Masterarbeit

im Studiengang Geoinformatik

der Universität Osnabrück

Optimization of LiDAR data line of sight analysis in service-oriented architectures using the OGC Web Processing Service

Eingereicht von

Florian Hillen

Erstprüfer: Dr. Bernhard Höfle, Ruprecht-Karls-Universität Heidelberg

Zweitprüfer: Prof. Dr.-Ing. Manfred Ehlers, Universität Osnabrück

Osnabrück, Mai 2011

Abstract

This work optimizes the calculation of the intervisibility of two or more points based on a triangulated irregular network (TIN). Two different approaches using the R-tree and a topological data structure are developed, implemented and tested to evaluate the TIN as a data model for visibility analysis. Furthermore, the algorithms are compared in calculation time and memory requirements to analyze their efficiency. The topological algorithm is up to 10 times faster on high-resolution TINs and outperforms the R-tree approach on every available TIN resolution. Apart from that the topological data structure is smaller and therefore easier to handle. Finally, the two approaches are integrated into a service-oriented architecture using the OGC Web Processing Service.

Kurzfassung

In dieser Arbeit wird die Berechnung des direkten Sichtkontakts zwischen zwei oder mehreren Punkten basierend auf einem unregelmäßigen Dreiecksnetz (TIN) optimiert. Dazu werden zwei Ansätze entwickelt und implementiert die zum einen den R-Baum und zum anderen eine topologische Datenstruktur verwenden. Zusätzlich wird auf Grundlage dieser Ansätze das TIN als Datenmodel für Sichtbarkeitsanalysen evaluiert. Darüber hinaus werden die Algorithmen hinsichtlich Berechnungszeit und Speicheraufwand verglichen um ihre Effizienz zu überprüfen. Der topologische Algorithmus ist dabei bis zu 10-mal schneller auf hoch aufgelösten TINs und überbietet den R-Baum Ansatz auf jeder verfügbaren TIN-Auflösung. Die topologische Datenstruktur ist zudem kleiner und somit einfacher zu handhaben. Abschließend werden die beiden Ansätze mit Hilfe des OGC Web Processing Service in eine serviceorientierte Architektur eingebunden.

Table of Contents

1 Introduction

In the recent years airborne laser scanning has replaced the traditional approaches to generate elevation data like radar interferometry and image matching. This is mainly due to the higher accuracy with more than 20 scanning points per square meter and the possibility to penetrate many natural objects like trees. In the course of this rising accuracy the data size and the need for fast analyses on this data increases as well. One of the trends these days in the course of mobile computing are location based services (see GSM Association, 2003). For those services and many other applications like navigation systems it is often necessary to know what portion of the client's environment is visible from its current geographical position. It is in the nature of things that this information is needed in the very moment and not after a certain time in which the position of the client already has changed. Therefore, fast and efficient visibility analyses are needed.

		Raster resolution of the digital elevation model [m]			
		1	**2**	**5**	**10**
Maximum calculation distance [m]	**1000**	> 4 min.	27.6 sec	2.8 sec	1.4 sec
	500	33.3 sec	10.4 sec	1.9 sec	1.0 sec
	300	15.2 sec	8.4 sec	1.7 sec	0.9 sec
	200	11.3 sec	7.6 sec	1.7 sec	0.9 sec
	100	9.3 sec	1.7 sec	1.7 sec	0.9 sec

Tab. 1: Calculation time of the visibility analysis in GRASS GIS according to the raster resolution of the digital terrain model and the maximum calculation distance.

Traditional approaches to calculate the visibility on a digital elevation model (DEM) are often satisfactory for short calculation distances and low data resolution but insufficient on highly accurate data and long calculation distances (see Tab. 1). For common airborne laser scanning DEM resolutions of one meter those traditional algorithms do not deliver the result in an appropriate time. Also higher distances are

often necessary for applications in mountainous regions or for significant points of interest. However, high accuracy and high-resolution 3D elevation data is one of the needs of the above mentioned applications and that is why new visibility approaches are needed.

The interoperability has become one of the main properties of software applications nowadays. A new development in geoinformatics according to the general trend of service-oriented architectures is the web-based Spatial Data Infrastructure (SDI) which provides the possibility to link distributed files for an analysis or calculation. It is also possible to distribute the computation effort on different computers. The Web Processing Service (WPS) therefore offers the possibility to integrate a defined set of geospatial functionalities into this infrastructure.

Over the past years many research approaches introduces new algorithms to calculate the visibility on digital surface data. Apart from different algorithm proposals, the approaches mainly differ in the way the elevation data is represented. Beside the traditional raster representation, the triangulated irregular network (TIN) has become the basis of many researches (see Goodchild and Lee, 1989; Lee, 1991; De Floriani and Magillo, 1999; Kidner et al., 2001; Rana and Morley, 2002; De Floriani and Magillo, 2003). This data model provides a better representation of the terrain while it saves less data points. The TIN can also integrate significant topographic features like ridges or peaks (cf. constraint triangulation).

1.1 Aim and structure of this work

The aim of this work is to optimize the calculation of the intervisibility of two or more points based on a triangulated irregular network (TIN) and the integration of the developed methods in a service-oriented architecture using the OGC Web Processing Service. Two different approaches based on the R-tree and a topological data structure are therefore developed, implemented and tested to evaluate the TIN as a data model for visibility analysis. Furthermore, the algorithms should be compared in calculation time and memory requirements to find out which one is more efficient. Finally, the possibility of integrating the two approaches into a service-oriented architecture should be examined.

Chapter 2 introduces the related works and explains the fundamentals of LiDAR and laser scanning, visibility analyses, service-oriented architecture as well as the OGC and the Web Processing Service. The available data is then described in chapter 3. In chapter 4 the methodology of this work is explained, which involves in particular the development of the two approaches of visibility analysis based on the R-tree and a topological data structure as well as the integration into a service-oriented architecture. The results of the data structure creation, the two algorithms and the developed system are discussed in chapter 5. Chapter 6 concludes with the research findings of this work and the future prospects.

2 Related Work

2.1 LiDAR and airborne laser scanning

Laser scanners are active remote sensing systems using a laser (Light Amplification by Stimulated Emission of Radiation) beam to measure the distance between the sensor and the illuminated surface or object and are therefore often described with the acronym LiDAR (Light Detection and Ranging) (Wehr and Lohr, 1999; Gomarasca, 2009). Based on the speed of light in vacuum the time taken by the laser pulse from the emission to the arrival of the reflection is used to calculate this distance (Fig. 1) (Gomarasca, 2009). The laser scanning systems are subdivided in terrestrial laser scanning (TLS), for stationary scanners, and airborne laser scanning (ALS), for scanners installed on any flying object like aircrafts, helicopters or even Unmanned Aerial Vehicles (UAVs).

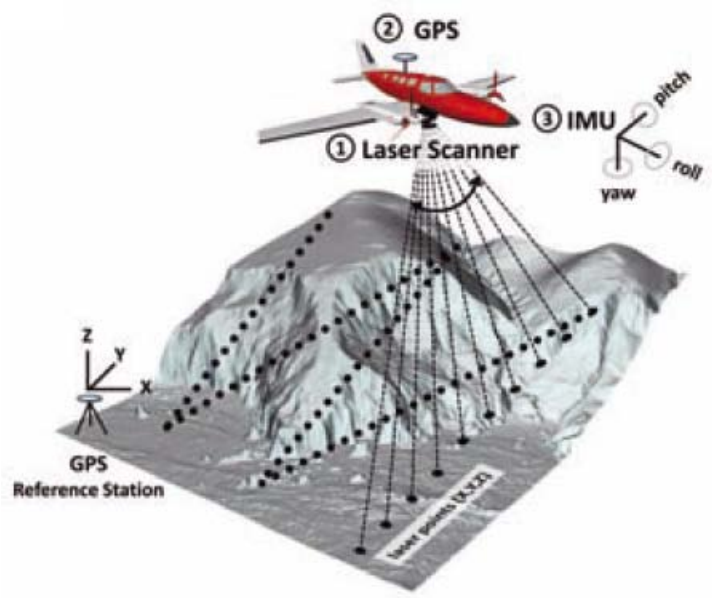


Fig. 1: The simplified principal of airborne laser scanning.

(modified: Höfle and Rutzinger, 2011:4)

Airborne laser scanning is often used to generate digital elevation models (DEMs) whose required accuracy is realized by a position and orientation system (POS),

including a differential GPS (DGPS) and an inertial measurement unit (IMU) (Fig. 2) (Wehr and Lohr, 1999; Ackermann, 1999). One of the major differences between airborne laser scanning and traditional techniques like radar interferometry and stereo-photogrammetry with image matching is the possibility to penetrate objects without a continuous surface like trees of cornfields. If such an object is scanned, the laser might see through small gaps in between the object and may deliver several reflections, as from the top of the object as well as the terrain below it. With those reflections it is possible to create a digital surface model (DSM) as well as a digital terrain model (DTM) from the resulting point cloud. Another advantage of the airborne laser scanning compared to photogrammetric approaches is the active emission of the laser beam which makes the system independent from the illumination of the sun. Thus, problems with shadows from trees or nearby houses do not exist. The accuracy of the extracted DEM mainly depends on the point density, which is affected by the flying speed and height, the pulse rate of the laser and the scan angle (Ackermann, 1999). Today's laser scanners reach point densities of 10-20 points/m² (Vosselman, 2008) which allows the generation of very detailed DEMs (e.g. planimetric resolution below 1 meter).

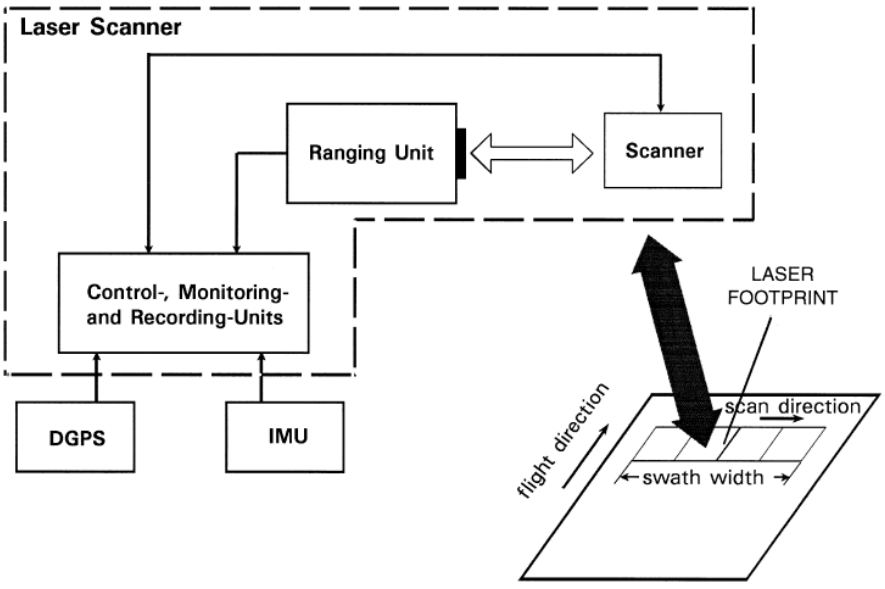


Fig. 2: A typical ALS system. (Wehr and Lohr, 1999:69)

Hence, airborne laser scanning is a highly accurate way to generate DEMs with a highly degree of automation (Ackermann, 1999) and is apart from that a standard in a wide range of applications nowadays (Large and Heritage, 2009) like mapping of roads or electrical lines, generation of 3D city models as well as rapid mapping and damage assessment after natural disasters (Wehr and Lohr, 1999). Airborne laser scanning is also an often used tool in many areas of research. For example the penetration characteristic of the laser was early used to determine tree heights by Næsset (1997) and is nowadays used to estimate the aboveground biomass in forests (Jochem et al., 2011). Another area in which the airborne laser scanning is widespread is glaciology. Early approaches compared the laser scanning with traditional photogrammetric techniques for glacier monitoring and change detection (Baltsavias et al., 2001; Geist et al., 2003; Arnold et al., 2006) while new studies analyze the possibility of surface classification for glacier surfaces to quantify their changes (Höfle et al., 2007). An overview of the use of airborne laser scanning for geomorphological applications and related fields is made by Höfle and Rutzinger (2011).

2.2 Visibility analysis

Along with the accuracy and the resolution of DEMs, the potential of the laser data increases significantly. Many applications nowadays are based on highly efficient analyses (see chapter 1). One of the possible analyses on elevation data is the visibility analysis which is described in the following.

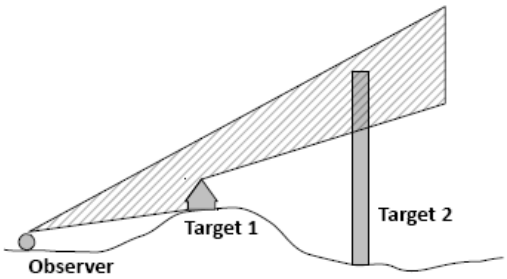


Fig. 3: The field of view of an observer. Target 1 is fully and target 2 is partially visible.

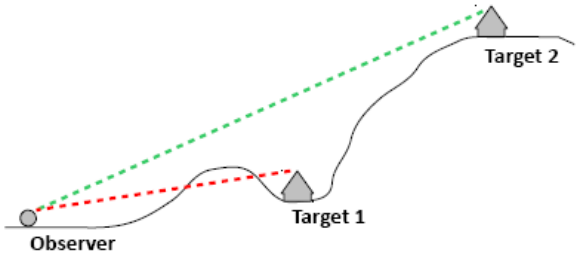


Fig. 4: The direct lines (LOS) between an observer and two targets. Target 2 is visible while target 1 is hidden behind the terrain.

Basically it is important for the visibility calculation to define in which case an object, for example a tower, is visible. If the tower has to be fully visible from the base to the top, the calculation has to be based on the terrain elevation below the target point. If it is enough to see just a part of the tower, the actual elevation of the target point has to be integrated in the visibility analysis (see Kidner et al., 2001). In this work an object is declared as visible if it is partially visible from the given viewpoint like target 2 in figure 3.

The visibility analysis can be classified in three categories according to the dimension of their output (Fig. 5) (De Floriani and Magillo, 1994):

- Point visibility
- Line visibility
- Region visibility

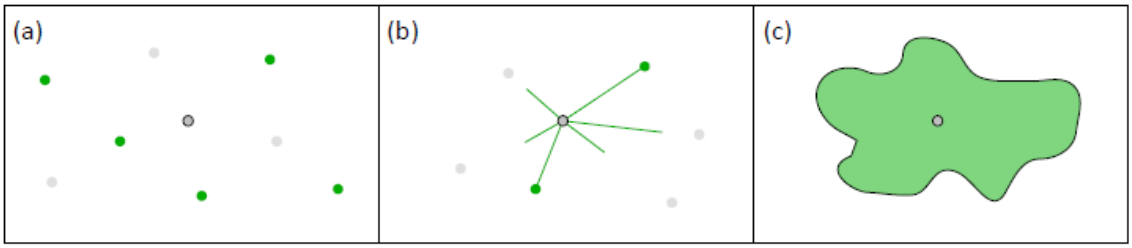


Fig. 5: Comparison of the three categories of visibility analyses. (a) Point visibility; (b) Line visibility; (c) Region visibility.

The point visibility analyses the intervisibility of two or more points and is based on the direct line between an observer point and a target point, named line-of-sight (LOS). If the LOS is intersected by any part of the terrain, we assume that the corresponding target is not visible from the observer (Fig. 4). The output is a set of points that are visible from the given viewpoint. The line visibility also uses the LOS to determine the visibility of a target. In contrast to the point visibility it delivers the actual visible part of the line-of-sight for any point pair. In this case it does not matter if the target is visible from the observer or not. The region visibility is the most complex calculation from the above-mentioned and delivers the visible area around the observer independent from any specific target (De Floriani and Magillo, 1994). In geographical information systems (GIS) this latter case is often called “viewshed analysis”.

Apart from the output dimension, visibility analyses mainly differ in the underlying data models (Fig. 6). The traditional way of presenting elevations is the raster data model, in which the data is stored in a regular grid in the x-y-plane with a specific resolution (Kidner et al., 2000). This makes the raster a simple data model and therefore easier to analyze (Andrade et al. 2011). As one can see in Figure 6a, all points in one cell have the same elevation, which results in a sudden step at the cell boundaries and which is, at low resolution, one of the reasons why raster data is often regarded as less accurate and not capable for terrain modeling (see Lee, 1991). The other traditional representation of elevation data is the Grid. It is based on a regular mesh in which the elevation data is stored in the center points of each cell. In this data model the height at the cell borders can be interpolated from the four nearest elevation points and the sudden steps are avoided. Anyway, at high resolution the raster certainly has a higher accuracy, but also requires more memory space even though this is not the decisive factor anymore these days (Andrade et al. 2011). Due to the straightforward analysis of raster data, there were many studies about intervisibility and viewshed calculation on raster data, in recent years especially in external memory. Franklin et al. (1994) studied the geometric aspects of the visibility problems in the placement of air defense missile batteries and therefore analyzed the three algorithms R3, R2 and Xdraw for calculating the viewshed around an observer and four visibility index algorithms. Cohen-Or and Shaked (1995) detected visible and hidden areas from a given viewpoint by working directly on a Digital Elevation Map rather than a polygonal representation of the surfaces. Their algorithm processed discrete LOS and tests the unit-sized terrain elements along their discrete cross-sections. Wang et al. (2000) proposed an algorithm for computing viewshed on gridded DEMs by using reference planes instead of sightlines and defined a target point as visible if it lies on or above the reference plane build by the viewpoint and a pair of adjacent points. Izraelevitz (2003) introduced an approach for the viewshed calculation which relies on an approximate line-of-sight computation in which previous calculated intervisibility information is reused. Haverkort et al. (2007) described a new application of the technique of distribution sweeping (see Goodrich et al., 1993) to calculate the viewshed in external memory based on the algorithm of Van Kreveld

(1996). Andrade et al. (2011) proposed another algorithm for the viewshed calculation in external memory as well which is adapted to the work on manipulating huge terrains of Franklin and Ray (1994) and Franklin (2002).

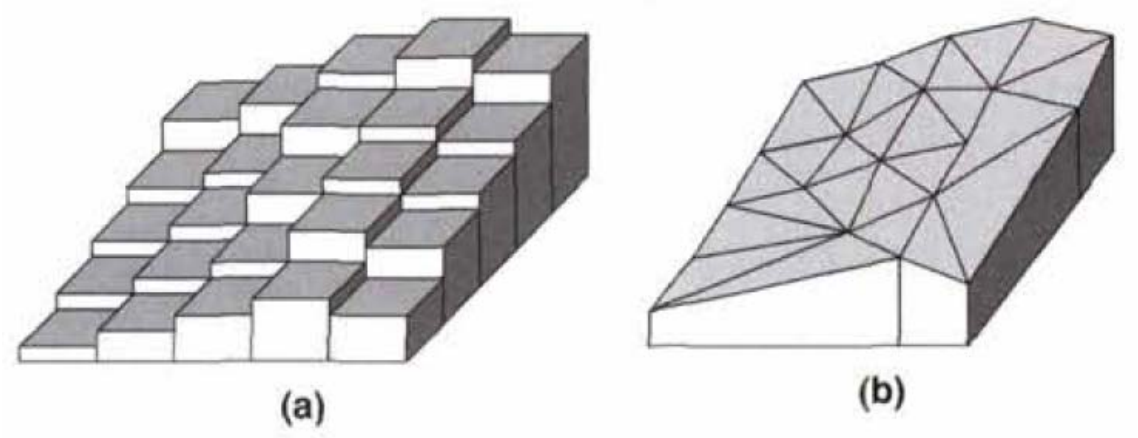


Fig. 6: Raster (a) and TIN (b) data models of the same terrain.
(Maloy and Dean, 2001:1294)

The triangulated irregular network (TIN), presented by Peucker et al. (1978), as another possibility for terrain modeling has become the basis in many research approaches. Goodchild and Lee (1989) demonstrated how to locate the minimum number of viewpoints to see an entire surface as well to locate a fixed number of viewpoints to maximize the visible area and defined the TIN as the most suitable data structure for their work. Lee (1991) exposed five visibility problems and introduced three different heuristic algorithms to demonstrate solutions for these problems based on a TIN. De Floriani and Magillo (1999) and De Floriani and Magillo (2003) gave an overview of algorithms for calculating visibility structures like the viewshed and the horizon as well as visibility queries based on TINs and Regular Square Grids (RSGs). Kidner et al. (2001) introduced an approach to analyze the intervisibility on Multiscale Implicit Triangulated Irregular Networks (Kidner et al., 2000) at multiple resolutions by searching for intersection between the LOS and the terrain and comparing the angles to the viewpoint. Rana and Morley (2002) optimized the visibility calculation by using topographic features like peaks, pits and passes derived from the TIN as observer points using ArcView.

As the name implies the TIN is a network of non-overlapping triangles build out of a set of irregular arranged points (Figure 6b). Hence, it builds a continuous surface without any sudden steps, as previously mentioned for the raster data. Furthermore, the TIN can adapt to characteristical surface points (Peucker and Douglas, 1975), like ridges, channels, peaks or pits and is thus a good way to represent the terrain surface (see Peucker et al., 1978; de Floriani, 1987).

Due to the irregular arranged points, the generation of the TIN could produce many different results, because the edges between the points can be chosen arbitrarily. This triangulation method may result in many large and thin or elongated triangles that are not matching the surface very well. To avoid this, the most common method to create a TIN is the Delaunay triangulation (see de Berg et al., 2008). This method only builds a triangle if no other point is within the circle build by its three points, which results in the creation of smaller but expanded triangles. With those triangles the characteristical points of a terrain surface can be modeled superiorly. Another advantage over the arbitrary triangulation is that the result is always the same, no matter with which points the algorithm begins (de Lange, 2005).

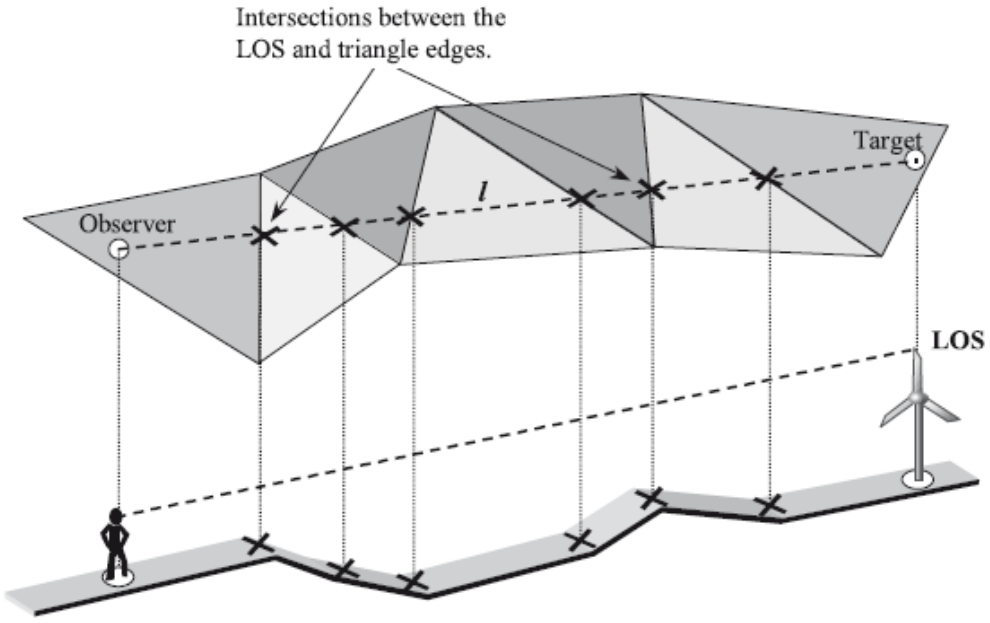


Fig. 7: Intervisibility calculation on a TIN. (Kidner et al., 2001:23)

This work will concentrate on the intervisibility on triangulated irregular networks. Kidner et al. (2001) present an algorithm for this purpose which builds the basis for the developed algorithms in chapter 4. In that algorithm the two triangles enclosing the observer and target first have to be identified. Afterwards the intersection points from the LOS with the triangles edges in the x-y plane has to be calculated (Fig. 7) and their height has to be determined by a linear interpolation. Finally, the angles from the observer to the intersection points are compared with the angle of the LOS. If one angle to an intersection point is higher than the angle of the LOS, the target is not visible.

2.3 Service-oriented architecture

"A Service-Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus." (Krafzig et al., 2004) (Fig. 8). It is important to clarify, that SOA is not a defined standard like XML-RPC (van Engelen et al., 2000) or SOAP (W3C, 2007) but an architecture or a framework for distributed systems (Melzer, 2010).

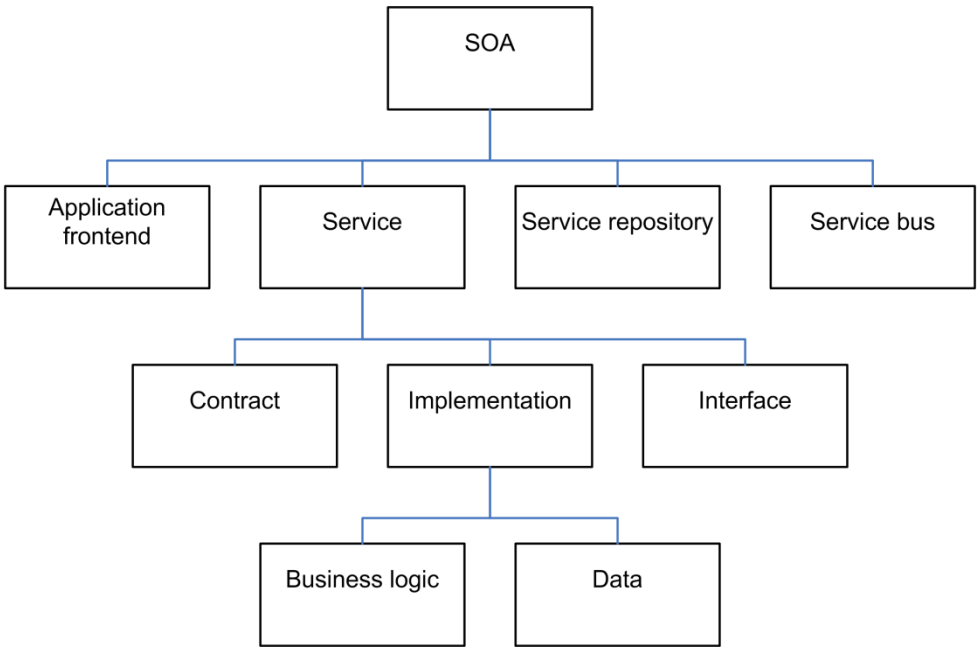


Fig. 8: Overview of the components of a SOA. (Krafzig et al., 2004:57)

The application frontend can be any kind of interface to control or start one or many services but is not necessarily controlled by an end-user. A service delivers the functionality to solve a given job which often reflects a precise workflow in a business. The contract of the service specifies the functionality, explains the usage and shows possible constraints. This contract has to be fulfilled in the implementation which can be considered as the atomic parts of the workflow and which includes the whole logic and data to solve the given job. The interface exposes the functionality to clients or others services that are connected with the service. All services can be stored in a service repository which delivers the needed information about the usage as well as the metadata of a specific service. A repository is not necessarily needed for a service-oriented architecture but it is often very useful. The service bus is responsible for the interoperability and the connection between the services and the frontend (Krafzig et al., 2004).

The main characteristics of a SOA are the loose coupling, which describes the dynamic linkage of services among themselves on demand, the usage of open standards for the interfaces and the principle of reuse (Melzer, 2010). The well-known implementations of the service-oriented architecture are web services which are forced to use existing interfaces and open protocols (Melzer, 2010; Offermann et al., 2006).

2.3.1 The Open Geospatial Consortium

The Open Geospatial Consortium (OGC) is a consortium of 419 partly meaningful members like ESRI, IBM, Google, Microsoft and the European Space Agency (ESA) (Open Geospatial Consortium, 2011a) as well as smaller companies and universities (Open Geospatial Consortium, 2011b). The OGC wants to establish the interoperability of geospatial services and the integration of geospatial content into other business processes (Mitchell et al., 2008). Therefore, many freely available interface standards for geospatial applications have been developed by the OGC (Open Geospatial Consortium, 2011b). The OGC Standards vary from implementation guidelines for web service developers, over the common architecture for simple feature geometry (see Open Geospatial Consortium, 2010a) to encoding definitions such as GML (Open Geospatial Consortium, 2011c).

Apart from the well-known standards like the Web Feature Service (WFS) and the Web Map Service (WMS) (see Open Geospatial Consortium, 2005; Open Geospatial Consortium, 2006), which are standard parts of geo-software for years, the Web Processing Service (WPS) standard is implemented and supported by more and more geospatial applications.

2.3.2 The Web Processing Service (WPS)

The Web Processing Service (WPS) is an OGC standard for publishing geospatial processes with the general goal of offering any kind of GIS functionality to the client across a network. In this context processes are containing pre-defined algorithms which work on spatial referenced data. Publishing is defined by the OGC as creating machine-readable as well as human-readable information. A high number of metadata allows the client to discover the needed service and explains how to use it. The WPS can process vector as well as raster data. This data can be delivered across a network or can be directly read from the server (Open Geospatial Consortium, 2007).

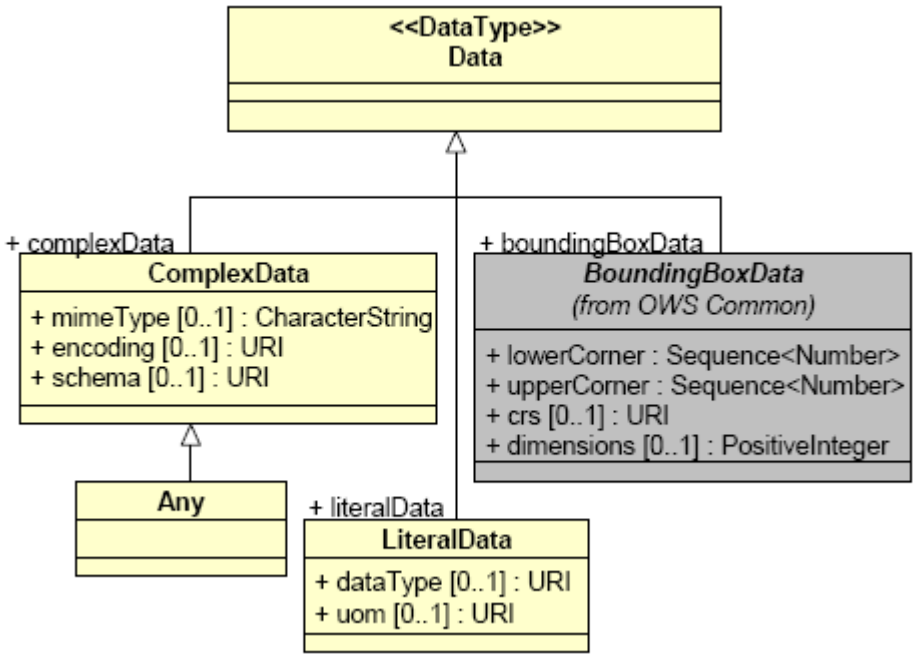


Fig. 9: UML diagram of the three different data types of the WPS.

(Open Geospatial Consortium, 2007:67)

The OGC defines three different types of data as input and output for the WPS (Fig. 9). The “ComplexData” defines any complex data type such as XML or raster data. The

value of this data type can be directly encoded into a client request or made available by a URL. The format of a “ComplexData” can be defined by a MIME-type or a URI to a given encoding or schema. In addition a process can define the maximum number of megabytes of the complex data.

The “LiteralData” represents any data that can be represented in a character string. Therefore, the data type and the unit of measure can be defined. Beyond that a process can specify a set of allowable values as well as a default value for this data type.

The “BoundingBoxData” is a data type which is originally defined in the OGC Web Services (OWS) Common Standard (see Open Geospatial Consortium, 2010b) and can additionally be used as a data type in the WPS. This data represents a bounding box defined by its lower and upper corners, the underlying coordinate reference system and the dimension of it.

The WPS can be requested by the client using HTTP GET with KVP (key value pair) encoding, HTTP POST with XML encoding and the Simple Object Access Protocol (SOAP). The general parameters for every OGC web service request are “service”, which will in the case of the WPS always be set as “WPS”, “request”, which defines the requested operation, and “version”.

The WPS provides three different operations which can be requested by the client:

- GetCapabilites
- DescribeProcess
- Execute

The “GetCapabilities” operation delivers a metadata document that describes all abilities of the service. Additional parameters of this operation are “AcceptVersions” and “Language”, if multilingual services are implemented. An exemplary request of this operation might therefore look like this:

```
http://foo.bar/foo?service=WPS&Request=GetCapabilities&version=1.0.0&
language=de-DE
```

The response of this operation is a XML file which contains metadata about the service, metadata about the provider, a list of the available operations as well as a list of the offered processes and many more information.

The “DescribeProcess” operation delivers detailed information about the offered processes. This includes the required input parameters with their allowed formats and the expected output. The parameter “Identifier” defines one or many processes that should be described:

```
http://foo.bar/foo?Service=WPS&Request=DescribeProcess&Version=1.0.0&
Language=de-DE&Identifier=intersection,union
```

This request delivers an XML file as well. This file contains general metadata about the process as well as metadata about the in- and outputs like the identifier, title, abstract and data type. Furthermore, the two process parameters “storeSupport”, which specifies whether complex data is stored by the WPS, and “statusSupported”, which indicates a quick response of the process status for long calculations, are listed.

The “Execute” operation runs one or more specified processes that are implemented on the WPS. Therefore, the processes have to be declared using the known parameter “Identifier”. All required data inputs have to be set in the parameter “DataInputs”. The response form can be defined using “ResponseDocument”, which returns the result as a part of the WPS response, or using “RawDataOutput”, which delivers the result without a WPS response document:

```
http://foo.bar/foo?request=Execute&service=WPS&version=1.0.0&
language=de-DE&Identifier=Buffer&DataInputs=Object=@xlink:
href=http%3A%2F%2Ffoo.bar%2Ffoo;BufferDistance=10&ResponseDocument=
BufferedPolygon
```

The response of this operation depends on the defined response form. The simplest case would be the “RawDataOutput” in which the result is directly returned to the client. In all other cases an XML document is responded whose structure depends on the following parameters. If “storeExecuteResponse” is true, the result is stored at a web accessible URL which is then included in the XML file as “executeResponseLocation”. In case that the “asReference” attribute is not set, the complex data will also be included in the response document. If “statusSupported” and “storeExecuteResponse” are true, the status of the response document is kept up to date. Therefore, the status element has the five possible choices “processAccepted”, “processStarted”, “processPaused”, “processSucceeded” and “processFailed” (Fig. 10).

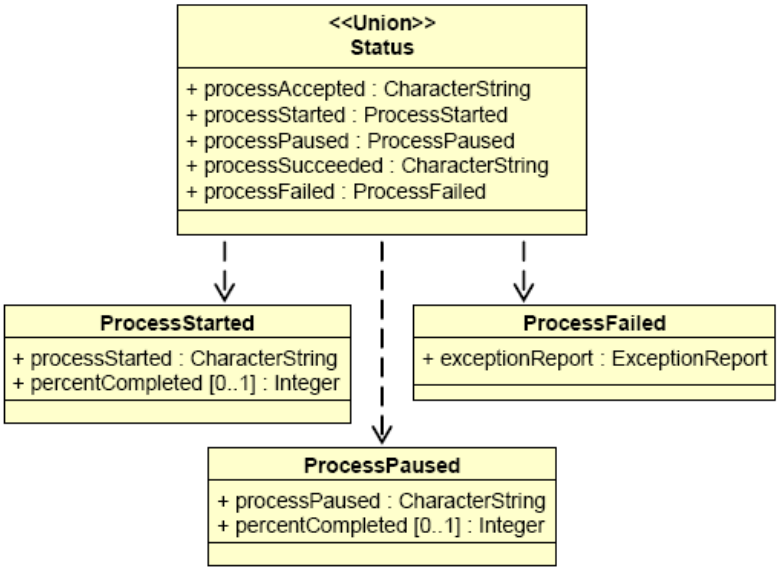


Fig. 10: UML diagram of the five possible choices of the status element of a WPS process. (Open Geospatial Consortium, 2007:68)

The typical workflow of an interaction with the WPS thus starts with the request of the server abilities using the "GetCapabilities" operation. This operation delivers among others a list of the offered processes whose detailed description can be retrieved with the "DescribeProcess" operation. The "Execute" operation finally runs the selected processes and returns the result in a specific output format.

2.3.3 The PyWPS

Since the first OGC specifications for the Web Processing Service in 2005, several implementations of the standard were developed (see Brauner et al., 2009). The company lat/lon GmbH in cooperation with the GIS and Remote Sensing Unit of the Department of Geography at the University of Bonn integrates a Java implemented WPS in its software package "deegree" with the name "deegree 3 processingService" (see deegree, 2011). Another Java implementation of the WPS is presented by the project WPSint (see Tigris, 2011). The ZOO Project defines itself as a open WPS platform and provides the possibility to develop processes in many different programming languages like C, Python, PHP, Java, Perl and Fortran (see ZOO Project, 2011). The 52°North Initiative for Geospatial Open Source Software GmbH presents one more Java based implementation with its "52°North WPS".

The WPS implementation used in this work is a project called “PyWPS” which started in April 2006 and was released in version 1.0.0 in August 2006 (see Čepický and Becchi, 2007). The PyWPS is implemented in Python and was originally developed to build a connection between the UMN MapServer and GRASS GIS. Later the functionality was extended to run without GRASS GIS in the background but with any other system or Python library. Apart from that, the PyWPS still supports the connection to GRASS GIS and therefore provides its whole functionality.

Due to its easy extensibility the PyWPS was used in some recent research approaches where it is for instance used in combination with Google Earth over the GridJet protocol and demonstrate its good efficiency (Wang et al., 2009). The current stable version of the PyWPS is 3.1.0 and the version 3.2 is in the development.

The processes of the PyWPS are written in Python as well and are represented by a class called “Process” with one mandatory method “execute”. This method is called when the process is executed by the client and contains the actual calculation logic.

In the initializing method of the class, the general metadata of the process as well as the input and output data has to be defined:

```
class Process(WPSProcess):
  def __init__(self):
     WPSProcess.__init__(self,
              identifier = "visTopo",
              title = "Analyze the visibility on the topological
                       data structure",
              abstract = """Analyze the visibility on the topological
                       data structure""",
              version = "1.0",
              storeSupported = True,
              statusSupported = True)

     self.observer_x = self.addLiteralInput(identifier = "observer_x",
              title = "Observer x-coordinate",
              type = type(0.0))
     […]
     self.visible = self.addLiteralOutput(identifier = "visible",
              title = "Visibility of the target")
```

```
        def execute(self):
            […]
```

In this sample process one can see the identifier, title, abstract and version of the process as well as the two WPS parameter “storeSupported” and “statusSupported”. One floating-point number is defined as a “LiteralData” input parameter for the process which in this case represents the X-coordinate of the observer point. Also defined in the previous example is a “LiteralData” output parameter which represents the visibility of the target and is part of the WPS response to the client.

3 Study area and data sets

3.1 Study area

The study area of this work is located in the city of Osnabrück in northern Germany. In Figure 11 one can see the area in which laser scanning data as a digital surface model (DSM) is available. The area is 5200 x 3600 meters (approx. 19 km²) large. The terrain is rather plane except for the some minor elevations like the Westerberg in the northwest and the Schinkelberg in the north. Also obvious in Figure 11 is the predominant urban structure of the study area with only a few green regions in the northwest, northeast and southwest apart from the urban green spaces.

Fig. 11: Aerial image of the study area (red) in Osnabrück. (Google Earth)

3.2 Available data structure

The digital surface model as can be seen in Figure 12 is available in the AAIGrid (Arc/Info ASCII Grid) format and is referenced to the DHDN Gauß-Krüger zone 3 which is based on the Bessel ellipsoid from 1841. It has 5269 columns and 3593 rows with the cell size 1 meter. The elevation varies between the minimum of 12.64 meters up to the maximum of 174.29 meters.

Fig. 12: Digital surface model (DSM) of Osnabrück (high elevations: bright; low elevations: dark).

3.3 TIN creation

For the following work, the raster elevation data has to be converted into a triangulated irregular network based on the center points of each grid cell. The raster to TIN conversion was performed with the 3D Analyst extension for the software ArcMap 9.3 by ESRI (see ESRI, 2010a). In this process some center points of the raster are used to create a candidate TIN which first of all covers the whole raster without representing the surface very well. Afterwards the TIN is incrementally improved by

adding new points to reach the predefined z-tolerance (the maximum difference in the z units) between the raster and the TIN (Fig. 13) (ESRI, 2010b).

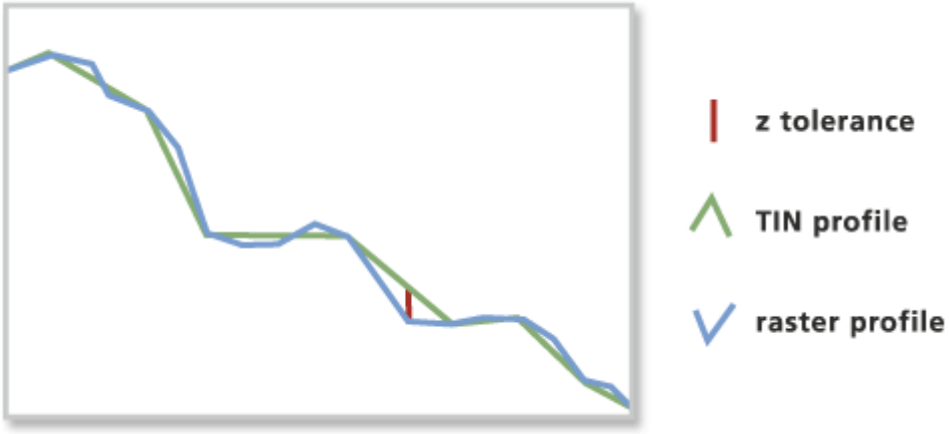


Fig. 13: Illustration of the general idea of the raster to TIN conversion. (ESRI, 2010b)

If the z-tolerance is very small or the raster data has a high roughness the points of the TIN strongly increases (ESRI, 2010b). To test the calculation speed of the upcoming methods on different data accuracies, the TIN is generated several times with varying z-tolerances and are saved as polygons in the ESRI-Shape format. Thus the below listed TINs are available in the following work and are mainly referred by its name from now on:

Name	Tolerance [m]	Triangles	Disk space [MB]
TIN15	15	165303 (2.89%)	35 (2.90%)
TIN10	10	531717 (9.31%)	112 (9.30%)
TIN8	8	836606 (14.65%)	176 (14.61%)
TIN2	2	5709027 (100.00%)	1204 (100.00%)

Tab. 2: Available TIN data sets (the percentages refer to the largest value in the respective column).

Figure 14 shows a part of the DSM of the city center of Osnabrück. Visible in this subset are the St. Katharinen church (red box), the castle of Osnabrück (blue box) and some common multi-level buildings (green boxes). As one can see the church spire is

the highest point in this region with 103 meters. The roof of the castle is with up to 25 meters just a bit higher than the other buildings around it with about 20 meters.

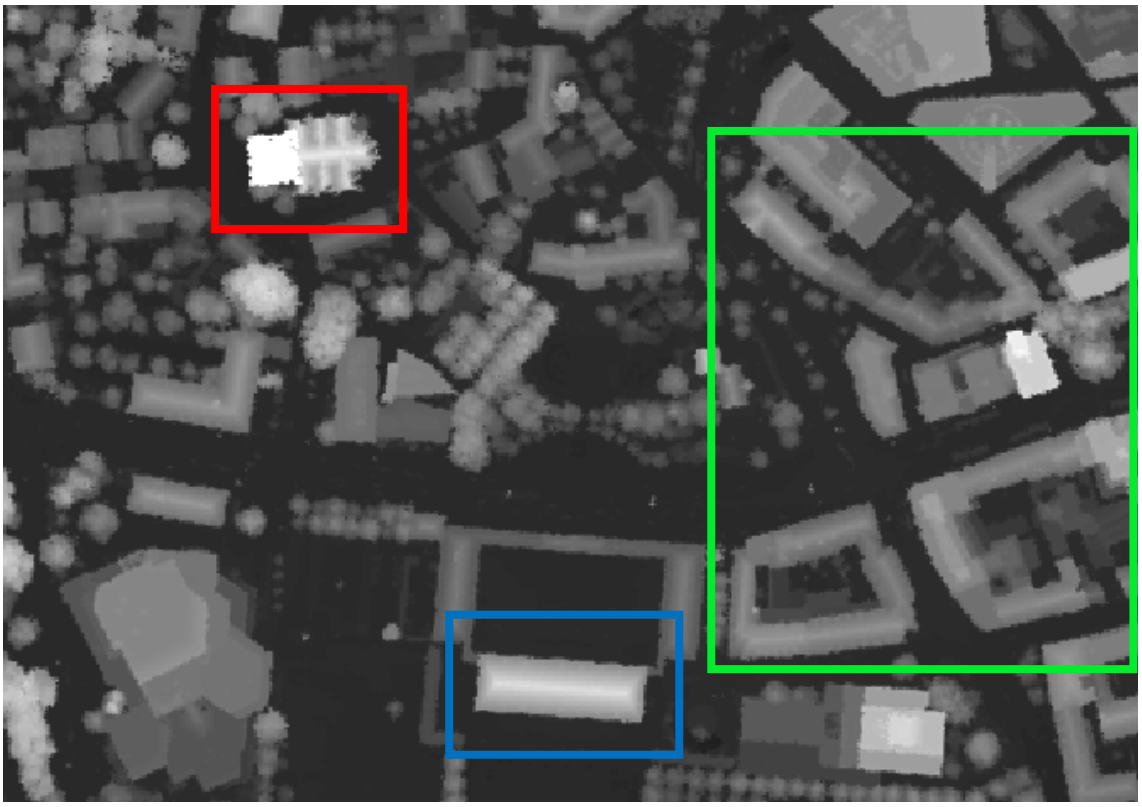


Fig. 14: DSM of a part of the city center of Osnabrück with the St. Katharinen church (red), the castle of Osnabrück (blue) and some common multi-level buildings (green).

The corresponding TIN subsets of the same part of the city center are shown in figure 15. In the top left picture one can see the TIN2. Almost every detail of the digital surface model is still available like for example the nave of St. Katharinen. Every building in the east can be optically separated from each other and also the structure of trees (around the church) is identifiable. The top right image in figure 15 shows the TIN8. The both characteristical buildings, the church and the castle, are still clearly recognizable. The buildings in the east are now merged to larger triangles and can hardly be differentiated. In the bottom left image of figure 15 the TIN10 is illustrated. The church and the castle are still identifiable but their shape is very simplified. The eastern buildings are highly aggregated and cannot be separated correctly. The image on the bottom right of figure 15 shows the TIN15. Exclusively the church can be recognized correctly. The castle is highly simplified and the buildings in the east are

merged to some few large elevations. It is obvious that the TIN15 data set can just be used as a test data set for performance analysis and that the results on it are not convenient. TIN8 and TIN10 are questionable due to their quality of representing the given structures but approximations on these data sets might be reasonable. TIN2 represents the part of the city well and should deliver correct and realistic visibility results.

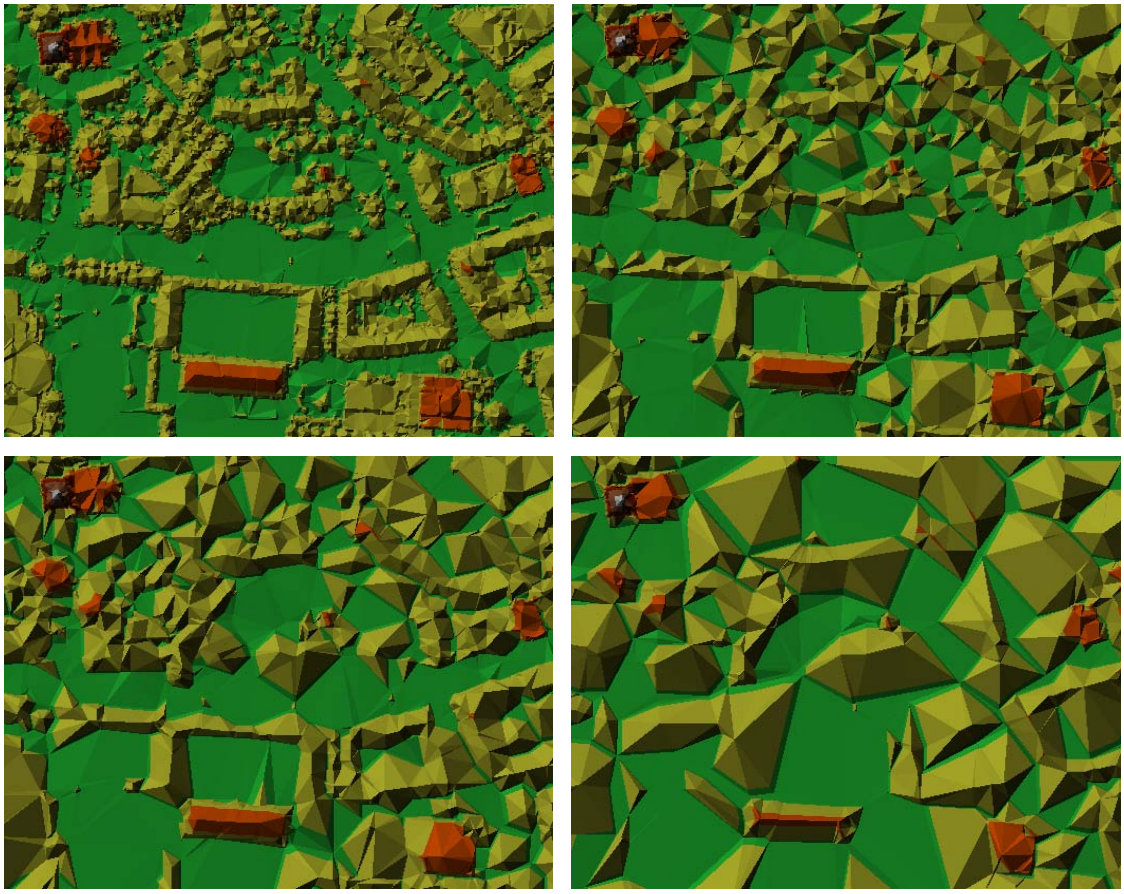


Fig. 15: The four available TINs of a part of the city center of Osnabrück. Top left: TIN2; Top right: TIN8; Bottom left: TIN10; Bottom right: TIN15.
(size of subfigures: approx. 340 x 280 meters)

The quality of the TIN in general has to be adapted to the question of the planned work and study area. In rural or mountainous regions the z-tolerance of the TIN can be chosen higher as small elevations are often not decisive in the visibility calculations and the distances between the observer and the target are often much higher. In urban regions the z-tolerance has to be lower as the distance between the observer

and the target are lower and also objects with a height about 2 meters can constrain the visibility. In this latter case it might be useful to integrate 3D building models or floor plans to define in which areas the tolerance can be chosen higher (open areas) or lower (built-up area).

4 Methodology

The methodology and the ideas of the developed visibility algorithms as well as the fundamentals of both algorithms and the programming logic are described in this chapter. Furthermore, a SOA-conform system design which integrates the two approaches is introduced and explained in detail.

4.1 Visibility analysis using the R-tree

In this chapter a straightforward approach is presented to analyze the intervisibility of two points. The idea is to incrementally parse the line-of-sight from the observer to the target and to detect and analyze every intersection with the TIN. For this, the R-tree, a specific data structure, is used.

4.1.1 Fundamentals of the R-tree

The R-tree is a hierarchical data structure for spatial searching based on the height balanced B+ tree and was originally presented by Antonin Guttman in 1984. It dynamically organizes d-dimensional geometric objects by storing their minimal bounding rectangles (MBR). All inner nodes of the R-tree are therefore containing the MBRs of their children. According to the B+ tree (see Kemper and Eickler, 2006), the R-tree saves the data objects by pointers in its leaves (Guttmann, 1984; Manolopoulos et al., 2005). Figure 16 shows a set of nested MBRs (a) as well as the corresponding R-tree structure (b). In the MBR R8 one can see the actual spatial data object.

One of the most frequent queries on an R-tree is the window or intersection query, which retrieves all MBRs that are intersecting a given search rectangle (Arge et al., 2004). Another usual query is the nearest rectangle query, which makes sense if no intersection of the search rectangle is expected.

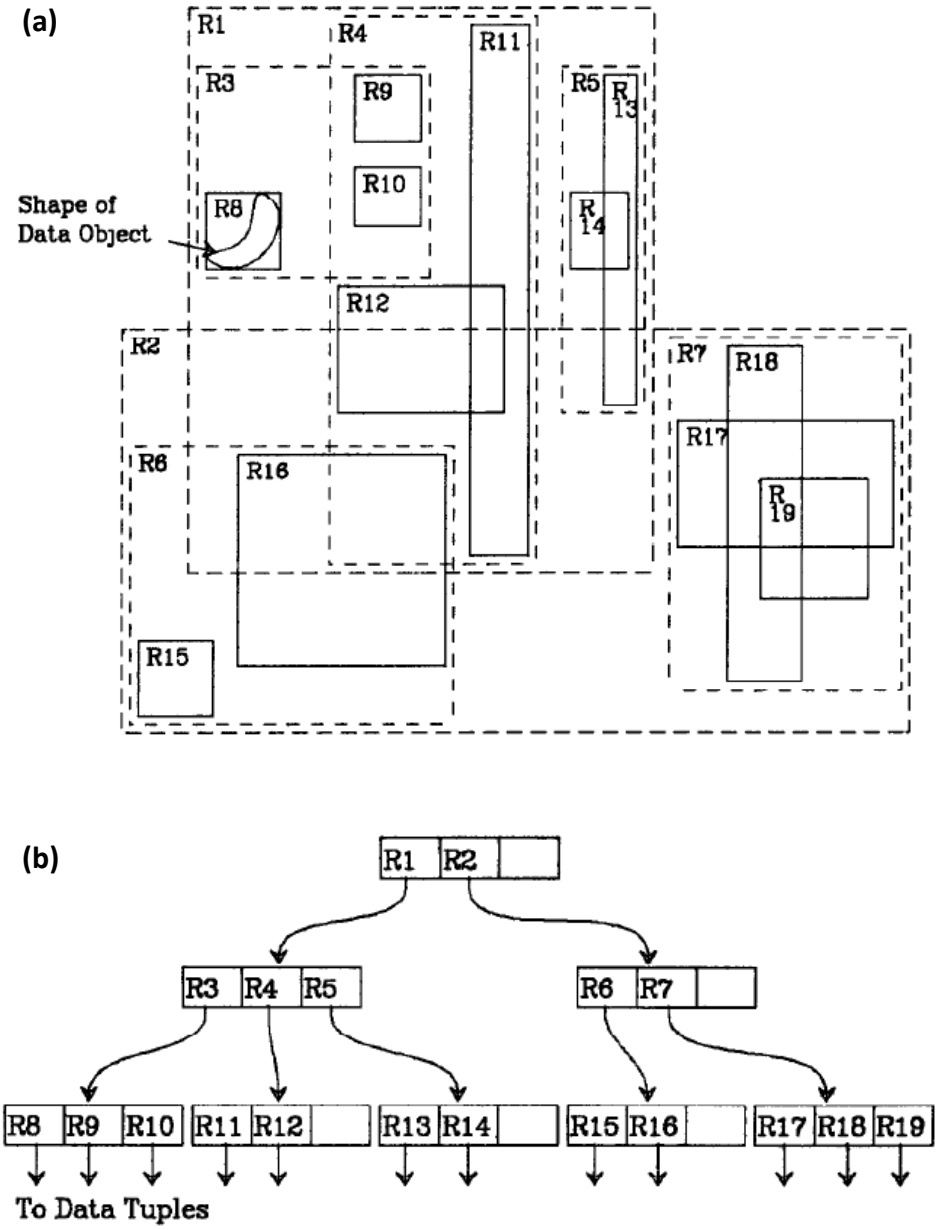


Fig. 16: (a) An example of nested minimally bounding rectangles (solid and dashed lines). R8 shows a sample data. (b) The corresponding R-tree data structure. (Guttman, 1984:49)

4.1.2 Algorithm

The general idea of the algorithm is to parse the line-of-sight with a given step size and overlapping factor and to determine the intersected triangle edges using the R-tree (Figure 17). Hence, the TIN has to be converted into an R-tree data structure. For this purpose an already existing Python module for the R-tree (see Python, 2011a) as well as the Python bindings for the Geospatial Data Abstraction Library (GDAL) and OGR (see Python, 2011b) are used. The latter module offers the possibility to directly read ESRI Shape files and to create own geometries according to the OGC simple features (see Open Geospatial Consortium, 2010a).

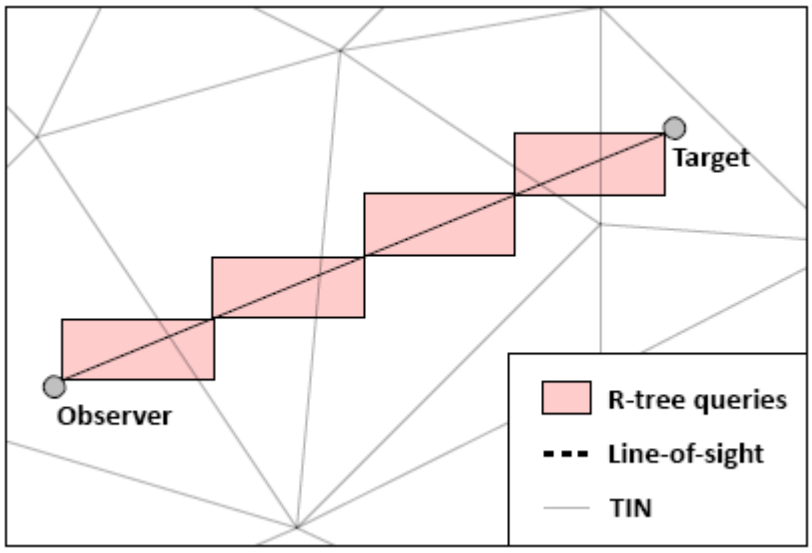


Fig. 17: General idea of the visibility analysis using the R-tree

First of all the TIN has to be imported using the classes of the OGR module. As the R-tree has to find single edges of a triangle, each triangle has to be split up. The easiest way to do this is to split the WKT (well-known text) string and rebuild the single edges of the triangle with OGR. Finally, the R-tree has to be build by adding these edges with its corresponding minimal bounding rectangle.

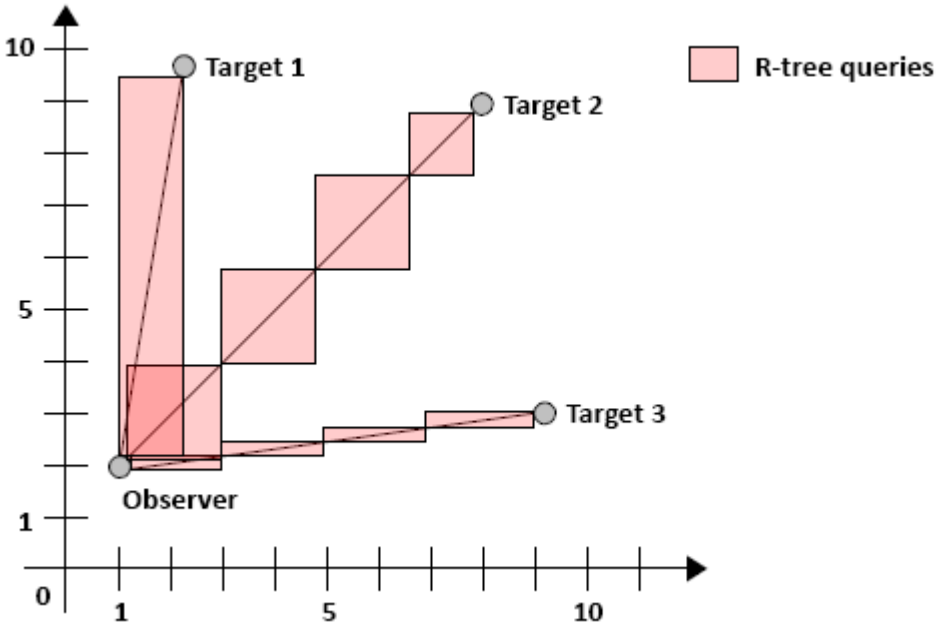


Fig. 18: Different lines-of-sight with different slopes.

Before the visibility can then be analyzed, the algorithm has to check some fundamental parameters of the given line-of-sight. As one can in figure 18 the slope is very important for the parsing direction. If the algorithm would always parse the LOS along the x-axis, lines with higher slopes will create in the worst case only one large window query (see R-tree query of target 1 in figure 18). In this case all triangles edges within the window query have to be checked for intersection. The actual pros of the algorithm, which is the abort after an early intersection with the LOS is found and the preselection of possible intersected edges, do not exist and this should be avoided. Thus the parsing direction has to switch if the slope is greater than 1 which then results in many small window queries, as intended.

After determining the parsing direction, the algorithm has to check whether the coordinate (x or y) has to be incremented or decremented based on the position of the observer. Let us assume a local coordinate system between the observer and an arbitrary target point with the observer point as the origin. In this coordinate system the two introduced parameters are representing four imaginary zones (Fig. 19). The algorithm has to handle these zones individually to parse any possible LOS correctly.

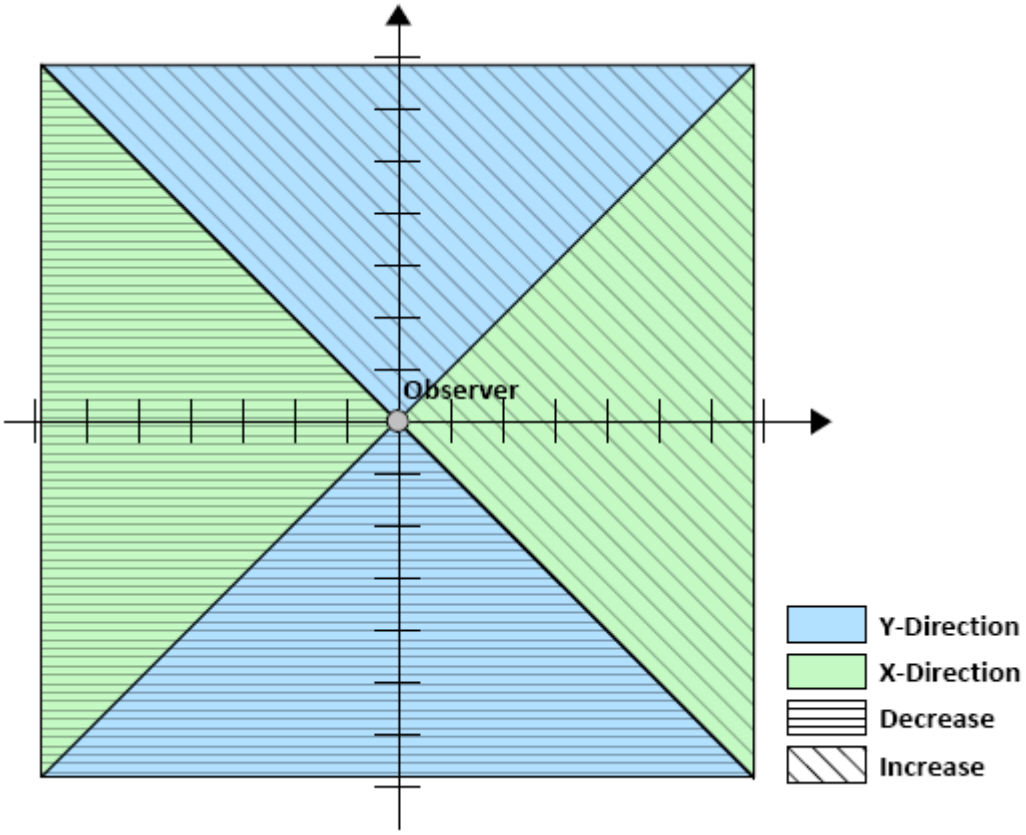


Fig. 19: The blue and green zones show the different parsing directions according to the slope of conceived lines-of-sight. The different line directions symbolize the zones in which the direction increases or decreases.

Another problem that the algorithm has to deal with is the possibility that the TIN intersects the LOS in exactly the same point in which one query window ends and the next query window starts (Fig. 20a). In this improbable case the intersection might not be recognized. This should be avoided by including an overlapping factor, which defines the regression on the LOS, for the query windows (Fig. 20b). In this work an overlapping factor of 0.5 is used for all calculations which mean that the new window query begins a half step behind the actual new starting point on the LOS. This results in an overlapping for the window queries of 25%.

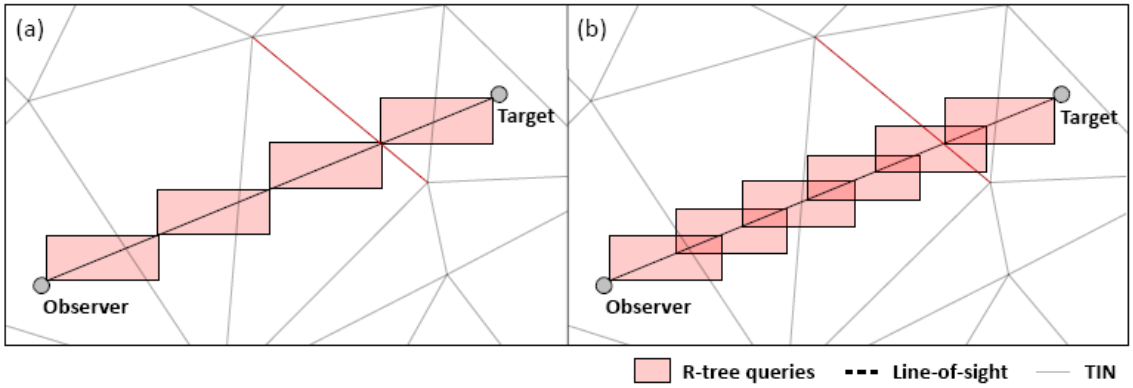


Fig. 20: Possibly undetected intersection (red line) with the TIN (a), avoided by overlapping query windows (b).

The whole logic of the algorithm is shown in figure 21. First the observer is defined as being the start point for the parsing and the slope of the LOS is calculated. The slope then defines if the LOS is parsed along the x- or the y-axis. Afterwards the actual parsing process begins by deciding if the x- or y-coordinate has to be increased or decreased. Now the query window can be calculated and the possibly intersecting edges can be determined by querying the R-tree. All possible edges are then checked for an intersection with the LOS. If an intersection point is found, its height is calculated by a linear interpolation between the edge points. Finally, the two available slopes have to be compared. If the slope of the LOS is less than the slope from the observer to the intersection point, the target cannot be visible and the parsing process should be aborted.

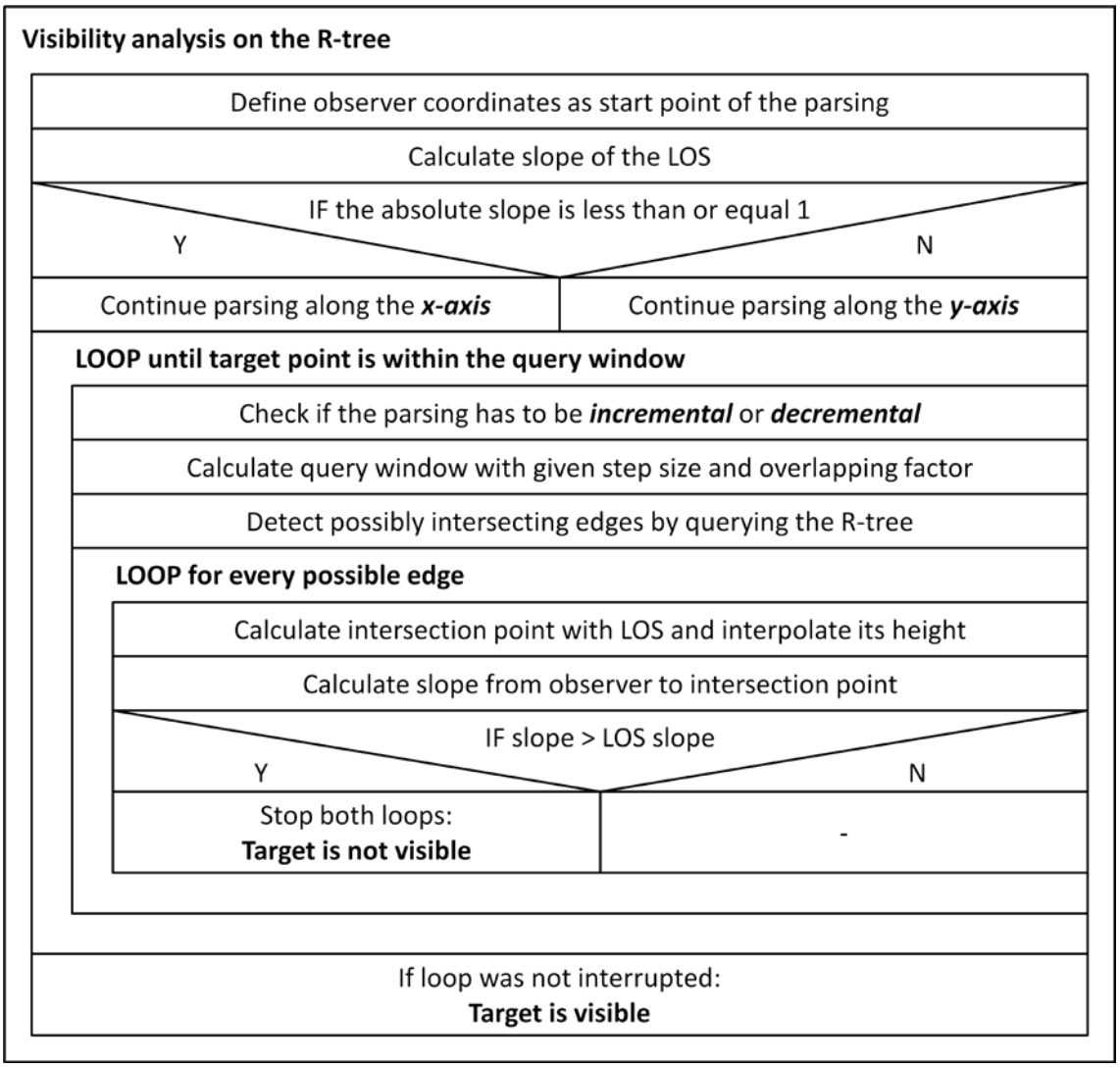


Fig. 21: The logic of the visibility algorithm.

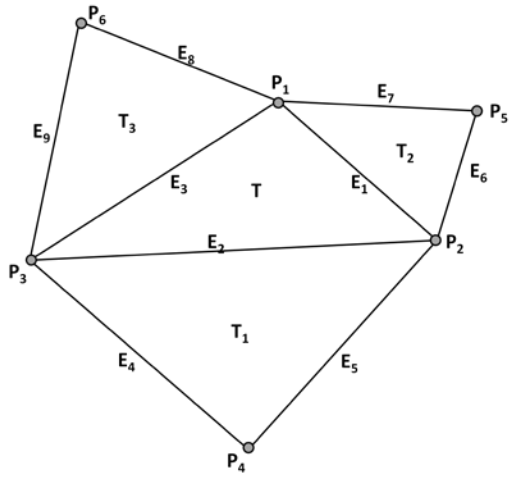
4.2 Visibility analysis using a topological data structure

In this chapter an advanced approach is presented to analyze the intervisibility of two points. The idea is to organize all triangles in a modified topological data structure, which makes the parsing of the LOS unnecessary. Starting at the triangle surrounding the observer point, the algorithm can navigate through the triangles independently as the neighboring triangles are directly stored in the data structure.

4.2.1 Fundamentals of the topological data structure

Topology relating to spatial data typically describes the spatial relationships between neighboring objects. Accordingly a topological data structure is representing these

objects including their relationships. The three primitives points, edges and areas (in this case triangles) are used to describe 2-dimensional data (Yeung and Hall, 2007; Thalheim and Libkin, 1998). Figure 22 shows four triangles with their points and edges as well as the corresponding data structure. It is obvious that no primitive is stored redundant. The points are represented by their x- and y-coordinates whereas the edges only save their start- and endpoint. The triangles are build by three edges and in some approaches are additionally saving their neighbors. With this information the objects and their relationship to each other can be derived.



Point	Coordinates
P1	X, Y
P2	X, Y
P3	X, Y
...	...

Edge	Points
E1	P1, P2
E2	P2, P3
E3	P3, P1
...	...

Triangle	Edges	Neighbors
T	E1, E2, E3	T1, T2, T3
T1	E2, E4, E5	T
...	...	...

Fig. 22: Four triangles with their points and edges and the corresponding topological data structure.

In this approach a quite similar data structure is used. Regarding the high number of triangles of the available TINs, the data structure is designed in a simpler way (Fig. 23). For each triangle its three points with their x-, y- and z-coordinates are saved from which the edges can be derived. Also the corresponding neighbors for each of these edges are saved separately.

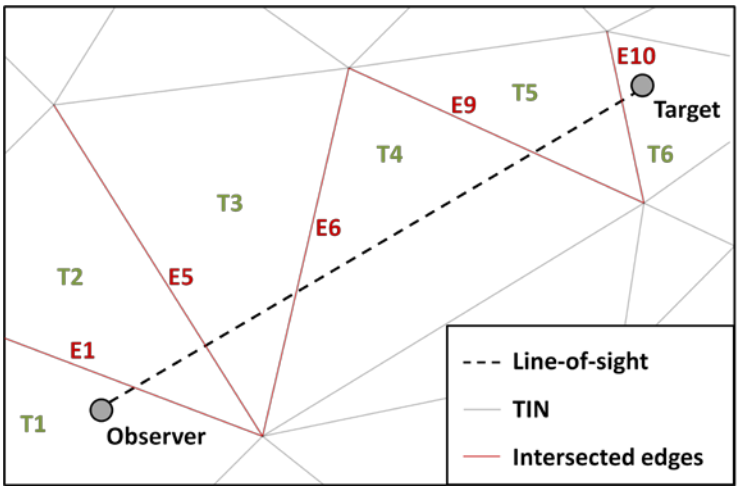
Triangle	Points
1	$(P11_x, P11_y, P11_z)$, $(P12_x, P12_y, P12_z)$, $(P13_x, P13_y, P13_z)$
2	$(P21_x, P21_y, P21_z)$, $(P22_x, P22_y, P22_z)$, $(P23_x, P23_y, P23_z)$
3	$(P31_x, P31_y, P31_z)$, $(P32_x, P32_y, P32_z)$, $(P33_x, P33_y, P33_z)$
...	...

Triangle	Neighbor triangles
1	(0, 0, 2)
2	(7, 0, 12)
3	(234, 12, 678)
...	...

Fig. 23: Data structure used in this approach.

4.2.2 Algorithm

The parsing of the line-of-sight might generate R-tree queries which do not contain any intersection point with the LOS at all but many possible intersecting lines which have to be analyzed. This results in a high amount of calculation time without delivering any meaningful output. The general idea of the algorithm is to just select the intersected edges by navigating through the TIN independently based on the topological data structure. If the first intersection with one edge of the initial triangle, which is surrounding the observer point, is found, the next intersection must be with one of the edges of the neighbor with whom the selected edge is shared. Figure 24 shows the principle of this idea. The initial triangle is T1 and its intersecting edge with the LOS is E1. The neighbor T2 with whom T1 shares the edge E1 can be directly derived from the data. The edges of T2 then have to be checked for a new intersection with the LOS to find the next triangle. Following this logic the algorithm navigates through the TIN until no new intersection can be found and the target triangle T6 is reached.



Triangle	Edges
T1	E1, E2, E3
T2	E1, E4, E5
T3	E5, E6, E7
T4	E6, E8, E9
T5	E9, E10, E11
T6	E10, E12, E13

Triangle	Neighbor triangles
T1	(T2, ..., ...)
T2	(T1, ..., T3)
T3	(T2, T4, ...)
T4	(T3, ..., T5)
T5	(T4, T6, ...)
T6	(T5, ..., ...)

Fig. 24: General idea of the visibility analysis using a topological data structure.

Accordingly before the visibility can be analyzed, two preliminary works have to be done. First the data structure has to be created and then the initial triangle has to be found. Therefore, the known Python modules for the R-tree and GDAL are used (see chapter 4.2.3).

The logic to create the topological data structure is shown in figure 25. First of all the TIN file has to be imported and the triangle points are extracted from the WKT strings. The triangles are saved in an associative array, in Python named dictionary, which is a collection of keys with associated values. In this case the ID-number of the triangle will be the key and the corresponding value will be the points of the triangle, saved in the Python data type tuple.

Subsequently the previous introduced R-tree helps to enhance the upcoming building of the topology. Apart from that it can be used to solve the problem of finding the initial triangle in the visibility analysis. Therefore, the triangle ID-numbers are inserted into an R-tree with the minimal bounding rectangle of the corresponding triangle.

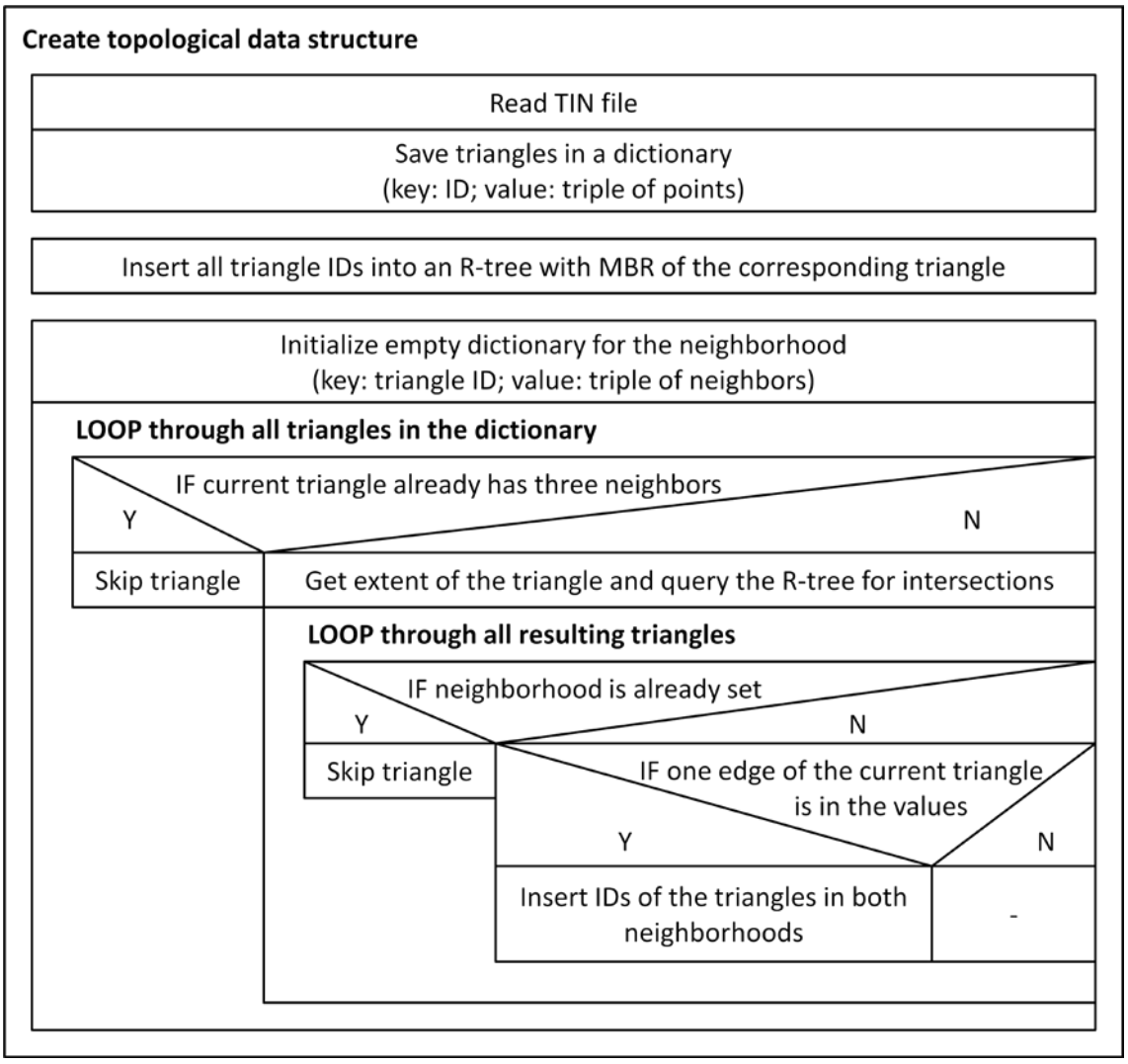


Fig. 25: The logic to create the topological data structure.

Finally, the topology has to be build. The neighborhood of the triangles is stored in a dictionary as well and has to be initialized with empty entries. The key for these entries will also be the ID-number and the value will contain the neighbors, stored in a tuple, whose position (1, 2 or 3) defines the shared edge of the triangle and its neighbor. Thus the corresponding neighbor can be directly addressed if an intersected edge is found during the visibility analysis. Typically the neighborhood is determined by looping through the triangles twice, as every triangle has to be compared with the whole set of triangles until its three neighbors are found. This process can be improved using the R-tree. Instead of comparing each triangle with every other, an R-tree window query can be used to narrow the number of triangles to the ones around the current selected triangle. Thus the calculation complexity decreases significantly.
Furthermore, for every detected neighbor, both neighborhoods, the one of the current selected triangle and the one of its neighbor are updated. Therefore, the algorithm fills multiple neighborhoods with one iteration only. Hence, it is important to check the triangles before starting the detection process, as the neighborhood might already been set.

After creating the topological data structure the visibility can be analyzed. Figure 26 shows the logic for this algorithm. As mentioned before, the R-tree is used to find the initial triangle. More precisely a nearest rectangle query with a small bounding box around the observer point delivers the possibly starting triangles. These triangles have to be checked for containing the observer point using the Python module OGR. After the initial triangle is found, the algorithm starts to analyze this triangle in a separate method. The three edges of the triangle are created and afterwards checked for intersection with the LOS. If one intersection is found, the algorithm saves the intersection point and determines the corresponding neighbor out of the neighbor dictionary. The height of the intersection point is then linearly interpolated and thus the slope from the observer to the intersection point can be calculated. If this slope is greater than the slope of the LOS the target point is not visible. Else the method is called recursively to analyze the neighbor triangle. This recursion will last until the end triangle is reached, which means that no new intersection is found between the three

triangle edges and the LOS. In this case the LOS is not intersected by the terrain and the target is visible.

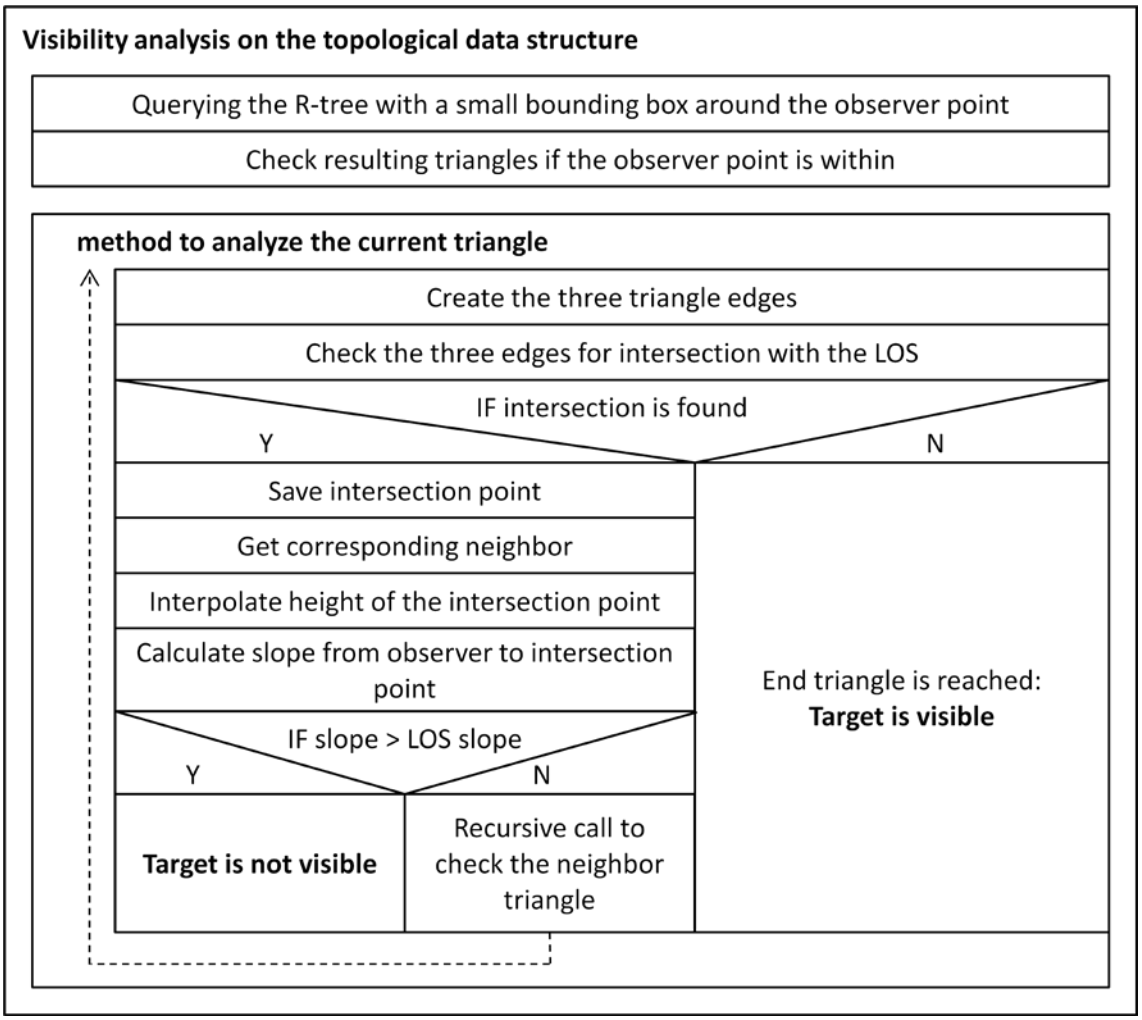


Fig. 26: The logic to analyze the visibility on the topological data structure.

4.3 Development of a SOA-conform system

In this chapter the introduced approaches are integrated into a service-oriented architecture in this case as a web service. Therefore, the PyWPS (see chapter 2.3.4) is used. It provides the whole functionality and structure which is required according to the OGC specification of the Web Processing Service. It also offers a simple framework to implement own processes using the programming language Python.

4.3.1 System design and components

The obvious idea to integrate the introduced approaches into a service-oriented architecture is to implement a PyWPS process for each algorithm. This would be easy but is practically not realizable. A web service in general can receive data from a client via HTTP. The transfer speed typically depends on the internet connection of the client. In this work we deal with data sizes between 35 and 1204 MB which have to be uploaded to the web service for every client request. Therefore, a simple XML-RPC server is implemented which communicates with the PyWPS via HTTP (see XML-RPC, 2010). Figure 27 shows the system design in a schematic diagram. In the following the different system components and their functionality are described in detail.

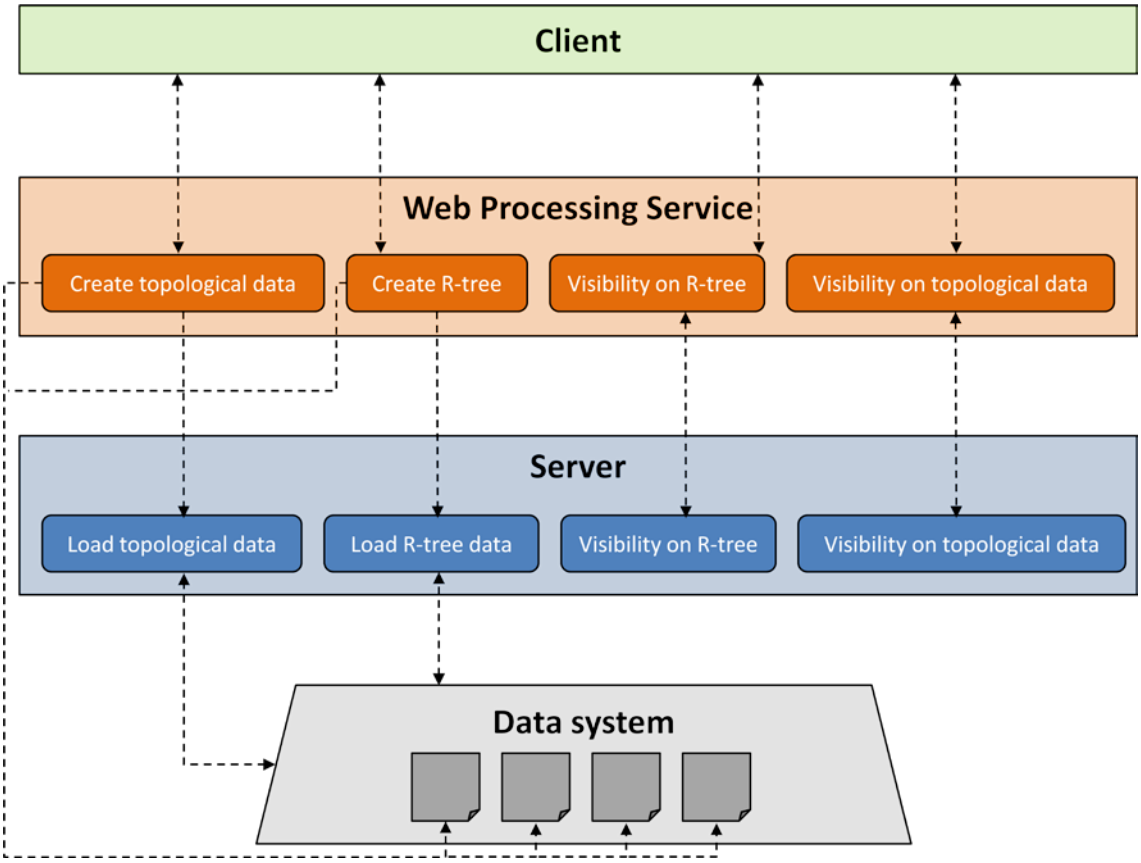


Fig. 27: Schematic diagram of the system design.

The client can interact with the WPS exclusively as the components behind the WPS are not accessible. In this case the client can be any type of application that can establish a connection to the World Wide Web as well any other service in an existing

service-oriented architecture that uses the functionality of this web service. The interoperability of all those different clients with the WPS is assured by the standardized interface of the WPS.

The WPS provides four operations with which the two different data structures can be created and the visibility on those can be analyzed. In figure 28 one can see the detailed functionality of those processes.

The input parameters of the two creation processes are the .shp and .shx file of the TIN. These files can be directly uploaded by the client via HTTP or can be downloaded by the WPS from another web server. After the input files are correctly transferred the processes create the respective data structure (see chapters 4.2.2 and 4.3.2) and save it to the data system of the web server. Afterwards it calls the server to load the data set and provide it under a random generated number. This ID-number of this user specific data set is then set as the return value of the processes. This data can be accessed with the two visibility processes of the WPS. These processes require the coordinates of the target and the observer as well as the ID of the previous generated data set. In addition the R-tree process needs the step size and the overlapping factor as input parameters. The processes then call the server to analyze the visibility according to the introduced approaches and finally return a simple “yes” or “no”.

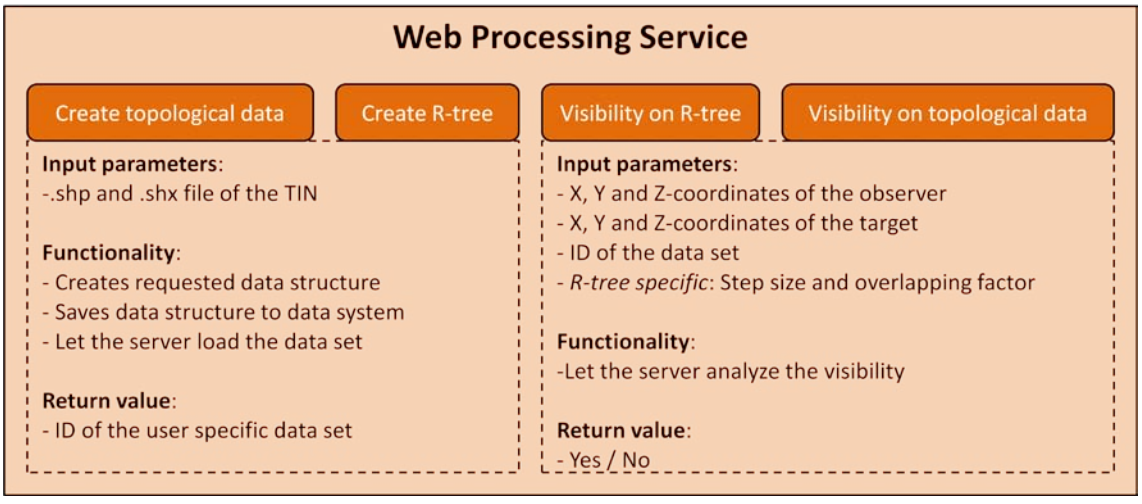


Fig. 28: The four WPS processes in detail.

The server is implemented using the standard Python module SimpleXMLRPCServer with which a program can be created that listens on a specific port and waits for

incoming requests. As mentioned before these requests can be exclusively sent by the WPS.

According to the WPS processes, the server provides four functions (Figure 27). The two loading functions receive the ID-number from the WPS and read the data out of the data system into local variables which therefore saves the data into the RAM of the web server. Thus the data is available while the server is running and can be accessed without reloading. If a client then wants to analyze the visibility, the data is already available and the analysis can start directly. The two visibility functions contain the presented visibility algorithms (see chapter 4.2.3 and 4.3.3). Hence, the actual visibility calculation is sourced out of the WPS into the server. This offers the possibility to simply replace the server component of the system with an implementation in a more efficient programming language than Python like C or C++.

5 Results and Discussion

In this chapter the two visibility algorithms are compared in calculation time and memory requirements to decide which one is more efficient. These parameters are determined by testing the algorithms on the given data sets. The calculation times of the traditional approach on raster data (see Tab. 1) are other benchmarks to which the upcoming performance results are compared.

The test data for the analyses are 52 observer-target pairs, which are randomly placed in the study area. Figure 29 shows the location of the observers (blue) and the targets (red) in which neighboring points are not necessarily building a pair. The distance between the pairs varies between 47.69 and 5899.96 meters whereas the most pairs (ca. 65%) are within 500 meters distance as this is probably the most common distance for applications in urban areas.

For the upcoming performance tests the algorithms were tested on a 4x Dual Xeon 2.93 GHz with 256 GB RAM to handle the large data volume.

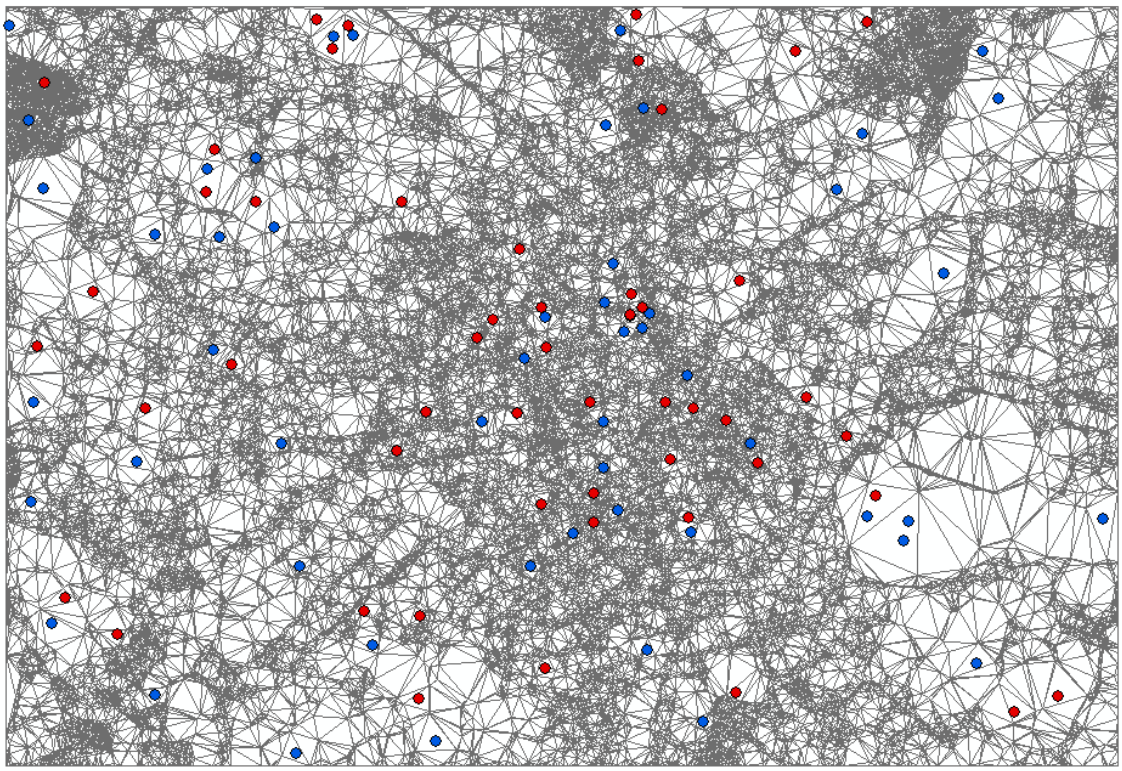


Fig. 29: The test data with 52 observer-target pairs. Blue: observer; Red: target.

5.1 Creation of the data structure

As mentioned in the chapter 4.1 and 4.2, the first step before analyzing the visibility is to transform the TINs into the appropriate data structure. Therefore, the creation time was logged and the creation process for both approaches was repeated 10 times. In the following comparison the median of the calculation times was preferred as it is not prone to outliers in contrast to the arithmetic mean. Those outliers can be the result of background processes on the testing machine which are affecting the normal calculation time. Table 3 shows the median of the calculation times and the total data size of the R-tree and the topological data structure for the different TINs.

	R-tree		**Topological data structure**	
Name	**Calculation time [sec]**	**Total data size [MB]**	**Calculation time [sec]**	**Total data size [MB]**
TIN15	134.54 (2.61%)	134 (2.89%)	65.06 (0.49%)	39.5 (2.85%)
TIN10	436.19 (8.46%)	431 (9.30%)	264.78 (2.01%)	125 (9.03%)
TIN8	711.66 (13.81%)	679 (14.65%)	532.79 (4.05%)	197 (14.23%)
TIN2	5150.24 (100.00%)	4633 (100.00%)	13125.65 (100.00%)	1384 (100.00%)

Tab. 3: Calculation time and the total data size of the R-tree and the topological data structure for the different TINs (the percentages refer to the largest value in the respective column).

It is obvious that both parameters increase due to the rising resolution of the TINs. However, it seems that the calculation time of the R-tree and the topological approach proceed differently as the calculation of the topological data structure is faster for TIN15, TIN10 and TIN8 but vastly increases for TIN2. This assumption is verified by the scatter plots shown in figure 30 between the number of triangles of the TIN and the calculation time. As one can see the calculation time of the topological data structure first increases slightly while the calculation time rises up according to the increased number of triangles. On the other hand, the calculation time of the R-tree seems to increase almost linearly. The total data size of both approaches increases linearly

(Fig. 31) whereas the R-tree uses over 3 times more disk space than the topological data structure because it saves each edge of the triangles.

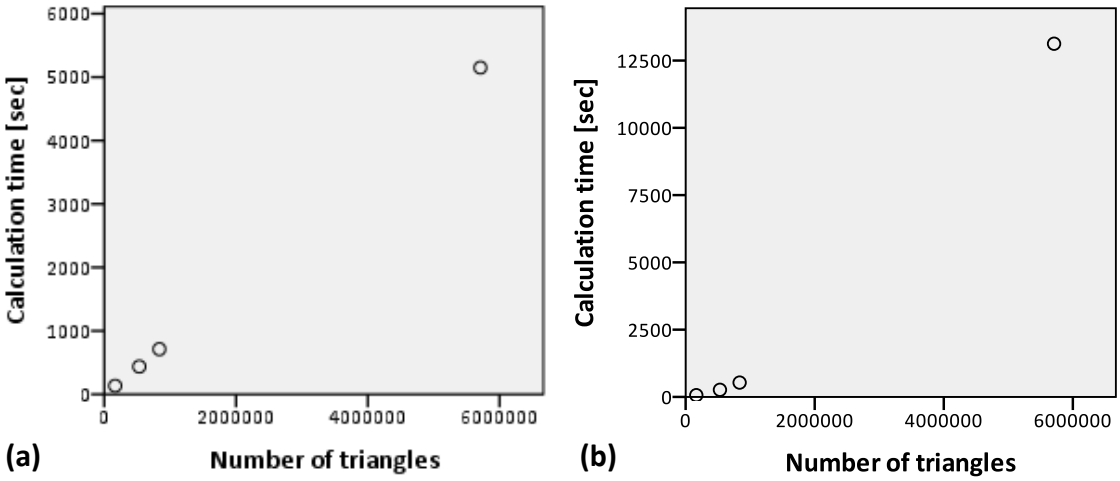


Fig. 30: Scatter plots between the number of triangles and the calculation time for the R-tree (a) and the topological data structure (b).

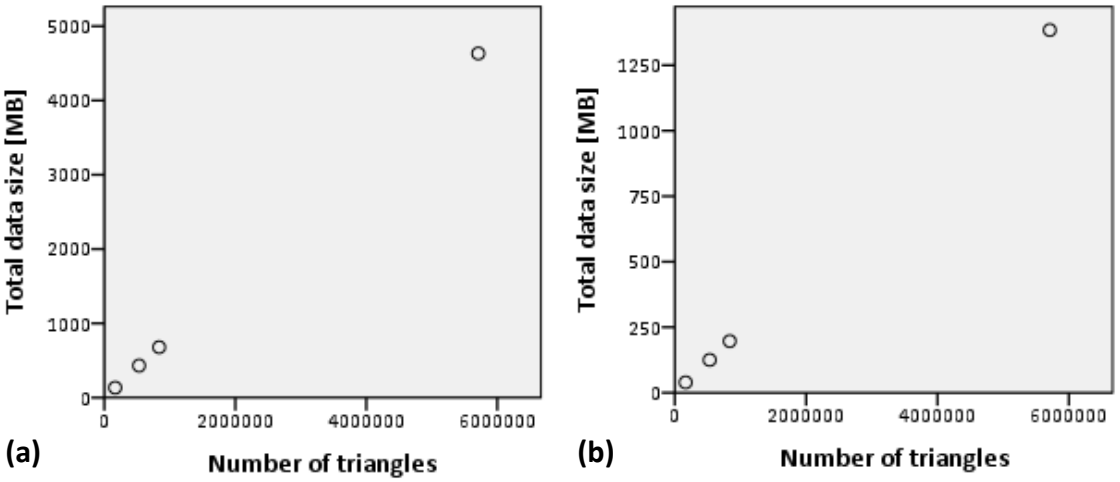


Fig. 31: Scatter plots between the number of triangles and the total data size of the R-tree (a) and the topological data structure (b).

This leads to the conclusion that the topological data structure can be created faster for low resolution TINs as the R-tree is more efficient for a higher resolution. The disadvantage of the topological approach on high-resolution TINs is mainly due to the use of an R-tree windows query during its generation process. The computation time of this spatial query increases according to the number of triangles that has to be

checked. In consideration of the memory requirements, the topological data structure is more efficient as it uses around one-third disk space compared to the R-tree.

5.2 Visibility analysis

To evaluate the visibility algorithms a test program was written which directly calls the visibility functions on the server and records the calculation times. This process is repeated 50 times for every observer-target pair on every available TIN. Afterwards these recorded times were processed into the following statistical parameters:

- Arithmetic mean
- Minimum value
- Maximum value
- Median
- Standard deviation

Also included in the following evaluation process is the length of the LOS as well as the number of triangles crossed by the LOS.

TIN	Parameter	Minimum	Maximum	Mean
TIN15	Standard deviation (R-tree)	0.00016	0.3148	0.1101
	Standard deviation (topological)	0.00003	0.1303	0.0203
TIN10	Standard deviation (R-tree)	0.00021	0.9479	0.3910
	Standard deviation (topological)	0.00010	0.4902	0.1056
TIN8	Standard deviation (R-tree)	0.00024	1.3671	0.6171
	Standard deviation (topological)	0.00005	0.8878	0.2028
TIN2	Standard deviation (R-tree)	0.01491	6.9087	5.1916
	Standard deviation (topological)	0.00031	6.8588	2.4262

Tab. 4: Descriptive statistics of the standard deviations of the computation times on the different TINs.

As mentioned in the previous chapter 5.2 the calculation time on the testing machine might be affected by background processes. The longer the actual calculation will take, the higher is the possibility that a background process affects it. The descriptive

statistics of the standard deviations support this assumption as the maximum and the mean of the standard deviation on TIN2 vastly increases (see Tab. 4). Based on this fact the standard parameter used to analyze the calculation time might vary between the median, the minimum and the mean as some test series are highly affected by background processes while other series are not.

5.2.1 Comparison of the two visibility approaches

Based on the different ideas of the two approaches, the calculation times depend on different parameters. The R-tree algorithm mainly depends on the distance between the observer and the target as it has to parse the LOS (Fig. 32).

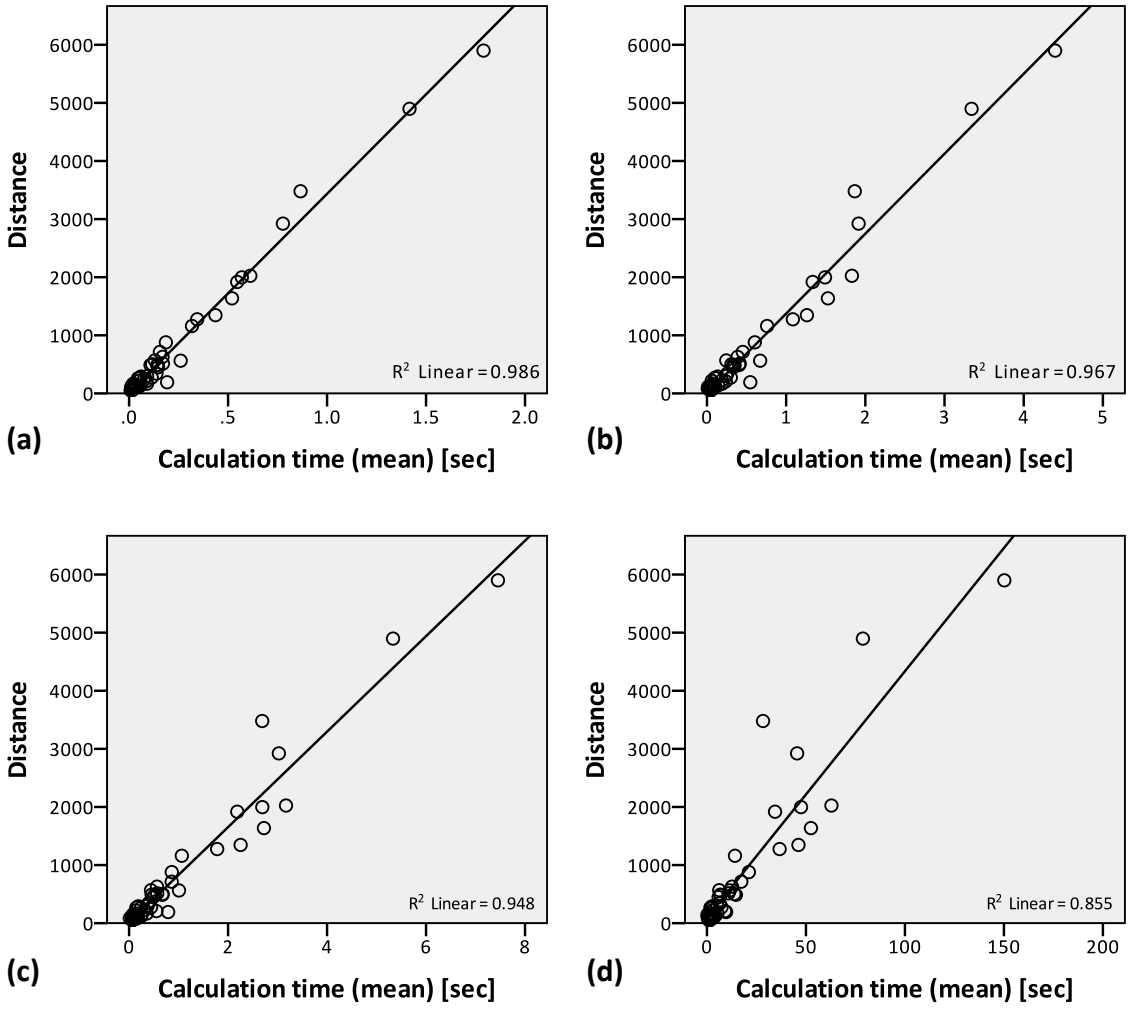


Fig. 32: Scatter plots between the distance from the observer to the target and the mean calculation time of the R-tree on TIN15 (a), TIN10 (b), TIN8 (c) and TIN2 (d).

The algorithm based on the topological data structure mainly depends on the number of crossed triangles by the LOS as its logic is based on exactly these triangles. The dependency is almost perfectly linear for every available TIN resolution (Fig. 33).

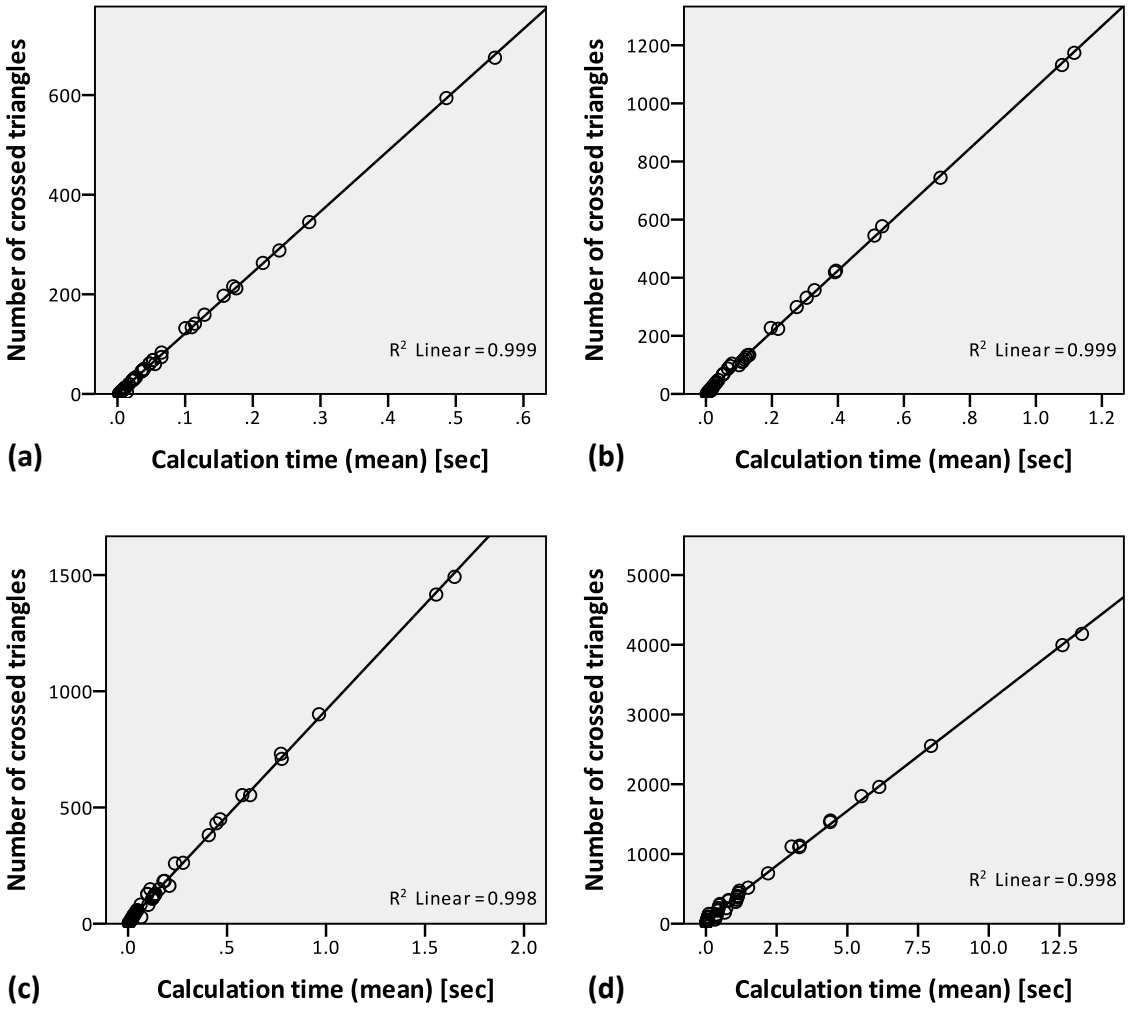


Fig. 33: Scatter plots between the number of crossed triangles by the LOS and the mean calculation time of the topological data structure on TIN15 (a), TIN10 (b), TIN8 (c) and TIN2 (d).

This approach seems to depend on the distance as well (Fig. 34), but the distance and the number of crossed triangles are highly correlated among each other. In consideration of the idea behind the algorithm the dependency on the distance between the observer and the target is not logical. If one assumes an arbitrary LOS intersects five triangles, then it does not matter how large these triangles are and how large the distance of the LOS is. Compared to the R-tree approach this is an enormous

advantage as the topological algorithm do not need any settings (like step size and overlapping factor for the R-tree) to work efficiently on any contrivable TIN resolution. The R-tree algorithm has to be adjusted to work efficiently on a higher or a considerable lower TIN resolution.

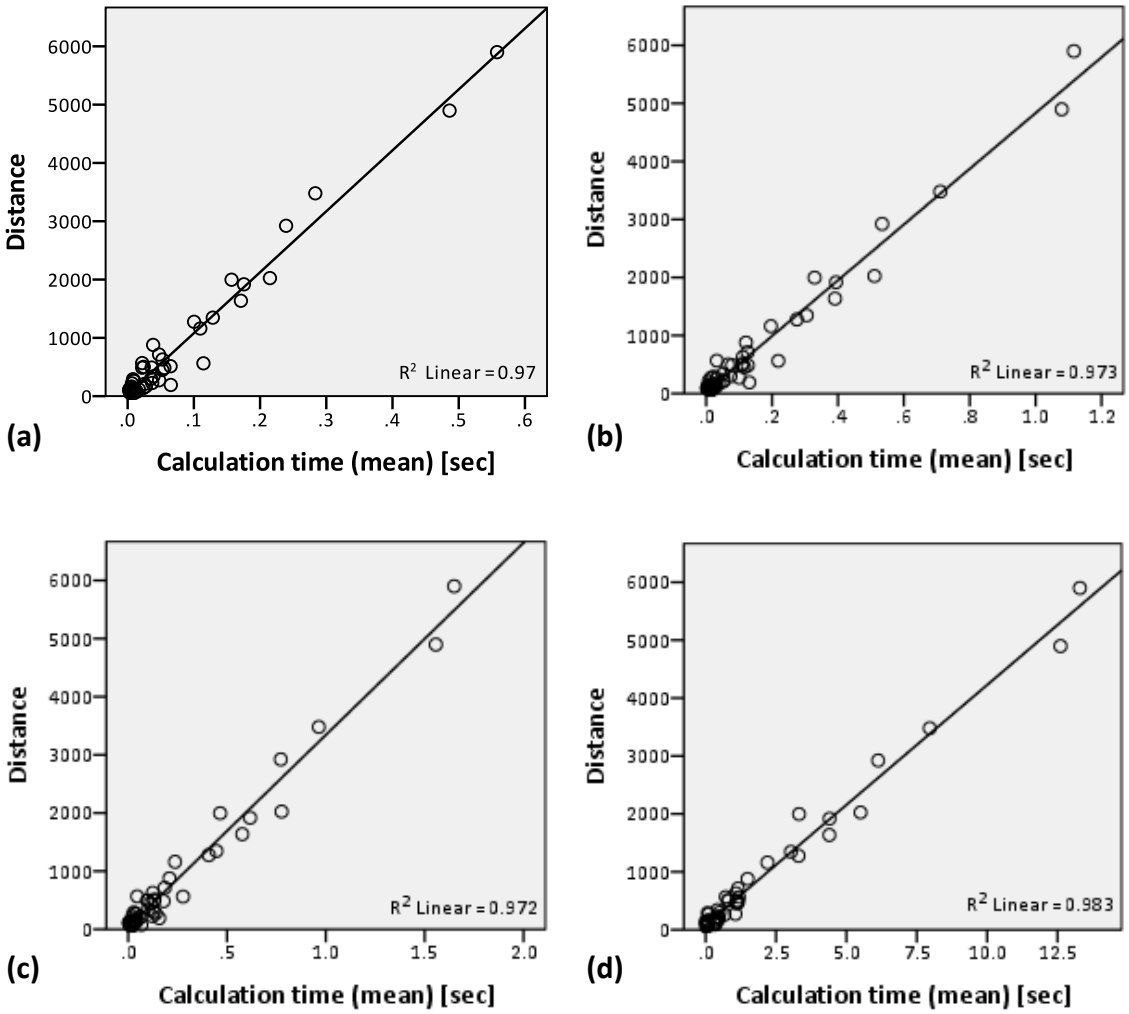


Fig. 34: Scatter plots between the distance from the observer to the target and the mean calculation time of the topological data structure on TIN15 (a), TIN10 (b), TIN8 (c) and TIN2 (d).

In a nutshell the dependencies of the two approaches reveal some weaknesses of the R-tree visibility algorithm. The rectangular window query might always involve triangle edges that are not intersected by the line-of-sight but have to be checked for intersection. The computing time for this is comparatively low but still increases due to

the TIN resolution. Adding the high number of R-tree queries by parsing the LOS, the total calculation time especially for TINs with higher resolutions increases significantly. In contrast, the topological approach generates as little calculations as possible and is therefore faster on every available TIN. Table 5 shows the descriptive statistics of the mean visibility calculation times on the R-tree and on the topological data structure. It is obvious that the topological approach outperforms the R-tree on every available TIN. For TIN2 the calculation in the mean is almost 10 times faster.

	R-Tree			**Topological data structure**		
TIN	**Minimum**	**Maximum**	**Mean**	**Minimum**	**Maximum**	**Mean**
TIN15	0.0079	1.7904	0.2195	0.0024	0.5579	0.0695
TIN10	0.0105	4.3966	0.5557	0.0021	1.1163	0.1522
TIN8	0.0149	7.4529	0.9157	0.0034	1.6491	0.2185
TIN2	0.2980	150.2021	15.7479	0.0069	13.2951	1.6332

Tab. 5: Descriptive statistics of the mean visibility calculation times on the R-tree and on the topological data structure.

These results are not expected as the topological data structure highly differs from the specification in the literature. Typically the three primitives are saved separately to avoid redundancy. One attempt to transform the TINs strictly into this data structure was discarded due to enormous computational complexity. To save for example one new point to the point data structure, the whole array has to be checked for redundancy before the point can finally be added. This results in a total calculation time for the smallest TIN (TIN15) of around 22 hours. It turned out that the best theory not necessarily is the best model for a practical realization.

5.3 The SOA-conform system

This chapter demonstrates the functionalities of the developed system in particular using the example of the topological approach and shows the potential by including some advanced processes. Before the system can be used, the server has to be initialized:

```
python serverVisibility.py <DATAFOLDER>
```

During the initialization the server reads all previous saved data from the given data folder. In that way the server also works correctly after a system crash or a system reboot. The output of the server can be written to a log file or directly examined in the console.

Once the server has started correctly, the WPS can send requests over HTTP. PyWPS processes can be executed using a web browser or directly on the command line:

```
wps.py "service=wps&version=1.0.0&request=getCapabilities"
```

With the previous command the capabilities of the WPS are listed, which among others contain the names of the available processes as well as some metadata about them:

```
<wps:ProcessOfferings>
        <wps:Process wps:processVersion="1.0">
              <ows:Identifier>createTopoData</ows:Identifier>
              <ows:Title>Create topological datastructure</ows:Title>
              <ows:Abstract>[…]</ows:Abstract>
        </wps:Process>
        <wps:Process wps:processVersion="1.0">
              <ows:Identifier>createRTree</ows:Identifier>
              <ows:Title>Create R-tree</ows:Title>
              <ows:Abstract>[…]</ows:Abstract>
        </wps:Process>
        <wps:Process wps:processVersion="1.0">
              <ows:Identifier>visTopo</ows:Identifier>
              <ows:Title>Visibility on the topological data
                   </ows:Title>
              <ows:Abstract>[…]</ows:Abstract>
        </wps:Process>
        <wps:Process wps:processVersion="1.0">
              <ows:Identifier>visRTree</ows:Identifier>
              <ows:Title>Visibility on R-tree</ows:Title>
              <ows:Abstract>[…]</ows:Abstract>
        </wps:Process>
        […]
 </wps:ProcessOfferings>
```

Before a process is executed, it is advisable to check the detailed metadata and to regard the required input parameters. These details can be derived by the "DescribeProcess" command:

```
wps.py "service=wps&version=1.0.0&request=describeProcess&
        identifier=createTopoData"

   […]
   <ows:Identifier>createTopoData</ows:Identifier>
   […]
   <DataInputs>
       <Input minOccurs="1" maxOccurs="1">
           <ows:Identifier>tinShp</ows:Identifier>
           <ows:Title>Path to shp file of the TIN</ows:Title>
           <ows:Abstract>Path to shp file of the TIN</ows:Abstract>
           <ComplexData>
               […]
           </ComplexData>
       </Input>
       […]
   </DataInputs>
```

Afterwards the WPS process "createTopoData" to create the topological data structure can be called. In this case the input data was previously uploaded to another web server and is automatically downloaded to a temporary folder on the data system:

```
wps.py "service=wps&version=1.0.0&request=execute&
        identifier=createTopoData&datainputs=
        tinShp=http://www.florianhillen.de/rastertin_tol15_subset.shp;
        tinShx=http://www.florianhillen.de/rastertin_tol15_subset.shx"

PyWPS Status [processpaused]: Getting input tinShx of process
createTopoData
PyWPS Status [processpaused]: Getting input tinShp of process
createTopoData
PyWPS Status [processstarted][0.0]: Process createTopoData started
[…]
<wps:ProcessOutputs>
     <wps:Output>
          <ows:Identifier>id</ows:Identifier>
          <ows:Title>ID of topological data</ows:Title>
          <wps:Data>
              <wps:LiteralData dataType="integer">
                  548904
              </wps:LiteralData>
          </wps:Data>
     </wps:Output>
</wps:ProcessOutputs>
```

The resulting data structure is directly saved by the process to the predefined data folder. The process then calls the server to load this data structure to the RAM using an arbitrary ID-number. This number, in this example 548904, is set as the return value

of the WPS process and is delivered to the client. Afterwards the client can run the corresponding visibility analysis on its specific data set using the ID-number. The result is returned in the XML format and in this case the given target is visible from the defined observer:

```
wps.py "service=wps&version=1.0.0&request=execute&identifier=visTopo&
        datainputs=observer_x=3434931.5;observer_y=5794119.2;
        observer_z=62;target_x=3434958.3;target_y=5794190.6;
        target_z=82;id=548904"

<wps:ProcessOutputs>
     <wps:Output>
         <ows:Identifier>visible</ows:Identifier>
         <ows:Title>Visibility of the target</ows:Title>
         <wps:Data>
             <wps:LiteralData dataType="integer">yes</wps:LiteralData>
         </wps:Data>
     </wps:Output>
</wps:ProcessOutputs>
```

The potential of the WPS in combination with the server is large. For example a WPS process can import given GML (Geography Markup Language) files that includes many observers and targets to start multiple visibility analyses. The result contains the visibility of every observer-target pair:

```
wps.py "service=wps&version=1.0.0&request=execute&
        identifier=multiVisTopoGML&datainputs=
        observers=http://www.florianhillen.de/observer.gml;
        targets=http://www.florianhillen.de/target.gml;id=548904"

<wps:ProcessOutputs>
     <wps:Output>
         <ows:Identifier>visible</ows:Identifier>
         <ows:Title>Visibility of the targets</ows:Title>
         <wps:Data>
             <wps:LiteralData dataType="integer">
                 <los id=1><visible>no</visible></los>
                 <los id=2><visible>no</visible></los>
                 <los id=3><visible>yes</visible></los>
                 <los id=4><visible>yes</visible></los>
              </wps:LiteralData>
            </wps:Data>
         </wps:Data>
     </wps:Output>
</wps:ProcessOutputs>
```

The basic functionality of the server can therefore be used to create more complex requests. Over the standardized interface of the WPS the system can be included in any implemented service-oriented architecture. The functionality can be as well used in geographical information systems (GIS) that support a connection to a WPS.

6 Conclusion

In this work two approaches to calculate the visibility on triangulated irregular networks have been proposed and successfully integrated into a service-oriented architecture. First of all, one has to say that it is possible to provide visibility analysis on laser data at the push of a button. This means that it succeeds to bring high accurate visibility queries together with a low calculation time which is important for location based services and many other geospatial applications.

Based on the concluding evaluation the following research statements can be made:

- It can be concluded that both approaches in general are much faster than the traditional calculation of visibility based on raster data and that the **TIN** is a good alternative data model for terrain analyses. Though it needs more preliminary processing effort as the most elevation data is delivered in raster format.
- The **R-tree** delivers a data structure that easily organizes the triangles without any large preprocessing effort and a linear creation time. The general idea of parsing the line-of-sight is attended by the fact that the algorithm might be “out of work” for certain regions (for example large triangles parsed with a low step size) or has to check too many inconclusive triangles (many triangle edges within one R-tree query). This creates a calculation overhead for almost any possible TIN resolution.
- The general **topological data structure** as proposed in the literature is not capable for a high number of triangles. A related but simpler data structure is therefore used in this work. After improving the creation process using the R-tree, the data structure is generated faster than the R-tree for low resolution TINs. Though the R-tree creation is faster on higher TIN resolutions because the spatial query during the topological generation process delivers more triangle edges which have to be analyzed according to the increased resolution. Furthermore, this approach needs around one-third less disk space compared to the R-tree.

- The **visibility analysis on the topological data structure** overshoots all expectations. The algorithm outperforms the R-tree approach on every available TIN resolution. On the largest examined TIN the calculation is in the mean almost 10 times faster than the calculation based on the R-tree (see Tab. 5). The high performance of this approach mainly depends on the fast navigation through the triangles based on the topological data structure which makes the parsing of the line-of-sight unnecessary. Thus, the algorithm almost entirely avoids the usage of spatial queries which are very time-consuming especially on large data sets. Another advantage is the minimized number of calculations to solve the visibility problem as only the actual intersected edges are processed by the algorithm.
- The integration of the approaches into a **service-oriented architecture** revealed some issues with the high data size of the input data. As it is not capable for the client to upload the TIN for every visibility request, a server was implemented to provide the user specific data sets. This extension of the system allows the WPS processes to quickly access the requested user data. The computational overhead which naturally results using a web service is thereby as low as possible.

From the previous research findings the following future prospects can be derived:

- One potential future work is to improve the visibility approaches using multi-core processors and multi-threading programming techniques. The R-tree approach can be enhanced by dividing the LOS and parsing the single parts simultaneously. This will reduce the calculation time according to the number of used processor cores. For the topological approach the TIN can be additionally passed through backwards. This means that two initial triangles, the ones surrounding the observer and the target point, has to be determined and will be used as the starting point for two simultaneous threads.
- Another improvement of the calculation speed might be the implementation of the system's server component in a more efficient programming language like C or C++ as described in chapter 4.4.2.

- Many approaches can be found by extending the developed SOA-conform system. The possibility to import data from a Web Feature Service into a WPS process will make it easier for the client to transfer his data. The output data can be modified as well by returning a geometry representing the visible part of the LOS for instance.
- Apart from that the topological approach can be used to solve other visibility problems with a higher dimension like the viewshed analysis. Starting with the initial triangle, the neighboring triangles within a defined radius can be incrementally checked for visibility to calculate the viewshed of a given observer. The R-tree approach can also be extended for this kind of problem as a window query around the observer point delivers the neighboring triangles as well.
- One disadvantage of the current service is the missing metadata of the available data on the server. If the user lost its ID-number, the specific data set cannot be queried anymore. The OGC Catalogue Service (CSW) could be used to create a metadata catalogue that is automatically filled once the client uploads new data to the server. Apart from the ID-number, the extent, the underlying coordinate system and the resolution of the data set can be saved as metadata. Thus, a possible other users receive an overview of the existing data and might reuse previously uploaded data sets. Another use-case of this extended system is that a location based service directly can determine an adequate data set and can request the demanded visibility query on it based on the current geographical position of the mobile device as well as the needed data resolution.

References

Internet links were last visited: 11.05.2011

52°North, 2011. 52°North WPS Geoprocessing Community - Welcome. URL: http://52north.org/communities/geoprocessing/wps/index.html

Ackermann, F., 1999. Airborne laser scanning – present status and future expectations. ISPRS Journal of Photogrammetry and Remote Sensing 54, pp. 64-67.

Andrade, M. V. A., Magalhães, S. V. G., Magalhães, M. A., Franklin, W. R., Cutler, B. M., 2011. Efficient viewshed computation on terrain in external memory. Geoinformatica, Vol. 15, No. 2, pp. 381-397.

Arge, L., de Berg, M., Haverkort, H. J., Yi, K., 2004. The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 347-358.

Arnold, N. S., Rees, W. G., Devereux, B. J., Amable, G. S., 2006. Evaluating the potential of high-resolution airborne LiDAR data in glaciology. International Journal of Remote Sensing 27 (5-6), pp. 1233-1251.

Baltsavias, E. P., Favey, E., Bauder, A., Bosch, H., Peteraki, M., 2001. Digital Surface Modelling by Airborne Laser Scanning and Digital Photogrammetry for Glacier Monitoring. Photogrammetric Record, 17(98), pp. 243–273.

Brauner, J., Foerster, T., Schaeffer, B., Baranski, B., 2009. Towards a Research Agenda for Geoprocessing Services. 12th AGILE International Conference on Geographic Information Science 2009.

Cohen-Or, D., Shaked, A., 1995. Visibility and Dead-Zones in Digital Terrain Maps. Computer Graphics Forum, 1995, Vol. 14, No. 3, pp. 171-180.

deegree, 2011. deegree3/ProcessingService - deegree Wiki. URL: http://wiki.deegree.org/deegreeWiki/deegree3/ProcessingService

de Berg, M., Cheong, O., van Kreveld, M., Overmars, M., 2008. Computational Geometry: Algorithms and Applications. Springer, Berlin, pp. 191-218.

De Floriani, L., 1987. Surface representations based on triangular grids. The Visual Computer, 1987, Vol. 3, No. 1, pp. 27-50.

De Floriani, L., Magillo, P., 1994. Visibility algorithms on triangulated digital terrain models. International Journal of Geographical Information Science, 1994, Vol. 8, No. 1, pp. 13-41.

De Floriani, L., Magillo, P., 1999. Intervisibility on Terrains. Geographic Information Systems: Principles, Techniques, Management and Applications, pp. 543-556.

De Floriani, L., Magillo, P., 2003. Algorithms for Visibility Computation on Terrains: a Survey. Environment and Planning B – Planning and Design 30, 5 (2003), pp. 709-728.

de Lange, N., 2005. Geoinformatik in Theorie und Praxis. Springer, Berlin, pp. 360-361.

ESRI, 2010a. ArcGIS Desktop Help 9.2 - Raster To TIN (3D Analyst). URL: http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?id=961&pid=959&topicname=Raster_To_TIN_%283D_Analyst%29

ESRI, 2010b. ArcGIS Desktop Help 9.2 - How Raster TIN (3D Analyst) works. URL: http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=How%20Raster%20TIN%20%283D%20Analyst%29%20works

Franklin, W. R, Ray, C. K., Mehta, S., 1994. Geometric Algorithms for Siting of Ait Defense Missile Batteries. Technical Report DAAL03-86-D-0001, Battelle, Columbus Division, Ohio.

Geist, T., Lutz, E., and Stötter, J., 2003. Airborne laser scanning technology and its potential for applications in glaciology. Proceedings, ISPRSWorkshop on 3-D reconstruction from airborne laserscanner and INSAR data, Dresden, Germany.

Gomarasca, M. A., 2009. Basics of Geomatics. Springer, Berlin, pp. 323-324.

Goodchild, M. F., Lee, J., 1989. Coverage Problems and Visibility Regions on Topographic Surfaces. Annals of Operations Research, 18 (1989), pp. 175-186.

Goodrich, M. T., Tsay, J.-J., Vengroff, D. E., Vitter, J. S., 1993. External-memory computational geometry. Proceedings of the IEEE Symposium on Foundations of Computer Science, 1993, pp. 714–723.

GSM Association, 2003. Permanent Reference Document: SE.23. Location Based Services, Version: 3.1.0. URL: http://www.gsmworld.com/documents/se23.pdf

Guttman, A., 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47-57.

Haverkort, H., Toma, L., Zhuang, Y., 2009. Computing Visibility on Terrains in External Memory. Proceedings of the 9th Workshop on Algorithm Engineering and Experiments / Workshop on Analytic Algorithms and Combinatorics (ALENEX/ANALCO 2007), pp. 13-22

Höfle, B., Geist, T., Rutzinger, M., Pfeifer, N., 2007. Glacier Surface Segmentation Using Airborne Laser Scanning Point Cloud and Intensity Data. International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences. Vol. XXXVI(Part 3/W52), pp. 195-200.

Höfle, B., Rutzinger, M., 2011. Topographic airborne LiDAR in geomorphology: A technological perspective. Zeitschrift für Geomorphologie, Vol. 55, Suppl. 2, pp. 1-29.

Izraelevitz, D., 2003. A Fast Algorithm for Approximate Viewshed Computation. Photogrammetric Engineering and Remote Sensing, Vol. 69, No. 7, 2003, pp. 767-774.

Jochem, A., Hollaus, M., Rutzinger, M., Höfle, B., 2011. Estimation of Aboveground Biomass in Alpine Forests: A Semi-Empirical Approach Considering Canopy Transparency Derived from Airborne LiDAR Data. Sensors. Vol. 11(1), pp. 278-295.

Kemper, A., Eickler, A., 2006. Datenbanksysteme: Eine Einführung. 6. Auflage, Oldenbourg Wissenschaftsverlag, München, pp. 215-216.

Kidner, D. B., Ware, J. M., Sparkes, A. J., Jones, C. B., 2000. Multiscale Terrain and Topographic Modelling with the Implicit TIN. Transactions in GIS, 2000, 4(4), pp. 379-408.

Kidner, D. B., Sparkes, A. J., Dorey, M. I., Ware, J. M., Jones, C. B., 2001. Visibility Analysis with the Multiscale Implicit TIN. Transactions in GIS, 2001, 5(1), pp. 19-37.

Krafzig, D., Banke, K., Slama, D., 2005. Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall International, 2005. pp. 56-63.

Large, A. R. G., Heritage, G. L., 2009. Laser Scanning – Evolution of the Discipline. Laser Scanning for the Environmental Sciences. Wiley-Blackwell, Oxford, UK.

Lee, J., 1991. Analyses of visibility sites on topographic surfaces. International Journal of Geographical Information Systems, 1991, Vol. 5, No. 4, pp. 413-429.

Maloy, M. A., Dean, D. J., 2001. An Accuracy Assessment of Various GIS-Based Viewshed Delineation Techniques. Photogrammetric Engineering and Remote Sensing, Vol. 67, No. 11, 2001, pp. 1293-1298.

Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y., 2005. R-Trees: Theory and Applications. Advanced Information and Knowledge Processing. Springer, Berlin, pp. 7-13.

Melzer, I., 2010. Service-orientierte Architekturen mit Web Services: Konzepte – Standards – Praxis. 4. Aufl. Spektrum Akademischer Verlag, pp. 9-31.

Mitchell, T., Emde, A., Christl, A., 2008. Web-Mapping mit Open Source-GIS-Tools. O'Reilly, p. 234.

Næsset, E., 1997. Determination of mean tree height of forest stands using airborne laser scanner data. ISPRS Journal of Photogrammetry and Remote Sensing 52(2), pp. 49-56.

Offermann, P., Schröpfer, C., Ahrens, M., 2006. Extending the UN/CEFACT Modeling Methodology and Core Components for Intra-organizational Service Orchestration. Service-Oriented Computing ICSOC 2006. Springer, pp. 154-165.

Open Geospatial Consortium, 2005. Web Feature Service Implementation Specification, Version 1.1.0. URL: http://portal.opengeospatial.org/files/?artifact_id=8339

Open Geospatial Consortium, 2006. OpenGIS Web Map Server Implementation Specification, Version 1.3.0. URL: http://portal.opengeospatial.org/files/?artifact_id=14416

Open Geospatial Consortium, 2007. OpenGIS Web Processing Service, Version 1.0.0. URL: http://portal.opengeospatial.org/files/?artifact_id=24151

Open Geospatial Consortium, 2010a. OpenGIS Implementation Standard for Geographic information – Simple feature access – Part 1: Common architecture, Version 1.2.1. URL: http://portal.opengeospatial.org/files/?artifact_id=25355

Open Geospatial Consortium, 2010b. OGC Web Services Common Standard, Version 2.0.0. URL: http://portal.opengeospatial.org/files/?artifact_id=38867

Open Geospatial Consortium, 2011a. OGC Members. URL: http://www.opengeospatial.org/ogc/members

Open Geospatial Consortium, 2011b. About OGC. URL: http://www.opengeospatial.org/ogc

Open Geospatial Consortium, 2011c. OGC Standards and Specifications. URL: http://www.opengeospatial.org/standards

Peucker, T. K., Fowler, R. J., Little, J. J., Mark, D. M., 1978. The Triangulated Irregular Network. Proceedings of the DTM Symposium, ASP-ACSM, pp. 24-31.

Python, 2011a. Python Package Index : Rtree 0.6.0. URL: http://pypi.python.org/pypi/Rtree

Python, 2011b. Python Package Index : GDAL 1.7.1. URL: http://pypi.python.org/pypi/GDAL/

PyWPS, 2011. Welcome to PyWPS - PyWPS. URL: http://pywps.wald.intevation.org/

Rana, S., Morley, J., 2002. Optimising visibility analyses using topographic features on the terrain. Centre for Advanced Spatial Analysis Working Paper Series. Paper 44.

Thalheim, B., Libkin, L., 1998. Semantics in Databases. Lecture Notes in Computer Science, Vol. 1358. Springer, pp. 124-130.

Tigris, 2011. wpsint.tigris.org. URL: http://wpsint.tigris.org/

van Engelen, R., Gallivan, K., Gupta, G., Cybenko, G., 2000. XML-RPC agents for distributed scientific computing. IMACS'2000 Conference, Lausanne, Switzerland.

van Kreveld, M., 1996. Variations on sweep algorithms: Efficient computation of extended viewsheds and classifications. Proceedings of the 7th Int. Symposium on Spatial Data Handling, 1996, pp. 15-27.

Vosselman, G., 2008. Analysis of planimetric accuracy of airborne laser scanning surveys. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Science 37 (Part B3a), pp. 99-104.

W3C, 2007. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). URL: http://www.w3.org/TR/soap12-part1/

Wang, F. Z., Helian, N., Wu, S., Guo, Y., Deng, D. Y., Meng, L., Zhang, W., Crowncroft, J., Bacon, J., Parker, M. A., 2009. Eight Times Acceleration of Geospatial Data Archiving and Distribution on the Grids. IEEE Transactions On Geoscience And Remote Sensing, Vol. 47, No. 5, May 2009, pp. 1444-1453.

Wang, J., Robinson, G. J., White, K., 2000. Generating Viewsheds without Using Sightlines. Photogrammetric Engineering and Remote Sensing, Vol. 66, No. 1, 2000, pp. 87-90.

Wehr, A., Lohr, U., 1999. Airborne laser scanning – an introduction and overview. ISPRS Journal of Photogrammetry and Remote Sensing 54, pp. 68-82.

XML-RPC, 2010. XML-RPC Home Page. URL: http://www.xmlrpc.com/

Yeung, A. K. W., Hall, G. B., 2007. Spatial database systems: design, implementation and project management. GeoJournal Library, Vol. 87. Springer, pp. 100-103.

ZOO Project, 2011. ZOO Project - Open WPS Platform. URL: http://www.zoo-project.org/

List of Figures

List of Tables

Appendix

A Source code examples

Source code of the **XML-RPC server**:

```
#! /usr/bin/python26

__author__="florian"
__date__ ="$16.03.2011 17:03:28$"

from SimpleXMLRPCServer import SimpleXMLRPCServer
from rtree import Rtree
from shapely.geometry import Point
from shapely.geometry import LineString
from shapely.geometry import Polygon
import cPickle
import os
import sys

class Lookup(dict):
    """
    a dictionary which can lookup value by key, or keys by value
    """
    def __init__(self, items=[]):
        """items can be a list of pair_lists or a dictionary"""
        dict.__init__(self, items)

    def get_key(self, value):
        """find the key(s) as a list given a value"""
        for item in self.items():
            if item[1] == value:
                return item[0]
        return ""

    def get_value(self, key):
        """find the value given a key"""
        return self[key]

class DataTopo:
    t = {}
    triangles = Lookup(t)
    n = {}
    neighbors = Lookup(n)
    idx = ""
    id = ""
    fileRoot = ""

    def __init__(self, id, fileRoot):
        print "             - initialising topo Data"
        self.id = id
        self.fileRoot = fileRoot
        print "                   *done*"
```

```
    # load topological data
    def loadData(self):
        fileTriangles = self.fileRoot+"topoTriangles_%s.pkl" % self.id
        fileNeighbors = self.fileRoot+"topoNeighbors_%s.pkl" % self.id
        fileRTree = self.fileRoot+"topoRTree_%s" % self.id

        print "             - now loading Triangles"
        input = open(fileTriangles, 'rb')
        self.triangles = cPickle.load(input)
        input.close()
        print "                   *done*"

        print "             - now loading Neighbors"
        input = open(fileNeighbors, 'rb')
        self.neighbors = cPickle.load(input)
        input.close()
        print "                   *done*"

        print "             - now loading RTree"
        self.idx = Rtree(fileRTree)
        print "                   *done*"

    def getSlopeXZ(self, observer, target):
        deltaZ = observer.z-target.z
        deltaX = observer.x-target.x
        if deltaX == 0:
            print "deltaX == 0"
        else:
            slope = deltaZ/deltaX
        return slope

    # get crossed triangles
    def crossedTrianglesVis(self, observer, los, slope, t, last, counter):
        # create the three triangle lines
        line1 = LineString(((self.triangles[t][0][0],
                    self.triangles[t][0][1], self.triangles[t][0][2]),
                          (self.triangles[t][1][0],
                    self.triangles[t][1][1], self.triangles[t][1][2])))
        line2 = LineString(((self.triangles[t][1][0],
                    self.triangles[t][1][1], self.triangles[t][1][2]),
                          (self.triangles[t][2][0],
                    self.triangles[t][2][1], self.triangles[t][2][2])))
        line3 = LineString(((self.triangles[t][2][0],
                    self.triangles[t][2][1], self.triangles[t][2][2]),
                          (self.triangles[t][0][0],
                    self.triangles[t][0][1], self.triangles[t][0][2])))

        # check which line is intersected by the LOS and set the next
          triangle
        intersection = Point()
        intersectionLine = LineString()
        i1 = los.intersection(line1)
        if i1.wkt != "GEOMETRYCOLLECTION EMPTY":
            if self.neighbors[t][0] != last:
                intersection = i1
                intersectionLine = line1
                n = self.neighbors[t][0]
        i2 = los.intersection(line2)
```

```
        if i2.wkt != "GEOMETRYCOLLECTION EMPTY":
            if self.neighbors[t][1] != last:
                intersection = i2
                intersectionLine = line2
                n = self.neighbors[t][1]
        i3 = los.intersection(line3)
        if i3.wkt != "GEOMETRYCOLLECTION EMPTY":
            if self.neighbors[t][2] != last:
                intersection = i3
                intersectionLine = line3
                n = self.neighbors[t][2]

        # if no intersection found so far, the last triangle is reached and
          the target is visible
        if intersection.wkt == "GEOMETRYCOLLECTION EMPTY":
            return counter

        # interpolate the height with the two points of the line
        positionVector = intersectionLine.coords[0]
        directionVector = (intersectionLine.coords[1][0] –
                               positionVector[0],
                           intersectionLine.coords[1][1] –
                               positionVector[1],
                           intersectionLine.coords[1][2] –
                               positionVector[2])
        if directionVector[0] == 0:
            lambdaLine = 0
        else:
            lambdaLine = (intersection.x - positionVector[0]) /
                               directionVector[0]
        intersectionHeight = positionVector[2] + lambdaLine *
                               directionVector[2]
        intersection = Point(intersection.x, intersection.y,
                             intersectionHeight)

        # calculate angle between observer and intersection point in xz-
          plane
        slopeIntersection = self.getSlopeXZ(observer, intersection)

        # angle of LOS must be higher than angle of observer-intersection
        if slope < slopeIntersection:
            return counter
        else:
            counter += 1
            return self.crossedTrianglesVis(observer, los, slope, n, t,
                                            counter)

    # Analyses the visibility between observer and target
    def analyseVisibility(self, observer, los, slope, t, last):
        # create the three triangle lines
        line1 = LineString(((self.triangles[t][0][0],
                    self.triangles[t][0][1], self.triangles[t][0][2]),
                            (self.triangles[t][1][0],
                    self.triangles[t][1][1], self.triangles[t][1][2])))
        line2 = LineString(((self.triangles[t][1][0],
                    self.triangles[t][1][1], self.triangles[t][1][2]),
                            (self.triangles[t][2][0],
                    self.triangles[t][2][1], self.triangles[t][2][2])))
        line3 = LineString(((self.triangles[t][2][0],
```

```
                self.triangles[t][2][1], self.triangles[t][2][2]),
                        (self.triangles[t][0][0],
                self.triangles[t][0][1], self.triangles[t][0][2])))

        # check which line is intersected by the LOS and set the next
          triangle
        intersection = Point()
        intersectionLine = LineString()
        i1 = los.intersection(line1)
        if i1.wkt != "GEOMETRYCOLLECTION EMPTY":
            if self.neighbors[t][0] != last:
                intersection = i1
                intersectionLine = line1
                n = self.neighbors[t][0]
        i2 = los.intersection(line2)
        if i2.wkt != "GEOMETRYCOLLECTION EMPTY":
            if self.neighbors[t][1] != last:
                intersection = i2
                intersectionLine = line2
                n = self.neighbors[t][1]
        i3 = los.intersection(line3)
        if i3.wkt != "GEOMETRYCOLLECTION EMPTY":
            if self.neighbors[t][2] != last:
                intersection = i3
                intersectionLine = line3
                n = self.neighbors[t][2]

        # if no intersection found so far, the last triangle is reached and
          the target is visible
        if intersection.wkt == "GEOMETRYCOLLECTION EMPTY":
            return True

        # interpolate the height with the two points of the line
        positionVector = intersectionLine.coords[0]
        directionVector = (intersectionLine.coords[1][0] -
                    positionVector[0],
                            intersectionLine.coords[1][1] -
                    positionVector[1],
                            intersectionLine.coords[1][2] -
                    positionVector[2])
        if directionVector[0] == 0:
            lambdaLine = 0
        else:
            lambdaLine = (intersection.x - positionVector[0]) /
                     directionVector[0]
        intersectionHeight = positionVector[2] + lambdaLine *
                     directionVector[2]
        intersection = Point(intersection.x, intersection.y,
                     intersectionHeight)

        # calculate angle between observer and intersection point in xz-
          plane
        slopeIntersection = self.getSlopeXZ(observer, intersection)

        # angle of LOS must be higher than angle of observer-intersection
        if slope < slopeIntersection:
            return False
        else:
            return self.analyseVisibility(observer, los, slope, n, t)
```

```
    # find initial triangle in the r-tree
    def getInitialTriangle(self, observer):
        hits = self.idx.nearest((observer.x, observer.y, observer.x,
                                 observer.y))

        for h in hits:
            t = self.triangles[h]

            poly = Polygon(((float(t[0][0]), float(t[0][1]),
                     float(t[0][2])),
                            (float(t[1][0]), float(t[1][1]),
                     float(t[1][1])),
                            (float(t[2][0]), float(t[2][1]),
                     float(t[2][1])),
                            (float(t[0][0]), float(t[0][1]),
                     float(t[0][1]))))

            if observer.within(poly):
                return h

        return False

class DataRTree:
    idx = {}
    id = ""
    fileRoot = ""

    def __init__(self, id, fileRoot):
        self.id = id
        self.fileRoot = fileRoot

    # load r-tree
    def loadData(self, ):
        fileRTree = self.fileRoot+"RTree_%s" % self.id
        self.idx = Rtree(fileRTree)

    # Calculate slope of LOS
    def getSlope(self, observer, target):
        deltaY = observer.y-target.y
        deltaX = observer.x-target.x
        if deltaX == 0:
            print "deltaX == 0"
        else:
            slope = deltaY/deltaX
        return slope

    # Detects the possible lines in the given bbox
    def detectPossibleLines(self, xMin,yMin,xMax,yMax):
        # check if minima and maxima are correct
        if xMin > xMax:
            temp = xMin
            xMin = xMax
            xMax = temp
        if yMin > yMax:
            temp = yMin
            yMin = yMax
            yMax = temp
```

```
        possibleLines = {}
        # determine possible lines from r-tree
        hits = self.idx.intersection((xMin,yMin,xMax,yMax), objects=True)
        for h in hits:
            possibleLines[(h.bbox[0],h.bbox[1],h.bbox[2],h.bbox[3])] =
                     h.object
        return possibleLines

    # Parsing the line of sight in the given step size with the defined
      overlap factor
    def parseLOS(self, observer, target, los, step, factor):
        # Calculate slope of LOS
        slope = self.getSlope(observer, target)

        # Start at observer position
        xStart = observer.x
        yStart = observer.y

        possibleLines = {}

        finished = 0
        # if the slope is below 1, increase X
        if abs(slope) <= 1:
            # do until target is reached
            while finished == 0:
                # lines from right to left
                if observer.x > target.x:
                    xEnd = xStart - step
                    yEnd = yStart - (step * slope)
                    xNewStart = xStart - (step * factor)
                    yNewStart = yStart - (step * slope * factor)
                    if xEnd <= target.x:
                        xEnd = target.x
                        yEnd = target.y
                        finished = 1
                # lines from left to right
                else:
                    xEnd = xStart + step
                    yEnd = yStart + (step * slope)
                    xNewStart = xStart + (step * factor)
                    yNewStart = yStart + (step * slope * factor)
                    if xEnd >= target.x:
                        xEnd = target.x
                        yEnd = target.y
                        finished = 1

                possibleLines = self.detectPossibleLines(xStart,yStart
                                                         ,xEnd,yEnd)
                visible = self.analyseVisibility(observer, target, los,
                                         possibleLines)
                if visible == False:
                    return False

                xStart = xNewStart
                yStart = yNewStart
        # if the slope is above 1, increase Y
        else:
            # do until target is reached
            while finished == 0:
```

```
                # lines from top to bottom
                if observer.y > target.y:
                    yEnd = yStart - step
                    xEnd = xStart - (step * (1/slope))
                    yNewStart = yStart - (step * factor)
                    xNewStart = xStart - (step * (1/slope) * factor)
                    if yEnd <= target.y:
                        yEnd = target.y
                        xEnd = target.x
                        finished = 1
                # lines from bottom to top
                else:
                    yEnd = yStart + step
                    xEnd = xStart + (step * (1/slope))
                    yNewStart = yStart + (step * factor)
                    xNewStart = xStart + (step * (1/slope) * factor)
                    if yEnd >= target.y:
                        yEnd = target.y
                        xEnd = target.x
                        finished = 1

                possibleLines = self.detectPossibleLines(xStart,yStart
                                                         ,xEnd,yEnd)
                visible = self.analyseVisibility(observer, target, los,
                                                 possibleLines)
                if visible == False:
                    return False

                xStart = xNewStart
                yStart = yNewStart

        return True

    def getSlopeXZ(self, observer, target):
        deltaZ = observer.z-target.z
        deltaX = observer.x-target.x
        if deltaX == 0:
            print "deltaX == 0"
        else:
            slope = deltaZ/deltaX
        return slope

    # Analyses the visibility between observer and target
    def analyseVisibility(self, observer, target, los, possibleLines):
        # calculate the angle of the LOS
        slope = self.getSlopeXZ(observer, target)

        # calculate intersection points with LOS
        for possibleLine in possibleLines.values():
            intersection = Point()
            possibleLine = LineString(((possibleLine.GetPoint(0)[0],
               possibleLine.GetPoint(0)[1], possibleLine.GetPoint(0)[2]),
                                       (possibleLine.GetPoint(1)[0],
               possibleLine.GetPoint(1)[1], possibleLine.GetPoint(1)[2])))
            intersection = los.intersection(possibleLine)
            if intersection.wkt == "GEOMETRYCOLLECTION EMPTY":
                continue

            # interpolate the height from the 2 points of the line
```

```
            positionVector = possibleLine.coords[0]
            directionVector = (possibleLine.coords[1][0] –
                    positionVector[0],
                                possibleLine.coords[1][1] –
                    positionVector[1],
                                possibleLine.coords[1][2] –
                    positionVector[2])
            if directionVector[0] == 0:
                lambdaLine = 0
            else:
                lambdaLine = (intersection.x - positionVector[0]) /
                    directionVector[0]
            intersectionHeight = positionVector[2] + lambdaLine *
                    directionVector[2]
            intersection = Point(intersection.x, intersection.y,
                    intersectionHeight)

            # calculate angle between observer and intersection point in
              xz-plane
            slopeIntersection = self.getSlopeXZ(observer, intersection)

            # angle of LOS must be higher than angle of observer-
              intersection
            if slope < slopeIntersection:
                return False

        return True

class Request:
    topoData = {}
    rtreeData = {}
    fileRoot = ""

    def __init__(self, fileRoot):
        sys.setrecursionlimit(100000)
        self.fileRoot = fileRoot

        print "Loading data from given directory %s ..." % fileRoot

        # Lade alle Daten aus fileRoot
        for file in os.listdir(fileRoot):
            ##print file.__str__().split("topoNeighbors_").split(".pkl")[0]
            beginning = file.__str__().split("_")[0]
            end = file.__str__().split(".")[1]
            if beginning == "topoTriangles":
                    id = int(file.__str__().split("_")[1].split(".")[0])
                    print "      - now loading topological Data with ID %s"
% id
                    self.loadTopoData(id)
                    print "             *done*"
            if beginning == "RTree" and end == "dat":
                    id = int(file.__str__().split("_")[1].split(".")[0])
                    print "      - now loading RTree with ID %s" % id
                    self.loadRTree(id)
                    print "             *done*"

        print "Finished!\n"

    # load topological data
```

```
    def loadTopoData(self, id):
        topo = DataTopo(id, self.fileRoot)
        topo.loadData()
        self.topoData[id] = topo

        return "done"

    # load r-tree
    def loadRTree(self, id):
        tree = DataRTree(id, self.fileRoot)
        tree.loadData()
        self.rtreeData[id] = tree

        return "done"

    def getCrossedTrianglesVis(self, id, observerCoords, targetCoords):
        observer = Point(observerCoords[0], observerCoords[1],
                         observerCoords[2])
        target = Point(targetCoords[0], targetCoords[1], targetCoords[2])
        los = LineString((observer.coords[0], target.coords[0]))

        # get topological data
        data = self.topoData[id]

        # get initial triangle
        t = data.getInitialTriangle(observer)

        # calculate the angle of the LOS
        slope = data.getSlopeXZ(observer, target)

        n = data.crossedTrianglesVis(observer, los, slope, t, "", 0)

        return n

    def getCrossedTriangles(self, id, observerCoords, targetCoords):
        observer = Point(observerCoords[0], observerCoords[1],
                         observerCoords[2])
        target = Point(targetCoords[0], targetCoords[1], targetCoords[2])
        los = LineString((observer.coords[0], target.coords[0]))

        # get topological data
        data = self.topoData[id]

        # get initial triangle
        t = data.getInitialTriangle(observer)

        n = data.crossedTriangles(observer, los, t, "", 0)
        return n

    # analyse visibility on topological data
    def visTopo(self, id, observerCoords, targetCoords):
        observer = Point(observerCoords[0], observerCoords[1],
                         observerCoords[2])
        target = Point(targetCoords[0], targetCoords[1], targetCoords[2])
        los = LineString((observer.coords[0], target.coords[0]))

        # get topological data
        data = self.topoData[id]
```

```
        # get initial triangle
        t = data.getInitialTriangle(observer)

        # calculate the angle of the LOS
        slope = data.getSlopeXZ(observer, target)

        # analyse visibility
        visible = data.analyseVisibility(observer, los, slope, t, "")

        if visible:
            return True
        else:
            return False

    # analyse visibility on rtree
    def visRTree(self, id, observerCoords, targetCoords, step, factor):
        observer = Point(observerCoords[0], observerCoords[1],
                         observerCoords[2])
        target = Point(targetCoords[0], targetCoords[1], targetCoords[2])
        los = LineString((observer.coords[0], target.coords[0]))

        # get r-tree
        data = self.rtreeData[id]

        # Parse line of sight and analyse visibility
        visible = data.parseLOS(observer, target, los, step, factor)

        if visible:
            return True
        else:
            return False

server = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)
server.register_instance(Request(sys.argv[1]))
#/home/florian/svn/svnhillen/entwicklung/Thesis/src/
#/data/studgi09/userdata/

try:
    print "Server running ..."
    server.serve_forever()
except KeyboardInterrupt:
    print "Exiting"
```

Source code of the **creation of the topological data structure**:

```
#! /usr/bin/python

__author__="florian"
__date__ ="$13.03.2011 14:35:24$"

from osgeo import ogr
from time import *
from rtree import Rtree
import cPickle
import sys

# reads the shapefile and creates the triangles
def createTriangles(shpfile, fileTriangles):
```

```
    shp = ogr.Open(shpfile)
    lyr= shp.GetLayer(0)
    lyr.ResetReading()

    print "\nCreating triangles ..."
    start = time()

    triangles = {}

    iTriangles = 1
    # save points, lines and triangles
    for feature in lyr:
        if iTriangles % 100000 == 0:
            print "Status: %s triangles created" % iTriangles
            print "     Running since: %s secs" % (time()-start)

        geom = feature.GetGeometryRef()
        wkt = geom.ExportToWkt()

        parts = wkt.split("POLYGON ((")
        parts = parts[1].split("))")
        parts = parts[0].split(",")

        # get point coords
        coords1 = parts[0].split(" ")
        coords2 = parts[1].split(" ")
        coords3 = parts[2].split(" ")

        # save triangle
        triangles[iTriangles] = ((float(coords1[0]), float(coords1[1]),
                        float(coords1[2])),
                                 (float(coords2[0]), float(coords2[1]),
                        float(coords2[2])),
                                 (float(coords3[0]), float(coords3[1]),
                        float(coords3[2])))
        iTriangles+=1

    end = time()
    duration = end-start
    print "Complete! (In %f seconds)" % duration

    # Saving geometries
    print "Saving triangles ..."
    start = time()

    output = open(fileTriangles, 'wb')
    cPickle.dump(triangles, output)
    output.close()

    end = time()
    duration = end-start
    print "Complete! (In %f seconds)" % duration

def getEnvelope(triangle):
    xmin = 999999999999
    xmax = 0
    ymin = 999999999999
    ymax = 0
```

```
    for i in range(0,3):
        if triangle[i][0] > xmax:
            xmax = triangle[i][0]
        if triangle[i][0] < xmin:
            xmin = triangle[i][0]
        if triangle[i][1] > ymax:
            ymax = triangle[i][1]
        if triangle[i][1] < ymin:
            ymin = triangle[i][1]

    return xmin, ymin, xmax, ymax

# creates r-tree with the lines of the given tin
def createRTree(fileTriangles, fileRTree):
    print "\nCreating R-Tree ..."
    start = time()

    print "Reading Triangles ..."
    input = open(fileTriangles, 'rb')
    triangles = cPickle.load(input)
    input.close()
    print "     done in %s secs" % (time()-start)

    idx = Rtree(fileRTree)

    for i in range(1,len(triangles)+1):
        t = triangles[i]
        extent = getEnvelope(t)

        idx.add(i, (extent[0],extent[1],extent[2],extent[3]))

    end = time()
    duration = end-start
    print "Complete! (In %f seconds)" % duration

# builds the topology of the triangles
def createTopology(fileTriangles, fileRTree, fileNeighbors):
    # building topology
    print "\nBuilding topology ..."
    start = time()

    print "Reading Triangles ..."
    input = open(fileTriangles, 'rb')
    triangles = cPickle.load(input)
    input.close()
    print "     done! (%s secs)" % (time()-start)

    print "Load R-Tree ..."
    idx = Rtree(fileRTree)
    print "     done! (%s secs)" % (time()-start)

    print "Finding neighbors ..."
    neighbors = {}
    count = len(triangles)

    # empty topology
    for i in range(1,count+1):
        neighbors[i] = ("", "", "")
```

```
    for i in range(1,count+1):
        if i % 10000 == 0:
            percentage = (float(i)/float(count))*100
            print "Status: %s %s" % (percentage, "%")
            print "     Running since: %s secs" % (time()-start)

        # skip triangles whose neighbors are already defined
        if "" not in neighbors[i]:
            continue

        extent = getEnvelope(triangles[i])

        line1 = (triangles[i][0], triangles[i][1])
        line2 = (triangles[i][1], triangles[i][2])
        line3 = (triangles[i][2], triangles[i][0])

        hits = idx.intersection((extent[0]+0.1,extent[1]+0.1,
                       extent[2]+0.1,extent[3]+0.1))
        for h in hits:
            k = h
            v = triangles[h]

            if "" not in neighbors[k]:
                continue
            if (line1[0] in v) and (line1[1] in v) and not(k == i):
                neighbors[i] = (k, neighbors[i][1], neighbors[i][2])
                # set neighbors neighbor to self
                if (v[0] in line1) and (v[1] in line1):
                    neighbors[k] = (i, neighbors[k][1], neighbors[k][2])
                if (v[1] in line1) and (v[2] in line1):
                    neighbors[k] = (neighbors[k][0], i, neighbors[k][2])
                if (v[2] in line1) and (v[0] in line1):
                    neighbors[k] = (neighbors[k][0], neighbors[k][1], i)
                # abort if all neighbors are found
                if "" not in neighbors[i]:
                    break
            if (line2[0] in v) and (line2[1] in v) and not(k == i):
                neighbors[i] = (neighbors[i][0], k, neighbors[i][2])
                # set neighbors neighbor to self
                if (v[0] in line2) and (v[1] in line2):
                    neighbors[k] = (i, neighbors[k][1], neighbors[k][2])
                if (v[1] in line2) and (v[2] in line2):
                    neighbors[k] = (neighbors[k][0], i, neighbors[k][2])
                if (v[2] in line2) and (v[0] in line2):
                    neighbors[k] = (neighbors[k][0], neighbors[k][1], i)
                # abort if all neighbors are found
                if "" not in neighbors[i]:
                    break
            if (line3[0] in v) and (line3[1] in v) and not(k == i):
                neighbors[i] = (neighbors[i][0], neighbors[i][1], k)
                # set neighbors neighbor to self
                if (v[0] in line3) and (v[1] in line3):
                    neighbors[k] = (i, neighbors[k][1], neighbors[k][2])
                if (v[1] in line3) and (v[2] in line3):
                    neighbors[k] = (neighbors[k][0], i, neighbors[k][2])
                if (v[2] in line3) and (v[0] in line3):
                    neighbors[k] = (neighbors[k][0], neighbors[k][1], i)
                # abort if all neighbors are found
                if "" not in neighbors[i]:
```

```
                    break

    end = time()
    duration = end-start
    print "Complete! (In %f seconds)" % duration

    # saving neighbors
    print "Saving neighbors ..."
    start = time()

    output = open(fileNeighbors, 'wb')
    cPickle.dump(neighbors, output)
    output.close()

    end = time()
    duration = end-start
    print "Complete! (In %f seconds)" % duration

if __name__ == "__main__":
    print "### Creating topological datastructure ###"
    start = time()

    print sys.argv[0]
    if len(sys.argv) == 4:
        shpfile = sys.argv[1]
        fileRoot = sys.argv[2]
        id = sys.argv[3]
    else:
        shpfile = "/media/Daten/Studium/- MASTER -/Thesis/Daten/
           Triangulierung/ArcGIS Versuche/rastertin_tol2_TinTriangle.shp"
        #shpfile = "/media/Daten/Studium/- MASTER -/Thesis/Daten/
           Triangulierung/ArcGIS Versuche/rasttin_TinTriangle.shp"
        #shpfile = "/media/Daten/Studium/- MASTER -/Thesis/Daten/
           Triangulierung/ArcGIS Versuche/rastertin_tol15_subset.shp"
        fileRoot = "/home/florian/"
        id = "1001"

    fileTriangles = "%stopoTriangles_%s.pkl" % (fileRoot, id)
    fileNeighbors = "%stopoNeighbors_%s.pkl" % (fileRoot, id)
    fileRTree = "%stopoRTree_%s" % (fileRoot, id)

    # create Datastructure
    createTriangles(shpfile, fileTriangles)
    createRTree(fileTriangles, fileRTree)
    createTopology(fileTriangles, fileRTree, fileNeighbors)

    end = time()
    duration = end-start
    print "\n### COMPLETE! (In %f seconds) ###" % duration
```

Source code of the **PyWPS process to create the R-tree data structure**:

```
# -*- coding: utf-8 -*-

__author__="florian"
__date__ ="$16.03.2011 13:02:15$"

from pywps.Process.Process import WPSProcess
```

```
from osgeo import ogr
from rtree import Rtree
from time import *
import xmlrpclib
import random
import os

class Process(WPSProcess):

    def __init__(self):
        WPSProcess.__init__(self,

                identifier = "createRTree",
                title= "Create R-tree",
                abstract = """This process creates an R-tree index
                            needed for the visibility analysis.""",
                version = "1.0",
                storeSupported = True,
                statusSupported = True)

        self.tinShp = self.addComplexInput(identifier = "tinShp",
                            title = "Path to shp file of the TIN",
                            abstract = """Path to shp file of the TIN""",
                            maxmegabites = 150)

        self.tinShx = self.addComplexInput(identifier = "tinShx",
                            title = "Path to shx file of the TIN",
                            abstract = """Path to shx file of the TIN""",
                            maxmegabites = 150)

        self.id = self.addLiteralOutput(identifier = "id",
                            title = "ID of R-tree")

    # creates r-tree with the lines of the given tin
    def createRTree (self, shpfile, filename):
        shp = ogr.Open(shpfile)
        lyr= shp.GetLayer(0)
        lyr.ResetReading()

        idx = Rtree(filename)

        i = 0
        for feature in lyr:
            geom = feature.GetGeometryRef()
            wkt = geom.ExportToWkt()

            parts = wkt.split("POLYGON ((")
            parts = parts[1].split("))")
            parts = parts[0].split(",")
            line1 = ogr.Geometry(ogr.wkbLineString)
            line2 = ogr.Geometry(ogr.wkbLineString)
            line3 = ogr.Geometry(ogr.wkbLineString)

            coords = parts[0].split(" ")
            line1.AddPoint(float(coords[0]), float(coords[1]),
                      float(coords[2]))
            coords = parts[1].split(" ")
            line1.AddPoint(float(coords[0]), float(coords[1]),
                      float(coords[2]))
```

```
            line2.AddPoint(float(coords[0]), float(coords[1]),
                      float(coords[2]))
            coords = parts[2].split(" ")
            line2.AddPoint(float(coords[0]), float(coords[1]),
                      float(coords[2]))
            line3.AddPoint(float(coords[0]), float(coords[1]),
                      float(coords[2]))
            coords = parts[3].split(" ")
            line3.AddPoint(float(coords[0]), float(coords[1]),
                      float(coords[2]))

            extent1 = line1.GetEnvelope()
            extent2 = line2.GetEnvelope()
            extent3 = line3.GetEnvelope()

            idx.add(i, (extent1[0],extent1[2],extent1[1], extent1[3]),
                      obj=line1)
            i+=1
            idx.add(i, (extent2[0],extent2[2],extent2[1], extent2[3]),
                      obj=line2)
            i+=1
            idx.add(i, (extent3[0],extent3[2],extent3[1], extent3[3]),
                      obj=line3)
            i+=1

    def execute(self):
        print "### Creating R-Tree ###"
        start = time()

        fileRoot = "/data/"

        # create random id and check if the id was not used before
        id = random.randint(1, 1000000)
        fileRTree = "RTree_%s" % id
        while os.path.exists(fileRoot+fileRTree):
            id = random.randint(1, 1000000)
            fileRTree = "RTree_%s" % id

        os.rename(self.tinShp.value, "./tin_%s.shp" % id)
        os.rename(self.tinShx.value, "./tin_%s.shx" % id)
        path = "./tin_%s.shp" % id

        # create datastructure
        self.createRTree(path, fileRoot+fileRTree)

        server = xmlrpclib.ServerProxy('http://localhost:9000')
        server.loadRTree(id)

        self.id.setValue(id)

        end = time()
        duration = end-start
        print "### Complete! (In %f seconds) ###" % duration

        return
```

B Test data

A list of the **52 observer-target pairs:**

ID	Observer X	Observer Y	Observer Z	Target X	Target Y	Target Z
1	3434931.50	5794119.20	63.32	3434958.30	5794190.60	62.55
2	3434840.10	5794256.80	70.48	3434967.20	5794296.60	66.59
3	3435054.10	5794200.40	77.05	3435016.40	5794229.60	75.47
4	3435020.00	5794136.60	67.80	3434958.30	5794197.90	62.64
5	3435039.97	5792610.63	75.54	3434558.42	5792519.67	68.20
6	3432712.45	5794579.66	90.25	3432990.68	5794980.96	93.36
7	3436254.56	5793124.29	65.69	3435981.68	5793621.90	77.90
8	3436056.59	5795055.86	72.79	3435740.90	5795446.46	77.11
9	3436629.11	5795446.46	67.00	3436083.34	5795585.58	68.95
10	3432712.45	5792396.60	77.56	3432530.53	5792685.54	77.19
11	3434258.78	5793691.46	74.29	3434312.29	5794173.01	89.16
12	3434831.30	5793472.08	64.34	3435259.35	5793755.66	66.02
13	3432123.88	5793311.56	95.66	3433964.50	5792771.15	64.64
14	3433306.37	5793589.79	68.32	3433878.89	5794729.48	96.46
15	3432182.74	5794793.68	82.70	3434986.47	5795617.68	64.83
16	3433375.93	5792118.37	82.52	3433472.24	5795596.28	64.88
17	3437196.27	5793231.30	66.65	3432418.17	5794306.78	89.44
18	3433274.27	5794611.76	95.33	3433188.66	5794734.83	94.39
19	3433648.81	5795521.37	66.26	3433547.15	5795457.16	63.77
20	3433557.85	5795516.02	63.13	3433622.06	5795569.52	65.36
21	3434488.86	5793006.58	66.36	3434542.36	5793300.86	65.89
22	3432220.19	5792733.69	78.86	3432284.40	5792856.76	77.58
23	3433017.44	5794563.61	97.51	3434424.65	5793728.91	74.87
24	3436275.97	5793220.60	66.13	3435152.34	5793509.53	64.43
25	3436447.19	5794392.39	71.81	3435794.41	5793803.82	66.54
26	3435938.88	5794788.33	65.27	3435478.72	5794360.28	65.43
27	3433397.33	5793006.58	65.77	3433857.48	5793552.34	63.69
28	3432985.33	5794028.54	74.26	3433070.94	5793958.99	71.95
29	3432958.58	5794890.00	98.36	3432953.23	5794777.63	101.39
30	3434831.30	5793691.46	79.96	3434772.44	5793782.42	68.91
31	3434456.75	5793991.09	82.15	3434563.77	5794039.25	67.94
32	3435232.59	5793910.83	73.06	3435125.58	5793782.42	67.86
33	3435526.88	5793584.44	69.16	3435564.33	5793493.48	66.60
34	3435248.65	5793167.09	64.94	3435237.95	5793236.65	72.39
35	3434686.83	5793161.74	63.92	3434788.49	5793215.25	79.80
36	3434900.86	5793268.76	68.80	3434788.49	5793349.02	80.36
37	3434558.42	5794183.71	72.95	3434537.01	5794231.87	68.22

38	3436083.34	5793242.00	66.37	3436120.80	5793338.31	71.63
39	3432621.49	5793498.83	80.88	3432664.30	5793755.66	76.91
40	3434879.45	5794440.54	64.42	3434435.35	5794510.10	66.84
41	3436704.02	5795221.73	67.03	3435414.52	5793696.81	87.66
42	3433188.66	5794938.15	79.42	3434232.03	5794087.40	77.59
43	3434847.35	5795093.32	77.06	3435109.53	5795168.23	106.06
44	3435023.92	5795173.58	117.54	3434997.17	5795398.30	73.93
45	3435307.50	5792268.19	77.60	3435462.67	5792407.31	67.72
46	3434039.41	5792177.23	72.64	3433959.15	5792380.55	66.42
47	3436602.35	5792546.42	67.94	3436778.92	5792316.34	73.45
48	3432134.58	5793782.42	90.27	3432150.64	5794049.95	85.26
49	3432016.87	5795569.52	72.88	3436987.60	5792391.25	67.99
50	3432113.18	5795120.07	91.05	3432188.09	5795296.64	91.03
51	3434911.56	5795542.77	66.24	3433991.25	5793739.61	65.34
52	3433739.77	5792632.03	65.67	3433702.32	5792792.55	70.40

Tab. 6: The X, Y and Z coordinates of the 52 observer-target pairs.