Masters Program in Geospatial Technologies

DESIGN AND IMPLEMENTATION OF 3D BUILDINGS INTEGRATION FOR A WEBGL-BASED VIRTUAL GLOBE

A Case Study of Valencian Cadastre and FIDE Building Model

Daniel Gastón Iglesias

Dissertation submitted in partial fulfilment of the requirements
for the Degree of *Master of Science in Geospatial Technologies*

UNIVERSIDADE NOVA ISEGI
UNIVERSITAT JAUME I
WESTFÄLISCHE WILHELMS-UNIVERSITÄT MÜNSTER

DESIGN AND IMPLEMENTATION OF 3D BUILDINGS INTEGRATION FOR A WEBGL-BASED VIRTUAL GLOBE

A Case Study of Valencian Cadastre and FIDE Building Model

Dissertation supervised by:

PhD Óscar Belmonte Fernández

PhD Joaquín Huerta Guijarro

PhD Pedro Cabral

March 2012

ACKNOWLEDGMENTS

I wish to express my deep gratitude to my supervisor PhD Joaquín Huerta for the time and effort he dedicated supporting me. I am also grateful to PhD Óscar Belmonte and PhD Pedro Cabral for their support and for accepting the responsibility of reviewing and commenting this document. I am grateful as well to Prodevelop workmates because of their general expertise working with JavaScript and GIS technologies.

My appreciation also goes to my Erasmus Mundus colleagues, for making this experience such a valuable and pleasant one.

Finally my appreciation and thanks go to my dearest parents and brothers, uncles Luis and J. Antonio, friends and girlfriend for their support, comprehension and help, which make this thesis possible.

Additionally I would like to thank Jaime I University and Prodevelop SL for their financial support to this research.

DESIGN AND IMPLEMENTATION OF 3D BUILDINGS INTEGRATION FOR A WEBGL-BASED VIRTUAL GLOBE

A Case Study of Valencian Cadastre and FIDE Building Model

ABSTRACT

Since nowadays Web applications are increasingly providing plenty of creative and interesting services relying on new standards and more powerful computers, it becomes important to create similar applications, to process and visualize geographic data taking advantage of such groundings. In this context, it results interesting to develop new Web-based geo-processing based on a 3D data representation, exploiting the recent WebGL graphic specification from a client-side point of view. This research explains the novel way in which whole Valencian cadastre was analyzed, processed and finally represented into a WebGL-based virtual globe. These improvements provide end-users firstly, an optimization of computer graphics performance, by natively accessing to graphics instructions; and secondly a functional data management and representation for the present and forthcoming geo-processing Web-based platform.

KEYWORDS

3D

Building Information Model

Building representation

Cadastre

Constrained Delaunay Triangulation

Extrusion

FIDE

Geo-process

Geographical Information System

Geospatial

GIS

HTML5

JavaScript

OpenGL ES

OpenSceneGraph

OSG

OSGEarth

OSGJS

ReadyMap

Scene Graph

Tile Map Service

Virtual Globe

WebGL

Web Map Service

ACRONYMS

3D	*Three dimensional*
AJAX	*Asynchronous JavaScript And XML*
BOE	*Official State Gazette (Spanish)*
CAD	*Computer-Aided Design*
CDT	*Constrained Delaunay Triangulation*
COLLADA	*COLLAborative Design Activity*
CSS	*Cascading Style Sheets*
DEM	*Digital Elevation Model*
DOM	*Document Object Model*
EPSG	*European Petroleum Survey Group*
FIDE	*Edification Exchange Data Format (Spanish)*
fps	*frames per second*
GDAL	*Geospatial Data Abstraction Library*
GIS	*Geographic Information System*
GPL	*General Public License*
GPS	*Global Positioning System*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IGN	*National Geographic Institute*
JS	*JavaScript*

JSON	*JavaScript Object Notation*
LGPL	*Lesser General Public License*
LOD	*Level Of Detail*
MIT	*Massachusetts Institute of Technology (License)*
OGC	*Open Geospatial Consortium*
OOL	*Object Oriented Language*
OpenGL	*Open Graphics Library*
OSG	*OpenSceneGraph*
OSGJS	*OpenSceneGraph JavaScript*
PHP	*Hypertext Preprocessor*
SDK	*Software Development Kit*
TMS	*Tile Map Service*
UTM	*Universal Transverse Mercator*
VBO	*Vertex Buffer Object*
WebGL	*Web-based Graphics Library*
WGS	*World Geodetic System*
WMS	*Web Map Service*
XML	*Extensible Markup Language*

INDEX

INDEX OF TABLES

INDEX OF FIGURES

Chapter 1

Introduction

1.1 Background

The widespread adoption of the World Wide Web has greatly changed the way that the current software is being developed, to the point that a big amount of new desktop-based software applications, are written for the Web instead of conventional computing architectures. Perhaps, the most common case where this trend is clearly noticed is Google Docs, which is a perfect example of how a traditional desktop application has successfully migrated to a Web-based environment.

In general terms, the software industry is shifting towards Web-based architectures (Taivalsaari, Mikkonen, Anttonen, & Salminen, 2011). In this type of Web-based software, applications rest on the Web as services; consisting of data, from a general point of view, that might be located anywhere in the world and not particularly in each of the computers as a conventional desktop approach.

Web-based software has some advantages over a desktop-based one. It requires no installation or even manual upgrades and it also, leverages what the Web means in terms of interaction and user sharing. This therefore combines the best of both worlds: the excellent usability of conventional desktop applications and the enormous potential of the World Wide Web (Taivalsaari, Mikkonen, Anttonen, & Salminen, 2011). Web-based software arrives with a new range of unexplored possibilities intrinsically related with new appearing market demands, which predicts an optimistic future on the field for forthcoming years. Additionally, end-user software security is improved since all information accesses follow the same traced routes for all the variety of users, therefore leveraging both data oneness and backups. Furthermore, Web applications can be deployed and shared instantly worldwide, with no middlemen or distributors, inciting to a increasing and already present, open source software development.

The trend for applications to shift to web-based software is gaining plausibility as new Web application features are introduced, that enrich the web browser with additional programming functionalities and interfaces.

An especially important step in this direction, along with the arrival of the HTML5 standard (W3C, 2011), is the introduction of WebGL (Khronos Group, 2012) as a standardized part of HTML5 (Lubbers, Albers, & Salim, 2010). WebGL is a graphic standard that enables the creation of three dimensional (3D) content that can natively run efficiently in a standard Web browser without any plug-in components or extensions (Taivalsaari, Mikkonen, Anttonen, & Salminen, 2011).

The possibility to natively add and graphically display 3D objects into existing Web applications is probably, the most outstanding change the Web is going to adopt for quite a while (Bochicchio, Longo, & Vaira, 2011). Displaying 3D graphics content on the Web has been possible in the past with APIs such as Flash, O3D, VRML and X3D (Ortiz, 2010), but only with certain browsers or if the necessary browser plugin was installed.

Nonetheless, considering the adoption of WebGL by most major browser developers, 3D capabilities are integrated directly into the web browser from now on. This means that 3D content can run smoothly in any standard compliant browser without the installation of applications or additional components. WebGL support has already been implemented and included in some of the forthcoming new versions of the most well-known browsers. However, most browsers demonstrate a lack of full functionality, with just Google Chrome and Mozilla Firefox providing the best support to date.

In the long term, the most important impact of WebGL may be its capabilities as a gaming environment. Even if most of people have began migrating to web-based content, up to the present it has been very difficult to convince hard-core game developers and players to take web-based software seriously. This is partly owing to suitable development APIs were not available, and partly because, up to this time, the execution speed of web-based software was completely inadequate for CPU-hungry gaming applications.

Considering the enormous income that the computer games sector generates, with a retail sales of 10.5$ billion dollars in 2009 (The Entertainment Software Association,

2011), expecting 70$ billion dollars by 2015 (DFC Intelligence, 2010)), and the widespread user immersion it entails, a fast and productive development of WebGL-based Web applications is expected.

Nevertheless, the use of WebGL is not explicitly limited to game applications. Concerning the area this project deals with, Geographic Information Systems (GIS), yet again WebGL is present. Web GIS has become a cheap and easy way to promote the wide-spread use of geospatial data, thus many software developers have paid attention and interest, to the distribution of both maps and tools that the general public can use without restrictions.

One of the most impressive and clarifying examples is the case of the outstanding Nokia Maps (Nokia, 2012). Nokia Maps —also known as Ovi Maps— is a free mapping service developed by Nokia for its mobile phones and Smartphone multimedia devices, which is also available on a website for all major browsers. In the earlier stages of development, it was released with a direct end-user obligation to download and install a proprietary plug-in —provided that such a plug-in was browser-compatible— in order for the user to properly visualize 3D content in the browser. However, due to the extensive work of both browser software developers and graphic cards driver updates, Nokia Maps —although still in beta version to date— has already migrated to WebGL, so as to show the same 3D content in a more optimized way.

It should be noted that the use of WebGL is not only limited to the development of 3D applications. WebGL will serve as an important feature in the development of more conventional desktop-style 2D applications as well. In fact, it is quite possible that WebGL will effectively replace the use of the earlier 2D canvas drawing API that has been supported by many Web browsers. Yet again, and similar to previous Nokia Maps example, Google Maps has migrated —within a 2D context this time— to WebGL (Google, 2012) Considering the calibre of high profile companies that have directly bet for WebGL-compliant applications' development, this once again demonstrates the importance of this graphic standard.

Most of the time, accessing to very low-level instructions as the ones that WebGL provides, is an arduous work, so that middleware software is often used to ease graphic software development tasks. As it has been depicted, graphical applications are in quite

demand in computer science today, and OpenSceneGraph (OSG), as a 3D graphics toolkit, is being widely used in fields such as virtual reality, scientific visualization, visual simulation, modelling games, mobile applications and GIS, which is the case in this present research.

OSG is, in fact, a rendering middleware application. It is a solution that raises the level of abstraction and eases the complexity of using a low-level WebGL 1.0 API, at the cost of flexibility. The concepts of modularity and object oriented programming are often applied to manage graph primitives, materials, and different data sets in visual user applications as the one concerning in this project. The application of these concepts saves much development time and makes it possible to combine new functionalities, such as modules and plug-ins.

Furthermore, OSG is actually a deferring rendering system based on the theory of scene graph, which records rendering commands and data in buffers, for a posterior rendering (Policarpo & Fonseca, 2005). This fact permits the system to perform several optimizations before rendering, as well as implementing a multithreading strategy for handling complex scenes (Wang & Quian, OpenSceneGraph, 2010).

A scene graph is a tree data structure that defines both a spatial and logical relationship of a graphical scene, for efficient management and rendering of graphic data (Guangwei & Zhitao, 2009). It is typically represented as a hierarchical graph, which contains a collection of graphic nodes including a top-level root node. A typical scene graph does not allow a directed cycle —where some nodes are connected in a closed chain— or an isolated element —a node that has no child or parent— inside itself.

Consequently, the use of scene graphs, and particularly the use of OSG, fits completely in the accomplishment of this project, not only for the current project development, but also for forthcoming releases. Expressly for the aim of this project, OSG perfectly fits in rendering big amount of buildings in real-time, which is actually a crucial part of this research.

This research focuses on an innovative way of building's attribute representation based on client-side architecture. Relying on this context, it is implemented the representation stage of a bigger project focused on the processing of precise building energetic demands. This processing relies not only on modelling and designing technologies, but

also on the analysis of 3D buildings locations in which this research has given solution. Consequently, the estimated building's energy consumption and corresponding emissions to the atmosphere are obtained, helping to minimize energetic consumption from a passive point of view.

Both commercial and residential edifications represent nearly a 40% of the world's energy consumption and in a similar level of CO2 emissions (Rodriguez, 2010). Nevertheless, regarding "Contribution of Working Group III to the Fourth Assessment Report of The Intergovernmental Panel on Climate Change"; the mentioned buildings sector represents the energetic sector with the highest potential in terms of energy reduction —estimation of 29% for 2020—(Barker T., 2007). Additionally, it is cited that is possible to achieve a 75% of energy savings, in individual new buildings compared with recent current practice, and generally at little or no extra cost (Barker T., 2007).

Such achievement requires the adoption of additional methodologies and technologies. The cited report enumerates the main ones, pointing out that the most energy saving occurs in new edifications. Among the cited technologies, it is stated in the first place the importance of a passive solar energy production as a technology with a highest applicability nowadays. According to that, this research will notably help on the analysis of solar incidence, implementing the representation stage of a building energetic demand processing.

1.2 Hypothesis

The development of a WebGL-based building's visualization application will provide a new Web-based geo-processing platform publicly accessible; creating a new GIS processing architecture based on a Web browser interactivity, interoperability and improved graphical performance.

1.3 Objectives and Limitations

Considering the mentioned background, and based on the hypothesis, this research is going to focus on the way to integrate and visualize 3D buildings coming from cadastre and FIDE data, into a WebGL-based globe.

Following, it is listed the objectives of the thesis research.

- To study in-depth the adopted Earth globe solution based on the WebGL standard.
- To use public cadastre data of the city of Valencia (Spain) provided by Spanish Authorities.
- To study, understand and use WebGL standard, based on OpenGL ES 2.0, for low-level instructions accessing.
- To study, understand and use GLSL as graphical language.
- To study, understand and use JavaScript language.
- To study, understand and use OSG and OSG Earth framework based on scene graph representations.
- To study, understand and use HTML and CSS as supporting languages for the suitable Web content visualization.
- To study, understand and use WMS and TMS protocols for request tiles to the correspondent servers.
- To propose a model to prepare, analyze and filter the data coming from the Spanish cadastre.
- To propose a model to represent 3D buildings derived from 2D geo-data, on the surface of a WebGL-based virtual globe.
- To represent not only the buildings' geometry but any interesting parameter present or not present on the treated data.
- To develop a fully functional Web application prototype for both testing and representing the buildings present in obtained cadastre data.
- To develop a complete solution for representing the FIDE buildings' models in the same way as done for cadastre data (See Appendix B).

GIS data representation and the consequent end-user interaction, is a large topic itself that has many issues to be considered. However, mainly due to the time reasons, many of these issues are not included into this research scope. Following, it is presented some of the limitations considered towards this research:

- This research does not intend to develop a complete GIS system, but an implementation of geo-data representation in a WebGL-compliant way.
- This research does not intend to develop any geo-process or analysis itself moreover than representing geo-data in an innovative way. Nevertheless it

establishes the way to represent any kind of results originated in third geo-processing, in a WebGL environment.

- This research focuses on using already implemented scene navigation functionalities, instead of developing new ones besides the firstly mentioned.

1.4 Dissertation Organization

The structure of this research consists of six major chapters which are distributed as *Introduction*, *State Of The Art*, *Analysis*, *Design and Implementation*, *Results* and finally *Conclusions*. Additionally, it has been included *Appendix A* and *Appendix B* to this research.

Introduction chapter reviews current and forthcoming technologies intrinsically related with foreseeable use trends, in addition to the reasons and objectives of this research. Straight after, *State Of The Art* chapter introduces current WebGL-based globes and a group of available JavaScript-based libraries that may give support to WebGL-based GIS developments.

Analysis chapter provides the technical reasoning about the application's requirements and processing the system needs to accomplish, as well as the temporal-economical planning. *Design and Implementation* chapter explains in-depth, the implemented solution making extensive use of graphical contents in order for the reader to clear understand it. Additionally, it is further divided in *Data Preparation* and *Data Visualization* sections, in consequence of relying on different used and implemented applications.

Results chapter describes in detail the obtained application's outputs, giving solutions to the hypothesis and research's objectives posed in this *Introduction* chapter. Finally *Conclusions* chapter presents the general achievements of the research. It summaries the contributions in knowledge provided by the research and suggests future directions and recommendations for the developed system.

Additionally, *Appendix A* describes the resulting outputs coming from different tested configurations. And finally, *Appendix B* describes the entire FIDE approach which provides roughly same functionalities as the implemented for the cadastre case.

1.5 Chapter Review

This chapter begins with a brief background strengthening the importance of WebGL graphic standard and Web-based applications, followed by the importance of Web-based geo-processes. Finally, explains that the background behind this research is the processing of precise buildings energetic demand, relying on the buildings geo-location. Additionally it describes a set of research's objectives, concluding with a section on the research's structure.

Chapter 2

State Of The Art

2.1 Introduction

General purpose Web applications, and more specifically GIS Web applications, have become more important around the world due to the existence of an interesting and useful geographical component. This is attached in some way to most of the data that an end-user either directly or indirectly deals with.

The recent appearance of HTML5 and WebGL standards, provide a wide range of new possibilities and market researches in mainly all sectors and areas, including GIS as this research does (Taivalsaari & Mikkonen, The Web as an Application Platform: The Saga Continues, 2011).

There are several applications where the ability to capture, display, and visualize 3D images comfortably would confer real benefits. Examples from the professional domain include CAD design, medical diagnostics, scientific-purpose visualization, and GIS visualization methods.

In the long term, the third dimension provides a better comprehension and modelling of the world around us, adding both an extra modern looking and an ability to represent specific parameters such as building or terrain elevations, which cannot be visualized from a conventional top-down view.

As GIS concepts are getting accepted and assimilated by the society, more Web applications are released, describing in many cases fully operational 3D globes. Until now, the only way to achieve such 3D representation, was implementing a Web application relying on the private-source Google Earth API, which as usual, required a plugin installation as a browser extension.

However, due to operating system security issues, administrator permissions, or even license concerns, many end-users have found it not suitable as a software development

framework. That is one of the reasons why some other solutions are increasingly appearing. The principal aim of most of the appearing Software Development Kits (SDK), is simply to provide an alternative solution, implemented mostly in JavaScript to be available in all WebGL-compliant browsers within open source environments.

Nevertheless, since WebGL is a rather recent standard specification, there is a little research found, encountering no fully complete solution done so far. However, due to market needs, trends, and technologies progressing, some WebGL globes are timidly appearing to date.

Each of selected and following presented WebGL globes adopts different structure and design, letting either an end-user or software developer to choose the one which fits the most his requirements, likings or purposes.

The encountered solutions are further divided into two separated groups in order to clarify the State Of The Art of this research. The first part, *3D Map Toolkits*, concerns Map Toolkits and their particular characteristics, detailing as well some other 3D related tools available on the Web, *3D on the Web*. The second part, *JavaScript 3D Render Engine Frameworks*, mentions and details the main 3D JavaScript libraries existent on the market, regarding special attention on the library selected in this research.

2.2 3D Map Toolkits

2.2.1 ReadyMap

(Pelican Mapping, 2011)

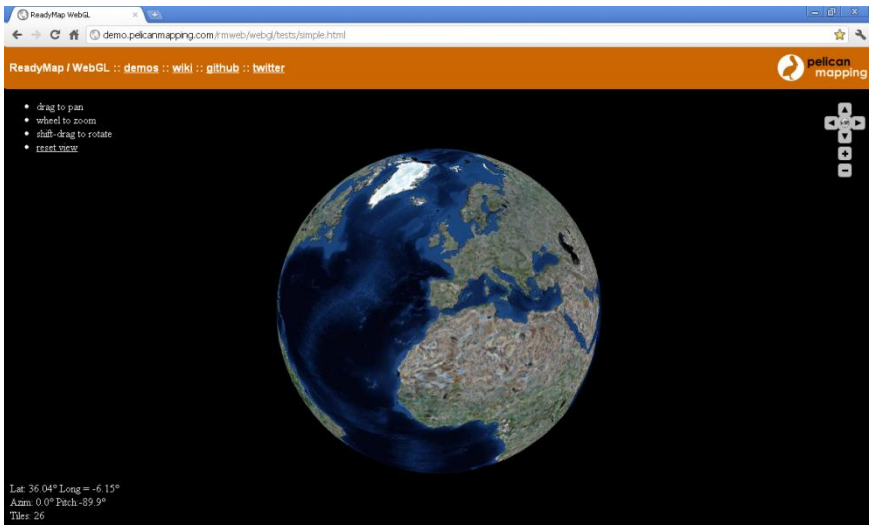


Figure 1. ReadyMap WebGL globe's appearance

ReadyMap globe has been developed under GNU LGPL license, by Pelican Mapping, a company settled in Fairfax, (USA). ReadyMap is a complete suite of tools for publishing and viewing maps in 2D and 3D, designed for desktop, mobile and Web environments. This last one is where WebGL SDK takes place.

ReadyMap WebGL SDK is a free JavaScript library that can be used to embed maps in a web page. It supports both 2D and 3D global maps, letting the user to geo-locate any kind of event on the map —GeoRSS or Flickr i.e.—, as well as manipulate the data and the on-screen camera perspective. Additionally it works well with open Web-mapping standards like TMS (Tile Map Service) and WMS (OGC Web Map Service), leveraging WebGL technology in order to render high-performance 3D maps.

The toolkit is built over OpenSceneGraph (OSG) library (OSG, 2012) (see Figure 2). OSG is an open source high performance 3D graphics toolkit, used by several applications' developers in fields so varied as visual simulation, games, virtual reality, scientific visualization, modelling and GIS.

Detailed next, it is presented a general schema of the ReadyMap structure.

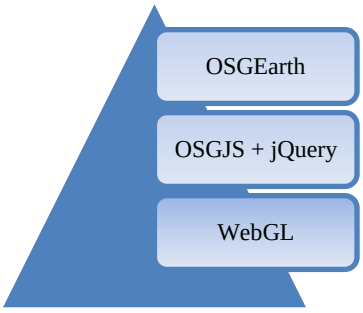


Figure 2. ReadyMap's architecture

OSG relies on the scene graph groundings. A scene graph is a tree data structure that defines the spatial and logical relationship of a graphical scene, for efficient management and rendering of graphic data. It is typically represented as a hierarchical graph, which contains a collection of graphic nodes; including a top-level root node, and a number of group nodes that serve together as the bottom layer of the tree. Additionally, each group node may have any number of children. Thus, these children nodes gather the information of their parent and can treated therefore as one independent unit.

OSG is written entirely in Standard C++ and OpenGL, running on a wide range of operating systems including Windows, OSX, GNU/Linux and Solaris

Due to OSG widespread, and also to both market research and trends, OSG has been gradually migrated to JavaScript being renamed "OSGJS". OSGJS is a WebGL framework based on OSG groundings, which makes possible firstly, to use an "OpenSceneGraph-like" toolbox which interacts with OpenGL via JavaScript calls, and secondly to export data models to the OSGJS format (OSGJS, 2012). Additionally, on top of OSGJS, ReadyMap makes extensive use of OSGEarth.

OSGEarth is a C++ terrain rendering toolkit that supports several kinds of data providing visualization and manipulation functionalities in many different ways. Concurrently as OSG, OSGEarth has been gradually ported to JavaScript as well, getting fully integrated into ReadyMap solution (OSGEarth, 2012).

Both, library and toolkit are fully merged with jQuery, the third component that ReadyMap relies on. jQuery is a JavaScript library that provides to ReadyMap a full-HTML-element processing binding ReadyMap objects to HTML5 tag elements.

2.2.2 WebGL Earth

(Klokan Technologies, 2011)

Figure 3. WebGL Earth globe's appearance

WebGL Earth is open source software available under standard GNU GPLv3 license, developed by Klokan technologies settled in Baar, Switzerland. It is designed for the

visualization of maps, satellite imagery and aerial photography on top of a virtual terrain using WebGL standard specification similarly as ReadyMap.

WebGL Earth provides some camera-dependent functionalities like rotation, zooming, or tilting which notably increases the user interactivity. Additionally it provides map visualization capabilities making the display of existing map solutions — OpenStreetMap and Bing i.e. — possible. Additionally, it is implemented a full support to custom Earth's map tiles loading. It also includes marker addition capabilities and viewpoint animation likewise the rest of solutions.

WebGL Earth is prepared to be used on third party websites in a very easy way, since developers have two options on how to use the source code for custom Web applications. On the one hand, it is possible to use the API so as to obtain a full customized application, alternatively on the other hand, it can be used the codebase with an aim of inserting a simple default map into a desired Web page.

In order to get ready to use WebGL Earth's SDK it is compulsory a prior setup of a Python Interpreter and Java Virtual Machine in order to successfully compile WebGL Earth source code, as well as the latest Closure Library.

2.2.3 OpenWebGlobe

(Geomatics Engineering departement at the University of Applied Sciences Northwestern Switzerland, 2011)

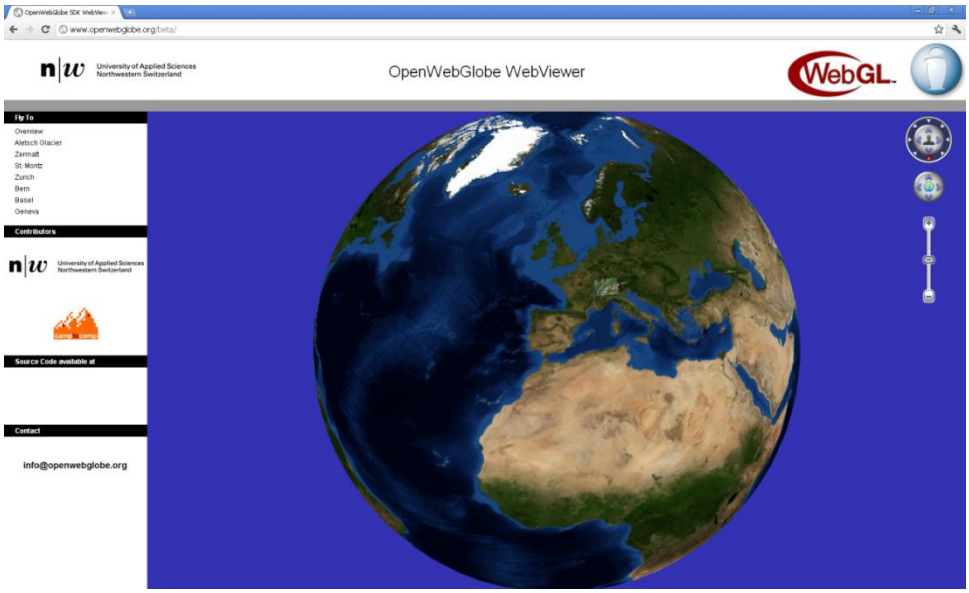


Figure 4. OpenWebGlobe globe's appearance

OpenWebGlobe has been created under MIT license by the Geomatics Engineering department at the University of Applied Sciences North-western Switzerland, settled in Basel, Switzerland.

OpenWebGlobe SDK is designed in an object oriented manner. The SDK is written entirely in C++ and has bindings to other languages like Python, C# and Visual Basic or JavaScript.

The solution the Geomatics Engineering department offers consists of, a WebGL globe that encompasses the software for processing very large volumes of geospatial data, in highly parallel and scalable computing environments. Additionally, it provides support for several data categories such as image data, elevation data, point of interest, vector data, and 3D objects.

Before streaming over the Internet, this data must be pre-processed, usually by comprising a transformation from a local to a global reference system, creating a set of pyramid layers, setting up levels of detail or tiling the data.

The latest released version to date is essentially a beta version with some of the functionalities spread out over the code not considered as a full complete solution.

2.3 3D on the Web

2.3.1 Google WebGLGlobe

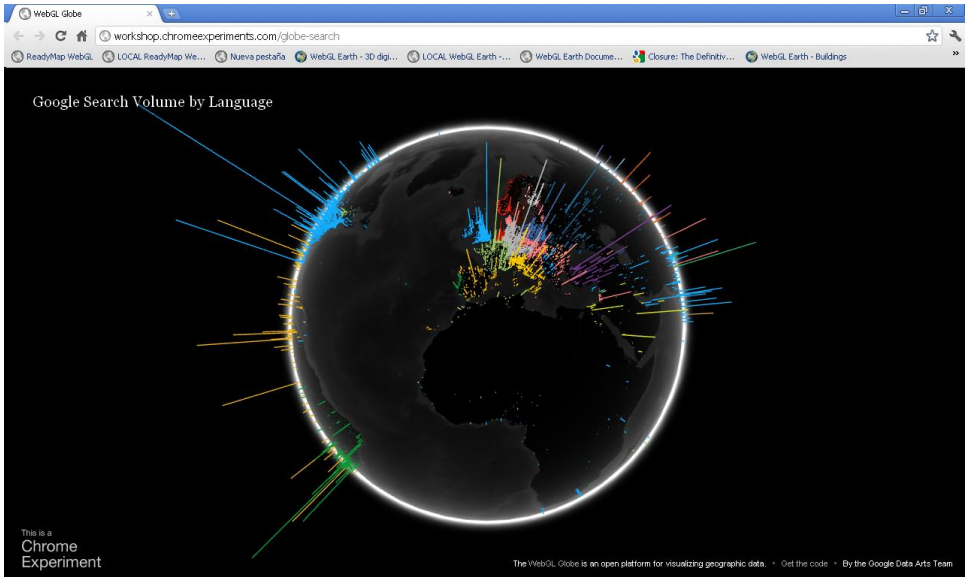


Figure 5. Google WebGL globe's appearance

In this section, the principal globe for data representation purposes based on WebGL is Google WebGL globe, within Chrome experiments framework.

WebGL Globe is an open platform for strictly geographic data visualization, created by the Google Data Arts Team, having as main features Latitude / longitude data spikes, colour gradients based on data value or type, and mouse panning and zooming functionalities.

In terms of technological dependence, WebGL Globe is based on both *Three.js* JavaScript 3D render engine, and "requestAnimationFrame.js" JavaScript file.

On top of the mention dependence, both JSON-described data and JavaScript-based shader files are loaded by the application, providing the globe the necessary data and the way the graphic card behaves to finally represent the processed data (Google, 2011).

Unlike previous mentioned Web applications, Chrome experiments, and WebGL globe involves a controversial terms of use, where by uploading code, the developer acknowledges and agrees that Google, or Google's licensors, own all legal right, including any intellectual property rights which subsist in the services (Google, 2012).

2.3.2 PhiloGL

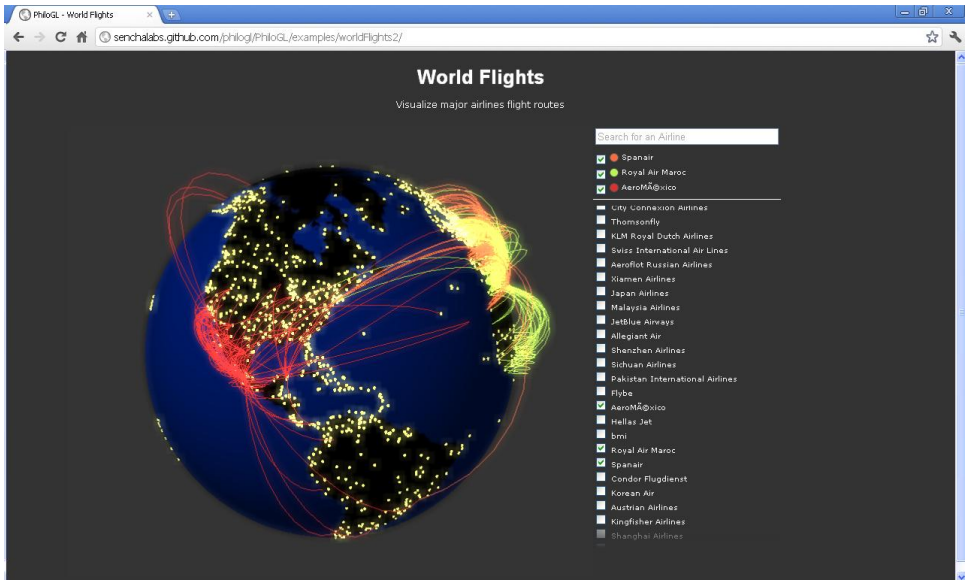


Figure 6. PhiloGL globe's appearance

PhiloGL is open source software, licensed under the MIT license and owned by the Sencha Labs foundation, that provides a WebGL framework for data visualization,

creative coding and game development. PhiloGL code is highly abbreviated compared to low level WebGL specification. This helps the users to build simple applications in a very little time only for data representation purposes (Sencha Labs, 2011).

2.4 WebGL Globes Comparison

Next, it is detailed a comparison's table between the main functionalities present in the variety of explained WebGL globes' solutions, followed by an explanation of each of introduced terms.

	ReadyMap	WebGL Earth	OpenWebGlobe	Google WebGLGlobe	PhiloGL
Viewer Functionality					
WebGL compatibility	Yes	Yes	Yes	Yes	Yes
Navigation					
Zoom In / Out	Yes	Yes	Yes	Yes	yes
Zoom Area	No	No	No	No	No
Zoom Full Extent	Yes	No	No	No	no
Zoom Predefined	Yes	No	No	No	no
Zoom Layer Extent, Next, Previous, Selected, X/Y	No	No	No	No	No
Bookmarks	Yes	Yes	No	No	no
Tilt	Yes	Yes	Yes	No	No
Pan	Yes	Yes	Yes	Yes	Yes
Data					
Add Local Layer	No	No	No	No	No
Add WMS	Yes	Yes	N/A	No	No
Add TMS	Yes	Yes	Yes	No	No
JSON	Yes	Yes	Yes	Yes	Yes
Other Data Formats	Yes (OSGJS)	No	No	Yes (COLLADA)	No
Selection					
Picking	No (labs)	No	No	No	No

Measuring					
Distance	No	No	No	No	No
Area	No	No	No	No	No
Angle	No	No	No	No	No
Coordinate display	Yes	Yes	No	No	No
Display Units	yes	Yes	No	No	No
Querying					
View Attributes	Yes	Yes	Yes	Yes	Yes
Find Address	Yes	Yes	No	No	No
Find Feature	No	No	No	No	No
Locate Feature by Query	No	No	No	No	Yes
Create Custom Query	No	No	No	No	No
Terms of Use					
License	GNU LGPL	GNU GPLv3	MIT	Owns all legal rights	MIT

Table 1. WebGL globes' comparison table

Navigation

- *Zoom In / Out*: zooms in and out to a specified centre point by a predefined magnification factor.
- *Zoom Area*: zooms in to a specified rectangular area extent.
- *Zoom Full Extent*: redraws so that the full extents of the map are displayed in the current window.
- *Zoom Predefined*: redraws so that a particular preselected zoom is applied.
- *Zoom Layer Extent, Next, Previous, Selected, X/Y*: additional zoom operations.
- *Bookmarks*: adds a bookmark to save the place with its particular coordinates.
- *Tilt*: moves the map so that the normal to user's view perspective is changed.
- *Pan*: moves the map around the map window in order to display areas that are outside of the current view extent, without changing magnification, just by dragging or by specifying a new view's centre point.

Data

- *Add Local Layer*: adds a new layer to the map from the local machine or network.
- *Add WMS*: Web Map Service support.
- *Add TMS*: Tile Map Service support.
- *JSON*: JSON data exchange format support.

- *Other Formats*: availability of other data exchange formats.

Selection

- *Picking*: selects a feature on-map based on geographic coordinates, relying upon screen coordinates selection.

Measuring

- *Distance*: measures a polyline (segments and total distance) drew by the user.
- *Area*: measures a polygon drawn by the user.
- *Angle*: measures an angle drawn by the user.
- *Coordinate Display*: displays X - Y coordinates of a specified point or points.
- *Display Units*: specifies units for displaying screen cursor position (e.g. Lat/Long or Projected).

Querying

- *View Attributes*: displays attribute data (from database) for a selected feature.
- *Find Address*: zooms to a location based on street address matching.
- *Find Feature*: zooms to a feature with a database attribute value matching a user-specified value.
- *Locate Feature by Query*: zooms to a feature with a database attributes matching a predefined query.
- *Create Custom Query*: builds database query to find features matching user-specified criteria.

Terms of Use

- *License*: concerns the terms for each of the Web applications.

2.5 JavaScript 3D Render Engine Frameworks

In this section it is explained the State Of The Art regarding to WebGL-based JavaScript libraries. Considering that many of the following JavaScript libraries are very recent releases, and the majority are still under early development, in many cases the information they have provided has been sparse or even inexistent.

Nonetheless, most of them are well documented with live examples which help for the user to get an overall idea about the graphic performance and the principal functionality that may be obtained from them.

2.5.1 OSGJS

OpenSceneGraph-JavaScript (OSGJS) is a library for modelling scenes as graphs. It attempts to transport the C++ based OpenSceneGraph functionality to JavaScript. Nevertheless, the support is incomplete. This can lead to difficulties in implementation when developers port code utilizing this library from C++ to JavaScript (OSGJS, 2012).

2.5.2 PhiloGL

PhiloGL is a JavaScript WebGL-based framework which tries to be as close to low level OpenGL instructions as possible, providing a yet tight abstraction to WebGL, focusing particularly on performance. PhiloGL also provides a module system covering Program and Shader management, JSON support, Web Worker management, Effects and Tweening (Garcia Belmonte, 2011).

2.5.3 SceneJS

SceneJS is a WebGL-based 3D engine geared towards the rendering of large numbers of individually pickable and articulated objects as required for engineering and data visualization applications.

On the inside, SceneJS is a fast draw list that is optimized for things like fast redraw and picking, while on the outside it is a convenient JSON-based scene graph API, that is easy to hook into the rest of the application stack (Kay, 2009).

SceneJS is a lean rendering kernel which omits functionality such as physics and visibility culling. However, its leanness makes it very efficient to update the scene state, making it practical to bolt on your choice of third-party toolkits such as TweenJS, JSBullet and jsBVH.

2.5.4 Three.js

Three.js is a lightweight 3D engine with a very low level of abstraction. Currently the engine can render using HTML 5's canvas, WebGL or SVG (Mrdoob, 2012). It is being actively developed in Github repository, leading a promising set of interesting forthcoming functionalities.

2.5.5 SpiderGL

SpiderGL is a JavaScript 3D graphics library which relies on WebGL for real-time rendering. The philosophy behind SpiderGL use, is to provide typical structures and algorithms for real-time 3D graphics rendering to Web developers, neither forcing them to comply with some specific paradigm —i.e. scene graphs— nor preventing low level access to WebGL (Visual Computing Laboratory at CNR Research Area of Pisa, 2012).

2.5.6 C3DL

The Canvas 3D JS Library (C3DL) is a JavaScript library that makes possible for software developers to write 3D applications relying on WebGL specification. It provides a set of math, scene, and 3D object classes, that makes WebGL more accessible, implementing the access to low level WebGL instructions (CATGames Research network at Seneca College, 2010).

2.5.7 CopperLicht

CopperLicht is a JavaScript 3D engine for game's design purposes ultimately focused to Web browser environments. CopperLicht includes a 3D world editor supporting the major 3D file formats as 3ds, obj or lwo for instance. Considering license terms, CopperLicht is free to use excepting for commercial purposes (Ambiera, 2010).

2.5.8 CubicVR

CubicVR is a lightweight, high-performance and implicit WebGL engine with a versatile collection of built-in features for quick implementations, providing functionality from a low-level use to full managed scenes and events with physics. It includes support for COLLADA format in both XML and JSON format; additionally it imports a variety of 3D models, animations and asset libraries (Cliffe, 2011).

2.5.9 GammaJS

GammaJS is a JavaScript library which can be used to develop 2.5D platform games for Web using WebGL graphic standard (Moore, Townsend, & Campbell, 2011).

2.5.10 GLGE

GLGE is a JavaScript library which wraps low level instructions into predefined instruction, easing the designing and developing of 3D content (Brunt, 2010).

2.5.11 Jax

Jax is a JavaScript development toolkit, which provides a wide range of functionalities, from code's generators to test suites designed specifically for WebGL applications development. It also provides support for both low-level WebGL instructions and JavaScript user interface interactivity, such as keyboard input or mouse handling (Jax, 2012).

2.5.12 O3D

O3D is a JavaScript API for creating Web applications in the browser. Originally built by Google as a browser plug-in (deprecated nowadays), this new implementation of O3D has been converted into a JavaScript library implemented on top of WebGL. Additionally, O3D provides a sample COLLADA converter, which can be used to both import files in the COLLADA format and write converters for other formats (Google, 2011).

2.5.13 Oak3D

Oak3D is a JavaScript framework for 3D graphics development based on WebGL standard. It is focused on optimizing the graphic 3D Web application's performance with GPU acceleration as the rest of mentioned libraries. Oak3D provides a set of simple and easy-to-use API interface which makes possible to software developers to implement 3D Web-based applications without considering the details that any low-level 3D graphics implementation entails (Oak3D, 2012).

2.5.14 X3DOM

X3DOM is a framework for integrating and manipulating X3D scenes as HTML5-DOM elements, which are ultimately rendered via WebGL, letting the software developer to manipulate existent 3D content by only removing, adding or changing DOM elements (Fraunhofer IGD, 2012).

2.5.15 Inka3D

Inka3D is an export plugin for Autodesk Maya. A wide range of features is supported including driven keys, shading networks, lighting and skinning, adding extra functionality to the scene's geometry. Moreover, all exported Maya parameters such as positions, rotations, scales, colours or even user defined extra attributes can be controlled using JavaScript (Inka3D, 2012).

2.5.16 KickJS

KickJS is an open source WebGL-based game engine built for modern web browsers such as the most recent version of Chrome and Firefox. KickJS provides a game oriented abstraction of WebGL, and makes game programming simpler and more accessible (KickJS, 2012).

2.6 Reasons for ReadyMap Choice

At this point, it is important to clarify that all the introduced solutions have been under early development by the time of this research, so that it has not been an easy task to choose one solution among many, also considering the amount and kind of similarities.

Nevertheless, relying on scene graphs concept has finally carried more weight since it has been considered a turning point for future releases, based on an increasing scene complexity (Wang & Quian, Why OSG?, 2010). Consequently, ReadyMap solution has been finally chosen mainly relying on:

- Improved performance: A set of scene graph techniques are already implemented in OSGJS, including view-frustum or occlusion culling besides a complete encapsulation of WebGL extensions and Graphics Library Shading Language (GLSL). Being able to treat the whole scene as a tree turns out into a great advantage since both parent and children hierarchical management is possible, optimizing costly graphical operations and WebGL changes of state.
- DOM access: Considering the jQuery community is a lot bigger than Closure Library; and more third-party plugins have been developed for jQuery than for others, it has been opted to use jQuery instead of the Closure Library present in some of the introduced solutions. Additionally, the developer of this research

has had previous experience with jQuery, carrying more weight this option among many.

- Active community: An active ReadyMap developer community makes possible to use the latest released functionalities as well as resolving eventual difficulties that may appear while implementing customized applications.
- License: Since it is an open source application, users and companies do not have to worry about software patent violations when using ReadyMap. This particular feature is also present in the majority of introduced solutions.
- GIS capabilities: It is expected a gradually migration of all the well-known OSGEarth functionalities to ReadyMap SDK. So far, GIS functions and transformations have been extensively implemented, nevertheless, a full migration would let ReadyMap gain more GIS support, defining a wide range of new geo-processing for client-side and server-side.

2.7 Chapter Review

This chapter has explained in-depth the State of The Art on WebGL-based globes, and the corresponding comparison between them. Additionally, it has been described a complete list of available JavaScript-based 3D libraries that may provide graphical solution to both data management and representation. Finally, it has been explained why ReadyMap has been chosen among the meagre variety of implemented solutions.

Chapter 3

Analysis

3.1 Introduction

In this chapter it is presented the analysis process accomplished at the beginning of this research, in order to establish which are the competences, features and functionalities that ReadyMap-based buildings' application has to provide to end-users. This section firstly describes application's requirements with regards to hardware, software and used data; straight after, it presents the corresponding use case model, followed by the high level architecture diagram where overall applications' dependences are featured. Additionally, it is presented the Gantt diagram and the research's budget diagrams, describing major and minor terms with regard to both time development and cost parameters.

3.2 Requirements

3.2.1 Introduction

For the correct development of this research, it has been necessary to satisfy the series of requirements following explained in this chapter. These requirements have mainly involved data and hardware-software user's configuration.

Additionally, it has been described in-depth the configuration set on the tested unit, in order to for the reader to help reproducing the testing environment, just in case it might be interesting.

3.2.2 Data acquisition

Considering that the case study has been located in the Valencian community, it has been necessary to obtain from the Spanish cadastre, the corresponding dataset with the area extent that fitted research's purposes. This data have been described as the cadastre data corresponding to Valencia city bounding area.

Considering that it was expected a huge amount of information present in the obtained dataset, it has been required to pre-process the data before represent it on-screen. Pre-processing has implicated the retrieving of necessary attributes for forthcoming processing.

Post-processed data must be stored in the server-side, and defined in an easy-well understood file format that a browser could recognize and later process. Thus, it has been expected that the browser once had retrieved the corresponding data from the server, it had internally converted it into JavaScript-based objects that the Web application could understand.

Application's user interface must provide the following set of functionalities:

- Render of ReadyMap WebGL globe within a 3D context view, embedded into a *Canvas* HTML5 element.
- User's interface must display several instances of buildings on the globe's surface in sync.
- Data retrieved from the server-side must be loaded in an asynchronous way in the client-side, aiming to provide real-time dynamism to the Web application.
- The data necessarily must be loaded and processed in a reasonable time considering the volume of data that Web application has to manage.
- ReadyMap WebGL globe's interface must provide pan, zoom, rotation and tilt as minimum movements' requirements for a correct scene exploration. Additionally, it has been expected some kind of helper —HTML element i.e. —, which would have let the end-user to move the camera-based perspective towards any present area or point of interest.
- Concerning source code, it has been highly important to reuse as much source code as possible, avoiding reimplementation and redesigning, therefore making the Web application as light and clear as possible.

3.2.3 Data requirements

With respect to cadastre data volume, likely to be recorded in Shapefile (SHP) format, it has been necessary to adopt a lightweight data exchange format conversion where to store the attributes and values coming from both topologic and semantic buildings' features.

Considering that this research has taken place in a Web environment, JavaScript Object Notation (JSON) has been the most suitable data exchange format for previously mentioned purpose, being much easier to load, read, parse and manipulate than Extensive Markup Language (XML) for instance. In view of JSON format is faster (Wang G. , 2011) and more convenient than XML (Azad, 2007), it has been adopted as exchange data format.

3.2.4 Software and hardware requirements

WebGL has not been designed to work on equipments based on antiquated graphic cards, so that ultimately both user's hardware and software configuration will determine whether the Web application will properly run or not.

Lately, graphic cards builders have realized about the importance of WebGL, and the new forthcoming range of market possibilities, so that graphic card drivers have begun to be updated in order to handle, if possible, the new graphic standard. For that reason, at last instance is user's responsibility to keep graphic card's drivers updated in order to properly visualize WebGL-based content.

Additionally, browser's choice will directly determine the way the Web application will render graphics on-screen. In order to provide the best user experience, some particular browsers —Google Chrome or Mozilla Firefox i.e.— are able to selectively enable or disable customized WebGL implementation support within its internal configuration. This has a clear aim to make new WebGL-based prototypical applications as compatible and trustworthy to developer's designs as possible. Furthermore, Khronos Group provides a real WebGL State of The Art with regard to browser graphic support (Khronos Group, 2011).

Next, it is detailed in the following *Test unit's hardware* and *Test unit's software sections*, the tested unit's configuration used along the development of this research.

3.2.4.1 Tested unit's hardware

Regarding the hardware's configuration present in tested unit, it is described as following.

Dell Vostro 1510
Intel Core 2 Duo T8100 2.10GHz
3 GB RAM

NVIDIA GeForce 8400M GS
Driver version 6.14.12.6658

3.2.4.2 Tested unit's software

Regarding the software's configuration present in tested unit, it is described as following.

Windows XP Professional SP3
Notepad++ v9.5.4
Apache HTTP Server 2.2
PHP 5.3.8.0
Quantum GIS 1.7.2
gvSIG 1.11

Google Chrome v16.0.912.63 m	(WebGL: Hardware accelerated)
Mozilla Firefox v9.0.1	(WebGL: Hardware accelerated)
Opera v11.60	(not compatible with WebGL)
Safari v5.1.2 (7534.52.7)	(not compatible with WebGL)
Internet Explorer v7.0.5730.13	(not compatible with WebGL)

3.3 Use Case Model

The Use Case model is a collection of diagrams and text that together document how users expect to interact with the system. Based on (A. Pender, 2002) , next figure depicts system's behaviour, highlighting who have been the actors within the system and which have been their goals, also showing the dependencies between elements.

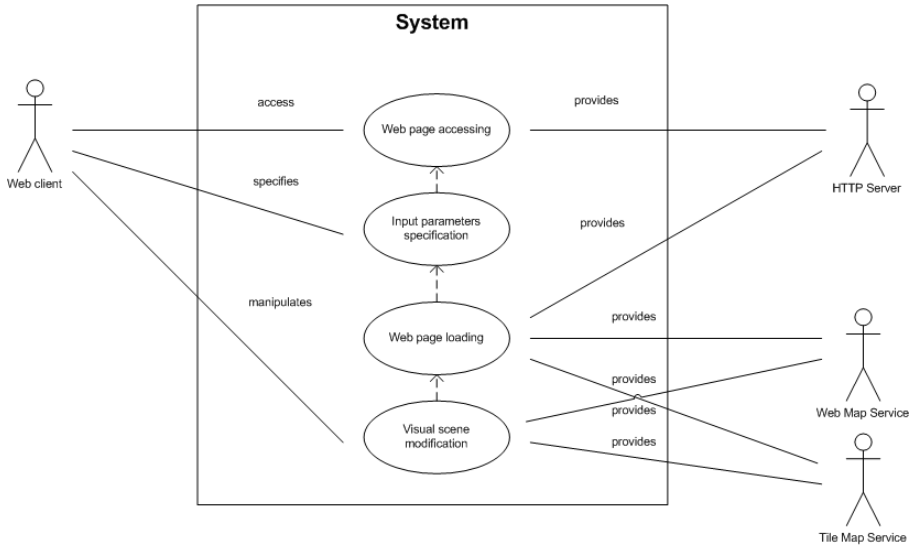


Figure 7. Web application's use case diagram

As seen, the purposed solution has concerned a simple use case model where only four use cases and four actors have been involved.

Regarding actors, they have been depicted as next:

- *Web Client***:** Web Client has been defined as the end-user of the Web application designed along this research. All processes have started due to the triggering action of this actor. It has been the responsible of accessing to the Web application, specifying input parameters and modifying visual parameters by exploring the scene.
- *HTTP Server*: This actor has been characterized along this project as an Apache HTTP local server mainly used for testing purposes. It is expected, in a close future, to further develop the Web application relying on remote server architecture. Additionally, HTTP server actor has been the responsible of providing, straight after a Web client request, the corresponding HTML file in addition to the associated files —such as CSS and JavaScript ones— which has provided full functionality and embellishment to the Web application. Furthermore, HTTP server is in charge of sending to Web client the JSON file with the building's geometries.
- *Web Map Service (WMS)*: Web Map Service actor has been the responsible of serving to the Web client, georeferenced map images over the Internet. Such images have been previously generated by a map server, and ultimately gathered from the corresponding geospatial database. Along this present research this actor will be shaped as PNOA WMS.
- *Tile Map Service (TMS)*: Tile Map Service actor has been the responsible of serving to the Web client, georeferenced map tiles over the Internet. Those tiles have been previously generated by a map server which ultimately gathered data from the corresponding geospatial database. Along this present research this actor will be defined as ReadyMap TMS, serving both Landsat 7 ETM and OpenStreetMaps imagery.

Regarding use cases, there have been depicted the following:

- *Web page accessing*: The end-user would request to HTTP server, a single HTML file and by default all its dependencies; and the HTTP server would bring the requested files back to the user. Once all the data have been loaded and pre-processed into the end-user computer, it would be thereafter possible to

specify the set of input parameters that would determine the Web application's behaviour.

- *Input parameters specification*: This particular use case has been exclusive for the end-user. Depending on the input, the application would behave in some different ways. The end-user would specify the desired WebGL rendering mode, a thematic colour visualization, and the desired number of children to add to the scene graph for testing purposes.
- *Web page loading*: At this point, the servers would provide to the Web application the requested data accordingly to the visual map extents. HTTP server would provide a pre-processed JSON file where the buildings topology and semantics would be defined. WMS server would provide the corresponding PNOA imagery, rather the same as TMS server which would provide the corresponding tiles of both Landsat and OSM sources.
- *Visual scene modification*: Finally, the end-user, once all the input data would have been loaded, processed and represented on-screen, would be able to explore and visualize the scene.

3.4 High Level Architecture

In this section it is described the high-level application's architecture diagram, so as to have a better understanding of the necessary components used on the development of this research. Next, it is presented the general schema:

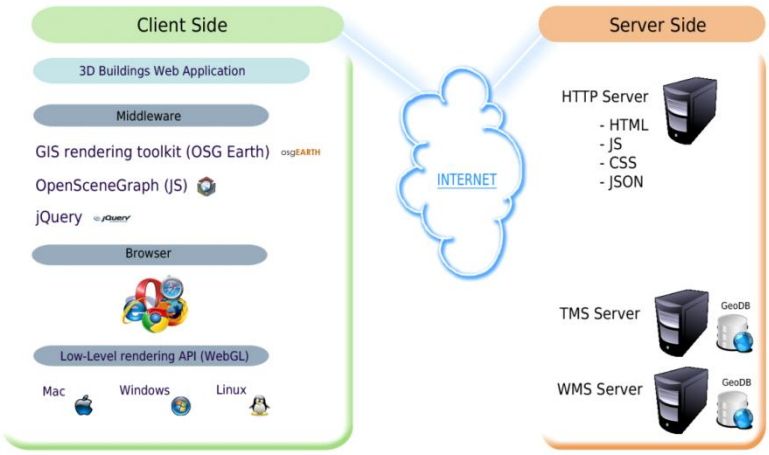


Figure 8. Web application's high level architecture diagram

As seen, the communication schema has been divided in two principal parts: client-side and server-side. It should be noted that, even if it has been depicted that HTTP server is located on the server side (see *Figure 8*), client's side software and HTTP Server have been actually located and executed on the same local machine for testing purposes, while rest of server types —TMS and WMS— have been located in a remote location. Additionally it has been planned a migration of HTTP server, from local to remote a location when this research's work would have completely finished, being therefore accessible to general public.

Client-side has been responsible of the user's interface presentation which has been expected to be a simple and clear interface, where it was going to be required that the end-user would fulfil a set of application parameters to manage Web application behaviour.

The main tool that the end-user would need to have installed in advance on his machine would be a Web browser. Unfortunately, not all range of browsers on market have been WebGL standard compliant to date; nevertheless, it is expected that browser's architecture will provide sooner or later a complete WebGL standardization, no matter which operating system the end-user would use.

Once the end-user had accessed the Web application, a series of compiled JavaScript files would be downloaded into de Web browser, which is actually the middleware's implementation. Middleware is computer software that connects software components trying to increase interoperability between systems with different architectures, simplifying therefore complex distributed applications (Verissimo & Rodrigues, 2001). In the context of this research, using middleware has been a crucial step to integrate and manage information coming from both graphical (OSG) and geographical (OSGEarth) sources.

In addition to the use of OSG and OSGEarth, it has been necessary to include jQuery JavaScript library in order to simplify HTML document traversing, event handling and AJAX requests. On top of this structure the buildings' Web application —that is proposed in this research— has taken place.

Server-side is something out of the boundaries of this research, yet it has been be explained along this document due to the existing interrelation between the server-side and the client-side.

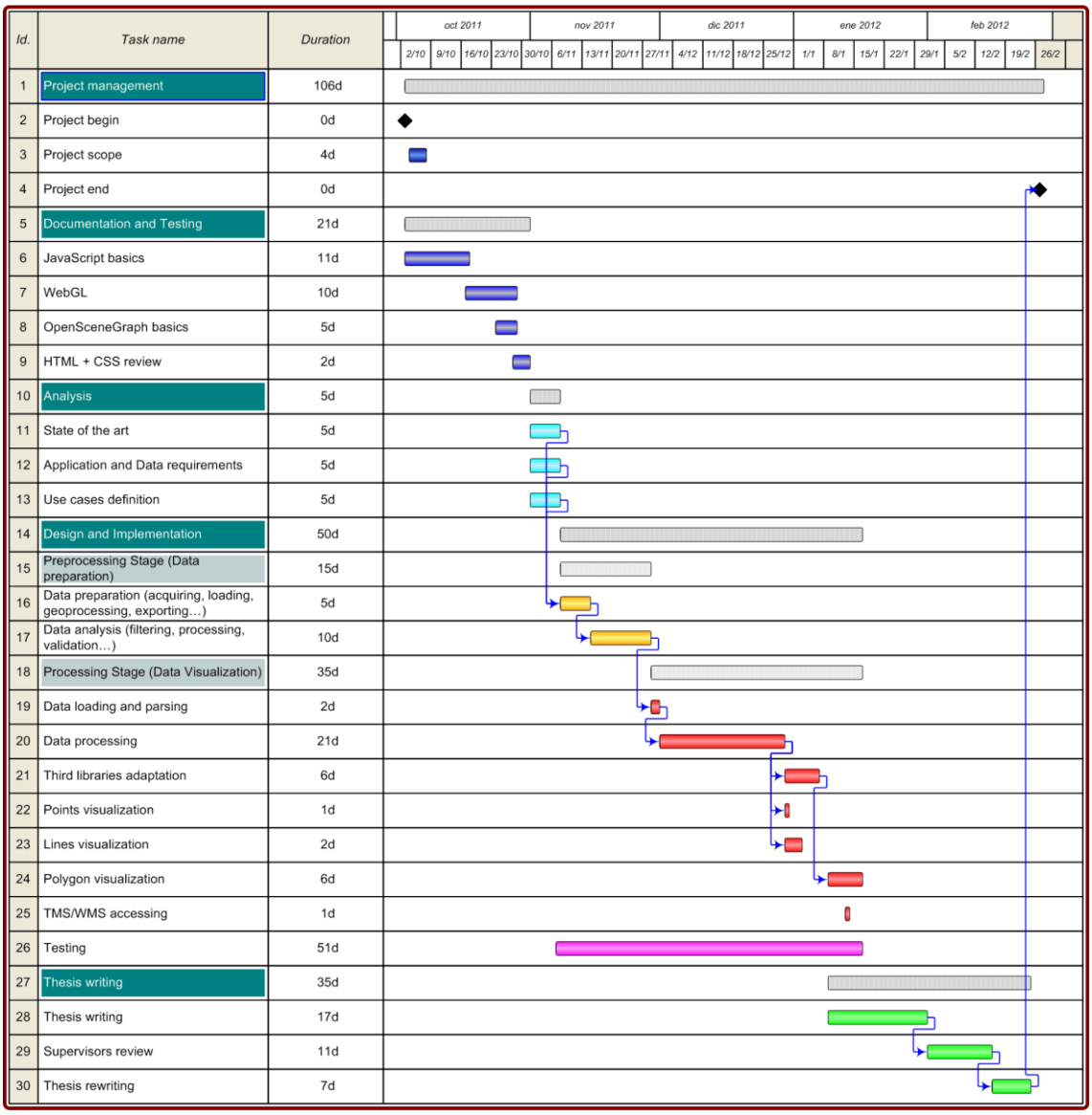
As mentioned, HTML server has run on a local machine during implementation and testing time, which is expected to be ported to a remote location once the Web application had finished. When referring to HTML server, it is referred in fact to a combination of Apache HTTP server and PHP, which is in fact the responsible of resolving the requests for corresponding set of tiles.

Thus, HTTP server has provided HTML, JSON, CSS and JavaScript files to the Web application requested via the corresponding AJAX requests from the client-side. Due to browser security restrictions, most AJAX requests are subject to the same origin policy, so that the request cannot successfully retrieve data from a different domain, sub domain, or protocol. For this reason, it has been used a local HTTP server which has solved such issues, besides to provide access to the Web application to external computers connected to the same network. Alternatively, it is possible to solve the mentioned origin policy issue by halting full browser's security processing —accessing to the deep internal configuration—, yet this is a discouraged solution due to both security and back-door reasons.

WMS and TMS servers have been neither designed nor implemented at all, nevertheless both would be accessed when the Web application would request the corresponding set of tiles and map imagery so as to place them on top of the globe's surface.

3.5 Planning (Gantt diagram)

With regard to project time-management matter, this research has been mainly divided into five groups, concerning from project planning to thesis writing stage. Following, it is presented a Gantt diagram depicting the working packages and their relative and absolute assigned time.



Id.	Task name	Duration
1	Project management	106d
2	Project begin	0d
3	Project scope	4d
4	Project end	0d
5	Documentation and Testing	21d
6	JavaScript basics	11d
7	WebGL	10d
8	OpenSceneGraph basics	5d
9	HTML + CSS review	2d
10	Analysis	5d
11	State of the art	5d
12	Application and Data requirements	5d
13	Use cases definition	5d
14	Design and Implementation	50d
15	Preprocessing Stage (Data preparation)	15d
16	Data preparation (acquiring, loading, geoprocessing, exporting…)	5d
17	Data analysis (filtering, processing, validation…)	10d
18	Processing Stage (Data Visualization)	35d
19	Data loading and parsing	2d
20	Data processing	21d
21	Third libraries adaptation	6d
22	Points visualization	1d
23	Lines visualization	2d
24	Polygon visualization	6d
25	TMS/WMS accessing	1d
26	Testing	51d
27	Thesis writing	35d
28	Thesis writing	17d
29	Supervisors review	11d
30	Thesis rewriting	7d

Table 2. Gantt diagram

- *Project management*: it has concerned all generalized assignments derived from the project's planning.
- *Documentation and Testing*: it has concerned documentation's labours such as reference books and thematic articles readings so as to define solid groundings for the rest of project's stages. Within this *Documentation and Testing* stage, it has been settled the basics of later used programming languages, being expected general improvements on programming skills at this point. A special effort has been given at learning JavaScript programming language considering that most of this research's functionalities are based on such language. Additionally both WebGL specification and pipeline has been studied in-depth considering they

represent a crucial step on the clear understand of WebGL-based OSG behaviour. Similarly, it has been necessary to assimilate OSG groundings in view of they play a connection role between low-level graphic card's instructions and the provided buildings' implementation.

- *Analysis*: it has gathered related application's analysis interpretations, which have provided a better description of the project's scope. Within this package, there have been detailed application's requirements, structure, architecture, in addition to the economic proposal in order to better define what this research has entailed.
- *Design and implementation*: this work package has been the research's core group since its structure concerns the most important processes in terms of time-consuming and expressed difficulty. In this *Design and implementation* stage it has taken place the vast majority of software implementations which have been divided in turn into two main stages. The first one has concerned the required set of planned implementations to analyze, filter and prepare the original cadastre data for the second group. The second group has implemented the buildings' integration functionality itself by using the data prepared in previous stage. Both stages have been consequently followed by a constant testing to check the success of the performed improvements.
- *Writing*: it has basically concerned thesis writing and reviewing tasks.

3.6 Budget

Both assimilating and estimating the costs and benefits of a GIS project is a crucial role of the strategic project planning and decision making process. Determining the economics of a GIS project is not an easy task and it should be clearly depicted.

The present research has taken, to one single programmer, roughly two months of full-time work in order to implement the solution, and one prior extra month of documentation and testing.

Thus, based on (Bonfatti, 1998) GIS cost approach it has been included the following budget description.

AMOUNT	DESCRIPTION	Unit cost	TOTAL
Hardware			
0	Not included		
Software			
1	Notepad ++	0€	0€
>1	Browser (any)	0€	0€
1	Apache HTTP Server + PHP	0€	0€
1	gvSIG	0€	0€
1	Quantum GIS	0€	0€
Data			
1	Cadastre data	0€	0€
Services			
352 hours	Software Developer work (not including documentation stage)	50.84€/h	17895,68 €
512 hours	Software Developer work (including documentation stage)	50.84€/h	26030,08€
Maintenance			
0	Not included		
		SUBTOTAL	26030,08 €
		VAT (18%)	3123,3 €
		TOTAL	**30715,49€**

Table 3. Budget table

3.7 Chapter Review

In this section the application's analysis has been described. Firstly, a group of system's requirements have been explained in-depth. Secondly, the application's use case model and the application's high level architecture diagram have been depicted; by regarding both diagrams it is possible to recognize the general activities the system has implemented as well as the overall activities' workflow. Finally there has been described both project's timing and estimated budget.

Chapter 4

Design and Implementation

4.1 Introduction

In this chapter it is described in detail the necessary steps to build up a building's representation on top of ReadyMap WebGL globe's surface, following the presented requirements (see *3.2 Requirements* section). The chapter is further divided into two main sections: *Pre-processing* and *Processing*.

Pre-processing section has detailed a series of required steps to prepare and adapt the original cadastre dataset, up to be finally transformed into a JSON structure (Butler, Daly, Doyle, Gillies, Schaub, & Schmidt, 2008). This stage has been contextualized firstly, in a GIS-based software environment so as to geo-process it; and secondly, on a JavaScript-based application which has processed and filtered the data. The consequent output is has been finally loaded by the Web application.

Processing section has described the transformations applied to the processed data coming from *Pre-processing* section. This processing has entirely taken place within the ReadyMap's architecture, and it has been further divided into two main groups. Firstly, it has described how data have been loaded and prepared, in addition to an extrusion processing which has shaped building's geometry. Secondly, it has described how data has been visualized and the background processing it entailed.

4.2 Pre-processing (Data preparation)

4.2.1 Introduction

In this section it is described the entire processing required to provide the Web application with a filtered and adapted dataset coming originally from the cadastre. This

section is additionally divided in three parts: *Data and Software*, *Data Preparation (gvSIG & QGIS)* and *Data Preparation (JavaScript)*.

In *Data and Software* it is firstly presented the original cadastre data's details as well as the software used in converting and transforming such data into another format that the Web application could manage.

Secondly, in *Data preparation* section —with regard to gvSIG and QGIS software—, it is described in-depth reprojection and exporting processes applied to the original data. The corresponding output of such geo-processes is depicted as a perfectly georeferenced data ready to be geometrically processed.

Finally, in *Data preparation* section —with regard to JavaScript environment— it is firstly described how the data were loaded by explaining the corresponding server communication. Secondly, it is exhaustively explained the geometrical filtering applied to the reprojected data and the consequent resulting output.

4.2.2 Data and Software

For the implementation of the building visualization's functionality, external data have been required. It has been described as a set of two vectorial layers extracted from Valencian cadastre dataset. These layers have been supplied by the Directorate General for Spanish Cadastre in the form of Electronic site (Treasury, 2012) at no cost.

Thus, the two vectorial layers used in this research are *Constru.shp* and *Masa.shp*. The general details for *Constru.shp* vectorial layer are the followings — it should be noted in boldface the attributes which have been used— (see also *Figure 9* for complete information):

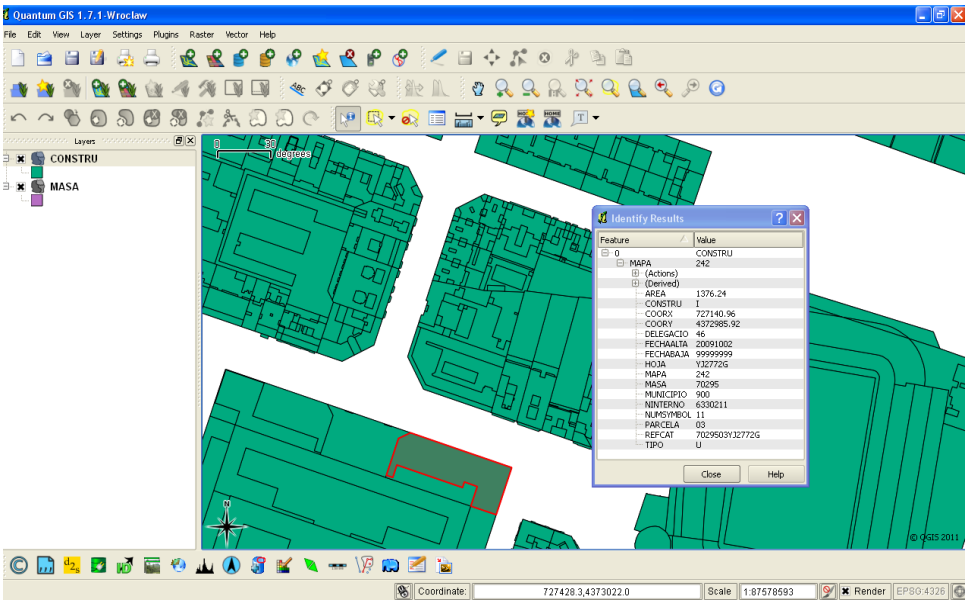


Figure 9. Constru.shp details

Type:	ESRI Shapefile	
Projection:	Universal Transverse Mercator (UTM)	
Zone:	30 Northern Hemisphere	
Datum:	European Datum 1950 (ED50)	
Spheroid:	International 1909 (Hayford Intl 1924)	
Units:	meters	
EPSG Code:	23030	
Area items:	235.968	
Vertex items:	2.596.327	
Covered Area:	422,64 km^2	
Attributes:	*Mapa*	Map number to which an item belongs
	Delegacio	Hacienda delegation id number
	Municipio	Municipality
	Masa	Block id
	Hoja	Cadastre reference
	Tipo	Type of parcels
	CoordX	Block centroid X coordinate
	CoordY	Block centroid Y coordinate
	NumSymbol	Internal number
	Area	Area in m^2
	FechaAlta	Date of construction/registration
	FechaBaja	Date of demolition/unregistration
	Ninterno	Sequential Number assigned by the system
	Parcela	Parcel code
	Constru	Constructed heights string
	RefCat	Parcel Cadastre reference

Concurrently, the general details for *Masa.shp* vectorial layer are the following — it should be noted the difference in building's geometry definition—(See also *Figure 10* for complete information):

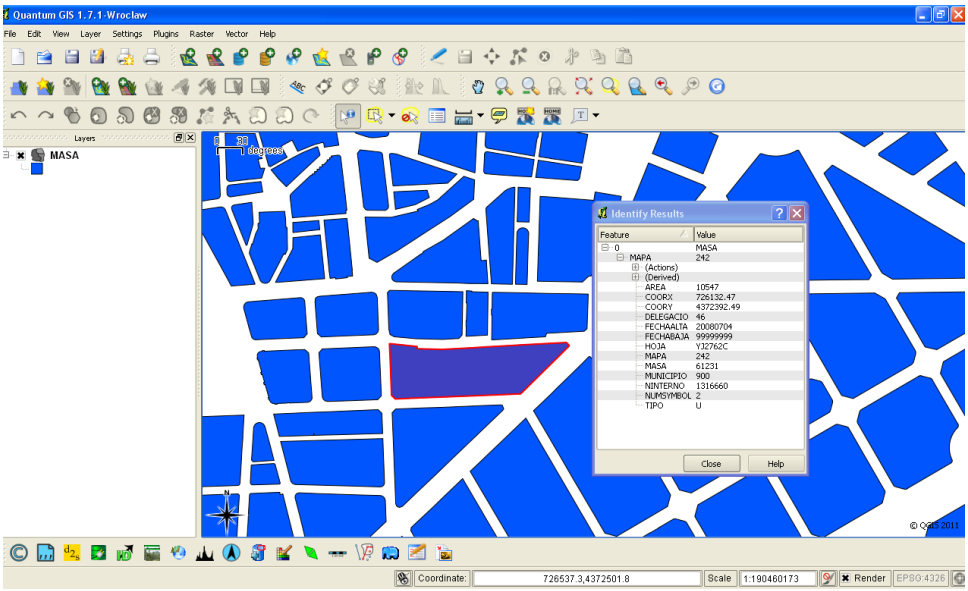


Figure 10. Masa.shp details

Type:	ESRI Shapefile	
Projection:	Universal Transverse Mercator (UTM)	
Zone:	30 Northern Hemisphere	
Datum:	European Datum 1950 (ED50)	
Spheroid:	International 1909 (Hayford Intl 1924)	
Units:	meters	
EPSG Code:	23030	
Area items:	4906	
Vertex items:	255.599	
Covered Area:	422,64 km2	
Attributes:	*Mapa*	Map number to which an item belongs
	Delegacio	Hacienda delegation id number
	Municipio	Municipality
	Masa	Block id
	Hoja	Cadastre reference
	Tipo	Type of parcels
	CoordX	Block centroid X coordinate
	CoordY	Block centroid Y coordinate
	NumSymbol	Internal number
	Area	Area in m2
	FechaAlta	Date of construction/registration
	FechaBaja	Date of demolition/unregistration

Ninterno Sequential Number assigned by the system

With respect to software and data workflow, following it is briefly summarized the general data pre-processing. Nevertheless *4.2.3 Data preparation (gvSIG & QGIS)* and *4.2.4 Data Preparation (JavaScript)* have explained it in-depth.

Given that the Web application requires geographical-based data coordinates to be processed and the original data are described in UTM coordinates, it has been necessary a map reprojection. In order to accomplish such task, gvSIG open source software has been used due to the robustness of the implemented geo-processing library (Cropper, 2010).

Additionally, Quantum GIS (QGIS) (OSGeo project, 2012) has been used for data visualization and exportation. It has been very helpful since it natively exports geographical data to JSON file format. Alternatively it is possible to execute the same assignment with GDAL library, particularly with OGR extension which is designed for vectorial data handling.

Thus, when data have been reprojected, exported and later on imported in QGIS, it has been ran a series of generalization processes in order to mitigate the amount of unuseful data to be exported, lightening therefore the resulting output files for the following JavaScript processing.

Once data have been processed in QGIS and further exported, it has been consequently loaded, parsed, and processed in a browser, for testing purposes, in order to obtain the final data output that Web Application can manage. This work has been done by creating a script, natively written in JavaScript, which is the responsible of yet again lightening and depurating prior data so as to obtain a final validated file.

4.2.3 Data Preparation (gvSIG & QGIS)

The main idea behind the use of both vectorial layers —*masa.shp* and *constru.shp*— has been to extract the interesting features of each of them to be merged in an ultimate output data. Each of the layers have provided some interesting and indispensable attributes separately, however due to the nature of this data, there have been a great amount of useless attributes, hence data filtering has been a must at this point.

On the one hand, it is needed to export building's geometry; on the other hand it is crucial to isolate and extract building's height information in the lightest possible way. Due to the amount of information to manage, it has been extremely necessary to minimize and optimize each data processing in order to improve both time and memory consumption.

Regarding building's geometry, both *constru.shp* and *masa.shp* provide a perfect geometrical outline of the several existing blocks of buildings —4906 blocks in fact—. Nevertheless *constru.shp* not only describes the blocks outline but also the internal area subdivisions —such as roofs, yards or balconies— (see *Figure 11*). It should be noted the main purpose has been to represent the contour of the building's ground floor and not the internal area subdivisions.

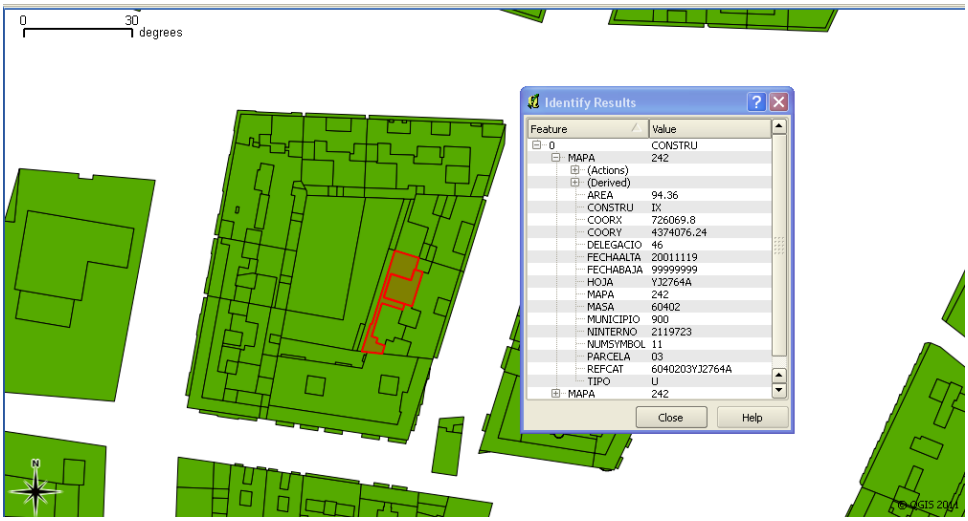


Figure 11. Constru.shp composition (closer detail)

Considering building's heights, they are only depicted in *constru.shp* —in *constru* attribute— (see *constru.shp* attributes). Despite building's heights have been essential for the final Web application, managing the complex geometry that *constru.shp* provides would have been firstly time-consuming processing; and secondly, it would have considerably decreased the probabilities of success in representing the Valencian cadastre, mainly due to the exaggerated amount of points required for representing the same building's outline.

Next table depicts the statistics related with the corresponding geometry of both layers which helps to understand the differences of complexity between them.

	Masa.shp	Constru.shp	Difference (%)
Vertex Items	255.599	2.596.327	+ 1015,78
Area Items	4096	235.968	+ 5760,93
Covered Area	422,64	422,64	+ 0,00

Table 4. Shapefiles geometry comparison

As a consequence of this preliminary data approach, it has been decided to extract height values from *Constru.shp*, and building's geometry from *Masa.shp*.

The main purpose behind this approach is to take advantage of *masa* attribute —do not confuse it with *masa.shp* layer— present in *constru.shp* (See *constru.shp* attributes).

Masa attribute describes to which particular building belongs each small area subdivision, therefore it is possible, once gathered all the small areas that belong to a particular building, to geo-process each of them in order to obtain the building's ground floor outline. In this way it would be only necessary to use *constru.shp* file.

Nonetheless, considering such geo-process would have worked fine, it is not interesting in terms of time-consuming and complexity, because *masa.shp* already provides such building's ground floor outline (see *Figure 10* for complete data).

As a result, from this point onwards, *masa.shp* has been used to describe the building's geometry, and concurrently *constru.shp* has been used to describe the building's height. The *masa* attribute, present in both layers, has played therefore a linking role between the buildings' geometries and the buildings' heights.

4.2.3.1 Reprojection and exportation

By examining the OSGEarth map functionalities present in ReadyMap, it has been decided to manage building's data in the form of geographical coordinates. Nonetheless, the data the Spanish cadastre had provided, had come in UTM coordinates, so that it has been required a data reprojection in order to adapt it into a new easy-to-use format for the Web application.

An apparently idle reprojection is the one which original data described in ED50 datum —EPSG:23030— (See *masa.shp* attributes) are reprojected to the widespread WGS84

datum —EPSG:4326—which is used by GPS devices and it describes geographical coordinates. Nonetheless for the case of Spain it is not the most suitable.

Considering ED50 datum was designed for fitting whole Europe, in the case of a reprojection to WGS84, it would work fine for central Europe, but not for the case of Spain. For the case of Spain, there would be point's displacements bigger than 200m, therefore it has been compulsory (BOE, 2007) to change the transformation to ETRS89.

Thus, data are reprojected from UTM coordinates to Geographical Coordinates using the following details:

Projection:	Geographic
Latitude of the origin:	0°
Longitude of the origin (central meridian):	0°
Scaling factor:	1
False easting:	0
False northing:	0
Datum:	ETRS89
Units:	degrees
EPSG Code:	4258

For this assignment the National Geographical Institute (IGN) provides a particular grid in NTV2 format (National Geographic Institute, 2012), to transform from ED50 datum into ETRS89 datum (see *Figure 12*), minimizing therefore the errors to minimum values. gvSIG has been used for reprojecting the data relying on the grid, resulting therefore in a new projected Shapefile layer defined as EPSG:4258.

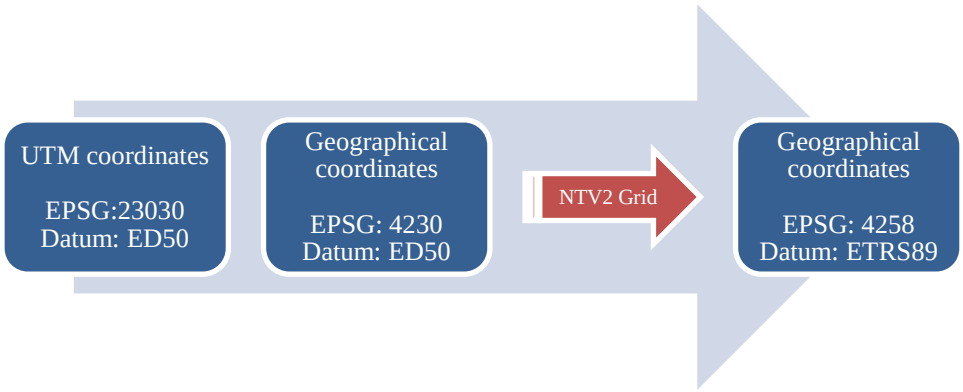


Figure 12. Reprojection's work flow

At this point data would be ready to be loaded in QGIS and later on exported to JSON. Next, it is depicted a piece of code representing the structure of a single building in

JSON format, extracted from the exportation of *masa.shp* into JSON —from now on called "masa.json"— (see *Figure 13*).


```
{ "type": "Feature", "id": 0, "properties": { "MAPA": 242, "DELEGACIO": 46,
"MUNICIPIO": 900, "MASA": "64514", "HOJA": "YJ2765A", "TIPO": "U",
"COORX": 726457.220000, "COORY": 4375194.490000, "NUMSYMBOL": 2,
"AREA": 7, "FECHAALTA": 20050331, "FECHABAJA": 99999999,
"NINTERNO": 1211371 }, "geometry": { "type": "Polygon", "coordinates": [ [ [
-0.366552, 39.495920 ], [ -0.366594, 39.495884 ], [ -0.366606, 39.495891 ], [ -
0.366564, 39.495927 ], [ -0.366552, 39.495920 ] ] ] } }
```


Figure 13. Masa.json code sample

Contrarily, for the case of *constru.shp* —where building's height have been described— it has been necessary a different approach mainly due to both the huge output file size —nearly 200MB—, and the unnecessary information it carries. Consequently, it has been required an attribute's table processing in order to lighten data by deleting useless and redundant data.

Considering that QGIS does not supply any table attribute's operations, it has been necessary to install an additional QGIS plugin in order to adapt, modify and export the associated *constru.shp* attribute's table, not to JSON but to CSV —from now on called *constru.csv*—.

In order to illustrate the data structure contained in *constru.csv*, a sample of the data contained in the file is shown below (see *Figure 14*).


```
MASA, CONSTRU,    AREA
36048, -II+I,     10.92
36048, VII,       4.59
36048, VII,       2.20
```


Figure 14. Constru.csv sample code

Masa attribute has been a connection between *masa.json* and *constru.csv* since it has been present and replicated in both layers with regard to the same geometry. It describes as explained (see *4.2.3 Data Preparation gvSIG & QGIS*), to which particular building belongs each small area subdivision.

Constru attribute depicts in a coded way defined by the Spanish Cadastre (Treasury, 2011) the "constructed heights" of each particular building's area subdivision.

Finally, *area* attribute has been neither used nor implemented in this particular research; nevertheless it has been also exported and loaded (see *4.2.4 Data Preparation JavaScript*) considering it for future releases to obtain more accurate buildings' heights.

4.2.4 Data Preparation (JavaScript)

In addition to gvSIG and QGIS, a JavaScript-based application has been implemented to load and process *masa.json* and *constru.csv* files. It has been proposed in order to obtain the corresponding extracted attributes from each of the files, merged together in a final JSON String. By using jQuery library —more specifically AJAX requests— it has been possible to easily load and parse both files so as to manage their attributes as JavaScript objects.

Furthermore, the way the data have been loaded is exactly the same as implemented on ReadyMap Web application so that; firstly, the JavaScript-based application has been set in the same framework as ReadyMap; and secondly, this piece of code has been reused minimizing therefore code redundancy.

In next *Server Communication* section it is described firstly, how the data have been loaded, and secondly the exhaustive processing of *constru.csv* —to get building's heights— and *masa.json* —to obtain a filtered building's outline— to be finally saved into a JSON exchange data file.

4.2.4.1 Server communication

At this point it is relevant to mention that, even if the current application has been implemented in JavaScript on the client-side mainly for testing purposes, it is expected a future release designed for the server-side, and further described as a Web Service.

With regard to time-consuming statistics when running the application, it has been concluded that Mozilla Firefox 8.0.1 has been the most-reliable browser for this particular data processing.

Browser	Time
Mozilla Firefox	Nearly 40min
Google Chrome	∞
Opera	∞

Table 5. Data preparation's script execution time

As illustrated (See *Table 5*), both Chrome and Opera were not able to finish the data processing for some reason. Apparently, Firebug developer plugin for Mozilla Firefox has worked much better managing and processing big amount of data, than similar solutions on compared competitors. Nonetheless, this research has no intention to establish any groundings or conclusions about browsers performance based on the obtained results.

Following, it is exemplified (see *Figure 15*) how data files have been loaded, by highlighting both AJAX requests, in order to make them ready to be processed. It should be noted that *getJSON* is an extension of the AJAX functionality within jQuery (The jQuery project, 2010) even if the name can be confusing.

```
<html>
<head>
<script type="text/javascript" src="jquery/jquery-1.4.2.js"></script>

<script type="text/javascript">

$(document).ready(function(){

        $.ajax({
                url: "../tests/data/CSV/constru.csv",
                datatype: "txt",
                success : function (data) { ... }

                $.getJSON("../tests/data/masa.json", function(json) { ...
}
                );
        });
 })

</script>
</head>
<body>
</body>
</html>
```

Figure 15. Data preparation's AJAX requests

4.2.4.2 Constru.csv processing

As illustrated (see *Figure 14*), *constru.csv* has three attributes: *masa*, *constru* and *area*. *masa* and *constru* have been loaded and processed, whereas *area* has been just loaded but not processed as said (see *4.2.3.1 Reprojection and exportation*).

Height attribute's value —*constru*— has not come defined as any numerical format but as a String format, describing not the real height but the constructed height. In this way it has been assumed that a constructed height is equivalent to three meters for future conversions.

After *constru.csv* processing it is expected a resulting output consisting of a new array of tuples of *masa* —unique for each building— and a calculated building's height for each *masa* value.

In order to accomplish the mentioned assignment, firstly, it has been essential to find out how many and how different height string characters were present in *constru* attribute. Following, it is demonstrated (see *Figure 16*) a sequence of extracted height string characters —out of 640 different ones— that were present in "constru.csv" table.

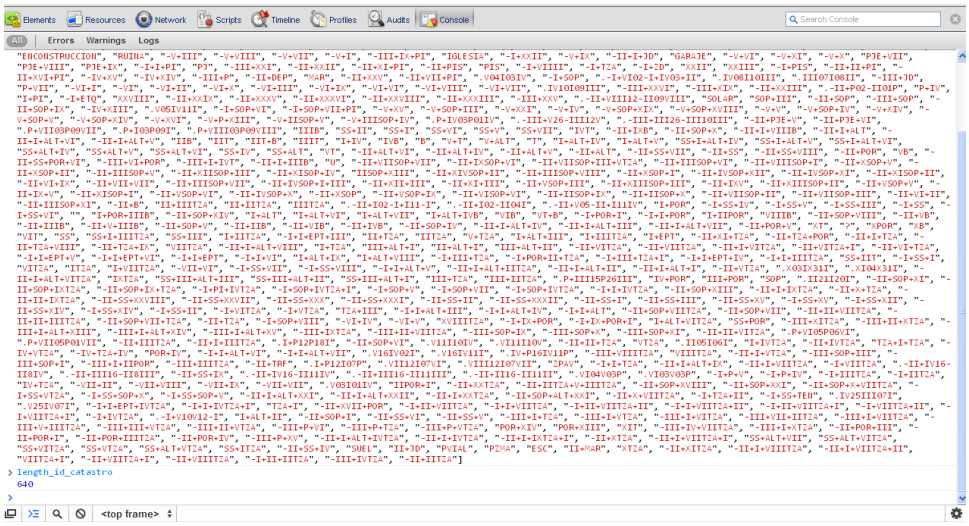


Figure 16. Data preparation heights attribute values

Secondly, it is has been necessary to divide those height's attributes into two main groups: underground values and on surface values. Undergrounds values, have been depicted as a dash followed by a value in Roman number format, whereas on surface

values have been described as non-dashed Roman numbers. Detailed next, it is exemplify such cases in order for the reader to clearly understand it (see *Figure 17*).

*-II+III → -(2*3m) + (3*3m) =6m (underground) + 9m (on surface)*

*-I+V → -(1*3m) + (5*3m) = 3m (underground) + 15m (on surface)*

*-I+X → -(1*3m) + (10*3m) = 3m (underground) + 30m (on surface)*

Figure 17. Height attribute codification I

Consequently, all underground —negative— heights have been ignored. This assignment has been accomplished by applying to each height string character a series of Regular Expression's filtering (Mozilla Developer Network, 2012) in order to delete the corresponding part of the string that does not match the application's requirements.

Thirdly, it has been necessary to delete concurrently, —using Regular Expression's filtering— redundant height information, such as balconies or terraces, described as a set of code values defined by the Spanish Treasury (Treasury, 2011). In this way undesired values are then wrapped into a set of well-known and desired height values. Next, it is detailed a small set of code values (See *Table 6*) and subsequently, a practical example to strengthen the concept (see *Figure 18*).

Code	Description
B	Balcony
T	Terrace
POR	Porch
ESC	Stairs
JD	Garden

Table 6. Heights attribute codification II

```
var alt_planta = 3;
for(var i = 0; i<altura.length; i++){

        if(altura[i] == 'I') { altura[i] = 1 * alt_planta; continue;}
        if(altura[i] == 'II') { altura[i] = 2 * alt_planta; continue;}
        if(altura[i] == 'III') { altura[i] = 3 * alt_planta; continue;}
```

Figure 18. Heights attribute codification III

Additionally, it has been further required to convert String-formatted height values into numerical-formatted values —such as Integer of Double. At this point, it has been used the assumed three-meter's height per constructed floor as next detail (see *Figure 19*).

```
-II+III+ESC      becomes      III
-I +III+JD       becomes      III
-I +III+POR+T+B  becomes      III
```

Figure 19. Heights definition algorithm code extraction

Regarding *Figure 14*, it can be noticed how *constru.csv* presents more than one height —*constru*— per *masa* attribute value —therefore per building—. Given that the desired output is to have a single array with unique *masa* values, it has been necessary to define how is achieved a single height value per building. Thus the definition of final height value is defined as the maximum of all the heights associated to each *masa* value.

As a result, it has been obtained a single array formed by unequivocal values of *masa* and building's height. In addition to this result, it is expected —in future releases— an improvement on height's calculations by considering *area* attribute into processing as mentioned.

4.2.4.3 Masa.json processing

Contrarily to *constru.csv* —focused on height's processing— related *masa.json* processing has focused on geographical and positional transformations. After processing, it is expected a remarkable simplification on the ground floors' outlines by deleting redundant points, without notably affecting to building's geometry.

At this point, it is relevant to mention that QGIS exports both latitude and longitude coordinates with six decimal numbers, contrarily to original data —*masa.shp* and *constru.shp*— which are described with up to fifteen decimal numbers. This exportation "handicap" produces in this particular research, due to Valencia location, a reasonable error displacement of 0.9m at most in both latitude and longitude coordinates. Nevertheless, considering the last version of ogr2ogr utility (GDAL, 2012) within OGR library —not stable by the time of realization of this thesis though— it has been tested

that it is possible to export from SHP to JSON with even more than fifteen decimal numbers.

Such numerical facts are extremely important because there is a relative likelihood that those two different vertices —being represented as two different vertices on QGIS— might be exported as two different vertices storing exactly the same coordinates. This is an important issue it has been needed to face; because the Web application and more precisely, the forthcoming triangulation functionality cannot handle such data.

Therefore, it has been crucial to check firstly, if any of the vertices defining a ground floor outline were replicated; in such case the corresponding vertex was deleted. This iterative process has swept the existing 4906 buildings present in *masa.json* (see *Figure 23*).

Secondly, it has been verified that no building was formed by less than three vertices after the prior processing, basically due to the impossibility to be represented as a polygon (see *Figure 23*).

Thirdly, it has been checked that no point was located on the virtual circumference centred on a secondary point. Thus in the case of a —redundant— point had fallen in such neighbourhood, it would be immediately deleted, monitoring at any moment the introduced geometrical error (see *Appendix A* and *Figure 23* for complete information).

Fourthly, it has been necessary to delete collinear vertices due basically to two reasons. On the one hand, it is logical to affirm —concerning this research— that if three vertices in a row are collinear, at least one of them is redundant so it must be deleted. On the other hand it is necessary to delete such vertices because roof triangulation's algorithm requires not having collinear vertices on the dataflow that receives as input. Mathematically it is not possible to define a triangle, with positive area bigger than zero, relying on any three vertices located in a same line.

Thus, collinear vertices processing has been implemented by picking three vertices in a row and computing the area of the triangle —according to Heron's formula (Heath, 1921)— they form. Therefore, if that area had been lower than a predefined area threshold or even zero, the middle vertex would have been considered collinear, and hence deleted. Seeing that, a new triangle would have been formed by picking a first

vertex —previously first as well—, a second vertex —previously third— and a newly third vertex, therefore computing again the area for this new triangle (see *Figure 20*).

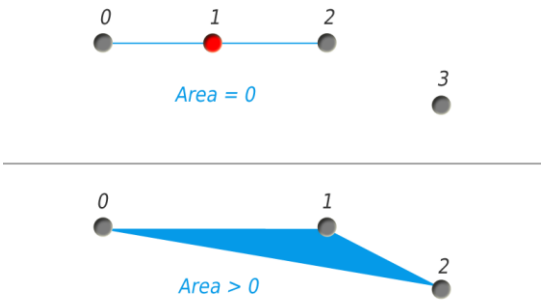


Figure 20. Area of triangle processing

In the case that any collinear vertex had been deleted, the algorithm would sweep again the totality of buildings' vertices to assure that the newly created geometry definition would have not defined any new collinear points. Additionally, it has been checked after deleting a vertex, that the building such point belonged to, had not been formed by less than three nodes (see *Appendix A* and *Figure 23* for complete information).

This step-based processing has helped to reduce redundancy and also small geometrical features that have not been exceptionally interesting to be represented on the final Web application.

4.2.4.4 Building's JSON creation

Once *constru.csv* and *masa.json* processing have finished and therefore the resulting data have been filtered, it has been subsequently wrapped into a JSON format structure, creating a verified JSON string that it is displayed in the end on the browser's console.

While wrapping processing, the buildings' heights values (see *4.2.3.2 Constru.csv processing*), and the corresponding buildings' vertices coordinates (see *4.2.3.3 Masa.json processing*) have been entirely merged.

In order to explain the obtained result, an output of the JavaScript-based application is presented below. It illustrates in up down direction:

- Internal output messages (in black).

- Sample of *masa* attribute values (in red).
- Sample of corresponding *masa* heights (in blue).
- Two lines of internal output messages (in black)
- JSON output String (in black)

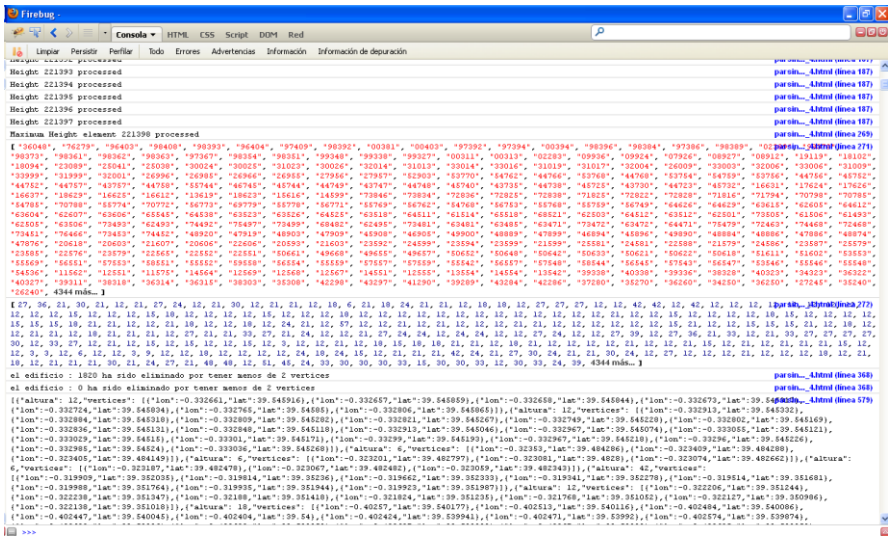


Figure 21. Data preparation's final output

As a result, JSON output has described the geometry associated to each ground floor's outline and its relative height (see *Figure 22* for complete information). Additionally, the JSON output has been copied and pasted into a new JSON file which in the end is going to be loaded by the Web application.

```
[{
   "height": 27,
   "vertices":   [{      "lon": -0.367753,
                         "lat": 39.494736},
                 {       "lon": -0.367794,
                         "lat": 39.4947},
                 {       "lon": -0.367806,
                         "lat": 39.494707}]
}, {...},{...},{...},{...}]
```

Figure 22. Prepared JSON sample

Following, it is presented a process diagram, where it is described the application's data flow and all the processes explained along this chapter so as to obtain a final JSON exchange data file.

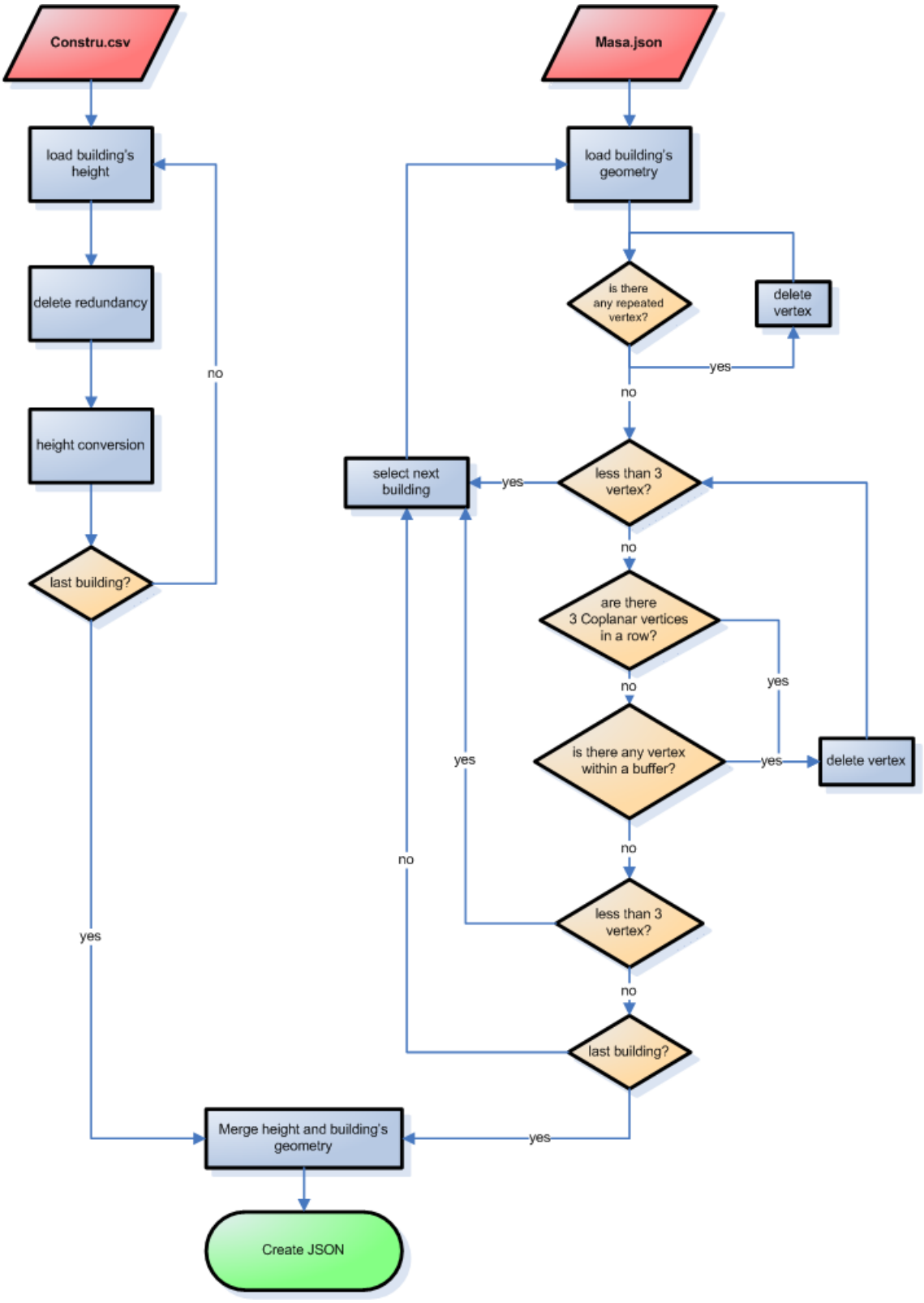


Figure 23. Data preparation process diagram

4.3 Processing (Data visualization)

4.3.1 Introduction

In this section it is described the entire processing required for the representation of buildings on top of the ReadyMap globe´s surface.

In *Application's Structure* and *Application's Behaviour* sections it is firstly explained the implementation's organization and the data workflow, taking special care in describing in-depth the implemented methods.

Secondly, in *File Organization* it is depicted the research's folders and files distribution; and similarly in *Data management*, the way the buildings' data have been treated and distributed, emphasizing the use of a LOD approach in this research.

Finally, in *Data Preparation* and *Data Visualization* it is explained the sequential processing firstly to load and transform the data coming from the *Pre-processing* stage; and secondly to graphically process such data in order to be finally represented on-screen.

4.3.2 Application Structure

Along this research, several programming languages —CSS, GLSL, JSON or HTML i.e.— have been used; nevertheless, JavaScript has been the main one, serving consequently as grounding for the rest of contributions. As a result, the application structure has been described strengthening JavaScript-based processing to clear understand the general application.

JavaScript is an object-oriented language (OOL). Mostly all that it is seen in a piece of JavaScript code has a good chance of being an object. In JavaScript, an object could be defined as a collection of named properties, a list of key-value pairs almost identical to an associative array in other languages (Stoyan, 2010).

In spite of JavaScript is an object-oriented language, there are no classes as other well known languages —Java or C# i.e.— so that there is not any kind of conventional inheritance in the way it is usually done on the mentioned OOL. Contrarily, JavaScript does inheritance via prototypes, which are objects as well.

According to these definitions, firstly, it is presented the application's distribution diagram (see *Figure 24*), which —to simplify— illustrates the placement of the implemented *building.js* file, where the core processing takes place. Secondly, it is briefly described each of the research-dependent folders in order to clear understand their general functionality within the Web application's context.

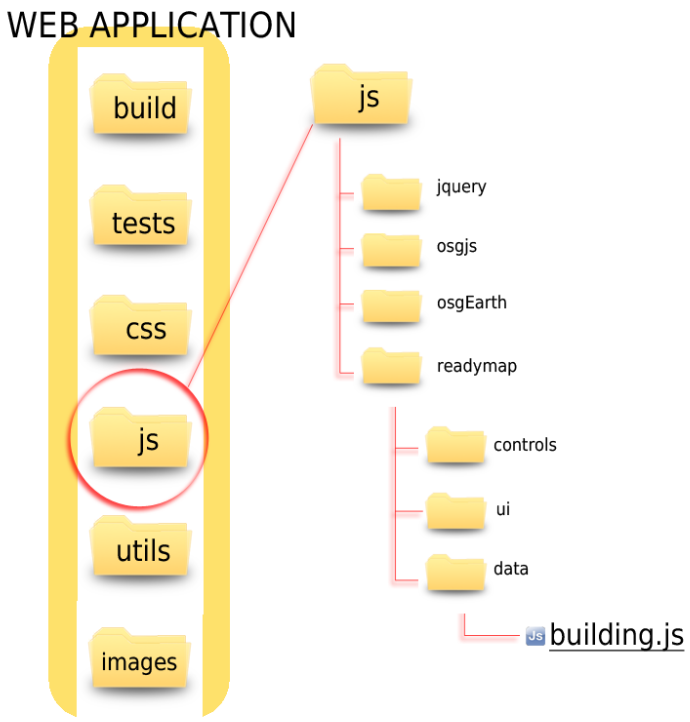


Figure 24. Application's structure diagram

Thus, the present packages are:

- *Build*: includes a single compiled JavaScript file made as an aggregation of all necessary JavaScript files located in *js* folder.
- *Tests*: includes both data and HTML files.
- *CSS*: includes applications' style sheet files.
- *Utils*: includes a Python compiler used to create a compiled JavaScript file.
- *Images*: includes a set of necessary images for the applications' user interface.
- *Js* :
    - *jquery*: includes jQuery JavaScript library compiled as a single file.
    - *osgjs*: includes OSGJS graphic library compiled as a single JavaScript file.

- *osgearth*: includes sixteen JavaScript files which provide GIS functionalities to OSGJS. OSGEarth provides geo-processes based on graphical OSGJS functionalities. Tile, HeightField, EllipsoidModel or MercatorProfile are just little examples.
- *Readymap*:
  - *Controls*: includes eight JavaScript files that provide globe's manipulation implementations as well as user-interaction functionalities.
  - *Ui*: includes six internal JavaScript files which provide items manipulation and position support.
  - *Data*: includes a series of JavaScript files that provides particular data functionality. WMS, TMS and building JavaScript files are located in this folder.

Additionally, Array.js, Math.js and Readymap.js files are placed in the *js* folder providing a necessary mathematical support.

4.3.3 Application Behaviour

Next, it is detailed the application Process Flow Diagram (see Figure 25) which represents the general data workflow along the Web application. Special care has been taken in depicting the processes, and the application's architecture components to which they belong to.

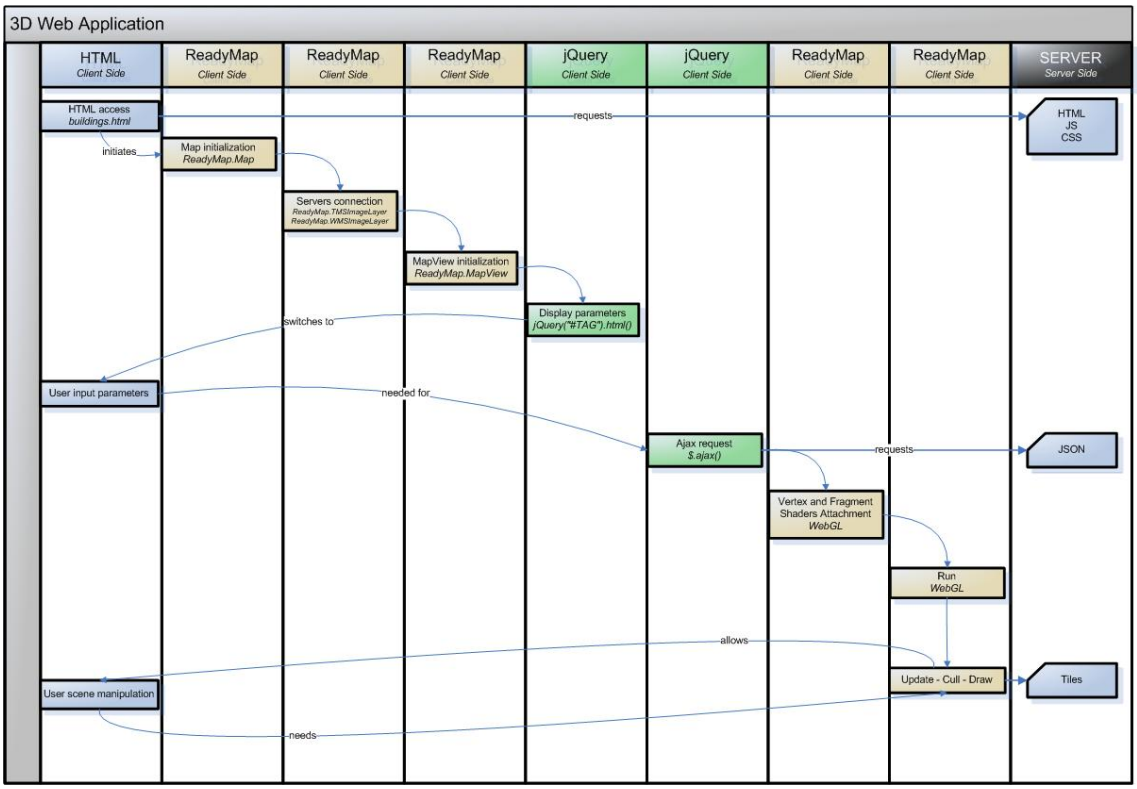


Figure 25. Application's process flow diagram

As observed most of processing takes place on the client's side, accessing to the server just for gathering the required set of files. Following data flow, the application firstly starts when end-user accesses through the browser to "buildings.html" located on server side.

Secondly, server provides in the same way all the application-dependent files such as JavaScript or CSS files and consequently Map initialization process starts, initializing a map object which establishes the grounding for following properties addition.

Thirdly, WMS and TMS server's requests are defined in addition to a mapView object, which is then created and ultimately linked to canvas HTML5 element. Additionally a series of geographical information —such as latitude or longitude coordinates— is represented on-screen.

At this point Web applications yields the control to the user, who is expected to input some data parameters which are going to define the ongoing application behaviour. End-user can choose the desired WebGL rendering mode, thematic colour representation and VBO management, which is going to determine the behaviour of the next processing AJAX stage.

AJAX stage relies on jQuery library functionality in contrast to previous processing based on implemented ReadyMap functionalities. AJAX processing is itself the core stage of this research and therefore it is going to be explained in more detail (see Figure 26).

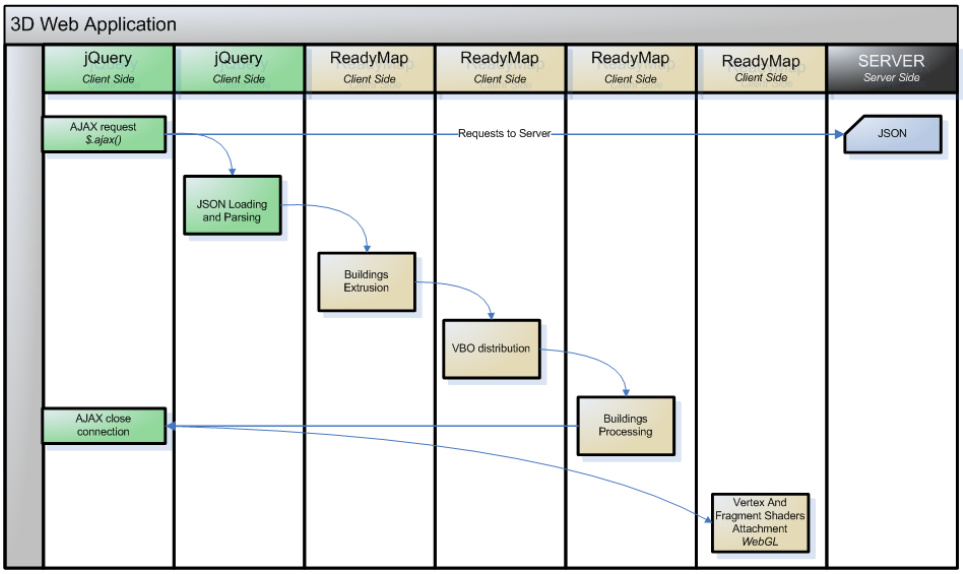


Figure 26. Application's process flow diagram detail

As seen, AJAX request gathers from the server the building's definition JSON which is loaded and parsed afterwards. Additionally, a series of buildings extrusion, children scene graph distribution and graphical buildings' reconstruction are implemented, being the core functionality of this buildings' representation solution.

Once AJAX request and buildings' processing are finished, both WebGL fragment shader and vertex shader are defined and initialized (see Figure 25), starting therefore a series of graphical operations —such as culling, updating or drawing— which optimizes in the end applications' graphical performance.

At this point graphic card contains the corresponding processed graphical data and consequently, from this point onwards, the end-user has full control of the application. By exploring the scene, the graphical information need to be updated, creating a closed loop and a direct dependency between on-screen camera movements and updating, culling and drawing methods.

4.3.4 File Organization

Four main files were either developed or used in order to provide as whole, the essential graphical application's results. a) *Buildings.html*, b) *jQuery.js*, c) *ReadyMap.js* and d) *Buildings.json*.

a. *Buildings.html*: It is an HTML file which loads, gathers and executes tasks relying on the following JavaScript and JSON files. It is the main grounding for a final graphical representation, especially based on canvas HTML5 element presence.
b. *jQuery.js*: It is a JavaScript Library file that simplifies HTML document traversing, event handling, animating, and AJAX interactions for web development.
c. *ReadyMap.js*: It is a JavaScript file which gathers and stores all implemented functionalities relative to OSG.js, OSGEarth.js and the ones particularly implemented for this research.
d. *Buildings.json*: It describes the geometry and semantic of the data set.

Alternatively, it has been slightly modified two CSS files which help to distribute and graphically embellish HTML elements visualization and consequently final building's representation.

4.3.5 Data Management

In this section it is presented the adopted methodology with regard to two new concepts present on the data management of this *Processing* stage: Level of Detail (LOD) and VBO.

For the object-based data models of GIS, LOD is embedded in rules of generalization. It results difficult to find well-defined metrics for the LOD definition. As a result, any ordinal scale would be appropriated to define LOD concept as far as it relies on such rules of generalization (Goodchild & Proctor, 1997).

Taking into consideration that it is expected a forthcoming LOD object inclusion on OSGJS; this research has focuses on rendering the building's dataset without considering any dynamic LOD building's representation.

Considering existing 3D urban data exchange formats, CityGML is one of the most used and widespread building exchange data format on market. CityGML is a common information model for the representation of 3D urban objects. It defines the classes and relations for the most relevant topographic objects in cities and regional models making consequently an extensive use of LOD concept (Open Geospatial Consortium, 2008).

The importance of LOD concept in both a map visualization and data definition is therefore plausible and something to be considered when developing visualization solutions as this research does (Goodchild & Proctor, 1997).

At this point, it is relevant to mention with regard to buildings' representation that, an optimized approach would have been to iteratively adapt and bind building's data representation to each particular LOD, optimizing in consequence the amount of buildings per area extension. In order to exemplify such approach, high altitude top-down views would represent a sparse and generalized group of buildings attached to a particular area's extent on-screen; whereas very low altitude top-down views would represent the totality of buildings attached to such area extent.

OSG has implemented an LOD object in its C++ version which mainly provides the LOD functionalities previously exemplified (OSG, 2009). Nevertheless OSGJS has not implemented yet any LOD object so that, it is still not possible to properly bind any building to a LOD object in order to define to what extent and detail it should be represented, so that it has been included as an improvement to come.

Focusing on graphical performance, two different approaches with regard to data management have been defined. They have been implemented to prove how the Web application would behave in view of different data management configurations. The major difference between them has been determined by the number of children to add to the scene graph's root in which this research relies on.

Each child on the scene graph represents a Vertex Buffer Object (VBO) virtual concept. WebGL uses primitives to draw 3D objects (further explained in *4.3.7 Data Visualization*) which are consequently defined as a series of attributes —such as colour or coordinates— which can be also stored in various arrays. The vertex data stored in such arrays are stored in client memory which needs in turn to be copied from the client memory to graphics memory. These data transferring should be done on every draw call

notably slowing down rendering performance which can be improved relying on the VBO concept. VBO mechanism allows these arrays data to be stored in high-performance memory, and consequently, to significantly improve the rendering performance and reduce the memory bandwidth (Munshi, Ginsburg, & Shreiner, Vertex Buffer Objects, 2009).

Regarding the likelihood of extreme cases as a result of adding VBO, it can be described mainly two: *array of structures* and *structure of arrays*.

- *Array of structures* defines a virtual array structure in which each particular building is added to the same VBO, which in the end is added to the root scene, so that such buffer contains the whole buildings' geometries.
- *Structure of arrays* defines a virtual arrays structure in which each particular building is added to the root scene as a single VBO, forcing the graphics card to execute expensive WebGL internal state changes.

The above cases have been defined as extreme situations likely to happen directly depending on user input. Every one of the weighted solution defined between those two illustrated cases has been also considered.

4.3.6 Data Preparation (ReadyMap)

This research has looked for giving solution to building's representation relying on a combination of the source code present on the simplest ReadyMap application, and the implementations done by the author of this research.

The Web application itself has depended on several sub-processes detailed in the following sections: *Server data loading and parsing*, *Buildings extrusion* and *VBO management*.

4.3.6.1 Server data loading and parsing

The data have been loaded into the browser in this *Processing* stage through a series of AJAX request —*$.ajax()*— provided by jQuery library (The jQuery project, 2010). Afterwards it has been used a JSON parsing method —*$.parseJSON*—, present in jQuery as well, in order to convert building´s JSON data into JavaScript objects fully comprehensible to any browser (The jQuery project, 2010).

4.3.6.2 Buildings extrusion

Considering that the input data has just provide the geometrical outline of each building, in addition to the corresponding height attribute; it has been necessary to define not only the building's ground floor, but also their roofs. Such assumption has been implemented by creating and alternatively inserting new points into building's geometry data definition. In this way, the height of each original point has been set to zero; and similarly, the height of the replicated point has been set as the building's height value —depicted in the JSON file— belonging to such point. Following *Figure 27* details the processing.

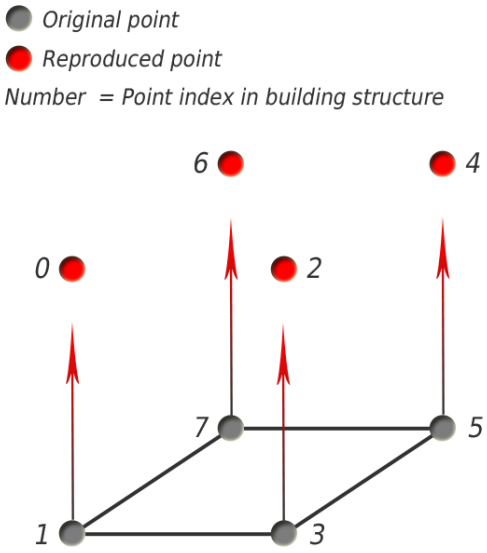


Figure 27. Buildings extrusion diagram

It is relevant to mention that QGIS has exported, the set of vertices in a clockwise order, being essential for the primitives drawing, detailed later on (see *4.3.7.3 Drawable Geometry*).

4.3.6.3 VBO management

The main idea behind using OSGJS is to manage the group of buildings as a group of children that are suspended from a root node. VBO management has determined the number of children that have been added to the root scene. Subsequently it has been assigned a similar number of buildings to be processed, to each of the children —VBO— (see *4.3.5 Data Management* for complete information). Such methodology has

been implemented, mainly for testing purposes, by calculating the total number of buildings and further dividing them into the number of user-defined VBO.

Each group of buildings, characterised by a first and last building numerical index, have been assigned to the corresponding instance of *BuildingNode*, and later on sent to *Data Visualization* stage (see *4.3.7 Data Visualization*).

BuildingNode is defined as a JavaScript object, additionally described as the complete set of attributes and methods implemented in this research, which give support to any ReadyMap-based building's representation.

In order to illustrate the process, the two extreme cases of building's representations — array of structures and structure of arrays— are detailed below (see *Figure 28*, *Figure 29* and also *4.3.5 Data Management* for complete information).

- **mapView**: ReadyMap.MapView
  - **root**: osg.Node
    - **children**: Array[3]
      - **0**: osgearth.MapNode
      - **1**: ReadyMap.BuildingNode
      - **2**: ReadyMap.BuildingNode

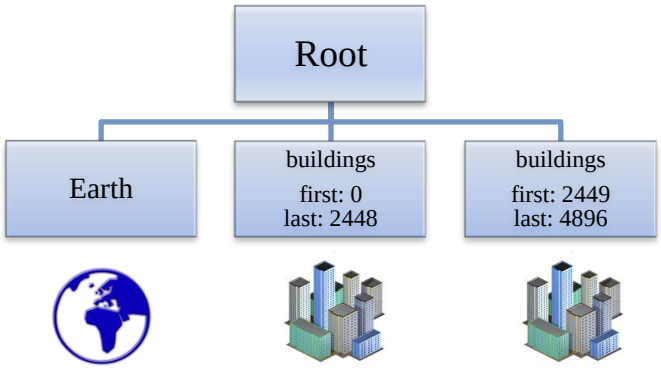


Figure 28. Array of structures graphical example

- **mapView**: ReadyMap.MapView
  - **root**: osg.Node
    - **children**: Array[4897]

- **0**: osgearth.MapNode
- **1**: ReadyMap.BuildingNode
- **2**: ReadyMap.BuildingNode

  …
- **4896**: ReadyMap.BuildingNode

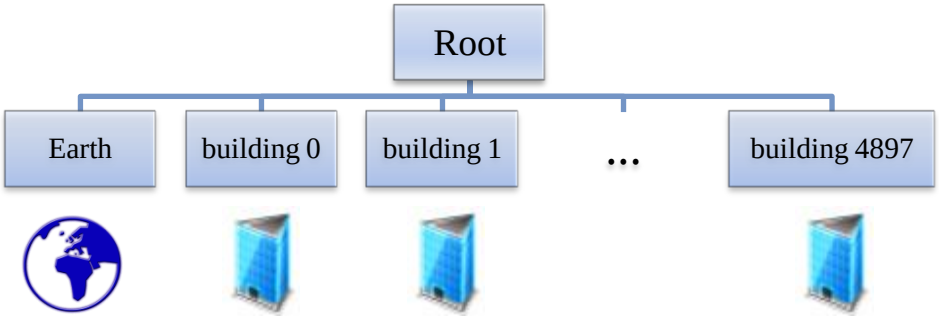


Figure 29. Structure of arrays graphical example

It should be noted the difference between "buildings" in *Figure 28*, and "building" in *Figure 29*.

4.3.7 Data Visualization (ReadyMap)

In this section it is described the core processing related to building's rendering process. It is further divided in four main parts: *Preliminary Concepts*, *Drawable Geometry*, *Reference Geometry* and finally *Drawable and Referenced Insertion*.

4.3.7.1 Preliminary Concepts

After the *Data Preparation* processing, it has resulted necessary to define a set of graphical and geographical processing to correctly visualize the data on the globe's surface. This complex assignment has been entirely done within OSGJS and OSGEarth frameworks, compiled as *ReadyMap.js* JavaScript file (see *Figure 24*).

On the one hand, OSGJS provides the development of high-performance 3D graphics applications relying on the concept of a scene graph, providing an object-oriented framework on top of WebGL. On the other hand, OSGEarth provides a 3D terrain rendering functionality following open standards. The consequence of making use of both has resulted in a synergic fusion of graphical and geographical fields shaped as a *BuildingNode*.

Each *BuildingNode* has been described by two principal attributes: *Geometry* and *MatrixTransform* (see *Figure 30*).

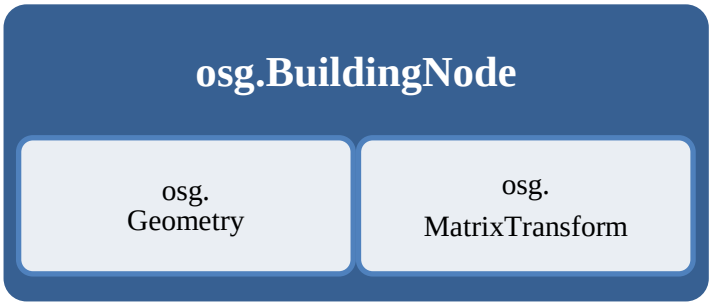


Figure 30. osg.BuildingNode object

It should be noted that *osg.BuildingNode* and *BuildingNode* refers to exactly the same JavaScript object, distinguished by a prior notation —"osg."— which has been used to call it from the source code.

As soon as an instance of a *BuildingNode* is made, it is defined and fulfilled the next three attributes: *origin*, *heightField* and *heightWeighting:*

- *origin* vertex has been created with the first tuple of coordinates —latitude and longitude— defined in the first vertex of the first building. This has been used in turn as a geometric anchor point for the rest of buildings associated to this particular *BuildingNode* (see *Figure 28* and further explained in *4.3.7.3 Referenced Geometry*).
- *heightField* has been defined to store the heights of the vertices associated to each building:

```
var heightField  = [    building0_height0, 0, building0_height0, 0,…
                        building1_height0, 0, building1_height0, 0,…
                        building2_height0, 0, building2_height0, 0,…    ]
```

- *heightWeighting* —also known as vertical exaggeration— has been defined to magnify the height of the buildings on the globe.

From this point onwards, the main core processing has taken place.

4.3.7.2 Drawable Geometry

The geometry for each particular BuildingNode has been described as a set of *primitives* and *attributes*, stored in a group of arrays.

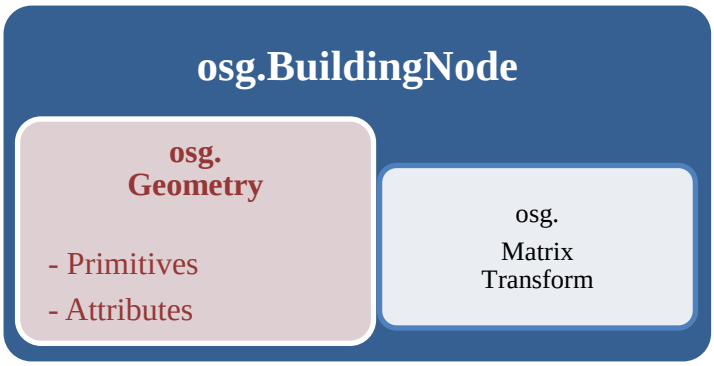


Figure 31. osg.Geometry object

Primitives have been defined as five one-dimensional element arrays where the indices of vertex's geometry have been stored in:

- *elements*: has stored the vertices' indices of the external building's faces.
- *roof*: has stored the vertices' indices of the face or faces that a building's roof consists of.
- *line1*: has stored the vertices indices connections of the building's edges.
- *line2:* has stored the vertices indices connections of the building's edges.
- *points*: has stored the vertices indices.

Primitive values have been stored into each *BuildingNode* subsequently binding them to an *osg.DrawElements* object. *osg.DrawElements* has managed the render of the indexed primitives by firstly defining the WebGL rendering mode —GL_TRIANGLES, GL_LINES or GL_POINTS i.e.— (Khronos Group, 2010); and secondly by assigning the corresponding set of *primitives*.

In order to illustrate how *osg.DrawElements* has been called, a code sample is presented below.

```
Var tris = new osg.DrawElements(gl.TRIANGLES, new
osg.BufferArray(gl.ELEMENT_ARRAY_BUFFER, elements, 1));

this.geometry.getPrimitives().push(tris);
```

Figure 32. osg.DrawElements geometry insertion

WebGL GL_TRIANGLES rendering mode has been used to represent *elements* and *roof* arrays data. WebGL GL_LINES rendering mode has been used to represent *line1* and *line2* arrays data. And finally WebGL GL_POINTS rendering mode has been used to represent *points* array data.

On the other hand, three main variable-dimensional arrays related to *Attributes* have been defined:

- *verts* (3dim): has stored vertices coordinates attribute.
- *normals* (3dim): has stored vertices normal attribute.
- *colours* (4dim): has stored vertices colour attribute.

Attribute values have been stored into each building's vertex in a similar way as before (see *Figure 33*), excepting the type of buffer that has been used.

```
this.geometry.getAttributes().Color = new osg.BufferArray(gl.ARRAY_BUFFER,
colors, 3);
```

Figure 33. osg.BufferArray binding

In order to get the best performance it has been used ARRAY_BUFFER to store the vertex data, and ELEMENT_ARRAY_BUFFER to store the indices of the primitives to be rendered (Munshi, Ginsburg, & Shreiner, Vertex Buffer Objects, 2009).

In the following sections *Primitive* and *Attribute* arrays processing are described in-depth.

4.3.7.2.1 Primitive Arrays

In this section it is described the way in which each particular rendering mode-dependent array has been filled. *Primitive* arrays have been filled at once, unlike *Attribute* arrays which have been filled iteratively per building's vertex.

The set of indices that have been stored in no matter which of the next rendering mode-dependent arrays, have been further referred to the matching index of coordinates — longitude, latitude, height— stored in *verts* array (see *4.3.7.3 Referenced Geometry* for further details).

Polygons rendering mode (GL_TRIANGLES)

In this section it is described the way *elements* and *roof* arrays have been filled with the consequent vertices indices, corresponding to the polygons rendering mode.

Relying on GL_TRIANGLES for polygons rendering, it is drawn a series of separated triangles, treating each triplet of indices gathered from either *elements* or *roof* arrays, as an independent triangle (Munshi, Ginsburg, & Shreiner, Triangles, 2009).

Building's side faces wrapping

Building's side faces wrapping has been accomplished sequentially storing in *elements* array, the corresponding triplet of indices. In this way a sequence of triangles got defined resulting in rectangular coplanar faces defined each two triangles (see *Figure 34*).

> *elements = [0,1,2,1,2,3,2,3,4,3,4,5…];*

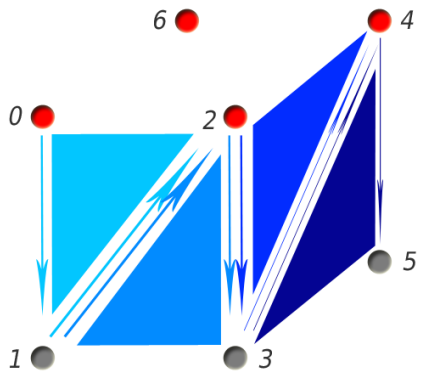


Figure 34. Triangle creation processing

It should be noted the order of vertices, the face creation flow, and the way each face has shared their triple of vertex with adjacent faces.

Building roof wrapping

Building's roof wrapping has been accomplished adapting a JavaScript-based Constrained Delaunay Triangulation library (CDT). Seeing that, roofs vertices have been triangulated to create a sequence of triangles which have ended up covering the entire roof surface.

Considering the inexistence of any wide range of Constrained Delaunay triangulation libraries written in JavaScript, it has been decided to adapt and modify *poly2tri* JavaScript library (Google, 2011) —based on (Zalik & Domiter, 2008)— as the most suitable alternative. In this way no additional triangulation Web Service has been required, implementing an innovative client-side processing for WebGL-based geometry.

Given that, explaining in-depth the way that *poly2tri* has worked is completely out of scope of this research due to extension reasons, it is following explained the way the input data has been transferred to the library and also the obtained output.

As a result of managing 3D data models, it has been firstly necessary to extract the vertex indices belonging to a same roof outline. Regarding *Figure 34* it can be noticed that *roof* elements are odd-value restricted.

Secondly, it has been required to gather the corresponding latitude and longitude coordinates with regard each of the previously extracted vertex indices. Such coordinates have spatially described each of the vertices on the same two-dimensional plane to be following transferred to the CDT.

Thirdly, the CDT has processed the sent data. The resulting output has been described as an ordered array of triangles which have defined the roof coplanar polygons. Each of those triangles has been consequently depicted as a triplet of coordinates.

Fourthly, inversely to the second step, it has been required to transform the obtained coordinates present in each of the ordered array of triangles into an ordered flow of vertex indices.

Finally, those vertex indices have been stored in *roof* array and additionally bound to GL_TRIANGLES rendering mode to be further rendered. Following this methodology it has been possible to represent any roof shape, with an optimized and graphically perfect set of coplanar polygons.

Relying on the results obtained from running several internal tests (see *Appendix A*), it has been finally achieved a successful accuracy percent of 99,37% in building roof creation, obtaining only a 0,63% of error, due to an indirect weighting between the data preparation processing (see 4.2.4 *Data Preparation*) and the CDT data processing.

Furthermore, it has been noticed that the more the closer the points have been either the original or prepared data, the higher has been the probability of failure.

Figure 35 illustrates a successfully triangulated complex shape.

Figure 35. Complex roof shape triangulation

Lines rendering mode (GL_LINES)

In this section it is described the way *line1* and *line2* arrays have been filled with the consequent vertices indices connections, corresponding to the lines rendering mode.

Relying on GL_LINES for lines rendering, it is drawn a series of separated lines, treating each tuple of indices gathered from either *line1* or *line2* arrays, as an independent line (Munshi, Ginsburg, & Shreiner, Lines, 2009).

Building vertices connection

Relying on the same building's structure (see *Figure 27*), the vertices connections have been depicted as following.

Considering that any building's geometry can be generalized as two parallel horizontal planes —ground floor and roof— being externally connected by a set of vertical planes —side faces—; it has been proposed a combination of two dataflow in order to cover the vertices alternatively in the next presented way.

The algorithm has been initialized on the vertex 0, defined on a positive and above zero value since it belonged to the building's roof, so that:

line1 has started in vertex 0 and has continues connecting vertices following the next pattern: 0, 1, 3, 2, 4, 5, 7….

line2 has started in vertex 0 and has continued connecting vertices following the next pattern: 0, 2, 3, 5, 4, 6, 7….

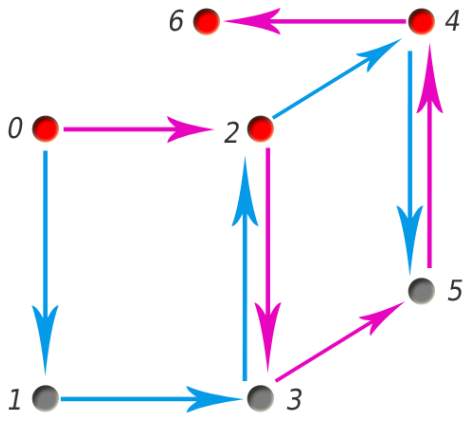


Figure 36. Lines creation processing

Consequently, *line1* and *line2* have swept and connected the total amount of vertices, defining side faces and roof contours with no extra roof treatment needed. This approach has entailed a cost of having as many numbers of repeated edges as numbers

of vertices the building's ground floor consists of minus one —three in case of a cube for instance—.

Finally, the application's rendering mode has been set to GL_LINES in order *line1* and *line2* to be treated as line representation data (see *Figure 36*), obtaining a 100% of successful vertices connections.

Points Sprites rendering mode (GL_POINTS)

In this section it is described the way that *points* array has been filled with the consequent vertices indices, corresponding to the point sprites rendering mode.

Relying on GL_POINTS for points rendering, it is drawn a series of isolated points, treating each vertex index gathered from *points* arrays, as a single point (Munshi, Ginsburg, & Shreiner, Point Sprites, 2009).

Simple vertices representation

Relying on building's structure (see *Figure 27*), the building's vertices representation has been accomplished sweeping the total amount of building's vertices, storing the corresponding index into *points* array and finally setting WebGL rendering mode to GL_POINTS.

4.3.7.2.2 Attribute Arrays

In this section it is described the way that *colour* and *normal* arrays are filled for each particular building. Unlike the *primitive* arrays; *attribute* arrays have been in sync and sequentially filled relying on per-vertex operations.

The particular case of *verts* array has been further described in the next section (see *4.3.7.3 Referenced Geometry*) considering that it has been intrinsically related with the matrix transformations.

Contrary to *primitive* arrays, *attribute* arrays processing do not depend on the rendering mode, so the following descriptions have been common and to all the rendering modes representations depicted in this research.

Colour gradient

Concerning data colour representation, it has been implemented two different colour representation for two different attributes sources: height colour theme and cadastre area colour theme. In each of these representations, a *colour* array has been fulfilled with as many four-dimensional vectors —red, green, blue, alpha—, as number of vertices the buildings consist of.

heightField data has been indirectly used for vertex colour assignment in the height colour theme representation. In this building's representation a set of colours have been predefined, so that each particular vertex colour has been depicted with regard to its height attribute value.

Furthermore, the previously defined vertical exaggeration (see *4.3.7.1 Preliminary Concepts*), has been used once more in colour transformations so that direct relation between height and colour, has not been distorted.

Additionally, ground floor vertices heights have been set to black considering that it is a common value for the totality of buildings. Consequently, it has been only changed the colour applied to the roof vertices. Following this methodology, the obtained result is depicted as a scaled colour gradient (see Figure 37) formed in Fragment Shader stage of the OpenGL ES 2.0 pipeline (Shreiner, OpenGL Programming Guide, 2010).

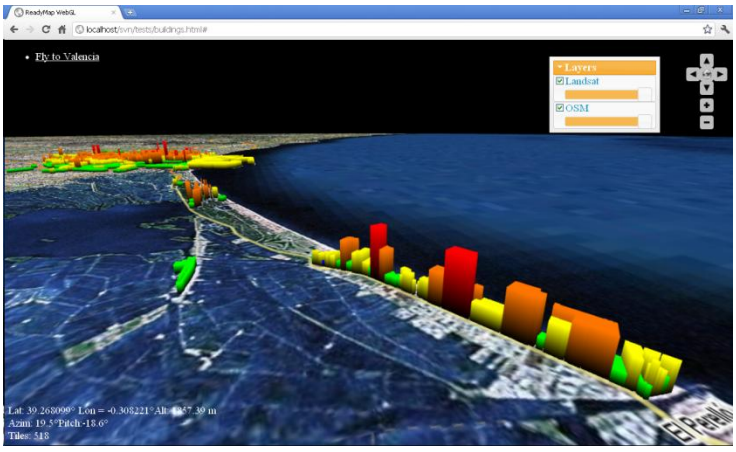


Figure 37. Building's colour gradient general perspective

The same processing has been applied to cadastre area colour theme, taking into account that the represented data have come from a different attribute, and consequently a different representation has been created.

Scene normals

A normal vector (or *normal*, for short) is a vector that points in a direction that is perpendicular to a surface. This vector is used by WebGL to determine how much light a plane receives at its vertices (Shreiner, OpenGL Programming Guide, 2010).

Due to research topic and background (see *1.1 Background*), *normal* processing has not been considered this time, being therefore postponed for future releases in case it is interesting.

Normals within the context of this research, would only embellish —consequently dirt— the colour representation, contrary to what it has been implemented, a solid face colouring for resulting incident solar radiation's output i.e. Nevertheless, *normal* data have been taking into account inserting unprocessed normal data into *normal* array in order to fulfil the required number of elements for validation purposes.

4.3.7.3 Referenced Geometry

In this section it is described the *verts* array processing and the sequence of Matrix transformations which have been applied to the original coordinates. These transformations have provided the new generated coordinates necessary to visualize the building's geometry on top of WebGL globe's surface.

As defined (see *4.3.7.3.1 Primitive Arrays*), *verts* has belonged intrinsically to *primitive* array group: nevertheless considering its associated Matrix Transformation processing, it is explained in this section.

4.3.7.3.1 Matrix Transformation

Matrix Transformations have consisted on a series of sequential steps (see *Figure 30* and *Figure 40*) which have been implemented to transform the original vertex coordinates into new generated ones.

These new coordinates have been further stored in *verts* array and have served to both place and locate each building's vertex on the globe's surface. This has been achieved by attaching to the corresponding *osg.Node* object, the data present in *verts*.

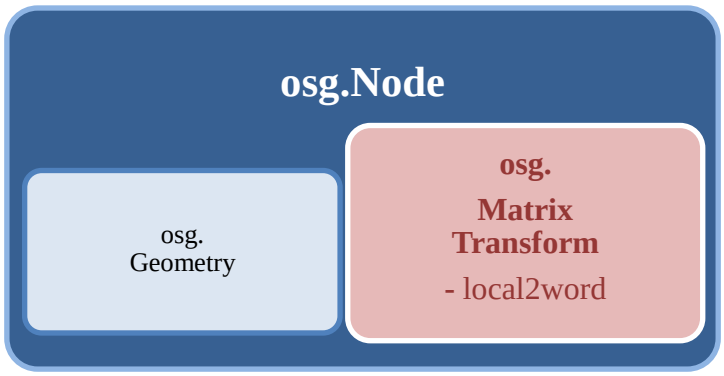


Figure 38. osg.MatrixTransform object

Considering such approach, two different processing have been implemented at this point: *Anchor point processing* and *ordinary point processing*.

Anchor point processing

Firstly, it has been necessary to define an anchor point for each VBO to be treated. Such point has been defined as the corresponding triplet of coordinates described in the first vertex of the first building that has been processed at that time.

Secondly, the latitude and longitude coordinates from such anchor point have been transformed from geographical coordinates degrees, to radians; while height value has rested unaltered described in meters.

Thirdly the new coordinates described in radians have been transformed to Earth-centred Earth-fixed coordinates (Iliffe & Lott, 2008).

Finally, it has been required to define a *world2local* transformation matrix in order to obtain the coordinates that have being used as an anchor point, all the rest of building's vertices belonging to the same VBO. *Figure 39* exemplifies the complete processing.

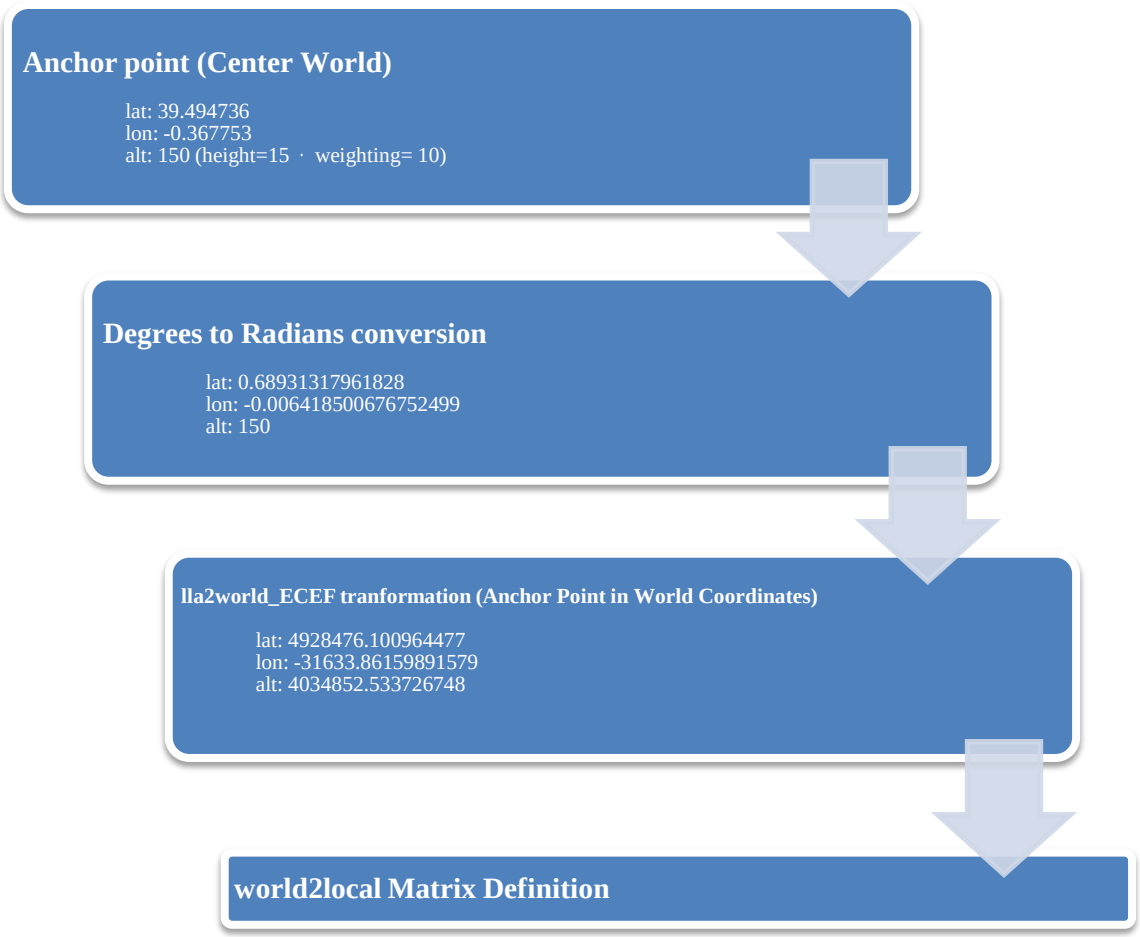


Figure 39. Anchor vertex geometric transformations

Ordinary point processing

Similarly to *Anchor Point processing*, for *ordinary point processing* it has been also necessary to load the original vertex data, convert it into radians and transform it to world coordinates.

Additionally, it has been required a new *world2local* transformation matrix described as the inverse of the *world2local* transformation matrix defined in previous *Anchor Point processing* stage.

This whole matrix processing has provided a complete methodology to transform each building's vertex, described as longitude, latitude and height, in a set of numerical values corresponding to WebGL Globe surface's coordinates. Following *Figure 40* describes the processing.

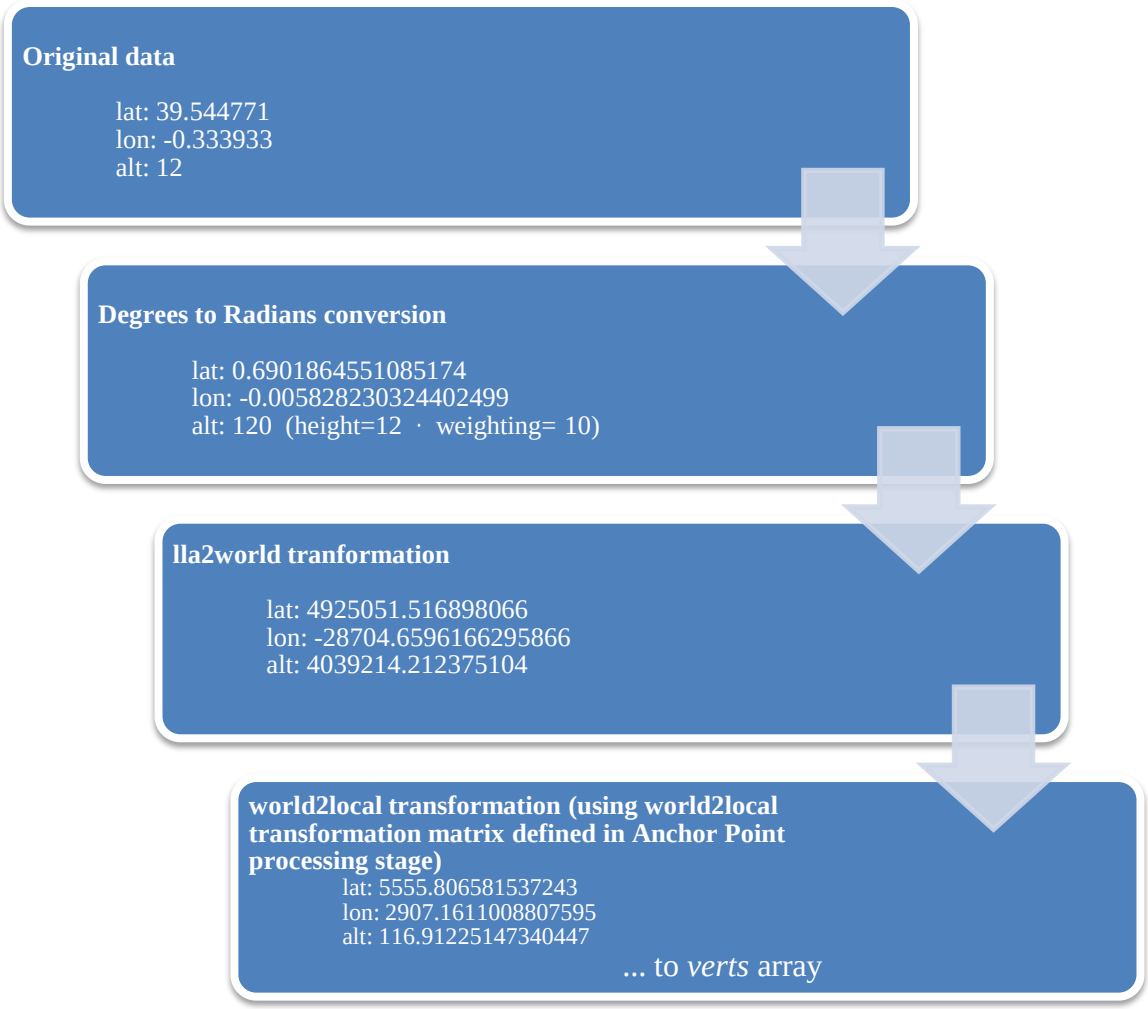


Figure 40. Ordinary vertex geometric transformations

4.3.7.4 Drawable and Referenced Geometry Insertion

In a final step, it has been attached to the same child:

- The *Drawable Geometry*, which have defined the topology of the buildings.
- The *Referenced Geometry*, which have accurately placed such topology on top of the globe's surface.

Next, it is illustrated this processing.

```
var xform = new osg.MatrixTransform();
xform.setMatrix(local2world);
xform.addChild(this.geometry);
this.addChild(xform);
```

Figure 41. Geometry and Matrix Transformation merging

4.4 Chapter Review

In a first part, it is explained in-depth the major processing of *pre-processing* stage with regard to data preparation. It is described how the data coming from the Valencian cadastre has been reprojected and further divided to process heights and ground floors' outlines separately. Furthermore, it is depicted a series of filtering applied to each particular building's geometry, in order to reduce the amount of redundant points without notably affecting to the building's geometries. Finally, it is shown how data's output is stored as a JSON data structure.

In a second part, it is explained the corresponding processes with regard to polygons, lines and points representations. It is described how each particular WebGL rendering mode has required a different data management. Additionally, it is depicted the coordinates transformations in order to place accurately each of the buildings on top of the globe's surface. And finally, it is presented how geometries and transformation matrix have been attached to the child.

Chapter 5

Results

5.1 Results Review

As soon as the Web application is accessed via the browser, end-user firstly receives a series of confirmation windows where it is required for him to choose the desired option (see *Figure 42*). It determines therefore the application's behaviour, and consequently the graphical appearance of the city of Valencia and the WMS/TMS layers to be visualized.

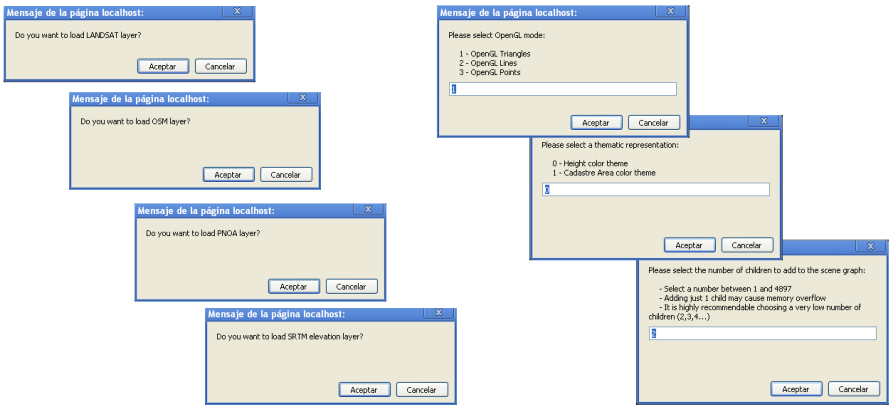


Figure 42. Dialogue boxes details

Secondly, it is required that end-user selects the desired WebGL rendering mode for the corresponding building's representation. As explained along this narrative, it has been implemented three different rendering modes which are subsequently represented on-screen as points, lines and polygons (see *Figure 43*, *Figure 44* and *Figure 45*).

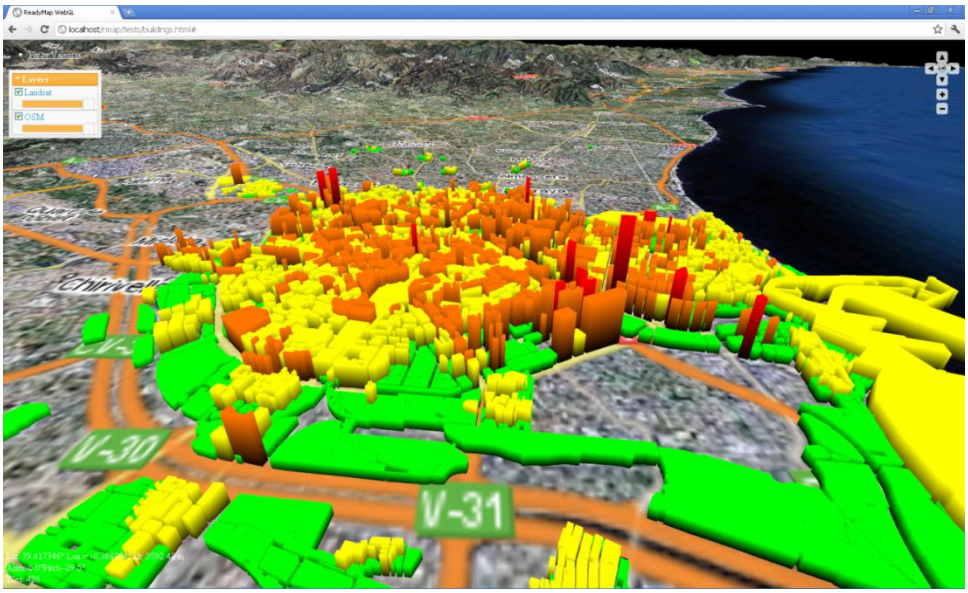


Figure 43. Polygon rendering mode representation

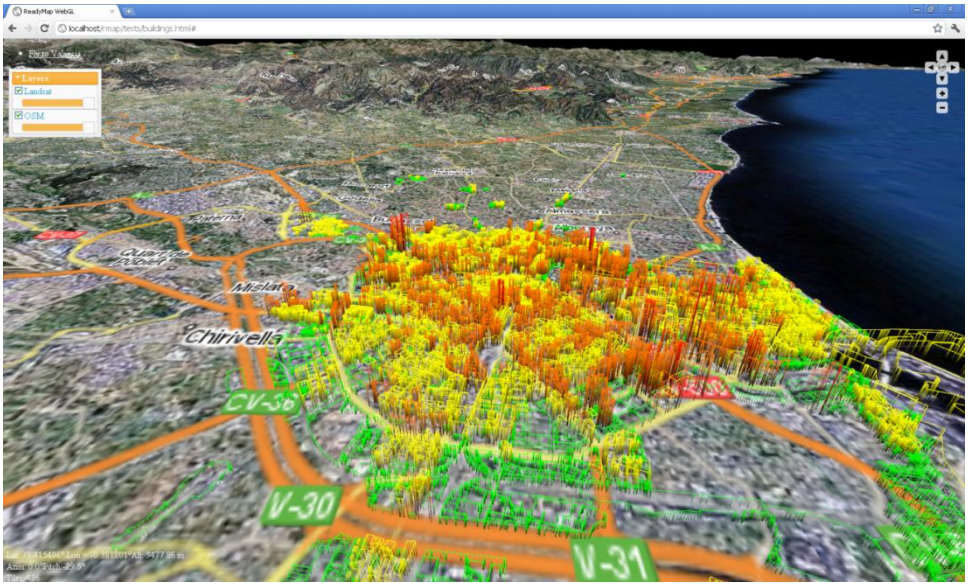


Figure 44. Line rendering mode representation

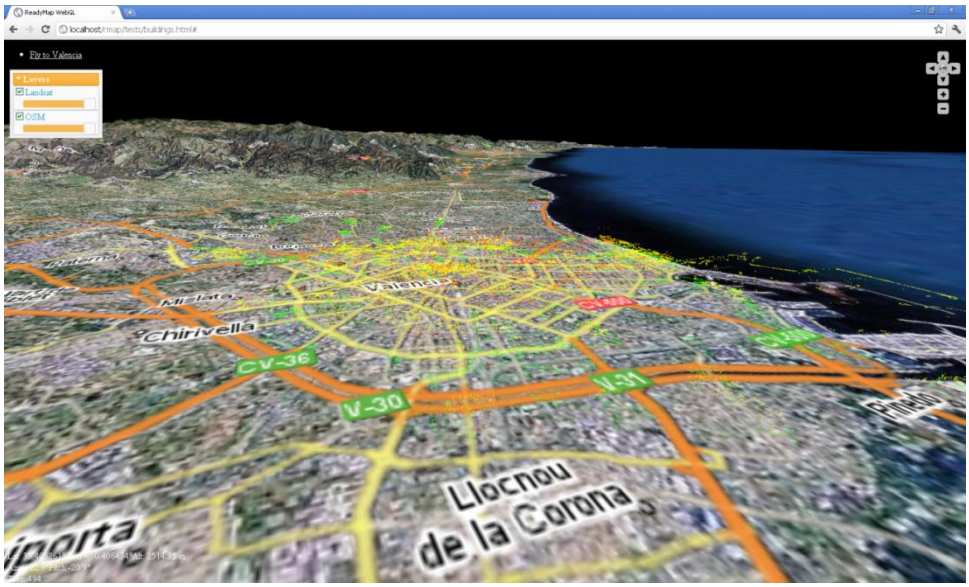


Figure 45. Points rendering mode representation

Thirdly, it is required for end-user to select the desired data representation. Thus, it has been implemented two different thematic colour-based data representations: cadastre's areas representation (see *Figure 46*) and heights representation (see *Figure 47*).

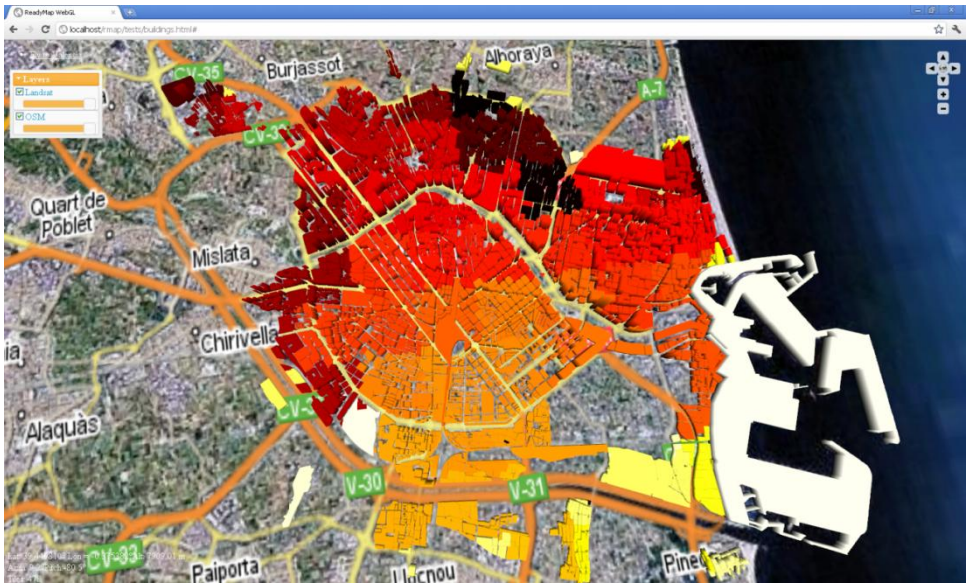


Figure 46. Cadastre area membership colour theme

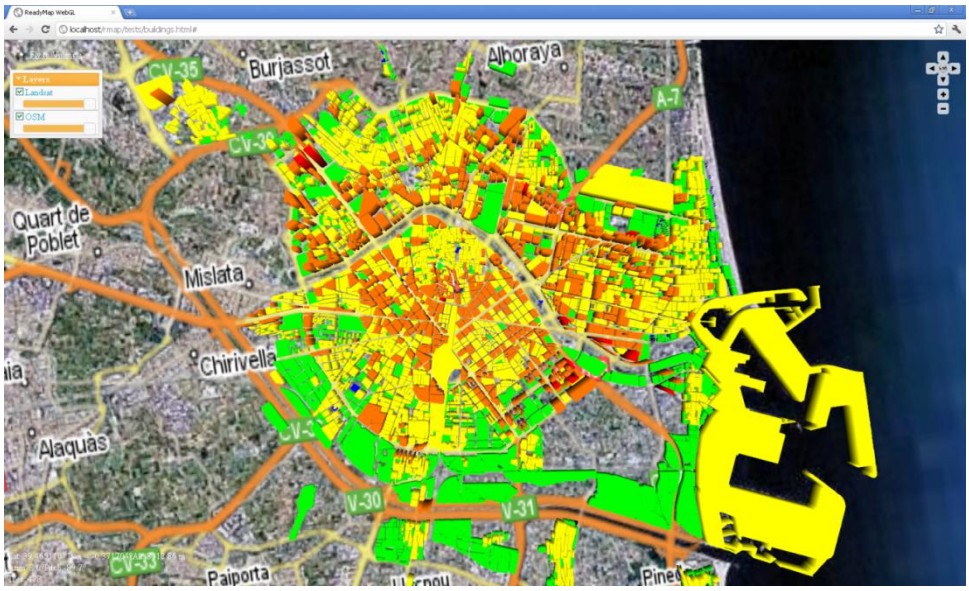


Figure 47. Height colour theme

Finally, it is required for the end-user to select the desired number of children to add to the scene. As a result it is concluded that the lower the number of children is defined, the more fluent the Web application has run.

The obtained frames per second (fps) with regard to the different number of children for a predefined static camera view are the ones described in next table.

Children	Frames per second (fps)
2	31.7
200	25.2
2000	7.8
4896	3.9

Table 7. Rendering performance

On the tested unit, there has been noticed a problem when attaching geometry to a single child. Therefore even considering there was not any confirmed memory overflow, for security reasons the minimum number of children to add has been set to two.

At this point, the application starts automatically loading, parsing and processing JSON file, thus it might be possible to have a black screen for a few seconds. As a result, ReadyMap globe, buildings' geometry and map's tiles are represented. Consequently, from this point onwards, end-user will have total control of the Web application, being

possible to explore the scene —panning, zooming, rotating or tilting the globe— and to change graphical parameters such as map layers' transparency.

For testing purposes, OSGJS provides a useful graphic statistics tool which makes possible to check in real time the application's performance, by simply adding "?stats=1" at the end of the Web address (see Figure 48).

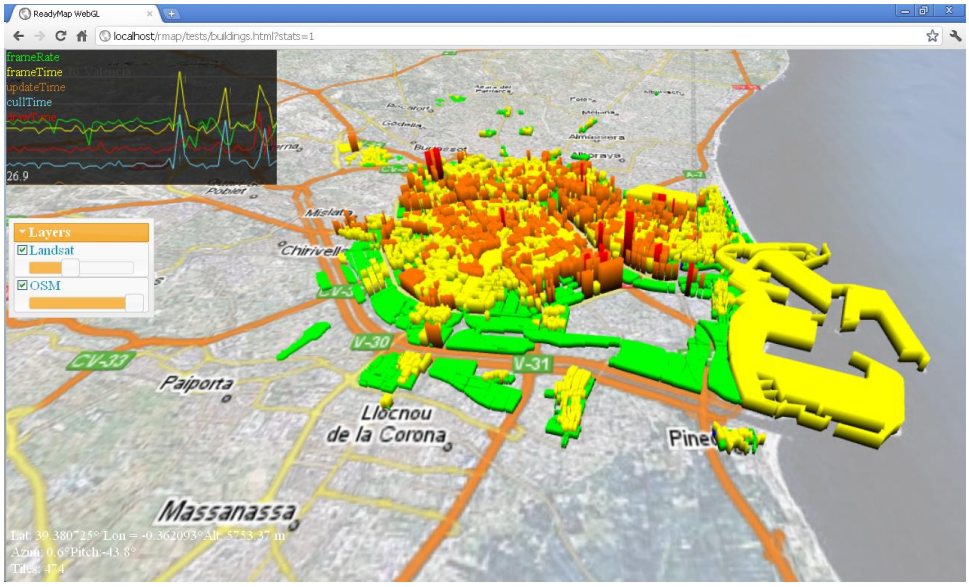


Figure 48. Graphic statistics panel

Application's source code and examples can be found at Github (Gastón Iglesias, 2012) in addition to the Documentation' Secretariat of the Erasmus Mundus Joint Master of Science Degree in Geospatial Technologies.

Chapter 6

Conclusions

6.1 Research Summary

The present research has analyzed, designed, implemented and tested each of the required stages to build an integrated solution to the generation of three-dimensional buildings. This was achieved on a WebGL standard-compliant framework relying on ReadyMap's globe.

The first chapter reviewed the impact of WebGL on Web-based graphical frameworks, mainly due to a notable improvement in performance of recent graphics cards. Web-based applications are becoming increasingly important up to the extent of being widely accepted that it is likely to be the new forthcoming processing platform. Many vendors and software developers have realized the potential of WebGL and the efforts on developing WebGL standard-compliant applications are increasing. GIS technologies are utilizing this progressive Web technology to discover new geo-processing possibilities. This research also solves the major part of WebGL-based visualization difficulties for current and forthcoming Web-based 3D geo-processing. To develop such a solution five main objectives, among some others, were defined:

- To propose a model for preparing, analyzing and filtering data from the Spanish cadastre.
- To propose a model to represent 3D buildings derived from 2D geo-data, on the surface of a WebGL-based virtual globe.
- To represent not only the buildings' geometry but any interesting parameter present or not present on the treated data.
- To develop a fully functional Web application prototype for both testing and representing the buildings present in obtained cadastre data.
- To develop a complete solution for representing the FIDE buildings' models in the same way as done for cadastre data (see *Appendix B*).

The second chapter reviewed the state of the art in WebGL-based virtual globes in addition to the existing JavaScript-based graphic libraries. Furthermore it was depicted the reasons of choosing ReadyMap framework, and the benefits of working on an architecture based on scene graphs.

Chapter three described an in-depth analysis about the data and application's requirements as well as the corresponding modelling of end-user use cases. Additionally it detailed the overall application's architecture and the corresponding time and economic planning.

Chapter four is the research's core chapter. It described in detail the required processing in order to finally render an on-screen visualization of the buildings recorded in the obtained cadastre data. Chapter four was divided into two main parts:

- The first part described the necessary steps in order to make the data ready to be loaded by the Web application. Thus it was discussed how data were first reprojected, and then divided into two files, named *constru.shp* and *masa.shp*. This was done in order to process separately the ground floor outlines, and heights of the buildings. Consequently, customized processing for each of the files was implemented. Thus, *constru.shp* was processed in order to obtain a single height value for each particular building, while *masa.shp* was filtered for redundant and collinear ground floor vertices of buildings, to both reduce unnecessary data and make the forthcoming triangulation process easier. Finally, both processing paths were merged by combining the buildings' geometry and height into a single JSON structure.
- The second part described the necessary steps in order to represent the processed data coming from the first part in the Web application. This second part detailed how data were processed so as to get an extruded building's three-dimensional geometry to pass like parameter to the main building's processing function. It also detailed the implementation done, firstly on each of the available rendering modes, which required different data management and additional adaptation of a Constrained Delaunay Triangulation library; secondly, on the management of Vertex Buffer Objects and what it entailed in terms of graphic performance; and finally, on the thematic-coloured data's representation. Additionally, this part

described the required geo-processing in order to accurately place each building on top of the ReadyMap globe's surface.

Finally, chapter five described the resulting Web application's outputs with regards to end-user choices. The descriptions were accompanied by a complete set of output images obtained from the application in order for the reader to get clearly understand the Web application's functionalities.

6.2 Research Contributions

This research proposed and implemented a complete methodology of three-dimensional thematic buildings' representation for the surface of a WebGL-based virtual globe, relying on an obtained dataset from Valencia's cadastre. In this methodology there was given full support for the Shapefile-based geo-data format, nevertheless the full processing can be applicable as well to any kind of geo-data as long as it can be exported into JSON format.

The implemented methodology demonstrated, from a graphical and geographical point of view, the possibility of importing, parsing, processing and finally representing any topological attribute —such as vertex coordinates— present in a dataset. Concurrently, the implemented methodology provides the functionality of representing semantic data as well, such as cadastre's area, a building's type definition or in general any kind of semantic attribute present in a cadastre dataset.

Additionally this research defined the groundings, within a WebGL standard-compliant framework, for the thematic representation of current and forthcoming urban-based geo-processing outputs, such as: incidents of solar radiation, noise impact, environmental impact of constructions, area population, disaster management and evacuation planning.

Furthermore, the existing JavaScript Constrained Delaunay Triangulation (CDT) library was adapted for this research. The adaptation allowed for the triangulation of vertices for GL_TRIANGLES rendering mode to be processed on the client's side instead of the conventional server side solution. Consequently, this improved the efficiency of time-consuming processing and reduced data traffic. This adaptation made possible to obtain any particular triangulation from any kind of complex face representation, as used in the FIDE methodology (See *Appendix B*).

Moreover by relying on OSGJS or scene graphs in general, the spatial organization was improved. In the case where there were just a small amount of buildings within a small area, the benefit is not clearly perceptible. However, in the cases where there were a very large number of buildings spread over an entire city, it was required to split them up into a grid to optimize the number of buildings per VBO having additional negative effects on efficiency for both too many and too few buildings per VBO. Consequently, scene graphs can help specifically on the breaking up of data spatially allows for faster culling since a large amount of geometry data can be processed and rejected quickly based on the viewing frustum.

Finally, the versatility and performance of WebGL in a browser was indirectly proved. In graphical terms, WebGL allows the web application to translate and update an entire city's geometry at a rate higher than 25 frames per second, in most tested units for a default application's configuration. Considering that it is a recent technology, still in the early stages of development, greater frame rate improvements are expected with future releases of both the WebGL framework and graphics hardware.

6.3 Future Work

The methodology discussed in this research, can be further improved in future works by:

- Implementing Level of Detail (LOD) functionality: OSG has migrated many of its functionalities to OSGJS version, nevertheless the LOD object structure has not been ported yet, so that this research has implemented the buildings' visualization without considering any graphical filtering with regard to the scene's extent. Consequently it is proposed to define a building's representation in sync with a map tile's creation, so that a top-down view with the view point farther away would represent fewer buildings than a closer and more detailed top-down view, therefore culling the geometry that is not in range.

- Considering digital terrain elevation data: This research has not considered, due to time constraints, buildings' heights above sea level. Nonetheless, it can be easily implemented following either of the following two approaches:

    - Dynamic height creation: the TMS server provides elevation tiles which are acquired in turn from SRTM elevation data. Consequently, it would be possible to define the building's zero level as the elevation value corresponding to the elevation tile where the building is projected. In this way terrain elevation and ground floor heights would match.
    - Static height creation: Considering it is possible to obtain either TMS data —SRTM elevation data is free and publicly accessible— or cadastre elevation data, it would be possible to add to the cadastre's data, an extra attribute depicting the height above sea level in order to be used while building's height definitions stage.

- Implementing standard exchange data models: Implementation for support on standard building's exchange data models, such as CityGML or IFC, will allow for greater versatility and portability.
- Condensing JSON structure: For this research, the JSON structure was proposed in terms of clarity for the reader. However, considering it is a server-stored file which ultimately will be sent over a network, it is highly recommendable to change JSON structure design into a more condensed set of values in order to reduce its file size and reduce data traffic.
- Obtaining pre-processing parameters: In order to obtain an accurate weighted vertex filtering, (See *Appendix A*) it must be considered to create a new script in order to obtain better optimized pre-processing parameters.

Bibliography

A. Pender, T. (2002). Use Case Model. In T. A. Pender, *UML Weekend Crash Course* (pp. 49-92). Wiley Publishing.

Ambiera. (2010). *copperLicht*. Retrieved February 12, 2012, from http://www.ambiera.com/copperlicht/index.html

Azad, K. (2007). *Better Explained*. Retrieved February 12, 2012, from http://betterexplained.com/articles/using-json-to-exchange-data/

Barker T., I. B. (2007). Technical Summary in: Climate Change 2007: Mitigation. Contribution of Working Group III to the Fourth Assessment. Cambridge, United Kingdom and New York, NY, USA: Cambridge University Press.

Belmonte, O., Carlos, G., & Maria, E. *IT Java-based projets development.*

Bochicchio, M., Longo, A., & Vaira, L. (2011). Extending Web Applications with 3D Features. *13th IEEE International Symposium on Web Systems Evolution (WSE)*, (pp. 93-96).

BOE. (2007, July 27). *Official Bulletin of the State.* Retrieved February 12, 2012, from Spanish National Geodetic Reference System: http://www.boe.es/aeboe/consultas/bases_datos/doc.php?id=BOE-A-2007-15822

Bonfatti, F. e. (1998). Guidelines for Best Practice in User Interface for GIS. *ESPRIT/ESSI project no. 21580*, (p. 96).

Brunt, P. (2010). *GLGE*. Retrieved February 12, 2012, from http://www.glge.org/

Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T., & Schmidt, C. (2008, June 16). *The GeoJSON Format Specification*. Retrieved February 12, 2012, from http://geojson.org/geojson-spec.html

CATGames Research network at Seneca College. (2010). *C3DL*. Retrieved February 12, 2012, from http://www.c3dl.org/

Cliffe, C. (2011). *CubicVR*. Retrieved February 12, 2012, from http://www.cubicvr.org/

Cropper, S. (2010). gvSIG is a viable robust alternative to commercially available GIS packages. *OSGeo Journal , 6*, 23-25.

de Smith, M. J., Goodchild, M., & Longley, P. (2009). Neighbourhood. In M. J. de Smith, M. Goodchild, & P. Longley, *Geospatial Analysis. A Comprehensive Guide to Principles, Techniques and Software Tools* (p. 70). Splint.

DFC Intelligence. (2010, May 25). *DFC Intelligence*. Retrieved February 12, 2012, from http://www.dfcint.com/wp/?p=277

FIDE Organization. (2009). *FIDE Conceptual Models*. Retrieved February 12, 2012, from http://www.fide.org.es/index.php/descargas/cat_view/24-modelos-conceptuales

Fraunhofer IGD. (2012). *X3DOM*. Retrieved February 12, 2012, from http://www.x3dom.org/

Garcia Belmonte, N. (2011). *PhiloGL*. Retrieved February 12, 2012, from http://senchalabs.github.com/philogl/

Garrido, S. (2011, July). *FIDE*. Retrieved February 12, 2012, from FIDE Geomertry's Descriptors: http://www.fide.org.es/index.php/descargas/doc_download/91-geometria

Gastón Iglesias, D. (2012, February 1). *Github ReadyMap building's repository*. Retrieved February 12, 2012, from https://github.com/RealFlow/godzi-webgl/tree/buildings

GDAL. (2012). *ogr2ogr*. Retrieved February 12, 2012, from http://www.gdal.org/ogr2ogr.html

Geomatics Engineering departement at the University of Applied Sciences Northwestern Switzerland. (2011). *OpenWebGlobe*. Retrieved February 12, 2012, from http://wiki.openwebglobe.org/doku.php?id=overview

Google. (2011). *Chrome Experiments*. Retrieved February 12, 2012, from http://www.chromeexperiments.com/globe

Google. (2012). *Chrome Expirements Terms of Service*. Retrieved February 12, 2012, from http://www.chromeexperiments.com/submit/

Google. (2012). *Google Maps*. Retrieved February 12, 2012, from http://maps.google.com/

Google. (2011). *O3D*. Retrieved February 12, 2012, from http://code.google.com/p/o3d/

Google. (2011). *Poli2tri*. Retrieved February 12, 2012, from http://code.google.com/p/poly2tri/source/checkout?repo=javascript

Guangwei, Y., & Zhitao, G. (2009). Scene Graph Organization and Rendering in 3D Substation Simulation System. *Power and Energy Engineering Conference (APPEEC)*, (pp. 1-4).

Heath, T. (1921). Heron's Triangle. In T. Heath, *A History of Greek Mathematics* (pp. 321–323). Oxford University Press.

Iliffe, J., & Lott, R. (2008). Conversion between ellipsoidal and geocentric cartesian coordinates. In J. Iliffe, & R. Lott, *Datums and Map Projections: For Remote Sensing, GIS and Surveying* (pp. 15-16). Whittles Publishing.

Illiffe, J., & Lott, R. (2008). Cartesian coordinates for engineering applications. In J. Illiffe, & R. Lott, *Datums and Map Projections: For Remote Sensing, GIS and Surveying* (p. 7). Whittles Publishing.

Inka3D. (2012). *Inka3D*. Retrieved February 12, 2012, from http://www.inka3d.com/

Jax. (2012). *Jax*. Retrieved February 12, 2012, from http://blog.jaxgl.com/

Kay, L. (2009). *SceneJS*. Retrieved February 12, 2012, from http://www.scenejs.com/

Khronos Group. (2011, December). *Blacklists And Whitelists*. Retrieved February 12, 2012, from http://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists

Khronos Group. (2010). *OpenGL ES2.0 Specification.* Retrieved February 12, 2012, from http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf

Khronos Group. (2012). *WebGL - OpenGL ES 2.0 for the Web*. Retrieved February 12, 2012, from WebGL - OpenGL ES 2.0 for the Web

KickJS. (2012). *KickJS*. Retrieved February 12, 2012, from http://www.kickjs.org/

Klokan Technologies. (2011). *WebGL Earth*. Retrieved February 12, 2012, from http://www.webglearth.org/about

Koordinaten. (2012). *Koordinaten*. Retrieved February 12, 2012, from http://www.koordinaten.de/informationen/formel.shtml

Lubbers, P., Albers, B., & Salim, F. (2010). The future of HTML5. In P. Lubbers, B. Albers, & F. Salim, *Pro HTML5 Programming* (pp. 259-267).

Milne, E. A. (1948). Vector Product. In E. A. Milne, *Vectorial Mechanics* (pp. 11-31). Methuen Publishing.

Moore, S., Townsend, R., & Campbell, J. (2011). *GammaJS*. Retrieved February 12, 2012, from http://gammajs.org/

Mozilla Developer Network. (2012). *Mozilla Developer Network*. Retrieved February 12, 2012, from https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/String/replace

Mrdoob. (2012). *Three.js*. Retrieved February 12, 2012, from https://github.com/mrdoob/three.js

Munshi, A., Ginsburg, D., & Shreiner, D. (2009). Lines. In A. Munshi, D. Ginsburg, & D. Shreiner, *OpenGL ES 2.0 Programming Guide* (p. 129). Addison-Wesley.

Munshi, A., Ginsburg, D., & Shreiner, D. (2009). Point Sprites. In A. Munshi, D. Ginsburg, & D. Shreiner, *OpenGL ES 2.0 Programming Guide* (p. 130). Addison-Wesley.

Munshi, A., Ginsburg, D., & Shreiner, D. (2009). Triangles. In A. Munshi, D. Ginsburg, & D. Shreiner, *OpenGL ES 2.0 Programming Guide* (p. 128). Addison-Wesley.

Munshi, A., Ginsburg, D., & Shreiner, D. (2009). Vertex Buffer Objects. In A. Munshi, D. Ginsburg, & D. Shreiner, *OpenGL ES 2.0 Programming Guide* (pp. 115-116). Addison-Wesley.

National Geographic Institute. (2012). *National Geographic Institute.* Retrieved Febuary 12, 2012, from Datum Grid: http://www.ign.es/ign/layoutIn/herramientas.do#DATUM

Nokia. (2012). *Nokia Maps 3D WebGL*. Retrieved February 12, 2012, from http://maps3d.svc.nokia.com/webgl

Oak3D. (2012). *Oak3D*. Retrieved February 12, 2012, from http://www.oak3d.com/

Open Geospatial Consortium. (2008). Multi-scale modelling (5 levels of detail, LOD). In G. Gröger, T. Kolbe, A. Czerwinski, & C. Nagel (Eds.), *OpenGIS® City Geography Markup Language (CityGML)* (1.0 ed., pp. 25-26).

Ortiz, S. (2010). Is 3D Finally Ready for the Web? *Computer , 43*, 14-16.

OSG. (2009). *OpenSceneGraph*. Retrieved February 12, 2012, from LOD class: http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00395.html

OSG. (2012, February 12). *OpenSceneGraph Website*. Retrieved February 12, 2012, from http://www.openscenegraph.org/projects/osg

OSGEarth. (2012). *OSGEarth*. Retrieved February 12, 2012, from http://osgearth.org/

OSGeo project. (2012). *Quantum GIS*. Retrieved February 12, 2012, from http://www.qgis.org/

OSGJS. (2012). *OpenSceneGraph JavaScript*. Retrieved February 12, 2012, from http://osgjs.org/

Pelican Mapping. (2011). *Github.ReadyMap SDK*. Retrieved February 12, 2012, from https://github.com/gwaldron/godzi-webgl/wiki/ReadyMap-SDK

Policarpo, F., & Fonseca, F. (2005). Deferred Shading Tutorial. *Pontifical Catholic University of Rio de Janeiro* , 1-27.

Rodriguez, J. (2010, Febrero 21). Energetic efficiency: business and ethics. *El País* , p. 27.

Sencha Labs. (2011). *PhiloGL*. Retrieved February 12, 2012, from http://www.senchalabs.org/philogl/

Shreiner, D. (2010). OpenGL Programming Guide. In D. Shreiner, *OpenGL Programming Guide* (p. 69). Addison - Wesley.

Shreiner, D. (2010). OpenGL Programming Guide. In D. Shreiner, *OpenGL Programming Guide* (pp. 1-29). Addison - Wesley.

Shreiner, D. (2010). OpenGL Programming Guide. In D. Shreiner, *OpenGL Programming Guide* (p. 50). Addison-Wesley.

Stoyan, S. (2010). In S. Stoyan, *JavaScript Patterns* (p. 3). Yahoo! Press.

Taivalsaari, A., & Mikkonen, T. (2011). The Web as an Application Platform: The Saga Continues. *37th EUROMICRO Conference on Software Engineering and Advanced Applications* .

Taivalsaari, A., Mikkonen, T., Anttonen, M., & Salminen, A. (2011). The Death of Binary Software:. *Ninth International Conference on Creating, Connecting and Collaborating through Computing* .

The Entertainment Software Association. (2011). *The Entertainment Software Association.* Retrieved February 12, 2012, from Essential Facts about the Computer and Video Game Industry: http://www.theesa.com/facts/pdfs/ESA_Essential_Facts_2010.PDF

The jQuery project. (2010). *jQuery*. Retrieved February 12, 2012, from http://api.jquery.com/category/ajax/

The jQuery project. (2010). *jQuery*. Retrieved February 12, 2012, from http://api.jquery.com/jQuery.ajax/

The jQuery project. (2010). *jQuery. parseJSON*. Retrieved February 12, 2012, from http://api.jquery.com/jQuery.parseJSON/

Treasury. (2012). *Electronic Cadastre Website*. Retrieved February 12, 2012, from http://www.sedecatastro.gob.es/OVCInicio.aspx

Treasury. (2011, March 03). *Treasury (Cadastre Website).* Retrieved February 12, 2012, from http://www.catastro.meh.es/ayuda/manual_descriptivo_shapefile.pdf

Verissimo, P., & Rodrigues, L. (2001). *Distributed Systems for System Architects.* Kluwer Academic Press.

Visual Computing Laboratory at CNR Research Area of Pisa. (2012). *SpiderGL*. Retrieved February 12, 2012, from http://spidergl.org/

W3C. (2011, May 25). *HTML5*. Retrieved February 12, 2012, from http://www.w3.org/TR/html5/

Wang, G. (2011). Improving Data Transmission in Web Applications via the Translation between XML and JSON. *Third International Conference on Communications and Mobile Computing*, (pp. 1-4).

Wang, R., & Quian, X. (2010). OpenSceneGraph. In R. Wang, & X. Quian, *OpenSceneGraph 3.0: Beginner's Guide* (p. 8). PACKT.

Wang, R., & Quian, x. (2010). Why OSG? In R. Wang, & x. Quian, *OpenSceneGraph 3.0* (pp. 12-13). PACKT Publishing.

Zalik, B., & Domiter, V. (2008). Sweep-line algorithm for constrained Delaunay triangulation. *International Journal of Geographical Information Science , 22* (4), 449-462.

Appendix A

In this appendix it is described the preliminary ran tests to obtain an optimized set of parameters. These parameters are further used on the cadastre's data preparation stage (see 4.2.4 *Data Preparation*) in order to obtain a filtered buildings' geometries.

This appendix is further divided in two main parts: *Introduction* and *Captures*. The first part describes the basic concepts in which the cadastre's data preparation stage relies on. The second section depicts a series of output images obtained from the application, which provide graphical and numerical results to the concepts introduced in the first part.

A1 Introduction

Mainly two parameters have been defined for the buildings' geometry filtering: *Area Trigger* and *Radius Threshold*.

Area Trigger has been defined as the numerical area value of the triangle that any three consecutive points have defined. It has been implemented due to the necessity of deleting collinear points, considering they made impossible to define valid triangles since they all lain on the same line.

On the following *Captures* section it has been depicted how different *Area Trigger* values have affected to ground floor's geometry up to a point where it has been impossible —for adapted Constrained Delaunay Triangulation library— to triangulate any the vertices that conformed such geometry.

Radius Threshold has been defined as the numerical value that describes the radius of a virtual circumference or neighbourhood, centred in each of the points that conforms each ground floor's geometry (de Smith, Goodchild, & Longley, 2009). Since measurements are relatively small, Earth's curvature does not have any notable influence on distances and it is correct to assume therefore, a 2D plain un-projected environment when processing neighbourhoods (Illiffe & Lott, 2008).

Subsequently, it has been implemented a simple Java application in order to obtain distance in meters —coming from geographical coordinates— relying on a spherical

Earth distance calculation's formula (Koordinaten, 2012). In this way, if any point was located within the virtual circumference area centred in a secondary point, it was therefore deleted. Consequently, the resulting ground floor's geometry would be lighter than the original one.

Area Trigger Code

Min	0.0000000282
Med	0.000000000282
Max	0.00000000000282

Radius Threshold Code	Distance in meters from a point centred in Valencia latitude: 39.470356 and longitude: -0.376781

R1	0.00001	(0.881 meters on terrain)
R5	0.000005	(0.425 meters on terrain)
R01	0.0000001	(nearly 0 meters on terrain)

In the following figures it is depicted firstly, the total amount of buildings that were present in the processed JSON file for each particular filtering configuration. Secondly, it is defined the number of buildings, with regard the total amount of buildings that were present in the JSON file, that have had any kind of triangulation problem and could not be represented.

A2 Captures

Detailed next, it has been depicted some captures, showing the results coming from the *Area Trigger* and *Radius Threshold* weighting, corresponding to *data preparation* stage.

For testing purposes, all the buildings have been represented with same height by default, in order the geometry not to be influenced by the viewer perspective.

A2.1 R1 Amax Test

Figure 49. R1 Amax test

Number of represented buildings: 4902 Error: 103

In the figure above, on the one hand it is depicted an accurate number of buildings —4902 out of 4906— with a very good definition at a cost of not filtering too many vertices, obtaining therefore a very high rate of redundancy. One the other hand, the triangulation algorithm is not able to handle such amount of unfiltered points, and that is the reason for the high number of Error buildings. It should be noted how the port of Valencia —on the right side—has not appeared comparing with following captures.

A2.2 R1 Amed Test

Figure 50. R1 Amed test

Number of represented buildings: 4897 Error: 41

In the figure above, it is depicted a better weighted solution considering the total number of represented buildings and the number of problematic ones. The filtering stage has run fine; consequently the vertex redundancy in building's outlines has been reduced. Additionally, the geometry did not seem to be extremely affected since buildings still preserved roughly the original shape. In general terms it is an overall solution to be considered, yet number of erratic buildings might be decreased, as it is described in following captures. It should be noted however, how the port of Valencia has appeared comparing to previous weighting result.

A2.3 R1 Amin Test

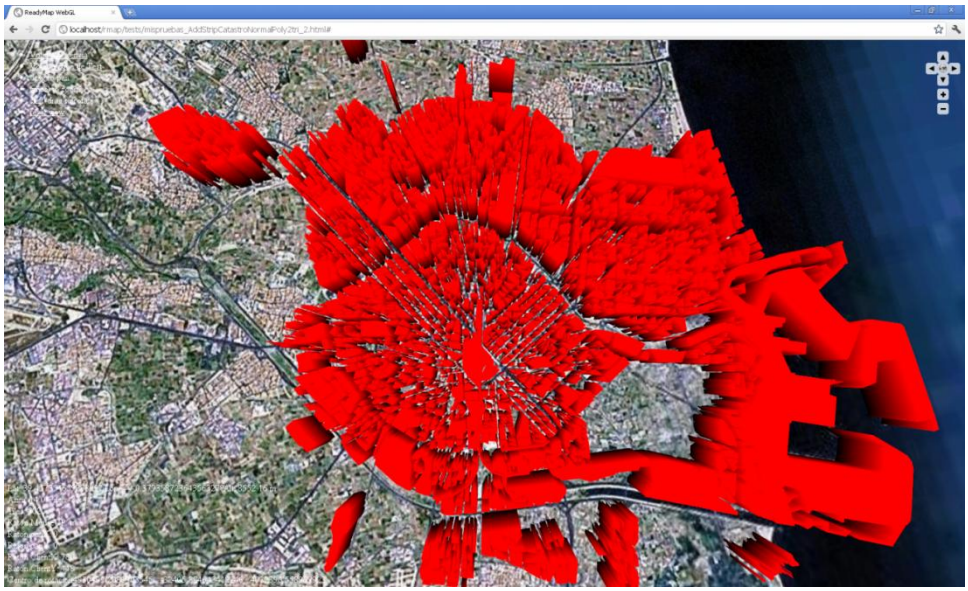


Figure 51. R1 Amin test

Number of represented buildings: 3259 Error: 6

In the figure above, it is depicted an extremely filtered geometry. Redundancy is so decreased and radius threshold is so big, that many buildings have entirely disappeared; resulting in a very low rate of 3529 represented buildings out of a total of 4906. As expected, the unrepresented buildings have decreased due firstly to a plausible decrease in the total number of buildings —since many of them have been completely filtered and deleted—. Secondly, unrepresented buildings also decreased due to both a decrease on the amount of vertices belonging to such buildings, and an increment of the distance between those vertices.

A2.4 R5 Amax Test

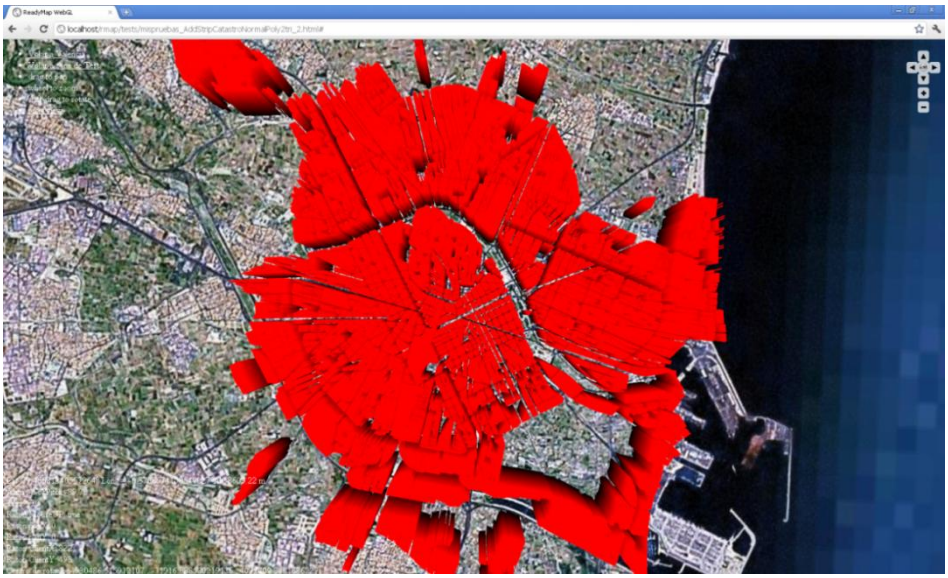


Figure 52. R5 Amax test

Number of represented buildings: 4906 Error: 100

In the figure above, it is depicted an accurate number of buildings with a very good definition, however the triangulation algorithm has still not been able to handle such amount of unfiltered points, and yet again that is the reason for the high number of inconsistent buildings. It should be noted that Valencia port is once more failing besides other smaller constructions.

A2.5 R5 Amed Test

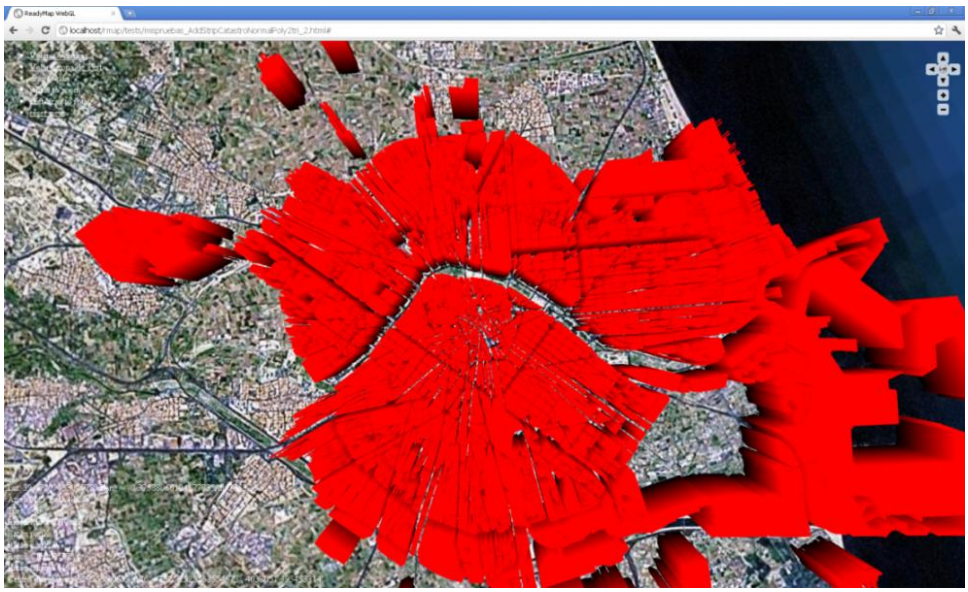


Figure 53. R5 Amed test

Number of represented buildings: 4897 Error: 31

In the figure above, it is depicted a correct weighted solution. It is presented a high number of represented buildings while error rate on them has been the smallest achieved. Geometry definition has been both accurate and optimized since most redundant points have been deleted, not affecting to the triangulation processing, and consequently representing all major and singular buildings in Valencia. This overall weighting has been considered as the reference one along this research's development.

A2.6 R5 Amin Test

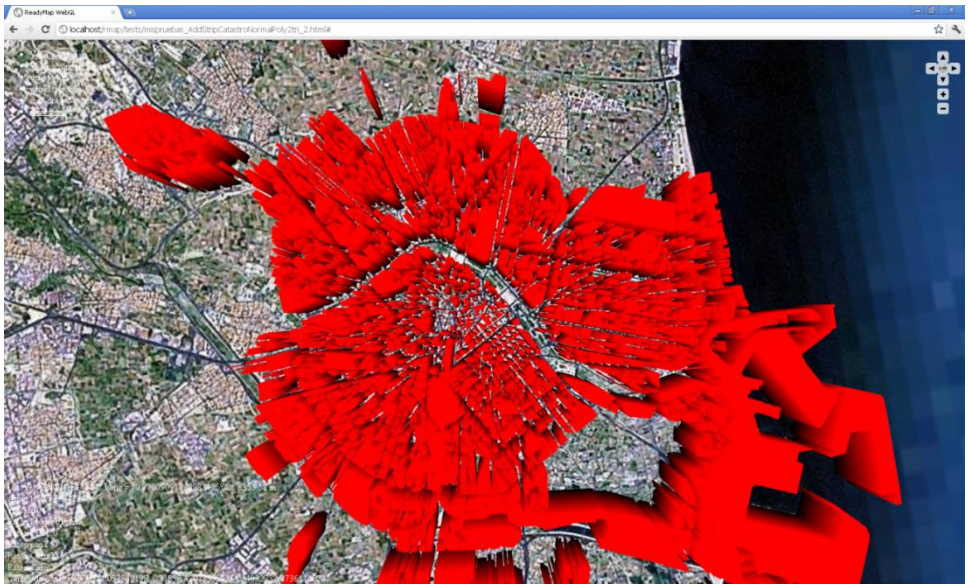


Figure 54. R5 Amin test

Number of represented buildings: 3260 Error: 7

In the figure above as expected, it is depicted a very poor geometry definition. Most buildings have become bitten and extremely distorted. Even if the error rate has been very low, the total amount of buildings has been extremely low as well. Considering data loss with regard this solution, it is concluded that it is not a suitable weighted solution.

A2.7 R01 Amax Test

Figure 55. R01 Amax test

Number of represented buildings: 4906 Error: 145

In the figure above, it is depicted an exact number of buildings in scene with a very good definition; however the triangulation algorithm has still not been able to handle such amount of unfiltered points. Curiously the number of problematic buildings has increased comparing to similar solutions, so that the perfect weighted solution might be the one that has been considered as reference, or some other between R1 and R01 cases.

A2.8 R01 Amed Test

Figure 56. R01 Amed test

Number of represented buildings: 4898 Error: 41

In the figure above, it is depicted a fine weighted solution similar to the reference one. It is represented a large number of total represented buildings whereas error rate has been relatively small, but higher than the reference. The number of problematic buildings has increased comparing to reference solution, so that the perfect weighted solution might be the one it has been considered as reference, or some other between R1 and R01 cases.

A2.9 R01 Amin Test

Figure 57. R01 Amin test

Number of represented buildings: 3260 Error: 6

Finally, in the figure above as expected, it is depicted a very poor geometry definition. Most buildings have been bitten again. Even if the error rate has been very low, the total amount of buildings has been extremely low as well. Apparently, errors in buildings have not decreased with regard to comparable solutions, so that it looks like *radius threshold* is set to its limits, not affecting to algorithm triangulation anymore. Considering data loss with regard this solution, it is concluded that it is not a suitable weighted solution.

A2.10 Summary table

	R1			R5			R01		
	Obtained	*Error*	*OK*	*Obtained*	*Error*	*OK*	*Obtained*	*Error*	*OK*
Amax	4902	103	4799	4906	100	4806	4906	145	4761
Amed	4897	41	4856	**4897**	**31**	**4866**	4898	41	4857
Amin	3259	6	3253	3260	7	3253	3260	6	3254
Total number of buildings in cadastre : 4906									

Table 8. Summary Table of implemented tests

Appendix B

B1 Introduction

Concurrently to cadastre data loading, it has been implemented a complete FIDE building's model support. FIDE is a building data exchange format widely used in Valencian community that is common to the distinct edification's areas agents, facilitating data exchange between them independently of the informatics applications they rely on.

FIDE format is based on the particularities, regulations and construction systems corresponding to the Spanish authorities; nevertheless, it inherits a major part or its format's structure from the international IFC building data exchange format, aiming therefore to facilitate formats interaction and systems interoperability.

FIDE format models are written in XML, depicting and extensive group of tags which altogether fully defines both building's topology and semantic.

Concerning FIDE *Appendix B* description, this work tries not to repeat common processing between cadastre data and FIDE data solutions, strengthening specially the adopted differences in developing approaches, in addition to application's analysis, processing and behaviour. Thus, this *Appendix B* is divided in three main sections: *Pre-processing*, *Processing* and *Results*.

In *Pre-processing* section it is described a series of Java-based data processing to further analyze, prepare and save the data, in order to be subsequently loaded by the Web application. In *Processing* section it is described in-depth the related data processing for an ultimate data representation. Finally, in *Results* section it is described the obtained results.

B2 Pre-processing (Data Preparation)

In this section it is depicted the processes and sub-processes related with data preparation, embracing the necessary steps to be done before the Web application has loaded the building's output file so as to be processed and represented. Additionally, it is described in detail the implemented Java-based application for FIDE data

management, further divided in three main parts: *Data Preparation*, *Application Structure* and *Application Behaviour*.

B2.1 Data Preparation (Java)

Explaining in-depth FIDE data exchange format is completely out of bounds due to both complexity and extension; nonetheless a set of basic concepts and explanations have been introduced along next sections in order for the reader to better understand the implemented pre-processing done in this research. For complete information, do refer to the official Web documentation (FIDE Organization, 2009).

Firstly, aiming to provide a complete FIDE support, it has been necessary to obtain a valid FIDE dataset, which have been kindly supplied by the Valencian Edification Institute.

Following, it is presented a virtual representation of the sample data (see *Figure58*).

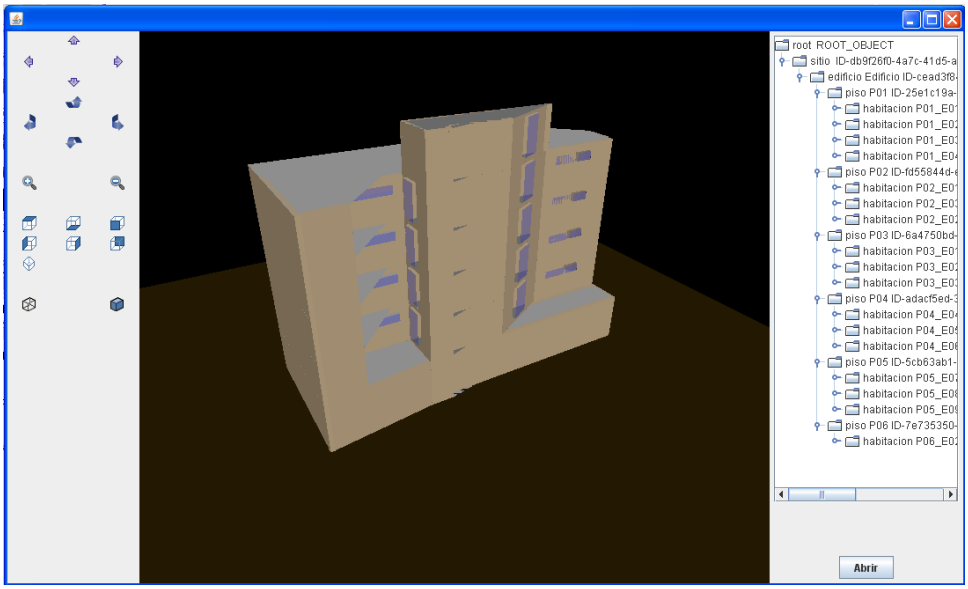


Figure 58. FIDE model representation

Contrary to cadastre's data case, FIDE data preparation methodology has been entirely implemented in Java (Belmonte, Carlos, & Maria), mainly due to a provided set of libraries which have helped to extract and gather information from the FIDE models.

Thus, the main objectives of *pre-processing* stage have been loading, parsing, processing and gathering the necessary information so as to create a valid JSON with the corresponding building's geometry to be finally loaded by the Web application.

B2.2 Application Structure

Next detailed, it is presented the structure of the Java-based FIDE processing program.

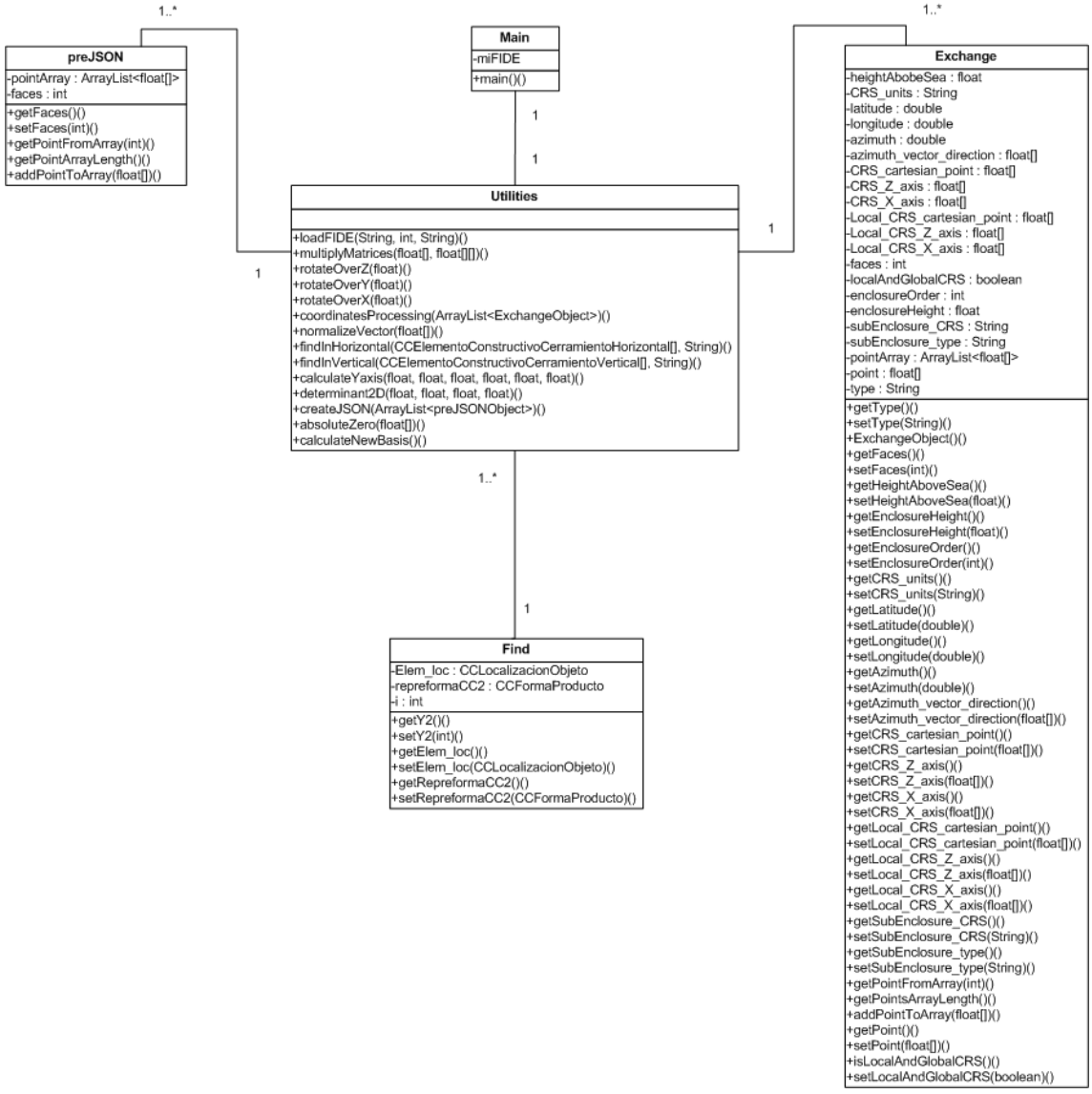


Figure 59. Java application's structure

As seen, it has been implemented five classes that provides solution to FIDE-JSON conversion, described as following:

- *Main* class: It has been used as grounding, being the executable class for the whole data preparation processing, being therefore the linking connection with the rest of classes. Within its main method, it has been done an instance of *Utilities* class, which has been in charge of the major part of the data processing.

- *Utilities* class: It has essentially consisted of a methods' aggregation, which have mainly defined the totality of processing considered in this part of the research. It has provided a set of methods and operations which has given complete solution to the project's methodology issues. *Utilities* class has been in charge of loading each FIDE model, parsing its content, extracting the useful data, storing data on secondary exchange objects and finally providing a set of mathematical methods for both graphical data management and coordinate systems transformations.
- *Find* class: It has given full support to *Utilities* class on finding particular set of elements along each FIDE data file.

Additionally, two exchange data classes —objects— have been created in order to store and transfer data along processing.

- *Exchange* class: It is a complex and detailed exchange data class which has served as a preliminary data storing layer, for extracted FIDE data model's attributes. Consequently, each *Exchange* object has carried the information of each particular geometric entity —such as ellipse, circle, polygon or box i.e.— corresponding to an unprocessed building's geometry. It should be noted therefore that a geometric entity may not be a single building's face but many —as in case of box entity, defining six faces—.
- *preJSON* class: It is a light-exchange data class, which has provided the minimum and essential processed data to *createJSON()* method (see *Figure59*). Additionally each *preJSON* object has carried not only the geometric vertices data, but also the information about the number of faces describing such data, in order to finally obtain a single JSON output depicting the corresponding geometry per face.

B2.3 Application Behaviour

Following detailed, it is presented the Java applications behaviour (see *Figure 60*).

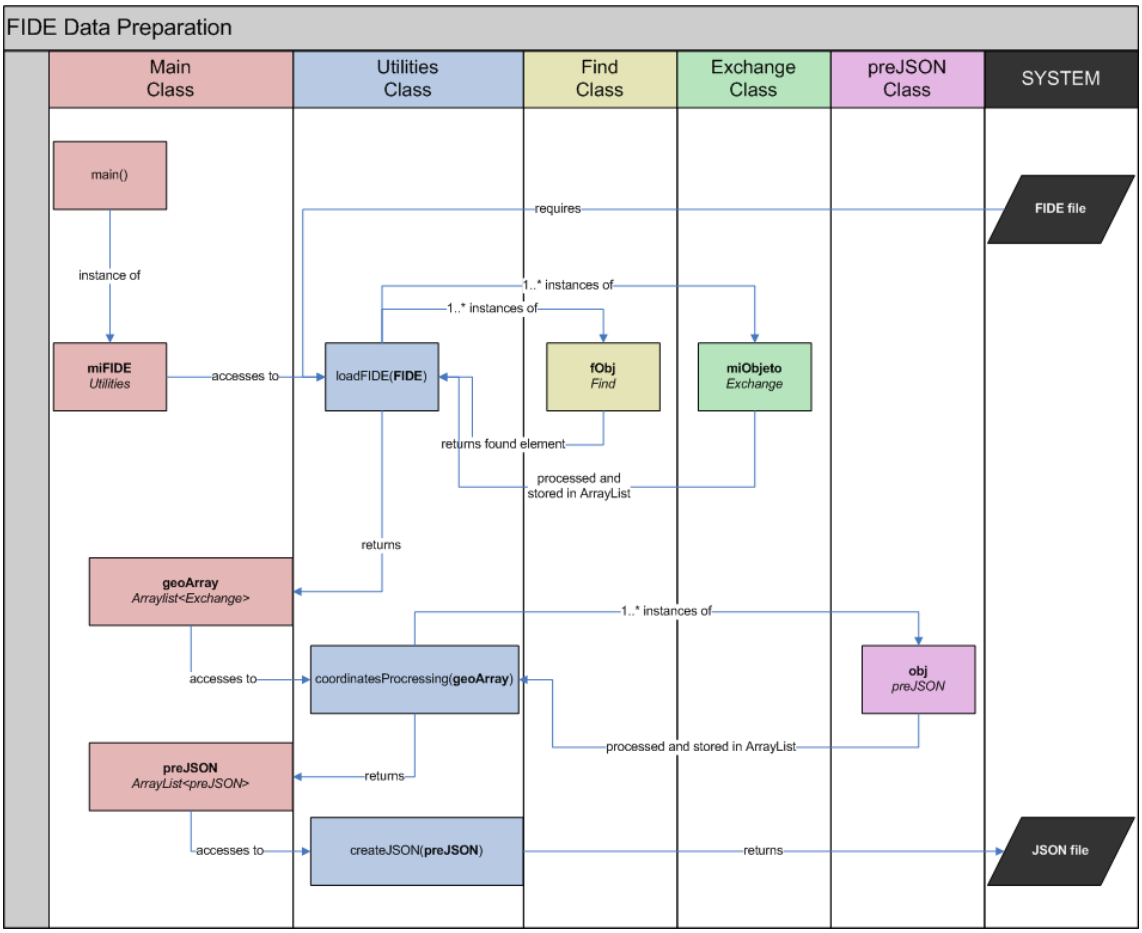


Figure 60. Java application's behaviour

Next, it is described the process flow diagram:

Firstly, the application accesses *main()* method in *Main* class, which immediately creates an instance of *Utilities* class.

At this point, the main target is to call to *loadFIDE()*, in order to extract all useful FIDE model's attributes, and save them into an array of *Exchange* objects. In this way it is obtained therefore as many objects as extracted geometric entities. Additionally, *loadFIDE()* takes advantage of *Find* class which provides a specific functionality on searching defined parameters along a collection of vertical and horizontal entities.

As observed, *loadFIDE()* returns an array of *Exchange* objects, which defines in turn the input for *coordinatesProcessing()*. Despite useful information is extracted and stored, it is required that *coordinatesProcessing()* accurately geo-locate each point on the ReadyMap globe's surface.

Therefore —and secondly— *coordinatesProcessing()* stores the newly processed geometrical information into an array of *preJSON* objects. Detailed next, it is presented a description of principal *coordinatesProcessing()* sub-processes.

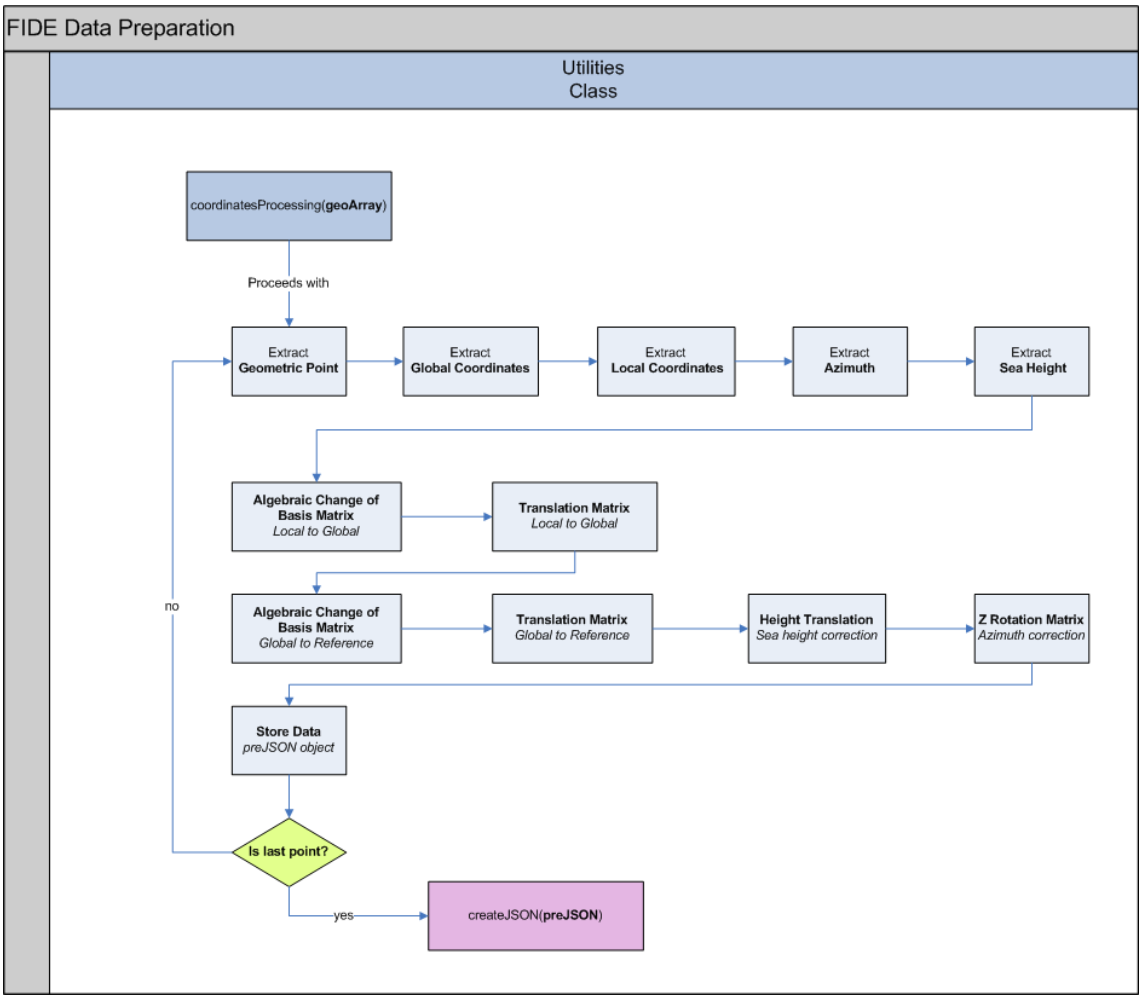


Figure 61. Java application's behaviour detail

As seen, this sub-processing involves firstly, an iterative extraction of each geometric point present in each geometric entity, its local coordinate reference system, the global coordinate reference system, a global azimuth and the corresponding height above sea level.

Once data extraction processing finishes, a set of coordinates' transformations begin. At this point it is necessary to mention that, having no purpose to go in-depth, a local coordinate reference system may depend on either a global or a local coordinate reference system (Garrido, 2011). Therefore it turns out necessary to adapt vertices' coordinates in order to define all the points of any local system reference with regard to

not the global, but to the reference system — it should be noted that a global reference system may be translated and rotated as well—.

The general geometric structure of a FIDE file concerns mainly three elements; place, building and enclosure. An enclosure may describe more enclosures within its geometry (see *Figure 62*) which in turn each of them may describe additional enclosures etcetera; that is why a local reference system may depend on another one as mentioned before.

Following the same example, a local reference system would define the reference context for an enclosure's geometry; a global reference system would define the reference context for a whole building's geometry; and finally the reference system would define the reference for the whole place, which can host alternatively one or more buildings' geometry.

Therefore, it turns out necessary to firstly transform any local coordinate into a new one with regard to the reference system. Consequently, it is firstly necessary to apply a sequence of change of basis and translations, from local to a global reference system, and straight after following the same procedure, from a global to a reference system.

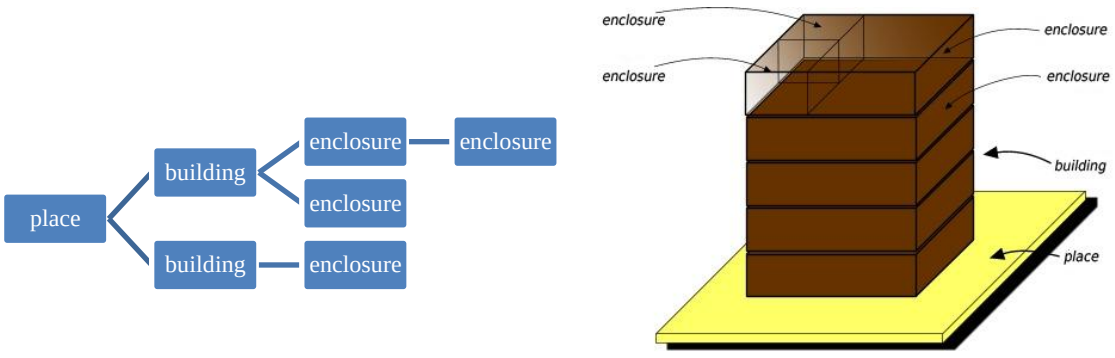


Figure 62. FIDE geometrical structure

Secondly, it is necessary to define a set of sequential corrections. On the one hand it is required to correct each point's height by summing the extracted height above sea level. This is especially important in cases such as this one, where the data are going to be finally represented on top of a Digital Elevation Model (DEM) surface. On the other hand it is necessary to adapt building's orientation by correcting Azimuth angle with regard to North direction. Additionally, it has been implemented a weighting in order to magnify building's size, applied not only as vertical exaggeration but also horizontal.

Finally, the information is stored in an array of *preJSON* objects, which is consequently sent to *createJSON()* in order to saved as a valid JSON file.

B3 Processing (Data Visualization)

In this section it is depicted the Web application's processes and sub-processes relying on the prepared data (see *B2 Pre-processing* section), in order to be finally represented on a WebGL standard-compliant Web browser. This section is additionally divided into two parts: *File Organization* and *Data Visualization* —the core data processing part—.

B3.1 File Organization

FIDE representation functionality relies on a similar file structure as buildings cadastre solution. Four main kinds of files were either developed or used in order to provide as whole, the essential graphical application's results. a) *FIDEbuildings.html*, b) *jQuery.js*, c) *ReadyMap.js* and d) *FIDE_x.json*.

a. *FIDEbuildings.html*: It is an HTML file which loads, gathers and executes tasks relying on the following JavaScript and JSON files. It is the main grounding for a final graphical representation, especially based on canvas HTML5 element presence.
b. *jQuery.js*: It is a JavaScript Library file that simplifies HTML document traversing, event handling, animating, and AJAX interactions for web development.
c. *ReadyMap.js*: It is a JavaScript file which gathers and stores all implemented functionalities relative to OSG.js, OSGEarth.js and the ones particularly implemented for this research.
d. *FIDE_x.json*: It describes the geometry and semantic of a particular FIDE model.

Alternatively it has been slightly modified two CSS files which help to distribute and graphically embellish HTML elements visualization and consequently final building's representation.

Additionally it has been included *cadastreBuildingsAndFIDE.html* file in which it is presented a merged representation of both cadastre and FIDE research's development.

B3.2 Data Visualization (ReadyMap)

In this section it is described the core processing related to FIDE building's rendering process. It is further divided in four main parts: *Preliminary Concepts*, *Drawable Geometry*, *Reference Geometry* and finally *Drawable and Referenced Insertion*.

Contrary to cadastre data processing, FIDE processing has not extruded any geometry since, as mentioned in this *Appendix A*, *Data Preparation* stage has described the building's geometry by a sequence of definitions per-face. Thus, comparing with cadastre data approach, FIDE prepared data have not needed any extra treatment before transferring to *FIDEbuildingNode*.

B3.2.1 Preliminary Concepts

FIDEBuildingNode has been defined as a JavaScript object, additionally described as the complete set of attributes and methods implemented in this research, which have given support to any ReadyMap-based FIDE building's representation.

As soon as an instance of a *FIDEBuildingNode* is made, it is defined and fulfilled the next two attributes: *origin* and *heightField:*

- *origin* vertex has been created with the first tuple of coordinates —latitude and longitude— defined in the first vertex present on the first face. This has been used in turn as a geometric anchor point for the rest of faces associated to this particular *FIDEBuildingNode* .
- *heightField* has been defined to store the heights of the vertices associated to each face:

var *heightField* = [face0_height0, face0_height1, face0_height2,…
face1_height0, face1_height1, face1_height2,…
face2_height0, face2_height1, face2_height2,…]

Finally, concerning the volume exaggeration variable has been set to one, contrary to cadastre approach. There has not been any on-globe faces magnifying; nonetheless such weighting has taken place in data pre-processing as mentioned (see *B2.3 Application Behaviour*).

B3.2.2 Drawable Geometry

The geometry for each particular *FIDEBuildingNode* has been described as a set of *primitives* and *attributes*, stored in a group of arrays.

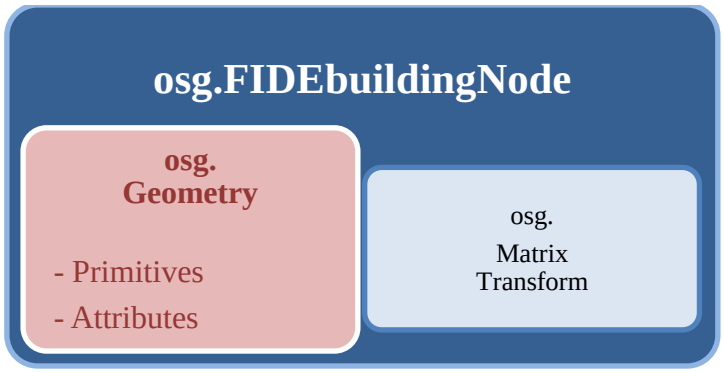


Figure 63. osg.Geometry object

Primitives have been defined as three one-dimensional element arrays where the indices of vertex's geometry have been stored in:

- *faces*: has stored the vertices indices of each face
- *lines*: has stored the vertices indices connections for each face.
- *points:* has stored the vertices indices.

Primitive values have been stored into each *FIDEBuildingNode* subsequently binding them to an *osg.DrawElements* object. *osg.DrawElements* has managed the render of the indexed primitives by firstly defining the WebGL rendering mode —GL_TRIANGLES, GL_LINES or GL_POINTS i.e.— (Khronos Group, 2010); and secondly by assigning the corresponding set of *primitives*.

In order to illustrate how *osg.DrawElements* has been called, a code sample is presented below.

```
Var tris = new osg.DrawElements(gl.TRIANGLES, new
osg.BufferArray(gl.ELEMENT_ARRAY_BUFFER, faces, 1));

this.geometry.getPrimitives().push(tris);
```

Figure 64. osg.DrawElements geometry insertion

WebGL GL_TRIANGLES rendering mode has been used to represent *faces* array data. WebGL GL_LINES rendering mode has been used to represent *lines* array data. And finally, WebGL GL_POINTS rendering mode has been used to represent *points* array data.

On the other hand, three main variable-dimensional arrays related to *Attributes* have been defined:

- *verts* (3dim): has stored vertices coordinates attribute.
- *normals* (3dim): has stored vertices normal attribute.
- *colours* (4dim): has stored vertices colour attribute.

Attribute values have been stored into each face vertex in a similar way as before (see *Figure 65*), excepting the type of buffer that has been used. In the following sections *Primitive* Arrays and *Attribute* Arrays are described.

```
this.geometry.getAttributes().Vertex = new osg.BufferArray(gl.ARRAY_BUFFER,
verts, 3);
```

Figure 65. osg.BufferArray binding

In order to get the best performance it has been used ARRAY_BUFFER to store the vertex data, and ELEMENT_ARRAY_BUFFER to store the indices of the primitives to be rendered (Munshi, Ginsburg, & Shreiner, Vertex Buffer Objects, 2009).

In the following sections, *Primitive* and *Attribute* arrays processing are described in-depth.

B3.2.2.1 Primitive Arrays

In this section it is described how *primitive* arrays are filled. *Primitive* arrays are filled at once, unlike *attribute* arrays which are filled iteratively per building's vertex.

Polygons rendering mode (GL_TRIANGLES)

In this section it described how *faces* array is filled, corresponding to polygons rendering mode, in which each vertices' indices are stored.

Building faces wrapping

Building faces wrapping has been accomplished adapting a JavaScript-based Constrained Delaunay Triangulation library (CDT) similarly to cadastre approach. Seeing that, face vertices have been triangulated to create a sequence of triangles which have ended up covering the entire face surface.

Unlike cadastre approach where only roofs' geometries were triangulated, FIDE processing has triangulated the complete sequence of faces that each FIDE building has been formed by. Thus, each face has been treated as a separated geometry resulting finally in an aggregation of faces that has defined either a building or a block of buildings.

As a result of managing an aggregation of faces, it has resulted necessary to determine the orientation of each of them. This has been crucial due to CDT triangulate the vertices corresponding to each of the faces on a two dimensional plane. According to that the orientation of each face may result in an inconsistent projected shape. For instance, if there is a vertical face whose vertices are described as (X,Y,Z), and it is further projected either on the XY or YZ plane, the result would be a single line (see *Figure 66*). Therefore, it has been implemented the following processing in order to solve such issue.

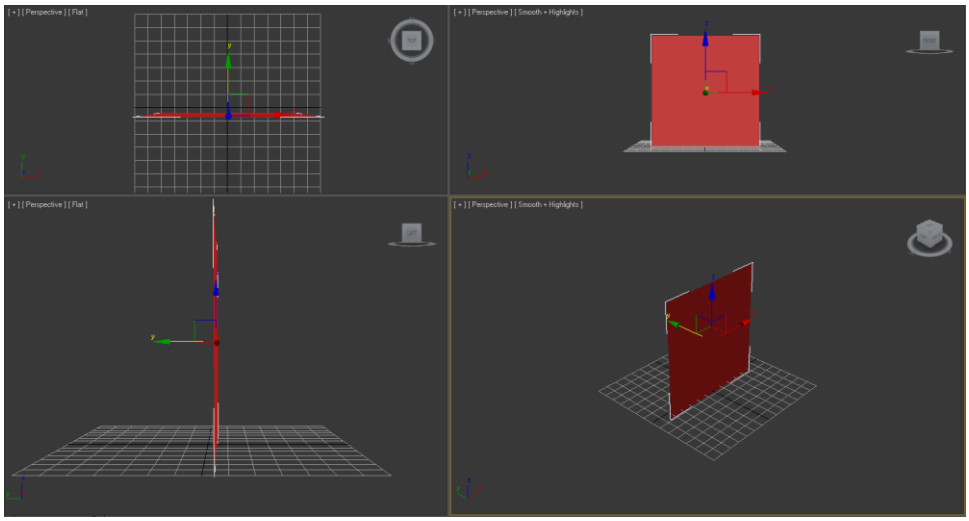


Figure 66. Vertical plane projections

At this point, relying on the original coordinates; it has been calculated firstly the two directional unit vectors corresponding to the face to be treated. Those directional unit vectors have been defined by picking the first three non collinear vertices in a row that each building's face defines.

Secondly, the *normal* of the plane defined by the two calculated unit vectors has been computed by doing the cross product of both vectors (Milne, 1948). This *normal* vector has been therefore described as (x,y,z) vector. Thus, depending on which direction it was pointing, it has been applied the consequent change of coordinates.

This methodology has been implemented by switching the coordinates to the projected plane that better described the face geometry (for instance in *Figure 66*, the upper right plane).

Thirdly, the processed coordinates have been triangulated. The resulting output has been defined as an array of triangles that formed the treated building's face. Each of resulting triangles has been described in turn as an array of three vertices.

Fourthly, and identically as cadastre data's approach, it has been necessary to sequentially extract the corresponding index associated to each triangulated coordinate's vertex in order to push it into *faces* array. It should be noted that any switch of coordinates previously applied must be considered at this point in order to find the right vertex coordinate's element.

Finally, each of the triples of vertices indices have been then bound to GL_TRIANGLES rendering mode. In this way it has been possible to represent no matter what face shape in an optimized way.

Lines rendering mode (GL_LINE_STRIP)

In this section it is described how *lines* array has been fulfilled, corresponding to line rendering mode, in which each vertices indices connections are stored.

Building vertices connection

FIDE model line representation has been accomplished storing the faces indices into *lines* array taking into consideration a special care on data endings due to GL_LINE_STRIP mode definition (Shreiner, OpenGL Programming Guide, 2010).

Points Sprites rendering mode (GL_POINTS)

In this section it is described how *points* array is fulfilled, corresponding to point sprites rendering mode, in which face vertex indices are stored.

Simple vertices representation

The faces vertices representation has been accomplished sweeping the total amount of faces vertices, storing the corresponding index into *points* array and finally setting WebGL rendering mode to GL_POINTS.

B3.2.2.2 Attribute Arrays

In this section it is depicted how *colours* array is filled for each face. Considering that *normals* and *verts* array processing has not changed comparing to cadastre buildings processing, it is not described in this *Appendix A*.

Face Colouring

As explained in the project's context of this research (see *1.1 Introduction*), the background behind this project has been the processing of precise buildings' energetic demand, relying on 3D buildings' geo-locations. The proposed technical approach to such issue has been defined as a Web service which would receive both FIDE and cadastre's geometry in JSON format as input, subsequently obtaining an output defined as an array of as many elements as FIDE building's faces. At each array position it

would have been stored the incident solar radiation for each face present in FIDE model. Additionally, those values would be limited to a zero-to-one scale.

Since the Web service is out of the bounds of this appendix and research, it has been implemented the incident solar radiation's face representation relying on a predefined array. The length of this array has been set to the number of faces present in the FIDE model.

Additionally, each of its elements has been defined with a random value constrained to same zero-to-one scale. In this way, each output value has been transformed into red-green-blue colour coordinates.

It has been implemented three different colour themes' representations:

- Height colour theme (see *Figure 67*).
- Solar radiation greyscale colour theme (see *Figure 68*).
- Solar radiation varied colour theme (see *Figure 69*).

Figure 67. Height colour theme

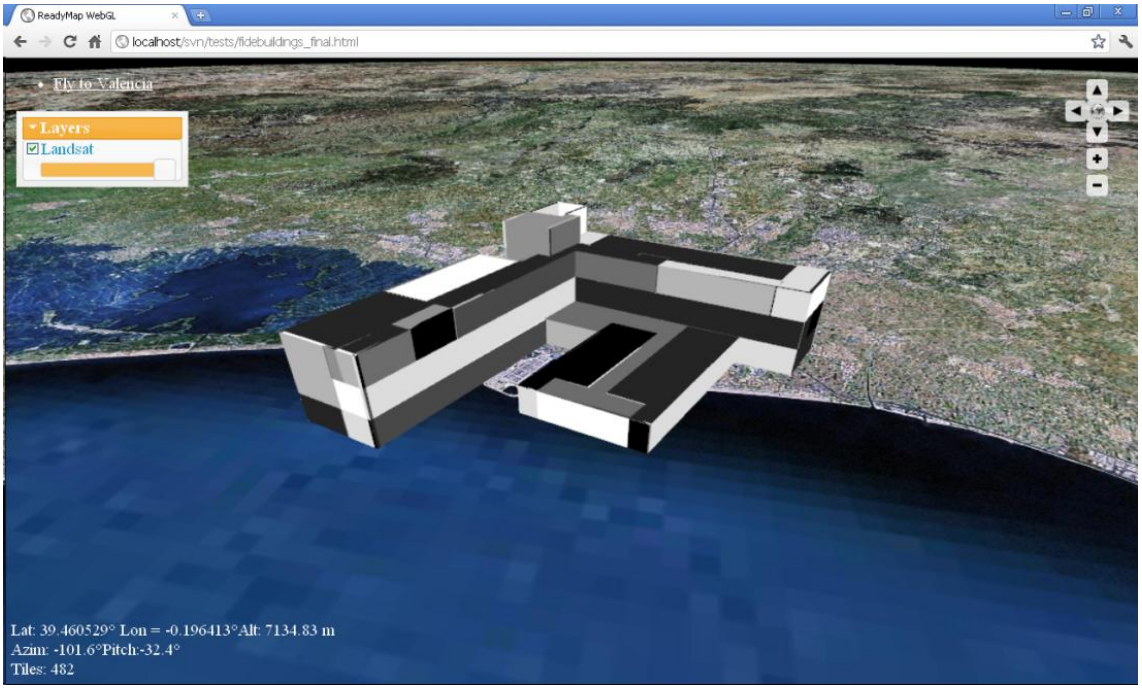


Figure 68. Solar incidence greyscale colour theme

Figure 69. Solar incidence varied colour theme

B3.2.3 Referenced Geometry

Considering that *Matrix Transformation* processing has not been changed comparing to cadastre buildings processing, it is not described in this appendix.

B3.2.4 Drawable and Referenced Geometry Insertion

In a final step, it has been attached to the same child:

- The *Drawable Geometry*, which have defined the topology of the faces that by aggregation have defined consequently a building.
- The *Referenced Geometry*, which have accurately placed such topologies on top of the globe's surface.

B4 Results

FIDE buildings implementation has defined a new generalized way to treat and finally represent geo-data on top of a WebGL-based globe's surface. Comparing to cadastre buildings' approach, FIDE approach has been based on a per-face data processing taking advantage of the innovations implemented on this research. By adapting a CDT library, it has been provided an innovative way of data representation for WebGL-based applications; additionally based on a client-side data processing contrary to conventional server-side-based approaches.

Indirectly, FIDE buildings implementation gives support to data representation coming from present and forthcoming Web-based geo-processing. Such implementation provides support for a new range of emerging Web-based market solutions.

2012

DESIGN AND IMPLEMENTATION OF 3D BUILDINGS INTEGRATION FOR A WEBGL-BASED VIRTUAL GLOBE

A Case Study of Valencian Cadastre and FIDE Building Model

Daniel Gastón Iglesias

Masters Program in **Geospatial Technologies**

Supported by:
Education and Culture
ERASMUS MUNDUS