



TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Dissertation

A Formal Approach to Implement Dimension Independent Spatial Analyses

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der
technischen Wissenschaften unter der Leitung von

O. Univ.Prof. Dipl.-Ing. Dr.techn. Andrew U. Frank

E127

Institut für Geoinformation und Kartographie

eingereicht an der Technischen Universität Wien

Fakultät für Mathematik und Geoinformation

Institut für Geoinformation und Kartographie

von

Dipl.-Ing. Farid Karimipour

Matrikelnummer 0728643

Bräuhausgasse 64/7

1050 Wien

Wien, June 2011

eigenhändige Unterschrift



TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Dissertation

A Formal Approach to Implement Dimension Independent Spatial Analyses

A thesis submitted in partial fulfillment of the requirements for the degree of
Doctor of Technical Sciences

submitted to the Vienna University of Technology
Institute of Geoinformation and Cartography
Faculty of Mathematics and Geoinformation

by

Dipl.-Ing. Farid Karimipour

Bräuhausgasse 64/7

1050 Wien

Advisory Committee:

O. Univ.Prof. Dipl.-Ing. Dr.techn. Andrew U. Frank

Institut for Geoinformation and Cartography

Vienna University of Technology

O. Univ.Prof. Dipl.-Ing. Dr.techn. Walter G. Kropatsch

Institute of Computer Graphics and Algorithms

Vienna University of Technology

Vienna, June 2011

autograph signature

To
my wife **Aida**,
my little **Lona**,
and
my parents.

ABSTRACT

Extension of 2D spatial analyses — i.e., a set of operations applied on a spatial data set — to higher dimensions, e.g., 3D and temporal, is one of the requirements to handle real world phenomena in GIS. The current approach is to design a technical solution to extend a certain 2D spatial analysis to a new multi-dimensional space with the least increase in complexity and speed. This technical approach has resulted in developments that cannot be generalized. The result of following such an approach in the software development stage is recoding each spatial analysis, separately, for each dimension. Therefore, the code for a, say, general 2D/3D static and moving objects supporting GIS is nearly four times the current code size, offering four variants: static 2D, moving 2D, static 3D, and moving 3D. The complexity of such a growth of code written in one of the currently popular programming languages, say, C++ is hard to manage, resulting in numerous bugs.

This thesis investigates spatial analyses based on their dimension independent characteristics (i.e., independent of the objects to which the analyses are applied), toward achieving a general solution. It intends to provide an integrated framework for spatial analyses of different multi-dimensional spaces a GIS should support. This framework will be independent of the objects to which the analyses are applied and spatial analyses are formally defined by combinations of the elements of this integrated framework.

To implement this approach, spatial analyses are formally expressed hierarchically where each analysis is defined as a combination of simpler ones. These definitions are independent of dimension and the hierarchy ends in a set of primary operations, which are not further decomposed. A set of required data types are also identified. Having implemented the dimensionally independent data types and operations, they all will be extended to a specific space (e.g., moving points) by applying the mappings between defined the spaces.

The required abstraction of the proposed approach is the subject of algebra that ignores those properties of operations that depend on the objects they are applied to. The desired spaces are structurally equivalent, so they are described by the same algebra. Having implemented the required data types and operations, their extension to a specific space is viable by applying the (structure preserving) mapping.

The proposed approach has been evaluated through implementation of Delaunay triangulation for 2D and 3D static and moving points in the functional programming language Haskell and their efficiency has been evaluated. The implementations were used in two applications, i.e., convex decomposition of polytops and optimum placement of a sensor network based on the moving Voronoi diagram, in order to show how the proposed approach can be practically used. The achieved results certify the hypothesis of the research says that “*studying spatial analyses based on their dimension independent characteristics leads to a consistent solution toward implementation of a multi-dimensional GIS*”.

Complexity and speed are factors used to evaluate the performance of an extension technique in current research. However, the aim here is to avoid recoding each spatial analysis for each dimension. Thus, the main concern of this research is on the mathematical validation of the conceptual framework and investigation of its implementation issues. Nevertheless, the results show that the proposed approach does not affect the big O complexity and speed for applying the spatial analyses on objects of higher dimensions.

KEYWORDS

Multi-dimensional spatial analyses, 3D spatial data, Moving objects, Formal theories, Abstraction, Algebraic structures, Algebraic specifications, Lifting, Delaunay triangulation, Voronoi diagram, Functional programming languages, Haskell

ACKNOWLEDGEMENT

I would like to express my gratitude to all those who contributed in one way or another in the journey of my PhD. It would have not been possible to complete this journey without their help and support.

In particular, I would like to express my deep thanks to Prof. Andrew U. Frank, my first advisor. He generously honored me with his invaluable knowledge and provided me with confidence to push forward. His guidance and ideas were essential for this work. My special thanks to Prof. Walter G. Kropatsch, my second advisor, for his invaluable comments to improve the work.

I also would like to thank my colleagues at the Institute for Geoinformation, with whom I had unforgettable nice time. I appreciate the discussion with Takeshi Shirabe, Gerhard Navratil, Gwen Wilke, Rizwan Bulbul and Barbara Hofer. I will not forget the help and support of Edith Unterweger, Christian Gruber and Banob Akladious.

Several people outside the Institute helped me with this work. I would like to acknowledge my colleagues in the 3D Topography project, especially Feriso Penninga, Christopher Gold, Peter Van Oosterom and Rod Thomson. My special thanks go to Hugo Ledoux for his constructive suggestions to improve the research. I would like to thank Cyrus Shahabi for his kind hospitality during my visit to the University of Southern California.

I dedicate this work to my family members: I would like to thank my wife Aida for her love and support. She always had an open ear for me, both in good and in bad times. And finally, I deeply thank my father and mother for their encouragement and never-ending support. I will not forget the first day of my school when I was fearful and my father stood behind the door of the class all the day.

THE STORY OF MY PHD

It was the first days of September 2004 that I met prof. Andrew Frank for the first time, who came to Iran to teach a course. The proposal of my master thesis had been accepted a week ago on whose topic I had been working for about a year and I was preparing myself for an easy and straightforward research. I did not know that, however, life is about the things that happen while you are busy with planning something else...

In our first discussion, I asked him to contribute in my master thesis, but he found the topic out of his interests. Instead, he suggested another topic. It was really a difficult decision: working on a ready and familiar topic or walk to a completely unfamiliar field of research. Though, it was a tempting suggestion. Working with one of the pioneers of GIS is not something that you can simply ignore, and of course I did not. I had a very hard time at the beginning. Everything was new and unfamiliar: from the concepts and fundamentals to the tools and even the implementation environment; but I tried to do my best to catch his interest, and fortunately I could. The results of my master thesis were satisfactory enough for him to accept that I keep working on it for the PhD. It was not the end of his generosity. He invited me to join him at Institute for Geoinformation, Technical University of Vienna, and I moved there and worked with him closely and learned from him as much as I could.

And now, I am at the end of this journey. When I look back, sometimes I wonder how brave I was to take the risk of this detouring, but I am happy that I took it. Five year continuous and close work with prof. Frank is not an experience that everyone can have, and I am happy that I was one of those few lucky people. He taught me not only how to do a research, but how to live in a scientific community. I learned that scientific community is a family that everyone tries to improve it, and it is not a race that each opponent tries to overtake the others, so you must not hide your promising results for yourself, but give them to the other members of this big family, without fearing of being overtaken. I learned that you can be a supervisor, and at the same time behave as a father, a bother or a close friend. And I learned a lot of other things about real life comparing with which the results of this research is like a drop.

Dear Prof. Frank! I will not forget your never-ending support. I hope I can use them in my life and also teach them to others.

TABLE OF CONTENTS

ABSTRACT.....	I
ACKNOWLEDGEMENT.....	III
TABLE OF CONTENTS.....	V
LIST OF FIGURES.....	X
LIST OF TABLES	XIII
1 INTRODUCTION.....	1
<i>1.1 Motivation.....</i>	<i>1</i>
<i>1.2 Goal and hypothesis.....</i>	<i>3</i>
<i>1.3 Assumptions</i>	<i>4</i>
<i>1.4 Approach and scientific background</i>	<i>4</i>
<i>1.5 Research design</i>	<i>6</i>
1.5.1 Construction of the abstract conceptual framework.....	6
1.5.2 Formal definition of spatial analyses	7
1.5.3 Definition of mappings	7
1.5.4 Development.....	7
1.5.5 Evaluation	7
<i>1.6 Major expected results.....</i>	<i>8</i>
<i>1.7 Intended audience</i>	<i>8</i>
<i>1.8 Organization of the thesis</i>	<i>9</i>

2	EXTENDING SPATIAL ANALYSES TO HIGHER DIMENSIONS: STATE-OF-THE-ART	10
2.1	<i>Delaunay triangulation</i>	10
2.1.1	Definition	10
2.1.2	Star and ears of a Delaunay triangulation	12
2.1.3	Constructing Delaunay triangulation in 2 and 3 dimensions	13
2.1.3.1	Flipping algorithm	14
2.1.3.2	Bowyer-Watson algorithm.....	20
2.2	<i>Dynamic spatial analyses</i>	21
2.2.1	Dynamic Delaunay triangulation	22
2.3	<i>Kinetic spatial analyses</i>	25
2.3.1	Kinetic Delaunay triangulation	26
2.4	<i>Event based approach to move several objects</i>	29
2.5	<i>Discussion</i>	30
2.6	<i>Summary</i>	33
3	FORMAL METHODS	34
3.1	<i>Abstraction</i>	34
3.1.1	Types of Abstraction.....	35
3.1.1.1	Procedural abstraction.....	35
3.1.1.2	Data abstraction	36
3.1.1.3	Iterative abstraction	36
3.2	<i>Algebraic structures</i>	36
3.2.1	Definition of algebra	36
3.2.2	Mappings between algebras.....	37
3.2.3	Algebraic representation of an abstract data type	39
3.3	<i>n-simplexes: an abstract data type for geometry</i>	40

3.3.1.1	Orientation of an n-simplex	42
3.3.1.2	Canonical representation of n-simplexes	43
3.3.1.3	Faces of an n-simplex	43
3.3.1.4	Boundary of an n-simplex.....	43
3.4	<i>Functional programming languages</i>	44
3.4.1	Why we use functional programming languages in this thesis?	45
3.4.2	Functional vs. structured programming languages	45
3.4.3	Evaluation of expressions in functional programming languages	47
3.4.3.1	Free and bound variables	47
3.4.3.2	Reduction	47
3.4.3.3	β -Reduction	48
3.4.3.4	Normal Form	49
3.4.3.5	Weak head normal form	49
3.4.3.6	Head normal form.....	50
3.4.3.7	Lazy evaluation.....	51
3.4.3.8	Outer-to-inner evaluation.....	51
3.4.3.9	Currying mechanism.....	52
3.5	<i>Summary</i>	52
4	PROPOSED APPROACH OF THE RESEARCH	54
4.1	<i>A review on the proposed approach</i>	54
4.2	<i>Formal definition of spatial analyses</i>	56
4.3	<i>Constructing an abstract hierarchical framework of spatial analyses</i>	57
4.3.1	Example: Constructing the hierarchical framework for Delaunay triangulation	58
4.4	<i>Summary</i>	59

5	EXTENSION TO N-DIMENSIONS.....	60
5.1	<i>Vector Algebra</i>	60
5.1.1	Operations on vectors.....	61
5.1.1.1	Inner product.....	62
5.1.1.2	Cross product.....	63
5.1.1.3	Triple product	63
5.1.2	Lifting <i>MakeND</i> to extend data types and primary operations to <i>n</i> -dimensional objects.....	64
5.2	<i>Definition of data types and classes</i>	65
5.3	<i>Operations of n-simplexes</i>	67
5.4	<i>Implementation of spatial analyses</i>	71
5.5	<i>Summary</i>	73
6	EXTENSION TO MOVING OBJECTS.....	74
6.1	<i>Definition of the lifting MakeMoving</i>	74
6.2	<i>Implementation of the mappings (liftings)</i>	75
6.3	<i>Extension of primitive operations to moving objects</i>	76
6.3.1	Extension of operations on <i>Ring</i>	76
6.3.2	Extension of operations with list arguments	76
6.4	<i>Summary</i>	78
7	RESULTS AND EVALUATION	79
7.1	<i>Implementation results</i>	79
7.2	<i>Evaluation</i>	88
7.3	<i>Applications</i>	94
7.4	<i>Summary</i>	101

8	CONCLUSION AND FUTURE WORK.....	102
8.1	<i>Results and major findings.....</i>	103
8.2	<i>Research contribution.....</i>	105
8.3	<i>Directions for future work</i>	108
 APPENDIX 1. THE FUNCTIONAL PROGRAMMING LANGUAGE HASKELL..		109
A1.1	<i>The Functional programming language Haskell.....</i>	109
A1.2	<i>Functions.....</i>	109
A1.3	<i>Lambda expressions</i>	110
A1.4	<i>Data types.....</i>	111
A1.5	<i>Lists</i>	112
A1.6	<i>Pattern matching</i>	115
A1.7	<i>Classes and instances.....</i>	115
 APPENDIX 2. THE HASKELL CODE		117
 BIBLIOGRAPHY		148

LIST OF FIGURES

Figure 1.1. Using lifting $S2M$ to extend static spatial analyses to their moving counterparts	5
Figure 1.2. The research design	6
Figure 2.1. 2D and 3D Delaunay triangulations.	11
Figure 2.2. 2D and 3D Voronoi diagram.	12
Figure 2.3. Duality of Delaunay triangulation and Voronoi diagram	12
Figure 2.4. The star of a vertex v in DT	13
Figure 2.5. Example of ears of a DT	13
Figure 2.6. Two possible triangulations of four 2D points	14
Figure 2.7. A big triangle that contains all of the vertexes	14
Figure 2.8. Position of a point with respect to a line segment	15
Figure 2.9. <i>Walking</i> algorithm to detect the triangle contains the new vertex	16
Figure 2.10. Inserting a new vertex into a triangle	16
Figure 2.11. Flipping algorithm to insert a vertex in a DT	17
Figure 2.12. A big tetrahedron that contains all of the vertexes	18
Figure 2.13. Position of a point respect to a tetrahedron	19
Figure 2.14. Inserting a new vertex into a tetrahedron	19
Figure 2.15. Two possible tetrahedralizations of five 3D points	20
Figure 2.16. A situation when the union of two tetrahedra is concave, so not flippable	20
Figure 2.17. A vertex is added to DT	21
Figure 2.18. Deleting a vertex v from a DT	22
Figure 2.19. Heller algorithm to delete a vertex from DT	23
Figure 2.20. 2-ear and 3-ear	24
Figure 2.21. Delete a vertex from DT	25
Figure 2.22. The opposite triangles of a vertex	27
Figure 2.23. A point moves in the circum-circle of an opposite triangle	28
Figure 2.24. A point moves out of the circum-circle of an imaginary triangle	28
Figure 2.25. Relation between different times defined for topological events	30
Figure 3.1. Abstract data type as a many-to-one mapping	35
Figure 3.2. Mapping elements and + operation of Roman numbers to their correspondences in Arabic numbers	37
Figure 3.3. Mapping R^+ to R through homomorphism \log	38

Figure 3.4. A homomorphism h between two concepts A and B with the same algebra	38
Figure 3.5. A simplicial complex that consists of 0-, 1- and 2-simplexes	41
Figure 4.1. The research approach to extend spatial analyses to n -dimensional static and moving objects.....	55
Figure 4.2. The hierarchy of spatial analyses and operations to define the case studies	59
Figure 5.1. A 2D vector P represented by its Cartesian coordinates	60
Figure 5.2. Addition of vectors and multiplication of a vector with a scalar.....	61
Figure 5.3. Functionality of the <code>addVertex</code>	70
Figure 5.4. Functionality of the <code>border</code> for a set of connected 2-simplexes	70
Figure 7.1. Delaunay triangulation of static points.....	80
Figure 7.2. Voronoi diagram of 2D static points	80
Figure 7.3. Delaunay triangulation of 2D moving points for some time instants.....	81
Figure 7.4. Delaunay triangulation of 3D moving points for some time instants.....	82
Figure 7.5. Voronoi diagram of 2D moving points for some time instants	83
Figure 7.6. Intensional and extensional definition of a function.....	84
Figure 7.7. Map of the street network.....	84
Figure 7.8. Paths of the simulated moving points on the street network	85
Figure 7.9. Delaunay triangulation of the simulated moving points on the street network for some time instants.....	86
Figure 7.10. Voronoi diagram of the simulated moving points on the street network for some time instants	87
Figure 7.11. Running time as a function of number of input points for 2D and 3D static and moving Delaunay triangulation	88
Figure 7.12. Number of reductions for applying different analyses on 2D/3D static/moving points.....	91
Figure 7.13. Number of reductions for applying <i>Sort</i> and <i>DT</i> on 2D/3D slow moving points	93
Figure 7.14. Number of reductions for applying <i>DT</i> on 2D and 3D moving points for multiple time instants where different number of points move	94
Figure 7.15. 2D and 3D non-convex polytop.....	94
Figure 7.16. AHD representation of the polytop of Figure 7.15.a.....	95
Figure 7.17. AHD representation of the polytop of Figure 7.15.b.....	96
Figure 7.18. Satellite image of Latyan dam and its reservoir	97
Figure 7.19. 3D view of Latyan dam and its reservoir	97

Figure 7.20. Points resulted from hydrography of Latyan dam reservoir	98
Figure 7.21. 3D TIN of Latyan dam reservoir	98
Figure 7.22. 3D TIN and surface of Latyan dam reservoir at water level of 1570m	99
Figure 7.23. Level-Surface-Volume diagram of Latyan dam reservoir	100
Figure 7.24. Using the Voronoi diagram for sensor network placement	101

LIST OF TABLES

Table 2.1. Some research to perform Delaunay triangulation in different dimensions	32
Table 3.1. 0- to 3-simplexes and their common names, representations and geometric configurations	41
Table 7.1. Running time as a function of number of input points for 2D/3D static/moving DT/CH	88
Table 7.2. Number of reductions for applying different analyses on 2D/3D static/moving points.....	90
Table 7.3. Number of reductions for applying <i>Sort</i> and <i>DT</i> on 2D/3D slow moving points	92
Table 7.4. Number of reductions for applying DT on 2D and3D moving points for multiple time instants where different number of points move	93
Table A1.1. Some standard manipulating functions over lists.....	114

1 INTRODUCTION

This chapter starts by describing the motivation for the work done in this thesis. The research goal, hypothesis and assumptions are presented. Both our general approach and the specific research design for verifying the research hypothesis are discussed. Expected results and the intended audience are explained. Finally, we present the organization of the thesis.

1.1 Motivation

“Position in space and time are fundamental for a GIS. They allow connecting other observations to locations in space and time” (Frank 2007, p. 86). Early geospatial information systems (GIS) dealt with position in a simple 2D Euclidean space. To handle real world phenomena, however, some applications need 3D and temporal objects. Extending the realm of GIS to these higher dimensional spaces has a wide range of requirements from data storage and data structure considerations to visualization strategies (Langran, 1989; Oosterom et al., 2008; Peuquet, 1999; Raper, 2000). Extension of 2D spatial analyses to higher dimensions is a major requirement in this regard. An spatial analysis is a set of operations applied on a spatial data set that result in a new data set, a value, etc.. Computing convex hull of a set of points, determining the position of a point with respect to a line, and computing the volume of a polytop are examples of spatial analyses that respectively result in a point set, Boolean value, and a numerical value.

Extension of 2D spatial analyses to higher dimensions has been the subject of many studies each has introduced a technical solution to extend a certain spatial analysis to a new multi-dimensional space with the least increase in complexity and speed. Complexity and speed are the parameters usually used for evaluating the efficiency of algorithms in computational geometry. “Computational geometry is defined as the systematic study of algorithms and data structures for geometric objects, with a focus on exact algorithms that are asymptotically fast” (Berg et al., 2008, p. 2). There are several successful results, considering complexity and speed as the evaluating parameters. In practice, however, almost no general solution has been introduced to interact with multi-dimensional, say, 3D and temporal, data. The main reason is that the extension techniques highly depend on the specific case studies. It has resulted in developments that cannot be generalized. In other

words, to extend a certain 2D spatial analysis to a new multi-dimensional space, a technical solution based on the characteristics of the analysis as well as the destination space must be designed. For instance, although several methods have been introduced to construct Delaunay triangulation and Voronoi diagram of 2D moving points (Albers et al., 1998; Bajaj and Bouma, 1990; Guibas et al., 1992; Mostafavi et al., 2003; Roos, 1991, 1993; Roos and Noltemeier, 1991; Vomacka, 2008), they have been modified to construct these structures for 3D moving points (Albers, 1991; Albers and Roos, 1992; Hashemi-Beni et al., 2007; Ledoux, 2008; Schaller and Meyer-Hermann, 2004). The result of following such an approach in the software development stage is recoding each spatial analysis, separately, for each dimension. Thus, the code for a, say, general 2D/3D static and moving objects supporting GIS is nearly four times the current code size, offering four variants: static 2D, moving 2D, static 3D, and moving 3D. The complexity of such a growth of code written in one of the currently popular programming languages, say, C++ is hard to manage, resulting in numerous bugs. The sheer size of the task explains why no commercial GIS has a comprehensive offer for treatment of 3D and moving objects.

Current approaches differentiate the same spatial analysis in different spaces despite their unification in the real world: to calculate the distance between two points, there are different methods depend on the type of the points (2D or 3D, static or moving, etc.), although the concept of distance in all of these multi-dimensional spaces are the same. GI science deals with the formal modeling of spatial process and interaction of humans with the environment in space and time (Frank, 2000). Thus, it is to have such a unified view point (Egenhofer and Mark, 1995). However, the space-based view of current approaches causes separation in how they deal with spatial analyses in different multi-dimensional spaces.

This research provides an integrated framework for spatial analyses in multi-dimensional spaces. Here, we study spatial analyses based on their dimension independent properties. This is similar to the approach proposed by Felix Klein in 1872 to study geometries via their invariants, which are independent of dimension (Klein, 1939). The framework will be independent of the dimension of the objects to which the analyses are applied and spatial analyses are formally defined by combinations of the elements of this integrated framework.

The proposed approach needs a more abstract view that ignores those properties of analyses that depend on the objects they are applied to. It enables us to have abstract descriptions of analyses. The required abstraction is the subject of algebra, which describes

an abstract class of objects and their behaviors. Structure of operations in algebra is independent of implementation. Thus, behavior of many things can be described by the same algebra as long as their behavior is structurally equivalent.

The multi-dimensional spaces a GIS must support are structurally equivalent, so they can be described using the same algebra and the required mappings between these spaces are defined. Having implemented the dimensionally independent data types and operations, they all will be extended to a specific space (e.g., moving points) by applying the mapping defined between the spaces.

In abstract, this research intends to integrate spatial analyses of multi-dimensional spaces through incorporation of algebraic specifications as formal abstractions. The research is based on the formalizations provided in GI science and GI theory (Bittner and Frank, 1997).

1.2 Goal and hypothesis

The goal of this research is to provide an integrated framework for spatial analyses of multi-dimensional spaces based on their dimension independent properties. This foundation can be used to formally define spatial analyses based on the elements of the framework. It intends to contribute the abstraction and algebraic specifications as indispensable concepts to provide the required abstraction which “captures the essence of the semantics of operations and objects” (Frank, 2007, p. 55). Our concentration is on mathematical validation of the proposed approach. However, we implement the approach for some case studies.

Complexity and speed are performance factors in most of the current research in computer science and computational geometry. However, the main concern of this research is on the mathematical validation of the conceptual framework and investigation of its implementation issues, so performance is not a key factor for evaluation of the results. “Performance” is one of the four areas (the others are “ontology and semantics”, “user interface” and “error and uncertain data”) that link the formal treatment of geospatial data to its use and should be excluded in developing a theory for geospatial data processing. “Without this clear separation, we taint the description of the things we presently understand with our ignorance in other areas” (Frank, 2007, pp. 24-25).

More specifically, we want to answer these questions:

1. What are the dimension independent properties of spatial analyses in different multi-dimensional spaces? Does this knowledge suffice to construct an abstract integrated framework based on which spatial analyses in any multi-dimensional space are described?
2. Having formally defined a spatial analysis as a combination of the elements of this abstract integrated framework, can it be extended to higher dimensions through the connections between the spaces?
3. How does it impact the performance comparing to the current approaches?
4. What are the barriers, if any, of implementation of this mathematically provable integration in a programming environment (here, Haskell)?

The hypothesis of the research is:

“Studying spatial analyses via their dimension independent properties leads to a consistent solution toward multi-dimensional GIS”.

1.3 Assumptions

As this research is done in the field of GIS, the investigated spaces are limited to those interesting for GI science, which are Euclidean 2D static, 3D static, 2D temporal and 3D temporal. On the other hand, the temporal data are divided into *kinetic* and *dynamic*. In the kinetic or *moving* case, the continuous movement of objects is considered. It means there is a fixed set of objects whose position change over time. In the dynamic case, the position of the objects is fixed, but new elements may be inserted or deleted over time. In this research, we concentrate only on moving objects, but this does not limit the approach to such cases. Nonetheless, methods for applying spatial analyses on dynamic objects are introduced in chapter 2.

1.4 Approach and scientific background

This research provides an integrated framework for spatial analyses in different multi-dimensional spaces through concentration on dimension independent structural properties of the spatial analyses. It proposes that abstract algebraic presentations provide the minimum

information about spatial analyses in a general context that can further be extended to a more specific space.

Delaunay triangulation is selected as the particular case study to answer the research questions posed and to verify our hypothesis. The proposed approach of the research will be used to develop this spatial analysis for points of different dimensions. Focusing on this particular analysis does not affect the general validity of the results, but it allows us to concentrate on validation of the proposed conceptual framework of the research.

Algebraic specifications provide the required formal abstraction of the research. The adopted approach is based on representation of spatial analyses in different multi-dimensional spaces using the same algebra. We formally express spatial analyses hierarchically where each analysis is defined as a combination of simpler ones. These definitions are independent of dimension and the hierarchy ends in a set of primary operations, which are not further decomposed. A set of required data types are also identified. Having implemented the dimensionally independent data types and operations, they all will be extended to a specific space (e.g., moving points) by applying the mappings defined between spaces. As an example, Figure 1.1 shows the general scheme of the described approach to extend static spatial analyses to their moving counterparts.

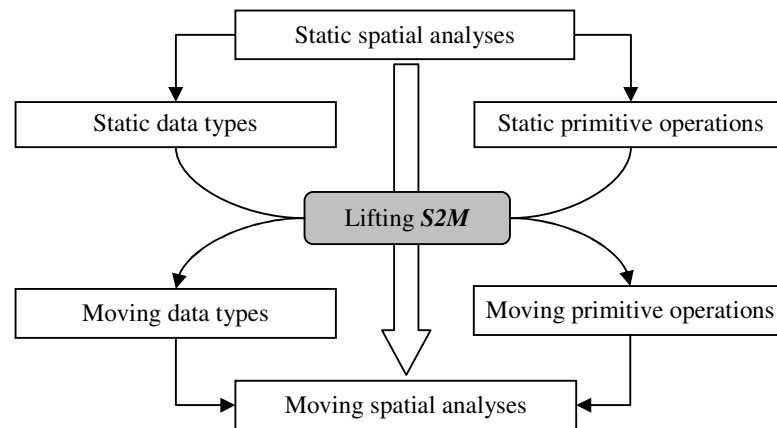


Figure 1.1. Using the lifting $S2M$ to extend static spatial analyses to their moving counterparts

The proposed formalization approach is a generalization of Guting's τ operation (Erwing et al., 1999; Guting et al., 2000; Guting et al., 2003; Guting and Schneider, 2005), which provides a mapping between analyses of static and moving points. However that mapping does not consider the concept of formalization and algebraic structures. The result

is the same if we limit the research to construct a mapping between spatial analyses of static and moving points, which has been implemented in the previous steps of this research (Karimipour, 2005; Karimipour et al., 2005a; Karimipour et al., 2005b; Karimipour et al., 2006). Navratil et al. used this formalization to lift the distance operation of two fixed points to points with different types of uncertainty (Navratil, 2006; Navratil et al., 2008).

1.5 Research design

The research adopts a five step methodology, which is shown in Figure 1.2 and described in the following subsections:

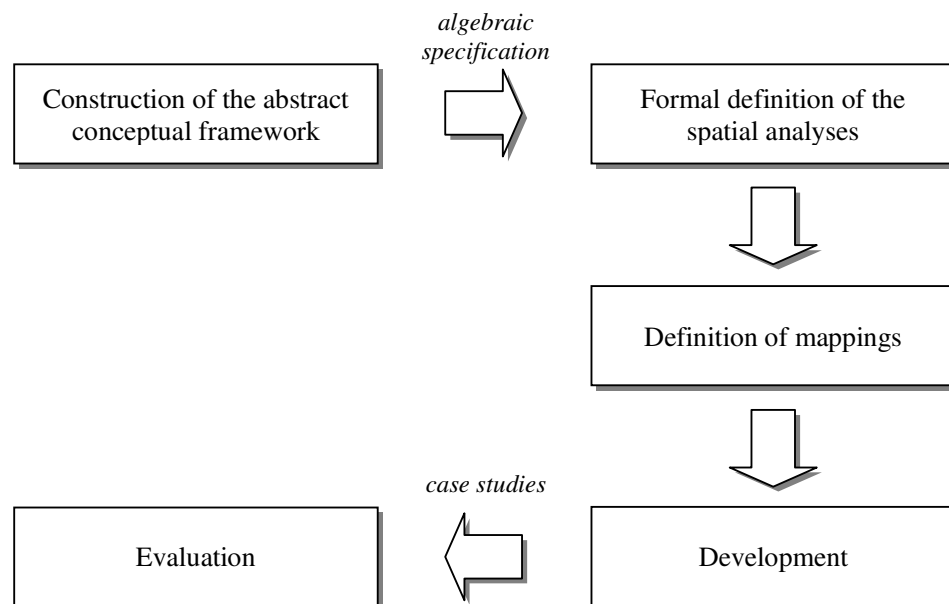


Figure 1.2. The research design

1.5.1 Construction of the abstract conceptual framework

The underlying strategy of the process starts with investigating the spatial analyses in different multi-dimensional spaces in order to define them based on their dimension independent structural properties. It leads to construct the abstract integrated framework of spatial analyses and operations. This framework, then, plays the role of a conceptual underline on which the GIS desired multi-dimensional spaces are constructed. It decomposes the spatial analyses to a set of data types and primitive operations that are not

further decomposed. Further spatial analyses will be defined according to combinations of these data types and primitive operations.

1.5.2 Formal definition of spatial analyses

The conceptual framework defined in the previous step, is formalized using algebraic specifications. The behavior of analyses are formally described as combinations of the elements of the integrate framework.

Algebraic specifications connect the conceptual framework to its implementation. The purpose is to formally describe the behavior of analyses. For this reason, we use them to describe the behavior of the analyses.

1.5.3 Definition of mappings

Spatial analyses are described based on the elements of the integrated framework, so mappings are defined to extend the spatial analyses to the desired multi-dimensional spaces. Different spaces are considered structurally equivalent and their mappings (liftings) are defined. This step will mathematically prove the proposed extension.

1.5.4 Development

The elements (i.e., operations) of the integrated framework are extended to a certain multi-dimensional space by applying the mappings provided in the previous step.

1.5.5 Evaluation

The constructed integrated frameworks and defined mappings will be evaluated using Delaunay triangulation to examine the implementation feasibility of the mathematically provable idea, to identify probable technical barriers, and to investigate the performance of the implementations. Without losing the generality, we assume the data are in general position. It helps us to focus on the main concern of the thesis — which is the mathematical validation of the approach and its implementation issues — and prevents from struggling with the subtleties of how to handle the degenerate cases, which is not our main concern.

The implementations were used in some applications, i.e., convex decomposition of polytopes and optimum placement of a sensor network based on the moving Voronoi diagram, to see how the implementation can be practically used.

The evaluation code is written in the functional programming language Haskell (Peyton Jones, 1987; Peyton Jones and Hughes, 1999; Thompson, 1999). Definitions are built in the form of functions. Functional programming languages use a similar syntax and have similar mathematical foundations as algebraic specifications. It defines each element as a function and also defines their relations explicitly as algebra (Hughes, 1989).

1.6 Major expected results

This thesis proposes a formal approach to implement dimension independent spatial analyses. The major expected results of this research are:

1. Introduction of a mathematical approach to construct an abstract integrated framework that expresses the essence of spatial analyses independent of the objects to which they are applied
2. Definition of the framework into a mathematical model with executable specifications in functional programming paradigm (Bird and Wadler, 1988; Thompson, 1999). This model will be independent of implementation details.
3. Identification of the barriers to implement the mathematically provable proposed idea in a programming environment (here, Haskell) through the selected case study (i.e., Delaunay triangulation) and investigation of the performance of the implementations.

1.7 Intended audience

Scholars in GIScience and related disciplines (e.g. computer science), who search for developing 2D spatial analyses to support higher dimensions, are the intended audience of this research. They can use the defined framework as a core and extend it by adding new elements of their own specific applications. We use this approach to implement spatial analyses for objects moving on a network, spatial analyses for non-convex polytopes of any dimension by convex decomposition of them (Bulbul, 2011; Karimipour, 2009; Karimipour

et al., 2010a; Karimipour et al., 2010b), and optimum placement of a sensor network based on the moving Voronoi diagram (Argany et al., 2010).

1.8 Organization of the thesis

The next chapter reviews previous approach concerning the extension of spatial analyses to higher dimensions and shows how they are applied to extend 2D Delaunay triangulation to support 3D and moving points. A discussion on comparing the current approach with the proposed approach of the research is then presented.

Chapter 3 introduces the formal methods that will be used in the proposed approach of the research. The concepts of abstraction, algebraic structures and simplicial complexes are presented in this regards. The functional programming languages are introduced and the reasons of using such an environment are presented.

Chapter 4 describes the proposed approach of the research to extend spatial analyses to different multi-dimensional spaces. It explains how the spatial analyses are decomposed and how the data types and primitive operations are defined. Different geometric and topological operations needed for the implementation of the case study is presented.

Chapters 5 and 6 describe the extension of spatial analyses to higher dimensions. In chapter 5, we use n -simplexes as an n -dimensional data type. A set of operations are defined on simplexes based on the concept of vector algebra. In chapter 6 we show how to extend the n -dimensional static spatial analyses to support moving objects. The required mappings are presented and discussed. The Haskell implementations of the theories are presented in these chapters.

Chapter 7 presents the implementation results for extending Delaunay triangulation to different dimensions. We then evaluate and discuss the performance of the implementations. At the end of this chapter, two applications developed upon the implementation are presented to show how the proposed approach can be practically used.

Chapter 8 summarizes the work done in this thesis. We present the results and major contribution as well as the research achievements. The chapter concludes with the possible directions for future work.

The proposed approach of the thesis has been implemented in the functional programming language Haskell. In Appendix 1, the main concepts and syntax of Haskell are described. The complete Haskell code of the implementations is given in Appendix 2.

2 EXTENDING SPATIAL ANALYSES TO HIGHER DIMENSIONS: STATE-OF-THE-ART

This chapter reviews the state-of-the-art in extending spatial analyses to higher dimensions and shows how the current approaches are applied to Delaunay triangulation, as the case study of the research. For this, Delaunay triangulation is introduced and some methods to construct this structure are presented. Current approaches to extend spatial analyses to higher dimensions are reviewed and their applications to Delaunay triangulation are explained. We use this information at the end of the chapter to compare the current approaches with the proposed approach of the research.

2.1 Delaunay triangulation

Delaunay Triangulation (DT) is a fundamental structure in computational geometry. This structure is commonly used in several applications, from computer graphics, visualization, computer vision, robotics, and image synthesis to mathematical and natural sciences (Cignoni et al., 1998).

Delaunay Triangulation is well known in geosciences for many years (Gold, 1979, 1994, 1998; Gold et al., 1977; Ledoux, 2008). It is the basic structure for many geoscientific applications such as terrain modeling, spatial interpolation and geological mapping problem. It is also widely used in 3D geoscientific modeling. “3D Delaunay triangulation is used in many geoscientific applications that collect data about spatial objects and domains such as features of the solid earth (aquifers), oceans (currents) or atmosphere (weather fronts), which fill 3D space”(Lattuada and Raper, 1995). Furthermore, there are several applications in geosciences for which constructing the 3D Delaunay triangulation is the basis, e.g., surface modeling, iso-surface extraction (Ledoux and Gold, 2007) and reconstruction of 3D complex geological objects (Yong et al., 2004).

2.1.1 Definition

Given a point set P in the plane, the Delaunay triangulation is a unique triangulation of the points in P (if the points are in general position), which satisfies the empty circum-circle

property: the circum-circle of each triangle does not contain any other point $p \in P$ (Delaunay, 1934; Guibas and Stolfi, 1985; Okabe et al., 2000; Stolfi, 1989a, b). This structure for a set of 3D points is the tetrahedralization of the points in which the circumsphere of each tetrahedron does not contain any other point of the point set. Figure 2.1 shows Delaunay triangulation of some 2D and 3D points.

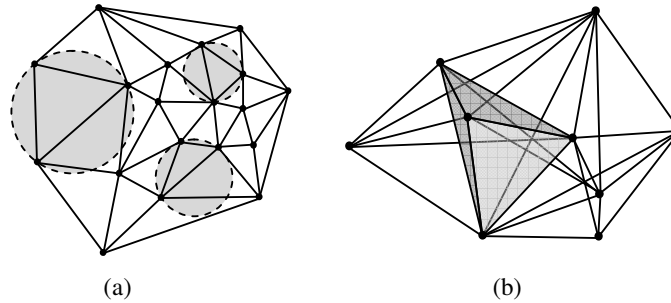


Figure 2.1. 2D and 3D Delaunay triangulations. (a) 2D: some of the circum-circles are shown (b) 3D: one of the tetrahedra is highlighted.

Delaunay triangulation is the dual structure of Voronoi diagram. The Voronoi diagram (VD) of a set of points is defined as follows: Let P be a set of points in an n -dimensional Euclidean space R^n . The Voronoi cell of a point $p \in P$, noted $V_p(P)$, is the set of points $x \in R^n$ that are closer to p than to any other point in P :

$$V_p(P) = \{x \in R^n \mid \|x-p\| \leq \|x-q\|, q \in P, q \neq p\} \quad (2-1)$$

The union of the Voronoi cells of all points $p \in P$ form the Voronoi diagram of P , noted $VD(P)$:

$$VD(P) = \bigcup_{p \in P} V_p(P) \quad (2-2)$$

Figure 2.2 shows 2D and 3D examples. The Voronoi diagram is a very simple structure and is used in many real-world applications (Ledoux, 2008). For an exhaustive surveys on Voronoi diagram and their applications, see (Aurenhammer, 1991; Okabe et al., 2000).

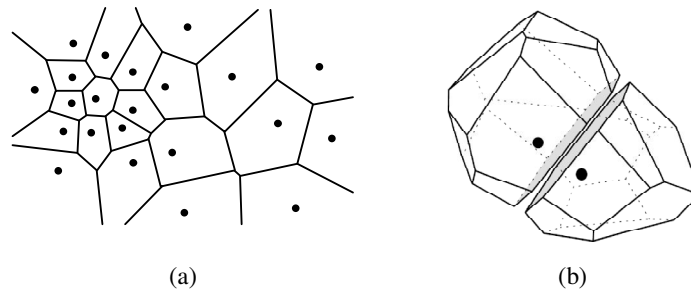


Figure 2.2. 2D and 3D Voronoi diagram: (a) VD of a set of points in the plane. (b) Two Voronoi cells adjacent to each other in \mathbb{R}^3 (they share the grey face).

Delaunay triangulation and Voronoi diagram are dual structures: the center of circum-circles (-spheres) of Delaunay triangulation are the Voronoi vertices; and joining the adjacent generator points in a Voronoi diagram yield their Delaunay triangulation (Figure 2.3). This duality is very useful because construction, manipulation and storage of the Voronoi diagram is more difficult than Delaunay triangulation, so all the operations can be performed on Delaunay triangulation, and the Voronoi diagram is extracted on demand (Ledoux, 2008).

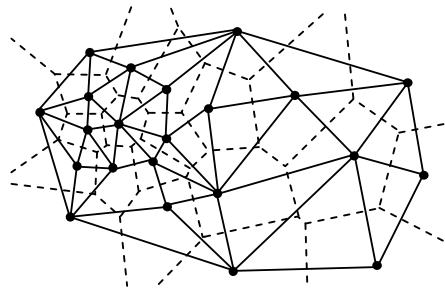


Figure 2.3. Duality of Delaunay triangulation (solid lines) and Voronoi diagram (dashed lines)

2.1.2 Star and ears of a Delaunay triangulation

Among the concepts to interact with the topology of the Delaunay triangulation, here we explain the star and ears of a Delaunay triangulation, which will be used later for moving or deleting a vertex in a Delaunay triangulation.

Consider a vertex v in an n -dimensional Delaunay triangulation. All the triangles (tetrahedra in 3D) that contain v form the $star(v)$, which has a star shape (Figure 2.4). For example, in 2D, the $star(v)$ contains the vertex v itself, and all the triangles and edges incident to v .

Let all the elements of dimension $(n-1)$ be a face in T , which are edges in 2D and triangles in 3D. An imaginary triangle (tetrahedron in 3D) that is formed by the vertexes of adjacent faces is an Ear of T :

- In 2D, an ear is constructed by three vertexes spanning two adjacent edges of two neighboring triangles (Figure 2.5.a).
- In 3D, an ear is constructed by four vertexes spanning either two adjacent faces (2-ear), or three faces incident to a vertex (3-ear) (Figure 2.5.b).

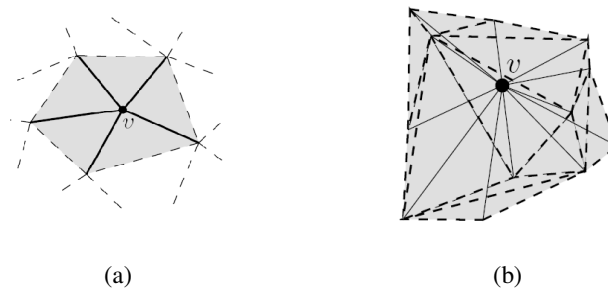


Figure 2.4. The star of a vertex v in DT: (a) 2D (b) 3D (Ledoux, 2008)

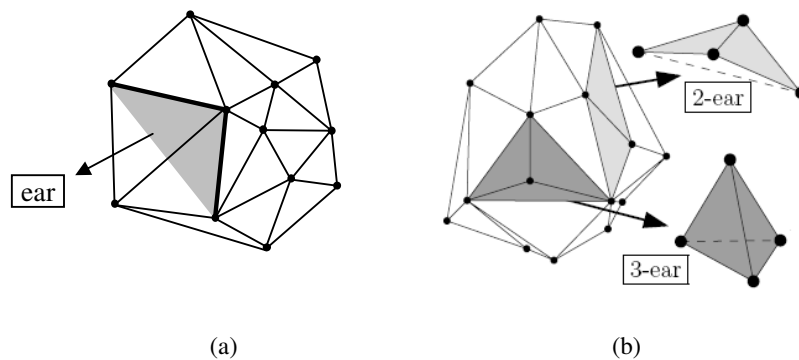


Figure 2.5. Example of ears of a DT: (a) 2D (b) 3D. In 2D, two adjacent edges (bold lines) form an ear. In 3D, two adjacent triangular faces (light grey) form a 2-ear, and three triangular faces incident to a vertex (dark grey) form a 3-ear (Ledoux, 2008).

2.1.3 Constructing Delaunay triangulation in 2 and 3 dimensions

The construction of the Delaunay triangulation is a classical problem of computational geometry. Many algorithms were proposed to construct the Delaunay triangulation of a set of points of different dimensions. Based on the paradigm used, they are classified into *incremental* (Bowyer, 1981; Edelsbrunner and Shah, 1992; Field, 1986; Joe, 1991; Lawson, 1977; Mucke, 1988; Watson, 1981), *divide and conquer* (Cignoni et al., 1998), and *sweep-*

line (Fortune, 1987) algorithms. There are also some other algorithms such as wrapping (Dwyer, 1991; Maus, 1984; Tanemura et al., 1983) and convex hull based (Brown, 1979; Edelsbrunner and Seidel, 1986) algorithms.

In the following, two incremental algorithms to construct Delaunay triangulation of a set of 2D points are introduced. Then, we show how they are adopted to support 3D points.

2.1.3.1 Flipping algorithm

This is an incremental algorithm that was originally introduced by Lawson (Lawson, 1977). It is based on the fact that there are two possible triangulations for four points in 2D, only one of which satisfies the circum-circle property (Figure 2.6). Replacing one configuration with the other one is called *flipping*. In 2D case, it is called *flip22* because there are two triangles before and after the flip operation.

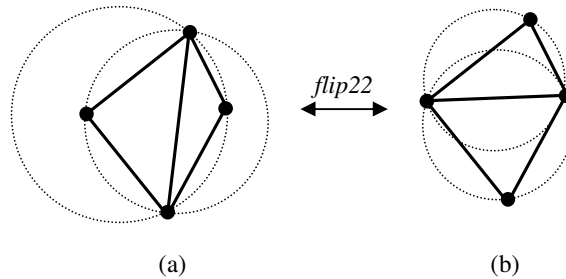


Figure 2.6. Two possible triangulations of four 2D points. Triangulation in (b) satisfies the circum-circle property.

Incremental algorithms start with the minimum number of points with a known structure. Here, we start with a big triangle that contains all of the vertexes (Figure 2.7). Other vertexes are inserted into the structure one by one and after each insertion, the structure is updated. To insert a vertex in a 2-dimensional Delaunay triangulation using the flipping algorithm, three steps are taken:

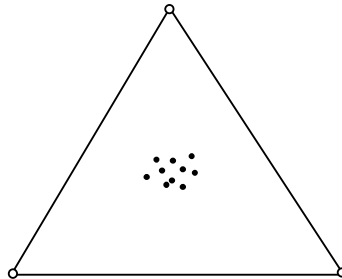


Figure 2.7. A big triangle that contains all of the vertexes

2.1.3.1.1 Finding the containing triangle

First step is to find the triangle that contains the new vertex. This is done through determining the position of the new vertex with respect to the edges of the triangle.

To determine the position of the point p with respect to a line segment \overline{ab} , the following determinant is used (Figure 2.8):

$$D = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_p & y_p & 1 \end{vmatrix} \quad (2-3)$$

If D is positive, then abp is clockwise (i.e., p is on right side of \overline{ab}); otherwise abp is counter-clockwise (i.e., p is on left side of \overline{ab}). A point p is inside a triangle abc (where abc is clockwise), if it is located on right side of all line segments \overline{ab} , \overline{bc} and \overline{ca} .

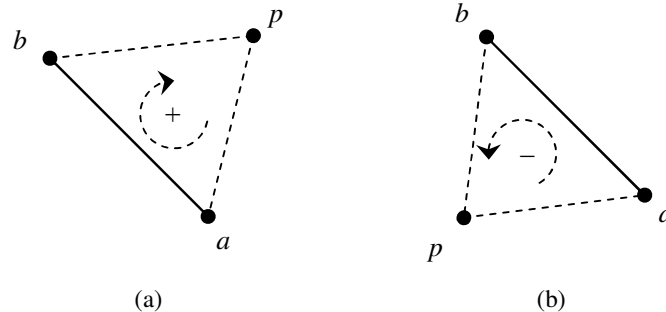


Figure 2.8. Position of a point with respect to a line segment: (a) positive (b) negative

To find a triangle that contains the new vertex, we can simply check all the triangles in order, until the containing triangle is reached. However, by using an algorithm called *walking*, this triangle can be found faster. As Figure 2.9 shows, this algorithm walks directly through the containing triangle. For this, if the new vertex is not in a triangle T , the next triangle to be checked is the one sharing the edge with T that makes a counter-clockwise order with the new vertex (Guibas and Stolfi, 1985; Stolfi, 1989a, b).

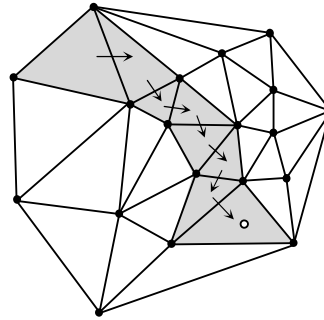


Figure 2.9. *Walking* algorithm to detect the triangle contains the new vertex

2.1.3.1.2 Insertion

In this step, the containing triangle is replaced with three new triangles that pass through the new vertex (Figure 2.10).

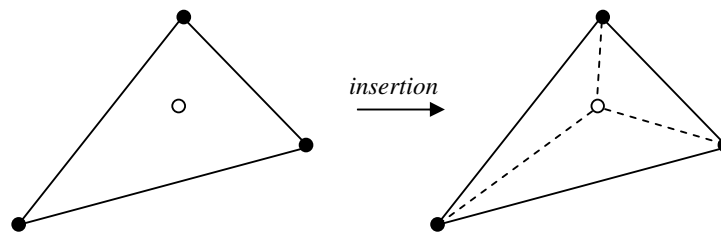


Figure 2.10. Inserting a new vertex into a triangle

2.1.3.1.3 Update

The three new triangles, created by the insertion process in the previous step, are pushed in a stack. Each element of the stack is checked against the circum-circle property. For a triangle $T = \langle a, b, c \rangle$ and a point p , the circum-circle property is satisfied, if the point p does not lie in the circum-circle of the triangle T . Its extension to 3D, called circum-sphere property, estates that the point p does not lie in the circum-sphere of the tetrahedron $T = \langle a, b, c, d \rangle$. The following determinants are used to test the circum-circle and circum-sphere properties for 2D and 3D cases, respectively (Guibas and Stolfi, 1985):

$$h = \begin{vmatrix} x_a & y_a & x_a^2 + y_a^2 & 1 \\ x_b & y_b & x_b^2 + y_b^2 & 1 \\ x_c & y_c & x_c^2 + y_c^2 & 1 \\ x_p & y_p & x_p^2 + y_p^2 & 1 \end{vmatrix} \quad \text{2D} \quad \begin{vmatrix} x_a & y_a & z_a & x_a^2 + y_a^2 + z_a^2 & 1 \\ x_b & y_b & z_b & x_b^2 + y_b^2 + z_b^2 & 1 \\ x_c & y_c & z_c & x_c^2 + y_c^2 + z_c^2 & 1 \\ x_d & y_d & z_d & x_d^2 + y_d^2 + z_d^2 & 1 \\ x_p & y_p & z_p & x_p^2 + y_p^2 + z_p^2 & 1 \end{vmatrix} \quad \text{3D} \quad (2-4)$$

A positive value for h indicates that p is inside the triangle abc , while it is outside if h is negative. If this property is not satisfied, then the triangle and its adjacent are flipped and the new triangles are pushed in the stack. This process continues until there is no element left in the stack. Figure 2.11 shows the entire process of inserting a new vertex into a Delaunay triangulation using the flipping algorithm.

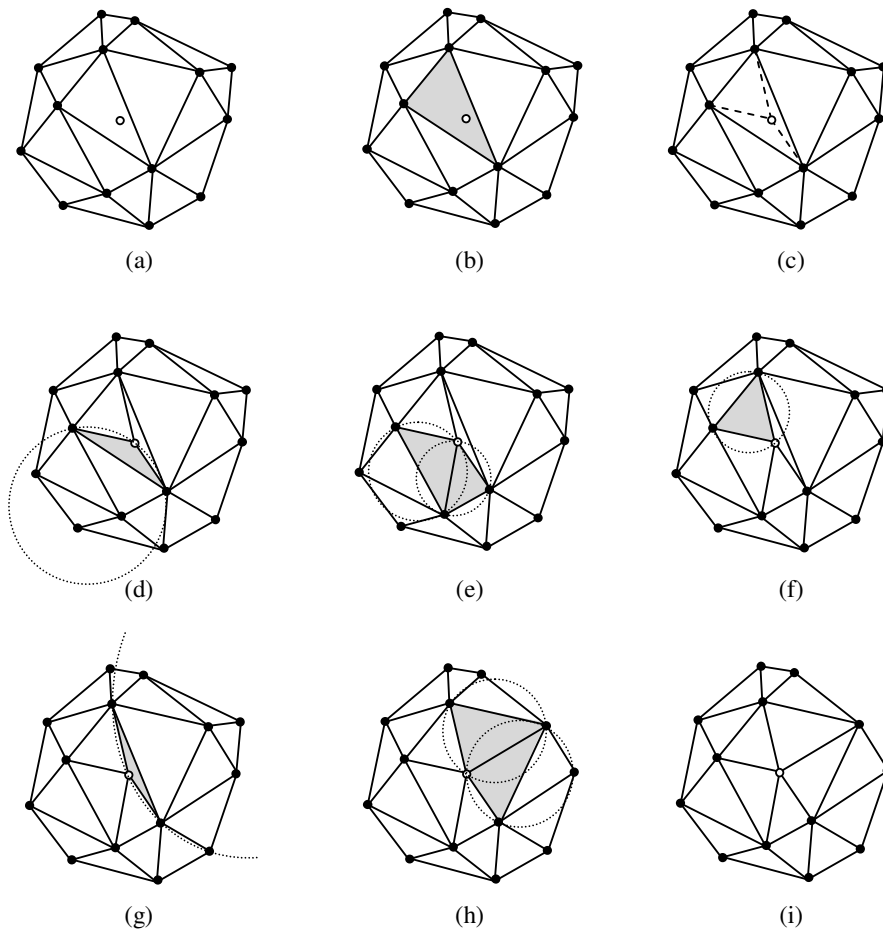


Figure 2.11. Flipping algorithm to insert a vertex in a DT: (a) Initial DT and the new vertex. (b) Detecting the triangle that contains the new vertex and (c) inserting the new vertex into it. (d) to (h) Checking the circum-circle property and applying flipping if required. (i) new DT that contains the inserted vertex (Ledoux, 2007).

The complexity of *flipping* algorithms is $O(n \log n)$ in 2D, where n is the number of input points: The number of triangles is $k*n$ (k is a constant); The procedure runs for each point and in each iteration, the *walking* algorithm finds the containing triangle in $k*\log n$ (k is a constant); and a constant number of adjacent triangles are updated, which altogether make the complexity $O(n \log n)$.

2.1.3.1.4 Extension to 3D

The general steps of flipping algorithm for 3D points are the same as 2D, but the details must be adopted:

- A big tetrahedron that contains all of the vertices is created (Figure 2.12).

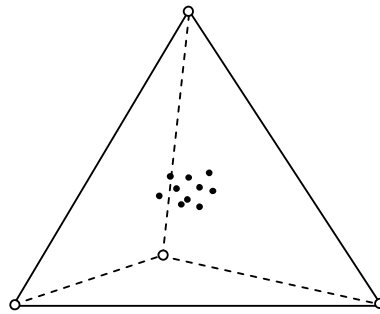


Figure 2.12. A big tetrahedron that contains all of the vertices

- To find the tetrahedron that contains a new vertex, the *walking* algorithm is adopted to interact with tetrahedra instead of triangles. The following determinant is used to determine the position of the new vertex respect to the triangular faces of each tetrahedron (Figure 2.13).

$$D = \begin{vmatrix} x_a & y_a & z_a & 1 \\ x_b & y_b & z_b & 1 \\ x_c & y_c & z_c & 1 \\ x_p & y_p & z_p & 1 \end{vmatrix} \quad (2-5)$$

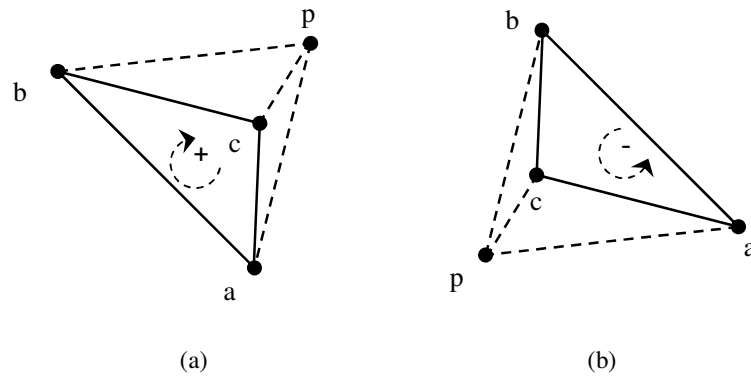


Figure 2.13. Position of a point respect to a tetrahedron: (a) positive (b) negative

- To insert a vertex into a tetrahedron, it is replaced with three new tetrahedra that pass through the new vertex (Figure 2.14).

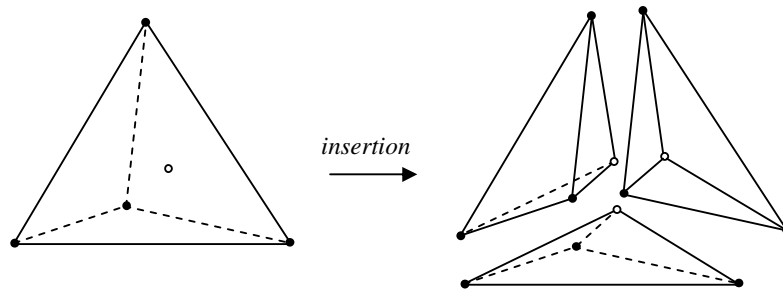


Figure 2.14. Inserting a new vertex into a tetrahedron

- To update the tetrahedralization, the *push-and-pop* process as 2D is performed, and the concept of flipping is generalized. Flipping, however, is different in 3D (Joe, 1989, 1991; Lawson, 1986), which prevents a dimension independent implementation. To tetrahedralize five 3D points, there are two possible solutions: one has two tetrahedra and the other has three (Figure 2.15). Flipping between the two configurations are called *flip23* and *flip32*, regarding the number of tetrahedra exist before and after the flip operation. Moreover, according to the geometry of a tetrahedron in the Delaunay triangulation with its adjacent, it is not always possible to perform a flip. It is the case when the union of two tetrahedra is concave (Figure 2.16). In such cases, no action is taken because the required flip will be performed by a later element in the stack. For more information, see (Edelsbrunner and Shah, 1992; Ledoux, 2006, 2007; Shewchuk, 2003).

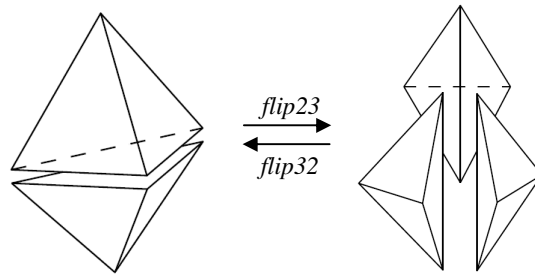


Figure 2.15. Two possible tetrahedralizations of five 3D points (Ledoux, 2007)

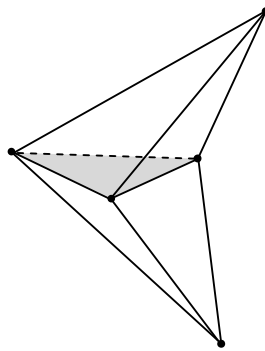


Figure 2.16. A situation when the union of two tetrahedra is concave, so not flippable

The triangulation of n points in 3D have $k \cdot n^2$ tetrahedra (Ledoux, 2007). After insertion of each point, the *walking* algorithm finds the containing tetrahedron in $k \cdot n$ (k is a constant); and a constant number of adjacent tetrahedra are updated, which altogether make the complexity $O(n^2)$.

2.1.3.2 Bowyer-Watson algorithm

This is a dimension independent incremental algorithm introduced by Bowyer and Watson (Bowyer, 1981; Kanaganathan and Goldstein, 1991; Watson, 1981). To add a vertex in a 2-dimensional Delaunay triangulation, all the triangles that violate the circum-circle property, i.e., whose circum-circle contains the new vertex (Figure 2.17.a), are deleted from the construction (Figure 2.17.b). This creates a hole, which is filled by new triangles that are created by joining the new vertex to each edge of the boundary of the hole (Figure 2.17.c).

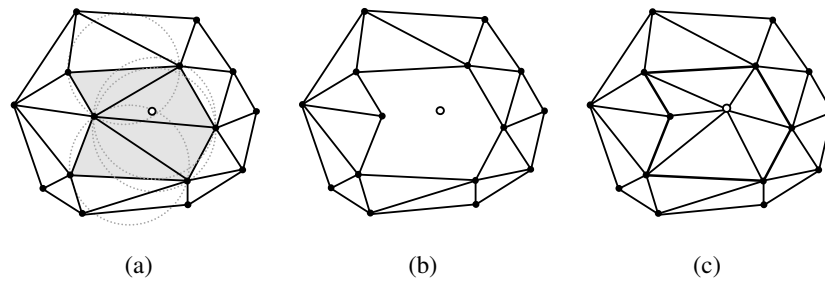


Figure 2.17. Inserted vertex, indicated as white, is added to DT: (a) and (b) all triangles whose circum-circle contains the new vertex are detected and deleted (c) The created hole is filled by new triangles, which are created by joining the new vertex to each edge of the boundary of the hole (Ledoux 2006).

2.1.3.2.1 Extension to 3D

To construct the Delaunay triangulation of a set of 3D points using the *Bower-Watson* algorithm, after each insertion, all the tetrahedra whose circum-sphere contain the new vertex are deleted, and the hole is filled by new tetrahedra that are created by joining the new vertex to each triangle of the boundary of the hole (Field, 1986; Watson, 1981).

The complexity of the *Bowyer-Watson* algorithms is $O(n \log n)$ in 2D and $O(n^2)$ in 3D, where n is the number of input points: The number of triangles is $k*n$ (k is a constant) in 2D and $k*n^2$ in 3D. The procedure runs for each point and in each iteration, the *walking* algorithm finds the triangle that contains the new point in $k*\log n$ and $k*n$ iterations for 2D and 3D, respectively (k is a constant). Regarding the fact that the violating triangles are connected, after detecting the first violating triangle, its adjacent triangles are checked and it continues until all of the adjacent triangles satisfy the test. Thus, the complexity of the *Bowyer-Watson* algorithm is $O(n \log n)$ in 2D and $O(n^2)$ in 3D.

2.2 Dynamic spatial analyses

In a *dynamic* set of points, the position of the points is fixed, but the number of the points may change over time, i.e., points may be inserted into or deleted from the point set.

Suppose that a structure, say, Delaunay triangulation is constructed for a set of objects. If a new object is inserted into or deleted from the set, the straightforward approach to update the structure is to reconstruct the whole structure from the scratch. Although this is simple, it is very inefficient; because usually inserting or deleting an object leaves a significant part of the structure unchanged, so most of the previous calculations are unnecessarily repeated.

A more efficient approach is to locally update the structure. In other words, only that part of the structure affected by the insertion or deletion is reconstructed. It is obvious that the affected part is different from an analysis to another, so each analysis needs its own updating strategy. In the following, we show how to locally update the Delaunay triangulation after insertion or deletion of a vertex.

2.2.1 Dynamic Delaunay triangulation

To insert a vertex in a Delaunay triangulation, the incremental algorithms proposed to construct the Delaunay triangulation, i.e., *flipping* and *Bowyer-Watson* algorithms can be properly used. These algorithms insert a vertex to an existing Delaunay triangulation and locally update the structure.

Deleting a vertex v from a Delaunay triangulation can be considered as the inverse problem of inserting a vertex in a Delaunay triangulation: The vertex v and all triangles incident to v are removed and the created hole is re-triangulated (Figure 2.18).

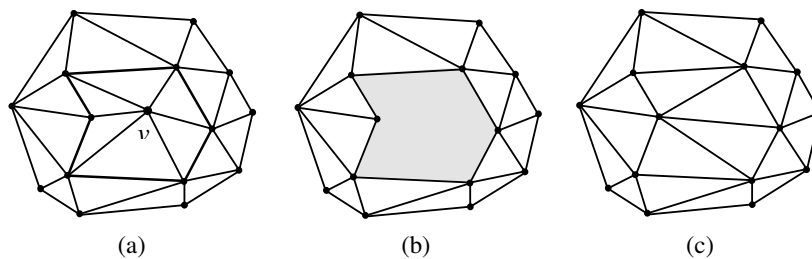


Figure 2.18. Deleting a vertex v from a DT: (a) DT Before deletion (b) The hole created by deleting the incident triangles (c) Re-triangulating the hole (Ledoux et al., 2005)

Heller (1990) proposed an algorithm to delete a vertex from a 2-dimensional Delaunay triangulation. In his algorithm, the ears of the vertex v are examined in counter-clockwise order (Figure 2.19.b) and the one with the smallest circum-circle (Figure 2.19.c) is flipped with its adjacent triangle with which it shares a link edge (Figure 2.19.d). This reduces the number of neighbors of v by one. The process continues until only three triangles left (Figure 2.19.e). Then, v is removed and the three triangles merged into one (Figure 2.19.f) (Heller, 1990).

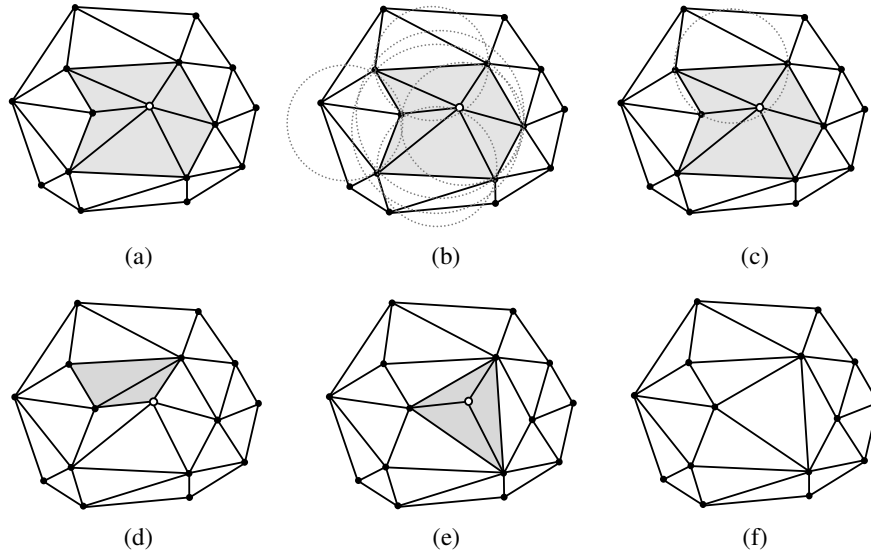


Figure 2.19. Heller algorithm to delete a vertex, indicated as white, from DT: (a) The initial DT. (b) Circum-circles of the triple neighboring vertices that form an imaginary triangle. (c) and (d) Flipping the imaginary triangle with the smallest circum-circle with its adjacent triangle. (e) Repeating the process until only three triangles left. (f) Removing the vertex and merging the three triangles into one.

Heller assumption was that among all the potential ears, the one with the smallest circum-circle has no other vertex inside and so could become a real triangle. However, Devillers (2002) showed, through a counter-example, that this assumption is wrong. Instead, he suggested ordering the ears (imaginary triangles) based on the power of the vertex to be removed with respect to that ear. This parameter is computed as follow (Devillers, 2002):

$$power(\langle a, b, c \rangle, v) = \frac{H(\langle a, b, c \rangle, v)}{D(a, b, c)} \quad (2-6)$$

where

$$D(a, b, c) = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix} \quad H(\langle a, b, c \rangle, v) = \begin{vmatrix} x_a & y_a & x_a^2 + y_a^2 & 1 \\ x_b & y_b & x_b^2 + y_b^2 & 1 \\ x_c & y_c & x_c^2 + y_c^2 & 1 \\ x_v & y_v & x_v^2 + y_v^2 & 1 \end{vmatrix} \quad (2-7)$$

It is proved that Devillers algorithm works for any dimensions (Devillers and Teillaud, 2003).

Mostafavi et al. (2003) proposed an algorithm that does not apply any order on the imaginary triangles. Instead, they test the imaginary triangles one by one, and if it is a valid imaginary triangle, it is flipped with its adjacent. An imaginary triangle $T = (v_1, v_2, v_3)$ is invalid if at least one of the following statements hold:

- $D(v_1, v_2, v_3)$ is negative. It means that T does not form an ear.
- $D(v_1, v_3, v)$ is negative, where v is the vertex to be deleted. It means that T encloses v .
- $H(\langle v_1, v_2, v_3 \rangle, x)$ is positive for at least one of the neighboring vertexes x . It means that there is, at least, one neighboring vertex that lies inside the circum-circle of T .

Although this algorithm is simpler, it becomes less efficient as the number of neighbors increases. However, this algorithm is equally efficient up to eight neighbors, which is mostly the case (Mostafavi et al., 2003).

To extend this algorithm to 3D, recall that there are two types of ears in 3D: 2-ears and 3-ears. Let \mathbf{P} be a polyhedron that is made up of triangular faces. A 2-ear is formed by two adjacent triangular faces abc and bcd sharing the edge bc (Figure 2.20.a); and a 3-ear is formed by three adjacent triangular faces abd , acd and bcd sharing the vertex d (Figure 2.20.b). A 2-ear is valid if and only if the line segment ad is inside \mathbf{P} ; and a 3-ear is valid if and only if the triangular face abc is inside \mathbf{P} . In the case of the deletion of a vertex v in a Delaunay triangulation, \mathbf{P} is a star-shaped polyhedron $star(v)$. An ear of $star(v)$ is valid if it is convex outwards from v .

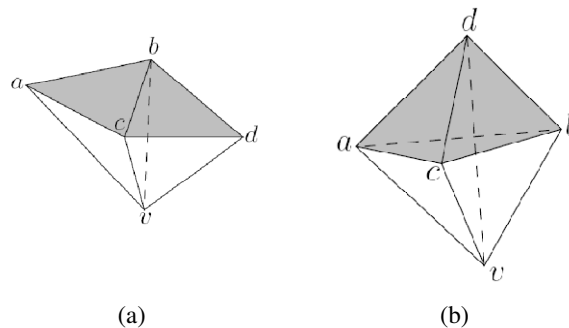


Figure 2.20. (a) 2-ear (b) 3-ear

Now, to delete a vertex v from a 3-dimensional Delaunay triangulation, all the ears of $star(v)$ are built and stored in a simple list. An ear e from the list (any ear) is popped. The

ear e is flipped if respects these three conditions: e is *valid*, *flippable* and *locally Delaunay* (Ledoux et al., 2005). For more details on a flippable tetrahedron, see (Ledoux, 2007). An ear e is locally Delaunay if its circum-sphere does not contain any other points on the boundary of $star(v)$.

Another approach suggested by Schaller and Meyer-Hermann (2004) moves the vertex to be deleted towards its nearest neighbor gradually and update the structure until the triangles between the two vertexes are very flat and can be clipped out of the triangulation without harming its validity. Figure 2.21 illustrates the idea of this algorithm. The updates are performed using the existing algorithms for kinetic Delaunay triangulation, which will be presented in the next section.

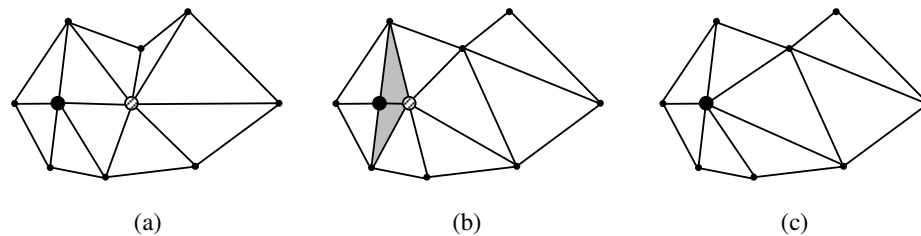


Figure 2.21. Delete a vertex from DT: (a) The vertex to be deleted (large hatched point) is moved gradually toward its nearest neighbor (large black point) and the DT is updated by flipping when required. (b) The movement continues until the inner simplexes (shaded region) can be safely deleted. (c) The two vertices are simply merged and the remaining opposing simplexes are connected as neighbors.

2.3 Kinetic spatial analyses

A *kinetic* or *moving* point is a point whose position changes over time, i.e., its position is a function of time:

$$\mathbf{P} = (p_1, p_2, \dots, p_n) = (f_1(t), f_2(t), \dots, f_n(t)) \quad (2-8)$$

Once insertion and deletion of an object have been implemented for a structure, the intuition to handle a moving object is that the object is deleted from the current position and re-inserted at the new position; after each deletion and insertion, the structure is updated. Although it is a very simple approach, it is computationally expensive because several unnecessary deletions and insertions are performed: an object is deleted and re-inserted, while this movement may not affect the topology of the structure, so it is not really kinetic. This approach nevertheless is an appropriate solution for many applications where the

intermediate states are not important (just the start and end states are of interest): the object is deleted from the start and re-inserted at the end.

A more efficient approach is using *event-based* methods. They are based on the concept of topological events, which is defined as “*for a structure D consisting of moving elements S , a topological event t is the moment when the movement of elements S change the topology of D* ”. Based on this concept, to handle the movement of an object in a structure, the topology of the structure is updated at topological events; elsewhere, only the geometry of the structure is modified, which does not need any computations. It is obvious that the topological events are different for each analysis, so each analysis needs its own movement handling strategy. In the following, some methods for moving the points in the Delaunay triangulation are presented.

2.3.1 Kinetic Delaunay triangulation

De Fabritiis and Coveney (2003) presented an approach to move the vertexes of a Delaunay triangulation: they gradually move the vertexes toward their destinations. After each movement, the triangles that violate the circum-circle property are detected and flipped. In 2D, each triangle T is checked with all of its neighbors. If the opposite vertex of a neighboring triangle T' lies in the circum-circle of T , then T and T' are flipped and put in a stack to be checked with their new neighbors. This process continues until there is no element left in the stack.

The idea of this approach is based on “delete and re-insert”, but it has some level of intelligence: a simple check is applied on all elements (triangles or tetrahedra here), but further operations (i.e., *flip*) are applied only when it is required. However, the main drawback is still there: This method uses a fixed time step to move all of the vertexes, no matter if this movement topologically affects the structure or not.

Extension of this method to 3D needs two types of check because two types of flips (*flip23* and *flip32*) are possible (Schaller and Meyer-Hermann, 2004):

- Each tetrahedron T is checked with its neighbor T' and a *flip23* is performed if the following two conditions are met:
 - The opposite vertex of the neighbor T' lies within the circum-sphere of T .
 - The five union vertexes of T and T' form a convex polyhedron.

- Each tetrahedron T is checked with two of its neighbors T_1 and T_2 and a *flip32* is performed if the following two conditions are met:
 - All of the pairs TT_1 , TT_2 and T_1T_2 violate the circum-sphere property.
 - T_1 and T_2 are also respective neighbors.

Another extension of this approach to 3D is that instead of performing a sequence of flips on the tetrahedra in order to locally restore the circum-sphere property, the validity of this property is checked for all the tetrahedra. The vertexes for which this property fails are moved back to the preceding position and then “delete and re-insert” is applied (De Fabritiis and Coveney, 2003).

To use the event-based update to move a vertex in a Delaunay triangulation, let p be a vertex in a Delaunay triangulation DT and P be the set of its neighboring vertexes, in clockwise order. Let T_r be the set of opposite triangles (tetrahedra in 3D) of p , i.e., neighbors of incident triangles (tetrahedra in 3D) to p , and T_i be the set of imaginary triangles (tetrahedra in 3D) that could be drawn from three (four in 3D) successive elements of P (Figure 2.22). Then, the topological events of DT caused by the point p are defined as follows (Albers et al., 1998; Ledoux, 2008; Mostafavi, 2002; Roos, 1991):

- If p moves in the circum-circle (-sphere in 3D) of an element of T_r (Figure 2.23), a flip is performed and the new triangles (tetrahedra in 3D) are updated (i.e., they are checked with their neighbors against the circume-circle (-sphere in 3D) property).
- If p moves out of the circum-circle (-sphere in 3D) of an element of T_i (Figure 2.24), a flip is performed and the new triangles (tetrahedra in 3D) are updated.

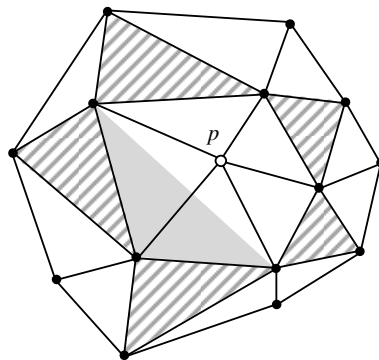


Figure 2.22. Hashed triangles are the opposite triangles of p . Shaded triangle is one of the imaginary triangles that could be drawn from three successive neighbors of p .

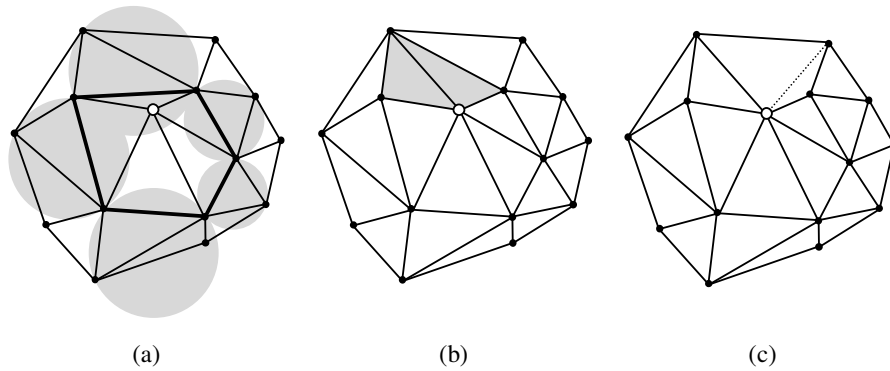


Figure 2.23. (a) The white point moves in the circum-circle of an opposite triangle. (b) The two triangles are flipped. (c) Final DT

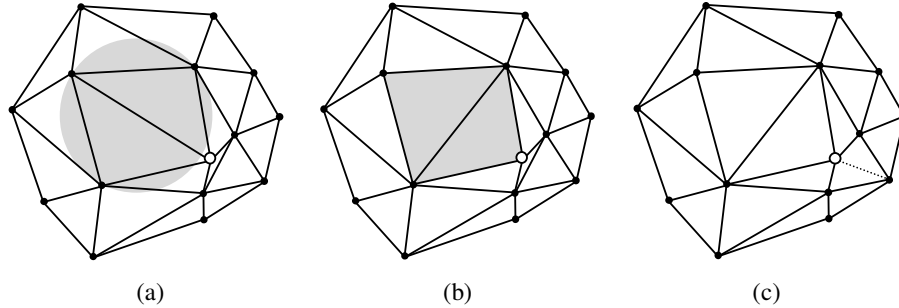


Figure 2.24. (a) The white point moves out of the circum-circle of an imaginary triangle (b) The two triangles are flipped. (c) Final DT

Roos (1991) proposed an algorithm to update a 2-dimensional Delaunay triangulation based on the concept of topological events. All the topological events for all the quadrilaterals (pair of adjacent triangles in the Delaunay triangulation) are computed and put in a priority queue, sorted according to the time they will arise. The time is computed by finding the zeros of the circum-circle equation developed into a polynomial. Then, the first topological event is popped from the queue, the Delaunay triangulation is modified with a *flip22*, and the queue is updated because the flip has changed locally some triangles. The algorithm continues until there are no topological events left in the queue (Guibas et al., 1992; Roos, 1991, 1993; Roos and Noltemeier, 1991). Similar algorithms have been proposed in (Bajaj and Bouma, 1990; Imai et al., 1989).

This algorithm has been extended to 3-dimensional Delaunay triangulation in (Albers, 1991; Albers et al., 1998; Albers and Roos, 1992). However, it is not very efficient in 3D because calculating the zeros of the circum-sphere equations cannot be done analytically, as is the case for the circum-circle equations (Gavrilova and Rokne, 2003; Vomacka, 2008).

Indeed, the polynomial for the 3-dimensional case has a high degree (8th degree) and iterative numerical solutions must be sought. That results in a much slower implementation, and it could also complicate the update of the Delaunay triangulation when the set of points contains degeneracies.

Mostafavi (2001) proposes a different algorithm and gives more implementation details. He focuses on the operations necessary to move a single point p , and then explain how to move many points (see section 2.4). In this algorithm, the topological events caused by a single point p are detected by intersecting the trajectory of the point p with the opposite and imaginary circum-circles, which were explained above (Gold, 1990; Gold et al., 1995; Mostafavi, 2002; Mostafavi and Gold, 2004). This algorithm has been equally extended to 3-dimensional Delaunay triangulation in (Ledoux, 2008).

2.4 Event based approach to move several objects

To move several objects in a structure based on the event based approach, the topological events of all objects must be determined and sorted based on the event time. Then, they are applied in the structure in order.

Note that objects may start moving at different times and so the events have different time origins. Therefore, they must be synchronized before sorting. For this, a global time scale G is considered whose origin is the occurrence of the first event. Then, three types of time are defined for topological events (Hashemi-Beni et al., 2007; Ledoux, 2008):

- ${}^e t_i^i$: The time between the topological event e and the topological event i . If d is the distance of the current position of the object to the topological event i , and the object is moving with a constant velocity v , then ${}^e t_i^i = d / v$.
- ${}^e t_g^i$: The time, in the time scale G , of the occurrence of the topological event i .
- ${}^e t_c$: The time, in the time scale G , passed from the start of the movement of the object.

The relation between the above times defined for topological events is (Figure 2.25):

$${}^e t_g^i = {}^e t_c + {}^e t_i^i \quad (2-9)$$

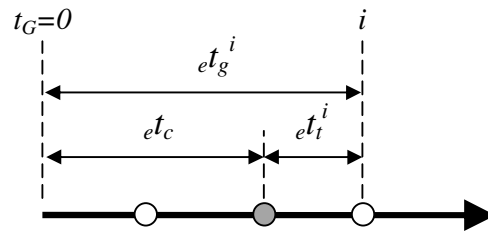


Figure 2.25. Relation between different times defined for topological events

Now, et_g^i is a synchronized event time that can be computed for all topological events using the Equation (2-9).

2.5 Discussion

The algorithms to perform spatial analyses can be classified as follows:

- *Dimension specific algorithms*: These algorithms are designed for a specific dimension and use the characteristics of the objects of that dimension in their definition, so they cannot be extended to any other dimension. The *Graham-scan* algorithm to compute the convex hull of 2D points (Berg et al., 2008; Karimipour et al., 2008) is an example of such algorithms.
- *n-dimensional algorithms*: These algorithms can be developed for objects of different dimensions. The general procedure is similar for any dimension, but the details are different. For example, in the *flipping* algorithm, presented in section 2.1.3.1, to construct Delaunay triangulation, the three step procedure *walk-insert-update* is applied to each 2D and 3D vertex, but the details are different in each dimension, especially for *flipping*. Because of such differences, they are separately implemented for each dimension.
- *Dimension independent algorithms*: These algorithms are dimension independent in their definition, such as *Bower-Watson* algorithms to construct Delaunay triangulation, presented in section 2.1.3.2. Although the definitions are independent of dimension, the data types and operational details are different. Therefore, because of lack of efficient geometric data structures, they are still implemented separately for each dimension. From an implementation point of view, there is no advantage in using dimension independent algorithms comparing to *n-dimensional* algorithms.

On the other hand, investigating the approaches presented in sections 2.2 and 2.3 to delete, insert or move an object in a structure, they efficiently update the structure so that the effected part is detected and locally updated. However, the detection and updating strategies differ for each analysis, which results in a new technique for extension of each analysis (Guibas, 1998; Guibas et al., 2004).

Considering the above discussion, the current approaches to extend spatial analyses to higher dimensions depend on the dimension and analysis, and aim to find the fastest algorithm to perform a certain analysis in a specific dimension (Boissonnat et al., 1998; Edelsbrunner, 1987; Goodman and O'Rourke, 1997; Skiena, 1998) (Table 2.1). The advantage is that the characteristics of the analysis and the dimension at hand are considered in designing the algorithm, so the ultimate simplicity and a fast implementation is achieved (CGAL website).

To establish a practical GIS, that supports a variety of spatial analyses, performance evaluation should not be limited to speed because it leads to extension strategies that depend on the dimension and the analysis, which results in recoding each spatial analysis for each dimension in the software development stage.

This research focuses on how to extend the spatial analyses to different dimensions with the smallest amount of recoding. That is, here the evaluation is on how much recoding is needed to extend an implemented spatial analysis to another dimension. Of course, this approach will cause losing a significant amount of information available for specific dimensions, so it may not provide the simplest and fastest solutions. In other words, this research believes that it is better to have a working comprehensive system, even if it is slow, than waiting for a fast system created in unknown future. Note that base on the Moor's law, computer speed doubles every 18 months on average!

Table 2.1. Some research to perform Delaunay triangulation in different dimensions

Data type		Title of the research	Reference
2D	Static	Computing dirichlet tessellation	(Bowyer, 1981)
		Numerical stability of algorithms for 2D Delaunay triangulations	(Fortune, 1992)
		A fast divide and conquer Delaunay triangulation algorithm	(Cignoni et al., 1998)
		A sweepline algorithm for Voronoi diagrams	(Fortune, 1987)
	Dynamic	Dynamic Voronoi diagrams and Delaunay triangulations	(Bajaj and Bouma, 1990)
		Delete and insert operations in Voronoi/Delaunay	(Mostafavi et al., 2003)
		Triangulation algorithms for adaptive terrain modelling	(Heller, 1990)
		Fully dynamic constrained Delaunay triangulations	(Kallmann et al., 2003)
	Moving	Delaunay triangulation of moving points	(Vomacka, 2008)
		Voronoi diagrams of moving points	(Albers et al., 1998)
		Voronoi diagrams of moving points in the plane	(Guibas et al., 1992)
		Voronoi diagrams of moving points in the plane	(Fu and Lee, 1991)
		Point location in the moving VD and related problems	(Devillers and Golin, 1993)
Dynamic Voronoi diagrams in motion planning	(Roos and Noltemeier, 1991)		
3D	Static	Implementing Watson's algorithm in three dimensions	(Field, 1986)
		Computing the 3D Voronoi diagram robustly	(Ledoux, 2007)
		Fully Incremental 3D Delaunay Refinement Mesh Generation	(Miller et al., 2002)
		A comparison of five implementations of 3D DT	(Liu and Snoeyink, 2005)
	Dynamic	Three-dimensional dynamic Voronoi diagrams	(Albers, 1991)
		Kinetic and dynamic Delaunay tetrahedralizations in three dimensions	(Schaller and Meyer-Hermann, 2004)
		Dynamic Voronoi diagrams	(Roos, 1991)
		On deletion in Delaunay triangulations	(Devillers, 2002)
		Flipping to robustly delete a vertex in a Delaunay tetrahedralization	(Ledoux et al., 2005)
		Perturbations and vertex removal in a 3D Delaunay triangulation	(Devillers and Teillaud, 2003)
	Moving	Voronoi diagrams of moving points in higher dimensions	(Albers and Roos, 1992)
		The kinetic 3D Voronoi diagram	(Ledoux, 2008)
		Kinetic and dynamic Delaunay tetrahedralizations in 3D	(Schaller and Meyer-Hermann, 2004)
		The kinetic 3D Voronoi diagram: a tool for simulating environmental processes	(Ledoux, 2008)
		Moving objects management in a 3D dynamic environment	(Hashemi-Beni et al., 2007)

2.6 Summary

In this chapter we reviewed the state-of-the-art in extending spatial analyses to higher dimensions and show how the current approach is applied to Delaunay triangulation, as the case study of the research. The current approach and the proposed approach of the research were compared.

Current approach to extend spatial analyses to higher dimensions depends on the dimension and analysis and aims to find the fastest algorithm to perform a certain analysis in a specific dimension. The result of following such an approach in the software development stage is recoding each spatial analysis for each dimension. We believe it is better to have a working comprehensive system, even if it is slow, than waiting for a fast system created in unknown future!

3 FORMAL METHODS

This chapter explains the formal methods used in this thesis to extend spatial analyses to different multi-dimensional spaces:

- The abstraction concepts needed to construct an integrated framework of spatial analyses are introduced, which leads to define spatial analyses based on their dimension independent properties.
- We explain the algebraic structures as the required abstraction to formally define spatial analyses as combination of the elements of an integrated framework.
- The simplicial complexes are introduced as an n -dimensional data type that enables dimension independent implementations.
- Finally, we introduce the functional programming languages and explain why they are used in this thesis. The main concepts of functional programming languages (especially their evaluation strategies) are described to an extent necessary for argue the implementations provided in the thesis.

3.1 Abstraction

In a broad meaning, abstraction is the process of generalizing a concept through reducing its information content. In computer science, abstraction is defined as removing the behavioral details of different objects to the lowest common denominator so that they can be interacted in the same manner. It is achieved by ignoring those characteristics that depends on the data types. The processes of this abstract level can be performed similarly because all the data types have similar characteristics in this level (Liskov and Guttag, 1988; Nordstrom et al., 1990; Pierce, 2005; Pierce, 2002). As shown in Figure 3.1, abstraction could be considered as a many-to-one mapping that maps different data types to a representative *abstract data type*, abbreviated *ADT* (Liskov and Guttag, 1988; Loeckx et al., 1996).

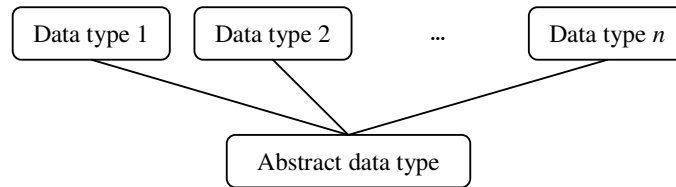


Figure 3.1. Abstract data type as a many-to-one mapping

3.1.1 Types of Abstraction

There are three type of abstraction and they are presented in the followings:

3.1.1.1 Procedural abstraction

Procedural abstraction specifies the number, types and the order of the inputs and output of a process. For example:

sort: a list of type $a \rightarrow$ a list of type a
 equality and order are defined on the elements of type a
effect: sorts the elements of the input list

(3-1)

says that the *sort* function sorts the elements of an input list on which equality and order are defined. However, it does not determine how to compute it. In procedural abstraction, the definition represented for application of a process is independent of characteristics of its elements and operations. For instance, using each of the definitions represented for equality and order in Equations (3-2) and (3-3) has no effect on the abstract definition of the *sort* function.

$$\begin{aligned}
 (a,b) = (c,d) &\Leftrightarrow a = c \wedge b = d \\
 (a,b) = (c,d) &\Leftrightarrow a = c \vee b = d \\
 (a,b) = (c,d) &\Leftrightarrow a = c
 \end{aligned}$$

(3-2)

$$\begin{aligned}
 (a,b) < (c,d) &\Leftrightarrow a < c \text{ and } b < d \\
 (a,b) < (c,d) &\Leftrightarrow a < c \\
 (a,b) < (c,d) &\Leftrightarrow a + b < c + d
 \end{aligned}$$

(3-3)

3.1.1.2 Data abstraction

Data abstraction defines a data type, a set of operations upon that data type, and specifies the relation between the data type and the operations. These together represent the characteristic of that data type.

The characteristic of a data type is specified through operations that can be applied on the objects of that data type. For instance, for the data type *queue*, the operations *push* (to add an element to the end of the queue), *pop* (to take the head of the queue), *size* (to get the number of elements of the queue), and *isNull* (to test if the queue is null) are defined.

3.1.1.3 Iterative abstraction

Iterative abstraction is used to avoid details of applying a process on iterative data types (e.g., sets, queues, lists, etc.). In other words, iterative abstraction specifies the elements of an iterative data type on which a process must be applied, but it does not specify the order of elements to be processed and how they are affected. For example, both “apply a function on all elements of a list” and “filtering those elements of a list that satisfy a certain criterion” can be defined using the following abstract representation:

```
for all elements of the set
do action
```

(3-4)

3.2 Algebraic structures

This section introduces algebraic structures and their related concepts. Definition of algebra and algebraic structures as well as their mappings are presented.

3.2.1 Definition of algebra

An algebra consists of a collection of elements, operations upon those elements, and axioms which are capable of expressing the interaction between operations and elements (Loeckx et al., 1996; MacLane and Birkhoff, 1999). An algebra G with elements S and operations W is denoted as $G[S, W]$.

An algebraic structure is a set of elements and operations that obey the rules of a certain algebra. i.e., these elements and operations can be substituted with their corresponding in the algebra. Algebraic structures describe structure independently of any

implementation and prior understanding. Thus, the same algebraic structure can describe the behavior of different things if their behavior is structurally equivalent (Dorst et al., 2007; MacLane and Birkhoff, 1999). For instance, although Roman numbers (I, II, III, ...) and Arabic numbers (1, 2, 3, ...) are two different representations of natural numbers, their elements and operations are structurally equivalent and so natural numbers algebra can describe both. *Sets, Groups, Rings, Fields, and Boolean algebra* are examples of algebraic structures. For more details, see (MacLane and Birkhoff, 1999).

3.2.2 Mappings between algebras

Two different concepts with equivalent structures could be mapped together through a mapping, called *homomorphism*. Homomorphisms are structure preserving mappings, i.e., a homomorphism maps corresponding elements and operations while preserving the structure (MacLane and Birkhoff, 1999). For instance, f in Figure 3.2 maps the elements and the $+$ operation of Roman numbers to their corresponding in Arabic numbers; and the structure of operation $+$ is preserved through this mapping.

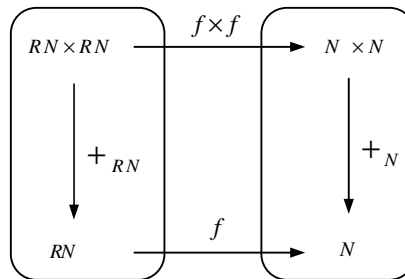


Figure 3.2. Mapping elements and $+$ operation of Roman numbers to their correspondences in Arabic numbers

Homomorphisms are commutative (Frank, 2007; MacLane and Birkhoff, 1999): for certain origin and destination, the result of mappings is independent of the path. As Figure 3.2 shows, adding two Roman numbers by $+_{RN}$ and then applying the mapping f gives the same result with applying the mapping f on the two Roman numbers first and adding them by $+_N$.

Note that homomorphisms do not necessarily map similar elements and operations (Frank, 2007; MacLane and Birkhoff, 1999). For example, logarithm (\log) could be considered as a homomorphism that maps R^+ to R , \times to $+$, and $\sqrt{\quad}$ to $\times \frac{1}{2}$ (Figure 3.3).

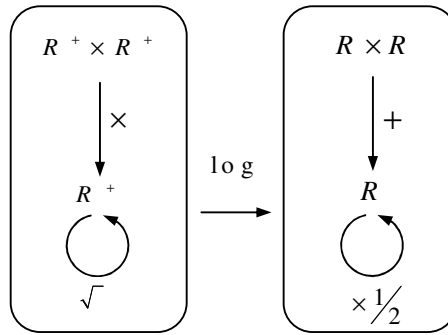


Figure 3.3. Mapping R^+ to R through homomorphism \log

Definition: Mathematically, a homomorphism, also called *lifting*, is defined as follows:

Let A and B be two different concepts represented by $G[S, W]$. Then, a homomorphism $h: A \rightarrow B$ is a set of mappings $(h_w)_{w \in W}$, where:

$$h_w : w_A \rightarrow w_B, \quad w \in W \tag{3-5}$$

and for any two corresponding operations

$$\begin{aligned} w_A \in W, \quad w_A : (s_1 \times \dots \times s_k) \rightarrow s, \quad k \geq 0 \\ w_B \in W, \quad w_B : (s_1 \times \dots \times s_k) \rightarrow s, \quad k \geq 0 \end{aligned} \tag{3-6}$$

the following equation is hold (Figure 3.4):

$$h(w_A(s_1, \dots, s_k)) = w_B(h(s_1), \dots, h(s_k)) \tag{3-7}$$

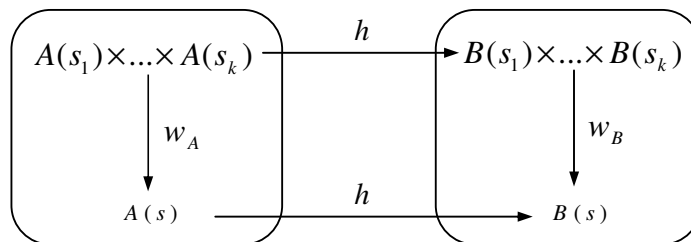


Figure 3.4. A homomorphism h between two concepts A and B with the same algebra

Equation (3-7) means that the result of applying the operation W_A on elements of A and then applying the mapping h yields the same result as applying the mapping h on elements of B and then applying the corresponding operation W_B , i.e., homomorphisms are commutative. For instance, Equation (3-7) for the homomorphism \log presented in Figure 3.3 is written as:

$$\begin{aligned}\log(x \times y) &= \log(x) + \log(y) \\ \log(\sqrt{x}) &= \frac{\log(x)}{2}\end{aligned}\tag{3-8}$$

Homomorphisms are functions, so they can be injective and surjective (Lawvere and Schanuel, 2005). Many of the homomorphisms studied in algebraic structures are bijective homomorphism, called *isomorphism*. If two structures A and B are related through an isomorphism, then A and B are *isomorphic*, denoted $A \cong B$. For isomorphic structures, the following relation holds:

$$A \in G \text{ and } A \cong B \Rightarrow B \in G\tag{3-9}$$

where G represents a certain algebraic structure. It means that for two structures A and B and an algebraic structure G , if A can be represented by G and the elements and operations of A and B are equivalent (i.e., A and B are isomorphic), then G can represent B , too.

3.2.3 Algebraic representation of an abstract data type

Above discussion leads to algebraic definition of an abstract data type as follow (Guttag and Horning, 1978):

“An abstract data type is a collection of different structures all of which are represented by the same algebra G and so they are isomorphic”.

This definition expresses that an abstract data type is a set of elements and operations of different data types that can be mapped together through an isomorphism. Regarding the Equation (3-9), if the elements and operations of a data type A represented by an abstract

data type D , are equivalent with the elements and operations of another data type B , then the abstract data type D can represent B , too.

3.3 n -simplexes: an abstract data type for geometry

An n -simplex S_n is formally defined as “the smallest convex set in an Euclidian space (denoted as R^m , with $n \leq m$), containing $n+1$ points v_0, \dots, v_n that do not lie in a hyperplane of dimension less than n ” (Hatcher, 2002). A simpler definition describes an n -simplex S_n as the simplest spanning geometric figure in the n -dimensional Euclidean space that contains $n+1$ points v_0, \dots, v_n of dimension n , providing that the vectors v_1-v_0, \dots, v_n-v_0 are linearly independent. An n -simplex S_n is represented by the list of its vertexes as:

$$S_n = \langle v_0, \dots, v_n \rangle \quad (3-10)$$

Each vertex itself is an n -dimensional point, so a detailed representation of an n -simplex is:

$$S_n = \langle (e_{01}, \dots, e_{0n}), \dots, (e_{n1}, \dots, e_{nn}) \rangle \quad (3-11)$$

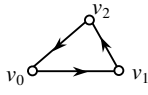
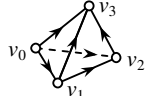
in which e_{ij} is the j^{th} defining coordinate of the i^{th} vertex. The n -simplexes are defined for any dimension. Table 3.1 shows 0- to 3-simplexes and their common names, representations and geometric configurations.

The concept of n -simplexes is extensively studied in the late 19th century by Henri Poincaré. It is the basis of the simplicial homology field, which is a part of algebraic or combinatorial topology (Hatcher, 2002).

For a given dimension n , an n -simplex is the elementary spatial object from which other complex objects of that dimension are constructed. Any subset of the vertexes of S_n represents a face of S_n . A simplicial complex C is a finite set of simplexes that satisfies the following conditions (Figure 3.5):

- Any face of a simplex from C is also in C .
- The intersection of any two simplexes $s_1, s_2 \in C$ is either empty or a face of both s_1 and s_2 .

Table 3.1. 0- to 3-simplexes and their common names, representations and geometric configurations

Dimension	Name		Representation	Geometric Configuration
0	<i>0-simplex</i>	Point	$S_0 = \langle v_0 \rangle$	$v_0 \circ$
1	<i>1-simplex</i>	Line segment	$S_1 = \langle v_0, v_1 \rangle$	$v_0 \circ \text{---} \circ v_1$
2	<i>2-simplex</i>	Triangle	$S_2 = \langle v_0, v_1, v_2 \rangle$	
3	<i>3-simplex</i>	Tetrahedron	$S_3 = \langle v_0, v_1, v_2, v_3 \rangle$	

Simplicial complexes have several properties (Alexandroff, 1961; Hatcher, 2002). They have been considered as a basic data type in developing spatial database systems (Penninga, 2008; Penninga and Oosterom, 2008; Schneider, 1997).

Simplicial complexes may consist of simplexes of different dimensions (Figure 3.5a). A *homogeneous* simplicial k -complex is a simplicial complex where every simplex of dimension less than k is the face of some simplex of dimension exactly k (Alexandroff, 1961; Hatcher, 2002). For example, a triangulation of a set of 2D points is a homogeneous simplicial 2-complex.

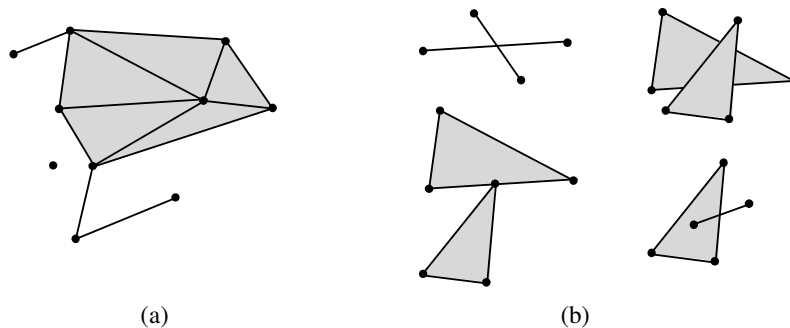


Figure 3.5. (a) A simplicial complex that consists of 0-, 1- and 2-simplexes. (b) Some configurations of simplexes that are not simplicial complex, because they violate axioms.

3.3.1.1 Orientation of an n -simplex

Vertexes of an n -simplex are ordered, which induces an orientation (either positive or negative) on the n -simplex. By convention, the orientation of a 0-simplex (point) is positive. The orientation of a 1-simplex (line segment) is positive from vertex v_0 to vertex v_1 and negative from vertex v_1 to vertex v_0 . For a 2-simplex (triangle), the orientation is defined based on the order in which the vertexes are listed: clockwise order is positive and counter-clockwise order is negative. The orientation of a 3-simplex (tetrahedron) is the sign of the volume constructed by its ordered vertexes (Alexandroff, 1961): based on the right-hand rule, a positive volume means that if the first three points are ordered so that they follow the direction of the curled fingers, then the thumb is pointing towards the 4th point.

The orientation of an n -simplex can be specified using the sign of the determinant of a matrix constructed as follows: for an n -simplex with vertexes $\langle v_0, \dots, v_n \rangle$, an element 1 is added to the end of each vertex and then they are arranged as the rows of a square matrix. For an n -simplex with vertexes $\langle (e_{01}, \dots, e_{0n}), \dots, (e_{n1}, \dots, e_{nn}) \rangle$, the result is:

$$\begin{vmatrix} e_{01} & \dots & e_{0n} & 1 \\ e_{11} & \dots & e_{1n} & 1 \\ \dots & \dots & \dots & \dots \\ e_{n1} & \dots & e_{nn} & 1 \end{vmatrix} = \begin{vmatrix} e_{11} - e_{01} & \dots & e_{1n} - e_{0n} \\ \dots & \dots & \dots \\ e_{n1} - e_{01} & \dots & e_{nn} - e_{0n} \end{vmatrix} \quad (3-12)$$

Non-negative values for this determinant indicate a positive orientation, while negative values mean a negative orientation. Similar to the determinant of a matrix, odd numbers of permutations of the vertexes of an n -simplex change the orientation, while even numbers of permutations maintain it unchanged. For instance, for the n -simplexes of Table 3.1:

$$\begin{aligned} S_0 &= \langle v_0 \rangle \\ S_1 &= \langle v_0, v_1 \rangle = - \langle v_1, v_0 \rangle \\ S_2 &= \langle v_0, v_1, v_2 \rangle = - \langle v_0, v_2, v_1 \rangle = \langle v_2, v_0, v_1 \rangle = \dots \\ S_3 &= \langle v_0, v_1, v_2, v_3 \rangle = - \langle v_0, v_1, v_3, v_2 \rangle = \langle v_0, v_3, v_1, v_2 \rangle = \dots \end{aligned} \quad (3-13)$$

3.3.1.2 Canonical representation of n -simplexes

Representation of an n -simplex by its vertexes is a situation where there are multiple representations for the same value. For a unified representation, we select a single preferred representation for each value, among the many equivalent ones, which is called *canonical representation*. We use a pair (*vertexes, orientation*) as the canonical representation in which the first element is the sorted list of its vertexes and the second element is the orientation of the n -simplex. For the vertexes of type (e_1, \dots, e_n) , they are sorted by e_1 -coordinate; in the case of equality of e_1 s, they are sorted by e_2 -coordinate, and so forth. For 2D points, this is sorting the points from left to right and from bottom up.

3.3.1.3 Faces of an n -simplex

For an n -simplex $S_n = \langle v_0, \dots, v_n \rangle$, any non-empty subset of vertexes $\{v_0, \dots, v_n\}$ is called a *face* of S_n . For example, all of the faces of the 3-simplex $S_3 = \langle v_0, v_1, v_2, v_3 \rangle$ are:

$$\begin{aligned}
 &\langle v_0 \rangle, \langle v_1 \rangle, \langle v_2 \rangle, \langle v_3 \rangle, \\
 &\langle v_0, v_1 \rangle, \langle v_0, v_2 \rangle, \langle v_0, v_3 \rangle, \langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_2, v_3 \rangle, \\
 &\langle v_0, v_1, v_2 \rangle, \langle v_0, v_1, v_3 \rangle, \langle v_0, v_2, v_3 \rangle, \langle v_1, v_2, v_3 \rangle, \\
 &\langle v_0, v_1, v_2, v_3 \rangle
 \end{aligned} \tag{3-14}$$

A face constructed from an improper subset is called an *improper face*. Thus, all of the faces illustrated in Equation (3-14), except the last one, are improper faces of S_3 . An n -simplex $S_n = \langle v_0, \dots, v_n \rangle$ has $\binom{n+1}{m+1}$ m -dimensional faces ($0 \leq m < n$) and $2^{(n+1)} - 1$ of improper faces altogether.

3.3.1.4 Boundary of an n -simplex

The *boundary* of the n -simplex $S_n = \langle v_0, \dots, v_n \rangle$, which is written as ∂S_n , is defined as follows:

$$\partial S_n = \sum_{i=0}^n (-1)^i \langle v_0, \dots, \bar{v}_i, \dots, v_n \rangle \tag{3-15}$$

where \bar{v}_i means omitting the vertex v_i from the vertex list. The boundary of an n -simplex is $n+1$ of $(n-1)$ -simplexes:

- The boundary of a 0-simplex (point) is an empty set;
- The boundary of a 1-simplex (line segment) is two 0-simplexes (points);
- The boundary of a 2-simplex (triangle) is three 1-simplexes (line segments);
- The boundary of a 3-simplex (tetrahedron) is four 2-simplexes (triangles).

For instance, for the n -simplexes of Table 3.1:

$$\begin{aligned}
 \partial S_0 &= \emptyset \\
 \partial S_1 &= \langle v_1 \rangle - \langle v_0 \rangle \\
 \partial S_2 &= \langle v_1, v_2 \rangle - \langle v_0, v_2 \rangle + \langle v_0, v_1 \rangle \\
 \partial S_3 &= \langle v_1, v_2, v_3 \rangle - \langle v_0, v_2, v_3 \rangle + \langle v_0, v_1, v_3 \rangle - \langle v_0, v_1, v_2 \rangle
 \end{aligned}
 \tag{3-16}$$

3.4 Functional programming languages

Functional programming is a paradigm in which functions are the central model components and are used as data; each parameter is considered as a function that is evaluated through a simplified form. Here, the focus is on function application, unlike the imperative programming languages that change the states (Hughes, 1989). Programming languages are classified by orders based on the variables used. A zero order language has no variables, only constants. A first order language has variables, which stand for objects, but not for predicates or functions. A second order language has variables that can stand for objects, predicates, or functions (sometimes called higher order). Functional programming languages fall in the category of second order, so they easily model processes; A process applies a set of defined functions to change the state of objects, which can be directly simulated by functional languages (Frank, 1997; Gunter, 1993).

3.4.1 Why we use functional programming languages in this thesis?

In this thesis, we focus on the properties of operations, instead of objects they are applied to. Functional languages are a direct solution to this purpose.

To construct the integrated framework of spatial analyses proposed in this research, we formally describe spatial analyses in a hierarchal way as combinations of simpler ones, until a set of primary un-decomposable operations are reached. Similarly, in a functional paradigm, a main function is defined through subsidiary functions, which are again defined through other subsidiary functions, and so on, until at the bottom level the functions, i.e., language primitives (called canonical expressions) are reached, which are not further simplified (Bird and Wadler, 1988). Therefore, the proposed integrated framework can be explicitly simulated in functional programming languages. In other words, functional programming languages are convenient tools to express algebraic specifications because both of them use a similar syntax and have similar mathematical foundations (Frank, 2000; Frank, 1999; Raubal, 2001).

Finally, in the proposed approach of the thesis, a spatial analysis is extended to a higher dimension using mappings (liftings) that maps different spaces to each other (e.g., 2D static to 2D moving). These liftings defined between functions are second order functions and can be modeled in functional programming languages.

It is essentially a formalization in a (dialect) of lambda calculus. Haskell — the functional programming language that we use — introduces “syntactic sugar” to abbreviate rewriting complex constructions and allows checking the results for syntactic completeness of definitions and semantic checking of the results.

3.4.2 Functional vs. structured programming languages

In *structured* programming languages – like C++, Pascal, Fortran and Java – a program consists of a set of blocks. A set of procedures are applied on input(s) of the block to produce an output.

Unlike *unstructured* languages (like Basic), blocks do not have multiple entries or exits. Therefore, commands like `goto`, which freely refer to any line of code, are not allowed. It enforces *modular* programming, which makes the programs simple, reusable and tractable (Bird and de More, 1997; Hughes, 1989).

In structured programming languages, a certain variable stands for a value that can be changed during the running time. i.e., the value assigned to a variable may change somewhere else. For example, assignments such as $a=a+1$ are allowed. It causes *side effects* in the programs. It means that re-assigning a variable in a block may affect the results of running another block. Thus, the order of running is important. For example, considering the following blocks A and B , the result of applying A then B is 13, while B then A yields 16:

$$\begin{array}{l}
 x = 5 \\
 \\
 \begin{array}{ll}
 \mathbf{Block A} & \mathbf{Block B} \\
 \{ & \{ \\
 \quad x = 2 * x & x = x + 3 \\
 \} & \}
 \end{array} \\
 \\
 \mathbf{B(A(x))} = 13 \\
 \mathbf{A(B(x))} = 16
 \end{array}
 \tag{3-17}$$

Functional programming is another programming paradigm that is constructed based on *function calls*. Here, a program is a function that calls other functions. For example, in the following expression:

$$\text{output} = \text{function 3} (\text{function2} (\text{function 1}))
 \tag{3-18}$$

function3 calls *function2*, which calls *function1*. Thus, the *function1* is evaluated first and passed to *function2*. Finally, *function3* applies on the result and produces the *output*.

Purely functional programming languages are based on λ -calculus — a mathematical theory of functions (Hankin, 2004; Michaelson, 1989). Like mathematics, functions produce only one result value and it is not changed, so there is no side effect. An expression always produces the same result because values can only be assigned once to a parameter (it is called *referential transparency*). Moreover, the final result is independent of the order of running the expressions. Functional programs are succinct because more than 90% of expressions in structured languages are assignments (Hughes, 1989), which do not exist in functional languages. Forbidding re-assignment enables lazy evaluation, which will be described in the following section.

On the other hand, *loop* expressions that are frequently used in structured languages are not allowed in functional languages because it is a re-assignment. Instead, the concept of

recursion is used, where definition of a function refers to itself. For example, the factorial function over natural numbers is demonstrated by the following definition (Thompson, 1999):

```
factorial (n) = if (n==0) then 1 else (n * factorial (n-1))
```

3.4.3 Evaluation of expressions in functional programming languages

In functional programming languages, expressions are evaluated through a complex and accurate mechanism that provides the maximum efficiency (Jeuring and Meijer, 1995; Peyton Jones, 1987). Following, we describe the main concepts, principals and rules used for expression evaluation in functional programming languages to an extent necessary for the implementations.

3.4.3.1 Free and bound variables

If t is a lambda term, and x is a variable, then " $\lambda x.t$ " is called a lambda abstraction. For example $\lambda x. + x 3$ is a function that adds 3 to its input x .

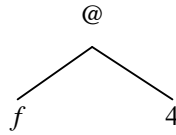
The abstraction operator, λ , is said to bind its variable wherever it occurs in the body of the expression. Variables that fall within the scope of a lambda are said to be *bound*. All other variables are called *free*. For example, in the expression $\lambda y. x \times y$, the variable y is a bound variable and x is free. Also note that a variable binds to its "nearest" lambda. For example, in the expression $\lambda x. y (\lambda x. z \times x)$, one single occurrence of x is bound by the second lambda.

3.4.3.2 Reduction

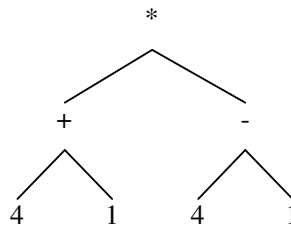
Suppose that the function f is defined as

$$f\ x = (x + 1) * (x - 1)$$

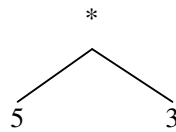
and we are required to evaluate $f(4)$. We can think of the program like this:



where the @ stands for function application. Applying f to 4 gives:



Applying the addition and the subtraction (in either order) gives:



Finally, we can execute the multiplication to get the result:

15

This simple example shows that *executing* a functional program consists of *evaluating* an expression; and the evaluation proceeds by means of a sequence of simple steps, called *reduction*. Each reduction performs a local transformation. Evaluation is complete when there are no further reducible expressions (called *redex*).

3.4.3.3 β -Reduction

Suppose the following lambda expression is given:

$(\lambda x. + x 1) 4$

In lambda calculus syntax, it denotes the application of a certain function, indicated by the lambda abstraction, to the argument 4. The rule for such function application is very simple: The result of applying a lambda expression to an argument is an instance of the body of the lambda abstraction in which occurrences of the formal parameter in the body are replaced with (copies of) the argument. Thus, the result of applying the lambda expression $(\lambda x. + x 1)$ to the argument 4 is:

$$+ 4 1$$

The $(+ 4 1)$ is an instance of the body $(+ x 1)$ in which occurrences of the formal parameter x are replaced with the argument 4. We write the reduction using the arrow \rightarrow :

$$(\lambda x. + x 1) 4 \rightarrow + 4 1$$

This operation is called β -reduction (Peyton Jones, 1987). Here are a few more examples of β -reduction:

```
If (not true) f g → if false f g → g
(λx. If (x > 0) f g) 2 → If (2 > 0) f g → f
head (cons 2 nil) → 2
(λx. head (cons (x+2) nil)) 2 → head (cons 4 nil) → 4
```

If an expression contains more than one redex, the order of reduction is from outer most to inner most expression (see section A1.1.3.7).

3.4.3.4 Normal Form

If an expression contains no redexes, then the evaluation is complete and the expression is said to be in *normal form*. Thus, the evaluation of an expression consists of successively reducing redexes until the expression is in normal form.

3.4.3.5 Weak head normal form

A lambda expression is in *weak head normal form* (WHNF) if and only if it is of the form

$$F \ E_1 \ E_2 \ \dots \ E_n$$

where $n \geq 0$ and $(F \ E_1 \ E_2 \ \dots \ E_m)$ is not a redex for any $m \leq n$. An expression has no *top-level redex* if and only if it is in weak head normal form (Peyton Jones, 1987). For example, the following expressions are in weak head normal form:

```
3
+ (- 4 3)      top-level + does not have enough arguments
```

The latest example is in weak head normal form, but not in normal form, since it contains inner redexes.

3.4.3.6 Head normal form

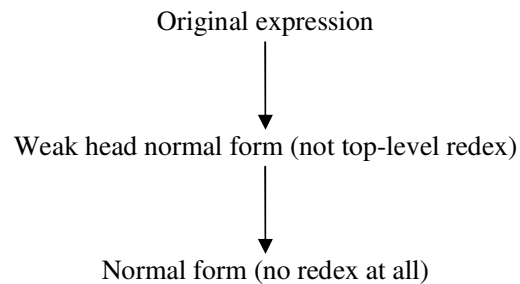
A lambda expression is in *head normal form* (HNF) if and only if it is of the form

$$\lambda x_1. \lambda x_2. \dots \lambda x_n. (\vee \ M_1 \ M_2 \ \dots \ M_m)$$

where $n, m \geq 0$ and $(\vee \ M_1 \ M_2 \ \dots \ M_p)$ is not a redex for any $p \leq m$. Anything in HNF is also in WHNF, but not vice versa (Peyton Jones, 1987). For example:

$$\lambda x. ((\lambda y. y) \ 3)$$

We can think of it like this:



3.4.3.7 *Lazy evaluation*

In ordinary imperative languages, arguments to a function are evaluated before the function is called (*call by value*). However, it is possible that the argument passed is never used in the body of the function, so that the work done in evaluating is wasted. This suggests that a better scheme might be to postpone the evaluation of the argument until its value is actually required (*call by need*). Call by need is in fact rarely implemented in imperative languages, because the evaluation of an argument may cause some side-effects to take place, and may produce a result which depends on the side effects (e.g., assignments) of other parts of the program. Hence, the exact time at which the argument is evaluated is crucial to the correct application of the program. However, it can be quite tricky to work out exactly when the argument be needed (and hence evaluated).

The order of execution of expressions is not important in functional languages. Therefore, the evaluation of an expression can be postponed till its value is actually needed to compute the overall result. On the other hand, if the value of an already evaluated expression is required again, the evaluated value can be safely used, because it has not been changed.

In the context of functional languages, call by need is often called *lazy evaluation*, since it postpones work until it becomes unavoidable. Any implementation of lazy evaluation has two ingredients (Peyton Jones, 1987):

- Arguments to functions should be evaluated only when their value is needed, not when the function is applied.
- Arguments should only be evaluated once; further uses of the argument within the function should use the value computed the first time. Since the language is functional and has no side-effect, this scheme gives the same results as re-evaluating the argument.

In a nutshell, arguments should be evaluated *at most* once and, if possible, not at all.

3.4.3.8 *Outer-to-inner evaluation*

A general form of an expression in functional languages is:

$$\text{output} = f_n (f_{n-1} (\dots f_2 (f_1))) \quad (3-19)$$

The value of f_k is dependent to the values of f_1 to f_{k-1} . However, f_k may not depend on some of the f_i s. $1 < i < k-1$. For example, if $f_1 = ((x+2)2*3+4$ and $f_2 = 3$, then the value of $f_2(f_1(5))$, which is equal to 3, is achievable without evaluating $f_1(5)$. To satisfy the lazy evaluation rule, the expressions must be evaluated from the most outer to the most inner ones. This can be expressed as “first, the most outer expression is evaluated and the next level expression is evaluated only if it is needed through achieving the normal form”. It is called *outer-to-inner evaluation* (Peyton Jones, 1987).

3.4.3.9 Currying mechanism

This mechanism transforms a function with multiple variables into multiple functions with single argument. In other words, consider the following function of n variables:

$$\text{output} = f(x_1, x_2, \dots, x_n) \quad (3-20)$$

Suppose that k of these variables are known. Substitution of the known variables in f results in a function of $n-k$ variables, which is the normal form of f_n . Further reductions need introducing the unknown variables. This mechanism is called *currying* in the functional programming.

For example, $f(x, y) = x+y$ is a binary function. However, if $y=5$, then $f(x) = x + 5$ is a unary function:

$$\begin{aligned} \text{plus}(x, y) &= x+y \\ y=5 &\rightarrow \text{plus5}(x) = x+5 \end{aligned} \quad (3-21)$$

3.5 Summary

In this chapter we explained the formal methods used in this thesis to extend spatial analyses to different multi-dimensional spaces. We started by introducing the abstraction concepts needed to construct an integrated framework of spatial analyses based on their dimension independent properties. Then, we explained the algebraic structures as the required abstraction to formally define spatial analyses as combinations of the elements of the integrated framework. The simplicial complexes were introduced as an n -dimensional data type that is needed for dimension independent implementations. We will develop the geometric and topological operations on n -simplexes in chapter 5. At the end of this chapter,

we introduced the functional programming languages and explain why they are used in this thesis. The main concepts of functional programming language, especially their evaluation strategies, were described to an extent necessary for arguing the implementations provided in the thesis.

4 PROPOSED APPROACH OF THE RESEARCH

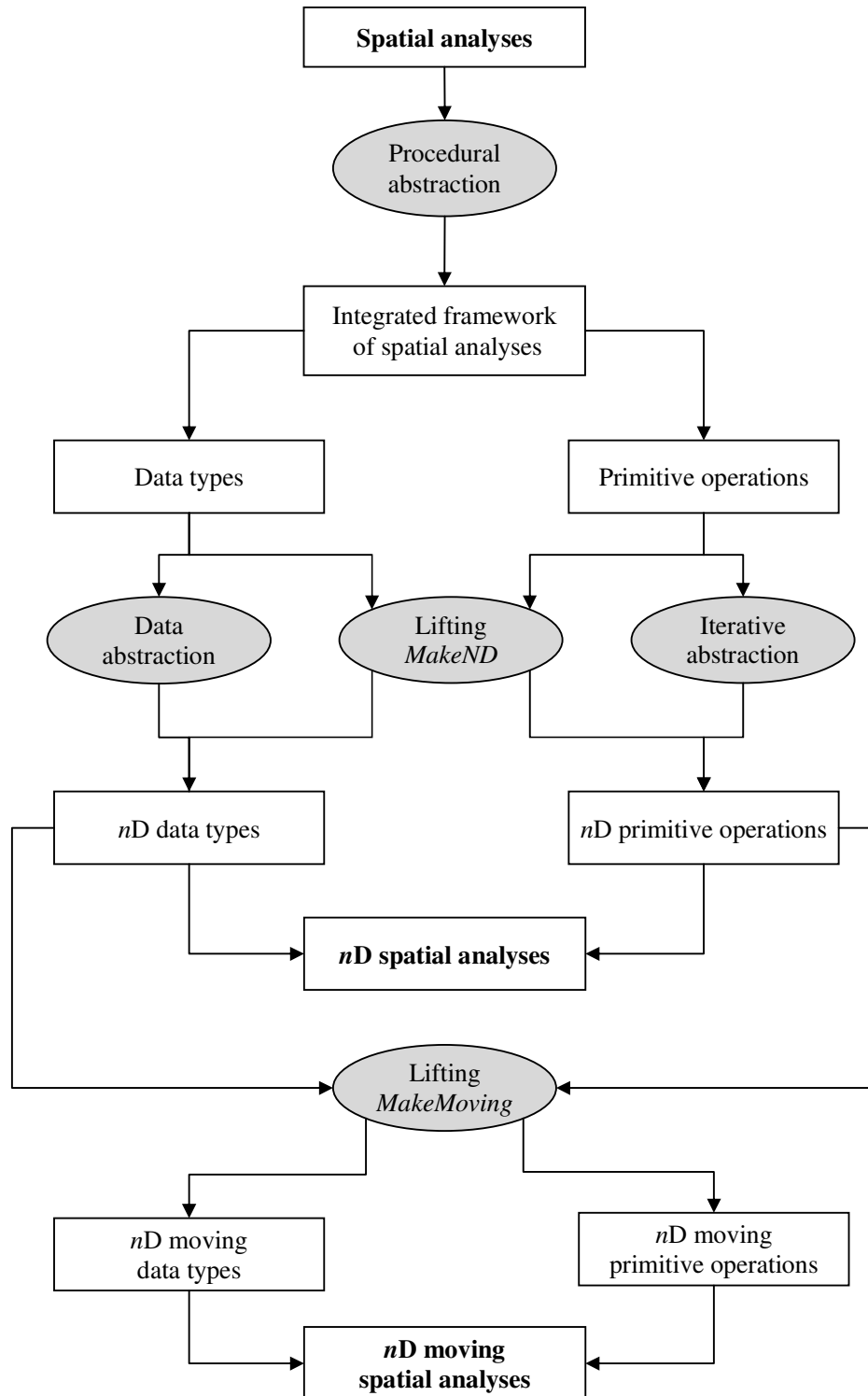
This chapter describes the proposed approach of the research to extend spatial analyses to different dimensions. We explain how to formally define dimension independent spatial analyses, which leads to an abstract integrated framework of spatial analyses. This idea is applied on the Delaunay triangulation, as the case study of the research. This framework will be the basis to develop the n -dimensional static and moving analyses in the next chapters.

4.1 A review on the proposed approach

This research proposes a formal approach to implement dimension independent spatial analyses. It studies spatial analyses based on their dimension independent characteristics and formally describes them using algebraic structures.

Figure 4.1 illustrates the research approach to extend spatial analyses to n -dimensional static and moving objects. First, the procedural abstraction is used to formally describe spatial analyses based on a set of primary operations, which are not further decomposed. These definitions are independent of dimension and results in a hierarchy that relates the spatial analyses and operations. In the next step, data abstraction is used to build n -dimensional data types in order to model objects of different dimensions. The operations to interact with these n -dimensional data types are also developed using the iterative abstraction and mapping to n -dimensional space. Then, all of the spatial analyses of the hierarchy, which are defined as combinations of primary operations, are immediately available in n -dimensional space, without any further efforts. Finally, the relevant mappings are applied to map these n -dimensional data types and primary operations to support n -dimensional moving objects, which provide spatial analyses for n -dimensional moving objects.

The rest of this chapter describes the formalization process to construct the abstract integrated framework and, as an example, uses it for the Delaunay triangulation as the case study of the research. Extension of the framework to n -dimensional and moving objects will be presented in the next two chapters.

Figure 4.1. The research approach to extend spatial analyses to n -dimensional static and moving objects

4.2 Formal definition of spatial analyses

To formally define a spatial analysis, it is expressed independent of the characteristics of the objects to which it is applied. We describe the overall process that is applied on the inputs to produce the outputs, which is procedural abstraction of spatial analyses. For instance, the formal definition of the Delaunay triangulation says that a set of input points is triangulated (by connecting points with lines, faces, etc.) so that the triangles satisfy the circum-circle property. This formal definition may have several implementations based on the algorithm used (e.g., *flipping*, *Bowyer-Watson*, etc.).

The above formal definition is then detailed by specifying an implementation algorithm. This algorithm must be multi-dimensional so that it can be later implemented independent of dimension. In section 2.5, we classified the multi-dimensional algorithms to *n-dimensional* (i.e., can be adopted to support different dimensions) and *dimension independent* (i.e., independent of dimension in their definition). Although the *n-dimensional* algorithms are developed for objects of any dimension, they cannot be used in our approach: in the abstract definition, an *n-dimensional* algorithm similarly works for different dimensions (i.e., the overall procedure is the same). However, the implementation details are different from one dimension to another. We are interested in dimension independent algorithms, which are independent of dimension in both definition and implementation.

For example, the abstract description of the *flipping* algorithm for the Delaunay triangulation (section 2.1.3.1) is as follows:

- 1- Construct an initial *n*-simplex that contains all of the vertexes
- 2- Incrementally insert the vertexes and update the structure as follows:
 - 2-1- Find the containing *n*-simplex *S* (*Walk*)
 - 2-2- Replace the *S* with new *n*-simplexes passing through the new vertex (*Insert*)
 - 2-3- Check the new *n*-simplexes with their neighbors against the circum-circle property (and apply *flipping* in case of failure) until this property is satisfied by all *n*-simplexes (*Update*)

This abstract description is valid for constructing the Delaunay triangulation of 2D and 3D points, but the details of *flipping* is different in 2D and 3D: there are two possible triangulations for four points in 2D (Figure 2.6). However, to tetrahedralize five 3D points, there are two possible solutions: one has two tetrahedra and the other has three (Figure

2.15). Moreover, according to the geometry of a tetrahedron in the 3D Delaunay triangulation with its adjacent, it is not always possible to perform a flip (Figure 2.16) and it must be left to be performed by a later element. These differences prevent a dimension independent implementation of flipping algorithm.

The abstract description of the *Bowyer-Watson* algorithm for the Delaunay triangulation (section 2.1.3.2) is as follows:

- 1- Construct an initial n -simplex that contains all of the vertexes
- 2- Incrementally insert the vertexes and update the structure as follows:
 - 2-1- Delete all the n -simplexes whose circum-circle contain the new vertex
 - 2-2- Join the new vertex to the nodes of the deleted part

This description is applicable to both 2D and 3D points with the same implementation details and so can be used in our approach.

The abstract description of the Voronoi diagram is as follows:

- 1- Construct the Delaunay triangulation of the point set
- 2- Compute the center of the circum-circles of the n -simplexes
- 3- Join the centers of the neighboring n -simplexes

4.3 Constructing an abstract hierarchical framework of spatial analyses

We describe a spatial analysis as a combination of some simpler analyses and operations. For instance, the convex hull calculation consists of determining the faces that have all of the other points of the set at one side (Berg et al., 2008). This later operation can be described as a repetitive determination of the position of a point with respect to a face, which can be described as a determinant calculation, and so forth. Eventually, this procedure provides a hierarchy of spatial analyses and operations in which, the elements of each level are described as combinations of elements of the lower levels. More precisely, if $f_{i,j}$ denotes the i th element of the j th level, then:

$$f_{i,j} = F(f_{p,q}), 0 \leq q < j, p \geq 0 \quad (4-1)$$

The elements of the lowest level of this hierarchy are primitive operations. All of the analyses and operations of the hierarchy are described as combinations of these primitive operations. On the other hand, a set of data types are needed for definition of these primitive operations.

4.3.1 Example: Constructing the hierarchical framework for Delaunay triangulation

In this section, we construct an abstract hierarchical framework for the Delaunay triangulation and Voronoi diagram, and identify the data types and primitive operations. Regarding the formal description of the *Bowyer-Watson* algorithm presented in section 4.2, the following spatial analyses and operations are needed:

- Constructing the Voronoi diagram needs the *Delaunay triangulation* to construct the Delaunay triangulation and *circle-center* that computes the center of the circum-circle of an n -simplex.
- Constructing the Delaunay triangulation needs *point-simplex-test* that identifies the position of a point with respect to an n -simplex, *point-circle-test* that identifies the position of a point with respect the circum-circle of an n -simplex, and *vertex-simplex-join* that joins a vertex to an n -simplex.
- All of the *circle-center*, *point-simplex-test* and *point-circle-test* are described based on the determinant calculations.
- The Determinant calculation consists of operators “+”, “-“ and “*”.

The above descriptions result in the hierarchy illustrated in Figure 4.2. The only data types required for the analyses and operations of this hierarchy are *numbers*, and a data type to represent a point.

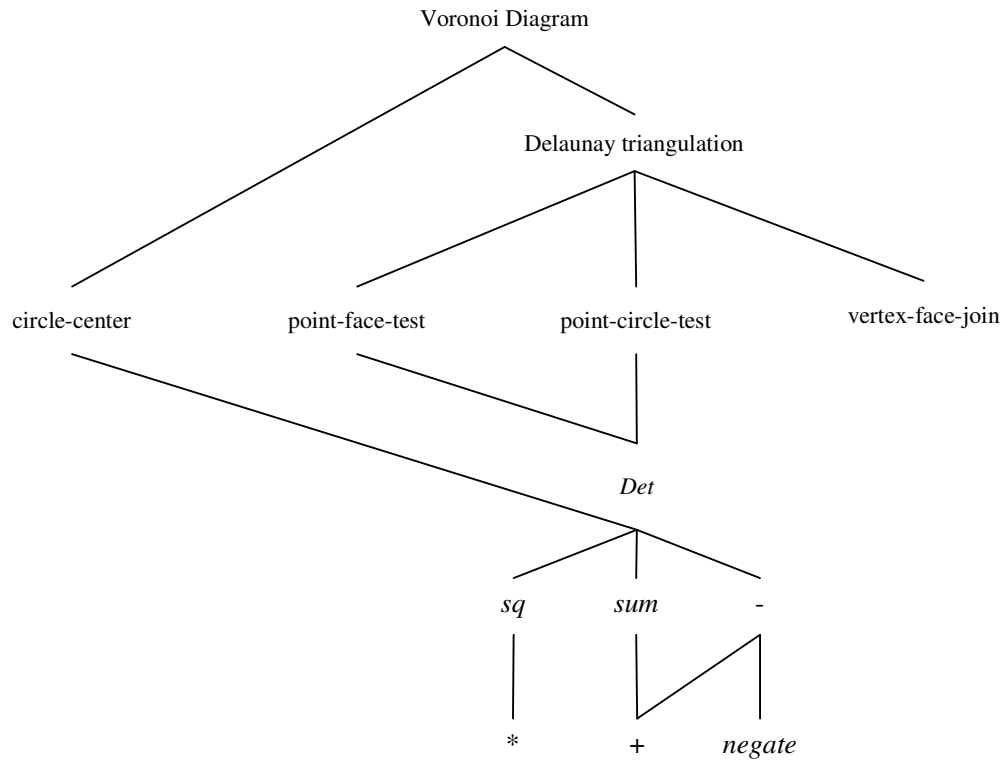


Figure 4.2. The hierarchy of spatial analyses and operations to define the case studies

4.4 Summary

This chapter presented the proposed approach of the research to extend spatial analyses to different dimensions. We described the process of formal definition of spatial analyses and construction of an abstract hierarchical framework of spatial analyses in which analyses and operations are defined as combinations of primitive operations.

5 EXTENSION TO N-DIMENSIONS

This chapter describes the extension of the formal integrated framework of spatial analyses – built in the previous chapter – to support n -dimensional objects. We use the n -simplexes as an n -dimensional data type and implement the operations on the n -simplexes based on vector algebra. As an example, we show how to use this approach to implement an n -dimensional Delaunay triangulation and its dual, Voronoi diagram.

We use the syntax of functional programming language Haskell, which is our implementation environment. In Appendix 1, the main concepts and syntax of Haskell are described. The complete Haskell code of the implementations is given in Appendix 2.

5.1 Vector Algebra

A *vector* \mathbf{V} in an n -dimensional space is an arrow that is determined by its length, denoted $|\mathbf{V}|$ and its direction, denoted by $\vec{}$. Figure 5.1 shows a vector \mathbf{P} in 2D space described by its Cartesian coordinates. Two arrows represent the same vector if they have the same length and are parallel. Vectors represent entities that are described by magnitude and direction. An object moving in space has, at any given time, a direction of motion, and a speed. This is represented by the velocity vector of the motion. The success and importance of vector algebra derives from the interplay between geometric interpretation and algebraic calculation.

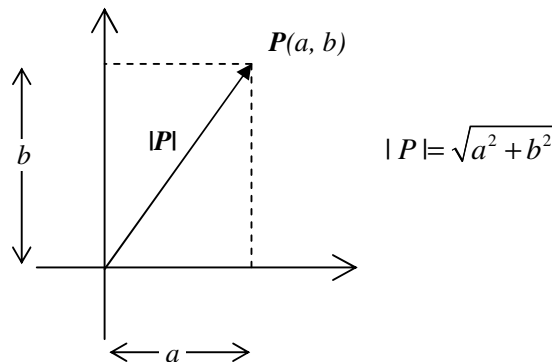


Figure 5.1. A 2D vector \mathbf{P} represented by its Cartesian coordinates

Vectors are added by adding their corresponding elements (Figure 5.2.a). Addition of vectors is commutative ($a+b=b+a$). Therefore, they form a group with the zero vector as the unit. Multiplication of a vector with a scalar k extends the vector k times, keeping the direction (Figure 5.2.b). This multiplication is distributive over addition, etc.

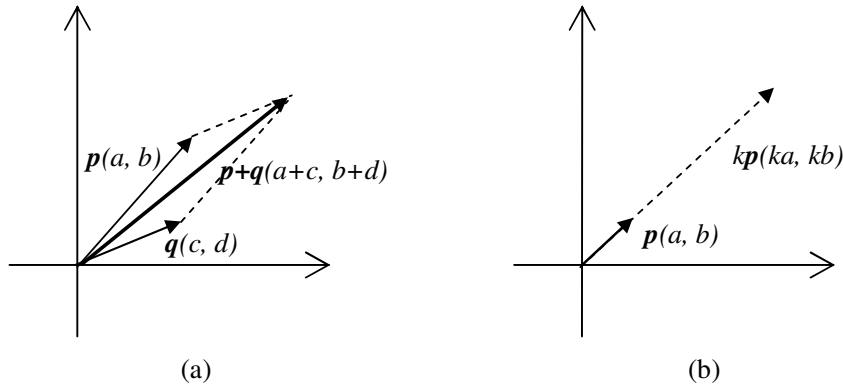


Figure 5.2. (a) Addition of vectors (b) Multiplication of a vector with a scalar

A vector space is a *module* over a field that consists of two kinds of things: *vectors*, which are a commutative group, and *scalars*, which form a ring with unit. These vectors and scalars are combined with an external operator scalar multiplication “ \cdot ” (MacLane and Birkhoff, 1999) with the following axioms:

$$\begin{aligned}
 &\text{Module } \langle M, +, 0 \rangle \text{ with group } \langle M, +, 0 \rangle \text{ and Ring with unit } \langle Q, +, *, 0, 1 \rangle \\
 &\text{for all } p, q \in Q \text{ and all } a, b \in M \\
 &p \cdot (a + b) = p \cdot a + p \cdot b \\
 &(p + q) \cdot a = p \cdot a + q \cdot a \\
 &(p * q) \cdot a = p \cdot (q \cdot a) \\
 &1 \cdot a = a
 \end{aligned} \tag{5-1}$$

5.1.1 Operations on vectors

An n -dimensional point is described as a vector in the n -dimensional Cartesian coordinate system. Then, geometric properties can be represented as operation on vectors. Apart from addition and scalar multiplication presented above, the inner (dot), cross and triple products of vectors are very often used.

5.1.1.1 Inner product

For the n -dimensional vectors $U(u_1, u_2, \dots, u_n)$ and $V(v_1, v_2, \dots, v_n)$, the inner product is a scalar defined as:

$$U \cdot V = (u_1, u_2, \dots, u_n) \cdot (v_1, v_2, \dots, v_n) = u_1 \cdot v_1 + u_2 \cdot v_2 + \dots + u_n \cdot v_n \quad (5-2)$$

The inner product is defined for all dimensions. For 2D and 3D vectors, it has some geometric properties:

- The inner product of a 2D or 3D vector with itself is the square of its length (called *norm*):

$$|U|^2 = U \cdot U = (u_1, u_2, \dots, u_n) \cdot (u_1, u_2, \dots, u_n) = u_1^2 + u_2^2 + \dots + u_n^2 \quad (5-3)$$

Then, the unit vector in the direction of a given vector is:

$$e_U = \frac{U}{|U|} \quad (5-4)$$

- The angle θ between two 2D or 3D vectors can be obtained using their inner product and their norms:

$$\cos \theta = \frac{U \cdot V}{|U| |V|} \quad (5-5)$$

Then, two 2D or 3D vectors are orthogonal if their inner product is zero:

$$U \cdot V = 0 \leftrightarrow U \perp V \quad (5-6)$$

5.1.1.2 Cross product

For the 3D vectors $U(u_1, u_2, u_3)$ and $V(v_1, v_2, v_3)$, the cross product is a vector orthogonal on both U and V and is defined as:

$$U \times V = (u_1, u_2, u_3) \times (v_1, v_2, v_3) = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1) \quad (5-7)$$

The length of $U \times V$ is twice the area of the triangle built from the two vectors:

$$A_{OUV} = \frac{|U \times V|}{2} \quad (5-8)$$

Two vectors are collinear if their cross product is zero.

5.1.1.3 Triple product

For the 3D vectors $U(u_1, u_2, u_3)$, $V(v_1, v_2, v_3)$ and $W(w_1, w_2, w_3)$, the triple product is a combination of a cross product and an inner product results in a scalar:

$$\langle U, V, W \rangle = U \cdot (V \times W)$$

$$\langle U, V, W \rangle = \begin{vmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix} \quad (5-9)$$

The triple product gives six times the volume of the parallelepiped built from the three vectors.

Vector algebra is used explicitly and implicitly in many of our calculations. The circum-circle test (Equation 2-4), clockwise and counter-clockwise tests (Equations 2-3 and 2-5) and orientation of an n -simplex (Equation 3-12) are examples of using vector algebra in geometric operations.

5.1.2 Lifting *MakeND* to extend data types and primary operations to n -dimensional objects

To define a lifting *MakeND* to extend data types and primary operations of the hierarchy to support n -dimensional objects, the following observations are considered:

- A point in the n -dimensional Euclidean space is a vector represented by n numerical elements. Then, the operations on points become vector operations.
- An n -simplex is represented as a set of $n+1$ points of dimension n and operations on simplexes are defined as vector operations.

Considering the above observations, a lifting to develop the vectors and their operations is defined as follow:

- The data type a must be mapped to a vector of data type a .
- An operation of m variables ($m \geq 0$) with the input $\bar{X} = (x_1, \dots, x_m)$ must be mapped to an operation that is applied to every elements of a set of input vectors, each of which consists of m elements.

Therefore, the lifting *MakeND* denoted as N is mathematically defined as:

$$\begin{aligned} a &\xrightarrow{N} [a_1, \dots, a_n] \\ f(\bar{X}) &\xrightarrow{N} [f(\bar{X}_1), \dots, f(\bar{X}_n)] \end{aligned} \quad (5-10)$$

As mentioned in chapter 3, this mapping must be commutative in order to be a lifting:

$$N(f_A(s_1, \dots, s_k)) = f_B(N(s_1), \dots, N(s_k)) \quad (5-11)$$

It is satisfied, because:

$$\begin{aligned}
& \forall f, g : (N(f).N(g)) [\bar{X}_1, \dots, \bar{X}_n] = \\
& (N(f)) (N(g) [\bar{X}_1, \dots, \bar{X}_n]) = (N(f)) [g(\bar{X}_1), \dots, g(\bar{X}_n)] = \\
& [f(g(\bar{X}_1)), \dots, f(g(\bar{X}_n))] = [f.g(\bar{X}_1), \dots, f.g(\bar{X}_n)] = \\
& (N(f.g)) [\bar{X}_1, \dots, \bar{X}_n]
\end{aligned} \tag{5-12}$$

5.2 Definition of data types and classes

We start by defining a class `Ring` to support numerical values and their operations. The class `Ring` has three basic operations `+` and `*` and `neg` (negation) as well as other operations `-`, `sq` and `sum`, which are described based on the basic operations:

```

class Ring q where
  (+), (-), (*) :: q -> q -> q
  neg, sq      :: q -> q
  sum         :: [q] -> q

  a - b = a + (neg b)
  sq a  = a * a
  sum ls = foldl (+) zero ls

```

Instances of this class are defined for different data types (here, `Int` and `Float`):

```

instance Ring Int where
  neg  = Prelude.negate
  a + b = a Prelude.+ b
  a * b = a Prelude.* b

instance Ring Float where
  neg  = Prelude.negate
  a + b = a Prelude.+ b
  a * b = a Prelude.* b

```

2D and 3D points are represented as a pair (x, y) or triple (x, y, z) , respectively. To have an n -dimensional representation, we represent a point as a vector, which is a list of its elements in the Cartesian coordinate system.

```
type Vec a = [a]
type StaticPt a = Vec a
```

To develop the n -simplex data type, we start by defining a vertex. A vertex is the same as a point, i.e., the `Vec` data type can be used equally for a vertex. However, their equality is explicitly indicated here, for the sake of clarity:

```
Vertex = Vec
```

Then an n -simplex is a list of vertexes:

```
Simplex :: [Vertex]
```

The canonical representation of an n -simplex is a pair of the sorted list of its vertexes and its orientation. We use a Boolean value for the orientation: *true* for positive and *false* for negative orientations.

```
CnSimplex :: (Simplex, Bool)
```

Dealing with n -simplexes and their operations is a case where the number of elements is not known:

- An n -dimensional point in the Euclidean space is represented by n numbers.
- An n -simplex is represented by $n+1$ points of dimension n .
- The operations on an n -simplex can take any number of points as input each of which can have any number of elements, *per se*. The same applies to the output. The situation is even worse if a number of operations are composed.

Here we use the *list* as an abstract data type that can model efficiently both of n -dimensional points and n -simplexes as well as their operations. A list is a collection of any number of elements of the same type. The advantages of using the list data type are:

- Elements of a list can be from any type, so a list can model a point, an n -simplex (as a set of points), or even any other data structure that may be needed (e.g., a pair whose first and second elements are a point and a list of simplexes, respectively).
- A list can have any number of elements, so it can be used to model points and simplexes of any dimension.
- List operations are independent of the number and type of the elements of the list, so the operations on points and simplexes can be equally used in any dimension.

Lists are very important and frequently used in functional programming languages. A complete description of lists and their operations is presented in Appendix 1.

5.3 Operations of n -simplexes

The first operation we define is the dimension of an n -simplex. It is the number of its vertexes, and so it is equal to `length` of the list:

```
simpDim = length
```

and dimension of a canonical n -simplex is the number of its vertexes:

```
cnSimpDim s = (length.vertexes) s
```

The orientation of an n -simplex uses the determinant of the matrix introduced in Equation (3-13). We create the required matrix and calculate its determinant:

```
getOrn s = det mat > 0
  where
    mat = map (1:) s
```

where `det` is a function that calculates the determinant of an square matrix.

To convert the primary representation of an n -simplex (i.e., list of its vertexes) to its canonical representation, we make a pair whose first and second elements are the sorted list of vertexes and the orientation of the n -simplex, respectively:

```
simp2cnSimp s = (sort s, getOrn s)
```

To convert the canonical to the primary representation, we apply the sign (orientation) to the list of the vertexes: If the sign is positive, no change is needed; if it is negative, however, the orientation of the simplex must be changed, which is achieved by swapping the first and the second elements:

```
cnSimp2simp ([v], b) = [v]
cnSimp2simp (vs, b) = if (b == true) then vs else swap vs
  where
    swap []      = []
    swap [v]    = [v]
    swap (v1:v2:vs) = (v2:v1:vs)
```

The operations to get the vertexes and orientation of an n -simplex are trivial:

```
vertexes (vs, b) = vs
orn      (vs, b) = b
```

Changing the orientation of an n -simplex is simply changing its sign:

```
changeOrn (vs, b) = (vs, not b)
```

The checks whether two n -simplexes have the same vertexes or orientation are:

```
eqVs  s1 s2 = vertexes s1 == vertexes s2
eqOrn s1 s2 = orn s1 == orn s2
```

Thus, the equality of two n -simplexes (i.e. consisting of the same vertexes and having the same orientation) is defined as follows:

```
eqSimps s1 s2 = eqVs s1 s2 & eqOrn s1 s2
```

The faces of dimension n of an n -simplex are the n combinations of its vertexes. Then, the function `simp2cnSimp` must be applied to all of them in order to have a canonical n -simplex:

```
faceN s n = (map simp2cnSimp) . (combine n) . vertexes $ s
```

To extract all of the faces of an n -simplex, we compute all of the faces of dimension i , and concatenates them:

```
faces s = concatMap faceN s [1.. n]
  where
    n = cnSimpDim s
```

The boundary operation for an n -simplex is implemented as:

```
boundary vs b = zip (removeEach vs) (cycle [b, not b])
```

In this definition, `removeEach vs` makes a list of all possible $(n-1)$ -simplex and `cycle [b, not b]` provides their corresponding sign. The two lists are zipped to make the final boundary.

To add a vertex to an n -simplex, we get the vertexes of the input simplex, add the new vertex to the front of the resultant vertex list and finally convert it to the canonical representation. Figure 5.3 shows the functionality of this operation for 2- and 3-simplexes.

```
addVertex v s = simp2cnSimp . (v:) . vertexes $ s
```

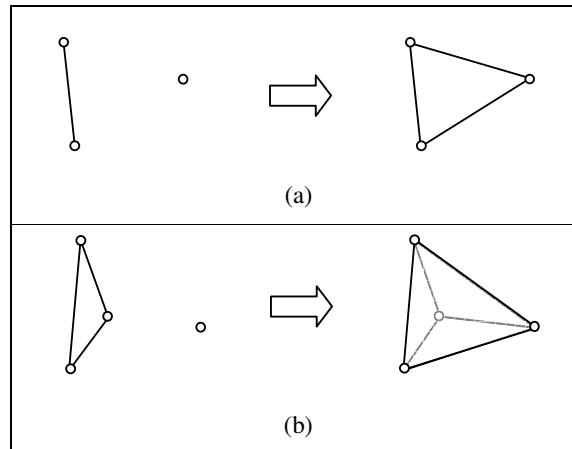


Figure 5.3. Functionality of the `addVertex`: a new vertex is added to a (a) 1-simplex (b) 2-simplex

The next is the `border` operation (we need this operation to extract the border of the hole created by removing the violating n -simplexes in the *Bowey-Watson* algorithm). As Figure 5.4 shows, this operation extracts the bordering $(n-1)$ -simplexes from a set of connected n -simplexes (a set of n -simplexes $S = \{s_1, s_2, \dots, s_m\}$ are connected if and only if for each $s_i \in S$, there is at least one $s_j \in S$ ($i \neq j$) such that $s_i \cap s_j$ is an $(n-1)$ -simplex). Note the difference between this operation and the `boundary` operation, which extracts the boundary of an individual n -simplex.

To implement this operation, we use the fact that bordering simplexes appear once and only once. Thus, to get the bordering simplexes, first we extract and concatenate the boundaries of all n -simplexes and then take the simplexes that appear once in this list:

```
border s = once . (concatMap boundary) $ s
```

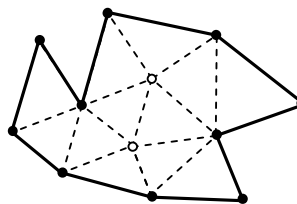


Figure 5.4. Functionality of the `border` for a set of connected 2-simplexes (dotted triangles), which results in their bordering 1-simplexes (bold edges)

The test whether three 2D points are in counter-clockwise (*ccw*) or clockwise (*cw*) order is often used in geometric algorithms (Knuth, 1992). Its extension to 3D checks

whether a point is on the right or left side of a plane goes through three points. The `ccw` and `cw` tests are implemented, generally, by adding the given n -dimensional point to the n -simplex and determining the orientation of the resultant $(n+1)$ -simplex:

```
ccw pt s = orn . (addVertex pt) $ s
cw  pt s = not (ccw s pt)
```

The test whether an n -dimensional point is inside the n -dimensional circum-sphere of an n -simplex is achieved by implementing the Equation (2-4):

```
inSphere p s = det (mat) >= 0  where
  mat = map (tr (s))
  tr x  = dx ++ [sum . map (sq (dx))]
  dx   = x - p
  sum (x) = fold ((+), 0, x)
```

5.4 Implementation of spatial analyses

To implement the spatial analysis, we use the n -simplexes to convert their formal description to implementable algorithms. For our case studies, it is as follows:

Algorithm *Bowyer-Watson-DT* (P)

Input. A set $P=\{p_0, \dots, p_m\}$ of n -dimensional points ($m \geq n$)

Output. A homogenous simplicial n -complex D that is the n -dimensional DT of P

1. $D \leftarrow$ A big n -simplex that contains all of the points $\{p_0, \dots, p_m\}$
2. **for** all points $p \in P$
3. $S \leftarrow$ Set of all n -simplexes $e \in D$ whose circum-sphere contains p
4. $B \leftarrow$ Set of $(n-1)$ -simplexes that make the border of S
5. $N \leftarrow$ Set of n -simplexes constructed by adding p to all $(n-1)$ -simplexes $b \in B$
6. $D \leftarrow \{D \setminus S\} \cup N$
7. **return** D

Algorithm *Voronoi* (P)

Input. A set $P = \{p_0, \dots, p_m\}$ of n -dimensional points ($m \geq n$)

Output. The Voronoi diagram of P

1. $D \leftarrow$ The Delaunay triangulation of the points $\{p_0, \dots, p_m\}$
2. $C \leftarrow$ The centers of the circum-spheres of the n -simplexes D
3. $VD = \{\}$
4. **for** all $s \in D$
5. $l \leftarrow$ the lines connecting the center of the n -simplex s to the center of all of its neighboring n -simplexes $ns \in D$
6. $VD \leftarrow VD \cup l$
7. **return** D

Implementations of these algorithms in Haskell are as follows (for complete implementation details, see Appendix 2):

```

delaunay :: [PtF] -> [CnSimplex]
delaunay pts = fold updateDT bigSimp pts
  where
    bigSimp = simple computations presented in appendix 2

updateDT dt pt = (dt \\ s) ++ n
  where
    s = filter inSphere pt dt
    n = map (addVertex pt) (border s)

Voronoi :: [PtF] -> [CnSimplex]
voronoi = connectNeighbors . map (center . delaunay)

```

where `updateDT` inserts a new vertex into a Delaunay triangulation and updates its structure, `center` computes the center of the circum-circles of a triangle, and `connectNeighbors` connects the center of the circum-circles of the neighboring triangles in a triangulation.

5.5 Summary

In this chapter we extended the formal integrated framework to support n -dimensional objects. We used the n -simplexes as an n -dimensional data type and implemented the operations on the n -simplexes based on the vector algebra. As an example, we showed how to use this approach to implement the n -dimensional Delaunay triangulation and Voronoi diagrams. The implementations in Haskell were presented.

6 EXTENSION TO MOVING OBJECTS

This chapter introduces a mapping called *MakeMoving* that extends the n -dimensional data types and operations developed in the previous chapter to moving objects.

6.1 Definition of the lifting *MakeMoving*

The moving data types and operations is structurally the same as their static corresponding elements, except that these data types as well as the input of the operations of moving objects are functions of time and the result is a function of time (Frank and Gruenbacher, 2001). The lifting to do this mapping is defined as follows:

- The data type a must be mapped to a data type a_t that is a function of time.
- An operation of m variables ($m \geq 0$) with the input $\bar{X} = (x_1, \dots, x_m)$ must be mapped to an operation all of whose inputs are functions of time.

Therefore, the lifting *MakeMoving* denoted as T is mathematically defined as:

$$\begin{aligned} a &\xrightarrow{T} (t \rightarrow a) = a_t \\ f(\bar{X}) &\xrightarrow{T} f(\bar{X})_t = f(\bar{X}_t) \end{aligned} \quad (6-1)$$

As expected, this lifting is commutative because:

$$\begin{aligned} \forall f, g : (T(f).T(g)) (\bar{X}_t) \\ &= (T(f)) (T(g) (\bar{X}_t)) = (T(f)) (g(\bar{X})_t) \\ &= (f(g(\bar{X})))_t = (f.g(\bar{X}))_t = (T(f.g)) (\bar{X}_t) \end{aligned} \quad (6-2)$$

In the hierarchical framework constructed in section 4.2 for spatial analyses, each analysis is defined as a combination of primary operations. These definitions are independent of dimension, so they are valid for any dimension. Thus, having implemented

the data types and primary operations for n -dimensional objects, the lifting *MakeMoving* will extend them to n -dimensional moving objects.

6.2 Implementation of the mappings (liftings)

First we define a class *lifting* to lift the data types and operations:

```
class Lifting f a where
  lift0 :: a -> f a
  lift1 :: (a -> b) -> f a -> f b
  lift2 :: (a -> b -> c) -> f a -> f b -> f c
  lift3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

The `lift0` lifts the data types and the `lift1`, `lift2` and `lift3` are used to lift operations with one, two, and three arguments, respectively. Lifting the operations with more arguments is done in a similar way.

A time instant is considered as a floating number. Then, a changing version of a value of type v is a function of time (instant) to that value:

```
type Instant      = Float
type Changing v   = Instant -> v
```

For example:

```
type MovingInt = Changing (Int)
type MovingPt a = Changing (Pt a)
```

An instance of the class *lifting* is implemented for extension to moving values:

```
instance Lifting ((->) Instant) a where
  lift0 a      = \t -> a
  lift1 op a    = \t -> op (a t)
  lift2 op a b  = \t -> op (a t) (b t)
  lift3 op a b c = \t -> op (a t) (b t) (c t)
```

6.3 Extension of primitive operations to moving objects

In this section, the primitive operations defined in the previous chapter are extended to moving objects using the lifting *MakeMoving*.

6.3.1 Extension of operations on *Ring*

The operations on *Ring* are extended to changing values by applying the above liftings:

```
instance Ring a => Ring (Changing a) where
  (+) = lift2 (+)
  (*) = lift2 (*)
  neg = lift1 neg
  sq  = lift1 sq
```

Note that only the primitive operations are lifted; the combined operations (i.e., -, sq and sum) are automatically lifted.

6.3.2 Extension of operations with list arguments

To extend the list operations to support moving objects, we customize the above liftings to support functions with list(s) as argument(s): the parameter τ must be added to all the elements of the list argument(s). For other elements, it is a simple lifting:

```
convert2Ft x = \t -> x t

lift0L      a      = map convert2Ft a

lift1L  op a      = op (map convert2Ft a)

lift2L  op a b    = op (map convert2Ft a) (convert2Ft b)
lift2LL op a b    = op (map convert2Ft a) (map convert2Ft b)
```

For example:

```

sort = lift1L sort
cw   = lift2L cw

```

In this case, all the elements of the list are functions of time, waiting for a time instant to be further processed, i.e., these liftings (referred to as *convert2Ft lifts* hereafter) result in a list of changing elements: $[\lambda t \rightarrow x1\ t, \lambda t \rightarrow x2\ t, \lambda t \rightarrow x3\ t, \dots]$.

Although the semantic of *convert2Ft lifts* is true, their executions do not terminate in some cases. The reason is that the process reaches a point that needs making a final decision for which the time instant must be specified: These are the cases where not only the values, but also the order or the number of elements of the list is changing, i.e., depends on the time instant (e.g., sorting and filtering the changing elements of a list). In lambda calculus language, in such cases, the process ends up in a *weak head normal form (WHNF)* expression that cannot be further reduced until the time instant is given (see section 3.4.3).

A solution to this problem is that the list of changing elements is converted to a changing list of elements, i.e., $\lambda t \rightarrow [x1\ t, x2\ t, x3\ t, \dots]$. In this case, the order and the number of elements of the list after applying the function is specified, which is in *head normal form (HNF)*, nevertheless it is still a function of time (these set of liftings is referred to as *lc2cl lifts* hereafter):

```

lc2cl :: [Changing a] -> Changing [a]
lc2cl ma = \t -> lift1 (\a -> a t) ma

lift0L    a      = lc2cl a

lift1L   op a     = lift1 op (lc2cl a)

lift2L   op a b   = lift2 op (lc2cl a) b
lift2LL  op a b   = lift2 op (lc2cl a) (lc2cl b)

```

For example:

```

head (IF (t>3) (\t. cons (t+1) nil) (\t. cons (t-2) nil)) 2

```

is in WHNF and so cannot be further processed, because the outer most expression cannot be β -reduced. However, after applying $lc2c1$, it will become:

```
 $\lambda t. \text{head (IF (t>3) (cons (t+1) nil) (cons (t-2) nil)) 2}$ 
```

that can be reduced to

```
 $\text{head (IF (2>3) (cons (2+1) nil) (cons (2-2) nil)) 2} \rightarrow$   
 $(\text{cons (2-2) nil}) \rightarrow (\text{cons 0 nil})$ 
```

which is in HNF. These new types of liftings work well. However, their efficiency still needs to be evaluated.

6.4 Summary

This chapter introduced a mapping called *MakeMoving* that extends the n -dimensional data types and operations developed in the previous chapters to moving objects.

7 RESULTS AND EVALUATION

This chapter presents the implementation results for extending the Delaunay triangulation to different dimensions. We then evaluate and discuss the performance of the implementations. Finally, two applications developed upon the implementations are presented to show how the proposed approach can be practically used.

7.1 Implementation results

The implementation of the dimension independent Delaunay triangulation and its dual, Voronoi diagram, was applied on a data set, given in Appendix 2 under the heading “Samples”, consists of four collections of twenty 2D static, 3D static, 2D moving and 3D moving points. For example:

```
pt2D      = [3, 4]           -- 2D static point
pt3D      = [1, 2, 1]       -- 3D static point
mpt2D t   = [(7-5*t), (2+5*t)] -- 2D moving point
mpt3D t   = [(3+2*t), (1-4*t), (2+3*t)] -- 3D moving point
```

Figures Figure 7.1 to Figure 7.5 illustrate the results of applying the implemented spatial analyses to data of different dimensions. In the case of moving points, the results for some time instants are presented. The Voronoi diagrams of 3D data sets are not shown because their representation in 2D is not informative.

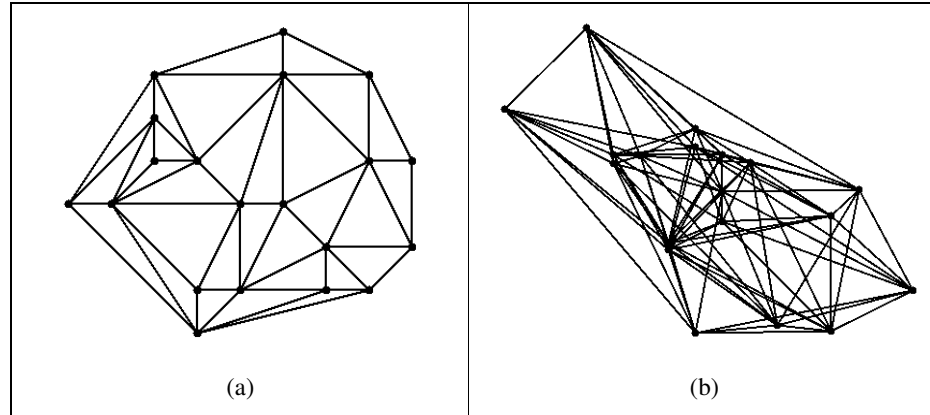


Figure 7.1. Delaunay triangulation of static points (a) 2D (b) 3D (projected)

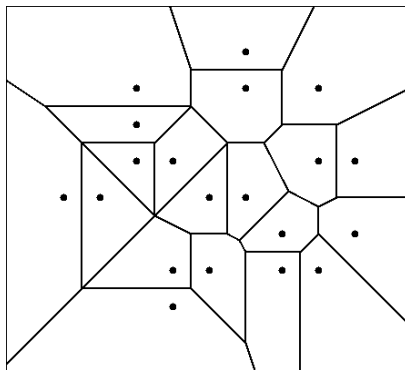


Figure 7.2. Voronoi diagram of 2D static points

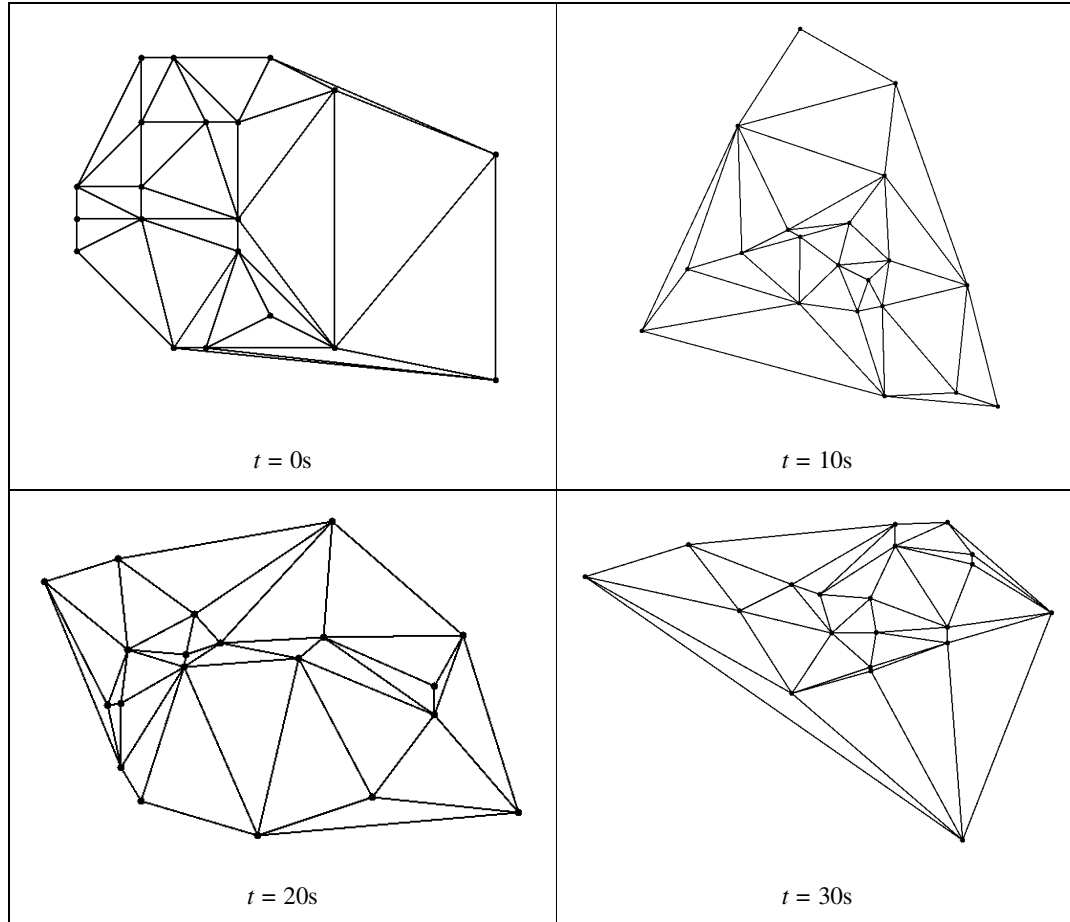


Figure 7.3. Delaunay triangulation of 2D moving points for some time instants

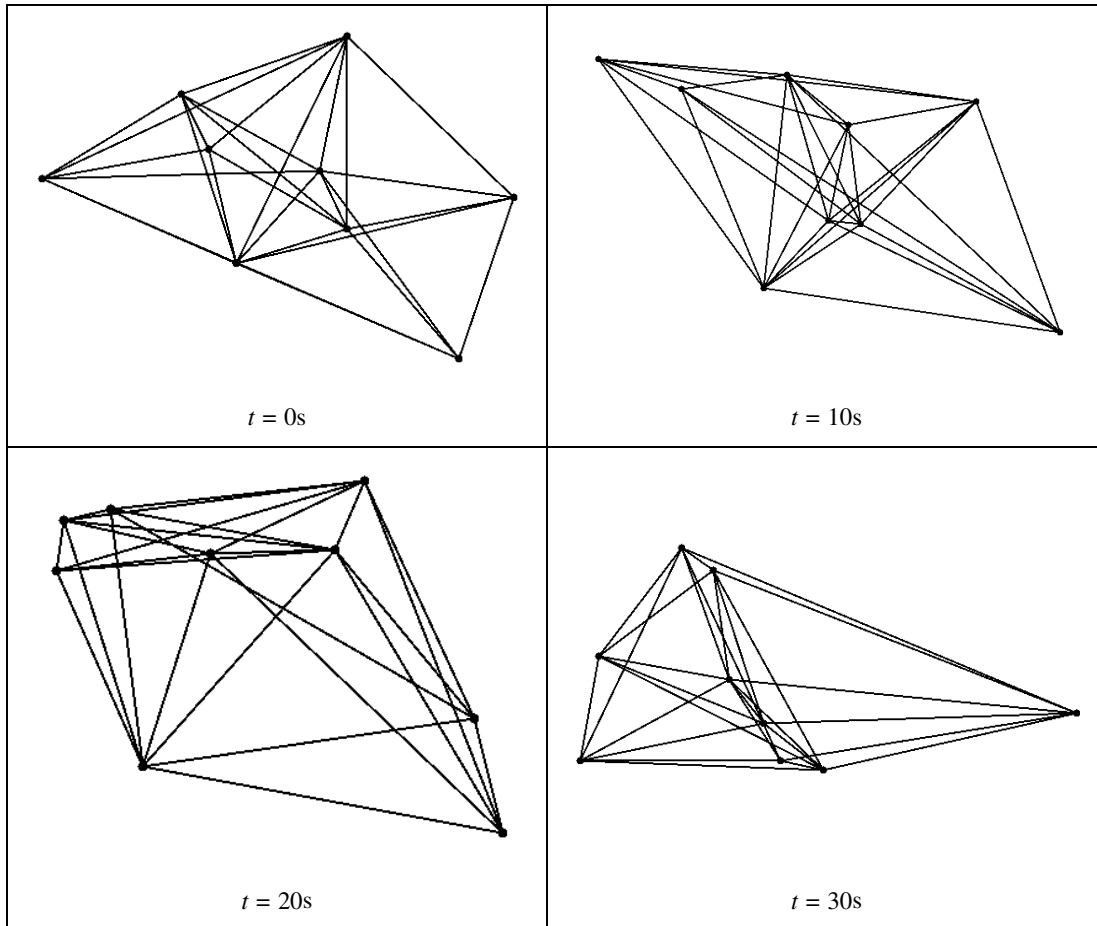


Figure 7.4. Delaunay triangulation of 3D moving points for some time instants (projected)

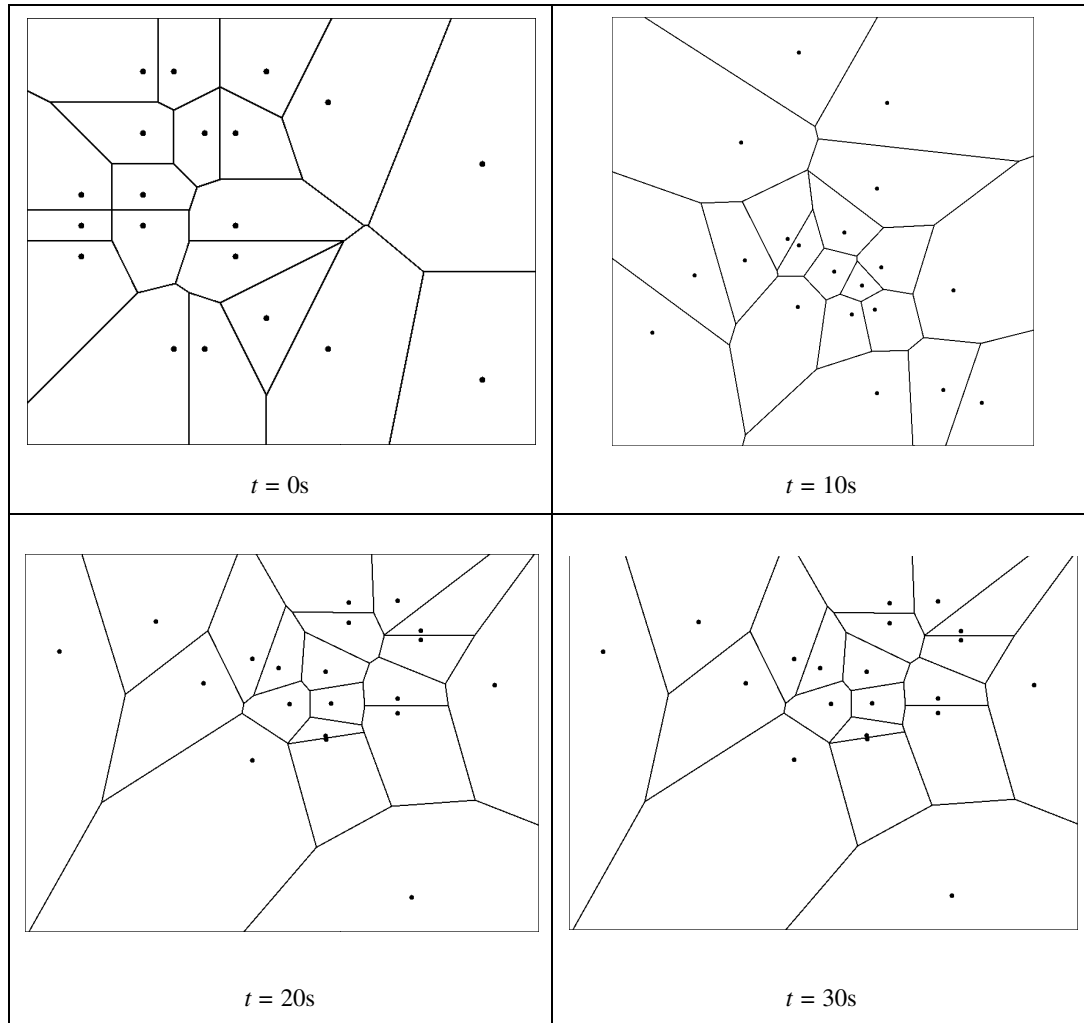


Figure 7.5. Voronoi diagram of 2D moving points for some time instants

In the above examples, the moving points were models as continuous functions of time. In this case, a value $f(t)$ is available for each time instant t . It is called *intensional* function definition (Figure 7.6.a). In contrast, in *extensional* definition of a function, the function is defined for a set of discrete values (Figure 7.6.b), between which we interpolate (Frank, 2007). Although there are examples of deploying intensionally defined functions in GIS (Mostafavi, 2002), moving objects are usually defined by extensional functions. The data collected by navigation systems for a moving car is an example of such data in which the positioning is accomplished at certain time intervals.

$f(x) = 3x^2 + \sin x - 3$ <p>(a)</p>	<table border="1"> <tr> <td>x</td> <td>1</td> <td>3</td> <td>6</td> <td>8</td> <td>14</td> <td>19</td> <td>25</td> <td>34</td> <td>...</td> </tr> <tr> <td>$f(x)$</td> <td>4</td> <td>8</td> <td>15</td> <td>18</td> <td>13</td> <td>9</td> <td>7</td> <td>3</td> <td>...</td> </tr> </table> <p>(b)</p>	x	1	3	6	8	14	19	25	34	...	$f(x)$	4	8	15	18	13	9	7	3	...
x	1	3	6	8	14	19	25	34	...												
$f(x)$	4	8	15	18	13	9	7	3	...												

Figure 7.6. (a) Intensional and (b) extensional definition of a function

To provide a continuous representation of an extensional function we developed an interpolation method: The position of each moving point is given for a set of discrete time instants results in a list of time-position pairs $P = \{(t_1, p_1), (t_2, p_2), \dots, (t_n, p_n)\}$. For a time instant t , if t is not available in P , then the position of the point is linearly interpolated using its neighbors in P . The implementations were applied on ten moving points simulated on a street network (Figures Figure 7.7 and Figure 7.8). Figures Figure 7.9 and Figure 7.10 show the result of applying the Delaunay triangulation and Voronoi diagram on the moving points at some time instants.

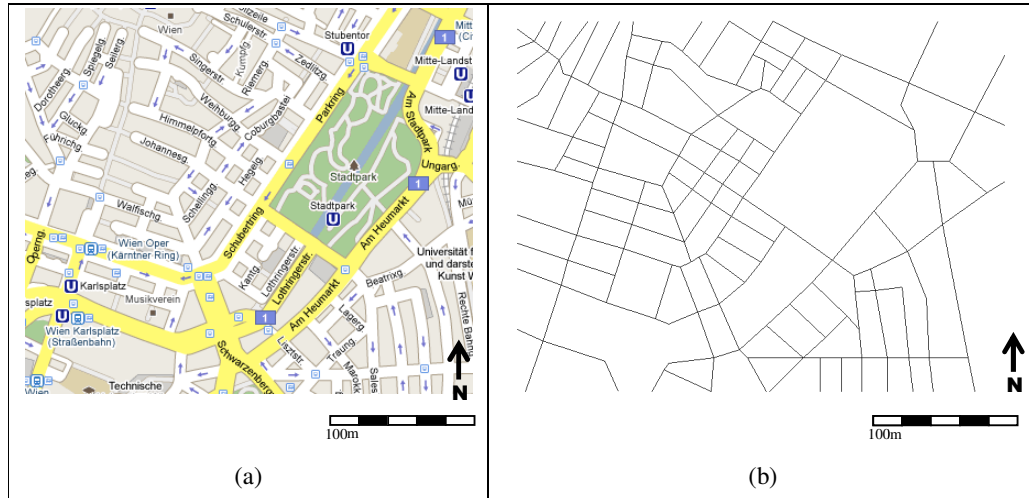


Figure 7.7. The study area (a) Map of the street network (b) Model of the street network

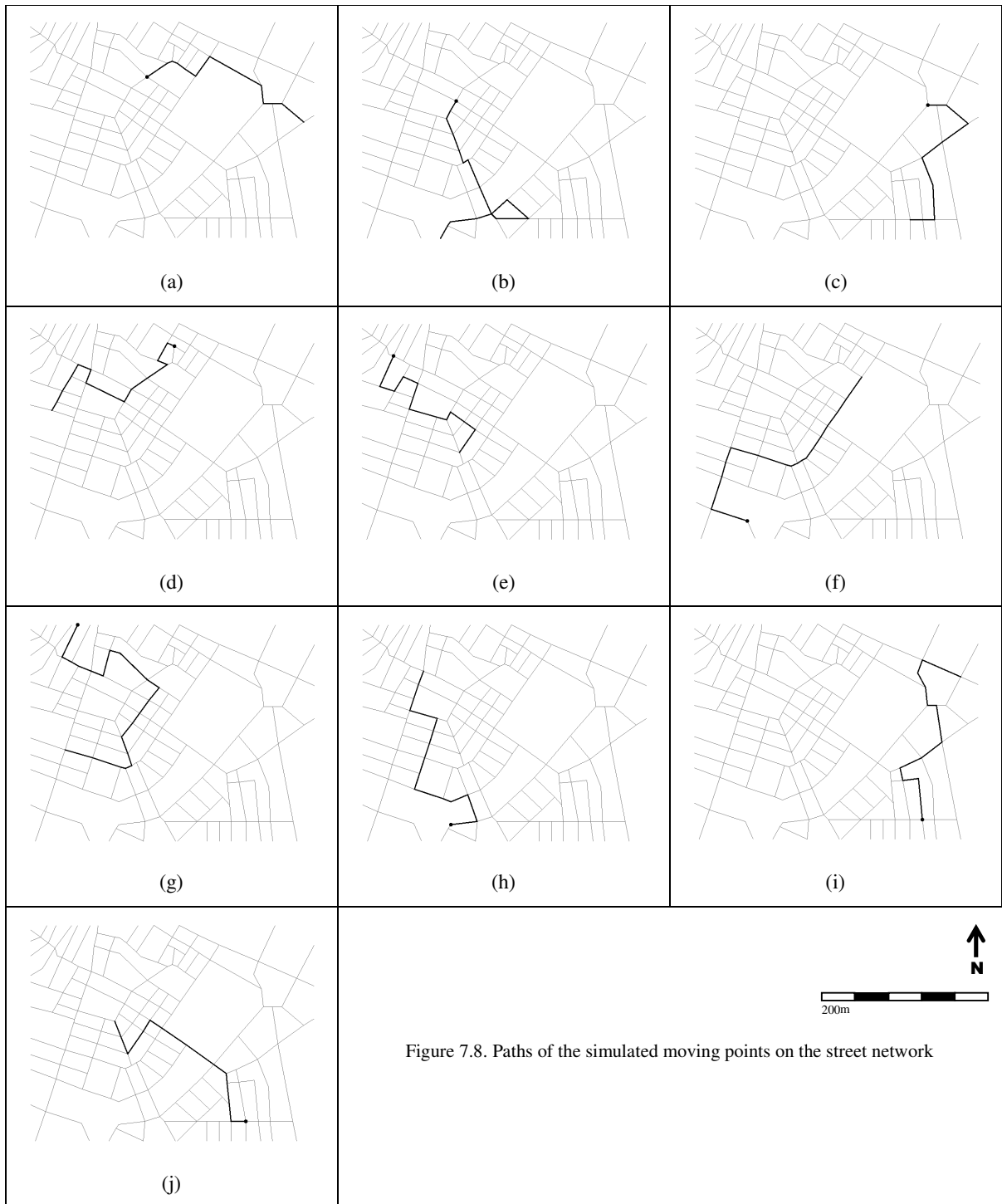


Figure 7.8. Paths of the simulated moving points on the street network

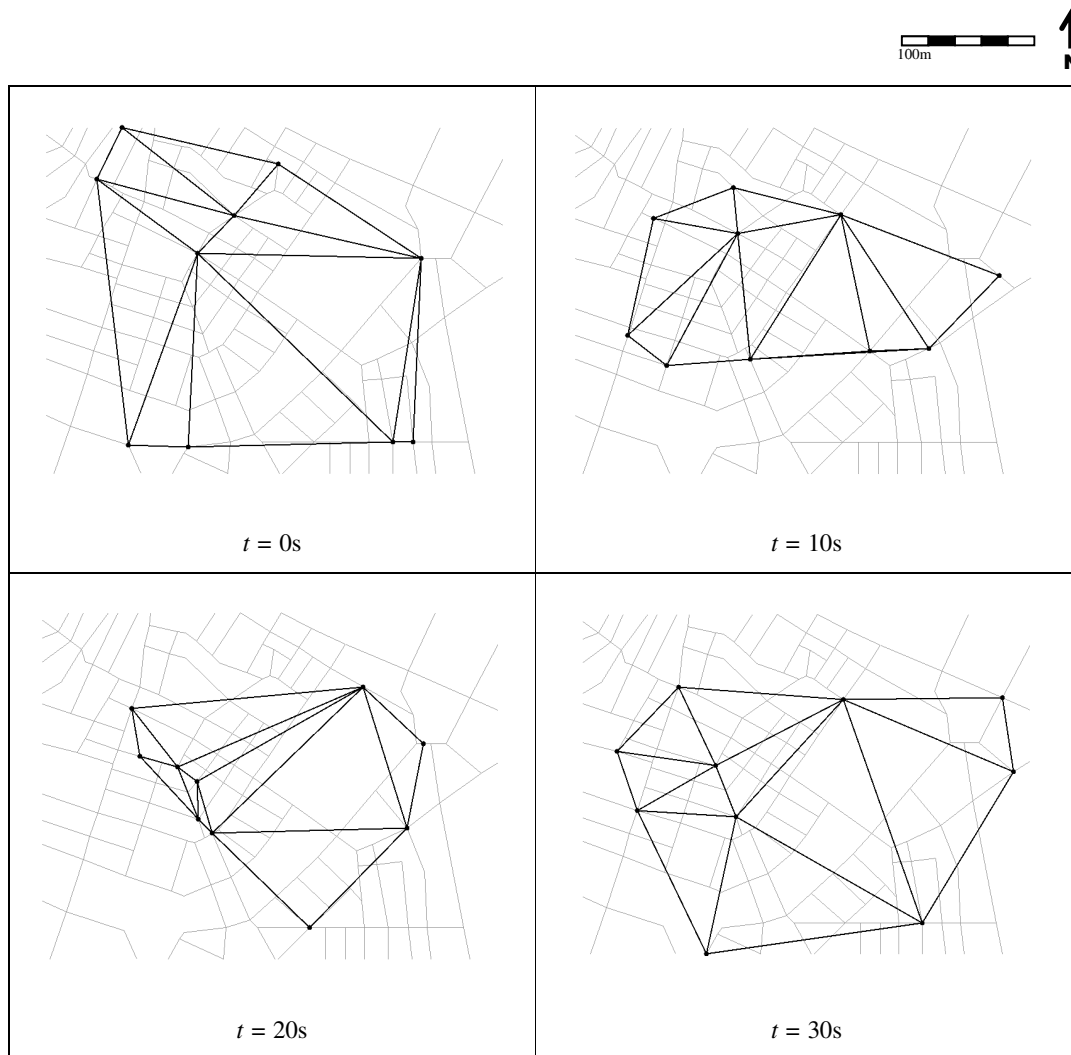


Figure 7.9. Delaunay triangulation of the simulated moving points on the street network for some time instants

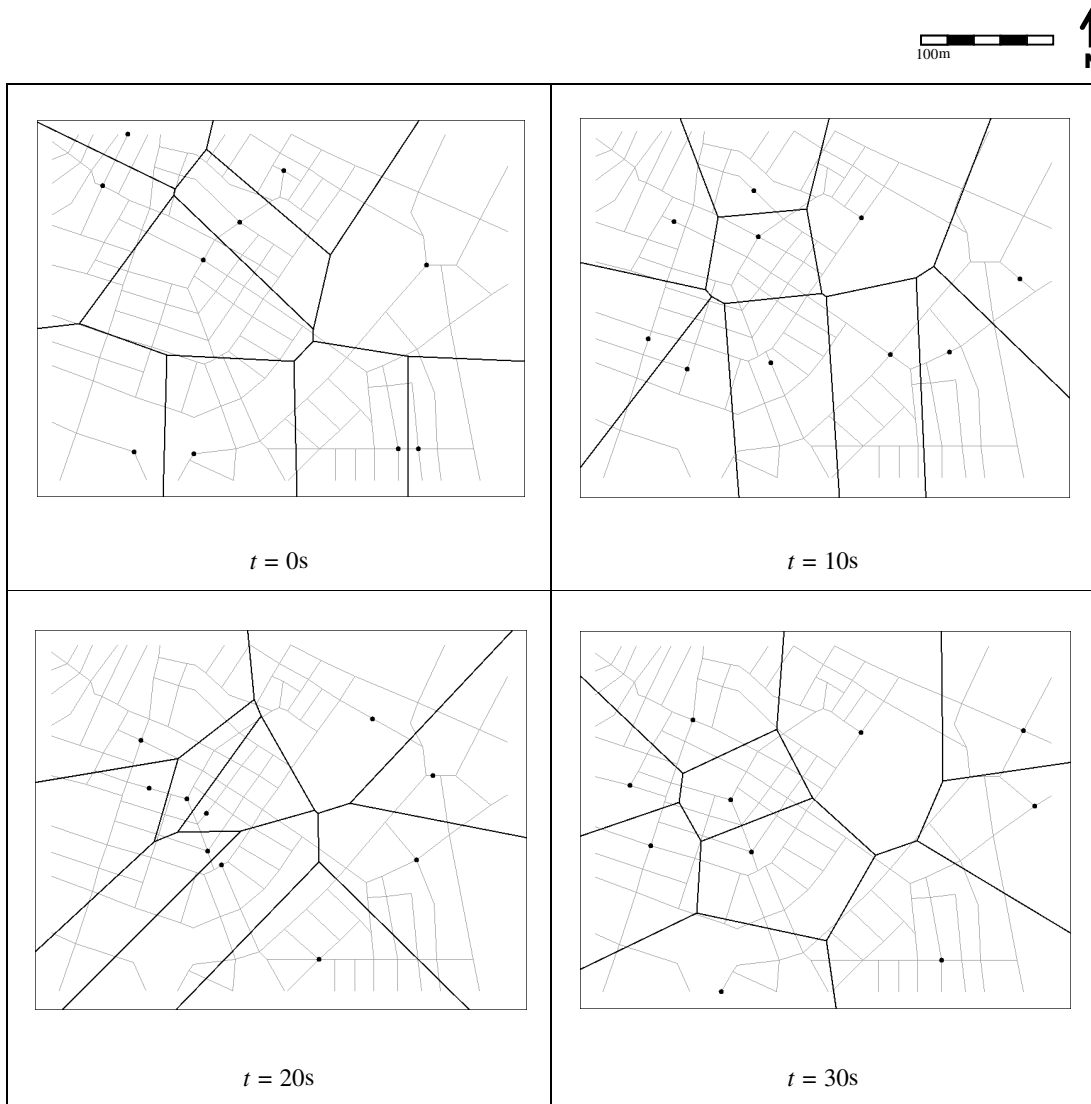


Figure 7.10. Voronoi diagram of the simulated moving points on the street network for some time instants

7.2 Evaluation

The goal of this research is to extend spatial analysis to different dimensions without recoding. Our main concern is on the mathematical validation of the conceptual framework and investigation of its implementation issues. The discussion presented in chapters 4 to 6 as well as the above implementations and results show that our goal is achieved.

This section evaluates the efficiency of the implementations and results. Table 7.1 and Figure 7.11 illustrate the running time as a function of number of input points for the 2D/3D static and moving Delaunay triangulation. Their investigation shows that the complexity of implementing the *Bowyer-Watson* algorithm to compute the Delaunay triangulation is $O(n \log n)$ and $O(n^2)$ respectively for 2D and 3D points, which was expected.

Table 7.1. Running time (in sec.) as a function of number of input points for 2D/3D static/moving DT/CH

No. of pints		10	50	250	500	1000	2000	4000	8000	16000	32000	64000
Static DT	2D	0.01	0.07	0.47	1.08	2.43	5.45	12.02	25.61	56.02	120.10	255.28
	3D	0.02	0.18	0.49	1.04	3.83	9.04	24.71	70.33	198.41	696.92	2859.35
Moving DT	2D	0.01	0.09	0.50	1.03	2.48	5.29	13.03	23.13	62.05	128.38	272.45
	3D	0.02	0.20	0.52	1.03	3.87	9.18	25.00	70.51	189.85	712.67	2777.85

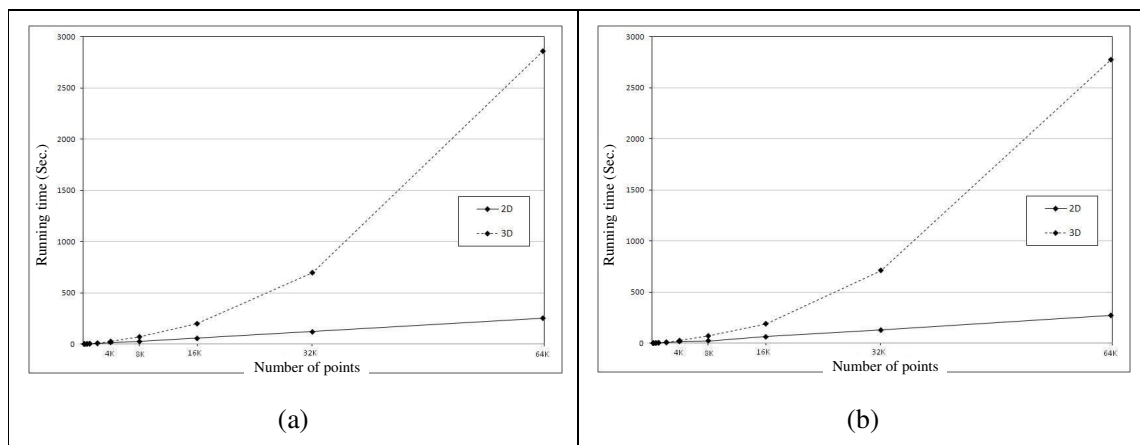


Figure 7.11. Running time as a function of number of input points for 2D and 3D static and moving Delaunay triangulation: (a) static (b) moving

On the other hand, for the same number of points, the running time to compute the Delaunay triangulation of 3D points is greater than 2D. This is because of the more and

bigger size of the matrixes that must be dealt with in 3D. Note that the complexity of matrix calculations depend on the size of the matrix (e.g., the complexity of determinant calculation is $O(n^3)$ (Kaltofen and Villard, 2004)). For instance, to check if a point is inside a tetrahedron (the case for 3D) it computes four 4×4 matrixes, while this is three 3×3 matrixes for a point against a triangle (the case for 2D); or to check if a point is inside the circum-sphere of a tetrahedron (the case for 3D) it computes a 4×4 matrix, while this is a 3×3 matrix for a point against the circum-circle of a triangle (the case for 2D). In abstract, the running time is a function of the number of n -simplexes as well as the size of the computation units, which depends on the dimension.

Finally, the running times to apply the Delaunay triangulation on the same number of static or moving points are quite similar. Note that the presented running times for moving points is the time needed to reduce the analyses to their simplest form, which are functions of time. To determine the final result for a certain time instant t , this t must be given to the time dependent function.

To compare the efficiency of the implementations to be applied on moving points at multiple time instants, we applied some spatial analyses on a data set containing 20 moving points. The analyses used in this evaluation are:

- Distance between two points
- The area (volume) of a triangle (tetrahedron) constructed by three (four) points.
- Clock-wise (CW) order test
- InSphere test
- Sorting a set of points
- Delaunay triangulation of a set of points

In the case of moving points – where the outputs of applying analyses to the moving points are functions of time – each analysis was applied for 1, 5, 10 and 20 different time instants. To evaluate the efficiency, GHC profiler was used. Among other detailed information, it gives the time and the number of reductions (number of steps to get the simplest form – see chapter 3) for running the code. The time parameter is not discussed here, because the running times are too short and not very informative. Moreover, the relationship of the number of reductions and running time is quite linear. Table 7.2 and Figure 7.12 illustrate the number of reductions for different cases. These results show that

the number of reductions for "static points" and "moving points for 1 time instant" are quite similar. It means that points with constant elements (static points) are treated the same as points with functional elements (moving points). It is because of this characteristic of functional languages that all values are functions: "3" is a constant function, while "2x" is a function of one parameter x .

Table 7.2. Number of reductions for applying different analyses on 2D/3D static/moving points. In the case of moving, the analyses are applied for multiple time instants

Analysis	Dim	Number of reductions				
		Static Points	Moving Points			
			1 time	5 times	10 times	20 times
Distance	2D	8,143	8,600	13,912	20,192	30,372
	3D	8,235	8,912	15,064	21,220	32,252
Volume	2D	10,694	11,084	25,936	42,384	77,776
	3D	17,682	18,348	61,040	114,400	221,120
CW test	2D	9,534	9,932	34,004	65,592	122,952
	3D	13,894	14,328	48,288	86,944	173,228
InSphere test	2D	40,745	41,172	132,040	260,696	497,036
	3D	86,559	87,172	284,152	551,628	1,070,580
Sort	2D	48,251	50,596	234,256	456,316	902,940
	3D	70,904	72,700	340,756	688,806	1,297,972
Delaunay triangulation	2D	12,684,532	12,693,748	62,968,188	123,809,361	244,840,892
	3D	28,800,201	28,811,448	140,345,336	270,689,236	571,929,332

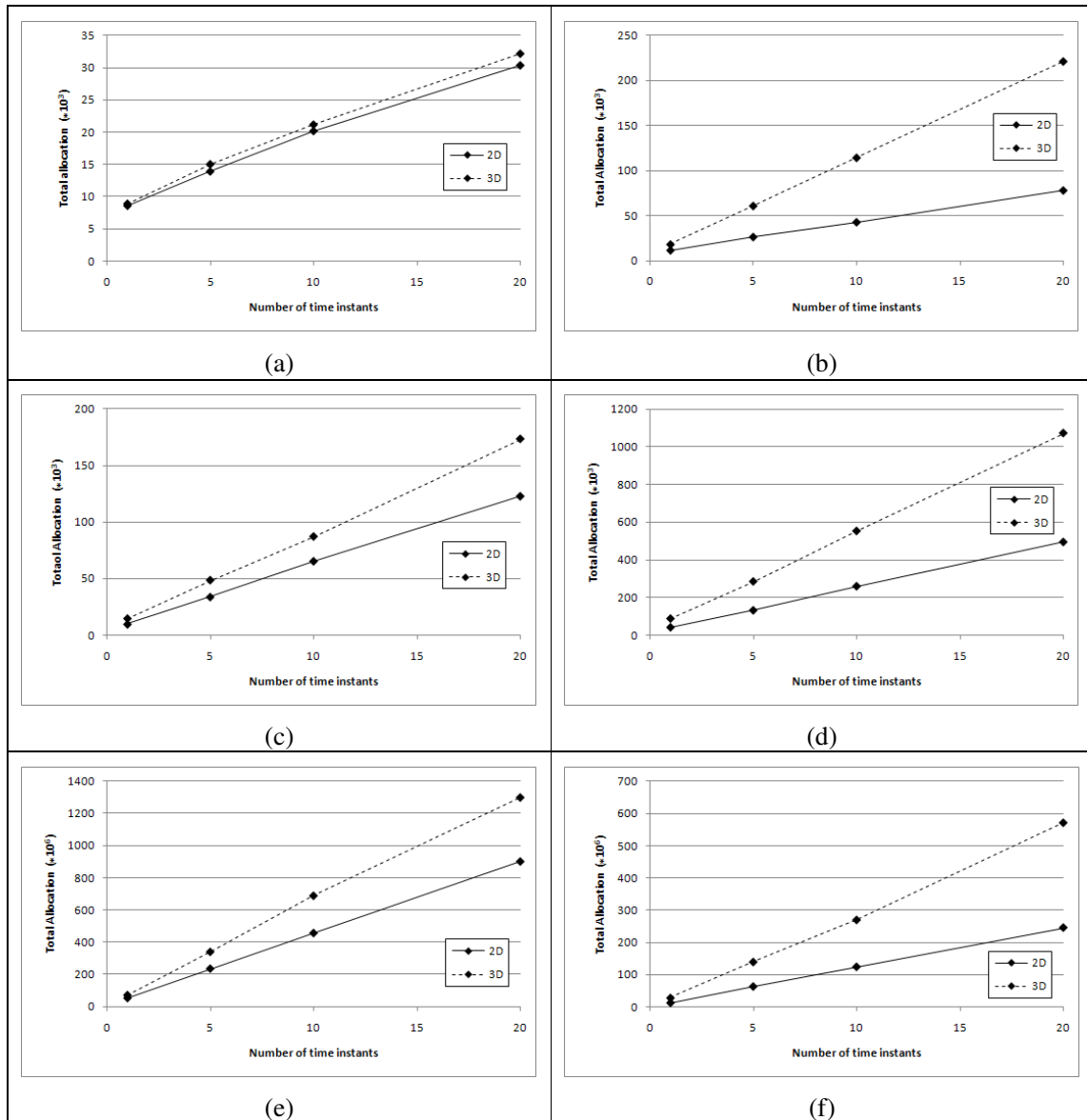


Figure 7.12. Number of reductions for applying different analyses on 2D/3D static/moving points. In the case of moving, the analyses are applied for multiple time instants (a) Distance (b) Volume (c) CW test (d) InSphere test (e) Sort (f) DT

For the *Distance*, *Volume*, *CW test* and *InSpher test*, if we consider applying the analysis on moving points for n_1 and n_2 number of time instants (e.g., $n_1 = 5$ and $n_2 = 20$), the increase of the number of reductions is less than n_2/n_1 (e.g., $n_2/n_1 = 20/5 = 4$). It shows that the lifting process works as we expected: the result is calculated as a function of time, and its value for a specific time instant, t_0 , is calculated through replacing the parameter t in the final function with t_0 .

For the *Sort* and *Delaunay triangulation*, if we consider applying the analysis on moving points for n_1 and n_2 number of time instants (e.g., $n_1 = 5$ and $n_2 = 20$), the increase of the number of reductions is about n_2/n_1 (e.g., $n_2/n_1 = 20/5 = 4$). The reason is that in the first event that a final decision is needed, *lc2cl* calculates the elements of the input list for the desired time instant and from now on it is processed as a list of static values.

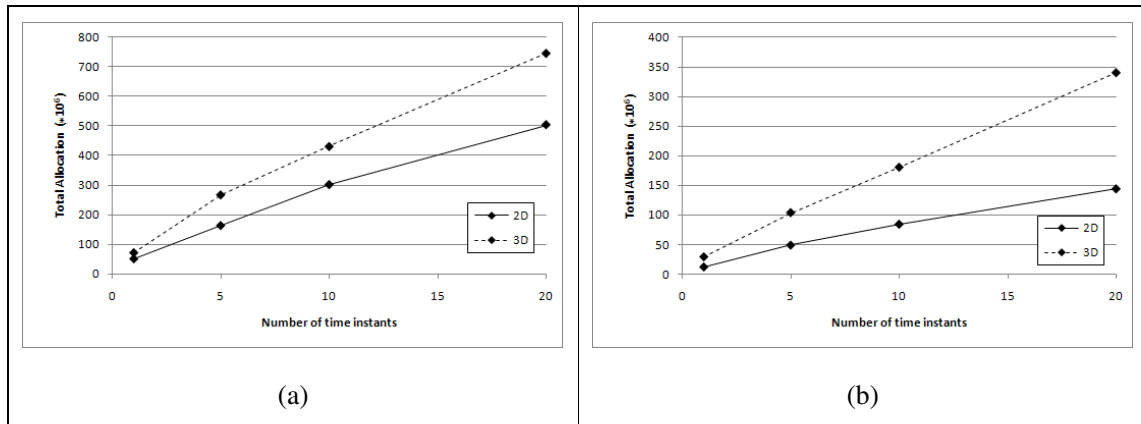
The above observation seems disappointing, but actually it is not, because the concept of our approach is true; and if the *convert2Ft lifts* worked, we would get the same results for the *Sort* and *Delaunay triangulation* as well. In other words, the concept of the approach is true, but the development environment is not completely supportive yet to interact with complex types of changing (e.g., changing the order or the number of the elements of a list). On the other hand, the *lc2cl lifts* enabled us to avoid rewriting the whole algorithms again; and in the worst case, it works as efficient as current approaches that recode each algorithm for each data type.

On the other hand, detection of topological events and locally updating the data structure in an imperative programming language must be handled manually. However, no effort is needed for such update in our implementation, because of the lazy evaluation (second ingredient of lazy evaluation in chapter 3). To certify this, we did two tests:

In the first test, the selected analyses were applied on some points move slower than the first data set. The results are shown in Table 7.3 and Figure 7.13. In this case, the increase in the number of reductions is less comparing to the first data set that moves faster (Table 7.2 and Figure 7.12). It means that the occurrence of the topological events is truly detected and the updates perform on these events; because if the points move slower, it takes longer time for the topological events to occur. Therefore, in a certain time interval, the number of topological events decrease, which results in less updates.

Table 7.3. Number of reductions for applying *Sort* and *DT* on 2D/3D slow moving points

Analysis	Dim	Number of reductions			
		1 time	5 times	10 times	20 times
Sort	2D	50,596	162,740	300,902	502,737
	3D	72,700	267,834	431,132	745,362
Delaunay triangulation	2D	12,693,748	49,569,245	84,893,407	144,673,912
	3D	28,811,448	103,604,529	180,512,763	340,523,480

Figure 7.13. Number of reductions for applying *Sort* and *DT* on 2D/3D slow moving points

In the second test, instead of all points, we moved one and two points of the point set. In the case of moving all points, it is most likely that the structure must be thoroughly updated. However, when only one or two points move, most of the structure is not affected after a movement and a local update would be enough. As shown in Table 7.4 and Figure 7.14, if one or two points move, the increase of the number of reductions for n_1 and n_2 number of time instants is significantly less than n_2/n_1 . It means that those parts of the structure that is not affected after the movement has been reused for updating. Note that the number of reductions when one point moves is less than the case of moving two points, because moving two points affects the structure more than moving one point.

Table 7.4. Number of reductions for applying *DT* on 2D and 3D moving points for multiple time instants where different number of points move

Dim	Case	Number of reductions			
		1 time	5 times	10 times	20 times
2D	All points move	12,693,748	62,968,188	123,809,361	244,840,892
	Two points move	12,693,748	39,798,093	73,498,820	143,452,905
	One point moves	12,693,748	16,544,532	24,662,760	39,620,932
3D	All points move	28,811,448	140,345,336	270,689,236	571,929,332
	Two points move	28,811,448	98,809,453	201,045,832	400,004,561
	One point moves	28,811,448	42,345,336	76,893,415	151,116,732

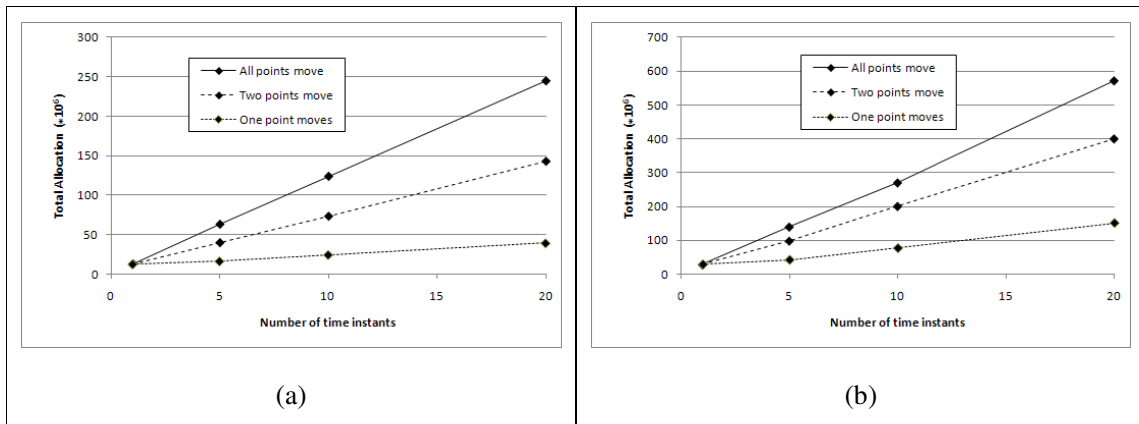


Figure 7.14. Number of reductions for applying DT on 2D and 3D moving points for multiple time instants where different number of points move (a) 2D (b) 3D

7.3 Applications

This section presents two applications developed upon the implementations in order to show how the proposed approach can be practically used.

We implement the convex decomposition of non-convex polytopes of any dimension, based on a method called Alternate Hierarchical Decomposition (AHD). This method uses an iterative procedure to represent a polytop as a tree of convex components. The root of this tree is the convex hull of the vertices of the polytop, and other convex components are located at the next levels with alternate signs (positive for even and negative for odd levels). The algorithm and the implementation details are described in (Bulbul, 2011; Karimipour, 2009). Figures Figure 7.16 and Figure 7.17 show the AHD representations of the 2D and 3D non-convex polytopes of Figure 7.15.

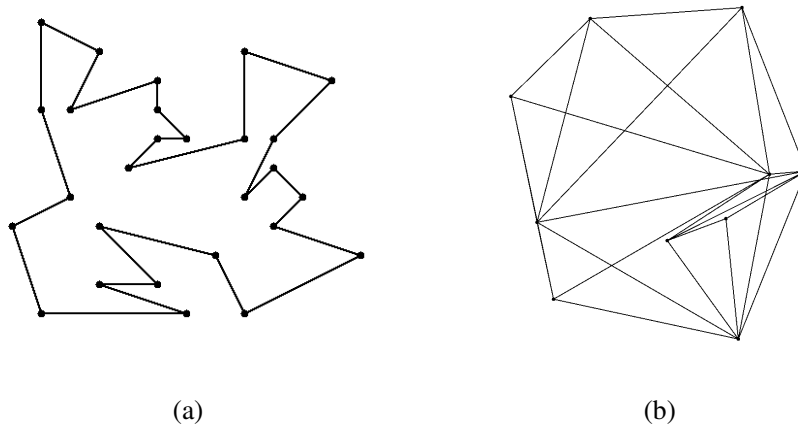


Figure 7.15. (a) 2D and (b) 3D non-convex polytop

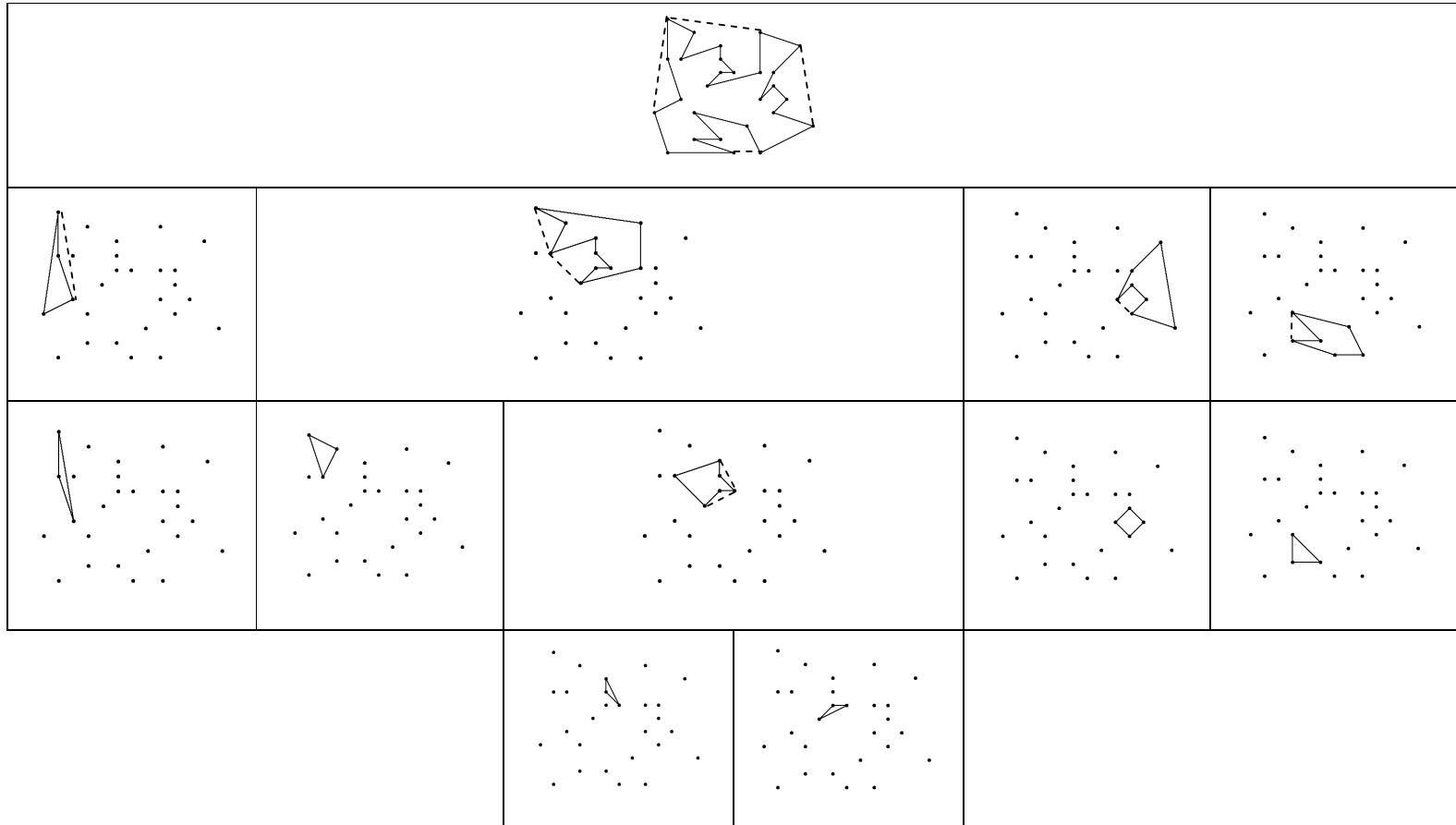


Figure 7.16. AHD representation of the polytope of Figure 7.15.a

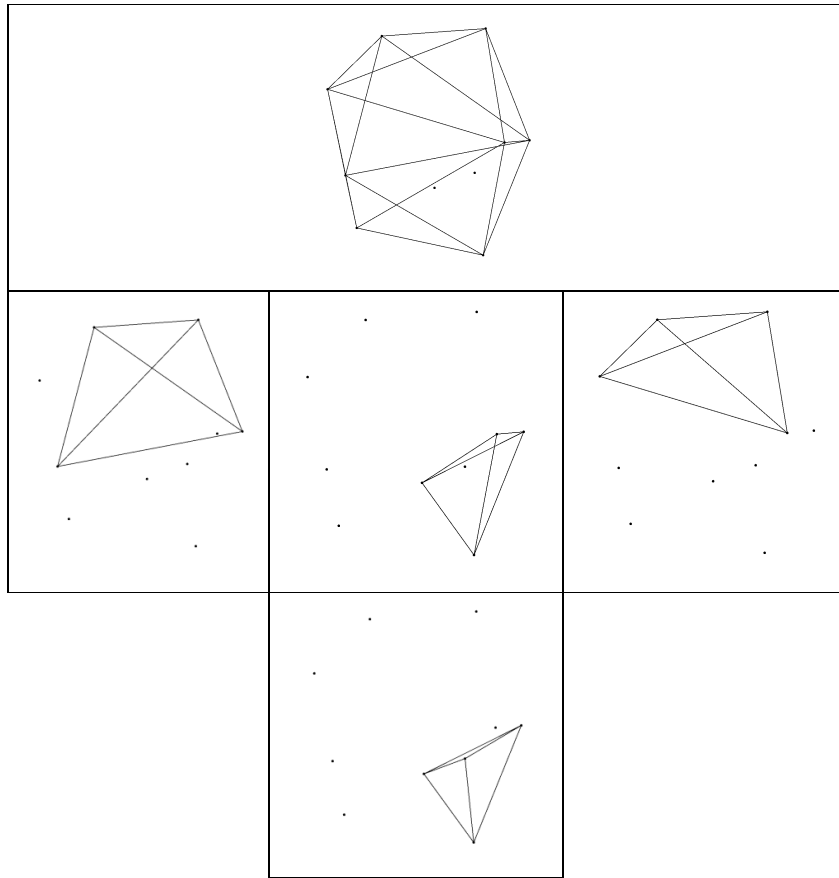


Figure 7.17. AHD representation of the polytop of Figure 7.15.b

We developed a method to calculate the volume of n -dimensional polytops based on the convex decomposition and Delaunay triangulation (Karimipour et al., 2010a; Karimipour et al., 2010b). It was used to calculate the area and volume of the reservoir of a dam at different water levels, which leads to a level-surface-volume diagram (Karimipour et al., 2010a; Karimipour et al., 2010b). This diagram is important for managing the water consumption and monitoring the dam construction: observing the daily water level, this diagram is used to estimate the surface area and water amount of the reservoir. This information helps the decision makers in applications like water usage allocation, dam deformation control and managing water release behind the dam.

The Lalyan dam – located in North East of Tehran, Iran – was selected as the case study (Figures Figure 7.18 and Figure 7.19). The bed of the dam reservoir was surveyed in a hydrographic process (Figure 7.20) and its 3D TIN was produced (Figure 7.21).



Figure 7.18. Satellite image of Latyan dam and its reservoir

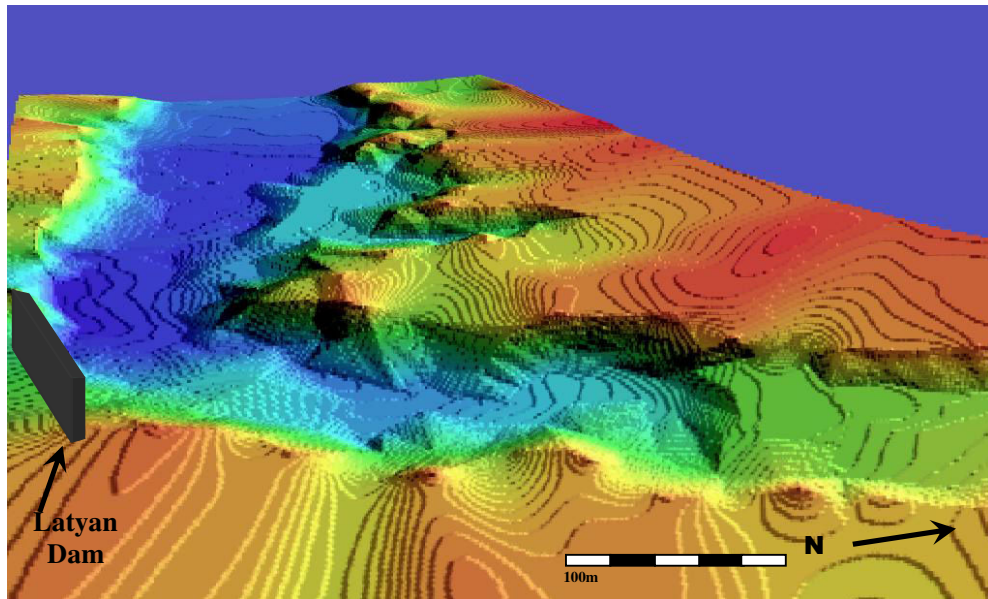


Figure 7.19. 3D view of Latyan dam and its reservoir

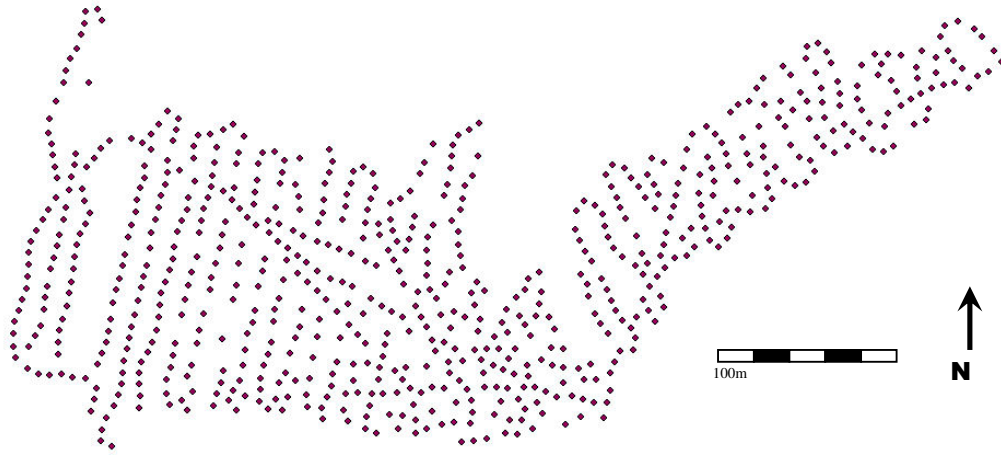


Figure 7.20. Points resulted from hydrography of Latyan dam reservoir

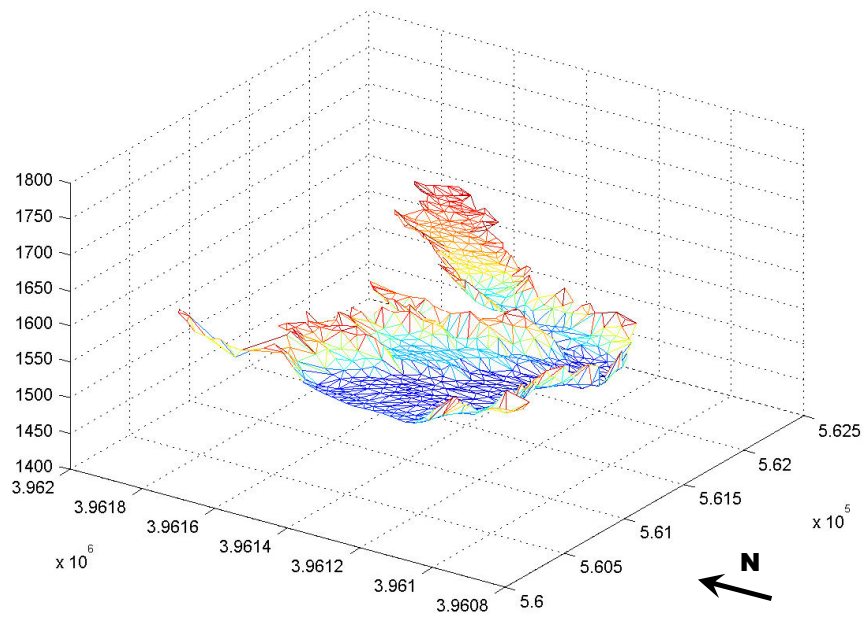


Figure 7.21. 3D TIN of Latyan dam reservoir

To calculate the area and volume of the reservoir at a certain water level, say h , the 3D TIN was intersected with the plan $z=h$, which results in the volume of the reservoir where $z < h$ and the surface of the reservoir at $z=h$. Figure 7.22 shows the results for the water level of 1570m. To calculate the area and volume of the results, the implemented n -dimensional Delaunay triangulation was used: For a convex n -dimensional structure, it is triangulated to a set of n -simplexes and then sum of the nD -volume (i.e., area for 2D, volume for 3D, etc.) of the components are calculated. The absolute value of the determinant used to specify the orientation of an n -simplex yields its nD -volume:

```
vSimp s = abs . det . map (1:) $ s
vConv p = sum . map vSimp . dt $ p
```

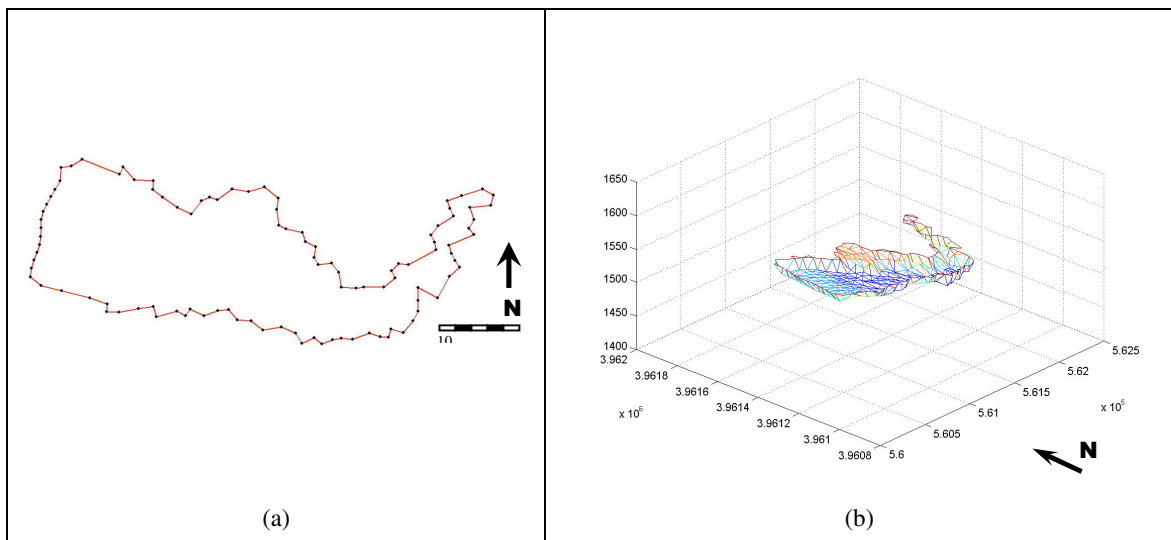


Figure 7.22. 3D TIN and surface of Latyan dam reservoir at water level of 1570m

As Figure 7.22 shows, our structures are non-convex. Therefore, first they must be decomposed to a set of convex components and then the above calculation is applied separately to each component. For this, the dimension independent decomposition of polytopes was used. Each component is triangulated using the implemented n -dimensional Delaunay triangulation. Calculating the area/volume of each component and summing up the results will provide the total area/volume of the reservoir at the desired water level. The function that takes an n -dimensional polytope and calculates its nD -volume as follow:

```
vPoly p = sum . map vConv . decompose $ p
```

By applying the explained process to different water levels, the level-surface-volume diagram was produced for the reservoir of the Latyan dam, which shows the surface area and volume of the reservoir at different water levels (Figure 7.23).

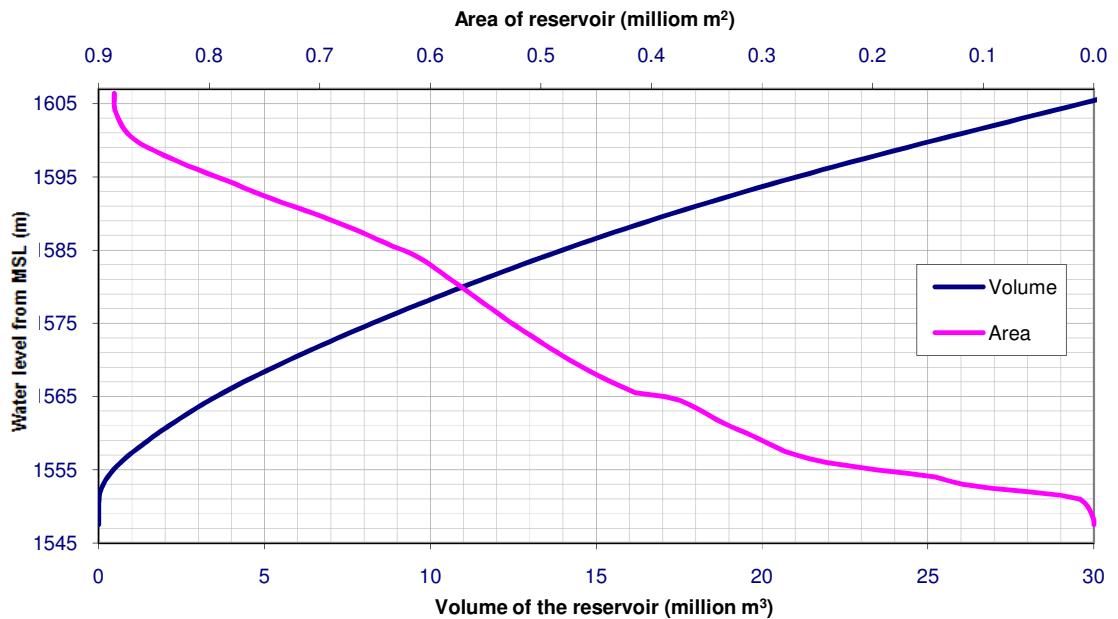


Figure 7.23. Level-Surface-Volume diagram of Latyan dam reservoir

As another application, we implemented an optimum placement algorithm proposed in (Ghosh and Das, 2008; Wang and LaPorta, 2004) to increase the coverage of a sensor network based on the moving Voronoi diagram: The Voronoi diagram of the sensors is constructed and each sensor moves toward its furthest Voronoi vertex (Figure 7.24.a) or it is placed at the center of the smallest enclosing circle of its Voronoi cell (Figure 7.24.b) (Argany et al., 2010). It changes the structure of the Voronoi diagram, so this process is applied iteratively till a certain threshold is reached.

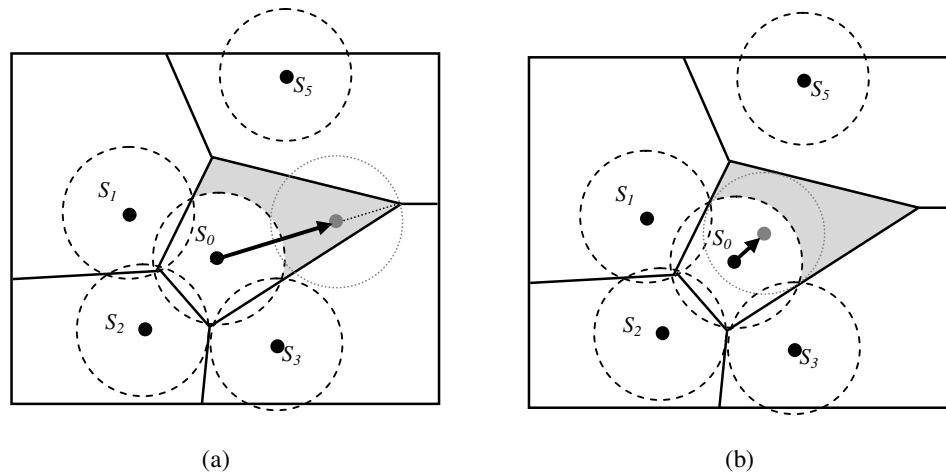


Figure 7.24. Using the Voronoi diagram for sensor network placement (a) Moving the sensors toward the furthest Voronoi vertexes (b) Placing the sensors at the center of the smallest enclosing circle of their Voronoi cells

7.4 Summary

In this chapter we implemented the proposed approach for the selected case studies in Haskell and evaluated the results. We applied the implementations to sample data sets and evaluated and discussed efficiency of the results. The results confirm that the concept of the proposed approach is true and works for analyses with individual inputs; but the development environment is not yet completely ready to support more complex types of changing inputs (e.g., lists). In other words, although the current implementation of functional programming languages support the changing values, but there are some cases of changing (e.g., changing the number and order of values in a list) that are not supported. Nevertheless, even in such cases, our implementation has two advantages: Firstly, we could extend the analyses to moving points without recoding the algorithm. Secondly, the lazy evaluation of Haskell helps us to be more efficient and prevent redoing the unchanged calculations.

We presented two applications developed upon the implementations in order to show how the proposed approach can be practically used. The achieved results certify the hypothesis of the research which says “*studying spatial analyses based on their dimension-independent characteristics leads to a consistent solution toward implementation of a multi-dimensional GIS*”.

8 CONCLUSION AND FUTURE WORK

This chapter summarizes the research of this thesis. It describes all stages through developing and testing the proposed approach to implement dimension independent spatial analyses. We then present the results and major findings of our work, as well as the research contribution. Finally, we propose directions for future research.

The goal of this research was to provide an integrated framework for spatial analyses of multi-dimensional spaces. The proposed approach is to formally define spatial analyses based on their dimension-independent properties. It leads to an integrated framework of spatial analyses, which will further be extended to different dimensions. This extension will be accomplished through the mappings between the spaces, which are independent of analyses. The abstraction and algebraic specifications were used as the formal methods to provide the required abstraction. We described these principals in more details in chapter 1.

To present the state-of-the art of extending spatial analyses to different dimensions, in chapter 2 we reviewed existing solutions to extend the Delaunay triangulation, as the case study of the research, to different dimensions. This information was used to compare the current approach with the proposed approach of the research to extend spatial analyses to different dimensions.

The formal methods used in this thesis were presented in chapter 3. Principals of abstraction, algebraic structures and n -simplexes were presented. The functional programming languages, as the programming environment in this thesis, were introduced and their principals were presented.

These principals were used in chapters 4 to 6 to develop the proposed approach of the research. We used the abstraction methods to develop the integrated framework of spatial analyses based on their dimension-independent properties. In this framework, spatial analyses are formally expressed in a hierarchical way in which each analysis is defined as a combination of simpler ones. These definitions are independent of dimension and the hierarchy ends in a set of primary operations, which are not further decomposed. The data types used in the operations of this hierarchy were also identified. Next, algebraic specification was used to formalize this conceptual integrated framework. It formally

describes the characteristic of analyses as combinations of the elements of the framework. On the other hand, as the spatial analyses are structurally equivalent in the required spaces, mappings (liftings) were defined between different spaces, independent of data types and analyses. Thus, having implemented the dimensionally independent data types and operations, they all will be extended to a specific space by applying the mapping function.

The proposed approach was evaluated through implementation of the Delaunay triangulation for 2D/3D static and moving points in the functional programming language Haskell. Having this spatial analysis at top of a hierarchy, it was decomposed to simpler operations till the primitive operations were achieved. The data types used in the operations of the hierarchy were also identified. On the other hand, the mappings between spaces were defined, which later were used to extend data types, operations and spatial analyses to points of different dimensions. The detailed explanation of the implementation was presented in chapter 7. We evaluated and discussed the performance of the implementations and presented two examples of using the implementations in practice.

8.1 Results and major findings

In this research we investigated studying spatial analyses based on their dimension-independent characteristics. This can be considered as an effort along the goal of GI science to model the interaction of human with the environment. This is different from the approach of current research that extends spatial analyses based on their dimension dependent characteristics. Although such an approach results in extensions with the minimum increase in complexity and speed, they must be implemented separately for each dimension. Here, we study spatial analyses based on their dimension-independent characteristics, and to develop the data types and operations of a space to another, mappings are defined between spaces. These mappings are independent of data types and analyses and only depend on the origin and target spaces. Thus, having implemented the dimensionally independent data types and operations, they all will be extended to a specific space by applying the mapping function. The results of using this approach for the case study verified the validity of the approach.

To construct the integrated framework, the concepts of abstraction and abstract data types were deployed. These concepts were used to define the required data types and data structures in a way that they can support objects of different dimensions. The operations to manipulate these data types and structures were developed independent of dimension, too.

This abstract viewpoint is a simulation of how people understand the environment. Thus, these concepts could be used for other GI related research.

An algebraic approach was used to define spatial analyses. It considers the unified nature of our unique physical reality when handling the included context and permits to combine the developed simple components to create a complex system. Using an algebraic approach, we constructed an integrated hierarchical framework that defines each spatial analysis as a combination of simpler ones, which eventually leads to set of primary operations that are not further decomposed. This hierarchy is independent of the space and only depends on conceptual relationships of spatial analyses. Thus, having developed the operations of a level to a certain dimension, all of the operations and analyses of the higher levels are immediately available in the new dimension. This shows the beauty and capability of algebraic views to interact with a complex system.

To interact with different dimensions, the algebraic structures were used. Different spaces were defined structurally equivalent and extension of the elements of a space to another was accomplished using mappings (liftings) defined between the spaces.

The achieved results of implementing the proposed approach certify the hypothesis of the research which says “*studying spatial analyses based on their dimension-independent characteristics leads to a consistent solution toward implementation of a multi-dimensional GIS*“. Of course, this abstraction applied in definition of data types, data structures and operations will cause losing a significant amount of information available for specific dimensions, so this approach may not provide the simplest and fastest solutions. Though, it does not harm the goal of this research, because our major goal is to present an approach to extend spatial analyses to higher dimensions with the minimum amount of recoding, so simplicity and speed are not our evaluating parameters. In other words, this research believes that it better to have a working comprehensive system, even if it is slow, than waiting for a fast system created in unknown future. Note that base on the Moor’s law, computer speed doubles every 18 months on average. Nevertheless, the results show that the proposed approach does not affect the big O complexity and speed for applying the spatial analyses on objects of higher dimensions.

The manipulations occurred in the environment are the results of processes that change the state of the real world objects. Although the goal of GI science is to study the process of the real world, because of deficiencies of the tools (e.g., modeling and programming environments) in practice, the focus of the current research is on studying the state of the

spatial objects (Hofer and Frank, 2008). A model of the reality is a collection of states that are converted to each other by processes. These conversions can be considered as functions that convert a state to another. To interact with the objects, the imperative programming languages can be used, but direct interaction with the processes and modeling their relations is possible through functional programming languages. The results of implementing the proposed approach of this research in Haskell certify this claim.

8.2 Research contribution

The major contribution of the research is providing a formal approach to implement spatial analyses in different dimensions, which eventually proves the hypothesis of the research, which says “*studying spatial analyses based on their dimension independent characteristics leads to a consistent solution toward implementation of a multi-dimensional GIS*“. We introduced a pure mathematical concept as an efficient tool to model and solve GI problems. More specifically, the major contributions of the research are as follows:

- Providing an exhaustive review on existing solutions to extend the 2D Delaunay triangulation to 3D, dynamic and kinetic points.
- Developing a mathematically provable framework to integrate spatial analyses via their dimension-independent properties, which can be extended to different multi-dimensional spaces (e.g., 3D, temporal, etc.):
 - Providing an abstract view to spatial analyses and formalizing this view using algebraic structures.
 - Providing a hierarchical framework of spatial analyses that eventually ends in a set of primitive operations.
- Constructing a strict connection between different multi-dimensional spaces that can be used to extend the program of an already implemented 2D spatial analysis to higher dimensions (e.g., 3D, moving, etc.) without recoding the whole process.
- Definition of the framework into a mathematical model with executable specifications in the functional programming paradigm. It introduces functional programming as a relevant and efficient environment to study spatial processes and their interactions.

- Identifying the barriers to implement the mathematically provable proposed idea in the programming language Haskell through the selected case study, i.e., Delaunay triangulation.

The contributions of the research has resulted in several articles, papers and reports published in different scientific journals, international conferences and symposiums as well as periodicals, including:

- **Karimipour, F.** and Ledoux, H. (2011). *Dynamic and Kinetic Delaunay Triangulation in 2D and 3D: A Survey*, Submitted to the Journal of Geoinformatica.
- **Karimipour, F.**, Delavar, M.R. and Frank, A.U. (2010). *A Simplex-Based Approach to Implement Dimension Independent Spatial Analyses*, Journal of Computer and Geosciences, **36**: 1223-1134.
- **Karimipour, F.**, Delavar, M.R. and A.U. Frank, A.U. (2010). *n-Dimensional Volume Calculation for Non-Convex Polytops*, 18th edition of the Haskell Communities and Activities Report, May 2010.
- **Karimipour, F.** (2009). *n-Dimensional Convex Decomposition of Polytopes*, 16th edition of the Haskell Communities and Activities Report, May 2009.
- Bulbul, R., **Karimipour, F.** and Frank, A.U. (2009). *A Simplex-based Dimension Independent Approach for Convex Decomposition of Nonconvex Polytopes*, In Proceedings of the GeoComputation 2009 Conference, Sydney, Australia, November 30 - December 2, 2009, pp. Unpaginated.
- **Karimipour, F.**, Delavar, M.R. and Frank, A.U. (2008). *A Mathematical Tool to Extend 2D Spatial Operations to Higher Dimensions*, In: O. Gervasi et al. (Eds.) Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2008), Perugia, Italy, June 30 - July 3, 2008, Lecture Notes in Computer Science (LNCS), Vol. 5072, Springer-Verlag, pp. 153-167.
- **Karimipour, F.**, Frank, A.U. and Delavar, M.R. (2008). *An Operation-Independent Approach to Extend 2D Spatial Operations to 3D and Moving Objects*, In: H. Sammet, C. Shahabi and W.G. Aref (Eds.) Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2008), Irvine, CA, USA, November 5-7, 2008.

- **Karimipour, F.** and Delavar, M.R. (2008). *Extension of Spatial Operations for Multi-dimensional GIS*, In: G. Navratil (Ed.) Proceedings of the Colloquium for Andrew U. Frank's 60th Birthday, Vienna, Austria, June 30 - July 1, 2008, Geoinfo Series, Vol. 39, pp. 117-123.
- **Karimipour, F.** (2008). *Simplex-Based Spatial Operations*, 15th edition of the Haskell Communities and Activities Report, November 2008.
- Rezayan, H., Frank, A.U., **Karimipour, F.** and Delavar, M.R. (2007). *Temporal Topological Relationships of Convex Spaces in Space Syntax Theory*, In: X. Tang, Y. Liu, Z. Jixian and W. Kainz (Eds.), *Advances in Spatio-Temporal Analysis*, Taylor and Francis, pp. 85-100.
- **Karimipour, F.**, Rezayan, H. and Delavar, M.R. (2006). *Formalization of Moving Objects Spatial Analyses Using Algebraic Structures*, In: W. Kuhn and M. Rabaul (Eds.) Proceedings of Extended Abstracts of GIScience 2006, Münster, Germany, September 20-23, 2006, IfGI Prints, Vol. 28, pp. 105-111.
- **Karimipour, F.**, Delavar, M.R. and Frank, A.U. (2005). *Applications of Category Theory for Dynamic GIS Analysis*, In Digital Proceedings of GIS Planet 2005, Estoril, Portugal, May 30- June 2, 2005.
- **Karimipour, F.**, Delavar, M.R., Frank, A.U. and Rezayan, H. (2005). *Point in Polygon Analysis for Moving Objects*, In: C. Gold (Ed.) Proceedings of the 4th Workshop on Dynamic & Multi-dimensional GIS (DMGIS 2005), Pontypridd, Wales, UK, September 5-8, 2005, ISPRS Working Group II/IV, International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, pp. 68-72.

Some of the above publications were included in the achievements of the 3D topography project (3D topography project, 2007) as the contribution of Technical University of Vienna, which was a partner of the project¹.

¹ RGI-011, 3D topography is an EU project that aims to enforce a major break-through in the application of 3D topography and the requirements of such a system such as data acquisition, data model, data storage, data analyses and database management.

8.3 Directions for future work

This research took a step toward deploying abstraction and algebraic structures to solve GI problems. It showed how to use these mathematical concepts to implement dimension-independent spatial analyses. The same manner of this research may be used for extension of other requirements of a multi-dimensional GIS (e.g. data structure, data model, etc.).

The proposed approach of the research was implemented for the Delaunay triangulation as well as some applications that use this analysis in their definitions. Using the proposed approach in implementation of further spatial analyses with more complex structures will evaluate this approach in terms of possibility and efficiency.

One of the major goals of this research was studying spatial processes independent of objects to which they are applied. It results in a better understanding of changes happen in our environment. However, the focus of most of current research is on studying spatial objects and how they are change from a state to another. A main reason is the modeling and programming environments, which are incapable of direct interacting with processes. This research introduced functional programming languages as an efficient environment to fill this gap. Using such functional environments in other GI research may help toward achieving more efficient models and simulations of the interaction of human with the environment in space and time.

APPENDIX 1. THE FUNCTIONAL PROGRAMMING LANGUAGE HASKELL

This Appendix introduces the functional programming language Haskell, which has been employed as the environment to implement the proposed approach of this thesis. The main concepts and syntax are described to an extent necessary to understand the implementations provided in the thesis.

A1.1 The Functional programming language Haskell

The Haskell is a functional programming language used to implement the proposed approach of this thesis. It is named after Haskell B. Curry who was one of the pioneers of the λ -calculus (Michaelson, 1989). Haskell is *purely functional*, *strongly typed*, and uses *lazy evaluation*. A variety of Haskell implementations is available; here we use the Glasgow Haskell Compiler (GHC). This section gives a short introduction to the syntax and functionality of Haskell. A detailed tutorial can be found in (Hudak et al., 2000; Peyton Jones and Hughes, 1999; Thompson, 1999).

A1.2. Functions

Functions in Haskell are defined as a series of *declarations*. As Haskell is typed strict, the order and types of the input parameter(s) and the output parameter of the function must be specified first. This is called *type signature*. The syntax of a type signature is as follow:

$$\text{function name} :: \text{type of input 1} \rightarrow \dots \rightarrow \text{type of input n} \rightarrow \text{type of output} \quad (\text{A1-1})$$

Similar to any functional language, Haskell obeys the outer-to-inner reduction rule in evaluating expressions. This is explicitly shown by removing the parentheses around the variables and the expression is evaluated from left to right:

$$\begin{aligned} y = f(x) &\rightarrow y = f x \\ y = f(x, y) &\rightarrow y = f x y \end{aligned} \quad (\text{A1-2})$$

For example, the function `add` that adds two integers is defined as:

```
add :: Int -> Int -> Int
add x y = x + y
```

Similar to the mathematics, *function composition* is possible in Haskell. It improves the structure of a program and thus its readability. The top-level functions are often specified by composing a number of functions together. Each part is designed and implemented separately – following a top-down approach. The output of one function becomes the input of another function, and so on. Therefore, the order plays an important role. The constraint by which functions can be composed is given by the signature of the function composition operator `(.)`:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

The following example increments its input then multiplies it by 2:

```
f = ((*2) . (+1)) x
```

A1.3 Lambda expressions

Anonymous functions can be made using lambda expressions. For example:

```
\x -> x + 1
```

is a function with one parameter that adds one to its input. In general:

```
\pattern1 pattern2 ... patternn -> expression (n>=1)
```

Lambda expressions are useful in defining in-line functions. For example `map` is defined as:

```
map (\xs -> zip xs [1..]) list
```

A1.4 Data types

Data types classify the variables based on their properties. A major strength of functional programming languages such as Haskell is that they are *typed strict*. It means every object has a particular type and only the operations of that type can be applied on that object. It assures that the program runs correctly and prevents the conceptual deviation in the results (Doets and Jan Eijck, 2004).

The following syntax is used in Haskell to assign a data type to a variable:

```
variable name :: data type (A1-3)
```

The “::” is read as “has type”. Haskell has several *predefined types* such as characters (`Char`), integers (`Int`), floating point numbers (`Float`), Double precision numbers (`Double`), rational numbers (`Ratio`), Booleans (`Bool`), strings (`String`), tuples `((a,b))`, and lists `([a])`.

```
`f' :: Char
4 :: Int
4.7 :: Float
4.73 :: Double
2 % 3 :: Ratio
True, False :: Bool
"gis" :: String
(4, 'f') :: (Int,Char)
[1,2,3,4] :: [Int]
```

As we frequently use the lists and their operations, they are more discussed separately in section A1.5.

User-defined data types are introduced with the keyword `data` and defined by the constructors of the type. For example, the data type `Point2D` in Cartesian space is defined by applying the constructor function `Pt2` to an integer (as the identity of the points) and two floating-point numbers (as the coordinates of the point):

```
data Point2D = Pt2 ID Float Float
```

Commonly used types can be assigned a synonym. In Haskell, it is called *type synonyms* and created with a type declaration. For example, the type `Dimension` behaves as the predefined type `Int`.

```
type Dimension = Int
```

A1.5 Lists

A list is a collection of any number of elements of the same type. For instance, all of the following collections are lists:

```
[1, 2, 7, 5, 1, 4] :: [Int]
['a', 'c', 'a', 'd', 'k'] :: [Char]
[(1, 'c'), (5, 'b'), (9, 'k'), (6, 'e')] :: [(Int, Char)]
[True, True, False, True, False, False] :: [Bool]
[[1, 2, 7], [5, 1], [8, 5, 2, 9], [3], [5, 1]] :: [[Int]]
```

where $[a]$ means a list of values of type a . The order of elements in a list is significant: $[1, 2, 3]$ is different from $[3, 2, 1]$, so we can talk about the first, the second, ... and the last elements of a list. The number of occurrences of an element does also matter: $[3]$ contains one element and $[3, 3]$ contains two, which happen to be the same.

The operator `'::'`, called *list constructor*, builds a list from an element and a list. Thus:

```
[1, 2, 7] = 1:[2, 7] = 1:2:[7] = 1:2:7:[]
```

The example shows that every non-empty list is built from an empty list `[]` by the repeated use of the list constructor `'::'`. This characteristic is used to define most of the functions over list, recursively. Table A1.1 presents a set of manipulating functions defined over lists and their implementations. A complete list of standard manipulating functions over lists and their implementations can be found in (Peyton Jones and Hughes, 1999).

Not that Haskell incorporates *higher-order functions* – functions that use functions as arguments and return functions as a result. The `map` function is an instructive example in

this respect. It takes a function and applies it to all elements in a list, such as incrementing the elements in a list as shown in Table A1.1.

Similar to the mathematics, it is possible to generate lists in Haskell using *list comprehension*. For example:

$\{(x, y) \mid x \in \{1,2,3\}, y \in \{4,5\}\}$	Definition in mathematics	
<code>[(x,y) x <- [1,2,3], y <- [4,5]]</code>	Definition in Haskell	(A1-4)
$\{(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)\}$	Results	

Haskell incorporates *polymorphic types* – types that are universally quantified in some way over all types, also called *parametric polymorphism*. This allows for defining functions applicable to various types. For example, the function `length` presented in table A1.1 to count the number of elements in a list can be applied to a list of integers, characters, etc..

Haskell uses the built-in infinite lists `[n ..]`, `[n, m ..]` so that `[0 ..] = [0, 1, 2, 3, ..]`. Regarding the lazy evaluation rule, an element of the list is evaluated only if its value is needed. One can extract finite portions from an infinite list by applying one of the predefined functions in Haskell such as `head`, `take`, etc.:

```
head [0,1,2 ..] = 1
take 5 [0,1,2 ..] = [1,2,3,4,5]
```

Table A1.1. Some standard manipulating functions over lists

Function and Syntax	Description	Example	Implementation
<code>length x</code>	returns the number of elements in the list <code>x</code>	<code>length [2,3,4] = 3</code> <code>length [] = 0</code>	<code>length [] = 0</code> <code>length (x:xs) = 1 + length xs</code>
<code>x ++ y</code>	concatenates two lists	<code>[2,3,4] ++ [4,5] = [2,3,4,4,5]</code> <code>[2,3,4] ++ [] = [2,3,4]</code>	<code>[] ++ y = y</code> <code>(x:xs) ++ y = x : (xs+y)</code>
<code>concat x</code>	for the list of lists <code>x</code> , puts all elements together in a single list	<code>concat [[1,2], [2,3,4], [3,4,5,6], [7,8]] = [1,2,2,3,4,3,4,5,6,7,8]</code>	<code>concat x = fold (++) [] x</code>
<code>concatMap f x</code>	for the list of lists <code>x</code> , applies the function <code>f</code> to all elements of <code>x</code> and then puts them together in a single list	<code>concatMap sum [[1,2], [3,4,5], [5,6]] = [3,12,11]</code>	<code>concatMap f x = concat.map f x</code>
<code>sum x</code>	calculates the sum of all elements of the list <code>x</code>	<code>sum [1,2,3,4] = 10</code>	<code>sum x = fold (+) 0 x</code>
<code>map f x</code>	applies the function <code>f</code> to every elements of the list <code>x</code>	<code>map (+1) [1,2,3] = [2,3,4]</code>	<code>map f [] = []</code> <code>map f (x:xs) = f x : map f xs</code>
<code>filter c x</code>	returns all elements of the list <code>x</code> that fulfill the condition <code>c</code>	<code>filter (>2) [1,2,3,4] = [3,4]</code> <code>filter (==2) [1,2,3,4] = [2]</code>	<code>filter f, [] = []</code> <code>filter f, (x:xs) = if f x == true then x : filter f xs else filter f xs</code>
<code>fold f a x</code>	combines the elements of the list <code>x</code> with the specified function <code>f</code> and the start value <code>a</code> (e.g., add all elements)	<code>fold (+) 0 [1,2,3,4] = 10</code> <code>fold (*) 1 [1,2,3,4] = 24</code>	<code>fold f a [] = a</code> <code>fold f a (x:xs) = fold f (f a x) xs</code>
<code>x \\<code>y</code></code>	drops elements of the list <code>x</code> that exist in the list <code>y</code> , i.e., <code>x - y</code>	<code>[1,2,3,4] \\<code>[3,4,5]</code> = [1,2]</code>	<code>x \\<code>y = [a a <-x and (not (a <- y))]</code></code>
<code>sort x</code>	sorts the elements of the list <code>x</code>	<code>sort [3,4,5,1,2,3,1,5,6,3] = [1,1,2,3,3,3,4,5,5,6]</code>	<code>sort (x:xs) = sort (filter (<x) xs) ++ filter (==x) xs ++ sort (filter (>x) xs)</code>

A1.6 Pattern matching

Pattern matching is a concept in Haskell to define functions. The left-hand sides of the equation contain patterns, which are matched against actual parameters during the application of the function. The process of pattern matching is sequential. If the match of an equation succeeds, the right-hand side gets evaluated and returned as the result of the function. If the match fails, the next equation is tried, and so on. If all equations fail, the result is an error. As an example for pattern matching we use the function `length`:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

When applying this function, the patterns `[]` and `(x:xs)` are matched against actual parameters, whereby `[]` matches only the empty list and `(x:xs)` matches any list with at least one argument — `x` being the first argument and `xs` the rest of the list. In general, patterns can be literal values, variables, wildcards, tuples, and constructors (Thompson, 1999).

A1.7 Classes and instances

A typical feature of Haskell is another type of polymorphism, called *ad hoc polymorphism* or *overloading*. Overloaded functions can be used for a variety of types – with different definitions being used for different types. Overloading therefore allows for the reuse of existing function names. In Haskell, classes are a mechanism for assigning types to overloaded functions.

A class is a collection of types over which a set of functions are defined. For example, the equality class `Eq` contains a set of types over which the equality operator (`==`) is defined:

```
class Eq a where
  (==) :: a -> a -> Bool
```

One then needs to define the members of the class – i.e., which types are instances of the class – and the actual behavior of the functions on each of these types. In other words, the class specifies the functions and their signature. Instances, however, define the functions applications. For example, following we define a 2D point as an instance of class `Eq`:

```
instance Eq Point where
  (==) (Point2D x1 y1) (Point2D x2 y2) = (x1==x2) && (y1==y2)
  (==) (Point3D x1 y1 z1) (Point3D x2 y2 z2) =
    (x1==x2) && (y1==y2) && (z1==z2)
```

APPENDIX 2. THE HASKELL CODE

This Appendix contains the complete code of the implementations of this thesis in Haskell programming language. Different modules of the program are presented.

Lifting

```
-- *** Definition of the required functors (Listing and Moving functors)

module Lifting where

import Prelude

-----

-- Listing functors
-----

-- Class "Functor" to lift operations with different number of operands
class Lifting f a where
  lift0 :: a -> f a
  lift1 :: (a -> b) -> f a -> f b
  lift2 :: (a -> b -> c) -> f a -> f b -> f c
  lift3 :: (a -> b-> c-> d) -> f a -> f b -> f c -> f d

-----

-- Moving liftings
-----

-- Type for changing (moving) values
type Instant    = Float
type Changing v = Instant -> v
```

```
-- Liftings to lift operations with static values to operations with
-- changing values
instance Lifting ((->) Instant) a where
  lift0 a = \t -> a
  lift1 op a = \t -> op (a t)
  lift2 op a b = \t -> op (a t) (b t)
  lift3 op a b c = \t -> op (a t) (b t) (c t)

-- Convert a list of changing values to a changing list of values
lc2cl :: [Changing a] -> Changing [a]
lc2cl ma = \t -> lift1 (\a -> a t) ma

-- Liftings to lift operations with list(s) of changing values as operand(s)
-- Lift of a function with no operands (constant value)
lift0L a = lc2cl a

-- Lift of a function with one operand
lift1L op a = lift1 op (lc2cl a)

-- Lift of a function with two operands
lift2L op a b = lift2 op (lc2cl a) b -- 1st operand is list
lift2LL op a b = lift2 op (lc2cl a) (lc2cl b) -- both operands are list

-- Lift of a function with three operands
lift3L op a b c = lift3 op (lc2cl a) b c -- 1st operand is list
lift3LL op a b c = lift3 op (lc2cl a) (lc2cl b) c -- 1st and 2nd operands are list
lift3LLL op a b c = lift3 op (lc2cl a) (lc2cl b) (lc2cl c) -- all operands are list
```

Ring

```
-- *** Definition of the class "Ring" contains primitive operations on numbers

module Ring where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)

import Lifting

infixl 6 +, -
infixl 7 *

-- Definition of the class "Ring" for primitive operations on individual values
class Ring q where
  (+), (-), (*) :: q -> q -> q
  neg, sq      :: q -> q
  sum         :: [q] -> q

  sq a    = a * a
  a - b   = a + (neg b)
  sum ls = foldl (+) zero ls

-- Instance of the class "Ring" for Integer values
instance Ring Int where
  neg  = Prelude.negate
  a + b = a Prelude.+ b
  a * b = a Prelude.* b

-- Instance of the class "Ring" for Floating values
instance Ring Float where
  neg  = Prelude.negate
  a + b = a Prelude.+ b
  a * b = a Prelude.* b
```

```
-- Lifting operations of the class "Ring" from individuals to lists
instance (Ring a) => Ring [a] where
  neg = lift1 neg
  (+) = lift2 (+)
  (*) = lift2 (*)

-- Lifting operations of the class "Ring" from static to changing values
instance Ring a => Ring (Changing a) where
  neg = lift1 neg
  (+) = lift2 (+)
  (*) = lift2 (*)
```


Vector

```
-- *** Definition of n-dimensional points with some additional
-- *** operations for 2D and 3D points

module Vector where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)

import Lifting
import Ring
import Samples
import MyList

-- Definition of an n-dimensional point as a list of numbers
type Pt a = [a]

-- Types for static and moving points
type StaticPt a = Pt a
type MovingPt a = Changing (Pt a)

-----
-- Definition of square distance between two n-dimensional points
-- and some operations for 2D and 3D points (x, y, z, xy, xyz)
class Ring c => Points p c where
  sqDist  :: p -> p -> c

-- Instance of the class "Points" for n-dimensional static points
instance Ring a => Points (StaticPt a) a where
  sqDist p1 p2 = sum.sq $ (p1 - p2)

-- Lifting operations of the class "Points" from static to changing values
instance Ring a => Points (MovingPt a) (Changing a) where
  sqDist = lift2 sqDist
```

```

-----
-- Convert a list of coordinates to a list of different elements
-- [[x1, y1, ...], [x2, y2, ...], ...] ==> [[x1, x2, ...], [y1, y2, ...], ...]
coord2List :: [Pt a] -> [[a]]
coord2List a = init.c2l $ a where
  c2l ([]:_) = [[]]
  c2l ps     = concatMap headL ps: c2l (map tail ps)
  headL a    = [head a]

-- Convert a list of different elements to a list of coordinates
-- [[x1, x2, ...], [y1, y2, ...], ...] ==> [[x1, y1, ...], [x2, y2, ...], ...]
list2Coord :: [[a]] -> [Pt a]
list2Coord ([]:_) = []
list2Coord l      = e1 : list2Coord e2
  where
    e1 = map head l
    e2 = map tail l

-- Type for points with Integer, Floating and Rational elements
type PtI = StaticPt Int
type PtF = StaticPt Float

-- Definition of the class "PointTests" contains some tests on points
class PointTests p bool where
  cw      :: [p] -> p -> bool
  ccw    :: [p] -> p -> bool
  inSphere :: [p] -> p -> bool

-- Instance of the class "PointTests" for n-dimensional static points
instance PointTests (StaticPt Float) Bool where
  cw ps p = (zero) <= (det $ map tr ps)
    where
      tr x = (x - p)

  ccw ps p = not.(cw ps) $ p

-- Note: ps must be in cw order
inSphere ps p = (zero) >= (det $ map tr allPts)
  where
    tr x  = x ++ [sum.map sq $ x] ++ [one]
    allPts = p:ps

```

```
-- Lifting operations of the class "PointTests" to n-dimensional moving points
instance PointTests (MovingPt Float) (Changing Bool) where
  ccw      = lift2L ccw
  cw       = lift2L cw
  inSphere = lift2L inSphere
```

Samples

```
-- *** Some sample 2D static, 2D moving, 3D static and 3D moving points
-- *** (Elements of the moving points are continuous functions of time)

module Samples where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)

import Lifting
import Ring
import Point

-- 2D static points
pt21, pt22, pt23, pt24, pt25, pt26, pt27, pt28, pt29, pt210, pt211,
pt212, pt213, pt214, pt215, pt216, pt217, pt218, pt219, pt220 :: StaticPt Float

pt21 = [3, 4]
pt22 = [1, 3]
pt23 = [4, 1]
pt24 = [8, 1]
pt25 = [7, 2]
pt26 = [9, 2]
pt27 = [5, 3]
pt28 = [8, 4]
pt29 = [6, 3]
pt210 = [5, 1]
pt211 = [4, 4]
pt212 = [6, 7]
pt213 = [6, 6]
pt214 = [8, 6]
pt215 = [3, 5]
pt216 = [4, 0]
pt217 = [7, 1]
pt218 = [2, 3]
pt219 = [3, 6]
pt220 = [9, 4]
```

```
pt2s = [pt21, pt22, pt23, pt24, pt25, pt26, pt27, pt28, pt29, pt210,
pt211, pt212, pt213, pt214, pt215, pt216, pt217, pt218, pt219, pt220]
```

```
-- 2D moving points
```

```
mpt21, mpt22, mpt23, mpt24, mpt25, mpt26, mpt27, mpt28, mpt29,
mpt210, mpt211, mpt212, mpt213, mpt214, mpt215, mpt216, mpt217, mpt218,
mpt219, mpt220 :: (MovingPt Float)
```

```
mpt21 t = [(7-5*sin t), (2+5*cos t)]
mpt22 t = [(5-3*sin t), (3-5*cos t)]
mpt23 t = [(4-3*cos t), (1-1*sin t)]
mpt24 t = [(2+7*cos t), (3+3*cos t)]
mpt25 t = [(3+9*sin t), (4+1*cos t)]
mpt26 t = [(6+1*sin t), (3+2*cos t)]
mpt27 t = [(1+1*sin t), (3-1*cos t)]
mpt28 t = [(8+6*cos t), (4-1*sin t)]
mpt29 t = [(9+4*sin t), (1-3*cos t)]
mpt210 t = [(5-3*sin t), (1+4*cos t)]
mpt211 t = [(3+1*sin t), (4-1*cos t)]
mpt212 t = [(6-2*sin t), (7-5*cos t)]
mpt213 t = [(6-5*cos t), (3+1*sin t)]
mpt214 t = [(8+6*cos t), (6-9*cos t)]
mpt215 t = [(3-3*sin t), (5+2*cos t)]
mpt216 t = [(4+2*sin t), (0-2*cos t)]
mpt217 t = [(7+2*sin t), (1-2*cos t)]
mpt218 t = [(9-3*cos t), (1+7*sin t)]
mpt219 t = [(3+5*sin t), (6-4*cos t)]
mpt220 t = [(4-2*sin t), (6+1*cos t)]
```

```
mpt2s = [mpt21, mpt22, mpt23, mpt24, mpt25, mpt26, mpt27, mpt28,
mpt29, mpt210, mpt211, mpt212, mpt213, mpt214, mpt215, mpt216, mpt217,
mpt218, mpt219, mpt220]
```

```
-- 3D static points
```

```
pt31, pt32, pt33, pt34, pt35, pt36, pt37, pt38, pt39, pt310, pt311,
pt312, pt313, pt314, pt315, pt316, pt317, pt318, pt319, pt320 :: StaticPt Float
```

```
pt31 = [1, 2, 1]
pt32 = [6, 2, 1]
pt33 = [4, 2, 5]
```

```
pt34 = [4, 5, 6]
pt35 = [3, 3, 2]
pt36 = [3, 1, 2]
pt37 = [1, 3, 4]
pt38 = [8, 4, 2]
pt39 = [9, 1, 4]
pt310 = [4, 5, 4]
pt311 = [8, 6, 7]
pt312 = [5, 4, 3]
pt313 = [9, 2, 6]
pt314 = [5, 6, 8]
pt315 = [3, 1, 4]
pt316 = [2, 8, 6]
pt317 = [8, 4, 2]
pt318 = [1, 6, 8]
pt319 = [9, 3, 9]
pt320 = [9, 1, 1]

pt3s = [pt31, pt32, pt33, pt34, pt35, pt36, pt37, pt38, pt39, pt310,
pt311, pt312, pt313, pt314, pt315, pt316, pt317, pt318, pt319, pt320]

-- 3D moving points
mpt31, mpt32, mpt33, mpt34, mpt35, mpt36, mpt37, mpt38, mpt39 :: (MovingPt Float)

mpt31 t = [(7-5*sin t), (2+5*cos t), (1+5*sin t)]
mpt32 t = [(5-3*cos t), (3-5*sin t), (3+4*cos t)]
mpt33 t = [(4-3*sin t), (1-1*cos t), (1-2*cos t)]
mpt34 t = [(2+7*cos t), (3+3*sin t), (3+4*sin t)]
mpt35 t = [(3+9*cos t), (4+1*sin t), (2-2*cos t)]
mpt36 t = [(6+1*sin t), (3+2*cos t), (5-3*sin t)]
mpt37 t = [(1+1*cos t), (3-1*sin t), (2+3*sin t)]
mpt38 t = [(8+6*sin t), (4-1*cos t), (4+2*cos t)]
mpt39 t = [(9+4*cos t), (1-3*sin t), (3-2*sin t)]

mpt3s = [mpt31, mpt32, mpt33, mpt34, mpt35, mpt36, mpt37, mpt38, mpt39]
```

n-Simplexes

```

-- *** Definition of n-dimensional points and n-simplexes based on
-- *** list data structure and then spatial operations as list
-- *** manipulating functions

module CnSimplex where

import List (sort, nub, (\\), inits, tails, union)
import Random

----- Data
type for points, n-simplexes and their operations
-----

-- Define a vertex as a list of Floating numbers
type Vertex = [Float]

-- Define a n-simplex as a list of vertexes
type Simplex = [Vertex]

-- Determine the dimension of a point
ptDim :: Vertex -> Int
ptDim = length

-- Determine the dimension of a n-simplex
simpDim :: Simplex -> Int
simpDim = length

-- Test if a list of points is a valid n-simplex
isSimplex :: [Vertex] -> Bool
isSimplex vs = (length vs == 1 + ptDim (head vs)) &&
               (allEq.map ptDim $ vs)
  where
    allEq x = all (==(head x)) x

-- Determine the orientation of a n-simplex "s" using the sign of
-- the determinant of area, volume, ...
getOrn :: Simplex -> Bool
getOrn s = (det $ map (1:) s) > 0

-----
-- Data type for canonical n-simplexes and their primary operations
-----
Canonical representation of a n-simplex
type CnSimplex = (Simplex, Bool)

-- Change the canonical representation of a n-simplex "s" to
-- its primary representation
simp2cnSimp :: Simplex -> CnSimplex
simp2cnSimp s = (sort s, getOrn s)

```

```

-- Change the primary representation of a n-simplex to
-- its canonical representation
cnSimp2simp :: CnSimplex -> Simplex
cnSimp2simp ([v], b) = [v]
cnSimp2simp (vs , b) = if b then vs else swap vs

swap [] = []
swap [v] = [v]
swap (v1:v2:vs) = v2:v1:vs

-- Test if an input is a valid canonical n-simplex
isCnSimplex :: ([Vertex], Bool) -> Bool
isCnSimplex (vs, b) = isSimplex vs

-- Get the vertexes of a canonical n-simplex
vertexes :: CnSimplex -> Simplex
vertexes = fst

-- Get the orientation of a canonical n-simplex
orn :: CnSimplex -> Bool
orn = snd

-- Determine the dimension of a canonical n-simplex
cnSimpDim :: CnSimplex -> Int
cnSimpDim = length.vertexes

-- Change the orientation of a canonical n-simplex
changeOrn :: CnSimplex -> CnSimplex
changeOrn (vs, b) = (vs, not b)

-- Check if two canonical n-simplexes "cs1" and "cs2"
-- have the same vertexes
eqVs :: CnSimplex -> CnSimplex -> Bool
eqVs cs1 cs2 = vertexes cs1 == vertexes cs2

-- Check if two canonical n-simplexes "cs1" and "cs2"
-- have the same orientation
eqOrn :: CnSimplex -> CnSimplex -> Bool
eqOrn cs1 cs2 = orn cs1 == orn cs2

-- Check if two n-simplexes "cs1" and "cs2" are equal
eqSimps :: CnSimplex -> CnSimplex -> Bool
eqSimps cs1 cs2 = (eqOrn cs1 cs2) &&
                  (eqVs cs1 cs2)

-- For a n-simplex "cs", give the faces of dimension "n"
faceN :: CnSimplex -> Int -> [CnSimplex]
faceN cs n = map simp2cnSimp.(flip combine n).vertexes $ cs

-- Give all faces of a n-simplex "cs"
faces :: CnSimplex -> [CnSimplex]
faces cs = concatMap (faceN cs) [1.. n]
  where
    n = cnSimpDim cs

```



```

-- Extract the boundary of a canonical n-simplex
boundary :: CnSimplex -> [CnSimplex]
boundary (vs , b) = zip (removeEach vs) (cycle [b, not b])

-----
Required operations on canonical n-simplexes
-- Note: Hereafter, we mean 'canonical n-simplex'
-- by n-simplex
----- Add
a vertex "v" to a n-simplex "cs"
addVertex :: Vertex -> CnSimplex -> CnSimplex
addVertex v cs = simp2cnSimp.(v:).cnSimp2simp $ cs

-- Extract the bordering (n-1)-simplexes from a list of connected n-
simplexes
border :: [CnSimplex] -> [CnSimplex]
border = foldr op [] . concatMap boundary
  where
    op x [] = [x]
    op x (y:xs) = if eqVs x y then xs else y:op x xs

-- Join two lists of n-simplexe
join :: [CnSimplex] -> [CnSimplex] -> [CnSimplex]
join = (++)

-- Determine the position of an n-dimensional point "pt" respect to
-- an (n-1)-simplex "cs"
ccw, cw :: CnSimplex -> Vertex -> Bool
ccw cs pt = orn.(addVertex pt) $ cs
cw cs pt = not (ccw cs pt)

-- Change the order of the vertexes of a n-simplex "cs" such that it
-- makes a ccw order respect to the vertexes of a given list of
-- n-simplexes "css"
mkCCWsimp :: [CnSimplex] -> CnSimplex -> CnSimplex
mkCCWsimp css cs = if ccw cs v then cs else changeOrn cs
  where
    v = aVertexIn css cs

-- Find a vertex in a list of n-simplexes "css" which is not a
-- vertex of a given n-simplex "cs"
aVertexIn :: [CnSimplex] -> CnSimplex -> Vertex
aVertexIn css cs = head ((nub (concatMap vertexes css)) \\ (vertexes cs))

-- Change the order of the vertexes of a n-simplex "css" such that
-- it makes a ccw order respect to the vertexes of other
-- n-simplexes in another list
mkCCWsims :: [CnSimplex] -> [CnSimplex]
mkCCWsims css = map (mkCCWsimp css) css

-- Determine the (n-1)-simplexes in a list "css" that make cw order
-- respect to a given vertex "v"
cwSims :: [CnSimplex] -> Vertex -> [CnSimplex]
cwSims css v = filter ((flip cw v).(mkCCWsimp css)) css

```

```
-- Determine the vertexes in a list of vertexes "vs" that make cw
-- order respect to a given (n-1)-simplex "cs"
cwVerts :: [Vertex] -> CnSimplex -> [Vertex]
cwVerts vs cs = filter (cw cs) vs
```

MyList

```
-- *** Some new operations on lists

module MyList where

import List (elemIndices, (\\), inits, tails)

-- Intersection of two lists "l1" and "l2"
intersect :: Eq a => [a] -> [a] -> [a]
intersect l1 l2 = [t | t <- l1, elem t l2]

-- Drop the elements of a list "l1" which are in a list "l2" (l1 - l2)
dropElems :: (Eq a) => [a] -> [a] -> [a]
dropElems l1 l2 = [x | x <- l1, notElem x l2]

-- Drop the elements of a list "l" which have a given value "v"
dropElem :: (Eq a) => [a] -> a -> [a]
dropElem l v = dropElems l [v]

-- Drop the Nth element of a list "l"
dropNthElem :: Int -> [a] -> [a]
dropNthElem n l = l1 ++ tail l2
  where
    (l1, l2) = splitAt n l

-- Check if the list "l1" is a subset of the list "l2"
isSubset :: Eq a => [a] -> [a] -> Bool
isSubset l1 l2 = null (l1 \\ l2)

-- Replace a given value "v1" with "v2" in a list "l"
replace :: (Eq a) => a -> a -> [a] -> [a]
replace v1 v2 l = map (rep v1 v2) l
  where
    rep v1 v2 v = if (v == v1) then v2 else v
```

```
-- Split a list "l" from the first appearance of a given value "v"
splitAtElem :: Eq a => a -> [a] -> ([a], [a])
splitAtElem v l = (flip splitAt l).head.(elemIndices v) $ l

-- Group all elements of a list respect a given "eq" function
groupAllBy :: Eq a => (a -> a -> Bool) -> [a] -> [[a]]
groupAllBy _ [] = []
groupAllBy eq (x:xs) = (x:ys) : groupAllBy eq zs
  where
    ys = [t | t <- xs, eq t x]
    zs = xs \\ ys

-- Group all elements of a list with (==) definition for equality
groupAll :: Eq a => [a] -> [[a]]
groupAll = groupAllBy (==)

-- Average of a numerical list "l"
ave :: (Fractional a) => [a] -> a
ave l = (sum l) / (fromIntegral.length $ l)

-- Repeat elements of a list "l"
repeatList :: [a] -> [a]
repeatList l = concat.repeat $ l

-- For two lists "l1" and "l2", find elements of "l2" whose corresponding
-- element in "l1" satisfies the condition "cond"
findByCond :: [a] -> [b] -> (a -> Bool) -> [b]
findByCond l1 l2 cond = map snd.(filter (cond.fst)) $ (zip l1 l2)

-- Determinant calculation
det :: (Num a) => [[a]] -> a
det [] = 1
det m = sum (alternate
              (zipWith (*) (map head m) (map det (removeEach (map tail m))))))
```

```
alternate :: (Num a) => [a] -> [a]
alternate = zipWith id (cycle [id, negate])

removeEach :: [a] -> [[a]]
removeEach xs = zipWith (++) (inits xs) (tail (tails xs))

-----
-- Code for permutation
-----

(^.) = (.) . (.)      -- (^.) uf bf x y = uf (bf x y)
(^^) = (.) . (.) . (.) -- (^^^) uf tf x y z = uf (tf x y z)
(^.) = (.) . flip (.)  -- (^.) f g = (. f) . g

shuffle :: [a] -> [[a]]
shuffle []      = [[]]
shuffle (x:xs) = concatMap (insertAll x) (shuffle xs)
  where
    insertAll :: a -> [a] -> [[a]]
    insertAll e []      = [[e]]
    insertAll e (x:xs) = (e:x:xs) : map (x:) (insertAll e xs)

combine, permute :: [a] -> Int -> [[a]]
combine _ r | r < 0    = error "Zero or more elements should be extracted."
combine _ 0            = [[]]
combine [] _          = []
combine (x:xs) r      = map (x:) (combine xs (r - 1)) ++ combine xs r

permute = concatMap shuffle .^ combine
```

Polyhedron

```
-- *** Definition of a type for polyhedron and some operations on it

module Polyhedron where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)
import List ((\\))

import Lifting
import Ring
import Vector
import MyList

-- Definition of a type for polyhedron
type PH = [PtF]

-- Dimension of a polyhedron
dimPH :: PH -> Int
dimPH = length

-- Check if two polyhedrons are equal (i.e., consists of the same points)
isEqPH :: PH -> PH -> Bool
isEqPH ph1 ph2 = null (ph1 \\ ph2)

-- Make the points of a polyhedron in a cw order
mkCW :: PH -> PH
mkCW pts = [pt] ++ mkCW2 pt pts'
  where
    (pt, pts') = (head pts, tail pts)

-- Make the points of a polyhedron in cw order respect to a given point "pt"
mkCW2 :: PtF -> PH -> PH
mkCW2 pt ph@(p1:p2:ps) = if cw ph pt then ph else (p2:p1:ps)
```

```
-- Make cw sub-polyhedrons of a given polyhedron
subCWphs :: PH -> [PH]
subCWphs ph = map (mkCWph ph) ph
  where
    mkCWph l p      = takeAcwPH (pairPtPHs l p)
    takeAcwPH (p, phs) = head.filter (flip cw p).shuffle $ phs
    pairPtPHs l p      = (p, dropElem l p)

-- Make sub-polyhedrons of a given polyhedron "ph"
subPHs :: PH -> [PH]
subPHs ph = map (dropElem ph) ph

-- Extract the extreme sub-polyhedrons from a list of connected polyhedrons
borderPHs :: [PH] -> [PH]
borderPHs = concat.filter (\x -> length x == 1).(groupAllBy isEqPH).concat.map subPHs

-- Test if a point is in a polyhedron
ptInPH :: PtF -> [PH] -> Bool
ptInPH p ch = null.filter (flip ccw p) $ ch
```

PolyhedronDS

```
-- *** A data structure to store polyhedrones and some operations on them

module PolyhedronDS where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)
import List (elemIndices)

import Lifting
import Ring
import Vector
import MyList
import Polyhedron

-- A data structure that stores the information of a polyhedron
-- PHds = (id, [points], [id of the adjacent polyhedron respect to points])
type PHds a = (Int, [Pt a], [Int])

type PHdsF = PHds Float

-----
Operations on PHds
-----

-- Get the id of a polyhedron
getPHid :: PHds a -> Int
getPHid (idPH, _, _) = idPH

-- Get the points of a polyhedron
getPts :: PHds a -> [Pt a]
getPts (_, pts, _) = pts

-- Get the adjacent polyhedrons of a given polyhedron
getAdjs :: PHds a -> [Int]
getAdjs (_, _, ths) = ths
```



```
-- Get the information of a polyhedron given an id and a PHds
getPHInfo :: Int -> [PHdsF] -> PHdsF
getPHInfo idPH ds = ds!!r
  where
    r = findPHbyID idPH ds

-- Update the information of a polyhedron in a given data structure
updatePHds :: PHdsF -> [PHdsF] -> [PHdsF]
updatePHds dsPH ds = dsNew
  where
    dsNew      = if (rPH == (-2))
                  then ds ++ [dsPH]
                  else ds' ++ [dsPH] ++ (tail ds'')
    (ds', ds'') = splitAt rPH ds
    rPH         = findPHbyID idPH ds
    idPH        = getPHid dsPH

-- Update the information of a list of polyhedrons in a given data structure
updatePHdss :: [PHdsF] -> [PHdsF] -> [PHdsF]
updatePHdss [] ds = ds
updatePHdss (d:dd) ds = updatePHdss dd dsNew
  where
    dsNew = updatePHds d ds

-- Update the adjacency information of a list of polyhedrons
updateAdjacency :: ([PHdsF], [PHdsF]) -> [PHdsF]
updateAdjacency (ds0, ds) = dsNew
  where
    ids = map getPHid ds0
    pts = map getPts ds0
    adjs = map (findAdjsOfPH ds) ds0

    dsNew = zip3 ids pts adjs
```

```

-- Find the adjacent polyhedrons of a given polyhedron
findAdjsofPH :: [PHdsF] -> PHdsF -> [Int]
findAdjsofPH ds ds0 = map (findAdjofPH ds idPH) phs
  where
    idPH = getPHid ds0
    ptsPH = getPts ds0
    phs = concatMap oppPH ptsPH
    oppPH = (\x -> [x]).(dropElem ptsPH)

-- Find the adjacent polyhedrons of a given polyhedron
findAdjofPH :: [PHdsF] -> Int -> PH -> Int
findAdjofPH ds n ph = if null adjPH then (-1) else getPHid.head $ adjPH
  where
    adjPH = filter (isAdjPH ph n) ds
    isAdjPH ps i th = ((/= i).getPHid $ th) && ((isSubset ps).getPts $ th)
    isSubset xs ls = all (== True).map (flip elem ls) $ xs

-- Change the adjacency information of a given polyhedron in a data structure
changeAdjPH :: [PHdsF] -> Int -> Int -> Int -> [PHdsF]
changeAdjPH ds t adjTOld adjTNew = dsNew
  where
    dsNew = if (t == -1) then ds else updatePHds dsTnew ds
    dsTnew = (t, pts, adjsTnew)

    pts = getPts dsT

    adjsTnew = replace adjTOld adjTNew adjsT
    adjsT = getAdjs dsT
    dsT = getPHinfo t ds

-- Find the row number of a polyhedron in a given data structure
findPHbyID :: Int -> [PHdsF] -> Int
findPHbyID idPH ds = if null r then (-2) else head r
  where
    r = (elemIndices idPH).map getPHid $ ds

```

```

-- Find the opposite polyhedron to a point in a given data structure
findOppPH :: Eq a => PHds a -> Pt a -> Int
findOppPH ds p = head (findByCond pts adj (==p))
  where
    pts = getPts ds
    adj = getAdjs ds

-- Find the opposite point to a polyhedron in a given data structure
findOppPt :: PHds a -> Int -> Pt a
findOppPt ds n = head (findByCond adj pts (==n))
  where
    pts = getPts ds
    adj = getAdjs ds

-- Make a single list of all edges of a polyhedron
mkEdgesOfPHs :: [PHdsF] -> PH -> [[PH]]
mkEdgesOfPHs ds ph = map edgesOfPH ds
  where
    edgesOfPH d = combine (thPts d) 2
    thPts d      = filter isIntPt (getPts d)
    isIntPt x    = elem x ph

-- Make a single list of all edges of a list of polyhedrons
mkListofAllEdges :: [PHdsF] -> PH -> [PH]
mkListofAllEdges ds ph = concat.(mkEdgesOfPHs ds) $ ph

-- Make a single list of all sub-polyhedrons of a polyhedron
mkSubPHs :: [PHdsF] -> PH -> [PH]
mkSubPHs ds ph = innerSimps ds
  where
    innerSimps d = filter isInnerSimp (map getPts d)
    isInnerSimp x = all (==True) (map (elem' ph) x)
    elem' a b     = elem b a

-- Make a single list of all sub-polyhedrons of a list of polyhedrons
mkListofAllSubPHs :: [PHdsF] -> PH -> [PH]
mkListofAllSubPHs ds ph = mkSubPHs ds ph

```

Delaunay

```

-- *** Delaunay Triangulation for n-dimensional static and moving points
-- *** (Bowyer/Watson algorithm)

module Delaunay where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)
import List (sort, nub, (\))

import Lifting
import Ring
import Vector
import Samples
import MyList
import Polyhedron
import PolyhedronDS

-- Delaunay polyhedronization for a given list of points and a data
-- structure (initially []). The output is the data structure
delaunay :: [PtF] -> [PHdsF] -> [PHdsF]
delaunay [] ds      = ds
delaunay ps []      = delaunay ps (boundingPH ps)
delaunay (p:ps) ds = delaunay ps dsNew
  where
    container = walk p ds
    dsNew      = if container == (-1) then ds else insert p ds container

-- Find the polyhedron that contains the inserted point
walk :: PtF -> [PHdsF] -> Int
walk p []      = -1
walk p (t:ts) = if ptInPH' p t then getPHid t else walk p ts'
  where
    idCCWph      = head (findByCond (ptRelPH p t) (getAdjs t) (== False))
    rowCCWph      = findPHbyID idCCWph ts
    (ts1, ts2)    = splitAt rowCCWph ts
    ts'           = ts2 ++ ts1

```

```

-- Insert the inserted point in the data structure
insert :: PtF -> [PHdsF] -> Int -> [PHdsF]
insert p ds container = dsNew
  where
    dsNew                = updatePHdss dsAdjAfterUpdateAdj dsHoleAfterUpdateAdj
    dsAdjAfterUpdateAdj = updateAdjacency (dsAdjs, dsHoleAfterUpdateAdj)
    dsHoleAfterUpdateAdj = updatePHdss dsHole ds
    dsHole                = updateAdjacency (dsHole0, dsUpToAdjs)

    (dsHole0, dsUpToAdjs) = fill (ds, deletedIDs, borderPHs, idBorderPHs, p, [])
    (deletedIDs, borderPHs, idBorderPHs) =
      mkHole (ds, p, [container], [], [])

    dsAdjs                = map (flip getPHinfo ds) idBorderPHs

-----
-- Initial bounding polyhedron
-----
-- Create an initial bounding polyhedron that contains all the points
boundingPH :: [PtF] -> [PHdsF]
boundingPH pts = [(1, boundingPts, replicate (dim+1) (-1))]
  where
    boundingPts          = ([p1, p2] ++ ps)
    p1                   = minimum' firstPt : map minimum' restPts
    p2                   = maximum' firstPt : map minimum' restPts
    ps                   = tail.map (mkAboundingPt coordList) $ coordList
    ([firstPt], restPts) = splitAt 1 coordList
    coordList            = coord2List pts
    dim                  = length.head $ pts

-- Make a bounding point
-- (average of the previous elements ++ max of the current element
-- ++ min of the next element)
mkAboundingPt :: [[Float]] -> [Float] -> [Float]
mkAboundingPt l p = map ave l1 ++ [maximum' l2] ++ map minimum' l3
  where
    (l1, l2, l3) = (fst s, head.snd $ s, tail.snd $ s)
    s            = splitAtElem p l

```

```

-- Expand the minimum and maximum respect
minimum' l = minimum l - scale * rangeL l
maximum' l = maximum l + scale * rangeL l
rangeL    l = abs (maximum l - minimum l)
scale     = 10

-----

-- Walk
-----

-- Position of a point respect to all ccw polyhedrons of a polyhedron
ptRelPH :: PtF -> PHdsF -> [Bool]
ptRelPH p ds = map (flip cw p) s
  where
    s = subCWpchs (getPts ds)

-- Test if a point is inside a polyhedron
ptInPH' :: PtF -> PHdsF -> Bool
ptInPH' p ds = and.ptRelPH p $ ds

-----

-- Insertion
-----

-- Delete all the polyhedrons that violate the Delaunay condition
mkHole (_, _, [], deletedIDs, borderPHs) =
    (sortedDeletedIDs, borderPHs, idBorderPHs)
  where
    idBorderPHs      = nub ([t | (t, _) <- borderPHs, t /= (-1)])
    sortedDeletedIDs = sort deletedIDs

mkHole (ds, p, (t:ts), deletedIDs, borderPHs) =
    mkHole (ds, p, tsNew, deletedIDsNew, borderPHsNew)
  where
    dsT = getPHinfo t ds
    pts = getPts dsT

    isInSphere = inSphere (mkCW pts) p

```

```

(tsNew, deletedIDsNew, borderPHsNew) =
    if not isInSphere
    then (ts, deletedIDs, borderPHs)
    else (tsNew2, deletedIDsNew2, borderPHsNew2)

deletedIDsNew2 = deletedIDs ++ [t]

adjs          = getAdjs dsT
idBorderPHs  = dropElems adjs (deletedIDsNew2)
idPHsToCheck = dropElems idBorderPHs ((-1):ts)

tsNew2       = ts ++ idPHsToCheck

borderPHsNew2 = (dropBorder borderPHs t) ++ newBorderPHs
newBorderPHs  = zip idBorderPHs cph

cph          = if all (==( -1)) idBorderPHs
              then combine pts dim
              else map findPH idBorderPHs

findPH t =   if t == (-1)
              then dropElem pts oppPt
              else intersect pts.getPts.flip getPHinfo ds $ t

oppPt      = findOppPt dsT (-1)
dropBorder l n = [(t, phs) | (t, phs) <- l, t /=n]

dim        = (length pts) - 1

-- Fill the hole created in the mkHole process
fill (ds, _, [], idBorderPHs, _, dsHole) =
    (dsHole, dsUpToAdjs)
where
    dsUpToAdjs = dsHole ++ dsAdjs

dsAdjs      = map (flip getPHinfo ds) idBorderPHs

```

```

fill (ds, deletedIDs, (ph:phs), idBorderPHs, p, dsHole) =
    fill (ds, deletedIDsNew, phs, idBorderPHs, p, dsHoleNew)
  where

    (insPH_id, deletedIDsNew) = if null deletedIDs
                                then (lastId+1, [])
                                else (head deletedIDs, tail deletedIDs)

    lastId    = maximum (map getPHid (dsHole ++ ds))
    insPH_pts = mkCW.(p:).snd $ ph
    dsHoleNew = dsHole ++ [(insPH_id, insPH_pts, [])]

-- Delaunay polyhedronization for a given list of points and a data
-- structure (initially []). The output is the edges of the polyhedrons
delaunay_Edges :: [PtF] -> [[PtF]]
delaunay_Edges pts = mkListofAllEdges (delaunay pts []) pts

-- Delaunay polyhedronization for a given list of points and a data
-- structure (initially []). The output is the polyhedrons
delaunay_PHS :: [PtF] -> [PH]
delaunay_PHS pts = mkListofAllSubPHs (delaunay pts []) pts

-- Definition of the class "Delaunay"
class Delaunay p1 p2 where
  delaunayPH :: p1 -> p2

-- Instance of the class "Deluaney" for n-dimensional static points
instance Delaunay [PtF] [PH] where
  delaunayPH = delaunay_Edges

-- Lifting the class "Deluaney" for n-dimensional moving points
instance Delaunay [Changing PtF] (Changing [PH]) where
  delaunayPH = lift1L delaunayPH

```


Voronoi

```

-- *** Create the Voronoi Diagram of a list of points
-- *** using their Delaunay Triangulation

module Voronoi where

import qualified Prelude
import Prelude hiding ((+), (-), (*), sum, map)
import List (nub)

import Lifting
import Ring
import Vector
import Samples
import MyList
import Polyhedron
import PolyhedronDS
import Delaunay

-- Voronoi diagram of a list of points
voronoi :: [PtF] -> [PH]
voronoi pts = map (replaceIdPHByCenter centers).voronoiEdges $ dt
  where
    dt      = delaunay pts []
    centers = findAllCenters dt

-- Create a list of all voronoi edges
voronoiEdges :: [PHdsF] -> [[Int]]
voronoiEdges ds = filter dropOuter.nub.concatMap link $ ds
  where
    dropOuter l = not (any (==( -1)) l)

-- Make a list of all neighbor polyhedrons
link :: PHds a -> [[Int]]
link ds = map (linkOrder idPH) idAdjs
  where
    idPH      = getPHid ds
    idAdjs    = getAdjs ds
    linkOrder p1 p2 = if p1 < p2 then [p1, p2] else [p2, p1]

```

```

-- Find centers of a list of polyhedrons and make a list of [(id, center)]
findAllCenters :: [PHds Float] -> [(Int, PtF)]
findAllCenters = map pairIdCenter
  where
    pairIdCenter d = (getPHid d, center.getPts $ d)

-- Replace id of a polyhedron by its center point
replaceIdPHByCenter :: Eq a => [(a, b)] -> [a] -> [b]
replaceIdPHByCenter centers ids = map findCenter ids
  where
    findCenter t = snd.head $ (filter ((==t).fst) centers)

-- Find the center of a list of points
center :: [PtF] -> PtF
center p = init (map (mkElem p2') p2)
  where
    p2          = coord2PabList p
    p2'         = zip (repeatList [1,(-1)]) p2
    mkElem x e  = (coff x e) * (a x e) / b
    a x e       = det $ map tr1 (list2Coord (dropElem (map snd x) e))
    coff x e    = fst.head $ (filter ((== e).snd) x)
    b           = ((-1)^dim) * (-2) * (det $ map tr2 (list2Coord p2))
    tr1 x       = x ++ [one]
    tr2 x       = (init $ x) ++ [one]
    dim         = length.head $ p

-- Convert a list of coordinates to a list of different elements plus their
-- lifting to a paraboloid
-- [[x1, y1, ...], [x2, y2, ...], ...] ==>
-- [[x1, x2, ...], [y1, y2, ...], ..., [x1^2+y1^2+..., x2^2+y2^2+..., ...]]
coord2PabList :: (Ring a) => [[a]] -> [[a]]
coord2PabList = coord2List.map liftToHyperB
  where
    liftToHyperB x = x ++ [sum.map sq $ x]

-- Definition of the class "Voronoi"
class Voronoi p1 p2
  where
    voronoiDiagram :: p1 -> p2

```

```
-- Instance of the class "Voronoi" for n-dimensional static points
instance Voronoi [PtF] [PH]
  where
    voronoiDiagram = voronoi

-- Lifting the class "Voronoi" to n-dimensional moving points
instance Voronoi [Changing PtF] (Changing [PH])
  where
    voronoiDiagram = lift1L Voronoi
```

BIBLIOGRAPHY

- Albers, G. (1991). *Three-Dimensional Dynamic Voronoi Diagrams*, Ph.D. Thesis, University of Wurzburg, Wurzburg, Germany, (In German).
- Albers, G., Mitchell, J.S.B., Guibas, L.J. and Roos, T. (1998). *Voronoi Diagrams of Moving Points*, *International Journal of Computational Geometry and Applications*, **8**: 365-380.
- Albers, G. and Roos, T. (1992). *Voronoi Diagrams of Moving Points in Higher Dimensional Spaces*, In Proceedings of the 3rd Scandinavian Workshop On Algorithm Theory (SWAT 92), Helsinki, Finland, Lecture Notes in Computer Science (LNCS), Vol. 621, Springer-Verlag, pp. 399-409.
- Alexandroff, P. (1961). *Elementary Concepts of Topology*, Dover Publications.
- Argany, M., Mostafavi, M.A. and Karimipour, F. (2010). *Voronoi-based Approaches for Geosensor Networks Coverage Determination and Optimisation: A Survey*, In Proceedings of the 7th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2010), Quebec, Canada, June 28 - 30, 2010, pp. 115-123.
- Aurenhammer, F. (1991). *Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure*, *ACM Computing Surveys, the Association for Computing Machinery*, **23**(3): 345-405.
- Bajaj, C. and Bouma, W. (1990). *Dynamic Voronoi Diagrams and Delaunay Triangulations*, In Proceedings of the 2nd Annual Canadian Conference on Computational Geometry, Ottawa, Canada, pp. 273-277.
- Berg, M.D., Cheong, O., Van-Krevelend, M. and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications (3rd Edition)*, Springer-Verlag.
- Bird, R. and de More, O. (1997). *Algebra of Programming*, Prentice Hall.
- Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*, Prentice Hall.
- Bittner, T. and Frank, A.U. (1997). *An Introduction to the Application of Formal Theories to GIS*, In: F. Dollinger and J. Strobl (Eds.) Proceedings of Angewandte Geographische Information sverarbeitung IX (AGIT), Salzburg, Austria, pp. 11-22.
- Boissonnat, J.D., Yvinec, M. and Bronniman, H. (1998). *Algorithmic Geometry*, Cambridge University Press.
- Bowyer, A. (1981). *Computing Dirichlet Tessellation*, *The Computer Journal, Oxford*, **24**(2): 162-166.

- Brown, K.Q. (1979). *Voronoi Diagrams from Convex Hulls*, *Information Processing Letters*, **9**(5): 223–228.
- Bulbul, R. (2011). *AHD: Alternate Hierarchical Decomposition - Towards LoD Based Dimension Independent Geometric Modeling*, PhD Thesis, Advisor: A.U. Frank, Department of Geoinformation, Technical University of Vienna, Vienna, Austria.
- Cignoni, P., Montani, C. and Scopigno, R. (1998). *A Fast Divide and Conquer Delaunay Triangulation Algorithm*, *Computer-Aided Design, Elsevier*, **30**(5): 333-341.
- De Fabritiis, G. and Coveney, P.V. (2003). *Dynamical Geometry for Multiscale Dissipative Particle Dynamics*, *Computer Physics Communications*, **153**: 209-226.
- Delaunay, B.N. (1934). *Sur la Sphere Vide (On the Empty Spher)*, *Izvestia Akademia Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7:793–800.
- Devillers, O. (2002). *On Deletion in Delaunay Triangulations*, *International Journal of Computational Geometry and Applications, World Scientific*, **12**(3): 193-205.
- Devillers, O. and Golin, M. (1993). *Dog Bites Postman: Point Location in the Moving Voronoi Diagram and Related Problems*, In Proceedings of the 1st Annual European Symposium on Algorithms (ESA'93), Honnef, Germany, September 30 - October 2, 1993, Lecture Notes in Computer Science (LNCS), Vol. 726, pp. 133-144.
- Devillers, O. and Teillaud, M. (2003). *Perturbations and Vertex Removal in a 3D Delaunay Triangulation*, In Proceedings of the ACM-SIAM symposium on Discrete Algorithms, Baltimore, USA January 12-14, 2003, pp. 313–319.
- Doets, K. and Jan Eijck, v. (2004). *The Haskell Road to Logic, Maths and Programming*, College Publications.
- Dorst, L., Fontijne, D. and Mann, S. (2007). *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*, Morgan Kaufmann.
- Dwyer, R.A. (1991). *Higher-dimensional Voronoi Diagrams in Linear Expected Time*, *Discrete and Computational Geometry*, **6**: 343–367.
- Edelsbrunner, H. (1987). *Algorithms in Combinatorial Geometry*, Springer-Verlag.
- Edelsbrunner, H. and Seidel, R. (1986). *Voronoi Diagrams and Arrangements*, *Discrete & Computational Geometry* **1**: 25–44.
- Edelsbrunner, H. and Shah, N.R. (1992). *Incremental Topological Flipping Works for Regular Triangulations*, In Proceedings of the 8th Annual Computational Geometry, Berlin, Germany, pp. 43-52.
- Egenhofer, M.J. and Mark, D.M. (1995). *Naïve Geography*, Technical Report, National Center for Geographic Information and Analysis.

- Erwing, M., Guting, R.H., Schneider, M. and Vazirgiannis, M. (1999). *Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases*, *Journal of GeoInformatica*, **3**(3): 269-296.
- Field, D.A. (1986). *Implementing Watson's Algorithm in Three Dimensions*, In Proceedings of the 2nd Annual Symposium on Computational Geometry, Yorktown Heights, New York, USA, pp. 246-259.
- Fortune, S. (1987). *A Sweepline Algorithm for Voronoi Diagrams*, *Algorithmica*, **2**: 153-174.
- Fortune, W. (1992). *Numerical Stability of Algorithms for 2D Delaunay Triangulations*, In Proceedings of the 8th Annual Computational Geometry, Berlin, Germany, pp. 83-92.
- Frank, A. (2000). *Geographic Information Science: New Method and Technology*, *Journal of Geographical Systems*, **2**(1): 99-105.
- Frank, A.U. (1997). *Higher Order Functions Necessary for Spatial Theory Development*, In Proceedings of 13th International Auto-Carto Conference, Seattle, USA, 7-10 April, 1997, pp. 11-22.
- Frank, A.U. (1999). *One Step Up the Abstraction Ladder: Combining Algebras – From Functional Pieces to a Whole*, In: C. Freksa and D. Mark (Eds.) Proceedings of the International Conference COSIT'99, Stade, Germany, August 25-29, 1999, Lecture Notes in Computer Science (LNCS), Vol. 1661, Springer-Verlag, pp. 95-107.
- Frank, A.U. (2007). *Practical Geometry - Mathematics for Geographic Information Systems*, Course Manuscript.
- Frank, A.U. and Gruenbacher, A. (2001). *Temporal Data: 2nd Order Concepts Lead to an Algebra for Spatio-Temporal Objects*, In Proceedings of the Workshop on Complex Reasoning on Geographical Data, Cyprus, December 1, 2001.
- Fu, J.-J. and Lee, C.T. (1991). *Voronoi Diagrams of Moving Points in the Plane*, *Journal of Computational Geometry & Applications*, **1**(1): 23-32.
- Gavrilova, M.L. and Rokne, J. (2003). *Updating the Topology of the Dynamic Voronoi Diagram for Spheres in Euclidean d-Dimensional Space*, *Computer Aided Geometric Design*, **20**: 231-242.
- Ghosh, A. and Das, S.K. (2008). *Coverage and Connectivity Issues in Wireless Sensor Networks: A survey*, *Pervasive and Mobile Computing*, **4**: 303–334.
- Gold, C. (1979). *Triangulation based Terrain Modelling - Where Are We Now?*, In: R.T. Aangeenburg (Ed.) Proceedings of Auto-Carto 4, International Symposium on Cartography and Computing, Baltimore, MD, USA, Vol. 2, pp. 104-111.
- Gold, C. (1990). *Spatial Data Structures - the Extension from One to Two Dimensions*, *Mapping and Spatial Modeling for Navigation*, **65**: 11-39.

- Gold, C. (1994). *A Review of Potential Applications of Voronoi Methods in Geomatics*, In Proceedings of Canadian Conference on GIS, Ottawa, ON, Canada, pp. 1647-1656
- Gold, C. (1998). *The Use of the Dynamic Voronoi Data Structure in Autonomous Marine Navigation*, In Proceedings of the 29th International Symposium on Robotics (ISR98), Birmingham, England, pp. 217-220.
- Gold, C., Charters, T.D. and Ramsden, J. (1977). *Automated Contour Mapping using Triangular Element Data Structures and an Interpolant over Each Triangular Domain*, In: J. George (Ed.) Proceedings of Sigraph '77, San Francisco, USA, Computer Graphics, Vol. 11, pp. 170-175.
- Gold, C.M., Remmele, P.R. and Roos, T. (1995). *Voronoi Diagrams of Line Segments Made Easy*, In Proceedings 7th Canadian Conference on Computational Geometry, Quebec City, Canada, pp. 223-228.
- Goodman, G.E. and Orourke, J. (1997). *Handbook of Discrete and Computational Geometry*, CRC Press.
- Guibas, L. (1998). *Kinetic Data Structures: A State of the Art Report*, In Proceedings of the 3rd Workshop on the Algorithmic Foundations of Robotics: The Algorithmic Perspective, Texas, USA, August 1998, Houston, pp. 191-209.
- Guibas, L., Karaveles, M. and Russel, D. (2004). *A Computational Framework for Handling Motion*, In Proceedings of the 6th Workshop on Algorithm Engineering and Experiments, New Orleans, USA, January 10, 2004, pp. 129-141.
- Guibas, L. and Stolfi, J. (1985). *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams*, *ACM Transactions on Graphics*, **4**(2): 74-123.
- Guibas, L.J., Mitchell, J.S. and Roos, T. (1992). *Voronoi Diagrams of Moving Points in the Plane*, In Proceedings of the 17th International Workshop, June 17-19, 1991, pp. 113-125.
- Gunter, C.A. (1993). *Semantics of Programming Languages*, The MIT Press.
- Guting, R.H., Bohlen, M., Erwig, M., Jensen, C., Lorentzos, N., Schneider, M. and Vazirgiannis, M. (2000). *A Foundation for Representing and Querying Moving Objects*, *ACM Transaction on Database Systems*, **25**: 1-42.
- Guting, R.H., Bohlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N., Nardelli, E., Schneider, M. and Viqueira, J.R.R. (2003). *Spatio-temporal Models and Languages: An Approach Based on Constraints*, In: M. Koubarakis and T. Sellis (Eds.), *Spatiotemporal Databases: The Chorochronos Approach*, Lecture Notes in Computer Science (LNCS), Vol. 2520, Springer-Verlag, pp. 117-176.
- Guting, R.H. and Schneider, M. (2005). *Moving Objects Databases*, Elsevier.
- Guttag, J.V. and Horning, J.J. (1978). *The Algebraic Specification of Abstract Data Types*, *Acta Informatica*, **10**: 27-52.

- Hankin, C. (2004). *An Introduction to Lambda Calculi for Computer Scientists*, King's College Publications.
- Hashemi-Beni, L., Mostafavi, M.A. and Gavrilova, M.L. (2007). *Moving Objects Management in a 3D Dynamic Environment*, In Proceedings of the Geocomputation 2007, NUI Maynooth, Ireland, September 3-5, 2007.
- Hatcher, A. (2002). *Algebraic Topology*, Available at: <http://www.math.cornell.edu/~hatcher/> (Accessed: June 2010).
- Heller, M. (1990). *Triangulation Algorithms for Adaptive Terrain Modelling*, In Proceedings of the 4th International Symposium on Spatial Data Handling, Zurich, Switzerland, July 23-27, 1990, pp. 163–174.
- Hofer, B. and Frank, A. (2008). *Towards a Method to Generally Describe Physical Spatial Processes*, In Proceedings of the 13th Symposium on Spatial Data Handling (SDH 2008), Montpellier, France, June 23-25, 2008.
- Hudak, P., Peterson, J. and Fasel, J. (2000). *A Gentle Introduction to Haskell - Version 98*, Available at: <http://www.haskell.org/tutorial/> (Accessed: June 2010).
- Hughes, J. (1989). *Why Functional Programming Matters?*, *The Computer Journal, Special Issue on Lazy Functional Programming*, **32**(2): 98-107.
- Imai, K., Sumino, S. and H, H.I. (1989). *Geometric Fitting of Two Corresponding Sets of Points*, In Proceedings 5th Annual Symposium on Computational Geometry, Saarbrücken, Germany, ACM Press, pp. 266-275.
- Jeuring, J. and Meijer, E. (1995). *Advanced Functional Programming*, In First International Spring School on Advanced Functional Programming Techniques, Bastad, Sweden, May 1995, Lecture Note in Computer Science (LNCS), Vol. 925, Springer-Verlag.
- Joe, B. (1989). *Three-dimensional Triangulations from Local Transformations*, *SIAM Journal on Scientific and Statistical Computing*, **10**(4): 718–741.
- Joe, B. (1991). *Construction of Three-dimensional Delaunay Triangulations using Local Transformations*, *Computer Aided Geometric Design* **8**: 123–142.
- Kallmann, M., Bieri, H. and Thalmann, D. (2003). *Fully Dynamic Constrained Delaunay Triangulations*, In: G. Brunnert, B. Hamann, H.Mueller and L. Linsen (Eds.), *Geometric Modelling for Scientific Visualization*, pp. 241-257.
- Kaltofen, E. and Villard, G. (2004). *On the Complexity of Computing Determinants*, *Computational Complexity*, **13**(3-4): 91-130.
- Kanaganathan, S. and Goldstein, N.B. (1991). *Comparison of Four-point Adding Algorithms for Delaunay-type 3-Dimensional Mesh Generators*, *IEEE Transaction on Magnetics*, **27**(3): 3444-3451.
- Karimipour, F. (2005). *Formalization of Spatial Analyses of Moving Objects Using Algebraic Structures*, M.Sc. Thesis, Advisors: M.R. Delavar and A.U. Frank,

Department of Surveying and Geomatics Engineering, College of Engineering, University of Tehran, Tehran, Iran, (In Persian).

- Karimipour, F. (2009). *n-Dimensional Convex Decomposition of Polytopes*, 16th edition of the Haskell Communities and Activities Report, May 2009, Available at: <http://www.haskell.org/communities/> (Accessed: June 2010).
- Karimipour, F., Delavar, M.R. and A.U. Frank, A.U. (2010a). *n-Dimensional Volume Calculation for Non-Convex Polytopes*, 18th edition of the Haskell Communities and Activities Report, May 2010, Available at: <http://www.haskell.org/communities/> (Accessed: June 2010).
- Karimipour, F., Delavar, M.R. and Frank, A.U. (2005a). *Applications of Category Theory for Dynamic GIS Analysis*, In Digital Proceedings of GIS Planet 2005, Estoril, Portugal, May 30- June 2, 2005.
- Karimipour, F., Delavar, M.R. and Frank, A.U. (2010b). *A Simplex-Based Approach to Implement Dimension Independent Spatial Analyses*, *Journal of Computer and Geosciences*, **36**: 1223-1134.
- Karimipour, F., Delavar, M.R., Frank, A.U. and Rezayan, H. (2005b). *Point in Polygon Analysis for Moving Objects*, In: C. Gold (Ed.) Proceedings of the 4th Workshop on Dynamic & Multi-dimensional GIS (DMGIS 2005), Pontypridd, Wales, UK, September 5-8, 2005, ISPRS Working Group II/IV, International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, pp. 68-72.
- Karimipour, F., Frank, A.U. and Delavar, M.R. (2008). *An Operation-Independent Approach to Extend 2D Spatial Operations to 3D and Moving Objects*, In: H. Sammet, C. Shahabi and W.G. Aref (Eds.) Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2008), Irvine, CA, USA, November 5-7, 2008.
- Karimipour, F., Rezayan, H. and Delavar, M.R. (2006). *Formalization of Moving Objects Spatial Analyses Using Algebraic Structures*, In: W. Kuhn and M. Rabaul (Eds.) Proceedings of Extended Abstracts of GIScience 2006, Münster, Germany, September 20-23, 2006, IfGI Prints, Vol. 28, pp. 105-111.
- Klein, F. (1939). *Elementary Mathematics from an Advanced Standpoint: Geometry*, Dover Publications, Reprinted 2004.
- Knuth, D.E. (1992). *Axioms and Hulls*, Lecture Notes in Computer Science (LNCS), Vol. 606, Springer-Verlag.
- Langran, G. (1989). *Time in Geographic Information Systems*, Ph.D. Thesis, University of Washington, Washington DC, USA.
- Lattuada, R. and Raper, J. (1995). *Applications of 3D Delaunay Triangulation Algorithms in Geoscientific Modeling.*, In Proceedings of the 3rd National Conference on GIS Research UK, Newcastle, UK, pp. 150–153.

- Lawson, C. (1986). *Properties of n-Dimensional Triangulations*, *Computer Aided Geometric Design*, **3**: 231-246.
- Lawson, C.L. (1977). *Software for CI Surface Interpolation*, In: J.R. Rice (Ed.), *Mathematical Software III*, Academic Press, pp. 161–194.
- Lawvere, F.W. and Schanuel, S.H. (2005). *Conceptual Mathematics: A First Introduction to Categories*, Cambridge University Press.
- Ledoux, H. (2006). *Modelling Three-dimensional Fields in Geo-Science with the Voronoi Diagram and its Dual*, Ph.D. Thesis, Advisor: C. Gold, School of Computing, University of Glamorgan, Pontypridd, Wales, UK.
- Ledoux, H. (2007). *Computing the 3D Voronoi Diagram Robustly: An Easy Explanation*, In *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering*, Pontypridd, Wales, UK, July 9-12, 2007, pp. 117-129.
- Ledoux, H. (2008). *The Kinetic 3D Voronoi Diagram: A Tool for Simulating Environmental Processes*, In: P.V. Oosterom, S. Zlatanova, F. Penninga and E. Fendel (Eds.) *Advances in 3D Geo Information Systems*, *Proceedings of the 2nd International Workshop on 3D Geoinformation*, Delft, the Netherlands, December 12-14, 2007, *Lecture Notes in Geoinformation and Cartography (LNCG)*, Springer-Verlag, pp. 361-380.
- Ledoux, H. and Gold, C. (2007). *The 3D Voronoi Diagram: A Tool for the Modelling of Geoscientific Datasets*, In *Proceedings of the GeoCongres 2007*, Quebec, Canada (http://www.gdmc.nl/publications/2007/3D_Voronoi_Diagram_Tool.pdf), octobre 2-5, 2007.
- Ledoux, H., Gold, C. and Baciú, G. (2005). *Flipping to Robustly Delete a Vertex in a Delaunay Tetrahedralization*, In *Proceedings International Conference on Computational Science and its Applications (ICCSA 2005)*, Singapore, *Lecture Notes in Computer Science (LNCS)*, Vol. 3480, Springer-Verlag, pp. 737-747.
- Liskov, B. and Guttag, J. (1988). *Abstraction and Specification in Programming Development*, The MIT Press.
- Liu, Y. and Snoeyink, J. (2005). *A Comparison of Five Implementations of 3D Delaunay Tessellation*, *Journal of Combinatorial and Computational Geometry*, **52**: 439-458.
- Loeckx, J., Ehrlich, H.D. and Wolf, M. (1996). *Specification of Abstract Data Types*, John Wiley.
- MacLane, S. and Birkhoff, G. (1999). *Algebra (3rd Edition)*, AMS Chelsea Publishing.
- Maus, A. (1984). *Delaunay Triangulation and the Convex Hull of n Points in Expected Linear Time*, *Journal of BIT*, **24**: 151-163.
- Michaelson, G. (1989). *An Introduction to Functional Programming through Lambda Calculus*, Addison Wesley.

- Miller, G.L., Steven, E.P. and Walkington, N.J. (2002). *Fully Incremental 3D Delaunay Refinement Mesh Generation*, In Proceedings of the 11th International Meshing Roundtable, Ithaca, New York, USA, September 15-18 2002, pp. 75-86.
- Mostafavi, M.A. (2002). *Development of a Global Dynamic Data Structure*, Ph.D. Thesis, Advisor: C. Gold, University of Laval, Laval, Canada.
- Mostafavi, M.A., C.Gold and Dakowiczb, M. (2003). *Delete and Insert Operations in Voronoi/Delaunay: Methods and Applications*, *Journal of Computers and Geosciences*, **29**: 523-530.
- Mostafavi, M.A. and Gold, C. (2004). *A Global Kinetic Spatial Data Structure for a Marine Simulation*, *International Journal of Geographical Information Science (IJGIS)*, **18**(3): 211-228.
- Mucke, E. (1988). *A Robust Implementation for Three-Dimensional Delaunay Triangulations*, *International Journal of Computational Geometry and Applications*, **8**(2): 255-276.
- Navratil, G. (2006). *Error Propagation for Free?*, In: W. Kuhn and M. Rabaul (Eds.) Proceedings of Extended Abstracts of GIScience 2006, Münster, Germany, September 20-23, 2006, IfGI Prints, Vol. 28, pp. 133-136.
- Navratil, G., Karimipour, F. and Frank, A.U. (2008). *Lifting Imprecise Values*, In: L. Bernard, A. Friis-Christensen and H. Pundt (Eds.) The European Information Society: Taking Geoinformation Science One Step Further, Proceedings of the 11th AGILE International Conference on Geographic Information Science (AGILE 2008), Girona, Spain, May 5-8, 2008, Lecture Notes in Geoinformation and Cartography (LNGC), Springer-Verlag, pp. 79-94.
- Nordstrom, B., Petersson, K. and Smith, J.M. (1990). *Programming in Martin-Lof's Type Theory: An Introduction*, Oxford University Press.
- Okabe, A., Boots, B., Sugihara, K. and Chiu, S.N. (2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams (2nd Edition)*, John Wiley.
- Oosterom, P.V., Zlatanova, S., Penninga, F. and Fendel, E. (Eds.), 2008. *Advances in 3D Geo Information Systems, Proceedings of the 2nd International Workshop on 3D Geoinformation*, Lecture Notes in Geoinformation and Cartography (LNCG). Springer-Verlag, Delft, the Netherlands, December 12-14, 2007.
- Penninga, F. (2008). *A Simplicial Complex-based Solution in a Spatial DBMS*, Ph.D. Thesis, Advisor: P.V. Oosterom, Delft University of Technology, Delft, the Netherlands.
- Penninga, F. and Oosterom, P.V. (2008). *A Simplicial Complex-based DBMS Approach to 3D Topographic Data Modeling*, *International Journal of Geographical Information Science (IJGIS)*, **22**(6-7): 751-779.
- Peuquet, D.J. (1999). *Time in GIS and Geographical Databases*, In: P.A. Longley, M.F. Goodchild, D.J. Maguire and D.W. Rhind (Eds.), Geographical Information

- System, Principals and Technical Issues (2nd Edition), John Wiley & Sons, pp. 91-103.
- Peyton Jones, S.L. (1987). *The Implementation of Functional Programming Languages*, Prentice Hall Int.
- Peyton Jones, S.L. and Hughes, J. (1999). *Haskell 98: A Non-Strict, Purely Functional Language*, Available at: <http://www.haskell.org/onlinereport/> (Accessed: June 2010).
- Pierce, B. (2005). *Advanced Topics in Types and Programming Languages*, The MIT Press.
- Pierce, B.C. (2002). *Types and Programming Languages*, The MIT Press.
- Raper, J. (2000). *Multi-dimensional Geographic Information Science*, Taylor & Francis.
- Raubal, M. (2001). *Agent-Based Simulation of Human Wayfinding: A Perceptual Model for Unfamiliar Buildings*, Ph.D. Thesis, Advisors: A.U. Frank and W. Kuhn, Institute for Geoinformation, Department of Geoinformation and Cartography, Vienna University of Technology, Vienna, Austria.
- Roos, T. (1991). *Dynamic Voronoi Diagrams*, Ph.D. Thesis, University of Wurzburg, Wurzburg, Germany.
- Roos, T. (1993). *Voronoi Diagrams Over Dynamic Scenes*, *Discrete Applied Mathematics*, **43**(3): 243-259.
- Roos, T. and Noltemeier, H. (1991). *Dynamic Voronoi Diagrams in Motion Planning*, In *Computational Geometry: Methods, Algorithms and Applications*, Proceedings of International Workshop on Computational Geometry (CG 91), Bern, Switzerland, March 21-22, 1991, Lecture Notes in Computer Science (LNCS), Vol. 553, Springer-Verlag, pp. 227-236.
- Schaller, G. and Meyer-Hermann, M. (2004). *Kinetic and Dynamic Delaunay Tetrahedralizations in Three Dimensions*, *Journal of Computer Physics Communications*, **162**(1): 9-23.
- Schneider, M. (Ed.), 1997. *Spatial Data Types for Database Systems - Finite Resolution Geometry for Geographic Information Systems*, Lecture Notes in Computer Sciences, 1288. Springer-Verlag, Berlin-Heidelberg, 275 pp.
- Shewchuk, J.R. (2003). *Updating and Constructing Constrained Delaunay and Constrained Regular Triangulations by Flips*, In Proceedings of the 19th Annual Symposium on Computational Geometry, San Diego, USA, ACM Press, pp. 181-190.
- Skiena, S.S. (1998). *The Algorithm Design Manual*, Springer-Verlag.
- Stolfi, J. (1989a). *Primitives for Computational Geometry*, Ph.D. Thesis, Computer Science Department, Stanford University, Palo Alto, USA.
- Stolfi, J. (1989b). *Primitives for Computational Geometry*, Digital Equipment Corporation.

- Tanemura, M., Ogawa, T. and Ogita, N. (1983). *A New Algorithm for Three Dimensional Voronoi Tessellation*, *Journal of Computational Physics*, **51**: 191–207.
- Thompson, S. (1999). *Haskell: The Craft of Functional Programming*, Addison Wesley.
- Vomacka, T. (2008). *Delaunay Triangulation of Moving Points*, In Proceedings of the 12th Central European Seminar on Computer Graphics, Budmerice Castle, Slovakia, April 24-26, 2008, pp. 67–74.
- Wang, G.C. and LaPorta, T. (2004). *Movement-Assisted Sensor Deployment*, In Proceedings of IEEE Infocom, INFOCOM'04, Hong Kong, March 2004, pp. 81-90.
- Watson, D.F. (1981). *Computing the n-Dimensional Delaunay Tessellation with Application to Voronoi Polytops*, *The Computer Journal*, **24**(2): 167-172.
- Yong, X., Sun, M. and Ma, A. (2004). *On the Reconstruction of Three-Dimensional Complex Geological Objects using Delaunay Triangulation*, *Journal of Future Generation Computer Systems*, **20**: 1227–1234.
- 3D Topography Project, (2007): <http://www.rgi-otb.nl/3dtopo/> (Accessed: June 2011).
- CGAL Website: <http://www.cgal.org/> (Accessed: June 2011).