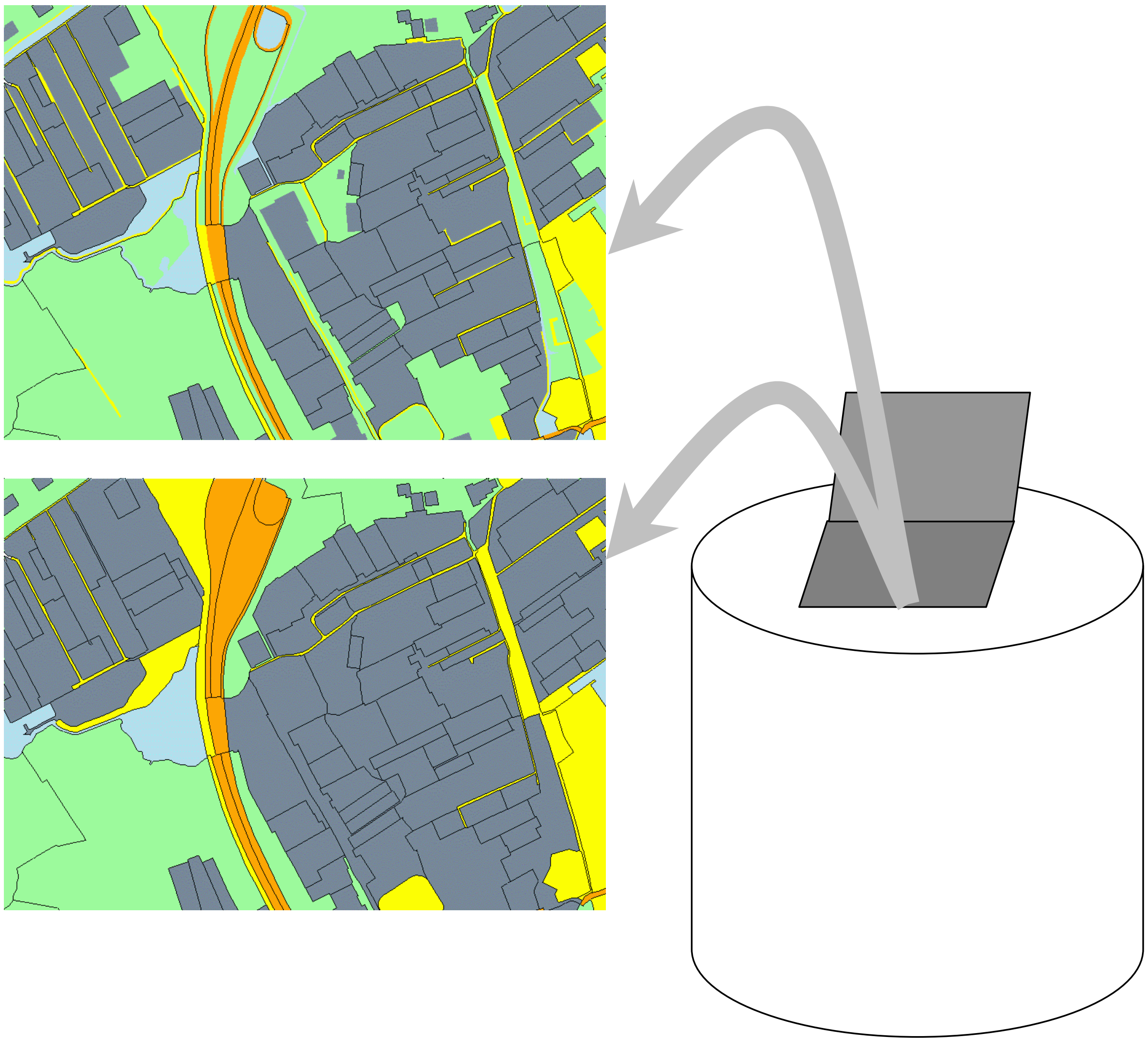
Development of a Topological Data Structure for On-the-Fly Map Generalization



Master of Science Thesis
M.J. Vermeij

Delft, June 2003

Section GIS-technology
Department of Geodesy
Faculty of Civil Engineering and Geosciences
Delft University of Technology

Development of a Topological Data Structure for On-the-Fly Map Generalization

Professor	Prof.dr.ir.P.J.M.van Oosterom
Supervisors	Drs. C.W. Quak
	Drs. T.P.M. Tijssen

Master of Science Thesis
M.J. Vermeij

Section GIS-technology
Department of Geodesy
Faculty of Civil Engineering and Geosciences
Delft University of Technology

Delft, June 2003

Preface

The thesis is the final and most tangible result of my graduation assignment at the Delft University of Technology. The research associated with this thesis was performed at the section GIS-technology of the department of Geodesy of the Faculty of Civil Engineering and Geosciences of the Delft University of Technology.

Of course I would like to thank all persons who in some way assisted in the research and the creation of this thesis. First of all the persons directly involved with my research, drs. W.Quak as my primary supervisor and prof.dr.ir. P. van Oosterom as my graduation professor and drs. T. Tijssen who also partly supervised the research. Furthermore I would like to thank all people from the section GIS-technology as well as other sections within the Department of Geodesy, and also fellow students, for useful input. Also I need to thank the Dutch Cadastre for providing the LKI data and the Dutch Topographic Service (TDN) for making the TOP10vector data available for the research. The TDN was kind enough to provide data for the entire area of the Netherlands. Unfortunately the developed data structure never matured enough to make use of more than a very small amount of the entire available data.

The subject of generalization and especially “on-the-fly” generalization is a very interesting field and still provides many challenges. I hope that the results presented in this thesis are useful in future developments.

Delft, june 2003
Maarten Vermeij

Summary

The field of automatic map generalization is still an active subject of scientific research. Use of automatic generalization can for example be found in web mapping applications. The problem with automatic generalization is not in the implementation of individual generalization operators, but more to apply them automatically in the right way. Even if fully automatic generalization yielding good results is achieved, it will require heavy computational effort to perform them. However interactive use of maps at various scales, such as in web mapping, requires quick responses. It is therefore necessary to, in some way, store the generalization process, such that maps at any scale can be retrieved easily and fast, without time consuming calculations. The performance of the generalization is especially important in a client-server architecture whereby the data is stored on a single server, and many clients can access that data simultaneously.

In thesis a data structure is presented that is intended to be used in a client-server architecture, with a DataBase Management System supporting spatial data as the server. Quick response times to client requests for generalized maps can be achieved by using specialised data structures that can store both geometric and thematic information, as well as information regarding the scales at which the data should be used in the creation of a map.
These data structures can be split into two groups multi-scale and scale-less data structures. Multi-scale data structures store complete maps at a number of different scales, e.g. maps at 1:10,000, 1:50;000 and 1:100,000. When a client requests a map at a certain scale, the server returns data from the closest scale stored, and simply stretches this to fit the map display. Since many features will be present at multiples scales, e.g. highways or large rivers, this type of data structure contains a large redundancy, as these features are stored at each scale. An advantage of such a data structure is however that every conceivable generalization operator can be utilised in creating the various fixed scale levels, including manual generalization. Thus each of the stored maps, can be cartographically very good.
A scale-less data structure does not store the data at any specific scale, but instead uses one integrated data set. Out of the data structure maps can be created at any scale, within reasonable limits. For each scale a different map face is created based upon the data present in the data structure.

Closely related to the concept of a scale-less data structure is a subject described of on-the-fly generalization. This term refers to a generalization process in which no generalized data sets are stores, instead every time a request is made for a map with a certain level of detail, a volatile map is created, which exactly satisfies the specifications of the request. One such data structure is the GAP-tree. GAP stands for Generalized Area Partitioning, which refers to the type of maps for which the data structure is intended, area-partitioning maps. The GAP-tree data structure stores a hierarchy of faces, whereby the face at the root of the tree is the most importance, and as the tree is traversed towards the leaves, less important faces are encountered. A map is created out of the GAP-tree by drawing all faces within the map face in decreasing order of importance, i.e. the most important face is drawn the first, and the least important face the last. By limiting the minimum importance, the total number of faces that is drawn is influenced, and thus the amount of detail of the final map. This GAP-tree data structure stores geometric objects, faces, which causes a redundancy when two faces share the same boundary. Redundancy in the boundaries of faces is unwanted because it requires more storage space and it also limits the possibilities of line generalization. If line generalization using e.g. the Douglas/Peucker algorithm is applied to both boundaries, different results can be created. In the map display this difference can be seen as double lines, or sliver polygons. To overcome the redundancy a topological data structure is needed.

In this thesis the development of such a topological data structure for on-the-fly generalization is described. The data structure consist of two tables, an edge table which stores the boundaries of all faces at the level of the input data, and a face table which stores thematic information regarding all faces but no geometric description of its shape. This face table contains a similar hierarchy as the GAP-tree, by storing a reference to a parent face for each face. Besides the geometric data in the edge table and the thematic information in the face table, both tables also store a 3 dimensional bounding box for each object. This bounding box has a 2D extent that equals the 2D bounding box of the edge, respectively the geometric representation of the face. The third dimension is used to store information regarding the level of detail (LoD) at which an object should be visible in a map. The value used for this LoD based selection is chosen such that a higher value corresponds to a lower LoD or smaller scale. By using a third geometric dimension to store this value, single SQL queries can be simultaneously selective for both geometric as well as LoD properties of the objects stored in the database. Especially when this is combined with the use of 3D indexes on both tables, efficient selection of the appropriate objects is realised. This is very important in the field of on-the-fly generalization.
However with the selection of the right objects the desired results of a generalized map is not yet obtained. For a map display explicit geometric descriptions are needed for all objects within the map face. It is therefore necessary to convert the topological description in the edges to a geometric description of the faces. This conversion requires, at least for the map area, that for all faces a complete boundary is present. For this to be realised using only edges from the edge table, a large number of edges outside of the map window is needed. This is not wanted as it would require for to much geographic data to be transferred from the server to the client. Another way to obtain a geometric description of all faces, is to include the query window in the list of edges used to create the geometric description of the faces, and to intersect the edges inside of the map window with the boundary of that window. With this new set of edges an area partitioning covering the entire map display can be created. However for proper display, e.g. using the right colors, each of the polygons in the area partitioning needs to be linked to the appropriate record in the face table. For this purpose each edge has a reference to the faces that lie left and right of it at the highest LoD. To obtain the reference to the faces present in a map display with a lower LoD, the face hierarchy has to be traversed. To be able to do this, the face hierarchy relevant to the map window and LoD is transferred to the client.

Using this topological data structure it is possible to create generalized maps on-the-fly. The system of course has both advantage and disadvantages. The first advantage is that it is a topological data structure, which reduces redundancy in the storage of geometric properties and allows the application of line generalization. Secondly the data structure can be efficiently queried using readily available 3D indexing method and does not need a custom build indexing type. Thirdly only geometric data that is present within the map window is needed. Finally the data structure allows the distribution of required computing power between the client and the server. By utilising client-side computing power the same server computer can server more clients simultaneously than if all processing was done at the server-side. The data structure also has some drawbacks.
The main problem is the fact that even in zoomed out maps, the original edges are still used. These edges are both to detailed and too large in number in zoomed out maps. The detail per edge can be reduced by applying line generalization. The number of edges however cannot be reduced so easily. In order to reduce this number a tree structure over the edges has to be constructed that enables the creation of large, complex edges out of multiple input edges, without the need for much computation. This additional data structure should also assure that the returned edges already have the right face references, so these do not have to be updated afterwards. This data structure was not concretised within the research. The use of topology does not only bring advantages, it also introduces the need for conversion routines from topology to geometry. This conversion has to be performed for each map display and can be quite computationally intensive.

Overall the data structure presented in this thesis is a step into the direction of the creation of a topological data structure for on-the-fly generalization. There is however still much work to do in this field; not only in the enhancement of the developed data structure, but also in the possibilities of the generalization algorithms used in on-the-fly generalization. The ultimate goal in this would be to be able to create efficiently and fast, on-the-fly generalized maps that are of good cartographic quality.

Samenvatting

Het terrain van automatische kaartgeneralisatie is nog steeds het onderwerp van wetenschappelijk onderzoek. Automatische generalisatie wordt bijvoorbeeld gebruikt in web mapping toepassingen. Het moeilijke bij automatische generalisatie is niet zo zeer het toepassen van afzonderlijke generalisatie operatoren, maar meer het automatisch correct toepassen van de hele set aan operatoren. Zelfs als volledig automatische generalisatie mogelijk is, die een goed resultaat opleverd, zal dit toch een zeer rekenintensieve bewerking zijn. Voor interactief gebruik van kaarten van meerdere schalen, zoals bijvoorbeeld bij web mapping, is het noodzakelijk dat kaarten met verschillende schalen gemakkelijk en snel kunnen worden verkregen, zonder tijdrovende berekingen uit te hoeven voeren. De snelheid van de generalisatie is met name van belang in client-server omgevingen waar de gegevens worden opgeslagen in een server en meerdere clients tegelijkertijd deze data bevragen.

In deze afstudeerscriptie wordt een datastructuur gepresenteerd die bedoeld is voor gebruik in zo'n client-server omgeving waarbij de server een DataBase Managment System is dat ruimtelijke data ondersteund. Snelle reactietijden voor bevragingen door de client kunnen bereikt worden door gebruik te maken van speciale data structuren die niet alleen geometrische en thematische informatie opslaat, maar ook gegevens over de kaartschalen waarvoor de gegevens geschikt zijn.
Binnen dit soort datastructuren zijn twee soorten te onderscheiden, multi-schaal en schaalloze datastructuren. Een multi-schaal structuur slaat volledige kaarten op voor meerdere schalen, bijvoorbeeld 1:10.000, 1:50.000 en 1:100.000. Als een client vraagt om een kaart met een bepaalde schaal geeft de server een kaart terug uit de dichtsbijzijnde opgeslagen schaal en rekt deze op om het kaartbeeld precies te vullen. Omdat veel objecten op meerdere schalen getoond moeten worden, bijvoorbeeld snelwegen of grote rivieren, worden deze in iedere kaartschaal opgeslagen. Dit zorgt echter voor een overtolligheid in de gegevens. Een voordeel van dit systeem is wel dat er geen beperkingen zijn aan de generalisatie methoden die worden toegepast. Alles is mogelijk, inclusief handmatige bewerkingen. Hierdoor kunnen deze kaarten van een zeer goede cartografische kwaliteit zijn.
Een schaalloze datastructuur slaat geen gegevens op met een bepaalde schaal. In plaats daarvan een geintegreerde structuur wordt gebruikt waarin alle gegevens worden bewaard. Uit deze structuur kunnen vervolgens kaarten worden gemaakt op iedere willekeurige schaal, binnen redelijk grenzen.

Nauw verwant met het concept van schaalloze datastructuren is het onderwerp van on-the-fly generalisatie. Met deze term wordt een generalisatie proces bedoeld waarbij geen gegeneraliseerde gegevens worden opgeslagen, maar waarbij iedere keer als een kaart wordt opgevraagd met een bepaalde schaal, een tijdelijke kaart wordt gemaakt, die precies voldoet aan de specificaties zoals opgegeven. Een zo een datastructuur is de GAP-tree of Generalized Area Partitioning tree. Deze structuur is bedoeld voor gebruik met zogenaamde planaire partities. In de GAP-tree wordt een hierarchie van faces opgeslagen, waarbij de face in de wortel van de boom het meest belangrijke is, en de faces in de bladeren het minst belangrijk. Een kaart kan gemaakt worden uit deze GAP-tree door alle faces te tekenen die binnen het kaart blad vallen. Bij het tekenen moeted faces in aflopende volgorde van belangrijkheid worden getekend. De meest belangrijke face dus eerste en de minst belangriijke als laatste. Door nu een minimum belangrijkheid op te gegeven kan het aantal faces dat moet worden getekend worden beinvloed en daarmee het detail niveau van de kaart. In de GAP-tree worden volledige geometrische objecten opgeslagen. Dit zorgt voor een redundantie wanneer twee faces (gedeeltelijk) dezelfde grens hebben. Deze redundantie is ongewenst, omdat het meer opslagruimte vereiste en omdat het het gebruik van lijngeneralisatie, met bijvoorbeeld het Douglas/Peucker algoritme, minder goed mogelijk maakt. Als lijn generlizatie zou worden toegepast op beide grenzen, kunnen verschillende

resultaten onstaan die in de kaart zichtbaar zijn als dubbele lijnen of zogenaamde sliver polygonen. Om deze redundatie te voorkomen is een topologische datastructuur nodig.

In deze scriptie beschrijft de ontwikkeling van een topologische data structuur voor on-the-fly generalisatie. De datastructuur bestaat uit twee tabellen, een edge tabel die degrenzen van de objecten opslaat op het detail niveau van de brondata, en een face table die de thematische informatie over de faces opslaat, maar geen geometrisch beschrijving van de faces bevat. Deze face tabel bevat een hierarchische structuur lijkend op die van de GAP-tree. Voor ieder face wordt een verwijzing naar een ‘vader’ face opgeslagen. Naast de geometrische gegevens in de edge tabel en de thematische informatie in the face table slaan beide tabellen nog een 3 dimensionale bounding box op voor ieder object. Deze bounding box heeft als 2D begrenzing de 2D bounding box van een edge dan wel de geometrische representatie van een face. De derde dimensie wordt gebruikt om informatie op te slaan over de detailniveaus waarvoor de afzonderlijke objecten zijn bedoeld. De waarden voor deze detailniveau informatie zijn zo gekozen dat hoger waarde overeenkomen met belangrijkere objecten. Door de derde geometrische dimensie hiervoor te gebruiken, kan met een enkele SQL-query tegelijker geselecteerd worden op ruimtelijke en de detailniveau criteria betreffende de objecten in de database. Zeker als dit gecombineerd wordt met het gebruik van 3D indices op beide tabellen, is een snelle en efficiente selectie van de relevante objecten mogelijk.
Echter met de selectie van de juiste objecten is het beoogde resultaat van een genereliseerde kaart nog niet bereikt. Om de kaart te kunnen tekenen is een expliciete geometrische beschrijving nodig van de objecten in de kaart. Het is daarom nodig dat de topologische beschrijving in de edges wordt omgezet naar een geometrische beschrijving van de faces. Voor deze conversie is het nodig dat voor alle faces binnen de zichtbare kaart een complete begrenzing beschikbaar is in de vorm van edges. Als hiervoor alleen de edges zouden worden gebruikt die uit de edge tabel komen, zou een groot aantal edges van buiten het kaartbeeld nodig zijn. Om dit te voorkomen worden de edges binnen de begrenzing van de kaart gesneden met de rand van de kaart. Tevens wordt de rand van de kaart toegevoegd als ware het de begrenzing van een face. Met deze nieuwe set van begrenzingen wordt een planaire partitie gemaakt die precies het hele kaart beeld vult. Voor een goede visualisatie, met bijvoorbeeld gebruikmaking van de juiste kleuren, is het nodig dat aan de polygonen in planaire partitie de juiste records uit de face table worden gekoppeld. Voor dit doel heeft iedere edge verwijzingen naar het vlak dat links en rechts van hem ligt op met hoogst detailniveau. Om een verwijzing naar de face op het active schaalniveau van de kaart te krijgen moet de face hierarchie worden doorlopen. Hiervoor wordt de hierarchie voorzover van belang binnen de kaart, naar de client verstuurd.

Het is mogelijk om met behulp van de een topologische datastructuur on-the-fly generaliseerde kaarten te maken. Het system heeft natuurlijk voor en nadelen. Het belangrijkste voordeel is het gebruik van een topologische structuur, waardoor er minder redundantie is in de geometrische gegevens en lijngeneralisatie mogelijk is. Ten tweede kan de datastructuur efficient bevraagd worden door gebruik te maken van bestaande 3D indices en is geen special indexeringsmethode nodig. Ten derde zijn alleen geometrische gegevens nodig die binnen de te maken kaart vallen. Ten slotte maakt de data structuur het mogelijk om de benodigde rekenkracht te verdelen tussen de client en server. Hierdoor is het mogelijk om met dezelfde server, meerdere clients tegelijk te verzorgen dan wanneer alles aan de serverkant plaats zou vinden.
Het grootste probleem met de datastructuur is het feit dat zelfs in ver uitgezoomde kaarten nog steeds de oorspronkelijke kleine edges worden gebruikt. Deze edges zijn te gedetailleerd en te groot in aantal voor deze kaarten. Het detail per edge kan eenvoudig verminderd worden met lijngeneralisatie technieken. Het grote aantal edges is echter niet makkelijk te verlagen. Voor het verlagen van dat aantal is een boomstructuur nodig die over de edges heen een hierarchy opbouwt. Uit deze structuur moeten op een makkelijke en snelle manier samengestelde edges kunnen worden gehaald. Deze structuur moet er ook

voor zorgen dat automatisch de juiste verwijzingen naar de faces worden meegegeven zodat deze ook niet meer hoeven te worden bijgewerkt. De ideeen voor deze structuur zijn echter niet geconcretiseerd binnen het onderzoek. Het gebruik van topologie brengt niet alleen voordelen, het vereist ook een omzetting naar expliciete geometrische beschrijvingen voor bijna iedere toepassing en voor iedere kaart. Deze omzetting is nog vrij rekenintensief.

De datastructuur die in deze scriptie wordt gepresenteerd is een stap in de richting van een topologische datastructuur voor on-the-fly generalisatie. Op dit terrein is echter nog veel werk nodig; niet alleen voor de verbetering van de ontwikkelde datastructuur, maar ook om het mogelijk te maken complexere generalisatie methoden te gebruiken in combinatie met on-the-fly generalisatie. Het uiteindelijke doel hierbij is om snel en efficient, on-the-fly gegeneraliseerde kaarten te kunnen maken, van een goede kartografische kwaliteit.

Table of Contents

Preface .... **iii**
Summary .... **v**
Samenvatting .... **ix**
Table of Contents .... **xiii**
1. Introduction **1**
1.1. Subject 1
1.2. Motivation 1
1.3. Goal 1
1.4. Conditions 2
1.5. Structure of this thesis 2
2. Automatic Cartographic Generalization **3**
2.1. Cartographic Generalization 3
2.2. Automatic generalization 6
2.3. Geo-DBMS and Generalization 7
2.4. Scale-less vs. Multi-scale 8
2.5. On-the-Fly generalization 8
2.6. Conclusion 12
3. Binary Line Generalization Tree **13**
3.1. Douglas/Peucker Line generalization 13
3.2. BLG-Tree 13
3.3. Implementation 14
3.4. Performance 20
4. Development of a Topological Data Structure for On-the-Fly Generalization **23**
4.1. General information on all methods 23
4.2. Polygon reconstruction based on geometric relationships between edges 24
4.3. Distinct Edge-Face Coding 27
4.4. Storing bounding boxes 28
4.5. Combined tree and bounding box data structure 30
4.6. Issues regarding all procedures 31
4.7. Client-side routines 32
5. Implementation of the data structure **33**
5.1. Implementation of a generalization Method 33
5.2. Server-side routines 33
5.3. Client-side routines 40
5.4. Conclusion 45
6. Analysis of the data structure **47**
6.1. TOP10Vector 47
6.2. Pre-processing 48
6.3. Performance of the generalization 50
6.4. Addition of BLG-tree 52
6.5. Advantages and disadvantages of the data structure 54
6.6. Possible enhancements of the data structure 56
7. Conclusions and recommendations **59**
7.1. Conclusions 59
7.2. Recommendations 61
References **63**
Literature 63
Software 63
Websites 63

Appendix A: Server computer system specifications 65
Appendix B: Source code server-side routines 67
Appendix C : Source code client-side routines...73
Appendix D: Class diagram of objects from client-side routines...................................79

1. Introduction

This chapter lays the foundation of the rest of this thesis. It presents the subject of the research, the goal, and the motivation. Also an overview of the structure of the remainder of this thesis is presented. This introduction might contain terms unknown by some readers, but these are explained further on in this thesis.

1.1. Subject

The overall term applying to the research subject is *server-side generalization*. The client-server architecture which is implied by this description consists of a DBMS which can serve multiple clients simultaneously and a client that is able to understand the data structures it receives from the server. Generalization is concerned with the reduction of the amount of detail present in map, to assure that the map is clearly readable. It come mainly in to play when smaller scale maps are derived from larger scale maps. Without generalization the map would become cluttered with small objects and not be easy to use.
The focus in the research is on the development of a new, or adaptation of an existing, data structure which supports on-the-fly generalization. In on-the-fly generalization the basis is one integrated data structure which contains all necessary information to create volatile maps of any scale at the request of a user. Each user request is answered by returning a map that precisely satisfies the user's demands for both spatial extent, as well as the amount of detail, or scale of the map. A more elaborate explanation of the term on-the-fly generalization is given in chapter 2.

1.2. Motivation

The direct motivation for the main subject, the development of a data structure for on-the-fly generalization, was the observation that some redundancy is present in the storage structure of the GAP-tree and also a basic idea for a new data structure supporting on-the-fly generalization, which could potentially overcome the observed redundancy. The observation came during the early stages of the research when the subject of the study was still only described under the broad term 'server-side generalization'. The focus of the research shifted to the GAP-tree and implementation of the GAP-tree in the standard DBMS Oracle 9i but this was later on ended in favour of the development of the new data structure.
The subject of on-the-fly generalization in general is an interesting subject for research as it can be very useful in any application where geographic data needs to be presented at a range of scales or levels of detail. An example application where on-the-fly generalization can be useful is web mapping.

1.3. Goal

The ultimate goal of the research was formulated as: an efficient data structure supporting on-the-fly generalization, based upon the ideas that come up in the preliminary studies. This goal is formulated in the following question:

How can on-the-fly generalization be implemented in a client-server architecture in an efficient way?

In order to provide an answer to that question the following sub-questions will have to be answered.

- *Which data structures and algorithms can be used that support efficient on-the-fly computation of a generalized data set?*

- *How should clients formulate their request for a generalized data set and how is the data returned?*

- *Can the query-time workload be distributed over the server and the client?*

Besides answering these questions a working implementation will be made that shows (part of) the working of the developed data structure. Some aspects of the data structure were only worked out theoretically late into the research and were not implemented.

1.4. Conditions

During the research some conditions and restriction were effective which are explicitly stated here.
The DBMS used in the research was Oracle 9i spatial. The DBMS was located on a SUN Enterprise E3500 server with Sun Solaris 7 (for precise specifications see Appendix A.1). This computer was also used to test the implementation of the client-side routines. Since the client applications were written in Java it was also relatively easy to test it on other computer platforms. Extensions to the Oracle DBMS in the form of stored procedures and functions could be created using the PL/SQL language or Java. The test data used was provided by the Dutch Topographic Service (Topografische Dienst Nederland, TDN). They were generous enough to provide TOP10vector data covering the entire area of the Netherlands for this research. This data was delivered in an ArcInfo format and was transferred to an Oracle database. Also data from the Dutch Cadastre was used in the form of the LKI database, containing geographic descriptions of parcels.
A Master of Science research at the department of Geodesy is scheduled to have a nominal time span of 1000 hours.

1.5. Structure of this thesis

After this introductory chapter, chapter two discusses the subject of automatic cartographic generalization. Attention is paid to generalization methods and issues related to automating the generalization process. The subject of on-the-fly generalization is discussed in more detail.
Chapter three describes the implementation of the BLG-tree data structure within the Oracle DBMS. This data structure can be used as part of more elaborate data structures for on-the-fly generalization.
The actual development of the data structure for on-the-fly generalization is presented in chapter four. This chapter discusses not only the final data structure, but also a number of steps that lead to the final result. Chapter five is concerned with the implementation of the data structure. To test the data structure an implementation had to be made of routines for filling the data structure and routines for retrieving data from it, as well as client-side functionality to properly interpret returned data from the server and to visualise it. The final data structure and accompanying routines was subject to analysis on several aspects, and the results of this are presented in chapter six. Chapter seven presents conclusions that can be drawn based upon the entire research and will explicitly answer the questions posed in this chapter. Furthermore recommendations for future research are presented.

2. Automatic Cartographic Generalization

The purpose of this chapter is to provide the reader with some background knowledge in the field of automatic cartographic generalization. Special attention will be paid to the problems that are still open in this field. The research associated with this thesis focussed on the subject of “On-the-Fly” generalization. This subject has some distinct demands and properties, which are discussed in this chapter.

2.1. Cartographic Generalization

Cartographic generalization has been defined by the International Cartographic Association as: ”The selection and simplified representation of detail appropriate to the scale and / or purpose of a map.”[1] This definition immediately makes clear that the process of cartographic generalization is concerned with a multitude of issues. The important issue in cartographic generalization is the reduction of the amount of detail that is presented in a map display. This should be in unison with the purpose and the scale of the map. The process of generalization starts with a data set containing more data than is useful for presentation in a map display. This base information could for example be a large scale base map from the national mapping service which are often present at scales such as 1:10.000 for for example Belgium, The Netherlands and 1:24.000 and 1:25.000 in the United States and Switzerland [19][20][21][22]. These maps contain detailed information covering large areas and are therefore well suited to be used as a basis to construct less detailed, generalized maps, which represent larger areas on smaller map faces. The reduction in detail required when smaller scale maps are created out of larger scale maps is a result of the fact that maps are mostly intended as a means of communication and should therefore be clear to understand. This demands a balance between the amount of information and the space that is available to present it in. A rule can be formulated that states that ‘the total amount of information present in a map of a certain size should be the same, regardless of the scale the map’ otherwise known as the constant information density rule [14]. Smaller scale maps can therefore only present information that is less detailed than larger scale maps. What information remains after the generalization process depends largely on the purpose of the map. Figure 2.1 shows how a reduction of information is used to maintain clear maps.

Generalization is a process, which is intricately woven throughout the entire process of gathering geo information until the presentation of it using a map or other means of visualization. It already starts when specifications are set up based upon which the surveying is performed, or other measurement techniques are chosen. These specifications define which parts of reality are of interest and should be recorded. Thus a primary selection process is involved even before the data is gathered. Such a selection process is also part of explicit cartographic generalization as described below. The measurements themselves are also subject to generalization, especially when manual surveying is used as the technique. The surveyor has to decide which objects are to be measured and exactly how to measure them. An important subject here is the idealisation of objects. For example, whereas in a map the separation between a road and adjacent fields appear crisp and well defined, in reality this border is not so well defined. The surveyor has to decide where to measure this border, which is always open to some interpretation, regardless of the detail of the specifications set up prior to the actual surveying. In a sense the choice to represent the side of the road by a line segments connecting individual points, is also a form of generalization. More extreme cases of idealisation can be found at watersides. As the water level varies over time, so does the exact position of the waterside. Whereas the idealisation of road sides propably has a margin of some decimetres of at most, water sides could have fuzzy boundaries spreading several meters.

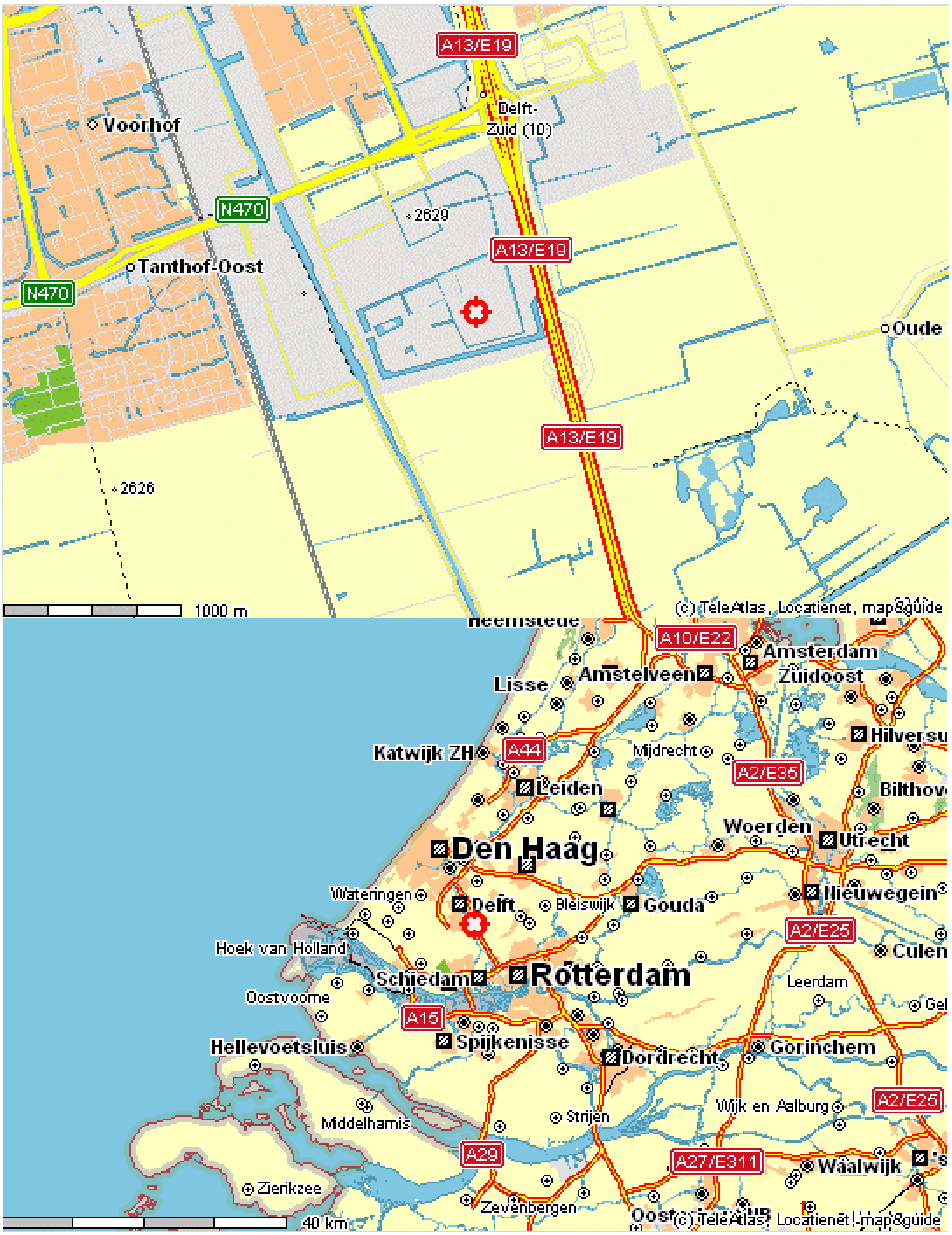


Figure 2.1: Example of need for generalization. Lower image would by totally clutterd if the detail level of the upper map was used.[24]

The subject of this thesis and it's associated research and of most other literature concerning cartographic generalization, is more focussed on the processing of geographic data after it has been collected and transformed to it's basic representation for example in the form of a topographic map. Nowadays this base information is usually available in some kind of digital form, e.g. in a file or in the tables of a spatial DBMS. For the remainder of this thesis it is assumed that all data is present in a known digital format and thus readily usable in various computer applications.

Generalization methods

Now that it is clear as to what is meant by cartographic generalization in the context of this thesis, some methods used in the generalization process will be presented. Generalization itself is performed by applying a number of functions on the input data set. Each of these functions in some way reduces the amount of detail and visual complexity of it's input. This will usually also result in a reduction in the amount of data to represent the output features, with respect to the input. The list presented here is based upon the list in [12]

Selection

The first step in the generalization process is the selection of which features should be present in the result. The criteria for this selection depend greatly on the purpose of the resulting data set. An often-used criterion is based upon the type or classification of objects. Only objects of certain types are selected, e.g. on a road map it would be less important to show individual buildings, so these need not be selected. Selection is probably the easiest way to reduce the total amount of information. Selection is also the first step in any generalization process since it establishes the set of objects, which is used as input for other generalization functions.

Simplification

The individual objects present after selection can be too detailed for immediate display. This amount of detail can be reduced through simplifications. To achieve this several operations are possible, such as the reduction of the number of points on a line, smoothing of a line etc.[12]

Exaggeration

Exaggeration is an operation that increases the visual importance of objects, often by enlarging their representation on a map display. Exaggeration is an important function in visualization of a generalized map.

(Re)Classification

Classifcation is the grouping of objects with similar properties into classes. Thus the number of different types of objects is reduced. If the number of classes it self needs to be reduced in the generalization process, the term re-classification can be used.

Symbolization

The representation of an object by an object of lower geometric dimensionality (collapse), or a simple representing object (symbol)

Aggregation

Aggregation is the union of adjacently or closely located objects into one larger object. Objects that are aggregated usually are of the same or similar themes.

Typification

Typification is the representation of a large number of discrete objects with similar shapes by a small number of objects with the same typified shape. The distribution pattern of the original objects must be preserved in the distribution of the new objects.

Anamorphose

Anamorphose is concerned with the solving of geometric and topologic conflicts between objects that are the result of the application of generalization procedures as described above. Mostly these conflicts are resolved by displacement of objects that are small, relative to the map scale, to satisfy spatial constraints, or by changing the shapes of larger objects, such as roads and coast lines.

2.2. Automatic generalization

Implementation within a computer program of each of the operators individually is very well possible. It is however a very complicated task to apply them all in conjunction to create a fully automatic generalization process that yields results which are as good as human intervened automatic generalization or computer assisted manual generalization, depending on the point of view. For fully automatic generalization to create good results the following requirements have to be met by the generalization method [3]:

- Contextual analysis – you can't generalise one map feature at a time in isolation. You have to consider groups of objects as a whole
- Adaptive processing – you can't apply a single algorithm to all features (even of a single class). You have to choose appropriate algorithms according to the circumstances of that feature
- Backtracking – you can't get it right first time every time. You have to be prepared to assess whether an operation has made things better or not, and be prepared to undo it and try something else.

Depending on the purpose and user of a map, maps of lesser cartographic quality can however be useful and some form of fully automatic generalization can be used. Whatever the purpose may be, research on the topic of automatic cartographic generalization is still needed with the final goal to have a fully automatic procedure that can create generalized maps as good as those created by manual generalization.

The importance of automatic cartographic generalization is closely related to the increased use of maps, especially in a digital form. The use of maps varies from the obvious route planning and navigation, to political and economic decision making. More and more the maps used have to be based on data which is in some way dynamic or at least subject to frequent updates. Furthermore in the analysis of geospatially related information, new map views are created over and over again to present exactly the information needed. The display of these ad hoc maps would benefit from automatic cartographic generalization, since manual generalization is a costly and time-consuming task, which would be impracticable to apply after every change of a map.

Another way in which automatic generalization is an important issue is in the field of web mapping, or the presentation of maps via the internet, as for example shown in figure 2.1 which was made using the locatienet website[24]. The advent of the internet made it possible to access data from all over the world via the computer. This includes the geographic data, often in the form of map displays. However the connection to the internet is often via a relatively small bandwidth which can cause long waiting time for detailed maps. A solution to this would be to reduce the amount of data through a generalization process which makes it suitable for transfer over the internet, yet still provide the cartographic detail that can be expected from the desired scale and map size. If this need for fast responses is combined with the desire for interactive maps, that allow the user to pan and zoom on the map and possibly also to change the selection of the objects or themes that should be displayed, then not only the need for automatic generalization becomes apparent, but also the need for quick response times. This requires a good performance of the combination of a data structure, generalization procedures and also the server computer.

Some comment is needed here on the term *scale* in computer or on-screen cartography, whereby maps are visualised on a computer monitor. On plain paper maps the scale is defined as the relative size of map-units to real world units. This is possible since the size of the map does not change after production. In on-screen cartography there is no strictly defined relation between the original units and the display units, due to the fact that it is relatively simple to zoom in and out, and thus alter the effective scale, and also due to the fact that the exact metric size of it's map display on the monitor, is not always known to the mapping software. It is always possible to present a scale value using for example m/pixel as the unit. But this value would not by meaningful to many users. A substitute for the scale indicator of a map can be the Level of Detail (LoD) of a map. There is no well-defined unit for this LoD but a practical choice is to refer to the scale of the paper map that presents the same amount of information per real-world unit.

2.3. Geo-DBMS and Generalization

Geo-DBMS

Historically much spatial data was stored using special file formats. This was necessary as spatial data types were not yet supported, and also because of the spatial data can be quite voluminous in terms of file size. With the advent of faster computer systems, which could easier handle large data volumes it became more and more feasible to use standard DBMS's for the storage of spatial data. Although these systems require more disk space to store the same data than the specialised file format, they offer much greater flexibility in the use, maintenance and protection of the data. For use with spatial data several DBMS vendors nowadays have special versions or add-on's available, that extend the standard DBMS to support spatial data types, and to provide functionality to use these data types. Another important aspect is the use of specialised spatial indexes such as the quad tree or the R-tree for quick access of records using spatial conditions [13]. A DBMS that explicitly supports the use of spatial data types is called a spatial DBMS, or in the context of geo-information a Geo-DBMS.

An important feature of these DBMS's is the ability to allow multiple users to access the same database simultaneously. The server application in this context is the DBMS that maintains the data. The client application used by the end-user accesses the server to perform queries to obtain the necessary data for the creation of a map display. In most cases the client and the server application will not reside on the same computer, but will be connected via a network. This connection can potentially be slow relative to the amount of data that has to be transferred in response to a client request. This is where the concept of server-side generalization becomes of importance. Most of the generalization functions that were described, reduce the amount of data that needs to be transferred. From this point of view it would be beneficiary to the response time of a server to a client, if generalization of the data transferred to the client were possible. However, since not all queries require the same level of detail some provision has to be made at the server-side that allows the creation of generalized maps with the LoD desired by the user.

Server-side generalization

To reduce the amount of data that has to be transferred from the server to the client some kind of method has to be employed that does reduce the amount of data but also retains the important information. This problem largely corresponds to the problem of cartographic generalization that has been introduced above. The generalization process than has to be implemented within the DBMS, or, if an even more distributed system is used, within a middle-tier application that is part of a multi-tier setup. A pre-requisite in this last setup is that the data tier and the middle tier are connected through a high bandwidth connection, possibly but not necessarily by running them on the same physical computer. Figure 2.2 shows the difference between a two tier and a multi tier system architecture. In the research a two tier approach was used, since the added functionality was implemented directly into the Database layer. The client application can be described as a thick client, since for use with the later on presented data structure, it has to do more than just displaying the data provided by the server.

In a two tier setup the client application directly accesses the DBMS to obtain data. A consequence of this is that the client application must be able to access that DBMS, i.e. a database driver must be available, and also that it can process the 'raw' data. In a multi-tier setup, the middle tier or application server is responsible for accessing the DBMS. Often in a multi tier setup the client only needs to use a standard web browser. In those cases the middle tier incorporates a web server that presents the client with a standard HTML format, possibly enhanced by Javascript or Java applets.

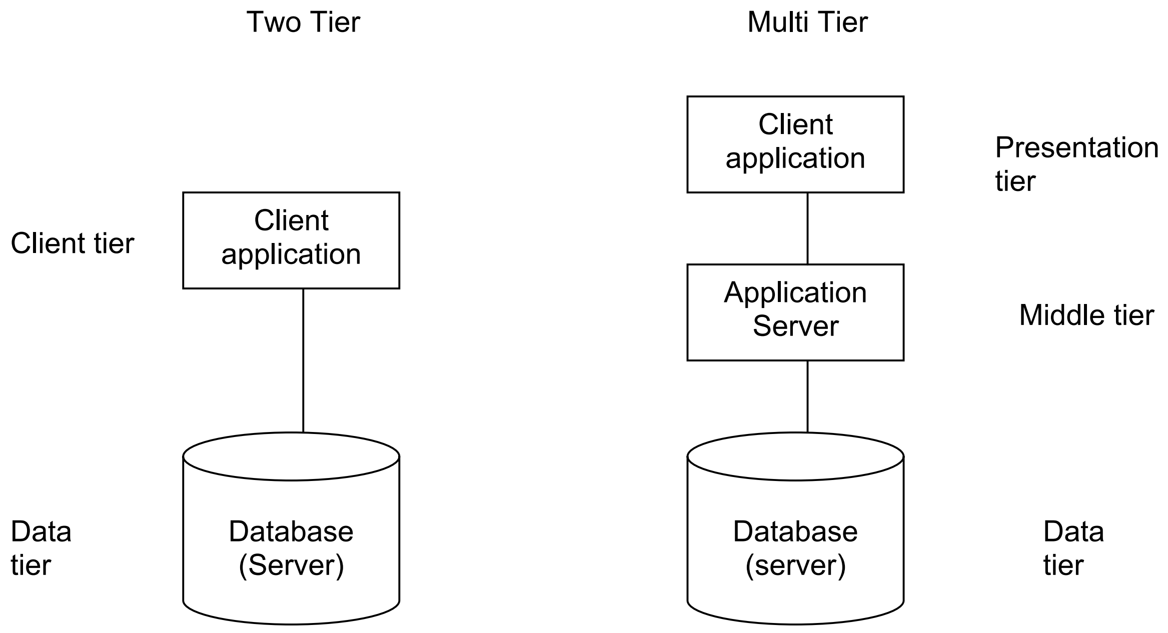


Figure 2.2: Two tier and multi tier setup

2.4. Scale-less vs. Multi-scale

Much of the literature on web mapping and generalization makes use of the terms scale-less and multi-scale, often without a clear discrimination between the two. In this thesis an important distinction is made in the sense that *scale-less* refers to a step-less transition between scales, whereas *multi-scale* refers to a discreet number of predefined scales. A scale-less data set is therefore a data set that can be used for any scale, with in reasonable limits, and can theoretically return different results for example for 1:10,000 as in 1:10,001. A multi-scale dataset can only produce results at a number of fixed scales, e.g. 1:10,000, 1:50,000 and 1:100,000. In both cases the term scale can be replaced by LoD for on-screen cartography, which will mostly be the case when using scale-less or multi-scale datasets.
The creation of multi-scale datasets is often achieved by repeatedly creating smaller-scale data sets out of larger-scale data sets and physically storing them. Each of these steps can be performed using automatic generalization, applying all necessary generalization functions, overviewed by a human operator if required, and can thus achieve good cartographic quality. Per scale level all geometric and topological conflicts can be solved prior to storing, since each level can be separately processed until a satisfying results has been achieved. Issues with regards to the automatic propagation of mutations throughout all levels have already been subject to extensive research [1], but are still difficult. A typical application of a multi scale data set would involve one master data set at the highest LoD. All lower detail levels would be derived from this master set. When and how mutations should be propagated to other scales would depend on the rules set up for this, e.g. periodically or after a certain number of mutations.

2.5. On-the-Fly generalization

Concept

On-the-fly map generalization is a term used to refer to a generalization process in which no generalized data sets are stored. Instead every time a request is made for a map with a certain level of detail, a volatile data set is created which exactly satisfies the specifications of the request. This data set can then for example be used for display purposes or other analysis. The concept of on-the-fly generalization is especially useful in interactive GIS applications where user demand fast response times. However generalization is a complex procedure requiring intensive computational effort of the hosting computer. To reduce the computational demand specialised data structures have to be used that enable acceptable response times, even for larger data sets. Unlike the solution of a multi-scale data set, which uses a number of pre-calculated and stored LoDs, the concept of on-the-fly generalization involves the use of a single integrated data set with little redundancy. The data structures applied in on-the-fly generalization therefore have to satisfy a number of demands:

- Support creation of data sets at ‘any’ LoD
- Allow quick response to client requests
- Minimize redundancy in the overall structure
- Minimize amount of data to be sent to a client

On top of these a number of less stringent demands can be formulated:

- Support a great number of generalization operators
- Support map-purpose dependant generalization

It is already difficult to address the first mentioned list of demands, addressing the second list, is especially hard since it requires extreme flexibility of the data structure. This flexibility in practice often conflicts with the need for quick response times and is therefore often sacrificed for speed.

Existing data structures

On-the-fly map generalization has been the subject of research for some time and a number of data structures have been developed. Here the focus will be on the GAP-tree and related data structures, which form the base for the data structures that are presented further on in this thesis.

The Generalized Area Partitioning tree is a data structure that is specifically designed for use with area partitionings [8]. An area partitioning is a division of a 2D surface into disjoined areas, such that each location on this surface belongs to exactly one area. Therefore no gaps or overlaps between these areas are allowed in an area partitioning. This concept is used in many map types, but can most clearly be recognised in choropleth maps.

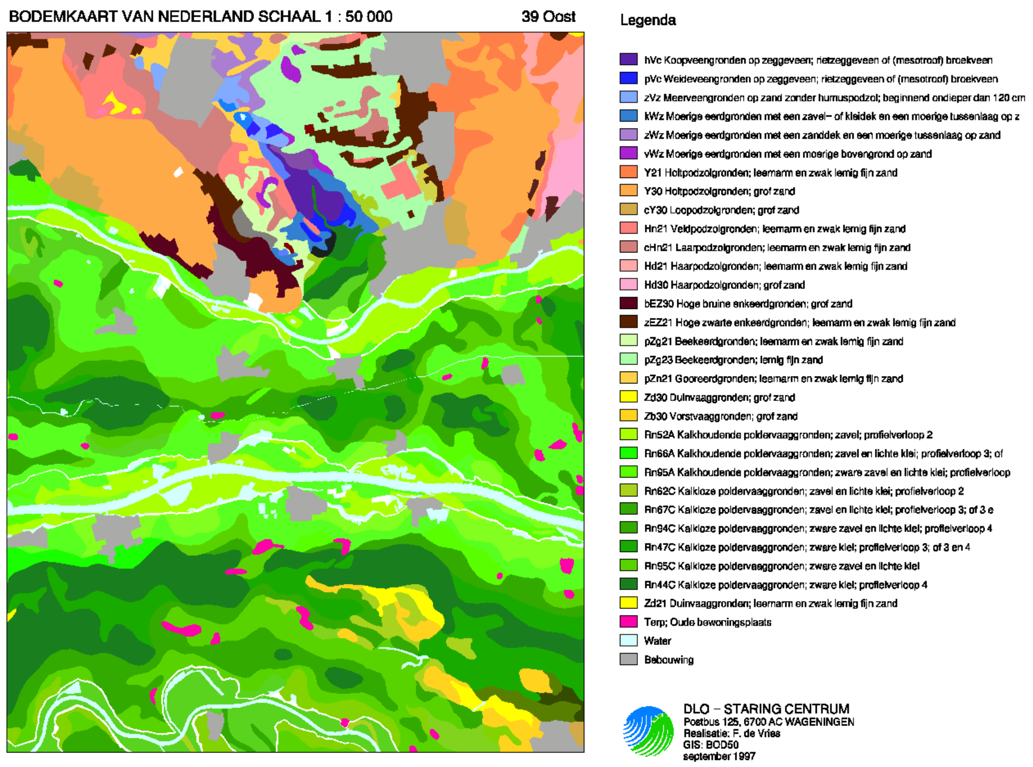

BODEMKAART VAN NEDERLAND SCHAAL 1 : 50 000
39 Oost
Legenda
hVc Koopveengronden op zeggeveen; rietzeggeveen of (mesotroof) broekveen
pVc Weideveengronden op zeggeveen; rietzeggeveen of (mesotroof) broekveen
zVz Meerveengronden op zand zonder humuspodzol; beginnend ondieper dan 120 cm
kWz Moerige eerdgronden met een zavel– of kleidek en een moerige tussenlaag op z
zWz Moerige eerdgronden met een zanddek en een moerige tussenlaag op zand
vWz Moerige eerdgronden met een moerige bovengrond op zand
Y21 Holtpodzolgronden; leemarm en zwak lemig fijn zand
Y30 Holtpodzolgronden; grof zand
cY30 Loopodzolgronden; grof zand
Hn21 Veldpodzolgronden; leemarm en zwak lemig fijn zand
cHn21 Laarpodzolgronden; leemarm en zwak lemig fijn zand
Hd21 Haarpodzolgronden; leemarm en zwak lemig fijn zand
Hd30 Haarpodzolgronden; grof zand
bEZ30 Hoge bruine enkeerdgronden; grof zand
zEZ21 Hoge zwarte enkeerdgronden; leemarm en zwak lemig fijn zand
pZg21 Beekeerdgronden; leemarm en zwak lemig fijn zand
pZg23 Beekeerdgronden; lemig fijn zand
pZn21 Gooreerdgronden; leemarm en zwak lemig fijn zand
Zd30 Duinvaaggronden; grof zand
Zb30 Vorstvaaggronden; grof zand
Rn52A Kalkhoudende poldervaaggronden; zavel; profielverloop 2
Rn66A Kalkhoudende poldervaaggronden; zavel en lichte klei; profielverloop 3; of
Rn95A Kalkhoudende poldervaaggronden; zware zavel en lichte klei; profielverloop
Rn62C Kalkloze poldervaaggronden; zavel en lichte klei; profielverloop 2
Rn67C Kalkloze poldervaaggronden; zavel en lichte klei; profielverloop 3; of 3 e
Rn94C Kalkloze poldervaaggronden; zware zavel en lichte klei; profielverloop 4
Rn47C Kalkloze poldervaaggronden; zware klei; profielverloop 3; of 3 en 4
Rn95C Kalkloze poldervaaggronden; zware zavel en lichte klei
Rn44C Kalkloze poldervaaggronden; zware klei; profielverloop 4
Zd21 Duinvaaggronden; leemarm en zwak lemig fijn zand
Terp; Oude bewoningsplaats
Water
Bebouwing
DLO – STARING CENTRUM
Postbus 125, 6700 AC WAGENINGEN
Realisatie: F. de Vries
GIS: BOD50
september 1997

Figure 2.3 Example for a choropleth map. [23]

Choropleths are thematic maps "in which quantitative spatial data is depicted through the use of shading or colour variations of individual unit areas" [6]. This type of map is mainly used for the display of thematic data such as soil type or population density. Figure 2.3 shows a choropleth map of the Netherlands, which indicates soil types. Other types of area partitioning maps are cadastral maps, and topographic maps.

The GAP-tree data structure stores a hierarchy of area objects in a tree structure, which allows efficient storage and retrieval of the objects at any LoD. This hierarchy is created by repeatedly simplifying the map display by leaving out the least significant area object. The significance of an object is concretised through an importance value, which is a function of the type of object or its thematic information, and it's size, usually in the form of an area or perimeter value. The leaving out of an object from the map creates a hole that needs to be filled to maintain a consistent map view. This usually means that the hole must be filled by the neighbouring features of the gap. For island features this is a simple task as they can by filled by the single surrounding feature. Other situations require more complex solutions. The creation of the GAP-tree requires the presence of a topological data structure with the following features:

- Node (0-cell). Point location with a list of references to edges, sorted on the angle of the line segment connected to the node;
- Edge (1-cell). Directed polyline with references to the left and right face;
- Face (2-cell). Area feature, with a list of references to all it's bounding edges, including possible inner boundaries.

Figure 2.4 shows these objects in a graphical context.

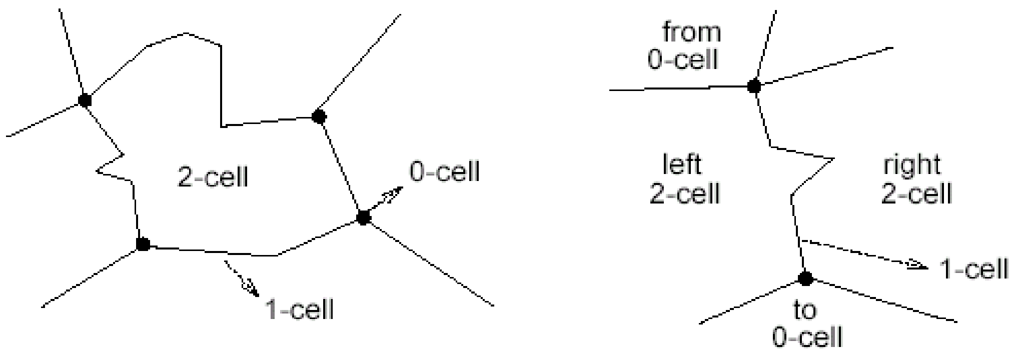


Figure 2.4. The topological data structure [8]

Once the data is present in this data structure, the GAP tree can be created using the following steps [8].:

1. For each face in the topological data structure an "unconnected empty node in the GAP-tree" is created;
2. Remove the least important area feature *a*, i.e., with the lowest importance *I(a)*, from the topological data structure;
3. Use a topological data structure to find the neighbours of *a* and determine for every neighbour *b* the length of the common boundary *L(a,b)*;
4. Fill the gap by selecting the neighbour of *b* with the highest value of the collapse function : Collapse(*a*,*b*) = *f*(*L*(*a*,*b*), CompatibleTypes (*a*,*b*), weight_factor(*b*)). The function CompatibleTypes(*a*,*b*) determines how close the two feature types of *a* and *b* are in the feature classification hierarchy associated with the data set.;

```
5. Store the polygon and other attributes of face a in its node in the
   GAP-tree and make a link in the tree from parent b to child a;
6. Adjust the topological data structure, importance value I(b), and the
   length of common boundaries I(b,c) for every neighbour c of the
   adjusted face b to the new collapsed situation.

Repeat the described steps 2-6 until all features in the topological data
structure are at the required importance level (for a certain display
operation)
```

To create a complete data structure, steps 2-6 need to be repeated until all faces have been merged into one large face. The GAP-tree that is then created can be stored so it can be used in subsequent queries. Figure 2.5 displays a small map and the complete GAP-tree that is associated with this map. In figure 2.6 the increase in detail, by progressively adding additional faces to the map is visualised. The GAP-tree faces need to be drawn in order from top to bottom.
Figure 2.7 shows the shape of the selected sub tree of a GAP-tree in relation with the zoom level.
For fast access of the appropriate nodes in a selection a reactive-tree index can be used [7]. This indexing method is especially suited for data with LoD properties, such as the GAP-tree nodes. However this indexing method is not available in most DBMS's. The task of adding a new indexing method and properly integrating it in the query system is very complicated and often impossible. The Reactive-tree index has been implemented in Postgres, the predecessor, of the PostgreSQL open-source DBMS. Some non-open-source DBMS's also provide the possibility to create new indexing types, such as the Oracle DBMS **Fout! Verwijzingsbron niet gevonden.**.

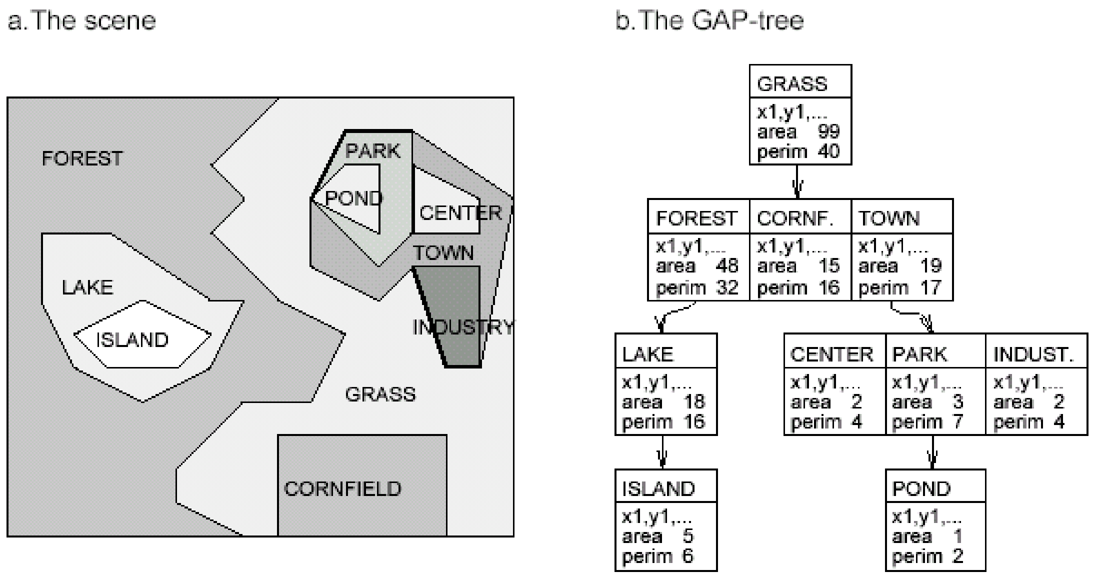

a.The scene
b.The GAP-tree
FOREST
PARK
POND
CENTER
TOWN
LAKE
ISLAND
INDUSTRY
GRASS
CORNFIELD
GRASS x1,y1,... area 99 perim 40
FOREST x1,y1,... area 48 perim 32
CORNF. x1,y1,... area 15 perim 16
TOWN x1,y1,... area 19 perim 17
LAKE x1,y1,... area 18 perim 16
CENTER x1,y1,... area 2 perim 4
PARK x1,y1,... area 3 perim 7
INDUST. x1,y1,... area 2 perim 4
ISLAND x1,y1,... area 5 perim 6
POND x1,y1,... area 1 perim 2

Figure 2.5: A map and its associated GAP-tree [8]

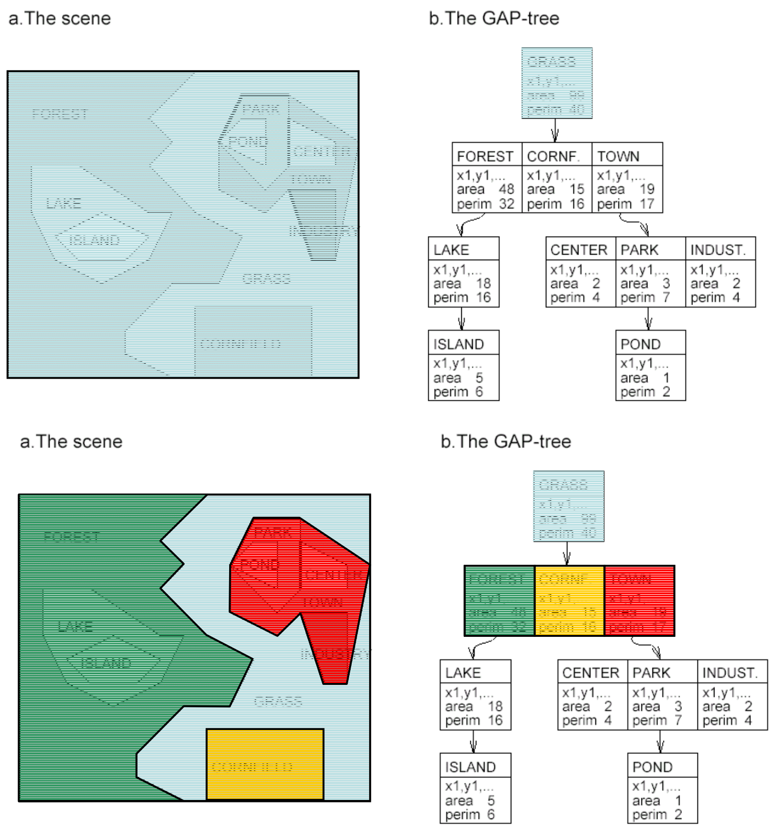

a.The scene
b.The GAP-tree
FOREST
PARK
POND
CENTER
TOWN
LAKE
ISLAND
INDUSTRY
GRASS
CORNFIELD
GRASS x1,y1,... area 99 perim 40
FOREST x1,y1,... area 48 perim 32
CORNF. x1,y1,... area 15 perim 16
TOWN x1,y1,... area 19 perim 17
LAKE x1,y1,... area 18 perim 16
CENTER x1,y1,... area 2 perim 4
PARK x1,y1,... area 3 perim 7
INDUST. x1,y1,... area 2 perim 4
ISLAND x1,y1,... area 5 perim 6
POND x1,y1,... area 1 perim 2

Figure 2.6: Increasing detail in GAP-tree data structure by drawing additional faces.

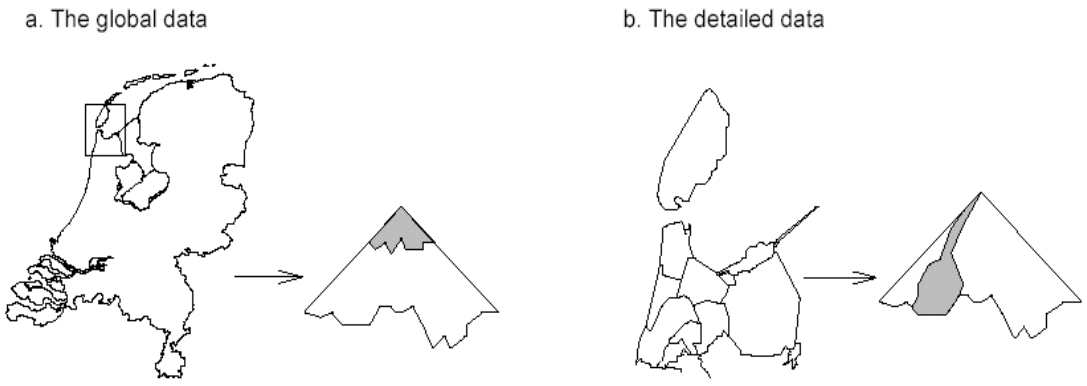


Figure 2.7. Shape of selected subtree of a GAP-tree, in low detail and high detail maps. The grayed area represents the selected nodes in the tree.[8] Note: The figure shown originates from [8] where it was used to indicate the use of the reactive tree index. However the shape of the selected sub tree of a GAP-tree selection is correctly visualised by this figure.

2.6. Conclusion

Generalization is an important issue in cartography and GIS in a wider view. A specific application of generalization is in the field of web-cartography, or indeed any application where the spatial data does not reside on the computer running the client or viewer application, where the amount of data that is transmitted should be kept small. This is especially the case when a low-bandwidth connection between the client and the server exist. One concept to tackle this problem is “on-the-fly” generalization, which tries to create a volatile generalized map using a single data structure. The overall subject of automatic cartographic generalization is still an important research subject with many open issues, and this also goes for the “on-the-fly” generalization methods.

3. Binary Line Generalization Tree

A considerable amount of the data stored in a GIS database consists of polylines. Whether it is used for describing linear features or to define the boundaries of polygons they are basically the same type of object. A reduction in the amount of data used in the representation of polylines could therefore be very beneficiary to a reduction of the total amount of data that has to be transferred from the database server to the client. However the generalization of linear feature requires a considerable amount of processing when done properly. To avoid this processing from having to be done at every request a data structure was implemented that can store the line generalization based upon the Douglas/Peucker algorithm [2], the Binary Line Generalization (BLG) tree [7]. This chapter discusses the workings of this algorithm and the specifics regarding the implementation in the DBMS used in the research.

3.1. Douglas/Peucker Line generalization

Many spatial data consists of linear features that are described by polylines and polygons. For generalization of line features often the term simplification is used. This line simplification has been the subject of extensive research over the last decennia and many different algorithms have been designed. One often-used method of simplification is the Douglas/Peucker algorithm. This algorithm, like many others used in generalization, is quite computational intensive. The standard approach to line generalization up to a specified tolerance using the Douglas/Peucker algorithm is to calculate the distance of all points to the line through the beginning and end of the polyline and then selecting the point with the largest distance. Should this maximum distance be larger then the specified tolerance then that point will be inserted into the resulting generalized polyline. The first and last points of the original polyline are by definition part of the output polyline. After the point is inserted, a left and right sub-polyline can be defined. The process can then be repeated for these sub polyline until the number of points in the next sub-polyline is equal to two or until the calculated maximum distance is smaller then the specified tolerance. Indicating that no additional points need to be inserted in the generalize polyline to satisfy the specified tolerance. The actual number of calculations that need to be performed depends on the number of points in the polygon and the specified tolerance but in general this process requires a large number of calculations. Also these calculations are performed on floating point data types, which are more time consuming then integer operations.

3.2. BLG-Tree

To prevent the calculations of the generalization from having to be performed for every request for a generalized polyline, a data structure can be used that stores information needed to create the generalized polyline. A useful structure for storing the Douglas/Peucker generalization is the BLG-tree. The BLG- or Binary Line Generalization tree stores the necessary information in a tree structure. Figure 3.1 displays this. The nodes of the tree represent the points in the polyline that is to be generalized. Within in a node 4 items are stored. First a reference to the point in the original polyline is stored. That point has a corresponding error value, which is the maximum distance value that is calculated using the Douglas/Peucker Algorithm. Furthermore the node has two references to its children, a left and a right subnode. To build this tree the Douglas/Peucker algorithm has to be executed with a tolerance of 0, or close to 0 if the application allows this. Thus all points that could ever be of importance in a generalized polyline, regardless of the required tolerance, are represented by a node in the BLG-tree.

During interactive use time the number of calculations that have to be performed is just a fraction of what has to been done to build the tree, more importantly, no expensive distance calculations have to be performed. To find out which points are to be included in the resulting polyline, the algorithm only has to descend the tree to the level where the error value stored in the node is smaller then the specified level of detail requires. The traversal of the tree mainly consists of simple comparisons and the following of references to sub nodes. Theses references refer to the indexes of the positions within the arrays. During this traversal the point indices encountered are to be stored in an array. The first and last point of the resulting and the original polygon are to be added separately since they are not present in the BLG-tree. After this the array of point indices that is now available, this has to be converted to a geometric object with the proper coordinates. The coordinates used are retrieved from the original polygon by their index in that array. In this process no geometric calculations.

3.3. Implementation

The Douglas/Peucker line generalization algorithm in combination with the BLG-tree data structure was the first generalization method that was implemented in this project. The primary reason for this choice was the fact that it is a relatively simple procedure, yet it can

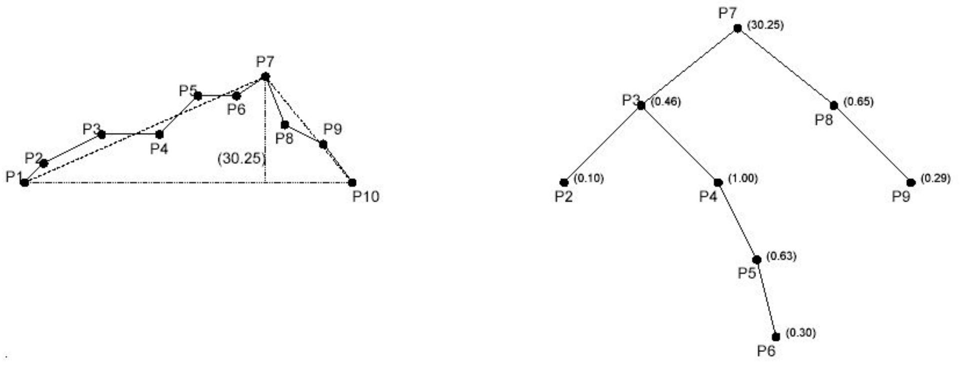


Figure 3.1. BLG-tree data structure. Left: Original polygon; Right: Corresponding BLG-tree. [7]

achieve very useful results. The first aspect to be described is the constructing of the tree and the storage of the tree within a database record. After that the retrieval of a generalized polyline using the BLG-tree is explained. Finally the some aspects regarding the compatibility of the implementation with the original data are discussed.

Building the tree

The BLG-tree presents some additional information for each record in the table containing the geometries. To allow for this information to be stored, an additional column needs to be appended to the table. Not all DBMSs support such an alteration of tables after their creation. Should this indeed not be possible then a new table has to be created containing the original columns plus the additional BLG-tree column and the original data needs to be copied to this new table. Afterwards the original table can be dropped. Fortunately the operation is possible in Oracle using the ALTER TABLE command. The BLG-tree column is to be used in conjunction with a standard geometry column. In Oracle the spatial type is called SDO_Geometry. This object type together with a large number of functions and procedures that work on it, as well as indexing methods are part of the ‘Spatial’ extension of the standard Oracle DBMS. The Oracle implementation of spatial data types uses one single object type that can contain any of the possible geometric types. This sdo_geometry type is defined in Oracle as [17]:

```
Create type sdo_geometry as object
(
   Sdo_gtype number,
   Sdo_srid number,
   Sdo_point sdo_point_type,
   Sdo_elem_info mdsys.sdo_elem_info_array,
   Sdo_ordinates mdsys.sdo_ordinatearray
);
```

This definition uses three other definitions:

```
Create type sdo_point_type as object
(
   X number,
   Y number,
   Z number
);
Create type sdo_elem_info_array as varray (1048576) of number;
Create type sdo_ordinate_array as varray (1048576) of number;
```

As mentioned above, this single SDO_GEOMETRY type is used to store any geometry description. The actual type of a geometry is indicated by the combination of the value of the sdo_gtype field. The following geometric types are defined in Oracle spatial:

- Point
- Line or Curve
- Polygon
- Collection
- MultiPoint
- MultiLine or MultiCurve
- MultiPolygon

The Douglas-Peucker line-simplification algorithm is primarily intended to be used on polylines. The implementation that was created only allows the use of the Oracle Line type. A special form of a polyline is a ring, which is a line from which the starting point and the end point lie at the same location. These lines need special treatment since it is not possible to fit a line through a single point. To support these rings two different versions of distance function were implemented. One that uses a point-line distance function, which calculates the distance of a point to an infinite line defined by two points. The other distance function is called the point-point distance, and calculates the distance between two different points. The need for and layout of the different treatment of rings is shown in figure 3.2. The BLG-tree creation algorithm only accesses a single distance function with three points as the input. Should the two points defining the infinite line have the exact same coordinate, then the point-point distance is calculated and returned.

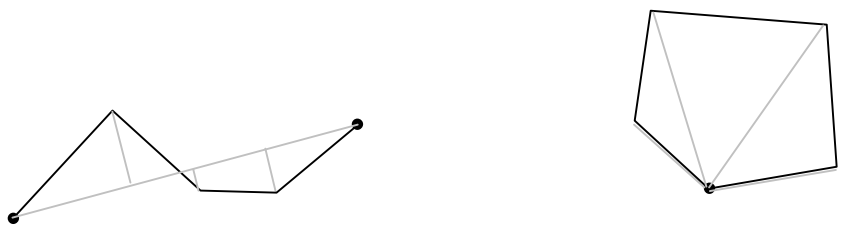


Figure 3.2 The standard situation (left) always uses point-line distance calculation. For rings (right) the first iteration uses point-to-point distance calculations.

The implementation of the calculation functions for both the point-line and the point-point distance both are based on the use of 2D Cartesian coordinates. Implementation of this distance function is easy and furthermore, the Dutch national grid RD uses Cartesian coordinates so data in that system could immediately be processed, which applies to all the data used in testing. The BLG-tree data structure itself does allow the use of other distance functions, so it is possible to uses other coordinate systems, provided that distance functions in that system are present.

The BLGTREE type used here is not a native data type in Oracle, it has to be explicitly defined using the following statements:

```
create or replace type floatarray as varray(524288) of float;
create or replace type intarray as varray(524288) of int;

create or replace type blgtree as object
(
  point intarray,
  error floatarray,
  leftnode intarray,
  rightnode intarray
);
```

The first two lines define the types FLOATARRAY and INTARRAY as VARRAY's of respectively FLOAT and INT types. A VARRAY is an array of variable length, which at maximum can contain the number of values as listed in the type definition. The array's as defined here can therefore contain a maximum of 524288 values. This value is half the size of the maximum size of the Oracle Sdo_ordinates array. This value corresponds to the maximum number of points that can be stored in an Oracle geometry, since each 2D points use 2 positions in the Sdo_ordinates array.
After execution of these statements it is possible to refer to the BLGTREE type in Oracle. To append a BLG-tree column to a table the following SQL command is issued:

```
ALTER TABLE <TABLENAME> ADD COLUMN (BLG BLGTREE);
```

The BLGTREE type consists of 4 array's of variable length. Four items with the same array index comprise one node of the BLG-tree. All four arrays therefore will have the same length. Usually this length will be equal to the number of points in the polyline minus 2 for the implicitly used points add both ends.
The point array stores references to the points in the original polygon. Thus the data in the original polyline is not copied and redundancy of that data is prevented. Furthermore the original data remains present and available to all applications that access this data. Storing the actual coordinates within the point array, instead of just the indices, and subsequently deleting the original geometry column could achieve a reduction of overall data size. But this would make all queries that require the original that, use the generalization process, which, while speeded up using the data structure, still takes effort then not generalizing. More on these performance issue can be read later on in this chapter.
The error array stores the error values related to that point. The values use the same unit as the original geometry. The leftnode and rightnode arrays contain references to the child node of the current node by storing their respective indices within the set of array's. To clarify this BLGTREE type an example will now be discussed.

In this example a polyline containing 9 points is processed. The resulting BLG-tree will then contain 7 nodes, since the begin- and end points of the polygon are always present in the generalized version and are therefore not stored in the tree. The following array's represent such a tree:

```
node :      0|      1|      2|      3|     4|       5|      6|
-------------------------------------------------------
pt   :      6|      2|      1|      5|     4|       3|      7|
lnode:      1|      2|      0|      4|     5|       0|      0|
rnode:      6|      3|      0|      0|     0|       0|      0|
Err  :1.5E-4|1.3E-4|3.2E-6|1.9E-7|1.7E7|1.16E-6|6.6E-8|
```

As can be seen in this example the root of the tree refers to points 6 in the polygon and has an error value of 1.5E-4. The left child node is located at index 1, which is associated to point 2 in the polygon. The right child node of node 0 refers to index 6 and thus to point 7. If the lnode or rnode array contains the value 0, it means that the current node has no child node in that direction. The procedures necessary for building the BLG-tree are implemented in java through stored procedures.

Retrieving data

When generalized data is requested a geometric shape has to be constructed that presents a generalized version of the original polyline. To create this polyline, the necessary points have to be retrieved from the BLG-tree. This is done, by descending the tree until the error that is associated to the encountered node, is smaller then the requested level of detail. The generalization functionality was implemented such that it can be accessed via function calls within SQL statements. The request for generalized data can be formulated as:

```
SELECT GENERALIZE(GEOMETRY,BLG,1000) FROM GEOMTABLE;
```

This request specifies the geometry column, the BLG-tree column and the tolerance that is requested. The GENERALIZE function is a PL/SQL function stored within the database and is responsible for returning the appropriate, simplified geometry. In order to do this, the current implementation of the function uses the entire geometry. Based on that geometry and the BLG-tree the function selects all appropriate points and constructs the resulting simplified geometry. This is a recursive process that is made clear in the following pseudo code:

```
//BLG-tree node
POINT    :     array with references to points in the original geometry
ERROR    :     array with Douglas/Peucker error values associated to the
               points refered to by the POINTARRAY
LEFTNODE:      array with references to left child nodes
RIGTHNODE:     array with references to right child nodes
               //Node(i)  constist  of   POINT(i),  ERROR(i),LEFTNODE(i)  and
               RIGHTNODE(i).
RESULT:        the resulting polyline

Function  generalize(INPUTGEOM geometry, INPUTBLG blgtree,
TOLERANCE float) : geometry
begin
      RESULT.addpoint(INPUTGEOM[0])
      Checknode(0)
      RESULT.addpoint(INPUTGEOM[INPUTGEOM.length])
      Return RESULT
end
```

```
procedure checknode(index integer)
begin
   if ERROR(index)>TOLERANCE
   then
   begin
      if leftnode[index]>0 then checknode(leftnode[index])
         RESULT.addpoint(INPUTGEOM[index])
      if rightnode[index]>0 then checknode(rightnode[index])
   end
end
```

The main function here is the generalize function. It receives the original geometry, BLG-tree there is a BLG-tree present then the function continues to create a generalized polyline else it returns the original input geometry. The resulting polyline is created by first adding the first point of the original polyline, then calling the CHECKNODE procedure to insert all the points that satisfy the tolerance factor and finally adding the last point of the original polyline. The CHECKNODE procedure is recursive in nature and calls itself for all subsequent sub nodes should the requested tolerance not yet be satisfied. In the CHECKNODE algorithm the following line:

```
if ERROR(index)>TOLERANCE
```

is used to check whether the point in the current node needs to be added to the result. The choice of the > sign instead of the >= assures that only just enough points are returned to satisfy the specified tolerance. Should the >= sign be used then when specifying a tolerance of 0 then all points of the original polyline are returned, including points on a straight line, which do not add to the shape of the polyline. Here a choice of for example –1 (actually any negative number will do) as the tolerance would return the original polyline, since all error values are by definition positive since they are distances. However out of performance consideration it would not be advisable to use this procedure to retrieve the original polygon since then the generalization overhead would be used unnecessary as the same data can be retrieved without generalizing.
The tolerance value specifies must given in the same units as those used in the input data. The choice of the exact value depends on the desired accuracy of the returned geometries. Should the tolerance value be set to the real-world width of a screen pixel, then there is no significant difference between visualisation of the generalized polyline and the original polyline. It must be noted that when both the original polyline and the generalized polyline are visualised, with that choice for the tolerance, that actually some differences can be seen between the two. This difference is due to the fact that both the visualisation of the original as well as the generalized polyline are subject to round off errors that occur when the real-world coordinates, often represented by float double values, are transformed and rounded to integer screen coordinates. Very small differences in the float representations can cause a jump from one pixel on the screen to the next, and thus cause another pixel to be coloured. Figure 3.4 shows this phenomenon.

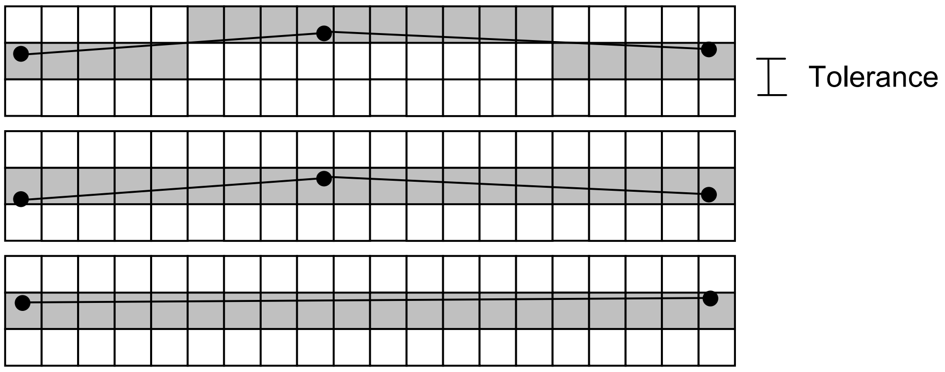


Figure 3.4: Even a tolerance value smaller than the pixel size can cause visible difference between the original and the generalized polyline. The top image represents the original polyline; the bottom image shows the generalized polyline. The middle image indicates that even a small change in the input line can change the on screen appearance.

Figure 3.5 gives an example of effects of the BLG-tree generalization on the test dataset. The blue lines in this figure represent the original data, the black lines are from the generalized set. The tolerance that was applied here was larger then the one required to precisely represent the original data at the pictured scale. That is the reason why in this picture differences between the blue and the black lines can be seen. In some places only the black lines are visible. This can be due to two reasons. First of all, the original dataset does not contain more detail then represented by the generalized line. Secondly the original data partly consists of other types then only the LINESTRING type and the implementation has been made such those other geometry types, such as CURVESTRING, which contains circular arcs instead of straight line segments, and MULTI LINESTRING, are returned in their original form. More on this topic can be found in the next paragraph. Should the correct tolerance have been used in the query, then no difference should have been visible between the original and the generalized data, apart from those caused by rounding of.

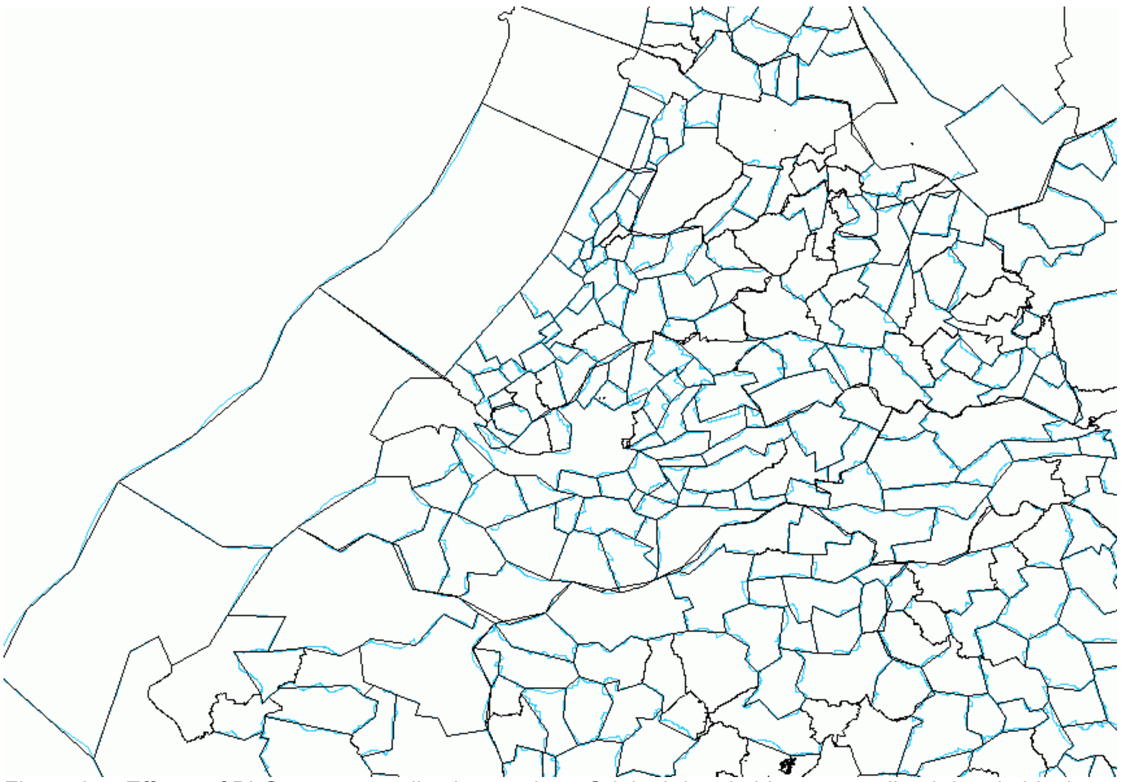


Figure 3.5. Effects of BLG-tree generalization on data. Original data in blue, generalized data in black.

Compatibility with original data
In Oracle the type of geometry used can vary per record. It was chosen to implement the BLG-tree only for the simple Oracle LINESTRING geometry type. The tree building algorithm checks whether the input geometry is of the appropriate type. If this is the case then a BLG-tree record will be created, else an empty record will be returned. During retrieval of a generalized geometry a similar process is used. Should the record be of the simple LINESTRING type and a BLG-tree record is present then a generalized LINESTRING is returned else the original geometry is returned, regardless of the actual type. Thus the algorithm always returns a useful answer that is compatible with the original dataset. The optimum result is of course achieved if all geometry records are of the simple LINESTRING type. It is possible to extend the capabilities of the BLG-tree implementation to include other data types but this was not attempted in this research.

3.4. Performance

One of the major reasons for the use of generalization is the reduction of data transferred from the database server to the client. This reduction of data should reduce the transfer time of a result. The reduction in transfer time depends greatly on the speed of the connection between the client and the server. Connections with smaller bandwidths should benefit more from the reduction than high bandwidth connections. Furthermore the generalization algorithm provides the server with a greater computational load when responding to a request to a generalize query then to a normal selection query. This could even result in the effect that a generalized set, with far less data then the original data set, is received slower then the original set. The reason for this is, the fact that the implementation of the BLG-tree data structure and the associated query-time routines, need to load the entire original geometry, as well as the entire accompanying BLG-tree.
To test the performance of the created algorithm a test set was used. This set, which will be referred to as LineBoundary, contains all simple polylines, or LINESTRINGs as they are called in Oracle, present in the table Gemeentegrenzen. This Gemeentegrenzen table contains all the municipality boundaries as they can be derived from the Dutch cadastral database LKI. The LineBoundary table was extended, according to the previously described method, with a column to contain the BLG-tree data. This column was subsequently filled using the developed CREATEBLG function. Thus a table containing 1501 records with polylines of varying length was created, with a properly filled BLG-tree column. This column could then be used to perform tests on using queries with the GENERALIZE function. For query-time performance testing a Java program was written which acts as a client for the Oracle database. This program conducts a query on the server and retrieves the result. The result is analyzed by recording the time it takes from issuing the query, to retrieving all the data. Furthermore some statistics are calculated based upon the retrieved data. These statistics contain, the total number of records returned, the total number of points in all those records and the average number of points per record. Since the returned data type in this test is always a LINESTRING, this last number indicates the average number of points per generalized LINESTRING. This test was executed for a range of tolerance levels. The test was first executed on the same computer as where the DBMS and the data reside. The effect of this is that these timing results depend greatly on the computational speed since data transfer only takes please in a very small, physically as well as logically, environment, which means that there is practically no hinder of a slow connection speed between client and server. This becomes immediately apparent when the timing results as shown in figure 3.6 are analyzed. The blue dot indicates the return time of the original, full size, data set. This contains 384268 points and is returned in just over 7 seconds. The most generalized dataset tested here returned 4468 in total. Which corresponds to almost 3 points per polyline, whilst 2 points per polyline are per definition present. This result isn't that surprising, since the generalizing algorithm actually needs to load the entire original polyline to create a generalized result. Apparently the connection between the client application and the server application within one computer is about as fast as the retrieval speed of data within the DBMS itself. Transfer to the client therefore takes about as long as transfer to the generalize

function in the database, which then still needs to process the request and formulate an answer in the form of generalize LINESTRINGs, which accounts for the additional time. Transfer to the client is, of course again very fast, and for small, that is highly generalized, result sets should be almost negligible.

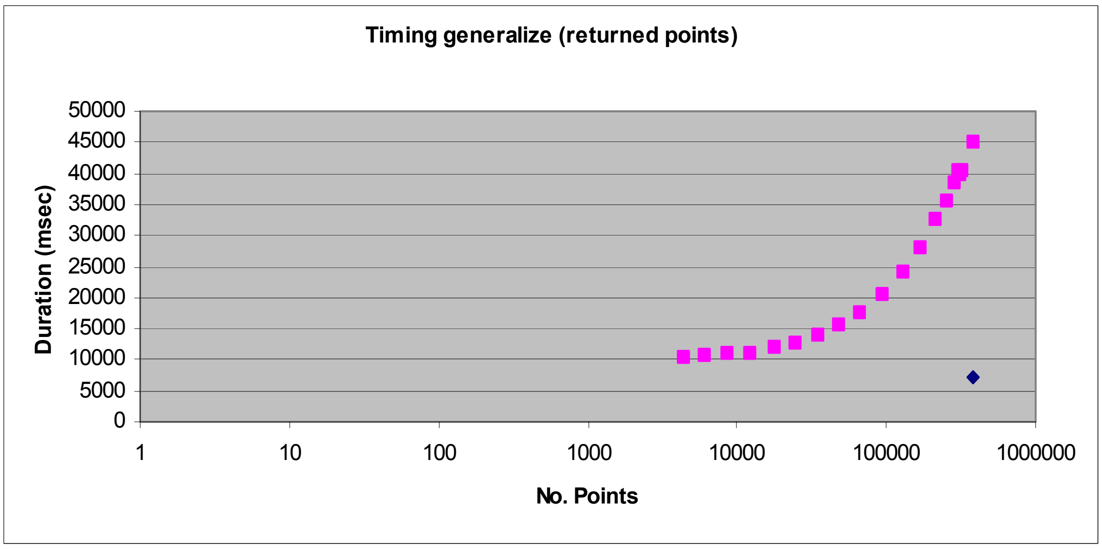


Figure 3.6. Timing results within server. The pink dots represent generalized data, the blue dot represents the original data.

To test the results when using a slower connection, the same program was also run on the other side of a 56k6 telephone modem connection. To avoid very long connection times this latter test was conducted on a limited amount of records and using high tolerance values. This resulted in the measurements as shown in figure 3.7.
From these measurements it can be concluded that when the connection between the database server and the client is relatively slow, there can be great benefits in transfer time of the necessary data for visualization. In the test dataset a tolerance of 400,000 corresponds to a visualization of the entire area on the Netherlands on an average computer screen, in the order of 1000 pixels in either direction.

The correlation between the applied tolerance and average number per polygon is displayed in figure 3.8. In the used data set the unit of the coordinates is millimetres, so tolerance of 1000 corresponds to a tolerance of one meter in reality. For many applications this could be an acceptable precision, and this setting reduces the average number of points from about 220 to about 70 per LINESTRING, which is a significant reduction of about 70%.

As is shown in this chapter, the use of the BLG-tree can provide a significant improvement in response time to queries over small bandwidth connections and is therefore a useful extension to the LINESTRING data type. The implementation as used in this research allows the original data to remain untouched and fully useable. This implementation might be useful in other generalization procedures that are described further on in this thesis whenever linear features are involved.

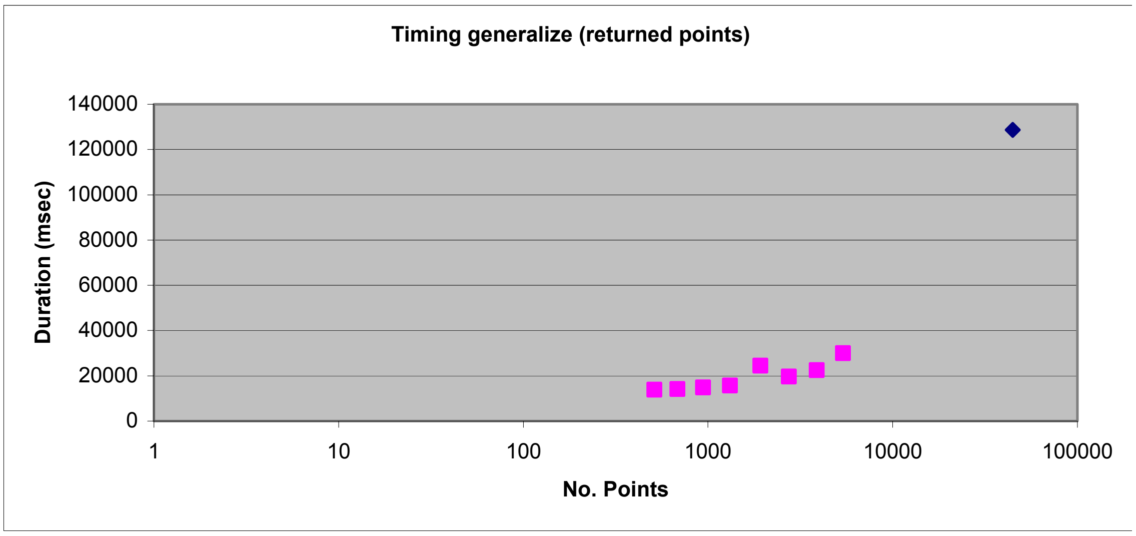


Figure 3.7. Timing results within 56k telephone modem connection

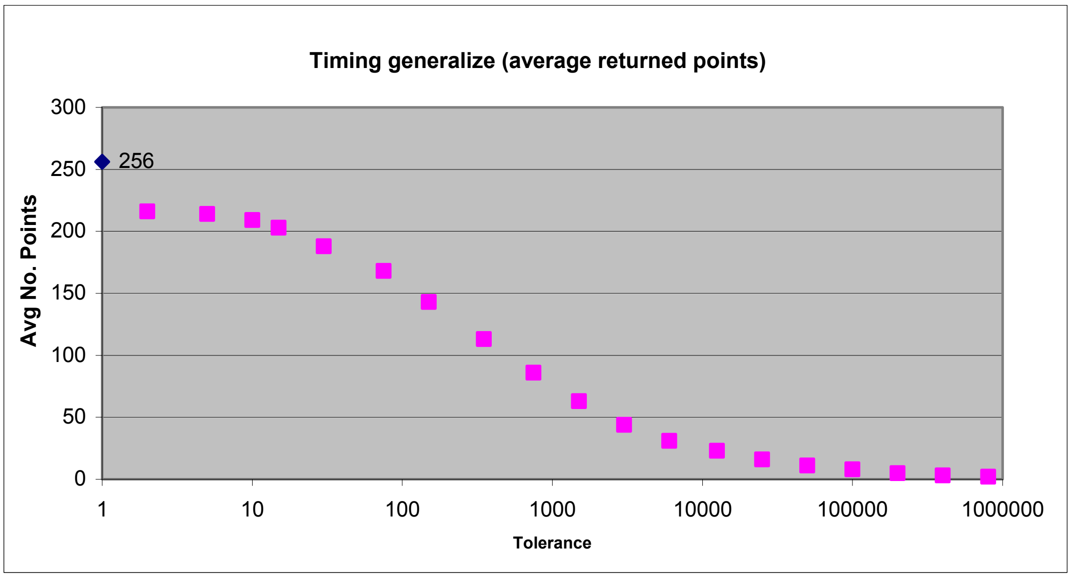


Figure 3.8. Influence of generalization on number of points per LINESTRING

4. Development of a Topological Data Structure for On-the-Fly Generalization

The goal of this research is to design, implement and test a scale-less data structure that allows the storage and retrieval of generalised spatial data within a ‘standard’ spatially enabled DBMS. The data structure must allow the on-the-fly creation of map display at a range of scales. This chapter presents algorithms and data structures, designed, used and sometimes discarded during the research. The discarded data structures are presented for several reasons. First they give an indication as to the development of the final data structure. Second, they were each discarded for specific reasons that influenced the exact shape of the final procedure. And third, they contain very promising features on some fields, whilst presenting problems in others. Should these problems be overcome in some way that retains the usability of these features, further development in that direction might be possible. Since the data structures are all variations on the same theme first some general aspects regarding all methods are discussed. After that the individual data structures are explained.

4.1. General information on all methods

Separating thematic and geometric information or introducing topology
A fundamental property of the data structures is the separation of the thematic and geometric data into two tables. This feature allows for changing either of the two without having to change the other. It is therefore possible to reuse the same boundary for more than one object. Without having to change, or duplicate it’s geometric properties.
The second, important feature is the use of a topological data structure for storing the geometric data. The topological storage is based upon edges.
Both the thematic and the geometric data need to be accompanied by some information that supports on-the-fly generalization. In all data structures this information is provided by two values, a lower and upper bound value for the LoD at which the feature should be present in a map display. An appropriate value for this scale-factor is the reciprocal value of the scale. A scale of 1:10,000 would then correspond to a scale-factor of 10,000. This value is also intuitively acceptable since it corresponds to the idea of height. The larger the scale-factor the further zoomed out the map is, or the higher the viewer is from the surface. Thus a higher scale-factor corresponds to less detail.
In the final tables these fields might not be present explicitly, but instead they can form the z-values used to construct a 3D bounding box. This 3D bounding box can than be used to create a single spatial index which supports the simultaneous selection on both the 2D query window, as well as the desired LoD. An edge table in any of the data structures has the following basic layout:

Column name	Type	Description
Oid	Integer	A unique identifier
Geometry	Geometry(2D LineString)	The geometric shape of the edge
Bbox3D	Geometry(3D Box)	A 3D boundingbox, with 2D extent bounding the geometry and 3D extent covering the appropriate LoD’s.

Table 4.1: Basic layout of Edge table.

Besides these fields, there will also be some additional fields relating to the assignment of thematic information to the bounded faces. The necessary data for that link depends on the actual data structure and will be given in their respective paragraphs.
The thematic information is stored in what will be called the 'face table', for the information is related to area features. The storage of the thematic information is mainly a simple table with a unique identifier and some fields containing the actual information. The actual information stored, depends on the data available in the input data and on the purpose of the generalized map. Depending on the data structure there can also be the need for some additional information to create a proper link between the geometric shapes and the thematic information, e.g. a point within the geometric boundary.

Column name	**Type**	**Description**
Oid	Integer	A unique identifier
Bbox3D	Geometry(3D Box)	A 3D boundingbox, with 2D extent bounding the geometry and 3D extent covering the appropriate LoD's.
Thematic Info	…	Any thematic information that needs to be stored about each face

Table 4.2: Basic layout of Face table.

Using third dimension for LoD information
An integral part of the setup of the data structure is the use of a third geometric dimension for the LoD information. The major advantage of this choice is that it enables the use of 3D indexing methods on the combined geometric and LoD data. This in turn allows the integral selection of records based on both the geometric as well as the LoD requirements. The 3D geometries that are used to support these indexes are 3D boxes, created as the 2D bounding boxes of the geometric shapes of respectively the edges and the faces, extended with the LoD values for the third dimension. Although the third dimension has no geometric meaning in the data structures, 3D indexing methods do interpret it as being geometric. As will become apparent in the following chapters, the values in this third dimension can vary greatly and more importantly, vary much more than their first and second dimension counterparts. As a consequence of this, very large 3D boxes might be constructed, which is not beneficiary to the workings of the indexing methods. If this presents a problem, linear scaling of the third dimension can be an option, to reduce the range in that direction.

4.2. Polygon reconstruction based on geometric relationships between edges

An important aspect of the procedure is the fact that the geometric shapes of generalized faces are determined at the client-side at query time. The geometric shape of the faces is not stored explicitly but instead should be reconstructed based on the separate edges. This process is named polygon reconstruction in this thesis. The polygon reconstruction starts with an unsorted list of the edges comprising the boundaries of all polygons that need to be reconstructed. This list of edges is obtained by a query of the client to the server.
The reconstruction algorithm needs two directed edges for each boundary, one for each adjoining polygon. Therefore each edge must be present twice in the edge list, each with an opposite direction to the other. Since the list initially only contains one edge per boundary segment, as returned by the server, this demand is satisfied by adding a reversed copy of each edge to the list. The next step is to create nodes at the start of each edge. Whenever two edges meet, i.e. multiple edges start at the same location, multiple edges will share the same node. Each node contains a list of all outgoing edges. This list needs to be sorted on

the angle of the first line segment of each outgoing edge in order for the reconstruction routine to be able to find the next edge in a ring. With the list of edges and nodes ready, rings can be detected. A ring is a closed sequence of line segments that encloses a region and does not contain any self-intersections. Rings are created using a program loop that adds edges to the ring until the end of the last added edge has the same coordinates as the beginning of the first edge. This program can be described by the following pseudo code.

```
Select arbitrary edge
Select the node at the beginning the edge
Store this edge in firstEdge
Loop
   Add linesegments in the edge to OutputRing
   Select the node at the end of the edge
   Select next edge in clockwise direction from edge at selected node
Until (selected edge)=(firstEdge)
```

Pseudo Listing 1: Description of ring reconstruction using edges and nodes.

The workings of this method are also visualised in figure 4.1. To prevent the routine from processing edges more than once, edges are marked when they have been processed. The selection of the arbitrary first edges then skips edges that have already been processed.

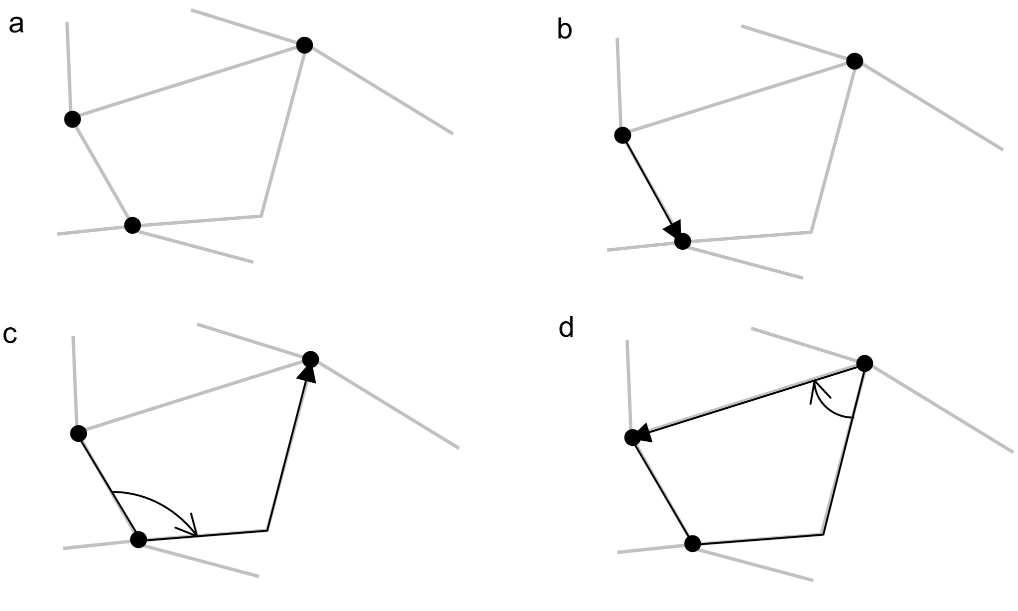


Figure 4.1: Ring reconstruction.
1. (Directed) edges and nodes
2. Select an edge to start with
3. At next node, select next edge clock wise
4. Repeat until next node of last edge is located at the start of the first edge

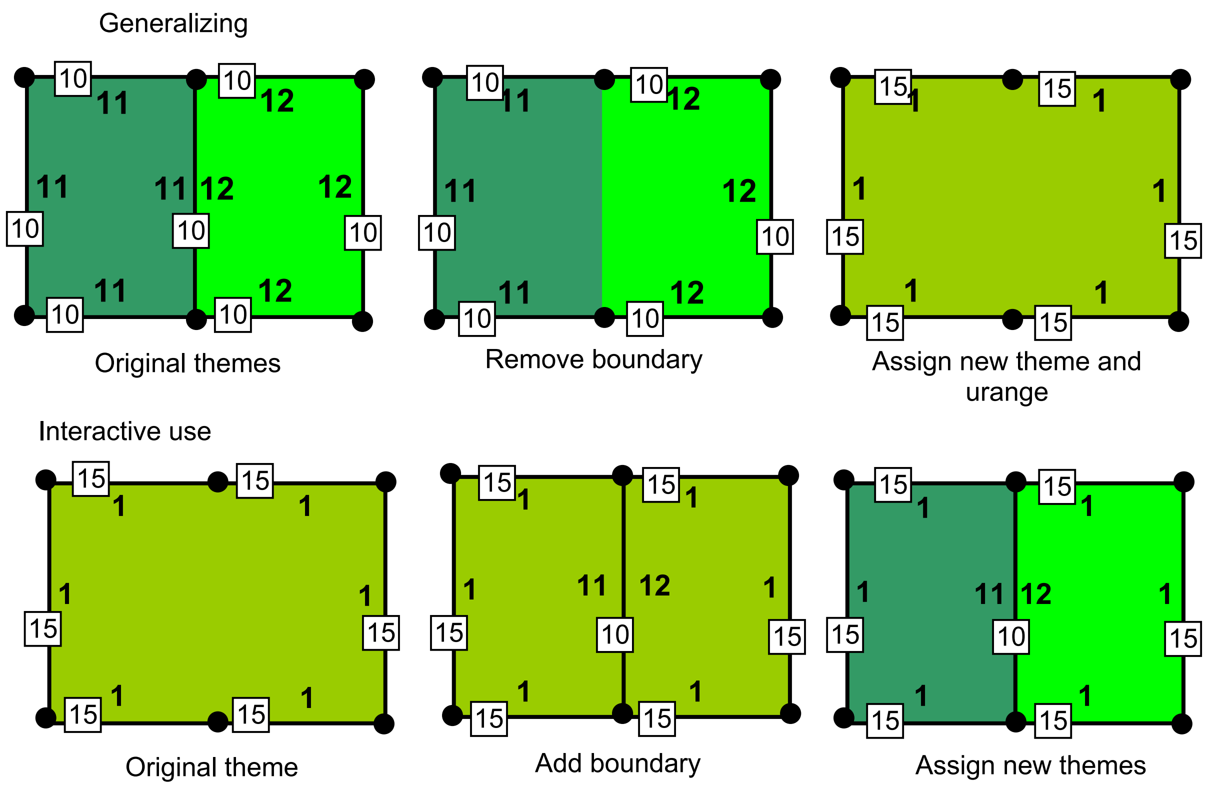
Using this algorithm it is possible to create all rings present in the set of rings. These rings should be separated into two groups. One that are the shells, or outer rings of polygons, and one that are the holes or rings in the interior of polygons. The described ring reconstruction algorithm automatically creates shells which are counter clockwise oriented rings and holes which are clockwise oriented. Holes should be assigned to the smallest shells they lay in, i.e. (area(holes)<area(shell)) and (hole lays within shell). The combination of a shell with zero or more holes is called a polygon. These polygons can be visualised on a clients display using standard functionality to show geometric objects.
Besides the geometric properties of the objects to show, proper visualization of the polygons also requires that appropriate symbolization is used. The symbolization should be based

upon the thematic information. It is therefore important that some kind of link exists between the polygons and the thematic information stored in the face table. The exact nature of this link varies with the specific data structure.

The polygon reconstruction method has two important prerequisites with regards to the creation of the topological elements describing the planar map, and secondly with the content of that topological data. Although the polygon reconstruction process does need a topological data structure consisting of edges and the nodes at which they touch each other, these nodes are not stored in the database. Instead they are determined only just prior to the reconstruction, based upon geometric properties of the edges. The position and content of the nodes are determined at query time based upon the coordinates of the first point of each edge. A node is located at the first point of each edge. Whenever two edges start at the same exact location they share the same node. For this method to create all the necessary nodes it is important that edges do not contain internal points that are located at nodes, since those would not be considered nodes, and therefore not be used in the reconstruction proces.
The second prerequisite is that the total set of topological data present after a query is enough to reconstruct at least all polygons that are totally or partially visible on the clients display. This second prerequisite appeared to be the most difficult to satisfy and the different ways to satisfy this demands determined the main difference between the variants of the developed data structure. In fact the impact of this demand only became really apparent after the concept of the first variant. In the remainder of this thesis this demand, or rather the lack of satisfying it will be referred to as ‘the problem of the missing edges’.

4.3. Distinct Edge-Face Coding

This rather cryptic title refers to the way face records are linked to the bounding edges of a face. A standard way of linking edges to faces, is by storing a reference to the left and right hand face of a directed edge. If this is done in a proper way, then all edges surrounding a face should have the same reference to that side of the face. The method is visualized in figure 4.2. This method is for example used in various data types used for the TOP10vector data. The TOP10vector line-objects are stored in what is called the bas-file. Each line-object in this file each line has a LPOLY and an RPOLY field which contains a numeric reference to the polygon at the left respectively the right side of that line. Since often a polygon will be bounded by more than one edge there is a redundancy in this method of storing, as normally all edges surrounding a polygon should have the same reference at that side of the polygon. This redundancy is exploited by the data structure presented here. As with the later on presented data structures geometric generalization is achieved by omitting edges from original map. The omission of edges results in larger planes, these will however not have consistent face references in all the edges. The correct face reference can be selected by taking the reference of the edge with the lowest scale-factor value, since this one will refer to the most detailed face. The source data in the example has a scale-factor of 10. When generalizing, faces are merged, by removing their common boundary. These edges are not actually removed, but are stored in a database. The remaining face is intended for use at higher scale-factors, i.e. zoomed out. The edges that bound that face are therefore assigned higher scale-factors and also new references to the new face they now bound. This entire process is visualised in the top row of figure 4.2



Legend

●	Node
1	Wood
11	Coniferous wood
12	Deciduous wood
$\boxed{10}$	scale-factor value

Figure 4.2 Concept of distinct edge-face coding.

In interactive use of the data, the process is reversed. If a map is requested with scale-factor 15 then all edges are retrieved with a scale-factor of 15 and larger. Using these edges all possible polygons are reconstructed. In the example this yields one polygon. All edges bounding this polygon have the same scale-factor and also the same face reference, i.e. face 1. When the user decides to zoom in, more edges are needed to provide more detail. Thus at a scale-factor of 10 or less the centre edge is needed. Again all possible polygons are reconstructed which results in two polygons. Bounding each of these polygons are both edges with scale-factor 15 and an edge with scale-factor 10. The rule with this data structure is that, of all edges bounding a face, the reference from the edge with the lowest scale-factor needs to be used to determine which face it is. Applying this rule to the bottom centre graphic, reveals that the left polygon is face 11 and the right polygon is face 12. Using this information the face can now be visualised with the correct styling.

4.4. Storing bounding boxes

A solution for the problem of the missing edges was found, by using bounding boxes of faces. Since bounding boxes are very simple geometries, even spatial calculations can be performed quickly and easily, especially when optimized functions are used, instead of functions applicable to general geometries. This method primarily uses bounding boxes to select faces and edges. Each face stores it's bounding box together with an upper and lower scale range. These are indexed as if they would form a 3D bounding box. The same is done for edges. During query time, the request for data in a query window, at a certain LoD. is transformed into a 3D spatial query, which returns all faces, of which the 3D bounding box intersects with a rectangle with x,y extent equal to the query window and a z value equal to the requested LoD. This query returns a number of faces. Based on the result of this query a new query window is constructed with the same z value but with x, y extent the bounding box of all returned faces. This new, larger query window is used to retrieve all necessary edges, by selecting all edges of which the 3D bounding box is intersected by the window. Actually this query window will return more edges than strictly necessary. The number of surplus

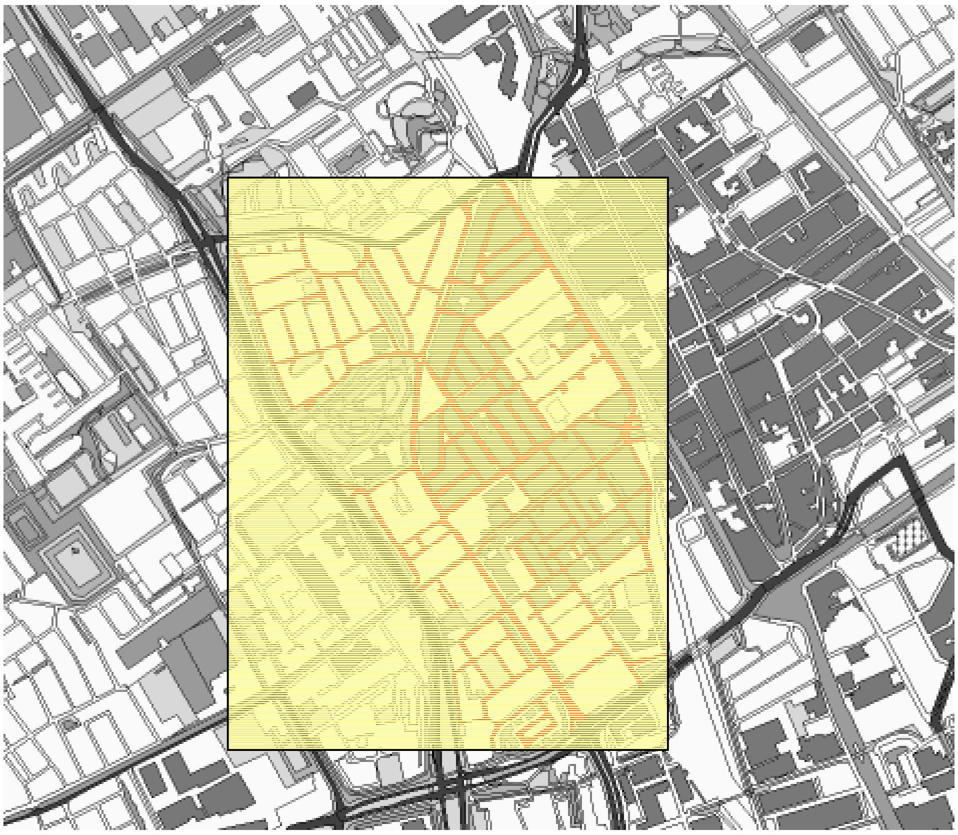
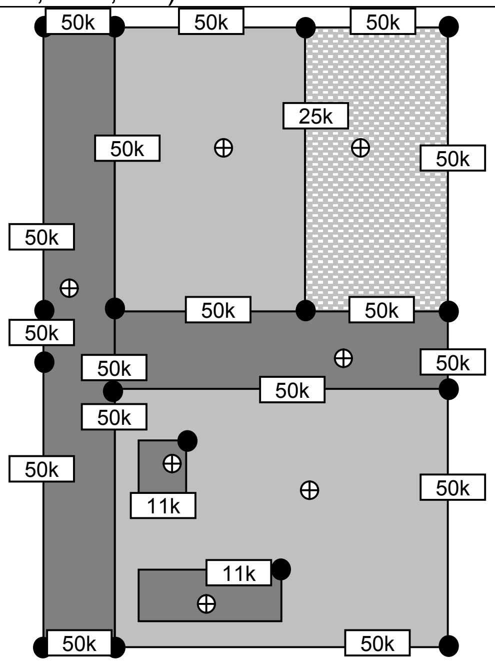
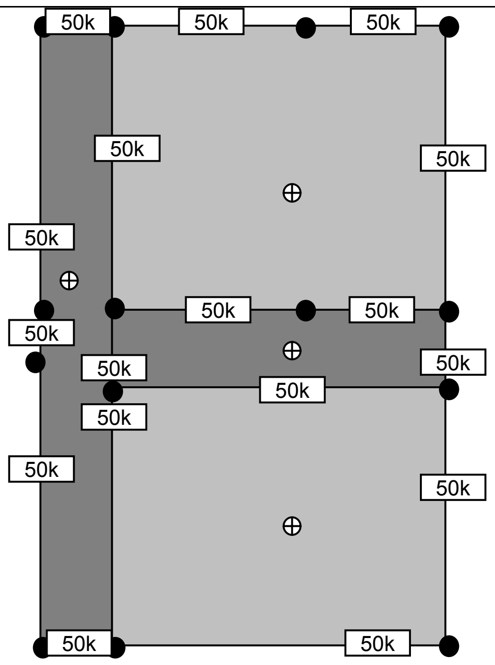


Figure 4.3: Example of a large and complex road polygon (red) and its bounding box (yellow).

edges depends on the requested LoD. and also on the shape and relative size of the polygons at any level of detail. The test data presented some problems in this field, because it contained a number of very large and complicated polygons. These large polygons were mainly road features. This resulted in the fact that a request for a detailed map of any part of the map, would require the transfer of a very large number of edges, since all edges within the bounding box of this road would be returned, even if it's bounding box only slightly overlaps the query window. This makes all the effort that has been done to reduce the data stream and server load undone. This problem is visualized in figure 4.3, which shows a single road polygon in red, and it's bounding box in transparent yellow. Should a request come in for a map at the original scale and the query window overlaps the yellow rectangle in any way, than all edges present within this query window would have to be returned. This problem was severe enough to look for another solution. Which is presented in the next paragraph.

Map with generalization information(Original level of detail, 1:10,000)	Generalized map for scale 1:45,000
 50k 50k 50k 25k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 11k 50k 11k 50k 50k	 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k 50k

Legend

50k ScaleFactor value corresponding to edge

● Calculated node

⊕ Face centroid (only centroids valid for the given level of detail are shown)

Despite these problems this data structure offers a very interesting feature, which might be

Figure 4.4 Linking polygons to face records using centroids.

exploited, should the data allow queries with little overhead. Instead of using the face references in the edges to indicate which faces lie on its left and right side and thus relate the faces to the polygons created at the client-side, centroids can be used to indicate this link. Whereby a centroid is a point that lies per definition within the boundaries of a face. The use of centroids allows the complete absence of any other data than geometric and scale information in the edge table. The advantage of that fact is that it enables the easy and extensive use of geometric generalization algorithms on the individual edges and possibly on combinations of edges. Although attention must than be paid to the appropriate locations of the centroid to avoid mismatches with the polygon as a result of geometric generalization of the boundary of a face. A face can then be assigned to a polygon by checking in which polygon it is located. A drawback of that procedure could however be the computing time required to perform the check for all faces. The usage of these cetroids for faces assignements can been seen in figure 4.4.

4.5. Combined tree and bounding box data structure

In this paragraph the final data structure is described, which provides acceptable results with regards to the selection and the amount of data. Actually the amount of geometric data can be even less then that in the original face data. This method is closely related to the first data structure presented, but solves the problem of the incomplete polygons in a different way than using bounding boxes. As in the distinct-edge-face coding procedure, the edges contain references to their left and right neighbouring faces. The problem was that the correct face reference could not be determined if not all edges were returned.
To counter this problem all edges will store the most detailed face references. Besides that, all faces will contain a reference to their parent face, which is the face that is created when two faces are merged. This parent-child relationship induces a tree structure in the face table, as can be seen in figure. Since the tree structure is oriented bottom-up, i.e. the children point to their parents instead of vice versa, there is no maximum number of children per parent.
The correct face for a certain level of detail can now be found by repeatedly following the link from a face to the parent face until the current level of detail applies to the found face. In generalized maps the edges surrounding a polygon will have different face references. To find the correct face record connected to the reconstructed polygon the face reference of a single edge is used. Which edge is chosen to start the search for the correct face, is arbitrary, since they all should lead to the correct face. Using this face-reference the tree-structure is traversed up until the encountered face satisfies the LoD requirements. The unique identifier of this face is than stored with the reconstructed polygon.

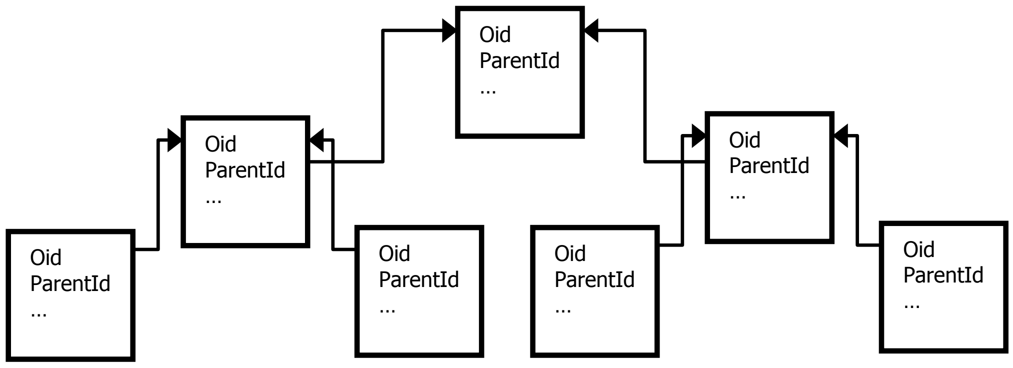


Figure 4.5. Hierarchy of faces through parent references. The ellipsis represents all additional information on the faces, e.g. thematical data.

This data structure looks very similar to the GAP-tree approach, but there are many differences. The first and primary is the fact that the geometry is not stored within the faces, but in a separate table. As a result of this the boundary between two faces is not stored twice, but only once. The boundaries are represented by objects called edges. These edges contain the geometric shape of the boundary as well as references to the left and right faces. This type of information is called topological information. The data structure can therefore be regarded as a topological data structure. The next difference is a direct result of this topological setup. This is difference is that, whilst “Often the boundaries of the areas in the GAP-tree are indeed non-redundant.” [8], the boundaries in the data structure presented here are by definition non-redundant. It is therefore possible to apply line simplification on these separate lines using for example the BLG-tree algorithm and data structure. Drawback here however is that the individual edges are relatively small and therefore benefit only slightly from this simplification. A solution to this problem would be to use some sort of hybrid solution between a scale-less database and a multi-scale data set, but more on that topic can be found in paragraph 5.4.

To be able to select the faces, without having to store its geometry, a bounding box is stored for each face. Efficient storage of a bounding box only requires two points per box, e.g. the upper-left and the lower-right corner.
Another difference with the GAP-tree data structure is that in that data structure the parent child relation was primarily top down. In other words, a parent record contains references to its child objects. In the newly developed data structure this relationship is bottom up. Each child contains a reference to its parent record. Since the relations are all from child to parent, there is no limit to the number of children that each parent has. Thus the generalization algorithm is allowed to merge more then two children into one parent.

At the client-side, the polygons are reconstructed out of the faces using the above-described method. This results in a planar partition. This is another difference with the GAP-tree method. The GAP-tree does not present a planar partition, but instead requires the polygons to be drawn in a certain order to create the appearance of a planar partition. It must be said that with some post processing at the client-side, it would also be possible to create a planar partition out of the polygons returned using the GAP-tree method.

4.6. Issues regarding all procedures

This paragraph lists some issues related to all the above-presented procedure.

Different detail levels present in one map
Normally a map should have a homogeneous level of detail over the entire map face. Which means that the geometric, as well as the thematic detail should be the same everywhere on the map. The geometric issue is addressed by the generalization method, however there can be inconsistencies in the thematic descriptions. It is not wanted that in one part of the map a distinction is made between coniferous wood and deciduous wood, whilst in an other part of the map only the general theme wood is used. To overcome this problem, either the client must know how the theme hierarchy is build up, or some processing on this part has to be performed at the server-side, at query time. Another way would be to reduce the amount of themes directly at the beginning, but this would cause a discrepancy between the thematic and geometric detail when zoomed in. This takes away some of the intentions of having a scale-less spatial database.

Large number of edges in small-scale maps
All of the approaches presented below are based upon the use of the original edges for all levels of detail. As a result of this the number of edges present in a small-scale map will be very large, even if a few number of (large) faces is present. Consequently there is still a quite extensive amount of data transfer to the client. It is possible to reduce the number of points that are used per edge using for example the BLG-tree, but this doesn't reduce the number of edges. Since there are at least two points per edge the number of points in the polygon can still be very large. A reduction of this number of points should be achieved at query time and can only be achieved by joining several edges together. This is not possible using these data structures since it is not known at the server-side which nodes contain only two edges, and thus would be a candidate to omitting in a response to a client's request. Techniques likes the dynamic joining of BLG-tree, as described in [7] can therefore not be directly applied on the edges in the data structures developed in this research. A reduction in the number of edges can only be achieved using more elaborate techniques which are shortly described in paragraph 6.6.

4.7. Client-side routines

The setup of the data structure that was developed allows the extensive use of client-side computing power. The presence of this computing power is valid assumption if relatively new computer systems are used as clients. This choice is not entirely obvious since there have always been waves throughout the history of computer science between the use of thick and thin clients. The miniaturization of computers and the tremendous increase in power combined with the increase of power-to-value ratio has made the client-side computers very powerful indeed. These computers are very well capable of running thick-client applications. This allows a reduction in the computing power needed at the server-side of the system, thus allowing more clients to be handled by the same server. An opposite trend is also present which advocates the use of thin clients. Arguments in favour of the reduction of client-side intelligence or complexity are mainly based on the subject of centralized maintenance and system administration. By reducing the intelligence needed at the client-side, there is also less need for regular support and updating of client-side applications. This can greatly reduce the overall cost of the entire system. However, most applications in client-server environments are primarily used for the processing of standard administrative data, which usually requires less computing power. GIS-data or spatial data in general is often used in conjunction with complex time consuming operations especially when compared to administrative tasks. The use of a thick client that executes some procedures locally is therefore justifiable. In this research the client-side processing power is used to reconstruct the topological data to a usable format at the client-side. The reconstructed data can then be used for further analysis or presentation.

5. Implementation of the data structure

This chapter presents details on the implementation of the data structure and associated routines as well as an analysis of this implementation and of the theoretical design of the data structure. The final data structure as presented in chapter 4 was subjected to testing, to attain information with regards to the usability and performance. For this testing it was necessary to actually perform some kind of generalization. Since the data structure was not intended to be used only in combination with one specific generalization method, the implementation used merely gives an indication as to the possibilities of this data structure. Due to limitations in time a relatively simple generalization method was chosen.

5.1. Implementation of a generalization Method

To implement the designed methods, the programming languages Java and PL/SQL were used in combination with an Oracle 9i spatial DBMS. The Java language is especially useful when applications need to be run on a number of different target systems. This would be the case for the client-side functionality. This part of the procedures was therefore written in Java. Another specific reason for this choice was the presence of the QuickGIS viewer application, which was written in Java and the source code of which was available. This made it possible to extend this application to test the client-side part of the procedures. PL/SQL, or Procedural Language/Structured Query Language is a procedural language extension to standard SQL that can be used to create stored procedures and functions in the Oracle DBMS. The Oracle DBMS also allows the use of Java to create these stored procedures, but it was found that the performance of these procedures was less then similar implementations in PL/SQL. Part of this difference in performance is related to the fact that PL/SQL uses the same data types as the DBMS itself and there is therefore no need for conversion of these types. On the other hand when Oracle data types are used in Java, there is a need for conversion, which can be quite extensive, especially when handling complex data types as are for example used for storing spatial data. Java however has the benefit of the presence of many open-source libraries that contain useful functions for virtually any type of applications. One such library is the Java Topology Suite [15], which was used in client-side routines in the research.

5.2. Server-side routines

At the server-side two different processes have to be performed. First the generalization information has to be gathered and stored using the specified data structure. Secondly a routine must be implemented that allows the server to respond to a client's request by sending the necessary information. Both of these distinct routines will be discussed in this paragraph.

All server-side routines are performed within the DBMS and use either standard DBMS functionality or stored procedures and functions. The stored procedures and functions were all written using the PL/SQL language.

Filling of the data structure

The filling of the data structure consists of several stages. First the input data must be set up to meet the requirements of the generalization routines. In this research the input data consists of TOP10Vector data from the TDN [20]. After a conversion process this data was available in database tables. Two tables were used. The first one contained explicit representations of the faces, containing both thematic and geometric data on the faces. This table was extended with a column storing the 'importance' of the face. The output column for the faces, or the table that will be used at query-time, contains columns for the upper and lower range level of detail values, and also a 2D bounding box. There is also a column, which explicitly stores the 3D bounding box which is the combination of the two. This later is primarily used to increase the speed of indexing the 3D bounding box. This does introduce some redundancy in the tables. To avoid this redundancy, the separate fields that make up the 3D bounding box can be removed after the correct setting of the values for the box field itself. The fields that can be removed are printed in grey in table 5.1. Table 5.1 list all the columns present in these two tables.

In the output table for the faces the geometry is not stored, this is done in a separate edge table. The input table for this is a table containing all the edges in the original map. An edge is present for each line that separates two faces. Besides it's geometric shape each edge record contains a reference to it's bounding faces, using a unique object id. For the input data all edges surrounding a face should have the same face reference at that side. The output table for the edges is almost identical to the input table, with the addition of the upper and lower level of detail and a 3D bounding box. The exact layout of these two tables is given in table 5.1. To enable quick selection of records, the input tables have a number of indexes. It is important to have indexes on at least the LFACE and RFACE fields of the edge table and furthermore on the oid fields of both tables. Also the importance value of the face table is indexed, for fast selection of the face with the smallest importance.

```
GenEdges
Name      Type
----------------------------
OID        NUMBER(11)
GEOMETRY  MDSYS.SDO_GEOMETRY
LENGTH    NUMBER(9,3)
LFACE     NUMBER(11)
RFACE     NUMBER(11)
LRANGE    NUMBER(11)
URANGE    NUMBER(11)
BBOX3D    MDSYS.SDO_GEOMETRY
MINX      NUMBER(9,3)
MINY      NUMBER(9,3)
MAXX      NUMBER(9,3)
MAXY      NUMBER(9,3)
```

```
GenFaces
Name       Type
------------------------------
OID        NUMBER(11)
PARENTID   NUMBER(11)
GEOMETRY   MDSYS.SDO_GEOMETRY
AREA       NUMBER(15,3)
IMPORTANCE NUMBER(15,3)
LRANGE     NUMBER(11)
URANGE     NUMBER(11)
BBOX3D     MDSYS.SDO_GEOMETRY
MINX       NUMBER(9,3)
MINY       NUMBER(9,3)
MAXX       NUMBER(9,3)
MAXY       NUMBER(9,3)
```

Table 5.1: Table definition for edge table (GenEdges) and face table (GenFaces)

Besides these tables 2 lookup tables are needed for the computation of the importance of a face and the compatibility of two faces. The importance of a face is a function of its area and the priority of theme of the face with the current generalization process. The compatibility is a function of the length of the common boundary of two face and the compatibility between their themes. The compatibility factors can be stored in a matrix, which in this case is stored in a database table. The definition of these two tables are listed in table 5.2.

```
Themepriority                    Compmatrix
Name           Type              Name            Type
------------------------------   -------------------------
THEME          NUMBER(5)         THEME1          NUMBER(5)
PRIORITY       NUMBER(6,3)       THEME2          NUMBER(5)
                                 COMPATIBILITY   NUMBER(6,3)
```

Table 5.2: Table definition for Themepriority table and compatibility matrix.

The content of the priority matrix and the compatibility matrix greatly influences the way a dataset is generalized. For the result of the generalization to have any value the content of these matrices have to be chosen very carefully. In practice this will main some iterations in which the values are varied until an acceptable result is found. Despite of the care that is put into this search for optimal parameters, the result is always subject to the very limited possibilities in generalization that is offered by this method. Serious applications of the data structure will likely require more sophisticated generalization procedures. Limitations in time however prevented the use of far more advanced methods.

With all input, output and auxiliary tables ready, the generalization process can start. This process was implemented through a stored procedure, the exact listings of which are given in appendix A. Here the process is described step by step. There might be small difference between the code listed here and the actual code in the appendix. These difference are due to the fact that the code here is meant to clarify the procedure. The presented code is written in PL/SQL, which uses -- to indicate comments and allows the use of SQL statements in the code. Here comments are additionally highlighted through the use of italic typesetting.

1. Main Loop
The generalization process is mainly contained within one large loop which assures that the generalization is continued until a specified condition is met, e.g. minimum LoD, or maximum total number of faces.

```
--Startup of the loop
--select the face with the smallest importance for removal into aface
--merging with a bounding face
select genfaces.*
into aface
from genfaces
where genfaces.oid =
(
  select min(oid)
  from genfaces
  where genfaces.importance=minimp
);

--The main loop
--The loop should be continued until a terminating condition is
--met.
--Conditions could be:
--  Minimum encountered importance is larger than a specified
--  maximum
--  There is only one face left so no further processing is
--  possible
--  A maximum number of iterations has been performed

--Start of mainloop
while (aface.area<maximp)loop
  --This is where the main generalization is performed

  ***INSIDE MAINLOOP***

  --Select face with smallest importance for next iteration
  select genfaces.*
  into aface
  from genfaces
  where genfaces.oid =
  (
    --since it is possible that multiple faces have the same (minimum)
    --importance, we take from those the one with the smallest oid,
    --so garanteed only on face is returned
    select min(oid)
    from genfaces
    where genfaces.importance=minimp
  );
```

```
end loop mainloop;
```

2. Inside the Main Loop

For each run through the generalization loop, one face (aface) is merged with one of it's neighboring faces (cface). This merging is actually performed by removing the merging faces from the input table and adding them to the output table. A new face is created that represents the union of these two faces and this new face is added to the input table. No geometric union is performed, instead the edge, or possibly edges, that separates the two faces is added to the output edge output table. Faces and edges that are removed have their upper bound level of detail value set to the importance of the face that was selected for removal. Although in the code the term scale is used, it is actually the importance value that is used. The reason for this is explained in the next paragraph.

```
--select neighbor with highest collapse value
maxcolval:=0;
--open cursor with all neighboring faces
--calculate collapse value for all neighboring faces
--and select face with highest collapse value into cface
for bface in neighbors(aface.oid) loop
  --calculate collapse value for all neighboring faces
  bndlength:=0;
  --calculate length of common boundary aface,bface
  for aedge in bndedges(aface.oid,bface.oid) loop
    bndlength:=bndlength+aedge.length;
  end loop;
  --calculate collapse value:=bndlength*compatibility
  select compatibility
  into colval
  from compmatrix
  where theme1=aface.theme and theme2=bface.theme;

  colval:=bndlength*colval;
  if colval>maxcolval
  then
    maxcolval:=colval;
    cface:=bface;
  end if;
end loop;

scale:=minimp;
--now we know which two faces need to be collapsed
--the collapse is performed by:
--setting the URange value of the collapsing faces to the current scale
--setting the URange value of the Edge(s) separating the two faces, to
--the current scale
aface.urange:=scale;
cface.urange:=scale;
for aedge in bndedges(aface.oid,cface.oid) loop
  aedge.urange:=scale;
  delete from genedges
  where oid=aedge.oid;
  insert into genedgesout(oid,geometry,length,lface,rface,lrange,urange)
  values(aedge.oid,aedge.geometry,aedge.length,aedge.lface,aedge.rface,
  aedge.lrange,aedge.urange);
end loop;

--A new face(uface) is created consisting of the entire set of
--boundaries of the two collapsing faces (including islands)
--excluding the common boundaries
```

```
--A number of properties of the new face need to be set
uface.oid:=1000001+teller;
--assigne the new face the theme of the more important face
uface.theme:=cface.theme;
uface.lrange:=scale;
uface.urange:=0;
uface.area:=aface.area+cface.area;
--update the bounding box (the bounding box of the union is the
--bounding box of the two individual bboxes)
uface.minx:=minn(aface.minx,cface.minx);
uface.miny:=minn(aface.miny,cface.miny);
uface.maxx:=maxn(aface.maxx,cface.maxx);
uface.maxy:=maxn(aface.maxy,cface.maxy);

--the the edges that now bound uface must
--have that faceid set to the face id of uface
--since the edge between aface and uface has already been deleted
--that leaves all remaining edges that have a reference to aface or
--cface;
update genedges
set lface =uface.oid
where lface=aface.oid or lface=cface.oid;

update genedges
set rface =uface.oid
where rface=aface.oid or rface=cface.oid;

--The new uface must have its importance set to the correct value
--to allow selection for collapsing later on the generalization process
select priority
into tempnumber
from themepriority tp
where uface.theme=tp.theme;

uface.importance:=uface.area*tempnumber;

--Insert the new face into the input table so
--it will be processed in following loops
insert into genfaces(oid, theme, lrange, urange, area, importance,
  minx, miny, maxx, maxy, centroid)
values(uface.oid, uface.theme, uface.lrange, uface.urange,
  uface.area, uface.importance,    uface.minx, uface.miny,
  uface.maxx,uface.maxy, uface.centroid);

--copy inputfaces (aface,cface) to output table
insert into
genfacesout(oid,parentid,theme,area,importance,lrange,urange,centroid,minx,
  miny,maxx,maxy)
values(aface.oid,uface.oid,aface.theme,aface.area,aface.importance,
  aface.lrange,aface.urange,aface.centroid,aface.minx,aface.miny,
  aface.maxx,aface.maxy);

insert into genfacesout(oid,parentid,theme,area,importance,lrange,urange,
  centroid,minx,miny,maxx,maxy)
values(cface.oid,uface.oid,cface.theme,cface.area,cface.importance,
  cface.lrange,cface.urange,cface.centroid,cface.minx,cface.miny,
  cface.maxx,cface.maxy);

--delete inputfaces from source tables so they won’t be selected in
--following loops
delete from genfaces
```

```
where oid=aface.oid;

delete from genfaces
where oid=cface.oid;
```

3. After the main loop

Once the main loop terminates there is a small number of operations that need to be performed to conclude the gathering of generalization information. The input face table will contain at least one, but depending on the specific loop terminating condition more, faces which need to be copied to the output table. Their upper bound level of detail value needs to be set to some maximum value to make sure they are selected at the coarsest level of detail.

```
--Update urange value for remaining face and edges
update genfaces set urange=maxscale
update genedges set urange=maxscale

--copy remaining faces in genfaces to genfacesout
insert                                                              into
genfacesout(oid,theme,area,importance,lrange,urange,centroid,minx,miny,maxx
,maxy)                                                            select
oid,theme,area,importance,lrange,urange,centroid,minx,miny,maxx,maxy    from
genfaces;

--copy remaining edges in genedges to genedgesout
insert    into    genedgesout(oid,geometry,length,lface,rface,lrange,urange)
select oid,geometry,length,lface,rface,lrange,urange from genedges;
```

This concludes the main portion of the generalization process. The output tables require some post processing however to be ready for querying. This post processing includes, the construction of 3D bounding boxes based upon the values for the 2D bounding box and the lrange and urange columns for both the edges and the faces. Furthermore indices need to be created on certain fields, most notably the newly create 3D bounding boxes. The code for this post processing is provided in the appendix, but does not contain any special techniques. Figure 5.1 gives a simple schematic overview of the filling of the data structure.

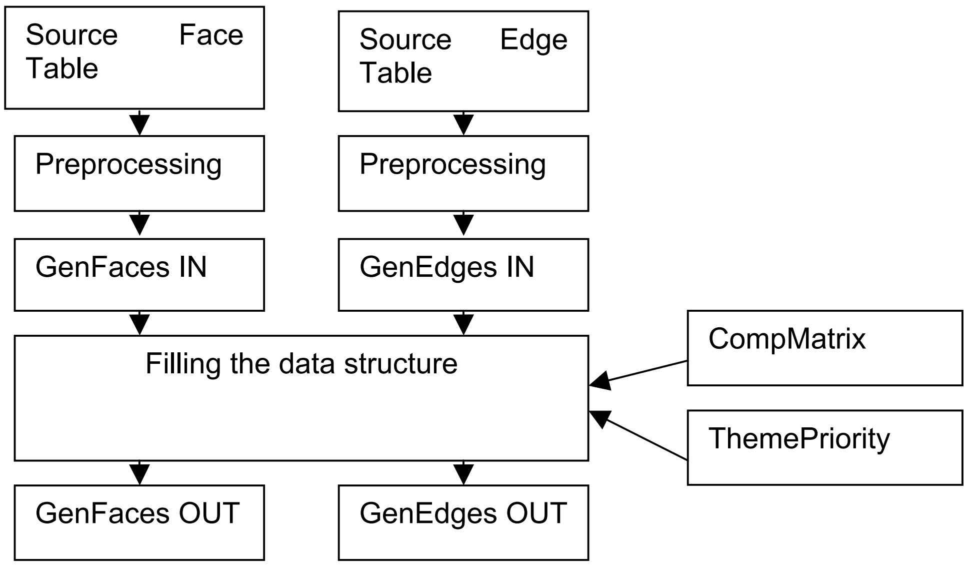


Figure 5.1: Schematic overview of procedure for filling the data structure

Importance value and scale-factor
The generalization information gathering procedure makes use of a parameter called importance factor. The value of this parameter is calculated as the product of the theme priority values as listed in the theme priority table and the area of a face. Appropriate values for the priorities are an important aspect in making the generalization produce a sensible result. The values used in the research are based upon values presented in [9]. These values range from 2 to 400. Although the use of carefully selected values for these parameters can result in the correct sequence of the selection of faces for merging, there is no clear relation between this importance value and a scale-factor [9]. It is likely that this relationship will vary per combination of data set and chosen theme priority values. One way to establish a proper relationship is to manually evaluate the visualisations of the data sets at various zoom levels, in combination with values for this importance. This evaluation would then result in a set of scale, importance combinations. These could then be used to interpolate any importance value given a certain scale. The evaluation would only have to be performed once, after the data structure has been created for a certain data set with a set of priority values. In the testing environment the importance value has to be entered directly into the field for the LoD. In any real applications this should be implemented a proper relationship between the LoD and the importance value, and optionally this LoD should be determined automatically depending on the on screen scale of the visualised map.

Response to client requests
Besides the gathering of the generalization information the server also is used at query time to provide the data to the client. In this process it receives a request from a client in the form of SQL queries. In the current setup the client needs to perform three separate queries to obtain all necessary thematic and geometric information to create a consistent map display. All of these queries are 3D spatial queries that in a single query specify both the spatial extent as well as the required level of detail. These queries use the Oracle sdo_filter function to enable the server to use its spatial index on the 3D bounding box to allow fast and efficient selection of the appropriate records. The setup of the de data structure does not impose a strict order in the various queries. But all are necessary to obtain enough information for proper visualisation.
The first query is intended to retrieve the thematic information from the face. This is a simple query and no processing of the output data is required at the server-side. The second query for the edges does require additional processing at the server-side. Here the face references need to be changed to refer to faces that actually fit the requirement for the level of detail requested. For this purpose a function was written in PL/SQL, which is listed here.

```
CREATE OR REPLACE FUNCTION GETRIGHTFACEID(startid integer, scale float)
RETURN number

--This function returns the faceid of the face that
--can be found by following a face's parentid until the
--right scale level has been found
IS
  aface genfacesout%rowtype;
BEGIN
  select genfacesout.* into aface from genfacesout where oid=startid;
  while aface.urange<scale loop
    select genfacesout.*
    into aface
    from genfacesout
    where oid=aface.parentid;
  end loop;
  return aface.oid;
END GETRIGHTFACEID;
/
```

This function is applied to both the left and right face-reference of each edge that is selected and transferred to the client. This does provide an additional workload at the server-side. In fact this routine took so long that updating of the face-references was transformed to a client-side process. The server-side update routine was therefore replaced by a third client query for the necessary update information, and subsequent updating of the necessary face-references at the client-side. At the client-side only one reference update per polygon is needed instead of one per edge at the server-side. Although not used in the current implementation the above listed routine, or a variant, can be used if calculation is shifted more to the server-side.

Use of Indices during interactive use
For efficient and quick selection of faces satisfying the conditions in a query indexes can be used. In this research a 3D R-tree index was used to index both the faces as well as the edges. The 3D R-tree index is a built in indexing method of Oracle spatial. In fact it is even possible to create a 4D R-tree index using only functionality readily available in Oracle spatial. The data used in the research is intrinsically 2D. The third dimension used in was intended to be the scalefactor as described in chapter 2. However, as was also stated there, the generalisation process used, uses an importance value which can not easily be transformed into a scalefactor. Instead the scalefactor, the importance value itself was used for the third dimension. This range of values in this dimension can be come very large, as it is the result of a priority value and an area. The more generalization is applied, the larger the area of the individual faces that remain will be, and thus the larger the importance value. Should the indexing method require similar value ranges in all dimensions, a linear scaling of the importance value can easily be applied, without changing' the results of queries.

5.3. Client-side routines

As stated before, the implementation of on-the-fly generalization using the data structure presented in chapter 4 requires considerable intelligence at the client. Here the implementation of the client-side routines is discussed, including the construction of the necessary database queries. In short the client-side routines can be split into the following main steps each of which will be explained more detailed:

1 Specify query window
2 Determine LoD
3 a) 1 Get edges using 3D query
      2 Intersect edges with query window
      3 Reconstruct faces
   b) Get face-reference update tree using 3D query
   c) Get faces using 3D query
4 Assign face records to reconstructed faces
5 Display faces.

All steps must be executed in the order in which they are numbered, but steps 3a, 3b and 3c do not have any specific order and can even be performed parallel.
The client-side routines were implemented in an existing GIS-viewer. The viewer used is called QuickGIS and was written in Java. Because the source code of this viewer was available, as well as insight into the exact workings of the application, since it was written by Wilko Quak, one of my supervisors, this viewer could rather easily be extended to incorporate the above listed operations.

The description of the routines will be in the order as listed above. The code however is organised in classes that are used throughout this process. Therefore first an overview is given of the classes needed in the reconstruction process. The interesting parts of the actual code itself can be found in appendix C.

Some parts of the implementation make use of a freely available library of objects that facilitates the use of geometric and topological data. This library is called the Java Topology Suite or JTS. The JTS library provides classes for geometric and also topologic objects and also a multitude of functions that can be applied to those objects. The routines at the client-side routines use both JTS objects or sub-classes thereof, such as the PolygonX class, as well as self-made objects that also have a JTS counterpart, e.g. the Node class. The JTS library was used for two reasons, first because of the presence of several very useful functions, such as a the check whether an object lies within another object, and secondly to allow future applications easy access to the final map data. For the topological elements such as nodes and edges self-made classes were created, to precisely control the way they work, and thus to properly implement the client-side routines for correct use of the data structure. In appendix D a class diagram of the below listed classes is available.

Node
This class implements a node. The important properties of a node are it's location and the sorted list of the outgoing edges. The node class was therefore inherited from the TreeSet class, which stores are list of objects in a tree structure. For the ordering in the tree set the angle of the first line segment of each edge is used. Due to this tree structure, the next outgoing edge, given a certain incoming angle can efficiently be selected.

NodeList
A nodelist contains a list all nodes within the total client side data set. It is intended to be used in combination with the below mentioned EdgeList. It implements functionality to create to create new nodes, or update existing nodes, based upon new Edges. For that purpose it implements an AddEdge procedure which checks whether there is already node at the location of the start of the edge. If there is such a node, then the edge will be added to the node's list of edges, otherwise a new node is created at that location and the edges is added to that node's list.

Edge
An edge is primarily a geometric description of (part of) the boundary of a face. To this end it stores the shape of this boundary using an array of double for the x and y coordinates. It also contains references to the faces on the left and on the right side. This reference has been implemented by an integer value, which contains the value of the unique identifier a face.

EdgeList
The total set of edges that are used for a polygon reconstruction session is stored in an object of the class EdgeList. This class maintains a list of all the edges. It is contains functionality to create a nodelist out based upon the present edges.

LinearRingX
The LinearRingx class is inherited from the JTS LinearRing class [15]. This subclass only adds a number of administrative properties such as a unique identifier and a scalefactor. It also stores its area and its orientation, to remove the need to calculate it explicitly every time it is needed.

PolygonX
A PolygonX is the polygon equivalent of the LinearRingX. It is derived from the JTS Polygon type and additionally has some properties to store a scalefactor, a unique identifier and a reference to the face it represents. A PolygonX has an outer shell which is a counter clockwise oriented ring and zero or more holes, which are clockwise oriented rings.

LinearRingXList

This list of LinearRingX's has the primary function to create PolygonX's based upon the rings it stores. PolygonsX's are created by adding the necessary holes to the outer shell. The selection of which holes belong to which PolygonX is handled by the LinearRingXList.

These classes are all used in the client-side process which will now be explained.

Step 1: Specify query window
The query window, is the real world area that is visible on the clients screen. This is usually selected by the user. This in contrast to the map window, which refers to the actual display area of the map on a computer screen or other output medium. The dimensions of the query window are specified in real world coordinates, e.g. using the Dutch national Datum RD. A map window is usually measured in pixels, as this is the coordinate system needed for the computer to display the graphics.

Step 2: Determine LoD
The LoD can either be specified explicitly by the user, or be determined automatically depending on the size of the query window and the map window. For automatic determination some relationship has to be established between the LoD values and the scale of the presentation on the display. The LoD value is used as the Z value in the query.

Step 3a: Get faces using 3D query
The faces have to be requested from the server using an SQL-query. This query uses the 2D extent of the query window, as well as the LoD for the third dimension. For the face query, all faces are selected of which the 3D bounding box is intersect by the 3D rectangle with x,y extent equal to the query window and the z value equal to the desired LoD. The following (simplified) lines of code are used to construct the 3D query for the selection of the faces:

```
Select * from FaceTable
Where mdsys.sdo_filter
(
  DataTable, mdsys.SDO_GEOMETRY
  (
    3003,  srid, NULL, mdsys.SDO_ELEM_INFO_ARRAY(1, 1003, 3),
    mdsys.SDO_ORDINATE_ARRAY(minx ,miny ,lod,maxx,maxy,lod)
   ),
   'mask=ANYINTERACT querytype = WINDOW'
)= 'TRUE'";
```

Listing 5.1: 3D Query for selection of faces

Figure 5.2 shows a schematic visualisation of the concept of the 3D query for the selection of faces. The light blue rectangle represents the 3D rectangle as defined in the query. Each of the boxes represents a 3D box of a face. The red boxes are intersected by the rectangle and are thus selected.

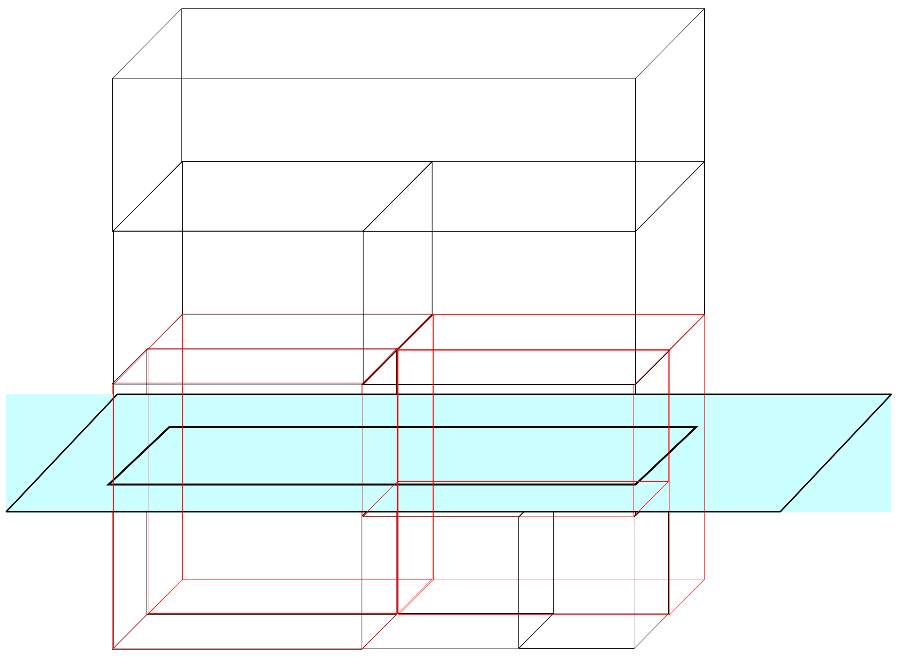


Figure 5.2: Selection of faces by intersection of 3D boxes with a 3D rectangle.

The mdsys.sdo_filter function is able to use, if available, a 3D R-tree index to speed up the selection process. A limitation of this function is that the query window must be an axis-parallel rectangle. This is often the case with queries for visualisation purposes. The complete query string is sent to the server which responds by sending all face records that fit the request. These records are stored in a list in the client application for further processing.

Step 3b: Get Face Reference update tree using 3D query
The face references stored with each edge refer to the edges at the LoD of the original, non-generalized, data. For lower LoD's, these need to be converted to refer to the actual faces present at that LoD. For this conversion a tree structure is needed which stores the relationships of faces, i.e. how faces are merged in the generalization process. This structure is present in the form as seen in figure 4.5. The link between a child and parent is based on a unique integer value for each face, the oid field.

Step 3c: Get edges using 3D query
This query is responsible for obtaining the geometric properties of the polygons that are to be reconstructed. Again a 3D query is used to address both the geometric as well as the LoD demands in a single query. The query that is formulated is therefore analogous to the query used to retrieve the faces.

Step 4: Intersect edges with query window
The polygon reconstruction process requires that edges connect to each other in nodes. This is already the case with edges that lie completely within the query window. However, a number of the selected edges will lie partly outside the query window. These window-intersecting edges, as shown in figure 5.3 a, prevent the polygon reconstruction process from creating a partitioning that covers the entire query window. The effect of which is shown in figure 5.3 b. The problem is solved by intersecting the window-intersecting edges with the query window and thus splitting edges that cross the query window into two parts as seen in figure 5.3 c. Polygon reconstruction using all the edges that are present of the splitting leads to the result shown in figure 5.3 d, which completely covers the query window with a planar partition.

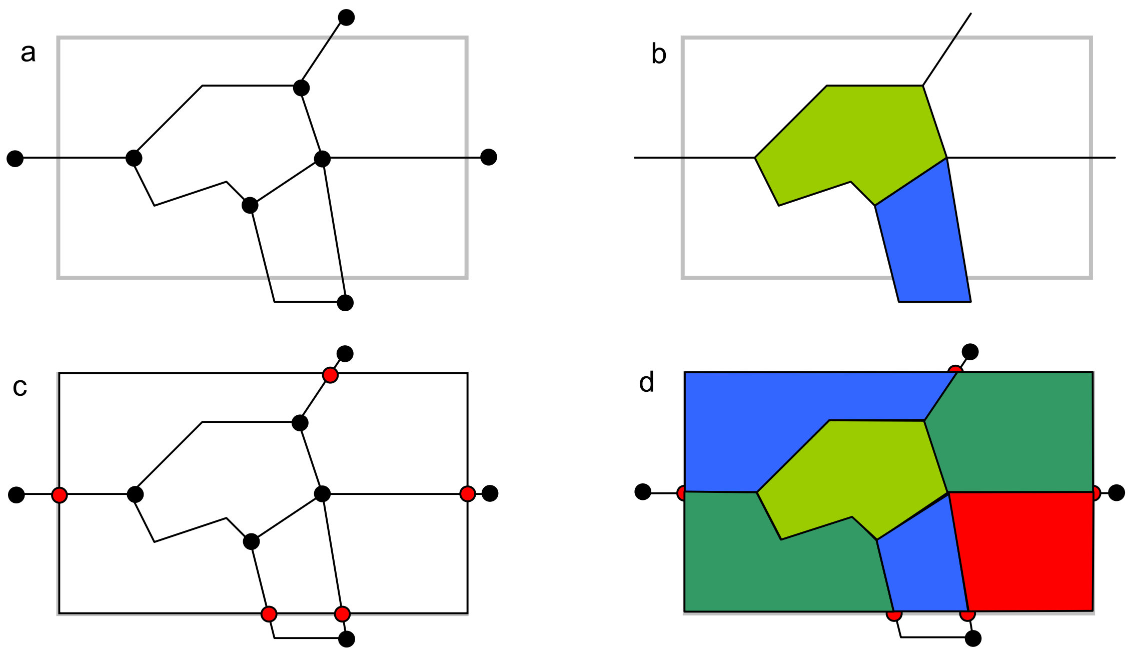


Figure 5.3. a. Edges in a query window, including dangling edges
b. Polygons reconstructed using present edges
c. - Use query window as edge
- Split window intersecting edges at intersection with query window
- Split query window edge at intersection with window intersecting edges
- Add query window edges to edge list
d. Create polygons using all edges in edge list.

Step 5: Reconstruct faces

The term face reconstructing refers to the process of selecting all edges that belong to one polygon feature and creating a single polygon object out of it, including possible inner rings, for display or other purposes. The reconstruction process requires that for every single edge that is returned in the query result a second edge is added with the opposite direction. From that point on each of those edges can be used as a directed edges. Both of the directed edges are stored in the same edge list.

Another prerequisite of the reconstruction process is the availability of a list of all nodes. A node is a location on the map where one or more edges start. Each node contains a list of all edges that start at that location. For polygon reconstruction it is necessary that the edges starting at a certain node be ordered on the angle of the first line segment. The nodes are created by looping through all directed edges in the edge list and for each edge determining if node is already present in the node list. If so, then the edge is added to that nodes own edge list, at the index determined by the angle. If not, then a new node is added to the node list with the first point of that edge as the location for the node.

Once all nodes have been created and assigned the appropriate edges, rings can be created. The reconstruction of a ring is started by arbitrarily taking an unprocessed edge from the edge list. Starting with this edge the next edge in the ring is located by selecting the node at the end of the edge and requesting the next outgoing edge, based on the incoming angle of the present edge. This next edge is appended to the ring under construction. This process is repeated unit the next node encountered is the same as the one at the start of the first edge. Each edge that is processed is marked, so it is only used in the construction of a single ring. This process automatically distinguishes between shells and holes, since all shells will obtain a counter clockwise orientation, whilst all holes will have a clockwise orientation. The process for ring creation was earlier visualised in figure 4.1.

Step 6: Assign face records to reconstructed faces
The reconstructed polygons need to be linked to the records with the face information. For this purpose each edge has been accompanied with references to its left and right faces. These reference are however only valid at the original LoD of the input data, as they al refer to the faces that were present at that level. To update these references the data acquired in step 4 is used. For each polygon only one face reference is needed. Which edge is used to provide this reference is not important, since the tree structure ensures that all references will lead up to the same face. The need for this updating to happen at the client side is discussed on page 41.

Step 7: Display faces.
Once all polygons are created and the correct face information is connected to them, they can be displayed, or used for other analysis purposes, since they are now represented by standard geometric objects. Of course display of the polygons is only one application, it is also possible to use the result for spatial analyses.

The execution of all these client side routines becomes quite an time consuming task, as will be shown in the next chapter, but can also already .

5.4. Conclusion

Using the developed data structure as presented in chapter 4 requires a some routines at both the server and the client side. In this chapter the routines for both side are described as. For the server side this encompasses both routines for the filling of the data structures, as well as routines used in interactive access of the data. The client only comes in to play during interactive use and routines used at that side are also discussed. With the general layout of the data structure known and necessary routines present, the data structure can be tested with real-life data. The next chapter describes the tests that were performed, and discusses the strong and weak points of the data structures, as well as concepts to overcome these weak points.

6. Analysis of the data structure

To analyse the developed data structure, testing was performed on real-life data. This chapter goes into detail how these tests were performed. Based on the test results, as well as earlier observed issues regarding the data structure, the advantages and disadvantages of the data structure are discussed. Also ideas for enhancement of the procedure are presented.

6.1. TOP10Vector

The data structure was tested using data from the Dutch Topographic Service[1] [10]. For the research TOP10vector data was available for the entire area of the Netherlands. Unfortunately however, the data structure never matured enough to process any more than a small area. The test performed applied to an area of 5×6.25 km of mostly urban nature. This area covers half a standard map sheet of the TOP10vector. Figure 6.1 displays all the polygons in the area. For creation of a proper map, the TOP10vector files also contain a number of other objects, including buildings and linear features such as railway tracks. These where however not used in the research and are therefore omitted in the display. The area visualised covers large parts of the city of Delft, with the city centre at the right, centre of the map.

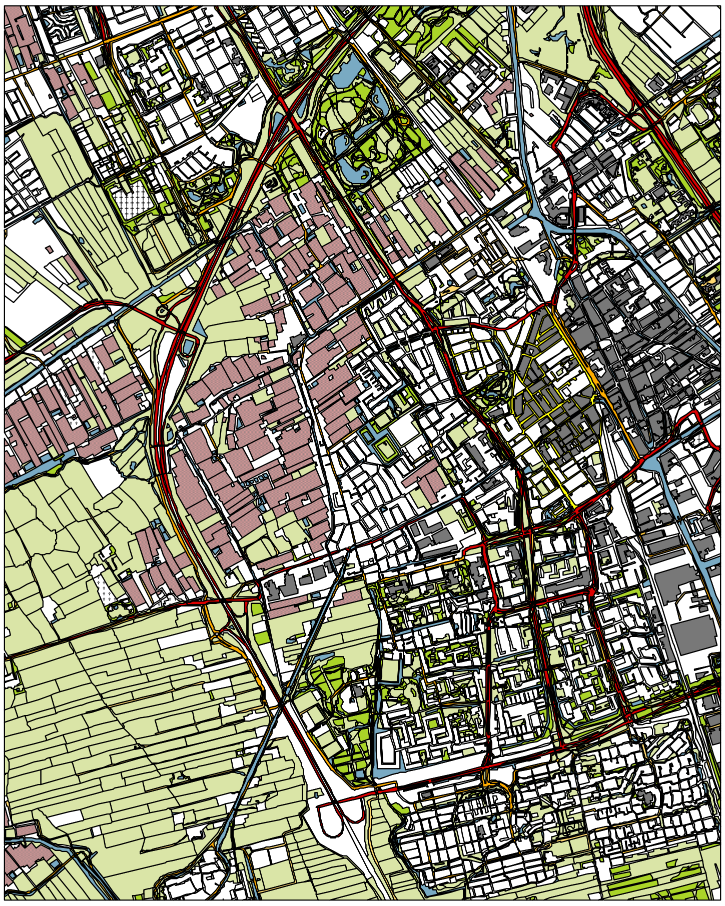


Figure 6.1: Map display of the test area, situated in and around the city of Delft.

[1] Dutch Topographic Service is a translation of Topografische Dienst Nederland (TDN)

6.2. Pre-processing

To be able to apply the generalization procedure on the TOP10vector data and create the data structure, the data needed to be stored in database tables. Out of the original data two tables were created, a table with polygons describing a planar partition, and a table with polylines which are the boundaries of the polygons. Table 6.1 shows the table definitions as they were used. Most of the attribute names are self-explanatory, but some requires an explanation. The c codes are so-called TDN-codes, these are numbers that indicate the type of objects, e.g. 3000 to 3999 are roads or related objects. The generalization process uses a priority value to calculate the importance of an area object. The values used for this priority were taken from [9], and are shown in table 6.2. As can be seen from that table the number of different themes that has been used in the generalization process is rather small. The input data had a much larger variety of themes, and also allows that use of up to 10 codes simultaneously. For this research only the c1 value was used, which is the primary indicator. The reduction of the number of different themes has two reasons, first of all it reduces the size of the priority table and the compatibility matrix, and secondly, the choice of these values greatly influence the generalization order and should therefore be made very carefully. Since the generalization itself was not the primary subject of this research, a small number of themes sufficed, as long as the process worked. The reduction of the number of different themes was achieved by assigning similar themes the exact same theme. For this purpose the following SQL statements were run against the inputtables:

```
--urban
update genfaces set theme=1013 where theme>=1000 and theme<2000;
--Main Roads
update genfaces set theme=2203 where theme>=2000 and theme<3000;
--Roads
update genfaces set theme=3533 where theme>=3000 and theme<4000;
--Rural
update genfaces set theme=5213 where theme>=5000 and theme<6000;
--Hydro
update genfaces set theme=6113 where theme>=6000 and theme<7000;
```

Listing 6.1: SQL code to reduce number of different themes.

The TDN-coding system was fortunately set up such that similar themes have code values from the same interval.
Table 6.3 shows the values used in the compatibility matrix these values are also based on values in [9].

```
top10vlakken                     |  top10lijnen
Name         Type                |   Name                 Type
---------------------------------|---------------------------------
GEOM         MDSYS.SDO_GEOMETRY  |   GEOM       MDSYS.SDO_GEOMETRY
AREA         FLOAT(126)          |   FNODE      NUMBER(38)
PERIMETER    FLOAT(126)          |   TNODE      NUMBER(38)
VLAKKEN_#    NUMBER(38)          |   LPOLY      NUMBER(38)
VLAKKEN_ID   NUMBER(38)          |   RPOLY      NUMBER(38)
IGDS_GGNO    NUMBER(38)          |   LENGTH     FLOAT(126)
C1           NUMBER(38)          |   LIJNEN_#   NUMBER(38)
C2           NUMBER(38)          |   LIJNEN_ID  NUMBER(38)
C3           NUMBER(38)          |   IGDS_GGNO  NUMBER(38)
C4           NUMBER(38)          |   C1         NUMBER(38)
C5           NUMBER(38)          |   C2         NUMBER(38)
C6           NUMBER(38)          |   C3         NUMBER(38)
C7           NUMBER(38)          |   C4         NUMBER(38)
C8           NUMBER(38)          |   C5         NUMBER(38)
C9           NUMBER(38)          |   C6         NUMBER(38)
C10          NUMBER(38)          |   C7         NUMBER(38)
SYMBOL       NUMBER(38)          |   C8         NUMBER(38)
                                 |   C9         NUMBER(38)
                                 |   C10        NUMBER(38)
                                 |   SYMBOL     NUMBER(38)
```

Table 6.1. Table definitions of tables for polygons and polylines from TOP10vector

```
     THEME |  PRIORITY
-----------|----------
      1013 |       400
      2203 |        80
      3533 |        75
      5213 |         5
      6113 |        10
```

Table 6.2. Content of theme priority table

Theme	**1013**	**2203**	**3533**	**5213**	**6113**
1013	1	0.003	0.005	0.9	0.1
2203	0.2	1	0.9	2	0.1
3533	0.2	0.9	1	0.2	0.1
5213	0.7	0.1	0.1	1	0.4
6113	0.005	0.005	0.005	0.6	1

Table 6.3. Content of theme compatibility matrix

During testing the top10vlakken table contained 7204 records and the top10lijnen table contained 22464 records. Using these values and tables, the generalization process as decribed in paragraph 5.1 was performed. The number of records that will be present in the output table can be determined before hand, given the fact that the generalization algorithm does not create any new edges, but instead, just copies the input edges. The number of edges in the faces table is one less then double the number of faces in the input face table.

That number is based on the fact that the generalization algorithm repeatedly creates one new face out of two others until only one large face is left. The number of records in the output tables is therefore 22464 for the edge table and 14407 for the face table. The importance values ranged from to 1.95 to $1.5*10^{10}$. This immediately shows, that without scaling of the importance value, the range of importance values can become very large.

6.3. Performance of the generalization

The filling of the data structure, took approximately 2 hours and 16 minutes for the input tables with the above described content. This is exclusive of the pre-processing actions such as the loading of the data and the reduction of the number of themes. Based on these values an estimation can be made as to the running time required to process all map sheets. There are 675 map sheets covering all of the Netherlands. Simple linear extrapolation of the timing results would yield a total running time of 675*2*2.16 hour = 2916 hour or 121.5 days. This number should be increased a little to also encompass the generalization on top of the individually generalized map sheets, but this would be less then the time necessary to process half a mapsheet as only 675*2=1350 polygons are to be processed from that level on. Still this projected processing time is far too large for serious application using the current implementation of the method. The main cause for this long processing time lays probably in the choice to implement the generalization method within the database as described above. The implementation is capable of handling data sets of virtually any size, as operations on these sets are all handled through standard database queries. This does however present a great performance penalty as for each new face to be generalized, several queries have to be performed on the input data set. Although the tests were not performed on a fast computer and although this generalization process does not have to be performed very often, the current implementation is not very useful for any serious application on very large data amounts. An alternative that for the DBMS based generalization procedure is a special application that performs that is optimized to perform the same tasks. Such an application could for example keep all necessary information within the main memory of the computer to ensure fast access times for all records. After generalization is complete, all results are written to the varies database tables.

Query-time performance

A very important aspect of the entire setup is the query time performance of the data structure. Since the process both involves actions of the server and the client, these will be discussed separately. It was chosen in the setup to move a substantial part of the query time procedure from the server to the client. This choice does impose a greater workload on the client than with most conventional client-server setups. The reason for this choice has been discussed earlier in this thesis. In this chapter the results of this decision for the complexity and performance of the server as well as the client-side are shown.

Query-time Server performance

The tasks of the server at query time do not require any other actions than the simple execution of a number of 3D spatial queries. Since all fields used in the selection process are indexed, these queries are performed efficiently. Not much is lost or can be gained at this stage. Typically any of the spatial queries took from under a second, to two or three seconds, including data transfer via the lan.

Query-time Client performance

A very important part of developed data structure is the query-time performance of the client. Especially since a substantial amount of processing has to be performed at the client. Of the processes, as listed in paragraph 5.1, the intersection of the edges with the query window is by far the most time consuming. This is primarily related to the inherent complexity of the intersection operation in general, as well as to the simple implementation that was chosen. It is possible to implement the intersection using more efficient algorithm, especially when taking into account that the query window only consists of axis parallel edges. One way of utilising this information is to use the R-tree index that is already present in the database.

Instead of requesting all edges of which the bounding box intersects or lies within the query window, first all edges of which the bounding box only intersect with the query window are selected. This far smaller set of edges is then intersected with the query window. After the intersection process, all edges that lie completely within the query window are requested from the database, again using the R-tree index for the selection. These edges are then added to the earlier created list of intersected edges. The polygon reconstruction process is executed precisely as before on this combined list.

To practically test the client-side performance the client was run a pc. This was easily possible since the client-side software was written in Java and could therefore be run on different platforms without any change in the source code, or re-compilation. The important properties of the pc for this research are that it houses a 2Ghz Pentium IV processor together with 512 Mb of memory. The operating system on this computer was Windows NT4. This computer is connected via a LAN to the server computer. Using this computer the queries still took considerable time, from the moment the first requests are sent to the server until the moment the final result is displayed on the clients screen. Using the given computer setup the queries took between 20 seconds up to 60 seconds. This value is far too large for serious use in on-the-fly generalization, which requires fast response times, e.g. 10 seconds. Especially since the given setup is currently an currently an above average system, this implicates that enhancements to the procedure are necessary. The majority of the 20 to 60 seconds is spent calculating the intersections with the query window. Also much time is needed for the polygon reconstruction.

The statistics that are shown concentrate on the data amounts, which is also a very important issue in the development of the data structure. Values will be shown for the real world size of the query window, the number of returned edges and faces, the number of intersections with the query window that were found and the number of polygons that were reconstructed. Also the LoD value that was used is indicated. However, as mentioned before, this value has no relations to an actual scale or the reciprocal value of a scale.

Query window size (km*km)	LoD	Faces	Edges	Intersections found	Polygons
2.2*1.5	1000000	123	804	67	138
2.2*1.5	500000	139	843	73	152
2.2*1.5	200000	173	952	96	191
2.2*1.5	100000	215	1072	115	239
2.2*1.5	50000	279	1214	133	308
2.2*1.5	20000	350	1336	156	383
2.2*1.5	10000	426	1433	176	468
2.2*1.5	1000	566	1601	191	617
2.2*1.5	100	574	1609	191	625

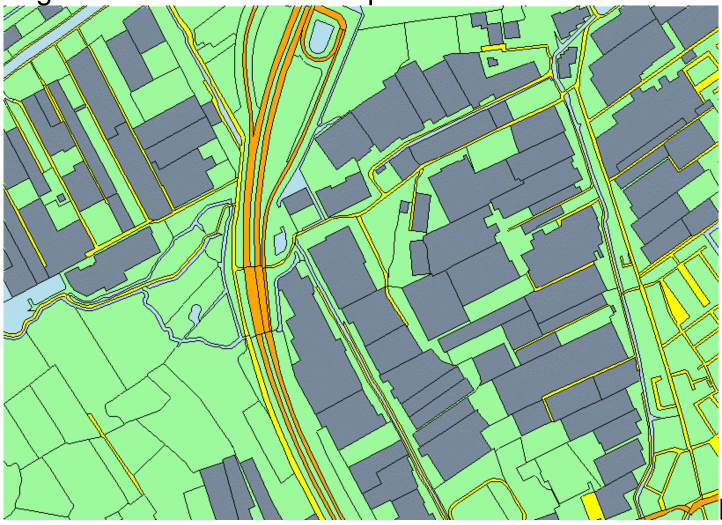
Table 6.4. Statistics on the query-time procedures.

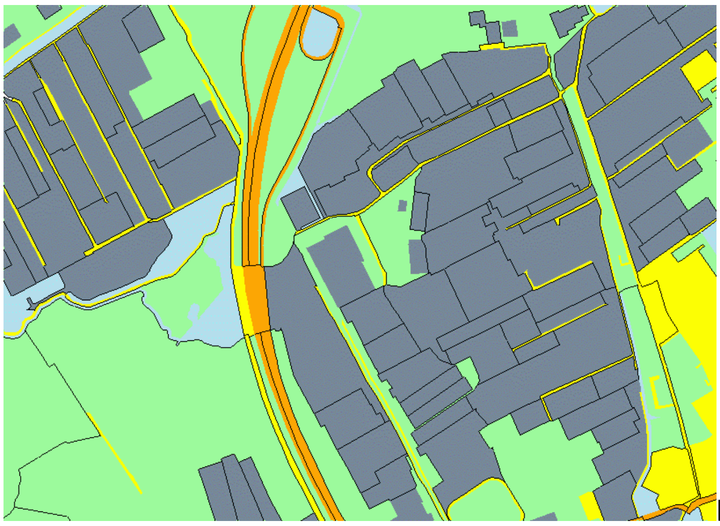
To show the cartographic results of these generalizations a number of map displays are shown in figure 6.2 The area in the map display corresponds to the 2.2*1.5 km area in table 6.4, which is a sub set of the entire map as seen in figure 6.1. The black lines in these pictures show the edges that were used to reconstruct the polygons. The images are shown in decreasing detail, which also illustrates how the generalization method operates. These image where made by taking screenshots of the running client application QuickGIS. A good spot to see the progressive generalization over all images is the curved highway access road at the top centre of the map.
Figure 6.3 shows the entire program window including a window displaying details on a selected faces. This detail window can be called by clicking on the face of which the

information is wanted. This functionality is based directly on the link between edges and faces.

6.4. Addition of BLG-tree

Although it has already been established that the sheer number of edges is a problem, the individual edges were still stored with a BLG-tree extension. Thus the number of points per edge can be reduced. To really see the effect of the reduction, manual input for the value of the BLG-tree tolerance was used. The values 0, 5 and 10 in map units where used. The map unit for the test data was meter. Depending on the scale these values had a significant impact on the visual appearance of the maps since values of 5 and 10 meters were often larger than the width of one pixel.


LoD: 20,000


LoD: 100,000

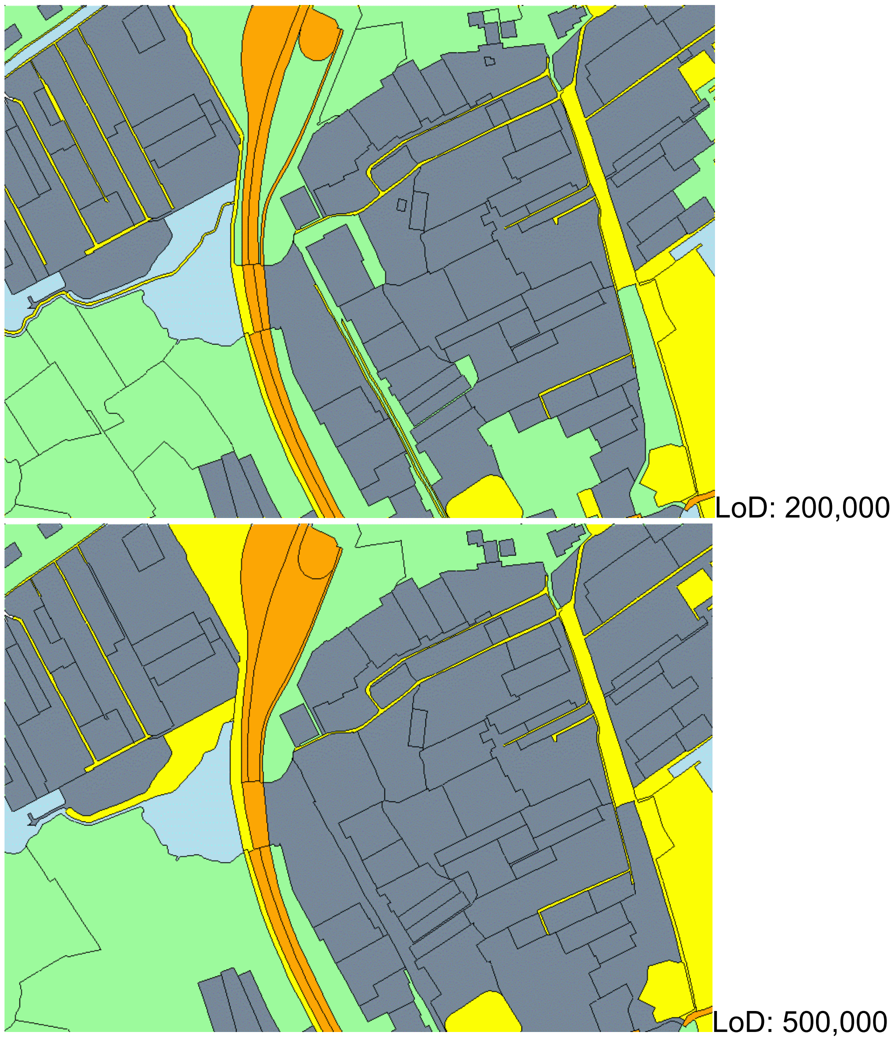
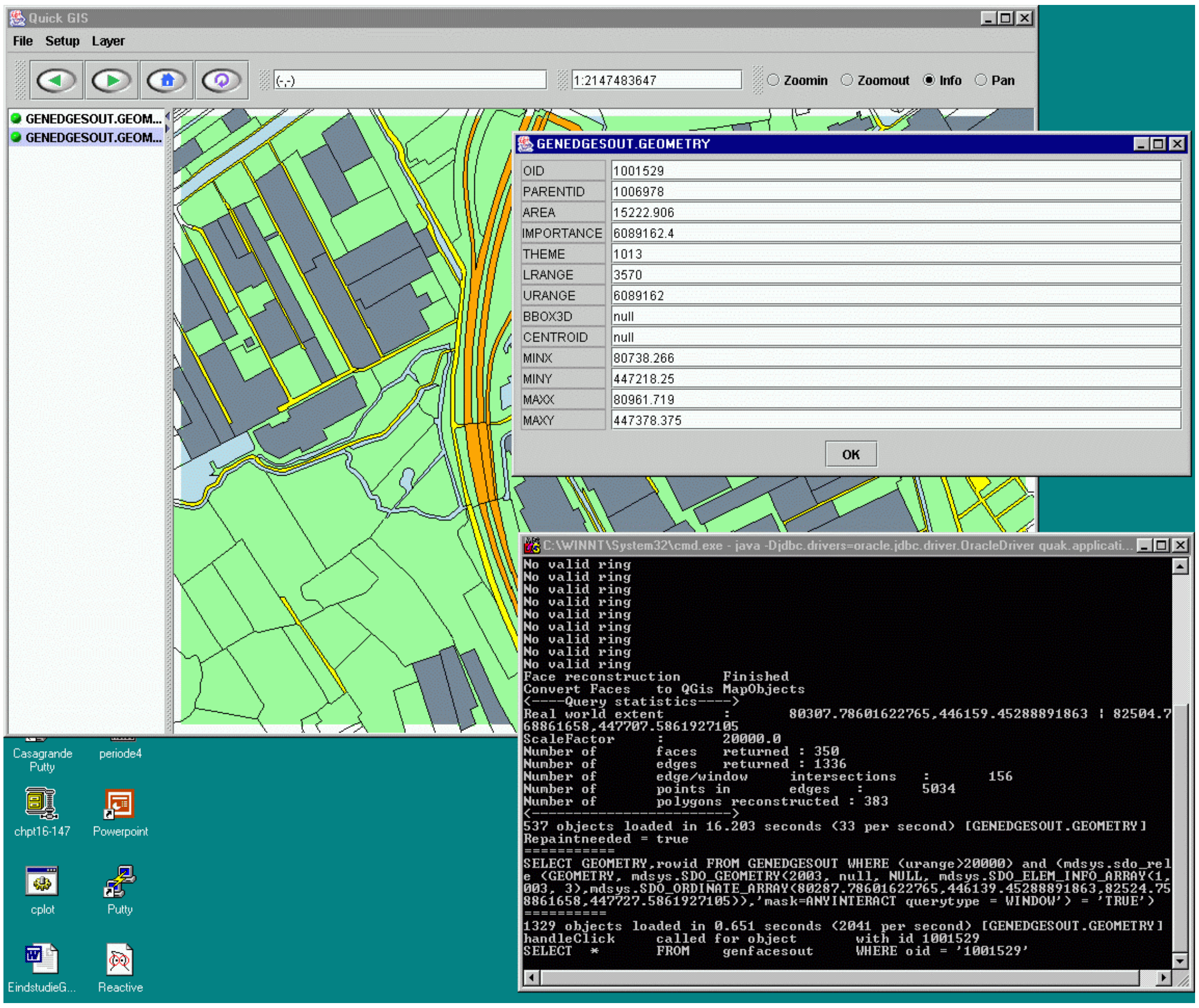


Figure 6.2: Cartographic result of the generalization method.



OID	1001529
PARENTID	1006978
AREA	15222.906
IMPORTANCE	6089162.4
THEME	1013
LRANGE	3570
URANGE	6089162
BBOX3D	null
CENTROID	null
MINX	80738.266
MINY	447218.25
MAXX	80961.719
MAXY	447378.375

```
No valid ring
No valid ring
No valid ring
No valid ring
No valid ring
No valid ring
No valid ring
No valid ring
No valid ring
Face reconstruction     Finished
Convert Faces   to QGis MapObjects
<----Query statistics---->
Real world extent       :       80307.78601622765,446159.45288891863 ! 82504.7
68861658,447707.5861927105
ScaleFactor     :       20000.0
Number of       faces   returned : 350
Number of       edges   returned : 1336
Number of       edge/window     intersections   :       156
Number of       points in       edges   :       5034
Number of       polygons reconstructed : 383
<------------------------>
537 objects loaded in 16.203 seconds (33 per second) [GENEDGESOUT.GEOMETRY]
Repaintneeded = true
===========
SELECT GEOMETRY,rowid FROM GENEDGESOUT WHERE (urange>20000) and (mdsys.sdo_rel
e (GEOMETRY, mdsys.SDO_GEOMETRY(2003, null, NULL, mdsys.SDO_ELEM_INFO_ARRAY(1,
003, 3),mdsys.SDO_ORDINATE_ARRAY(80287.78601622765,446139.45288891863,82524.75
8861658,447727.5861927105)),'mask=ANYINTERACT querytype = WINDOW') = 'TRUE')
==========
1329 objects loaded in 0.651 seconds (2041 per second) [GENEDGESOUT.GEOMETRY]
handleClick     called for object       with id 1001529
SELECT  *       FROM    genfacesout     WHERE oid = '1001529'
```

Figure 6.3: Screenshot of a running QuickGIS application in the back. The upper right window shows details of a selected face. The lower right window shows the text output of QuickGIS, used for debugging and monitoring.

6.5. Advantages and disadvantages of the data structure

The procedure as presented in this thesis is not yet fully matured and needs enhancements at several points. This paragraph describes the advantages and disadvantages of the data structure as it exists at the moment. Special interest will go into the comparison to the GAP-tree data structure. Since that data structure was not used or tested in this research no quantified performance comparison can be made. However based upon the fundamentals of both data structures, some remarks can be made on that subject. Paragraph 5.6 will go into more detail as to the possibilities of enhancement of the overall data structure.

Advantages

Use of a topological data structure

The developed data structure uses topological properties to store each boundary only ones. This has two advantages. First it means that each boundary between two objects needs to be stored only once. Non-topological data structures store common boundaries of objects twice, one for each of the two adjacent objects. This reduces the total amount of geometric data present in the database. Second, by storing these boundaries only once, they can easily be simplified using a line simplification algorithm, without running the risk of creating sliver polygons. The GAP-tree data structure stores entire geometries and is therefore prone to redundancy in the storing of the boundaries.

Only use standard 3D spatial indices
Efficient selection of appropriate records in both the edge as the face table can be achieved using a standard 3D R-tree index on the 3D bounding boxes of the objects. This approach is an alternative to the reactive tree index, used in conjunction with the GAP-tree in the respective articles and associated researches [8][9]. In fact the use of 3D indexes with the third dimension employed to support selection based on LoD information, could be used for many other data structures allowing the storage of multiple LoDs. Since the Oracle implementation of the R-tree index even allows 4D data to be indexed, that DBMS can also use the R-tree to access 3D geometric data with LoD information.

Only transfer edges completely or partially within query window
The data structure only requires those edges that satisfies the required LoD and lie completely or partially within the query window. Absolutely no edge outside of the query window is needed to construct a valid and complete, area-partitioning map. This feature, presented here as an advantage, is however closely related to the disadvantage of having to transfer face-reference update information.

Distribution of computing power between client and server
At query-time the data structure puts little additional strain on the server on top of that of using standard spatial queries. The added computational need is in the form of having to perform 3D instead of 2D queries, and the need to transfer the child-parent relations.

Disadvantages
Small edges
All edges present in the data structure are those that bound the original faces at the input LoD. These exact same edges become the boundaries of the larger faces and lower LoDs. Subsequently these larger faces are bounded by a large number of small edges, which represent an increasingly large amount of data as larger query windows are used at lower LoDs. Although each of these edges can individually be simplified, e.g. by using a BLG-tree, this cannot be done for a combination of edges. The primary reason for this is the fact that in addition to storing the geometric shape of a boundary, an edge also stores references to its left and right face. Combining multiple edges into one large edge, would cause this combined edge to have to contain multiple reference to left and right faces. Depending on the LoD of a query, multiple separate edges would have to be returned for one single compound edge. No solution was found in this research to alter the result set of a query in such a way that the content of the result set of that query could be constructed largely by self-written code.

Need to transfer face-reference update information
The price for not having to transfer any edges that lie completely outside of the query window and having face references in the edges, is that all the child-parent relations that are present within the query window, up to the specified LoD need to be transferred to the client. This is an additional amount of data that becomes larger with a decrease in detail and larger maps, as it is the connection between the original face and the generalized faces.

Line intersection process needed
In order for the polygon reconstruction process to work correctly it is needed that the query window is added to the list of edges and also that all edges that intersect this query window, are split into two new edges at this query window, as well as the query window edge being split at those intersections. The computation of this intersection is quite a laborious task, especially when large number of edges are involved. This is reflected by long processing times at the client. However the fact that the intersections need to be calculated is more important than whether it is done at the client side.

Topology processing required

The use of topology is both an advantage as well as a disadvantage. The advantages have already been discussed. The main disadvantage is the need for heavy processing to convert the topological data structure to explicit geometric objects. The explicit geometric description is needed for many applications, but most recognisable in the display of maps. Any topological data structure does have this requirement, this disadvantages is therefore not so much related to the developed data structure, but more to the choice to use topology in the data structure.

6.6. Possible enhancements of the data structure

This paragraph discusses ideas concerning the possible enhancements to the data structure that increase the usability of the data structure.

Increase generalization information gathering speed and functionality
On-the-fly generalization requires a pre-processing of all data that can potentially be requested by a user. Although this process of filling the scale-less topological data structure only has to be performed once on all data, and possibly on parts of the data for updates, the process still needs to be performed in a realistic time span. The implementation used in the research, which took 2 hours 16 minutes for half a map sheet, would for example not be useful for processing the TOP10vector data covering the entire area of the Netherlands.
Another issue in the generalization process is the ability to support as many of the generalization operators as possible. At the moment only the aggregation of adjacent faces is supported. This is mainly the result of the fact that the generalization information gathering procedure requires a area portioning as the input. The fact that the client-side polygon reconstructing also creates a area portioning is less of a limitations, since essentially every map display presents such a visualisation, except when transparent layers are used.

Reduce number of edges
It has already been described that the data structure is not yet ready for serious application. The main reason for this is the problem is the use of the original edges even at lower LoDs. Although the use of a BLG-tree data structure on the individual edges, does reduce the complexity of each of these edges, the sheer number still remains a serious issue. In order to reduce this number at higher LoDs at least two methods are possible. The first is to re-introduce the use of explicitely stored LoD's. When a new LoD is stored, all edges that are only incident with two other edges are combined into on large, possibly simplified edge. Although this concepts breaks with the ideal of a scale-less data set, the current developed data structure and query time routines are able handle such a setup without need for any adaptation, as long as the lrange and urange values are set correctly. This is another indication of the separation between the generalization method and the storage of the data, which is an important property of the data structure.

The second method for reducing the number of edges at lower LoDs is more complicated, but should, if realised, be more attractive then the first method. The primary idea of this method is to create a tree structure over the edges. This tree structure together with some stored procedure, should enable the database to dynamically create edge records that are to be sent to the client. This yields two problems. First of all, what is the exact, theoretical layout of this tree structure, and secondly, how can this structure be implemented with the database in such away, that it is indeed possible to dynamically alter both the content and the number of records that are returned. This is different from the GENERALIZE function that was created for the BLG-tree implementation. That function only needed to create a new geometry based on the input of the function. No selection of records has to be performed within the function it self, since the selection of what records are to be processed had already been taken care of by the DBMS based upon the clients query.
The ideal for this edge data structure would be to store large edges, and then to dynamically 'create' the smaller edges if the LoD' requires that. Besides being able to create the output edges this data structure must also encompass the storage of the appropriate face references for all edges that it must be able to create. Although some ideas about the shape

of the data structure came up both theoretical as well as practical obstacles prevented it from being realised.

An important practical problem was the creation of records that are to be returned. In simple spoken language the query ‘Give me all edges with a window with a LoD corresponding to a 1:10,000 map’. The first step in answering this query would be for the database to select all ‘super-edges’ that fit the query window demand. For each of these super-edges the edges corresponding to the indicated LoD, should be generated. It was however unclear how the creation of these edge records should be implemented.

This tree-structure should be able to return a different number of edges, depending on the requested LoD. Another subject of attention for the edge tree would be the handling of edges and parts of edges that fall outside of the query window. The amount of information that is not visible in the final map should be as small possible. However the topological data structures need information from edges outside of the query window to be able to construct the geometric objects and depending, on the data structure, the assignment of the thematic information that is linked to the reconstructed geometries. As seen in the implementation of the data structure the absence of data outside of the map window can be overcome, but only at the price of having to use computational intensive routines. Also the some properties of faces require the presence of the entire objects, e.g. the area and the total length of the boundary. Of course this kind of information calculated during the filling of the data structure, but this would require additional storage space for the face table.

Both methods also immediately reduce the size of the edge-face reference update tree that is retrieved with each query when working with lower LoDs. The edge tree method actually removes the need for this additional query totally, as the references can automatically be set to the correct value if the edge-tree data structure correctly supports this feature. But this should be possible given the other demands on this data structure. The less complex method of the storage of LoDs still requires performing an update of the edge-face references, but the overall size of that tree is limited, since each stored LoD restarts with correct references for that scale. Therefore only a tree up to the first higher LoD needs to be retrieved.

7. Conclusions and recommendations

This final chapter presents conclusions that can be drawn based upon the research presented in this thesis. A number of conclusions refer back to the research questions posed in chapter one, some other are based upon experiences gained during the research, which do not fit any of the questions but are interesting enough to be presented. Furthermore some recommendations are made with regards to the developed data structure and future research on this data structure as well as in the field of 'on-the-fly' generalization in general.

7.1. Conclusions

Answers to the research questions

How can on-the-fly generalization be implemented in a client-server architecture in an efficient way?
This broad targeted question can in general be answered as, by using a data structure that can store geometric and thematic data as well as LoD information, without redundancy in any of the three. The basic geometric information used in the research and subsequent data structure was the presence of an area partitioning. To store such an area partitioning, without redundancy in the geometric data, it is necessary to use a topological data structure that eliminates the need to store the boundaries of faces twice. Furthermore also if these boundaries do not alter between LoDs they should not be stored more than once. The drawback of these data structure comes with visualisation, since for visualisation it is necessary that the geometric properties of the objects to be displayed are available. This conversion from the topological description to an explicit geometric description which is also linked to the thematic information that is available on the object can be a time consuming task, especially if it has to be conducted for a large number of objects and even more if the individual objects are geometrically complex. The current implementation makes use of topological data, both stored explicitly as well as derived from geometric properties. It also requires the alteration of geometric properties by clipping edges with the query window. A practically usable data structure should only have to use stored topological references.
The answer to the question of whether the increase in complexity of the data structure together with the increased need for computational effort outweighs the benefits of having a scientifically more appealing data structure which reduces redundancy, would depend on the purpose of the application utilising the data structure. The current state of the data structure can however not be regarded as practically applicable for serious applications. To allow the data structure to be effectively useable the issues presented in paragraph 5.4 need to be addressed. Furthermore the possibilities of the generalization procedure used in filling the data structure need to be greatly enhanced. This filling process needs to both be able to perform a fully automatic generalization as well as have the means to store the transitions in the map face effectively in the data structure. The first issue here is still a subject of scientific research. The second issue, though important for the field of on-the-fly generalization in general, and especially for the data structure developed in this research, is not likely to be addressed in the near future, as it only becomes effectively relevant when a mature data structure is available.

Which data structures and algorithms can be used that support efficient on-the-fly computation of a generalized data set?
To answer this question both existing data structures as well as the developed ones need to be considered. In the existing data structures a difference must be made between truly on-the-fly generalization data structures and multi-scale data sets. Multi-scale data sets can be created using existing generalization methods and tools. The storage of the individual LoDs should be no problem using standard applications, available for single LoD data sets. The

use of multi-scale data also requires less advanced clients then truly on-the-fly data structures.
The number of data structures for on-the-fly generalization is still very limited. The most matured, but still limited in its use as a stand-alone data source, is the GAP-tree. Although this structure does allow for good response times, the graphical quality of generalization of topographic maps is still not suitable for use in general public applications. As with the developed data structure this is not a direct consequence of the data structure itself, but more of the way the input data is transferred to the data structure.

How should clients formulate their request for a generalized dataset?
This question is mainly concerned with the aspect of the LoD part of the query. The specification of the spatial extent of a query is a straightforward task, which any client accessing spatial data should be able to handle. More interesting is the specification of the LoD. Not only is this a problem in the queries, it is a problem in the setup of any data structure for on-the-fly generalization. The approach used in this research, do directly employ the importance value for LoD purposes is for from ideal. A useful choice for the LoD specification is the mentioned scale-factor, which is the reciprocal value of a scale. This value should be used to indicate the amount of detail that should be present within a certain real-world extent. For example, a scale-factor of 10,000 would imply that the amount of detail per real world km^2 should be equal to the amount given in a 1:10,000 map per 10*10 cm^2 map face. A mature data structure for on-the-fly generalization should be able to provide a conversion between the internally used importance values and externally usable LoD values.
As an overall answer to this question, the request should be formulated as the combination of a query-window and a LoD specification, with the above mentioned remarks on the LoD in mind. Using such a specification it possible to vary both the amount of detail, as well as the spatial extent of the query, which one of the purposes of on-the-fly generalization.

Can the query-time workload be distributed over the server and the client?
In general this question can be answered affirmatively, as the implementation created in the research shows. However the overall processing power needed at query time should still be kept as low as possible, to allow short response times on systems with a reasonable performance level. The current state of the data structure requires too much client-side processing time at query time, even for fast computer systems.
The distribution of the workload can especially be interesting if it helps in the reduction of the data transfer between server and client. As shown in this thesis, it is possible to create a data structure that reduces redundancy in storage and transfer by utilising topological properties. The transfer of topological data, intended for visualisation, implies that client-side processing is required to create a fully geometric description needed for visualisation.

General conclusions
The current state of development of data structures for on-the-fly generalization is not sufficient to use them as the sole source of data for more than simple choropleth maps. As can be seen in the images in this thesis and also in other articles on the GAP-tree, the generalization of topographic maps, such as the TOP10vector, pure on-the-fly generalization does not yet yield results that are easy to interpret for occasional map users. That kind of audience is however very likely to use the various web-mapping applications e.g. for online route planning. For on-the-fly generalization to be seriously applicable, whether using topological data structures or not, the cartographic results should be better then currently achieved.

In general the use of a topological data structure requires more resources than pure geometric representations. Depending on the implementation, some savings can be achieved at the storage area, but this greatly increases the need for computing power, since geometric representations are still needed for many applications. It must be said that the need for computing power with topological data structures need not be as large as apparent in developed routines.

7.2. Recommendations

After reading this thesis it should be clear the primary recommendation of this thesis is to enhance the data structure by developing the missing structure needed to efficiently store the edges in such a way that the actual content and layout of the edges can depend on the LoD. Ideas as to how this can be achieved can be found in paragraph 6.6.

In general the topic of on-the-fly generalization is still a subject that needs more research to increase its usability. It is very well possible that such research would reveal that the on-the-fly generalization is most effectively used in combination with other methods of ‘stored generalization’. Indeed there is already research which points in that direction. The ultimate goal of such coalescence would be to reduce as much as possible the need for the storage of multi-scale data. Future research should therefore also explore which generalization operators can be used effectively with on-the-fly generalization.

Since an important part of the data structure is based upon topological relationships it would be interesting to see whether a more efficient implementation of the data structure and associated algorithms can be achieved using the topology implementation which should become available in Oracle 10i. Another product that supports the use of topology is Radius Topology which extents Oracle 9i to support topology.

References

Literature

[1] Cecconi, A., Galanda, M. *Adaptive zooming in web cartography*. Zurich: Department of Geography, University of Zurich, 2002.
[2] Douglas, D., Peucker, T. 'Algorithms for the reduction of the number of points required to represent a line or its caricature'. *The Canadian Cartographer,* 12(2)(1973), p.112-122.
[3] Hardy, H., Hayles, M., Revell, P., *Clarity, a new environment for generalization using agents, java, xml and topology*, Cambridge: Laser-Scan Ltd., april 2003.
[4] Herring, R., *The OpenGIS™ Abstract Specification, Topic 1: Feature Geometry (ISO 19107 Spatial Schema) Version 5, 2001*, Open GIS Consortium, Inc. 2001. (http://www.opengis.org/techno/abstract/01-101.pdf)
[5] Kilpeläinen, T. *Multiple representation and generalization of geo-databases for topographic maps.* Finnish Geodetic Institute, 1997.
[6] McDonell, R., Kemp, K., *International GIS Dictionary.* Cambridge: GeoInformation International, 1995.
[7] Oosterom, P. van, *Reactive Data Structures for Geographic Information Systems*. Leiden, 1990.
(http://www.gdmc.nl/oosterom/thesis.ps)
[8] Oosterom, P. van, *The GAP-tree, an approach to "On-the-Fly" Map Generalization of an Area Partitioning.* Den Haag: TNO Physics and Electronics Laboratory, 1993.
(http://www.gdmc.nl/oosterom/gap.ps)
[9] Putten, J. van, Oosterom, P. van, *New Results with Generalized Area Partionings.* Rijksuniversiteit Leiden, 1998.
[10] Topografische Dienst Nederland, *Productbeschrijving TOP10vector.* Topografische Dienst Nederland, 1998.
[11] Vivid Solutions, *JTS Technical Specifications.* july 3, 2002 .
(http://www.vividsolutions.com/Jts/bin/JTS%20Technical%20Specs.pdf)
[12] VBB Viak, *Automatic generalization of Geographic Data,* 1997.
(http://129.187.92.218/publications/meng/paper/generalization1997.pdf)
[13] Worboys, M.GIS: *A Computing Perspective*. London: Taylor Francis Ltd, 1998.
[14] Woodruff, A., Landay, J., Stonebraker, M., *Constant Information Density in Zoomable Interfaces*, Berkeley: Department of Electrical Engineering and Computer Sciences, University of California.

Software

[15] Java Topology Suite (JTS), Version 1.2, Vivid Solutions Inc.
[16] Java 2.0, Version 1.4, Sun Microsystems, Inc.
[17] *Oracle 9i spatial* , Version 9.2.0.2.0. Oracle Corporation.

Websites

[18] http://factfinder.census.gov
[19] http://www.ngi.be
[20] http://www.tdn.nl
[21] http://www.usgs.gov
[22] http://www.swisstopo.ch
[23] http://cgi.girs.wageningen-ur.nl/cgi/geodesk/geodata/bodemkaart50000.gif
[24] http://www.locatienet.nl

Appendix A: Server computer system specifications

Hardware and OS

- Sun Enterprise E3500 server with Sun Solaris 7
- Two 400 MHZ UltraSPARC CPUs with 8Mb CPU cache each
- 2Gb main memory
- two mirrored internal disks of 18.2 Gb, fiber channel;
- two internal RAID0 sets (3*18.2Gb each)

four external, hardware controlled, RAID5 sets (6*18.2 Gb each)

Appendix B: Source code server-side routines

Data structure filling routine (PL/SQL)

```
CREATE OR REPLACE PROCEDURE MVGENERALIZE(maximp number)
--This file handles the creation of the actual generalization information
--the necessary tables must already be present

--maximp is the maximum number allowed for the minimum return
area///importance
--this is a terminating condidition for the generalization.
--the other condition is the number of face that remain (ie. don't
generalize if only 1 face is left)

IS
aface genfaces%rowtype;
bface genfaces%rowtype;
cface genfaces%rowtype;
uface genfaces%rowtype;
aedge genedges%rowtype;

--cursor with all edges seperating the two faces
cursor bndedges(faceid1 integer ,faceid2 integer)
is
   select genedges.*
   from genedges
   where (lface=faceid1 and rface=faceid2)
     or (rface=faceid1 and lface=faceid2);

--cursor with all neighboring faces
cursor neighbors(faceid integer)
is
   select genfaces.*
   from genfaces
   where
   oid in
   (
     select distinct genfaces.oid
     from genfaces
     where genfaces.oid in (select rface from genedges where lface=faceid)
       or genfaces.oid in (select lface from genedges where rface=faceid)
   );

--cursor with all enclosing edges of a face
cursor facesedges(faceid1 integer)
is
   select genedges.*
   from genedges
   where (lface=faceid1) or (rface=faceid1);

maxcolval number;   --maximum collapse value encountered
colval number;
minimp number;
teller integer;
bndlength number;   --total combined length of the boundary between two
                      faces
scale number;
newoid integer;
tempnumber number;
numrec number;
```

```
logstr varchar2(255);
logfile utl_file.file_type;

function minn (x1 number, x2 number) return number
IS
BEGIN
   if (x1<x2)then return x1;
   else return x2;
   end if;
END;

function maxn (x1 number, x2 number) return number
IS
BEGIN
   if (x1>x2)then return x1;
   else return x2;
   end if;
END;

--now comes the main procedure
BEGIN
--some lines to enable realtime logging for progress monitoring
logfile:=utl_file.fopen('/var/tmp','maarten_mvgeneralize6.log','a');
logstr:='Begin of mvgeneralize';
utl_file.put_line(logfile,logstr);
select to_char(sysdate,'DD-MM-YYYY HH24:MI:SS') into logstr from DUAL;
logstr := '**** starting run at '||logstr||' *****';
utl_file.put_line(logfile,logstr);
utl_file.fflush(logfile);

--determine smallest importance value of all faces
select min(importance)
into minimp
from genfaces;

teller:=0;

--select face with smallest importances (could be more then one, so select
--one with smallest id)
select genfaces.*
into aface
from genfaces
where genfaces.oid =
(
  select min(oid)
  from genfaces
  where genfaces.importance=minimp
);

select count(*)
into numrec
from genfaces;

while (numrec>1) loop

utl_file.put_line(logfile,'Minimp: '||minimp);
utl_file.put_line(logfile,'Area  : '||aface.area);
utl_file.put_line(logfile,'Teller: '||teller);
utl_file.fflush(logfile);

--select neighbor with highest collapse value
```

```
maxcolval:=0;
--open cursor with all neighboring faces
for bface in neighbors(aface.oid) loop
  --calculate collapse value for all neighboring faces
  bndlength:=0;
  --calculate length of common boundary aface,bface
  for aedge in bndedges(aface.oid,bface.oid) loop
    bndlength:=bndlength+aedge.length;
  end loop;
  --calculate collapse value:=bndlength*compatibility
  select compatibility
  into colval
  from compmatrix
  where theme1=aface.theme
  and theme2=bface.theme;

  colval:=bndlength*colval;
  if colval>maxcolval
  then
    maxcolval:=colval;
    cface:=bface;
  end if;
end loop;

scale:=minimp;
--now we know which two faces need to be collapsed
--the collapse is performed by:
--setting the URange value of the collapsing faces to the current scale
--setting the URange value of the Edge to the current scale
--creating a new face consisting of the entire set of boundaries of the two
  collapsing faces (including islands)
--excluding the common boundaries
aface.urange:=scale;
cface.urange:=scale;

utl_file.put_line(logfile,'AFace : '||aface.oid);
utl_file.put_line(logfile,'CFace : '||cface.oid);

for aedge in bndedges(aface.oid,cface.oid) loop
  aedge.urange:=scale;
  delete from genedges
  where oid=aedge.oid;

  utl_file.put_line(logfile,'Edge Deleted: '||aedge.oid);
  insert into genedgesout(oid,geometry,length,lface,rface,lrange,urange)
  values(aedge.oid,aedge.geometry,aedge.length,aedge.lface,
  aedge.rface,aedge.lrange,aedge.urange);

end loop;

--create new face which is the union of the aface and bface
--a new, unique oid is needed for the new face
--the lrange value of the new face needs to be set to the current scale
--any of the two centroids of the input faces will do as the centroid for
  the new face
--(a point in one of the two face will also be in the union of the two)

uface.oid:=1000001+teller;
uface.theme:=cface.theme;--assigne the new face the theme of the more
important face
uface.lrange:=scale;
uface.urange:=0;
uface.area:=aface.area+cface.area;
uface.centroid:=cface.centroid;
--update bounding box (bounding box of the union is the bounding box of the
  two individual bboxes)
uface.minx:=minn(aface.minx,cface.minx);
uface.miny:=minn(aface.miny,cface.miny);
uface.maxx:=maxn(aface.maxx,cface.maxx);
uface.maxy:=maxn(aface.maxy,cface.maxy);

--the edges the edges that now bound uface must
--have that faceid set to the face id of uface
--since the edge between aface and uface has already been deleted
--that leave all remaining edges that have areference to aface of cface;
update genedges
set lface =uface.oid
where lface=aface.oid or lface=cface.oid;

update genedges
set rface =uface.oid
where rface=aface.oid or rface=cface.oid;

--update importance of new face
select priority
into tempnumber
from themepriority tp
where uface.theme=tp.theme;

uface.importance:=uface.area*tempnumber;

utl_file.put_line(logfile,'UFace : '||Uface.oid);
utl_file.put_line(logfile,'UFace imp: '||Uface.importance);

insert into genfaces
(
  oid, theme, lrange, urange, area, importance, minx, miny,
  maxx, maxy, centroid
)
values
(
  uface.oid, uface.theme, uface.lrange, uface.urange,
  uface.area, uface.importance, uface.minx, uface.miny,
  uface.maxx, uface.maxy, uface.centroid
);

--copy inputfaces to output table
insert into genfacesout
(
  oid, parentid, theme, area, importance,
  lrange, urange, centroid, minx, miny,
  maxx, maxy
)
values
(
  aface.oid, uface.oid, aface.theme, aface.area, aface.importance,
  aface.lrange, aface.urange, aface.centroid, aface.minx,
  aface.miny, aface.maxx, aface.maxy
);

insert into genfacesout
(
  oid, parentid, theme, area, importance,
  lrange, urange, centroid, minx, miny,
  maxx, maxy
)
values
(
  cface.oid, uface.oid, cface.theme, cface.area, cface.importance,
  cface.lrange, cface.urange, cface.centroid, cface.minx, cface.miny,
  cface.maxx, cface.maxy
);

--delete inputfaces from source tables
delete from genfaces
where oid=aface.oid;

delete from genfaces
where oid=cface.oid;

--update the boundaries to refer to the new face
for aedge in bndedges(aface.oid,cface.oid) loop
  update genedges set lface=uface.oid where lface=aface.oid;
  update genedges set rface=uface.oid where rface=aface.oid;
  update genedges set lface=uface.oid where lface=cface.oid;
  update genedges set rface=uface.oid where rface=cface.oid;
end loop;

teller:=teller+1;
--determine the minimp/area for the next loop
select count(*)
into numrec
from genfaces;

select min(importance)
into minimp
from genfaces;

select genfaces.*
into aface
from genfaces
where genfaces.oid =
(
  select min(oid)
  from genfaces
  where genfaces.importance=minimp
);
end loop mainloop;

--copy remaining faces in genfaces to genfacesout
insert into genfacesout
(
  oid, theme, area, importance, lrange, urange, centroid,
  minx, miny, maxx, maxy
)
select oid, theme, area,I mportance, lrange, urange,c entroid,
  minx, miny, maxx, maxy
from genfaces;

--copy remaining edges in genedges to genedgesout
insert into genedgesout
(
  oid, geometry, length, lface, rface, lrange, urange
```

```
)
select oid, geometry, length, lface, rface, lrange, urange
from genedges;

utl_file.put_line(logfile,'Number of generalizations: '||teller);
utl_file.put_line(logfile,'---------End of mvgeneralize-------');
select to_char(sysdate,'DD-MM-YYYY HH24:MI:SS') into logstr from DUAL;
logstr := '**** ending run at '||logstr||' *****';
utl_file.put_line(logfile,logstr);
utl_file.fflush(logfile);
utl_file.fclose(logfile);
END MVGENERALIZE;
/
```

Appendix C : Source code client-side routines

This appendix shows code snippets of the client-side routines (Java).

Construction of a 3D query based on 2D query window and LoD information.

```
   query="SELECT  oid,generalize(geometry,blg,"+getBLGFactor()+") as
   geometry,      lface, rface,     urange FROM GENEDGESOUT WHERE ";

   areawhere      =     "" +
      "mdsys.sdo_filter(" + spatialWhereTable + ",   mdsys.SDO_GEOMETRY(" +
      "3003,      "     +     "NULL" + ", NULL, mdsys.SDO_ELEM_INFO_ARRAY(1,
       1003, 3)," +
      "mdsys.SDO_ORDINATE_ARRAY(" +
            viewarea.getMinX() + "," + viewarea.getMinY()  +     ","   +
      lod+","+
            viewarea.getMaxX() + "," + viewarea.getMaxY()  +     ","   +
      lod+ "))," +
      "'mask=ANYINTERACT      querytype   =     WINDOW') = 'TRUE'";

   query=query+areawhere;
```

Calculating intersetions with query-window

```
//for splitting   the   edges we need     a     list that   can
//append records to     its   own   end   whilst traversing itself
//edgelist has been     implemented as a treeset which does not   have this
  functionality
//there     for   the   edges are   put   in a linkedlist   (at   least
  their     references are)
LinkedList elinked=new LinkedList();
it1   =     aedgelist.iterator();
while (it1.hasNext())
{
  elinked.add(it1.next());
}
LinkedList e2linked=new LinkedList();
e2linked.add(windowEdge.makeReverseEdge());

System.out.println("Topo EdgeList   length:     "+elinked.size());
System.out.println("Window EdgeList length:     "+e2linked.size());
//for each edge   in e2linked check whether     it intersects     with any
edge  in elinked
//create segments when they   intersect
int   i1=0;
LineStringIntersector   li1= new LineStringIntersector();
System.out.println("Start     Intersection Search");
int   intersect=0;
int   intersectcount=0;
System.out.println("elinked.size:  "+elinked.size());
while (i1<elinked.size())
{
  intersect=0;
  int i2=0;
  Edge aedge=(Edge)(elinked.get(i1));
  while     ((i2<e2linked.size())   && (intersect==0))
```

```
{
  Edge bedge=(Edge)(e2linked.get(i2));

 com.vividsolutions.jts.geom.Coordinate coord=new
   com.vividsolutions.jts.geom.Coordinate();
 if (li1.IntersectEdges(aedge,bedge)==1)
 {
   //Intersection Found
   intersect=1;
   intersectcount++;

   int    ipt1=li1.index1;
   int    ipt2=li1.index2;
   coord=li1.ico1;

    //temp copy original coordinates
    double x[]=(double[])(aedge.x.clone());
    double y[]=(double[])(aedge.y.clone());
    //resize original linestring
    aedge.x=new double[ipt1+2];
    aedge.y=new double[ipt1+2];
    //copy all origal coordinates upto and including the insertion index
    for   (int i3=0;i3<= ipt1;i3++)
    {
      aedge.x[i3]=x[i3];
      aedge.y[i3]=y[i3];
    }
    //append coordinates of intersection point
    aedge.x[ipt1+1]=coord.x;
    aedge.y[ipt1+1]=coord.y;
    elinked.set(i1,null);
    elinked.add(aedge);

    Edge aedge2=new   Edge();
    //create new edge for   second half of split edge
    aedge2.x=new double[x.length-ipt1];
    aedge2.y=new double[x.length-ipt1];
    aedge2.x[0]=coord.x;
    aedge2.y[0]=coord.y;
    for   (int i3=(ipt1+1);i3<(x.length);i3++)
    {
      aedge2.x[i3-ipt1]=x[i3];
      aedge2.y[i3-ipt1]=y[i3];
    }
    aedge2.oid=aedge.oid;
    aedge2.leftPlaneId=aedge.leftPlaneId;
    aedge2.rightPlaneId=aedge.rightPlaneId;
    aedge2.scaleFactor=aedge.scaleFactor;
    aedge2.tag=aedge.tag;
    //add new   edge to     edgelist
    elinked.add(aedge2);

    //process   query-window      edges
    x=(double[])(bedge.x.clone());
    y=(double[])(bedge.y.clone());
    bedge.x=new double[ipt2+2];
    bedge.y=new double[ipt2+2];
    for   (int i3=0;i3<= ipt2;i3++)
    {
      bedge.x[i3]=x[i3];
      bedge.y[i3]=y[i3];
```

```
        }
        bedge.x[ipt2+1]=coord.x;
        bedge.y[ipt2+1]=coord.y;

        e2linked.remove(i2);
        e2linked.add(bedge);

        aedge2=new Edge();
        aedge2.x=new double[x.length-ipt2];
        aedge2.y=new double[x.length-ipt2];
        aedge2.x[0]=coord.x;
        aedge2.y[0]=coord.y;

        for   (int i3=ipt2+1;i3<(x.length);i3++)
        {
          aedge2.x[i3-ipt2]=x[i3];
          aedge2.y[i3-ipt2]=y[i3];
        }

        aedge2.oid=bedge.oid;
        aedge2.leftPlaneId=bedge.leftPlaneId;
        aedge2.rightPlaneId=bedge.rightPlaneId;
        aedge2.scaleFactor=bedge.scaleFactor;
        aedge2.tag=bedge.tag;
        e2linked.add(aedge2);
        //System.out.println("bedge1 length: "+bedge.x.length);
        //System.out.println("bedge2 length: "+aedge2.x.length);
      }
      i2++;
    }
    i1++;
}
```

Creating rings
Note: the function was erroneously called createPolygon and never renamed, it does however create single rings and not polygons with holes.

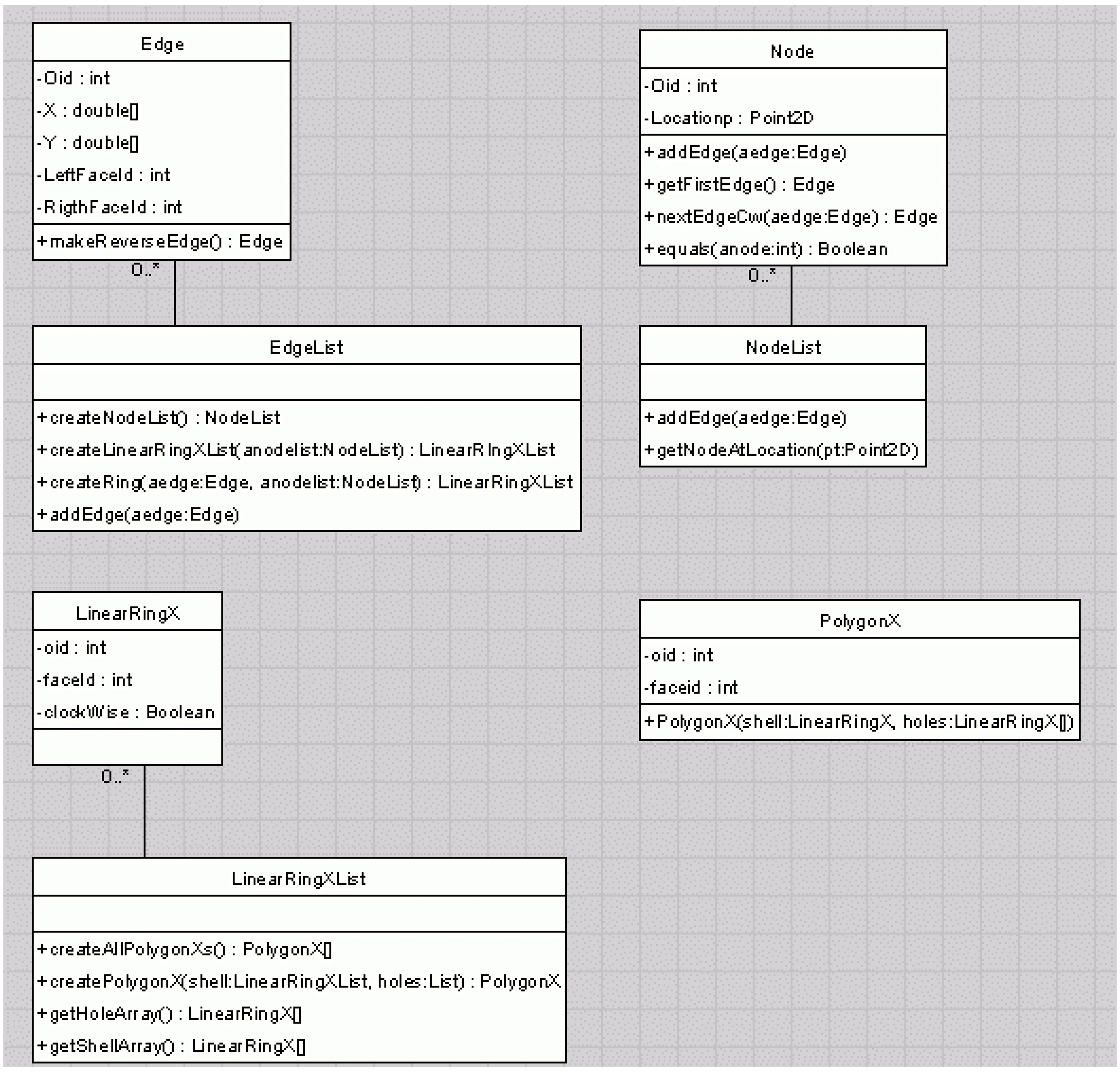
```
public LinearRingX createPolygon(Edge aedge, NodeList anodelist, boolean
useNodeTag)
{
   CoordinateList clist=new CoordinateList();
   Coordinate coord;
   Node anode;
   Node bnode;
   anode=anodelist.getNodeAtLocation(aedge.getFirstPoint());
   double tempscale=aedge.scaleFactor;
   int tempLeftId=aedge.leftPlaneId;
   int tempRightId=aedge.rightPlaneId;
   do
   {
      //if specified mark the edge as processed
      if (useNodeTag)
      {
        aedge.tag=1;
      }
      //determing smallest scalefactor
      if (aedge.scaleFactor<tempscale)
      {
        tempscale=aedge.scaleFactor;
        tempLeftId=aedge.leftPlaneId;
        tempRightId=aedge.rightPlaneId;
      }
      //add all points in the edge except the last one (which will be added
        by the next edge
      int l1=aedge.length();
      if (l1>0)
      {
        for (int i1=0;i1<(l1-1);i1++)
        {
          clist.add(coord=new Coordinate(aedge.x[i1],aedge.y[i1],0),false);
        }
      }
      //getNode at end of edge
      bnode = anodelist.getNodeAtLocation(aedge.getLastPoint());
      //the second point of the next edge may not be the same as the
        prelast point in the current edge (since that would mean a loose
        end)
      Point2D checkpt =aedge.getPoint(aedge.length()-2);
      //select next edge
      aedge=bnode.nextEdgeCw(aedge);
      if (checkpt.equals(aedge.getPoint(1)))
      {
        return null;
      }
   }
   while (!bnode.equals(anode));
   //terminate loop when startpoint has been reached
   //terminate polygon with starting point
   clist.closeRing();
   //clist.add(coord=new
   Coordinate(anode.location.getX(),anode.location.getX(),0),false);
   /**
     *A polygon/ring must contain at least 4 points (including double
        start/end
     *to cover an area.
     *So a null object is returned if less points are present
     */
   if (clist.size()<=3)
   {
     return null;
   }
   //return polygon as LinearRing
   //note the holes/islands are not yet present
   LinearRingX    aring    =new    LinearRingX(clist.toCoordinateArray(),new
   PrecisionModel(),0);
   //assign minimum scalefactor
     aring.scaleFactor=tempscale;
   //assign plane identifier
   //leftPlaneId for CW oriented
   //rightPlaneId for CCW oriented
   if (alg.isCCW(aring.getCoordinates()))
   {
      aring.planeId=tempLeftId;
      aring.CW=false;
   }
   else
   {
      aring.planeId=tempLeftId;
      aring.CW=true;
   }
   return aring;
}
```

Create Polygons

```
/**
  *Create a PolygonX with the specified shell as the outer boundary and any
    possible hole present in Holes
  */

public PolygonX createPolygonX(LinearRingX shell,LinkedList holes)
{
   //for each hole, test whether it fits within the shell
   //if so, at it to the hole list
   //for comparison the LinearRings must be converted to polygons
   Polygon shellPol= new Polygon(shell, new PrecisionModel(),0);
   double shellArea=Math.abs(alg.signedArea(shell.getCoordinates()));
   LinkedList validHoles=new LinkedList();
   //list valid holes
   Iterator it1 = holes.iterator();
   LinearRingX aring;
   while (it1.hasNext())
   {
     aring=(LinearRingX)(it1.next());
     Polygon aringPol=new Polygon(aring, new PrecisionModel(),0);
     //holes must have an area smaller!! than its parent otherwise they
       would be fitted within the island
     double ringArea=Math.abs(alg.signedArea(aring.getCoordinates()));
      //System.out.println("Ringarea: "+ringArea);
      //System.out.println("Contained: "+shellPol.contains(aringPol));
      try
      {
        if ((ringArea<shellArea)&&(shellPol.contains(aringPol)))
        {
          //add the hole to the validHoles list
          validHoles.add(aring);
          //System.out.println("Hole added!");
          //remove the hole from the overall holes list
          it1.remove();
        }
      }catch (Exception e)
      {
      }
      LinearRingX[] returnHoles=new LinearRingX[validHoles.size()];
      validHoles.toArray(returnHoles);
      return new PolygonX(shell,returnHoles,new PrecisionModel(),0);
   }
}
```

Appendix D: Class diagram of objects from client-side routines



Edge
-Oid : int
-X : double[]
-Y : double[]
-LeftFaceId : int
-RigthFaceId : int
+makeReverseEdge() : Edge
0..*
Node
-Oid : int
-Locationp : Point2D
+addEdge(aedge:Edge)
+getFirstEdge() : Edge
+nextEdgeCw(aedge:Edge) : Edge
+equals(anode:int) : Boolean
0..*
EdgeList
+createNodeList() : NodeList
+createLinearRingXList(anodelist:NodeList) : LinearRingXList
+createRing(aedge:Edge, anodelist:NodeList) : LinearRingXList
+addEdge(aedge:Edge)
NodeList
+addEdge(aedge:Edge)
+getNodeAtLocation(pt:Point2D)
LinearRingX
-oid : int
-faceId : int
-clockWise : Boolean
0..*
PolygonX
-oid : int
-faceid : int
+PolygonX(shell:LinearRingX, holes:LinearRingX[])
LinearRingXList
+createAllPolygonXs() : PolygonX[]
+createPolygonX(shell:LinearRingXList, holes:List) : PolygonX
+getHoleArray() : LinearRingX[]
+getShellArray() : LinearRingX[]

Due to problems with the program used in creating this diagram, the super classes of the objects are not shown in the diagram. They are therefore listed in the next table

Class	**Super Class**
Edge	-
EdgeList	TreeSet
Node	TreeSet
NodeList	TreeMap
LinearRingX	jts.geom.LinearRing
LinearRingXList	LinkedList
PolygonX	jts.geom.Polygon