Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Design and development of a framework based on OGC web services for the visualization of three dimensional large-scale geospatial data

Laureando:
Antonio Nozzi

Relatore:
Ch.mo Prof. Massimo Rumor

Padova, 23 Ottobre 2012
Anno Accademico 2011-2012

Abstract

Most of the current 2D GIS representations are tied to the limits of traditional GIS software and are, more properly, simplified abstractions of the real aspects of the territory. 3D GIS tools try to describe reality, and the phenomena that take place in it, in the proper dimensions, making it possible to solve spatial problems not addressable in 2D. 3D representations are also more intuitive and can be used to communicate territorial information to non-experts in a immediate and realistic way.

3D GIS visualization models have a variety of applications in geography, urban studies and other scopes. Such models often require to render significant amounts of three dimensional geospatial data, so being very demanding in terms of computing capability and memory usage. In order to efficiently manage large scale data rendering and reach a reasonable compromise between quality and performances, some optimizations are needed.

The aim of this project is to design a streaming framework for the visualization of three dimensional large-scale geospatial data. A simple idea is implemented: just the bare necessities have to be loaded and rendered. The 3D scene is so incrementally built and dynamically updated run-time, taking into account the movements of the camera and its field of view. To effectively and efficiently achieve this behavior, proper mechanisms of tiling and caching have been implemented.

The framework implementation focuses on textured terrain streaming. Despite the scope limitation, the defined streaming paradigm has general validity and can be applied to more complex 3D environments. The addition of other features on top of the terrain is straightforward and does not imply substantial modifications to the framework.

In order to make the framework standard compliant and platform independent, it has been designed to work with OGC web services and the widely adopted web-based approach has been chosen. As a result, any WebGL compliant browser can run web applications built on top of this framework without the use of plug-ins or additional software.

Contents

List of Figures

List of Tables

Acronyms

AFL	**A**cademic **F**ree **L**icense
AJAX	**A**synchronous **J**avaScript **A**nd **X**ML
AMD	**A**synchronous **M**odule **D**efinition
API	**A**pplication **P**rogramming **I**nterface
BSD	**B**erkeley **S**oftware **D**istribution
C3DL	**C**anvas **3D** **L**ibrary
CAD	**C**omputer-**A**ided **D**esign/**D**rafting
CNR	Italian **N**ational **C**ouncil of **R**esearch
COLLADA	**COLLA**borative **D**esign **A**ctivity
CPU	**C**entral **P**rocessing **U**nit
CSS	**C**ascading **S**yle **S**heets
DEM	**D**igital **E**levation **M**odel
DOM	**D**ocument **O**bject **M**odel
DSM	**D**igital **S**urface **M**odel
DTM	**D**igital **T**errain **M**odel
EPL	**E**clipse **P**ublic **L**icense
FMC	**F**undamental **M**odeling **C**oncepts
FOV	**F**ield **O**f **V**iew
fps	**f**rames **p**er **s**econd
GIF	**G**raphics **I**nterchange **F**ormat
GIS	**G**eographic **I**nformation **S**ystem
GLSL	Open**GL** **S**hading **L**anguage
GML	**G**eography **M**arkup **L**anguage
GNU	recursive acronym, **G**NU is **N**ot **U**nix
GPL	GNU **G**eneral **P**ublic **L**icense

GPU	**G**raphical **P**rocessing **U**nit
GeoDBMS	**Geo**graphic **D**ata**B**ase **M**anagement **S**ystem
GeoTIFF	**Geo**graphic **T**agged **I**mage **F**ile **F**ormat
HDF	**H**ierarchical **D**ata **F**ormat
HTML	**H**yper**T**ext **M**arkup **L**anguage
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
ISTI	**I**nstitute of **I**nformation **S**cience and **T**echnology
JPEG	**J**oint **P**hotographic **E**xperts **G**roup
JSON	**J**ava**S**cript **O**bject **N**otation
KML	**K**eyhole **M**arkup **L**anguage
LOD	**L**evel **O**f **D**etail
MIME	**M**ultipurpose **I**nternet **M**ail **E**xtensions
MIT	**M**assachussets **I**nstitute of **T**echnology
NetCDF	**Net**work **C**ommon **D**ata **F**orm
NITF	**N**ews **I**ndustry **T**ext **F**ormat
OBJ	**OBJ**ect
OGC	**O**pen **G**eospatial **C**onsortium
OpenGL ES	OpenGL for **E**mbedded **S**ystems
OpenGL	Open **G**raphics **L**ibrary
OS	**O**perative **S**ystem
PNG	**P**ortable **N**etwork **G**raphics
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
SRS	**S**oftware **R**equirements **S**pecification
SVG	**S**calable **V**ector **G**raphics
UML	**U**nified **M**odeling **L**anguage
W3C	**W**orld **W**ide **W**eb **C**onsortium
WCS	**W**eb **C**overage **S**ervice
WFS	**W**eb **F**eature **S**ervice
WHATWG	**W**eb **H**ypertext **A**pplication **T**echnology **W**orking **G**roup
WMS	**W**eb **M**ap **S**ervice
XHTML	E**x**tensible **H**yper**T**ext **M**arkup **L**anguage
XML	E**x**tensible **M**arkup **L**anguage

Chapter 1

Introduction to 3D GIS

1.1 Geographic Information System

A Geographic Information System (GIS) is an organized collection of hardware, software and data designed to efficiently store, manage, share, analyze, manipulate, and display geographical information for informing decision making. In a more practical sense, GIS applications are tools that allow users to create interactive queries, analyze the spatial information, edit data in maps, and present the result of all these operations.

Spatial features are stored in a coordinate system (latitude/longitude, state plane, universal transverse mercator, etc), which references a particular place on earth. Descriptive attributes in a tabular form are associated with spatial features. Spatial data and associated attributes in the same coordinate system can then be layered together for mapping and analysis. GIS can be used for scientific investigations, resource management and development planning.

GIS differs from Computer-Aided Design/Drafting (CAD) and other graphical computer applications in that all spatial data are geographically referenced to a map projection in a earth coordinate system. For the most part, spatial data can be re-projected from one coordinate system into another, thus data from various sources can be brought together into a common database and integrated using GIS software.

Another property of a GIS database is that it has a topology, which defines the spatial relationships between features. The fundamental components of spatial data in a GIS are points, lines (arcs) and polygons. When topological relationships exist between features, it is possible to perform analysis on spatial data, such as modeling the flow through connecting lines in a network, combining adjacent polygons that have similar characteristics and overlaying geographic features.

1.2 Adding the third dimension: 3D GIS

Most of currently available GISes provide tools for managing two dimensional data. For an increasing number of applications 2D representation is not sufficient. This especially holds for geology, for architectural design or for large-scale urban planning where the complexity of the objects results from their rich 3D spatial structure. For example, city models may contain buildings, roads including bridges or tunnels, subways, sewer, gas pipe networks or just the vegetation. All this information can be stored in 2D layers which might be sufficient for some applications, however this representation appears to be inadequate in the long run.

3D GIS visualization models have a variety of applications in geography, urban studies and other scopes: site location analysis, emergency facilities planning, design review, geography education, tourism, marketing, etc. While they are generally used to simply visualize the built environment, there are early signs of them being used as 3D interfaces to more sophisticated simulation models.

3D GIS tools often have to render significant amounts of three dimensional geospatial data. A hypothetical complex 3D urban model, which may include features such as buildings, transportation objects, water bodies, vegetation objects and city furniture, could require to render a huge number of 3D objects, maybe quite detailed and textured. Doubtless, this task could be quite hard for graphics hardware and very expensive in terms of computing capability and memory usage. Wishing to make 3D GIS software accessible to a wide range of hardware, not just high-end hardware, some optimizations and tricks have to be considered while developing applications.

Chapter 2

Project requirements analysis

2.1 Problem definition

The aim of this project is to design a lightweight streaming framework able to efficiently visualize three dimensional large-scale geospatial data. When dealing with a complex 3D environment, an incremental approach may be used to lighten the rendering process. In practice, the framework should be able to dynamically build and update the 3D scene as needed, for example taking into account the movements of the camera and its field of view. This simple idea permits to load and render just a limited amount of data at a time, thus significantly reducing the GPU load and improving performances.

In order to test the planned strategy in a specific context, the framework implementation focuses on textured terrain streaming. Despite the scope limitation, the defined streaming paradigm has general validity and can be applied to more complex 3D environments. The terrain is just a base, above which other spatial features (buildings, vegetation, city furniture, etc) can be placed. The addition is straightforward and does not implies substantial modifications to the fundamental principles of the framework.

2.2 Basic project requirements

Current GIS requirements are moving towards a comprehensive set of multi-platform tools. WebGIS is currently one of the most deployed approach, thanks to its intrinsic multi-platform support and the absence of dedicated software installed client-side (just a modern web browser is usually needed).

Standards compliance is also an important requirement. The use of well defined standards simplifies the development and increments interoperability and software quality. The Open Geospatial Consortium (OGC) maintains a variety of open geospatial standards. Many geospatial servers (GeoServer, MapServer, etc) and GeoDBMSes (PostGIS, Oracle Spatial, etc) well support these standards and provide a solid base on top of which scalable and reliable WebGIS applications can be developed.

The computing power needed for 3D GIS visualization and processing is more and more onerous than in the 2D case, so most of the products currently available work installing a software client that directly access the graphical card using OpenGL or DirectX standard libraries. Reliable multi-platform software is yet to come and commercial software is often only available on Microsoft Windows.

Basing the analysis on these simple observations, the project should try to resolve the defined problem in accordance with the following requirements:

- The developed framework has to provide multi-platform support through the employment of a web-based approach.
- A full support for 3D hardware acceleration must be provided.
- OGC standards compliance must be ensured.

Chapter 3

Development platform

3.1 Overview

The requirements defined in Chapter 2 are essential to choose the right development platform to accomplish the project.

As previously established, the developed framework should serve as a basis for multi-platform web-based applications. With the introduction of HTML5 (cf. Section 3.2) and derived technologies, like WebGL, web applications are dramatically incrementing their potential, so operations before unimaginable are now possible. WebGL (cf. Section 3.3) allows 3D graphics rendering inside a web browser with full hardware acceleration support. Moreover, being based on an open cross-platform 3D library (OpenGL), it is well supported by most of the existing operative systems. Since GIS are in continuous evolution and, as said before, are moving toward a web-based approach, the enormous potential of these new technologies is exploited.

In order to deal with WebGL, a proper programming language has to be chosen. The most natural choice is JavaScript. JavaScript is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. In order to ease the development, a JavaScript framework should be used (cf. Section 3.4.1). Considering the low-level nature of WebGL, using an higher-level WebGL framework is also strongly recommended (cf. Section 3.4.2).

3.2 HTML5

HTML5 is a markup language for structuring and presenting content for the World Wide Web. It is the fifth revision of the HTML standard and, on October 2012, is still under development. Following its immediate predecessors HTML 4.01 and XHTML 1.1, HTML5 extends and improves the markup available for documents and introduces markup and Application Programming Interfaces (APIs) for complex web applications developing and multimedia handling.

3.2.1 History

The Web Hypertext Application Technology Working Group (WHATWG) began its work on the new standard in 2004. At that time, the World Wide Web Consortium (W3C) was focusing on the XHTML 2.0 draft and HTML 4.01 had not been updated since 2000. In 2007, the HTML5 specification developed by WHATWG was adopted by the new W3C HTML working group, while the XHTML 2.0 draft was definitively abandoned in 2009. W3C and WHATWG are currently working together on the development of HTML5, but with some degree of separation (agreed on July 2012). W3C will continue the HTML5 specification work, focusing on a single definitive standard which is considered as a "snapshot" by WHATWG. The WHATWG will continue its work with HTML as a "living standard", that is, a standard that is never complete and is always being updated and improved.

Although HTML5 has been well known among web developers for years, it became the topic of mainstream media around April 2010 after Apple's then-CEO Steve Jobs issued a public letter titled "Thoughts on Flash" where he concludes that "[Adobe] Flash is no longer necessary to watch video or consume any kind of web content" and that "new open standards created in the mobile era, such as HTML5, will win on mobile devices (and PCs too)".

3.2.2 Features overview

The HTML5 is no longer based on Standard Generalized Markup Language (SGML) despite the similarity of its markup. It has, however, been designed to be backward compatible with common parsing of older versions of HTML. HTML5 also defines in some detail the required processing for invalid documents so that syntax errors will be treated uniformly by all conforming browsers and user agents.

HTML5 introduces a number of new elements and attributes that reflect typical usage on modern websites. In particular, it introduces the new *video*, *audio* and *canvas* elements, as well as the integration of Scalable Vector Graphics (SVG) content and MathML for mathematical formulas. These features are designed to make it easy to include and handle multimedia and graphical content on the web without having to resort to external proprietary plug-ins and APIs. Other new elements, such as *section*, *article*, *header*, *nav* and *footer*, are designed to semantically enrich the document structure, replacing, in some specific cases, more generic elements, such as *div*. Some deprecated elements from HTML 4.01 have been dropped, including purely presentational elements such as *font* and *center*, whose effects have long been superseded by Cascading Style Sheets (CSS). Moreover, other elements from HTML 4.01 have been changed, redefined or oficially standardized.

In addition to specifying markup, HTML5 specifies JavaScript APIs that can be used to develop complex web applications or to provide animation within web pages. Existing Document Object Model (DOM) interfaces are extended and de facto features are documented. Some new APIs are also defined:

- **Canvas 2D context**, for immediate mode 2D rendering inside a *canvas* element.
- **Drag-and-drop**, which integrates this common pointing device gesture into web pages.
- **Cross-document messaging**, which allows a document to communicate with one another across different origins or source domains.
- **Microdata**, which nest semantics within existing content on web pages. Search engines, web crawlers and browsers can extract and process Microdata from a web page and use it to provide a richer browsing experience.
- **Web Storage**, a key-value pair storage framework that provides behavior similar to cookies but with larger storage capacity and improved API.

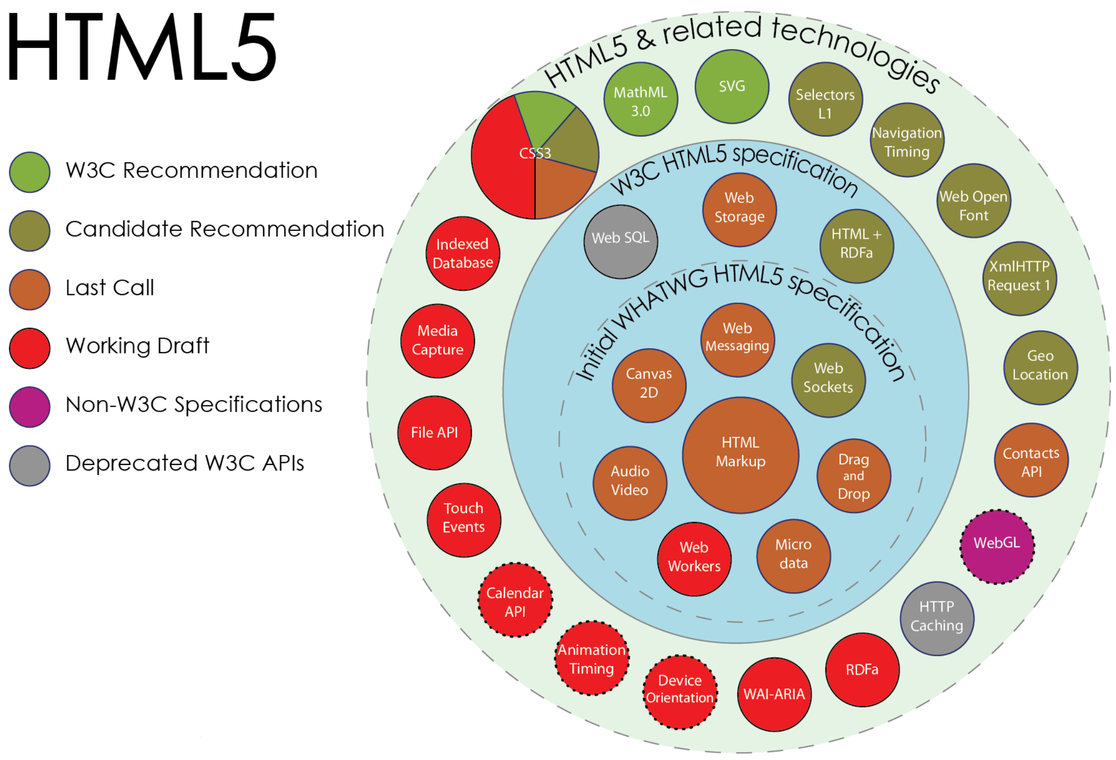


FIGURE 3.1: HTML5 related APIs: specification process status.

Not all of the above technologies are included in the W3C HTML5 specification, though they are in the WHATWG HTML specification. Some related technologies, which are not part of either the W3C HTML5 or the WHATWG HTML specification, are maintained by W3C separately. Here we have some examples:

- **Geolocation**, an interface to retrieve geographical location information for a client-side device.
- **Web SQL Database**, a local SQL database (no longer maintained).
- **Indexed Database**, a web browser standard interface for a local database of records holding simple values and hierarchical objects.
- **File**, which handles file uploads and file manipulations.
- **File Writer**, an API for writing files from web applications.
- **Web Audio**, a high-level API for processing and synthesizing audio in web applications.

Other related technologies are also maintained outside the W3C and the WHATWG. A suitable example is WebGL, a 3D context for the HTML5 *canvas* element maintained by the Khronos Group.

3.2.3 The *canvas* element

Canvas was initially introduced by Apple for use inside their own Mac OS X WebKit component in 2004, powering applications like Dashboard widgets and the Safari browser. In 2006, it was adopted by Opera web browser and standardized by the WHATWG on the new proposed HTML5 specification.

The canvas element allows for dynamic scriptable rendering of 2D shapes, bitmap images and 3D graphics. It implements a low-level raster-based procedural model that directly updates a bitmap and does not have a built-in scene-graph. The drawing approach defined by canvas is therefore quite different from the one defined by SVG, probably the main alternative to canvas for in-browser graphics manipulation. SVG is built on a higher-level vector-based approach, in which every drawn element is recorded as a DOM object that is subsequently rendered to a bitmap. As a result, while SVG objects can be modified through their attributes, canvas objects must instead be redrawn.

A canvas element consists of a drawable region defined in HTML code with height and width attributes. JavaScript code may access the area through the proper APIs, called “contexts”. Two contexts are currently available: canvas 2D context, maintained by W3C and WHATWG, and WebGL rendering context, maintained by Khronos Group.

Canvas 2D rendering context

The 2D context represents a flat Cartesian surface whose origin is at the top left corner, with the coordinate space (expressed in pixels) having x values increasing when going right and y values increasing when going down. Each context maintains a stack of drawing states. A state records the drawing style (fill and stroke colors, line width, dash patterns, font, text alignment, etc) and the transformations (translation, rotation, scaling) applied when an object, text or shape, is placed onto the canvas. The current state can be easily saved and restored, allowing to pass from a state to another with only few lines of code.

Only one basic shape is supported: filled or stroked rectangle. In order to draw more complex shapes, paths have to be used. We can think of a path as a collection of pixels going from a starting point to an ending point, possibly composed by sub-paths. Drawing a shape using paths consists of four basic steps:

1. begin the path,
2. add points to the current path using lines, arcs or more complex primitives (rectangles, circumferences, ellipses, quadratic curves) as connections,
3. close path,
4. fill or stroke path.

Simple text can also be drawn. The 2D context allows to set up text alignment, both horizontal and vertical, and to define font rules using CSS syntax. As for shapes, text can be filled or stroked.

The *drawImage* method can be used to draw images onto the canvas. This method can be invoked with three different set of arguments, specifying if the image has to be scaled, clipped or both. A source rectangle and a destination rectangle are defined. When the *drawImage* method is invoked, the region of the image specified by the source rectangle is painted on the region of the canvas specified by the destination rectangle, possibly filtering the original image data (the actual employed filter is selected by the web browser) if scaling is needed. *DrawImage* can take either an *img* element, a *video* element or another *canvas* element as source. If the source is a video or an animated image, only one frame (the one at the current playback position in the first case, the poster frame in the second case) is used as the source image.

The 2D context allows to retrieve pixels data from canvas using the *getImageData* method. This method returns an *ImageData* object representing the canvas bitmap for the region specified by a source rectangle. Width and height properties of an *ImageData object* represent respectively the number of pixels per row and the number of rows in the image data, while image data is stored as a canvas pixel *ArrayBuffer* referenced by the data property. In a Canvas pixel *ArrayBuffer*, data is represented in left-to-right order, row by row top to bottom, starting with the top left, with each pixel's red, green, blue, and alpha components being given in that order for each pixel. Each component of each pixel represented in this array must be in the range 0..255, representing the 8 bit value for that component. Image data can be drawn back onto canvas using the *putImageData* method. Therefore, image data can be taken from the canvas, elaborated and put back to the canvas.

3.3 WebGL

WebGL is a cross-platform, cross-browser and plug-in free API for immediate mode 3D rendering inside web pages. WebGL is derived from OpenGL ES 2.0 and provides similar rendering functionality, but in an HTML environment. It is designed as a rendering context for the HTML5 canvas element, so a full integration with all DOM interfaces is provided and any DOM-compatible language can interact with this API (JavaScript is the most natural choice). Everyone familiar with OpenGL ES 2.0 will recognize WebGL as a shader-based API using OpenGL Shading Language (GLSL), with constructs that are semantically similar to those of the underlying OpenGL ES 2.0 API (the WebGL specification stays very close to the OpenGL ES 2.0 specification, with some minor concessions). WebGL is a very low-level 3D API with flexible primitives that can be applied to any use case. Third party libraries can provide an API on top of WebGL that is more tailored to specific areas, thus adding a convenience layer to WebGL that can accelerate and simplify development.

3.3.1 History

WebGL grew out of the Canvas 3D experiments started by Vladimir Vukicevic at Mozilla. Vukicevic first demonstrated a Canvas 3D prototype in 2006. By the end of 2007, both Mozilla and Opera had made their own separate implementations. In early 2009 the non-profit technology consortium Khronos Group started the WebGL Working Group, with initial participation from Apple, Google, Mozilla, Opera and others. Version 1.0 of the WebGL specification, based on version 2.0 of the OpenGL ES specification, was released on March 2011.

3.3.2 OpenGL ES

OpenGL ES is an API for advanced 3D graphics targeted at hand-held and embedded devices. In the desktop world there are two standard 3D APIs, DirectX and OpenGL. DirectX is the de facto standard 3D API for any system running the Microsoft Windows operating system and is used by the majority of 3D games on that platform. OpenGL is a cross-platform standard 3D API for desktop systems running Linux, various implementations of UNIX, Mac OS X, and Microsoft Windows. It is a widely accepted standard 3D API that has seen significant real-world usage. Due to the widespread adoption of OpenGL as a 3D API, it made sense to start with the desktop OpenGL API in developing an open standard 3D API for hand-held and embedded devices and modifying it to meet the needs and constraints of this kind of devices (limited processing capabilities and memory availability, low memory bandwidth, sensitivity to power consumption, etc). The following criteria have been adopted in the definition of the OpenGL ES specification:

- In order to create an API suitable for constrained devices, any redundancy from the OpenGL API was removed. In any case where there was more than one way of performing the same operation, the most useful method was taken and the redundant techniques were removed.
- Despite the simplifications, OpenGL ES was designed to maintain some degree of compatibility with OpenGL. Any application written to the embedded subset of functionality in OpenGL should also run on OpenGL ES.
- New features were introduced to address specific constraints of hand-held and embedded devices. For example, to increase performance while keeping power consumption reasonable.
- Any OpenGL ES implementation should meet certain acceptable and agreed-on standards for image quality, correctness and robustness. In order to achieve this goal, every OpenGL ES implementation must pass some conformance tests to be considered compliant.

Three versions of the OpenGL ES specification have been released by Khronos Group so far. The OpenGL ES 1.0 and 1.1 specifications implement a fixed function pipeline and are derived from the OpenGL 1.3 and 1.5 specifications, respectively. The OpenGL ES 2.0 specification implements a programmable graphics pipeline and is derived from the OpenGL 2.0 specification. Being derived from a revision of the OpenGL specification means that the corresponding OpenGL specification was used as the baseline for determining the feature set in the particular revision of OpenGL ES.

3.3.3 OpenGL ES 2.0 Graphics pipeline

OpenGL ES 2.0 implements a graphics pipeline with programmable shading. The diagram in Figure 3.2 describes its structure highlighting the two programmable stages (shaded boxes).

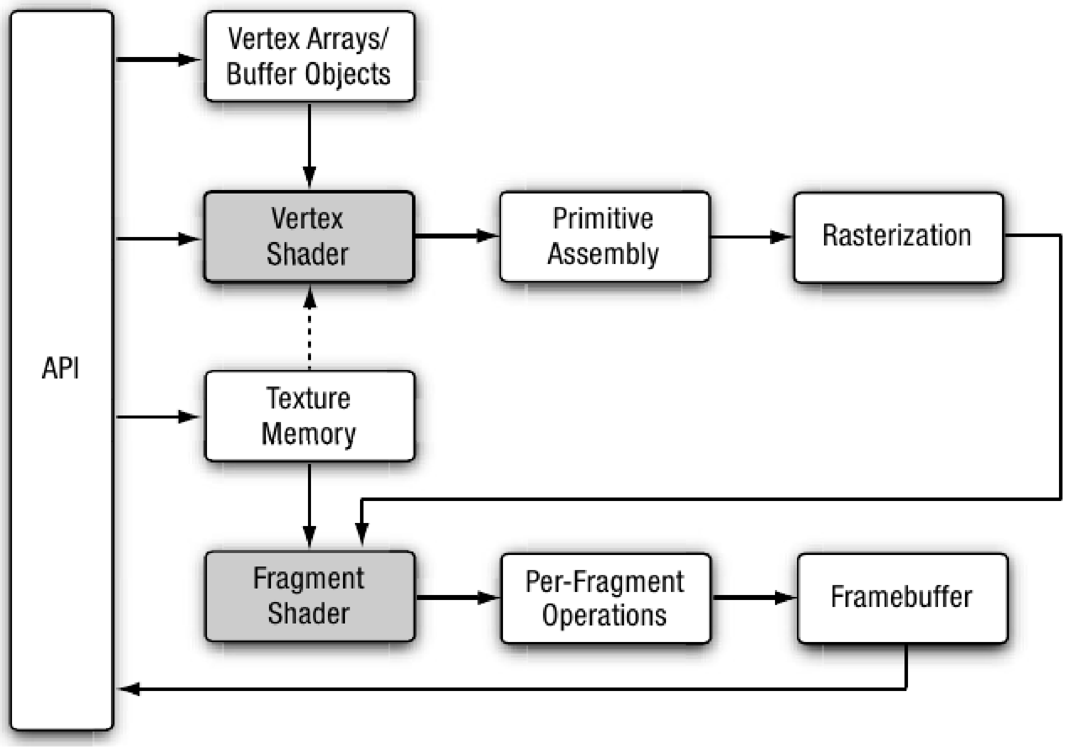


FIGURE 3.2: OpenGL ES 2.0 programmable graphics pipeline.

Vertex shader

The vertex shader implements a general purpose programmable method for operating on vertices. Vertex shaders can be used for traditional vertex-based operations such as transforming the position by a matrix, computing the lighting equation to generate a per-vertex color, and generating or transforming texture coordinates. Alternately, more complex operations can be specified. The inputs to the vertex shader consist of the following:

- **Attributes**, per-vertex data supplied using vertex arrays.
- **Uniforms**, constant data used by the vertex shader.
- **Shader program**, vertex shader program source code or executable that describes the operations that will be performed on the vertex.

The outputs of the vertex shader are called varying variables.

Primitive assembly

In the primitive assembly stage, the shaded vertices are assembled into individual geometric primitives (triangles, lines, point-sprites, etc). For each primitive, it must be determined whether the primitive lies within the view frustum (the region of 3D space that is visible on the screen). If the primitive is not completely inside the view frustum, the primitive might need to be clipped to the view frustum. If the primitive is completely outside, it is discarded. A culling operation can also be performed that discards primitives based on whether they face forward or backward. After clipping and culling processes, the position of each vertex is converted into screen coordinates.

Rasterization

Rasterization is the process that converts primitives into a set of two-dimensional fragments, which are then processed by the fragment shader. The varying values assigned to each vertex of a primitive are interpolated in order to generate varying values for each fragment. The two dimensional fragments produced by the rasterization stage represent pixels that can be drawn on the screen.

Fragment shader

The fragment shader implements a general-purpose programmable method for operating on fragments. The fragment shader is executed for each generated fragment by the rasterization stage and takes the following inputs:

- **Varying variables**, outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation.
- **Uniforms**, constant data used by the fragment shader.
- **Shader program**, fragment shader program source code or executable that describes the operations that will be performed on the fragment.

The fragment shader can either discard the fragment or generate a color value for it.

Per-fragment operations

The per-fragment operations stage performs the following functions and tests on each fragment:

- **Pixel ownership test**. Determines if a pixel is currently owned by the OpenGL ES context.
- **Scissor test**. If enabled, fragments lying outside a specified rectangular region are discarded.
- **Stencil and depth tests**. If enabled, determine if a fragment has to be rejected or not based on its stencil and depth values (implementing stenciling and hidden surface removal).
- **Blending**. If enabled, pixels output by the fragment shader may be blended with pixel values already present in the framebuffer.
- **Dithering**. If enabled, it can be used to minimize the artifacts that can occur from using limited precision to store color values in the framebuffer.

At the end of the per-fragment stage, either the fragment is rejected or a fragment color, depth, or stencil value is written to the framebuffer at location.

3.4 Frameworks selection

A development framework is a universal, reusable software platform (usually an integrated set of libraries) used to develop applications. Software frameworks are designed to facilitate and speed up the development process providing a ready-to-use set of common functionality and high-level abstractions. Some frameworks also provide a basic architectural structure for a particular class of applications. Developers can use this structure (properly edited and extended) as a basis to build specific applications.

When choosing the best framework for a given project, the provided functionality is not the only aspect to evaluate. Other aspects, such as performances, security, flexibility and learning curve, are likewise important. The purpose of this section is to identify both a JavaScript framework and a WebGL framework that fit this project. In order to accomplish this goal, some well-known frameworks will be analyzed and compared.

3.4.1 JavaScript framework selection

The fundamental purpose of a JavaScript framework is to extend and improve native JavaScript functionality. Most of JavaScript frameworks are also designed to hide the differences between browser-specific implementations, thus providing a cross-browser programming environment.

In order to meet project's needs, a JavaScript framework must be:

- browser independent;
- modular;
- easy to use and quickly adoptable;
- well supported and documented;

It must also provide support for:

- basic DOM manipulation;
- events handling;

- asynchronous programming (deferred, promises, etc);
- class-based programming (for better code organization).

The following analysis will consider the two most widely used and valued JavaScript frameworks: jQuery and Dojo Toolkit.

jQuery

jQuery is an open source project released under MIT (Massachussetts Institute of Technology) License. It was first released in August 2006 at BarCamp NYC by John Resig and is probably the most popular JavaScript framework in use today.

jQuery is a cross-browser JavaScript framework designed to simplify the client-side scripting of HTML. Its syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop AJAX (Asynchronous JavaScript and XML) applications. jQuery also provides capabilities for developers to create plug-ins on top of the basic JavaScript framework. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets.

Dojo Toolkit

Dojo Toolkit is an open source project dual-licensed under the Modified BSD (Berkeley Software Distribution) License or the Academic Free License (AFL). It was started by Alex Russell, Dylan Schiemann, David Schontzler, and others in 2004. The first stable release dates back to November 2007.

Dojo Toolkit is a modular JavaScript framework designed to ease the rapid development of cross-platform, JavaScript/AJAX-based applications and web sites. A full distribution of the toolkit consists of three main packages:

- **Dojo**: sometimes referred to as the "core", this is the main part of Dojo and the most generally applicable packages and modules are contained in here. The core covers a wide range of functionality like AJAX, DOM manipulation, class-type programming, events, promises, data stores, drag-and-drop and internationalization libraries.

- **Dijit**: contains an extensive set of widgets (user interface components) and the underlying system to support them. It is built fully on-top of the Dojo core.
- **DojoX**: this is a package dedicated to the development of additional Dojo functionality. It contains a collection of packages and modules that provide a vast array of additional functionality that are built upon both the Dojo core and Dijit. Packages and modules contained in DojoX have varying degrees of maturity: some of the modules are extremely mature and some are highly experimental.

Modularity is one of the most representative characteristics of Dojo. Starting from 1.7 version, a full support for Asynchronous Module Definition (AMD) has been introduced. AMD allows to selectively load modules at startup, thus eliminating the need to load the entire toolkit. Moreover, it is possible to define and load custom modules, and any third-party AMD-compliant module.

Chosen framework

Both jQuery and Dojo are cross-browser, and provide the required functionality about DOM manipulation, event handling and asynchronous programming. However, Dojo provides an higher level of modularity (further improved by the AMD system) and a good support for class-based programming than jQuery.

Unlike jQuery, Dojo Toolkit is developed and supported according to an enterprise approach. For this reason, it is more reliable and better fits long-term, large-scale projects. In practice, Dojo Toolkit offers better solutions for the development of complex web applications, while jQuery better fits the development of dynamic web sites.

Because of its enterprise-level approach (and its not so fine documentation), Dojo may have a bit steeper learning curve than jQuery. Nevertheless, it remains the best choice for this project.

3.4.2 WebGL framework selection

As previously mentioned, WebGL is a very low-level API, so even the most trivial task requires a considerable amount of lines of code to be accomplished. For this reason, native WebGL is not the best choice to develop complex 3D applications.

A WebGL framework/engine is designed to hide most of low-level details, and to provide ready-to-use high-level primitives for both basic and advanced functionality. A good engine has to achieve a reasonable level of abstraction, allowing the developer to direct access lower-level capabilities if needed.

In order to meet project's needs, a WebGL framework must be:

- browser independent;
- based on a scene-graph (to efficiently manage complex 3D scenes);
- easy to use and quickly adoptable;
- well supported and documented.

It must also provide support for:

- cameras (ready-to-use perspective and orthographic views);
- materials, lights and textures;
- geometry utilities (to easily reproduce basic shapes such as, planes, cubes, etc);
- math utilities (to easily handle 3D transformations);
- basic ready-to-use control systems;
- direct loading of 3D models in different formats.

Nice to have features (for future developments):

- full access to shaders;
- integrated Level Of Detail (LOD) system;

- basic support for physics and animations.

The following analysis will consider some popular WebGL frameworks: C3DL, SpiderGL and Three.js.

C3DL

The Canvas 3D JS Library (C3DL) is an open source project released under MIT License. It was first developed as part of the CATGames Research Network at Seneca College (Toronto, Canada), which was working on a middle layer API for Canvas 3D (the precursor of WebGL). The development seems currently stuck: the last release (2.2) dates back to March 2011, and no development news has been published since June 2011.

C3DL is an easy to use WebGL framework thought for the development of simple 3D web applications. It provides basic support for scene organization, cameras, materials, textures, lights, shaders customization and COLLADA 3D models loading.

The framework has a good API reference and offers some useful introductory tutorials, but lacks meaningful code examples and community support.

SpiderGL

SpiderGL is an open source project released under Modified BSD License. It was designed by Marco Di Benedetto, Federico Ponchio, Fabio Ganovelli and Roberto Scopigno from The Visual Computing Lab of ISTI-CNR (Institute of Information Science and Technology - Italian National Council of Research). This framework has been abandoned for a very long time. The community resumed the support to the library very recently (Summer 2012).

SpiderGL is a JavaScript framework designed to develop web applications using WebGL. It provides data structures and algorithms to ease the use of WebGL, to define and manipulate shapes, to import 3D models in various formats (JSON and OBJ currently supported) and to handle asynchronous data loading.

SpiderGL was designed keeping in mind three fundamental qualities:

- **Efficiency**: working with JavaScript and WebGL, efficiency is not only a matter of asymptotic bounds, but finding the most efficient mechanism to do an operation is also crucial.
- **Simplicity/Short learning time**: users should be able to reuse as much as possible of their former knowledge on the subject and take advantage of the library quickly. For this reason SpiderGL carefully avoids over-abstractions: almost all of the function names in the library have a one to one correspondence with OpenGL commands or with geometric/mathematic entities.
- **Flexibility**: SpiderGL does not try to hide native WebGL functions. Conversely, WebGL and SpiderGL calls can be used almost seamlessly.

SpiderGL is composed of five modules:

- **GL**: gives access to WebGL functionalities.
- **MESH**: manages 3D model definition and rendering.
- **ASYNC**: for asynchronous loading.
- **UI**: provides user interface utilities.
- **Space**: provides math and geometry utilities.

Three.js

Three.js is an open source project released under MIT License and supported by a large community of active developers. It was first released on GitHub by Ricardo Cabello in April 2010 and it is currently under heavy development (51th revision released in September 2012). The lack of an exhaustive traditional documentation (still under construction) is well compensated by a large repository of code examples and an excellent community support (support for developers committing to the library is provided via the Issues forum on GitHub, while support for developers building applications and web pages is provided via StackOverflow; real-time on-line support is also provided using IRC via Freenode).

Three.js aims to create a lightweight JavaScript 3D engine with a very low level of complexity. The engine can render 3D graphics using WebGL, the canvas 2D context or SVG and provides an interface that completely abstracts the underlying rendering mechanisms, so that the same code can be ported on multiple renderers.

Three.js is a scene-graph based engine built on three basic elements:

- A **Scene**, containing a set of 3D objects (that can be added, removed or updated run-time).
- A **Camera**, used by the renderer to set the viewable area.
- A **Renderer** that, receiving a scene and a camera as input, renders the scene using the canvas 2D context, SVG or WebGL.

Features overview:

- **Cameras**: perspective and orthographic.
- **Controllers**: first person, trackball, fly, path and roll.
- **Animations**: morph and keyframe.
- **Lights**: ambient, direction, point and spot.
- **Materials**: basic, Lambert, Phong and more.
- **Shaders**: access to full WebGL capabilities; lens flare, depth pass and extensive post-processing library.
- **Objects**: meshes, particles, sprites, lines, ribbons and bones, all with level of detail (LOD).
- **Geometry**: plane, cube, sphere, torus, 3D text and more.
- **Loaders**: support for direct loading of 3D models in different formats (binary, COLLADA, JSON and more).
- **Utilities**: full set of time and 3D math functions including frustum, Quaternion, matrix, UVs and more.
- **Export/Import**: utilities to create Three.js-compatible JSON files from within: Blender, CTM, FBX, 3D Max, and OBJ.

Chosen framework

C3DL provide too much limited functionality to fit a complex project. The scene is just a list of models, so no hierarchical organization is provided (in other words, no scene-graph). Just basic geometric primitives are provided, so the use of 3D models is quite obliged. Nevertheless, only COLLADA models are supported. The camera implementation supports FOV (Field of View), near plane and far plane, but lacks methods to automatically compute a perspective or orthographic projection. Moreover, lights and materials management are very limited and no integrated control systems are provided. Considering its functional limitations and the poor support (the development is stuck for more than one year), C3DL has been discarded.

SpiderGL offers good basic features, however it does not provide a scene-graph, and a good materials system is also missing. Nevertheless, the most important defect with SpiderGL is the lack of support. The development has just been resumed after a long period of inactivity and no assurance about the future has been given. Moreover, there is no documentation apart the source code and there are no official communication channels to support developers or to submit bugs and bug-fixes. Considering the lack of support, the steep learning curve (due to an excessive low-level approach) and the functional limitations, SpiderGL has been discarded.

Three.js is currently the most popular WebGL framework. It is in continuous development and is supported by an active community of developers. The auto-explicative source code and the large repository of meaningful examples allow to quickly take full advantage of the provided features. Three.js is based on a scene-graph approach, provides a complete light system (ambient, directional, point and spot light provided) and implements a good materials system (with the possibility to directly access shaders for full customization). It provides geometry utilities that allow the in-line construction of both basic (cubes, planes, etc) and complex (spheres, cylinders, tetrahedrons, octahedrons, toruses, etc) shapes, and a full set of math utilities (vectors, matrices, quaternions, etc) to handle 3D transformations. The camera can be perspective, orthographic or combined (allowing to automatically switch from one type of view to the other). Many different 3D model formats are supported, and a specific JSON format (compatible with 3D editors such as Blender) is provided. Three.js also defines many different built-in control systems and provides basic support for animations and LOD. Physics is not directly supported, but can be integrated using an extension named Phisijs.

Three.js is currently the most complete WebGL framework and well fits the project's needs, so it is definitely the best choice.

Chapter 4

Geospatial data formats and services

4.1 Overview

The Open Geospatial Consortium (OGC) maintains many standards for geospatial data encoding and handling . This Chapter analyses the most relevant standards for this project's purposes. Both the encoding formats (Section 4.2) and the web services (Section 4.3) will be considered.

4.2 OGC 3D-capable formats

The OGC provides two suitable standards for representing 3D geospatial features: KML and CityGML. KML (Keyhole Markup Language) mainly focuses on geographic annotations (place marks, textual descriptions, tagging, etc), but can also be used to define the location and orientation of textured 3D objects encoded as COLLADA models. On the other hand, CityGML is a GML (Geography Markup Language) extension specifically thought for the representation of 3D urban objects, so it natively embeds the encoding of 3D information, without requiring external models. Considering the project's scope, CityGML is definitely more complete and suitable for our needs. Sections 4.2.1 and 4.2.2 provide a brief description of KML and CityGML respectively, outlining their main features and characteristics.

4.2.1 KML

Keyhole Markup Language (KML) is an XML grammar used to encode and transport representations of geographic data for display in an earth browser, such as a 3D virtual globe, 2D web browser application, or 2D mobile application. Put simply: KML encodes what to show in an earth browser, and how to show it. Each geographic reference always has a longitude and a latitude. Other data, such as tilt, heading and altitude, can make the view more specific. KML uses a tag-based structure with nested elements and attributes and is based on the XML standard.

KML was developed for use with Google Earth, which was originally named Keyhole Earth Viewer. It was created by Keyhole, Inc, which was acquired by Google in 2004. KML became an international standard of the Open Geospatial Consortium in 2008. KML is complementary to most of the key existing OGC standards including Geography Markup Language (GML), Web Feature Service (WFS) and Web Map Service (WMS). Currently, KML 2.2 utilizes certain geometry elements derived from GML 2.1.2. These elements include point, line string, linear ring, and polygon.

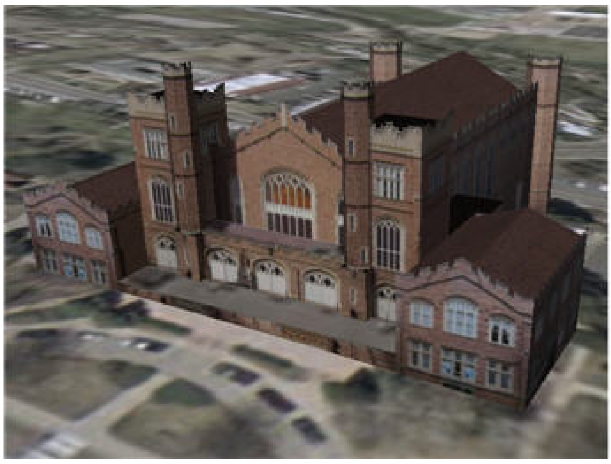


FIGURE 4.1: Textured COLLADA model georeferenced by KML (from Google Earth).

KML can be used to:

- Annotate the Earth.
- Specify icons and labels to identify locations on the surface of the planet.
- Create different camera positions to define unique views for KML features.
- Define image overlays to attach to the ground or screen.

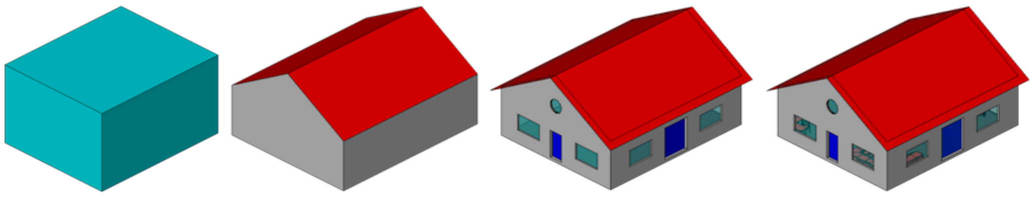
- Define styles to specify KML feature appearance.
- Write HTML descriptions of KML features, including hyperlinks and embedded images.
- Organize KML features into hierarchies.
- Locate and update retrieved KML documents from local or remote network locations.
- Define the location and orientation of textured 3D objects.

KML files are very often distributed in KMZ files, which are zipped files with a ".kmz" extension. The contents of a KMZ file are a single root KML document and optionally any overlays, images, icons, and COLLADA 3D models referenced in the KML.

The KML community is wide and varied. Casual users create KML Placemarks to identify their homes, describe journeys, and plan cross-country hikes and cycling ventures. Scientists use KML to provide detailed mappings of resources, models, and trends such as volcanic eruptions, weather patterns, earthquake activity, and mineral deposits. Real estate professionals, architects, and city development agencies use KML to propose construction and visualize plans. Students and teachers use KML to explore people, places, and events, both historic and current. Organizations such as National Geographic, UNESCO, and the Smithsonian have all used KML to display their rich sets of global data.

4.2.2 CityGML

CityGML is an open data model and an XML-based format for the storage and exchange virtual 3D city models. It is implemented as an application schema for the Geography Markup Language 3 (GML3), the extendible international standard for spatial data exchange issued by the Open Geospatial Consortium (OGC). It defines the classes and relations for the most relevant topographic objects in cities and regional models with respect to their geometrical, topological, semantical and appearance properties. Included are generalization hierarchies between thematic classes, aggregations, relations between objects, and spatial properties. This thematic information goes beyond graphic exchange formats and makes it possible to employ virtual 3D city models for sophisticated analysis tasks in different application domains like simulations, urban data mining, facility management, and thematic inquiries. The OGC Members adopted version 1.0.0 of CityGML as an official OGC Standard in August 2008. In Spring 2012, the OGC Members approved version CityGML 2.0.0. CityGML is intended to become an open standard and therefore can be used free of charge.



(a) LOD1. (b) LOD2. (c) LOD3. (d) LOD4.

FIGURE 4.2: CityGML house model.

Features of CityGML:

- Geospatial information model (ontology) for urban landscapes based on the ISO 191xx family.
- GML3 representation of 3D geometries, based on the ISO 19107 model.
- Representation of object surface characteristics (e.g. textures, materials).
- Taxonomies and aggregations.
  - Digital Terrain Models as a combination of (including nested) triangulated irregular networks (TINs), regular rasters, break and skeleton lines, mass points.
  - Sites (currently buildings; bridges and tunnels in the future).
  - Vegetation (areas, volumes and solitary objects with vegetation classification).
  - Water bodies (volumes, surfaces).
  - Transportation facilities (both graph structures and 3D surface data).
  - Land use (representation of areas of the earth's surface dedicated to a specific land use).
  - City furniture.
  - Generic city objects and attributes.

  - User-definable (recursive) grouping.

- Multiscale model with 5 well-defined consecutive Levels of Detail (LOD):
  - LOD0 – regional, landscape.
  - LOD1 – city, region.
  - LOD2 – city districts, projects.
  - LOD3 – architectural models (outside), landmarks.
  - LOD4 – architectural models (interior).
- Multiple representations in different LODs simultaneously; generalization relations between objects in different LODs.
- Optional topological connections between feature (sub)geometries.
- Application domain extensions: Specific hooks in the CityGML schema allow to define application specific extensions, for example for noise pollution simulation.

4.3 OGC web services

As mentioned, CityGML is an application schema of GML. CityGML features can then be efficiently handled through a Web Feature Service (WFS), an OGC standard designed to manipulate and retrieve GML-encoded geospatial features. The OGC provides other two powerful standards for handling geospatial data in different forms and with different purposes: WMS and WCS. WMS (Web Map Service) dynamically produces spatially referenced maps from geographic information. WMS yields just a pictorial rendering of maps in a graphical format, so no detailed information about the underlying data is provided. WCS (Web Coverage Service) handles geographic data in the form of "coverages". A coverage is the digital representation of some spatio-temporal phenomenon defined by a range of different values for each location/time. Sections 4.3.1, 4.3.2 and 4.3.3 provide a brief description of WFS, WMS and WCS respectively, outlining their main features and characteristics. Section 4.3.4 provides some information about GeoServer, the chosen geospatial server implementing the described standards.

4.3.1 Web Feature Service

Web Feature Service (WFS) is an OGC open standard which defines interfaces for data access and manipulation operations on geographic features using HTTP. Via these interfaces, a client can retrieve, combine, and manage geospatial data encoded in GML from different sources.

The standard specification defines the following operations:

- **GetCapabilities** (mandatory). A web feature service must be able to describe its capabilities. Specifically, it must indicate which feature types it can service and what operations are supported on each feature type.
- **DescribeFeatureType** (mandatory). A web feature service must be able, upon request, to describe the structure of any feature type it can service.
- **GetFeature** (mandatory). A web feature service must be able to service a request to retrieve feature instances. In addition, the client should be able to specify which feature properties to fetch and should be able to constrain the query spatially and non-spatially.
- **GetGmlObject** (optional). A web feature service may be able to service a request to retrieve element instances by traversing XLinks that refer to their XML IDs. In addition, the client should be able to specify whether nested XLinks embedded in returned element data should also be retrieved.
- **Transaction** (optional). A web feature service may be able to service transaction requests. A transaction request is composed of operations that modify features; that is create, update, and delete operations on geographic features.
- **LockFeature** (optional). A web feature service may be able to process a lock request on one or more instances of a feature type for the duration of a transaction. This ensures that serializable transactions are supported.

Based on the operation descriptions above, three classes of web feature services can be defined:

- **Basic WFS**. A basic WFS would implement the GetCapabilities, DescribeFeatureType and GetFeature operations. This would be considered a READ-ONLY web feature service.
- **XLink WFS**. An XLink WFS would support all the operations of a basic web feature service and in addition it would implement the GetGmlObject operation for local and/or remote XLinks, and offer the option for the GetGmlObject operation to be performed during GetFeature operations.
- **Transaction WFS**. A transaction web feature service would support all the operations of a basic web feature service and in addition it would implement the Transaction operation. Optionally, a transaction WFS could implement the GetGmlObject and/or LockFeature operations.

4.3.2 Web Map Service

Web Map Service is an OGC open standard for serving georeferenced maps over the Internet. The Open Geospatial Consortium became involved in developing standards for web mapping after a paper was published in 1997 by Allan Doyle, outlining a "WWW Mapping Framework". The OGC established a task force to come up with a strategy, and organized the "Web Mapping Testbed" initiative, inviting pilot web mapping projects that built upon ideas by Doyle and the OGC task force. Results of the pilot projects were demonstrated in September 1999, and a second phase of pilot projects ended in April 2000. The Open Geospatial Consortium released WMS version 1.0.0 in April 2000, followed by version 1.1.0 in June 2001, and version 1.1.1 in January 2002. WMS version 1.3.0, which is currently the last published, has been released in January 2004.

The standard specifies the behavior of a service that dynamically produces spatially referenced maps from geographic information. WMS can use one or more distributed geospatial databases as source of geographic information and returns maps in a variety of formats (JPEG, PNG, GIF, GeoTIFF, SVG, etc) that can be displayed in a browser application. Note that this standard is only applicable to pictorial renderings of maps in a graphical format, it is not applicable to retrieval of actual feature data or coverage data values.

WMS is based on a simple HTTP interface which specifies the following operations:

- **GetCapabilities** (mandatory), returns service metadata, which is a machine-readable (and human-readable) description of the server's information content and acceptable request parameter values. The response to a GetCapabilities request shall be an XML document formatted according to a standard defined XML Schema.
- **GetMap** (mandatory), returns a map. The response to a valid GetMap request shall be a map of the spatially referenced information layer requested in the desired style, and having the specified coordinate reference system, bounding box, size, format and transparency. An invalid GetMap request shall yield an error output in the requested Exceptions format (or a network protocol error response in extreme cases).
- **GetFeatureInfo** (optional), provides more information about features in the pictures of maps that were returned by previous GetMap requests. The canonical use case for GetFeatureInfo is that a user sees the response of a GetMap request and chooses a point (I,J) on that map for which to obtain more information. The server shall return a response according to the requested format if the request is valid, or issue a service exception otherwise. The nature of the response is at the discretion of the service provider, but it shall pertain to the feature(s) nearest to (I,J). This operation is only supported for those Layers for which the attribute queryable=1 (true) has been defined or inherited.

4.3.3 Web Coverage Service

The Web Coverage Service (WCS) supports electronic retrieval of geospatial data as "coverages" – that is, digital geospatial information representing space-varying phenomena. A WCS provides access to potentially detailed and rich sets of geospatial information, in forms that are useful for client-side rendering, multi-valued coverages, and input into scientific models and other clients. The WCS may be compared to the OGC

Web Map Service (WMS) and the Web Feature Service (WFS). Like them it allows clients to choose portions of a server's information holdings based on spatial constraints and other criteria. Unlike the WMS, which portrays spatial data to return static maps (rendered as pictures by the server), the Web Coverage Service provides available data together with their detailed descriptions, defines a rich syntax for requests against these data, and returns data with its original semantics (instead of pictures) which may be interpreted, extrapolated, etc. – and not just portrayed. Unlike WFS, which returns discrete geospatial features, the Web Coverage Service returns coverages representing space-varying phenomena that relate a spatio-temporal domain to a (possibly multidimensional) range of properties.

Coverages have a domain comprised of regularly (sometimes irregularly) spaced locations along 0, 1, 2, or 3 axes of a spatial coordinate reference system. Their domain may also have a time dimension, which may be regularly or irregularly spaced. A coverage defines, at each location in the domain, a set of fields that may be scalar-valued (such as elevation), or vector-valued (such as brightness values in different parts of the electromagnetic spectrum). These fields (and their values) are known as the range of the coverage.

The coverage model is defined by the OGC standard *GML 3.2.1 Application Schema - Coverages* (often referred to as GMLCOV) which in turn is based on the Geography Markup Language (GML) 3.2. This standard defines an abstract type for coverages which consists of the following components:

- **Coverage domain**. The extent where valid values are available.
- **Range set**. The set of values the coverage consists of, together with their locations.
- **Range type**. A type definition of the range set values.
- **Metadata**. A slot where any kind of metadata can be added.

This abstract coverage is refined into several concrete coverage types, which can be instantiated.

As coverages are conceptually concisely defined through GML, a natural representation is GML itself. However, this is not mandatory: any of a series of data formats can be used to encode a coverage, such as GeoTIFF, NetCDF, HDF-EOS, or NITF. As some of these encoding formats are not capable of incorporating all metadata making up a coverage, GMLCOV foresees a multi-part MIME encoding (see Figure 4.3) where the first component encodes the coverage description (domain extent, range type, metadata, etc) and the second part consists of the range set payload using some encoding format.

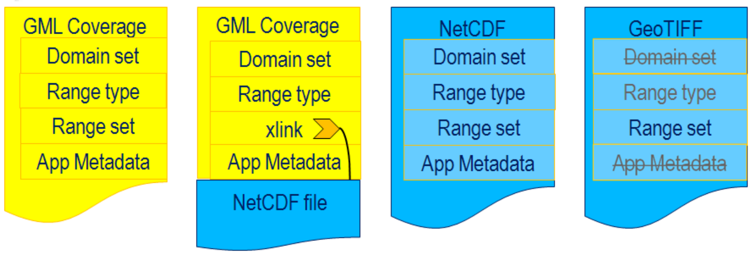


FIGURE 4.3: Different coverage encodings.

The WCS interface specifies three operations that may be requested by a WCS client and performed by a WCS server:

- **GetCapabilities** (mandatory). This operation allows a client to request the service metadata (or Capabilities) document. This XML document describes the abilities of the specific server implementation, usually including brief descriptions of the coverages available on the server. This operation also supports negotiation of the specification version being used for client-server interactions. Clients would generally request the GetCapabilities operation and cache its result for use throughout a session, or reuse it for multiple sessions. When the GetCapabilities operation does not return descriptions of its available coverages, that information must be available from a separate source, such as an image catalog.

- **DescribeCoverage** (mandatory). This operation allows a client to request full descriptions of one or more coverages served by a particular WCS server. The server responds with an XML document that fully describes the identified coverages.

- **GetCoverage** (mandatory). This operation allows a client to request a coverage comprised of selected range properties at a selected set of geographic locations. The server extracts the response data from the selected coverage, and encodes it in a known coverage format. The GetCoverage operation is normally run after GetCapabilities and DescribeCoverage operation responses have shown what requests are allowed and what data are available.

4.3.4 GeoServer

WMS, WFS and WCS are implemented by many geospatial servers. For this project, GeoServer has been chosen.

GeoServer is an open source software licensed under the GNU General Public License (GPL). It was started in 2001 and is currently under constant development (the last stable version, 2.2, has been released in September 2012). Being a community-driven project, GeoServer is developed, tested, and supported by a diverse group of individuals and organizations from around the world.

GeoServer is a geospatial server written in Java that allows users to share and edit geographic data. Designed for interoperability, it publishes data from any major spatial data source (both vectorial and raster data formats are supported) using the OGC open standards. GeoServer is the reference implementation of the Web Feature Service (WFS) and Web Coverage Service (WCS) standards, as well as a high performance certified compliant Web Map Service (WMS).

4.4 Terrain data management

As introduced in Section 2.1, the implementation of the designed streaming framework focuses on textured terrain. This Section describes the simple strategy adopted to store and retrieve terrain data and textures via WMS. Sections 4.4.1 and 4.4.2 provides a brief description for the basic concepts involved: Digital Elevation Model (DEM) and heightmap.

The information needed to represent terrain is usually expressed as elevation data stored in a Digital Elevation Model. A DEM is a collection of values representing the different elevation of the points composing a surface, so it is in fact a coverage.

WCS could seem the right solution to manage and retrieve terrain data, but:

- WCS is the least popular among the OGC web services. Just few efficient implementations of WCS are available and not so many GISes fully support it, while WMS and WFS are preferred.
- A graphical representation of a DEM, called heightmap, is enough for a 3D engine to generate the 3D representation of the corresponding piece of terrain. The additional information provided by a WCS is not needed.

A georeferenced heightmap (usually a gray-scale raster image) can be easily generated (e.g. with tools like GDAL) from the data contained in a DEM. Such a heightmap, representing a specific portion of the Earth's surface, can be easily handled and retrieved through a Web Map Service. This service allow to retrieve a specific area of a heightmap (in a light raster format like PNG) by simply defining its geographic bounding box. The retrieved heightmap chunk can be then directly converted into a 3D mesh representing the corresponding terrain tile.

The same approach can be used to retrieve textures. Any available WMS layer (or combination of layers) can be used as texture. Just two WMS calls are so sufficient for retrieving both the needed data to build a 3D terrain tile and the corresponding texture to map on it.

The described process is simple, efficient and, very important, is based on a widely adopted and implemented OGC standard. For these reasons, it is the right choice for this project.

4.4.1 Digital Elevation Models

Digital Elevation Models (DEMs) are collections of data used to represent the different elevations of surface area. In their most intuitive form they come as a two-dimensional matrix, where the rows represent latitudes and the columns longitudes (according to the coordinate system, different notations may be used). The individual cells are then filled with the elevation (usually given in meters above or below sea level) for the given latitude/longitude combination. The resolution at which the equally-spaced elevation samples are captured defines the quality of the model. Most common resolutions go from 1/6 arc-second to 1 arc-second (about 5 meters and 30 meters respectively in linear notation).

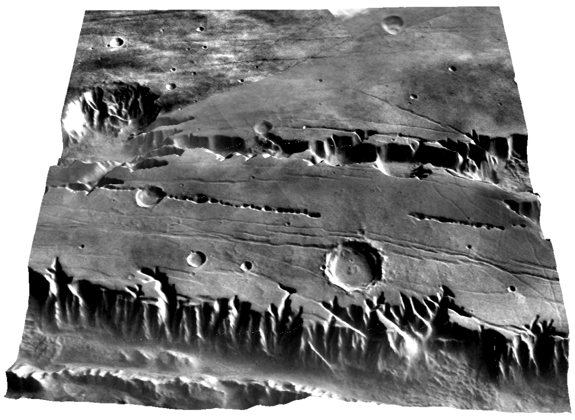


Figure 4.4: 3D rendering of a DEM of Tithonium Chasma on Mars.

Often the term “Digital Elevation Model” is confused with either the term “Digital Surface Model” (DSM) or the term “Digital Terrain Model” (DTM). In specific literature, the usage of this terms is not well defined. In most cases a DSM represents the earth’s surface and includes all objects on it. In contrast to a DSM, a DTM represents the bare ground surface without any objects like plants and buildings (see Figure 4.5). The term DEM is often used as a generic term for DSMs and DTMs, only representing height information without any further definition about the surface. As this project deals with bare ground surface, henceforth the term DTM will be used.

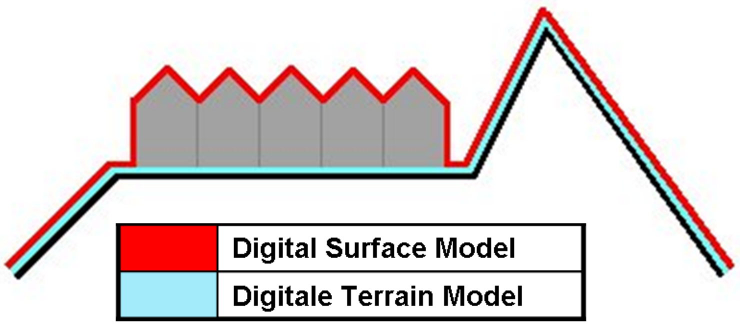
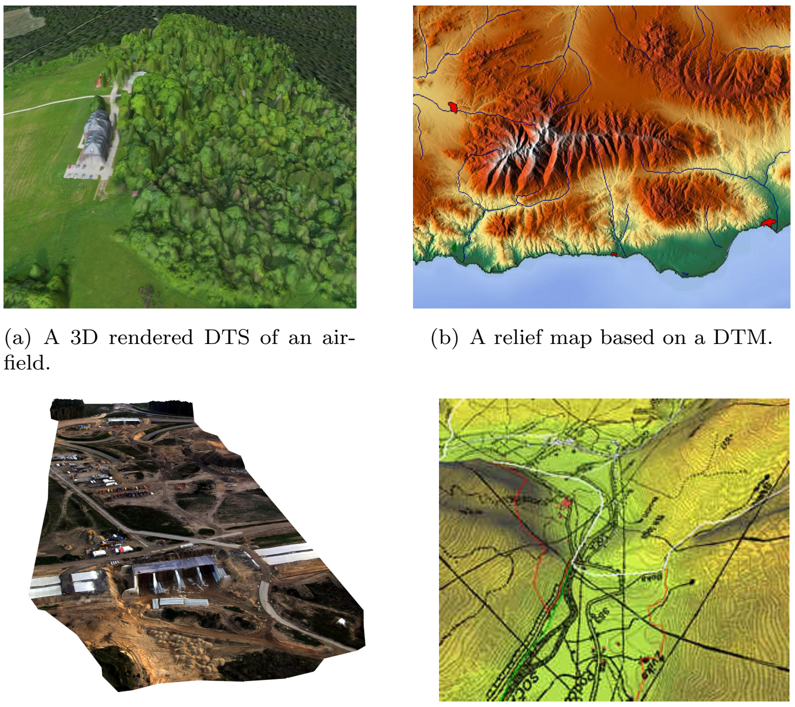


Figure 4.5: DSM vs DTM.

Digital elevation models may be produced in a number of ways. Currently, remote sensing techniques, such as interferometry, are preferred to direct surveys. DEM are widely used in many scientific, engineering and technical scopes. Common uses includes: 3D rendering, GIS, surface analysis, creation of relief maps and physical models (for example raised-relief maps), terrain analyses in geomorphology and physical geography, modelling water flow or mass movement (such as avalanches and landslides), infrastructure design and precision farming and forestry, flight simulation and 3D flight planning, archeology.



(a) A 3D rendered DTS of an airfield.

(b) A relief map based on a DTM.

(c) A 3D rendered DTS of a construction site.

(d) A topographic map combining DTM, textures and vectorial graphics.

FIGURE 4.6: Some examples about how a DEM could be used.

4.4.2 Heightmaps

In computer graphics, a heightmap is a raster image used to store elevation data for 3D rendering. A heightmap can be used as pattern for bump mapping or displacement mapping, or it can be converted into a 3D mesh to reproduce terrain.

A heightmap contains one channel interpreted as a distance of displacement or “height” from the “floor” of a surface, and sometimes visualized as a gray-scale image, with black representing minimum height and white representing maximum height. When the map is rendered, the designer can specify the amount of displacement for each unit of the height channel, which corresponds to the “contrast” of the image. Heightmaps can be stored in existing gray-scale image formats, with or without specialized metadata, or in specialized file formats such as Daylon Leveller, GenesisIV and Terragen documents.

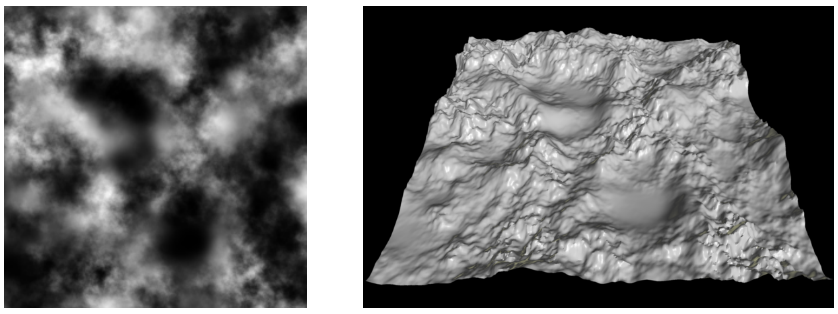

(a) Sample heightmap. (b) The same heightmap converted to a 3D mesh.

FIGURE 4.7: 3D rendering of a heightmap.

It is also possible to exploit the use of individual color channels to increase detail. For example, a standard RGB 8-bit image can only show 256 values of gray and hence only 256 heights. By using colors, a greater number of heights can be stored (for an 8-bit image, 256^3 = 16,777,216 heights can be represented (256^4 = 4,294,967,296 if the alpha channel is also used)). This technique is especially useful where height varies slightly over a large area. In this case, using only gray values, because the heights must be mapped to only 256 values, the rendered terrain may appear flat, with "steps" in certain places.

Heightmaps are an ideal way to store digital terrain elevations and are widely used in terrain rendering software and modern video games. They can be generated using real world elevation data as source (e.g. a DEM). Alternatively, they can be created by hand with a classical paint program or a special terrain editor. These editors visualize the terrain in 3D and allow the user to modify its surface. Normally there are tools to raise, lower, smooth or erode the terrain. Another way to create a heightmap is to use a random generation algorithm, such as a 2D Perlin noise function.

Chapter 5

Software design and development

5.1 Introduction

Software design is a process of problem-solving and planning for a software solution. Once the software purpose is identified and the specifications are determined, a plan for the solution has to be developed. It includes low-level components and algorithm implementation issues as well as the architectural view.

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- **Extensibility**: new capabilities can be added to the software without major changes to the underlying architecture.
- **Robustness**: the software is able to operate under stress or tolerate unpredictable or invalid input.
- **Reliability**: the software is able to perform a required function under stated conditions for a specified period of time.
- **Fault-tolerance**: the software is resistant to and able to recover from failures.
- **Security**: the software is able to withstand hostile acts and influences.
- **Maintainability**: the software can be easily maintained in order to isolate/correct defects, meet new requirements, cope with a changed environment.

- **Interoperability**: the software is able to operate with other products designed for interoperability.
- **Modularity**: the resulting software comprises well defined, independent components. Each component could be implemented and tested in isolation before being integrated to form the desired software. This allows division of work and leads to better maintainability.
- **Reusability**: the modular components designed should capture the essence of the functionality expected out of them, nothing more. This single-minded purpose make the components reusable whenever there are similar needs in other projects.

5.2 Software development process

The software development process defines the sequence of different activities that take place during software development. The software engineering defines many different software development models, each one with its own characteristics. The use of a well defined development model helps to improve the software quality.

Considering the experimental nature of this project, an agile model, based on the iterative/incremental approach, has been chosen.

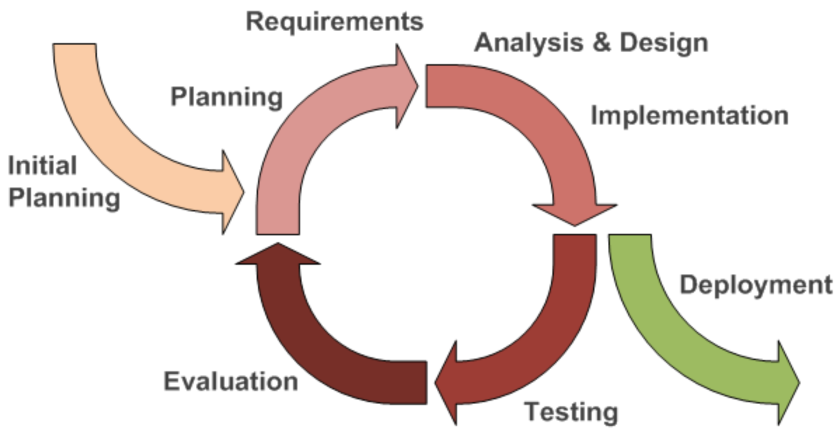


Figure 5.1: Iterative/incremental software development process.

The basic idea behind this model is to develop a software through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during development of earlier parts or versions of the software. Iterative development starts with an initial planning followed by many iterations of requirement, analysis, design, testing and implementation steps and ends with deployment. After each iteration, analysis is done to make sure that the required functionality is implemented and additional functionalities that need to be implemented are identified. Such iterations make modification and implementation easy. Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster.

In an experimental project, requirements may evolve over time and unexpected issues may arise. Thanks to its reactivity and flexibility, the agile approach is particularly suitable to handle this kind of projects. Agile processes use feedback, rather than rigorous planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software.

5.3 Requirements analysis

Requirements analysis encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product. Systematic requirements analysis is also known as *requirements engineering* and is critical to the success of a development project.

Requirements must be actionable, measurable, testable, related to identified business need or opportunities, and defined to a level of detail sufficient for system design. It is helpful to use some categorization scheme as a checklist for requirements coverage, to reduce the risk of not considering some important aspects of the system. For example, requirements may be categorized according to the FURPS+ model, a useful mnemonic which stands for:

- **F**unctional (features, capabilities, security, etc).
- **U**sability (ease of use, human factors, documentation, etc).
- **R**eliability (availability, frequency of failure, recoverability, predictability, etc).
- **P**erformance (response times, throughput, accuracy, resource usage, etc).

- **S**upportability (adaptability, maintainability, reconfigurability, etc).

The "+" in the acronym indicates auxiliary factors, such as implementation, interface, operations, packaging and legal. Some of these requirements are collectively called the quality attributes or *quality requirements.* These include usability, reliability, performance and supportability.

In common usage, requirements are simply categorized as *functional* or *non-functional.* The former defines the desired behavior of a system, specifying the required functions to be implemented. The latter impose constraints on the design or implementation, rather than specific behaviors.

A Software Requirements Specification (SRS) includes both functional and non-functional requirements, thus providing a complete description of the behavior of the system to be developed. Writing *use cases* is an excellent technique to understand, describe and document both functional and non-functional requirements, and many developers prefer them to large, monolithic documents. A use case informally defines the interactions between external actors and the system under consideration. An actor specifies a role played by a person or thing when interacting with the system. Use cases treat the system as a black box, and every interaction with it, including system responses, are perceived as from an external point of view. This is a deliberate policy, because it forces the developer to focus on what the system must do, not how it has to be done, and avoids the trap of making assumptions about how the functionality will be accomplished. A use case should:

- Describe what the system shall do for the actor to achieve a particular goal.
- Include no implementation-specific language.
- Be at the appropriate level of detail.
- Not include detail regarding user interfaces and screens.

Following is the SRS of the system in analysis. Both functional and non-functional requirements are documented as use case templates and diagram. Besides the core functionality of the streaming framework some basic features usually provided by a standard 3D viewer are considered. These features are completely independent from the framework logic and have been included just to better contextualize the analysis.

5.3.1 Functional requirements use cases

USE CASE: run on multi-platform	
Summary	The system should be able to run on multiple platforms and environments
Priority	Desired
Use frequency	Always
Actors	User
Stakeholders	
Prerequisites	Different target platforms available
Main scenario	Run on multi-platform
Scenario extensions	
Notes	

Table 5.1: USE CASE: run on multi-platform

USE CASE: view the scene	
Summary	The system should be able to display a 3D scene incrementally built and dynamically updated at run-time
Priority	Essential
Use frequency	Always
Actors	User
Stakeholders	Scene, scene elements
Prerequisites	Data loaded and processed
Main scenario	Display a 3D scene incrementally built and dynamically updated at run-time
Scenario extensions	
Notes	

Table 5.2: USE CASE: view the scene

USE CASE: update the scene content	
Summary	The system should be able to update the content of the scene according to the current Field Of View (FOV)
Priority	Essential
Use frequency	Very Often
Actors	User
Stakeholders	Scene, scene elements
Prerequisites	The FOV has changed
Main scenario	Update the content of the scene according to the current FOV
Scenario extensions	
Notes	

Table 5.3: USE CASE: update the scene content

USE CASE: load terrain	
Summary	The system should be able to load the proper chunk of a Digita Terrain Model (DTM) when required
Priority	Essential
Use frequency	Often
Actors	User
Stakeholders	DTM, data source
Prerequisites	Missing DTM chunk required; required DTM chunk available
Main scenario	Load the proper chunk of DTM when required
Scenario extensions	
Notes	The loading process is triggered by the observer's movements: when a “void” region (not yet loaded) is approached, the corresponding missing data are loaded

Table 5.4: USE CASE: load terrain

USE CASE: load features	
Summary	The system should be able to load the proper features when required
Priority	Essential
Use frequency	Often
Actors	User
Stakeholders	Features, data source
Prerequisites	Missing features required; required features available
Main scenario	Load the proper features when required
Scenario extensions	
Notes	The loading process is triggered by the observer's movements: when a “void” region (not yet loaded) is approached, the corresponding missing data are loaded; features may include buildings, vegetation objects, city furniture, etc

Table 5.5: USE CASE: load features

USE CASE: navigate through the scene	
Summary	The user should be able to move through the scene
Priority	Essential
Use frequency	Very Often
Actors	User
Stakeholders	Camera, scene
Prerequisites	Scene rendered
Main scenario	The user moves through the scene
Scenario extensions	
Notes	Basic movements such as pan, zoom and rotate should be allowed

Table 5.6: USE CASE: navigate through the scene

USE CASE: pick a scene element	
Summary	The user should be able to pick a scene element by clicking on it
Priority	Optional
Use frequency	Sometimes
Actors	User
Stakeholders	Scene element
Prerequisites	Scene rendered
Main scenario	The user pick a scene element by clicking on it
Scenario extensions	Pick by search
Notes	

Table 5.7: USE CASE: pick a scene element

USE CASE: get element information	
Summary	The system should be able to show information about the picked element
Priority	Optional
Use frequency	Sometimes
Actors	User
Stakeholders	Picked element
Prerequisites	Scene rendered and picked element
Main scenario	Display information about the picked element
Scenario extensions	
Notes	

Table 5.8: USE CASE: get element information

USE CASE: get scene information	
Summary	The system should be able to show information about the scene
Priority	Optional
Use frequency	Sometimes
Actors	User
Stakeholders	Scene
Prerequisites	Scene rendered
Main scenario	Display scene information
Scenario extensions	
Notes	

Table 5.9: USE CASE: get scene information

5.3.2 Non-functional requirements use cases

USE CASE: run inside a web browser	
Summary	The system should be able to run inside a web browser without installing any specific client software
Priority	Desired
Use frequency	Always
Actors	User
Stakeholders	
Prerequisites	WebGL-compliant web browser
Main scenario	The system runs inside a web browser
Scenario extensions	
Notes	

Table 5.10: USE CASE: run inside a web browser

USE CASE: use hardware acceleration	
Summary	The system should be able to use hardware acceleration
Priority	Essential
Use frequency	Always
Actors	User
Stakeholders	Graphics hardware
Prerequisites	Hardware acceleration available
Main scenario	Use hardware acceleration
Scenario extensions	
Notes	

Table 5.11: USE CASE: use hardware acceleration

USE CASE: load terrain via WMS	
Summary	The system should be able to load chunks of a DTM via Web Map Service (WMS)
Priority	Essential
Use frequency	Often
Actors	User
Stakeholders	DTM, geospatial server
Prerequisites	DTM available via WMS
Main scenario	Load chunks of DTM via WMS
Scenario extensions	
Notes	

Table 5.12: USE CASE: load terrain via WMS

USE CASE: load features via WFS	
Summary	The system should be able to load features via Web Feature Service (WFS)
Priority	Essential
Use frequency	Often
Actors	User
Stakeholders	Features, geospatial server
Prerequisites	Features available via WFS
Main scenario	Load features via WFS
Scenario extensions	
Notes	

Table 5.13: USE CASE: load features via WFS

USE CASE: load material textures	
Summary	The system should be able to load and map textures on terrain and features
Priority	Desired
Use frequency	Often
Actors	User
Stakeholders	Terrain, features
Prerequisites	Textures available, 3D models loaded
Main scenario	Load and map textures on terrain and features
Scenario extensions	
Notes	

Table 5.14: USE CASE: load material textures

USE CASE: generate default materials	
Summary	The system should be able to generate default materials, for both terrain and features, in case of missing textures/material informations
Priority	Desired
Use frequency	Often
Actors	User
Stakeholders	Terrain, features
Prerequisites	Missing textures
Main scenario	Create and apply default materials
Scenario extensions	
Notes	

Table 5.15: USE CASE: generate default materials

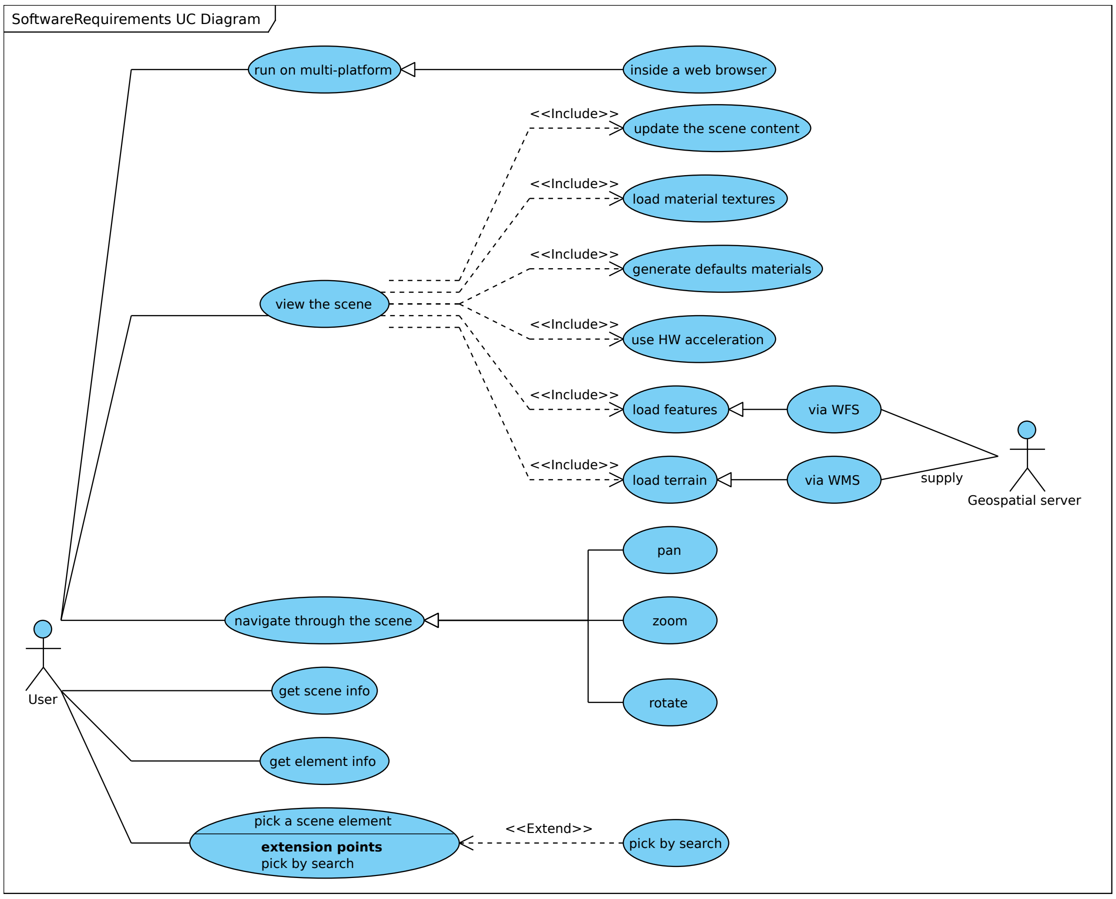


FIGURE 5.2: Use case diagram for the Software Requirements Specification.

5.4 Software design

The purpose of the design phase is to define the structure of the software. The model constructed in the software requirements phase is transformed into the architectural design by allocating functions to software components and defining the control and data flow between them. This phase may involve several iterations of the design. Technically difficult or critical parts of the design have to be identified. Prototyping of these parts of the software may be necessary to confirm the basic design assumptions.

As introduced in Section 2.1, the framework implementation focuses on textured terrain, while streaming of other geospatial features (buildings, vegetation, city furniture, etc) is left to future developments (cf. Section 6.2). Sometimes, dealing with just a limited part of the problem helps to better understand the design steps needed to solve it, and this is the case. Despite the scope limitation, the designed streaming paradigm has general validity, and can be applied to more complex environments without substantial modifications to its logic.

Following is a complete description of the system design. In order to better understand it, Section 5.4.1 provides a brief introduction to the Three.js architecture and to some basic 3D rendering concepts. The designed streaming logic is described in Section 5.4.2, while Section 5.4.3 describes how this logic has been traduced into software components.

5.4.1 Three.js architecture

Three.js is an open source, cross-browser WebGL framework released under MIT License. It aims to create a lightweight JavaScript 3D engine with a very low level of complexity. The engine can render 3D graphics using WebGL, the canvas 2D context or SVG and provides an interface that completely abstracts the underlying rendering mechanisms, so that the same code can be ported on multiple renderers.

Three.js is a scene-graph based engine built on three basic elements:

- A **Scene**, containing a hierarchical organized set of 3D objects (that can be added, removed or updated run-time).
- A **Camera**, used by the renderer to set the viewable area.

- A **Renderer** that, receiving a scene and a camera as input, renders the scene using WebGL (or canvas 2D context, or SVG; henceforth just WebGL will be considered).

Scene-graph

The scene-graph is a hierarchical data structure used to group 3D objects. It is specifically a tree consisting of parent nodes containing any number of child nodes and child nodes containing one parent node. The root node, that is the scene object, has no parent (see Figure 5.3). Each non-root node could be a generic 3D object, a mesh, a line, a LOD object, a particle, a particle system, a camera, a light, etc (see Figure 5.4).

Using a scene-graph in a 3D engine has multiple benefits:

- Allows to efficiently handle complex 3D scenes.
- Allows to group objects (e.g. group semantically or spatially related objects).
- Simplifies global attributes management (e.g. setting the position of an entire subtree, instead of setting the position of each object in it).
- Aids in persisting the state of the scene, or part of it (e.g. an entire subtree can be backed up by saving a reference to its root node).

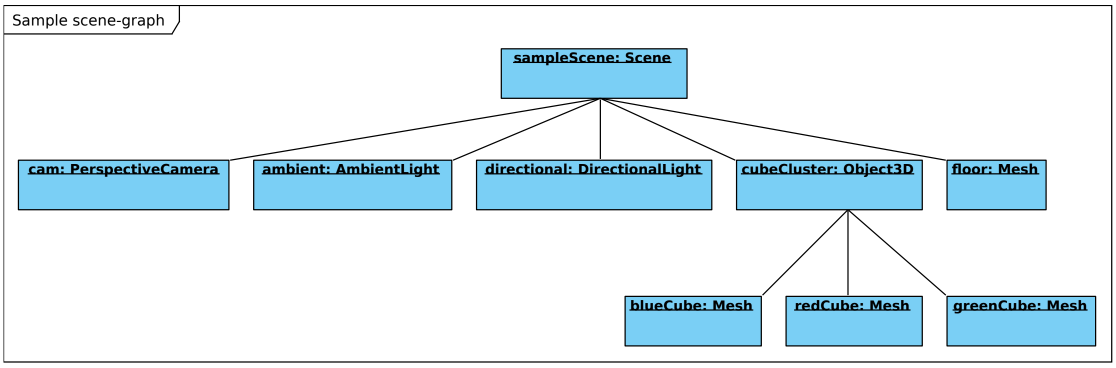


Figure 5.3: Object Diagram representing a sample scene-graph.

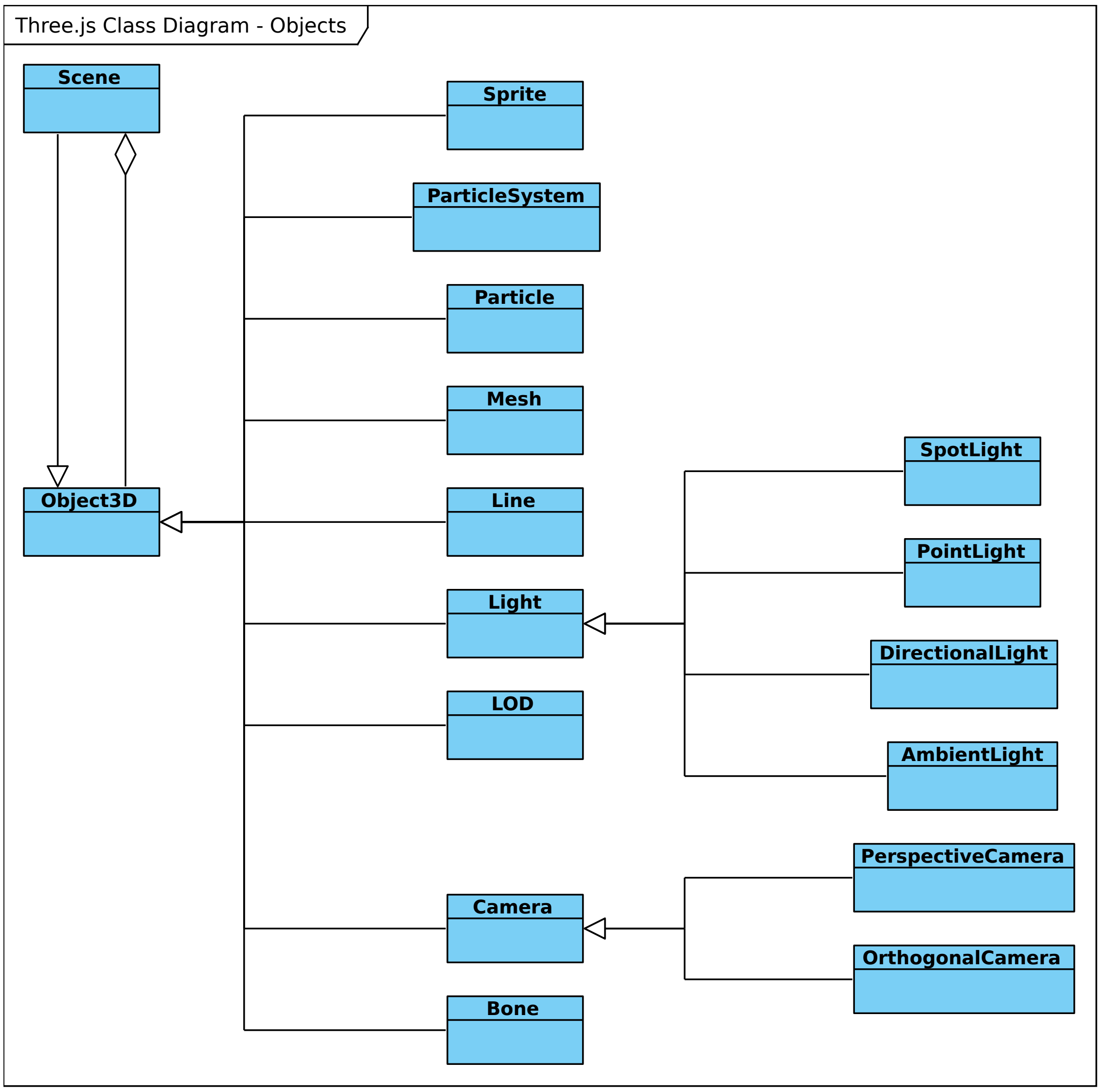


FIGURE 5.4: Simplified Class Diagram representing Three.js objects.

Camera

A camera model is used by the renderer to project the 3D scene to the screen. The placement, orientation and settings (FOV, near and far distance, aspect ratio) of the camera define a *view frustum*. In 3D computer graphics, the view frustum is the region of space in the modeled world that may appear on the screen (it is the field of view of the notional camera). The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid (hence the name). The planes that cut the frustum perpendicular to the viewing direction are called the *near plane* and the *far plane*. Objects closer to the camera than the near plane or beyond the far plane are not drawn. Sometimes, the far plane is placed infinitely far away from the camera so all objects within the frustum are drawn regardless of their distance from the camera.

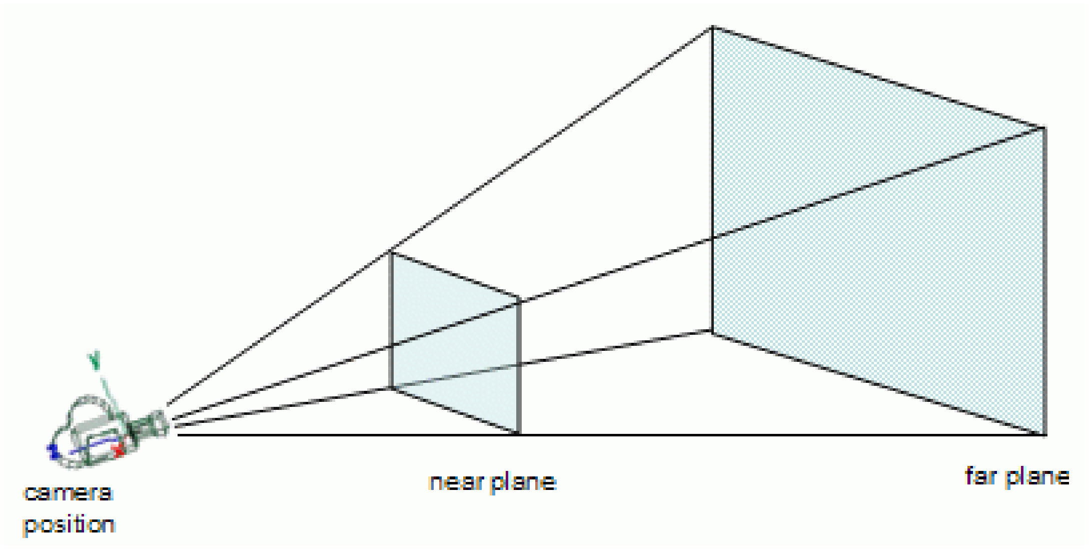


FIGURE 5.5: View frustum representation.

The view frustum is used by the renderer to perform two essential processes: *culling* and *clipping*. View frustum culling is the process of removing objects that lie completely outside the viewing frustum from the rendering process. View frustum clipping is the process of clipping to the view frustum objects that lie partially outside the viewing frustum. Rendering these objects, or parts of objects would be a waste of time since they are not directly visible. To make culling fast, it is done using bounding volumes surrounding the objects rather than the objects themselves.

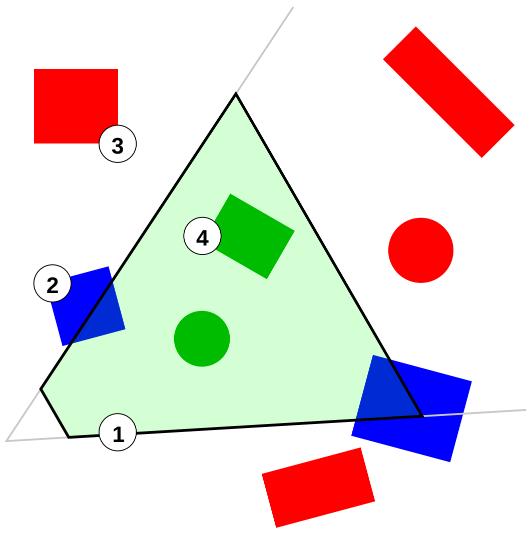


FIGURE 5.6: View frustum culling and clipping: 1) view frustum, 2) clipped objects, 3) culled objects, 4) completely visible objects.

WebGL Renderer

The WebGL renderer is the component that actually accesses the WebGL capabilities and renders the given scene-graph through the given camera. Its main responsibilities include:

- Transforming scene data from world space to screen space.
- Performing view frustum culling and clipping processes.
- Drawing the transformed scene to the screen.

5.4.2 Streaming logic

The system has to load and render a large-scale DTM stored as a georeferenced gray-scale heightmap and accessible via WMS (cf. Section 4.4). In order to efficiently accomplish this task, an incremental approach is applied. Instead of loading the entire DTM at once, the system load it chunk by chunk following the observer's movements. The system also dynamically update the scene so that it contains, at each time, only the terrain chunks lying inside the field of view or near it. Both the loading and the updating processes are executed run-time without interfering with the rendering process. This behavior, graphically summarized by the diagram in Figure 5.7, is mainly based on *tiling* and *caching*.

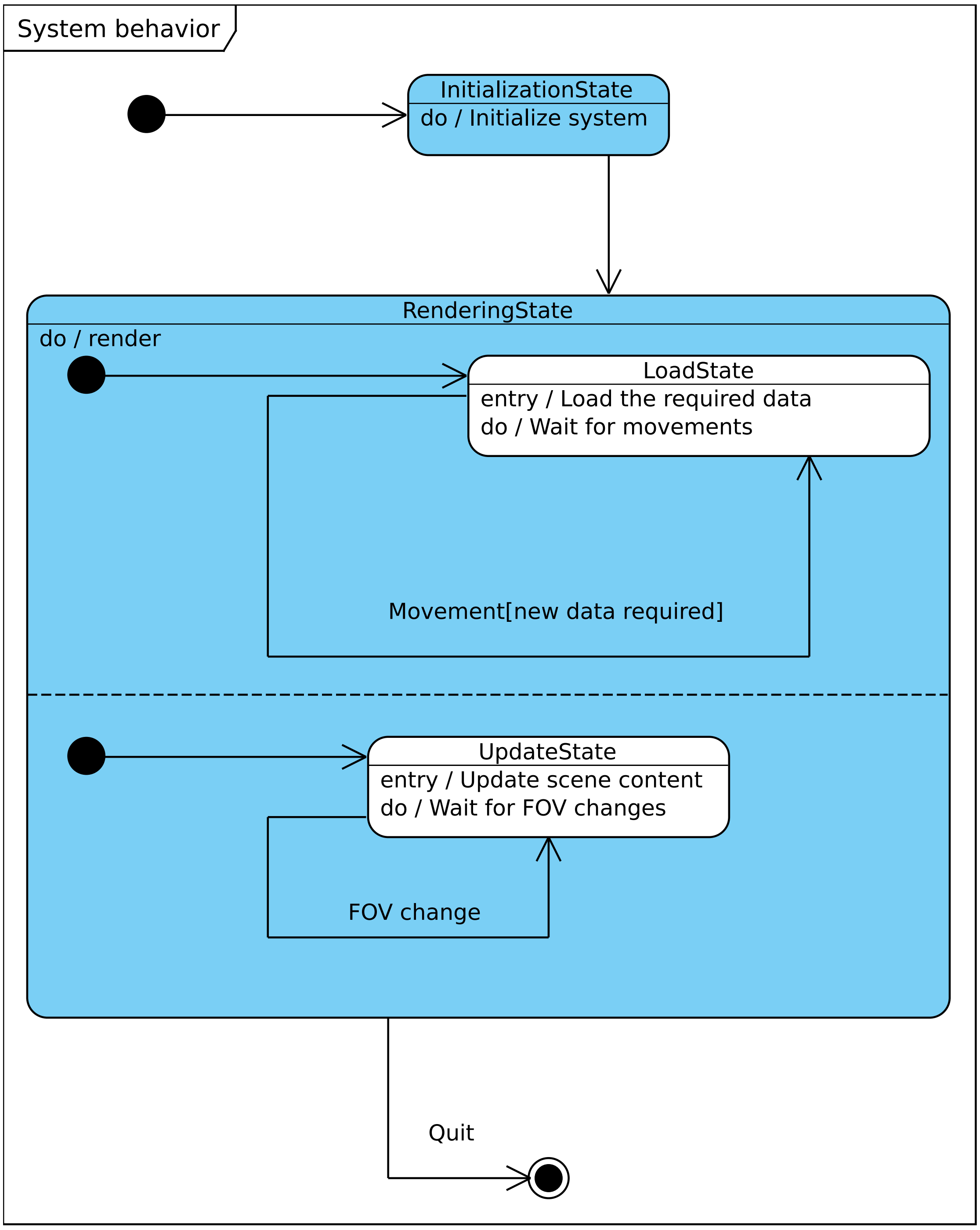
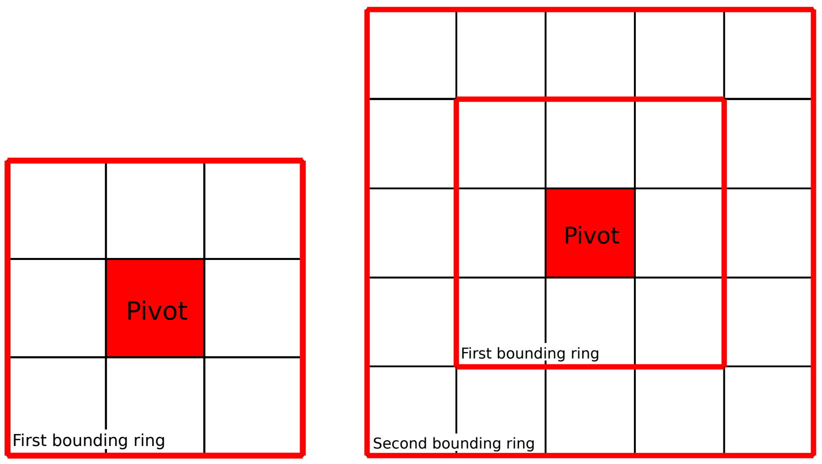


Figure 5.7: State Diagram describing how the system behaves.

Tiling

What has been roughly called "chunks" are regularly displaced square tiles of fixed size. Following is the description of how the tiling process works.

Let S be the starting camera position. At startup, the engine loads a *cluster* of tiles consisting of the tile centered on S, which we call *pivot*, plus a fixed number (at least one) of *bounding rings* (see Figure 5.8).

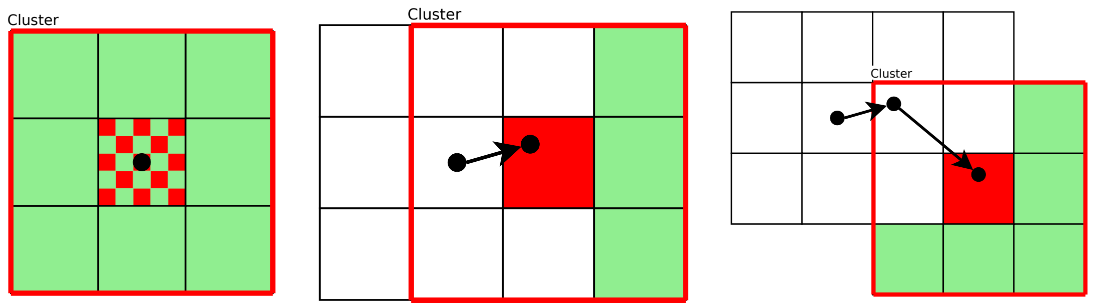


(a) Cluster multiplier set to 1 (default): one bounding ring.

(b) Cluster multiplier set to 2: two bounding rings.

FIGURE 5.8: The size of a cluster is defined by the *cluster multiplier* (which can be set according to the needs).

As soon as the camera crosses the pivot's boundary, the just reached tile become the new pivot and the cluster centered on it is considered. Once the new tiles have been loaded, the process start over (see Figure 5.9).



(a) Startup.

(b) First move.

(c) Second move.

FIGURE 5.9: The loading process (pivot and cluster in red, loaded tiles in green).

Considering the regular shape of the tiles and their regular distribution, checking the position of the camera with respect to the pivot and placing the new tiles in the correct position are quite simple tasks. Let l be the tile's edge (in 3D units), $P = (X, Z)$ the pivot position and $p = (x, z)$ the camera position. The camera is inside the pivot if and only if:

$$(X - \frac{l}{2} \leq x \leq X + \frac{l}{2}) \wedge (Z - \frac{l}{2} \leq z \leq Z + \frac{l}{2}) \tag{5.1}$$

The other possible cases define the relative position of the camera with respect to the pivot. The positions of the tiles are computed by simply adding the proper positive or negative offset ($\pm n \cdot l$) to the pivot's position (see Figure 5.10).

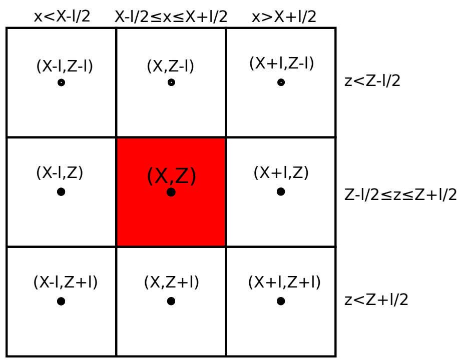


FIGURE 5.10: Relative positions with respect to the pivot.

For each tile a two-values index is assigned: *(I,J)*. This index denotes the tile's position inside an ideal grid. The first loaded tile has index (0,0), other indices are assigned accordingly (see Figure 5.11). The index is used as key in the caching process and to compute the geographic bounding box (bbox) of the tile.

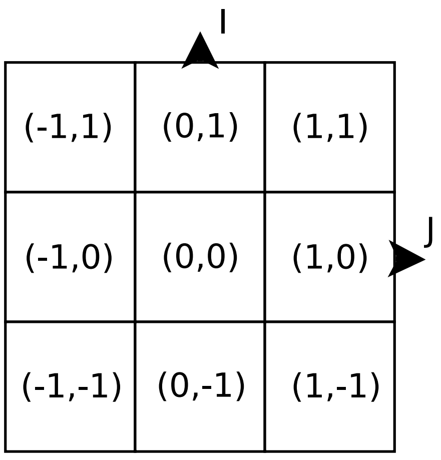


FIGURE 5.11: Tiles indexing.

Georeferencing

In order to keep a perfect one to one correspondence between the 3D environment and the real world, the geographic bbox of each tile is computed with an analogous method. Given the geographic starting point chosen by the user *O=(X,Y)* and the tile's edge l the bbox of the tile centered on it is computed according to:

$$\begin{aligned} minX &= X - \frac{l}{2}, \\ maxX &= X + \frac{l}{2}, \\ minY &= Y - \frac{l}{2}, \\ maxY &= Y + \frac{l}{2} \end{aligned} \tag{5.2}$$

This bbox is used as reference: other bboxes are obtained by simply adding the proper offsets. For example, a tile with index *(I,J)* has bbox:

$$\begin{aligned} minX &= O.minX + lI, \\ maxX &= O.maxX + lI, \\ minY &= O.minY + lJ, \\ maxY &= O.maxY + lJ \end{aligned} \tag{5.3}$$

Note that, in this case, l represents the real-world dimension of the tile, so it is expressed in meters (or angular units, according to the geographical coordinate systems). This value is fixed and is obtained by multiplying the tile dimension expressed in pixels for the heightmap resolution. For example, if we consider 256x256 px tiles of a heightmap with resolution 2.5 m, the edge of each tile is 640 m long.

Tiles creation

When a tile is loaded, its geographic bbox is used to get, via WMS, the corresponding heightmap chunk. The obtained raster image is drawn on a *canvas* element using the *drawImage* method. In order to limit the number of vertices composing each mesh, the image is scaled to a fixed size (default: 256x256px to 32x32px). The elevation data stored in the heightmap are retrieved using the *getImageData* method (both *getImageData* and *drawImage* are provided by the 2D context, cf. Sections 3.2.3, 4.4.2). The retrieved data, properly adjusted to respect the proportions, are used to set the right altitude of each point composing the 3D mesh which represents the tile. Another WMS call is needed to retrieve the texture to map on the 3D mesh (any available layer, or combination of layer can be used). Once the texture is applied, the tile can be rendered.

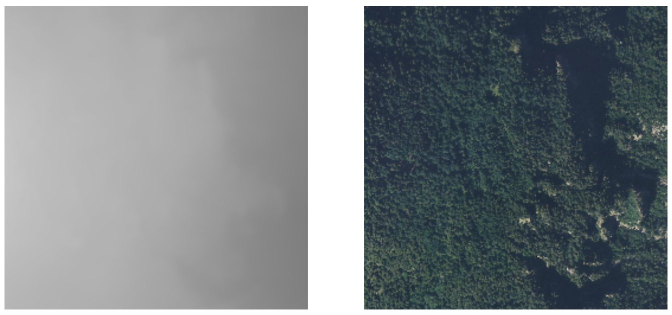

(a) Heightmap. (b) Texture (orthophoto).

FIGURE 5.12: An heightmap and the corresponding texture.

The described approach based on tiling has a drawback. Very often, adjacent tiles do not match perfectly. The reason is simple: the corresponding points on the shared edge have different elevation values. To (partially) solve this problem, the following strategy has been employed.

For each tile we load not only its heightmap (*core*), but also the heightmaps corresponding to the adjacent tiles (*support*). In order to build the 3D mesh we consider the interior points of the core heightmap and the proper sides of the support heightmaps (see Figure 5.13). Exploiting this trick, it is possible to match adjacent tiles in exchange for a negligible loss of information. This strategy may seem quite onerous. However the additional heightmaps are cached by the browser, so they are preserved for future uses.

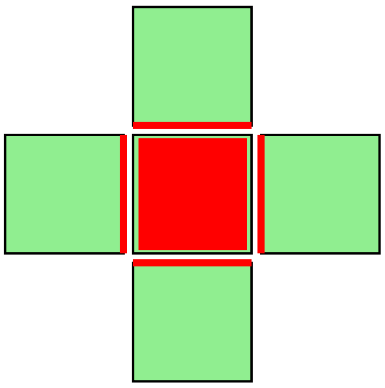


FIGURE 5.13: A trick to match tiles (loaded heightmaps in green, considered data in red).

Some (rare) mismatches may still arise between corners, so this strategy needs to be improved (cf.Section 6.2).

Scene updating and caching

As mentioned, the scene is constantly updated so that it contains, at each time, only a limited number of tiles (those lying inside the view frustum or near it). The logic underlying this process is following described.

Whenever the camera changes its target, the view frustum is computed and projected on the horizontal plane. The projection of each vertex falls on a specific tile with index (I_k, J_k), where k identifies the vertex. The set of tiles to be placed into the scene-graph includes all the those with index (I, J) such that:

$$min_k\{(I_k, J_k)\} \leq (I, J) \leq max_k\{(I_k, J_k)\}$$

Figure 5.14 graphically describes this behavior.

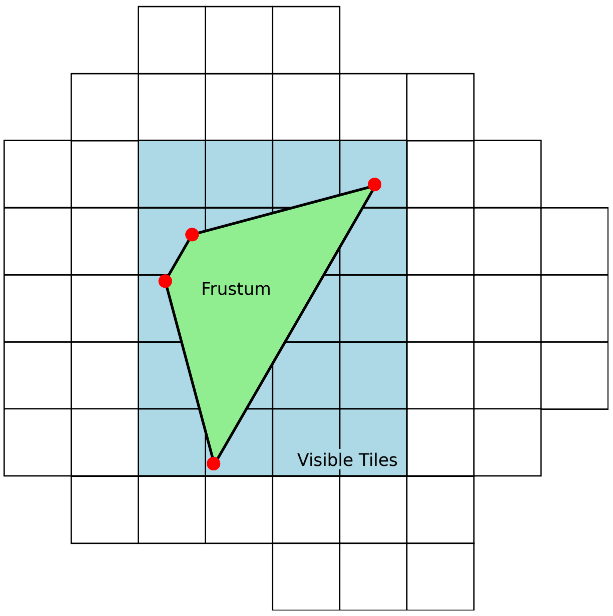


Figure 5.14: Tiles to be placed into the scene-graph (light green) according to the current view frustum (green)

Once such a set is defined, the included tiles not yet in scene are added to it, while the not included tiles still in scene are removed.

When a tile is removed, it is not destroyed but stored into a cache (using its index as key). Therefore, the updating process simply moves a reference to the tile between the scene-graph and the cache.

The cache can contain just a limited number of tiles. Once the limit is reached, the cache is purged according to a policy based on creation time and proximity to the current camera position.

5.4.3 System architecture

The system architecture is quite simple. The described logic is implemented by a limited number of well defined software components. Each component is responsible for a particular step of the streaming process previously described.

The source code is organized in modules, according to the structure proposed by Dojo Toolkit (cf. Section 3.4.1). Following are the defined modules accompanied by a brief description of their roles:

- **view.Main**. Responsible for the system initialization, it sets up the rendering context (builds the renderer and attach it to a *canvas* element) and the 3D environment (initializes scene, camera, controls, lights, sky-box, etc), starts and handles the rendering loop and interacts with the underlying streaming engine.
- **view.shared**. Stores some basic configuration values (tile edge, FOV, near and far distance, etc) which are shared by each module.
- **stream.Engine**. This is the core component, responsible for the tiling and the caching processes described in the previous section. It checks the camera position and its field of view, triggering a new load or update process when needed.
- **stream.TileManager**. Handles the tiles creation process and manage the cache.
- **model.Tile**. Defines the *Tile* object and the interface needed to handle it (georeferencing, 3D model building, texturing).
- **utils.Frustum**. Provides utility functions used by the Engine to compute the current view frustum and define which tiles to place into the scene-graph.

- **utils.wms**. Provides utility functions for building WMS request URLs.
- **utils.Cache**. Defines the *Cache* object (implemented as an associative array) and the interface needed to manage it.

5.5 Development work-flow

The development process has been engineered to keep track of code changes and bugs. This allows a better control of the project life cycle and permits easy backup and recovery in case of data loss or bad code. The aim of this strategy is to be proactive and not reactive while concentrating on code quality to minimize rework.

Development tools are an essential part of this process, following are the ones used in this project:

- **Integrated Development Environment** (IDE): *Eclipse.* Eclipse is an open source IDE maintained by the Eclipse Foundation and released under the Eclipse Public License (EPL). It is written primarily in Java and in its default form it is meant for Java developers. Users can extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules.
- **Version Control System** (VCS): *Git.* Git is a distributed VCS with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Git's current software maintenance is overseen by Junio Hamano. Git is a free software distributed under the terms of the GNU General Public License (GPL) version 2.

The main goals of the defined work-flow are the following:

- Maintain a local copy of the project up to date with the latest version in the repository.
- Keep the repository up to date with latest code changes.

In order to achieve these goals, the following tasks have been followed on regular basis:

- Update local copy with the latest version in the repository.
- Make changes to the local copy.
- Publish changes.

5.5.1 Documentation

As previously mentioned, this is an experimental project designed to be further developed, possibly by different people. For this reason, comprehensibility is a crucial aspect. In order to support further development and code analysis, a simple, but exhaustive documentation has been produced. Moreover, the code has been organized according to widely understandable patterns and many comments have been provided to improve the comprehensibility.

Chapter 6

Results and future developments

6.1 Development results

The result of the development process is a fully functional 3D viewer with basic navigation features. This viewer implements the designed streaming strategy with excellent results in terms of performances (50-60 fps during navigation with occasional drops to 30-40 fps while loading) and memory usage (20-30 MB).

6.1.1 Testing data

The viewer streams a DTM (resolution: 2.5 m) covering an area of about 72 Km^2 around Caldaro Sulla Strada del Vino (Bz, Italy). The native ArcINFO Ascii Grid files has been converted to a GeoTIFF grey-scale heightmap to be stored on GeoServer and retrieved via WMS. The 3D model has been textured using different available WMS layers (default: orthophoto).

6.1.2 Testing platform

- **CPU**: Intel Core 2 Quad Q9450 2.66 GHz.
- **GPU**: Amd Radeon HD5870.
- **System memory**: 8 GB DDR2 800 MHz.
- **Video memory**: 1 GB GDDR5 4800 MHz

- **OS**: Microsoft Windows (8), Apple Mas OS X (Snow Lion), GNU Linux (Ubuntu 12.04).
- **Web Browser**: Mozilla Firefox (16.0.1), Google Chrome (22.0.1229.94).

6.1.3 Screenshots

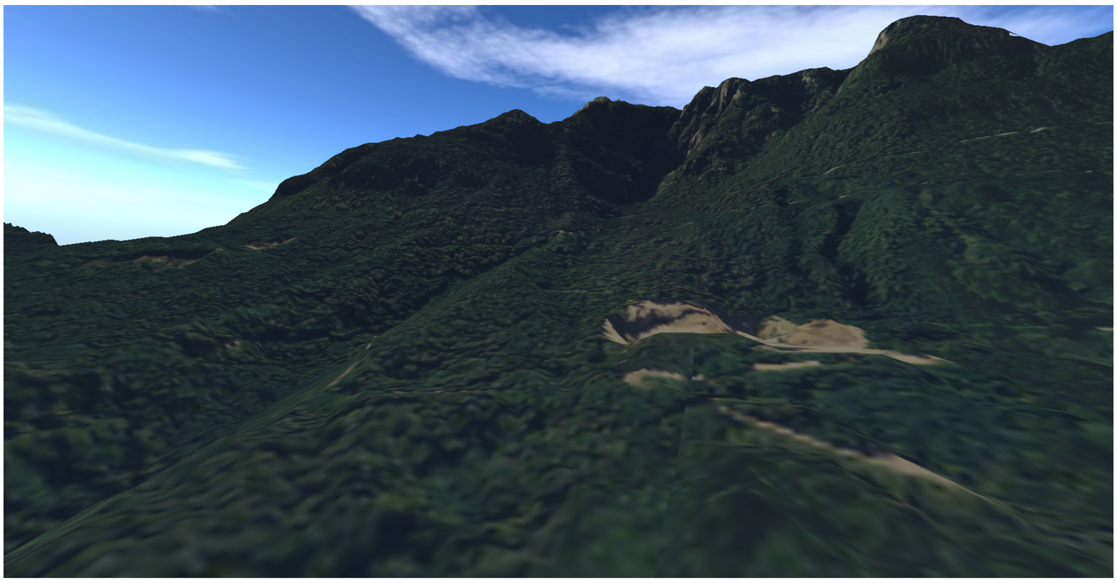


Figure 6.1: Screenshot - 1.

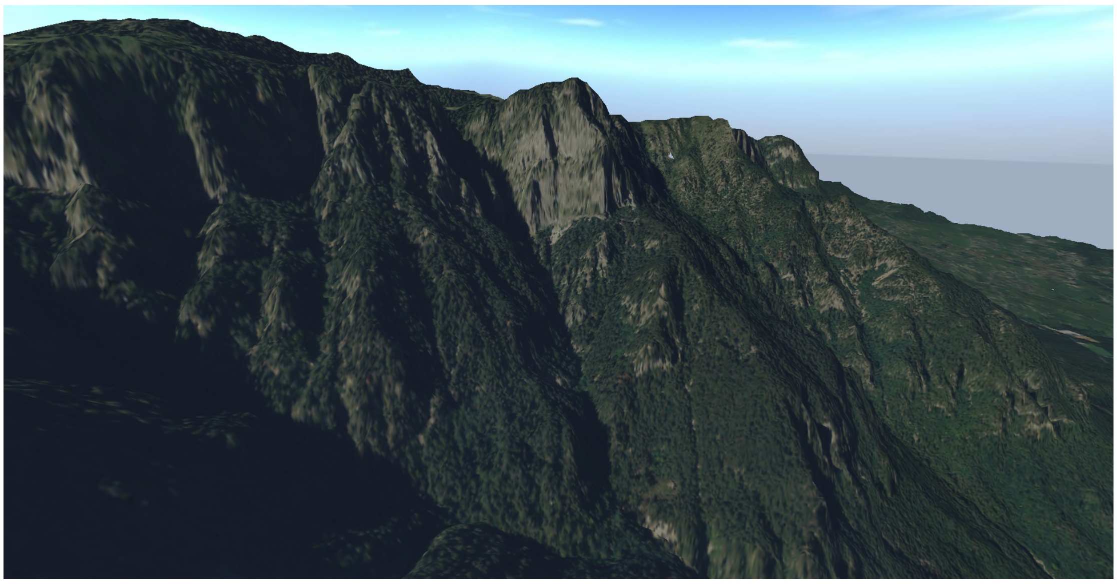


Figure 6.2: Screenshot - 2.

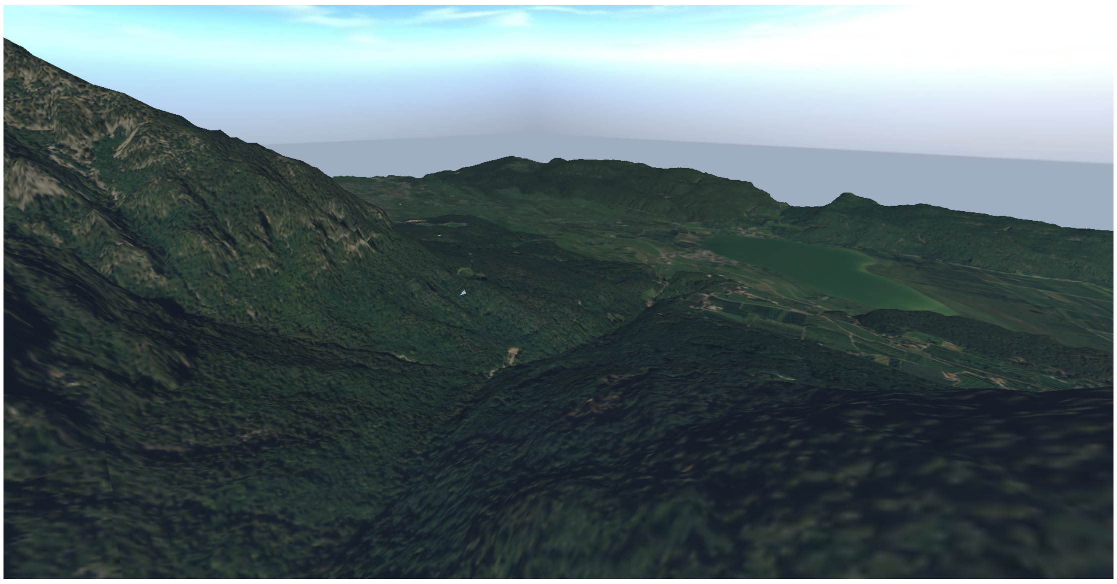


Figure 6.3: Screenshot - 3.

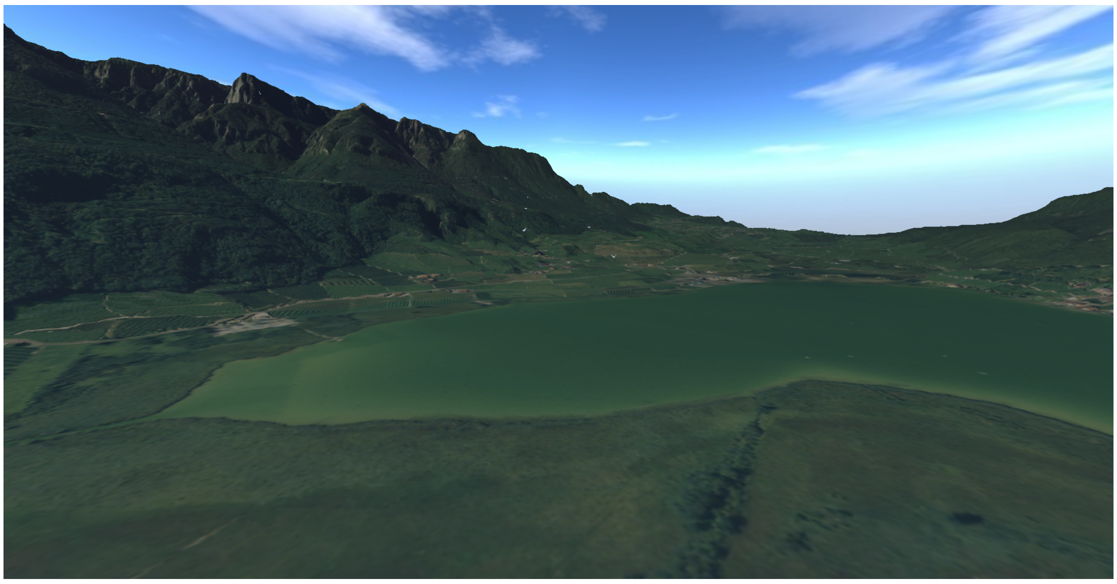


Figure 6.4: Screenshot - 4.

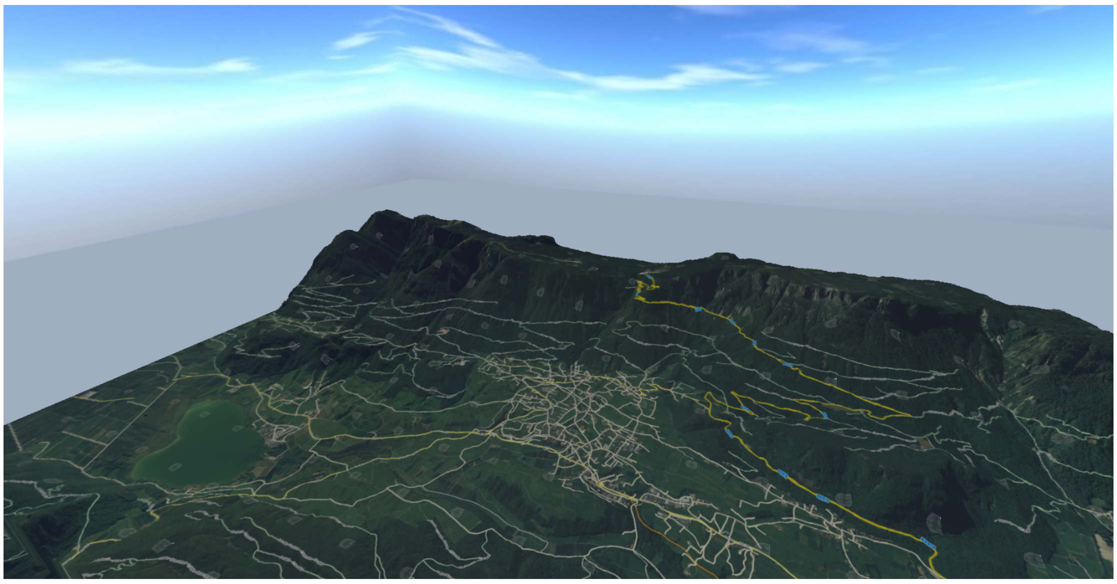


FIGURE 6.5: Screenshot - 5.

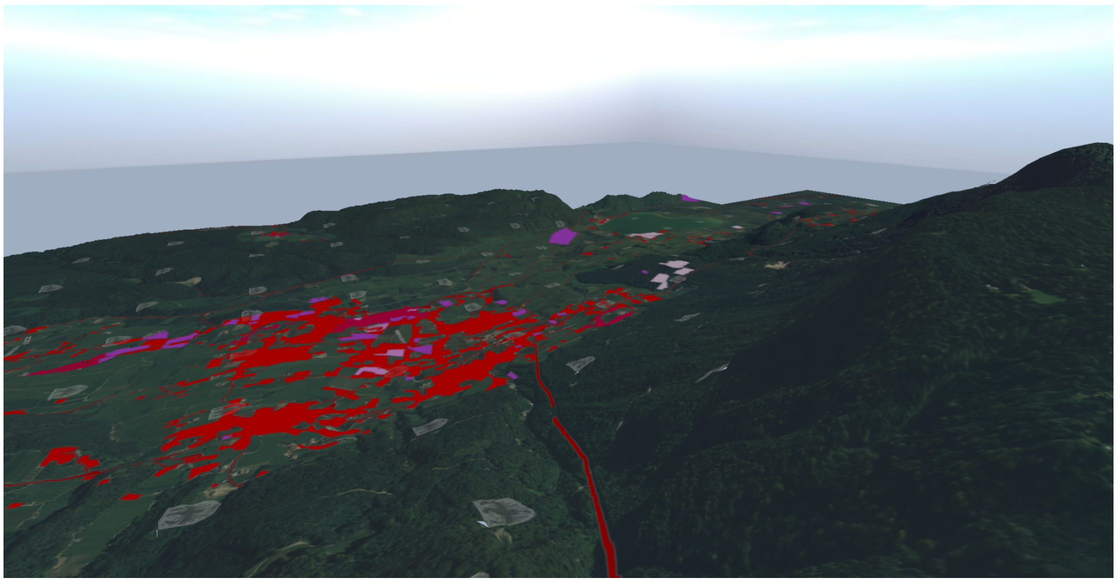


FIGURE 6.6: Screenshot - 6.

6.2 Future developments

The developed software is a fully functional base on which develop a complete streaming system for three dimensional geospatial data. Future developments include:

- **Improvements**:
  - Tiles matching system refinement.
  - Heightmaps processing through shaders.
  - Navigation system refinement.
  - Improved lights and shadows.
- **New features**:
  - Level Of Detail (LOD) system.
  - Streaming of CityGML-encoded geospatial features retrieved via WFS (Web Feature Service).

Bibliography

[1] Web Hypertext Application Technology Working Group (WHATWG). HTML living standard, September 2012. URL http://whatwg.org/html.

[2] World Wide Web Consortium (W3C). HTML5 working draft, March 2012. URL http://www.w3.org/html5.

[3] Khronos Group. WebGL 1.0 specification, February 2011. URL https://www.khronos.org/registry/webgl/specs/1.0/.

[4] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL ES 2.0 Programming Guide*. Addison-Wesley, 2008. ISBN 978-0321502797.

[5] jQuery website. URL http://jquery.com.

[6] Dojo Toolkit website. URL http://dojotoolkit.org.

[7] C3DL website. URL http://www.c3dl.org.

[8] SpiderGL website. URL http://spidergl.org.

[9] Three.js website. URL http://threejs.org.

[10] Open Geospatial Consortium (OGC) website. URL http://threejs.org.

[11] Open Geospatial Consortium (OGC). KML Standard Specification, April 2008. URL http://portal.opengeospatial.org/files/?artifact_id=27810.

[12] Open Geospatial Consortium (OGC). CityGML Standard Specification, April 2012. URL https://portal.opengeospatial.org/files/?artifact_id=47842.

[13] Open Geospatial Consortium (OGC). WFS Standard Specification, May 2005. URL http://portal.opengeospatial.org/files/?artifact_id=8339.

[14] Open Geospatial Consortium (OGC). WMS Standard Specification, March 2006. URL `http://portal.opengeospatial.org/files/?artifact_id=14416`.

[15] Open Geospatial Consortium (OGC). WCS Standard Specification, March 2008. URL `http://portal.opengeospatial.org/files/?artifact_id=27297`.

[16] Open Geospatial Consortium (OGC). GML 3.2.1 Application Schema - Coverages (GMLCOV), May 2012. URL `https://portal.opengeospatial.org/files/?artifact_id=48553`.

Acknowledgements

I would like to thank everyone assisted (and beared) me during my research, especially my family and my friends in 3DGIS.