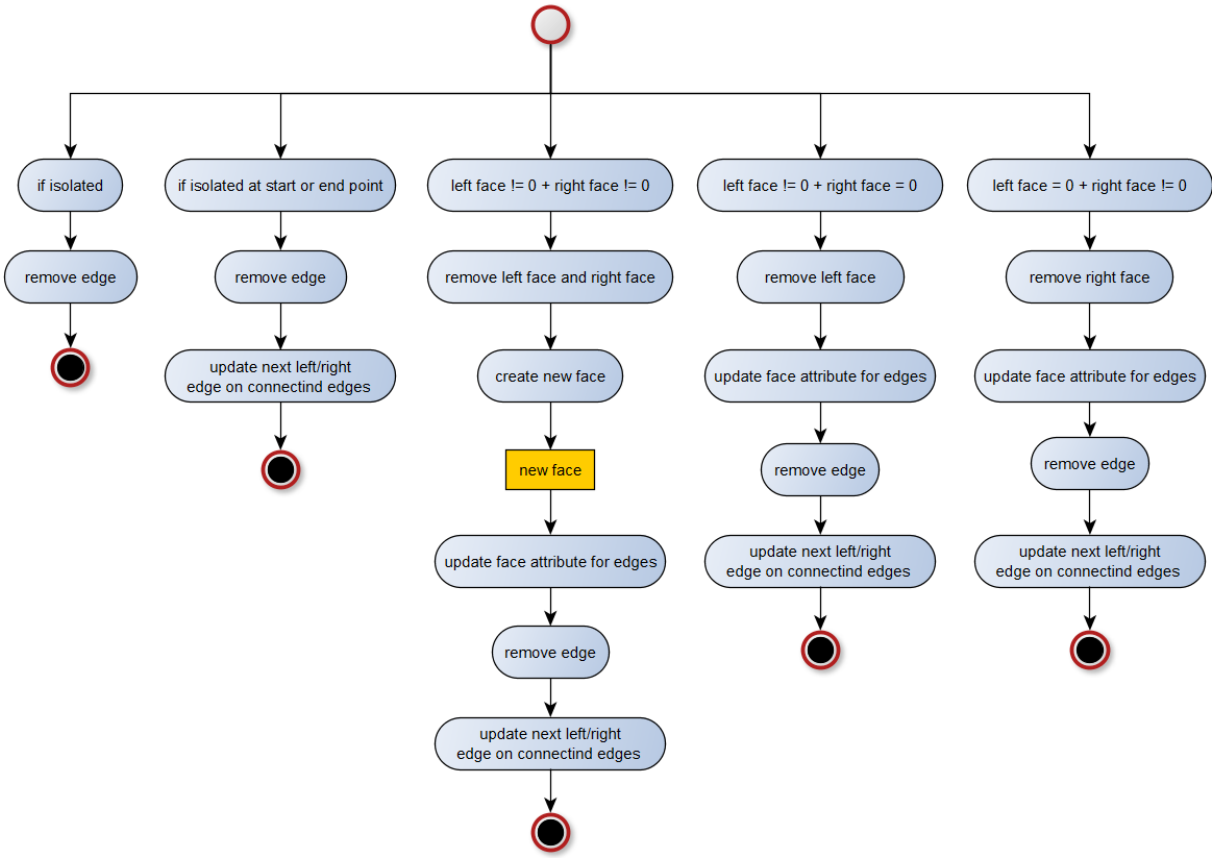

Figure 7.8. An activity diagram over the function remEdgeNewFace().

As mentioned, removing an isolated edge will not affect any other data and is done without further action. If the edge is not isolated but yet not a part of any face, the edges with attributes pointing to the removed edge are updated. If the edge is a part of one face, this face is removed and the left/right face attributed are updated to the value 0 (universal face). The edge itself is then removed and the edges with attributes pointing towards the removed edge are updated. If the edge is a part of two different faces, removing it can be seen as merging the two faces together. The old faces are thus removed and a new face is created in their place. The left/right face attribute are updated to the value of the new face. The edge is removed and the edges with attributes pointing towards the removed edge are updated.

7.4.6 SplitEdge

As mention, an edge can be added with different operations in the databases studied. One standard operations is to add an edge connecting an existing edge with its own start or end point in an arbitrary point along the existing edge. If the connecting point is not the start or end point of the existing edge, the edge should be split in this point, creating two new edges. The function *splitEdge()* is implemented in for this purpose. Since the user does not tell the client application what type of edge is being added, this action is a part of the process of addEdgeNewFace(). Splitting an edge is a type of intersection, only a permitted one. Before

creating an edge the function *intersections()* is called. If an intersection is found where an edge is to be split, the function splitEdge() is called. This action is accordingly a part of a preparation step, before the new edge is created. An activity diagram over the function is shown in figure 7.9.

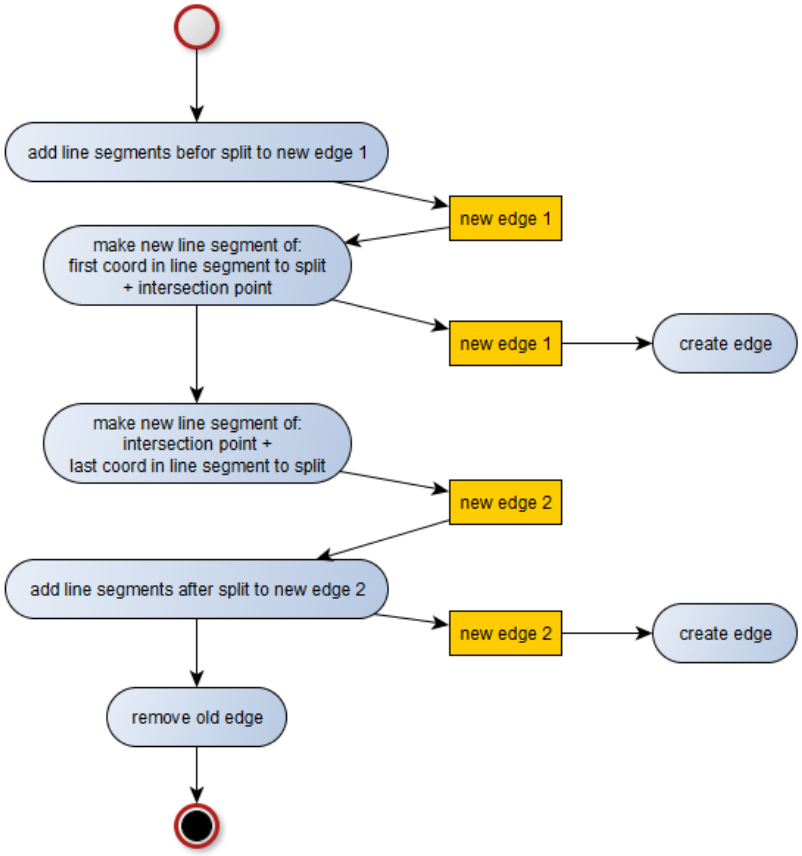

Figure 7.9. An activity diagram over the function splitEdge().

The *id* of the edge and the *index* of the line segment to be split, as well as the *intersection point* are used as input values when calling the splitEdge() function. The line segment index is used as a breaking point; all line segments before are added to one edge and all line segments after are added to another. A control is performed on the line segments to be split. If the intersection point is the line segments start or end point, the whole line segments is simply added to the correct new edge. If the intersection point is somewhere along the line segment, the segment is split into two and each part is added to one new edges respectively. The old edge is removed (with remEdgeNewFace()) and the two new edges are created (with addEdgeNewFace()).

7.4.7 Finding the Next Edge Attributes

In section 5.1 it is explained how the nextLeftEdge and nextRightEdge attributes are set. For the client application, this process is implemented through the functions *nextLeftEdge()*, *nextRightEdge()* and a number of help functions. The concept is to find the closest connecting edge from the start and end point of the edge when looking at the angle in a

clockwise direction. A description of the function nextLeftEdge() follows, assuming it is understood how the function nextRightEdge() works in a corresponding way.

When searching for the nextLeftEdge, the possible edges are the ones connected to the end point of the edge. All edges are looped over and the connecting ones are saved to a list. The angle between the edge itself and all the connecting edges are calculated and compared. Since the edges can consist of several line segments, the angles are calculated from the first or the last line segment, not the whole edge. The id of the edge with the smallest angle is returned. Depending on the direction of this edge (if it the first or the last line segment of the edge that has been compared) the id will be returned as a positive or negative integer.

The angles between two vectors $\mathbf{a}$ and $\mathbf{b}$ (representing line segments), are calculated using the *dot product* (scalar product), the *cross product* (vector product) and the *arctangent* function. The dot product is proportional to the cosine of the in-between lying angle and has the following relation (MathIsFun 2016a):

$$\mathbf{a} \cdot \mathbf{b} = a_x * b_x + a_y * b_y = |\mathbf{a}| * |\mathbf{b}| * \cos\theta$$

The cross product is proportional to the sinus of the angle and has the following relation (MathIsFun 2016b):

$$\mathbf{a} \times \mathbf{b} = |\mathbf{a}| * |\mathbf{b}| * sin\,\theta$$

The result of the cross product is a vector $\mathbf{v}$ defined in the 3-dimentional space, and can also be expresses as:

$$\begin{pmatrix} v(x) \\ v(y) \\ v(z) \end{pmatrix} = \begin{pmatrix} a_y * b_z + a_z * b_y \\ a_z * b_x + a_x * b_z \\ a_x * b_y + a_y * b_x \end{pmatrix}$$

Since only looking in the two dimensional plane in this implementation, the z axis value is set to zero. If eliminating all the multiples of 0, a simplified expression of $\mathbf{a} \times \mathbf{b}$ will appear:

$$\mathbf{a} \times \mathbf{b} = \ a_x * b_y + a_y * b_x \ = |\mathbf{a}| * |\mathbf{b}| * sin\,\theta$$

This expression is equal to the *determinant* of the two vectors when represented in a matrix. The angle is calculated using the dot product, the cross product (the determinant) and the relationship:

$$tan\,\theta = \frac{sin\,\theta}{cos\,\theta} \quad \left(\leftrightarrow \frac{sin\,\theta}{cos\,\theta} = \frac{\frac{a_x * b_y + a_y * b_x}{|\mathbf{a}| * |\mathbf{b}|}}{\frac{a_x * b_x + a_y * b_y}{|\mathbf{a}| * |\mathbf{b}|}} = \frac{a_x * b_y + a_y * b_x}{a_x * b_x + a_y * b_y} = \frac{\mathbf{a} \times \mathbf{b}}{\mathbf{a} \cdot \mathbf{b}} \right)$$

To ensure that the angle is looked at in the clockwise direction a normalized arctangents function is used. Arctangents is defined between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$, the function atan2 is used to complement this and is defined the following way:

$$\operatorname{atan2}(y,x) = \begin{cases} \arctan\frac{y}{x} & \text{if } x > 0, \\ \arctan\frac{y}{x} + \pi & \text{if } x < 0 \text{ and } y \geq 0, \\ \arctan\frac{y}{x} - \pi & \text{if } x < 0 \text{ and } y < 0, \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0, \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0, \\ undefined & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

The following calculations are performed for each edge in JavaScript code:

```
function angleBetweenVectors (a1, b1, c1) {
  var a = [b1[0] - a1[0], b1[1] - a1[1]]
  var b = [c1[0] - a1[0], c1[1] - a1[1]]
  var dotProduct = dot(a, b)
  var determinant = (a[0] * b[1] - a[1] * b[0])
  var angle = atan2Normalized(determinant, dotProduct)
  return (Math.round(angle * 1000) / 1000)
}
```

```
function dot (a, b) {
  var normal = 0
  if (a.length !== b.length) {
    throw new Error('line segment has to many break point')
  } else {
  for (var i = 0; i < a.length; i++) {
    normal += a[i] * b[i]
  }
  return normal
}
```

```
function atan2Normalized (x, y) {
  var result = Math.atan2(x, y)
  if (result < 0) {
    result += (2 * Math.PI)
  }
  return result
}
```

7.4.8 Finding new Faces

There are several function involved to check if any new faces have appeared after adding an edge. The edge is created and the next left/right edge attribute references to and from the edge are updated before checking for new faces. This information is used as a part of the control. The main function is walkingOnEdgeSide(), which in turn uses several help functions. walkingOnEdgeSide() takes as input the nextLeftEdge (if checking for the left face) or the nextRightEdge (if checking for the right face) of the new edge. It then checks the next edge attribute for that edge, and so on, until it has walked all the way back to itself. While doing this, all visited edges are saved to a list in the right order. Walking along the edges until finding the start edges again does not necessarily mean that a face has been found. Figure

7.10 show three different scenarios where a new edge is added and the walk from both nextLeftEdge and nextRightEdge will lead back to the edge itself. However, only after one of the six walks should a new face actually be created (top left square).

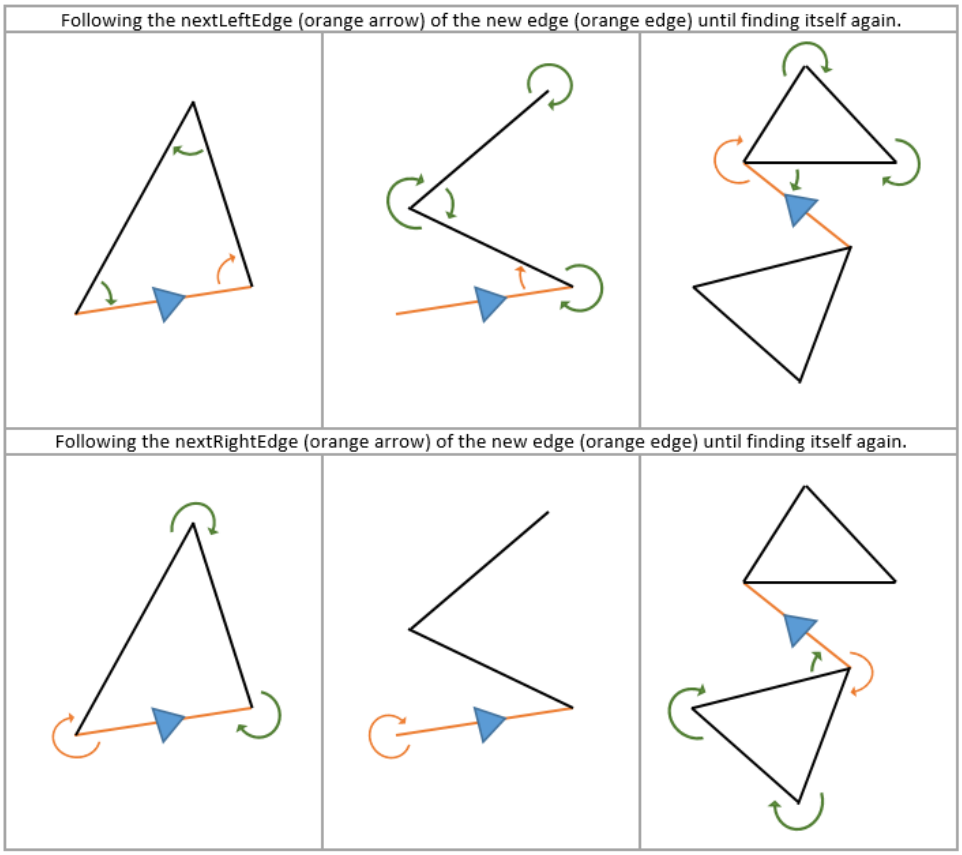

Figure 7.10. Three scenarios, six walks along the edges (left and right for each scenario), where only one of the walks should result in finding a new face (top left square).

To make sure that a face is created only when supposed to, *Gauss area formula* (also known as the *Shoelace formula*) is used. The formula is primarily used to determine the area of a polygon (convex or concave, not self-intersecting). To do so, it uses all the coordinates of the polygon. Depending on the order of the coordinates, clockwise or counter-clockwise, the resulted area will be positive or negative. This concept is used when establishing new faces. As mentioned, all visited edges during the walk is save to a list in the right order. Performing Gauss area formula with the coordinates of the edges in the given order will reveal if the walk constitutes a face (inside of a polygon) or not (outside of a polygon, like in the bottom left square in figure 7.10, or no polygon at all). Since the edges can consist of several line segments, all the break point needs to be considered in the calculations.

Gauss area formula has the following definition (Worboys and Duckham 2004):

$$area = \frac{1}{2} * \sum_{i=1}^{n} x_i(y_{i+1} + y_{i-1})$$

Figure 7.11 illustrates how the break points are defined in the formula.

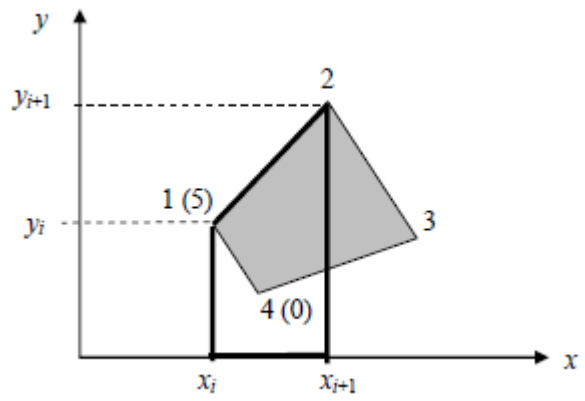

Figure 7.11. A polygon with four break point (Harrie 2014).

7.4.9 Logic and Graphic

The edges and faces are stored in OpenLayers Geometries to make them easy to render. Only the coordinates of the edges and faces are required to render them.

The edges are drawn on top of a background map. The map used is the *Open Street Map* (OSM), which goes under the open license ODbL (Open Data Commons Open Database License) (OpenStreetMap 2016). It uses the projection EPSG:3857, which is a Cartesian coordinate system representing the coordinates as x and y values *(x, y)*. The geometry of the edges and faces are described according to this projection in the client application.

7.4.10 Load to and from File

When saving a topology to a local file, the GeoJSON format is used. The OpenLayers Geometries representing the edges (ol.geom.LineString()) are converted into GeoJSON objects using the OpenLayers Feature *ol.format.GeoJSON* and the function *writeFeatures()*. The GeoJSON objects are written to the file as a string using the function *JSON.stringify()*.

When loading an existing topology from a file, the data string is interpreted into GeoJSON objects, which are converted into OpenLayers LineString (the chain backwards). The ol.geom.LineString holds all the attribute data of the edges. However, when adding them to the topology in the client application, only the coordinates of the edges are used. Just as when drawing the edges, the geometry is sent to the function addEdgeNewFace(), and new edges are created. The same relations between the edges will rule but the edges will most likely get new id:s, starting from id 1.

It might seem unnecessary to convert the topology to and from GeoJSON objects holding all the attributes of the edges, when only using the coordinate during the import. For this solution, simply selecting the geometries would be enough. However, writing to GeoJSON does not contribute with any additional work for this implementation. Also, it can be of value to know that it is possible to load all the attributes of the OpenLayers Geometries for further developments. Topological data stored in the PostGIS database can be imported to GQIS. From QGIS, the data can be exported to GeoJSON objects retaining all the attributes. This could be a possible conversion method.

7.5 Delimitations

The implementation phases is set to eight weeks. During this time the requirements in chapter 6.1 are strived to be meet. It becomes clear when past half time, where limitations need to be drawn in order to finish the implementation at a good stage. The client application can therefore be looked at as a prototype or a non-completed version of itself.

There are limitations due to the dependency towards the OpenLayers library. When using the drawing tool from OpenLayers a snap functionality is added to ensure that the right point is selected when clicking negligibly close to an existing point. An edge is not added to the topology or to the ol.geom.LineString() until the whole edge is drawn. This means that the snap function will not work towards the edge that is currently being drawn. It should be possible to add ring edges in a topology, (edges that have the same start and end point), however this is not possible in the client application due to the snap functionality limitations.

The client application preforms all the calculations based on the geometry of the edges and how they connect to each other. If a new edge connects an existing, its attributes are set in regards to the connecting edge. If the new edge is isolated it is assumed that it lies within the universal face and its attributes are set in regards to that. However, it should be possible to create an isolated edge inside a face that is not the universal face (a topology in a topology). If doing so in the client application the left/right face attributes will be incorrect (they will be 0). Furthermore, the case of connection this edge to other edges in the topology is not supported.

The way an existing topology is loaded to the client application, by only using the coordinates of the edges, is a severe limitation in the process of working with a subset of data. If the data are to be merged back into a database with other data, the attributes of the subset cannot be changed in a way that contradicts or duplicates the other data, or causes it to lose its connection with it. Loading topological data into the client application while keeping its exact attributes can be done, but requires loading the faces as well. The faces are normally created as a sub step of the addEdgeNewFace() function, which is not called when using this technique. Another significant aspect is to consider all the previously used id:s, possibly just the largest one. This concept is straight forward to apply, but due to time limitations it has not been tested in this project.

The reference system used in the client application is predefined and based on the projection of the background map. If loading data from a database that stores the data in a different reference system, issues could occur. To solve this limitation, an idea is to let the user select what reference system to work in before loading the existing topology (or creating a new one).

7.6 Result

The result is a GIS client application where the user can create a topology by adding and removing edge data. An edge is created using a drawing tool on top of a map. The geometry of the drawn edge is verified before the edge is added to the topology. The edge is not

added if its geometry intersects another edge in an incorrect way. If it intersects in a way where an existing edge is to be split, the split operation is performed before adding the new edge. When the edge is added, all the attributes of the edge is updated, as well as the attributes of all the affected edges and faces. When an edge is removed, the attributes of all the affected edges and faces are updated.

The client application has a simple and intuitive interface (see figure 7.2 and 7.12-7.15). There are two modes: drawing edges and selecting edges. In the drawing mode, the mouse arrow becomes a drawing tool when held over the map. A position is selected by a single click, an edge is confirmed complete by a double click. A double click at the first position will not result in an edge being added, it will just register that position as the start point of the edge. When in selection mode, the user can select an edge by clicking on it. The selected edge is marked blue. Only one edge at once can be selected. If the user wants to delete the selected edge, the **Delete** button is used.

The user can load a local GeoJSON file by pressing the button **Choose file**. The edges in the file are in that case added to the topology. The user can save the current topology by pressing the **Save Topology** button. The user is asked to pick a name for the file, after which it is saved to the Downloads folder.

The interface shows a summary of the amount of edges and faces currently in the topology. The empty topology consists of one face (the universal face) and zero edges.

Figure 7.12 – 7.15 below show the process of moving an edge between two faces. The user simply needs to selects the edge, delete it and draw a new one. The client application will perform the calculations meanwhile, making sure the changes are done in a correct way and that the attributes are updated during the process.

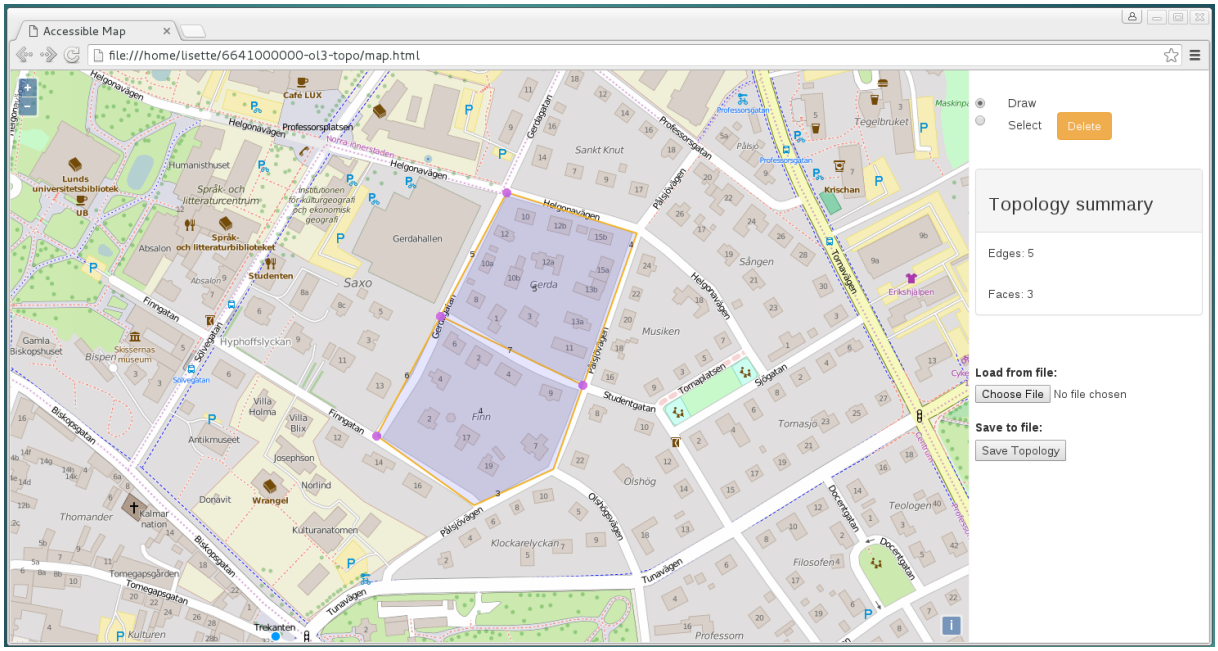

Figure 7.12 Topology consisting of five edges and two faces (excluding the universal face).

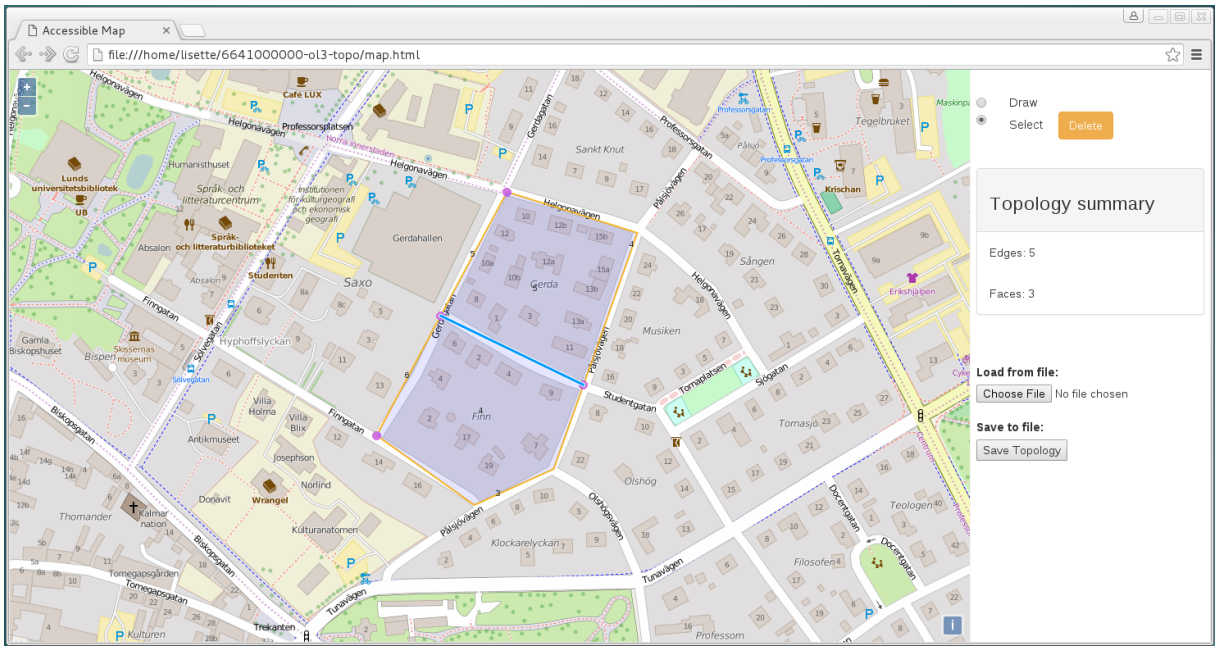

Figure 7.13. One edge (marked blue) is selected in the Topology from figure 7.12.

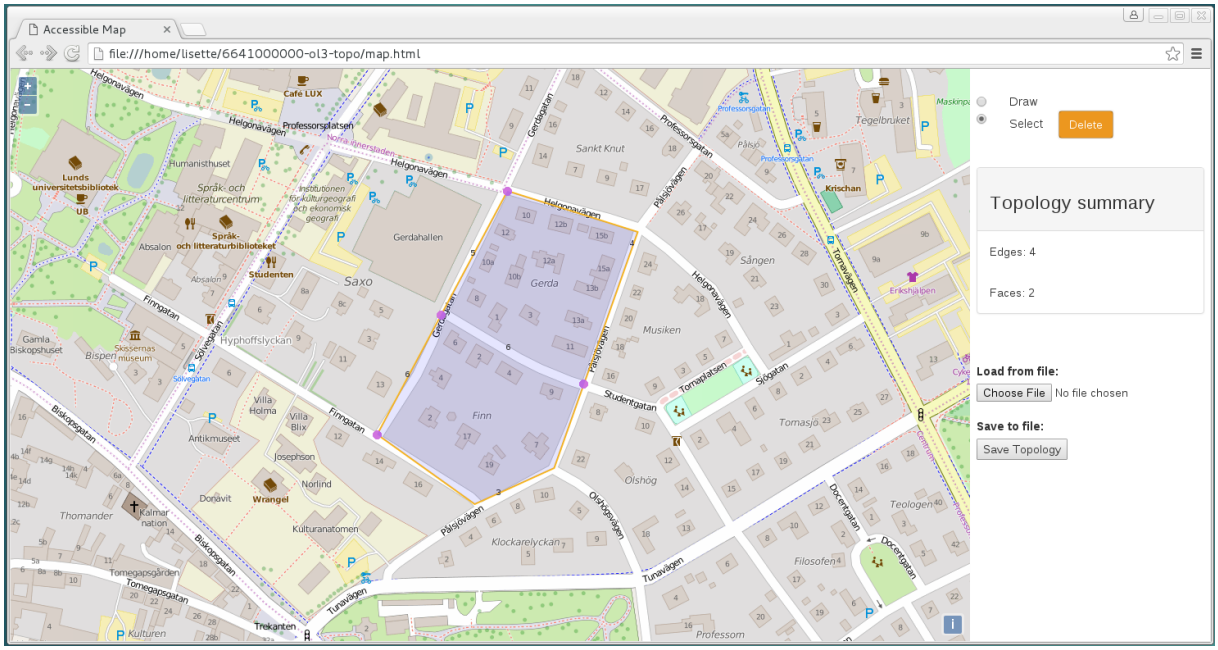

Figure 7.14. The selected edge from figure 7.13 removed.

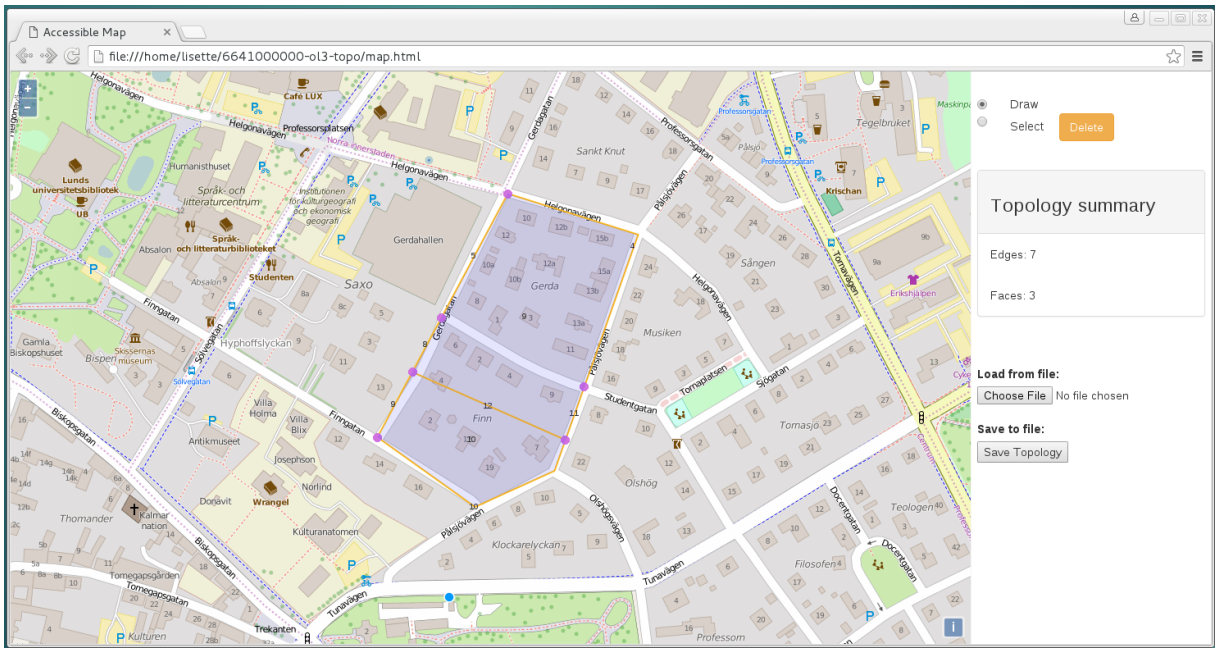

Figure 7.15. A new edge added to the topology in figure 7.14

7.7 Evaluation

The evaluation reconnects to the requirements set up in section 6.1. Below the symbols ☑ is used when a requirement is met, and the symbol ☒ when it is not.

User requirements

1 Edges

1.1 The user should be able to create isolated edges ☑

1.2 The user should be able to create edges that connects to one or several other edges by

- connecting its own start and/or end point to the other edge(s) start and/or end point(s). ☒
- connecting its own start and/or end point to an arbitrary point along the other edge(s) geometry. ☒

1.3 The user should be able to remove an isolated edge. ☑

1.4 The user should be able to remove an edge connected to other edges

- connecting its own start and/or end point to the other edge(s) start and/or end point(s). ☑
- connecting its own start and/or end point to an arbitrary point along the other edge(s) geometry. ☑

1.5 The user should not be able to create an edge intersecting itself (non-simple geometry). ☑

1.6 The user should not be able to create an edge intersecting another edges geometry, if it is not with its own start and/or end point (see functional requirement 3.3). ☒

2 Faces

2.1 The user should be able to create a face by enclosing a surface when creating an edge. ☒

3 Interface and drawing tools

3.1 The user should be able to create an edge by drawing a line string. ☑
3.2 The user should be able to remove an edge by selecting it and confirming the removal. ☑

Functional requirements

1 Nodes

1.1 A node should be created at the start point and at the end point of all new edges. However, if there is already a node in the start point or the end point, a new node should not be created there. ☒

1.2 The start node and end node of an edge should be removed when the edge is removed. However, if another edges is connected to the node, the node should not be removed. ☒

2 Edges

2.1 When creating a valid edge, the edge should be added to the topology and the attributes of the edge should be updated accordingly to existing topology. ☑

2.2 An edge should be split if another edge connects to the edge with its start and/or end point in an arbitrary point that is not the edges own start and/or end point. ☑

2.3 When creating an edge intersecting itself or another edge, the edge should not be added to the topology. ☑

3 Faces

3.1 A face should be created and added to the topology when a surface is enclosed. ☒

3.2 A face should be removed when one of the edges enclosing the surface of the face is removed. ☑

4 Interface and Drawing tools

4.1 The client application should provide an intuitive interface where topological data can be manipulated. Most desirable with a background map. ☑

5 Data Management

5.1 The client application should be able to store the data on local files. ☑

5.2 The client application should be able to load from a file with existing data. ☑

User requirement 1.2, 1.6 and 2.1, as well as functional requirement 2.2 and 3.1 are all connected to the issue of not being able to connect the edge that is being drawn to itself. When connecting the edge that is being drawn to existing edges, all the mentioned requirements are fulfilled.

Functional requirement 1.1 and 1.2 are not met due to the change of thought regarding nodes, and the fact that they are not implemented.

8 Discussion

The main aim of this project is to developing a client application where topological data can be loaded, created, deleted and rendered without any connection to a database. This has been achieved and the aim is considered met. The client application is not complete, but represents a valuable initial step towards developing a complete topological client application, or a general topological library.

The greatest challenge during this implementation was figuring out how to perform the calculations to maintain the topological relationships. The methods used for the PostGIS database were difficult to interpret. PostGIS is mostly implemented in C. However extensions and updates are partly implemented in SQL, resulting in the software consisting of a combination of the languages. There was short of knowledge regarding both these languages, and due to time limitations there was no option to study them thoroughly. SQL is however fairly readable, leading to some help and hint during the process. In addition, linear algebra and geometrical algorithms have been used as a tool when producing and performing the calculations.

When looking at the code after finishing the implementation it is recognized how the pure topological calculations can be separated from the functionality required for this solution in particular. The files containing the logic of the client application is stated to be a topological library. Still, the functions are fairly custom for this implementation. Making a more general JavaScript library is considered valuable. The functions naturally need access to the topology, which could be sent through the functions as an argument (parts of it or as a whole). If the functions were structured this way, it would simplify building a client application where multiple instances of topologies could be worked on simultaneously. Moreover, a general library could possibly be used when developing applications for other devices, depending on the storage model and the developing languages used. The lack of knowledge regarding implementing in a more general and independent way, and with the main focus on identifying ways to keep true to the topological rules, the library is not as general as it could be.

Developing a similar client application for other devices such as mobile phone and tablets could expand the usability. As mentioned in section 7.1, a significant question is then whether to make it a hybrid application or a native application. Creating a native application is a potential solution, but would involve developing a customary interface. Plug-ins for apps is an alternative to this; a well-established GIS app is then used as a platform, already supplied with a functional interface. A hybrid application is thought to be the most straight forward solution, especially if using the code implemented in this project. The files would in that case be compiled, loaded into an app and run in a local web environment within the app. Using either option, it is crucial to find a solution where the data are stored with accessible attributes. The same logical operations can in that case easily be applied. For the

data to be loaded from or to a database the storage model needs to support being converted to and from a serialization format.

Essential guidelines during the implementation phase was the functionality considered as standard in the topologically structured databases. The client application has not reached a state where it offers complete functionality. The issues related to not being able to connect to the edge that is currently drawn, is considered applicable if using other tools when selecting the coordinates. If developing a general library, the logic and the graphics would not rely on each other the way they do in this implementation, hence the problems could be solved. There is also functionality ruled out due to the decision to not include nodes. There are functions such as *remove node* and *heal edge when removing node* in the database, which naturally are not implemented. Whether the general library should support nodes or not is hard to state. Preferably, they should be include in such way where the developer personally can decide whether to implement them or not.

The projection used is EPSG:3857, representing the coordinates as x and y values *(x, y)*. As mention in section 7.4.7, the angles between the edges are calculated using the dot product and the cross product. These formulas are applicable for Cartesian coordinate systems *(x, y, z)*, meaning that the calculation are not guaranteed to cover the case of a geographic coordinate system *(longitude, latitude, altitude)*. Further investigations are required if this is of interest for the developer.

Section 5.1 presents the topological data structure ISO/IEC 13249-3:2016, implemented by Oracle spatial. When going over the attributes of the edges, PREV_LEFT_EDGE_ID and PREV_RIGHT_EDGE_ID are declared. In PostGIS these attributes are missing, additionally having the attributes abs_next_left_edge and abs_next_right_edge. For this project, the latter option is replicated. As far as this implementation goes, the attributes PREV_LEFT_EDGE_ID and PREV_RIGHT_EDGE_ID are only needed once, they are then calculated at that time. The abs_next_left_edge and abs_next_right_edge attributes are the absolute value of the next_left_edge and next_right_edge attributes. When using JavaScript, the function *Math.abs()* is available giving the same result for integers. Using this function rather than having an extra attribute is therefore an option. However, it could be discovered that these attributes are more important if going further with the implementation and adding more functionality.

The interface is simple and intuitive, all to make the process of managing topological data as easy as possible for the user. There are two modes functionality wise - creating edges and selecting + deleting edges, and there is only two states to work in. The default state is to create edges, illustrated by a filled radio button labeled **Draw**. To swap state, the (only) other radio button, labeled **Select**, is pressed. There is really no room for misinterpretation when working in the interface.

There are two ways of changing the topology, one where the user adds to it and one where the user subtracts from it. To ease the process of changing the topology, an idea is to let the

user move the edges as well. Since all the edges relate to each other, the logical operations would presumably remain the same; the moved edge is deleted and a new edge with the new geometry is created. The user would however be under the impression that the edge is being moved. This would result in the edge getting a new id every time it is moved. The mapping library Leaflet has an extension to its drawing tool with a similar functionality (Leaflet.draw.topology). It operates on a layer of topological data and allows the user to snap to an arbitrary point in a topology and move that point while all the connecting data follows. It could be of interest to study the technique behind this, and integrate the concept with the developed client application. The topological layer can be of the type GeoJSON, (and LayerGroup and FeatureGroup) so a possible integration might not be a big step away. Since the user cannot create a topology using Leaflet, the functionality is handled server side towards a database, which would not be the case if integrated with the client application.

Loading a subset of the topology from the database into the client application is at this state possible, but only for edges. They are added using their geometry, ignoring the rest of the attributes. This is an easy way of ensuring that the topology loaded to the client application is correct. However, it obstructs the possibility of loading the subset of data back into the database. The initial step would be to let the topological subset keep its attribute data, additionally loading the face data of the subset to the client application. The question is then whether to include the node data or not. To ensure that all the relations are correct when loading the subset back to the database, the nodes need to retain their id:s. This could be done in several ways, for instance by introducing nodes as a constraint for creating edges or by having the client application supporting node data but managing their existent automatically. Still, there is the issue of not interfering with the data in the database. Adding edges intersecting the geometry of the data in the database must be unpermitted, as well as using unavailable id numbers. Since the calculations are based on the geometries, one thought is to set an outer boundary within which the subset can be manipulated. When loaded back, only the elements connecting to the boundary need to be updated, done in either the client application or in the database direct. Another idea is to not set any limitations when editing the topology in the client application. Instead all the intersecting elements needs to be removed from either the client application or the database when loaded back. This could possibly cause correct data to disappear and result in a lot of adjustments afterwards.

As described, loading a subset of topological data back into a database is a complicated procedure. Solving it would complete the process shown in figure 1.1, which would be much valuable. When distributing large volumes of topological data, the possibility of working with subsets could improve the efficiency and standard of the work. Furthermore, it is common that two or more users need to work with the same dataset at once. By letting them work with subsets of the data, non-interfering work can be guaranteed, and the data can be merge back together in a controlled way afterwards.

9 Conclusions

Implementing the data in a topologically structured way can be a huge advantage when performing certain operations. Even so, the concept is many times not used when favorably needed. The complicity of the structure makes is challenging to implementing and integrate with other structures, which is thought to be one of the reasons for this.

The aim of this project is to develop a GIS client application where topologically structured data can be created and manipulated. This was achieved, resulting in a web based application written in JavaScript and HTML. When looking at the result, it can be stated that the storage structure of the data can be kept simple. The main information lies within the attributes and coordinates of the edges, as long as they are accessible the calculations can be performed. Additional information might be necessary to complete the graphical functionality.

The challenging part was to figure out what rules to follow and how, in order to keep true to the topological relationships. Depending on what user experience the developer wishes to provide, the functions required can differ between similar client applications. The general calculations are however the same.

Having access to a library providing topologically oriented functions would be of great value. If the complicated parts of the implementation were pre-defined and easy accessible, the developer could have limited knowledge about topologies and yet create similar applications as the one developed in this project. This could possible lead to a more open attitude towards topological data structures and encourage developers to create topologically structured web and native application, both for computers, tables and mobile phones.

Making this a web based application was successful. Established libraries were used as a tool in regards to the background map, intersection calculations and the graphical representation of the edges and faces among others. Nothing indicates that making a native application would bring issues not detected when developing the web based application. Depending on the languages used the selection of help tools might differ though.

References

Butler H, Daly M, Doyle A, Gillies S, Schaub T, Schmidt C, 2008. GeoJSON: The GeoJSON Format Specification
http://geojson.org/geojson-spec.html [2016-05-12]

Chaffer and Swedberg 2013. *Learning jQuery*. 4 Vol. Birmingham: Packt Publishing

Crockford 2008. *JavaScript – The Good parts*. Californian: O'Reilly

Theobald D. M. 2001. Esri: ArcUser Magazine - Understanding Topology and Shapefiles
http://www.esri.com/news/arcuser/0401/topo.html [2016-02-18]

Donna J. Peuquet, Zhan Ci-Xiang. 1987. An algorithm to determine the directional relationship between arbitrary-shaped polygons in the plane. *Pattern Recognition*, 20: 65-74.

Esri 2016. Esri: Understanding spatial relationships.
http://edndoc.esri.com/arcsde/9.1/general_topics/understand_spatial_relations.htm [2016-01-29]

Flanagan 2001. *JavaScript - The Definitive Guide.* 4 Vol. Californian: O'Reilly

Fulton and Fulton 2011. *HTML5 Cancas*. 1 Vol. Californian: O'Reilly

GeoTools 2015. GeoTools: Point Set Theory and the DE-9IM Matrix.
Http://docs.geotools.org/latest/userguide/library/jts/dim9.html [2016-01-29]

Harold and Means 2004. *XML IN A NUTSHELL – A Desktop Quick Reference.* Californian: O'Reilly

Harrie L. 2014. *Lecture Notes in GIS Algorithms*. GIS Centre and Department of Earth and Ecosystem Sciences. Lund University.

Harrtell B. 2016. GitHub: JSTS
https://github.com/bjornharrtell/jsts [2016-05-20]

Herring, John R. 2011. OGC: Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. OGC 06-103r4 Version: 1.2.1

JSON 2016. Introducing GeoJSON.
http://www.json.org/ [2016-05-16]

JTS 2016. Tsusiat Software: JTS Topology Suite
http://tsusiatsoftware.net/jts/main.html [2016-05-20]

jQuery 2016. jQuery: What is jQuery?
https://jquery.com/ [2016-05-17]

Lake, Burggraf, Trninic´ and Rae 2004. *Geographical Mark-up Language - Foundation for the Geo-Web*. England: John Wiley & Sons, Ltd

Landon Blake, 2007, Spatial Relationships In GIS - An Introduction, *OSGeo Journal*, Volyme 1

MathCamp 2016. Rio Rancho Math Camp: What is Topology?
http://rioranchomathcamp.com/topology.asp

MathIsFun 2016a. Math is fun: Dot Product.
https://www.mathsisfun.com/algebra/vectors-dot-product.html [2016-05-26]

MathIsFun 2016b. Math is fun: Cross Product.
https://www.mathsisfun.com/algebra/vectors-cross-product.html [2016-05-26]

Max J. Egenhofer, Jayant Sharma, David Mark, 1993. A Critical Comparison of the 4-Intersection Models for Spatial Relations: Formal Analysis, R. McMaster and M. Armstrong (eds), Autocartio 11.

Max J. Egenhofer, John R. Herring. 1990. A Mathematical Framework for the Definition of Topological Relationships. *Fourth International Symposium on Spatial Data Handling*: 803-813, Zurich, Switzerland.

Max J. Egenhofer, Robert D. Franzosa. 1991. Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5:2: 161-174.

Musciano and Kennedy 2002. *HTML & XHTML*. 5 Vol. Californian: O'Reilly

OpenLayers 2016. OpenLayers 3.
http://openlayers.org/ [2016-05-20]

OpenStreetMap 2016. OpenStreetMap: Copyright and License.
https://www.openstreetmap.org/copyright [2016-06-02]

Oracle 2016. Oracle Help Center: Spatial Topology and Network Data Models
https://docs.oracle.com/cd/B19306_01/appdev.102/b14256/sdo_topo_concepts.htm

Portele, Clemens. 2012. OGC: Geography Markup Language (GML) — Extended schemas and encoding rules. OGC 10-129r1 Version: 3.3.0

PostGIS 2016. PostGIS: About PostGIS.
http://postgis.net/ [2016-02-26]

Santiago 2015. *The book of OpenLayers 3 – Theory & Practice.* Victoria, Canada: Lean Publishing

Shekhar and Chawla 2003. *Spatial Databases - A Tour.* New Jersey: Prentice Hall

TopoJSON 2013. Bostock M, Metcalf C, 2013. GitHub: TopoJSON - The TopoJSON Format Specification.
https://github.com/mbostock/topojson-specification/blob/master/README.md [2016-03-25]

TopoJSON 2015. Michael N, 2015. GitHub: TopoJSON – Home.
https://github.com/mbostock/topojson/wiki [2015-03-25]

Worboys and Duckham 2004. *GIS – A Computing Perspective.* 2 Vol. America: CRC Press

W3Schools 2016. W3Schools: CSS Introduction.
http://www.w3schools.com/css/css_intro.asp [2016-05-18]

W3Schools 2016. W3Schools: JSON Tutorial.
http://www.w3schools.com/json/ [2016-05-16]

Appendix A

Additional notes on figure 6.1.4 (copied from Oracle 2016)

- *E* elements (E1, E2, and so on) are edges, *F* elements (F0, F1, and so on) are faces, and *N* elements (N1, N2, and so on) are nodes.
- **F0** (face zero) is created for every topology. It is the universe face containing everything else in the topology. There is no geometry associated with the universe face. F0 has the face ID value of -1 (negative 1).
- There is a node created for every point geometry and for every start and end node of an edge. For example, face F1 has only an edge (a closed edge), E1, that has the same node as the start and end nodes (N1). F1 also has edge E2, with start node N21 and end node N22.
- An **isolated node** (also called an **island node**) is a node that is isolated in a face. For example, node N4 is an isolated node in face F2.
- An **isolated edge** (also called an **island edge**) is an edge that is isolated in a face. For example, edge E25 is an isolated edge in face F1.
- A **loop edge** is an edge that has the same node as its start node and end node. For example, edge E1 is a loop edge starting and ending at node N1.
- An edge cannot have an isolated (island) node on it. The edge can be broken up into two edges by adding a node on the edge. For example, if there was originally a single edge between nodes N16 and N18, adding node N17 resulted in two edges: E6 and E7.
- Information about the topological relationships is stored in special edge, face, and node information tables. For example, the edge information table contains the following information about edges E9 and E10. (Note the direction of the arrowheads for each edge.) The next and previous edges are based on the left and right faces of the edge.
  For edge E9, the start node is N15 and the end node is N14, the next left edge is E19 and the previous left edge is -E21, the next right edge is -E22 and the previous right edge is E20, the left face is F3 and the right face is F6.
  For edge E10, the start node is N13 and the end node is N14, the next left edge is -E20 and the previous left edge is E18, the next right edge is E17 and the previous right edge is -E19, the left face is F7 and the right face is F4.

Institutionen av naturgeografi och ekosystemvetenskap, Lunds Universitet.

Student examensarbete (Seminarieuppsatser) i geografisk informationsteknik. Uppsatserna finns tillgängliga på institutionens geobibliotek, Sölvegatan 12, 223 62 LUND. Serien startade 2010. Hela listan och själva uppsatserna är även tillgängliga på LUP student papers och via Geobiblioteket (www.geobib.lu.se)

Serie examensarbete i geografisk informationsteknik

1 *Patrik Carlsson och Ulrik Nilsson* (2010) Tredimensionella GIS vid fastighetsförvaltning.
2 *Karin Ekman och Anna Felleson* (2010) Att välja grundläggande karttjänst – Utveckling av jämförelsemodell och testverktyg för utvärdering
3 *Jakob Mattsson* (2011) Synkronisering av vägdatabaser med KML och GeoRSS - En fallstudie i Trafikverkets verksamhet
4 *Patrik Andersson and Anders Jürisoo* (2011) Effective use of open source GIS in rural planning in South Africa
5 *Nariman Emamian och Martin Fredriksson* (2012)Visualiserig av bygglovsärenden med hjälp av Open Source-verktyg - En undersökning kring hur man kan effektivisera ärendehantering med hjälp av en webbapplikation
6 *Gustav Ekstedt and Torkel Endoff* (2012) Design and Development of a Mobile GIS Application for Municipal Field Work
7 *Karl Söderberg* (2012) Smartphones and 3D Augmented Reality for disaster management - A study of smartphones ability to visualise 3D objects in augmented reality to aid emergency workers in disaster management
8 *Viktoria Strömberg* (2012) Volymberäkning i samhällsbyggnadsprojekt
9 *Daniel Persson* (2013) Lagring och webbaserad visualisering av 3D-stadsmodeller - En pilotstudie i Kristianstad kommun
10 *Danebjer Lisette och Nyberg Magdalena* (2013) Utbyte av geodata - studie av leveransstrukturer enligt Sveriges kommuner och landstings objekttypskatalog
11 *Alexander Quist* (2013) Undersökning och utveckling av ett mobilt GIS-system för kommunal verksamhet
12 *Nariman Emamian* (2014) Visning av geotekniska provborrningar i en webbmiljö
13 *Martin Fredriksson* (2014) Integrering av BIM och GIS med spatiala databaser – En prestandaanalys
14 *Niklas Krave* (2014) Utveckling av en visualiseringsapplikation för solinstrålningsdata
15 *Magdalena Nyb*erg (2015) Designing a generic user interface for distribution of open geodata: based on FME server technology
16 *Anna Larsson* (2015) Samredovisning av BIM- och GIS-data
17 *Anton Lundkvist* (2015) Development of a WEB GI System for Disaster Management
18 *Ellen Walleij* (2015) mapping in Agricultural Development – Introducing GIS at a smallholders farmers' cooperative in Malawi
19 *Frida Christiansson* (2016) Lagring av 3D - geodata - en fallstudie i Malmö Stad