TECHNICAL UNIVERSITY OF CRETE

Design and implementation of a spatially-enabled wiki

by

Dimitrios Strevinas

A thesis submitted in partial fulfillment for the
Diploma of Computer Engineering

in the
Department of Electronics and Computer Engineering

May 2009

TECHNICAL UNIVERSITY OF CRETE

Abstract

Department of Electronics and Computer Engineering

Diploma of Computer Engineering

by Dimitrios Strevinas

The availability of large volumes of digital spatial content in modern applications, local datastores (e.g.,institutions and organization intranets) and the internet has generated interest in methods and tools for effective management of the shared content. Spatial information access is divided by thematic interest across a vast amount of web places, institutions and agencies, reducing the reachability and creativity that can emerge by combining multiple sources of geospatial content. Additionally, it is up to the provider, whether spatial content is described in a complete manner or demonstrated by means of rendered graphic samples.

We propose an approach, concerning the management of spatial information, based on the integration of the wiki architecture, state–of–the–art web–mapping applications and mature and major geospatial libraries. Targeting at the centralization of content within wiki systems and promoting their collaborative nature, such an architecture could benefit spatial analysis, spatial decision support and, of course, education. Thus the emerged project, named GeoMoin, is a modular collaborative web–mapping application, developed to achieve these goals.

Acknowledgements

This dissertation could not have been completed without the help of many people. I am foremost grateful to my supervisor, Professor Vasilis Samoladas, for assigning this project which introduced me to the world of GIS, the encouragement and the confidence he showed to me and to my work.

Special thanks are given to Thomas Waldmann and Radomir Dopieralski from the MoinMoin team; Steve Lime, Frank Warmerdam, Paul Ramsey, Tom Kralides, Daniel Morissette, Aaron Racicot, Howard Butler, Sean Gillies and Yewondwossen Assefa from the OSGeo and GIS Python for their invaluable and tireless support, ideas and corrections over the long days in the IRC channels.

Above all, my deepest thanks go to my family, my friends and Maria for their love, support and encouragement in all aspects of my life.

Contents

Abstract ii

Acknowledgements iii

List of Figures viii

Abbreviations x

1 Introduction 1
- 1.1 Motivations . . . 2
- 1.2 Contributions . . . 3
- 1.3 Thesis structure . . . 4

2 Wikis and MoinMoin 5
- 2.1 Introduction to wiki concepts . . . 5
  - 2.1.1 What is a wiki . . . 5
  - 2.1.2 Editing, linking and searching in a wiki environment . . . 6
  - 2.1.3 Controlling changes . . . 7
  - 2.1.4 Trust and security within a wiki . . . 8
  - 2.1.5 Architectures of wikis . . . 9
- 2.2 The MoinMoin wiki . . . 10
- 2.3 MoinMoin features . . . 10
- 2.4 MoinMoin architecure . . . 12
- 2.5 Extensions within MoinMoin . . . 13
  - 2.5.1 Actions . . . 14
  - 2.5.2 Macros . . . 16
  - 2.5.3 Parsers . . . 19
  - 2.5.4 Formatters . . . 22
  - 2.5.5 Themes . . . 23
- 2.6 XML-RPC and WikiRPC . . . 23
- 2.7 MoinMoin event sequence . . . 26

3 GIS concepts and web–mapping 30
- 3.1 A GIS overview . . . 30
- 3.2 Spatial data models . . . 32

3.2.1 Vector data model . . . 32
3.2.2 Raster data model . . . 34
3.2.3 Image data . . . 36
3.2.4 Data accuracy and quality . . . 37
3.3 Important GIS issues . . . 40
3.3.1 Organizing non–spatial data . . . 40
3.3.1.1 Tabular models . . . 40
3.3.1.2 Hierarchical models . . . 41
3.3.1.3 Network models . . . 41
3.3.1.4 Relational models . . . 41
3.3.1.5 Object–oriented models . . . 43
3.3.2 Map projections in overview . . . 43
3.3.2.1 Metric properties of maps . . . 44
3.3.2.2 Construction of a map projection . . . 45
3.3.2.3 Different projection surfaces and their "development" . . . 46
3.3.2.4 Projection definitions . . . 48
3.3.3 Geocoding principles . . . 49
3.3.4 Topological overlays . . . 50
3.4 WebGIS and Web mapping . . . 50
3.4.1 Types of web–mapping . . . 52
3.4.2 Advantages of web–mapping and WebGIS . . . 54
3.4.3 Disadvantages and problematic issues . . . 55
3.4.4 WebGIS and Spatial Desicion Support . . . 56

4 The MapServer Package 58
4.1 What is MapServer . . . 58
4.2 Setting the terrain for FOSS GIS development . . . 59
4.2.1 The Open Geospatial Consortium . . . 59
4.2.2 The Open Source Geospatial Foundation . . . 60
4.3 Intoducing MapServer capabilities . . . 61
4.4 Mapserver's configuration: The Mapfile . . . 63
4.4.1 A simple mapfile . . . 64
4.4.2 Associating datasets to a LAYER object . . . 66
4.4.3 Using projections within the mapfile . . . 69
4.5 Introducing MapScript . . . 73
4.5.1 MapScript objects discussion . . . 74
4.5.2 Rendering a map . . . 74
4.5.3 Accessing the features of a layer . . . 75
4.5.4 Computations on spatial features . . . 78
4.6 OGC web–services within MapServer . . . 79
4.6.1 The WMS service overview . . . 80
4.6.2 The WFS service overview . . . 81
4.6.3 Integration with MapServer . . . 84
4.6.3.1 Mapserver as WMS client . . . 84
4.6.3.2 Mapserver as WMS client . . . 85
4.6.3.3 Mapserver as WFS client . . . 87

5 The GeoMoin Web Application 89
5.1 Overview . . . 89
5.2 Design and implementation in overview . . . 90
5.2.1 Application architecture . . . 91
5.2.2 Overview of system components . . . 94
5.3 Filesystem data services: Requirements and Implementation . . . 95
5.3.1 Spatial data in the Filesystem and retrieval requiremenents . . . 96
5.3.2 Mapfiles storage and retrieval . . . 98
5.3.3 Markfiles and their usage . . . 99
5.4 DBMS data services: Installation and Implementation . . . 100
5.4.1 DBMS architecture in GeoMoin . . . 100
5.4.2 PostGIS extension for PostgreSQL . . . 101
5.4.3 PygreSQL PGDB module for DBMS access . . . 103
5.4.4 GeoMoin database schema specification . . . 104
5.4.4.1 Implementing the schema on PostgreSQL . . . 108
5.4.5 GeoMoin DBMS Wrapper: pgUtils . . . 111
5.5 OWS data services . . . 116
5.6 Business tier . . . 116
5.6.1 InstallManager . . . 116
5.6.2 Uploading a dataset . . . 117
5.6.3 Invoking the installer action . . . 118
5.6.4 Installing the datasets using OgrTypes . . . 118
5.7 Dataset controlling system: LayerManager . . . 121
5.7.1 Querying and managing the installed datasets . . . 122
5.7.2 Generation a UMN mapfile layer definition . . . 124
5.7.3 Creating an annotation layer . . . 125
5.7.4 WMS/WFS access through LayerManager . . . 127
5.8 Map controlling system: MapManager . . . 128
5.8.1 Creating and editing a mapfile . . . 128
5.9 Spatial visualization system . . . 133
5.9.1 Definition and implementation of the renderer . . . 133
5.9.2 The map rendering process . . . 135
5.9.3 Managing the layer status and position before rendering . . . 137
5.9.4 Defining the extent and navigation tools . . . 139
5.9.5 Query capabilities . . . 144
5.9.6 Annotating a rendered map . . . 151
5.9.6.1 Enabling an annotation layer in the MapRenderer . . . 152
5.9.6.2 Annotation table structure . . . 153
5.9.6.3 Creating an annotation . . . 154
5.9.6.4 Displaying and managing annotations . . . 160
5.9.7 Using XML-RPC to manage the WIKI content . . . 165
5.10 GeoMoin mapplets . . . 166
5.10.1 Synchronous mapplets . . . 166
5.10.2 Asynchronous mapplets . . . 171
5.11 User services and AJAX . . . 178
5.11.1 MapManager . . . 181
5.11.2 LayerManager . . . 184

5.11.2.1 Querying and extracting dataset information 184
5.11.2.2 Creating annotation layers 186
5.11.2.3 WMS managing 187
5.11.3 Spatial Visualization System 191
5.11.3.1 Navigation tools and queries 192
5.11.3.2 Annotation interactions 194
5.11.3.3 Layer management and general information 195
5.11.3.4 Mapplet interactions 198
5.11.4 Interactions between text and maps 200

6 Conclusion **204**
6.1 Case–study: A map of greece 204
6.2 Summary of theoritical results 210
6.3 Future work 211
6.3.1 Global availability of GeoMoin 212
6.3.2 GeoMoin within a multi–server environment 212

A Technical Issues **214**
A.1 Installation 214
A.1.1 Pre–required general libraries 215
A.1.2 PostgreSQL and Postgis. 218
A.1.3 Apache web server 219
A.1.4 MapServer and GDAL 220
A.1.5 MoinMoin wiki 221
A.1.6 GeoMoin Installation 222

Bibliography **225**

List of Figures

3.1 Spatial data models . . . 32
3.2 Satellite imagery . . . 36
3.3 A relational table schema . . . 42
3.4 Linkages within a relational database file . . . 44
3.5 The cylindrical projection . . . 47
3.6 The conical projectial . . . 47
3.7 Azimuthal projections . . . 47
3.8 overlaying . . . 51
3.9 Web Mapping . . . 52

4.1 Mapserver architecture . . . 62
4.2 World map with lat/long . . . 70
4.3 Wisconsin area projections . . . 72
4.4 Mapfile basic structure . . . 74
4.5 WMS retrieved map . . . 81

5.1 3-Tier architecture . . . 92
5.2 DBMS access . . . 100
5.3 Geometry types . . . 102
5.4 Access control flowchart for a SELECT query. . . . 113
5.5 Access control flowchart for an INSERT query. . . . 113
5.6 Access control flowchart for an UPDATE query. . . . 115
5.7 Installing dataset to the filesystem . . . 119
5.8 Installing dataset to the database . . . 121
5.9 Ogrtypes structure . . . 124
5.10 Python dictionary containing parsed map . . . 130
5.11 Setting the layer status . . . 138
5.12 Tolerance and buffer size . . . 147
5.13 Annotation table registration . . . 151
5.14 Annotation table content . . . 153
5.15 An annotation popup . . . 165
5.16 Synchronous mapplet flowchart . . . 167
5.17 Asynchronous mapplet flowchart . . . 172
5.18 Classic Web Application Model versus AJAX Model . . . 179
5.19 ManagerManager panel: select map . . . 181
5.20 ManagerManager panel: edit raw . . . 182
5.21 ManagerManager panel: edit gui . . . 182
5.22 ManagerManager panel: edit layers . . . 183

5.23 ManagerManager panel: edit web object . . . 183
5.24 LayerManager panel: Search query panel . . . 184
5.25 LayerManager panel: Search query results . . . 185
5.26 LayerManager panel: Layer information . . . 185
5.27 LayerManager panel: Modifying layer info . . . 186
5.28 LayerManager panel: Adding annotation fieldnames . . . 187
5.29 LayerManager panel: Creating annotation table . . . 188
5.30 LayerManager panel: Annotation layer creation . . . 188
5.31 LayerManager panel: WMS widget . . . 189
5.32 LayerManager panel: WMS registration . . . 189
5.33 LayerManager panel: WMS demo connection . . . 190
5.34 SVS: layout . . . 191
5.35 SVS: rendering . . . 191
5.36 SVS: toolbox . . . 192
5.37 SVS: Query options . . . 193
5.38 SVS: query results . . . 194
5.39 SVS: Annotation options . . . 195
5.40 SVS: annotation insertion . . . 196
5.41 SVS: displaying an annotation . . . 196
5.42 SVS: information tab . . . 197
5.43 SVS: Layer managing tab . . . 197
5.44 SVS: Synchonous mapplet example . . . 198
5.45 SVS: Asynchonous mapplet part 1 . . . 199
5.46 SVS: Asynchonous mapplet part 2 . . . 199
5.47 SVS: Wiki tools . . . 200
5.48 Attaching a rendering to a page . . . 201
5.49 Displaying the attached map . . . 201
5.50 Attaching a BookMap to a page . . . 202
5.51 A gallery using BookMaps . . . 203

6.1 Case study: Attaching datasets . . . 205
6.2 Case study: Installing datasets . . . 205
6.3 Case study: Dataset information . . . 206
6.4 Case study: Make layer queryable . . . 207
6.5 Case study: Creating a label . . . 208
6.6 Case study: Importing layers . . . 208
6.7 Case study: Adding the map to wiki page . . . 209
6.8 Case study: Setting the map options . . . 209
6.9 Case study: Visualizing the map . . . 210

Abbreviations

FOSS	**F**ree and **O**pen **S**ource **S**oftware
GIS	**G**eographical **I**nformation **S**ystems
OGC	**O**pen **G**eospatial **C**onsortium
OSGeo	**O**pen **S**ource **Geo**spatial Foundation
OWS	**O**GC **W**eb **S**ervices
sDSS	**S**patial **D**ecision **S**upport **S**ystems
WebGIS	**W**orld-Wide W**eb** with **G**eographical **I**nformation **S**ystems
WCS	**W**eb **C**overage **S**ervice
WFS	**W**eb **F**eature **S**ervice
WKB	**W**ell **K**nown **B**inary (Representation)
WKT	**W**ell **K**nown **T**ext (Representation)
WMS	**W**eb **M**apping **S**ervice

... to my grandfather, Dimitris

Chapter 1

Introduction

In recent years, we have seen a tremendous explosion of digital spatial information available on the internet, local agencies, institutions and governments. Additionally online web–mapping applications like Google Earth, GoogleMaps and YahooMaps increased the public interest concerning what can be achieved through the use of digital cartography, while raising the standards of quality, accuracy of the content along with the funds, crucial for the maintainance of such state–of–the–art services. Moreover, the continuous advances of Free and Open Source software (FOSS) within the GIS terrain, make them a respective alternative to commercial high–cost applications. Web–mapping applications like MapServer, GeoServer and OpenLayers, spatial libraries like GDAL and GEOS along with powerful desktop applications like GRASS are particularly good examples of such software.

On the other hand, the digital spatial data is the most important, yet expensive part of a GIS[1]. The importancy lies in the fact that they serve as the input for the GIS applications. The expense can be attributed to the man–hours spent to be generated plus the huge cost of the equipment used to automate or faciliate the operation. While there are many private sector firms specializing in providing digital data, federal, provincial and state government agencies are an excellent source. Large costs associated with data capture and input, prove government departments as the only agencies with financial resources and manpower funding to invest in data compilation. Yet, the financing and technical expertise vary from one country to another, resulting in not enough attention being given to the quality of the data or the processes by which they are prepared. Additionally, spatial information, depending on the thematic content they cover, can be seen scattered loosely around local institutions, web datastores and web services[1] proving themselves:

[1]In example, OGC web services are a form of providing spatial content to the web in a predefined architectural manner

- Difficult to be located.
- Non–properly described.
- Non–qualified compared to the cost in order to obtained.

Fortunately, the advances in the incorporation of the GIS technlogies within the Web (not to mention the mobile devices), introducing the concept of WebGIS and web–mapping applications, has opened new markets and caught the attention of financiers resulting in the upgrowth of both the technologies and the methods used to produce and provide geospatial information. These efforts result in broad societal and commercial benefits in areas such as education, decision-making for a sustainable future, land-use planning, agricultural, and crisis management.

1.1 Motivations

In this section a list of important issues, concerning the need of an application like GeoMoin, is presented. The listing is by no means complete as long as different requirements are emerging over time, but surely addresses the initial motivations behind the development of the software.

- **Geospatial content is not easily accessible**; Many different providers exist but it is difficult to locate the appropriate one.
- **Geospatial content is expensive**; Accurate and qualified datasets are usually associated with high costs (e.g. satellite imagery). As long as the copyrights are preserved, a centralized web–place can exist where users contribute accurate, qualified, properly described and, above all, free spatial content.
- **Spatially–enabling a web site requires expertise and consideration of financial resources;** Today, spatial information is an important multimedia source of information which can fit within every modern organization's web–site. Designing the web architecture in order to support spatial content can become a difficult task that requires expertise and sometimes a considerable amount of time. Additionally, different solutions and experts can be employed for such case, requiring considerable financial resources.
- **Free and Open Source spatial applications can match commercial solutions;** Geographic Information Science is a field that is being benefited by the development and use of open–source packages. Complicated GIS application can

be developed and distributed at no cost leaving the financial need to the demanding part of creating the spatial information itself.

- **Access to easily configurable web–application promotes education;** Based on open–source solutions, an application fully functional within the web can exist, where users can learn about the different aspects of building–block applications like MapServer, create maps without diving in the technical configuration details, visualize datasets, develop new tools, and analyze geographic information.

1.2 Contributions

The GeoMoin project, described in this thesis report, is being developed in a continuous effort to assist the GIS community targeting at the motivations presented in the previous sections. Following the traces of the projects like the Wikipedia, we believe that the collaborative work within a properly developed context can achieve these goals.
Concerning spatial data acquisition and benefit, GeoMoin provides tools that allow its users to contribute content within its storage subsystems; content that can be tagged with metadata and described, discussed and maintained within the pages of the wiki system which supports the whole application. Users can submit queries, which on matching, return ready–to–use datasets (by this concept, layers) fullfilling their needs. These layers are properly constructed in a form that can be used *as–is* in order to be used in fully navigation thematic maps **within** the GeoMoin application. That is, datasets, apart from being downloaded, can be visualized within the context of the application by user–defined renderings. Additionally, GeoMoin provides a framework for embedding and visualizing OGC's OWS services and particularly WMS and WFS. By supporting this feature, remote datastores like NASA's Jet Propulsion Laboratory web services can be directly accessible providing hands–on rendered images and XML–formatted raw features respectively.
As stated before, spatial datasets can be directly embedded into fully–navigational and queryable maps within the wiki pages. By building configuration files, called *mapfiles*, the maps can be fully customized in terms of the facilities they offer, while proper wiki–markup configuration directives customize the linking between the pages of the wiki system and the maps themselves. The wiki pages are now automatically spatially–aware, allowing the system to host spatial information from web–sites and small organizations that do not have the sufficient funds and extertise to built or manage a fully functional web–mapping service within their web infrastructure.
As far as spatial desicion support facilities is concerned, the GeoMoin system is suitable for providing navigational and querying results over different spatial scenarios and case

studies, whilst user–developed modules, called *mapplets*, can fit in the context of the application providing tools able to spatially analyse geographic information and interactively intermediate in the rendering and navigational procedure. As soon as, GeoMoin fits natively within the MoinMoin[2] project, properly developed tools can include report extractions, and subsystems that can manage desicion support scenarios.

1.3 Thesis structure

The rest of this thesis is organized as follows:

Chapter 2 provides a review of the underlying web architecture behind the concept of wikis. It describes, in particular, MoinMoin, which is the wiki behind the GeoMoin application. The different methods, or modules, capable of expanding the wiki capabilities are described using examples.

Chapter 3 provides a introduction to the concept of Geographic Information Systems (GIS) analyzing the input data structures, projection, methodologies concerning spatial and non–spatial organizational modeling, methods for spatial analysis and finally the methods and requirements concerning the development of web–mapping applications.

Chapter 4 introduces the readed to the free and open source packages which are the building bocks behind the development of GeoMoin. The UMN Mapserver web–mapping framework is described in details along with the frequently used GDAL library.

Chapter 5 presents in full detail the architecture of the GeoMoin project in the first sections, while in the rest of the sections each module is analysed and the workarounds used are presented. The logical division of the chapter is based in the 3–tier architecture of GeoMoin being data storage tier, business logic tier and user services tier respectively.

Chapter 6 provides samples case studies, by using GeoMoin, along with performance discussion and results. It summarises the goals achieved and provides direction for future research and development.

Appendix A provides technical details over miscellaneous issues presented throughout the chapters along with installation instructions.

[2]MoinMoin is the wiki that supports the GeoMoin application and is described in Chapter 2

Chapter 2

Wikis and MoinMoin

2.1 Introduction to wiki concepts

The current chapter is dedication to a discussion concerning the logic of the online collaborative environments called **wikis** and, particularly, *MoinMoin* which will be used throughout this work. Starting at describing the concept behind the need of wikis, their architectures and design rules are mentioned [1] , while the specific issues and implementations within *MoinMoin* are as extensively described as possible.

2.1.1 What is a wiki

A wiki is a collection of web pages designed to enable anyone who accesses it to contribute or modify content. Ward Cunningham the developer of the first wiki software, WikiWikiWeb, describes the essence of the wiki concept as follows [3]:

- A wiki invites all users to edit any page or to create new pages within the wiki Web site, using only a plain-vanilla Web browser without any extra add-ons.
- Wiki promotes meaningful topic associations between different pages by making page link creation almost intuitively easy and showing whether an intended target page exists or not.
- A wiki is not a carefully-crafted site for casual visitors. Instead, it seeks to involve the visitor in an ongoing process of creation and collaboration that constantly changes the Web site landscape.

[1] Numerous elements of this chapter are attributed to the Wikipedia article on wikis [2].

Table 2.1: Wiki linking from HTML to wiki markup.

Wiki syntax	Equivelant HTML	Rendered output
"Take some more [[tea]]," the March Hare said to Alice, very earnestly.	"Take some more tea ," the March Hare said to Alice, very earnestly.	"Take some more tea," the March Hare said to Alice, very earnestly.

Every wiki engine introduces the wiki owner to numerous "plugins" which can used to mutate a classic wiki instance to a:

- **Content management system (CMS)** A web–location where users create, edit, manage and publish content in a consistently orgazinised fashion.
- **Concurrent version system (CVS)** A web–location that keeps track of all work and all changes in a set of files, and allows several developers (potentially widely separated in space and/or time) to collaborate
- **Blog space** Users can maintain their own blog or online diary in a wiki, or create a blog community
- **Social network** Web–location that can be used to enthrust social activities (eg. users can provide jobs and opportunities to other members of the wiki)

2.1.2 Editing, linking and searching in a wiki environment

Ordinarly the structure and formating of a wiki page is presented in a simple markup language, in order to provide a steep learing curve for non experienced users. This markup language, of course, is afterwards translated to html and presented to the user's browser. An example of a wiki syntax for linking is shown in table 2.1.2.

Although limiting access to HTML and Cascading Style Sheets (CSS) within wikis, limits user ability to alter the structure and formatting of wiki content, there are some benefits. Limited access to CSS promotes consistency in the look and feel and, sometimes, having JavaScript disabled prevents a user from implementing code, which may limit access for other users. Increasingly, wikis are making *WYSIWYG* ("What You See Is What You Get") editing available to users, usually by means of JavaScript or an ActiveX control that translates graphically-entered formatting instructions, such as "bold" and "italics", into the corresponding HTML tags or wikitext. In those implementations, the markup of a newly edited, marked-up version of the page is generated and submitted to the server

transparently, and the user is shielded from this technical detail. However, WYSIWYG controls do not always provide all of the features available in wikitext.
Wikis, ordinarly, keep track of the changes the user has made. Ofter, every version of the page is separetely stored. This means that authors can revert to an older version of the page, should it be necessary because a mistake has been made or the page has been vandalised.

Usually a wiki page contains a large number of links to other pages for the user to navigate through. This can be thought as a form of non–linear navigation which is generally "native" to wiki architectures as opposing to other web structured schemes of design. Wikis offer the *backlink* feature which provide a list of pages that link to the current page.
It is typical for a wiki page, to contain links to wiki pages that do not **yet** exist. This is a form of inviting users to create add content that needs to exist.
Links and pages in a wiki instance use the *CamelCase* format. The word "CamelCase" itself is in this format too. A word like that is produced by removing the spaces between the words and capitalising the first letters. The wiki system can, of course, split the words and lower the letters before presenting the user with the link, but generally this behaviour is not used.

As far as searching in wiki is concerned, the user is always presented with a **title search**, and sometimes with a **full–text search**. The desicions on whether a wiki will support advanced search features depends on whether the storage is based on a database or the filesystem. Generally filesystem–based wikis can use open–source libraries like *Xapian*[2] or *Lucene*[3] which provide full text indexing and searching capabilities. Alternatively, external search engines such as Google can sometimes be used on wikis with limited searching functions in order to obtain more precise results. However, a search engine's indexes can be very out of date (days, weeks or months) for many websites.

2.1.3 Controlling changes

Wikis are generally designed with the philosophy of making it easy to correct mistakes, rather than making it difficult to make them. Thus, while wikis are very open, they provide a means to verify the validity of recent additions to the body of pages. The most prominent, on almost every wiki, is the *Recent Changes* facility; a specific list

[2]Xapian is an open source probabilistic information retrieval library, released under the GNU General Public License (GPL). That is, it is a full text search engine library for programmers.

[3]Lucene is a free/open source information retrieval library, originally created in Java by Doug Cutting. It is supported by the Apache Software Foundation and is released under the Apache Software License. Lucene has been ported to programming languages including Delphi, Perl, C#, C++, Python, Ruby and PHP.

numbering recent edits, or a list of edits made within a given time frame. Some wikis can filter the list to remove minor edits and edits made by automatic importing scripts (bots).
From the change log, other functions are accessible in most wikis: the *revision history* shows previous page versions, while the *diff* feature highlights the changes between two revisions. Using the revision history, an editor can view and restore a previous version of the article. The diff feature can be used to decide whether or not this is necessary. A regular wiki user can view the diff of an edit listed on the "Recent Changes" page and, if it is an unacceptable edit, consult the history, restoring a previous revision; this process is more or less streamlined, depending on the wiki software used.
In case unacceptable edits are missed on the "recent changes" page, some wiki engines provide additional content control. It can be monitored to ensure that a page, or a set of pages, keeps its quality. A person willing to maintain pages will be warned of modifications to the pages, allowing him or her to verify the validity of new editions quickly.

2.1.4 Trust and security within a wiki

Critics of publically-editable wiki systems argue that these systems could be easily tampered with, while proponents argue that the community of users can catch malicious content and correct it. Lars Aronsson, a data systems specialist, summarizes the controversy as follows:

> " Most people, when they first learn about the wiki concept, assume that a Web site that can be edited by anybody would soon be rendered useless by destructive input. It sounds like offering free spray cans next to a grey concrete wall. The only likely outcome would be ugly graffiti and simple tagging, and many artistic efforts would not be long lived. Still, it seems to work very well. "

The open philosophy of most wikis, allowing anyone to edit content, does not ensure that every editor is well–meaning. Vandalism can be a major problem. In larger wiki sites, such as those run by the Wikimedia Foundation, vandalism can go unnoticed for a period of time. Wikis by their very nature are susceptible to intentional disruption, known as *trolling*[4] or to *WikiSpam*[5]. Wikis tend to take a soft security approach to

[4] An Internet troll, or simply troll in Internet slang, is a person who posts controversial and usually irrelevant or off-topic messages in an online community, such as an online discussion forum or chat room, with the intention of baiting other users into an emotional response or to generally disrupt normal on-topic discussion.

[5] WikiSpam is a wiki–wide problem. It won't be solved but wikiwide. These spammers add links to wikis, blogs, bulletin boards, and guestbooks so their Google *pagerank* increases.

the problem of vandalism; making damage easy to undo rather than attempting to prevent damage. Larger wikis often employ sophisticated methods, such as bots that automatically identify and revert vandalism and JavaScript enhancements that show characters that have been added in each edit. In this way vandalism can be limited to just "minor vandalism", where the characters added/eliminated are so few that bots do not identify them and users do not pay much attention to them.
The amount of vandalism a wiki receives depends on how open the wiki is. For instance, some wikis allow unregistered users, identified by their IP addresses, to edit content, whilst others limit this function to just registered users. Most wikis allow anonymous editing without an account, but give registered users additional editing functions; on most wikis, becoming a registered user is a short and simple process. Some wikis require an additional waiting period before gaining access to certain tools. For example, on the English Wikipedia, registered users can only rename pages if their account is at least four days old. Other wikis, such as the Portuguese Wikipedia, use an editing requirement instead of a time requirement, granting extra tools after the user has made a certain number of edits to prove their trustworthiness and usefulness as an editor. Basically, "closed up" wikis are more secure and reliable but grow slowly, whilst more open wikis grow at a steady rate but result in being an easy target for vandalism. A clear example of this would be that of Wikipedia and Citizendium. The first is extremely open, allowing anyone with a computer and internet access to edit it, making it grow rapidly, whilst the latter requires the users' real name and a biography of themselves, affecting the growth of the wiki but creating an almost "vandalism–free" ambiance.

Another workaround used by many wikis, including MoinMoin, is the presence of *Access Control Lists* (ACLs) in each wiki page. That is, the user can decide whether a page can be editable, by which users or by which groups when the wiki supports user groups.

2.1.5 Architectures of wikis

Wiki architecture is similar to any other software architecture, requiring a local network or "hosted" site to deliver the product to the customer or end user. Thus a wiki can be viewed as a web–location which can be hosted within the provider's hardware or within an intranet or local network. Sometimes users create "personal wikis" where the wiki is positioned as a personal knowledge management system, designed to run on a single computer and align to personal storage.
The wiki architecture is based on a server–side delivery of content, where the server provides the content, along with the editing tools and the various facilities. In most scenarios, wiki instances operate within a web–server content (such as Apache) while wikis like MoinMoin can additionally operate in a stand–alone mode serving as a daemon.

Moreover various architectures can be used to built and provide the wiki system, ranging from *cgi* (common gateway interface) and web–development languages like Python, PHP, ASP.NET, C# to *fastcgi*[6] and *mod_python*[7].

2.2 The MoinMoin wiki

The wiki serving the GeoMoin project is MoinMoin[8] [9]. Mainly developed by Jurgen Hermann and Thomas Waldmann this cross–platform wiki engine is implemented in Python and was initially based on the *PikiPiki* wiki engine. Distributed under the terms of the GNU General Public License, MoinMoin is free software. It is shipped with a feature set of fine grained access control, simple user groups, GUI editor, easy install, simple but efficient spam protection, easy theming combined with a simple code base making it often the wiki of choice for many open source projects (e.g. Apache, Debian, Ubuntu, Fedora) and corporate wikis. Moreover python language is well–known for its fast learning curve and high level programming facilities.
MoinMoin wiki has an extensive support of unicode pages, supporting multilanguage content and interface. There is an ongoing attempt by the MoinMoin community to translate both the interface and the documentation in many different languages.

2.3 MoinMoin features

MoinMoin support a vast number of features some of which are unique between the different wiki flavors[10]. As far as markup features for the wiki page editing is concerned, almost all the HTML markup directives are supported, including but not limited to headings, text forms (eg bold,italic), links, tables, lists e.t.c.

Base features

- Backups of all page revisions

[6]FastCGI is an open extension to CGI that provides high performance for all Internet applications without the penalties of Web server APIs.

[7]Mod_python is an Apache module that embeds the Python interpreter within the server. With mod_python developers can write web-based applications in Python that will run many times faster than traditional CGI and will have access to advanced features such as ability to retain database connections and other data between hits and access to Apache internals.

[8]http://moinmo.in/

[9]During the publication of this work the current version of MoinMoin package is 1.8.0

[10]The web sites http://moinmo.in/WikiEngineComparison and http://www.wikimatrix.org contain an extensive comparison on the supported features between many wiki engines, with the second being dedicating to this kind of requests

- Page revision list
- Diffs between arbitrary page versions
- Recent changes
- Subpages
- I18N (foreign and multi-language) support
- Unicode support, standard encoding is utf–8
- Lots of help pages
- Grouping of pages within categories
- RSS feed for RecentChanges
- Wiki page templates
- Configurable edit locking/warning to avoid editing conflicts
- Copying/deleting/renaming of pages, optionally including subpages

Advanced features

- GUI (WYSIWYG) and text (markup) editor
- Large number of macros
- Attachents
- Email and jabber notification
- WikiSynchronisation – keeps wiki content in sync
- Caches bytecode-compiled versions of pages to speed up.
- Theming
- Underlay directory for storing read-only pages
- Access control lists
- Reuse of authentication done by web server
- Modular authentication makes it easy to support single–sign–on
- Pluggable Parsers support many different input formats.

- Pluggable Formatters support many different output formats.
- Antispam features, surge protection

MoinMoin, moreover, includes many extensibility features and environments described later in the chapter.

2.4 MoinMoin architecure

MoinMoin's storage mechanisms are based on flat files and folders,meaning that the user information and wiki pages are stored in a special location in the filesystem of the server. This methodology faciliates the manipulation of the content on the server especially in the the case of managing revisions if the wiki is attacked by spammers.
Currently a storage abstraction layer is being developed in order to let the system administrator choose between relying on a flat file system or a relational database one.
MoinMoin is a multi–platform software tested on the following Operating System (OS) environments:

- Linux distributions
- Unix distributions
- Mac OS X
- Windows
- OpenVMS

Additionally, any other platform that support the python interpreter could host a MoinMoin wiki instance.
In the current work, MoinMoin wiki is served through the Apache2 webserver, with MoinMoin acting as a Common Gateway Interface (CGI). Of course there are lots of alternatives for system administrators including:

- Standard CGI (for apache,IIS and others)
- WSGI e.g. Apache with mod_wsgi[11]

[11]The aim of mod_wsgi is to implement a simple–to–use Apache module which can host any Python application which supports the Python WSGI interface. The module would be suitable for use in hosting high performance production web sites, as well as average personal sites running on commodity web hosting services.

- Built-in standalone Python http server
- Apache or lighttpd with FastCGI
- Apache with mod_python
- Twisted python network engine
- Zeus Web Server with FastCGI
- Sun Java System Web Server with FastCGI

Before explaining the cycling of an HTTP request through MoinMoin, the following section introduces the concept of MoinMoin's extensions (plugins) and, of course, its basic modules.

2.5 Extensions within MoinMoin

Apart from the basic modules that built–up the MoinMoin package, there exists a number of mechanisms that extend MoinMoin's capabilities and features. These mechanisms are called plugins. Through plugin development a python programmer has access to MoinMoin's Application Programming Interface (API). Plugins can serve a number of functions including the handling of user input, reaction to events and formatting of the output on the client's web–browser. Many programmers have contributed to MoinMoin's development, creating plugins which can be easily installed to an instance of the wiki. GeoMoin, specifically, is a plugin collection enabling MoinMoin to handle spatial content. Below is a list of the main types of plugins supported:

- actions
- parsers
- formatters
- macros
- themes
- WikiRPC

For each type of the above plugins, the programmer can develop his own module which will be executed by MoinMoin upon user's request. There are lots of built–in plugins in the system which exist in every instance of wiki. Of course, user–specific plugins

are installed in a different filesystem location than the one that built–in plugins are installed to, with the first, overriding the second. That is, the programmer can create a new "FullSearch.py" macro not by altering the original macro, but creating a macro with the same name in the special directory of the wiki instance.
Upon changing or installing a new plugin there exist different methodologies for them to become active, based on different enviroments, that is, for simple CGI, the plugin will be available upon the next HTTP request. For the standalone or the Twisted framework, the MoinMoin server must be restarted. On installations that are based on FastCGI and mod_python the web–server must be restarted. Before describing the types of plugins, it is useful to inform that on MoinMoin's web–site: *moinmo.in* which is itself a wiki, there exists the CategoryMarket page which lists the so–called "markets" for each type of plugins, being ActionMarcet, ParserMarcet etc.

2.5.1 Actions

The action is what the MoinMoin code does with the client request (and the requested page). The default action is to parse the page as wiki text and output HTML. However other actions can be defined to allow arbitrary processing. Thus actions can server as the moral equivalent of CGI scripts, but inside a MoinMoin context.
By writing an action, MoinMoin can be extented to do any processing the programmer needs, which, most often, is page–oriented[12], but in general any processing is allowed, as for example, printing all the users registered in the wiki.
There are built–in action which are page–oriented like:

show Parses the page using the defined formatter and outputs html

edit Imports and outputs the defined editor (text or GUI) so that the user can update the page

diff Makes a comparison between two versions of the same page

highlight Highlights certain words on a page

Any action script is executed before any output is generated, so that the script can decide what the output should be. It can generate a pre-defined output in place of the page, it can send back the original page or provide the original page with a message in a handy popup–like window. Below there is a simple action that determines if a user is a wiki superuser and outputs some text:

[12]Meaning that those actions have impact on a particular page

```
""" In this action we will create a page on-the-fly
    which checks if a user is superuser and sends some
    message to the client """
# The execute function is taking control when the
# action is invoked
def execute(pagename, request):
    # Here we get the user object from the request object
    # which holds lots of valuable data
    user = request.user
    # Send http headers to the client
    request.emit_http_headers()
    # Add a pop-up like message
    request.theme.add_msg("Action completed!")
    # Set the title of the on-the-fly page
    request.theme.send_title("We want to say")
    # Use the user object to do the check
    if user.isSuperUser():
        # The request object gives direct access
        # to the output stream
        request.write("<br>You are a super user")
    else:
        request.write("<br>You are a simple user")
    # Tell the default formatter to end the page content
    request.write(request.formatter.endContent())
    # We now send the footer of the page
    request.theme.send_footer(request.page.page_name)
    # And some closing html data
    request.theme.send_closing_html()
```

Now let's look at an action that inserts a tuple in an database and return a status message to the page.

```
# The python module for accessing various databases
import pgdb

def execute(pagename, request):
    # Create a connection object
    dbConn = pgdb.connect(user="user1",password="qwerty",
                          host="127.0.0.1",database="mydb")
```

```
try:
   # Create an execution cursor
   cur = dbConn.cursor()
   cur.execute("INSERT INTO users(id,name) VALUES (10,'somebody');")
   # Commit the current transaction
   dbConn.commit()
   cur.close()
   # Send the fancy message
   request.theme.add_msg("Database operation complete!")
except:
   # In case of error send another fancy message
   request.theme.add_msg("An error was detected!")
# In addition to the fancy message, send the original
# page content too!
request.page.send_page()
```

Studying MoinMoin's built–in plugins is an excellent way for the module designer to get a grasp of the various work–arounds that can be achieved using the action modules.

2.5.2 Macros

A macro is a mechanism for incorporating dynamic content in the middle of page output. A macro definition usually includes a small number of parameters, and is used to output information which is processed and/or formatted when the page is accessed. Like the action modules, macros have direct access to the request,formatter and page object being that of *macro.request*, *macro.formatter* and *macro.request.page* respectively.
A part that needs attention is that a macro is prohibited from altering the output before and after its definition. That is, for example, using the theme object derived from the request object the developer **cannot** send data to a page by writing eg. *request.theme.send_msg("Hello World")*
Built–in macros exist in every installation of a MoinMoin wiki that are responsible for doing specific tasks like outputting system information, recent changes etc. A macro developer can easily create a macro using a simple environment like the one described earlier in the actions section. The following simple macro output an html form that requires the user to enter some information and then using the submit button a MoinMoin action is taking place to handle the data.

```
# In this macro we will use the default formatter's
```

```
# rawHTML function to print the form we define
def execute(macro, args):
     html=""
     html+="<p> The following arguments where found: %s </p>"%args
     html+="""
        <form name="simple" method="POST">
           <p> Please enter some information </p>
           Input 1: <input type="text" name="input1"><br>
           Input 2: <input type="text" name="input2"><br>
           <input type="submit" name="action" value="PgExample">
        </form>
          """
     return macro.formatter.rawHTML(html)
```

In the above example someone can clearly note that in the HTML form statement there is no *action* directive described. But viewing the HTML *submit* input we can see that the *name* of the submit control is named **action** which makes MoinMoin handle the request and search for an action named **"value"** to handle the posting of the form. In this case the action is called **"PgExecute"** and is implemented below:

```
""" Simple action that capture the HTTP request and
     displays the contents of 2 particular form variables"""
def execute(pagename, request):
   #  Get HTTP request parameters
   parms = request.form
   input1=""
   input2=""
   # The parms is a dictionary, check to see if it contains
   # the values we are interested in
   if (parms.has_key('input1') and parms["input1"][0]!=""):
     input1=parms["input1"][0]
   if (parms.has_key('input2') and parms["input2"][0]!=""):
     input2=parms["input2"][0]
   # Output what we found
   request.emit_http_headers()
   request.theme.send_title("Results")
   request.write("You have issued: <br>")
   request.write(input1)
   request.write("<br>")
```

```
request.write(input2)
request.write(request.formatter.endContent())
request.theme.send_footer(request.page.page_name)
request.theme.send_closing_html()
```

Instead of submitting a form that way (using the name "action") one can implentend two different workarounds:

- Hard code an action directive in the form tag which can move the control to either a python script somewhere in the filesystem or, of course, to any web page which can handle this input being that jsp, php etc.
- Put whatever **name** and **value** pair for the sumbit button he wants. So the same page will be re-invoked, thus the same script will be re-invoked, so the form data print can be handled by the macro itself.

An example of the second usage is posted below:

```
# In this macro we will use the default formatter's
# rawHTML function to print the form we define
def execute(macro, args):
   #  Get HTTP request parameters
   parms = macro.request.form
   html=""
   # If the form was submitted in previous invocation
   if (parms.has_key('mode') and parms["mode"][0]=="work"):
       input1=parms["input1"][0]
       input2=parms["input2"][0]
       html+="You have issued: <br>"
       html+=input1 + " and " + input2 + "<br>"
   # if it is the first invocation
   else:
       html+="<p> The following arguments where found: %s </p>"%args
       html+="""
        <form name="simple" method="POST">
           <p> Please enter some information </p>
           Input 1: <input type="text" name="input1"><br>
           Input 2: <input type="text" name="input2"><br>
           <input type="submit" name="mode" value="work">
```

```
        </form>
          """
   return macro.formatter.rawHTML(html)
```

Of course, as stated before, macros are not used only for form handling and html output. As soon as a programmer can import every python module he chooses, every possible task can be performed. The last thing important to mentioned on macros, is how they can be defined in a page so they can be execute. A simple wiki page containing a macro is shown below:

```
Welcome to my page. This page contains a macro.
<<MyMacro( param1, param2, param3)>>
```

Note that the anchors are needed for the content to be treated as a wiki markup for macro execution.

2.5.3 Parsers

A parser, or, more formally, syntactic analyser, is a process that analyzes a sequence of tokens to determine grammatical structure with respect to a given (more or less) formal grammar. In a MoinMoin context, a parser is a python module that is provided with plain text, reads the input, analyzes the content and uses a formatter to display the appropriate output to the wiki page. Like every other module, parsers have access to MoinMoin objects already instantiated.
Let's have a look at a request for a parser execution within a wiki page:

```
This page contains a parser

{{{
#!map ,

  mapfile=1214071817.69.30350_usa_laea.map
  need_options=true
  need_reference=false
  # some comments go here
  need_legend=true
  autopopup=false
  debug=true
```

```
  annotation=goodies

}}}

[[FullSearch(CategorySamples)]]
----
CategoryCategory
```

It is clear that in order for MoinMoin to treat a parser statement as executable, it is important that the call is surrounded by **three left-sided** and **three right-sided** brackets, having the name of the parser issued, beginning with the symbols **#!** and ending with a **comma**, while the content follows.

The content can be of any type the developer decides, as long as the appropriate analysis is developed within the module. A typical usage of a parser module within MoinMoin is the *colorization and proper output* of some input written in a well–known format. For example, using the *text_pascal* parser, one can feed it with pascal code and on viewing of the page, a very stylish representation will be produced. For parser modules like that, a developer can create a class that derives from the built–in ParserBase class which contains the framework for colorizing the input, so that the developer is only requested to define the *rules* of the language as long as the *reserved words*.

Parsers, of course, can be used for far more complicated tasks than colorising a well–known format. In that scenario, like the other modules, the developer is introduced to a method of defining parsers within MoinMoin. A very simple example of a parser is already in the built–in parser collection and is responsible for wrapping a *raw text input* into html <pre> directives:

```
# -*- coding: iso-8859-1 -*-
"""
    MoinMoin - Plain Text Parser, fallback for text/*

    @copyright: 2000-2002 Juergen Hermann <jh@web.de>
    @license: GNU GPL, see COPYING for details.
"""

Dependencies = []

class Parser:
    """
```

```
        Send plain text in a HTML <pre> element.
    """
    extensions = '*'
    Dependencies = []

    def __init__(self, raw, request, **kw):
        # The raw parameter contains the input text as a WHOLE
        self.raw = raw
        # Parsers have direct access to request and form obejcts
        self.request = request
        self.form = request.form
        self._ = request.getText

    def format(self, formatter):
        """ Send the text. """
        # This is the output of a <pre> html markup
        self.request.write(formatter.preformatted(1))
        # Print the text, replacing tabs with spaces
        self.request.write(formatter.text(self.raw.expandtabs()))
        # This is the equivelant of </pre> markup
        self.request.write(formatter.preformatted(0))
```

In that case the developer is needed to define a class name *Parser* which must contains two important functions, __**init**__ and **format**, with first being the first function called when the parser is executed and the last being the last one to be executed handling the output of the parsing results. Please note that no call for the format function is done inside __init__. That is, those functions are called automatically in the correct order.
A developer can create as many classes as he wants, thus doing the actual parsing in a different class. A workaround frequently used is the following:

- __init__ is called, which holds the raw data to be parsed
- Instantiate a class named **pdp** (Page Data Processor) inside __init__ which does the actual parsing and returns the parsed data in a special structure or a simple dictionary.
- Instantiate a user–specified class inside __init__ which given as input the parsing structure computed above does the required calculations or procedures and afterwards saves the output in a particular structure. The output could vary from simple text or html to special formats supported by MoinMoin like pdf.

- __init__ finishes its execution.
- *format* function takes control, which having the required visibility to the output structure, uses a predefined formatter to render the output to the client–browser.

In the current work, the above workaround is used in the parser that renders and projects the maps to the client browser. The result is clean well–structured source code.

2.5.4 Formatters

A formatter is used to output the page in a particular format. The default formatter for MoinMoin is the html formatter. The user can command a page to be rendered using another formatter either by editing the page and inserting a *#format* followed by the name of the installed formatter, or by selecting an action that recreates an on-the-fly presentation of the page using the selected parser.
Generally speaking, a formatter provides to MoinMoin an interface for outputting standard document elements. As we stated earlier, MoinMoin uses the idea of separate parsers (e.g. for parsing the wiki syntax) and formatters (e.g. for outputting HTML code) with a SAX[13]–like interface between the two. The idea is that it is possible to output DocBook instead of HTML, only by writing a docbook-formatter that implements the formatter interface, while all parsers that use the interface will automatically be supported. This formatter interface is implemented in the class **FormatterBase**.
Writing a formatter is a demanding project that requires that the developer is specialized in the particular format, so it is out of the scope of this work to provide information on how to build one.
An important element to note is that every module described earlier has direct access to the formatter object used in the particular page so it is recommended that this interface is being, rather than directly outputting HTML, so that the document can be delivered to the client in any supported output format. The FormatterBase class contains everything a formatter can do and is implemented in the file *__init__.py* located in the *formatter* under the MoinMoin package installation location.

[13]The Simple API for XML (SAX) is a serial access parser API for XML. SAX provides a mechanism for reading data from an XML document. It is a popular alternative to the Document Object Model (DOM).

2.5.5 Themes

A theme determines the visual appearance of the output. There can be different screen layouts, icons and CSS per theme. The default theme is "modern". A user can set a different theme in the *user preferences*. Describing themes in detail is out of the scope of the current work.

2.6 XML-RPC and WikiRPC

XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism. XML-RPC defines only a handful of data types and commands, and the entire description can be printed on two pages of paper. This is in stark contrast to most RPC systems, where the standards documents often run into the hundreds of pages and require considerable software support in order to be used. It is beyond the scope of this document to discuss the complete XML-RPC specification, but a sample request–response is include below:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
   </methodCall>


<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
         </param>
      </params>
   </methodResponse>
```

In general, XML-RPC defines standard data types and their XML representation, which can be used in order to formulate requests to particular methods and responses.

WikiRPC stands for an implementation of the XML-RPC specification in regard to wiki systems. WikiRPC is a standard and is currently at its second version. Acting as a specification, WikiRPC is implemented by the following wiki engines:

- TWiki
- MoinMoin
- UseModWiki
- PhpWiki
- OpenWiki
- WikiGateway

WikiRPC defines standard functions in order to manage the wiki content using client tools:

getRecentChanges(Date timestamp) Get list of changed pages since timestamp, which should be in UTC. The result is an array, where each element is a struct.

int getRPCVersionSupported() Return the WikiRPC version implemented

utf8 getPage(utf8 pagename) Get the raw Wiki text of page, latest version.

utf8 getPageVersion(utf8 pagename, int version) Get the raw Wiki text of page.

utf8 getPageHTML(utf8 pagename) Return page in rendered HTML, latest version.

utf8 getPageHTMLVersion(utf8 pagename, int version) Return page in rendered HTML.

array getAllPages() Returns a list of all pages. The result is an array of utf8 pagenames.

struct getPageInfo(utf8 pagename) returns a struct with information about a particular page.

struct getPageInfoVersion(utf8 pagename, int version) Returns a struct just like plain getPageInfo(), but this time for a specific version.

array listLinks(utf8 pagename) Lists all links for a given page.

array getBackLinks(utf8 page) Returns the pages that link to this page.

putPage(utf8 page, utf8 content, struct attributes) Writes the content of the page.

array listAttachments(utf8 page) Lists attachments on a given page. The array consists of utf8 attachment names that can be fed to getAttachment (or putAttachment).

putAttachment(utf8 attachmentName, base64 content) (over)writes an attachment.

struct getAttachmentInfo(utf8 attachmentName) Returns attachment information.

MoinMoin is implementing the above specified functions as a subset containing many other facilities. The following example posted in the official website of MoinMoin return the names of all the pages included in a wiki instance:

```
1 import xmlrpclib
2 srcwiki = xmlrpclib.ServerProxy("http://localhost/mywiki?action=xmlrpc2")
3
4 allpages = srcwiki.getAllPages()
5 for pagename in allpages:
6     pagedata = srcwiki.getPage(pagename)
7     print "Got %s." % pagename
8     print pagedata
```

It is important to note that a programmer can create individual WikiRPC methods as plugins in order to perform special remote operations within the wiki instance. The small plugin below checks if a particular page exists in the wiki instance:

```
from MoinMoin import Page
import xmlrpclib

def execute(xmlrpcobj,pagename):
    request = xmlrpcobj.request
    thepage = Page.Page(request,pagename)
    if thepage.isStandardPage(False)==True:
        return xmlrpclib.Boolean(1)
    else:
        return xmlrpclib.Boolean(0)
```

WikiRPC plugins are extensively used in GeoMoin wherever the MoinMoin request object does not exist (eg when remote scripts are called using AJAX functions).

2.7 MoinMoin event sequence

Now that the MoinMoin extensions are described, we can provide a sequence of the order the events are taking place in MoinMoin upon an HTTP request [4]. This sequence is crucial to understanding the inner working between the MoinMoin modules and components. The listing below provides the events as a summary:

- The user issues an HTTP request.
- Determine the theme to use
  - perform the action for the page, which is, in the default case is:
- Parse the page input
- Format the page for output
  - in the process of outputting the page, execute any macros or parsers that are encoded in the page. Any output from the macros or parsers is incorporated (inline) with the other output from the page.
- Send the result back to the client

Assuming that the architecture of the system is based on Apache web–server with MoinMoin acting as a CGI application. We won't run into much detail on how apache communicates with the CGI applications. In the process of describing the cycle an HTTP request follows before a response is returned to the client, it is possible that many modules and function contained in the MoinMoin application will be stated and whenever possible, described to detail. Based on the fact that MoinMoin has a state–of–the–art architecture as far as the layers of abstraction is concerned, we could state that a migration from standard CGI to FastCGI or mod_python, would change the roll of event only partly.

When the user–agent issues the HTTP request to the server, Apache takes control identifying an executable command. The CGI protocol definition describes from a webserver's point–of–view how the control will be passed to the cgi application along with the parameters, and the returning of the output. The HTTP request variables are passed to the CGI program using enviroment variables. In our case the CGI that is ordered

to be executed is **moin.cgi** whose main role is the dispatching of the control to the appropriate MoinMoin server class object which in case is ServerCGI written in python. The fact that moin.cgi is very small in terms of lines of code give us the ability to post it here:

```
#!/usr/bin/env python
# -*- coding: iso-8859-1 -*-
"""
    MoinMoin - CGI Driver Script

    @copyright: 2000-2005 by Juergen Hermann <jh@web.de>,
                2008 by MoinMoin:ThomasWaldmann
    @license: GNU GPL, see COPYING for details.
"""

import sys, os

sys.path.insert(0, '/usr/local/lib/python2.5/site-packages')
sys.path.insert(0, '/usr/local/share/moin/mywiki')

from MoinMoin.server.server_cgi import CgiConfig, run

class Config(CgiConfig):
    name = 'moin'

run(Config)
```

The server_cgi module is located here: *MoinMoin/server/server_cgi.py* and is responsible for dispatching the control to the appropriate *Request Subclass*. Request sub classes handle all the information that came from the client. There is a request sub class for each type of server environment, hiding the differences between different servers behind a *common API*. This common API is a class that every Request Subclass derives from, called *RequestBase*, and can found in the following location: *MoinMoin/request/__init__.py*. That is, RequestBase is the last level of abstraction between the different types of web–serving.

In example, request_cgi located at: *MoinMoin/request/request_cgi.py* implements the functions that read from the input stream, write back to it and capture the HTTP request parameters storing them to cgi.FieldStorage[14] structures. Finally the request_cgi

[14] cgi is a Python module for accessing features of the Common Gateway Interface. Fieldstorage is a module specialised in preserving the information in a structured manner.

object moves to the higher abstraction level by calling RequestBase's **run function**. From now on, there are no differences between the various architectural decisions.
The run function, upon execution, initializes the theme which was predefined to be used and afterwards, based on the url sent with the HTTP request, decides what pagename (url) or action was requested and invokes:

- Page().send_page() to output normal wiki pages
- MoinMoin.action.handler() to get a function handler to the particular function's definition, if the url contained a request for a particular action

If a page was requested to be rendered the following steps are being executed:

- Loads the raw page (into body)
- Decides what formatter to use (default is text_html formatter)
- Creates formatter (or uses default from Request object already instantiated)
- Reads processing instructions and access controls (ACLS) at the top of the page
- Sends page header if needed
- Decides what Parser to use (default is "wiki" parser)
- Creates Parser object
- Calls *send_page_content()* to:
  - use a cached page *if possible*
  - use *Parser(body, request).format(formatter)* to create the page *if not*
  - send the page body
- Sends footnotes if any
- Sends footer if needed

The *Parser(body, request).format(formatter)* function parses the body of the raw page looking for special strings and calls the supplied formatter to translate the parts into the output language (usually HTML). Additionally the formatter calls the macros found in the page and creates and calls any parser declared.
When the page is ready for output, *request.write(text)* is called, which is implemented for every different web–serving environment, and outputs the data to the user agent.

Thus, without running into specific detail, the above call–graph is executed upon an HTTP request. It is clear that MoinMoin offers a well–structured environment for web–application development giving the programmer a complete, well–organised API, supporting both low level and high level tasks. Throughout the development of GeoMoin, MoinMoin acted transparently, introducing solutions to every web–development difficulty that appeared, while its development team was highly active and helpful on both the mailing list and the MoinMoin irc room.

Chapter 3

GIS concepts and web–mapping

3.1 A GIS overview

A geographic information system (GIS) is an information system for capturing, storing, analyzing, managing and presenting data which are spatially referenced [5]. In the strictest sense, it is any information system capable of integrating, storing, editing, analyzing, sharing, and displaying geographically referenced information. In a more generic sense, GIS applications are tools that allow users to create interactive queries (user created searches), analyze spatial information, edit data, maps, and present the results of all these operations. That is, thanks to the advent of sophisticated computer techniques has proliferated the multi-disciplinary application of geo-processing methodologies, and provided data integration capabilities that were logistically impossible before.
Geographic information systems technology can be used for scientific investigations, resource management, asset management, environmental impact assessment, urban planning, cartography, criminology, geographic history, marketing, logistics and decision support just to name a few. For example, GIS might allow emergency planners to easily calculate emergency response times in the event of a natural disaster, GIS might be used to find wetlands that need protection from pollution, or GIS can be used by a company to site a new business location to take advantage of a previously underserved market.

A GIS system has four main functional subsystems [1]. These are:

Data input subsystem This subsystem allows the user to capture, collect, and transform spatial and thematic data into digital form. The data inputs are usually derived from a combination of hard copy maps, aerial photographs, remotely sensed images, reports, survey documents, etc.

Data storage and retrieval This subsystem organizes the data, spatial and attribute, in a form which permits it to be quickly retrieved by the user for analysis, and permits rapid and accurate updates to be made to the database. This component usually involves use of a database management system (DBMS) for maintaining attribute data. Spatial data is usually encoded and maintained in a proprietary file format.

Data manipulation and analysis This subsystem allows the user to define and execute spatial and attribute procedures to generate derived information. This subsystem is commonly thought of as the heart of a GIS, and usually distinguishes it from other database information systems and computer-aided drafting (CAD) systems.

Data output This subsystem allows the user to generate graphic displays, normally maps, tabular reports representing derived information products as well as various information in the various forms supported by the GIS product (eg. Geography Markup Language [GML]).

An operational GIS also has a series of components that combine to make the system work [1]. These components are critical to a successful GIS.

Hardware This is the computer system on which a GIS operates. Today, GIS software runs on a wide range of hardware types, from centralized computer servers to desktop computers used in stand-alone or networked configurations.

Software GIS software provides the functions and tools needed to store, analyze, and display geographic information. A review of the key GIS software subsystems is provided above.

Data The most important component of a GIS is the data. Geographic data and related tabular data can be collected in-house, compiled to custom specifications and requirements, or occasionally purchased from a commercial data provider. A GIS can integrate spatial data with other existing data resources, often stored in a corporate DBMS. The integration of spatial data (often proprietary to the GIS software), and tabular data stored in a DBMS is a key functionality afforded by GIS.

People GIS technology is of limited value without the people who manage the system and develop plans for applying it to real world problems. GIS users range from technical specialists who design and maintain the system to those who use it to help them perform their everyday work. The identification of GIS specialists versus end users is often critical to the proper implementation of GIS technology.

Methods A successful GIS operates according to a well-designed implementation plan and business rules, which are the models and operating practices unique to each organization.

3.2 Spatial data models

Traditionally spatial data has been stored and presented in the form of a map. Three basic types of spatial data models have evolved for storing geographic data digitally. These are referred to as :

- Vector
- Raster
- Image

The diagram 3.2 reflects the two primary spatial data encoding techniques. These are vector and raster. Image data utilizes techniques very similar to raster data, however typically lacks the internal formats required for analysis and modeling of the data. Images reflects pictures or photographs of the landscape.

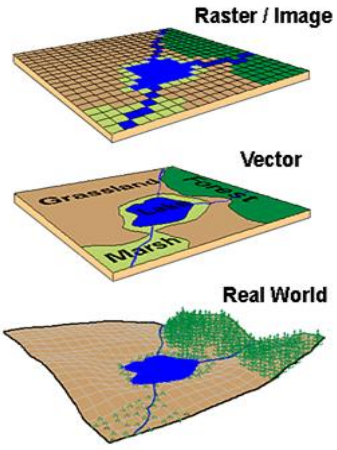


FIGURE 3.1: Spatial data types.

3.2.1 Vector data model

In the vector data model, objects are constructed from points and edges as primitives. A point is represented by its coordinates, whereas more complex and surfacic objects are

represented by structures (lists,sets,arrays) on the point representation. In particular can be represented by the finite set of its vertices.
There exists a large number of variants to represent spatial obejcts in a vector model. The following is a simple representation [6]:

- A point is represented by its coordinates *[x: real,y: real [,z: real]]*
- A polyline is represented by a *list* of points $< p_1, \ldots. \ p_n >$, each p_i being a a vertex. Each pair (p_i,p_{i+1}), with $i < n$, represents one of the polyline's edges.
- A polygon is also represented by a list of points. The notable difference is that the list represents a *closed* polyline, and therefore the pair (p_n,p_1) is also an edge of the polygon. We could also note that in a different representation, a polygon could be a list of lines, whereas a line is consisting of exactly two points.
- A region is simply a set of polygons: { *polygon* }

A few remarks are noteworthy. First, there are n way of choosing where to start the boundary description of a polygon object; once the starting vertex has been chosen there are two ways of scanning the vertices, called *clockwise* and *anticlockwise* orders. Second, there is no apparent distinction between a polyline structure and a polygon one. It is up to the software that manipulates geometric data to interpret properly the structure, and to check if the representation is valid; that is, to verify that the polyline is closed for the polygon.

By considering collections and no longer individual objects, we become interested in the *relationships* among objects of the same collection. In the Appendix we describe the three commonly used representations of collections of objects, respectively called *spaghetti, network* and *topological* models. The *topological* model is often referred to as an *intelligent data structure* because spatial relationships between geographic features are easily derived when using them. Primarily for this reason the topologic model is the dominant vector data structure currently used in GIS technology. Many of the complex data analysis functions cannot effectively be undertaken without a topologic vector data structure.

Government agencies and other organizations collect spatial information and georeferenced socioeconomic data in their own areas of responsibility. Along with international standardization committes, these organizations help define geospatial data standards, whose data formats have very strict rules about their content and characteristics of the data they contain. Data are also available in formats specific to particular GIS or Computer Aided Design (CAD) packages. For example, the DXF format was intended as a

TABLE 3.1: Popular vector datasets

Format Name	Software Platform	Type	Developer	Comments
Arc Export	ARC INFO	Transfer	Environmental Systems Research Institute, Inc. (ESRI)	Transfers data across ARC/INFO platforms.
AutoCAD Drawing Files	AutoCAD	Internal	Autodesk	
Digital Line graphs (DLG)	Many	Transfer	United States Geological Survey (USGS)	Used to publish USGS digital maps.
Hewlett-Packard Graphic Language	Many	Internal	Hewlett-Packard	Used to control HP plotters.
MapInfo Data Transfer Files	MapInfo	Transfer	MapInfo Corp.	
MapInfo Map Files	MapInfo	Internal	MapInfo Corp.	
MicroStation Design Files (DGN)	MicroStation	Internal	Bentley Systems, Inc.	
Spatial Data Transfer System (SDTS)	Many	Transfer	US Government	New US standard for vector and raster geographic data.
TIGER	Many	Transfer	US Census Bureau	Used to publish US Census Bureau maps.

data transfer format between CAD software packages and became a popular GIS data format.

Another popular format with extensive application throughout web–mapping applications is the ESRI Shapefiles; A "shapefile" commonly refers to a collection of files with ".shp", ".shx", ".dbf", and other extensions on a common prefix name (e.g., "lakes.*"). The actual shapefile relates specifically to files with the ".shp" extension, however this file alone is incomplete for distribution, as the other supporting files are required. Shapefiles spatially describe geometries: points, polylines, and polygons. These, for example, could represent water wells, rivers, and lakes, respectively. Each item may also have attributes that describe the items, such as the name or temperature. Extensive information concerning the ESRI shapefiles can be found in the *ESRI Shapefile Technical Description* [7].

Table 3.2.1 contains a listing and the associative descriptions on some of the most popular vector formats in the market today.

3.2.2 Raster data model

[8]Raster data models incorporate the use of a grid-cell data structure where the geographic area is divided into cells identified by row and column. This data structure is commonly called raster. While the term raster implies a regularly spaced grid, other tessellated data structures do exist in grid based GIS systems. In particular, the quadtree data structure has found some acceptance as an alternative raster data model.

The size of cells in a tessellated data structure is selected on the basis of the data accuracy and the resolution needed by the user. There is no explicit coding of geographic coordinates required since that is implicit in the layout of the cells. A raster data structure is in fact a matrix where any coordinate can be quickly calculated if the origin point is known, and the size of the grid cells is known. Since grid-cells can be handled as two-dimensional arrays in computer encoding many analytical operations are easy to program. This makes tessellated data structures a popular choice for many GIS software. Topology is not a relevant concept with tessellated structures since adjacency and connectivity are implicit in the location of a particular cell in the data matrix.
Several tessellated data structures exist, however only two are commonly used in GIS's. The most popular cell structure is the regularly spaced matrix or raster structure. This data structure involves a division of spatial data into regularly spaced cells. Each cell is of the same shape and size. Squares are most commonly utilized. Different tessellated data structures can use different tessellation modes such as:

- Grid squares of same size
- Hexagonal cells of same size

Irregural tesselations can include:

- Cadastral zones
- Thiessen polygons

Since geographic data is rarely distinguished by regularly spaced shapes, cells must be classified as to the most common attribute for the cell. The problem of determining the proper resolution for a particular data layer can be a concern. If one selects too coarse a cell size then data may be overly generalized. If one selects too fine a cell size then too many cells may be created resulting in a large data volumes, slower processing times, and a more cumbersome data set. As well, one can imply an accuracy greater than that of the original data capture process and this may result in some erroneous results during analysis.
As well, since most data is captured in a vector format, e.g. digitizing, data must be converted to the raster data structure. This is called vector-raster conversion. Most GIS software allows the user to define the raster grid (cell) size for vector-raster conversion. It is imperative that the original scale, e.g. accuracy, of the data be known prior to conversion. The accuracy of the data, often referred to as the resolution, should determine the cell size of the output raster map during conversion.

It is important to understand that the selection of a particular data structure can provide advantages during the analysis stage. For example, the vector data model does not

handle continuous data, e.g. elevation, very well while the raster data model is more ideally suited for this type of analysis. Accordingly, the raster structure does not handle linear data analysis, e.g. shortest path, very well while vector systems do.

3.2.3 Image data

Image data is most often used to represent graphic or pictorial data. The term image inherently reflects a graphic representation, and in the GIS world, differs significantly from raster data. Most often, image data is used to store remotely sensed imagery, e.g. satellite scenes or orthophotos, or ancillary graphics such as photographs, scanned plan documents, etc. Image data is typically used in GIS systems as background display data (if the image has been rectified and georeferenced); or as a graphic attribute. Remote sensing software makes use of image data for image classification and processing. Typically, these kinds of data must be converted into a raster format (and perhaps vector) to be used analytically with the GIS.

Image data is typically stored in a variety of de facto industry standard proprietary formats. These often reflect the most popular image processing systems. Other graphic image formats, such as TIFF, GIF, PCX, etc., are used to store ancillary image data. Most GIS software will read such formats and allow you to display these kinds of data.

FIGURE 3.2: Image data is most often used for remotely sensed imagery such as satellite imagery or digital orthophotos.

It is common, that both image data and raster datasets are required to be self–descriptive concerning their existance in the physical space. That is, to establish relations between imagery to map projection and coordinate system, in other words to *georeference* their contents. When data from different sources need to be combined and then used in a GIS

Table 3.2: Vector data: Pros and Cos

Advantages	Data can be represented at its original resolution and form without generalization.
	Graphic output is usually more aesthetically pleasing.
	Since most data is in vector form no data conversion is required.
	A vector dataset is easily maintaned in terms of inserting and updating features.
	Accurate geographic location of data is maintained.
	Allows for efficient encoding of topology, and as a result more efficient operations that require topological information.
Disadvantages	The location of each vertex needs to be stored explicitly.
	For effective analysis, vector data must be converted into a topological structure. This is often processing intensive and usually requires extensive data cleaning. As well, topology is static, and any updating or editing of the vector data requires re-building of the topology.
	Algorithms for manipulative and analysis functions are complex and may be processing intensive.
	Continuous data, such as elevation data, is not effectively represented in vector form.

application, it becomes essential to have a common referencing system. This is brought about by using various georeferencing techniques. Using the GPS technology, a user can georeference a spatial point of interest using the latitute/longitude pair.
In order to georeference imagery, desktop GIS application (ArcMap, ERDAS Imagine, GRASS) provide the essential toolboxes; one first needs to establish control points, input the known geographic coordinates of these control points, choose the coordinate system and other projection parameters and then minimize residuals. Residuals are the difference between the actual coordinates of the control points and the coordinates predicted by the geographic model created using the control points. They provide a method of determining the level of accuracy of the georeferencing process.
Raster formats like GeoTIFF can internally contain georeferencing information, while bitmaps and JPEG imagery need to be accompanied with files like the *world file* which contains the required information in order to be used in spatial analysis and overlays.

3.2.4 Data accuracy and quality

The quality of data sources for GIS processing is becoming an ever increasing concern among GIS application specialists. With the influx of GIS software on the commercial market and the accelerating application of GIS technology to problem solving and decision making roles, the quality and reliability of GIS products is coming under closer

Table 3.3: Raster data: Pros and Cos

Advantages	The geographic location of each cell is implied by its position in the cell matrix. Accordingly no geographic coordinates are stored.
	Due to the nature of the data storage technique data analysis is usually easy to program and quick to perform.
	Discrete data, e.g. forestry stands, is accommodated equally well as continuous data, e.g. elevation data, and facilitates the integrating of the two data types.
	Grid-cell systems are very compatible with raster-based output devices, e.g. electrostatic plotters, graphic terminals.
Disadvantages	The cell size determines the resolution at which the data is represented.
	It is especially difficult to adequately represent linear features depending on the cell resolution. Accordingly, network linkages are difficult to establish.
	Raster maps inherently reflect only one attribute or characteristic for an area.
	Since most input data is in vector form, data must undergo vector-to-raster conversion. Besides increased processing requirements this may introduce data integrity concerns due to generalization and choice of inappropriate cell size.
	Most output maps from grid-cell systems do not conform to high-quality cartographic needs.

scrutiny. Much concern has been raised as to the relative error that may be inherent in GIS processing methodologies. While research is ongoing, and no finite standards have yet been adopted in the commercial GIS marketplace, several practical recommendations have been identified which help to locate possible error sources, and define the quality of data. The following review of data quality focuses on three distinct components, data accuracy, quality, and error.

Accuracy

The fundamental issue with respect to data is accuracy. Accuracy is the closeness of results of observations to the true values or values accepted as being true. This implies that observations of most spatial phenomena are usually only considered to estimates of the true value. The difference between observed and true (or accepted as being true) values indicates the accuracy of the observations.

Basically two types of accuracy exist. These are positional and attribute accuracy. Positional accuracyis the expected deviance in the geographic location of an object from its true ground position. This is what we commonly think of when the term accuracy is discussed. There are two components to positional accuracy. These are relative and

absolute accuracy. Absolute accuracy concerns the accuracy of data elements with respect to a coordinate scheme, e.g. UTM. Relative accuracy concerns the positioning of map features relative to one another.
Often relative accuracy is of greater concern than absolute accuracy. For example, most GIS users can live with the fact that their survey township coordinates do not coincide exactly with the survey fabric, however, the absence of one or two parcels from a tax map can have immediate and costly consequences.
Attribute accuracy is equally as important as positional accuracy. It also reflects estimates of the truth. Interpreting and depicting boundaries and characteristics for forest stands or soil polygons can be exceedingly difficult and subjective. Most resource specialists will attest to this fact. Accordingly, the degree of homogeneity found within such mapped boundaries is not nearly as high in reality as it would appear to be on most maps.

Quality

Quality can simply be defined as the fitness for use for a specific data set. Data that is appropriate for use with one application may not be fit for use with another. It is fully dependant on the scale, accuracy, and extent of the data set, as well as the quality of other data sets to be used. The recent U.S. Spatial Data Transfer Standard (SDTS) identifies five components to data quality definitions. These are :

Lineage The lineage of data is concerned with historical and compilation aspects of the data such as the source of the data and their content

Positional Accuracy The identification of positional accuracy is important. This includes consideration of inherent error (source error) and operational error (introduced error). A more detailed review is provided in the next section.

Attribute Accuracy Consideration of the accuracy of attributes also helps to define the quality of the data. This quality component concerns the identification of the reliability, or level of purity (homogeneity), in a data set.

Logical Consistency This component is concerned with determining the faithfulness of the data structure for a data set. This typically involves spatial data inconsistencies such as incorrect line intersections, duplicate lines or boundaries, or gaps in lines. These are referred to as spatial or topological errors.

Completeness The final quality component involves a statement about the completeness of the data set. This includes consideration of holes in the data, unclassified areas, and any compilation procedures that may have caused data to be eliminated.

3.3 Important GIS issues

3.3.1 Organizing non–spatial data

GIS use raster and vector representations to model location, but they must also record information about the real-world phenomena positioned at each location and the attributes of these phenomena. That is, the GIS must provide a linkage between spatial and non-spatial[1] data. These linkages make the GIS "intelligent" insofar as the user can store and examine information about where things are and what they are like.
A separate data model is used to store and maintain attribute data for GIS software. These data models may exist internally within the GIS software, or may be reflected in external commercial Database Management Software (DBMS). A variety of different data models exist for the storage and management of attribute data [1] [9]. The most common are:

- Tabular
- Hierarchical
- Network
- Relational
- Object-Oriented

3.3.1.1 Tabular models

The simple tabular model stores attribute data as sequential data files with fixed formats (or comma delimited for ASCII data), for the location of attribute values in a predefined record structure. This type of data model is currently outdated in the GIS arena.
All records in this data base have the same number of "fields". Individual records have different data in each field with one field serving as a key to locate a particular record. For example, a social security number may be the key field in a record of a name, address, phone number, sex, ethnicity, place of birth, date of birth of a person, and so on.
For a person there could be hundreds of fields associated with the record. When the number of fields becomes lengthy a flat file is cumbersome to search. Also the key field is usually determined by the programmer and searching by other determinants may be difficult for the user.
This method lacks any means of checking data integrity, as well as being inefficient with respect to data storage, e.g. limited indexing capability for attributes or records, etc.

[1] Non–spatial data are also called *attribute* or *characteristic* data

3.3.1.2 Hierarchical models

Hierarchical files store data in more than one type of record. This method is usually described as a "parent-child, one-to-many" relationship. One field is key to all records, but data in one record does not have to be repeated in another. This system allows records with similar attributes to be associated together. The records are linked to each other by a key field in a hierarchy of files. Each record, except for the master record, has a higher level record file linked by a key field "pointer". In other words, one record may lead to another and so on in a relatively descending pattern. An advantage is that when the relationship is clearly defined, and queries follow a standard routine, a very efficient data structure results. The database is arranged according to its use and needs. Access to different records is readily available, or easy to deny to a user by not furnishing that particular file of the database. One of the disadvantages is one must access the master record, with the key field determinant, in order to link "downward" to other records.

3.3.1.3 Network models

The network database organizes data in a network or plex structure. Any column in a plex structure can be linked to any other. Like a tree structure, a plex structure can be described in terms of parents and children. This model allows for children to have more than one parent.
Network DBMS have not found much more acceptance in GIS than the hierarchical DBMS. They have the same flexibility limitations as hierarchical databases; however, the more powerful structure for representing data relationships allows a more realistic modelling of geographic phenomenon. However, network databases tend to become overly complex too easily. In this regard it is easy to lose control and understanding of the relationships between elements.

3.3.1.4 Relational models

The relational database organizes data in tables. Each table, is identified by a unique table name, and is organized by rows and columns. Each column within a table also has a unique name. Columns store the values for a specific attribute, e.g. cover group, tree height. Rows represent one record in the table. In a GIS each row is usually linked to a separate spatial feature, e.g. a forestry stand. Accordingly, each row would be comprised of several columns, each column containing a specific value for that geographic feature. The following figure presents a sample table for forest inventory features. This table has 4 rows and 4 columns. The forest stand number would be the label for the spatial

feature as well as the primary key for the database table. This serves as the linkage between the spatial definition of the feature and the attribute data for the feature.

UNIQUE STAND NUMBER	AVG. TREE HEIGHT	STAND SITE INDEX	STAND AGE
001	3	G	100
002	4	M	80
003	4	M	60
004	4	G	120

Data is often stored in several tables. Tables can be joined or referenced to each other by common columns (relational fields). Usually the common column is an identification number for a selected geographic feature. This identification number acts as the primary key for the table. The ability to join tables through use of a common column is the essence of the relational model. Such relational joins are usually ad hoc in nature and form the basis of for querying in a relational GIS product. Unlike the other previously discussed database types, relationships are implicit in the character of the data as opposed to explicit characteristics of the database set up.

There are many different designs of DBMSs, but in GIS the relational design has been the most useful. In the relational design, data are stored conceptually as a collection of tables. Common fields in different tables are used to link them together. This surprisingly simple design has been so widely used primarily because of its flexibility and very wide deployment in applications both within and without GIS.

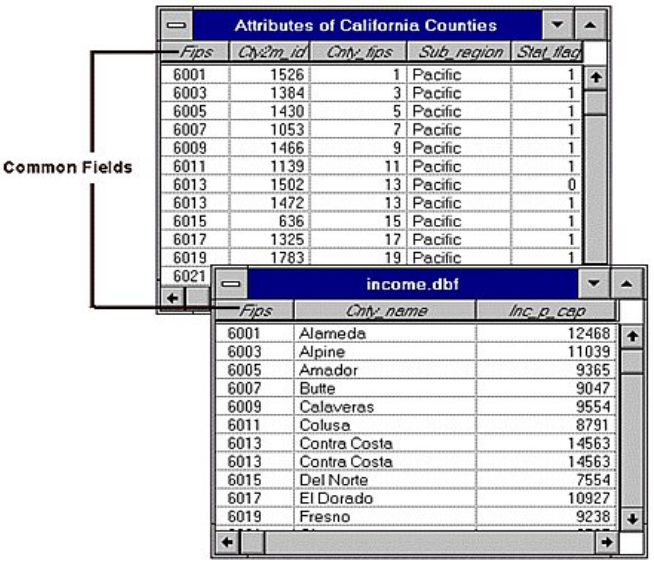


FIGURE 3.3: In the relational design, data are stored conceptually as a collection of tables. Common fields in different tables are used to link them together.

In fact, most GIS software provides an internal relational data model, as well as support for *commercial off-the-shelf* (COTS) relational DBMS'. COTS DBMS' are referred to as external DBMS'. This approach supports both users with small data sets, where an internal data model is sufficient, and customers with larger data sets who utilize a DBMS for other corporate data storage requirements. With an external DBMS the GIS software can simply connect to the database, and the user can make use of the inherent capabilities of the DBMS. External DBMS' tend to have much more extensive querying and data integrity capabilities than the GIS' internal relational model. The emergence and use of the external DBMS is a trend that has resulted in the proliferation of GIS technology into more traditional data processing environments.

The relational DBMS is attractive because of its:

- simplicity in organization and data modelling.
- flexibility - data can be manipulated in an ad hoc manner by joining tables.
- efficiency of storage - by the proper design of data tables redundant data can be minimized
- the non-procedural nature - queries on a relational database do not need to take into account the internal organization of the data.

The diagram 3.4 illustrates the basic linkage between a vector spatial data (topologic model) and attributes maintained in a relational database file.

3.3.1.5 Object–oriented models

he object-oriented database model manages data through objects. An object is a collection of data elements and operations that together are considered a single entity. This approach has the attraction that querying is very natural, as features can be bundled together with attributes at the database administrator's discretion. To date, only a few GIS packages are promoting the use of this attribute data model. However, initial impressions indicate that this approach may hold many operational benefits with respect to geographic data processing.

3.3.2 Map projections in overview

A map projection is a way to represent the curved surface of the Earth on the flat surface of a map. Strictly speaking, a map projection can be viewed as a *method* or a

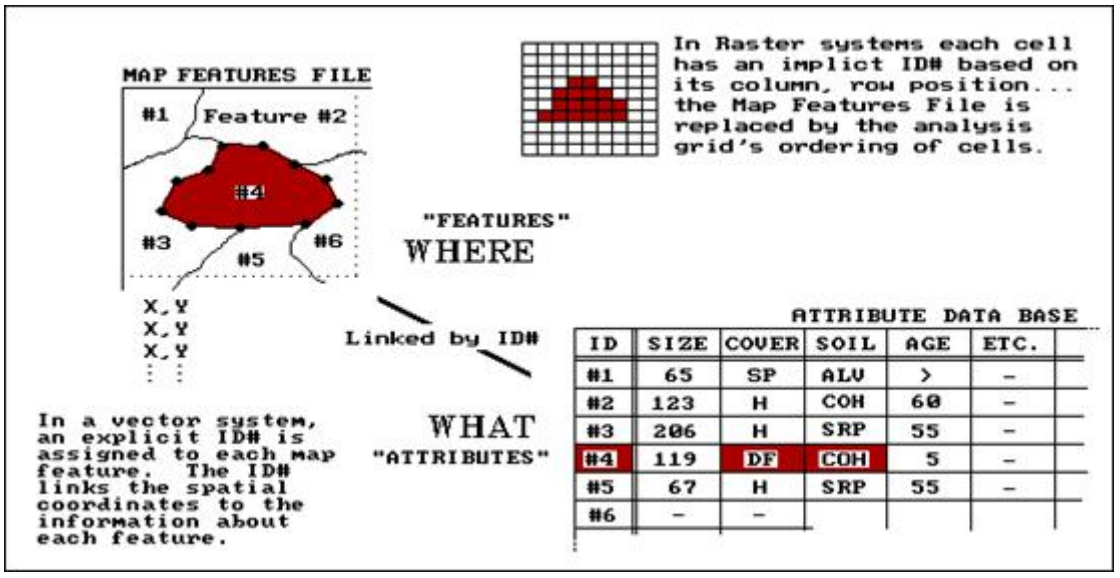


FIGURE 3.4: Basic linkages between a vector spatial data (topologic model) and attributes maintained in a relational database file.

function defined on the earth's surface and with values on the plane, and not necessarily a geometric projection.[2]

Flat maps could not exist without map projections, because a sphere cannot be laid flat over a plane without distortions. One can see this mathematically as a consequence of Gauss's *Theorema Egregium*[3]. Flat maps can be more useful than globes in many situations: they are more compact and easier to store; they readily accommodate an enormous range of scales; they are viewed easily on computer displays; they can facilitate measuring properties of the terrain being mapped; they can show larger portions of the earth's surface at once; and they are cheaper to produce and transport.

3.3.2.1 Metric properties of maps

Many properties can be measured on the earth's surface independently of its geography. Some of these properties are:

- Area
- Shape

[2]Extensive information concerning this section can be found at [10] [11] [12] [13]

[3]Theorema Egregium is a foundational result in differential geometry proved by Carl Friedrich Gauss that concerns the curvature of surfaces. Informally, the theorem says that the Gaussian curvature of a surface can be determined entirely by measuring angles and distances on the surface itself, without further reference to the particular way in which the surface is situtated in the ambient 3-dimensional Euclidean space. Thus the Gaussian curvature is an intrinsic invariant of a surface.

- Direction
- Bearing
- Distance
- Scale

Map projections can be constructed to preserve one or some of these properties, though not all of them simultaneously. Each projection preserves or compromises or approximates basic metric properties in different ways. The purpose of the map, then, determines which projection should form the base for the map. Since many purposes exist for maps, so do many projections exist upon which to construct them.

Another major concern that drives the choice of a projection is the compatibility of data sets. Data sets are geographic information. As such, their collection depends on the chosen model of the earth. Different models assign slightly different coordinates to the same location, so it is important that the model be known and that chosen projection be compatible with that model. On small areas (large scale) data compatibility issues are more important since metric distortions are minimal at this level. In very large areas (small scale), on the other hand, distortion is a more important factor to consider.

3.3.2.2 Construction of a map projection

The overview of the construction of a map projection involves three basic steps:

- Selection of a model for the shape of the earth or planetary body (usually choosing between a sphere or ellipsoid)
- Transformation of geographic coordinates (longitude and latitude) to plane coordinates (eastings and northings or x,y)
- Reduction of the scale

Because the real earth's shape is irregular, information is lost in the first step, in which an approximating, regular model is chosen. Reducing the scale may be considered to be part of transforming geographic coordinates to plane coordinates.

Most map projections, both practically and theoretically, are not "projections" in any physical sense. Rather, they depend on mathematical formulae that have no direct physical interpretation. However, in understanding the concept of a map projection it is helpful to think of a globe with a light source placed at some definite point with respect to it, projecting features of the globe onto a surface.

3.3.2.3 Different projection surfaces and their "development"

A surface that can be unfolded or unrolled into a flat plane or sheet without stretching, tearing or shrinking is called a 'developable surface'. The cylinder, cone and of course the plane are all developable surfaces. The sphere and ellipsoid are not developable surfaces. Any projection that attempts to project a sphere (or an ellipsoid) on a flat sheet will have to distort the image (similar to the impossibility of making a flat sheet from an orange peel).

One way of describing a projection is to project first from the earth's surface to a developable surface such as a cylinder or cone, followed by the simple second step of unrolling the surface into a plane. While the first step inevitably distorts some properties of the globe, the developable surface can then be unfolded without further distortion.

Once a choice is made between projecting onto a cylinder, cone, or plane, the orientation of the shape must be chosen. The orientation is how the shape is placed with respect to the globe. The orientation of the projection surface can be normal (inline with the earth's axis), transverse (at right angles to the earth's axis) or oblique (any angle in between). These surfaces may also be either tangent or secant to the spherical or ellipsoidal globe. Tangent means the surface touches but does not slice through the globe; secant means the surface does slice through the globe. Insofar as preserving metric properties go, it is never advantageous to move the developable surface away from contact with the globe, so that practice is not discussed here.

Below we provide an overview of some of the most popular projection along with images that respond to the develepment phase from the sphere to the flat surface.

Cylindrical Projection

The term "cylindrical projection" is used to refer to any projection in which meridians are mapped to equally spaced vertical lines and circles of latitude (parallels) are mapped to horizontal lines. One can imagine it as the projection of the spherical earth on a cylinder that is wrapped around it. Figure 3.5 shows a graphical representation of this projection.

Conical Projection

The term "conical projection" is used to refer to any projection in which Earth's surface is projected onto a tangent or secant cone, which is then cut from apex to base and laid flat. Figure 3.6 shows a graphical representation of this projection.

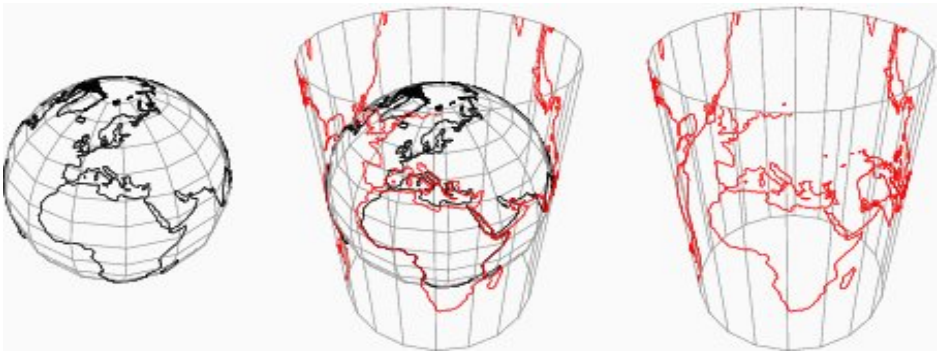


FIGURE 3.5: The cylindrical projection.

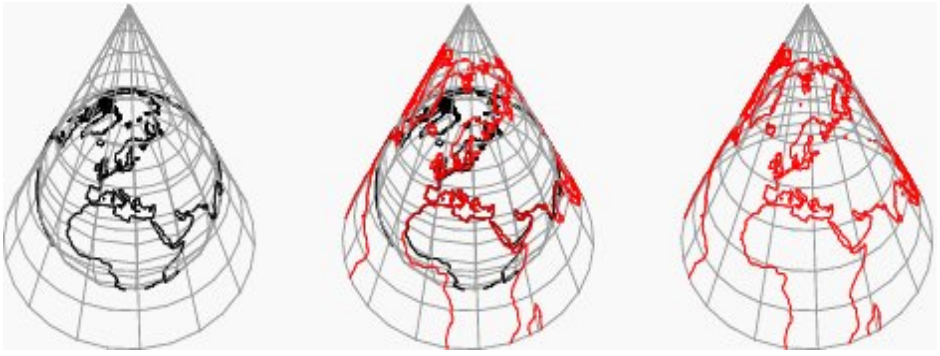


FIGURE 3.6: The conical projection.

Azimuthal Projection

Given a reference point A and two other points B and C on a surface, the azimuth from B to C is the angle formed by the minimum-distance lines AB and AC. In other words, it represents the angle one sitting on A and looking at B must turn in order to look at C. The bearing from A to C is the azimuth considering a pole as reference B. Figure 3.7 shows a graphical representation of this projection.

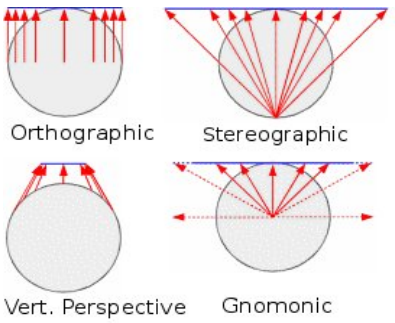


FIGURE 3.7: The azimuthal projection.

3.3.2.4 Projection definitions

It is obvious, that each projection, no matter the selected surface, has numerous unique attribute associated with it. Therefore, there are many standardized ways to define a projection along with its properties. Below is a representation of *US national atlas equal area* projection with its definition using the "human–readble" *Open Geospatial Concortium*'s (OGC) Well–Known–Text (WKT).

```
PROJCS["US National Atlas Equal Area",
    GEOGCS["Unspecified datum based upon the Clarke 1866 Authalic Sphere",
        DATUM["Not_specified_based_on_Clarke_1866_Authalic_Sphere",
            SPHEROID["Clarke 1866 Authalic Sphere",6370997,0,
                AUTHORITY["EPSG","7052"]],
            AUTHORITY["EPSG","6052"]],
        PRIMEM["Greenwich",0,
            AUTHORITY["EPSG","8901"]],
        UNIT["degree",0.01745329251994328,
            AUTHORITY["EPSG","9122"]],
        AUTHORITY["EPSG","4052"]],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    PROJECTION["Lambert_Azimuthal_Equal_Area"],file:///home/motley/Desktop/Thesis/myt

    PARAMETER["latitude_of_center",45],
    PARAMETER["longitude_of_center",-100],
    PARAMETER["false_easting",0],
    PARAMETER["false_northing",0],
    AUTHORITY["EPSG","2163"],
    AXIS["X",EAST],
    AXIS["Y",NORTH]]
```

Of course, each software vendor can define its own method of defining projections as long as it supports the standard options. In sake of simplicity there exist a special definition for each projection, based on a simple *European Petroluem Survey Group (EPSG) code*. The EPSG code is just a number which acts as a *"primary"* key to each software's database of map projection defiitions. For example, the above projection definition is assigned a EPSG code of **2163**.

3.3.3 Geocoding principles

Geocoding is the process of finding associated geographic coordinates (often expressed as latitude and longitude) from other geographic data, such as street addresses, or zip codes (postal codes). With geographic coordinates the features can be mapped and entered into Geographic Information Systems, or the coordinates can be embedded into media such as digital photographs via geotagging. Reverse Geocoding is the opposite: finding an associated textual location such as a street address, from geographic coordinates. A **geocoder** is a piece of software or a (web) service that helps in this process.
A simple method of geocoding is *address interpolation*. This method makes use of data from a street geographic information system where the street network is already mapped within the geographic coordinate space. Each street segment is attributed with address ranges (e.g. house numbers from one segment to the next). Geocoding takes an address, matches it to a street and specific segment (such as a block, in towns that use the "block" convention). Geocoding then interpolates the position of the address, within the range along the segment. As an example[14] we can use the *742 Evergreen Terrace* address; Let us assume that this segment (for instance, a block) of Evergreen Terrace runs from 700 to 799. Even-numbered addresses would fall on one side (e.g. west side) of Evergreen Terrace, with odd-numbered addresses on the other side (e.g. east side). 742 Evergreen Terrace would (probably) be located slightly less than halfway up the block, on the west side of the street. A point would be mapped at that location along the street, perhaps offset some distance to the west of the street centerline.
Difficulties arise when:

- Distinguishing between ambiguous addresses such as 742 Evergreen Terrace and 742 W Evergreen Terrace.
- Geocoding new addresses for a street that is not yet added to the geographic information system database.

A very common error is to believe the accuracy ratings of a given map's geocodable attributes. Such "accuracy" currently touted by most vendors has no bearing on an address being attributed to the correct segment, being attributed to the correct "side" of the segment, nor resulting in an accurate position along that correct segment. With the geocoding process used for U.S. Census TIGER datasets, 5-7.5% of the addresses may be allocated to a different census tract, while 50% of the geocoded points might be located to a different property parcel. [15]

3.3.4 Topological overlays

The combination of *several* spatial datasets (points, lines or polygons) creates a new output vector dataset, visually similar to stacking several maps of the same region and is the most required and common technique in geographic data processing. These overlays are similar to mathematical Venn diagram overlays. A *union* overlay combines the geographic features and attribute tables of both inputs into a single new output. An *intersect* overlay defines the area where both inputs overlap and retains a set of attribute fields for each. A *symmetric difference* overlay defines an output area that includes the total area of both inputs except for the overlapping area.
Topological overlay is predominantly concerned with overlaying polygon data with polygon data, e.g. soils and forest cover. However, there are requirements for overlaying point, linear, and polygon data in selected combinations, e.g. point in polygon, line in polygon, and polygon on polygon are the most common. Vector and raster based software differ considerably in their approach to topological overlay; In raster data analysis, the overlay of datasets is accomplished through a process known as *local operation on multiple rasters* or *map algebra*, through a function that combines the values of each raster's matrix. This function may weight some inputs more than others through use of an *index model* that reflects the influence of various factors upon a geographic phenomenon [5].

Generally, GIS software implements the overlay of different vector data layers by combining the spatial and attribute data files of the layers to create a new data layer. Again, different GIS software utilize varying approaches for the display and reporting of overlay results. Some systems require that topological overlay occur on only two data layers at a time, creating a third layer. Figure 3.8 illustrates a typical overlay requirements where several different layers are spatially joined to created a new topological layer. By combining multiple layers in a topological fashion complex queries can be answered concerning attributes of any layer [1].

3.4 WebGIS and Web mapping

The synergy of Geographical Information Systems and Web Technology allows access to dynamic geospatial information without burdening the users with complicated and expensive software. The World Wide Web provides GIS users easy access to spatial data through a simple browser interface or sometimes by a lightweight client side application. The concept of **WebGIS** is based on how the map is produced and responds to users' interactions over the Web. The publication and distribution of spatial data

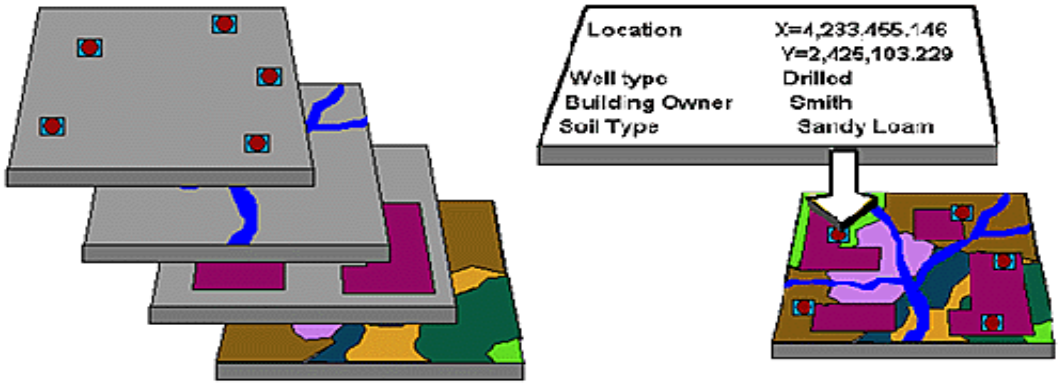


FIGURE 3.8: Sample overlaying.

are increasingly important activities enabling organizations to share domain-specific dynamic spatial information over the Web.

Web mapping is the process of designing, implementing, generating and delivering maps on the World Wide Web. WebGIS is similar to web mapping but with an emphasis on analysis, processing of project specific geodata and exploratory aspects. Often the terms webGIS and web mapping are used synonymously, even if they don't mean exactly the same. In fact, the border between web maps and web GIS is blurry. Web maps are often a presentation media in webGIS while they are increasingly gaining analytical capabilities. In the current work, we present GeoMoin which is originally a web–mapping application, but due to its ability of supporting analytical operations either natively or by building extensions via scripting, allows for a classification within the more general realm of WebGIS applications. Thus, throughout the document, the two terms, will be used synonymously.

The use of the web as a dissemination medium for maps can be regarded as a major advancement in cartography and opens many new opportunities, such as realtime maps, cheaper dissemination, more frequent and cheaper updates of data and software, personalized map content, distributed data sources and sharing of geographic information. It also implicates many challenges due to technical restrictions (low display resolution and limited bandwidth, in particular with mobile computing devices, many of which are physically small, and use slow wireless Internet connections) and security issues, reliability issues and technical complexity. While the first web maps were primarily static, due to technical restrictions, today's web maps can be fully interactive and integrate multiple media. This means that both web mapping and web cartography also have to deal with interactivity, usability and multimedia issues.

3.4.1 Types of web–mapping

[4]A first classification of web maps has been made by Kraak[5]. He distinguished static and dynamic web maps and further distinguished interactive and view only web maps. However, today in the light of an increased number of different web map types, this classification needs some revision. Today, there are additional possibilities regarding distributed data sources, collaborative maps, personalized maps, etc.

The following graphic lists potential types of web maps. While the graphic shows in principle an order of increasing sophistication, the allocation within the order is not explicit. Many maps fall into more than one category and it is not always clear that a personalized web map is more complex or sophisticated than an interactive web map. Individual web map types and their application within GeoMoin are discussed below.

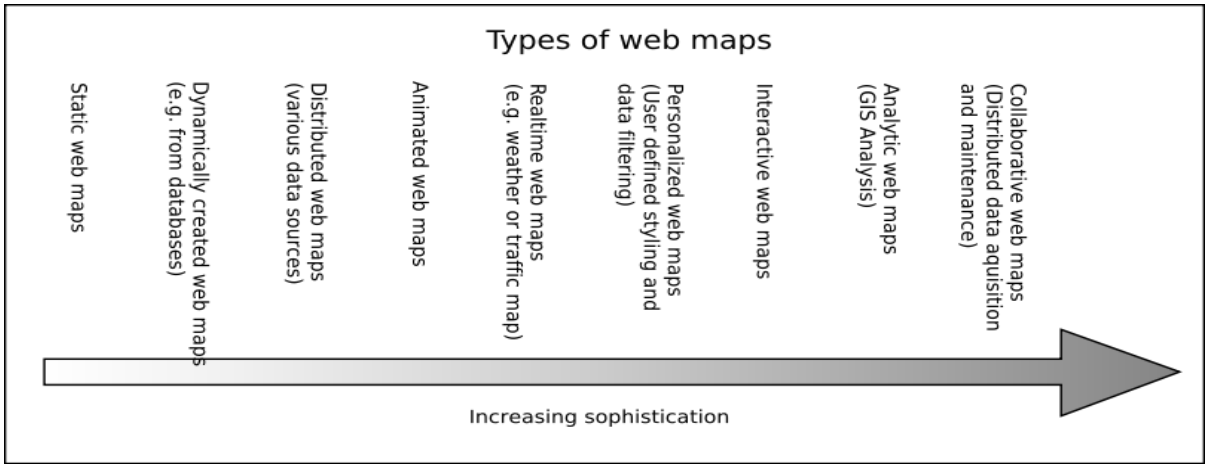


FIGURE 3.9: Web mapping applications' classification.

Static Static web pages are view only with no animation and interactivity. They are only created once, often manually and infrequently updated. Typical graphics formats for static web maps are PNG, JPEG, GIF, or TIFF (e.g., drg) for raster files, SVG, PDF or SWF for vector files. Often, these maps are scanned paper maps and had not been designed as screen maps. GeoMoin can be configured to present static maps without any interaction with the end user.

Dynamic These maps are created on demand each time the user reloads the webpages, often from dynamic data sources, such as databases. The webserver generates the map using a web map server or a self written software. The use of MapServer

[4]The following information are presented from the Wikipedia article on web mapping.

[5]Kraak, Menno Jan (2001): Settings and needs for web cartography, in: Kraak and Allan Brown (eds), Web Cartography, Francis and Taylor, New York, p. 34.

and dynamic web development allow the presentation of dynamic spatial content within GeoMoin.

Distributed These maps are created from distributed data sources. The WMS protocol offers a standardized method to access maps on other servers. WMS servers can collect these different sources, reproject the map layers, if necessary, and send them back as a combined image containing all requested map layers. One server may offer a topographic base map, while other servers may offer thematic layers. GeoMoin makes extensive use of distributed OGC web services (OWS) and especially WMS and WFS, while every map can be configured to act as a client for the distributed content, or as a server of the above web–services.

Animated Animated Maps show changes in the map over time by animating one of the graphical or temporal variables. End–user browsers usually need to support third–party software like Quicktime, Flash player, Java applets. This feature is not developed within GeoMoin.

Realtime Realtime maps show the situation of a phenomenon in close to realtime (only a few seconds or minutes delay). Data is collected by sensors and the maps are generated or updated at regular intervals or immediately on demand. Examples are weather maps, traffic maps or vehicle monitoring systems. This feature is not developed and tested within GeoMoin, but the extensive use of the PostgreSQL spatially–enabled database could easily provide means of implementing realtime features like GPS tracking, or traffic information announcements.

Personalized Personalized web maps allow the map user to apply his own data filtering, selective content and the application of personal styling and map symbolization. The OGC (Open Geospatial Consortium) provides the SLD standard (Styled Layer Description) that may be sent to a WMS server for the application of individual styles. The fact the every user can create or manage maps and geospatial content allows for high personalization within GeoMoin.

Interactive Interactivity is one of the major advantages of screen based maps and web maps. It helps to compensate for the disadvantages of screen and web maps. Interactivity helps to explore maps, change map parameters, navigate and interact with the map, reveal additional information, link to other resources, and much more. Technically, it is achieved through the combination of events, scripting and DOM manipulations. Using GeoMoin, users have full control of the rendering of a map including panning, zooming, querying, displaying and creating annotations along with many other features.

Analytic These web maps offer GIS analysis, either with geodata provided, or with geodata uploaded by the map user. As already mentioned, the borderline between analytic web maps and web GIS is blurry. Often, parts of the analysis are carried out by a serverside GIS and the client displays the result of the analysis. GeoMoin's native components and extension via the use of mapplets allow GeoMoin to be used as a mean of light–weight spatial analysis.

Collaborative Collaborative maps are still new, immature and complex to implement, but show a lot of potential. The method parallels the Wikipedia project where various people collaborate to create and improve maps on the web. Technically, an application allowing simultaneous editing across the web would have to ensure that geometric features being edited by one person are locked, so they can't be edited by other persons at the same time. Also, a minimal quality check would have to be made, before data goes public. GeoMoin was primarily developed to serve as a collaborative mean of distributing spatial content over the web. Users can upload and install spatial content, create or manage maps and improve existing maps and spatial information contributed by other users.

3.4.2 Advantages of web–mapping and WebGIS

- Web maps can easily deliver up to date information. If maps are generated automatically from databases, they can display information in almost realtime. They don't need to be printed, mastered and distributed. Examples:

- Software and hardware infrastructure for web maps is cheap. Web server hardware is cheaply available and many open source tools exist for producing web maps.

- Product updates can easily be distributed. Because web maps distribute both logic and data with each request or loading, product updates can happen every time the web user reloads the application. In traditional cartography, when dealing with printed maps or interactive maps distributed on offline media (CD, DVD, etc.), a map update caused serious efforts, triggering a reprint or remastering as well as a redistribution of the media. With web maps, data and product updates are easier, cheaper, and faster, and can occur more often.

- They work across browsers and operating systems. If web maps are implemented based on open standards, the underlying operating system and browser do not matter.

- Web maps can combine distributed data sources. Using open standards and documented APIs one can integrate (mash up) different data sources, if the projection

system, map scale and data quality match. The use of centralized data sources removes the burden for individual organizations to maintain copies of the same data sets. The down side is that one has to rely on and trust the external data sources.

- Web maps allow for personalization. By using user profiles, personal filters and personal styling and symbolization, users can configure and design their own maps, if the web mapping systems supports personalization. Accessibility issues can be treated in the same way. If users can store their favourite colors and patterns they can avoid color combinations they can't easily distinguish (e.g. due to color blindness).

- Web maps enable collaborative mapping. Similar to the Wikipedia project, web mapping technologies, such as DHTML/Ajax, SVG, Java, Adobe Flash, etc. enable distributed data acquisition and collaborative efforts. Examples for such projects are the OpenStreetMap project or the Google Earth community. As with other open projects, quality assurance is very important however.

- Web maps support hyperlinking to other information on the web. Just like any other web page or a wiki, web maps can act like an index to other information on the web. Any sensitive area in a map, a label text, etc. can provide hyperlinks to additional information. As an example a map showing public transport options can directly link to the corresponding section in the online train time table.

- It is easy to integrate multimedia in and with web maps. Current web browsers support the playback of video, audio and animation (SVG, SWF, Quicktime, and other multimedia frameworks).

3.4.3 Disadvantages and problematic issues

- *Reliability issues* – the reliability of the internet and web server infrastructure is not yet good enough. Esp. if a web map relies on external, distributed data sources, the original author often cannot guarantee the availability of the information.

- *Geodata is expensive* – Unlike in the US, where geodata collected by governmental institutions is usually available for free or cheap, geodata is usually very expensive in Europe or other parts of the world.

- *Bandwidth issues* – Web maps usually need a relatively high bandwidth.

- *Limited screen space* – Like with other screen based maps, web maps have the problem of limited screen space. This is in particular a problem for mobile web

maps and location based services where maps have to be displayed in very small screens with resolutions as low as 100100 pixels. Hopefully, technological advances will help to overcome these limitations.

- *Quality and accuracy issues* – Many web maps are of poor quality, both in symbolization, content and data accuracy.
- *Complex to develop* – Despite the increasing availability of free and commercial tools to create web mapping and web GIS applications, it is still a complex task to create interactive web maps. Many technologies, modules, services and data sources have to be mastered and integrated.
- *Immature development tools* – Compared to the development of standalone applications with integrated development tools, the development and debugging environments of a conglomerate of different web technologies is still awkward and uncomfortable.
- *Copyright issues* – Many people are still reluctant to publish geodata, esp. in the light that geodata is expensive in some parts of the world. They fear copyright infringements of other people using their data without proper requests for permission.
- *Privacy issues* – With detailed information available and the combination of distributed data sources, it is possible to find out and combine a lot of private and personal information of individual persons. Properties and estates of individuals are now accessible through high resolution aerial and satellite images throughout the world to anyone.

3.4.4 WebGIS and Spatial Desicion Support

A **Spatial Decision Support System (sDSS)**[6] is an interactive, computer-based system, designed to support a user or group of users in achieving a higher effectiveness of decision making while solving a semi-structured spatial problem. It is designed to assist the spatial planner with guidance in making land use decisions. For example, when deciding where to build a new airport many contrasting criteria, such as noise pollution vs. employment prospects or the knock on effect on transportation links, which make the decision difficult. A system which models decisions could be used to help identify the most effective decision path.

A spatial decision support system typically consists of the following components [7].

[6] An sDSS is sometimes referred to as a Policy Support System

[7] This concept fits dialog, data and modelling concepts outlined in Sprague, R. H. and H. J. Watson (1996) Decision support for management. Upper Saddle River, N.J.: Prentice Hall.

1. A database management system – This system holds and handles the geographical data. A standalone system for this is called a Geographical Information System, (GIS).

2. A library of potential models and methods that can be used to forecast the possible outcomes of decisions.

3. An interface to aid the users interaction with the computer system and to assist in analysis of outcomes.

An sDSS usually exists in the form of a computer model or collection of interlinked computer models, including a land use model while various techniques are available to simulate land use dynamics. An sDSS typically uses a variety of spatial and nonspatial information, like data on land use, transportation, water management, demographics, agriculture or climate. By using two (or, better, more) known points in history the models can be calibrated and then projections into the future can be made to analyze different spatial policy options. Using these techniques spatial planners can investigate the effects of different scenarios, and provide information to make informed decisions. To allow the user to easily adapt the system to deal with possible intervention possibilities, an interface allows for simple modification to be made.

Spatial decision support applications is leaning towards WebGIS applications, as they allow researchers and stakeholders to benefit from sharing, analyzing and visualizing large, up-to-date geospatial data sets with minimal effort and cost. Moreover, the integration of open-source, open-standards software packages and web design technologies result in a powerful tool for the developing of interactive web mapping portals for spatial analysis. FOSS offers a level of flexibility, availability and lowered cost that is typically unavailable with commercial software, while an architecture and design based on open standards ensures system interoperability and data reusability.

Chapter 4

The MapServer Package

4.1 What is MapServer

MapServer is an Open Source development environment for building spatially-enabled internet applications [16]. It was originally developed by the *University of Minnesota (UMN)* in collaboration with NASA, due to the NASA's need to make its satellite imagery available to the public. Later the project was hosted by the TerraSIP project, a NASA sponsored project between the UMN and a consortium of land management interests. MapServer is now a project of OSGeo [4.2.2], and is maintained by a growing number of developers (nearing 20) from around the world. It is supported by a diverse group of organizations that fund enhancements and maintenance, and administered within OSGeo by the MapServer Project Steering Committee made up of developers and other contributors. Mapserver is lead–developed by *Stephen Lime* and by the time this work is written, its release version is *5.2.0*.

It is important to state that MapServer is not a full–featured GIS system in terms of *spatial analysis* and *data processing*, nor does it aspire to be. Instead, **MapServer excels at rendering spatial data (maps, images, and vector data) for the web**. Beyond browsing GIS data, MapServer allows a developer to create "geographic image maps", that is, maps that can direct users to content. For example, the *Minnesota DNR Recreation Compass*[1] provides users with more than 10,000 web pages, reports and maps via a single application. The same application serves as a "map engine" for other portions of the site, providing spatial context where needed.

[1] http://www.dnr.state.mn.us/maps/compass.html

Slighty altering Stephen Lime's foreward on Bill Kropla's book *"Beginning MapServer: Open Source GIS Development"* we could note that, at its essence, MapServer is conceptually very simple, but unless someone shares the thought processes of the core developers, the learning curve can be a bit steep. For many open source projects, documentation is a weak point, and MapServer is no exception. That is, the documentation is scattered loosely across mailing lists, sample applications and reference web–pages. Therefore, it is important to dedicate some sections describing the fundamental aspects of MapServer, addressing its features and some important "work–arounds".

4.2 Setting the terrain for FOSS GIS development

4.2.1 The Open Geospatial Consortium

[17]The Open Geospatial Consortium, frequently refered as *OGC*, is an international voluntary consensus standards organization. In the OGC, many commercial, governmental, nonprofit and research organizations worldwide collaborate in an open process, encouraging development and implementation of standards for geospatial content and services, GIS data processing and exchange. It was previously known as *Open GIS Consortium*.

Most of the OGC specifications are based on a generalized architecture captured in a set of documents collectively called the *Abstract Specification*, which describes a basic data model for geographic features to be represented. Atop the Abstract Specification is a growing number of *specifications*, or *standards*, that have been (or are being) developed to serve specific needs for interoperable location and geospatial technology, including GIS.

Below we provide a list of the most important OGC specification up–to–date:

OGC Reference Model a complete set of reference models.

WMS Web Map Service: Provides map images.

WFS Web Feature Service: For retrieving or altering feature descriptions. Provides information in the Geography Markup Language (GML) format.

WCS Web Coverage Service: Provide coverage objects from a specified region.

WPS Web Processing Service: Remote processing service.

CSW Web Catalog Service: Access to catalog information.

SFS Simple Features - SQL

GML Geography Markup Language: XML format for geographical information.

KML Keyhole Markup Language: XML-based language schema for expressing geographic annotation and visualization on existing or future Web-based, two-dimensional maps and three-dimensional Earth browsers.

WSC Web Service Common

The OGC has a close relationship with ISO/TC 211 (Geographic Information/Geomatics). The OGC abstract specification is being progressively replaced by volumes from the ISO 19100 series under development by this committee. Further, the OGC standards Web Map Service, GML and Simple Features Access are ISO standards.

The OGC works with other international standards bodies including World Wide Web Consortium (W3C) and Organization for the Advancement of Structured Information Standards (OASIS).

4.2.2 The Open Source Geospatial Foundation

[18]The Open Source Geospatial Foundation (OSGeo), is a non-profit non-governmental organization whose mission is to support and promote the collaborative development of open geospatial technologies and data. The foundation was formed in February 2006 to provide financial, organizational and legal support to the broader Free and open source geospatial community. It will also serve as an independent legal entity to which community members can contribute code, funding and other resources, secure in the knowledge that their contributions will be maintained for public benefit. OSGeo projects include:

- *Geospatial Libraries:*
  - FDO
  - GDAL/OGR
  - GeoTools
- *Desktop Applications:*
  - GRASS GIS
  - OSSIM
  - Quantum GIS
- *Web Mapping:*
  - Mapbender
  - Mapserver

- Mapbuilder
- Mapguide OpenSource
- OpenLayers

4.3 Intoducing MapServer capabilities

[19] MapServer creates maps from spatial information stored in digital format, being able to handle both vector and raster data. Using the OGR library it can access over 20 different vector type formats including shapefiles, PostGIS and ArcSDE geometries, OPeNDAP, Mitab/MapInfo and USGS TIGER datasets.
Additionally a map rendered by MapServer can simultaneously use both vector and raster formats. For example, an aerial or satellite photo of a region can be overlayed below rendered vector data to provide a clearer picture or "background" of how these vector elements relate to real–world features. Natively, MapServer suppports two raster/image formats: GeoTIFF and EPPL7, but using the GDAL library can expand its abilities to a large number of formats including Windows bitmap, GIF and JPEG.

Mapserver can operator in **two different modes:** *CGI* and *MapScript*. In CGI mode, mapserver function in a web server environment as a CGI script, accepting requests and generating responses. Being a fact that a CGI implementations are particularly slow due to the creation of many different processes each timee the script is called, this modes serves for easy set–up and producing of an easy–to–build and straightforward low–traffic application. On the contrary in MapScript mode, a programmer has access to MapServer's API through Perl,Python or PHP. This interface allows for a flexible, feature–rich application giving the programmer the ability to access MapServers' advanced features. The requirements for a project like GeoMoin, set the selection of mapscript as mandatory.

Natively[2], MapServer is a *template–based* application. When first executed in response to a HTTP request, it reads a configuration file,described later, the *mapfile* which contains the layer definitions and other components of a map. Next, it reads one or more HTML templates identified in the map. Each template consist of the traditional HTML tags and some special Mapserver *substitution strings*. These strings specify, for example, the path where the rendered image is stored or the zoom level and direction. Thus, after MapServer substitutes the strings with the correct values, it sends the data stream to the web–server which afterwards forwards it to the user–agent. When the user–agent

[2]With the word "natively" we mean that the templating capabilities are crucial when MapServer operates in CGI mode, although MapScript can still take advantage of it.

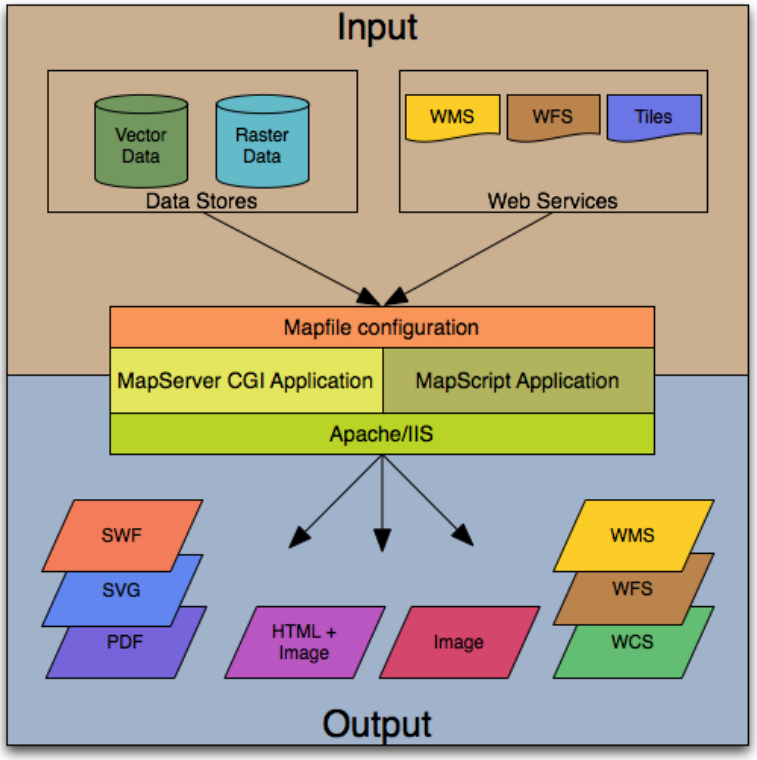


FIGURE 4.1: Mapserver architecture (source: UMN MapServer documentation)

changes any HTML form elements on the page and submits the results, MapServer recieves a new request and the cycle of events starts again.
The use of mapscript exposes the programmer to a more advanced interface allowing access to every function and data type used to built the cgi version itself. In case of mapscript programming, the mapserver libraries are being imported, mapserver objects are instantiated using mapfiles and script–specific needs. After the business logic of the application is executed the output can be either provided with a web–programming interface (PHP,python etc) to the web–client or even a linux shell as a standalone application.

Mapserver automatically performs several tasks when generating a map. It labels features and prevents collisions between neighboring labels. It provides for the use of bitmapped or TrueType fonts. Label sizes can be fixed or configured to scale with the scale of the map. It can also create *legends* and *scalebars* or *reference maps*.

Mapserver builds map, by stacking layers on top of each other. As each is rendered, it is placed on top of the stack. Every layer displays features selected from a single data set. Features to be displayed can be selected by using Regular Expressions, string comparisons or logical expressions. Because of the similarity of data and the similarity

of the styling parameters (like scale,color and labels), one can think of a layer definition as a theme. The display of layers is under user control, allowing him to decide which layers will be rendered. With the use of MapScript, layers can be generated on the fly and can be populated with dynamic data.

MapServer additionally provides query capabilities which will be described later in detail, but in CGI mode it lacks the tools that allow the kind of analysis provided by a true GIS. It is important to note, that a mapserver programmer, may not only rely on the native tools supported by the mapscript library. On the contrary, one can import libraries like *GDAL* and *GEOS*, which in conjuction to mapscript introduce a highly advanced framework for GIS developement.

This overview described some of MapServer's features and also shown why it is not a full GIS: it provides no integrated DBMS tools, in it's native mode the analytical capabilities are limited, and it has not tools for georeferencing. But since MapServer's function can be accessed via an API, it can serve as the foundation of a powerful spatially aware application that has many analytical and reporting functions like a typical GIS. In addition, while there are no integrated means of manilupating spatial data, there are third–party tool set that perform these kinds of actions.
Summarizing, when a run as CGI in a web environment, MapServer can render maps, display attribute data and perform rudimentary spatial queries. When accessed via the API, the application becomes significantly more powerful. In this environment, MapServer can perform the same task it would as CGI, but it also has access to external databases via program control, as well as more complex logic and a larger repertoire of possible behaviours.

4.4 Mapserver's configuration: The Mapfile

In this section, we will describe the most important element of a MapServer application, the *mapfile*. The mapfile defines a collection of mapping objects that together determine the appearance and *behaviour* of a map as displayed in the web–browser. Based on the same datasets, an application can provide different mapfiles which respond differently to user actions. Although it might seem that the use of a static configuration file would have limited functionality, the design of MapServer and the format of the mapfile allow the development of powerful applications.
Mapfile definitions follow the typical "keyword–value pair" parsing styling. Some values are lists of items separated by white space and frequently enclosed in quotes. Single quotes and double quotes are both acceptable. It is important to note that the mapfile is *partially–case–sensitive*, that is, for example some database access methods require

case–sensitivity. A complete reference to mapfile directives is provided by the MapServer community here [20] while important tutorials are provided here [21].

4.4.1 A simple mapfile

Let's present a simple mapfile as a "Hello World" application with a line–by–line description, which is helpful for the reader to get an insight of how things work, before presenting more complex mapfile directives:

```
01    # This is a comment
02    NAME "Helloworld"
```

Line 01 is a single line comment, which can exist even after a mapfile directive. Note that MapServer does not support multi–line comments. Line 02 defines the name of the map. In a CGI mode, the NAME acts as the prefix for the rendered images filenames, so it is a good practise to be kept short and without spaces.

```
03    SIZE 400 300
04    IMAGECOLOR 250 0 0
05    IMAGETYPE png
06    EXTENT -1.00 -1.00 1.00 1.00
```

Line 03 defines the length and width of the rendered image in pixels. Line 04 defines the background color of the map in a R G B (red green blue) fashion, in that case using a red background. Line 05 instructs MapServer to create a rendered image of type "PNG". The geographic extent, the rectangular area covered in real coordinates, is crucial to a mapfile design. Along with the map scale and the output size, some computations take place to define what will be rendered. For example, if a dataset is used consisting of the capital cities in the world and the extent is "136 30 150 45", the rendered output will be an image of 400 pixels length and 300 pixels width, consisting of cities spatially–included in the rectangle defined using those coordinates. This example though will not have any real–world association, so an arbitrary extent can be used provide that if follows the rule below:

> The EXTENT format is of type: minx miny maxx maxy, with the first pair of coordinates describing the minimum point in the map which is the *lower left corner* and the second pair describing the maximum point of the map which is the *upper right corner*.

```
07    WEB
08       TEMPLATE "/var/www/hello.html"
09       IMAGEPATH "/var/www/tmp/"
10       IMAGEURL "/tmp/"
11    END
```

In order for a user–agent to display a map, it must be embed in a web–page or a *template*. A WEB object defines the name of this template and its crucial parameters. MapServer afterwards uses these parameters to compute the string substitutions which were mentioned earlier. Lines 07 and 11 begin and end a web object as a parsing directive. Line 08 defines the name of the template file along with its path in the filesystem. Line 09 shows where the rendered images will be stored in the filesystem. This folder must be accessible via the web–server so that the CGI application can have access to the rendered image. Thus, IMAGEURL on line 10 defines its web location with respect to the web server's DocumentRoot.
Now MapServer knows what type of image to produce, what size and how to embed it in a HTML page to send it to the user–agent. But it doesn't know **what** to render. This role is attributed to a special mapfile object, the *Layer*.

A layer is referencing a single dataset and contains a set of elements tat will be rendered in a particular scale, using a particular projection.

```
12    LAYER
13       STATUS default
14       TYPE point
```

Line 12 starts a layer definition. Line 13 defines that the layer will always be rendered. STATUS keyword can also be *"off"* or *"on"*, specifying if the layer will be visible or not, *as long as the user does not change its status*, that is, the user can set which layer will be visible or not within the application. A "default" is *always on*.
Each layer has a geometric type associated with it. This means that the dataset elements will be treated by MapServer as a point, or polygon or whatever the "TYPE" directive defines. Thus, if a geometry is created to be used as *line* (a list of points) and the mapfile's layer definition commands this geometry to be treated as *point*, it is obvious that the rendered image will contain only points and NOT lines connecting the line coordinates. In our example, the geometry will be of type "point".

Now that MapServer knows the type of the data of the layer, the next step is to provide them. Data can be contained in many format, as well as, in many sources. For this

example we will use a special source of data, the *inline data*. These data are enclose between the FEATURE parsing instruction as shown below:

```
15        FEATURE
16           POINTS 0.0 0.0 END
17           TEXT "Hello World"
18        END
```

Line 16 defines where the point will be draws in respect to the map extent. Having as extent: -1 -1 1 1, this point will be drawn in the middle of the map. Line 17 defines a text label associated with this point, which can be rendered on the image.

The only thing that MapServer is still not aware, is *how* the data will be rendered. For example, what color the point will have, will it be a point or a symbol or how what font the label will use. These tasks are governed by the CLASS object.

```
19        CLASS
20           STYLE
21               COLOR 0 255 0
22               SIZE 10
23           END
24           LABEL
25               TYPE bitmap
                 SIZE 4
                 COLOR 120 0 0
26           END
27        END
28    END #end layer
29    END #end mapfile
```

In line 19 a CLASS object is being created. Lines 20–23 declare a STYLE object which the defines the color and the size of the dot that wil represent the point on the rendered image. Line 24–26 declare a LABEL object which defines how the label will be rendered, in that case using bitmap fonts. Finally, the layer object and the mapfile object are closed.

4.4.2 Associating datasets to a LAYER object

In the previous example we saw that a layer object can be populated with *inline* elements using the *FEATURE* directive. Although straight–forward, this methodology doesn't

work for real–world problems that MapServer was created to solve. For example, if a designer wants to reference all the restaurants in a single town, he comes across the following issues:

- There is probably a large number of features to be referenced.
- There is a need for manipulating those features in the future.
- It is common that these features should be available to many different maps.

Georeferencing real–world features is a process commonly done using full GIS frameworks like GRASS. These GIS provide a dataset in a particular format contained these features along with the non–spatial attribute they contain. A typical example, is ESRI shapefiles or TIGER/Line datasets. This formats gathers a set of files containing the *georeferenced* features, the non–spatial attributes and optionally the projection parameter and a precompiled data indexing.
MapServer is natively aware of some formats used in the industry, but, as we mentioned before, using OSGeo's library GDAL/OGR it can support many off–the–shelf raster or vector formats.

In order to add an ESRI dataset as input to a layer object, some directives must be added. Let's assume that the ESRI dataset's file consist of the following:

countries.shp The main shapefile file. Maps the geographic features to the tables containg the attribute information.

countries.dbf The database of the feature attribute, resembling a relational model.

countries.shx This file provided the indexing of the features for faster update and retrieval.

First of all, in the global MAP object a directive defining the path where the dataset is located can be declared:

```
SHAPEPATH "/var/www/shapefolder"
```

Finally in the layer level (the layer object):

```
LAYER
    DATA "countries"
```

The SHAPEPATH directive can be ommited provided that the DATA directive will include the full path to the .shp file:

```
LAYER
    DATA "/var/www/shapefolder/countries.shp"
```

As a second example, we will add a layer definition using a MapInfo (MITAB) dataset. MapServer does not provide native support for this type of datasets, but it can through the OGR[3] library. In this case a layer definition should look like this:

```
LAYER
   NAME "mitab_layer"
   TYPE POLYGON
   CONNECTIONTYPE OGR
   CONNECTION "/var/www/mitabfolder/countries.tab"
   STATUS ON
   CLASS
     ...
   END
   ...
END
```

The new directive CONNECTIONTYPE defines a connection through the OGR library, while the CONNECTION directive defines the name of the main MapInfo file.

Finally, we must provide an example of a connection to a DBMS holding the dataset. Assuming that the spatially–aware DBMS is PostgreSQL and is equipped with the PostGIS extension which will be explained later, a layer definition should look like this:

```
LAYER
  NAME postgis_layer
  STATUS ON
  TYPE POLYGON
  CONNECTIONTYPE POSTGIS
  CONNECTION "host=127.0.0.1 dbname=mydb port=5433 user=user1 pass=qwerty"
  DATA "wkb_geometry FROM countries"
  CLASS
```

[3]The OGR Simple Features Library is a C++ open source library (and commandline tools) providing read (and usually write) access to a variety of vector file formats including ESRI Shapefiles, and MapInfo mid/mif and TAB formats.

```
      ...
  END
END
```

The CONNECTIONTYPE defines that a connection to a PostGreSQL database will be established. The CONNECTION defines the connection string which must be provided with the appropriate option like the host and port on which the DBMS is serving, the database name and the credential. Finally in the DATA directive an SQL statement must be provided that returns a table of a Well–Known–Binary (WKB) geometry, described later, derived from the table which holds the dataset. The name "wkb_geometry" along with the name "the_geom" is a very typical naming convention for a spatial column in a database [4].

4.4.3 Using projections within the mapfile

Earlier, we described *projections* as a mean of transfering the three dimensional earth's surface to a two dimensional flat surface. Many projection surfaces exist while each one can have different attributes. For example there can exist a unique *Azimuthal* projection for the area covering Cyprus island located in the Mediterranean sea.
To render a map using MapServer, both the map designer and the application itself must be aware of the projection which was used to create the spatial information. It is common that the spatial information are *unprojected*, that is, they use latitute/longitude pairs in degrees describing the spheroid earth's surface. This representation is called WGS84 and uses the unique EPSG code of 4326. The mapping to an image follows a simple transformation of the geographic coordinates to pixel values. A world map as a flat surface based on WGS84 can be visualized on feature 4.2.

The application can be aware of what projection was used to create data by adding a PROJECTION directive in the LAYER level as shown below:

```
PROJECTION
   "proj=latlong"
   "ellps=WGS84"
   "datum=WGS84"
END
```

[4]The reason behind that is that if a PostGIS table was created using the shp2pgsql tool a column named "the_geom" would emerge and if the ogr2ogr tool was used, a column named "wkb_geometry" would emerge. There exists a quick workaround to grab all the geometry columns contained in a table by issueing this SQL command: "SELECT column_name FROM information_schema.columns WHERE table_name='[TABLE]' and data_type='USER-DEFINED"'

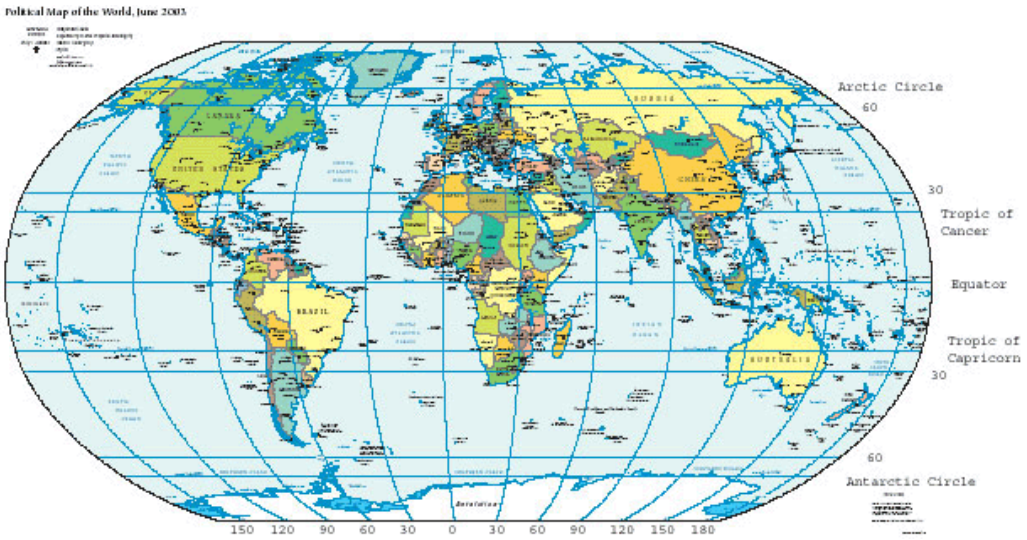


FIGURE 4.2: World map overlaid with latitute/longitude lines

Alternatively the EPSG code can be used as shown below:

```
PROJECTION
    "init=epsg:4326"
END
```

As long as a mapfile can have multiple layer objects, if all layers contain spatial data **in the same projection** there in no need for PROJECTION objects to be declared. MapServer will assume that all data are on the same projection.

MapServer can also support an *output projection* defined on the MAP level. This method can be used to *reproject* a map using a user–defined projection. It is a good practice to always define map level projection whenever possible. A map level definition should look this:

```
PROJECTION
    # This is the definition of Lambert Azimuthal Equal-Area
    # projection for the Continental U.S.
    "proj=laea"
    "ellps=clrk66"
    "lat_0=45"
    "lon_0=-100"
END

PROJECTION
```

```
    # Alternatively, an EPSG code can be specified.
    # This is the EPSG code for Lambert Azimuthal Equal-Area
    # projection for the U.S.
    #
    "init=epsg:2163"
END
```

If a LAYER level projection is now defined, the PROJ.4 library (described in Appendix A) will be used to reproject the spatial data to the MAP level projection. It is importand to note that if a MAP–level projection is specified, and then only **one** other LAYER projection object, MapServer will assume that all of *the other layers are in the specified MAP-level projection.* So it is a good practice to define LAYER level projections (also refered to as "input projections") to all the layers in the mapfile in order to avoid misconceptions caused by the MapServer or to avoid future problems while editing a huge mapfile.
Below we provide two images [4.3] of the Wisconsin area with the first being unprojected and the later, being projected using the Lambert Azimuthal Equal-Area projection for the Continental U.S. The pictures have the same size, but the difference between the two projections is clearly noticable. These images are produced using MapServer.

It is crucial to note that when a map is reprojected to a new MAP level projection (also refered to as *output projection*), *the MAP extents must also be reprojected to the new coordinate system.* A workaround to achieve this is to use the **cs2cs** utility provide by the Proj.4 library. Issuing the command

```
cs2cs +proj=latlong +datum=WGS84 +to
      +proj=laea +ellps=clrk66 +lat_0=45 +lon_0=-100
```

the user can type the first coordinate pair (in the example, it is the minx miny of the map extent) and afterwards the second one (in the example, it is the maxx maxy of the map extent) and the program will output the reprojected values for the extent coordinates. Specifically, the minx/miny coordinate for the area covered in the pictures is *-97.5 41.619778* unprojected. Running the command it outputs the following coordinates: *208398.01 -372335.44 0.000* [5]. Afterwards the map extent can be substituted using the new coordinates.
Usually there exists some distortion while projecting from one system to another, so the map is not centered as someone would expect. Using Desktop GIS application like ArcMAP, manual edits can happen in order to align the data in great detail.

[5]The third number is the altitude and can be ignored if it is not needed.

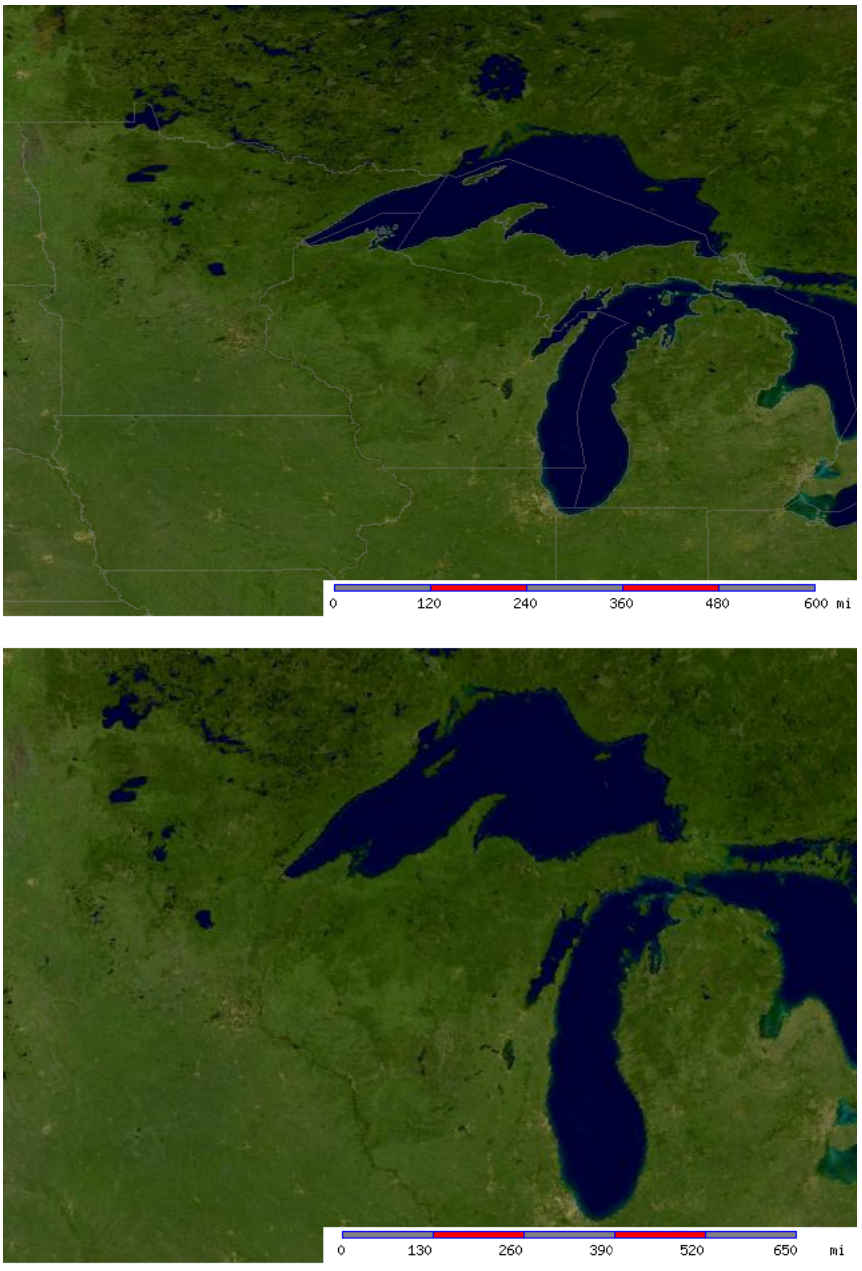


FIGURE 4.3: Satellite images of Wisconsin area unprojected and projected using the LAEA projection respectively.

4.5 Introducing MapScript

As we stated earlier MapServer was natively created to act as a CGI executable. But the reduced abilities caused by a single pre–compiled program cannot match today's needs for a web–mapping application, even to the degree of capabilities that MapServer was designed to offer. Thus, the MapServer team introduces *MapScript*, a more powerful way to use MapServer. MapScript provided access to all of MapServer's underlying functionally, making it available as a convenient API.
MapScript is based on an object oriented architecture, so the API is more properly characterized as a collection of class, methods and attributes. The API is available through the following programming languages:

- PHP
- Python
- Perl
- C#
- Tcl
- Ruby
- Java

Since different languages have different structures and syntax, the API exhibits some language–specific differences. In addition to this there are two parallel maintenance efforts. For example, PHP/MapScript is maintained manually, while Perl and Python versions make use of a software interface generator (SWIG) to auatomate the process.
In this work we are interested and make exclusive use of the Python/MapScript. The reason behind this is that the software we created is based on the MoinMoin wiki engine which is written in Python.
In this section we will describe the basic MapScript functionalities, in order to achieve a degree of abstraction between the GeoMoin's specific workarounds and the more "low–level" details and standard methodologies MapScript offers. Thus this section is not a complete guide to MapScript features, nor it was intended to be. However with those demonstrations, a programmer can dig into the details of the API and broaded his understanding of MapScript.

4.5.1 MapScript objects discussion

In the previous section we discussed some issues surrounding the basic MapServer configuration script, the mapfile. We have seen the mapfile's parsing elements are based on clearly separated parsing logic. For example, the whole mapfile's contents are included in a structure named MAP, whereas each layer containing spatial data is included in its own structure called LAYER, which is included in the MAP structure itself. Feature 4.4 gathers the basic elements of a mapfile and show the parent–child connections between them.

MapScript is based to this particular relationship between the MapServer elements to

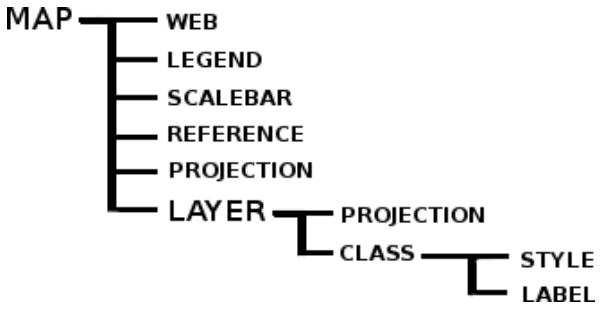


FIGURE 4.4: The basic structure of a mapfile.

provide its object oriented relationships between these elements. Thus, there exists an object called **mapObj** and an object called **layerObj** whose member attribute *layerObj.map*, returns the mapObj.

In addition to the elements of the mapfile that are directly associated with MapScript obejct, mapscript offers objects in order to create new features and facilities. Object derived by these classes include the **pointObj**, **lineObj** and **rectObj**, describing a geographic point, polyline and rectangle respectively.

4.5.2 Rendering a map

The basic function MapScript can perform is to render maps based on a given mapfile. The following python script uses a mapfile and saves the created image in the filesystem.

```
01      import mapscript
02      try:
03        map = mapscript.mapObj("/tmp/mapfile.map")
04      except:
05        print "Error occured while reading the mapfile"
06      image = map.draw()
```

```
07      image.save("/tmp/render.png")
```

In line 01 the mapscript library is being imported to give access to its members. Lines 02–05 create a map object,feeding the constructor with the path to the mapfile, while wrapping the whole call inside a try directive to catch possible exception [6]. Line 06 uses the map object's **draw()** function to render the map. This function call returns an **imageObj** whose **save([path])** function creates and stores the rendered image in the filesystem. Of course, a web–gis application, upon instantiation of the mapObj, includes its business logic in order to drive the rendering to a particular state.

4.5.3 Accessing the features of a layer

In order for a MapServer developer to add extensive analytical capabilities to an application, it is crucial that there is a method of accessing the underlying geographic features within a layer. In terms of MapScript, to access the features of a layerObj.
For this example, let's assume that there exist a mapfile consisting of one layer that holds the cities of USA georeferenced as points. Assuming that the dataset is in the ESRI shapefile format we could have a mapfile that looks like this:

```
NAME "usa_cities"
UNITS DD
EXTENT -180 0 -60 90
SIZE 640 480
IMAGECOLOR 255 255 255
IMAGETYPE PNG

SYMBOL
     NAME "Circle"
     FILLED true
     TYPE ellipse
     POINTS 1 1 END
END

WEB
     IMAGEPATH "/var/www/tmp/"
     IMAGEURL "/tmp"
END
```

[6] Generally, leaving the exception to occur is, of course, a method to debug problems associated with the parsing ofthe mapfile.

```
LAYER
     NAME "cities"
     DATA "/var/www/mapdata/citiesx020.shp"
     STATUS default
     TYPE point
     CLASS
          NAME "US Cities"
          STYLE
               SYMBOL "Circle"
               SIZE 6
               COLOR 0 255 0
          END
     END
END
END # mapfile
```

The above mapfile would be rendered as white image, consisting of green circles of 6 degrees size. Of course such a rendering would be meaningless because the particular dataset contains 35432 features (enough to create a green image). But this example addresses the use of MapScript to analyse these features.
According to the previous example we get the map object like that:

```
01     import mapscript
02     map = mapscript.mapObj("/tmp/mapfile.map")
```

The next thing that needs to be done is to access the layers contained in the map object:

```
03     num_of_layers = map.numlayers
04     for i in range(num_of_layers):
05         layer = map.getLayer(i)
```

Line 03 returns the number of layers which were parsed. In this example the variable num_of_layers will hold a value of *1*. Now we use the python's range function which returns a list of numbers up to the number given as a paramater but not including it. Thus the call *range(num_of_layers)* will return *[0]*. Finally, the map object's member function *getLayer(int index)* is called which returns a reference to the layer at particular index, starting from 0 which is fine because the range function includes the 0. Thus, the variable *layer* now holds a reference to a *layerObj*.

Now that we have a reference to a particular layer, we must try to iterate through all the features it includes as we did before with the layers:

```
06          result = layer.whichShapes(map.extent)
07          layer.open()
08          while (1):
09              shape = layer.nextShape()
10              if shape==None:
11                  break
```

Line 06 is crucial because it opens the underlying layer. This is required before operations like getFeature() will work, but is not required before a draw call. Line 07 is a little more complicated. We discussed earlier that a map extent is the whole geographic region that will be covered by the next render of a map. In fact the map extent, in terms of MapScript is a rectangle object or *rectObj* consisting of the lower–left and upper–right coordinates. One thing that needs special attention is that during the usage of a mapping application showing a particular map, *the map extent is not static*, on the contrary, when the user zooms in or pans after a zoom, the map extent is altered to fit the new rectange the user requested to view. Layer object's member function *whichShapes(rectObj rect)* does the handy job of creating a structure that *will allow access to the features contained inside and only inside the rectange given as a parameter.*
In our example, as the parameterized rectangle, we use the whole extent of the map, which is returned using the call *map.extent*. The result of the function call in line 07 is MS_SUCCESS or MS_FAILURE, containing the values 0 and 1 respectively [7].
The internal structure, preprocessed by whichShape(rectObj) gives the ability to use the layer object's function call *nextShape()*. This function uses this internal structure to return the next feature that is ready to be accessed. In MapScript the geographic features are handled by the *shapeObj class*, which can handle all types of geometric types including points, polylines, rectangles. If the reference to the shapeObj return ny nextShape() is null (or *None*), it means that the iteration has already accessed all the features contained in the current map extent.
In our example, we know that the geometry type of the layer is POINT, so the last thing left to do, is to access the point object itself:

```
10              lineobj  = shape.get(0)
11              pointobj = lineobj.get(0)
12              print " | " + repr(point.x) + " " + repr(point.y) + " | "
```

[7] These variable are accessed using for example, *mapscript.MS_SUCCESS*

In line 10 we can see that the shapeObj has a member function called *get(idx)* which returns the line in the particular index given as a parameter. The line returned is handled by MapScript's *lineObj*. In line 11, same as above, the lineObj's *get(idx)* function return a reference to a MapScript *pointObj* in the particular index given as a parameter. This structure is not peculiar if we think of the representation of the spatial object in the vector model discussed in section 3.2.1, where the points act as vetrices within the line work. Line 12 prints the point coordinates on the output stream.
As soon as a reference to a *shapeObj* exists, instead of accessing its geometry, the function call *shapeObj.getValue(index)* can be used to access a *non–spatial* value (or attribute) at the index given as a parameter. Mapscript layerObj's function *getItem(int i)* and attribute *numitems* can be used to return a particular *fieldname*[8] and the *number of fields* respectively.

4.5.4 Computations on spatial features

Using mapscript, developers have access to functions available to analyse and compute geographic relationships between geographic features. The mapscript object *ShapeObj* (described above) can be seen as an abstaction object between the different types of geometries being points, lines, polygons, or multipolygons. Thus, the object gives a very useful collection of function which are either native in mapserver, or supported through GDAL or GEOS. Spatial operations include:

contains(pointObj—shapeObj) Returning true, if the shape or point is within the object calling the member function

crosses(shapeObj) Returning true, if the shapes are crossing each other

difference(shapeObj) Returning the difference between the supplied and existing shape

disjoint(shapeObj) Returning true, if the shapes are disjoint

distanceToPoint(pointObj) Returning the distance from a point object in specified map or layer units

distanceToShape(shapeObj) Returning the **minimum** distance from a shape object in specified map or layer units

equals(shapeObj) Returning true if the two shapes are equals in terms of geometry only

[8] With the name *fieldname* we refer to the *name* of a single column containing non–spatial information.

intersects(shapeObj) Returning true if the supplied shape intersects the existing one

intersection(shapeObj) Calculates the spatial intersection between the two shapes

overlaps(shapeObj) Returning true if the supplied shape overlaps the existing one

touches(shapeObj) Returning true if the supplied shape touches the existing one based on the arcs defined by the vertices

Union(shapeObj) Returns the union of the existing and supplied shape.

4.6 OGC web–services within MapServer

Web Services, in general, reflect the advantages of the Web as a mean that provides *services*, not just information. By the term "services" we do not refer to monolithic web applications, but, rather, to component services that can be plugged together to build larger, more comprehensive services and systems. For example, OpenID offers an authentication service exported on the Web and MoinMoin supports it in order to authenticate a user.

Within the broader context of web services, OGC Web Services (OWS) represent an evolutionary, standards–based framework that enables seamless integration of a variety of online geoprocessing and location services. OWS allows distributed geoprocessing systems to communicate with each other using familiar technologies like XML and HTTP. Thus, OWS are self–contained, self–describing, modular services that can be published, located and invoked across the Web. An OGC Web Service can be treated as a "black box" that performs a task, such as providing driving directions. As long as OGC web services can describe the operation they perform in Metadata (Capabilities), it is possible to search the services and understand what a particular web accessible service can perform.

OWS architecure is multi–tier and attention can be drawn to the Information Management Services tier that contains services designed to store and provide access to data, with each server normally handling multiple separate datasets. In addition, metadata describing multiple datasets can be stored and searched. In GeoMoin, we are interested in two types of services included in the OWS specification: *WMS* and *WFS*.

4.6.1 The WMS service overview

WMS is a service that dynamically reproduces spatially referenced maps of client–specific ground rectangles from one or more client–selected geographic datasets, returning predefined pictorial renderigs of maps in an image format. As an example, we will use the WMS service provided by *www.geosignal.org*. First of all, the user or an application can issue a *GetCapabilities* query of the WMS service that will return an XML by issuing the following HTTP GET query string:

```
http://www.geosignal.org/cgi-bin/wmsmap?version=1.1.1& \
                    service=WMS&request=GetCapabilities
```

The result of the previous query is reflected in an XML document containing metadata about the service, and the layers that are accessible through it. Some accessible layer of France look like that:

```
<Layer queryable="0" opaque="0" cascaded="0">
<Name>RASTER1000k</Name>
<Title>Raster France 1/1 000 000</Title>
<SRS>EPSG:27582</SRS>
<ScaleHint min="280.633" max="935.443"/>
</Layer>

<Layer queryable="0" opaque="0" cascaded="0">
<Name>RASTER500k</Name>
<Title>Raster France 1/500 000</Title>
<SRS>EPSG:27582</SRS>
<ScaleHint min="140.316" max="280.633"/>
</Layer>

<Layer queryable="0" opaque="0" cascaded="0">
<Name>RASTER250k</Name>
<Title>Raster France 1/250 000</Title>
<SRS>EPSG:27582</SRS>
<ScaleHint min="65.481" max="140.316"/>
</Layer>
```

Issuing a HTTP GET query string like:

```
http://www.geosignal.org/cgi-bin/wmsmap?
                    SERVICE=WMS&VERSION=1.1.1&
                    REQUEST=GetMap&
                    BBOX=-6.062580,41.163200,10.878300,51.291800&
                    SRS=EPSG:4326&LAYERS=RASTER4000k
```

Results in the map shown in figure 4.5

FIGURE 4.5: Map retrieved using the OWS WMS service.

4.6.2 The WFS service overview

WFS is a service that retrieves spatial features and feature collections stored that meet client–specific selection criteria. WFS returns results in the *GML* format. The Geography Markup Language (GML) is the XML grammar defined by the Open Geospatial

Consortium (OGC) to express geographical features. GML serves as a modeling language for geographic systems as well as an open interchangable format for geographic transactions on the Internet. As an example for the WFS service we will use the WFS provided by the *DMSolutions* group. The corresponding URL containing access to the GetCapabilities request is featured below:

```
http://www2.dmsolutions.ca/cgi-bin/mswfs_gmap?SERVICE=WFS&VERSION=1.1.1&
                                                REQUEST=GetCapabilities
```

The GetCapabilities request results in an XML document describing the service's features and the datasets that it provides:

```
<FeatureType>
<Name>prov_land</Name>
<Title>Canadian Land</Title>
<SRS>EPSG:42304</SRS>
<LatLongBoundingBox minx="-173.537" miny="35.8775"
                    maxx="-11.9603" maxy="83.8009"/>
</FeatureType>

<FeatureType>
<Name>land_fn</Name>
<Title>US Land</Title>
<SRS>EPSG:42304</SRS>
<LatLongBoundingBox minx="-178.838" miny="31.8844"
                    maxx="179.94" maxy="89.8254"/>
</FeatureType>

<FeatureType>
<Name>park</Name>
<Title>Parks</Title>
<SRS>EPSG:42304</SRS>
<LatLongBoundingBox minx="-173.433" miny="41.4271"
                    maxx="-13.3643" maxy="83.7466"/>
</FeatureType>
```

Drawing our attention to the park dataset, we can get the first feature issuing the following HTTP GET query string:

```
http://www2.dmsolutions.ca/cgi-bin/mswfs_gmap?
             SERVICE=WFS&VERSION=1.0.0&
             REQUEST=GetFeature&TYPENAME=park&
             MAXFEATURES=1
```

The above query result in the following GML:

```
<wfs:FeatureCollection>
<gml:boundedBy>
<gml:Box srsName="EPSG:42304">
<gml:coordinates>
-2261310.750000,-67422.421875 2840366.000000,3830124.250000
</gml:coordinates>
</gml:Box>
</gml:boundedBy>
<gml:featureMember>
<myns:park>
<gml:boundedBy>

<gml:Box srsName="EPSG:42304">
<gml:coordinates>
245524.015625,3585946.750000 504494.156250,3830124.250000
</gml:coordinates>
</gml:Box>
</gml:boundedBy>
<myns:msGeometry>
<gml:Polygon srsName="EPSG:42304">
<gml:outerBoundaryIs>
<gml:LinearRing>
<gml:coordinates>
389366.843750,3791519.750000 419768.875000,3775503.000000
503425.843750,3765282.250000 503337.593750,3764874.500000
504494.156250,3756882.500000 504296.625000,3753103.750000
498288.875000,3743100.500000 497375.906250,3740939.250000
.....
</gml:coordinates>
</gml:LinearRing>
</gml:outerBoundaryIs>
</gml:Polygon>
```

```
</myns:msGeometry>
<myns:AREA>38346293248.000</myns:AREA>
<myns:PERIMETER>1357483.000</myns:PERIMETER>
<myns:PARK_>2</myns:PARK_>
<myns:PARK_ID>40</myns:PARK_ID>
<myns:NAME_E>Ellesmere Island National Park Reserve</myns:NAME_E>
<myns:NAME_F>R?serve de parc national de I'?le-d'Ellesmere</myns:NAME_F>
<myns:YEAR_EST>1986</myns:YEAR_EST>
<myns:REG_CODE>61</myns:REG_CODE>
<myns:AREA_KMSQ>39500.000</myns:AREA_KMSQ>
</myns:park>
</gml:featureMember>
</wfs:FeatureCollection>
```

4.6.3 Integration with MapServer

Mapserver is able to take advantages of the architectures provided above either by using online OWS resources as datasets to build maps (client) or providing such services for remote applications (server). Thus MapServer can be parameterized to be used as:

- WMS client
- WMS server
- WFS client
- WFS server

Below, we will describe the steps needed to achieve the above modes, excluding the WFS server.

4.6.3.1 Mapserver as WMS client

In order to use MapServer as a WMS client, the only parameterizing that takes place is to import the online resource as a LAYER in the mapfile used to build a map. WMS layers are accessed via the WMS connection type in the Mapfile. Here is an example of a layer using this connection type:

```
LAYER
  NAME "prov_bound"
```

```
  TYPE RASTER
  STATUS ON
  CONNECTION "http://www2.dmsolutions.ca/cgi-bin/mswms_gmap?"
  CONNECTIONTYPE WMS
  METADATA
    "wms_srs"             "EPSG:42304"
    "wms_name"            "prov_bound"
    "wms_server_version"  "1.1.1"
    "wms_format"          "image/gif"
  END
END
```

We can see that the type of the layer is RASTER as we refer to WMS imagery while the CONNECTIONTYPE is WMS in order that the mapserver business logic understands that the URL provided within the CONNECTION string is used to locate the remote WMS server. The METADATA is the key element to a successful WMS integration as it is the place where the whole configuration is taking place. In the example above, the *wms_srs* is used to declare that the projection used, will match the one presented in the Capabilities document of the WMS server's response. The *wms_name* identifies the layer whose data the imagery will depict. The last two options define the remote WMS server's version and the output format of the image that will be generated by the server. Of course, setting options that are not equivelant to the ones presented in the server's Capabilities document will result in mapserver exception, thus a fault in the application. That is, the building of mapfile layers using WMS must be carefully created, examining and learning what a remote WMS service exactly has to offer.

4.6.3.2 Mapserver as WMS client

Parameterizing mapserver to act as a WMS server requires more than adding directives to the mapfile. In case of MapScript, a proper script must be create which will be part of the URL that remote clients connect to in order to request the imagery. Beginning with the mapfile requirements, the mapfile will be parameterized to act a "server" mapfile. This is accomplished using the WEB object of the mapfile:

```
WEB
  ...
  METADATA
    "wms_title"           "WMS Demo Server"
    "wms_onlineresource"  "http://my.host.com/mapserv.py?"
```

```
    "wms_srs"              "EPSG:4269 EPSG:4326"
  END
END
```

The first option declares the name of the particular WMS server instance, the second is the URL that will be presented to client in order to know how to access the imagery. The last option is the projections that are supported by the server, that is, only EPSG of 4269 and 4326 are allowed.
Finally, within every layer contained in the mapfile, a directive "DUMP TRUE" must be added in order to inform the server that the layer is subject to remote WMS requests. Additionally, the METADATA of every layer can contain options that declare the name and the description of each layer presented.
The last part of the WMS configuration is to create the script that will be used by the remote WMS client to request the imagery:

```
01    map =  mapscript.mapObj("WMStest.map")
02    req = mapscript.OWSRequest()
03    mapscript.msIO_installStdoutToBuffer()
04    req.loadParams()
05    map.OWSDispatch( req )
06    content_type = mapscript.msIO_stripStdoutBufferContentType()
07    content = mapscript.msIO_getStdoutBufferBytes()

08    if content_type == 'application/vnd.ogc.se_xml':
09        content_type = 'text/xml'
10    print 'Content-type: ' + content_type
11    print
12    print content
```

In line 01, the map is accessed and a map object is instantiated. Line O2 forms an object that will hold and parse the HTTP request options passed to the server script. This is done by issuing a loadParams() execution, presented in line 04. Line 05 involves the actual processing of the request within the context of the mapfile internally, while the content type as a MIME type and the content itself as a binary representation of the image can be fetched using line 06 to 07 respectively. Finally, a proper document must be returned to the client containing the MIME type and the binary so that the client can use the imagery provided.

4.6.3.3 Mapserver as WFS client

MapServer can retrieve and display data from a WFS server. The following document explains how to display data from a WFS server using MapServer. A WFS layer is a regular mapfile layer, which can use CLASS objects, with expressions, etc.
As of MapServer 4.4, the suggested method to define a WFS Client layer is through the CONNECTION parameter and the layers METADATA. The necessary mapfile parameters are defined below:

CONNECTIONTYPE Must be "wfs"

CONNECTION The URL to the WFS Server. e.g. http://www2.dmsolutions.ca/cgi-bin/mswfs_gmap?

METADATA The LAYERs must contain a METADATA object with the following parameters:

wfs_typename The name of the layer found in the GetCapabilities.

wfs_version WFS version, currently 1.0.0

wfs_maxfeatures (opt.) Limits the number of GML features to return.

wfs_latlongboundingbox The bounding box of this layer in geographic coordinates in the format "lon_min lat_min lon_max lat_max". If it is set then MapServer will request the layer only when the map view overlaps that bounding box.

wfs_filter This can be included to include a filter encoding parameter in the getFeature request. The content of the wfs_filter is a valid filter encoding element. For example a filter like:

```
METADATA
  "wfs_filter"   "<PropertyIsGreaterThan>
                  <PropertyName>POP_RANGE</PropertyName>
                  <Literal>4</Literal>
                  </PropertyIsGreaterThan>"
END
```

will provided GML features where the population range is greated than 4.

Thus, a properly described WFS layer can be te following:

```
LAYER
  NAME "park"
```

```
  TYPE POLYGON
  STATUS ON
  CONNECTIONTYPE WFS
  CONNECTION "http://www2.dmsolutions.ca/cgi-bin/mswfs_gmap?"
  METADATA
    "wfs_typename"          "park"
    "wfs_version"           "1.0.0"
    "wfs_request_method"    "GET"
    "wfs_connectiontimeout" "60"
    "wfs_maxfeatures"       "1"
  END
  PROJECTION
    "init=epsg:42304"
  END
  CLASS
    NAME "Parks"
    STYLE
      COLOR 200 255 0
      OUTLINECOLOR 120 120 120
    END
  END
END # Layer
```

Chapter 5

The GeoMoin Web Application

5.1 Overview

GeoMoin is an open–source web–gis application mainly developed in order to assist GIS communities to provide their spatial information to the public. In those terms, it is being developed at the Technical University of Crete (TUC), issued under General Public Licence (GPL).
GeoMoin, in its essense, is a wiki that can handle geo–spatial information. In order to explain the need of a merging between the wiki and GIS technology we can provide the well–known Wikipedia as an example. Wikipedia is an online encyclopedia, based on the MediaWiki, where its contributors are allowed to post, update, comment or delete articles which consitute the pages of the wiki system. That is, Wikipedia is a collaboration between many users of the Web providing a centralized point that serves information. Therefore, GeoMoin itself, was created in order to become such a collaboration, where users (or communities) can upload, render, manipulate **geospatial information**.
While GeoMoin is an on–going developing effort we can list some situations where it can be proved useful:

- A community that collects spatial information in regard to a particular region of the earth's surface may need to provide its datasets to the public so that they are easily accessible and used.
- The same community needs to render these spatial information and provide a graphical representation to the web, allowing **user interaction**.
- A web user wants to find ESRI datasets containing elevation information for a particular region. It is easier to search a centralized location specializing in spatial information, than digging through the web.

- The fact that the application is being developed with user–interaction and interoperability in mind, allows GeoMoin to serve as an educational tool.
- A restaurant chain wants to provide its costumers with an interactive map showing its restaurants in a particular region. It can eliminate the web–design cost of adding spatial content to its webpages by using a map of that region and annotate the points of interest. Each annotation of course can link to a particular page in the company's web–site.
- A GIS programmer wants to use the GPS technology to geotag locations and offer the results for online navigation. GeoMoin can be used as a framework for the development of an application like that.

It is important to state that MapServer is not a full–featured GIS system in terms of *spatial analysis* and *data processing*, nor does it aspire to be. Desktop GIS applications such as ArcGIS and GRASS excel in providing highly specialized tools for such interactions with spatial information. GeoMoin serves as a mean of providing the result of these computational and analytics operations to the web, in an interactive and user–friendly manner.
Of course, viewed as a web developing framework, GeoMoin can be enhanced with features that can be found in a desktop GIS application as it gives the programmer access to libraries like GDAL, but its nature as a web application does not allow the cost–effectiveness of such interactions, with huge datasets, in terms of network bandwidth, server CPU and memory requirements.

5.2 Design and implementation in overview

[22]WebGIS applications can be classified as *fat-client* and *thin-client*. In fat-client systems, a significant proportion (often, the bulk) of data processing happens at the client, whereas the server is primarily responsible for data storage (e.g. Tsou, 2004). By contrast, thin-client systems strive to minimize processing on the client; except for presentation and user interaction, data processing occurs at the server. For GeoMoin, the thin-client approach was adopted, based on the following criteria:

- The system must be accessible on the internet, where users may have in–sufficient network resources to download and process massive data locally. Only visualization data should be transmitted to the client (rendered graphics, query results e.t.c.).

- The system must be accessible from different platforms and environments, such as UNIX and Windows operating systems, as well as all popular web browsers, without dependence on additional software (specialized plug-ins, local applications, etc.) and with almost–zero configuration. An exception to this rule applies to the rendering part which is delivered through Asynchronous Javascript and XML (AJAX), as long as web clients that do not support this architecture still exist and are being used.

- GeoMoin should be portable across a broad range of server platforms and technologies, with the lowest possible dependence on proprietary software, in order to allow for deployment to servers with maximum versatility and minimum cost.

- GeoMoin servers must be scalable, both with respect to the number of concurrent users and in terms of storage capacity and computational performance. At the same time, server administration should be simple, and require only modest technical expertise.

The design that is outlined in this section demonstrates that, by using state-of-the-art web programming facilities, combined with a server that integrates a broad range of open-source tools and applications, it is possible to achieve all of the above goals with currently available WebGIS technology.
Of course without organized stressing of the web–application it is not clear that the above criteria can be satisfied without serious compromises to application functionality and user-interface quality.

5.2.1 Application architecture

GeoMoin is architected as a 3-tier system, as shown in figure 5.1, thus the whole functionality is distributed in 3 levels of abstraction. The *user services*, the *business services* and the *data services*.

The ***user services tier*** provides the graphical user interface through which the user interacts with the web–application. The technology used here is Dynamic HTML (DHTML) and AJAX (Asynchronous Javascript And XML). AJAX (which is discussed in more detail later) is a state-of-the-art approach to web application design, which provides superior user-interface functionality and interactivity compared to traditional (i.e. form-based) web applications, based solely on standard web browser features. By adopting AJAX, a portable, friendly, functional, highly-interactive user interface, without resorting to proprietary applet technologies (including Java applets) were provided.

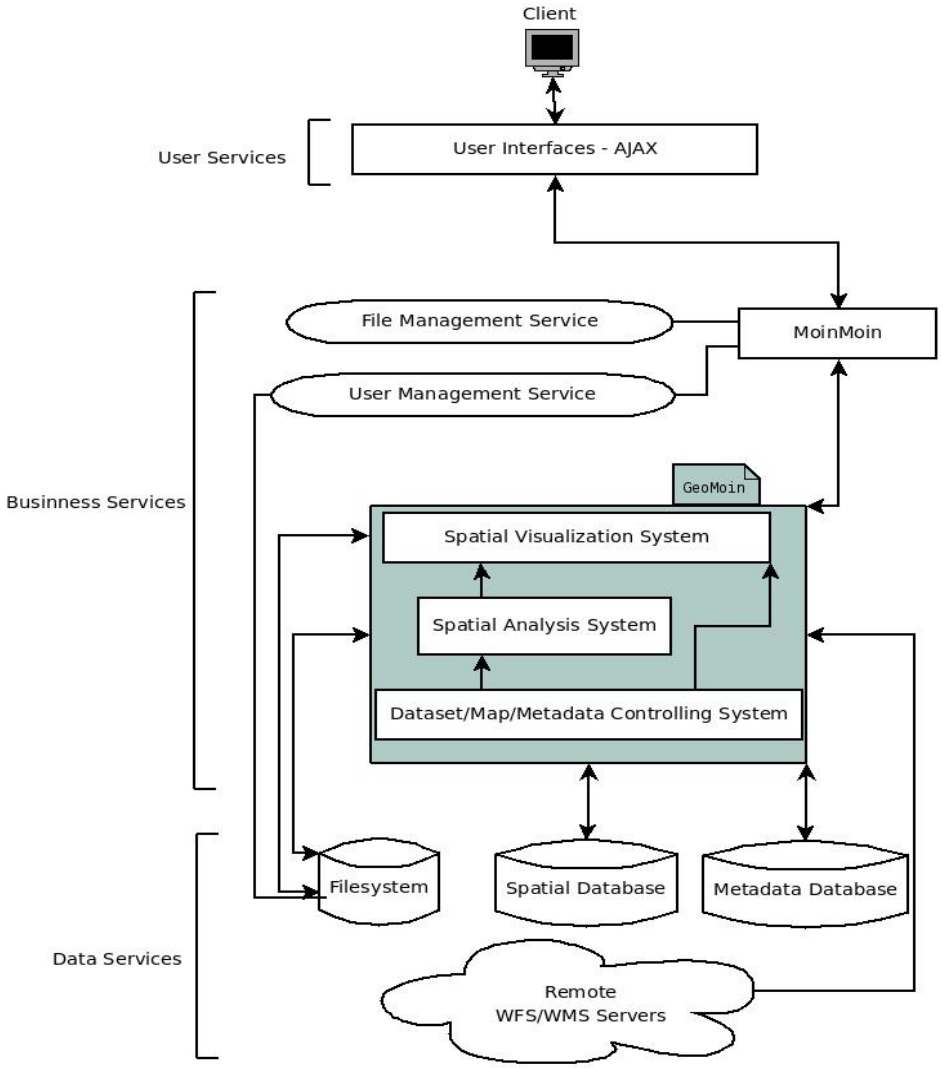


FIGURE 5.1: The 3-Tier architecture of GeoMoin

MoinMoin, viewed as a framework for building web applications in the Python programming language, provides a convenient and high–level method for generating server side HTML and can match the capabilities of developing a WebGIS using PHP or JSP.
It is very important to highlight that GeoMoin, while considered a web–application, it nests into a higher–level web–application – the MoinMoin wiki – whose interactions between the user and the server are based on the traditional HTML form methodology using submit actions that refresh the web page. On the contrary, using the AJAX technology, submitting user information is achieved in a partial, asynchronous manner, meaning that the page will submit only the required information while the page *will not be refreshed*. It is clear, that it becomes very important for the application to be carefully built in a manner that will not compromise user–navigation and web–design practices while confusing the user. Thus, the following guideline is being adopted:

The AJAX functionally is being used only in occasions of demanding computational requirements such as the rendering of a map, in order to avoid multiple reinvocations of the same operations. Additionally AJAX interactions are clearly stated to the end user.

In other cases of low computational requirements, such as the generation of a mapfile, traditional web–design is adopted.

The ***business services tier*** encapsulates the business logic for the entire application. The user and session management system is embedded in the MoinMoin wiki providing user registration, authentication using *local accounts* or *OpenID*[1], and personalization functionality. In addition the pages of the wiki which are stored in the filesystem are accessible through MoinMoin's file management and indexing service. Focusing solely on the GeoMoin application, we can describe three subsystems:

- The Spatial Visualization System, which is based on MapServer and GDAL, is responsible for image representation of geospatial data, the management of annotations and the querying of the spatial data amongst other capabilities. The spatial visualization system will be described in detail later in the chapter.
- The Dataset/Map/Metadata Controlling System is reflected to the end user through two main applications: the LayerManager and the MapManager. The LayerManager is responsible for the management of the layers that contain geo–spatial information either the reside in the filesystem or in PostGIS. The MapManager accesses the filesystem to retrieve, parse, create or update the users' mapfiles. The metadata are information stored in PostGIS which have non–spatial nature and are responsible for identifying and characterizing datasets in order to provide them through search queries. The Dataset/Map/Metadata Controlling System will be described in detail later in the chapter.
- The Spatial Analysis System implements the spatial decision support capabilities in the form of a number of spatial analysis services, such as spatial query processing over geographic databases, and is available through the use of GeoMoin **mapplets**. An effort is being made to provide a common API in order to allow the indepented development and contribution of the mapplets which can be considered as add–ons to the GeoMoin capabilities.

The ***data services tier*** provides services that store, retrieve and update information through a simple data model. The main storage mechanisms are (a) the file system

[1]OpenID eliminates the need for multiple usernames across different websites, providing a centraliced secure authentication service. Location: openid.net

and (b) the PostgreSQL database management system (DBMS) through the use of the PostGIS spatial extension. The file system is used for storing spatial (vector or raster) datasets and mapfiles in a variety of formats, managing per-user directories. The spatial database has a dual role; it stores metadata related to datasets stored in the file system, and it can be used to manage (store, update, search, process) vectorized[2] spatial content for application-specific purposes. Finally, as Data Services in GeoMoin application, the OGC's open web–services (OWS) are included and specifically the web mapping service (WMS) and web feature service (WFS). Details concerning the Data Services tier will be available in the following sections.

5.2.2 Overview of system components

Currently GeoMoin is mainly based on the following open–source software packages:

Apache2 The web–server hosting the system.

MoinMoin The wiki system providing the web–site desing and a framework for development.

UMN Mapserver Mapserver's mapscript is behind use cases such as the rendering and the querying of geospatial information.

GDAL library The library that is used to interact with the geospatial information.

PostgreSQL The database management system (DBMS) used in GeoMoin.

PostGIS An extension for PostgreSQL DBMS that provides the support for spatial data types. Moreover, it provides spatial analysis functions to the database.

In addition to the main packages, the following ones should be considered:

pgdb This package acts as a driver in order to communicate with the DBMS. In addition, it provides a complete level of abstraction between the different DBMSs.

Geopy An on–going python library that handles the connection to various online GeoCoder such as GoogleMaps, Yahoo Maps, Virtual Earth etc. Geocoders can be viewed as databases of mappings between real–world locations (eg. roads,states) and their coordinates.

[2] An attempt is being made to include raster datasets in PostGIS through the PGCHIP library which will in the Appendix.

OverlibMWS The DHTML javascript library specialized in the generation of popups. Based on Erik Bosrup's overLIB, this is an enhanced version attributed to Foteos Macrides and is available through the Artistic License, Version 2.0 which is compatible with GPL.

OWSLib OWSLib is a Python package for working with OGC web map (WMS) and feature services (WFS). It provides a common API for accessing service metadata and wrappers for GetCapabilities, GetMap, and GetFeature requests. It is initially developed by Sean Hussein Gillies and is available under GPL.

Cheetah Cheetah is a template management system that is extensively used for the generation of mapserver mapfiles within the application.

The above packages are integrated using Python, a mature and highly portable scripting language. Although MapServer and GDAL in natively written in C++, the existing SWIG bindings are readily available and can be easily integrated. Please note that the current choice of third–party packages is not binding, for example we could rely on different web–server hosting the application or in different wiki modes (eg FastCGI) instead of the CGI default. In addition, there exist spatial–extensions for every popular DBMSs including MySQL and Oracle.

When it comes to describing a system like GeoMoin a down–top approach needs to be adopted, because the lower tier level of the architecture contains indepented components and associations between the upper tiers that can be confusing if they are introduced from the top.

In general, with the term Data Services we refer to whole subsystem that provides the stored information to the higher level mechanism. Those mechanisms do not only include the GeoMoin application, but additionally the MoinMoin wiki itself.
As it was described above, three sources of information can be identified: the **filesystem**, the **DBMS** and the remote **OWS services**.

5.3 Filesystem data services: Requirements and Implementation

The filesystem as source of information contains four important elements, the wiki pages, the user information, the spatial data and the mapfiles. The wiki pages are stored in the filesystem under specific directories accessible by MoinMoin and each page is a file that contains the wiki markup language. In MoinMoin these directories' contents are

indexed using Xapian[3] in order to be efficiently accessible. Details of the page storage subsystem is out of the scope of this document. The user information are stored in the filesystem too under a special directory called "user". They contain all the information for a user account including its password while MoinMoin takes care of the security of access. Different wiki implementations store the user information in DBMS's while MoinMoin is leaning towards this direction in the future. It is not important to dive into the details of the user management as it is handled successfully by MoinMoin.

5.3.1 Spatial data in the Filesystem and retrieval requiremenents

The spatial data, managed by the GeoMoin application, are stored in a special directory named by each user's unique ID. The parent directory can be easily changed by editing the GeoMoin's configuration files.
Spatial datasets that can be installed in GeoMoin include:

Vector Data

- ESRI Shapefile
- Mapinfo mitab
- Geographic Markup Language (GML)

Raster Data

- GeoTiff
- Portable Network Graphics (PNG)
- JPEG
- BMP

This list can be easily enhanced to include every different spatial dataset distribution as long as the respective module (described later) is updated to include them.
The main problem an application that manages spatial datasets has to overcome is the

[3]Xapian is a highly adaptable toolkit which allows developers to easily add advanced indexing and search facilities to their own applications. It supports the Probabilistic Information Retrieval model and also supports a rich set of boolean query operators. Xapian is an Open Source Search Engine Library, released under the GPL. It's written in C++, with bindings to allow use from Perl, Python, PHP, Java, Tcl, C# and Ruby.

lack of description embedded in the various formats. Different vendors apply different methods of adding metadata in their datasets, while some formats do not contain metadata at all. There are two approaches to this problem:

- Feature extraction
- Human annotation

As far as feature extraction is concerned, this approach is still vendor depended. GDAL library ships with a utility called *ogrinfo* which accesses and displays information over every type of dataset that it can support. A sample output of this utility can be:

```
root@jack-daniels:/var/www/GeoMoin/mapdata/1214071817.69.30350# \
ogrinfo countries.shp countries -al -so
INFO: Open of ‘countries.shp’
      using driver ‘ESRI Shapefile’ successful.

Layer name: countries
Geometry: Polygon
Feature Count: 609
Extent: (-180.000000, -90.000000) - (180.000000, 83.623596)
Layer SRS WKT:
(unknown)
COUNTRY: String (254.0)
STATE: String (254.0)
REGION: String (254.0)
CONTINENT: String (254.0)
```

The fact is that when it comes to retrieval of information, we are not solely concerned about the technical spatial metadata. Every dataset must be additionally characterized by a full description, a date of installation, the user that provided it, etc. These kinds of metadata have to do with human annotation, and as soon as each spatial dataset has a specific role of existance as a multimedia type, those metadata should be not only accurate, but as complete as possible. In GeoMoin, the following methodology is used: Specific types of metadata that describe the installed layers are solely stored in the GeoMoin database while full description and usage examples can be optionally included in an independent wikipage. Moreover, metadata for this layer can include the link to the page.
Thus, the retrieval methods, which will be described later on this chapter, communicate

with the DBMS in order to complete a retrieval query.

5.3.2 Mapfiles storage and retrieval

In chapter 4 we explained that a mapfile is a script that given as an input to MapServer cgi or mapscript, it contains all the directives which are needed in order to render and query the spatial datasets that are assigned to it. These mapfiles in GeoMoin are created by its users and are stored by default under a specific directory in the filesystem. This directory can be easily changed by editing the GeoMoin configuration. As long as the mapfile is a single–file configuration mean there is no need for a per–userid directory, on the contrary, they are stored in a filename consisting of the userid and the specific name that the user selected for the map (eg. 1214071817.69.30350_world.map). The userid part is crucial in order to prohibit two different users create a map under the same name, although it prohibits a user create a map with the same name twice.
The creation of a mapfile is based on a templating mechanism created for Python which is called **Cheetah**. Cheetah is a very powerful open source template engine and code generation tool but it is out of the scope of this document to introduce the reader to its capabilities[4]. Altough the filling of the directives in a mapfile is a matter of a higher–level subsystem – called MapManager –/, the reason that Cheetah templating is described here is because it gives the ability to **hide** specific directory information from the end–user, as long as security is concerned. There exist mapfile directives that require the complete path of a directory in which an action is required to be performed. For example when a layer is defined, its DATA directive contains the complete path to the dataset, in case of filesystem access. One more important security issue is the one that concerns the access to the DBMS in order to populate a layer from there. The CONNECTION string could be:

```
CONNECTION "host=zeppelin dbname=gisdb user=gis-user
                              password=431ro1489 port=5433"
```

Thus templating variables are used and Cheetah, in case of mapfile usage, substitutes them with the values contained in the config script. Moreover, this approach allows for quick migration of the directories or the database credentials, avoiding any batch editing in the mapfile repository. A Cheetah'ed CONNECTION string could be:

```
CONNECTION "host=$db_host dbname=$db_name user=$db_user
```

[4]For a high-level introduction to Cheetah please refer to the User's Guide at http://cheetahtemplate.org/learn.html

```
                        password=$db_pass port=$db_port"
    DATA "wkb_geometry FROM ${db_schema}citiesx020"
```

Retrieving the mapfiles is achieved using the *os* python library and as long as the access controls of the mapfile repository are correctly set, no security leaks should exist. Moreover, the business logic incorporated disallows users to manage mapfiles owned by others.

5.3.3 Markfiles and their usage

The HTTP is a state–less protocol meaning that the server doesn't keep information on previous HTTP requests. Thus, unlike desktop development, information which is generated by the web–applications and reside in the server memory segments must be stored somewhere else to achieve consistency. One method of storing information is to use the session memory space which is unique for every user connected and authenticated to the server. MoinMoin provides the session object as a python dictionary which can be updated with new values. The pitfall of that approach is that upon client logout the session object is destroyed and information are lost and moreover two identical operations may share the same session field between them, something that leads to obvious inconsistencies.
A web–application like GeoMoin may possibly create loads of information that need to be stored safely to maintain state. There are two possible approaches:

- Store information in a DBMS
- Store information in the filesystem

In GeoMoin the second approach was chosen; when a sub–system generates information that needs to be stored, it makes use of Python's Pickle module. Pickle is an approach of storing and loading serializable python data types in simple text files using the methods pickle() and unpickle() respectively. For example a python dictionary containing serializable elements can be stored quickly and efficiently. GeoMoin's pickled information are called ***markfiles***. Markfiles are stored under a special directory named ***usermarks*** and, specifically, the files created include the user id thus overwrite problems are resolved. For example, information about the layers that a user selected to create a mapfile are stored in a unique markfile.

5.4 DBMS data services: Installation and Implementation

5.4.1 DBMS architecture in GeoMoin

GeoMoin is developed in order to be highly depended on a DBMS system. Currently the DBMS serving GeoMoin is PostgreSQL. DBMS is used for managing the following:

- The metadata that are linked to the datasets installed in the system.
- The vector datasets that are installed in the DBMS itself using PostGIS.
- The support for creating, inserting, updating and deleting spatial annotations and annotation tables.
- Information about the OGC Web Services that are identified.

Figure 5.2 shows the interaction between the elements of GeoMoin and the DBMS.

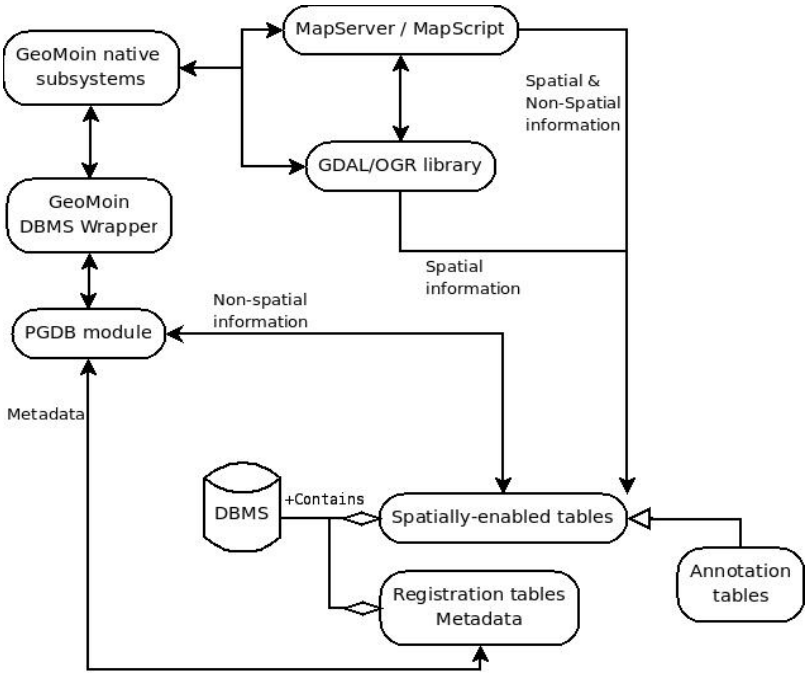


FIGURE 5.2: Graphical view of the DBMS interaction in GeoMoin

5.4.2 PostGIS extension for PostgreSQL

[23]PostGIS is developed by Refractions Research Inc, as a spatial database technology research project. Refractions is a GIS and database consulting company in Victoria,British Columbia, Canada, specializing in data integration and custom software development. Refractions Research Inc plans on supporting and developing PostGIS to support a range of important GIS functionality, including full OpenGIS support, advanced topological constructs (coverages, surfaces, networks), desktop user interface tools for viewing and editing GIS data, and web-based access tools.
The GIS objects supported by PostGIS are a superset of the "Simple Features" defined by the OpenGIS Consortium (OGC). As of version 1.1 [24], PostGIS supports all the objects and functions specified in the OGC "Simple Features for SQL" specification.
The purpose of the above specification is to define a standard SQL schema that supports storage, retrieval, query and update of simple geospatial feature collections via the ODBC API. A simple feature is defined by the OpenGIS Abstract specification to have both spatial and non-spatial attributes. Spatial attributes are geometry valued, and simple features are based on **2D geometry** with linear interpolation between vertices.

Simple geospatial feature collections will conceptually be stored as tables with geometry valued columns in a Relational DBMS (RDBMS), each feature will be stored as a row in a table. The non-spatial attributes of features will be mapped onto columns whose types are drawn from the set of standard ODBC/SQL92 data types. The spatial attributes of features will be mapped onto columns whose SQL data types are based on the underlying concept of additional geometric data types for SQL. A table whose rows represent Open GIS features shall be referred to as a feature table. Such a table shall contain one or more geometry valued columns. Feature table implementations are described for two target SQL environments: SQL92 and SQL92 with Geometry Types. In case of PostGIS, SQL92 with Geometry Types is adopted.
The term SQL92 with Geometry Types is used to refer to a SQL92 environment that has been extended with a set of Geometry Types. In this environment a geometry-valued column is implemented as a column whose SQL type is drawn from the set of Geometry Types. "Simple Features for SQL" specification describes a standard set of SQL Geometry Types based on the OpenGIS Geometry Model, together with the SQL functions on those types. This specification does not attempt to standardize any part of the mechanism by which the Geometry Types are added to and maintained in the SQL environment: The standard SQL3 mechanism for extending the type system of a SQL database is through the definition of user defined Abstract Data Types. In figure 5.3 the class hierarchy for the Geometry Types can be seen.

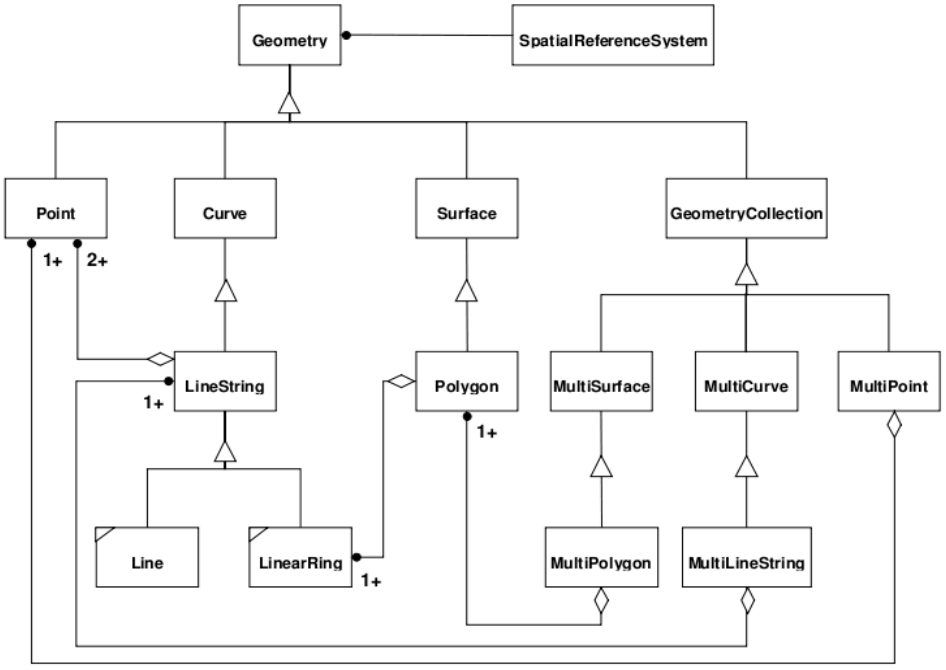


FIGURE 5.3: Class hierarchy of the geometry types.

The OpenGIS specification defines two standard ways of expressing spatial objects: the Well-Known Text (WKT) form and the Well-Known Binary (WKB) form. The Well-known Text Representation of Spatial Reference Systems provides a standard textual representation for spatial reference system information. The Well-known Binary Representation for Geometry (WKBGeometry), provides a portable representation of a Geometry value as a contiguous stream of bytes. It permits Geometry values to be exchanged between an ODBC client and an SQL database in binary form. Both WKT and WKB include information about the type of the object and the coordinates which form the object.
Examples of the text representations (WKT) of the spatial objects of the features are as follows:

- POINT(0 0)
- LINESTRING(0 0,1 1,1 2)
- MULTIPOINT(0 0,1 2)
- MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))
- MULTIPOLYGON(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1-1)))

- GEOMETRYCOLLECTION(POINT(2 3),LINESTRING((2 3,3 4)))

The OpenGIS specification also requires that the internal storage format of spatial objects include a spatial referencing system identifier (SRID). The SRID is required when creating spatial objects for insertion into the database.
Input/Output of these formats are available using the following interfaces:

bytea WKB = asBinary(geometry);
text WKT = asText(geometry);
geometry = GeomFromWKB(bytea WKB, SRID);
geometry = GeometryFromText(text WKT, SRID);

For example, a valid insert statement to create and insert an OGC spatial object would be:

```
INSERT INTO geotable ( the_geom, the_name )
          VALUES ( GeomFromText(POINT(-126.4 45.32), 312), A Place);
```

As it is obvious, the fact that PostGIS adopts SQL92 with Geometry Types allows for creation of a standard SQL table and adding a geometry column like:

```
CREATE TABLE parks (
   park_id   INTEGER,
   park_name VARCHAR,
   park_date DATE,
   park_type VARCHAR
);
SELECT AddGeometryColumn(parks, park_geom, 128, MULTIPOLYGON, 2 );
```

The sql function AddGeometryColumn of the Simple Features for SQL specification is parameterized with the table that will be spatially enabled, the name of the geometry column that will be created, the SRID of the spatial reference system used and the number of dimensions.

5.4.3 PygreSQL PGDB module for DBMS access

Continuing the description of the components presented in figure 5.2, PygreSQL's PGDB module is a implementation of the DB-API 2.0 specification targeting on PostgreSQL. DB-API 2.0 defines standard methods of accessing and managing the DBMS. The usage

of this module is very simple and can be described with an example.
Before any sql query is issued a pgdbCnx object must be instantiated defining the connection to the DBMS. This is done by executing the following function:

```
pgdbCnx = pgdb.connect(user='gisuser',password='gispass',\
                host='localhost',database='gisdb')
```

Upon connection a cursor must be instantiated that will handle the SQL queries:

```
cursor = pgdbCnx.cursor()
```

After defining the cursor an sql query can be issued and the results can be captured. Finally the cursor is closed and the connection is terminated:

```
cur.execute ("SELECT road_id, AsText(road_geom) \
                AS geom, road_name FROM roads;")
row = cur.fetchall()
cur.close()
pgdbCnx.close()
```

In conjuction with PostGIS the above query will return the unique id and the text representation of the geometry of a spatially enabled table called roads.
In case of an SQL insert, create, delete or update query a commit command must be issued:

```
pgdbCnx.commit()
```

It is important to note that many PostGIS functions alter geometry information while returning results and are issued using SQL SELECT statements. That fact that information is altered within the DBMS, requires a commit() even while issuing a SELECT statement.

5.4.4 GeoMoin database schema specification

As we described earlier, in GeoMoin the need for a DBMS is targeting on providing management for the following:

- The metadata that are linked to the datasets installed in the system.

- The vector datasets that are installed in the DBMS itself using PostGIS
- The support for creating, inserting, updating and deleting spatial annotations and annotation tables
- Information about the OGC Web Services that are identified.

Thus the following needs for the very basic database schema are specified[5]:

1. A table must exist under the default name "filesystem" which stores information about the datasets that are installed in the filesystem. It must consist of the following fields:

   id The sequential primary key, unique for each dataset.

   name The name of the dataset.

   type The format of the dataset e.g. ESRI for ESRI shapefiles, GML for Geography markup language etc.

   path The path where the dataset can be found in the filesystem, include the user's unique id as part of the path.

   owner The unique user id of the registered MoinMoin user that installed the dataset.

   description A short description of the dataset provided by its owner.

   wikipage This field acts as a link to a wiki page that fully describes the contents and the usage for the dataset.

   creation_date The date in which the dataset was installed to the system.

   All the fields in the above listing act as metadata for dataset installed in the filesystem.

2. A table must exist under the default name "postgis" which stores information about the datasets that are installed in PostGIS. In addition, this table stores information about the annotation tables that are created and managed by the business–tier of the GeoMoin architecture. The PostGIS tables referenced by the tuples of this table are, of course, spatially–enabled containing one or more geometry columns.
   The table must consist of the following fields:

   id The sequential primary key, unique for each dataset.

[5]With the term "very basic", we refer to features that are important in order that the higher levels of the architecture can operate. Additional features can be added in future versions of the software.

name The name of the dataset.

owner The unique user id of the registered MoinMoin user that installed the dataset.

viewable A boolean value that flags if the current dataset can be viewable by other wiki users in addition to the owner.

editable A boolean value that flags if the current dataset can be editable by other wiki users in addition to the owner in terms of updating or inserting tuples.

type The type of the dataset can either be "ANNOTATION" describing a spatially–enabled table that can act as an annotation layer or "POSTGIS" describing a spatially–enabled table that was created by a third–party group and is provided as–is.

directlink Used only if the table is an annotation; it flags whether the annotation table will be rendered as a group of links point to predefined webpages. If false, it marks the display of a popup containing the annotation information.

description A short description of the dataset provided by its owner.

wikipage This field acts as a link to a wiki page that fully describes the contents and the usage for the dataset.

creation_date The date in which the dataset was installed to the system.

3. A table must exist under the default name "wms" which stores information about the wms servers registered in the system.

   id The sequential primary key, unique for each wms server.

   wmsname A name for the server.

   url The url pointing to the cgi on the remote server

   immut A boolean value that flags whether the dataset can be deleted or edited.

   description A description for the server. For example, if it is currently down.

   owner The unique id of the user that added the server.

4. A table must exist under the default name "wfs" which stores information about the wfs servers registered in the system.

   id The sequential primary key, unique for each wfw server.

   wmsname A name for the server.

   url The url pointing to the cgi on the remote server

   immut A boolean value that flags whether the dataset can be deleted or edited.

   description A description for the server. For example, if it is currently down.

owner The unique id of the user that added the server.

5. For every spatially–enabled table, a Generalized Search Tree (GiST) must be created to index the main geometry column that is contains. Information about GiST indexing and PostGIS performance can be found in the appendix.

6. For every spatially–enabled table that **is not** an annotation table there must exist the ability to cluster the geometry indices to achieve reordering of all the data rows in the same order as the index criteria yielding performance advantages. Excluding annotation tables, we are left with the **read–only** tables produced by installing spatial datasets in PostgreSQL. It is important to highlight that clustering tables that contain features with NULL geometry cannot happen. Eventually, indexing using GiST tables with NULL geometries is allowed. Thus, it is recommended that a mechanism should exist that checks whether a dataset can be installed in PostgreSQL setting a NOT NULL constraint in its geometry field, allowing clustering. Otherwise, clustering can be ommited.

7. An annotation table in GeoMoin is a spatially–enabled PostGIS dataset that can be created by any authenticated user, where every authenticated user can add to or edit the spatial information contained to it. By the time this work is written an annotation table consists of only one geometry column of the geometry type: **point** or **polygon**. Details on how these spatial features are obtained, managed and inserted can be found in the later sections. An annotation table must consist of the following fields:

   id A sequential primary key, unique for every spatial feature contained.

   owner The unique user id of the registered MoinMoin user that added this feature.

   editable A boolean value that flags whether the current feature can be editable by other wiki users in addition to the owner.

   directlink A boolean value that flags whether the feature annotation will be handled as a direct link to a web–page by the business logic.

   geometry column A geometry column presented in Well–Known–Binary (WKB) format that defines the spatial attributes of the feature

   1..* fieldnames One or more fieldnames defined by the creator of the annotation table, that act as the non–spatial information of the feature. Types supported up till now: **integer**, **float**, **text**, **date**.

   Each unique [id] of an annotation tuple can act as a foreign key to another spatial or non–spatial table in the RDBMS. Thus with the development of the essential business logic the relational aspects of the database can be utilised to achieved

more complex integration between spatial and non–spatial features. One simple example of such integration could be the developement of a comment board for each annotation on a map.

8. The DBMS's encoding must be UTF–8

5.4.4.1 Implementing the schema on PostgreSQL

In this section, the above requirements will be converted in sql and the required database and schema will be created using a unix shell and the console–based *psql* utility that ships with PostgreSQL.

Requirements:

- PostgreSQL must be installed
- The PostGIS extensions that must be available in their sql format are: *lwpostgis.sql* and *spatial_ref_sys.sql*.
- The default user "postgres" must exist in order to manage the DBMS.

In the unix shell the *su* command must be issued to change the active user to postgres

```
su postgres
```

The following commands will create a template database and define the procedural language format:

```
createdb postgistemplate
createlang plpgsql postgistemplate
```

The PostGIS extension sql scripts must now be installed in the template database using the psql utility:

```
psql -d postgistemplate -f lwpostgis.sql
psql -d postgistemplate -f spatial_ref_sys.sql
```

Upon successful installation of the sql script, the spatial extension to PostgreSQL is now available. Now the roles and access controls to the template must be defined consisting of a group and a user to whom permissions are assigned:

```
psql
CREATE ROLE gisgroup NOSUPERUSER NOINHERIT CREATEDB NOCREATEROLE
CREATE ROLE gis LOGIN PASSWORD 'password' NOINHERIT;
GRANT gisgroup TO gis;
\q
```

Before creating the actual database the geometry_columns and spatial_ref_sys tables which were created using PostGIS must be owned by the gis role created above:

```
psql -d postgistemplate
ALTER TABLE geometry_columns OWNER TO gis;
ALTER TABLE spatial_ref_sys OWNER TO gis;
```

Still in the psql command–line terminal, now the database will be created under a new PostgreSQL schema:

```
CREATE SCHEMA gis_schema AUTHORIZATION gis;
# quit to terminal
\q
createdb -T postgistemplate -O gis gisdb
```

Now we can either use psql or the pgAdmin III[6] to review the changes that happened to the PostgreSQL.
Finally the tables that are defined under the [1], [2], [3] and [4] requirements of the above specification must be created:

```
psql -d gisdb

CREATE TABLE gis_schema.postgis (
  id serial NOT NULL,
  name text NOT NULL,
  owner text NOT NULL,
  viewable boolean NOT NULL DEFAULT true,
  editable boolean NOT NULL DEFAULT true,
  type text NOT NULL,
directlink boolean NOT NULL DEFAULT false,
  description text DEFAULT 'Not Defined'::character varying,
  wikipage text DEFAULT 'Not Defined'::character varying,
```

[6]PgAdmin III is a GUI tool that can be used to manage a PostgreSQL DBMS.

```
      creation_date date NOT NULL,
      CONSTRAINT postgis_pkey PRIMARY KEY (id)
    ) WITH (OIDS=FALSE);
    ALTER TABLE gis_schema.postgis OWNER TO gis;

    CREATE TABLE gis_schema.filesystem (
      id serial NOT NULL,
      name text NOT NULL,
      type text NOT NULL,
      path text NOT NULL,
      owner text NOT NULL,
      description text DEFAULT 'Not Defined'::character varying,
      wikipage text DEFAULT 'Not Defined'::character varying,
      creation_date date NOT NULL,
      CONSTRAINT filesystem_pkey PRIMARY KEY (id)
    ) WITH (OIDS=FALSE);
    ALTER TABLE gis_schema.filesystem OWNER TO gis;

CREATE TABLE gis_schema.wms
(
   id serial NOT NULL,
   wmsname text,
   url text,
   immut boolean,
   owner text,
 description text,
 CONSTRAINT wms_pkey PRIMARY KEY (id)
)
WITH (OIDS=FALSE);
ALTER TABLE gis_schema.wms OWNER TO gis;

CREATE TABLE gis_schema.wfs
(
   id serial NOT NULL,
   wfsname text,
   url text,
   immut boolean,
   owner text,
 description text,
```

```
 CONSTRAINT wfs_pkey PRIMARY KEY (id)
)
WITH (OIDS=FALSE);
ALTER TABLE gis_schema.wfs OWNER TO gis;
```

In order to create a generalized search tree (GiST) to index a geometry column, the following SQL statement must be executed:

```
CREATE INDEX [tablename]_[geom_column]_gist
       ON gis_schema.[tablename]
       USING gist ([geom_column])
```

In order to cluster a non–editable (read–only) spatially enabled table using the GiST index:

```
ALTER TABLE gis_schema.[tablename]
      ALTER COLUMN [geom_column] SET not null;
CLUSTER [tablename]_[geom_column]_gist
        ON gis_schema.[tablename];
```

Requirement [7] concerning annotation table generation is a subject of the business tier and will be discussed in the appropriate section.

5.4.5 GeoMoin DBMS Wrapper: pgUtils

In GeoMoin, database access between the data services and the business services can be achieved either by using OSGeo's GDAL/OGR library (if MapServer functions are performed or high–level management of the spatial data is needed) or by accessing the DBMS using PygreSQL's PGDB module. The second approach is preferred when it comes to accessing the "filesystem" and "postgis" tables or there is a need to edit an PostGIS annotation table. In the last occasion, between the PGDB module and the business services in the 2nd level tier, GeoMoin's module *pgUtils* is responsible for the following tasks:

- Builds the actual SQL queries that manage the above parts of database content.
- Manipulates the calls to the PGDB interface in order to prevent SQL injection attacks.

- Acts accordingly in terms of access control, checking if the user that sent the query fullfills the appropriate access limitiations in order to manage the content.

The main call that the business services make, is to the **executeQuery** function which is responsible to call an appropriate handler depending on the sql query type. executeQuery is parameterized with the following arguments:

queryType The sql query type to be performed. It can be an insert,update,delete,drop or select statement

moinUser The MoinMoin's user's unique id that issued the call.

tables A list of the sql tables included in the call.

viewColumns A list of the table fieldnames that the query spots on.

constraint The string that follows the SQL's WHERE clause.

dict A dictionary which contains a mapping between fieldnames and values when an query is processed. It also contains the mappings in order to substitute the constraint string with values.

isReg A boolean value that flags whether the call was made by an authenticated user.

In case the user issuing the call is an unauthenticated user, he is only allowed to perform SQL SELECT queries. Every query performed, whether a user is authenticated or not, is filtered through a flexible access control mechanism which is query–type–depended. Illustrating it with an example, let us suppose that a user requires to view the details of a particular annotation depicted on a map. First of all, the required information are gathered in order to form a SQL SELECT statement. In GeoMoin DBMS schema specification 5.4.4 it was defined that if an annotation table included in the "postgis" registration table is marked as viewable, then all its tuples are viewable (tuples in an annotation table contain fields that constraint only the editability of themselves). Thus the following flowchart [5.4] shows the roll of access control events when a SELECT query is performed.

As it is shown in the flowchart, the table owner has an over–all access to the table.

If the query is consisting of an INSERT statement, then the access control mechanism has to evaluate the editability of the tables checking their registrations in the "postgis" table. Flowchart [5.5] shows the access control procedure for these types of queries. In case of an SQL DROP query, the trigger of the drop sequence is one of the following:

- The user wants to drop a PostGIS annotation table.

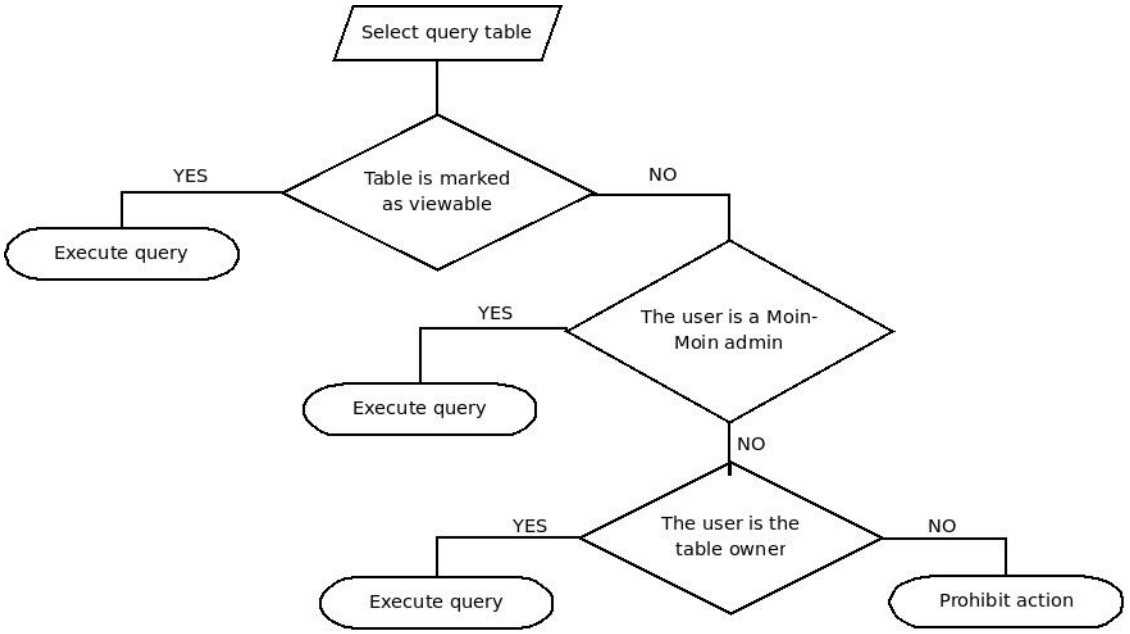


FIGURE 5.4: Access control flowchart for a SELECT query.

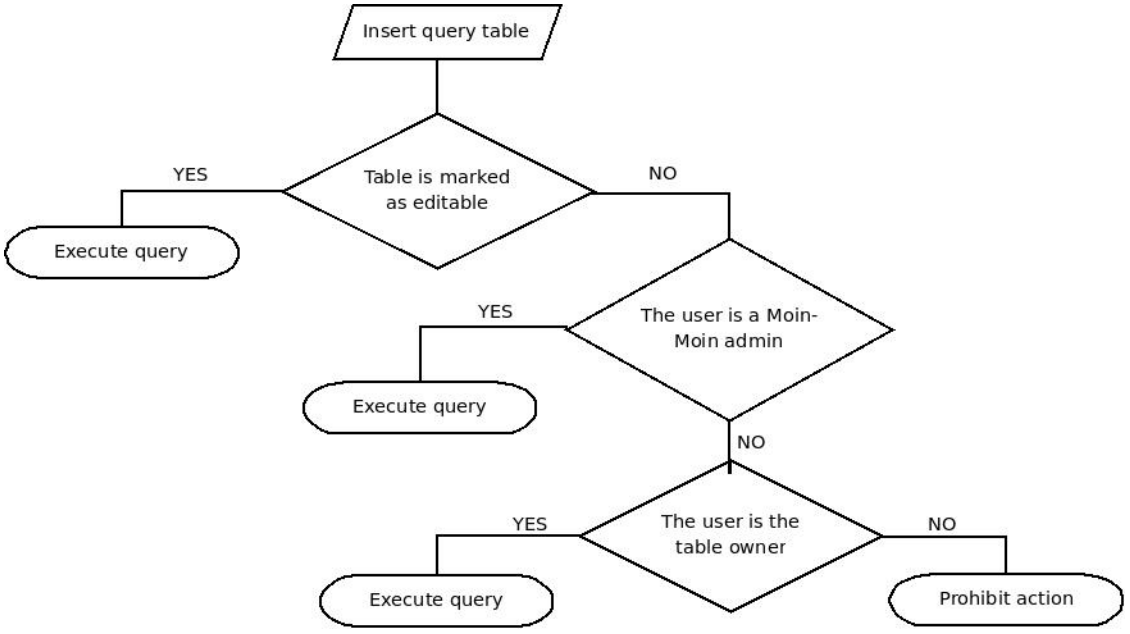


FIGURE 5.5: Access control flowchart for an INSERT query.

- The user wants to drop a PostGIS table which is NOT an annotation table.
- The user wants to remove a filesystem dataset

In the first two cases, GeoMoin allows **only the owner of the PostGIS table or a GeoMoin administrator** to execute the DROP query, evaluating the ownership by checking the registration in the "postgis" table.
In the third case, GeoMoin allows **only the owner of the filesystem dataset or a GeoMoin administrator** to execute the DROP query, evaluating the ownership by checking the registration in the "filesystem" table. It must be noted that in case a filesystem dataset is evaluated to be deleted, pgUtils wrapper accesses the filesystem and deletes the files consisting of the name of the dataset and every possible extension. For example, if an ESRI shapefile dataset is registered in the "filesystem" table with the name "countries", it will be associated with filesystem elements like: countries.shp, countries.dbf, countries.prj.

The last SQL statement that is processed by pgUtils is the UPDATE. An update query intents to access and update specific tuples in an database table, thus GeoMoin defines access controls that handle each tuple access. In the GeoMoin DBMS schema specification [5.4.4] it was defined that every annotation table, for each of its tuples must contain metadata that define the user that created the specified tuple and whether this tuple can be edited by the public or not. Thus the flowchart [5.6] shows the access control checks that take place when an update is performed. It is important to note that the registration tables "filesystem" and "postgis" are also managed using pgUtils, in order to update or insert data in them. The only difference is that they are not subjective to access control restriction as they must be editable by every user of the wiki. Of course, the business logic of application prohibits direct access, on the contrary, it allows operations that are solely targeting to the datasets attributed to the user that installed them.

In order to pass query strings to pgUtils, they must be edited in order to prevent malicious users from executing SQL injection attacks. The manipulation of the query string can be illustrated using the following example.

```
SELECT query to be executed:
    select adm,years
    from gis_schema.postgis
    where name='indiana' and pop<=12000;
```

The above query, if we assume that the "name" and "pop" values in the WHERE clause are filled by the user using an HTML form, then a "pop" value of: "12; DROP TABLE

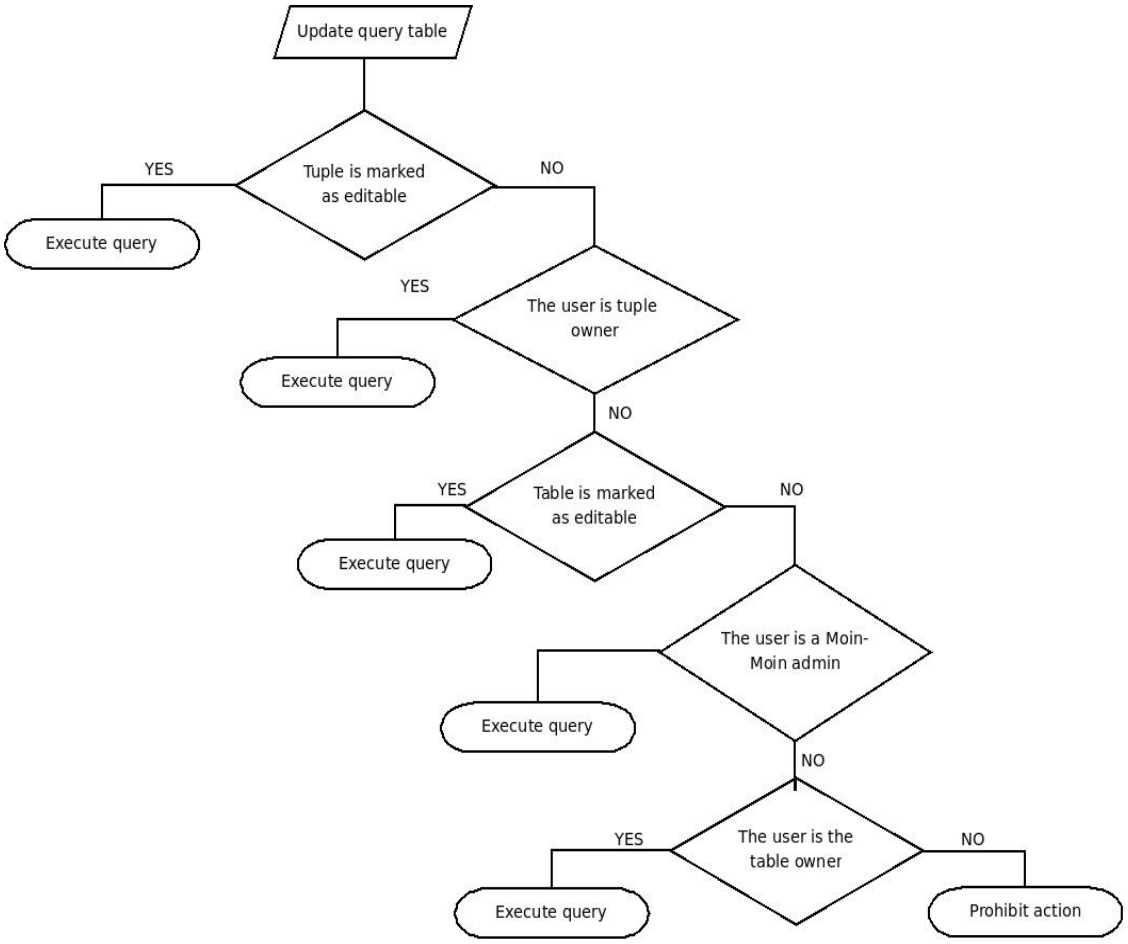


FIGURE 5.6: Access control flowchart for an UPDATE query.

postgis; –" instead of "12000" could easily drop the postgis registration table. Thus before executing the query using the PGB module, the following manipulation must take place:

```
SELECT query to be executed:
    select adm,years
    from gis_schema.postgis
    where name=%(name)s and pop<=%(pop)s;
Additionally the following python dictionary must be provided:
    dict={"name":"indiana","pop":"12000"}
```

The pgdb module is responsible for substituting the values after they are sanitized to exclude SQL injection possibilities.

5.5 OWS data services

In Chapter 4 we discussed the ability of mapserver to render resources based on remote WMS or WFS services. In GeoMoin we make use of those services in order to give the ability to the users, to create maps based on remote datasets that do not exist within a GeoMoin installation. In the specification presented above, two tables are responsible for storing information about the current WMS and WFS servers and are called "WMS" and "WFS" respectively. Thus, these tables can be considered as keepers of metadata concerning datasets that are stored to remote locations. The users can select particular layers from within each server and use the exactly as if they selected layers stored in the filesystem or the Postgresql DB.

Summing up the discussion about the Data Services, we described GeoMoin's support for spatial datasets including datasets contained in the filesystem, PostgreSQL and OGC web–services. It must be noted, that currently GeoMoin does not support all the types of datasets (vector,raster) that are accessible through OGR/GDAL and in addition, it does not support all the OGC Web Services including the Web Coverage Service (WCS). GeoMoin, though, can be easily expanded to include many of the features that does not exist in the current release.

5.6 Business tier

In the following two sections, the Business tier of the GeoMoin architecture will be described, and in particular, the Map/Dataset Controlling System.

5.6.1 InstallManager

In order to install spatial information to GeoMoin the user must upload the datasets and install them to the system. The component dealing with these particular functions is called *InstallManager*. This feature is a MoinMoin action module that it can be accessible from every page in the wiki. The nature of the installer as an action was selected in order to faciliate the binding between a wiki page and the dataset; a page that contains an attached dataset is supposed to contain information about the author, origin or a sample usage of it. The procedure that takes place in order a dataset to be installed is:

1. Uploading of the dataset in an archived format (currently only .zip archive is supported).

2. Selection of the installer action in order to install the dataset.
3. Selection whether a dataset should be installed in the filesystem or database.
4. Dataset installation takes place.

Below, the elements of the above listing are described in detail.

5.6.2 Uploading a dataset

In MoinMoin, every page created, support its own attachements to be uploaded. That is, the attachments are bound to that specific page. A user, in order to upload spatial information can use the default MoinMoin *Attachments* action. In order to maintain as much consistency as possible, a dedicated uploading module for spatial information was not created, but some restrictions concerning the nature of the attachment were applied; the user must package the datasets in format accessible using GeoMoin[7] and add the directive "DATASET_" in front of the package name. Thus the user can package the following ESRI shapefiles,

```
usa_states.shp
usa_states.dbf
usa_states.shx
usa_cities.shp
usa_cities.dbf
usa_cities.shx
```

in a archive named: "DATASET_usasimple.zip". Finally, the archive can be uploaded to the server.

Limitations concerning the types of spatial dataset that GeoMoin can process, also exist. Currently GeoMoin support the following types:

- ESRI Shapefiles
- Mapinfo
- Geography Markup Language (GML)
- Various raster formats

It is importand to note that, currently, an archive cannot contain different types of datasets.

[7] Currently only the zip archive format is supported.

5.6.3 Invoking the installer action

When the uploading of the datasets in the wiki page is complete, the user can select the *InstallSpatial* action in order to install them to the system.

When InstallSpatial is executed a listing of all the spatial attachements is gathered and presented to the user along with the following options concerning the installation.

- The user can select to install all the datasets contained in the attachement, or a specific one.
- The user can select whether the datasets should be installed in the filesystem or PostGIS.

In case they are installed in PostGIS the user can enforce a geometry type. This method solves installation problems where a dataset created using a polygon geometry is rejected by PostGIS which recognises it as multipolygon. That is, different GIS products, support different vector models.

The type of the dataset contained in an attachement is recognised using the extension of the files. For example, an ESRI dataset will always contain a file with the extension ".shp", while the other files associated to this particular dataset will have always the same name with different extensions. Using the method presented above, all the ESRI datasets contained in an attachment can be isolated, just by checking the extension and keeping the rest of the filename, thus the user can be presented with a listing on which dataset will be installed.

5.6.4 Installing the datasets using OgrTypes

OgrTypes is a GeoMoin module that acts as a "driver" for the spatial formats that GeoMoin supports. Every python class implemented in OgrTypes is, in fact, a driver for every type of spatial dataset. Along with an on–going number of facilities that the drivers provide to GeoMoin's higher level functions, each one contains a unique installation method for the dataset format it manages. For example, the ESRI driver contains two functions called *install_filesystem* and *install_postgis* which install the dataset to the filesystem and the postgis respectively.

Figure 5.7 shows the flowchart that describes the events that take place when the attachement is going to be stored to the filesystem. The extraction and gathering of the archived datasets in the install list is a subject of the InstallManager, which, for each

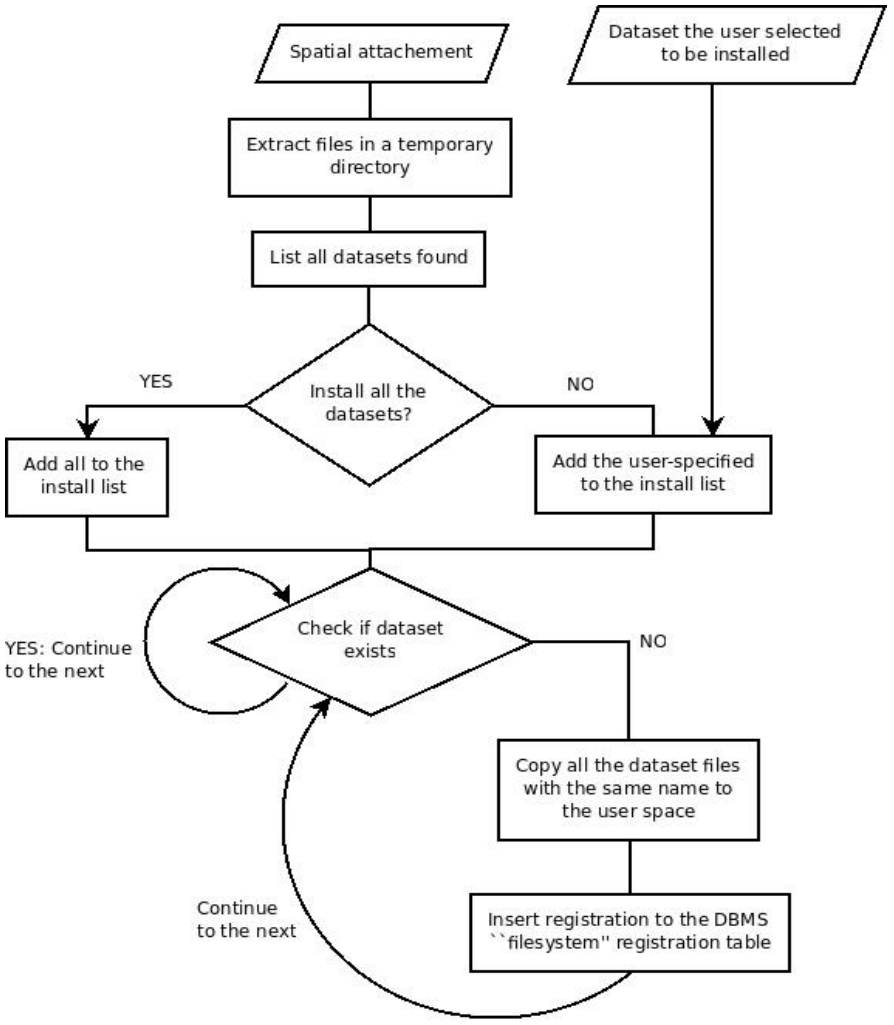


FIGURE 5.7: Installing a spatial attachment to the filesystem.

datasets, instantiates the correspondive driver and executes the appropriate function. In order to move the datasets to the filesystem the os python module is used, which copy the required files from the temporary space to the current user's unique space in the filesystem. The registration of the table in the PostgreSQL table "filesystem" is a INSERT query handled using the **pgutils** module described in a previous section. More insformation can be found directly in the source code of the application.

It is important to note that every different dataset format may have a specific installation routine, although the majority require the same processing steps. For example, GML datasets can contain multiple layers within a single gml file. Thus, the OGR library must be used to extract the layer names and check whether they exist in the registration table. If not, the multiple layers must be solely registered in the "filesystem" table.

After the installation script has finished executing, a detailed listing of the results is presented to the user, showing the datasets that were successfully installed and those that generated problems.

Installing a dataset to the PostgreSQL database is a more complicated routine and additionally, error prone. The reason behind this, is the need of an intermediate translation of a dataset to the sql language equivelant, which is accomplished with the execution of the OGR/GDAL utility *ogr2ogr*. The following command can be used to install an ESRI dataset to PostgreSQL:

```
ogr2ogr -append -update -f PostgreSQL PG:"host=localhost dbname=gisdb  \
        user=gis password=qwerty port=5433"\
        -lco SCHEMA=gis_schema /tmp/usa_datasets/statebounds.shp
```

A spatially enabled table is created in the DBMS under the name of the layer processed. Below, we present two possible errors that can make PostgreSQL reject a particular dataset:

- The geometry of source dataset does not match the geometry constraints (for that particular source geometry) applied by the PostGIS extension due to different vector model adopted. For example, a polygon source geometry, may be recognized as a multipolygon one.
- The database encoding[8] does not match the encoding of the byte sequences containing in the source dataset.

The first problem can be bypassed, by solely installing the particular dataset while enforcing a geometry conversion. This conversion can be done by adding "-nlt enforced_type" in the arguments of ogr2ogr utility, for example, adding "-nlt multipolygon" converts a geometry to multipolygon. Of course a user may intentionally enforce a geometry conversion in order to achieve polygon geometries to be converted to multipoint ones etc. In both cases, it is mandatory the user that contributes the spatial content is completely aware of information and the datatype contained within, if not, the user can upload the dataset and possibly notify the administrator or another user to install it.
The second problem cannot be fixed by GeoMoin; it requires the removal or the replacing of the suspected byte sequences to ones that match the encoding of PostgreSQL.

Figure 5.8 shows the flowchart that describes the events that take place when the attachement is going to be stored to the database. The registration of the table in the PostgreSQL table "postgis" is a INSERT query handled using the **pgutils** module described in a previous section. More insformation can be found directly in the source code of the application. Likewise datasets installed in the filesystem, different datasets have

[8] GeoMoin's PostgreSQL uses the UTF-8 encoding.

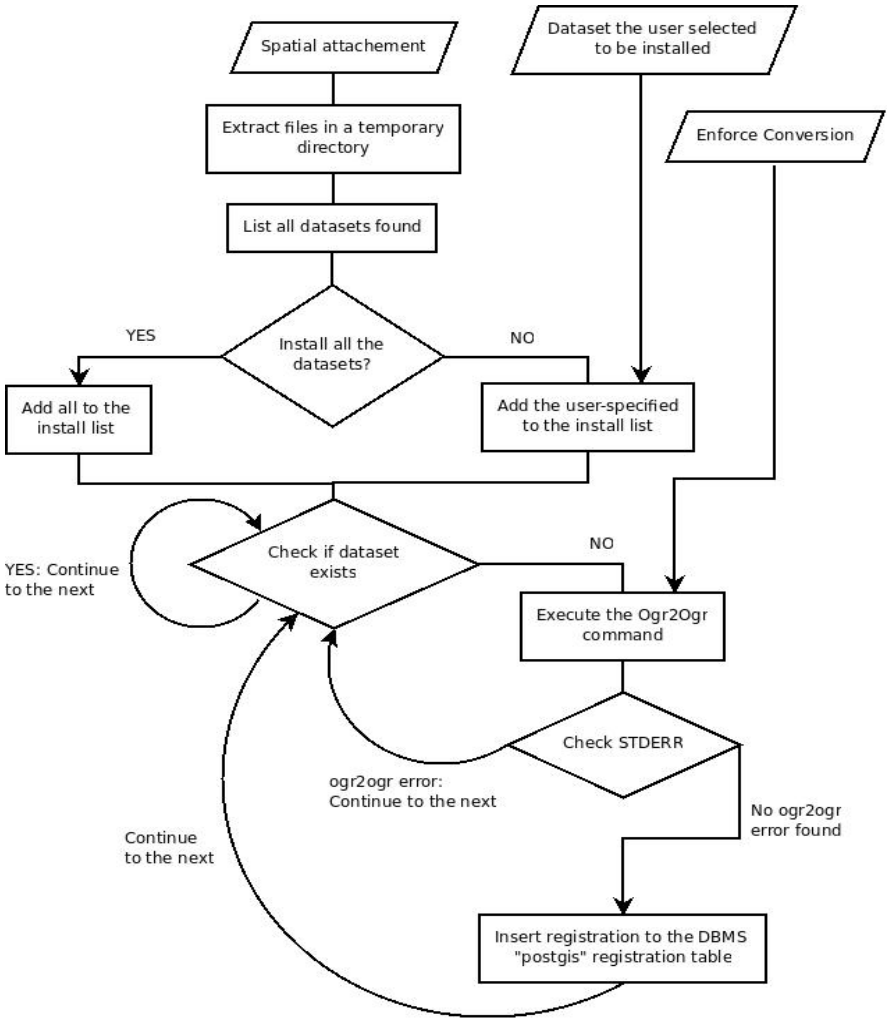


FIGURE 5.8: Installing a spatial attachment to the database.

different install procedures; GML for example, must be opened using the OGR library to check whether multiple layers exist thus checking for duplicates can take place.
After the installation script has finished executing, a detailed listing of the results is presented to the user, showing the datasets that were successfully installed and those that generated problems.

5.7 Dataset controlling system: LayerManager

The lower logical levels of the business logic in GeoMoin include a sub–system that is responsible for managing that spatial data contained in Data Service tier and it goes with the user–friendly name LayerManager. It is developent as a MoinMoin macro, which gives the ability to be included in many pages of the wiki.

In general, interaction between the user and LayerManager is HTML form–based, meaning that upon submission the page will reload and the macro will be reinvocated, this time accessing the submitted form elements.

LayerManager is particularly useful in the following use case because it allows the user to:

- Query the spatial datasets' metadata in the database and get a listing of those, matching the query criteria.
- Update the metadata of the spatial datasets that he installed.
- Delete the spatial datasets that he installed.
- View spatial attributes about every dataset using the OGR/GDAL.
- Create a markfile containing layer definitions of the spatial datasets selected, in order to be used with the MapManager.
- Create a new annotation tables in PostGIS, with an unlimited number of fieldnames and only one geometry column.
- Add,edit,delete WMS/WFS servers.
- Connect to WMS/WFS servers in order to access the layers they offer. The user can add layer definition for the WxS layers in his markfile, in order to use them with the MapManager.

5.7.1 Querying and managing the installed datasets

Using the LayerManager the user is able to query the metadata contained in the registration tables "filesystem" and "postgis" in order to view the installed datasets that match the query criteria. Connection to the DBMS is established through the PgUtils wrapper issuing the appropriate *viewquery* commands. A query execute could look like this:

```
select *
from gis_schema.filesystem
where type='ESRI' and (creation_date>2008-02-01 and creation_date<2008-06-01)
```

The above query will return all the ESRI shapefiles installed in the system between those dates.
The user can choose from the following query criteria:

- The dataset name
- A date range as in the above example
- Datasets installed by a particular user
- The dataset type
- A part of the description of the dataset

Of course all the above query criteria can be combined with each other to construct very specific queries. Spatial query capabilities are not implemented yet, but such queries are demanding in terms of computing resources.
The results presented to the user are divided in three basic categories; the datasets installed in the filesystem, the dataset installed in PostGIS which are not annotation layers and the annotation layers in the PostGIS.

For every spatial dataset the query has returned, the user can use an appropriate control which opens the layer information of the particular dataset. In the previous section we described the role of the *OgrTypes* module, which depending on the type of the dataset, dispatches the control to the appropriate sub–class that can handle the particular dataset. Almost every OgrTypes sub–class contains a function called *ogrInfo* which is responsible to present spatial information about a dataset using the OSGeo's GDAL/OGR library. The information that are presented to the user are the following:

- Internal layer name
- Number of features contained
- The geographic extent
- The geometry type according to GDAL (not mapserver)
- The fieldnames of the non–spatial information included
- The WKT projection definition along with the Proj.4 represention
- The geometry columns contained
- A sample layer definition which can be used in order to add this dataset as a UMN mapfile layer

Through the LayerManager the user has the ability to manage the datasets that were installed in the system by his account. The registration tables "filesystem" and "postgis", which contain the datasets' metadata, can be updated to include the user requested values. In addition, the user can delete any dataset that he contributed to GeoMoin.

5.7.2 Generation a UMN mapfile layer definition

GeoMoin gives the ability to create a dynamic layer definition of a spatial dataset, provided that the type of the dataset is supported by the system. OgrTypes module is responsible for this type of requests. Feature 5.9 shows the flowchart of the actions that need to be performed in order to create the layer definition.

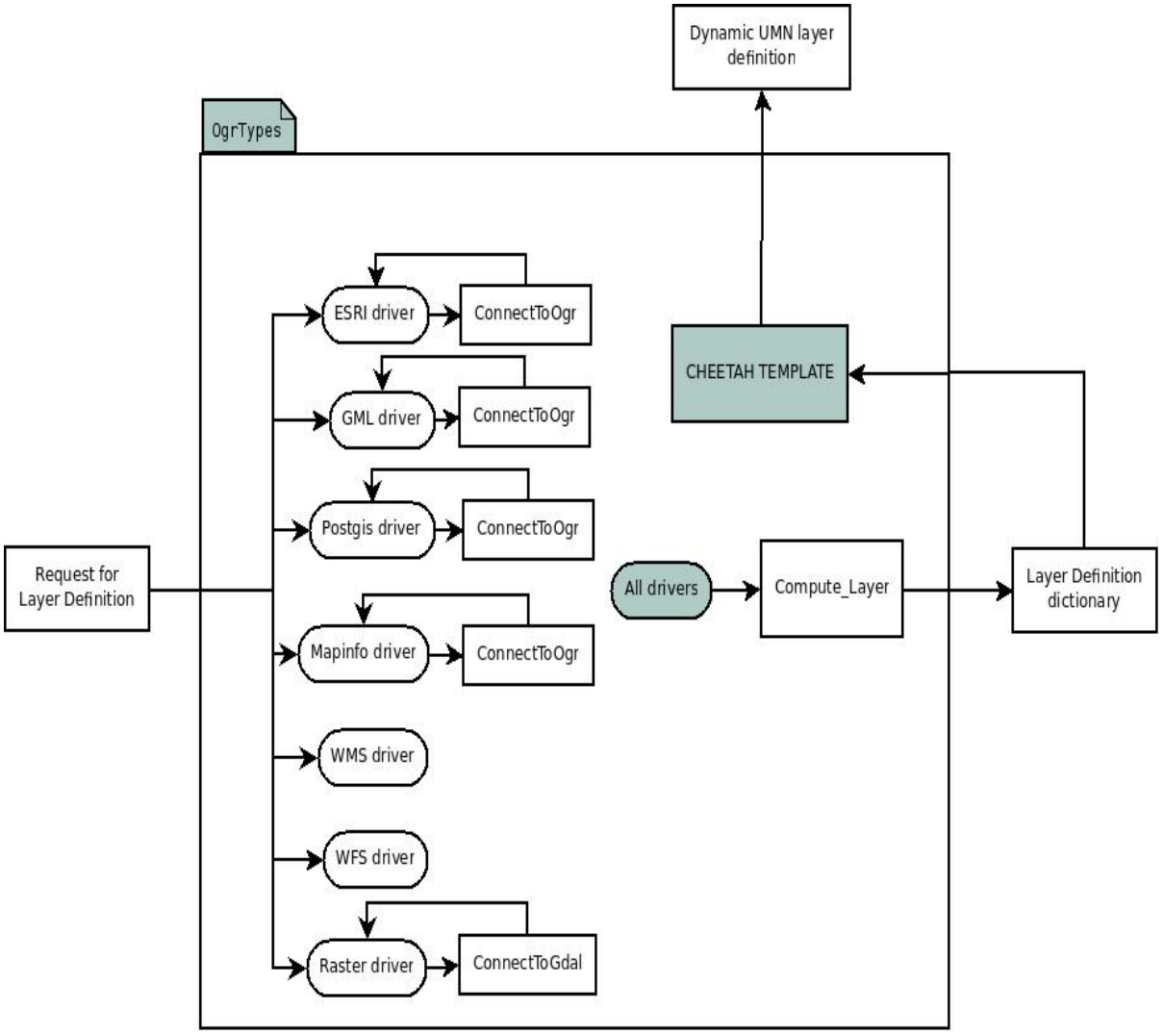


FIGURE 5.9: OgrTypes and layer definitions.

The request for a layer definition contains information about the type of the dataset, the name and its location if it stored in the filesystem. The re–invocation of the Layer-Manager (after the submit) can use the HTTP request object to get these information and call the appropriate functions. According to the type of the dataset the appropriate driver class is instantiated from the ogrtypes modules. Each driver contains the following functions:

connectToOgr Use the OGR library (for vectors) and GDAL (for rasters) in order to extract feature information for the layer being accessed. We are interested in the following elements:

- The layer name
- The extent its features cover
- The name of the non spatial columns
- The geometry type
- The number of features in the layer
- If it is a PostgreSQL layer, the name of the geometry column. (This is actually accessed by a query in the "postgis" registration table)

The values returned from the feature extraction procedure are used to set the member variables of the driver instance, in order that other functions can have access to them.

compute_layer Creates a pre–defines dictionary, wrapping the information gathered by the connectToOgr function, along with a draft generation of a mapfile CLASS directives that will be used to render the layer.

After the dictionary is created, the following use–case can take place:

If the user requested information on a dataset, the above procedure can take place, and the generated dictionary can be used to create a mapserver mapfile definition of the layer and be presented to user. Additionally, the function **ogr_info** displays various information about the dataset.

If the user requested that the dataset should be used to create a map, the above procedure can take place, and the generated dictionary will be pickled in a special markfile. This markfile will be later used by the MapManager to generate the actual mapfile layer definition.

5.7.3 Creating an annotation layer

The LayerManager gives the ability to create and install new annotation layers in the PostGIS DBMS. An annotation layer constists of only one spatially–enabled column, which holds the geometry of the element annotated, and an unlimited number of non–spatial fields that can keep the following types of information:

- text

- integer
- float
- date

Additionally, the user is requested to fill the metadata of the new annotation table which includes:

- The annotation name
- An optional description
- The wiki page associated to the annotation layer
- Selections concerning whether the table can be editable and viewable by other GeoMoin users.
- The geometry type of the annotation; It can be a point or a polygon.

Upon submitting the request, the new annotation table is installed in the DBMS and can be accessed through the LayerManager. The installation procedure creates a transaction declaring a PGDB cursor and the following procedure is executed using SQL statements:

1. The table is created using the non–spatial fieldnames defined in 7th requirement in 5.4.4.
2. The ownership is set to the database user responsible for GeoMoin requests.
3. A geometry columns is generated and added to the table using the postgis function *AddGeometryColumn*
4. A GiST index is created on the the geometry columns generated above.
5. A registration for the new annotation table is added to the postgis registration table.
6. If one of the above statements fail the cursor is used to rollback the transaction.
7. If everything is successfull the cursor is succefully commited.

At any time, the owner of the annotation table can edit the registration information such as the decription of the table, or its access controls or even delete the whole table.

5.7.4 WMS/WFS access through LayerManager

Remote servers providing WMS and WFS content can be accessible through the LayerManager. GeoMoin's users can contribute, registering new WMS/WFS servers which can be accessible by everyone. As far as the managing of the WxS servers is concerned, the user can:

- Add a new server to the system inserting the following information: a name for the WxS server, a description and the url of the online resource.
- Update the above information or delete a registration
- Connect to a WxS server. The layers will be listed so the user can select which one to use.

All the information about the WxS servers installed in the system are stored in a special registration table in the DBMS called "wms" and "wfs" respectively. The calls to the pgdb module are passed through the pgUtils [5.4.5] module in access any access control restrictrions are needed in the future.

Concerning the client connection to a WxS online resource GeoMoin uses the OWSLib[9] module, which provides a common API for accessing the services' metadata and wrappers for GetCapabilities, GetMap, and GetFeature requests. Connecting to a WMS server using OWSLib is straightforward:

```
>>>from owslib.wms import WebMapService
>>>wms = WebMapService('http://wms.jpl.nasa.gov/wms.cgi', version='1.1.1')
>>>print wms.capabilities.service
'OGC:WMS'
>>>print wms.capabilities.title
'JPL Global Imagery Service'
>>> print ",".join(layer.name for layer in wms.capabilities.contents)
['global_mosaic', 'global_mosaic_base', 'us_landsat_wgs84']
```

Issuing the above commands, a list for the layers contains in the WMS is returned to the user. Of course every layer can have many styles associated to it, thus every possible style option is returned. Selecting a layer from the listing described above, the user can generate a UMN mapfile layer definition using the WFS or the WMS driver of MoinMoin's OgrTypes module. The current MapServer version defines that the majority

[9] OWSLib is created by Sean Gillies and distributed under GPL.

of the information needed to render a WxS layer must be include in the METADATA directive of the layer definition. For example a mapfile layer acting as a client for NASA's WMS service could look like:

```
LAYER
    NAME "BMNG"
    CONNECTIONTYPE WMS
    CONNECTION "http://wms.jpl.nasa.gov/wms.cgi?"
    STATUS ON
    TYPE raster
    METADATA
      "wms_srs"  "EPSG:4326"
      "wms_transparent"  "TRUE"
      "wms_name"  "BMNG"
      "wms_format"  "image/jpeg"
      "wms_server_version"  "1.1.1"
      "wms_style"  "default"
    END
END
```

Thus the information gathered using OWSLib, concerning the service metadata, are passed using an HTTP request to OgrTypes and the layer definition is created. Finally, the user can use the MapManager to import the layer to a mapfile.

5.8 Map controlling system: MapManager

The business logic in GeoMoin includes a sub–system that is responsible for managing the MapServer mapfiles that are created by the users of the web application and it goes with the user–friendly name MapManager. Strictly speaking, MapManager is a GUI where the GeoMoin user can manage almost every directive that a mapfile consists of.

5.8.1 Creating and editing a mapfile

There are two possible ways for a mapfile to be edited:

- Accessing the mapfile's content in a raw form using a text editor.
- Accessing the mapfile's content in a GUI where every directive can be indepentently managed.

Mapfiles are part of the Data Services and are stored in the filesystem under directories unique for every user and named under their MoinMoin unique userid. Using this approach, two mapfiles sharing the same name, created by different users will not override each other.
MapManager is implemented as a MoinMoin macro, thus it can be loaded in every page in which the macro directive is appended. The logic of the macro execution is HTML form–based, meaning that the user is presented with forms and upon submission the page is reloaded. The macro will be executed again and will check the query string in order to dispatch the control to the appropriate functions.
When the macro is first executed it checks whether the user is a MoinMoin authenticated user. If the user has successfully logged in, the following elements are presented:

- A listing of the user's mapfile in order to load them to the text or the GUI editor. Additionally the user can delete the map he selected.
- A form where the user can create a new mapfile.
- A form where the user can manage the layers that were selected in the LayerManager.

If the user selects a text edit, the next invocation of the macro outputs a HTML TEXTAREA containing the mapfile's contents where he make the appropriate changes and the submit the edit. The last invocation rewrites the current mapfile and presents the elements listed before.
In case a GUI edit is needed, the mapfile must be properly parsed and displayed to the user along with controls in order to change the directives. The first design of the MapManager was instantiating a mapscript Map Object from the mapfile and used the associated getter/setter methods to access the directives. This method was a hands–on approach concerning the parsing of the mapfile as it is left to mapserver library. The drawback was the lack of support for many directives using Python Mapscript. In example, the GRID directive as soon as the map object is instantiated, is embedded in the layer object that contains it, leaving no accessors to the GRID directives. Also, some directives have values, although they are not defined in the mapfile, that is, they are attributed initial values.
Thus, currently, a unofficial parser was created and is being used for the needs of MapManager. Although there exist some serious limitations:

- Commentation is not supported. When the map is saved the comments are lost.

- Every directive must be contained in its own line. Native mapfile parsing allows multiple directives to be declared in a single line, although it is not a recommended practice as far as the readability of the mapfile's content is concerned.

this method is far more convenient than relying on mapserver objects just to parse and update a text mapfile. The parsing is done using the module **mapparser** and it is better left to the commentation of the module itself to describe the methods used to analyse the mapfile syntax.
When the parsing has finished, the information are stored in a Python dictionary which has the structure shown in figure 5.10:

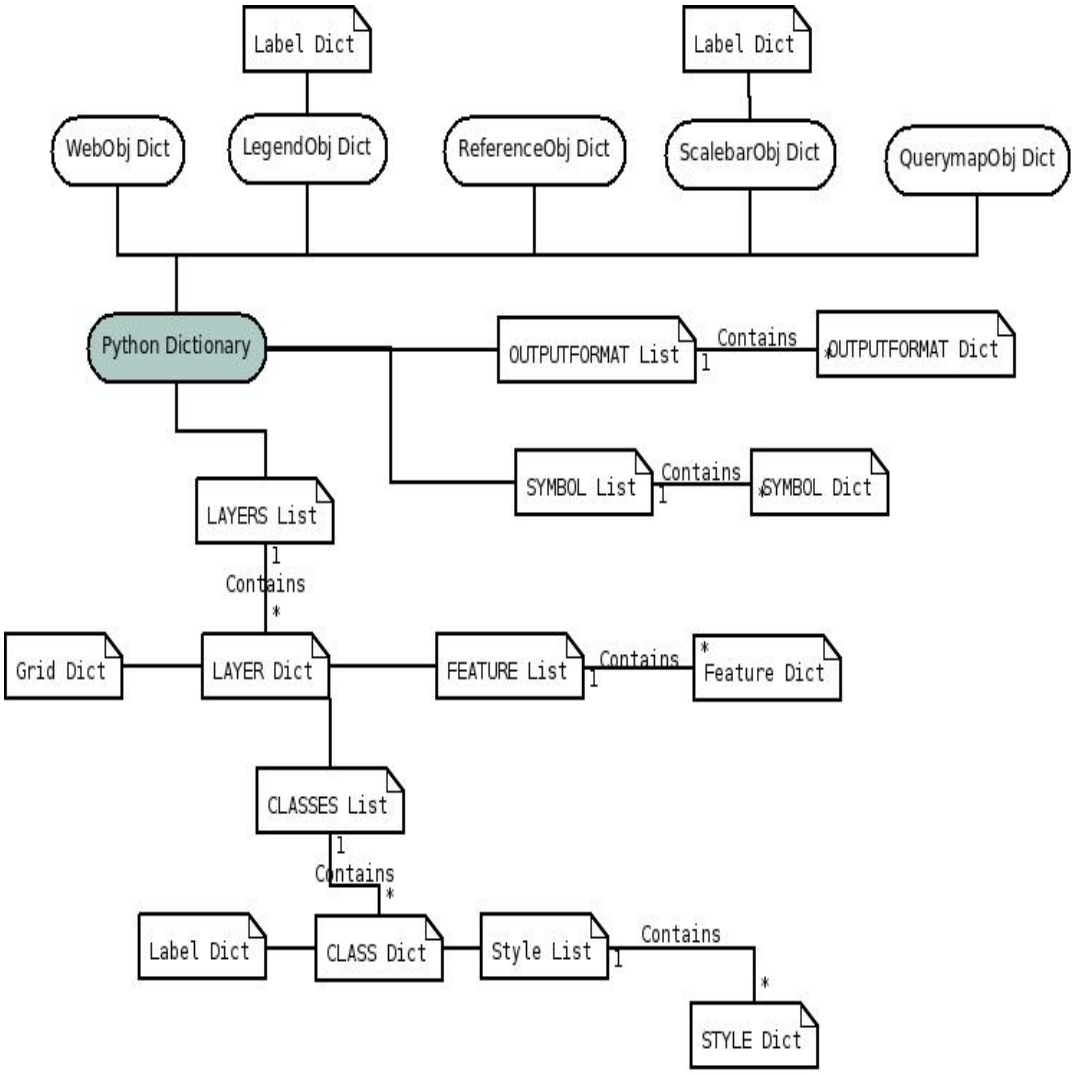


FIGURE 5.10: A python dictionary containing the parsed map object.

Moreover, as soon as the parsing of all the mapfile directives has taken place, the macro creates two markfiles (defined in section 5.3.3) and stores the LAYERS list separately

from the rest Map Object elements. The role of the markfiles is that they maintain data consinstency between many invocations of the macro. Thus in the next invocations the mapfile will not be parsed again, instead, the markfiles are read and the dictionary will be loaded. Another reason for keeping markfiles is that the updating of the directives can occur within many invocations before submitting the new map file, whereas, in case of a re–parsing, the previous changes would be lost.

The markfile holding the layers that were parsed from the mapfile populates the first HTML FORM that is presented to the user. Using that form the following can be accomplished:

- A specific layer definition can be update or deleted
- A class can be created,updated,deleted for a specific layer
- A style can be created,updated,deleted for a specific class
- A label can be created,updated,deleted for a specific class
- An inline feature can be created,updated,deleted for a specific layer
- A grid object can be created for the layer

Upon submission the markfile holding the mapfile's layer information will be rewritten in order to contain the new directive values.

The markfile holding the layers that were generated from the LayerManager populates the first HTML FORM that is presented to the user. Using that form the following can be accomplished:

- A specific layer definition can be update or deleted
- A class can be created,updated,deleted for a specific layer
- A style can be created,updated,deleted for a specific class
- A label can be created,updated,deleted for a specific class
- An inline feature can be created,updated,deleted for a specific layer
- A grid object can be created for the layer

Upon submission the markfile holding the LayerManager's layer information will be rewritten in order to contain the new directive values.

The markfile holding the map elements populates the second HTML FORM that is presented to the user. Actions that can be submitted are:

- Save the map using information from the markfiles
- Update the map object
- Create or update a legend object
- Create or update a scalebar object
- Create or update a reference object
- Create or update a web object
- Create or update an inline symbol
- Create or update an output format object

Upon submission the markfile holding the map information will be rewritten in order to contain the new directive values.

The central html form that is presented to the user, is populated with the contents of the map dictionary. The HTML FORM elements that are used vary from checkboxes and textboxes to drop–down lists according to the requirements of the mapfile elements. More information about the specific mapfile elements can be found in commentation of the source code inside the macro. Additional information about the interaction between a GeoMoin user and the HTML forms will be described when the User Services tier is explained.

When the user finishes the update of specific mapfile elements he can:

- Cancel the whole operation, deleting the current markfiles.
- Save the map information as a new mapfile in the filesystem.

If the user selects to save the map in the filesystem, the python dictionaries for the map, the mapfile layers, and the LayerManager layers, must be used to generate native mapfile definitions. The best approach to problem like that, is the deployment of a templating engine and specifically Python's *Cheetah*[10]. The template *map.tmpl* is responsible for generating all the mapfile directives and objects (eg. scalebar, web) except the layers. Respectively, the template *layer.tmpl* is responsible for building layer definitions for the layers in the mapfile itself or the LayerManager, including the generation of the class objects, grid object, etc which are generated using their own templates.

[10]Cheetah is an open source template engine and code generation tool, written in Python. It can be used standalone or combined with other tools and frameworks. Web development is its principle use, but Cheetah is very flexible and is also being used to generate C++ game code, Java, sql, form emails and even Python code.

5.9 Spatial visualization system

The spatial visualization system, is the most important part in GeoMoin's business services tier, implemending the rendering of the spatial information in the form of an image which is presented to the end user through the web browser. Strictly speaking, the term "visualization system" does not only refer to a subsystem responsible for image renderings, but also to all the functions that can be supported through mapscript including the querying of the spatial information illustrated by the rendered map.
It is important to note that the spatial visualization system is not responsible for the output of the information in an output format[11]. It produces the results and gathers the required information which will be later send to the appropriate scripts for an HTML output.
The native features supported by the renderer are the following:

- Graphical output of the map
- User tools for controlling the map rendering (pan,zoom,layer ordering e.t.c)
- Managing of annotation tables
- Spatial and attribute queries
- Tools providing integration between the application and the whole wiki

The feature listing can be enhanced by the use of *mapplets* (described later) which add to the functionality of the system.

In the section below various aspects of the renderer implementation will be described, starting from the nature of the renderer as a MoinMoin plugin.

5.9.1 Definition and implementation of the renderer

The rendering and navigation in a GeoMoin map can be considered as a three–step process with the last two steps being repeated as long as the user navigates through the map. Throughout this process information are being logically exchanged between the business and user tier of the GeoMoin architecture. Five building blocks can be identified throughout the whole process:

- A parser that initializes the map rendering information.

[11] Output of the information to the user browser through the web–server is attributed to the user–services tier [5.1].

- The main application that handles the spatial visualization and all the various spatial routines. For the rest of the document this application will be refered to as **MapRenderer**.
- The AJAX framework, that serves as an intermediate between the main application and the user request through the browser.
- XML-RPC scripts following the WikiRPC specification, used to connect and manage the wiki through the main application for various business logic purposes.
- The Mapplets, which that can be considered as MapRenderer plugins.

First of all, when the wiki page containing a map is requested by the client, a **MoinMoin parser** is executed which resides in the wiki markup. The parser is responsible for initializing the map while gathering the required information needed for rendering and creating the initialization HTML markup that will be delivered to the user browser. In the body of the MoinMoin parser, the user can parameterize many aspects of the rendering procedure. Below we define a list of the options available to the user:

mapfile The name of the mapfile that will be processed and rendered.

owner The name of the user that owns this mapfile. The mapfile and the owner directives will be used to fetch the mapfile from the filesystem.

minimal Flags whether the output of the rendering procedure will be a minimal subset of the whole, including only the rendered image along with some navigational tools.

need_reference Flags whether a reference image must be generated.

reference_image A reference image name, included in the wiki page that contains the parser, as an attachment.

need_legend Flags whether a legend must be generated.

need_scalebar Flags whether a scalebar must be generated.

debug Debugging on/off.

override_projection The proj.4 definition that can be used to render a map using a different projection than the one defined in the mapfile.

annotation (0..*) Defines the name of the layers in the mapfile that should be managed using the annotation controls. Of course, the layer defined here must be actual annotation tables installed in PostgreSQL,

mapplet (0..*) Name of the mapplet that will be included in application displayed to the user.

The above list will be enhanced in future versions to include more rendering options.

When the parser is executed, the above information are parsed by the class named *PageDataProcessor* (PDP) while notification messages are presented to the user in case of any misconfiguration. A *projector* object is finally instantiated, which, having access to PDP's member variables, outputs a draft HTML output to the user browser using the appropriate MoinMoin formatter, in particular, *rawHTML*.
As long as, the HTML output is sent to the user browser, the user, using AJAX controls can navigate through the map. It is important to note that during the initialization of the map when the wiki page loads, the MapRenderer should be automatically called in order to create an initial rendering of the map and engage the navigation tools. Thus using the event *onLoad* on the body of the page, the AJAX functions are called and the initial rendering is taking place.

In the following sections, the building blocks of the spatial visualization system are expanded and described, while the AJAX functionality is discussed at the end in order to natively move to the user–tier of the architecture.

5.9.2 The map rendering process

The main role of the spatial visualization system is to render a map in a graphic format and to deliver it to the end–user. The MapRenderer is the building block responsible for this operation as well as many others, managing spatial content.

Rendering a map using Mapscript is a process which follows a predefined workflow as far as a simple image presentation of the map is concerned. On the other hand, in case special handling of the information rendered is needed, the implementation can be complicated. In the current section, we will describe the standard procedure excluding the feedback from the user controls or the querying rendering results.

The name of the map specified in the parser's body, along with the owner of the map itself, define the absolute path of the mapfile in the filesystem. Before instantiating an mapscript **mapObj** object, the map must be filtered through the Cheetah template mechanism in order to substitute the system–specific details that are included in the mapfile as template definitions. For example, a layer in the filesystem included in the mapfile should not be presented by its full path to avoid the exposure of the server's filesystem structure as much as possible. The resulting map is saved in a temporary file

and can be used to instantiate the mapObj which will drive the rendering procedure. The whole operation is implement as follows:

```
namespace={}
namespace["default_fontset"]=DefaultConfig.font_path
namespace["default_symbolset"]=DefaultConfig.symbol_path
namespace["default_imagepath"]=DefaultConfig.img_path
namespace["default_imageurl"]=DefaultConfig.img_path
namespace["default_attach_path"]=DefaultConfig.attach_path
namespace["default_log"]=DefaultConfig.mapserver_log_path
namespace["db_host"]=DefaultConfig.db_host
namespace["db_user"]=DefaultConfig.db_user
namespace["db_pass"]=DefaultConfig.db_pass
namespace["db_name"]=DefaultConfig.db_name
namespace["db_port"]=DefaultConfig.db_port
namespace["db_schema_exp"]=DefaultConfig.db_schema_exp
mapfile=DefaultConfig.map_path+pdp.mapfile
t = Template(file=str(mapfile),searchList=[namespace])
# Here we create the temporary map with the substituted values
fd, tmap = tempfile.mkstemp(suffix='.map')
f = os.fdopen(fd, 'wb')
f.write(t.respond())
f.close()
```

The map can now be parsed by mapscript and a mapObj can be returned:

```
map =  mapscript.mapObj(tmap)
```

The map object can be used in order to alter all the elements of the map rendering or to query the map. While these option will be discussed in the following section, now the actual rendering procedure is presented.
Depending on whether the user selected additional rendering of a scalebar, legend and reference image, these objects must be drawn and saved in the filesystem with unique filenames in order to be used by the HTML displayer. The code snippet of the above actions is:

```
self.map_id = str(random.randrange(999999)).zfill(6)
map_id=self.map_id
image_name = pdp.mapfile[:-4] + map_id + ".png"
```

```
self.image_url="/tmp/" + image_name

if pdp.need_reference==True:
   ref_name = pdp.mapfile[:-4] + "ref" + map_id + ".png"
   self.ref_url="/tmp/" + ref_name

image=map.draw()

if pdp.need_reference==True:
    ref = map.drawReferenceMap()
    ref.save(DefaultConfig.img_path + ref_name)
```

The scalebar and legend object are created the same way as the reference image. It should be noted that if a query is executed then mapObj's function *draw()* must be substituted with *drawQuery()* in order to render a map with the query results highlighted. The member variables, image_url, ref_url, leg_url, scalebar_url depicting the map, the reference image, the legend and the scalebar respectively, are returned using AJAX responses along with the *map_id*. Map_id will be used in the next invocation of the application, in order to delete the images created in the current one which will be, of course, updated.

5.9.3 Managing the layer status and position before rendering

When the map is initialized using the parser, the map rendering will be executed using the mapfile and the parser body directives. In terms of layer display, currently the parser body contains no relative directives, thus leaving these information only to be managed through the mapfile. In the mapfile, the status of the layer can be flagged with the values ON,OFF and DEFAULT under the directive STATUS. The DEFAULT turns the layer on permanently while the other two value define the initial status. Additionally, the layer order definition in the mapfile defines the drawing layer order, with the first layer defined, being drawn in the bottom of rendering queue. These mapfile directives define the initial corresponding member variables' values of the mapObj. The following flowchart [5.11] depicts the events that take place in order to compute the layer status within the renderer while considering the user input. The first time the maprenderer is executed, it uses the default layer status defined in the mapfile and outputs a listing of the layers along with the appropriate controls where the user can submit manual changes. In the next invocation of the MapRenderer, the HTTP request object will contain the new layer status. We would note that if the user turns all the layers off,

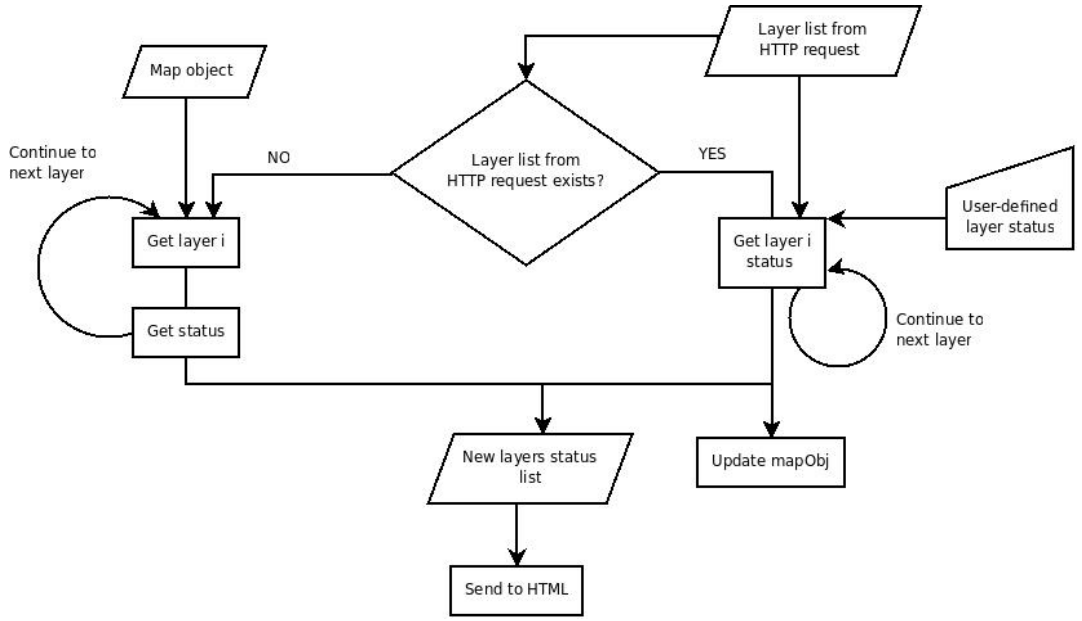


FIGURE 5.11: Setting the layer status.

then automatically the status is set to the mapfile defaults.
Using the map object, the layers' status can be accessed as below:

```
for elem in range(mapobj.getLayerOrder()):
    layerobj = mapobj.get(elem)
    print layerobj.status
```

The member variable layerobj.status is mutable and can be set to e.g mapscript.MS_ON, for the layer to be rendered.

The last element that needs to be described in this section is the layer ordering, which defines the order in which the layers are drawn in the image. Access to the drawing order within mapscript is implemented through the function getLayersDrawingOrder() and setLayersDrawingOrder() respectively [12].When the map is initialized, the layer order is accessed from the map object and is included in the HTML page as an HTML HIDDEN field. Later invocations of MapRenderer read the HTTP Request element produced by this hidden field, thus the application state, as far as the layer ordering is concerned, is maintained. In addition, the user can select to move a layer up or down the layer hierarchy, thus resulting in a different rendering, with the layer that was moved to be shown in the appropriate ordering position. Below we present in pseudo-code the steps used to calculate layer ordering:

[12]Up till Mapserver version 2.10 the documented function getLayersDrawingOrder didn't include the proper SWIG typemap, thus access to these functions returned an unusable SWIG pointer to an integer array. Thus the typemaps were implemented unofficially.

```
# Read the state maintenance Hidden field
if HTTP_REQUEST contains layer_order:
    layer_order = LAYER_ORDER_FROM_HTTP_REQUEST(layer_order)
    # Convert list to python tuple
    mapObj.setLayerOrder(tuple(layer_order))

# Check if the layer is checked to be moved
if HTTP_REQUEST contains "moveup":
    layer= INTEGER_HTTP_REQUEST("moveup")
    resCode = mapObj.moveLayerUp(layer)
if HTTP_REQUEST contains "movedown":
    layer= INTEGER_HTTP_REQUEST("movedown")
    resCode = map.moveLayerDown(layer)
# Get the resulting drawing order
self.orderList = map.getLayerOrder()
# Wrap it in a string to send to the HTML page
#  for application maintenance
self.orderList = "_".join(["%d" % (i) for i in self.orderList])
```

5.9.4 Defining the extent and navigation tools

The most important user controls over a digital map are the panning and the zoom tool. The pan tool moves the center of the map to the specified click point while the zoom tools enlarges the map in a specified scale. Practically all these operations on a digital map alter only one element, and this is the displayed *map extent*. Map extent stands for the rectangle defined by the lower left and upper right geographical coordinates which define the geographic region displayed in the rendered map. Thus, a pan action to the left alters the horizontal part of those coordinates and a 2x zoom action to the center of the map divides the coordinate pairs by 2. Also, observing the mapfile we can see that the only directive existing that can alter the scale of the map is the EXTENT directive which can be accessible through mapscript too.
In Mapscript, the extent is defined as a rectangle object with the member variables minx,miny,maxx,maxy. Thus, the state of the web application in terms of the rendering scale can be maintained by exporting the current extent to HTML and returning it to the next invocation along with the user control options. The basic navigation controls on a GeoMoin map are:

- Panning by mouse click on the map

- Zooming in or out in the map by defining the zoom size
- Zooming in by drawing a rectange on the image
- Recenter at a specified geographic point by defining the geographic coordinates
- Pan using the reference map
- Zoom to a rectangle defined using the mouse drag action

Before any user action is performed, the renderer must set the extent of the map to match the one contained in the HTTP request (if it exists). This can be accomplished by the following lines:

```
if parms.getfirst('extent'):
    extent = parms.getfirst('extent').split(' ')
    try:
       map.setExtent(float(extent[0]),float(extent[1]),
                     float(extent[2]),float(extent[3]))
    except:
       map.setExtent(max_extent.minx, max_extent.miny,
                     max_extent.maxx, max_extent.maxy)
```

If the extent is somehow malformed, it returns to the original one defined in the mapfile.

When a mouse click is issued within the rendered image, an appropriate javascript function is executed that computes the exact pixel coordinates of the click. The coordinates are included in the AJAX call that will execute the new invocation of the MapRenderer. Mapscript defines a very useful function called *zoomPoint(int zoomfactor, pointObj imgpoint, int width, int height, rectObj extent, rectObj maxextent)*. This function zooms by *zoomfactor* to *imgpoint* in pixel units within the image of *height* and *width* dimensions and georeferenced *extent* while zooming can be constrained to a maximum *maxextent*. One important thing to note about this function is that there is no need to geo–transform the image click coordinates to real coordinates, having to deal with the different map projections and the distortions of such convertions. Also, as max extent, the extent defined in the mapfile is set. One last thing to note is that the zoomfactor is set to 1; the function will produce as a panning result.
So the first two actions listed above can be accomplished by using the zoomPoint function with a zoomfactor of 1 and greater than 1 respectively. The zooming to the center of the map is done by dividing the image width and height by 2 and using the results as the click coordinates. For a recentering at a specified geographical coordinate we must take care of the following issues:

- The geographical coordinates issued must be unprojected using the lat/lon coordinate pairs, thus a conversion to the current map projection must take place.
- Finally, a conversion between the geographical coordinates and image coordinates must take place.

In order to convert coordinate pairs between projections, *OSGeo's osr* python module must be used. The following lines do this conversion:

```
\label{proj2lat}
        from osgeo import osr
        # We create the source and destination projections object
        src = osr.SpatialReference()
        dst = osr.SpatialReference()
        # The input projection is the user issued
        # one and is using the EPSG 4326 (latlon)
        src.ImportFromEPSG(4326)

        # Below we define a sample projection in
        # order to transform the coordinate pairs.
        # The projection is defined in the proj4 format
        projection="+proj=laea +ellps=clrk66 +lat_0=45 +lon_0=-100
                          +a=6370997 +b=6370997 +units=m"
        # Set the projection definition to initialize the object
        dst.ImportFromProj4(projection)

        coords=(-104,41)
        # Now we must check if the projections are the same thing
        #if dst.IsSame(src):
        if dst.IsSameGeogCS(src):
            return coords
        else:
            # We instantiate an osr Coordinate Transformation object
            transformer = osr.CoordinateTransformation(src,dst)
            result=transformer.TransformPoint(coords[0],coords[1])
            print (result[0],result[1])
```

Of course, a geographic reprojection can be accomplished if the input coordinate pair is residing in the region defined by the output projection. For example, the LAEA

projection in the above script, defines the area of USA, thus a latlon pair of (63,45) would have no meaning and if a result is returned, the distortion would be enormous. Having the reprojected coordinates, they must be converted to image coordinates. For this issue we use the following function:

```
def map2img (width, height, x, y, ext):
   ppd_x=0  # pixel per degrees on horizontal axis
   ppd_y=0  # pixel per degrees on vertical axis
   new_x=0  # The horizontal image coordinate
   new_y=0  # The vertical image coordinate

   # The horiz pixel per degree is the image width
   #  divided by the horizontal extent
   ppd_x = width / (ext.maxx - ext.minx)
   ppd_y = height / (ext.maxy - ext.miny)

   # (pixel/degrees) * degrees --> pixels
   new_x=ppd_x * (x - ext.minx)
   # Same idea as above, but the vertical pixel position
   # is counted from the top of the image
   new_y=height - ppd_y * (y - ext.miny)
   return (new_x,new_y)
```

Of course, the above calculations introduce a degree of distortion when the scale denominator of the map is set to a high value. Thus, in smaller extents the resulting coordinates are more accurate.
After the image coordinates are calculated, the map object's *zoomPoint* can be used to zoom or recenter the map at the specified geographic coordinate pair. Finally, we should note that in case of a zoom using the drawing of a rubber-band box on the image, a special javascript is engages which follows the mouse MouseDown, MouseUp, MouseMove javascript events. After the mouse button is released the image coordinates of the rubber–band box are translated to geographic coordinates and the new extent is used to draw a new instance of the map.

The last navigation method to discuss is the reference image panning. While the map is at a particular zoom level, a mouse click on the image map transfers the viewpoint of the new rendered image to a new extent defined by the previous zoom size and the mouse click coordinates in pixels. Please note that in general, a reference image is much smaller than the digital map it generalizes. In detail the computations are the following:

- If the mouse click was on the rendered map, which would be the coordinates in analogy to the reference image click coordinates? This is a percentage factor implemented as:

```
clkpoint.x = map.width * clkpoint.x / reference_image.width
clkpoint.y = map.height * clkpoint.y / reference_image.height
```

- Currently the center coordinates of the rendered map can be computed as (map.width/2, map.height/2). These coordinates must be translated in analogy to the original map extent. Firstly, the image coordinates of the current extent are calculated:

```
(imgxmin_curext,imgymin_curext) = utils.map2img(map.width,map.height,
                                map.extent.minx,map.extent.miny,max_extent)
(imgxmax_curext,imgymax_curext) = utils.map2img(map.width,map.height,
                                map.extent.maxx,map.extent.maxy,max_extent)
```

  And the center is:

```
imgxcenter_curext=(imgxmax_curext+imgxmin_curext)/2
imgycenter_curext=(imgymin_curext+imgymax_curext)/2
```

- Now a percentage of the difference between the requested click point and the above image point in comparison to the image size must be calculated. A positive *percx* means that the click was on the West, while a positive *percy* means that the click was on the South, because image pixels count for UL corner of the image.

```
percx = (clkpoint.x - imgxcenter_curext)/map.width
percy = (clkpoint.y - imgycenter_curext)/map.height
```

- The new extents are calculated based on the above percentages (the same percentage generated above using the pixel coordinates, applies to geographic coordinates too):

```
map.extent.minx = map.extent.minx + (percx*max_extent.maxx)*2
map.extent.miny = map.extent.miny - (percy*max_extent.maxy)*2
map.extent.maxx = map.extent.maxx + (percx*max_extent.maxx)*2
map.extent.maxy = map.extent.maxy - (percy*max_extent.maxy)*2
```

- Check if the rectangle defined by the new extent, is contained within the rectangle specified by the maximum extent and in that case, reset the extents to the valid values defined by the maximum extent.

5.9.5 Query capabilities

In addition to its rendering capabilities, Mapserver provides a powerful query facility, supporting both spatial queries (which select features based on their geographic location) and attribute queries (which selects features based on attribute values). The cgi version of Mapserver supports the *query* mode where the querying capabilities are enabled. In the case of GeoMoin we are interested in the Mapscript API for issuing the query commands. The particular API provides many types of query functions which can be used in conjuction to each other, in order to implement more complex queries. Before a layer is queried, the user creating the mapfile must ensure that for every layer, to produce results, a *TEMPLATE* directive must be included in its definition or, alternatively, in a class definition of the layer. The second approach, sets as queriable, only the layer features represented by the particular class definition. This method of turning a layer queriable can be identified as a Mapserver "quirk" and relies on the mapserver mode as a CGI, containing templating mechanisms and is important for backwards compatibility. In the next Mapserver version, hopefully, a more specific directive could turn on/off the query facilities for a layer. Additionally, each queriable layer must define two directives named *TOLERANCE* and *TOLERANCEUNITS*. The first defines the sensitivity for point and line based queries, meaning that a feature is returned if a spatial query is *at most* within *TOLERANCE* units far from the feature. The *TOLERANCEUNITS* directive defines the unit types for the previous value and it can be one the following types: pixels, feet, inches, kilometers, meters, miles, dd (decimal degrees) and defaults to pixels. For example, a layer definition containing the above requirements could be:

```
LAYER
  NAME "cities"
  DATA "\${default_attach_path}1214071817.69.30350/cities.shp"
  STATUS ON
  TYPE point
  PROJECTION
      "+init=epsg:4326"
  END
  TOLERANCE 10.0
  TOLERANCEUNITS miles

  CLASS
      TEMPLATE "ttt"
      NAME "cities"
```

```
        STYLE
            SYMBOL "Circle"
            SIZE 5
            COLOR 234 25 14
        END
    END
END
```

It is obvious that the TEMPLATE directive does not need to contain a valid url.

The mapscript objects that can be queried are both the layers and map itself, meaning that both objects contain specific query functions. Using the map object querying functions the following queries can be defined:

- Return the layers' features that intersect a geographic point within a tolerance buffer
- Return the layers' features that intersect a particular polygon
- Return the layers' features that intersect the results of a previous result set on a polygon layer
- Return the layers' features that intersect a given feature

Alternatively, using the layer object, provides access to a richer selection of queries

- Return the layer's features that intersect a geographic point within a tolerance buffer
- Return the layer's features that intersect a particular rectangle
- Return the layer's features that intersect the results of a previous result set on a polygon layer
- Return the layer's features that intersect a given feature of type polygon
- Return a layer feature at the particular index
- Return the layer's features matching the results of a query on their non–spatial fields

These basic query types, in Mapscript, are supported by specific functions which can be combined to built more specific and complicated queries.

In GeoMoin those functions integrated resulted in the following query types:

Query by point Returns specific layers' or all the layers' features that contain a geographic point within a tolerance buffer. For each layer single or multiple results can be displayed. Different tolerances for every layer can be specified. The query is issued by a mouse click.

Query by item Query a specific layer using a query string on its non–spatial attributes. For each layer, single or multiple results can be displayed.

Query by feature Return specific layers' or all the layers' features that intersect a polygon based on the previous point query result of the polygon layer that produced this result. For each layer, single or multiple results can be displayed.

Query by itemfeature Return specific layers' or all the layers' features that intersect a polygon based on the previous item query results of the polygon layer that produced this result. For each layer, single or multiple results can be displayed.

Query by real coordinates Return specific layers' or all the layers' features that intersect a rectangle specified by its geographic lower left and upper right coordinates.

Query by image coordinates Return specific layers' or all the layers' features that intersect a rectangle specified by its image upper left and lower right coordinates.

Of course, it is a relatively simple task to add new query cappabilities to GeoMoin by enhancing the above or creating new. In that case it is important to address some issues concerning the Mapscript query API. The most often–used function is *queryByPoint(pointObj point, int mode, float buffer)*. This function can either be accessible through a map object resulting in the querying of all the layers or through the layer object resulting in the querying of the particular layer. *mode* can be set to *MS_SINGLE* or *MS_MULTIPLE* resulting in single or multiple results for every layer. The last two parameter: *buffer* and *point* are worth mentioning.

With *buffer*, the function refers to the radius of an imaginery circle, centered by the geographic coordinates of the query point, within which, the query produces results. The diagram [5.12] below describes the interaction between the TOLERANCE value mentioned above and the buffer parameter. It is obvious that if the result margin is bigger that zero then the feature is returned. Attention must be particularly paid to

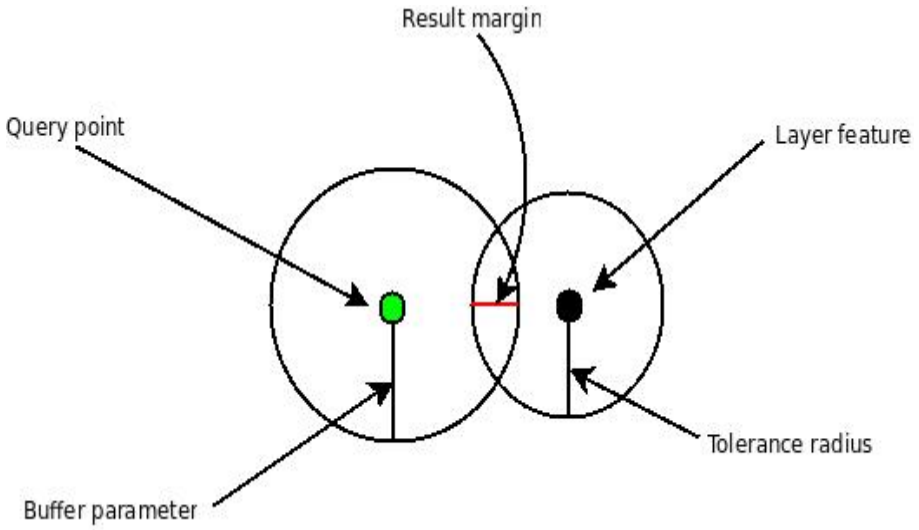


FIGURE 5.12: Interaction between the tolerance value and the buffer parameter.

the correspondance of the units that the buffer and the tolerance values are expressed; while tolerance units type is set by the TOLERANCEUNITS directive described earlier, the buffer parameter of the query functions (in mapscript) was observed that is not internaly converted to the type specified by the mapfile definition and it defaults to decimal degrees. This results in a comparison between decimal degrees and the type specified by TOLERANCEUNITS. Thus a conversion must be made to turn the user–specified buffer to the type defined by TOLERANCEUNITS. This is done manually using the following function:

```
def compute_tolerance(tolerance,layer):
    if layer.toleranceunits==mapscript.MS_MILES:
        return tolerance/69.04   # Convert DD to miles with distortion
    elif layer.toleranceunits==mapscript.MS_FEET:
        return tolerance/69.04*5280
    elif layer.toleranceunits==mapscript.MS_INCHES:
        return tolerance/69.04*63360
    elif layer.toleranceunits==mapscript.MS_KILOMETERS:
        return tolerance/69.04*1.609344
    elif layer.toleranceunits==mapscript.MS_METERS:
        return tolerance/69.04*1609.344
    # Below the dpp function calculates the decimal degrees
    #  per pixel considering the map image size and the current extent
    elif layer.toleranceunits==mapscript.MS_PIXELS:
        return toler*dpp(map.width,map.height,map.extent)
```

```
    elif layer.toleranceunits==mapscript.MS_DD:
        return tolerance
```

From the above, it is assumed that the user should set the *buffer* to an appropriate value considering the units set by the TOLERANCEUNITS directive. Below we will discuss the different GeoMoin query type indepedently.

Query by POINT

The point query performs the simplest spatial query: a point query. The image coordinates of a mouse click in the rendered image are returned to the renderer. These coordinates are used to search the specified layers one after another until Mapserver finds the feature within the specified tolerance from the click point. For every layer queried, single or multiple results can be returned containing the non–spatial attributes of every features involved. The mapscript function perfomed for this query is based on the layer object and its usage is illustrated in the following pseudo–algorithm:

```
for all queried layers selected by user
     # Mode can be MS_SINGLE or MS_MULTIPLE
     # buffer is the tolerance specified by the user for
     #    the particular layer involved
     layer.queryByPoint( pointObj point, int mode, float buffer )
     layer.open()
     result = layer.get_Results()
     output_Results(result)
```

It could be possible to use the map object's queryByPoint function with the drawback of setting a global tolerance buffer for all the layers involved in the query.

Query by ITEM

This kind of query tells Mapserver to check the attribute table of a user–selected layer to match a user–defined query string. Let us assume that a layer named *Countries* contains a non–spatial fieldname by the name *Countryname*. The user can select the particular layer, set as *Query Attribute* the word *Countryname* and as *Query String* the word *Germany*. The result set will contain the feature corresponding to the country of Germany. The Query String follows the same syntax as the mapfile FILTER directive and can also be a Regular Expression like "*/Ger./*".It is important to outline that for shapefiles and connections through OGR, queries like the one above are valid, but for connections through SDE,OracleSpatial and PostGIS, the query string is only consulted by Mapserver and should be a native SQL WHERE clause. A pseudocode of this interactions with mapscript:

```
for all queried layers selected by user
     # Mode can be MS_SINGLE or MS_MULTIPLE to produce
     #   single or multiple results
     layer.queryByAttributes(mapObj,Query Attribute,Query String,mode))
     layer.open()
     result = layer.getResults()
     outputResults(result)
```

Query by FEATURE

Feature query performs a spatial query that uses a feature from one layer, the *slayer* to query another layer. In Mapserver only polygon layers can be selected as an *slayer*. The whole operation is issued by a mouse click to the *slayer* from which the polygon geometry selected is extracted. The result set is internally saved by mapserver, so issuing the actual *queryByFeature(slayer)* on the map object, the internal result set is consulted and a query is being made to all the queriable layers. Every layer having features that intersect the polygon will be returned. A pseudocode of this interaction with mapscript:

```
slayer.queryByPoint(mapObj map, pointObj point, MS_SINGLE))
map.queryByFeatures(slayer)
for all layers selected by user
   layer.open()
   result = layer.getResults()
   outputResults(result)
```

It is important to note in case of overlapping polygons within the same layer, the first one resulting is selected to continue the query. Additionally in the current Mapscript version if the mode is set to MS_MULTIPLE resulting in more that one polygon, the queryByFeatures function breaks the stability of the application and is currently reported as a bug.

Query by ITEMFEATURE

The idea of the ITEMFEATURE query combines the logic of the FEATURE and the ITEM query. The attribute part of the operation is a non–spatial query which results in the polygon geometry. This polygon will be compared against all the features of all the layers in order to check for intersection. The features intersecting are returned. A pseudocode of this interaction with mapscript:

```
slayer.queryByAttributes(mapObj,Query Attribute,Query String,MS_SINGLE))
map.queryByFeatures(slayer)
```

```
for all layers selected by user
   layer.open()
   result = layer.getResults()
   outputResults(result)
```

Query by Real Rectangle

Using this query, a user can define a rectangle based on lower left and upper right *georeferenced* coordinates and submit it, ordering Mapserver to return all the layers' features that interesect it. Mapscript offers the very useful queryByRect function on both the map and the layer level. In GeoMoin the map's function is used in order to query all the layers.

Query by Image Rectangle

Using this query, the user can draw a rubber–band box on top of the image using Javascript whose coordinates will be transalted to the desired query bounding box, ordering Mapserver to return all the layers' features that interesect it. Mapscript does not natively support rectangle queries based on image coordinates, thus georeferenced coordinates must be manually exported. In a previous section, we defined the *map2img* function which converts an image coordinate pair to a georeferenced one. The following function named *img2map* convert georeferenced cordinate pairs to image ones:

```
def img2map (width, height, x, y, ext):
  dpp_x = 0
  dpp_y = 0
  dpp_x = (ext.maxx-ext.minx)/width  # degrees per pixel
  dpp_y = (ext.maxy-ext.miny)/height
  x = minx + dpp_x*x          # degrees from left
  y = maxy - dpp_y*y          # degrees from top because
                              # pixels count down from top
  return (x, y)
```

After the coordinate pairs are calculated, the query is issued using the *queryByRect* function on the map object level.

An important issue concerning the queries in mapscript, is the optical representation of the query result; in addition to the queries being returned in a text form (return of the computed information is accomplished through AJAX responces), the user is introduced to a graphical output of the query results. This is achieved using a different rendering function than the one discussed in 5.9.2, the *mapObj.drawQuery()* function. This function does completely the same job as the *mapObj.draw()* while highlighting

the features included in the result set. The highlighting color can be set in the mapfile within the mapfile under the object named *QUERYMAP*.

5.9.6 Annotating a rendered map

One of the most important tools that Geomoin offers, is the ability to link pairs of coordinates with annotations containing information about the particular geographic feature that they describes. Through the use of annotations, the users can, in loose terms, create their own spatial datasets or contribute to already created ones by adding or editing particular features. That is, an annotation is nothing more or nothing less than a dynamically generated dataset that has its unique methods of displaying the contents included in it. GeoMoin supports an unlimited number of annotation datasets with each of them having its unique characteristics along with a unique role of existence.

The annotation information, provided by the user, are stored in special PostgreSQL tables updated to include a spatial PostGIS column. Using the LayerManager [5.7], a user can create a brand new annotation table defining an unlimited number of non–spatial columns, enough to fully describe a geographic feature. Up till now, the supported types of columns in an annotation table can be:

- text
- integer
- float
- date

In order to present an example of an annotation table, let's assume that the user created it using the LayerManager. In the "postgis" registration table the addition could look like 5.13

The *type* field is setting the table type to be managed as annotation. The *viewable* field

id	name	owner	viewable	editable	type	description	wikipage	creation_date
5	hydrogp020	1214071817	TRUE	TRUE	POSTGIS	Not Defined	datasets	2008-10-03
7	gr	1214071817	TRUE	TRUE	POSTGIS	Not Defined	datasets	2008-10-03
9	goodies	1214071817	TRUE	FALSE	ANNOTATION	Restaurants in Chania	Goodies	2008-10-12

FIGURE 5.13: Registration of an annotation table.

describes if the table can be viewable by all the users of GeoMoin, while the *editable*

disallows users from inserting or managing annotation, unless they are the particular annotation or the table owners (*owner* field contains the user id of the table owner). Additionally, a description exists, describing the contents of the annotation table[13], while a wikipage will act as a link to a wiki page that fully describes the dataset.

5.9.6.1 Enabling an annotation layer in the MapRenderer

Adding an annotation layer to a mapfile follows the same principles like any other postgis–enabled database table; the user can select it in the LayerManager in order to create a layer definition in his markfile, and afterwards use the MapManager to import the layer in a map. Thus a mapfile layer definition of the above annotation can be [14]:

```
LAYER
  NAME "goodies"
  CONNECTIONTYPE POSTGIS
  CONNECTION "host=\$db_host dbname=\$db_name
              user=\$db_user password=\$db_pass port=\$db_port"
  DATA "wkb_geometry FROM \${db_schema_exp}goodies"
  STATUS ON
  TYPE point

  TOLERANCE 10.0
  TOLERANCEUNITS miles

  CLASS
      TEMPLATE "ttt"
      NAME "goodies"
      STYLE
          SYMBOL "Pin"
          SIZE 26
          OFFSET 13 -13
      END
  END
END
```

[13]It is preferable that the users creating annotation tables, provide a complete and accurate description of the contents

[14]It is interesting to note that the querying capabilities are independent of the layer's type as annotation and can be used as described in the previous section.

After the registration is inserted and the annotation layer definition is added to the mapfile, it is ready to be included in a map rendering. In order the layer, to be treated as an annotation layer, it must be additionally defined in the renderer's parser body, otherwise, **the annotation controls will not be available to the user** the layer will be displayed as a typical PostGIS layer. This is done by adding the line 05 while editing the MoinMoin wiki page containing the parser:

```
01   {{{
02      #!renderer ,
03      mapfile=demo2.map
04      owner=user1
05      annotation=goodies    # Include this line
        .
        .
     }}}
```

5.9.6.2 Annotation table structure

Before explaining an annotation table structure, the sample contents of the "goodies" layer are presented below: The columns named: *id, owner, editable and wkb_geometry*

id [PK]	owner text	editable boolean	address text	phone text	open text	wkb_geometry geometry
15	1214071817	FALSE	Elm street 14	30-234-23-13	12:00 to 24:00	0101000020E610
16	1214071817	TRUE	Arlington Road 2	32-342-23-32	08:00 to 06:00	0101000020E610
17	1214071817	TRUE	Abbey Road 12	950-20-11-88	12:30 to 12:35	0101000020E610

FIGURE 5.14: Contents of an annotation table.

are assumed to be system columns. The first two define the primary key and the userid of the user that created a particular annotation. The editable field is a boolean value that flags whether the tuple can be editable, even if the whole table is marked as non–editable. Figures 5.4, 5.5, 5.6 presented on previous section provide the flowchart of the SQL operations considering the access controls.

The last system column named ***wkb_geometry*** contains the Well–Known–Binary representation of the geographic point or polygon annotated. It is very important to distinguish between these two types of annotation geometries because they are handled very differently by the GeoMoin annotation subsystem.
A GeoMoin user, while creating an annotation layer, can select the geometric type of

its geographic content, whether it is a polygon or a point. The difference between the management of these two types is that when a user (using the MapRenderer) wants to add a new *point* annotation, this annotation is created by the coordinates of the mouse click on the rendered image. On the contrary, creating a polygon annotation, **requires the selection of a pre-existing polygon feature which will act as the annotation geometry source**. For example, in a rendering of a world map containing a polygon layer of the countries' borders, the user can annotate Germany by selecting the appropriate tool and source layer while clicking anywhere within the polygon rendering of the country. Both methods are discussed later.

5.9.6.3 Creating an annotation

In order to create a new GeoMoin annotation, two steps must be considered; first of all, the details of the annotation layer, that will hold the annotation, must be gathered in order to create an HTML form, and secondly, upon filling the required details, the annotation must be stored and rendered. While the second step is common for both point and polygon annotation types, the first one requires different handling.

Every time the MapRenderer is executed, a list of all the annotations included in the parser body is gathered. A check is being made to ensure that every annotation layer included in the parser body exists in the PostgreSQL DBMS and if the mapfile has a definition for that layer. Each element of the list contains a pointer to an object named *AnnotationLayer* which holds as member variables:

- The name of the annotation layer, which should be an exact match of the mapfile layer name and PostGIS name.
- The geometric type of the annotation whether it is **point** or **polygon**. This value is gathered from the mapscript layer object.
- A boolean value flagging whether this annotation table is viewable. This value is gathered using the pgUtils module for the particular database table.
- A boolean value flagging whether this annotation table is editable. This value is gathered using the pgUtils module for the particular database table.
- A boolean value flagging whether this annotation will act as a direct link to a web page or a popup containing the details will be rendered. This value is gathered using the pgUtils module for the particular database table.

Every AnnotationLayer contains the following member functions concerning the insertion of a new annotation:

CreateAnnotationForm This function collects the information that will be used by the user–services tier in order to display a form that handles **point** annotation insertions.

CreatePolygonAnnotationForm This function collects the information that will be used by the user–services tier in order to display a form that handles **polygon** annotation insertions.

AddAnnotation This function is executed when the submission of one of the above forms happens, in order to add the new annotation to PostgreSQL.

In order to create an annotation, the user must select either the point or the polygon annotation tool from the user interface. Additionally, the annotation layer that will hold the user information must be selected in case multiple annotation layers were declared[15]. When the user issues a mouse click on the rendered image of the map, the above information are sent using an HTTP request to the MapRenderer along with the image coordinates of the click point. Upon receiving the input, MapRenderer calls either *CreateAnnotationForm* or *CreatePolygonAnnotationForm* in regard to the geomtry type of the annotation layer selected.
First of all the image coordinates of the mouse click must be converted to geographic coordinates using the function *img2map*. Afterwards, the user field names[16] of the annotation tables are collected using the pgUtils module. These field names will be used by the user–services tier to output the HTML TEXT elements in order to insert the new annotation details.
Finally, the geometry of the annotation feature must be generated; if the annotation layer is of type **point** the following steps must take place in order to create the geometry:

- The geographic coordinates produced by the img2map function are converted to their correspondive latitude/longitude pair using the function ***proj2LATLONG***

```
from osgeo import osr

def proj2LATLONG(projection,x,y):
   """
     @param projection: the proj4 definition of the imput projection
     @param x,y: the coordinates projected using the input projection
```

[15]The polygon annotation tool is only valid for annotation layers of point geometry, while the polygon tool is valid for annotation layers of polygon geometry

[16]As user field names we refer to the PostgreSQL table field names that are managed by the end–users. In the contrary system field names refer to fieldnames which control the annotation in regard to the business logic (eg. “editable” and “owner” are system field names

```
  @rtype: tuple
  @return: the latlon coordinate pair """
# Create the source and destination projections
src = osr.SpatialReference()
dst = osr.SpatialReference()
# The projection paramater is "encoded" in proj4 wkt definition
# thus it is imported using:
src.ImportFromProj4(projection)

# The output projection is the annotations projection which is always
# latlon, thus EPSG 4326
dst.ImportFromEPSG(4326)

# Check if the two projections are the same
if dst.IsSameGeogCS(src):
    return (x,y)
else:
    # A coordinate transformation object is created
    # to handle the reprojection
    trans = osr.CoordinateTransformation(src,dst)
    result=trans.TransformPoint(x,y)
    return (result[0],result[1])
```

- An SQL call of the GeoMoin SQL function **geomoin_wkb** that will create the Well–Known–Binary representation of the coordinates is formulated. It is important to highlight that the SQL function is not executed at this particular step. An example of an SQL call formulation could be:

```
sqlcall = "gis_schema.geomoin_wkb(10.662,47.507,4326)"
```

Below the implementation of the geomoin_wkb function is presented:

```
CREATE OR REPLACE FUNCTION
      -- parameters: latitude, longitude, projection EPSG
      gis_schema.geomoin_wkb(double precision, double precision, integer)
-- Return the well-known-binary representation
RETURNS geometry AS
   $BODY$
   DECLARE
 point public.geometry;
```

```
wkb public.geometry;
  BEGIN
    -- Postgis function that creates a point using a pair of coordinates
    select into point st_makepoint from public.ST_MakePoint(\$1,\$2);
    -- Postgis function to convert a point object to its Well-Known-
    -- Binary representation using the specified projection EPSG
    select into wkb * from public.geomfromwkb(point,\$3);
    return wkb;
  END;
  $BODY$
  LANGUAGE 'plpgsql' VOLATILE
  ALTER FUNCTION gis_schema.geomoin_wkb OWNER TO postgres;
```

As long as the above information are collected, they are formulated into an XML document and are sent to the user–services tier as a response to the AJAX call of MapRenderer.

If the annotation layer is of **polygon** type, the generation of the geometry is very different. In addition to the user selection of the polygon annotation layer that will hold the information, the user should also select a polygon layer that will act as the source of the polygon geometry.

- The geographic coordinates produced by the img2map function are converted to their correspondive latitude/longitude pair using the function ***proj2LATLONG*** exactly as in point annotations.
- The source layer selected by the user is instantiated using mapscipt:

```
layer = map.getLayerByName(sourceLayer)
if layer.type != mapscript.MS_LAYER_POLYGON:
   return (False,"Only polygon source layers allowed")
```

- A mapscript point object is instantiated and a *point query* is executed at the particular source layer:

```
point = mapscript.pointObj(clickpoint[0],clickpoint[1])
# One result is permitted thus MS_SINGLE is parameterized
# Tolerance is 0
resCode = layer.queryByPoint(map, point, mapscript.MS_SINGLE, 0)
if resCode == mapscript.MS_FAILURE:
   return (False,"No polygon was returned. Is the layer queryable?")
```

- Afterwards, the query results must be opened and the geometry features must be extracted. It is very common, that a polygon layer in MapServer can contain features that are viewed as multipolygon in the representations of other GIS packages. Thus a point query in the area of Philippines can result in a single **mapscript** feature object which contains multiple mapscript line objects (closed lines) represent the Philippines individual islands. Thus in MapRenderer, all the line objects contained in the features are serially scanned and the particular closed line that **contains** the click coordinates is returned:

```
resultSet=layer.getResults()
layer.open()
# the MS_SINGLE parameter result in only one feature
result = resultSet.getResult(0)
shapeobj = layer.getFeature(result.shapeindex, result.tileindex)

# If the real geometry is multipolygon, we must check in which lineObj
#  the click point is contained
for i in range(shapeobj.numlines):
   # Create a temporary shape object
   tempshape = mapscript.shapeObj(mapscript.MS_SHAPE_POLYGON)
   # Get the line object of the source shape
   lineobj = feature.get(i)
   tempshape.add(lineobj)
   # If the click point is contained
   if tempshape.contains(point):
       ...
       ...
```

- As soon as the mapscript line object that contains the click point is tracked, the Well–Known–Text representation of the geometry can be returned using mapscript's *shapeObj.toWKT()*. The polygon shape contained in the WKT can consist of four coordinates pairs representing a simple rectangle, up to an unlimited number of coordinate pairs defining high resolution polygons. In later section, it is mentioned that a polygon annotation is presented in the map rendering as an HTML AREA. Thus in case of a large number of coordinate pairs, a simplification must take place so that the HTML output is not overloaded with a huge number of image coordinate pairs per polygon annotation. Simplification stands for sets of operations that reduce the number of vertices of a polygon feature preserving as much as possible vertices that make the resulting polygon resemble the original.

PostGIS extension contains a function called **simplify** [17] that takes as input a Well–Known–Binary representation and a simplification factor. A possible value for this simplification factor can be the **map units per pixel** multiplied by 2 or 3.

```
simplify_factor = 2 * utils.dpp(map.width,map.height,map.extent)
```

Finally, the GeoMoin SQL function *geomoin_simplify* is executed that takes as input the **WKT** of the geometry and the simplification factor. The function stores the simplified **WKB** is a temporary PostgreSQL table returning the primary key of the insert value to be used later. A sample SQL call to this function could be:

```
SELECT id FROM gis_schema.geomoin_simplify(
    TEXT('POLYGON ((39.155 -6.586, 39.167 -6.61, ... ))') , 1.117875
)
```

The implementation of the *geomoin_simplify* is presented below:

```
CREATE OR REPLACE FUNCTION gis_schema.geomoin_simplify(text, double precision)
  -- returns the primary key of the temporary table
  RETURNS bigint AS
$BODY$
DECLARE
id bigint;
wkb public.geometry;
wkbsimpl public.geometry;
BEGIN
    -- Postgis function. Create geometry from WKT and particular EPSG
    select into wkb * from public.geomfromtext($1,4326);
    -- Postgis function. Simplify a geometry by a factor
    select into wkbsimpl * from public.simplify(wkb,$2);
    -- Insert simplified data in the temporary table
    insert into gis_schema.geomtemp(wkb_geometry) values (wkbsimpl);
    -- Get the current value of the table primary key sequence
    select into id * from currval('gis_schema.geomtemp_id_seq');
    return id;
END;

$BODY$
```

[17] The PostGIS simplify function is using the Douglas Peuker algorithm for simplifying polygons.

- Finally, an SQL call of the GeoMoin SQL function **geomoin_gettempgeom** is made that will read the temporary table's tuple defined by the above id. It is important to highlight that the SQL function is not executed up to this particular step. The execution will actually happen when the insertion of the annotation will be made in a later stage (later invocation of MapRenderer).

Just like in the case of a point annotation, the information collected above are formulated into an XML document and are sent to the user–services tier as a response to the AJAX call of MapRenderer and the HTML form in rendered.

When the user submits the required information after filling the user field names of the annotation, MapRenderer is re–executed and handles the annotation insertion using function *AddAnnotation* of the particular AnnotationLayer object. This function gathers the field names of the annotation table and searches the HTTP Request object to get the correspondive values. The SQL call created above that will return the geometry of the annotation is also included in the HTTP request. From these information a SQL INSERT statement is formulated using the GeoMoin's pgUtils *module*. A sample SQL INSERT statement for adding a point annotation could be:

```
INSERT INTO
   gis_schema.pointannot (name,surname,wkb_geometry,editable,owner)
VALUES
   (john, doe, gis_schema.geomoin_wkb(-41.393, 21.781, 4326), False, user1)
```

5.9.6.4 Displaying and managing annotations

Upon each execution of the MapRenderer, the annotation layers included in the parser body must be opened and their information must be returned to the user–services tier in order to create the popup that contain the annotation details. The triggering of the popup view is done via a mouse click inside the HTML AREA that surrounds the geographic region or point annotated. Using the popup view the user can view, update or delete a particular annotation provided that he has the appropriate access controls over the annotation table or the tuple holding the annotation. In order to view an annotation, MapRenderer should return only the values contained in the user field names of every annotation in the tables. In order to update an annotation, MapRenderer should return the above values which will be contained in specific HTML TEXT controls along with the **primary key** of each annotation tuple. Finally. deleting an annotation requires only the primary key of the annotation. For every update or delete action, MapRenderer will be re–invocated and will call the appropriate handling functions discussed later.

In order to collect the information required for user–services to output the annotation details, every AnnotationLayer object has the following member functions:

CreateTTimagemap This function opens the point annotation layer and gathers the features. MarkSpot function is called for each feature.

CreateTTimagemapPolygon This function opens the polygon annotation layer and gathers the features. MarkSpot function is called for each feature.

MarkSpot This function opens a particular feature and collects the required information in order to view/manage/delete the annotation specified by the particular feature.

In *CreateTTimagemap* and *CreateTTimagemapPolygon* functions, the following steps must take place:

1. First of all, the user field names of the annotation table are gathered by a call to pgUtils function *GetFieldNames*.

2. A mapscript layer object must be instantiated using the layer name in order to get the features. The layer is afterwards opened and the STYLE object is also instantiated.

```
layer=map.getLayerByName(layername)
layer.open()
# Get the first class of the layer, then
# get the first style and the size
stylesize =(layer.getClass(0)).getStyle(0).size
```

3. An appropriate handling must take place in order to avoid annotations outside the currently displayed extent to be rendered. This is achieved using the mapscript function ***whichShapes***. As an input to this function a mapscript rectangle object must be provided which in this case is the specific extent of the map. There exists a particular implication where in case the map is projected using a particular projection while the source layer is projected using a different projection, the correct result are not returned. This possibly happens because the extent provided in whichShapes is not reprojected to the projection defined by the source layer. Thus this must be done manually using the following code:

```
(minx,miny)=utils.proj2LATLONG(map.getProjection(),
                               map.extent.minx, map.extent.miny)
```

```
(maxx,maxy)=utils.proj2LATLONG(map.getProjection(),
                        map.extent.maxx, map.extent.maxy)
newextent=mapscript.rectObj(minx,miny,maxx,maxy)
status = layer.whichShapes(newextent)
```

4. Next, each shape returned is accessed and the non–spatial fields are included in a list named "values" that will be provided to the MarkSpot function discussed later. Furthermore, for the reason that GeoMoin support only point and polygon annotation, each shape object contains only one line object which can be accessed:

```
while (1):
  # Get the shape object
  shape = layer.nextShape()
  if shape==None:  break

  values=[]
  for i in range(shape.numvalues):
     values.append(shape.getValue(i))

  lineobj=shape.get(0)
```

From this point, each line object returned can contain either a point or a polygon. Using the geometry of each annotation the HTML AREA pixel coordinates must be generated. For point annotations where a single coordinate pair define the position of the annotation, *CreateTTimagemap* function continues the procedure:

1. A mapscript point contained in the line object is returned:

```
hotSpot = lineobj.get(0)
```

2. The point object returned above, must be reprojected from latlon to the map projection in order to be useful:

```
newspot = utils.LATLONG2proj(map.getProjection(),
                             hotSpot.x,hotSpot.y)
```

3. The pixel coordinates that correspond to the particular point in regard to the current extent must be calculated:

```
position=utils.map2img(map.width, map.height,
                        newspot.x, newspot.y, map.extent)
```

4. Particularly for point annotations in GeoMoin, the HTML AREA coordinates generated are consisting the upper–left and bottom–right pixel coordinates of the rectangle which has as a center the point annotation coordinates and as a vertice size, the size of the mapscript style object divided by two. The horizontal and the vertical coordinates are stored in the lists names "hor" and "ver" in order to be used by the function MarkSpot:

```
x=position[0]
y=position[1]

newsize=stylesize/2
# xm and ym: coords of the UL pixel
# xp and yp: coords of the LR pixel
xm=x-newsize
if (xm<0):  xm=0
ym=y-newsize
if (ym<0):  ym=0
xp=x+newsize
if (xp>map.width):  xp=map.width
yp=y+newsize
if (yp>map.height): yp=map.height

hor = [xm,xp]
ver = [ym,yp]
```

5. Finally, the MarkSpot function is executed, parameterized with the field names of the annotation table, the values for the field name and the horizontal and vertical list created above:

```
MarkSpot(values,fieldnames,hor,ver)
```

   Some additional parameters are also included in the call, but do not worth mentioning and they are documented in the source code.

If the annotation layer is of type *polygon* the following steps must take place:

1. The point objects contained in the line object define the vertices of the polygon geometry and will be used to produce the polygonal HTML AREA that will trigger the annotation:

```
for i in range(lineobj.numpoints):
    point=lineobj.get(i)
```

2. As in point layers, the point object must be reprojected to the map projection and converted to pixel coordinates in regard to the current extent:

```
newpoint=utils.LATLONG2proj(map.getProjection(),hotSpot.x,hotSpot.y)
position=utils.map2img(map.width,map.height,
                       newpoint.x,newpoint.y,map.extent)
```

3. The image coordinates are appended to the horizontal and vertical lists as above.

4. Finally the MarkSpot function is executed as above.

The MarkSpot function is responsible for creating the XML part for each of the annotations. The elements included in the XML output of each annotation:

- The values of the user field names of the feature. In many cases the contents are parsed and a resulting value is finally returned. For example, if the value is "PHOTO:world.jpg", the result is a HTML IMG element pointing to the image included in the wiki page attachements
- The above values are also include, unparsed.
- The editability of the feature. Locks editing and deleting functions.
- The primary key of the annotation tuple within the annotation table.
- The coordinates that were calculated by the *CreateTTimagemap* and *CreateTTimagemapPolygon* functions.

A screenshot of the resulting HTML popup rendering is shown below:

In case the user selects the update or delete actions the MapRenderer is re–invocated and the function *UpAnnotation* or *DelAnnotation* is called respectively. Both function read the HTTP request; in case of updating the annotation the user field names of the annotation layer updated are requested along with the integer primary key that is used to locate the annotation tuple in the PostgreSQL annotation table, whereas in case of deleting an annotation only the primary key in need. Both functions use the PgUtils module to formulate SQL UPDATE and DELETE queries in order to manage the requested annotations. A sample SQL UPDATE statement could be:

```
UPDATE gis_schema.polyannot SET
       description='A description',number=32 where id=16
```

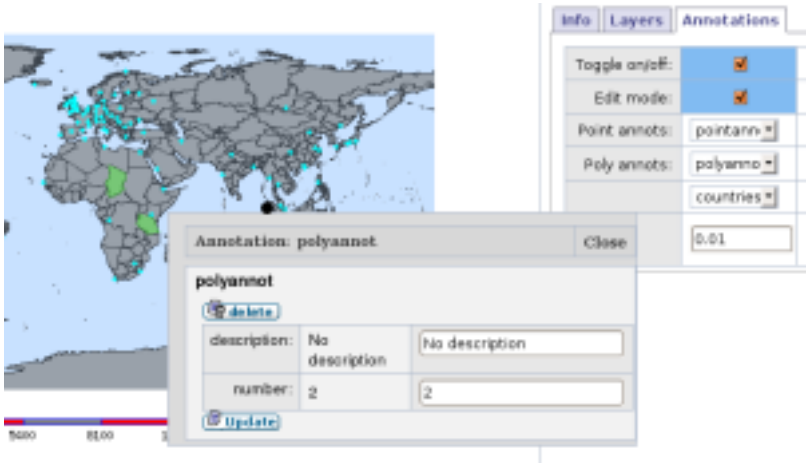


FIGURE 5.15: Sample annotation.

5.9.7 Using XML-RPC to manage the WIKI content

In chapter 2 we described the main sequence of the events that are happening within the MoinMoin application, when a request from a user is issued. In order to interconnect the different modules and provide a maintenance of the application state, the MoinMoin request object exists. This object is important in order to access the wiki through the MoinMoin API, in terms of editing pages, uploading attachments, getting link structures e.t.c. When the MapRenderer is called using the AJAX interface, variables derived from the parser are passed in form of an HTTP POST request, while the MoinMoin request object is impossible to be transfered to the MapRenderer in order to control various aspects of the wiki instance. Thus the only clean and documented solution to this problem is to use MoinMoin's ability to access the wiki instance through the use of XML–RPC calls in terms of WikiRPC described in Chapter 2.
When MapRenderer is called through AJAX, an authentication token of the particular session is formulated and passed within the HTTP request. MapRenderer, using this token can authenticate to MoinMoin **within the context of the session that created the particular token**. Thus, using this token, the following abstracted code can upload an attachment to a wiki page using XML–RPC:

```
token = ReadHTTPRequest("token")
rpcwiki = xmlrpclib.ServerProxy("http://localhost/wiki?action=xmlrpc2")
mcall = MultiCall(rpcwiki)
mcall.applyAuthToken(token)
mcall.putAttachment(pagename,attachmentname,base64object)
results = mcall()
```

Not only MapRenderer, but every application called through AJAX, in order to manage the wiki, can be provided with an authentication token. Using MoinMoin's multicall function, various XMLRPC calls can be formulated, assigned to a particular session token and finally executed using the mcall() function. Member functions of the multicall object are either the native MoinMoin wikirpc functions, or the functions created by the user. GeoMoin defines its own method to be included in the multicall object. Native MoinMoin wikirpc function include the function proposed in the WikiRPC specification along with custom MoinMoin functions.

5.10 GeoMoin mapplets

In many occasions, GeoMoin's built–in functionalities may be considered limited, compared to the needs of a particular use case of the software. For example, a GeoMoin administrator could need the ability to incorporate geocoding functionallity to the interactive navigation. In particular, he would like to allow the users input an address and return the correspondive results, allowing them to zoom to a road listed within the result set. Manipulating the source code of the application could add such a functionality, but it can become cumbersome as far as future updates of the application must be incorporated. Thus, a mechanism of creating user–specific add–on was created and these add–ons are called ***mapplets***.
Mapplets, strictly speaking, are structured python functions that are executed in certain places within the GeoMoin flowchart. GeoMoin mapplets are created and installed by the administrators of a GeoMoin installation and not from users (unlike GoogleMaps) because they give access to the complete python interface including system and spatial libraries. Thus, careful design must take place in order to prohibit unauthorized access to information held in the server's filesystem.
In order to discuss how the mapplets fit in the event flowchart, the two types of mapplets are presented along with their correspondive examples that will be developed for demonstration later in the section.

5.10.1 Synchronous mapplets

Synchronous mapplets are mapplets that directly alter the business logic of the Spatial Visualisation System, and particularly the rendering process of a map. With the term *synchronous* we refer to a situation where the mapplet is presented as an HTML FORM and upon submission of the information an AJAX call to the MapRenderer is being made, thus a new rendering is computed according to the previous information. A flowchart of these operations is presented in figure 5.16. As it is shown in the diagram, when the

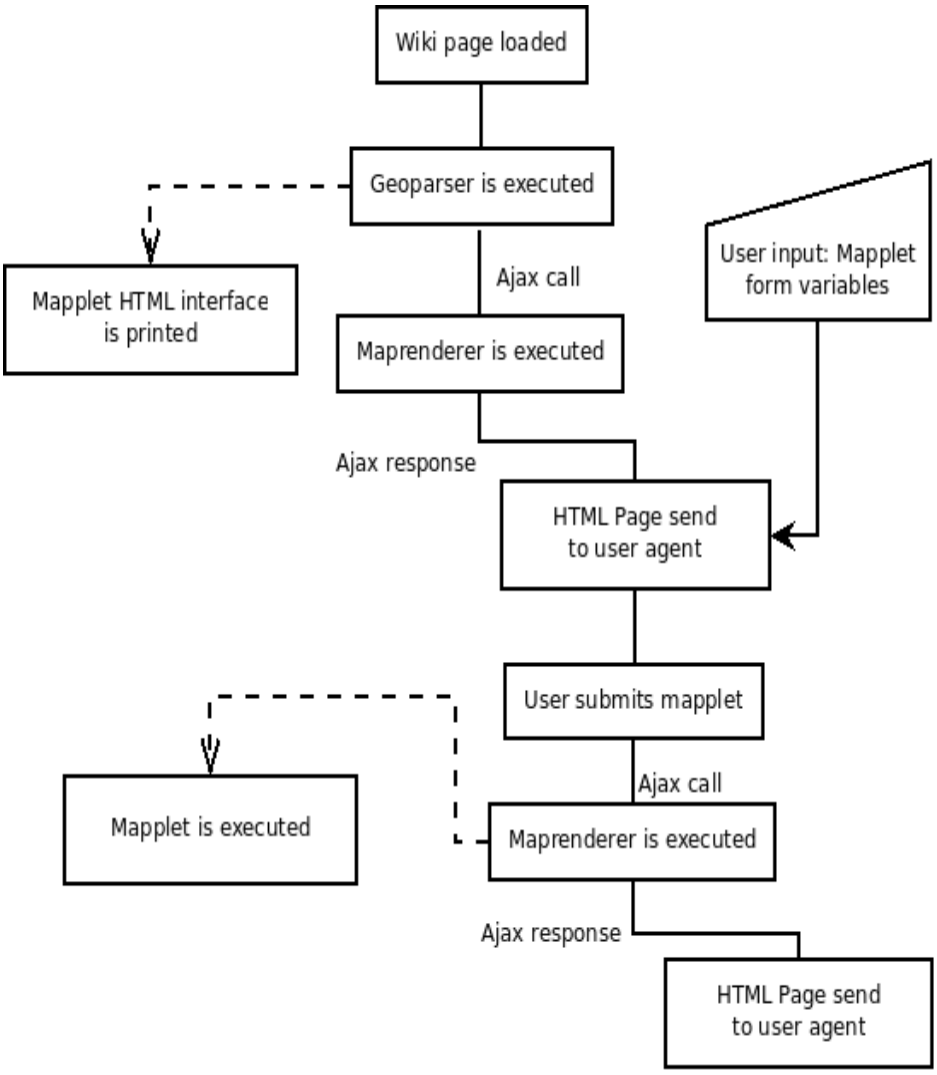


FIGURE 5.16: Synchronous mapplet flowchart.

MoinMoin parser *geoparser* is executed, the mapplet's HTML interface is generated by calling the suitable function. Likewise, when the maprenderer is executed, the mapplet is executed too, altering the rendering procedure.
As a demonstration mapplet we will create a mapplet that displays a Tissot indicatrix [18] over the current map rendering.

In order to faciliate the user in creating a mapplet, a mapplet template is included that contains the parameters to be changes and the function declarations to be implemented.

[18]Tissots indicatrix, or ellipse of distortion, is a concept developed by French mathematician Nicolas Auguste Tissot, in 1859 and 1871, to measure and illustrate distortions due to map projection. It is the theoretical figure that results from the orthogonal projection of an infinitesimal circle with unit radius, defined in a geometric model of the Earth (a sphere or an ellipsoid), on the projection plane. Tissot proved that this figure is normally an ellipse, whose axes indicate the two principal directions of the projection at a certain point, i.e., the directions along which its scale is maximum and minimum. When the Tissots indicatrix reduces to a circle it means that, at that particular point, the scale is independent of direction.

We will run through the step by step creation of the mapplet from the template. First of all the module import part is presented:

```
############ START IMMUTABLE #############
# -*- coding: iso-8859-1 -*-
import mapscript
import urllib
from MoinMoin.GeoBase.config import DefaultConfig
############ END IMMUTABLE   #############
from osgeo import ogr,osr
import os
```

Additional modules can be included after the *immutable* tag, in our case the OGR module is included. Afterwards, the following paramaters must be filled:

```
### Name of the mapplet
mappletname = "tissot"
description = "This mapplet displays a Tissot Indicatrix over the current rendering."
### Remote AJAX application to run
ajaxscript = "maprender.py"
### AJAX function to handle the call
ajaxfunction = "ajaxRunMapplet"
### Mapplet HTML names needed for execution
inputvars = []
```

The **mappletname** defines the name of the mapplet, which must be kept as short and simple as possible, as it defines the name of the HTML elements that will be sent using the HTTP request. The **description** is self–explanatory. Using **ajaxscript**, the remote script that will be executed using an AJAX call is declared. In *synchronous mapplets* this script is **always** the MapRenderer addressed by the *maprender.py* module. The **ajaxfunction** parameter defines the javascript function that implements the call to the above script (defined via *ajaxscript*). Again, in synchronous mapplets the function is built–in the GeoMoin javascript codebank and called *ajaxRunMapplet*. For customizing purposes, different AJAX functions can be implemented. Finally, the **inputvars** contains the names of the HTML controls that are used to execute the mapplet (in this example, no controls are defined as a simple button will engage the Tissot indicatrix rendering).
Following, the templated functions for the output of the interface and the execution of the mapplet are presented:

```
############ START IMMUTABLE #############
def formvar(name):
    return  mappletname + "_" + name

def execute(map,httpreq,options):
    inputdict={}
    for elem in inputvars:
        if utils.existHttp(httpreq,formvar(elem))==False:
            return "Please provide a %s"%elem
        inputdict[elem] = httpreq.getfirst(formvar(elem))
    (code,message) = userexecute(map,options,inputdict)
    return (code,message)

def output(parms):
    #ajaxscript = "geocoderAJAX.py"
    ajaxapp = "%s/%s"%(DefaultConfig.geomoincgi,ajaxscript)
    ajaxcall = "%s('%s','%s','%s')\""%(ajaxfunction,ajaxapp,parms,mappletname)
    output =  "<form name=\"%s\" method=\"POST\">"%mappletname
    output+="<fieldset><legend style=\"background-color:#E7E7E7;\" >
             <b>Description</b></legend>%s</fieldset>"%description
    output+=useroutput(ajaxcall)
    output+="<hr>"
    output+='<div name="%s" id="%s"></div>'%(formvar("results"),formvar("results"))
    output+="</form>"
    return output
############ END IMMUTABLE   #############
```

It the **output** function, the ajax call is formed along with the HTML form that will be send to the user. The **useroutput** function is user–implements and contains the mapplet specific HTML controls. The DIV TAG before the form closing is used to host the results of the mapplet interactions.
Moving to the **execute** function, first of all, the information contained in the HTTP request object must be gathered and included in a python dictionary. The *inputvars* list contains the names of the variables to be checked (of course they must agree with the HTML controls created in the custom output function). As stated above, the mapplet execution is done within the maprenderer execution thus it gives access to the mapscript map object and to other variables that drive the rendering procedure. In the execute function the map object is named **map** and the miscellaneous variables are in form of a python dictionary which is populated within the maprenderer and is called **options**.

Along with the **inputvars** dictionary, the user can create a custom execute function that has unlimited options over the rendering procedure. It is important to note that currently the **options** dictionary contains:

max_extent The maximum extent, or initial extent of the map.

userid The user's unique id.

layerlist The list of the layers that was computed. Each element of the list is a list itself containing information about every layer.

Finally, the custom functions for the output of the interface and the execution of the mapplet must be created. Respectively they are called *useroutput* and *userexecute*. The useroutput function below:

```
def useroutput(ajaxcall):
    output=""
    output+="<br> <input type=\"button\" name=\"%s\" value=\"Enable\"
                        onClick=\"%s\">"%(formvar("enable"),ajaxcall)
    output+="<input type=\"hidden\" name=\"MAPPLET_COMMAND\"
                        value=\"%s\">"%mappletname
    return output
```

This example has two HTML controls, first is a button that upon mouse click engages the Javascript call that will make the ajax call to the remote script. Secondly, an essential HIDDEN variable with the mapplet name as value, named MAPPLET_COMMAND, must always be included.
The **userexecute** is the most important function of the mapplet as it defines its business logic. In the current example, the computation of the Tissot Indicatrix will be presented as pseudocode:

```
def userexecute(map,options,inputdict):
    userid = options["userid"]
    path = DefaultConfig.temp_path+"tissot_"+userid+".shp"
    # Now we must grab the projection. If no map level
    # projection is specified, assume it is latlong
    projection=""
    try:
        projection = map.getProjection()
    except:
```

```
        projection = "+init=epsg:4326"
        pass

    % Open OGR driver for ESRI Shapefiles and create shapefile
    % Create a polygon layer within the datasource
    % Use OSR to create spatial reference object
         for latlong and map projection
    % Use FOR to create a logical grid over the latlong extent
           -180 to 180 and -90 to 90, step by 30
    % Create a OGR point feature to every x,y grid point
    % Reproject point using orthogonal projection
    % Buffer it by 390 kilometers
    % Reproject to latlong
    % Save ESRI shapefile
    % Create a new MapScript map layer and assign the datasource
    % Define a style for printing the polygon contents
    return (True,"OK")
```

Later in chapter, where the Spatial Visualization System GUI is presented, the interactions between this sample mapplet and the rendering are presented in screen captures.

5.10.2 Asynchronous mapplets

In many occasions, a particular idea for a mapplet may require a preprocessing that is usually time–consuming or (requires) multiple user interactions, before the map rendering of the mapplet–driver process is executed. The idea behind asynchronous mapplets is using AJAX ability to make asynchronous calls to remote scripts, independently of any other AJAX call used to navigate through the map. Suppose we want to create a mapplet where the user inputs a name of a city or address and as a final result we require a map rendering that contains an graphical arrow pointing to this place. Analysing the requirements, it is important that a database of real–life world locations exists so that it can be queried. These databases are called **Geocoders** and exist as services in the Internet provided by different vendors like Google, Yahoo, Microsoft etc. The fact that we depend on external resources in order to gather information, introduces time dependencies that are not known before–hand. Thus a mechanism should allow connections to these archives independently, allowing the user to navigate through the map or do any other processing (like executing another mapplet), without having to wait for the remote server to respond with the results.
When the results are ready, they must be presented to user, which chooses the one

matching his requirements. Afterwards, with a click of a button, the rendering process is re–invocated and the result is presented in the map image as an arrow. It is clear enough that the last requirement is closely reminding of a synchronous mapplet. That is, asynchronous mapplets are mapplets that incorporate an asynchronous AJAX call to a remote script in order to do a pre–processing and, finally, a call to the **maprenderer** (like every synchronous mapplet) in order to execute the rendering logic of the mapplet within the Spatial Visualization System.

The flowchart of the operations described above is shown in figure 5.17.

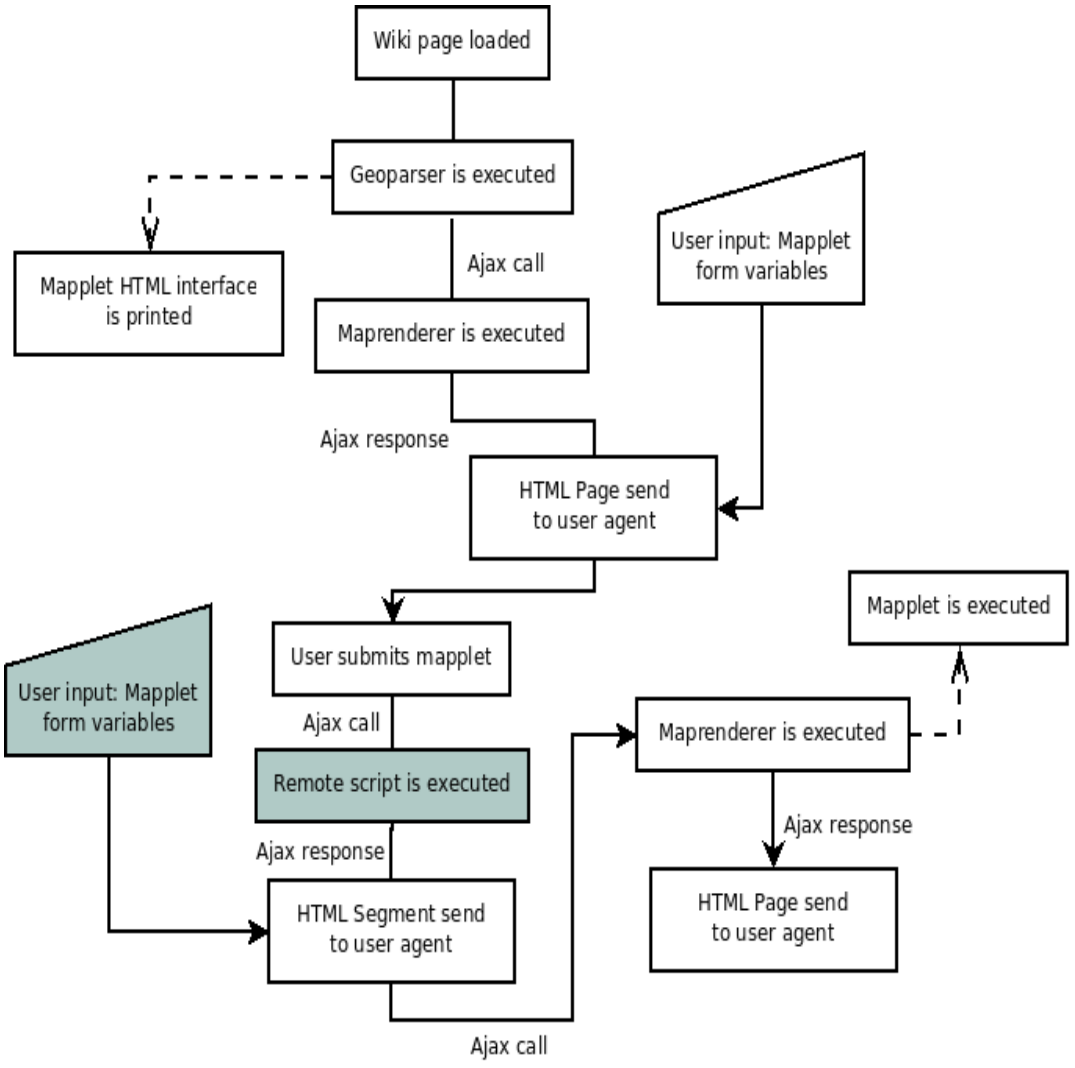


FIGURE 5.17: Asynchronous mapplet flowchart.

Below, the same template as synchronous mapplets is used to create the first building block of the Geocoder mapplet.

```
### Name of the mapplet
```

```
mappletname = "geocoder"
description = "This mapplet connects to a remote geocoder,
               queries with a given string and returns locations."
### Remote AJAX application to run
ajaxscript = "geocoderAJAX.py"
### AJAX function to handle the call
ajaxfunction = "geoCoder"
### Mapplet HTML names needed for execution
inputvars = ["coordinates"]
```

The name and description of the mapplet are found in the first two line of the parameter section of the template. This time, the script that is going to be executed is not the maprender.py but custom built *geocoderAJAX.py* which is described later in the section. Subsequently, the javascript function the creates the AJAX call is a custom built one and is named *geoCoder*. After the *geocoderAJAX.py* and the user selects the appropriate location to draw the arrow, a HTML FORM element containing the coordinates of the location is included in the HTTP Request. This variable will be used by the **userexecute** function (described in Synchronous mapplets) to include the arrow in the map rednering process.
The **useroutput** function is used to create the initial interface of the mapplet (and is execute when the MoinMoin parser is engaged) is presented below:

```
def useroutput(ajaxcall):
    output=""
    output+= "<br>Location:"
    output+= '<input type="text" name="%s"
                     style="width:30%%">'%(formvar("location"))
    output+= '<input type=button name="%s" value="Search"
                                  onClick="%s"><br>'%(formvar("submit"),ajaxcall)
    output+= '''<br>Geocoder:
                     <select name="%s">
                       <option value="Google">Google</option>
                       <option value="Yahoo">Yahoo</option>
                       <option value="GeoNames">GeoNames</option>
                       <option value="VirtualEarth">VirtualEarth</option>
                       <option value="GeocoderDotUS">GeocoderDotUS</option>
                       <option value="Wikipedia" DISABLED>Wikipedia</option>
                     </select>
             '''%(formvar("geoserver"))
```

```
    return output
```

The first form element is named **location** and is a textbox where the user inputs the name of the location to be geocoded. Secondly, a button the engages the Javascript call is included. Finally, using the SELECT box, the user can choose between different online geocoders to query for the particular location.
When the javascript call is made, *geocoderAJAX.py* is executed and the results are presented to the user using additional HTML form elements. These interactions will be addressed later. Supposing the user chooses a location matching his criteria, upon submission, *maprender.py* will be executed, which will call the **userexecute** function of the mapplet. This function draws the arrow in the map rendering and is presented below (containing pseudocode to hide unnecessary details):

```
def userexecute(map,options,inputdict):
    point = inputdict["coordinates"].split("_")
    zoom = 1
    res=utils.LATLONG2proj(map.getProjection(),eval(point[0]),eval(point[1]))
    % Create a new mapscript point layer object in the map
    % Create a class and a style for this layer
    % Set a particular symbol depicting an arrow
    % Create a point feature and add the coordinates
    % Assign this feature to the layer
    % Get the image coordinates from the geographic coordinates
    % Zoom to the particular image coordinates
    return (True,"OK")
```

The remote script used to geocode the location submitted using the initial mapplet form resembles the templated script for creating the basic mapplet. The first part of the script is presented below:

```
#!/usr/bin/env python
# -*- coding: iso-8859-1 -*-
############ START IMMUTABLE #############
import cgi,time
import cgitb; cgitb.enable(display=0, logdir="/tmp")
from MoinMoin.GeoBase.config import DefaultConfig
############ END IMMUTABLE   #############

from MoinMoin.GeoBase import utils
```

```
from geopy import geocoders

# HTML variables from the initial mapplet interface
inputvars = ["location","geoserver"]
```

The first line allows the module to run as a script. After the module imports, the user–defined import are declared, including **geopy**, a open–source library for accessing various geocoders using Python. Finally, the **inputvars** define the names of the variables which are expected to be included in HTTP Request.
The following templated part of the script contains some functions needed to maintain consistency between different mapplets:

```
############ START IMMUTABLE #############
def setOutputXML(output):
    print "Content-type: text/xml"
    print ""
    print "<?xml version=\"1.0\"?>"
    print output

def setOutputText(output):
    print "Content-type: text/plain"
    print ""
    print output

def AJAXexecute():
    # AJAX script to run
    ajaxscript = "%s/maprender.py"%DefaultConfig.geomoincgi
    # The event is used to submit the form using AJAX
    ajaxcall = "ajaxRunMapplet('%s','%s','%s')\""%(ajaxscript,"{includeparms}",mapple
    inputdict={}
    for elem in inputvars:
        if utils.existHttp(httpreq,mappletname+"_"+elem)==False:
            return "Please provide a %s"%elem
        inputdict[elem] = httpreq.getfirst(mappletname+"_"+elem)
    result = userAJAXexecute(ajaxcall,inputdict)
    return result

def formvar(name):
    return  mappletname + "_" + name
```

```
httpreq = cgi.FieldStorage()
mappletname = httpreq.getfirst("mappletname")
results = AJAXexecute()
############ END IMMUTABLE   #############
# The output can be XML or TEXT
setOutputText(results)
```

The **setOutputXML** is used in order that the script returns an XML document to the Javascript function that called it. This xml document can be accessed in Javascript using the XMLHttpRequest().responseXML object.
The **setOutputText** is used in order that the script returns a plaintext document to the Javascript function that called it. This plaintext document can be accessed in Javascript using the XMLHttpRequest().responseText object.
The **AJAXexecute** defines that after the remote execution of the script has taken place and the results are presented to the user, the *maprenderer.py* should be invocated using an AJAX call in order to "fit" the results within the rendering procedure. All the variables contained in the *inputvars* list are read from the HTTP request and a call to the custom–built **userAJAXexecute** function is taking place.
The final line is executed as soon as the script is invoked. A call to AJAXexecute is being made and the results can be wrapped either in XML or Text representation, based on which function is called. For our example, text output is send back which contains the HTML code that will be produced from the **userAJAXexecute** presented below:

```
def userAJAXexecute(ajaxcall,inputdict):
        execOut=""
        geoserver = inputdict["geoserver"]
        location = buffer(inputdict["location"])
        if geoserver=="Google":
            g = geocoders.Google(api_key=DefaultConfig.g_api)
        % Do the analogous task for all the geocoders available
results = g.geocode(str(location),exactly_one=False)
        % Create an HTML table
        for place, (lng, lat) in results:
           % Add a row to the table containing the place and coordinates
           % Add a radio button and set as value the string created by
                     combining the latitude and the longitude
        % Close the HTML table
        execOut = '<input type=button name="%s" value="Zoom" onClick="%s">'
```

```
                %(formvar("submit"),ajaxcall)
    execOut = '<input type="hidden" name="MAPPLET_COMMAND" value="%s">'
                %(mappletname)
    return execOut
```

The Javascript function that is handling the AJAX call will check the AJAX response object and create an HTML form including the HTML code from *userAJAXexecute* above. When the user inputs the required information (in this example, select the geocoded address to be shown in the map), an AJAX call to the MapRenderer will take place and the rendering procedure will include the instructions calling the function **userexecute** of the first script we analysed.
A careful reader would point out that many different AJAX calls can be incorporated before the control returns to the main mapplet script. Thus mapplets can become as complicated as possible. But it is recommended that a simple execution path is maintained until a better mapplet API would be built. It is visualized that mapplet generation would be achieved through a user friendly graphical interface with different mapplet components that can be incorporated to create new advanced mapplets in a very standardized and effective way.

5.11 User services and AJAX

In GeoMoin, the user services tier, is the logic, defined to act as an intermediate between the end–user and the business logic of the system. Architecturally, the definition of the user services, along with their level of abstraction from the other lower layers, is clear and straight–forward; upon user request and within the context of the currently viewed application state, certain functions can be executed in order to process and modify information. As soon as the results of the involved operations are generated, the user must be informed of these results and the new application state.
Implementing the above specification, involves the adoptation of a web–design methodology; The fact that GeoMoin is a web mapping application, implies that the execution time for rendering or querying large vector or raster datasets, does not allow the implementation of the user services using the classic HTML form methodology; reloading the complete web–page upon user interaction is both time consuming and annoying to the end–user.

In order to address the issue of partially reloading specific elements of the page, throughout the Spatial Visualization System, the *AJAX technology* is being used. **Ajax** (Asynchronous JavaScript and XML) is a group of interrelated web development techniques used for creating interactive web applications or rich Internet applications. With Ajax, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page. Data is retrieved using the XMLHttpRequest object or through the use of Remote Scripting in browsers that do not support it. Despite the name, the use of XML, or its asynchronous use is not required.

The term Ajax has come to represent a broad group of web technologies that can be used to implement a web application that communicates with a server in the background, without interfering with the current state of the page. In the article that coined the term Ajax, Jesse James Garrett explained that it refers specifically to these technologies:

- XHTML and CSS for presentation
- the Document Object Model for dynamic display of and interaction with data
- XML and XSLT for the interchange and manipulation of data, respectively
- the XMLHttpRequest object for asynchronous communication
- JavaScript to bring these technologies together

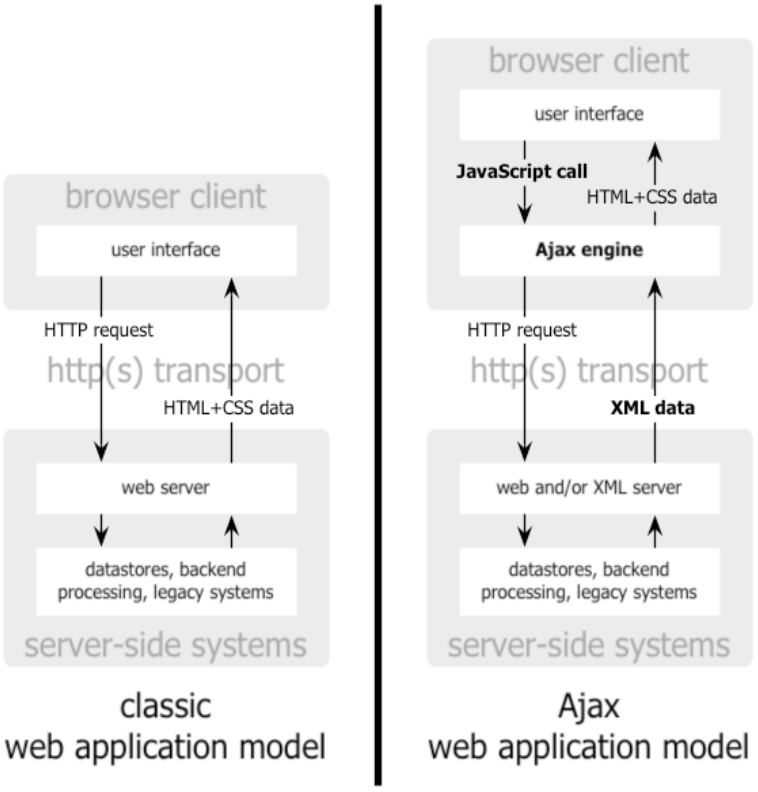


FIGURE 5.18: Classic Web Application Model versus AJAX Model.

Since then, however, there have been a number of developments in the technologies used in an Ajax application, and the definition of the term Ajax. In particular, it has been noted that:

- JavaScript is not the only client-side scripting language that can be used for implementing an Ajax application. Other languages such as VBScript are also capable of the required functionality.
- XML is not required for data interchange and therefore XSLT is not required for the manipulation of data. JavaScript Object Notation (JSON) is often used as an alternative format for data interchange, although other formats such as preformatted HTML or plain text can also be used.

JavaScript is the scripting language that nearly all browsers support, which will fetch data behind the scenes from the server, and XML is the popular language that can be used to store data in an easy format. Heres an overview of how Ajax works:

1. The user interacts with the web page by submitting an action.

2. The particular javascript function associated with the action submmited is called. The function creates the XMLHttpRequest object and uses it to send the HTTP request to the server script or application.

3. Asynchronous data retrieval of the default ajax implementation allows the above two steps to happen multiple times before any results are returned

4. As soon as the XMLHttpRequest object returns a final state (e.g. 200 means Successfull), data in form of XML or plain text can be exported from the object. These information are the server script's responses to HTTP request that was posted.

5. Using the Document Object Model of the initial HTML web page, partial content can be updated using the information gathered above.

The AJAX technology, is influencing a growing rate of web sites and particularly web sites that are implementing complex kinds of applications (e.g GoogleMaps, PassPack). In order to faciliate the development process of such state–of–the–art web design, various packages called **Ajax Frameworks** have emerged. These frameworks provide implementations of AJAX methodologies that can be used with moderate or, in some cases, non at all, interaction with the AJAX specific functions; that is, the developer can be primarily concerned with the web application design. Such frameworks include and are not limited to packages like Backbase, Dojo Toolkit, Ext, JQuery.
The GeoMoin application is depended and developed using the MoinMoin wiki as a Web Development Framework. MoinMoin, currently, does not support or provide AJAX functionality to applications developed within it. Moreover, the fact that MoinMoin follows a form–based design strategy, does not allow the overuse of custom AJAX functionality; a uniformity of web–design practice must exist, in order not to confuse the end–user. Thus, as far as, the GeoMoin business logic components, they are provided to the user services as such:

MapManager The subsystem that manages the mapfiles is following a form–based web design strategy.

LayerManager The subsystem that manages the installed spatial datasets is following a form–based web design strategy.

InstallManager The subsystem that installs spatial datasets is following a form–based web design strategy.

Spatial Visualization System The main component of the GeoMoin web application is using AJAX. Moderate to high rendering execution times, should be handled with proper web interaction between the system and the user.

The following sections, deal with the above subsystems in terms of the interaction with the end–user. The business logic behind the subsystem was extensively desribed in previous section while the source code of the application in properly comment to act as a reference.

5.11.1 MapManager

When the wiki page containing the MapManager is loaded, the user is presented with a list of the maps he attributed to the system. [5.19].

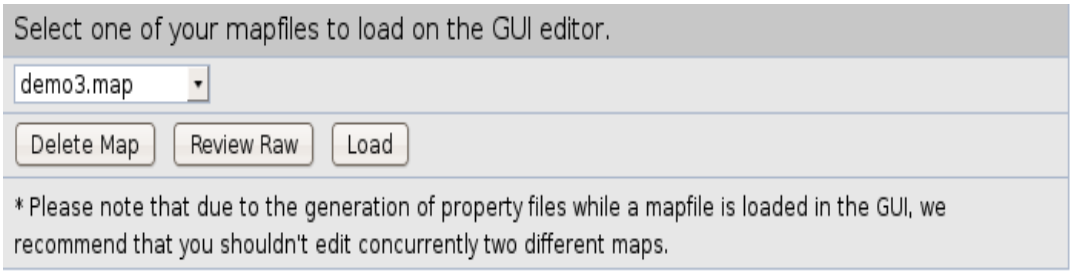


FIGURE 5.19: Panel for map selection.

The user has can perform the following actions:

- Delete a map.
- Manage the map using a text box. [5.20]
- Load the map to the graphical editor. The user can view the map level directives of a map 5.21 and a list of the layers included [5.22]

The layers that were generated using the LayerManager, are presented to the user right below the mapfile layers using the same method and controls. Using the GUI editor the user has access to controls in order to:

- Modify[19] the map settings.
- Modify the web, legend, reference, scalebar.
- Create and modify symbols and output formats.
- Modify the layers' parameter.
- Insert inline features to a layers.

[19]Modifying, includes the action of deleting an element.

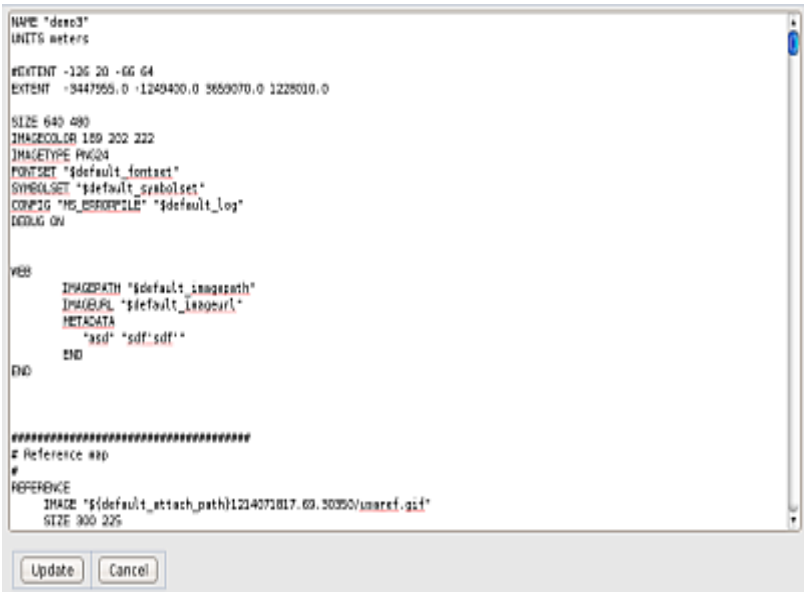


FIGURE 5.20: Editing a raw version of the map.

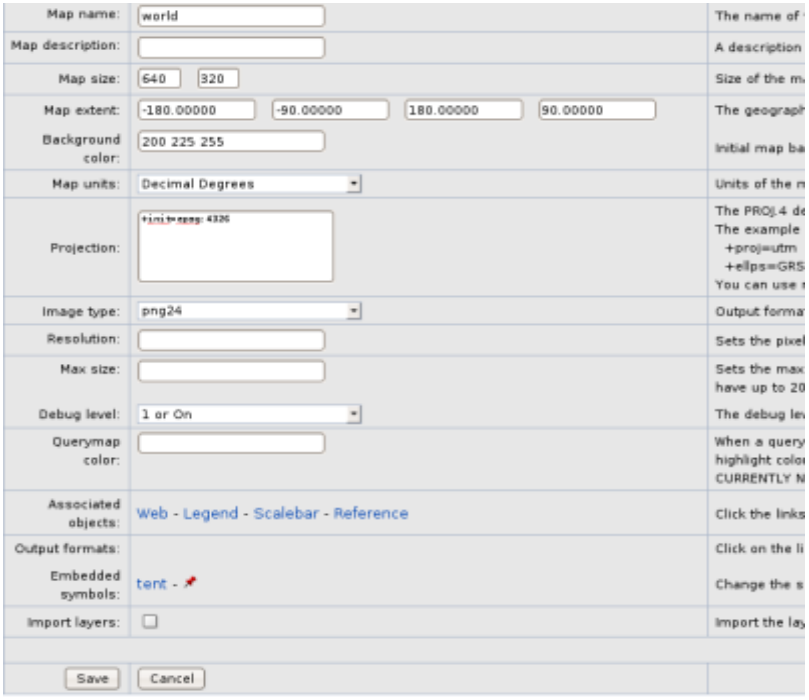


FIGURE 5.21: Editing using the gui editor.

FIGURE 5.22: The layers contained in the map.

- Modify and create new classes for a layer.
- Modify and create new labels and styles for a layer.
- Add the layers that were generated from the LayerManager to the current map.
- Save the resulting map.

In order to display information about particular elements, GeoMoin makes extensive use of the javascript popup library *overlibMWS*. For example, using this library, the user can access the WEB element's options as shown in figure 5.23. Of course, using the

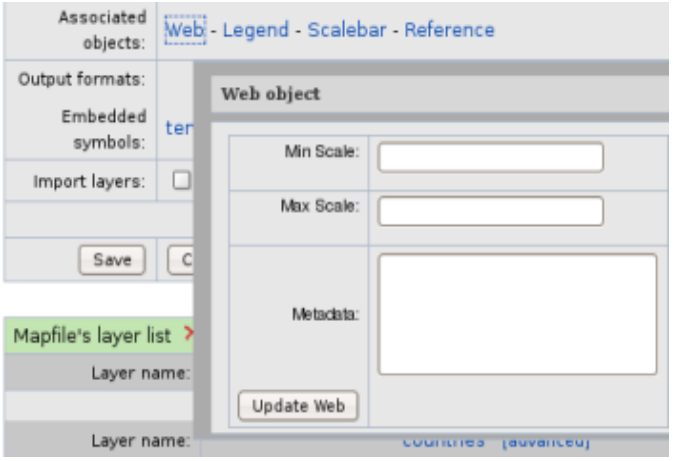


FIGURE 5.23: Popup showing web element.

MapManager, the user can create a new map; the layers for the map can be obtained using the LayerManager, while all the options and controls are the same as in the event of editing a map.

5.11.2 LayerManager

The LayerManager provides a centralised place where GeoMoin datasets can accessed in order to extract information or import the to mapfiles. Additinally, annotation layers can be created and OGC OWS services can be contacted.

5.11.2.1 Querying and extracting dataset information

As soon as the macro implement the LayerManager is executed by MoinMoin, search panel [5.24] is displayed to the user in order to query the datasets installed in the system. Using the panel displayed above the user can:

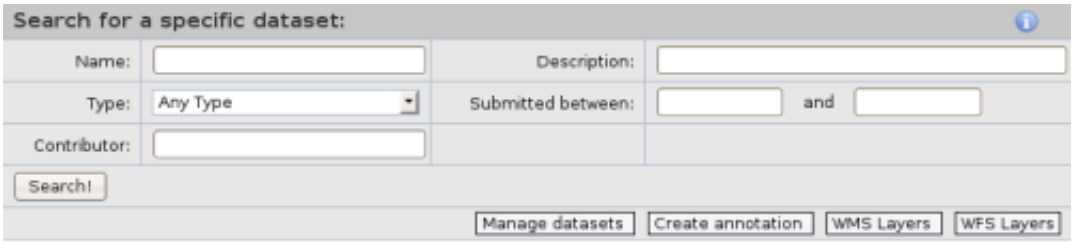


FIGURE 5.24: Search query panel.

- Query the database using the predefined query controls.
- Manage the datasets the currently authenticated user installed in GeoMoin
- Create new annotation layers
- Connect to OGC's OWS online servers and capture the datasets they offer.

Providing a demostration query provides the results shown in figure 5.25. In addition to providing the metadata associated with the resultin datasets, the user can click on the layer name in order to get a detailed description of the dataset information using feature extraction [5.26](achieved through the GDAL library[20]). Using the **MARK** control, the user can select particular layers from the listing and create MapServer mapfile definitions automatically. These definitions are stored in user–specific special files called markfiles (discussed in previous section). Using MapManager, the user can import these definitions in new or existing maps. It is important to note that the same

[20]Feature extraction using the gdal library takes considerable time when feature information as accessed. Additionally, in later version of GeoMoin, more extensive analysis on the spatial datasets will be possible.

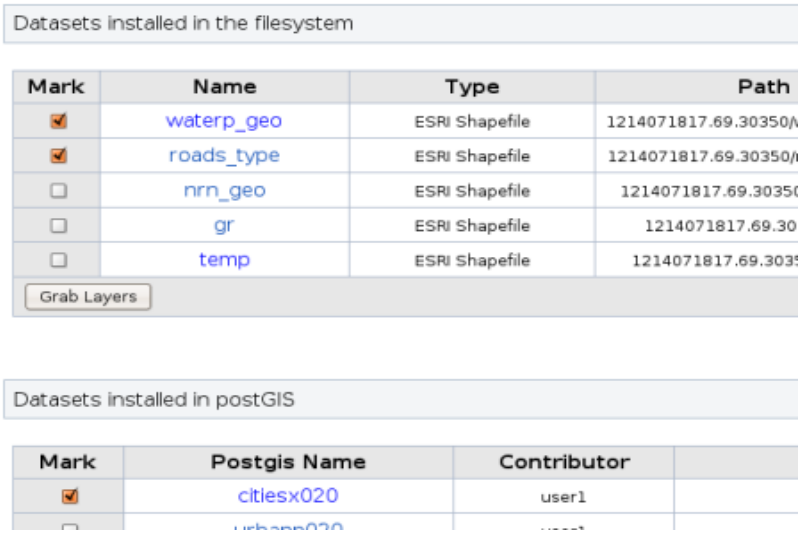


FIGURE 5.25: Search query results.

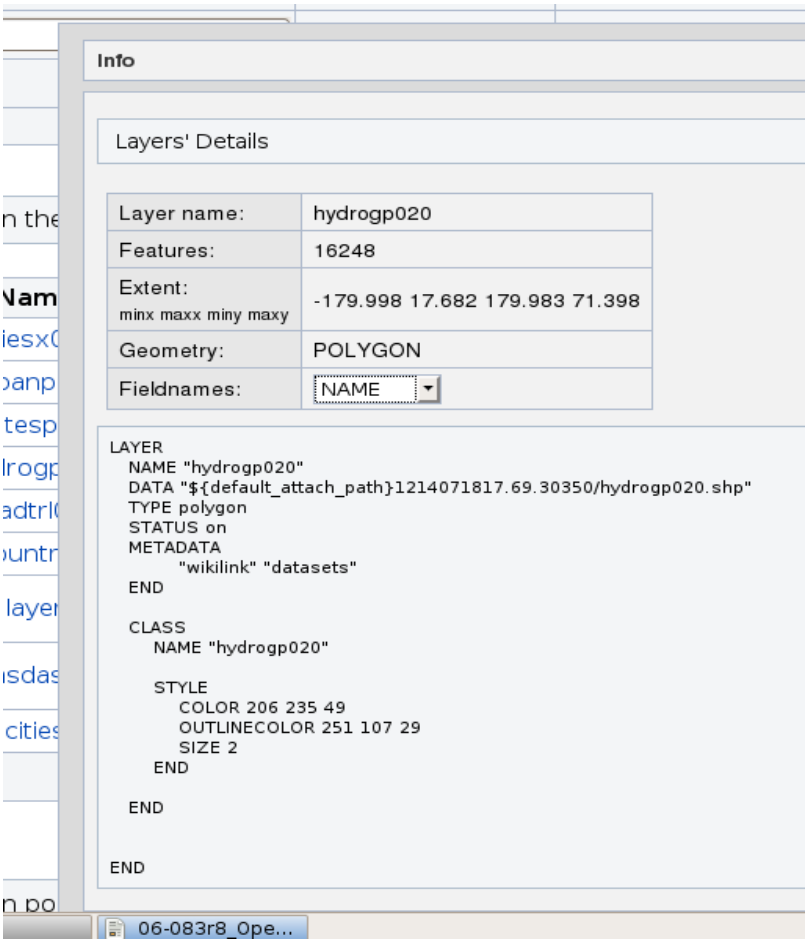


FIGURE 5.26: Layer information.

layer can be *marked* multiple times, thus using MapManager different options for every definition can be defined.

In this first GeoMoin version, managing the user–installed datasets, is the simple process of updating the metadata information used to link the datasets with the GeoMoin business logic. The user can modify the description of the layers along with the wiki page or the url associated to them. As far as annotation layers are concerned, additional management can take place, shown in figure 5.27. The following HTML SELECT

FIGURE 5.27: Modifying layer info.

controls manage the following features:

Viewable The user can select whether the annotation layer will be viewable by other users of the wiki. Note that if other users contributed in the layer, inserting new features, they will not be abled to access them even while they are owners of the tuples.

Editable The user can select whether the annotation layer will be editable by other users in terms of *inserting* new features. This means that if the editability is turned off, users that already created content will be able to manage it, but will not be able to create new.

DirectLink The user can select whether this annotation layer will react as a direct link to a web site or wiki page.

5.11.2.2 Creating annotation layers

Creating new annotation layers, is a process of generating a user–defined PostgreSQL table containing a PostGIS geospatial column. In order to create an annotation layer, the user must define fieldnames that will the non–spatial information of the layer [5.28]. The PostgreSQL datatypes that are currently supported by GeoMoin are:

- text
- integer

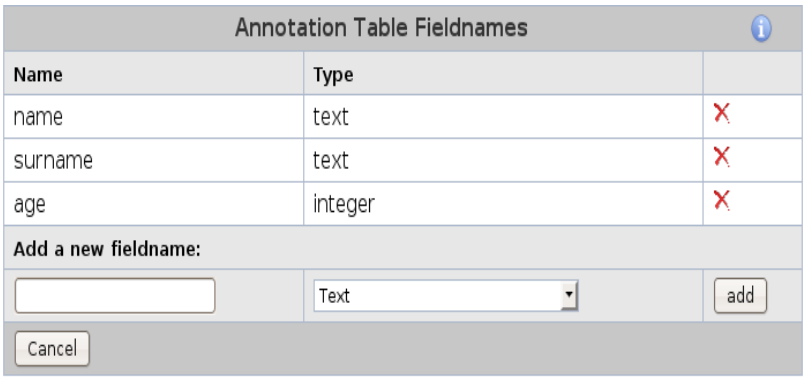


FIGURE 5.28: Adding annotation fieldnames.

- double precision
- Date

In order to create the annotation layer the following information must be provided:

Name The name of the annotation layer which will be used as the name of the PostgreSQL table. It will also act as the layer name when the annotation is embedded in a mapfile.

Description This metadata information must describe the layer's content in order to be easily accessible using the search queries.

Wikipage The wikipage associated with datasets, that includes detailed information about the layer's contents

Viewable/Editable/Directlink These features where described previously, in the part discussing the dataset management.

Geometry type The user can set the geometry type of the annotation features, currently in GeoMoin **point** and **polygon** geometries are supported. The form controls are displayed in the figure 5.29:

5.11.2.3 WMS managing

Using the LayerManager the user can import remote WMS servers in GeoMoin, access the datasets they offer and create mapfile layer definitions in order to use them within the MapManager. Unlike the LayerManager tools described above, the GeoMoin's OWS section relies completely on **AJAX** in order to provide a more usable interface due to

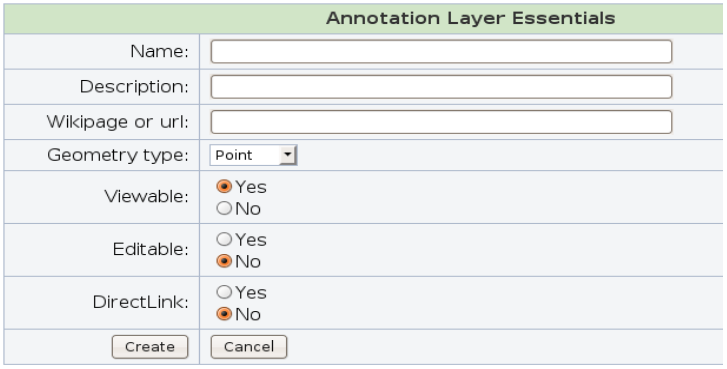


FIGURE 5.29: Creating annotation table.

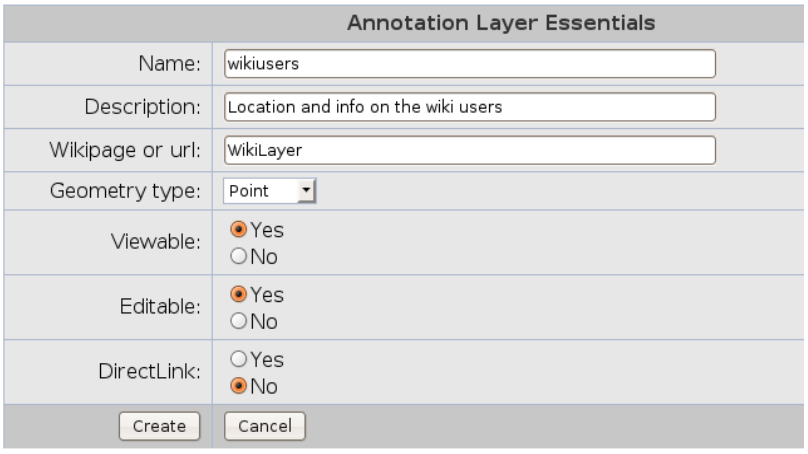


FIGURE 5.30: Annotation layer creation.

the time durations needed to remotely connect to the OWS services or preview the layers they offer. Following the conventional HTML submit architecture, the user would have been introduced to a blank web browser image, waiting for the GeoMoin server to connect to the remote WMS and provide the results.
The widget featured in [5.31] allows the user to:

- Connect to a WMS.
- Register a new WMS.
- Edit the settings for an existing WMS registration.
- Delete an existing WMS registration.

Creating or editing an existing WMS registration displayed the HTML controls shown in feature [5.32]. The user can add/edit the following options:

FIGURE 5.31: WMS widget.

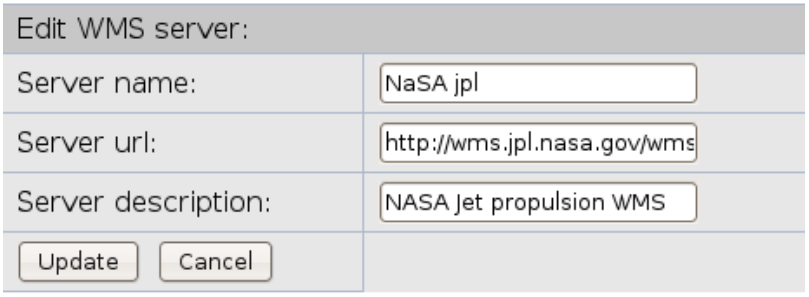


FIGURE 5.32: WMS registration.

- The name of the WMS service; should be kept short and descriptive.
- The url used to connect to the service.
- A detailed description of the services the WMS offers.

Upon connection to the Nasa JPL WMS, some general information about the service are displayed along with widgets that control each layer that is provided. In figure [5.33], the CONUS mosaic layer is expanded and a preview of the image provided, is requested, based on the options selected. The user can choose between creating a layer definition to use within the MapManager by clicking the "Grab Layer" button or preview the spatial image using the "Preview" button. Each of these controls are based on the options selected by the user which are, of course, provided by the capabilities of the current WMS:

LatLon box The maximum latitude/longitude bounding box the resulting image will cover. In the future the user will be able to select a particular extent in order to minimize the load of the server and the network.

Proj4 options The Proj4 definition (usually EPSG codes) used to project the geospatial extent with.

Style Different styles provided for the particular layer. Found in the capabilities XML document of the WMS.

Image type Different type for the encoding of the returned image.

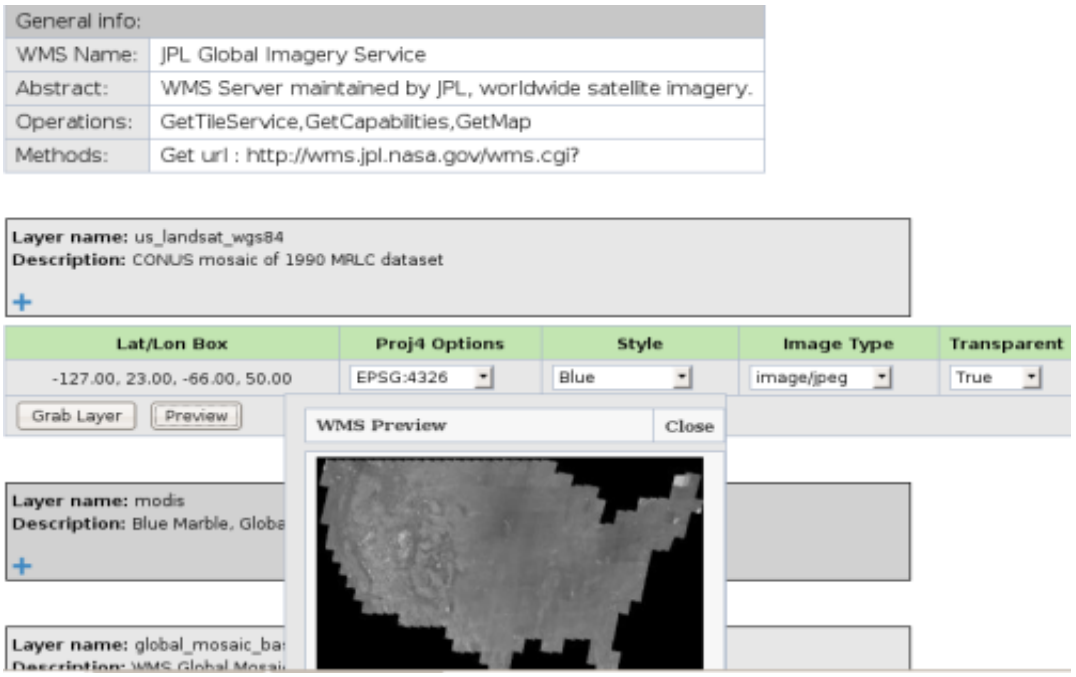


FIGURE 5.33: WMS demo connection.

Transparent Boolean control, requesting the service to provide an transparent image wherever "neutral" pixel values exist. It can be overrided by setting the mapfile's OFFSITE directive afterwards.

It is important to note that previewing a layer in a client–side process; GeoMoin creates the url that will point the browser in displaying the parameterized image and appends it in the HTML of the popup window as an HTML IMG tag. It is common that a preview may **not** be displayed due to misconfigurations in the remote WMS, service denials due to overload of requests or possible bugs within the OWSLib used to access WMS services. Continuous testing of the WMS services within GeoMoin will hopefully provide a stable interface for these types of requests.

5.11.3 Spatial Visualization System

The spatial visualization system (SVS) provides the rendering of the mapfiles along with the controls in order to perform spatial and non–spatial operations. The user–services part of the SVS is extensively using the AJAX technology to call the server script and the DOM and DHTML/javascript in order to manage the displayed information in the web page containing the rendering. The layout 5.34 depicts the separation of the different web components within the HTML document. Starting from the map rendering (the

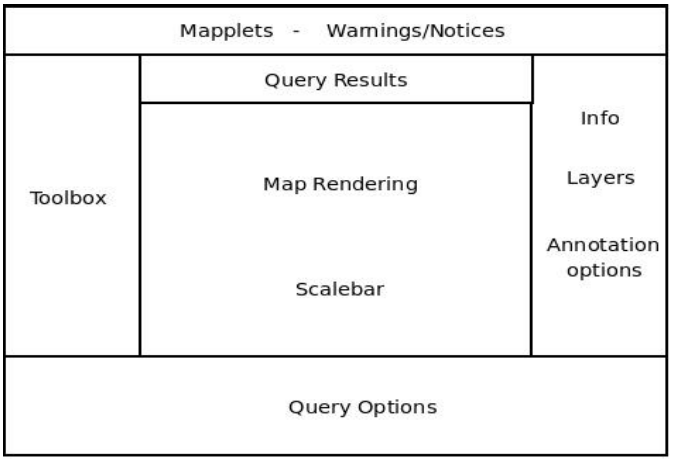


FIGURE 5.34: Layout of the SVS.

center of the layout, feature 5.35 provides a demonstration of a world map using NASA's JPL WMS service along with some other layers. Feature 5.36 shows the toolbox the user

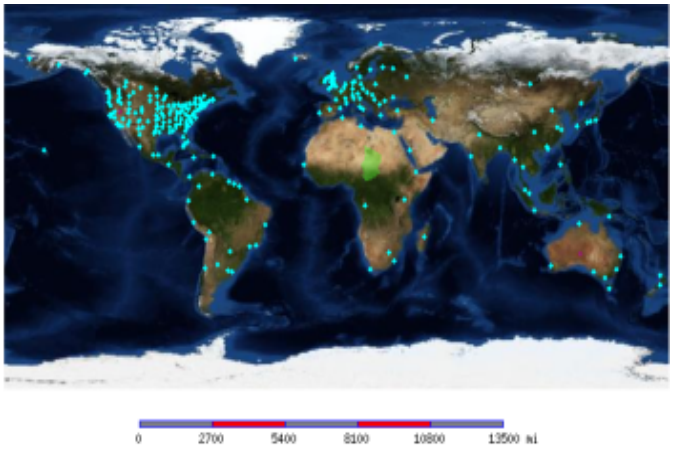


FIGURE 5.35: Demonstration of map rendering.

has access too.

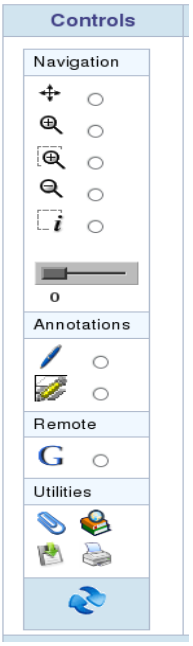


FIGURE 5.36: Toolbox

5.11.3.1 Navigation tools and queries

Using the navigation tools, the user can pan,zoom in or zoom out the map. Using the javascript slider below, particular zoom level can be selected in a scale from 1 to 20. The above tools, as soon as they are selected, are activated using a mouse click on the map rendering. Using javascript, the pixel coordinates of the mouse click event are calculated using the following method[21]:

```
pos_x = event.offsetX ? (event.offsetX) :
                        event.pageX - document.mapform.maprender.x;
pos_y = event.offsetY ? (event.offsetY) :
                        event.pageY - document.mapform.maprender.y;
```

Below the zoom tools, the query tool is used to enable the query mode on a map. Strictly speaking, "query mode" is a state that is linked to the cgi version of the mapserver package. The term "query mode" is used here to describe the situation where activation options like a mouse click on the image or the pressing of the refresh button, result in the processing of a particular query, **as long as the query tool is selected**. In order to perform a query, certain query type and options must be selected. [5.37] In the list below we describe the different query options:

[21] Thus, there is no need in defining the image as a server–side imagemap using the HTML IsMap directive.

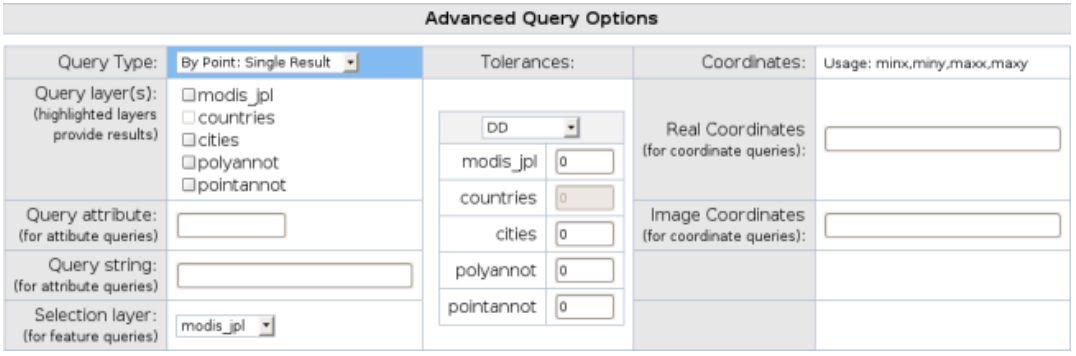


FIGURE 5.37: Query options.

Query type The different query types that are supported by GeoMoin. Details can be found in section 5.9.5. The most of the query type require a spatial activation; a mouse click on the map, while others (like attribute queries) require a refreshing of the map.

Query layers The layers checked in this section will be included in the query results.

Tolerances For every activated layer, a tolerance buffer can be defined along with the unit type of the buffer. Features that fall within the tolerance buffer are returned.

Selection layer Many query types require that querying function should be performed on a particular layer. For example, feature queries return all the features contained in the polygon pointed by a mouse click, thus the source layer must be defined. Additionally, attribute queries (performing on non–spatial data) must be performed on a particular layer.

Query attribute Used in attribute (non–spatial) queries. Defines the name of the attribute that will be included in an SQL WHERE–like query. For PostgreSQL layers this field can be omitted.

Query string This controls is used in conjuction with the query attribute and defines the value that will be used to search against the dataset. Can be a regular expression. For PostgreSQL selection layers a native WHERE clause must be defined.

Real Coordinates Using this controls a rectangle of real coordinates (LL,UR) can be defined and all the features that are within or intersect that geographic region are returned.

Image Coordinates Basically it is the same as the above control while the image coordinates defined are transformed to real coordinates before the query is performed.

Finally, after the appropriate options are defined, the query can be performed either by a mouse click or by clicking the refresh icon according to the query type requirements. On submitting, the appropriate AJAX function is called passing the control to the MapRenderer, which will execute the query and return the results in an XML form. Afterwards the appropriate javascript function will read the results and modify the DOM. A demonstration of an FEATURE query is shown in image 5.38. The query is defined to return all the cities in TEXAS,US.

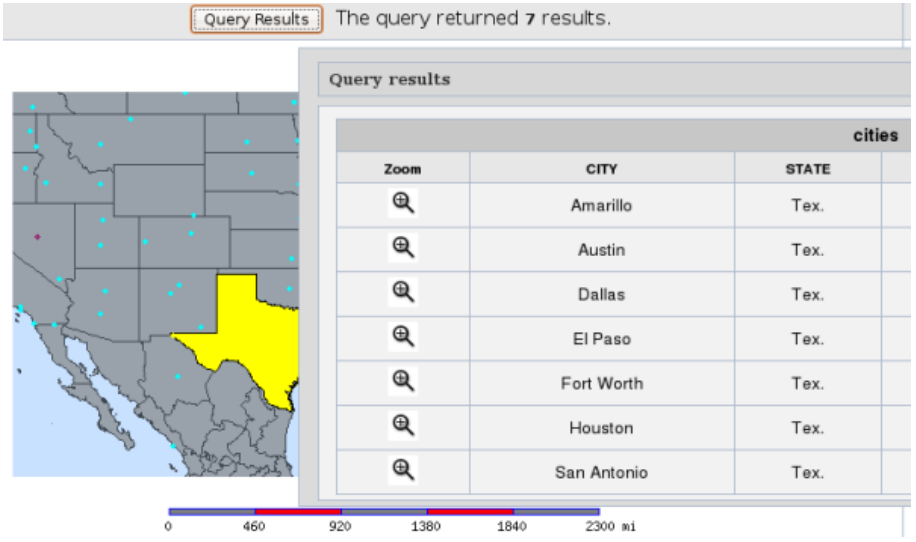


FIGURE 5.38: Demonstration of query results.

5.11.3.2 Annotation interactions

Continuing the discussion concerning the toolbox, new geographic features can be dynamically insert using the annotation tools; the first tool is used to draw point annotations while the second is used in order to draw polygon annotation based on a existing polygon. Activation of the annotation insert form is performed by a mouse click on the rendered map, while the image coordinates are calculated using the same commands as above. In order to describe the annotation interactions, the panel containing the annotation options must be presented. Thus, in feature 5.39 the following controls can be listed:

Toggle This controls enables or disables the action performed if a mouse click is issued over an annotation. By default, the annotations are toggled and on mouse click, a popup is displayed containing the annotation details. Particulary useful, if a zoom is needed near the annotation.

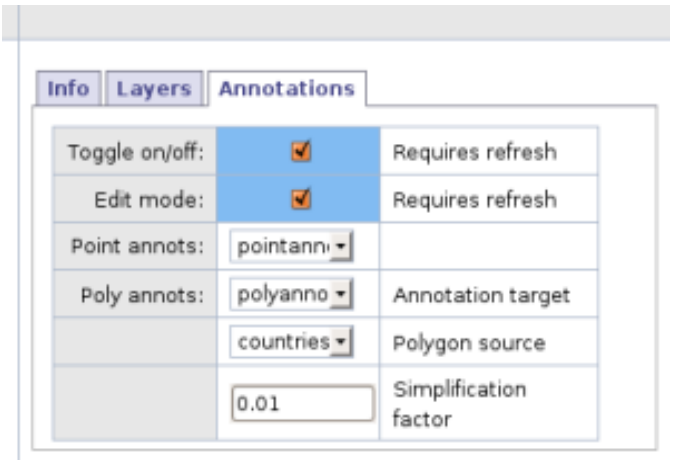


FIGURE 5.39: Annotation options.

Edit mode This controls toggles on or off the edit mode on the annotation. If the user does not have the appropriate access controls to handle the annotations, then this control does not alter anything.

Point Annots Selects which annotation layer will hold the new point annotations.

Poly Annots The first select box defines the layer that will hold the new polygon annotations while the second defines the source polygon layer that will be used to capture the geometry.

Simplification factor For polygon annotations, the simplification function provides good results. In case an override in needed, this is done by using this control.

The annotation of Greece will be demonstrated in the following example. Feature 5.40 shows the insertion form, while feature 5.41 depicts the annotation after it was inserted and selected. From the popup diplaying the annotation details, it is obvious that the user demonstrating the popup interaction has complete access over the annotation table, as he is able to insert new annotation and manage existing ones. The controls associated with the annotation management are the removal of the annotation using the Delete button and the updating of the annotation as long as all the controls are filled out. The same principle of operation, applies also to point annotations.

5.11.3.3 Layer management and general information

The right section of the layout of the web page displaying the map includes a tab that displays information about the current state of the rendering along with a tab that can be

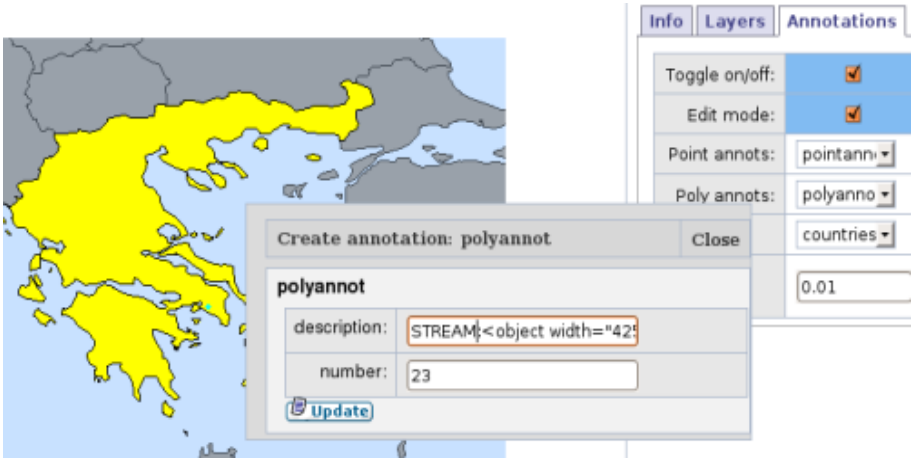


FIGURE 5.40: Inserting a new polygon annotation.

FIGURE 5.41: Viewing a polygon annotation.

used to manage the layers. Figure 5.42 depicts the info tab which provides information on the following issues:

- The reference map displaying the current extent in rectangle. The user can click anywhere in the reference map in order to move the viewpoint accordingly without changing the size of the geographic bounding box.
- The current map scale, defined in association with the map units defined in the mapfile.
- The geographic coordinates of the last click point on the map.
- The current geographic extent the rendering is displaying.
- The legend graphic show the layers (in strict terms, the classes) displayed along with the symbol used to render the features contained in the datasets.

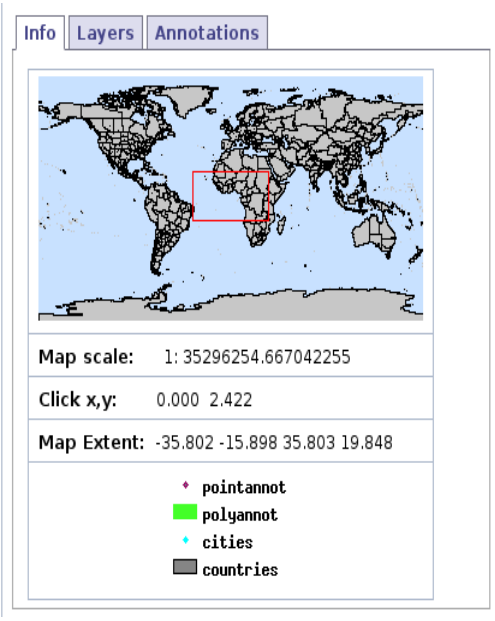


FIGURE 5.42: Information tab.

Below, figure 5.43 displayed the layer managing tab. Using the layer managing, the user

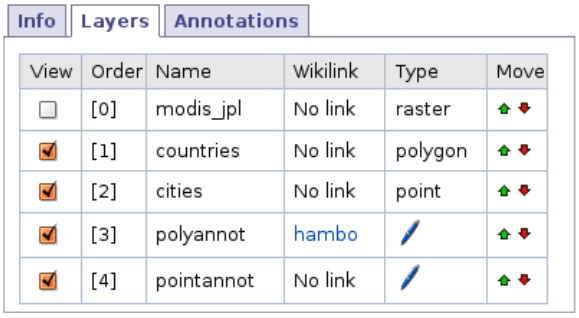


FIGURE 5.43: Layer managing tab.

has access to the following actions and information:

View Using this control, layers can be excluded from the mapserver rendering queue, thus not rendered. The action does not require refreshing the map, because the appropriate AJAX function is called while listesting on the onClick event of the checkbox.

Order Displays the initial ordering number of each layer, while the position in respect to the other layers define the current layer order.

Name The name of the layer defined in the mapfile that was used to render the current map.

Wikilink The link to wiki page associated with a particular layer. Information about the link address are captured from the layer's metadata directive "wikilocation"

Type The mapserver internal type of the layer rendered. For GeoMoin annotation layers a pen defines their ability to hold annotation information.

Move These controls move a particular layer up or down within the layers' drawing order. It is importand to note that move a layer up means that it will be rendered earlier, thus it will de hidden behind the layer it was swapped with.

5.11.3.4 Mapplet interactions

In this section we will provide some images depicted the interactions between the user and the mapplets. Starting with a synchronous mapplet that displays a Tissot indicatrix over the current rendering: When the "enable" button on the mapplet is pressed the

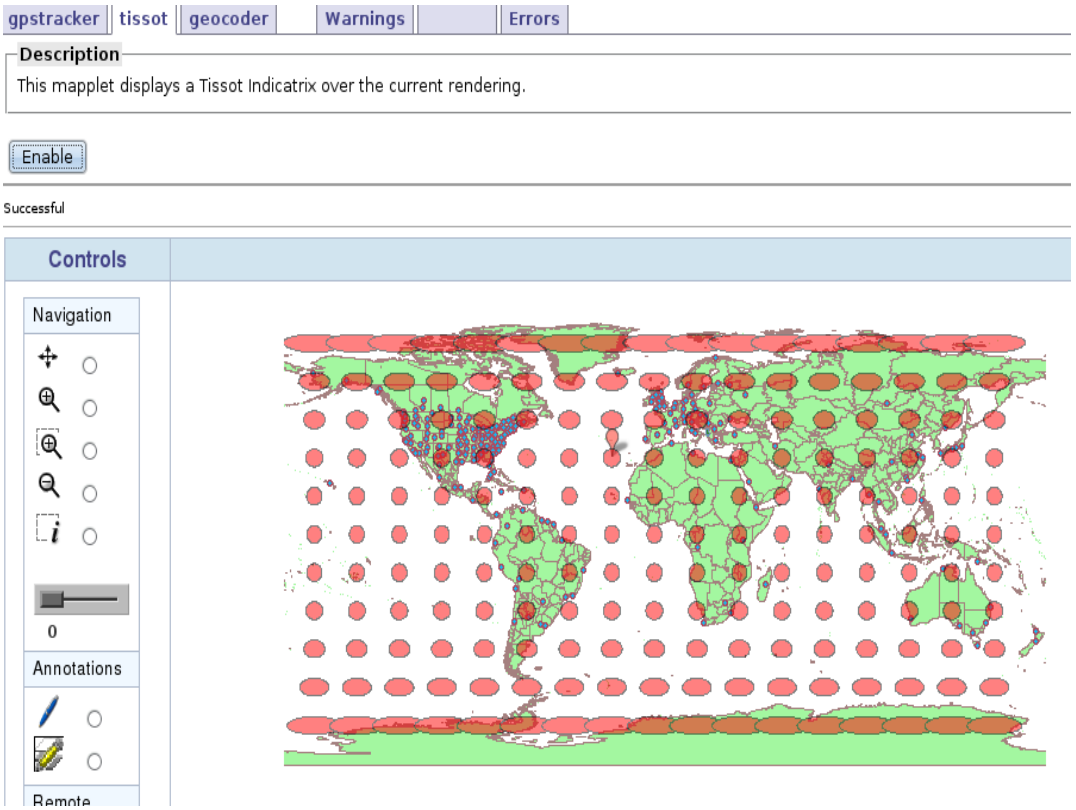


FIGURE 5.44: Synchonous mapplet example.

rendering take place immediately producing the desired result.
Moving on to asynchonous mapplets, the geocoding example presented earlier is visualised below: The user is required to input the desired location eg. "Abbey Ln, Maltby, Rotherham, UK" along with a geocoder (in this case Google) and press "Search". Ajax

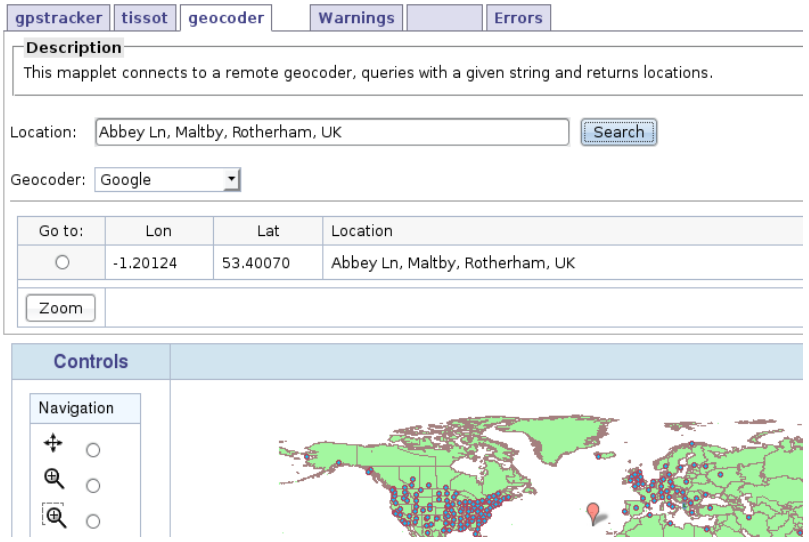


FIGURE 5.45: Asynchonous mapplet part 1.

will partially alter the HTML page to present the result, leaving the rest of the page intact. This means that while the geocoding query is taking place, the user can independently zoom, pan or create annotations. As soon as the results are fetched and the user selects the one that matches his criteria, he can press "Zoom" so that the map is panned/zoom to the particular location:

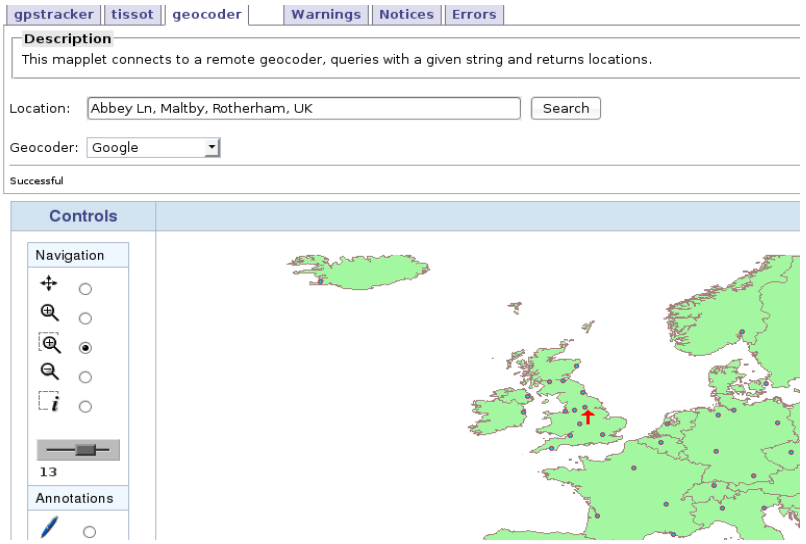


FIGURE 5.46: Asynchonous mapplet part 2.

5.11.4 Interactions between text and maps

A geospatial wiki does not solely targets to the development of tools that will display and manage spatial content. These tools should serve as a mean of incorporating this multimedia type along with various other types the wiki supports, such as text, sound, image or video. This subject clearly separates GeoMoin from a web application designed to specialize in spatial rendering such as GoogleMaps. In order to achieve an ideal integration, both GeoMoin must support and enhance MoinMoin's services, while MoinMoin can be enhanced support GeoMoin's service. For example, when a user creates an account in the wiki, he could be prompted with a map where he can annotate the location where he stays. There are no limitations to what can be achieved by combining the base wiki system and GeoMoin, as far as spatial support is concert. In this section, we will present some examples and tools that can be used to enhance a wiki page with the services GeoMoin offers.

In previous sections it was clearly stated that the Spatial Visualization System can be engaged from every wiki page of the system. That is, a wiki page can incorporate a fully interactive rendering of a map the user has created. While the user navigates through the map, it would be possible that a particular rendering worths to be saved either as an image or as a "bookmark". Saving the rendering as an image could be used to include it in wiki page as image within text. Saving a "bookmark" could be used in order to continue allow future navigation starting from this particular rendering. GeoMoin supports both actions by using the tools shown in figure 5.47.

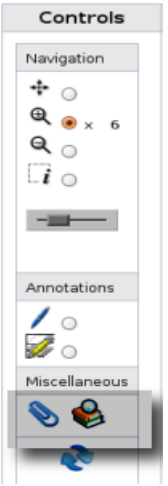


FIGURE 5.47: Wiki tools

The first tool, allows the current rendering to be saved as an image within the attachments of a particular wiki page. This allows the wiki page to contain this rendering

regardless of whether the page containing the map would be deleted in the future (5.48). The attachment can be embedded within the wiki text of the page:

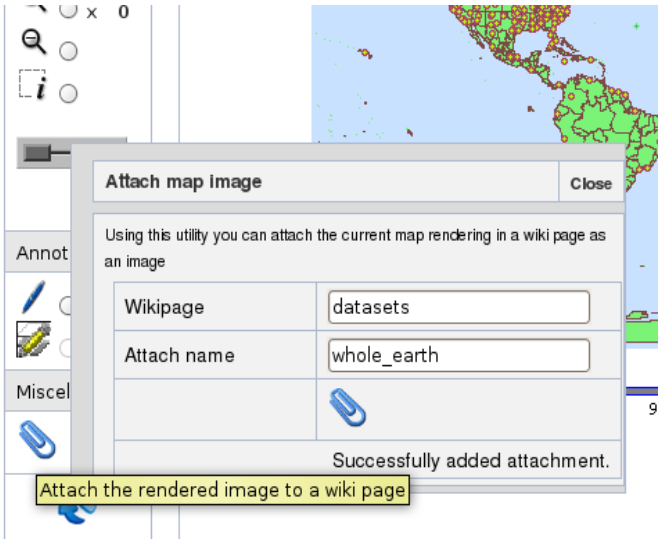


FIGURE 5.48: Attaching a rendering to a page

```
'''Here you can see an image of the earth's surface:'''

{{attachment:whole_earth.png}}
```

The resulting output can be seen in figure 5.49.

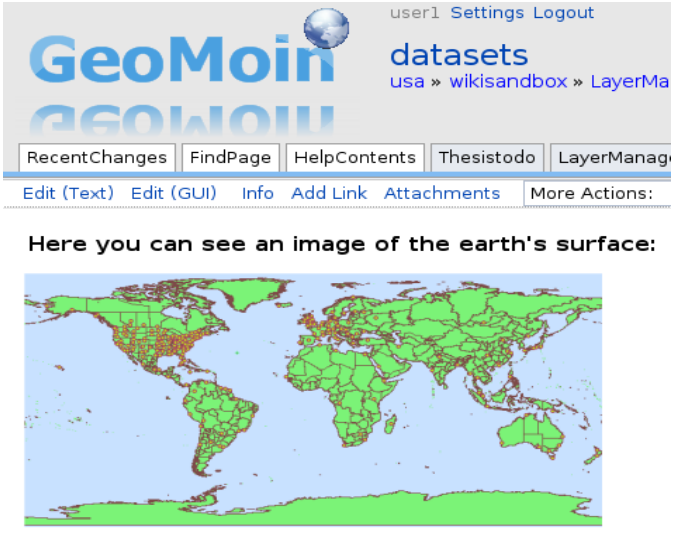


FIGURE 5.49: Displaying the attached map

The second tools, creates a bookmark called ***BookMap***, which is attached to a particular wikipage. The BookMap is a file that contains the required information in order to create a link to the page that generated it, as well as information used to roll back to the particular rendering which generated it (5.50). The BookMap is now part of the

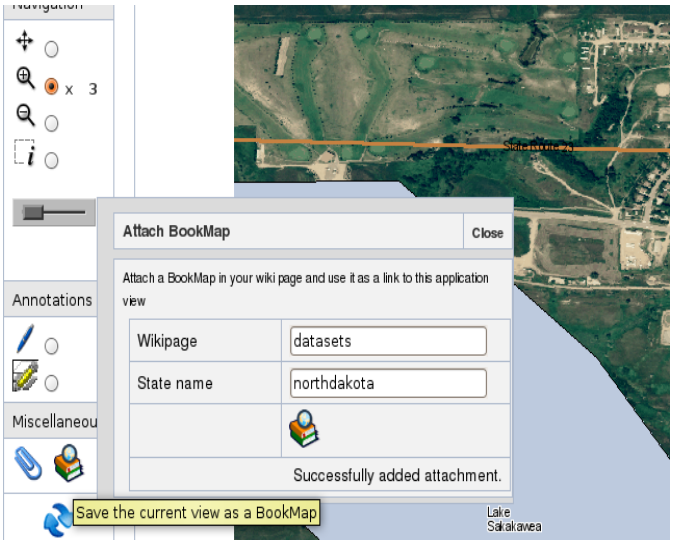


FIGURE 5.50: Attaching a BookMap to a page

attachment of the wikipage that was defined. In order to use the book to create a link to the wiki page that hosts the map, the wiki text can include the following GeoMoin macro:

```
Click this link to <<BookMap(northdakota,navigate in North Dakota)>>
```

The first argument is the name of the BookMap attached. The second argument, is the text that will be displayed as the link. The macro can also be configured to display an image instead of text by issuing:

```
Click the image below to navigate to through North Dakota:
<<BookMap(northdakota,north.png,1)>>
```

Now the second argument is the attached image (could be a rendering created using the tool described before) while the third argument flags whether the second argument should be treated as an image or text. Thus, we can incorporate the above tools and the MoinMoin syntax to generate a image gallery that directly links to the displayed renderings for further navigation (5.51):

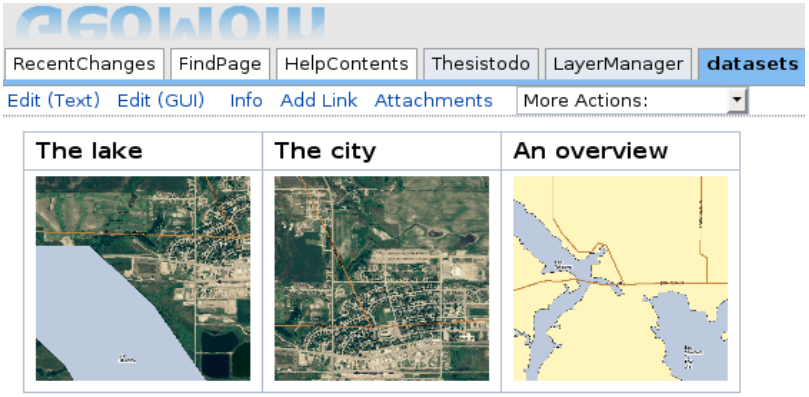


FIGURE 5.51: A gallery using BookMaps

Chapter 6

Conclusion

The last chapter of the report is dedicated to the outcomes of the development effort within GeoMoin. The best way to demonstrate the resulting elements is to provide a complete case–study including the installation of datasets and the generation of a particular map. This is the subject of the first section of the chapter. The next section is dedicated to a discussion concerning the goals that can be achieved using GeoMoin in relevance to the motivations presented in the first chapter of this work. Finally, in the last section, directions for future research and development are provided in an effort to address important issues that are not yet part of GeoMoin.

6.1 Case–study: A map of greece

The case study presented is the generation of a interactive map of the geographic region covered by the country of greece. First of all, in order to create a map, some datasets are selected and uploaded to the wiki.:

- A vector dataset of polygon type, showing both greece and divisions within it.
- A raster image depicting the island of Crete and the south Peloponisos.

First of all we will create an empty wiki page named **GreeceMap** which will be used to upload the datasets and host the map. Using the wiki attachements we can upload the above datasets [6.1] (for each dataset a unique archive is created and the part *dataset_* is always used as prefix). After the datasets are attached, the action **InstallSpatial** from the list of actions is selected, resulting in the output shown in figure 6.2. In order to install the datasets in this example, the raster is installed in the filesystem while the vector is installed in PostgreSQL database. It is important to note the geometry must

FIGURE 6.1: Case study: Attaching datasets.

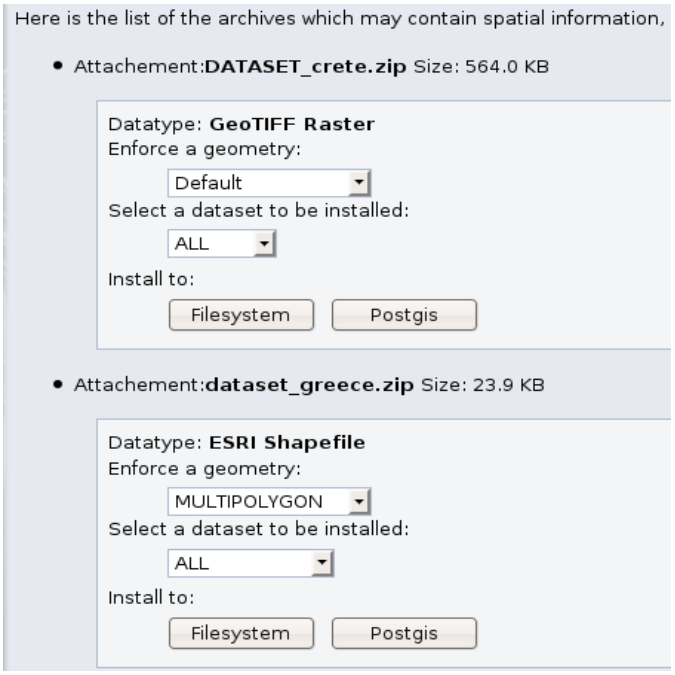
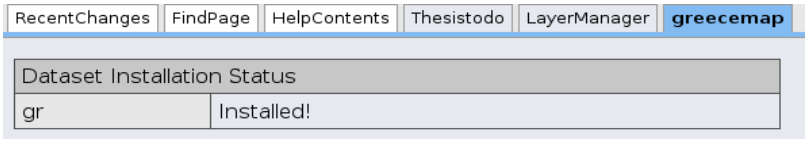


FIGURE 6.2: Case study: Installing datasets.

be enforced to MULTIPOLYGON (as shown in the figure) because PostGIS includes constraints concerning the correct type of geometry included within the feature tables. As soon as each dataset is installed, the system will notify the user for the successful operation (Figure 6.1). By now, LayerManager must be visited for the following tasks:



- To check if the dataset was installed successfully.
- To get information on the dataset, particularly the geographic extent.
- To create layer definitions, in order to make the mapfile.

Within LayerManager, we can query the database of dataset registrations using the name of the vector dataset (in this case "gr"), along with the user name that installed the dataset in order to avoid duplicate names (in this case "user1"). The vector dataset will be returned and its named can be click to provide the details shown in figure 6.3. The geographic extent along with the fieldnames can be seen. This extent will be used

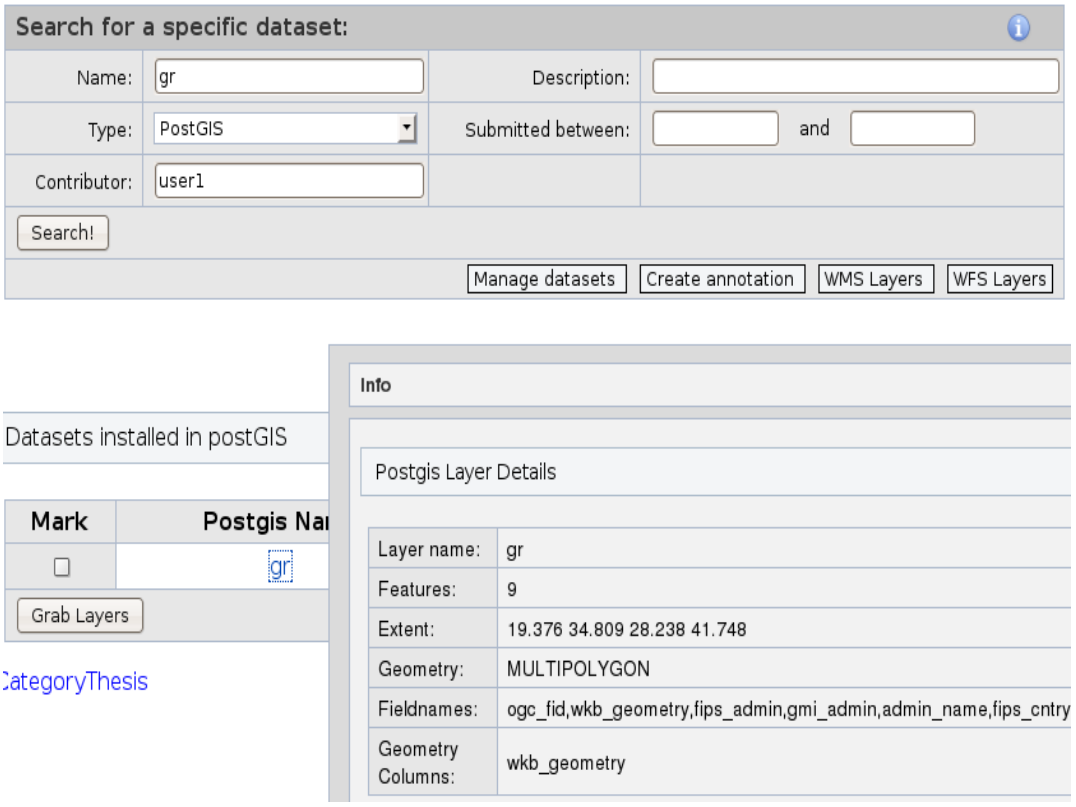
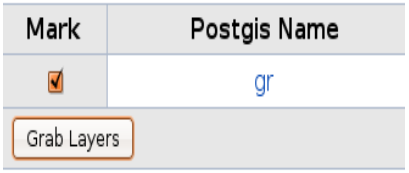


FIGURE 6.3: Case study: Dataset information.

in the MapManager to create the map.
In order to create the layer definitions for both the vector and the raster dataset, the must be properly returned using the appropriate queries and afterwards **marked** as shown in figure 6.1. By now the layer definitions are created and we are ready to proceed to the MapManager in order to create the actual mapfile. Within MapManager, clicking the LayerManager tab, shows the two layer that where imported from the LayerManager previously. The raster will be maintained as–is, but as far as the vector is concerned, two operation must take place:



- Make the layer queryable.
- Add a label for a particular fieldname.

In order to make the layer queryable, the name of the layer must be click to show the basic layer options, and the attribute named **queryable** must be set to **true** (shown in Fig.6.4).

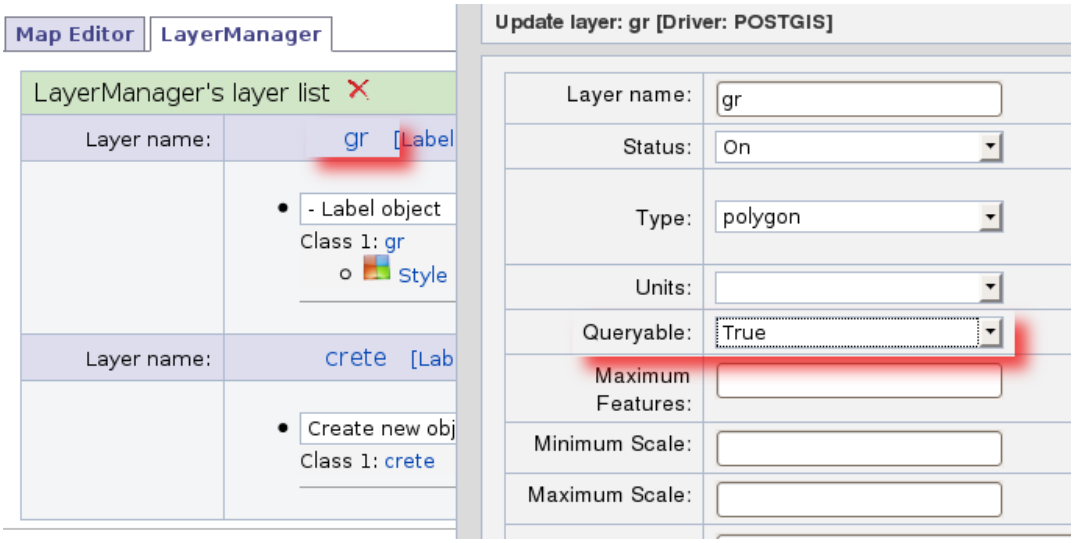


FIGURE 6.4: Case study: Make layer queryable.

In order to create a label, the item "Label Object" must be selected from the drop–down box, resulting the generation of a draft label object as shown in figure 6.5. The first icon popups a windows that can be used to define the general label options such as the fieldname to be labeled, whereas the second icon popups a windows that contains attribute managing the label's position and colorising options.
As soon as the layer options are filled, the tab containing the map options can be used to insert the map details. Some basic parameters that are used in this example are:

- The name of the map.
- The size of the map in pixels.

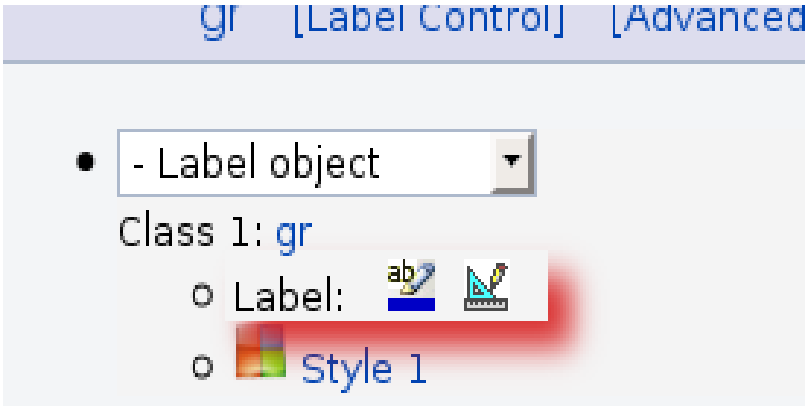


FIGURE 6.5: Case study: Creating a label.

- The ground units (usually relative to the projection used).
- The image type of the resulting renderings (e.g. png, tiff).
- The extent, the map covers (can be found using the layer info from the LayerManager).
- The background color of the image.
- The map projection.

A completed form of the above element is presented in figure 6.8.

Finally, the layers must be imported to the map and the map must be saved in the filesystem (Fig. 6.6).

FIGURE 6.6: Case study: Importing layers.

By now, the map is created and is ready to be embedded in any wiki page of the system. Thus, we open the page created before (named "greecemap") and the parser that engages the Spatial Visualisation System is defined as shown in figure 6.7.

Finally the page revision must be saved and the resulting fully navigational map is shown in figure 6.9.

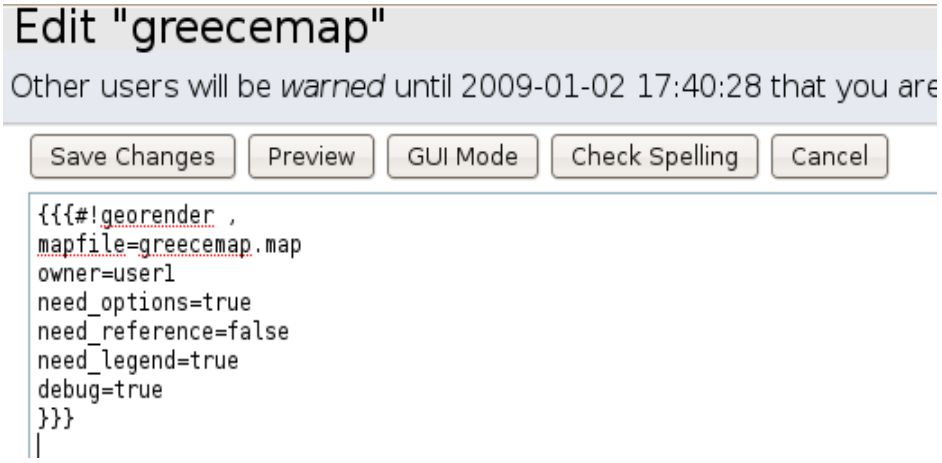


FIGURE 6.7: Case study: Adding the map to wiki page.

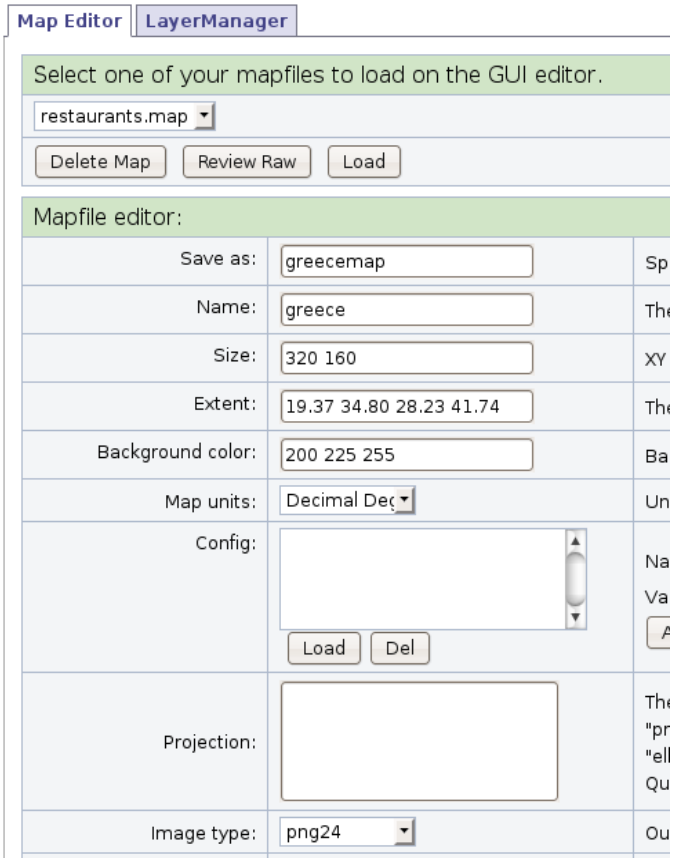


FIGURE 6.8: Case study: Setting the map options.

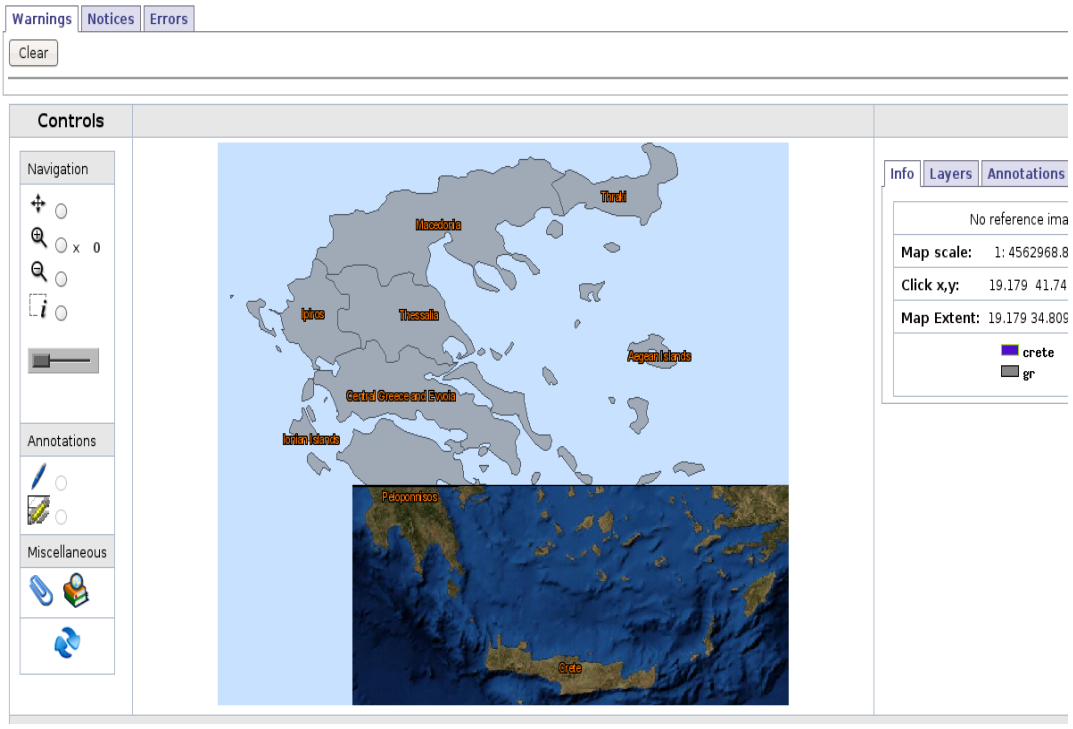


FIGURE 6.9: Case study: Visualizing the map.

6.2 Summary of theoritical results

In this section, the goals which have been theoritically achieved, through the development of GeoMoin, will be addressed in reference to Chapter 1: "Motivations".
Using spatially–enabled wiki applications, the world of collaborative web and GIS comes together. GeoMoin is such an application, providing ease of access and contribution of geographic information by a whole community of its users. The users can visualize the contributed content within the Spatial Visualization system of the application, or, optionally, download it, to use it in Desktop GIS applications [1]. Additionally, online OWS servers can be registered and used as local datasets, with usability limited only to the network throughput between the OWS server and the GeoMoin server. Institutions could also buy spatial datasets and install them in a GeoMoin server allowing users to create maps and interactive wiki pages disabling the right to download them by using MoinMoin access controls or by simply not posting the dataset archives in the wikipage attachments, thus online users could benefit from datasets they would not have access to by other means.
The fact that GeoMoin is a ready–to–use collaborative mapping application allows organizations or simple users to add it in their wikis. No expertise in needed to operate a

[1] It is important to note that the handling of copyrighted geospatial information without the explicit terms set by the owner, within a public GeoMoin server, is not encouraged by any means.

working GeoMoin installation and, apparently, no specialized personel. Thus, enhanced maps can be created that can hold information of any kind in regard to the needs of the users. For example, a group dedicating an effort to create a web–site dealing with the World War II, could plan the use of wiki instead of another static or dynamic web architecture and use GeoMoin to add spatial content. In that scenario, maps can be generated depicting various concepts of the subject or incorporating today information in regard to this era.
As far as, education advances is concerned, GIS software and libraries like MapServer, GDAL, GEOS spatial libraries are open–source solutions that are increasingly being used by universities worldwide. Providing an easy to use application like GeoMoin, users can take advantages of the features these packages offer and learn the various aspects of the Web–GIS technology without being concerned about technical issues which are handled as much transparently as possible. In a developer level, GeoMoin can act as a framework where users and students can build additional components either by modifying the open–source code or by implementing new mapplets to incorporate new features. For example, creating a mapplet, the user can visualise a GPS tracklog or waypoint file within the current map rendering. By creating a new driver in the ogrtypes module, the user can install GPS logs, in the database or in the filesystem by converting to a well–known format like ESRI Shapefile.

The opportunities of the collaborative web–mapping applications are countless and are provided by applications of very different architectures and forms. For example, WikiMapia is "map–based", meaning that the users are presented with a shared map where they can add annotations by drawing features, while, in the other hand, GeoMoin is more "content–based" as it is a component that enhances a MoinMoin wiki to provide spatial support to each page of the wiki system. GeoMoin is a carefully developed application but not yet "mature" as it is not yet tested against real–world scenarios. Thus, it is open to future enhancements which is the subject of the final section of this chapter.

6.3 Future work

We present some open issues for future work in the following sub-sections. It is important to note that the nature of this project as the developent of a web–mapping framework typically follows the development procedure of all the open–source projects of this type. As such, a project management system like **trac** along with a version control system can be maintained where users can publish enhancements or defects. Thus, the future work presented in this document cannot include minor specialized issues.

6.3.1 Global availability of GeoMoin

Up until the submission of this work, there has not been an effort to make an instance of GeoMoin publically available on the web. While the inner parts that constitute GeoMoin like the MoinMoin wiki, MapServer and PostGIS are already recognized for their ability to handle and serve large scales of information individually, the combination of these tools to form an online repository of geographic information and interactive maps within an organization, institute or a dedicated resource (like Wikipedia's achievement on encyclopedia content) is highly desirable.
This effort requires testbenches between the different environments that the wiki can be served within the context of a web server like FastCGI, modpython, ModWSGI or Twisted to choose the most efficient, less error–prone and easily configurable options.
Additionally, new design concepts or requirements may arise concerning the availability and reachability of geographic content and maps within the server. For example, in a large scale scenario the MoinMoin indexing and search modules could be extented to include information not only for the wikipages and their raw content but also the spatial content and maps represented within the page.
Summarizing, it is obvious that this effort deals with many different aspects of a collaborative web mapping application in terms of its availability options and the feedback that arises from the usage scenarios that lead to new design concepts. As such, the multiple branches can be identified, analysed and provided as specific future enhancements.

6.3.2 GeoMoin within a multi–server environment

In order to get the most of a web mapping application, the application itself should be able to be provided from within multiple servers, in a scalable manner. The options for balancing the service falls into three main categories:

- Multiple instances of the wiki serving the same content.
- Decentralization of the modules that provide the spatial operations and rendering.
- Decentralization of the geospatial content

Maintaining multiple instances of the wiki can provide a per–location or a fail–safe access to the service. There exist methods for mirroring the content of the wiki using MoinMoin's Wiki–RPC interface. One tool, licenced under GPL that perform such a job can be found in `http://www.merten-home.de/FreeSoftware/moinupdate/`.
The computational cost of a web–mapping application that is served within a wiki is

attributed to the modules the render the geospatial content. GeoMoin's spatial visualization system is using the MapServer to render maps and provide query results through mapscript and is called within the AJAX interface as a remote script. Thus, multiple servers, serving the spatial visualization system can be maintained and selected based on specific criteria.

Maintaining multiple instances of the wiki or(/and) multiple servers serving the rendering process requires individual study of the methods that mapserver uses to obtain geographic information. For GeoMoin, these are the local datasets (shapefiles, GML e.t.c.), the PostgreSQL database and the OWS services. The OWS services excel at providing spatial content through the internet, while the PostgreSQL is ideal for servers that are maintained within an intranet. The local datasets are the less flexible source as they reside in the local server's filesystem (of course, a network filesystem can be used to serve these types of datasets to an intranet).

An interesting idea that could provide considerable speedup is the employment of different servers providing the spatial visualization system handling different part of the total extent that is going to be rendered. Thus, the rendering module can compute the required extent, divide it in equal parts and distribute it to different servers. The parts returned can be combined to form the whole rendering or AJAX can be employed to partially fill the response to the client thus providing a tiling effect.

Appendix A

Technical Issues

A.1 Installation

In this section we will describe the steps needed to perform the installation and configuration of a working GeoMoin instance. Due to the fact that GeoMoin is depended upon many different packages, the whole operation may appear cumbersome and error–prone. Thus, each step is presented and analysed as much as possible, in order to prevent frustration. Of course, familiarity with many aspects of a linux distribution is assumed.
Throughout the discussion, each step of the installation procedure was reproduced on a **clean** installation of the linux distribution **Ubuntu 8.10 Desktop i386**[1]. Ubuntu is shipped along with a graphical utility called **Synaptic** and command–line tools like **apt–get** which can be used to automatically download and install additional packages found in online **repositories**. In many cases, automatic installation of some packages is encouraged, to avoid conflicts and errors based on in misconfiguration of their depencencies.

The following sections are dedicated to the installation of:

- Pre–required general libraries.
- PostgreSQL DBMS and Postgis.
- Apache web server.
- MapServer and GDAL.
- MoinMoin wiki.

[1]It is preferable that in order to follow the procedure on a different linux distribution, a request to the author should be made.

When the above requirements are satisfied, installation of GeoMoin can take place. The installation is divided in three parts:

- Creation the GeoMoin spatial database.
- Installation of a custom MoinMoin wiki instance.
- Installation of GeoMoin modules and shared files within the above instance.

A.1.1 Pre–required general libraries

Using the Linux package managers, the following packages must be installed or ensured that are installed by default:

g++ This is the GNU C++ compiler, a fairly portable optimizing compiler for C++.

Python2.5 The python programming language in its 2.5 version.

Python-dev Python header files and static libraries

swig SWIG is a compiler that makes it easy to integrate C and C++ code with other languages. From version 1.3 and above.

python-cheetah Cheetah is a python based template engine. It is used in GeoMoin in order to generate MapServer mapfiles.

libgd2-xpm GD is graphics library. Mapserver natively uses it to render images. Note that GD has its own list of dependencies including zlib, libpng, FreeType and libJPEG. These provide GD with image compression and support for TrueType fonts. Apparently after its installation the following packages must be ensured:

libfreetype and dev FreeType is a font rendering engine used by GD. libfreetype-dev is suggested too.

libJPEG and dev Used to render JPEG images.

libPNG and dev Used to render PNG images.

zlib and dev Data compression used by GD.

libgd2-xpm-dev Development files and headers for the GD library.

libcurl3 libcurl is designed to be a solid, usable, reliable and portable multi-protocol file transfer library. Used by Mapserver for OWS requests including WMS.

libcurl4-gnutls-dev These files (ie. includes, static library, manual pages) allow to build software which uses libcurl.

fontconfig Fontconfig is a font configuration and customization library. It is designed to locate fonts within the system and select them according to requirements specified by applications.

libfontconfig1 This package contains the runtime library needed to launch applications using fontconfig.

libgeos-c1 Geometry engine for GIS, C library.

libgeos2c2a Geometry engine for GIS, C++ library.

libgeos-dev Geometry engine for GIS, development files.

After the installation of the above packages is successful, some important packages that Mapserver and GDAL are using, must be manually configured and installed too. These are: Proj.4, Shapelib, geopy and OWSLib.

Building and Installing Proj.4

Proj.4 is a library of cartographic projection routines. It can be accessed by MapServer of standalone mode to perform projection on an entire dataset. It is available at http://trac.osgeo.org/proj/. The following steps are used to built Proj.4:

```
tar -xvzf proj-4.6.1.tar.gz -C /usr/local/src/
cd /usr/local/src/proj-4.6.1
./configure
make
make install
ldconfig
```

Building and Installing ShapeLIB

shapelib is a library of C routines for creating and manipulating shapefiles. Several utilities are included in the distribution to perform these actions. One can create shapefiles (which include DBF files), dump the contents of shapefiles or change the projection of shapefiles. Some of the utilities depend on Proj.4. The shapelib can be accessed through http://dl.maptools.org/dl/shapelib/. The following steps are used to built Shapelib:

```
tar -xvzf shapelib-1.2.10.tar.gz -C /usr/local/src/
cd /usr/local/src/shapelib-1.2.10
make
cp dbfadd /usr/local/bin/ (repeat for dbfcreate, dbfdump, shpadd,
                                      shpcreate,shpdump,shprewind,
```

```
                                    shptest)
make test                  (ensure it is successful)
make lib                   (optional library built)
make lib_install
ldconfig
```

Building and Installing GeoPy

GeoPy makes it easy for developers to locate the coordinates of addresses, cities, countries, and landmarks across the globe using third-party geocoders and other sources of data, such as wikis. That is, geopy, is a python module that provides a simple to use interface, in order to connect to various geocoders. The official site can be found at http://exogen.case.edu/projects/geopy/. In order to install it, we will use the Cheese-Shop, an online python repository. Thus the following are needed:

Python-setuptools Binaries that allow the connection and installation from the repository

BeautifulSoup A simple HTML/XML parser

SimpleJSON An JSON parser

geopy The actual module

The installation procedure is the following:

```
apt-get install python-setuptools
easy_install BeautifulSoup
easy_install simplejson
easy_install geopy
```

The installation can be validate by issuing: "import geopy" in a python command–line like above.

Building and Installing OWSLib

OWSLib is a set of python modules for accessing OWS services. Currently supports WMS, WFS, WCS. The installation is straight–forward using the **easy_install** tool described above thus:

```
easy_install OWSLib
```

A.1.2 PostgreSQL and Postgis.

GeoMoin currently uses the PostgreSQL DBMS for its database needs, along with the Apache web server in order to provide the service to the network. Due to the fact that the installation of these enviroments is highly depended in each individual linux distribution and their tuning is the subject of specialised books, we decided that the installation should be done using the before–mentioned package managers. Apparently, this approach will minimize problems based on individual configurations. Of course, a system administrator could easily import the later steps in a working PostgreSQL or Apache installation.
As far as the DBMS is concerned, the following packages are needed:

postgresql-8.3 The 8.3 version of the PostgreSQL DBMS. Includes the command line psql and pg_config utilities.

postgresql-server-dev-8.3 Header files for compiling SSI code to link into PostgreSQL's backend; for example, for C functions to be called from SQL.

python-pygresql PyGreSQL is the Python module that interfaces to a PostgreSQL database.

libpq-dev Version 8.3 header files and static library for compiling C programs to link with the libpq library in order to communicate with a PostgreSQL database backend.

The configuration of the DBMS can be done using the following configuration file:

/etc/postgresql/8.3/main/postgresql.conf

By default, the server is listening for incoming connections to "localhost" on port 5432 (although it could be 5433). Additionally, the required executables installed, can be found in **/usr/lib/postgresql/8.3/bin**. The **psql** is used from command–line in order to administer the database. A basic configuration could be the following:

```
user1@zeppelin:~$ su root
root@zeppelin:~$ passwd postgres     # in order to set a password
root@zeppelin:~$ su postgres
postgres@zeppelin:~$ psql            # connect to DBMS
postgres=# ALTER user postgres WITH password '*******';
postgres=# \q
```

The graphical frontend **pgAdmin3** can be used with the specified password in order to connect easily to every database created.

As soon as the Postgresql DBMS is installed, the spatial extension called **postgis** can be built and imported. Afterwards, the GeoMoin database, schema, functions and tables are additionally installed. PostGIS can be obtained from http://postgis.refractions.net/download/. In order to built it, the following command are needed:

```
tar -xvzf postgis-1.3.5.tar.gz -C /usr/local/src/
cd /usr/local/src/postgis-1.3.5
./configure
make
make install
ldconfig
```

The Postgresql installation will be identified automatically. We are particularly interested in the following files contained under the path **/usr/share/postgresql/8.3/contrib/**:

lwpostgis.sql Contains the required geometry types and functions.

spatial_ref_sys.sql Contains the spatial reference system, or else, the different projections supported.

A.1.3 Apache web server

As far as the web server is concerned, the **Apache2** package[2] is needed. In the directory **/etc/apache2**, the following files are worth–mentioning:

apache2.conf Provides the majority of tuning options available using the web server.

ports.conf Defines the port the web server is listening to. Default is 80.

envvars Sets the default connected user for the system. Default is **www-data** for both name and group.

There are two different place where aliases and scriptaliases can be set, under /etc/apache2/:

httpd.conf This is the general configuration file.

[2]Specifically, 2.2.8 version was tested.

sites–available/default This configuration is subject to each unique virtual host running within the web server

In the current installation, we will use the first option to install the document root and the cgi executables of GeoMoin and MoinMoin.
Finally, it is important to mention that the webserver can be managed using the following command:

apache2ctl -k [start/stop/restart]

Opening a browser and pointing to http://localhost will respond successfully.

A.1.4 MapServer and GDAL

In this section, we will move to the build and installation of GDAL geospatial library, Mapserver and mapscript. GDAL (Geospatial Data Abstraction Library), strictly speaking, is a translator library for raster data. It provides the ability to import and project georeferenced raster images. Generally, the GDAL contains OGR Simple Features Library which provides abstracted access to reading and some writing of a variety of vector formats.

Building and Installing GDAL

GDAL can be found at http://trac.osgeo.org/gdal/wiki/DownloadSource. In order to build the library, the following command must be issued (during the configure, the path to pg_config and geos-config may be required, so ensure the output of the configure step):

```
tar -xvzf gdal-1.5.3.tar.gz -C /usr/local/src/
./configure --with-python --with-pg --with-geos
make
make install
ldconfig
```

The –with-python and –with-pg options provide the python typemaps and the postgis driver. To test the installation:

```
user1@localhost:~\$ python
> from osgeo import gdal
> from osgeo import ogr
> exit()
```

Building and Installing Mapserver

Mapserver can be found at http://mapserver.org/download.html. In order to build mapserver:

```
tar -xvzf mapserver-5.2.1.tar.gz -C /usr/local/src/
cd /usr/local/src/mapserver-5.2.1/
./configure --with-proj --with-gdal --with-ogr --with-postgis
            --with-wfs --with-wms --with-wmsclient --with-wfsclient
            --with-geos --with-jpeg --with-png
make
```

Natively, the make command will output the cgi version of mapserver which is not needed in GeoMoin. Thus we can move on the essential part of the mapscript installation.

Building and Installing Mapscript

Mapscript source code is shipped within the mapserver package. The installation is straight–forward and can be done issueing these commands:

```
cd /usr/local/src/mapserver-5.2.1/mapscript/python
python setup.py build
python setup.py install
ldconfig
```

To test the installation, open a command–line python and issue: **import mapscript**. Additionally the further directory tests/cases contains test units for all the possible mapscript usages. An experienced user can debug the installation using **runalltests.py**.

A.1.5 MoinMoin wiki

In this chapter we will leave the GIS software realm, in order to install a fresh instance of the MoinMoin wiki in the system. We are interested in the CGI version of MoinMoin which will use the Apache web server as its host enviroment. The following installation guide, is a summarization of the complete and particularly useful guide found in MoinMoin's website http://moinmo.in. The source code of the release is currently in version 1.8.1 and can be downloaded from http://moinmo.in/MoinMoinDownload. The first step, is install the package in the desired path. In this guide, this path is **/usr/local/**. By default, **/usr** is used, but it is a cleaner approach to install user–defined packages to the local folder. Issuing:

```
tar -xvzf moin-1.8.1.tar.gz -C /usr/local/src/
cd /usr/local/src/moin-1.8.1
python setup.py install --prefix='/usr/local' --record=install.log
```

The following directory structure was created under /usr/local/:

lib/python2.5/site-packages/MoinMoin Contains the MoinMoin library, that can be imported using python.

share/moin Contains the shared files of the application (like htdocs for apache)

A.1.6 GeoMoin Installation

The installation of GeoMoin is a procedure that is highly depended to the packages described above. Issuing,

```
tar -xvzf geomoin-1.0.0.tar.gz -C /usr/local/src/
cd /usr/local/src/geomoin-1.0.0
```

the GeoMoin installation files are extracted to the requested path. In order to faciliate the installation, three shell scripts were created:

installsql Installs the GeoMoin database within the Postgresql DBMS.

installwiki Creates a new MoinMoin wiki instance. This instance will host the GeoMoin application.

installbase Installs the GeoMoin specific python modules, MoinMoin plugins and shared files.

Before executing any of the above scripts, the file named **install.conf** (located in the base path) must be edited to reflect the installation options. Parameters for the database creation, as well as, aliases and scriptaliases for the apache configuration concerning the wiki should be filled. Leaving the file, as–is, will hopefully achieve a proper installation but it is highly recommended to review all the changes that will happen to the current system state.

Installing the GeoMoin database

In order to install the database the user should issue:

```
su postgres
./installsql
exit
```

The su is required to execute the script commands as the postgres user who has native access to psql utility. The following installation step are executed within the script:

1. Create a postgis template database. Useful in order to create more than one postgis enabled databases.
2. Import lwpostgis.sql and spatial_ref_sys.sql to the database template.
3. Create the new role for accessing the database through GeoMoin.
4. Create the GeoMoin database.
5. Install the GeoMoin specific sql functions.
6. Install the GeoMoin default tables.

If a postgis–enabled DBMS already exists, only the final two step are required (check the script), as well as the proper changes in the configuration file.

Installing a MoinMoin wiki instance

In order to install the new wiki instance the user should issue:

```
sudo ./installwiki
```

The script copies the appropriate elements from the MoinMoin shared documents and creates aliases for the Apache environment. The installation procedure for apache cgi is perfectly described in the MoinMoin web–site under the url:

http://moinmo.in/HelpOnInstalling.

If a MoinMoin wiki already exists in the system, the configuration script must be edited in order to become aware of the installation. Additionally it must be ensured the MoinMoin version is 1.7.3 or above.

Installing the GeoMoin base system

In order to install the database the user should issue:

```
sudo ./installbase
```

The following installation step are executed within the script:

1. The GeoMoin shared files are installed including the cgi scripts and the javascripts.
2. The Apache aliases for the cgi and the shared files are registered.
3. The GeoMoin python modules are installed within the MoinMoin source package.
4. The configuration module used by all the modules is generated from the install.conf script.

Completing the installation

By now, the base system including the database, the wiki and the GeoMoin shared files and libraries is installed. Thus the wiki must be configured to enable GeoMoin. Using a web–browser, the default url to access the wiki is **http://localhost/mywiki**[3]. The wiki shows up and the user must create a new account. The next step is to set this account as the superuser of the wiki instance[4]; this is done manually by editing the **wikiconfig.py**[5] residing in the folder of the wiki instance which by default is **/usr/local/share/moin/mywiki/**. A commented line like:

```
#superuser = [u"YourName", ]
```

must be uncomment and include the name of the user that was previously created.
Secondly, it is for security reasons that the XMLRPC facilities are disabled by default in MoinMoin, thus we need to enable them. Within the **multiconfig.py**[6] around line **725** the **actions_excluded** parameter is declared. This parameter excludes actions unless an action is removed from the list. Thus the action **xmlrpc** must be excluded.
Finally, in order that the proper visual styling including css and wiki logo are set, the user must select **settings→preferences→Prefered theme** and choose **geomoin** as the theme to use. The wikiconfig.py contains an option that sets the default theme that new or anonymous users get. This is very important and can be set by locating the following line and changing it to "**geomoin**":

```
# The default theme anonymous or new users get
theme_default = 'modern'   # MUST BE SET TO "GEOMOIN"
```

[3]Depending on the aliases set in the config file.
[4]If a wiki already exists including the superuser, this step should be ommited
[5]This file reflects the configuration of this unique wiki instance, thus it contains a huge number of options that more advanced wiki users can utilize.
[6]The multiconfig.py contains various parameters affecting all the wiki instances and is located by default at: /usr/local/lib/python2.5/site-packages/MoinMoin/config

Bibliography

[1] David J. Buckey. Introduction to gis. URL `http://bgis.sanbi.org/gis-primer/index.htm`.

[2] Wikipedia. Wikipedia entry on wikis. . URL `http://en.wikipedia.org/wiki/Wiki`.

[3] Bo Leuf. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley, 2001.

[4] MoinMoin team. Moinmoin code structure help page. URL `http://moinmo.in/MoinDev/CodeStructure`.

[5] Wikipedia. Wikipedia entry on geographic information system. . URL `http://en.wikipedia.org/wiki/Geographic_information_system`.

[6] Philippe Rigaux, Michel O. Scholl, and Agnes Voisard. *Spatial Databases: With Application to GIS*. Morgan Kaufmann, 2001.

[7] Inc. Environmental Systems Research Institute. Esri shapefile technical description. Technical report, Environmental Systems Research Institute, Inc., 1998.

[8] Francis Harvey. *A Primer of GIS: Fundamental Geographic and Cartographic Concepts*. The Guilford Press, 2008.

[9] Shelly Sommer and Tasha Wade. *A to Z GIS: An Illustrated Dictionary of Geographic Information Systems*. Esri Press 2nd edition, 2006.

[10] nationalatlas.gov. Map projections: From spherical earth to flat map. URL `http://nationalatlas.gov/articles/mapping/a_projections.html`.

[11] USGS. Map projections. URL `http://egsc.usgs.gov/isb/pubs/MapProjections/projections.html`.

[12] Bugayevskly L. *Map Projections: A Reference Manual*. TAYLOR & FRANCIS, 1995.

[13] Wikipedia. Wikipedia article on map projections. . URL `http://en.wikipedia.org/wiki/Map_projection`.

[14] Wikipedia. Wikipedia article on geocoding. . URL `http://en.wikipedia.org/wiki/Geocoding`.

[15] JERRY H. RATCLIFFE. On the accuracy of tiger-type geocoded address data in relation to cadastral and census areal units. 2001. School of Policing Studies, Charles Sturt University, NSW Police College.

[16] Wikipedia. Wikipedia article on mapserver. . URL `http://en.wikipedia.org/wiki/MapServer`.

[17] Wikipedia. Wikipedia article on open geospatial consortium. . URL `http://en.wikipedia.org/wiki/Open_Geospatial_Consortium`.

[18] Wikipedia. Wikipedia article on open source geospatial foundation. . URL `http://en.wikipedia.org/wiki/Osgeo`.

[19] Bill Kropla. *Beginning MapServer: Open Source GIS Development*. Apress, 2005.

[20] Mapserver mapfile documentation. Technical report, OSGeo and MapServer community. URL `http://mapserver.gis.umn.edu/docs/reference/mapfile`.

[21] Pericles S. Nacionales. Mapserver 5.x tutorials. 2007. URL `http://biometry.gis.umn.edu/tutorial/`.

[22] Salah Juba, Vasilis Samoladas, Nikos Boretos, Ioannis Manakos, and Christos G. Karydas. Ims: a web-based map server for spatial decision support. Mediterranean Agronomic Institute of Chania, Department of Electronic and Computer Engineering, Technical University of Crete.

[23] Refractions Inc. Postgis 1.3.5 manual. URL `http://postgis.refractionds.net/documentation/manual-1.3/`.

[24] Inc. Open GIS Consortium. Opengis simple features specification for sql revision 1.1.

[25] ESRI (2003). Spatial data standards and interoperability. URL `http://www.esri.com/library/whitepapers/pdfs/spatial-data-standards.pdf`.

[26] Paul Bolstad. *GIS Fundamentals, a First Text on Geographic Information Systems, 3rd Edition*. Eider Press, 2008.

[27] Alex Martelli, Anna Ravenscroft, and David Ascher. *Python Cookbook*. O'Reilly Media, Inc., 2005.

[28] OSGeo and MapServer Community. Python mapscript documentation. Technical report. URL `http://mapserver.gis.umn.edu/docs/reference/mapscript/index_html`.