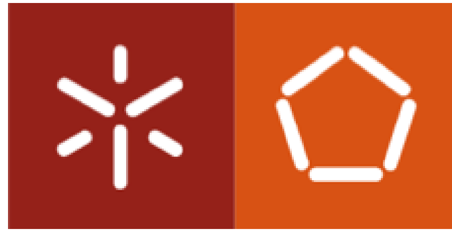


Universidade do Minho
Escola de Engenharia

Nuno Miguel Carvalho Oliveira

Web 3D Service
An Open Source Implementation

Dissertação de Mestrado

Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação do

Professor Jorge Gustavo Rocha

Outubro de 2013

Web 3D Service
An Open Source Implementation

Dissertation

Nuno Miguel Carvalho Oliveira
(PG18391)

Universidade do Minho

October, 2013

Acknowledgements

I would like to express my gratitude to my advisor, Prof Jorge Rocha, for his excellent guidance, dedication and to have introduced me the GIS field.

Besides my advisor, I would like to thank Paulo Machado for always give his best suggestions and provide the best uses cases for W3DS.

I would also like to thank my family. They were always supporting and encouraging me.

Finally, I would like to thank GeoServer team for their dedication to open source and their support.

Abstract

Geographic Information Systems (GIS) represents some of the most interesting challenges for software engineering of our time. The data volume, the complexity and the critical use cases demands expertise in different computer science domains, besides the knowledge of some geography principles.

The development of standards related to geographic information representation and its manipulation, either *de facto* or *de jure* standards is crucial to the development of large scale GIS applications.

By default georeferenced data have been represented in a two dimensional plane for simplicity. Nowadays the evolution of graphics hardware and the emergence of technologies like WebGL give us the necessary support to make 3D GIS possible. But newer and more sophisticated GPUs are not enough to support mature 3D applications. Standards related to 3D representation and manipulation are necessary.

In this work we analyze the Open Geospatial Consortium Web 3D Service draft specification and provide an open source implementation for it.

Contents

1 Introduction **8**

1.1 Motivation 9

1.2 Goals 11

1.3 Dissertation Structure 12

2 State Of The Art **13**

2.1 3D GIS 14

2.1.1 Web 3D Evolution 16

2.1.2 Web 3D GIS Clients 17

2.2 Web Services Architectures 19

2.2.1 OGC Web Services 20

2.2.2 Map Servers 22

3 Web 3D Service **24**

3.1 GetCapabilities 25

3.2 GetScene 27

3.3 GetFeatureInfo 31

3.4 GetTile 32

3.5 Styling 34

4 Architecture and Implementation **36**

4.1 Base Framework 37

4.2 Java Technologies 38
4.2.1 Maven 39
4.2.2 Spring Framework 46
4.3 Architecture 51
4.4 Implementation 54
4.4.1 Service 54
4.4.2 Types 56
4.4.3 Styles 57
4.4.4 Responses 59
4.4.5 Web 60

5 Results and Evaluation 62
5.1 Use Case 62
5.2 Dataset Preparation 64
5.2.1 Terrain Preparation 64
5.2.2 Preparation of 3D Features 67
5.3 GeoServer Configuration 68
5.4 GetCapabilities Request 70
5.5 GetTile Request 72
5.6 GetScene Request 73
5.6.1 X3D 73
5.6.2 KML 77

6 Conclusion 80
6.1 Publications 81
6.2 Future work 81

List of Figures

2.1 3D Computer Aided Design (CAD) models that represent the 3D entities and the Digital Terrain Model (DTM), as proposed by Cambray in 1993 (*Source [6]*). 15
2.2 Types of clients as described on Web 3D Service (W3DS) specification (*Source [31]*). 19

3.1 Example of 3D styles (*Source [12]*). 35

4.1 Maven clean and default life cycles and its build phases. 44
4.2 Simplified description of the Dispatcher and the W3DS components. . . . 53
4.3 Print screen of W3DS service configuration page. 60
4.4 Print screen of W3DS layer configuration page. 61

5.1 Architecture of the use case. 63
5.2 Cutting tiles. 65
5.3 Perfect composition of several tiles, at different resolutions. 67
5.4 Comparison between an infrastructure and is Collada model. 68

6.1 X3D models renders by X3DOM 82

List of Tables

3.1 W3DS operations. 25
3.2 W3DS *GetScene* operation parameters. 28
3.3 *GetFeatureInfo* operation parameters. 32
3.4 *GetTile* operation parameters. 33

Listings

3.1 KVP encoded GetCapabilities for GET request. 25
3.2 Example of a TileSet element. 27
3.3 KVP encoded GetScene for GET request. 30
3.4 KVP encoded GetFeatureInfo for GET request. 32
3.5 XML encoded GetTile for POST request. 34
4.1 Multi-module *pom* that aggregate some data base drivers. 41
4.2 Simple *pom* example. 42
4.3 Partial snapshot of Hibernate dependency tree produced by `mvn dependency:tree` command. 46
4.4 Spring bean XML definition. 49
4.5 Spring beans related to the W3DS *service* component 55

4.6 Spring beans related to the W3DS *types* component. 57

4.7 Example of the inclusion of a 3D model using extended SLD. 58

4.8 W3DS response formats registered as Spring beans. 59

4.9 W3DS Apache Wicket components registered as Spring beans. 61

5.1 W3DS service meta-data. 70

5.2 *GetCapabilities* operation meta-data. 70

5.3 Description of a W3DS tiled layer. 71

List of Acronyms

GIS Geographic Information Systems

GPS Global Position System

VRML Virtual Reality Modeling Language

OGC Open Geospatial Consortium

KML Keyhole Markup Language

GML Geography Markup Language

WMS Web Map Service

WFS Web Feature Service

WCS Web Coverage Service

WPS Web Processing Service

SLD Styled Layer Descriptor

XML Extensible Markup Language

W3DS Web 3D Service

X3D Extensible 3D

CAD Computer Aided Design

DTM Digital Terrain Model

LOD Level Of Detail

SQL Structured Query Language

WPVS Web Perspective View Service

SRTM Shuttle Radar Topography Mission

JTS Java Topology Suite

POM Project Object Model

OWS Open Web Service

NRW North Rhine-Westphalia

SDI Spatial Data Infrastructure

OWS OGC Web Service

CRS Coordinate Reference System

EPSG European Petroleum Survey Group

KVP Keyword Value Pair

MIME Multipurpose Internet Mail Extensions

JAK Java Api For KML

ORM OGC Reference Model

CGI Common Gateway Interface

WTS Web Terrain Service

GDAL Geospatial Data Abstraction Library

CGAL Computational Geometry Algorithms Library

Chapter 1

Introduction

Is difficult to define a precise bird date to Geographic Information Systems (GIS). In the late 1960s the world saw the development of the *Canadian Geographic Information System*, which is one of the earliest GIS developed. This was a direct consequence of the computer hardware development. Since, the development of more powerful GIS and the evolution of computer science have always work side by side.

At the beginning, applications so complex as GIS was only supported by computers which have the size of a truck. Typically, GIS applications were used by big companies with very specific use cases or by governments. The cost and complexity of maintaining such systems was so high, that they only were used when the answer to the question *Where is what ?*, was not practical but fundamental. Nowadays we have GIS applications an hundred times more powerful which run on common devices, that can be carried on or pocket. This evolution didn't only affect how practical they became but radically changes the way we are using it.

One of the most important aspect of every GIS is who we represents the data, or in other words, which kind of data he can handle. The fact is, that an evolution on the way we represent the data is also the sign of a big change on the potentiality of GIS systems. The first big jump was given from hand drawn maps to digital values stored on disk. In the 1970s, this leads to the automated map drafting, were digital stored

values are converted in sets of x y coordinates and draw by plotters. The next step have been the representation of geographical data in a vector format. Were map features are represented as sets of points, lines or polygons. At the same time, the raster data model also start to be used. This one, represents the data as a grid over a projected area and store the values from each cell.

In the current days, vectors and raster models are still the most used way to storing GIS data. Which data structure to use, is determined by the nature of the data and the kind of processing we want to perform. This leads to one duality in map visualization. In the current days when we look at map in most of the cases what we see is a mixture of vector and raster data.

With a new way to represent the data, the quantity of available GIS information start to grow. No matter which kind of model we use to represent our data, it needs to be stored and accessed. In meantime GIS market has become attractive for software companies, that start building a big among of systems who needed to be feed with GIS data. The increasing demands from customers, companies and organizations for interoperability force them addressing the needs of standards.

1.1 Motivation

Complex GIS software likes the ones used in city management are the result of the interaction of several organizations. Such interoperability is guaranteed by the respect of standards [17]. Most of them developed by Open Geospatial Consortium (OGC), an international consortium who develop and promotes the use of open standards in geographical information. Along the years OGC has produced a large number of standards that falls on two big categories: formats and services.

Some of OGC most know and used formats, are Keyhole Markup Language (KML) [13] and Geography Markup Language (GML) [11]. Both of them can be used two express different kind of vectorial data and is meta-data, they also support 3D information. The

Web Map Service (WMS) [10] and Web Feature Service (WFS) [15] are certainly the most OGC used services.

The WMS service serve georeferenced images produced by the server who uses GIS data. The produced images are a mixture of vector and raster layers, which can represent any kind of information. Frequently WMS is associated in another OGC standard, the Styled Layer Descriptor (SLD) [12] who's principal function is to describe a layer for the rendering process of a WMS. SLDs gives a way to customize the final appearance of the produced map.

Where WMS shows a representation of the GIS data hold by the server WFS give us the way to access and edit the vectorial data. WFS can use a variety of formats to encoding the data, however the most used are the Extensible Markup Language (XML) based ones, like GML. Due to the increasing number of web based GIS the GeoJSON format, which is more JavaScript friendly, have been widely adopted by GIS community. However, GeoJSON is not an OGC format.

Most of the actual GIS represents the information in two dimensions. This abstraction of the real world, forces the user to mentally translate what we see on the map before using it. In some uses cases, for example meteorology, geology and architecture, we cannot reflect, analyze or even display the relevant information precisely. With a 3D visualization some of that process become simplified and intuitive.

City administration have become one of the top use cases for 3D GIS ([21], [37], [25] and [24]), some of their operations needs precise and comprehensive knowledge about the all urban space. For example, frequently we need to simulate emergency situations, in 2D visualization this is made analytically and the results are traduced to a map and finally interpreted. In 3D we can in real-time see our simulations and directly see the results whit an extra accuracy. In other situations, like infrastructures management, we need more interaction whit the features that the one provided by the 2D mapping.

The idea of 3D mapping is not new, in 1997 [38] presents a web 3D GIS that use Virtual Reality Modeling Language (VRML) [29] to represent the 3D data. But, like other 3D

GIS, they have facing a major problem: the cost of rendering complex georeferenced 3D scenes. Nowadays, the constant evolution of graphical hardware, have provided common devices whit an enormous capacity of 3D rendering. Unfortunately, the techniques and environments needed to use that capabilities are not very friendly whit the new generation of GIS, who in the meantime have mostly become web applications.

Luckily the emergence of technologies such as WebGL [23], have definitely open the doors of 3D capabilities to modern GIS. In consequence, a big number of GIS using 3D visualization have appeared in the last years. However, with no standards to guaranty interoperability and maintainability, most of the examples we found can only see as show-off applications that can't be deployed in real scenarios. In order to fight this chaotic situation we need standards that fully supports the needs of 3D GIS.

The OGC Web 3D Service (W3DS) specification draft is the most advanced candidate standard related with 3D GIS [31]. W3DS is a portrayal service where 3D GIS data is delivered encoded in a format that can be interpreted by a 3D client. W3DS is similar to WMS, cause both provide a view over the data. However, the result of WMS request (e.i. an image) can be seen everywhere, but the result of W3DS requires a client with extra capabilities. The W3DS referenced format specification to encode the result of a request is Extensible 3D (X3D) [5]. Georeferenced three-dimensional scenes encoded in X3D can be directly included in HTML5 pages and natively view in every browser which supports WebGL. At this moment, no W3DS open-source implementation is available.

1.2 Goals

The main goal of this work is to make an open-source implementation of a W3DS. Such implementation will follow the most recent W3DS specification , version 0.4.1.

To accomplish such goal, an open source map server will be used to take advantage of all logic already developed.

To illustrate the W3DS usage, an use case is provided, with several 2.5D layers.

1.3 Dissertation Structure

The first chapter starts with an overview of GIS systems and its technological evolution, representing the main motivations to develop this work. 3D, OGC formats and services related concepts are also introduced. The interoperability importance is highlighted and serves as motivation for the goal of this work.

The second chapter presents the state of the art, approaching two major topics: 3D GIS and OGC web services architecture description. It will be explored the 3D Geographic Information Systems (GIS) field presenting some of the most interesting works in that area. A brief overview about 3D web technologies and 3D GIS web clients will also be presented. It will be discussed the OGC web services architecture description where presenting the three majors open source map servers: MapServer, Deegree and GeoServer.

On the third chapter the W3DS standard is discussed. It starts by an overview of the 3D visualization pipeline. The four operations of the service: *GetCapabilities*, *GetScene*, *GetFeatureInfo* and *GetTile* will be described.

The fourth chapter is dedicated to our open source implementation and architecture of the W3DS. It introduces our base framework, i.e. GeoServer, and related technologies. Special attention is given to Maven and Spring Framework since they are responsible for the flexibility and extensibility of GeoServer. We also present the main components of our implementation and its integration in the GeoServer architecture.

On the fifth chapter we present the W3DS service in action, based on the management of telecommunications infrastructures. The data preparation is also described. Special focus is given to 3D tiled terrains.

Some concluding remarks are presented in the last chapter. Future work is discussed that results from the feedback received by users.

Chapter 2

State Of The Art

The Open Geospatial Consortium (OGC) Web 3D Service (W3DS) standard is the central component of this work. It stands on two major fields, the 3D Geographic Information Systems (GIS) for obvious reasons and because it is a OGC service it is connected to the web service architecture view of OGC. The full stack required by a 3D GIS application includes concepts from both of those fields.

Geographic Information Systems (GIS) community have been talking about 3D GIS from several years. A lot of prototypes have been made but they failed in dressing a standard pipeline for 3D GIS applications. The lack of standards to provide interoperability between the stages of the pipeline have limited the prototypes to the use case for what they have be made.

Most of the modern GIS have migrated to the web. A recent 3D GIS application will need to provide a web based 3D visualization. In the 90's several 3D web based applications failed not because the lack of an appropriate 3D web standard but due to technologies issues. Nowadays recent technologies like WebGL definitively open the doors for 3D web applications.

In the enterprise view of OGC interoperability is essential. Web services are key elements in that interoperability model. OGC promote several open standards that define web services, like the Web Map Service (WMS) or the Web Feature Service (WFS). Most

of that services are implemented by several proprietary and open source map servers.

Open source and open standards are two different concepts that are frequently confused. The main goal of an open standard is to guarantee interoperability between different systems regardless how they are implemented. Open source is about free software that are available under a license that give to users a total control over the software including is source code.

Most of the reference implementations for open standards are made by open source projects. Open source map servers like GeoServer or Deegree are the reference implementation for several OGC standards. Is wise to implement the W3DS on top of one of this servers.

In the first section of this chapter we will explore the 3D GIS field. We will present some of the most interesting works in that area. A brief overview about 3D web technologies and 3D GIS web clients will also be presented. The second part of this state of the art will inside on the OGC web services architecture description where we will present three major open source map servers: MapServer, Deegree and GeoServer.

2.1 3D GIS

When we take a closer look at the evolution of 3D GIS, what is interesting is that they are not a new concept. In 1993 [6] presents a 3D GIS that use Computer Aided Design (CAD) models to represent the 3D entities and the Digital Terrain Model (DTM) (*Figure 2.1*). The models have three different kinds of approximation, the first two are used to index and accelerate the rendering process and the last one is a detailed representation of the 3D model.

In 1997 [38] presents one of the first 3D Web GIS. The HTML pages were produced dynamically and the 3D data was directly retrieved from the database using Structured Query Language (SQL). The produced 3D scenes are encoded in Virtual Reality Modeling Language (VRML) which can be interpreted by the browser VRML plug-in. The reference

[7] gathers some interesting publications about 3D GIS, that can be seen as the state of the art of 90s.

Most of the recent works inside on city administration, which have become one of the top use cases for 3D GIS ([21], [37], [25] and [24]). The problem is that most of that 3D GIS projects are very specific for the use case for which they were developed. Although they still interesting projects, most of them fail in dressing a stack of standards and technologies for developing 3D GIS applications.

Pilot 3D was one of the first projects where the authors worry about interoperable 3D visualization of GIS data [2]. *Pilot 3D* result from the initiative Geodata Infrastructure North Rhine-Westphalia (GDI-NRW). Another result from the GDI-NRW was the first draft of W3DS. The reference [3] presents a 3D Spatial Data Infrastructure (SDI) for the city of Heidelberg where W3DS is a central piece. This work was the first and still one of the most complete attempts in defining a 3D SDI based on open standards.

Nowadays, OSM-3D is one of the most ambitious and interesting project related to 3D GIS. Is main objective is to provide a 3D view of OpenStreetMap data integrated with the elevation data provided by the Shuttle Radar Topography Mission (SRTM). Is implementation is made on top of OGC standards, including W3DS for 3D visualization. The reference [19] makes an overview about the current state of OSM-3D in Germany and provides a good discussion about the generation of 3D building models.

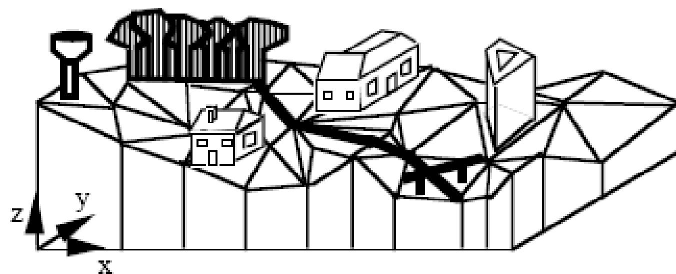


Figure 2.1: 3D CAD models that represent the 3D entities and the DTM, as proposed by Cambray in 1993 (*Source [6]*).

2.1.1 Web 3D Evolution

VRML was the first web based 3D format, released in 1995 and ISO certified in 1997. The main goal of VRML was to give a way to represent 3D virtual worlds that can be integrated on web pages. A VRML scene is composed by geometric primitives like points, segments and polygons. The scene may also include multimedia content like hyperlinks, images, sounds and videos. The aspect can be customized using lights effects and defining materials properties. VRML scenes can be explored in desktop software or in web browsers, using some compatible plug-in. The reference [29] make a good overview about the format.

In 2001, the Web3D Consortium, who have become the main supporter of VRML, releases Extensible 3D (X3D) a Extensible Markup Language (XML) encoding version of VRML ([5]). The XML based encoding of X3D make him more suitable to be natively integrated in HTML pages. X3D also add new features like the support of shaders, better events handling, new geometric primitives and others short cuts for 3D rendering. X3D brings up the concept of working groups, their job is to extends X3D to custom support of certain areas, like medicine, GIS and CAD. The GIS working group have provided X3D with the capability to natively support the needs of GIS applications. The main features are the full support of georeferenced coordinates and custom events for geographical scenes.

Even if at this time, VRML and X3D stay the most used web based 3D formats, their use decreases significantly when compared to ten years ago. When VRML was released everyone have tried to make use of it, quickly we have seen the appearance of 3D web content everywhere. Some companies have invest large quantities of money to move their websites to 3D. The same thing happened to GIS applications, we have assist to a massif jump from 2D to 3D, companies and governments have even start buying 3D georeferenced data. The problem was that technology don't follow that movement. Computers with the capability of rendering complex 3D scenes at acceptable frames rates were not common and those that existed were too expensive. With the poor quality offered by 3D web and

passed the new sensation of a third dimension, people start looking again to a 2D web.

Around 2009 appears WebGL and the doors to a 3D web are definitively open ([28]). WebGL specification is based on OpenGL ES 2. Even if it is only a draft, it already have been implemented by the majors web browsers and plugins have been provided for those that don't support natively WebGL ([23]). WebGL gives us the possibility to use 3D hardware acceleration from the JavaScript of web pages, like OpenGL does for desktop 3D applications. It's integration with HTML5 give the possibility to directly embed on web pages complex interactive 3D scenes.

Recently a new web based 3D format is being adopted: XML3D. This is the only major web 3D format that is not supported by Web3D Consortium, however is a candidate to become a WC3 standard. Unlike the others formats, the main goal of XML3D is to be an extension to HTML5 specification ([33]). The authors claim that even if using X3D or VRML we can integrate 3D content to a web page, the separation of the two concepts is to well defined. On his side XML3D definition is based on others successful standards of W3C like HTML, DOM and CSS. All the interactions with the 3D scenes are made using the web standard way, i.e. using DOM events and JavaScript. XML3D is independent of the 3D rendering API used, in [34] the authors use a modified version of Chromium Browser that use OpenGL, however the top rendering technology for XML3D is WebGL.

2.1.2 Web 3D GIS Clients

The result of a *GetScene* request from a W3DS is a 3D scene encoded in a specific format that need to be interpreted and rendered by a graphical engine. Such requirement requires clients with extra capabilities. The W3DS specification [31] identifies three types of clients and classify them according to they complexity degree: thick, medium and thin. In *Figure 2.2* we can see the balancing scheme between the client and server for every type of client.

The pipeline for 3D GIS data visualization is composed of four major steps: data access, scene preparation, rendering and visualization. For every type of client the that

data is accessed on server and visualized on the client side. A thin server like WFS require a very powerful client capable a generate 3D content from 2D data. On the opposite side a thick server like Web Perspective View Service (WPVS) [20] would be responsible to render the 3D scene from a certain perspective and send only that frame to the client. On the middle we have W3DS were the server side produce the 3D scene and the client is responsible for rendering and visualization.

The most well know and used 3D client is certainly Google Earth which is capable of handling 3D scenes encoded in Keyhole Markup Language (KML) and works on most of the platforms (windows, linux, android and a web plugin is also available). Google Earth belongs also to a particular subset of clients: 3D globes, who have become the most common and intuitive way to visualize and interact whit 3D GIS data.

In most of the cases, a georeferenced 3D scene to appear in a familiar spatial context requires at least two other components: a sky and a DTM. That integration is not always trivial, mostly the positioning on the DTM, 3D globes give that for free. The references [35],[18] and [32] gives a good overview about the fundamentals and challenges behind 3D globes.

Lately several globes have emerged from different sources, however most of them only serve as proof of concept without capabilities to support real use cases. One of the most promising ones is OpenWebGlobe, which already showed the capabilities to support real applications needs and implement some high level features like point cloud streaming support ([22]).

For some use cases using a 3D globe to represent georeferenced 3D scenes is not suitable, in that situations we need libraries that provide mechanisms to facilitate the makeover of custom clients, like OpenLayers does for 2D web mapping. If we follow the advice of W3DS specification and encode our scenes in X3D the X3DOM library will be indispensable [4].

X3DOM appears around 2009 and is development is mainly supported by Fraunhofer IGD, is main objective is to provide an integration between HTML5 and X3D. WebGL or

Flash can be used as back-end rendering API, however Flash is only used when WebGL is not available. X3DOM helps the development of custom clients providing a intuitive way to integrate 3D scenes to web pages and interact with them.

From non web based clients the library OsgEarth, build on top of Open Scene Graph, provides a good set of functionalities. OpenLayers3 is currently being developed and if it reach is objectives it will certainly become on of the most useful library for build 3D web GIS clients.

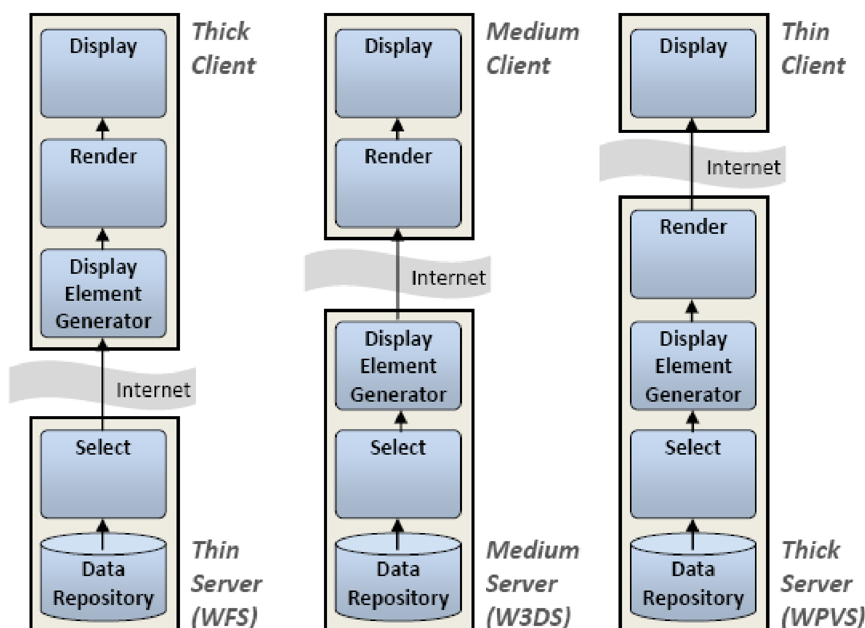


Figure 2.2: Types of clients as described on W3DS specification (*Source [31]*).

2.2 Web Services Architectures

The OGC Reference Model (ORM) [17] introduce some OGC standards and discuss why they are fundamental to modern GIS applications. It also provides an overview about the different types of architectures that can be build on the top of that standards. Most of that architectures are web based and are closely related with others open standards promoted by open organizations like the World Wide Web Consortium (W3C).

Independently of the used open standards or architectures, OGC claims that the key to

successful GIS applications is interoperability. The non-interoperability leads to complex GIS applications with limited resources and an high cost of development. Interoperability guarantee the share of data and computation resources between different organizations. ORM references the [8] and [9] whites papers that provide a interesting discussion about interoperability.

The WMS and the WFS are good examples of interoperability achieved through open standards based on web architectures. Since the development of WMS and WFS a considerable amount of other web based services have been developed by OGC. Implementations of OGC web services tend to be grouped in map servers. It is a wise approach, since common aspects of the implementation are only implemented once. We also benefit in having a common architecture for a group of services.

2.2.1 OGC Web Services

OGC interoperability is mainly based on service oriented architectures. OGC services have a well defined interface that is described by a standard. A service will receive requests and use its parameters to execute operations. In most of the cases a result or an exception message will be returned. Services can be chained, i.e. a service can invoke others services and is execution can be distributed. With this approach we reach an high decoupling between the client and the server side implementation. The reference [17] provide a good overview about OGC service architectures.

Most OGC services act as high level APIs to access data sets in a variety of ways, like WMS or W3DS service for example. Other services, like the Web Processing Service (WPS), are used to make powerful computations over large data sets. In the document [16] OGC introduces the OGC Web Service (OWS), which is a service interface that define some properties and guidelines that should be implemented by others OGC services. OWS interface standard focus on three major common aspects:

- Operations and request contents, i.e. common operations that should be implemented by all OGC services.

- Parameters and data structures included in operation requests and responses that are common to all operations.

- Extensible Markup Language (XML) and Keyword Value Pair (KVP) encoding of operation requests and responses.

Currently only a common operation is defined, the *GetCapabilities* operation. This operation is mandatory in every service that implement the OWS interface. The main goal of this operation is to provide information about the service and the available content. Service information include the available operations description and metadata about the service. The available content will change from a service to another. For example, the available content returned by a *GetCapabilities* operation from a WMS will contain information about the existing layers and the available styles.

Three mandatory parameters common to all services that implement OWS interface are defined: *service*, *version* and *request*. The *service* parameter identifies the service which the request is destined. Note that frequently the URL path already reference the service, but the *service* parameter should always be used to identify the service regardless to the URL. The *version* parameter identifies the version of the service that should be used. Generally map servers are very permissible with this parameter, if the provided versions is invalid or not available the most recent version existing on the server will be used. The *request* parameter identifies the service operation that should be executed, regardless to others parameters or anything else.

Other optional parameters that frequently appear are also described. One of them is the *crs* which represents a coordinate reference system in the European Petroleum Survey Group (EPSG) form, for example `EPSG:4326`. Another frequent parameter is the *bbox* which represents a referenced bounding box and match this format `minX,minY,maxX,maxY`. Sometimes a parameter is associated to a list of items, for example the WMS style parameter. Lists are represented as sequence of items separated be a coma, like `style1,style2,...` for example.

The HTTP protocol supports two request methods: POST and GET. A service that implement the OWS interface must at least provide one of this methods, both can also be provided. GET requests should be encoded in KVP. A GET request URL is composed of the URL path followed by a question mark `?` and ended with a list of server-specific parameters of the form `name=value&`. In a POST request the parameters are transmitted in the body of the message that can be encoded as XML document formatted as specified by one or more XML schemas.

2.2.2 Map Servers

A map server is a web architecture based software that normally implement several OGC services. This are the common aspects of every map server. However, as they evolve as a software project they tend to favor some technologies and some standards in favour of others. These differences with the time become stronger and produce a set of map servers that provide the same core functionalities but in a different way.

This differences affects both user and developers in different ways. For most of the users the main differences will be related with software usability. For example, some map servers provide a graphical user interface and others not. Developers are concerned with the core architecture and the technology stack. Working in a project written in C++ or in a project written in Java involves a lot of differences.

MapServer, Deegree and GeoServer are three major map servers. Their success is mainly due to the enormous quantity of standards they support, their extensible architectures and the diversity of the community behind them, which involve some of the major players in the GIS field. However, conceptually their are very different. They take different approaches on several fundamentals aspects of their architectures. Although, the OGC standards on which they relies guarantee the interoperability between them.

MapServer is an open source set of tools that can be used to produce several types of spatial applications. It was originally developed by the university of Minnesota for the ForNet project in cooperation with NASA and Minnesota Department of Natural

Resources (MNDNR). The core of MapServer is write in C but it provides bindings for other languages like Java, .NET, Python, Ruby, PHP and Perl.

In is most reduced form MapServer can be seen as a rendering engine for maps. It can be installed as Common Gateway Interface (CGI) script that will interpret the request parameters and produce a map based on a MapFile. For most advanced use cases it can be used to make available OGC services like WMS or WFS. MapServer can be extended in a variety of ways and also be used as an API by other programming languages.

Deegree is an open source project written in Java developed by lat/lon. It implements a great number of OGC standards and is the reference implementation for some of them. It also implements some of the new 3D OGC candidates services like the WPVS. Deegree provides the components necessary to build a Spatial Data Infrastructure (SDI) in a modular way.

Deegree is composed of five major products. The Deegree web services which contains the implementations of the OGC services. The iGeoPortal which is the web based portal framework of Deegree project. The iGeoSecurity that contains security related components. The iGeo3D product who is related with the storage and visualization of 3D geodata. The iGeo3D product doesn't includes W3DS. The iGeoDesktop that represent the SDI desktop GIS of Deegree.

GeoServer is an open source geospatial driven server written in Java. The heart of GeoServer is interoperability. It reads data from the major spatial data sources and publishes it using OGC services. GeoServer is the reference for the OGC WFS and WCS services. It also a high performance certified compliant WMS implementation.

GeoServer is built on the top of recent Java technologies like Maven and Spring Framework. It includes in is software stack other import geospatial libraries like Java Topology Suite (JTS) and GeoTools. GeoServer have a modular architecture that can be easily extended. It is supported by a powerful community and have clear policies related to the development process.

Chapter 3

Web 3D Service

The Web 3D Service (W3DS) is a portrayal service proposal for three-dimensional spatial data. The first proposal was presented back in 2005 by Kolbe and Quadt [30]. Since then, some improvements were integrated. In 2009, version 0.4.0 was accepted as public discussion paper by the Open Geospatial Consortium (OGC). Afterwards, a version 0.4.1 was rewritten. This is the last version available, and it dates from 2011 [31].

The W3DS service delivers scenes, which are composed by display elements representing real world features. It does not provide the raw spatial data with attributes, like the Web Feature Service (WFS) service does. It only provides a view over the data, accordingly, for example, to the level of detail.

Unlike the Web Map Service (WMS), it does not provide rendered images. It filters the data to be delivered according to several parameters, like a bounding box, but the result will be a graph of nodes with properties attached to each node, like shapes, materials and geometric transformations.

This graph of display elements must be handled by the client. W3DS clients must implement the necessary logic to take advantage of the W3DS operations. Typically, clients will continuously request scenes from the service, trying to minimize the data delivered to the client, while providing the best user experience. All four proposed operations, tagged as mandatory or optional are listed in *Table 3.1*.

Operation	**Use**
GetCapabilities	Mandatory
GetScene	Mandatory
GetFeatureInfo	Optional
GetTile	Optional

Table 3.1: W3DS operations.

Like other OGC services, information about the service, the supported operations, available layers and their properties, can be retrieved using the *GetCapabilities* operation. The *GetFeatureInfo* operation returns information about the features and their attributes. In the previous versions of the standard a *GetLayerInfo* operation was also available, it possible to obtain information about the layer.

Two operations are provided to return 3D data: *GetScene* and *GetTile*. These two operations differ essentially in how the features are selected. *GetScene* allows the definition of an arbitrary rectangular box to spatially filter the features to compose the scene returned to the client. *GetTile* returns a scene on-the-fly formed by features within a specific delimited cell, within a well-defined grid.

3.1 GetCapabilities

Like all OGC services that implement the OGC Web Service (OWS) interface, W3DS implements the mandatory *GetCapabilities* operation. As defined in the OWS standard this operation will return information about the abilities of the server. W3DS specification also says that this operation should preferentially be invoked by HTTP GET requests and the result should be Extensible Markup Language (XML) base encoded. The *GetCapabilities* operation can be requested using only the OWS mandatory parameters, i.e. the *service*, *request* and *version* parameters. In *Listing 3.1* is represented a KVP encoded *GetCapabilities* for a GET request.

Listing 3.1: KVP encoded GetCapabilities for GET request.

```
1 http://3dwebgis.di.uminho.pt/geoserver3D/w3ds?
2 SERVICE=w3ds&REQUEST=getCapabilities&VERSION=0.4.1
```

The result of the W3DS *GetCapabilities* request includes the standard content, i.e. meta information about the server and a description about is content. Although, the content description also contain some optional elements specific to W3DS. At the top level of content description we have the optional *layers* and *background* elements. The layers description, beyond the standard content, also contains the *queryable* and *tiled* optional properties. It also contains two extra optional elements: the *LodSet* and the *TileSet* elements.

The *background* element provide meta data that describe a background available for a *GetScene* request. Backgrounds are useful when we use a simple client that just renders the scene produced by the server without adding any extra content. Extensible 3D (X3D) have a default element to add a background to X3D scenes. A background can be an image, a simple color or a *degradé* for example. The background element is ignored by more advanced clients, like Google Earth for example.

The layer description *queryable* property is used to indicate if the layer supports the *GetFeatureInfo* operation. Sometimes is useful to deactivate the *GetFeatureInfo* operation for layers that have a lot of noise information, like Digital Terrain Model (DTM) layers, or for layers that contain sensitive information. The *tiled* property identifies the layers that support the *GetTile* operation. A layer that support the *GetTile* operation must be tiled and its description must contain a *TileSet* element.

The *LodSet* element describe the Level Of Detail (LOD) available for a layer. A layer with a *LodSet* provide several representations for the objects it contains. A layer that contains 3D buildings is a typical candidate for a *LodSet*. A building can at least have three different representations. The first will be a simple representation of the building using only 3D primitive geometries like box or pyramids. The second representation will include some simplified textures and features of the building. The third representation

provides a very detailed representation of the building. Sometimes a four level that provide the indoor of the building is also provided.

Some layers are suitable to be provided as a set of adjacent rectangular tiles. DTM layers are the typical candidates to be tiled. Using the *GetTile* operation we can efficiently access a tile using is level, row and column. To know which tiles to request a client need information about the available levels and the base grid. This information is provided by the *TileSet* element. In *Listing 3.2* is represented a *TileSet*.

The *LowerCorner* element gives us the minimum X axis and Y axis coordinates of the base grid. The *TileSizes* are related by powers of two and provide information about the existing number of levels and the size of the tiles of each level. The *NumBaseCols* and *NumBaseRows* elements give us respectively the number of columns and the numbers of rows of the base grid. A *TileSet* can be seen as a pyramid of images or a quadtree structure, it provide an efficient way to store and access spatial data.

Listing 3.2: Example of a TileSet element.

```
1 <TileSet>
2         <Identifier>dem_tileset</ows:Identifier>
3         <CRS>EPSG:4326</CRS>
4         <TileSizes>180 90 45 22.5 11.25 5.625 2.8125 1.40625 0.703125 0.3515625
              0.17578125</TileSizes>
5         <NumHeightLevels>1 1 1 1 1 1 1 1 1 1 1</NumHeightLevels>
6         <LowerCorner>-180.0 -90.0</LowerCorner>
7         <NumBaseCols>2</NumBaseCols>
8         <NumBaseRows>1</NumBaseRows>
9 </TileSet>
```

3.2 GetScene

The mandatory *GetScene* operation is the principal operation of W3DS. Is goal is to compose a 3D scene from the available GIS data. A 3D scene can be composed of a

variety of elements, from natural world to man made structures. A 3D scene can also contain elements that will be used in the rendering process, like a background or a light source. Scenes produced by *GetScene* operation to be visualize require a client capable of renders 3D content. Several scenes can be merged by the client in order to provide the best user experience.

In *Table 3.2* are listed all the available parameters that can be used in a *GetScene* request. Four mandatory parameters and eleven optional parameters are available. Note that the mandatory parameters inherited from OWS interface have been omitted.

Operation	**Definition**	**Use**
crs	CRS of the returned scene	Mandatory
boundingBox	Bounding rectangle surrounding selected dataset	Mandatory
format	Format encoding of the scene	Mandatory
layers	List of layers to retrieve the data from	Mandatory
minHeight	Vertical lower limit for boundingBox selection criteria	Optional
maxHeight	Vertical upper limit for boundingBox selection criteria	Optional
spatialSelection	Indicates method of selecting objects with BoundingBox	Optional
styles	List of server styles to be applied to the layers	Optional
lods	List of LODs requested for the layer	Optional
lodSelection	Indicates method for selecting LODs	Optional
time	Date and time	Optional
origin	Offset vector which shall be applied to the scene	Optional
background	Identifier of the background to be used	Optional
light	Add light source	Optional
viewpoints	Add Viewpoints to choose from	Optional

Table 3.2: W3DS *GetScene* operation parameters.

The *crs* parameter let us define the coordinate system that should be used for a specific request. The used *crs* must appear as available in the result of the *GetCapabilities* operation. If any of the requested layers is project in a different coordinate system the service is responsible to transform it to requested one. If the layer can't be transformed an exception should be returned.

We define the spatial sub set of data that should be used using the *boundingBox* parameter. The *boundingBox* can have two dimensions, i.e. be a rectangular space along the X and Y axis or having 3D dimensions using the Z axis to define lower and upper

vertical limits. If the bounding box is invalid an exception should be returned. Note that if the boding box is outside the valid range of the coordinate reference system its still valid but an empty scene must be returned.

A scene can be encoded in any format available on the server. The reference format to encode the produced scenes is X3D. As already referenced, a scene can contain elements that are used in the rendering process. X3D supports all elements defined in the specification, from backgrounds to models with different LODs. Al tough, other formats may not support that elements, for example KML will not support extra light sources. The server should silently ignore that elements if the requested format don't support them.

The *layers* parameter let us define the layers that should be included in our scene. The parameter contains a list of identifiers separated by a comma. The order in which the layers appear should not affect the final visual aspect of the scene. Each requested layer must appear in the *GetCapabilities* response as a valid layer. Tiled layers can also be used in *GetScene* requests. The specification don't specifies the result if a layer is invalid.

Most of Geographic Information Systems (GIS) libraries are not prepared to deal with 3D bounding box. The *minHeight* and *maxHeight* parameters are an alternative to add a lower and a upper vertical limit without using a 3D dimensional bounding box. The *spatialSelection* parameter is also related with the bounding box parameter. It give us control about how the intersections between the bounding box and 3D elements are handled.

Three values are possible to the *spatialSelection* parameter: `overlaps`, `contains_center` and `cut`. The default value is `overlaps`, if the bounding box intersects any geometry of the model this one should be considered. In the `contains_center` method the model is added to scene only if the bounding box contains is center. The center of the model should be computed using is convex hull. The `cut` spatial selection method defines that only features that are completely contained by the bounding box should be returned. Features that intersects the bounding box should be splitted.

The *styles* and *lods* parameters are lists which the length must be equal to the length of the list in the *layers* parameter. Each element must correspond to a layer. An empty space can be used, meaning that the default value of the correspondent layer should be used. The *styles* parameter let us define the style that should be applied to a layer. If a style is invalid the default one should be used in substitution. The *lods* parameter let us define for each layer which LOD should be used.

The *lodSelection* parameter tells the server how it should interpret LOD values. Three values are available: `equals`, `equals_or_smaller` and `combined`. The default value is `equals`, with this method a model is included only if it have a LOD that correspond to the requested one. The `equals_or_smaller` defines that if requested LOD is not available for a model the next lower LOD should be used. Although, if no lower LOD is available the feature should be omitted from the scene. The `combined` lets the server includes several LODs for the same model. In this case the client is responsible to select the correct model LOD tor tenderize based on the viewpoint.

The five last optional parameters, i.e. *time*, *origin*, *background*, *light* and *viewpoints* are associated with the rendering process. They are very tied to the X3D format. In practice, most of the optional parameters are omitted. Typical *GetScene* requests are similar to the one showed in *Listing 3.3*.

Listing 3.3: KVP encoded GetScene for GET request.

```
1 http://3dwebgis.di.uminho.pt/geoserver3D/w3ds?
2 VERSION=0.4.1&
3 SERVICE=w3ds&
4 REQUEST=GetScene&
5 CRS=EPSG:4326&
6 FORMAT=model/x3d&
7 LAYERS=buildings_3d,dem_3d&
8 BOUNDINGBOX=-8.301200,41.437741,-8.294825,41.444161&
9 STYLES=buildings_by_type,dem_texture_igp
```

3.3 GetFeatureInfo

A basic use case for any GIS is the possibility to click on a feature of the map and obtain some information about it. W3DS provide the optional *GetFeatureInfo* operation for that use case. Without being a operation so complex as *GetScene* operation for example, is implementation involve some interesting challenges.

The main challenge is to return information about the correct feature. Lets say we are exploring a 3D scene composed of three kind of layers: DTM, buildings and street furniture. We want obtain some information about a street lamp. Depending on the distance between our viewpoint and the lamp we will more or less precisely click on the lamp. If the lamp is near a building and have other furniture like trees around it can be difficult to precisely select the correct feature.

In *Table 3.3* are listed all the available parameters that can be used in a *GetFeature-Info* request. OWS interface inherited parameters have been omitted. Four mandatory parameters and one optional parameter are available.

The *crs* parameter is used to define in which coordinate system the *coordinate* parameter should be interpreted. The specification don't precise if coordinates that eventually appear in the result should be transformed. Although, if possible it will be a good practice to guarantee that all the returned coordinates will be project on the requested coordinate system.

The *layers* parameter have the same function that it have in the *GetScene* operation. The only particularity is that a *layer* to be considered by *GetFeatureInfo* operation must be tagged as queryable in the result of *GetCapabilities* operation.

The *format* parameter let us define the format in which the result should be encoded. The most common are XML based ones or JSON.

The *coordinate* parameter represents a coordinate on the request coordinate reference system. The coordinate can have a Z value, by default Z value is considered to be zero. Is based on this parameter that the server will try to select the correct feature from where

information should be returned. If not feature can be found a the given position a empty result should be returned.

The natural way of implementing the feature selection algorithm is to define an offset and return all the features that lies on the produced area. In W3DS *GetFeatureInfo* we will use a sphere centered in the *coordinate* parameter and consider as candidate features all the ones that lies inside that sphere. The *featurecount* parameter give us a way of limiting the number of features from where return information. By default only one feature should be used.

Operation	**Definition**	**Use**
crs	Coordinate Reference System	Mandatory
layers	List of layers to retrieve the data from	Mandatory
format	Response encoding format	Mandatory
coordinate	Position to search for features	Mandatory
featurecount	Maximum number o features that should be considered	Optional

Table 3.3: *GetFeatureInfo* operation parameters.

Typical *GetFeatureInfo* requests are similar to the one showed in *Listing 3.4*.

Listing 3.4: KVP encoded GetFeatureInfo for GET request.

```
1 http://3dwebgis.di.uminho.pt/geoserver3D/w3ds?
2 VERSION=0.4.1&
3 SERVICE=w3ds&
4 REQUEST=GetFeatureInfo&
5 CRS=EPSG:4326&
6 FORMAT=application/json&
7 LAYERS=buildings_3d&
8 COORDINATE=-8.301200,41.437741,220&
9 FEATURECOUNT=10
```

3.4 GetTile

The usual way to select a sub set of spatial data is by defining a rectangular bounding box which is used as spatial selection filter. This method provides a great flexibility and

can be used to select data from several layers, however to some kind of layers it can be very expensive to compute that selection. In some use cases is preferred to have pre computed sub sets of the data. We refer to that sub sets of data as tiles. In the context of W3DS, typical candidates layers to be tiled are terrains and 2.5D layers.

A tiled space can be seen as a georeferenced grid where each cell will correspond to a tile. A tile is characterized by its row number, column number and level. Tile level value is related to the LOD concept. The tiles size between levels should always be related by powers of two, i.e. a tile of level zero will correspond to four tiles of level one and so on. All levels share the same origin and tiles of several levels should seamlessly fit together in a heights-resolution scene.

W3DS provide the optional *GetTile* operation to access tiled layers. The *GetTile* parameters are listed in *Table 3.4*. Parameters inherited from OWS interface have been omitted. Are available six mandatory parameters and two optional parameters.

Operation	**Definition**	**Use**
crs	Coordinate Reference System	Mandatory
layer	Identifier of the layer	Mandatory
format	Tile encoding format	Mandatory
level	Level of requested tile	Mandatory
x	Row index of requested tile	Mandatory
y	Column index of requested tile	Mandatory
z	Height level index of requested tile	Optional
style	Identifies of server style to be applied	Optional

Table 3.4: *GetTile* operation parameters.

As usual a *crs* parameter is provided. If the tile available on the server is not projected on the requested coordinate reference system the server is responsible to transform it to the requested one. If the tile can't be transformed to the requested coordinate reference system an exception should returned.

GetTile operation can only use a layer per request. The used layer must be marked as *tiled* and have an associated *TileSet*, otherwise an exception should be returned. The optional *style* parameter let us specify a style that should be applied to the returned tile. The *format* parameter have the same function as the same parameter in *GetScene*

operation.

We identify the tile that we want using the x, y and *level* parameters. A tileset can define a vertical stratification for certain tiles. The optional z parameter let us select a specific vertical tile. By default all vertical tiles are selected. The specification don't defines the response to return if a requested tile is not available.

The *Listing 3.5* show an example of a POST *GetTile* request with the most common parameters.

Listing 3.5: XML encoded GetTile for POST request.

```
1 <GetTile service="W3DS" version="0.4.1" request="GetTile">
2     <CRS>EPSG:27492</CRS>
3     <Format>model/x3d</Format>
4     <Layer>dtm</Layer>
5     <Level>1</Level>
6     <X>5</X>
7     <Y>7</Y>
8     <Style>igp_texture</Style>
9 </GetTile>
```

3.5 Styling

Any service who's objective is to provide a view over GIS data needs to give some mechanism to configure the final visual aspect of the produced view. The normal way of doing this is to provide a *style* parameter who references the style we want to apply to our view. WMS *GetMap* operation and W3DS *GetTile* and *GetScene* operations provide such mechanism.

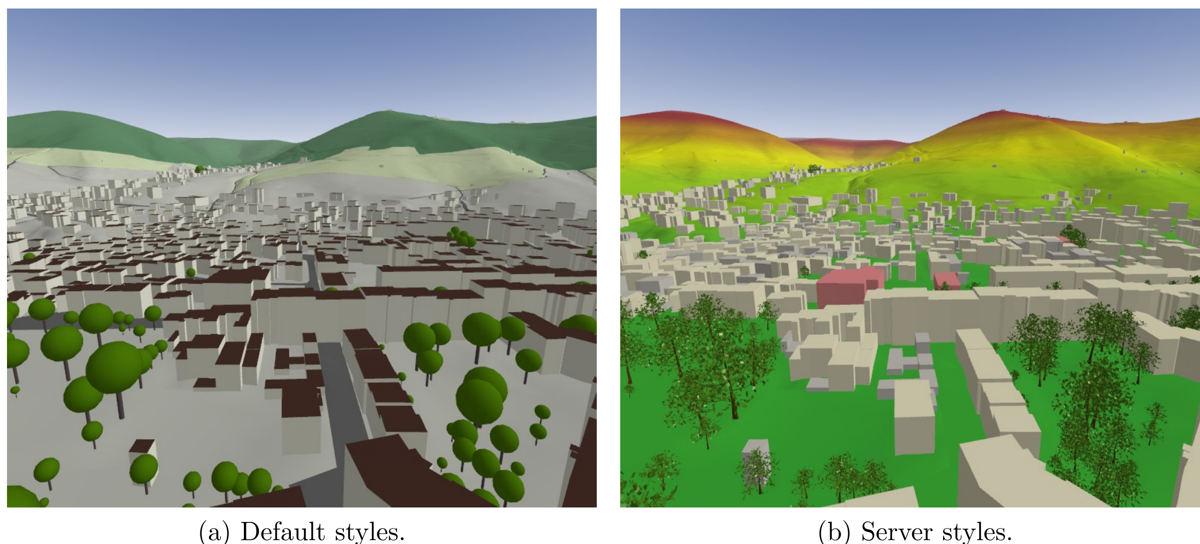
Once we know the style to apply to our view we need to find and interpret the requested style and provide a view according to his definitions. The power of styling cannot be underestimated, different styles can tell us different thinks using exactly the same data. This is why is important to guarantee interoperability between map servers

and their implementation of styling.

WMS is typically associated with the Styled Layer Descriptor (SLD) standard [12] who defines styling capabilities for 2D maps. Around 2009 was submitted to OGC, by the team behind W3DS draft standard, a candidate draft for a styled layer descriptor profile for 3D portrayal services standards [14]. If the currently 3D services OGC candidates, i.e. W3DS and WVS become OGC standards the document [14] should be merged with the SLD standard.

Styling 3D capabilities have not yet be extensively explored by the community. The main possibilities given by 3D styling are related with the rendering process and the inclusion of 3D extra models. We can for example change the lighting model or some rendering algorithms. Billboard elements like trees can be added at runtime to the scene using styles reducing server side complexity.

In *Figure 3.1* we can see a styling example. The server style change the rendering algorithm of the terrain relating the elevation with different colors. The trees models are changed to more detailed ones and the aspects of the buildings are also changed.



(a) Default styles. (b) Server styles.

Figure 3.1: Example of 3D styles (*Source [12]*).

Chapter 4

Architecture and Implementation

One of the main steps in a software project is the definition of is architecture. The implementation of the Web 3D Service (W3DS) is straight forward in terms of its functionalities, since all the operations and protocols are well defined in the specification. The main challenge will be the integration of our implementation in the architecture of an existing open source map server.

There are several Open Geospatial Consortium (OGC) compliant and reference open source implementations of several web map services, like GeoServer, MapServer or Deegree. It is wise to use such implementations to develop the W3DS component, taking advantage of existing code to manage the request pool, parsing the operations and other common tasks.

We choose to develop the W3DS component on top of GeoServer. It has detailed technical documentation, has clear policies regarding new contributions and uses recent development tools. It is written in Java and uses sophisticated technologies making it easier in terms of flexibility, extensibility and maintainability. These three design goals are particularly important in projects like GeoServer, since it implements several different web services and it is used by a large community.

In the first part of this chapter we will present our base framework. In the second chapter we will provide and overview about the main Java technologies used in our

implementation. In the last part we will describe our implementation and is integration in GeoServer architecture.

4.1 Base Framework

GeoServer is a mature open source map server build in Java. Is being used by several organizations in production environments for years. One of is main design goal is interoperability. He reads data for major spatial data sources and publish it using open standards. GeoServer is supported by a multidisciplinary community composed of individuals and organizations from around the world.

GeoServer is the reference implementation of Web Feature Service (WFS) and Web Coverage Service (WCS) and its Web Map Service (WMS) implementation is OGC certified. Several other OGC standards are also supported like Geography Markup Language (GML), Keyhole Markup Language (KML), Styled Layer Descriptor (SLD) or Web Processing Service (WPS). Some vendor-specific extensions are provided for the implemented standards. Although not portable this extensions turns GeoServer more powerful.

Another aspect of GeoServer highly appreciated by users is the graphical web interface. That interface allow users to easily in a intuitive way manage their GeoServer instance. They can configure several aspects of the server from the log level through the layers and data-sources. It also give us the possibility to interact with the server using demo requests or even preview user published layers.

Being a community-driven and a very extensible project, GeoServer receives several open-source contributions from the community. That contributions can extend GeoServer in a variety of ways and can become problematic to maintain if not correctly managed. Based on this, in order to contribute to GeoServer we need to follow some contributing rules available on GeoServer wiki.

A contribution is proposed as a patch that can change some lines or add a new module.

There are several ways to generate a valid patch, GeoServer documentation refer three: GitHub pull request, Git diff and Unix diff. W3DS module was proposed as a GitHub pull request. A GeoServer module can be classified as: core, extension or community.

Core modules are the ones distributed in the main distribution. Grossly classified they fall on two categories: architectural and functional. Architectural modules provides the base objects required by any other GeoServer module to work. The platform module is a good example of an architectural module. Functional modules provides functionalities that will be used by common users. WFS, WCS and WMS can be considered functional modules.

Extensions modules are provided as separated artifacts from the main distribution. They typically implement stable functionalities that are useful for some users but not for all. To activate an extension we only need to add is artifact to the class-path of the main distribution. An extension module must respect some requirements, like having a maintainer, a defined road-map or a significant code tests coverage. An extension module can be retrograded to a community module if its fail some of that requirements or become problematic for any other reason.

The requirements for a community module are really low, we only need the approval of one Project Steering Committee (PSC) member. Core modules and extensions started as community modules. When a community module become stable enough it maybe promoted to an extension or core module. Community modules group all the unstable or experimental modules that are not part of the release process. At the moment W3DS is a community module.

4.2 Java Technologies

The choose of GeoServer as our map server dive us in the Java world. Along the years Java has made is entry in most of the fields of software engineer and still influencing it. Java technologies have reached an high degree of maturity and successful projects are

using them from years.

Java Enterprise Edition (Java EE) is a specification that provide an environment to run professional Java applications. Java EE applications are made of several components that run on an applicational server. *GlassFish* is the reference implementation for Java EE specifications, however it exists other well know and widely used implementations like JBoss, WebLogic, WebSphere and more recently TomEE.

GeoServer is made to run on Java EE environment, although only a small subset of Java EE components are used. The Java EE components needed by GeoServer are available in some light applicational servers like Tomcat or Jetty. Most of the available examples for GeoServer run on Tomcat and uses Jetty for is integration tests.

GeoServer includes in is software stack a considerable amount of other third party libraries like Java Topology Suite (JTS) or GeoTools. In the next sections we will describe to important projects on which also GeoServer depends: Maven and the Spring Framework. We focus on this two dependencies because their are mainly responsible for the high maintainability and extensibility of GeoServer.

4.2.1 Maven

Maven is an open-source build automation tool for software projects, mainly used for Java projects it can also take care of projects written in other languages like C# or Scala. Maven purchase the same objective as Unix Make command or Ant tool, but is based on different concepts. Unlike Make or Ant were we need to define the build tasks of our project in Maven we use a declarative approach, i.e. we describe our project and let Maven create the build tasks for us.

The plugin architecture on which Maven is based grants him an high capacity of extensibility. Some of is base behavior is given by plugins like the dependency management. Plugins can be used for several proposes like IDE integration or even add support for a new language. A plugin can be associated with one or more Maven goals. The big number of existing maven plugins have contributed to turn maven in a Swiss army knife for

build modern and flexible applications.

Another appreciated aspect of Maven are the standard procedures he introduces. A developer already introduced to Maven will easily understand the structure of a new project based on Maven. No time will be spend understanding obscure build scripts or IDE custom project source folder organization.

Project Structure

Maven give us the possibility to separate our projects in several modules that have they own life cycle. A module can contain any piece of the architecture and can also contains sub-modules which let us group components of our architecture by some criteria. Such granularity can be used to have a flexible code organization.

The concept of modules can be coupled with the profile support given by Maven. Profiles allow us to activate or deactivate some build goals based on some criteria. One of the typical use cases for profiles is the running control of tests. When used with modules profiles can be used to control which components are included in the final distribution or even to produced several distributions.

Maven defines a standard structure for its projects. Its possible to redefine the default locations but its highly recommended to follow the standard structure. Configurations will be short, it will be easy to integrate new plugins and the learning curve for a new member of the project will be shorter.

Even following the standard structure of Maven a module can become very complex, but the complexity is hierarchized in sub-directories. The top module folder contains a `pom.xml` configuration file. A `src` folder that will contain all the necessary elements to build the module. A `target` folder where will be saved all the output produced by the build process.

Sometimes on big projects also appears the `bin` and `conf` folders. The first contains some executable scripts need by the build process. The second contain some configuration files that are used by the *resources* Maven plugin. At this level are also expected metadata

folders like the ones used by SVN, CVS or GIT.

The `src` folder typically contains a `main` folder and a `test` folder. The first will contain the elements used to build the main artifact and the second the unit tests. Sometimes this level also contain an `it` folder which contains the integration tests. The directory structure of the level below these folders is similar. Typically it will contain, for Java modules, a `java` folder and a `resource` folder.

In the case of an aggregator module, i.e. a module used to group a set of sub-modules. The top level will contain a `pom.xml` file and the sub-modules folders. The `pom.xml` will reference the sub-modules using the modules section as show in *Listing 4.1*. Note that the type of packing for the aggregate module is `pom`.

Listing 4.1: Multi-module *pom* that aggregate some data base drivers.

```
1  <project>
2    <groupId>org.example</groupId>
3    <artifactId>data-base-drivers</artifactId>
4    <packaging>pom</packaging>
5    <name>Data Base Drivers</name>
6    <version>1.0-SNAPSHOT</version>
7    <modules>
8      <module>postgres</module>
9      <module>oracle</module>
10   </modules>
11 </project>
```

Pom

The core component of a Maven module is the Open Web Service (OWS), an Extensible Markup Language (XML) based document were we describe our module properties like the version, dependencies and plugins configurations. In *Listing 4.2* we can see a basic *pom* example that will produce a jar artifact and have a dependency to the JDBC PostGres database.

A Maven artifact is uniquely identified by is *groupId*, *artifactId*, *version* and *type* and is the result of a Maven build. An artifact can be of any type, most common are: jar, war, zip and ear. By default a produced artifact is always uploaded to the local maven repository but can be also be upload to any other repository using the `deploy` command. An artifact can be requested by other Maven projects using the `dependency` tag.

Listing 4.2: Simple *pom* example.

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <parent>
4      <groupId>org.example</groupId>
5      <artifactId>data-base-drivers</artifactId>
6      <version>1.0-SNAPSHOT</version>
7    </parent>
8    <groupId>org.example</groupId>
9    <artifactId>postgres</artifactId>
10   <version>1.0-SNAPSHOT</version>
11   <dependencies>
12     <dependency>
13       <groupId>org.postgresql</groupId>
14       <artifactId>postgresql</artifactId>
15       <version>9.2-1002-jdbc4</version>
16     </dependency>
17   </dependencies>
18 </project>
```

In a multi-modules projects, which is frequently the case, properties, dependencies and plugins configurations can be inherited by sub-modules using `parent` element. The `postgres` module (*Listing 4.2*) will inherit content from is parent `data-base-drivers` (*Listing 4.1*).

In big multi-modules projects is common to have in the top *pom* the sections: dependency management and plugin management.

The dependency management let us declare dependencies that won't be inserted in the final distribution unless a sub-module depends on it. To use a dependency that have been declared in the dependency management a sub-module only need to use the *groupId* and *artifacId*. It this mechanism the dependencies of the hole project are managed in one place.

The plugin management section is the place where we configure the default behavior and properties for most of the used plugins, like its version for example. Although the plugin will run only for the plugins who declare it. The plugin configuration defined in the plugin management is inherited by all the sub-modules and will me merged it any configuration existing in the sub-module.

Build Life Cycle

The build life cycles are a good example of the standard procedures introduced by Maven. A build life cycle clearly defines the process for building and distributing some artifact. Maven come with three build-in life cycles:

default handles the building, testing and deployment of the project.

clean handles the project cleaning.

site handles the creation of the project site documentation.

A build life cycle is composed of several build phases. When we execute the Maven command `mvn install` we are executing the build phase `install` of the Maven default life cycle (*Figure 4.1*). Build phases are executed sequentially, i.e. when we call the `install` build phase all the previous build phases will also be executed.

Build phases are made of goals. A goal represent a well bounded an defined task. A goal can be associated with zero or more build phases. A goal can be directly called outside any build phase or build life cycle. A frequently used one is the `eclipse:eclipse` goal which use Maven Eclipse plugin to produce IDE integration files.

In a Maven command we can mixture calls to build phases and specific goals. Maven grant to us that all the goals will be executed in the correct order. Multiple goals bound to a phase are executed in the same order as they are declared in the *pom*.

Plugins are the standard way to add new goals to build phases. A plugin is always associated with one or more goals. The plugin configuration section `executions` give us a powerful control on the execution work-flow of goals. For example, choose the build phase where the goal will be bound or the execution order.

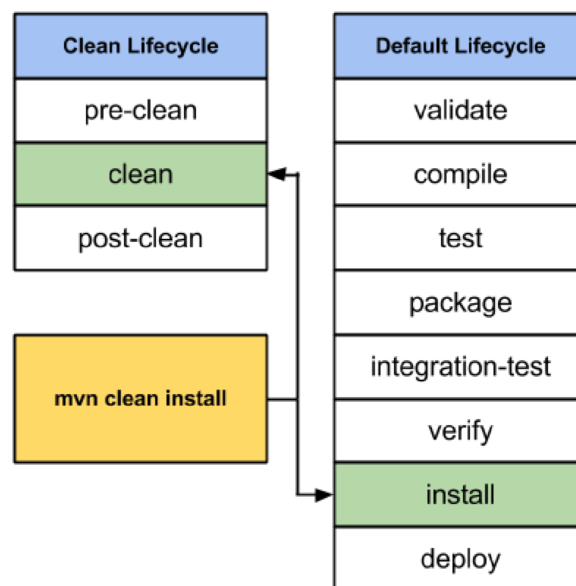


Figure 4.1: Maven `clean` and `default` life cycles and its build phases. The command `maven clean install` will execute the `clean` build phase from the `clean` life cycle and the `install` build phase from the `default` life cycle.

Dependencies

In Java projects if we want to use an external library we need to put its jar on the class-path of the java virtual machine. The problem is that the library we want to use have also is own dependencies that we need to provide and so on. In a big project were we easily have dozens of dependencies and sometimes the same library in different version this quickly become a nightmare.

The dependencies management is one of the most emblematic and well know features of Maven. Most of the recent Java projects cannot live without this feature. With is dependency management Maven give us a standard way to easily deal with project dependencies. The basic idea is that if we want to use a library in our project we only need to add is Maven artifact as dependency. Maven will do the rest for us.

A key feature of Maven dependency management is the possibility to automatically download artifacts from remote repositories when the artifact is not available locally. Maven comes with some default public repositories but give us the possibility to configure extras repositories and manually install artifacts.

The more recent versions of Maven handle for us the transitive dependencies, i.e. the dependencies on which a declared dependency depends. In *Listing 4.3* are show some transitives dependencies of Hibernate. If we want to use Maven in our project we only need to had a dependency for Hibernate. Maven will automatically download and install all libraries on which Hibernate depends like ANTLR for example.

Transitives dependencies have a major drawback. If some dependencies have the same transitive dependency Maven will only retain the first one in the dependency tree regardless to is version. The problem is that frequently we have the same transitives dependencies but in different versions. The common solution is to use the exclusion section to exclude the incompatible versions and use a version that work for both. If no compatible version can be founded to the transitive dependency, we also need to upgrade or downgrade the main dependencies until a common version for the transitive dependency can be founded.

Sometimes we need to limit the transitivity of a dependency, i.e. define for which build phases a transitive dependency is required. Based on this use case Maven give us the dependency scope. By default are available six dependency scopes: `compile` , `provided` , `runtime` , `test` , `system` and `import`.

The default scope is `compile` and it adds no limitations to the transitivity. A dependency which scope is `provided` is expected to be provided by the JDK or the container

and is not transitive. An example are the Java EE APIs which will be provided by the Java EE container. A `runtime` dependency is not required for compilation only for execution. A `test` dependency is only used for testing proposes, for example the JUnit library.

The last two scopes don't actually affect the transitivity of the dependency. The `system` scope is used to identify dependencies that are not available in a repository. The `import` scope is basically used to include a a dependency management section from another OWS.

Listing 4.3: Partial snapshot of Hibernate dependency tree produced by `mvn dependency:tree` command.

```
1 [INFO] +- org.hibernate:hibernate:jar:3.2.5.ga:compile
2 [INFO] | +- net.sf.ehcache:ehcache:jar:1.2.3:compile
3 [INFO] | +- commons-logging:commons-logging:jar:1.0.4:compile
4 [INFO] | +- asm:asm-attrs:jar:1.5.3:compile
5 [INFO] | +- dom4j:dom4j:jar:1.6.1:compile
6 [INFO] | +- antlr:antlr:jar:2.7.6:compile
7 [INFO] | +- cglib:cglib:jar:2.1_3:compile
8 [INFO] | +- asm:asm:jar:1.5.3:compile
```

4.2.2 Spring Framework

When we build an Enterprise Java project we have a major choice: Java EE or Spring Framework ? Nowadays the answer to that question is complicated. Both Java EE and Spring Framework, especially Java EE, have learned from they mistakes. However when GeoServer adopted Spring Framework the answer to hat question was short: Java EE was horrible and Spring was great.

Java EE is a standard with a large community that include some of the industry majors. However Spring become very widespread with a huge professional community. The main reason why Spring comes up and become so popular was that the use of

first versions of Java EE was really painful. Bad applicational servers, lots of XML configurations, heavy applications and a complicated development are some of the main reasons.

Spring is a lightweight container when compared to Java EE. Is easy to use, can be deployed on web containers like Tomcat, uses convention over configuration and is deploy took just a few seconds. Spring also offers more flexibility and powerful tools. An example is the aspect-oriented programming provided by Spring against the interceptor mechanism given by Java EE.

Spring is based on two major concepts: inversion of control and aspect-oriented programming. Its distribution is composed of several modules that give support to a widely number of common software components. This modular architecture allow us to use Spring in a incremental way. The core of Spring is based on beans and their factory. Beans are used by dependency injection container to arm Spring with all the necessary decoupling between implementation and its interfaces.

Inversion of Control

Java interfaces or abstract classes let us define the behavior of an object regardless to is implementation. But at some moment in our implementation we need to explicit call a constructor to create the object. A constructor always reference a concrete implementation. So when we call a constructor the decoupling between interfaces and implementation is broken.

The factory pattern help us solving this issue handling the creation of our objects. When we want to create an object we pass is abstract type to the factory and it gives us back on object of the desired type. With this method the dependency between the interface and the implementation is removed. However the factory will also need at some point to call a constructor, which implies that it will have to know the implementation.

The factory solution have the advantage of grouping the problem in one place. When we need to change the implementations we only have to modify the factory. However

no matter how customizable is our factory we still need to recompile all the code, i.e. the factory will always explicitly know which implementation is used. Although this solution is acceptable from most of the situations it still unacceptable from some advanced situations.

Lets take as use case an enterprise scheduler platform who's main work is to run some scheduled tasks at a specific time. The behavior of a task is well defined by an interface and a factory is used to produce the tasks objects. Our platform is sold to several customers and some of them have their own implementation for the task interface. We will need to have a different factory for every client that have a custom implementation for the task interface.

If the custom task interface implementation is owned by the client that mean that will have a different version of our platform for every client, which is unacceptable. The solution is to have the task implementation completely decoupled from is interface, i.e. our platform will never know the implementation that is been used. This can be reached using inversion of control pattern (IoC).

IoC is an architecture pattern where the control flow of the application is delegated to a framework. IoC is sometimes compared to the Hollywood principle: *Don't call us, we'll call you.* Is not the application that handles the calls to the framework but the framework that handles the calls to the application. Spring Framework implements IoC using the dependency injection pattern which is a specific kind of IoC.

Dependency injection allow us to completely remove hard dependencies between different software components. The main application may only use abstracts types, a suitable implementation will by injected at runtime by the framework. The IoC container will be responsible about life cycle of the objects managed by him. The objects handled by the IoC container are defined using XML based configuration files or using Java annotations metadata.

Modern Java applications use dependency injection to provide extensions points in their work-flow that are used to build plugins. Dependency injection is also largely used in

testing environments. The decoupling provided between implementation and abstraction can be used to mock objects instead of using complex production real objects.

Beans

A bean is a Java object who's life cycle is entirely managed by the IoC container. Beans are defined in XML based configuration files, typically called `applicationcontext.xml`, in a `<bean/>` section (*Listing 4.4*). Spring gives us a large set of options to customize our beans, but we should keep in mind one of the base design aspects of Spring: convention over configuration.

The truly mandatory properties of a bean are is *id* and *class* for the others properties Spring will provided a default value if no one is defined. The *id* property uniquely identify a bean and the *class* property defines the Java type of the bean. In *Listing 4.4* we can see a bean definition who's *id* is `Logger` and is *class* is `LoggerImpl`.

Listing 4.4: Spring bean XML definition.

```
1 <bean id="Logger"
2      class="org.utils.LoggerImpl"
3      scope="singleton"
4      init-method="init"
5      destroy-method="destroy"
6      parent="defaultBean">
7      <constructor-arg ref="logConfProperties"/>
8      <property name="level" value="${logger.level}"/>
9 </bean>
```

The scope property let us configure how the IoC container will manage a request for a new bean. Are available five scopes types: singleton, prototype, request, session and global-session. The default scope is singleton and a bean who's scope is singleton will only have one instance. The IoC will cache the first instance created and will always return the same instance. This scope should be used for stateless beans.

The prototype scope can be seen as the opposite of singleton scope. A new instance

of the bean will always be created by the IoC container. This scope should be used for statefull beans. The last three types of scopes are only available in a Spring web-aware context. The request scope is bound to an HTTP request, the session scope is bound to a an HTTP session and the global-session scope is bound to an HTTP global session.

The life cycle of a bean can be quite complex but a reasonable application should only need to work on two phases: bean initialization and bean destruction. Spring provides two properties for these uses cases: `init-method` and `destroy-method`. Both of them let us reference a method of the bean class. The first will be called after the bean instance is created and the second before the instance is removed from the container.

Spring beans configuration supports inheritance. The parent property lets us reference another bean from where configurations should be inherited. Beans inheritance is typically used when the bean class extends from another class. Generally we define an abstract bean where we made the configuration related with the super class and make the abstract bean parent of our bean.

The `constructor-arg` element let us defines parameters that are used to call a constructor. Spring will automatically find the correct constructor based on the number of `constructor-arg` elements and the type of values used by them. We can also set properties of our new bean using `property` element. The attribute `name` references the Java name of the property. Spring will the `name` attribute to call by reflection the correct setter.

Values can expressed using `value` or `ref` attributes. The first is used when the value is a basic type like string or integer. The second is used when we the value is an object. In this case we reference another bean using is id. In Spring configuration files when can reference properties defined on bundle files using `${...}` syntax. We can also use Spring Expression Language (SpEL) for more advanced use cases.

4.3 Architecture

Our architecture is highly coupled and influenced by GeoServer architecture. The bad news is that most of the components of our W3DS implementation will not work with another map server. The good news is that the redundant tasks are handled by GeoServer and our components only focus on fundamentals aspects of W3DS specification. We also benefice from a well defined architecture that already made is proofs from we can inherit some interesting components like the testing environment for example.

Our W3DS implementation will be contributed to GeoServer as a community module. Community modules are added under the community source folder of GeoServer source code. A community module is only activated by a Maven profile, in this case the `w3ds` profile. We need also to add a dependency behind the `w3ds` profile to our w3ds community module in the `web-app` module. With this dependency and the `w3ds` profile activated our W3DS implementation will be included in the GeoServer `war`. With W3DS registered as a community module we reach the first stage of GeoServer architecture integration.

The central component of GeoServer architecture is the `Dispacther` element. The main responsibility of the `Dispatcher` is to handle OGC Web Service (OWS) requests. Its control the complete execution cycle of every request providing extensions points for the main execution steps. That extensions points are defined as Spring beans. Implementing a new service consist in providing a set of beans that will be executed in that extensions points.

Secondary extensions points are also provided, they are typically executed after every major step of the pipeline. For example, after the service of a request as been identified the method `fireServiceDispatchedCallback` will execute the `serviceDispatched` method of every registered callback. Callbacks are Spring beans that implement the `DispatcherCallback` interface. This mechanism allow us to provide extensions for existing services like a security mechanism or the handling of an additional parameter.

The execution pipeline of a OWS request is composed of six main steps. The three

first steps are related to the identification of the request, i.e. the service, the operation and the version. In the four step we create an `Operation` object. Using the operation object created in the previous step in the five step we execute the operation. In the last step we use the result `Object` to write a response encoded in the requested format. This six steps can be encapsulated in three stages: read, execute and write (*Figure 4.2*).

A OWS HTTP request can be encoded in two forms: Keyword Value Pair (KVP) and XML. The first is typically associated with a GET request and the second with a POST request. At least three parameters should be provided: the service, the operation of the service to execute and the service version. Based on the operation parameters types the correct KVP or XML parser are used.

KVP parsers are registered as Spring beans that extend from the `KvpRequestReader` class and XML request readers are Spring beans that extends from the abstract class `XmlRequestReader`. Both of the request parsers types implement a `read` method. Based on the return type of this method the `Dispatcher` will pick the correct parser for every parameter type of the operation.

The execution step is very simple, we already know the service operation and the parameters, we just have to call by reflection the execution method. A service is registered as Spring bean were we explicitly declared all the available operations. Based on the service, operation and version parameters we find the correct operation to execute simply matching the service and operation names. If the provided version is not valid the most recent one available is used.

The execution step produce a generic result, i.e. is Java type is `Object`. In the last step we will encode the result in a suitable format and serialize it to an output-stream. Response encoders are registered as Spring beans that extend the `Response` abstract class. Response encoders implement to fundamentals methods: `canHandle` and `write`. The first is used to test if this encoder can handle our result the second is used to write the result to an output-stream.

In *Figure 4.2* we can see the Spring beans implemented for the W3DS and how they

are used in `Dispatcher` life cycle. GeoServer also allow us to extend is Web interface using Apache Wicket and Spring beans. Briefly described, to extend the graphical interface of GeoServer we produce our Apache Wicket components and inject them as Spring beans. We can also implement a `ServiceInfo` object that describe our service. This object will be used in logging and other reporting tasks.

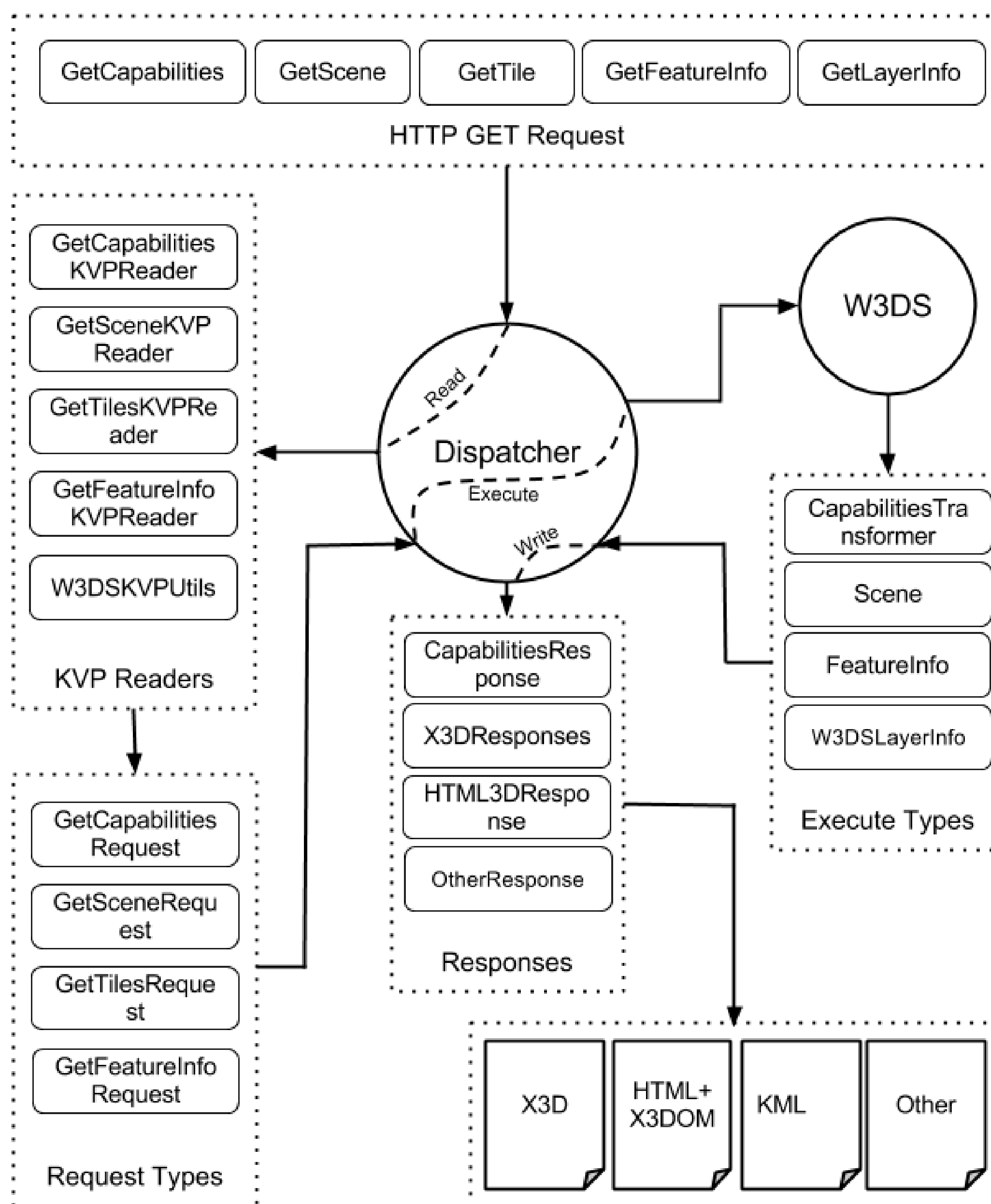


Figure 4.2: Simplified description of the Dispatcher and the W3DS components.

4.4 Implementation

Our W3DS implementation is composed of five major modules: *service, types, styles, responses* and *web*. The *service* component contains the objects closely related tot he W3DS specification. The *types* component contain all the types used to represent W3DS related concepts. We need to extend GeoServer SLD support to include 3D specifies, the *styles* component contain that extensions. The *responses* component will contain the encoders for the supported formats. The *web* component contain our extensions to the GeoServer Web interface.

As explained in the previous section, most of our objects will need to be registered as Spring beans. We use a single Spring beans file for registering all our beans. The file is called `applicationcontext.xml` and its located in the resource folder of the W3DS module. Frequently we inject in our beans two GeoServer components: the `Catalog` bean and the `GeoServer` bean. The first give us useful methods to access the GeoServer catalog, it is used to retrieve 3D styles for example. The second give us access to generic functionalities, like logging for example.

4.4.1 Service

The main object of the *service* component is the `W3DS` class. Its contains the elements necessary to register W3DS as a GeoServer service. The sub components on it depends fall on two categories: service metadata or service operations.

GeoServer needs some metadata information about a service, like the available versions or is name for example. That information can be hard-coded in the `W3DSInfo` class or loaded from a XML based configuration file. We implement the `W3DSXStreamServiceLoader` class that will find a `w3ds.xml` file in the class-path and load W3DS service information from that file at server start-up.

W3DS operations *GetCapabilities*, *GetScene*, *GetTile* and *GetFeatureInfo* are methods of `W3DS` class. When the `Dispatcher` executes a W3DS operation, behind the scenes it

will call by reflection the correspondent method of the `W3DS` object. In *Listing 4.5* we can see the `w3dsserviceregister` where we register all the W3DS operations that are available on the the `W3DS` object.

Listing 4.5: Spring beans related to the W3DS *service* component

```
1  <bean id="w3dsservice" class="org.geoserver.w3ds.service.W3DS">
2          <constructor-arg ref="geoServer" />
3  </bean>
4  <bean id="w3dsLoader" class="org.geoserver.w3ds.service.W3DSXStreamLoader">
5          <constructor-arg ref="resourceLoader" />
6  </bean>
7  <bean id="w3dsserviceregister" class="org.geoserver.platform.Service">
8          <constructor-arg index="0" value="w3ds" />
9          <constructor-arg index="1" ref="w3dsservice" />
10         <constructor-arg index="2" value="0.4.1" />
11         <constructor-arg index="3">
12                 <list>
13                         <value>GetCapabilities</value>
14                         <value>GetScene</value>
15                         <value>GetFeatureInfo</value>
16                         <value>GetTile</value>
17                 </list>
18         </constructor-arg>
19 </bean>
20 <bean id="w3dsURLMapping" class="org.geoserver.ows.OWSHandlerMapping">
21         <constructor-arg ref="catalog" />
22         <property name="alwaysUseFullPath" value="true" />
23         <property name="mappings">
24                 <props>
25                         <prop key="/w3ds">dispatcher</prop>
26                 </props>
27         </property>
28 </bean>
```

In *Listing 4.5* are showed all the registered Spring beans related to the *service* component. The `w3dsservice` bean register our `W3DS` object. Our XML configuration loader is registered as the `w3dsLoader` bean. The `w3dsserviceregister` register the W3DS service on GeoServer, note the reference to the `w3dsservice` bean. The `w3dsURLMapping` bean makes possible to directly reference the W3DS service on HTTP requests.

4.4.2 Types

The *type* module contain all the types related to the W3DS service. Every request have their own type except the *GetCapabilities* which use the default support given by GeoServer to OWS *GetCapabilities* requests. For every request we have a KVP parser that will parser requests parameters and produce the correct request type. All the KVP parsers created are registered as Spring beans (*Listing 4.6*).

A request to a *GetScene* operation is mapped by the `GetSceneKvpRequestReader` on a `GetSceneRequest` object. The *GetTile* operation is mapped on a `GetTileRequest` object by the KVP parser `GetTileKvpRequestReader`. The *GetFeatureInfo* operation is mapped on a `GetFeatureInfoRequest` object by the `GetFeatureInfoKvpRequestReader` KVP parser.

All the KVP parsers created are registered as Spring beans (*Listing 4.6*). KVP parsers are also responsible to validate the request parameters and produced the the correct error messages when the value of parameter is not valid or a mandatory parameter is missing.

We have the necessity to extend the layer representation given by GeoServer to include W3DS specific information, like 3D styles for example. We create the `W3DSLayer` class which contains all the information necessary to represent a layer as needed by W3DS operations. We also create the `LOD` and `TileSet` objects. The first is used to handle information related to the level of detail of a layer. The second is used to handle information of tiled layers that can be request by the *GetTile* operation.

We also have the necessity to create objects to handle the result of W3DS operations. That objects will be used by response converters to encode the result in the requested

format. The execution result of *GetScene* and *GetTile* operations are mapped on the `Scene` object and the result of a `GetFeature` info is mapped on to a `FeatureInfo` object.

Listing 4.6: Spring beans related to the W3DS *types* component.

```
<bean id="getSceneKvpReaderRequestReader"
      class="org.geoserver.w3ds.kvp.GetSceneKvpRequestReader">
        <constructor-arg value="org.geoserver.w3ds.types.GetSceneRequest" />
        <constructor-arg ref="catalog" />
        <constructor-arg ref="geoServer" />
</bean>
<bean id="getFeatureInfoKvpRequestReader"
      class="org.geoserver.w3ds.kvp.GetFeatureInfoKvpRequestReader">
        <constructor-arg value="org.geoserver.w3ds.types.GetFeatureInfoRequest" />
        <constructor-arg ref="catalog" />
        <constructor-arg ref="geoServer" />
</bean>
<bean id="getTileKvpReaderRequestReader"
      class="org.geoserver.w3ds.kvp.GetTileKvpRequestReader">
        <constructor-arg value="org.geoserver.w3ds.types.GetTileRequest" />
        <constructor-arg ref="catalog" />
        <constructor-arg ref="geoServer" />
</bean>
```

4.4.3 Styles

GeoServer have a good support to SLDs and even provide some extensions. Although SLD cannot represent all the information necessary to 3D GIS. One of the critical aspects was the inclusion of 3D models.

Like terrain, urban models can be very expensive to handle. City models also do not change often. W3DS does not provide a specific operation that lets us retrieve a single 3D model as `GetTile` does for tiled terrains. This makes it more difficult to develop an efficient cache system. However, the main formats (Extensible 3D (X3D), KML and

XML3D) used to encode `GetScene` or `GetTile` responses give the possibility of referencing external 3D models.

Instead of having our 3D models stored in the database and translated to a specific format by W3DS, we only place a reference to that georeferenced model. This functionality can be achieved by extending SLDs to support the inclusion of external 3D models (*Listing 4.7*).

In that situation the rendering engine on the client side will be responsible for requesting and handling the 3D model. On the client side, only one model is downloaded for each model type. On the server side, instead of storing thousands of models, no model at all is stored in the database.

For some applications the domain model is composed of thousands of elements from a limited number of different types, the use of this strategy greatly improved performance. This improves the performance and storage space on the server side, minimizes data transferred between the server and the client, and also improves efficiency on the client side.

Listing 4.7: Example of the inclusion of a 3D model using extended SLD.

```
1  <PointSymbolizer>
2         <Graphic model="true">
3                <altitudeMode>relativeToGround</altitudeMode>
4                <altitude>50</altitude>
5                <heading>0</heading>
6                <tilt>45</tilt>
7                <roll>0</roll>
8                <href>http://localhost:8080/models/airplane.x3d</href>
9                </Graphic>
10 </PointSymbolizer>
```

4.4.4 Responses

We provided three response formats: KML, X3D and HTML+X3D. The HTML+X3D is a wrapper around the X3D format, i.e. it uses the result of the X3D response format and put it on HTML page that uses X3DOM library to renders the X3D elements.

The KML responses are produced using the Java Api For KML (JAK) library, which is also be used by the new GeoServer KML response encoder for the WMS service. Behind the scenes W3DS delegates the production of KML to WMS and only add afterwards the 3D elements.

X3D response format is the default format for *GetScene* and *GetTile* operations. The lack of a complete X3D API for Java force us to develop a light one. Our X3D builder support all the needs of W3DS, from georeferenced scenes to the inclusion of 3D models.

All W3DS responses extend from the `Response` class and are registered as Spring beans. Note that are our response formats are not stateless so they are not registered as singletons like all the other W3DS beans. This `Response` class have two fundamentals methods: `canHandle` and `write`. The first is used by the `Dispatcher` to ask a response if it can handle the request. The second is used to write the response encoded in the requested format.

Listing 4.8: W3DS response formats registered as Spring beans.

```
1 <bean id="W3DSX3DResponse"
2       class="org.geoserver.w3ds.responses.X3DResponse"
3       singleton="false" />
4 <bean id="W3DSHtmlX3DResponse"
5       class="org.geoserver.w3ds.responses.HtmlX3DResponse"
6       singleton="false" />
7 <bean id="W3DSKmlResponse"
8       class="org.geoserver.w3ds.responses.KmlResponse"
9       singleton="false" />
```

4.4.5 Web

GeoServer administration web interface is a very appreciated feature by users. We extend that web interface to include W3DS specific configurations. We add an administration web page for the service, i.e. where we can activate or deactivate the service and provide some metadata about the service (*Figure 4.3*). We also add a layer configuration page from where we can configure W3DS layers (*Figure 4.4*).

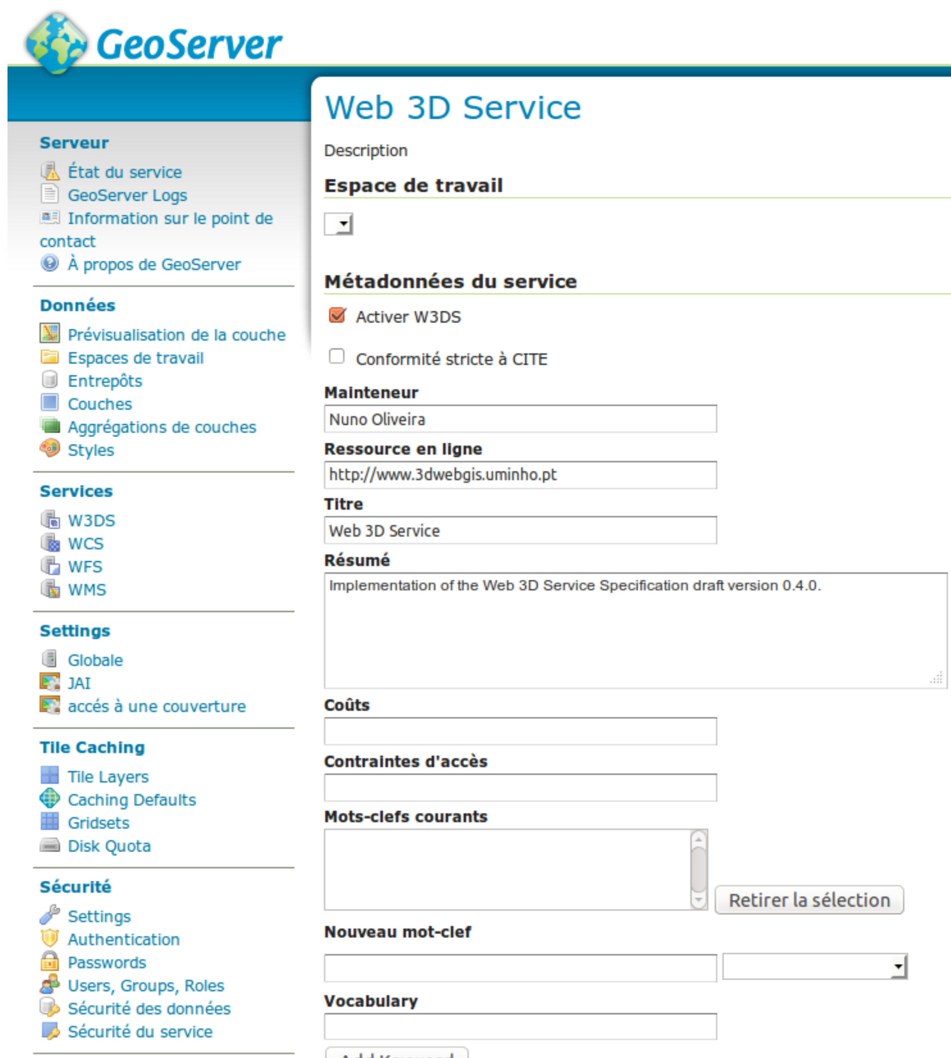


Figure 4.3: Print screen of W3DS service configuration page.

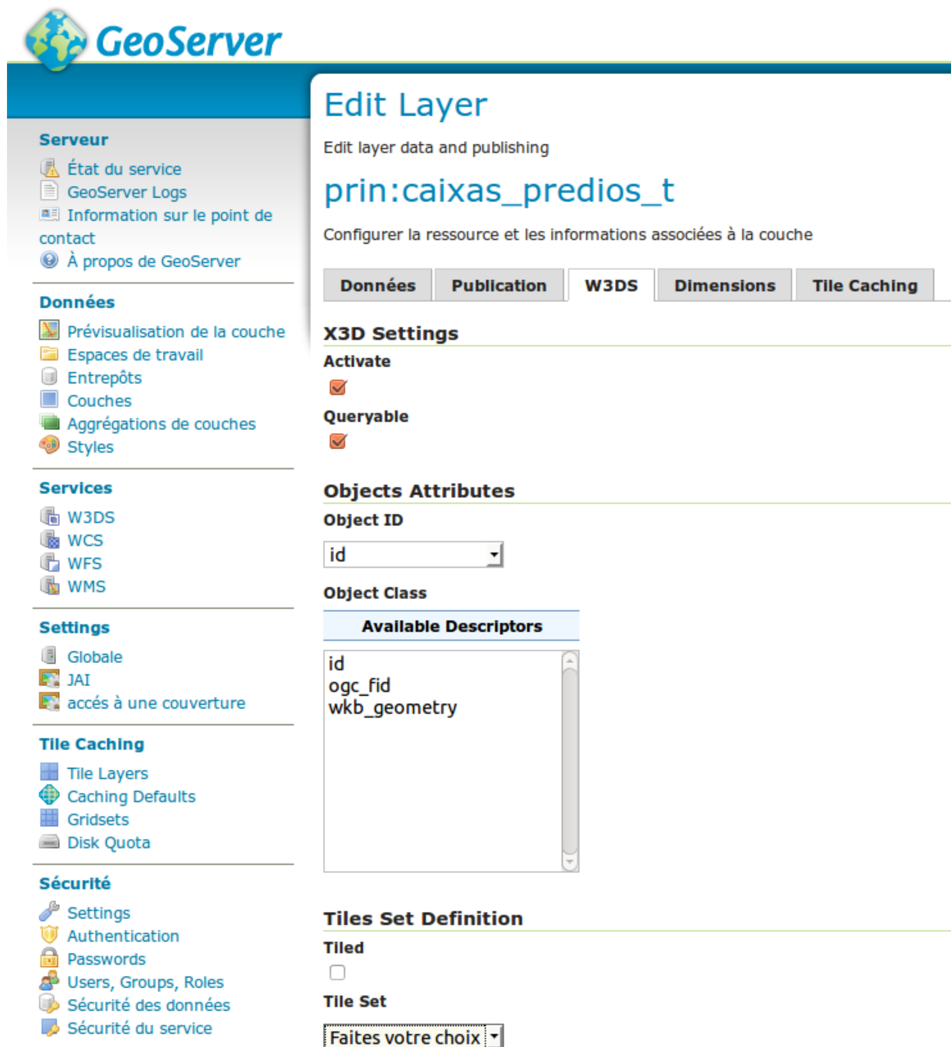


Figure 4.4: Print screen of W3DS layer configuration page.

GeoServer web interface is based on Apache Wicket, to extend that interface we provide Spring beans that provide Apache Wicket web components (*Listing 4.9*). GeoServer provide abstract class that help us extending is interface. For example, to add a new layer configuration tab we can extend form `LayerEditTabPanelInfo`. We also inherit some useful features, like the internationalization.

Listing 4.9: W3DS Apache Wicket components registered as Spring beans.

```
1 <bean id="W3DSEditTabPanelInfo"
2      class="org.geoserver.web.data.resource.LayerEditTabPanelInfo">
3        <property name="id" value="W3DSEditTabPanelInfo" />
4 </bean>
```

Chapter 5

Results and Evaluation

In the previous chapter, we presented the architecture and the implementation of the W3DS service we made on top of GeoServer. In this chapter we will use the developed service to present some preliminary results. To accomplish this we will consider a use case related to telecommunication infrastructures.

In the first section we will describe our use case. The second section is related with the data preparation. We will present our algorithm to produce 3D tiled terrains and the process we use to create telecommunications infrastructures 3D representations. In third section we will describe how to configure a GeoServer instance with W3DS module activated. The remaining sections describe the results of several requests in different contexts.

5.1 Use Case

Our use case is based on the 3D georeferenced visualization of telecommunications infrastructures. The data volume, the type of data and how the elements are related together provide some interesting challenges. We also need to manage context elements like a Digital Terrain Model (DTM), buildings and street furniture. Our use case can be separated on two parts. In the first one we prepare the data to be served by W3DS and

in the second part we provide a web 3D visualization over the 3D GIS data served by GeoServer.

In *Figure 5.1* is represented the architecture of our use case. The entry point of the server side will be the W3DS service. The client will use W3DS operations to obtain the necessary data. The supported formats are HTML, KML and X3D. The HTML format produce an HTML page that use X3DOM library to renders the X3D content. The KML response format will produce scenes to be consumed by Google Earth web plugin or is desktop version. The X3D format produced scenes can be viewed using a web browser plugin or using some desktop viewer.

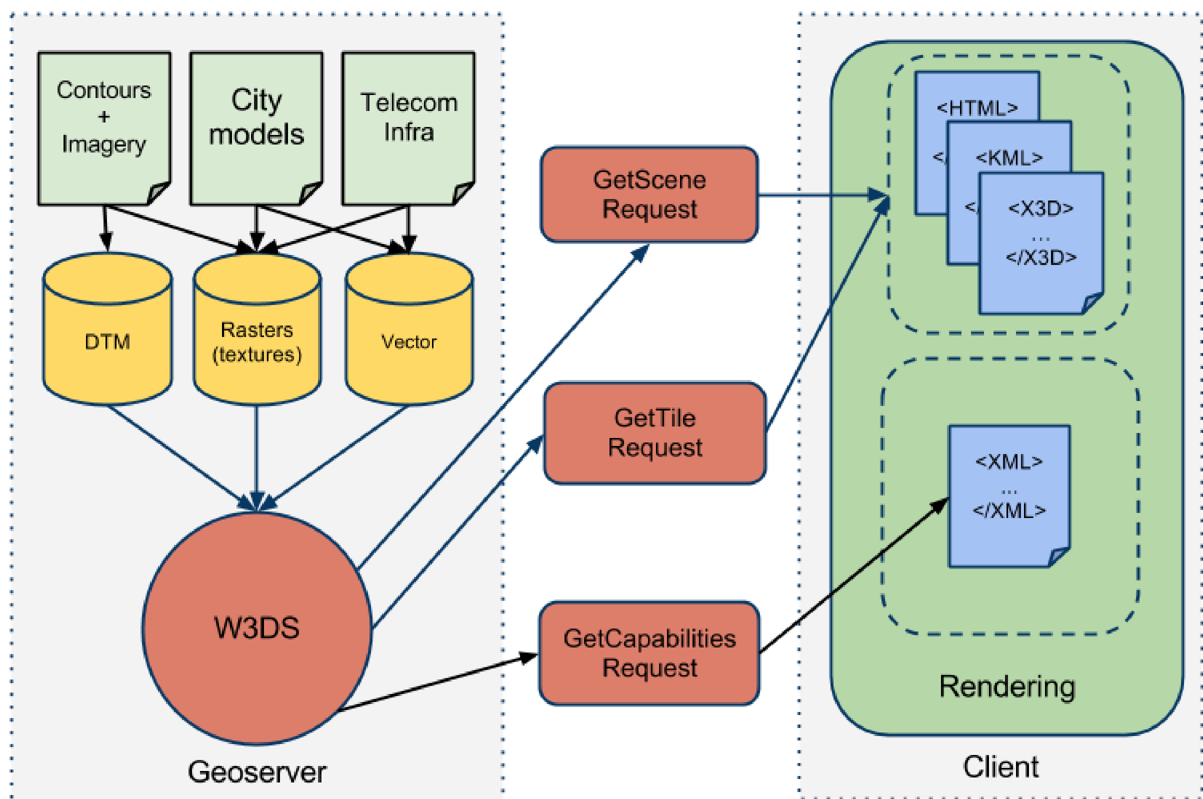


Figure 5.1: Architecture of the use case.

Our web client will be build on top of Google Earth web plugin which is the only viable option to build a client with the minimum requirements. We will need to write the integration code between Google Earth and the W3DS service. Google Earth will only be used as a rendering engine. Using Google Earth API to obtain information about

our location we will request in runtime the needed data from the server and inject it on Google Earth.

One of the requirements of our use is that it must be possible to provide a 3D visualization over telecommunication infrastructures encoded as georeferenced 2D lines and points. The solution is to use 3D SLDs to provide a 3D representation of the geometries. Based on this we will need to provide 3D models for the telecommunication infrastructures. Those models will be encoded in Collada and X3D.

5.2 Dataset Preparation

Our data set is composed of several entities with a very different topology. For example poles are represented as 3D models but the cables between them are represented as lines. Some of these cables and junctions are below ground level, while others are some meters over ground level like aerial cables for example. The telecommunication infrastructures and city models must be integrated with the DTM. In the X3D view we will need to manage this integration. In our web client Google Earth will do this integration for us.

5.2.1 Terrain Preparation

We cannot found a tool that easily let us compute a 3D georeferenced terrain from a set of points and store it tilled or not in a Postgres/PotsGIS database. Initially we expected that Geospatial Data Abstraction Library (GDAL) [36] will give us that functionality. Instead of building a custom script for our specific use case we create an extension to GDAL build on top of Computational Geometry Algorithms Library (CGAL) [1] with the capability to manage 3D georeferenced terrains.

Must of the rendering techniques for tilled terrains implies a client that can merge tiles with different LODs at runtime. Ideally a scene returned by a *GetScene* request should already by ready to visualization with no extra processing on the client side. Our algorithm guarantees that all tiles will perfectly fit together at any resolution.

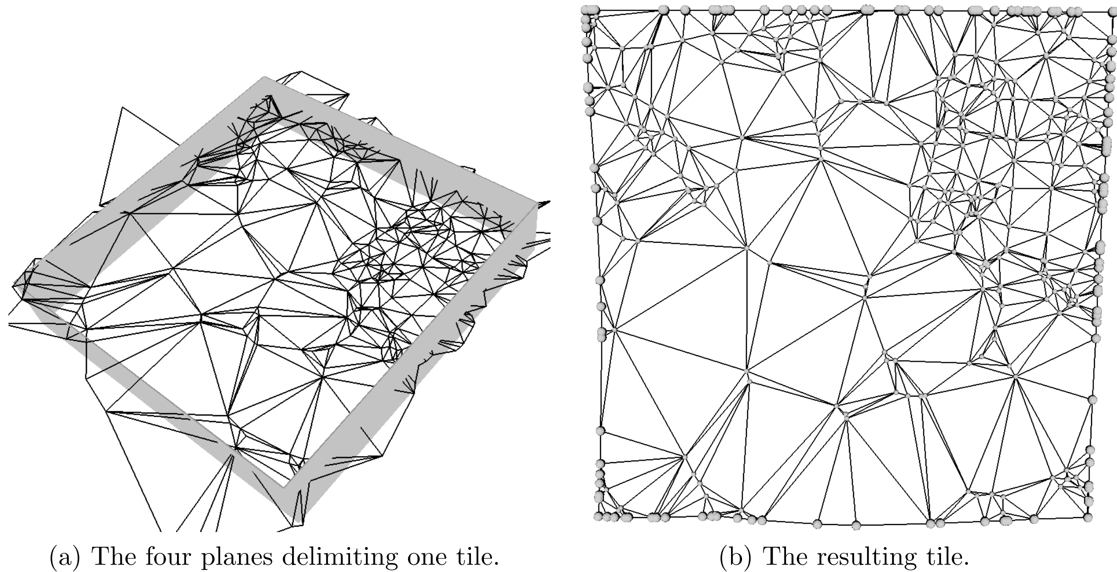

(a) The four planes delimiting one tile. (b) The resulting tile.

Figure 5.2: Cutting tiles.

Basic Tiling Algorithm

Although we can create tiles from any source supported by GDAL, we will describe our algorithm starting from the simplest source, which is a list of points with elevation.

The first step is to create the Triangle Irregular Network (TIN) from the points, using the Delaunay triangulation. This triangulation is done using the CGAL library. The Delaunay triangulation in CGAL is high configurable. For the TIN generation, we only need the 2.5 properties of the terrain, and the simple Delaunay triangulation applied to 2.5 data is adequate and the fastest approach.

The generated TIN can became quite large. In our approach, as we will show, we need to calculate the overall TIN. It is necessary to calculate this large TIN before dividing it into tiles.

To divide the TIN in tiles, we use 4 vertical planes to cut each tile. These planes start from the ground (elevation zero) and go to the highest possible elevation. Interception points are calculated. These are points on the edges that cross the vertical planes. This process is illustrated in *Figure 5.2*.

Special care must be taken to create the four corners. These corner points are the ones in the vertical line where the vertical planes intercept. The corner is the point where

that vertical line intercepts the TIN. It might be on a triangle that has no points inside the tile. The same corner can be shared by 4 different tiles.

All interceptions points will be used by both adjacent tiles divided by the same plane. If we keep these points in each tile, we guarantee that they will stitch perfectly.

Auxiliary Data Structure

The tiling algorithm described can scale quite well, maintaining a constant time per tile, if an adequate data structure is provided. We created a simple spatial index, called *InitialGrid*, to access all triangles that might be within one or more adjacent tiles. Using this index, for each tile, we only intercept the four planes with a small fraction of all triangles. The amount of time to cut each tile is thus constant, with a complexity of $O(1)$, in the big-O notation.

Joining Tiles Generated At Different Levels

Tiles can be hierarchically organized. One tile can be split into four other tiles, occupying the same surface, but with higher accuracy (with more mass points or triangles within the same area). The notion of level is independent of the Level Of Detail (LOD) used in the *GetScene* operation.

Tiles can and should be generated at different levels. To do so, we start with the most accurate data. The described algorithm is used to provide the tiles for the higher level. Only afterwards, the next lower level of tiles is computed. The next lower level will occupy the area of four existing tiles. The interception points calculated in the previous levels around each four tiles are preserved. All other mass points within the four tiles are used to compute the lower level tile. So, the Delaunay triangulation is done to compute the lower level TIN, considering less mass points, but preserving all points of the border. For each lower level, the user can decide how many mass point are discarded (values like 1/4 or even 1/8 have been used, preserving the surface shape, while significantly reducing the number of triangles in each tile.

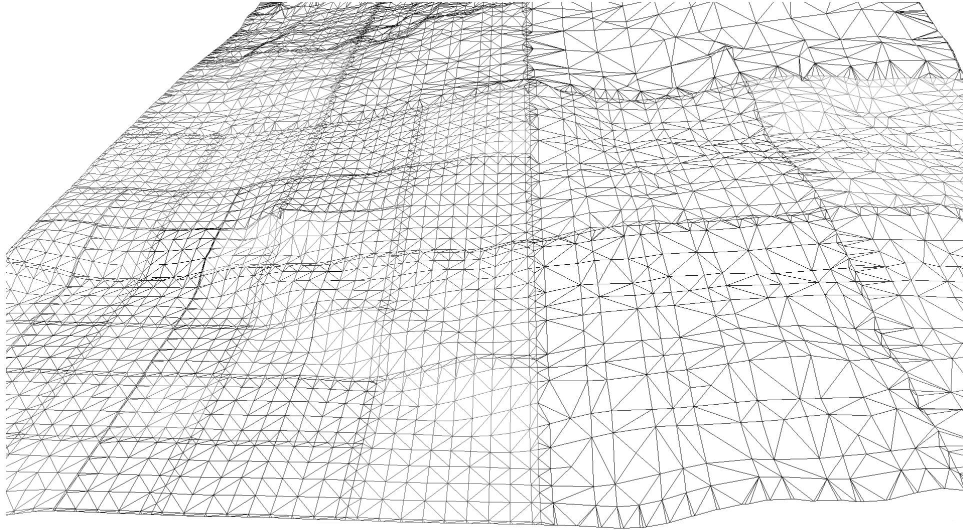


Figure 5.3: Perfect composition of several tiles, at different resolutions.

With such an algorithm, we preserve the points used in different tile levels. If the points are preserved, tile stitching will be perfect even when we put tiles from different levels side by side, as shown in *Figure 5.3*.

Tile Storage

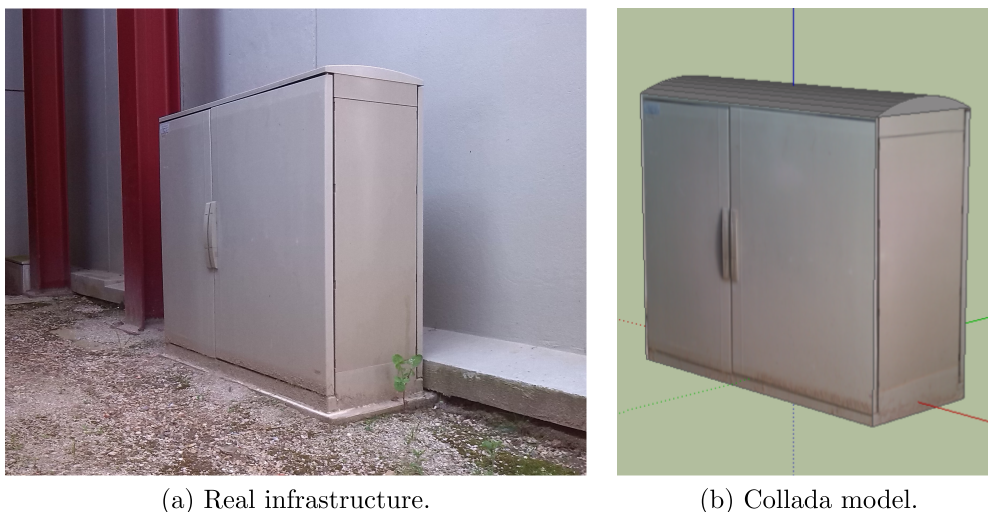
After being calculated, each tile is stored in a spatial database. Each tile is a row in the database and can be retrieved by its level, row and column number as keys. Alternatively, tiles can be stored as files, and served from the file system. The hierarchical organization of the file system can use the three different keys (level, row and column) to organize the tiles in folders within folders.

5.2.2 Preparation of 3D Features

We receive buildings and telecommunication infrastructures represented as 2D georeferenced geometries stored in a Postgres/PotsGIS database. For every building we have is base shape represented as a multi-polygon geometry and is height. We create some PL/SQL functions that make the extrude of the buildings. Using this approach we obtain a 3D representation for every building encoded in a multi-polygon geometry.

Telecommunication infrastructures was represented as 2D lines or 2D points, i.e. cables are represented as lines and the others infrastructures as points. Using the library we develop to manage terrains we give a 3D representation to every type of cable. The 3D representation for cables will only be useful in X3D format, Google Earth will ignore it and will positioning the cables using is own algorithms.

Every infrastructure represented as a 2D point can be represented as a 3D model. For every infrastructure we produce two models. The first is a 3D representation encoded using 3D primitives geometries that are directly stored in the database. The second representation is encoded in Collada and is more detailed. The Collada models have been produced using *Google SketchUp 7* (*Figure 5.4*).



(a) Real infrastructure. (b) Collada model.

Figure 5.4: Comparison between an infrastructure and is Collada model.

5.3 GeoServer Configuration

To fully support 3D, new options were added to the GeoServer administration interface, as described in 4.4.5. In this section we illustrate the use of such options.

The *Figure 5.5* show the configuration of W3DS service. In *Figure 5.6* we can see the edition of an SLD with 3D properties. Note that a shortcut for W3DS service is also available on the main menu on the left.

Figure 5.5: Configuration of W3DS service.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<StyledLayerDescriptor version="1.0.0"
    xsi:schemaLocation="http://www.opengis.net/sld StyledLayerDescriptor.xsd"
    xmlns="http://www.opengis.net/sld"
    xmlns:ogc="http://www.opengis.net/ogc"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <NamedLayer>
    <Name>IGP Texture</Name>
    <UserStyle>
      <Title>IGP Texture</Title>
      <FeatureTypeStyle>
        <Rule>
          <Name>DEM</Name>
          <Title>DEM</Title>
          <PolygonSymbolizer>
            <Fill>
              <DiffuseColor>0 1 0</DiffuseColor>
              <TextureUrl>
                <ogc:PropertyName>texture_igp</ogc:PropertyName>
              </TextureUrl>
            </Fill>
          </PolygonSymbolizer>
        </Rule>
      </FeatureTypeStyle>
    </UserStyle>
  </NamedLayer>
</StyledLayerDescriptor>
```

Figure 5.6: Edition of an SLD with 3D properties.

5.4 GetCapabilities Request

The next listings show the most interesting parts of a W3DS *GetCapabilities* response.

Listing 5.1: W3DS service meta-data.

```
1  <ows:ServiceIdentification>
2   <ows:Title>GeoServer Web 3D Service</ows:Title>
3   <ows:Abstract>A Web 3D Service Implementation.</ows:Abstract>
4   <ows:ServiceType>OGC W3DS</ows:ServiceType>
5   <ows:ServiceTypeVersion>0.4.1</ows:ServiceTypeVersion>
6   <ows:Fees>NONE</ows:Fees>
7   <ows:AccessConstraints>NONE</ows:AccessConstraints>
8  </ows:ServiceIdentification>
```

Listing 5.2: *GetCapabilities* operation meta-data.

```
1  <ows:OperationsMetadata>
2   <ows:Operation name="GetCapabilities">
3    <ows:DCP>
4     <ows:HTTP>
5      <ows:Get xlink:href="http://3dwebgis.di.uminho.pt/geoserver3D/ows?">
6       <ows:Constraint name="GetEncoding">
7        <ows:AllowedValues>
8         <ows:Value>KVP</ows:Value>
9        </ows:AllowedValues>
10      </ows:Constraint>
11     </ows:Get>
12    </ows:HTTP>
13   </ows:DCP>
14  </ows:Operation>
```

Listing 5.3: Description of a W3DS tiled layer.

```
1  <w3ds:Contents>
2   <w3ds:Layer>
3    <ows:Title>tiled_dtm</ows:Title>
4    <ows:Abstract>Tiled DTM</ows:Abstract>
5    <ows:Identifier>geoserver3D:tiled_dtm</ows:Identifier>
6    <ows:BoundingBox crs="EPSG:27492">
7     <ows:LowerCorner>-11000.0 198000.0</ows:LowerCorner>
8     <ows:UpperCorner>-10000.0 199000.0</ows:UpperCorner>
9    </ows:BoundingBox>
10   <ows:OutputFormat>model/x3d+xml</ows:OutputFormat>
11   <ows:OutputFormat>text/html</ows:OutputFormat>
12   <w3ds:DefaultCRS>EPSG:27492</w3ds:DefaultCRS>
13   <w3ds:Queriable>true</w3ds:Queriable>
14   <w3ds:Tiled>true</w3ds:Tiled>
15   <w3ds:TileSet>
16    <ows:Identifier>dtm_tileset</ows:Identifier>
17    <w3ds:CRS>EPSG:27492</w3ds:CRS>
18    <w3ds:TileSizes>490.0</w3ds:TileSizes>
19    <w3ds:LowerCorner>-17046.156 193553.047</w3ds:LowerCorner>
20   </w3ds:TileSet>
21   <w3ds:Style>
22    <ows:Identifier>dtm_gray</ows:Identifier>
23    <w3ds:IsDefault>true</w3ds:IsDefault>
24   </w3ds:Style>
25   <w3ds:Style>
26    <ows:Identifier>dtm_texture_osm</ows:Identifier>
27    <w3ds:IsDefault>false</w3ds:IsDefault>
28   </w3ds:Style>
29  </w3ds:Layer>
```

5.5 GetTile Request

The *Figure 5.7* and *Figure 5.8* present two tiles obtained from W3DS using the *GetTile* operation. Both represent the same tile with different textures. The texture was changed requesting the tiles with different styles. The URL of the textures are WMS requests, i.e. the client request at runtime the textures from a WMS server. The two tiles are encoded in X3D and have been render on Google Chrome web browser using X3DOM library.

Figure 5.7: Tile obtained using W3DS *GetTile* operation. The aerial image texture is obtained from a WMS server.

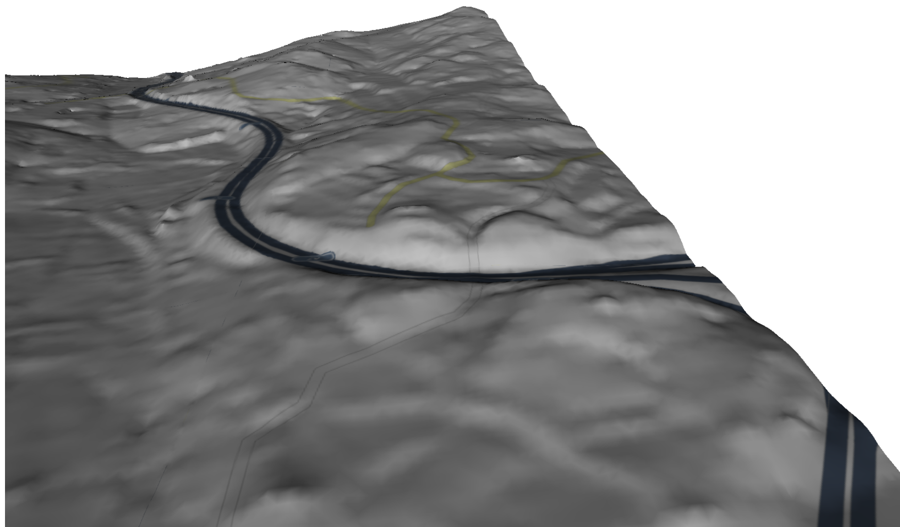


Figure 5.8: Tile obtained using W3DS *GetTile* operation. The OSM image texture is obtained from a WMS server.

5.6 GetScene Request

The next figures present different views of our use case. All of them have been obtained using W3DS *GetScene* operation.

The scenes showed in sub-section 5.6.1 have been produced using vectorial 3D data retrieved from the database. The only elements that are not retrieved from the database are the terrain textures. The scenes are encoded in X3D and no styling transformation have been used, i.e. the absolute values of the retrieved geometries are used. Since only vectorial information retrieved from the database is used, all the scenes are encoded using X3D simple geometries.

In sub-section 5.6.2 we present some scenes obtained using our web client developed on top of Google Earth web plugin. Except the cables all the others infrastructures are represented as Collada models. The cables are represented as simple lines and positioned using SLD properties. The scenes are encoded in KML and we only reference the Collada models using an HTTP URL. Our Collada models are stored in the file system and served by an Apache server. The Google Earth plugin is responsible to completely manage the Collada models.

Our web client provide two methods for requesting the 3D infrastructures from the server. In the default one we simply navigate on the globe and based on our position W3DS *GetScene* requests are performed. The 3D scenes returned by the server are integrated on the Google Earth globe providing a 3D view of the telecommunications infrastructures. In the second method we can directly select the area where we want to see the 3D infrastructures. All the remaining 3D infrastructures on the globe are removed and a *GetScene* request is performed for the selected area. The returned scene is integrated on the globe and our view point is updated.

5.6.1 X3D

Figure 5.9: Aerial cables are represented as red lines. The 3D buildings, the aerial cables and the OSM texture fit together perfectly.

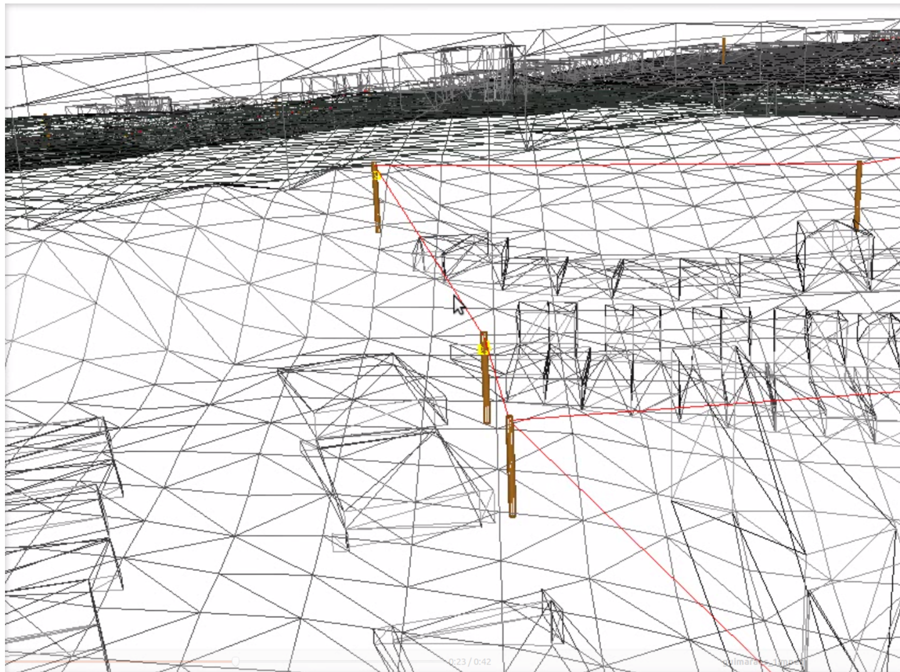


Figure 5.10: In the wire frame mod we can see all the geometries that have been retrieved from the database. We can also see that the telecommunication infrastructures and the buildings are correctly positioned on the DTM.

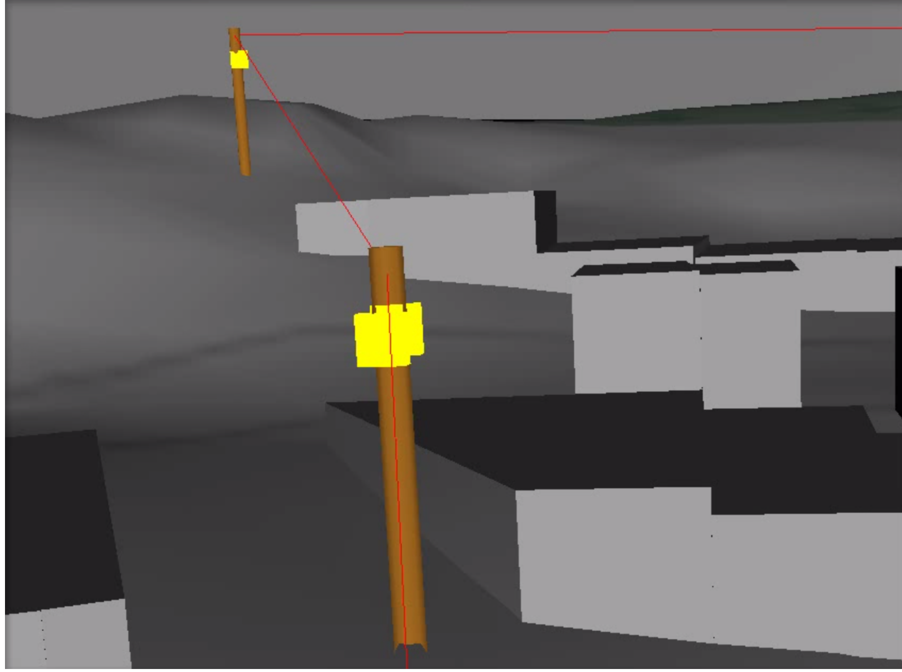


Figure 5.11: In this scene we can visualize the correct positioning of several telecommunication infrastructures. The front pole have two distributions boxes on his top that don't overlap. An aerial cable connect the two poles.

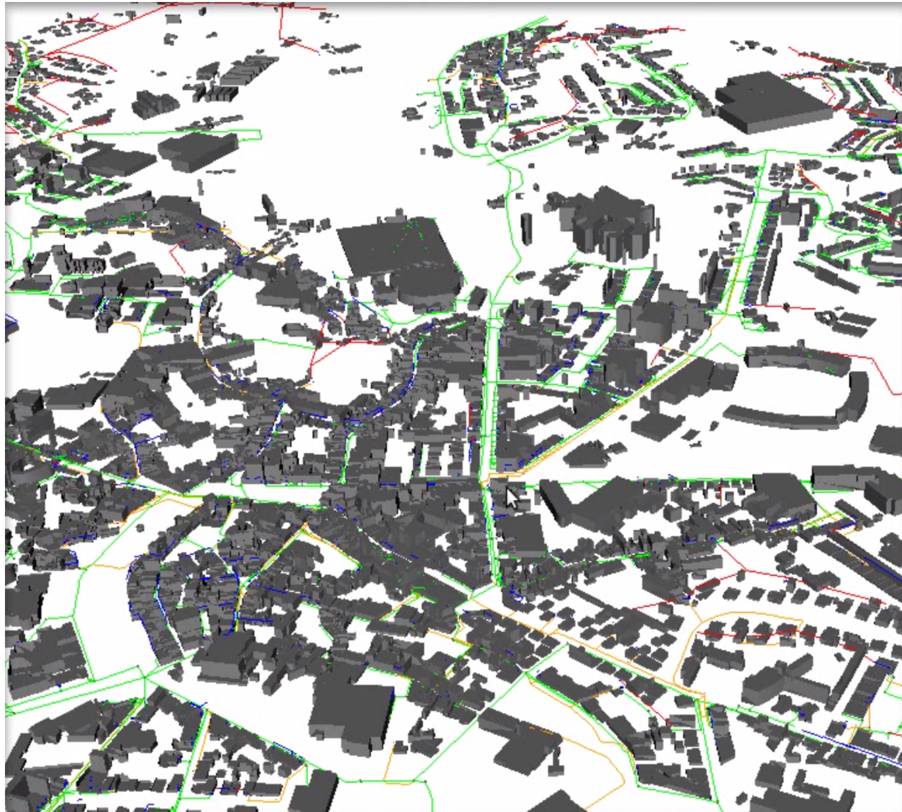


Figure 5.12: Scene without a DTM. In green we can see the underground conducts. We can see that the conducts follow the city streets.

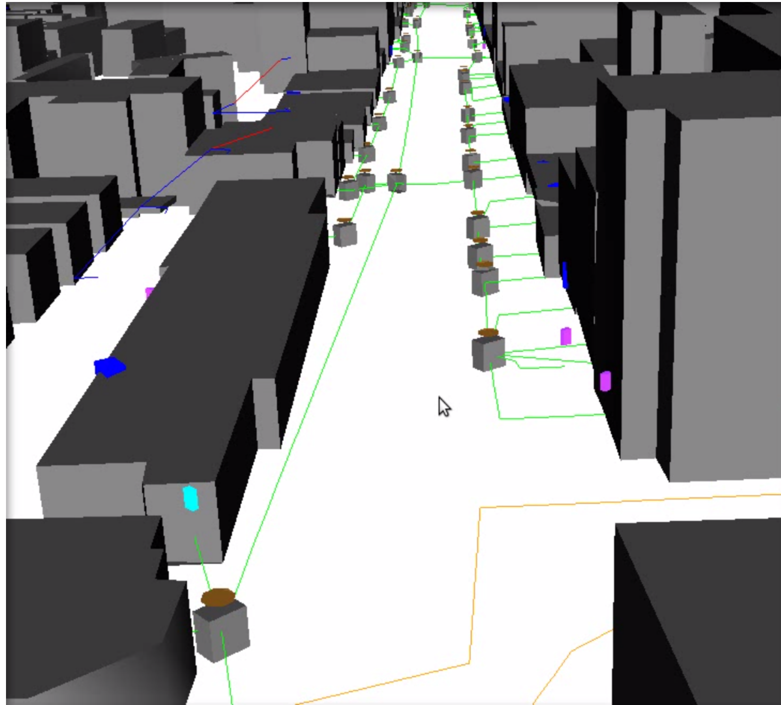


Figure 5.13: In this scene we can some underground connected telecommunications infrastructures.

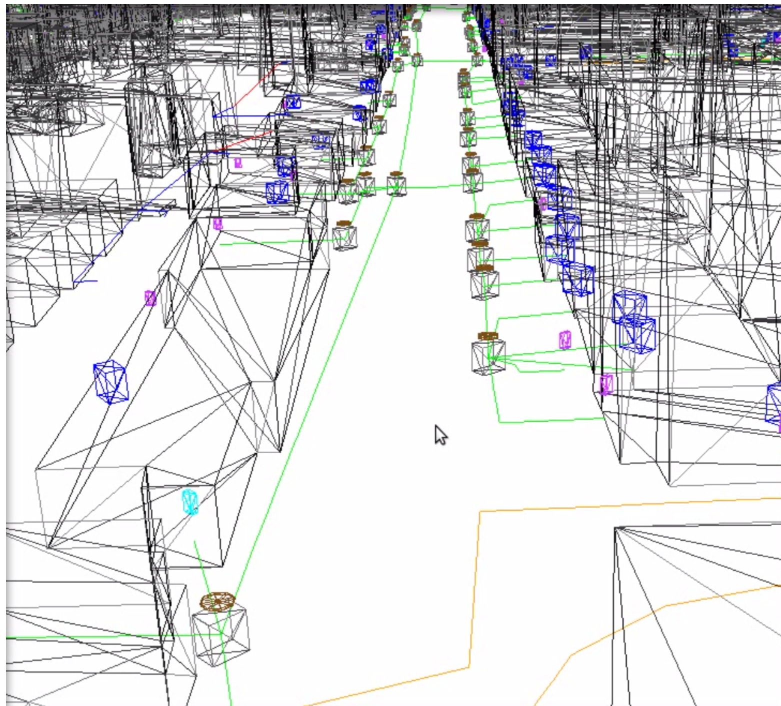


Figure 5.14: Wire frame mode where we can see the telecommunication infrastructures inside the buildings. Note that all the elements are connected to the network by conducts.

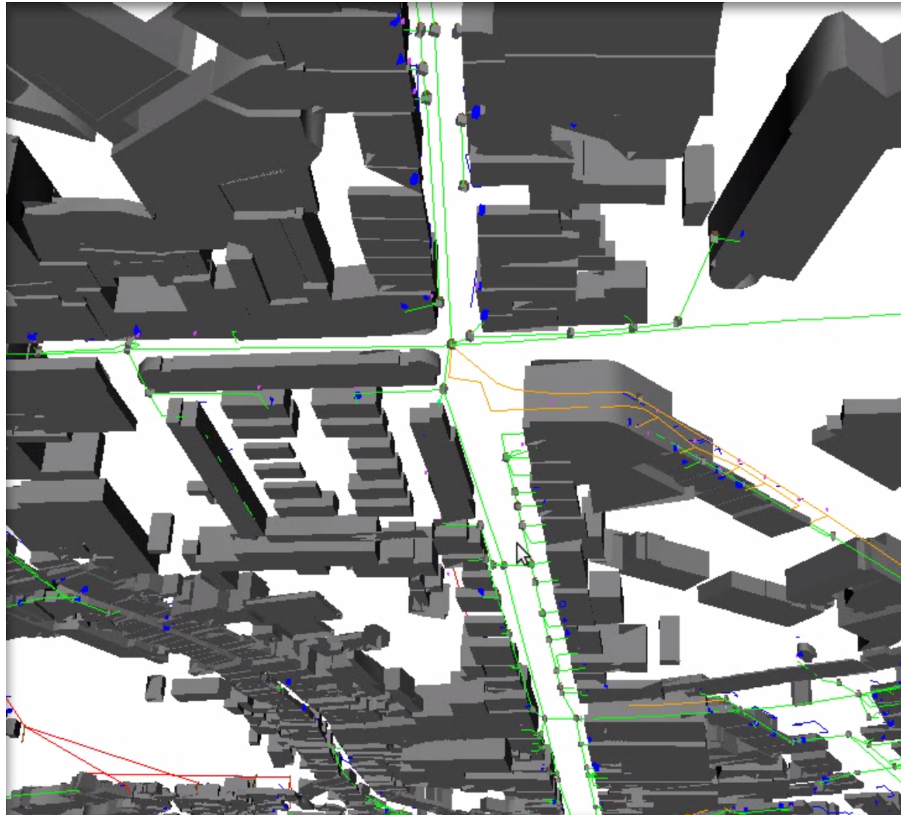


Figure 5.15: In this scene we can see the integration between the underground infrastructures and the city buildings.

5.6.2 KML

Figure 5.16: The yellow bounding box identifies the area where we want to see 3D infrastructures. A *GetScene* request will be performed and the returned scene will be integrated in the globe.

Figure 5.17: In this scene we can see telecommunications infrastructures integrated with Google Earth globe.

Figure 5.18: The detail of an infrastructure Collada model. The real infrastructure can be seen in *Figure 5.4*.

Figure 5.19: In this scene we can see the perfect integration between three telecommunications infrastructures and a Google Earth building.

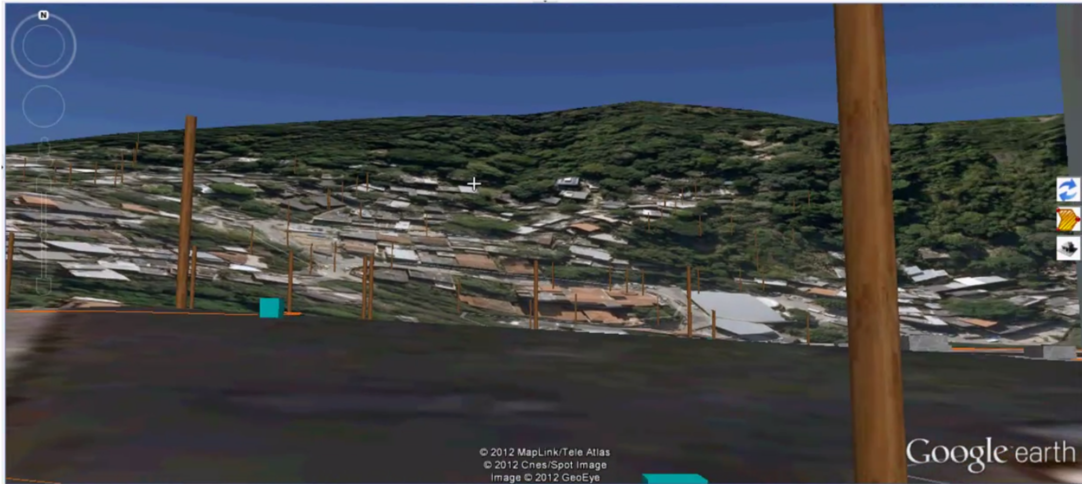


Figure 5.20: Several poles positioned on Google Earth DTM.

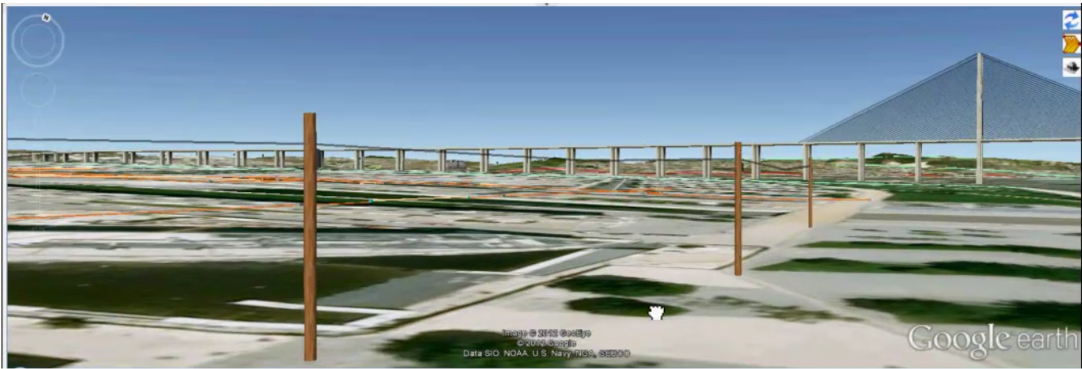


Figure 5.21: Several poles connected by aerial cables.

Figure 5.22: Perfect integration between four telecommunications infrastructures.

Chapter 6

Conclusion

We have presented why Geographic Information Systems (GIS) related standards have appear and why they are fundamental to guarantee interoperability and investment in application development. The lack of standards that fully supports the 3D GIS requirements is responsible for a chaotic situation of lots of 3D GIS applications that can't work together. In a real use case, no one will use a GIS system with very specific capabilities and zero interoperability.

We presented the Web 3D Service (W3DS) which is the most advanced standard related to 3D GIS. However even if the W3DS specification have been released some years ago it is still a draft. With no open source implementation which everyone can use and customize, is difficult to have a community that will provide fundamental feed back for the evolution of the specification.

In this project, we developed an open source implementation of a W3DS build on top of GeoServer. Besides the technological advantages of GeoServer, the feedback and the support from its community turned to be a very good choice.

Our implementation was adopted as community module by GeoServer.

During this project, we also prepared a use case related with 3D telecommunication infrastructures.

From this use case, several lessons were learned.

We identify some bottlenecks in the 3D GIS pipeline. We spend most of the time on data preparation. Managing 3D terrains is still a complex and a fastidious task. In a production environment the only viable client option is Google Earth consuming KML produced by W3DS. X3DOM geospatial support is in development stage and still having several problems.

Storing 3D data in a relational database can be useful, but for large data sets it can become a bottleneck. 3D models should be managed by clients, i.e. the produced 3D formats should only reference 3D models and it is client responsibility to manage and integrate that models in their rendering process. 3D models can be stored in a file system or in some document driven database like MongoDB. The chosen format to represent the 3D models should also provide a tool that help users creating their 3D models.

6.1 Publications

The results of this work were presented in two national conferences (SASIG 2012 and SASIG 2013) and four international conferences (SIG Libre 2012, AGILE 2012, CUPUM 2013 and ICCSA 2013). From these presentations, two publications were accepted and published by Springer, in the LNCS series, [27, 26].

6.2 Future work

Future works on W3DS module fall on two categories. The first ones are related with Open Geospatial Consortium (OGC) standards. i.e. W3DS and is related standards. The second ones are related with vendor options and will manly inside on the response formats. The second category will depend on the evolution of 3D GIS clients and 3D web formats.

We need to improve our implementation of W3DS specification. We only support some few optional parameters of W3DS operations. We also need to discuss if some

of that parameters should be removed from the specification, like the ones related with rendering properties. Currently we extend the Styled Layer Descriptor (SLD) capabilities to support 3D needs. We need to review and implement 3D SLD specification.

Recently X3DOM geospatial support have been improved and Blender now supports the export to X3D (*Figure 6.1*). We need to provide a response format in XML3D. KML response format is very basic we need to improve that support.

Currently OpenLayers 3 (OL3) is been developed. This is a complete rewrite of OpenLayers library with a modern and flexible approach. OL3 also comes with a set of new functionalities, one of them is the support of 3D content. The current support for 3D is built on top of Cesium. We need to provide a W3DS integration and implement a render build on top of X3DOM library.

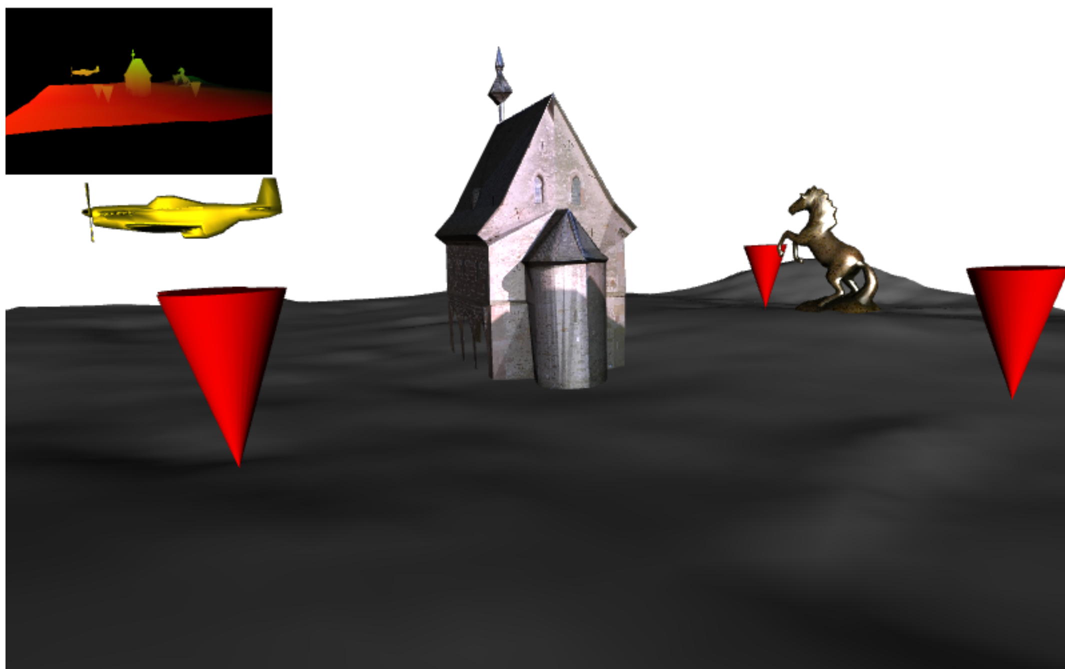


Figure 6.1: W3DS *GetScene* result renders by X3DOM library on Google Chrome. The models are positioned using 3D SLD. The horse model was exported from Blender.

Bibliography

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[2] Thomas Altmaier, Angela Kolbe. Applications and solutions for interoperable 3d geo-visualization. 2003.

[3] Jens Basanow, Pascal Neis, Steffen Neubauer, Arne Schilling, and Alexander Zipf. Towards 3d spatial data infrastructures (3d-sdi) based on open standards — experiences, results and future issues. 2008.

[4] Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. X3DOM - A DOM-based HTML5/ X3D Integration Model. 2009.

[5] Don Brutzman and Leonard Daly. *X3D: extensible 3D graphics for Web authors.* 2007.

[6] Béatrix Cambray. Three-dimensional (3D) modeling in a geographical database. 1993.

[7] Alessandro Carosio. La troiséme dimension dans les systémes d'information geographique et la mensuration officielle, 1999.

[8] The Open Geospatial Consortium. The havoc of non-interoperability, 2004.

[9] The Open Geospatial Consortium. The importance of going "open", 2005.

[10] The Open Geospatial Consortium. OpenGIS® Web Map Server Implementation Specification, 2006.

[11] The Open Geospatial Consortium. OpenGIS® Geography Markup Language (GML) Encoding, 2007.

[12] The Open Geospatial Consortium. Styled Layer Descriptor profile of the Web Map Service Implementation Specification, 2007.

[13] The Open Geospatial Consortium. OGC® Kml, 2008.

[14] The Open Geospatial Consortium. Draft for Candidate OpenGIS Styled Layer Descriptor profile of 3D Portrayal Services Standard, 2009.

[15] The Open Geospatial Consortium. OpenGIS® Web Feature Service 2.0 Interface Standard, 2010.

[16] The Open Geospatial Consortium. Ogc web services common standard, 2011.

[17] The Open Geospatial Consortium. OGC® Reference Model, 2011.

[18] Patrick Cozzi and Kevin Ring. *3D Engines Design for Virtual Globes*. 2011.

[19] Marcus Goetz and Alexander Zipf. OpenStreetMap in 3D - Detailed Insights on the Current Situation in Germany. 2012.

[20] Benjamin Hagedorn. Web View Service Discussion Paper, 2010.

[21] Georg Held, Alias Abdul-Rahman, and Siyka Zlatanova. Web 3D GIS for urban environments. 2001.

[22] B. Loesch, M. Christen, and S. Nebiker. OpenWebGlobe - An Open Source SDK For Creating Large-Scale Virtual Globes On A WebGL Basis. 2012.

[23] Chris Marrin. Webgl specification, 2013.

[24] J. Moser, F. Albrecht, and B. Kosar. Beyond visualization - 3D GIS analyses for virtual city models. 2010.

[25] Masahiko Murata. 3D-GIS Application for urban planning based on 3D city model. 2005.

[26] Nuno Oliveira and Jorge Gustavo Rocha. Tiling 3D Terrain Models. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, Carmelo M. Torre, Hong-Quang Nguyen, David Taniar, Osvaldo Gervasi, and Bernady O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2013*, volume 7971 of *Lecture Notes in Computer Science*, pages 550–561. Springer-Verlag Berlin, 2013.

[27] Nuno Oliveira and Jorge Gustavo Rocha. Web 3D Service Implementation. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, Carmelo M. Torre, Hong-Quang Nguyen, David Taniar, Osvaldo Gervasi, and Bernady O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2013*, volume 7973 of *Lecture Notes in Computer Science*, pages 538–549. Springer-Verlag Berlin, 2013.

[28] Sixto Ortiz. Is 3D Finally Ready for the Web? 2010.

[29] Mark Pesce. *VRML: Browsing and Building Cyberspace*. 1995.

[30] Thomas Quadt, Udo Kolbe. Web 3d service, 2005.

[31] Thomas Schilling, Arne Kolbe. Draft for Candidate OpenGIS Web 3D Service Interface Standard, 2011.

[32] Olaf Schroth, Ellen Pond, Cam Campbell, Petr Cizek, Stephen Bohus, and Stephen R. J. Sheppard. Tool or Toy? Virtual Globes in Landscape Planning. 2011.

[33] Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. XML3D: interactive 3D graphics for the web. 2010.

[34] Kristian Sons and Philipp Slusallek. Demo: XML3D - Interactive 3D Graphics for the Web. 1993.

[35] Benjamin T. Tuttle, Sharolyn Anderson, and Russell Huff. Virtual globes: An overview of their history, uses, and future challenges. 2008.

[36] Frank Warmerdam. GDAL, Geospatial Data Abstraction Library. http://www.gdal.org.

[37] Peter Zeile, Ralph Schildwächter, Tony Poesch, and Pierre Wettels. Production of virtual 3D city models from geodata and visualization with 3D game engines . A Case Study from the UNESCO World Heritage City of Bamberg Problem Status / Starting Point. 2004.

[38] Siyka Zlatanova. VRML for 3D GIS. 1997.