Universidade do Minho

University of Minho
Informatics Department

Master in Informatics

OL3 - JavaScript library for 3D scenes

Bruno Gustavo Rego Amaral da Costa

Supervised by:

Professor Doutor Jorge Gustavo Rocha

Braga, October 30, 2012

Abstract

The main purpose of this thesis is the implementation of an open source JavaScript library for the navigation of dynamic three-dimensional scenes through the browser, using asynchronous data from different servers and without the need for plugins.

Due to the recent developments of Hypertext Markup Language (HTML)5 and the new capabilities of JavaScript for supporting 3D graphics using Web Graphics Library (WebGL), users can now break free from the traditional two dimensional web, and experience three-dimensional scenes through the browser. Using these new specifications a library for the visualisation and navigation of three-dimensional scenes is a step towards a more immersive experience of the web. The library should be capable of loading geometry data from servers asynchronously and provide tools for a free and immersive navigation of the scenes obtained from the server.

The library was developed and most functionalities were implemented, such as asynchronous loading of data and navigation. Data management is partly implemented, as there are still some issues regarding the overlapping of geometry, but the issue is well defined and a solution is nearly implemented.

Overall, the library has it's strong points and issues, yet it is almost ready to be usable. Further work is still required and additional possibilities and functions can be studied and implemented.

Resumo

O objetivo desta tese é a implementação de uma biblioteca JavaScript open source para a navegação de cenas tridimensionais dinâmicas através do *browser*, utilizando dados assíncronos de diferentes servidores e sem a necessidade de *plugins*.

Tendo em conta os recentes desenvolvimentos do HTML5 e as novas capacidades do JavaScript para suportar gráficos 3D recorrendo ao WebGL, os utilizadores podem agora libertar-se da barreira bidimensional da *web* tradicional, e experiênciar cenas tridimensionais através do *browser*. Utilizando estas novas especificações, uma biblioteca para a navegação e visualização de cenas tridimensionais é um passo em frente para uma experiência mais imersiva da *web*. A biblioteca deve ser capaz de carregar geometria a partir de servidores assincronamente e fornecer ferramentas para uma navegação livre e imersiva das cenas obtidas do servidor.

A biblioteca foi desenvolvida e a maioria das funcionalidades foram implementadas tais como carregamento de dados assíncronos e navegação. Gestão dos dados está parcialmente implementada, no entanto ainda existem algumas falhas no sobreposicionamento de geometria, no entanto a falha está bem definida e uma solução está prestes a ser implementada.

De um ponto de vista geral, a biblioteca tem pontos fortes e algumas falhas, no entanto está quase pronta a ser utilizada. Trabalho futuro ainda é preciso, tal como o estudo de novas funcionalidades e a sua implementação.

Contents

Listings

List of Figures

Acronyms

AJAX Asynchronous JavaScript and XML. x, 20, 35, 39

API Application programming interface. x, 2, 5, 6, 16, 20, 32, 38, 67

CAD Computer-aided Design. x, 11

CRS Coordinate Reference System. x, 40, 41

CSS Cascading Style Sheets. x, 16

DID Distributed Interactive Simulation. x, 11

DOM Document Object Model. x, 5, 7, 15–18, 43, 53, 68, 70

DTD Document Type Definition. x, 12

GeoJSON Geo JavaScript Object Notation. x, 21

GeoRSS GeoRSS-Simple. x, 21

GIS Geographic Information System. x

GLSL OpenGL Shading Language. x, 5

GML Geography Markup Language. x, 21

HTML Hypertext Markup Language. i, iii, x, 1, 2, 5–7, 15, 17, 21, 25, 28, 29, 35, 42–44, 51, 53, 62, 64, 67, 68, 70, 71

HTTP Hypertext Transfer Protocol. x, 38

KML Keyhole Markup Language. x, 21

LOD Level of Detail. x, 32, 33, 40, 54–58, 70, 71

NURBS Non-uniform Rational Basis Spline. x, 11

OpenGL Open Graphics Library. x, 5, 6, 9, 16, 17

OpenGL ES OpenGL for Embedded Systems. x, 5, 6

OS Operating System. x, 17

SAI Scene Access Interface. x, 20

SVG Scalable Vector Graphics. x, 16, 17, 19

URL Uniform Resource Locator. x, 39, 52

VRML Virtual Reality Modeling Language. x, 1, 9, 16, 68

VRML97 Virtual Reality Modeling Language 2.0. x, 9, 11

W3C World Wide Web Consortium. x, 31

W3DS Web 3D Service. x, 2, 31, 32, 39–42, 52–55, 57, 58, 64, 67–70

WebGL Web Graphics Library. i, iii, x, 1, 2, 5–7, 9, 28, 67

WMS Web Map Server. x, 22, 23, 31, 69

X3D Extensible 3D. x, 2, 9–20, 31, 32, 35, 43, 53, 67, 68

X3DOM Extensible 3D Document Object Model. x, 2, 15, 17–20, 33, 35, 43–46, 48, 62, 64, 67, 68, 70

XHR XMLHttpRequest. x, 38

XML Extensible Markup Language. x, 9, 11–13, 17, 32, 33, 38–40, 68

Chapter 1

Introduction

With the development of HTML5 and the increased capabilities of JavaScript, including the support for 3D graphics through the use of WebGL, one should consider the impact and the new opportunities those specifications allow. Until now, the usage of 3D graphics was made using plug-ins, like Virtual Reality Modeling Language (VRML) or Flash forcing the user to install third party software. With HTML5 and WebGL, the browser can now take advantage of the processing power offered by the graphic card, thus allowing the users to access 3D contents using the computer display card's Graphics Processing Unit out of the box.

One field which would greatly benefit from the adaptation to a three-dimensional navigation system would be mapping software. The current practice in map navigation through the browser is made in a two dimensional system; users can pan the map around and zoom in and out, but the viewpoint is always from above and there is no perception of the space. As we experience the world in a multidimensional way and not in a top-down two-dimensional point of view, browsing a map as a three-dimensional representation seems more similar to our experiences on our daily life than a two dimensional one.

Using the emerging specifications and the similarity to our daily life which three dimensions allow, the creation of a library to navigate three-dimensional scenes is a step towards a better immersive experience in map navigation. Taking as reference the existing libraries of two-dimensional maps, an analysis was made of the essential elements for navigation and then adapting them to a three-dimensional world. From this analysis, two key points were found: the first is obtaining the data which represents the geometry of the scene through asynchronous requests, the second is to supply the user with an adequate Application programming interface (API) which allows him to create a rich web application whose functionality is the navigation of a scene with all the freedom three dimensions can allow.

The goal of this work is to develop a client side library to help the development of rich web 3D applications. The library will provide the core components to create such 3D web applications, like getting 3D data from remote servers, combine different 3D scenes and navigation controls.

1.1 Objectives

The objective of this thesis is the investigation of the current technologies for 3D scene visualization in the browser as well as the available open source client-side web mapping solutions.

The practical result of the investigation is the development of a JavaScript library that allows users to visualize and navigate dynamic three-dimensional scenes through the browser, using asynchronous data from different servers through HTML5 and Extensible 3D (X3D), without the need for plug-ins. The objective of this library is to provide a platform for the development of rich client-side web applications that allow navigation in 3D scenes with asynchronous loading of geometry data.

1.2 Thesis Outline

Chapter 2 (State of the Art) gives an overview of the current technologies which enable the creation of the library such as WebGL, X3D, Extensible 3D Document Object Model (X3DOM) and Web 3D Service (W3DS). A study was also made gathering information about existing

JavaScript libraries used in scene navigation, which were analyzed as a foundation for the structure and usability of OL3, namely OpenLayers and WebGL Earth.

Chapter 3 (Implementation) illustrates the implementation of the Ol3 library. Beginning with a small introduction about Object Oriented Programming in JavaScript and name spaces, required to create the Ol3 object, this section presents the basic functions needed for the library; asynchronous communication with the web server, the GetCapabilities request and the bounding box. After describing the previous elements, this chapter details the library's different objects, the Scene, the Camera and the Layer.

Chapter 4 (OL3 in Action) provides a description of how the code was managed during the development stages, as well as how to obtain the code and implement new classes. This chapter also has a small tutorial in how to setup a small scene in the browser.

Chapter 5 (Conclusions and Future Work) is a summary of the thesis, as well as a discussion of the issues left unresolved and a reflection of the paths left open and how they might be approached.

Chapter 2

State of the Art

Prior to the development of the library, a study was made. Existing libraries were found and their capabilities were studied, in order to identify their strong points and flaws. Due to the innovations presented by this library there are few reference works, thus we present scarce information, and there are no open source libraries for map browsing which allow the users to navigate a map in three dimensions using the browser.

The current state of technologies was also reviewed, as to find suitable tools to develop the library, and to provide the readers with some background information about the new specifications used in the development of the library.

2.1 WebGL

WebGL is an API which is used to create 3D graphics in a Web browser. Its specification[2] was released in March 2011 and is managed by the Khronos Group. This specification, based in OpenGL for Embedded Systems (OpenGL ES) 2.0, allows the use of OpenGL Shading Language (GLSL) and is semantically similar to the standard Open Graphics Library (OpenGL) API. WebGL uses HTML5 Canvas element as Document Object Model (DOM) interface and, being a DOM API, WebGL can be used with DOM compatible languages like JavaScript, and is supported in most browsers. There are also libraries for

WebGL development, such as WebGLU, X3DOM, Processing.js, SpiderGL and PhiloGL.

Desktop browser support is widely available, as Mozilla Firefox, Google Chrome, Safari and Opera have WebGL implemented, leaving Internet Explorer as the notable exception. WebGL has also penetrated the mobile platforms, as some mobile browsers have WebGL support, mostly present in the Android operating system.

The HTML Canvas element is a rendering destination in web pages, and allows the use of different rendering APIs such as CanvasRenderingContext2D and WebGLRenderingContext. This context is where the WebGL API resides. This API can be used with Libraries or with OpenGL ES 2.0.

In order to use the WebGL API, one must create a WebGLRenderingContext object for a specified HTMLCanvasElement. To do so, the `getContext()` method must be called. When the context is created, OpenGL creates a view port rectangle, with the same width and height as the Canvas element.

```
if (!window.WebGLRenderingContext) {
    // the browser doesn't even know what WebGL is
    window.location = "http://get.webgl.org";
  } else {
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("webgl");
    if (!ctx) {
      // browser supports WebGL but initialisation failed.
      window.location = "http://get.webgl.org/troubleshooting";
    }
  }
```

Listing 2.1: JavaScript code required to initialize the WebGLRenderingContext

Next will be presented a small example of how to create a simple cube with WebGL.

```
<body onload="init()">
  <canvas id="glcanvas" width="640" height="480"></canvas>
```

```
</body>
```

Listing 2.2: Html required to initialize the scene

The HTML canvas element will contain the WebGL view port. When the body element is loaded, the `init()` function will run.

```javascript
function init() {
    var canvas = document.getElementById("glcanvas");
    initWebGL(canvas);
    if (gl) {
        gl.clearColor(0.0, 0.0, 0.0, 1.0);
        gl.enable(gl.DEPTH_TEST);
        gl.depthFunc(gl.LEQUAL);
        gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
        initBuffers();
        setInterval(drawScene, 15);
    }
}
```

Listing 2.3: init function

The `init` function, as depicted in listing 2.3, will obtain the canvas element from the DOM and will use the `initWebGL`, listing 2.4, to create the WebGLRenderingContext.

Having a valid WebGLRenderingContext object, the buffers are initiated using the `initBuffers` function, as shown in listing 2.5. This function will create the vertices array, vertex index array and the colors array. Then it will create the necessary array buffers. The `drawScene` function, as shown in listing 2.6, will be executed at a regular interval clearing the canvas and drawing the cube.

```javascript
function initWebGL(canvas) {
    gl = null;
    try {
        gl = canvas.getContext("webgl") || canvas.getContext("experimental-
webgl");
    }
    catch(e) {}
}
```

Listing 2.4: initWebGL function

```
function initBuffers() {
␣␣␣␣var vertices = [
␣␣␣␣-1.0, -1.0, 1.0,
␣␣␣␣1.0, -1.0, 1.0,
␣␣␣␣//cut due to large amount of lines
␣␣␣␣-1.0, 1.0, -1.0
␣␣␣␣];

var colors = [
  [1.0, 1.0, 1.0, 1.0], // Front face: white
  [1.0, 0.0, 0.0, 1.0], // Back face: red
  [0.0, 1.0, 0.0, 1.0], // Top face: green
  [0.0, 0.0, 1.0, 1.0], // Bottom face: blue
  [1.0, 1.0, 0.0, 1.0], // Right face: yellow
  [1.0, 0.0, 1.0, 1.0] // Left face: purple
];
var generatedColors = [];

for (j=0; j<6; j++) {
  var c = colors[j];
  for (var i=0; i<4; i++) {
    generatedColors = generatedColors.concat(c);
  }
}

cubeVerticesColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVerticesColorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(generatedColors), gl.
    STATIC_DRAW);

cubeVerticesIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVerticesIndexBuffer);
 var cubeVertexIndices = [
  0, 1, 2, 0, 2, 3,
  4, 5, 6, 4, 6, 7,
  8, 9, 10, 8, 10, 11,
  12, 13, 14, 12, 14, 15,
  16, 17, 18, 16, 18, 19,
  20, 21, 22, 20, 22, 23
];
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeVertexIndices),
     gl.STATIC_DRAW);
}
```

Listing 2.5: initBuffers function

```
function drawScene() {
␣␣␣␣gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
␣␣␣␣perspectiveMatrix = makePerspective(45, 640.0/480.0, 0.1, 100.0);
␣␣␣␣loadIdentity();
␣␣␣␣mvTranslate([-0.0, 0.0, -6.0]);
␣␣␣␣gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVerticesIndexBuffer);
␣␣␣␣setMatrixUniforms();
␣␣␣␣gl.drawElements(gl.TRIANGLES, 36, gl.UNSIGNED_SHORT, 0);
}
```

Listing 2.6: drawScene function

As can be seen from the example, the syntax is quite similar to OpenGL, and the transition from OpenGL to WebGL seamlessly.

2.2 Extensible 3D

X3D is a open standards file format used to represent real-time 3D scenes and their incorporation into non-3D content. A development from VRML, it extends the capabilities of Virtual Reality Modeling Language 2.0 (VRML97) and allows the user to encode the scenes using the VRML97 syntax as well as Extensible Markup Language (XML).

Due to its wide array of components and profiles which provide different features, X3D can be used in diverse fields such as engineering and scientific visualisations or medical visualisation as well as multimedia and entertainment. Besides 3D graphics, these being polygonal geometry, parametric geometry, hierarchical transformations, lightning, materials, texture mapping, X3D also supports shaders (both pixel and vertex shaders). 2D graphics can also be used, such as text, 2D vector graphics and can be composed in both 2D and 3D.

Besides the creation of graphics, X3D provides the users with animation tools such as timers, interpolators, humanoid animation and morphing, and to allow for richer experiences when viewing the scenes, X3D allows the mapping of audio and video to geometry.

Interaction is also a part of X3D , through the use of mouse and keyboard input, as well as camera and user movement within the scene, collision, proximity and visibility detection and physical simulation.
Networking is also possible, enabling the creation of scenes with assets located in multiple locations within a network or in the World Wide Web.

X3D uses a scene graph to display the various graphic nodes that create the 3D scene. This scene graph has a tree structure, directed and acyclic. This means the scene has a beginning of the graph, the different nodes have a parent-child relationship and there are no cycles in the graph. The 3D scene is therefore defined in a hierarchical structure, with the different nodes properly organized and their relations evident.

2.2.1 Profiles and Components

X3D has a modular structure and has different profiles, made up by components. Profiles are a set of functionalities and components, which allow the users to have different levels of support and make the scene-graph more portable and easily translated to other formats. All profiles are a superset of the previous profile.

X3D has seven different profiles; Core, Interchange, Interactive, MPEG-4Interactive, CADInterchage, Immersive and Full.

Core This profile provides minimal definitions, and is not intended for regular use. Since this profile only includes metadata nodes and no geometry nodes, coupled with specific components chosen by users, a scene can be defined.

Interchange The Interchange profile has all the basic nodes needed to define a geometry, appearance and keyframe-based animation.

Interactive The Interactive enhances the Interchange profile by adding user interaction nodes to the scene.

MPEG-4Interactive Implements the MPEG-4 multimedia specification.

CADInterchange Created in order to support Computer-aided Design (CAD) models. Has some nodes from the Interchange profile plus new nodes required for the CAD support.

Immersive This profile is the most similar to VRML97. Has all the definitions from Interactive as well as nodes for the support of 2D geometry, environmental effects and events.

Full This profile, as the name implies, includes all the nodes defined by the X3D specification. Besides providing all the functionality the previous profiles, this also adds other capabilities such as Non-uniform Rational Basis Spline (NURBS) support, GeoSpatial Humanoid animation and Distributed Interactive Simulation (DID).

Each X3D node is part of a component and, according to its level, possesses either the same or enhanced features. The usage of components can provide node functionality which is not present at the chosen profile, thus enabling the user to ensure cross platform compatibility and a controlled environment. This also provides the advantage of loading a smaller profile and allow selective functionalities of different or larger profiles to be used.

2.2.2 X3D File Structure

The X3D files rely on the XML-syntax or the VRML97 encoding to define a scene graph. A small example of the XML-syntax used is given below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.2//EN" "http://www.web3d.org/
   specifications/x3d-3.2.dtd">
```

```
3  <X3D profile='Immersive' version='3.2'>
4   <head>
5    <meta name='filename' content='sample.x3d'/> <meta name='
       description' content='a simple blue light'/>
6    <meta name='author' content='author's name'/>
7   </head>
8   <Scene>
9    <Viewpoint centerOfRotation='0 -1 0' position='0 -1 7'/>
10   <NavigationInfo type='"EXAMINE' 'ANY"'/>
11   <Transform rotation='0 1 0 3'>
12    <Shape>
13     <Sphere/>
14     <Appearance>
15      <Material DEF='LightBlue' diffuseColor='0.1 0.5 1'/>
16     </Appearance>
17    </Shape>
18   </Transform>
19  </Scene>
20 </X3D>
```

Listing 2.7: X3D syntax necessary for the creation of a scene with a blue sphere

Given the previous example of the X3D scene graph, the following describes the structure of a X3D file:

- A file header
- Start of the X3D root node
- A X3D header section
- The X3D scene graph
- End of the X3D root node

File header The file header, as shown in lines 1-2 of Listing 2.7, is comprised of a XML declaration and an optional Document Type Definition (DTD).

Start of the root node The root node, line 3 of Listing 2.7, has a XSD declaration for the X3D file and must include a version and a profile. There are three available versions; 3.0, 3.1 and 3.2. The profile will inform which of the X3D available profiles will be used. The root node is also responsible for the beginning of the scene graph.

X3D header section In the X3D header section, as seen in lines 4-8 of Listing 2.7, there is information pertaining to the file or its author, copyright, description and other relevant information the author decides to include. These are defined by a name-value pair, where the name defines the attribute and the value the corresponding content of said attribute. It is also possible to reference components in the header, and this allows the user to use components which aren't part of the current profile, without the need to load all the components of a higher profile.

The X3D scene graph As shown in lines 9-20 of Listing 2.7, the scene graph is where all nodes which represent elements in the scene are present. Nodes can be used as a XML opening and closing element pair, as seen on lines 12 and 18 in Listing 2.7 or as a singleton element, line 13 of Listing 2.7, which may or may not include attributes.

End of the X3D root node This node closes the X3D scene graph.

2.2.3 User interaction

X3D scenes are not static, users can interact with the scenes. One possible interaction is the navigation in the three-dimensional space created by the author. Using the navigation node, line 14 of Listing 2.7, one can move in several modes through the scene. Below is a short description of the navigation capabilities:

EXAMINE This mode is used for rotating solitary objects.

FLY The FLY mode allows zooming in, out and move around the scene.

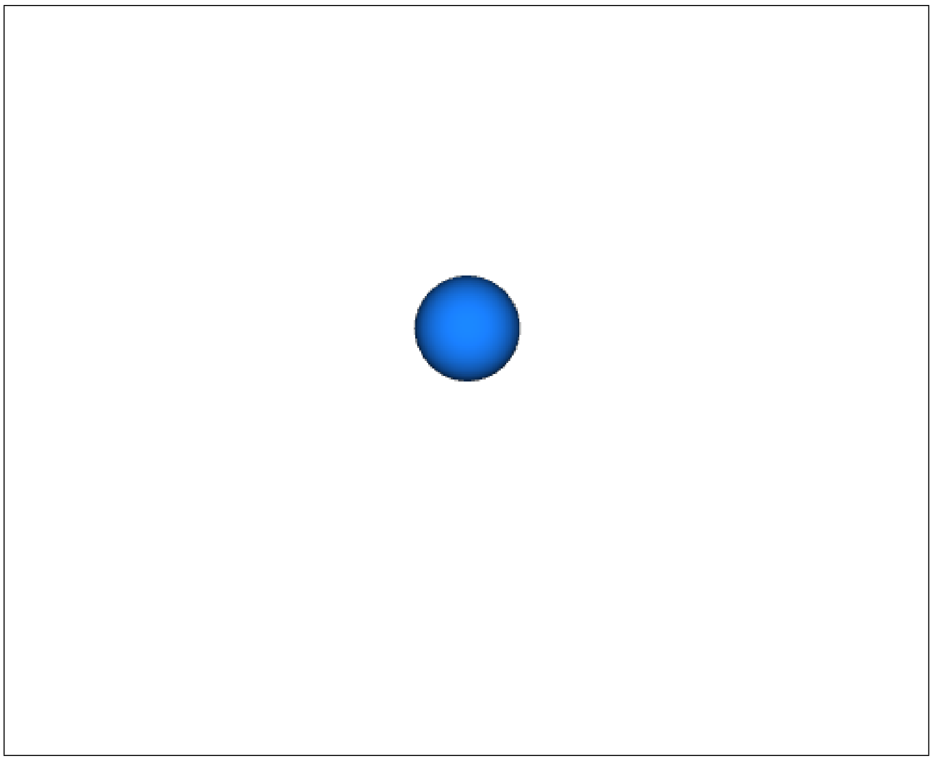


Figure 2.1: Graphical result of Listing 2.7

LOOKAT This mode allows for the user to select geometry of interest using the pointer.

WALK Used for exploration, but contrary to the FLY mode this is on the ground, from a first person point of view.

ANY Allows for the user select any mode available from the previously described.

NONE Gives user zero control of navigation, thus enabling the author to create customized navigation which will be part of an application where the X3D scene will be present.

In FLY, WALK and NONE modes, there is collision detection between the viewing camera and the geometry which disables the camera from passing through objects present in the scene. WALK mode also implements terrain following, in which there is an avatar where the camera is placed and the author can define the heights where the user is capable of surpassing and those which he can't. One can also make use of animation in order to constrain the user movement and guide the him through the scene.

2.3 X3DOM

The main purpose of the X3DOM is to create a human readable 3D scene graph which can be embedded in the HTML DOM, and allow the development of rich applications with the same ease and approach used in developing HTML applications.

Since there is no method that allows the update or synchronisation of the X3D elements, thus only allowing a single import of the DOM elements, the X3D scene model is static; X3DOM creates a bridge between the X3D scene model and the HTML5, providing a seamless integration between the two which allows the manipulation of the 3D content by changing the DOM and support for some HTML events on 3D objects.

X3D is used to define the scene graph and render the scene, and all interaction and scene graph manipulation will be handled using the standard DOM based scripting like all HTML documents. With this approach, X3DOM aims to improve the 3D Web by using less technology, by reducing the 3D system to a visualisation component and use Web technology for scripting and dynamics.

According to the HTML5 specification, X3D is referenced as a the method to declare 3D scenes, although there is no integration mode defined. X3DOM's purpose is to integrate 3D content directly into the DOM tree, as is text, images, audio and video.

When developing X3DOM, the authors reflected upon the state of 3D graphics in the Web[6].A short summary of their study follows.

- Rendering with plugins

  Flash Until version 10, there was no support for 3D in Adobe Flash, therefore users had to use 2D vector shapes and math to create simple 3D rendering systems. Since version 10, Flash now supports simple 3D transformations, yet those are still very limited.

Silverlight Microsoft Silverlight developed this plugin based on the .NET Framework to create something similar to Adobe Flash. Likewise, this plugin has little support for 3D, mainly allowing users to transform 2D graphics in a 3D space.

Java, Java3D, JOGL and JavaFX Developed by Sun, Java3D incorporated the VRML and X3D design, yet failed to thrive in the Web, later being dropped by Sun.

O3D Developed by Google, this API relies on two layers; a lower level implemented using C/C++ which is the browser plugin and a higher level implemented in JavaScript. Targeted to JavaScript programmers, it fails to offer a efficient method to define the scene content in a declarative way, forcing users to rely on JavaScript to create and alter the scene-graph content. Another drawback of O3D is the performance, hindered by the need to implement all the logic and behaviours in JavaScript.

- Rendering without plugins

CSS and SVG approach Although not a true 3D, using Scalable Vector Graphics (SVG), Cascading Style Sheets (CSS) and the Canvas element, developers built 3D pipelines. Apple also improved its WebKit engine and made possible to apply 3D transformations to 2D DOM elements.

Hardware accelerated rendering This proposal intends to incorporate 3D rendering in the browsers. Mozilla's Canvas3D or 3D-Context from Opera are two examples which wrap OpenGL and allow users to call OpenGL commands for the Canvas element. Although functional, these implementations have a drawback; just like O3D, there is no efficient method to define the scene content and the performance is not up to par.

Rendering with plugins has two drawbacks; the first being that they are not installed by default in the systems, thus leaving that task to the users, which may cause issues with the browser or the Operating System (OS). The second issue is that the application and event models are present inside the plugin, and not in the DOM.

2.3.1 X3Dom Architecture

Since the current JavaScript/OpenGL implementation lacks a scene graph implementation like the one present in X3D, X3DOM proposes a simple solution to the problem using HTML5 and X3D.

The most important reason for this approach, is that the HTML5 specification uses X3D for declarative 3D scenes although the DOM integration is not defined, nor how to access the scene graph content. Another reason for this implementation strategy is that X3D can be defined using a XML encoding and there is a DOM tree interface present in the binding interface, yet it lacks a live updating mechanism and the ability to change the DOM content.

X3DOM's purpose is to produce the results in place rather than in a plugin, much like the way SVG is implemented in the browser. In order to do so, X3DOMs implementation only supports a subset of the X3D specification and has the X3D nodes mapped as DOM elements. X3D will be used to render the scene, leaving all manipulations of the scene graph to the standard DOM-based browser scripting.

The authors intend to reduce the 3D system to a mere visualisation component, leaving all else to current web technologies. X3DOM will then be a connector between the HTML5 and X3D.

Connector

This connector provides the bridge between the browser font ends with the X3D back ends, allowing for the communication of changes in the DOM or the X3D representation. The front end adapter will access the DOM tree contents and be able to read and write the DOM representation of the X3D scene. As for the back end adapter, it should have access to the X3D runtime context and reflect the DOM tree. It is up to the connector to keep both adapters in sync therefore being able to reflect changes in both directions.

Model Updates

All changes to the DOM tree or DOM elements must affect the X3D tree by means of the back end adapter. These changes not only include X3D nodes but all the different X3D structures.

Observer Responses

The X3D execution model can change the X3D tree, when user interaction or timed events require, therefore, the connector must reflect those changes in the DOM representation of the scene. To achieve this, specific X3D tree elements will have observers, which will ensure the required changes are made.

Media streams

The connector must handle media downstream and upstream. Some elements in the X3D can require media elements such as texture images, movies and sound, therefore X3DOM must be able to access these through the browser streaming mechanism. It is also required for the back end to access the graphics context from the front end.

Scalability and Multi-Profile Support

X3DOM allows the integration of 3D structures in the DOM. This works perfectly with small amount of data, yet when a more demanding size of data is required, users can rely on the Inline node. This object allows dividing the 3D scene in multiple files and locations. One feature of notice is that each inlined scene has support for different profiles, allowing for a better control of necessary modules.

As stated in 2.2.1, X3D has support for multiple profiles allowing users to create scalable scenes. X3DOM supports this feature and also implements a specific profile as seen in 2.3.1.

This X3DOM profile is an extension of the Interactive Profile with added

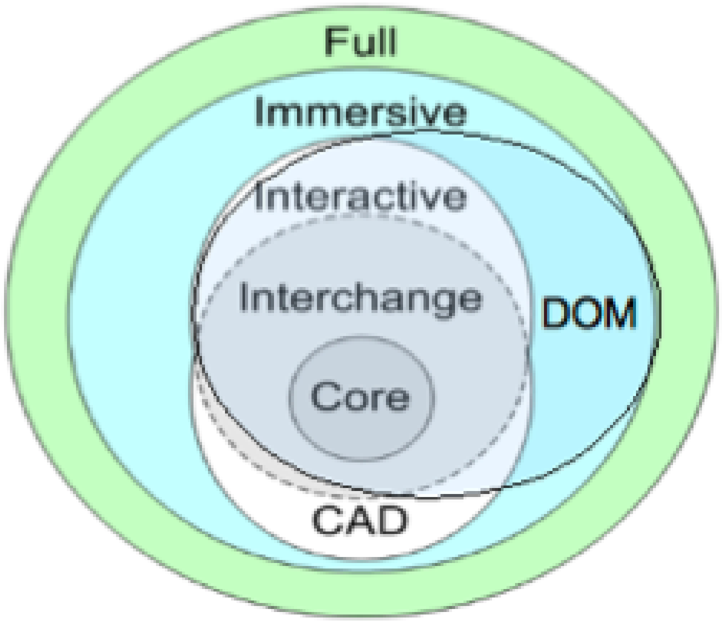


Figure 2.2: X3DOM Profile compared with existing X3D Profiles

animation and event-handling components.

Scripting and Prototypes are not supported, thus scripting must rely on the browser side.

X3D Elements as Single Point of Access

In order to allow a single point of access for the manipulation of the X3D element, the authors propose the following:

X3D Element Attributes Based in the SVG-spec, the X3D element must possess attributes to configure the render and execution engine. This includes such attributes as `xmlns, x, y, width, height, version` and `baseProfile`. The last two are required in order to request a specific X3D profile and runtime version.

SAI Interface The X3D specification allows the creation of bindings for different languages and runtime manipulation of the scene. X3DOM through the Scene Access Interface (SAI) allows the interfacing with the X3D element using browser-side scripting.

2.4 OpenLayers

OpenLayers is an open source client side JavaScript API created for the visualization of geographical data in browsers.

OpenLayers works using Client / Server model, where OpenLayers is the web map client and remote services are required to provide the data.

The client automatically request the data as needed, while the users navigate the map. These requests behind the scenes are done asynchronously, using Asynchronous JavaScript and XML (AJAX).

Since OpenLayers does not contain any data, we must rely on third-party services to provide the data. The map services supported by OpenLayers are:

- Web Map Service
- Web Feature Service
- Google Maps
- OpenStreetMap
- Virtual Earth
- Yahoo! Maps
- UMN MapServer
- MapGuide Open Source
- GeoServer
- ka-Map
- World Wind servers

- ArcGIS Server

Besides the previous services, OpenLayers also includes support for GeoRSS-Simple (GeoRSS), Keyhole Markup Language (KML), Geography Markup Language (GML), and Geo JavaScript Object Notation (GeoJSON).

OpenLayers supports multiple services simultaneously, allowing the user to gather data from different servers and displaying the results in different layers. One example is using a service as the main map source, and request a sewer line map which will be overlaid over the geographical data.

2.4.1 Using OpenLayers

Creating a Map

In order to create a map in a HTML document, one needs to obey to conditions; include the OpenLayers JavaScript library in the document and create an element to place the map.

```
1 <html>
2     <head>
3         <title>OpenLayers Example</title>
4         <script src="http://openlayers.org/api/OpenLayers.js"></script>
5     </head>
6     <body>
7         <div style="width:100%; height:100%" id="mapID"></div>
8     </body>
9 </html>
```

Listing 2.8: Creating a map with OpenLayers

After having included the library as seen in line 4 of 2.8 and creating the element for the map, line 7 of 2.8, the map is initialized with the following constructor:

```
var map = new OpenLayers.Map('mapId')
```

Listing 2.9: OpenLayers constructor

The constructor takes one string as a parameter, that string is the id of the HTML element created for the map, as seen on line 7 of listing 2.8

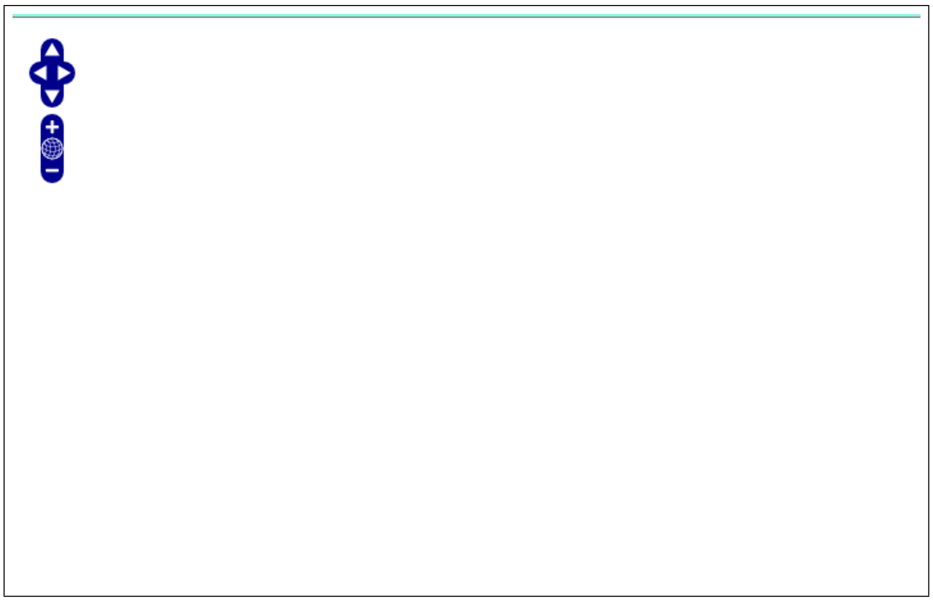


Figure 2.3: HTML page with an OpenLayers map

The map is created, and controls are visible the top left corner, yet no map images are visible. This is due to the fact that the map has no layers, therefore nothing is displayed.

Creating a layer and populating the map

To display data in a map, users must create a layer and associate it with a desired Web Map Server (WMS). This is accomplished using the following JavaScript:

```
var wms = new OpenLayers.Layer.WMS(
    "OpenLayers WMS",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {layers: 'basic'}
);
map.addLayer(wms);
```

Listing 2.10: Creating a OpenLayers layer

Creating a layer is quite simple; one must user the Layer constructor and then add the object to the map.

In the provided example, listing 2.10, we have the constructor being used in lines 1 to 5, where lines 2 and 3 are the required parameters, line 4 has optional parameters and in line 6 the layer is added to the previously created map as shown in listing 2.9.

A layer requires the user to provide the layer name and the layer url as parameters. Additional parameters can be supplied, but in this case, only which layer from the WMS is to be used.

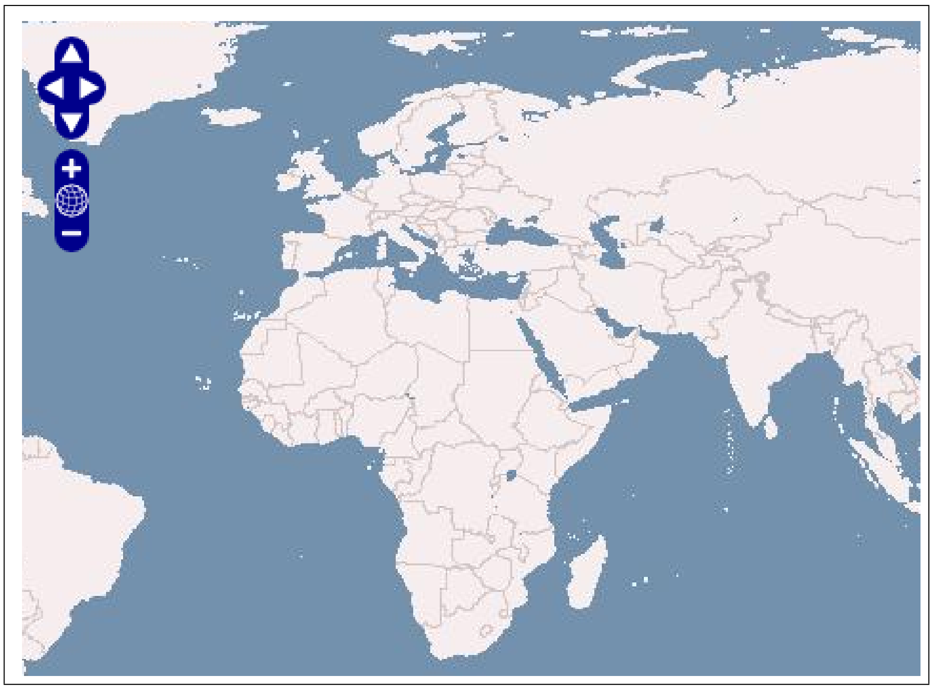


Figure 2.4: OpenLayers map with a simple WMS layer

OpenLayers allows the use of several layers simultaneously, by overlaying images and using transparency. To create an overlay, users must create a new layer, like demonstrated in listing 2.10, but add specific optional parameters.

```
var dm_wms = new OpenLayers.Layer.WMS(
    "Canadian Data",
    "http://www2.dmsolutions.ca/cgi-bin/mswms_gmap",
    {
        layers: "bathymetry,land_fn,park,drain_fn,drainage,"+
        "prov_bound,fedlimit,rail,road,popplace",
        transparent: "true",
        format: "image/png"
9  ␣␣},
10 ␣␣{isBaseLayer: false}
11 );
12 map.addLayer(dm_wms);
```

Listing 2.11: OpenLayers map with different layers

The main difference between listings 2.11 and 2.10 is that in 2.11 line 9 the provided parameter defines the layer as an overlay, and in line 6 the layer transparency is forced, in order not to cover the base layer created in 2.10.

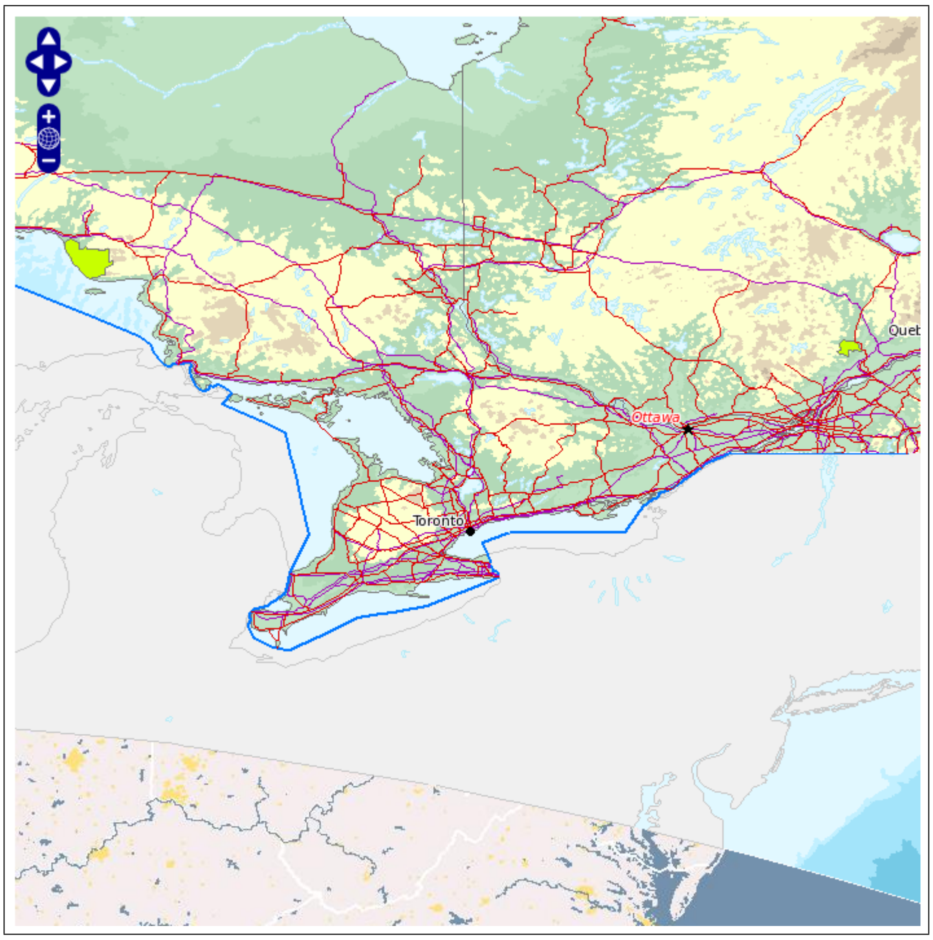


Figure 2.5: OpenLayers map with a multiple layers

Controls and User Interface

OpenLayers adds default controls to the map, yet additional control elements can be added for more interactivity. These can reside in the map or other HTML elements present in the page.

```
map.addControl(new OpenLayers.Control.LayerSwitcher({'ascending':false}));
```

Listing 2.12: Creating additional controls

This code(2.12) adds a Layer Switcher which allows users to toggle layer visibility, as seen below.

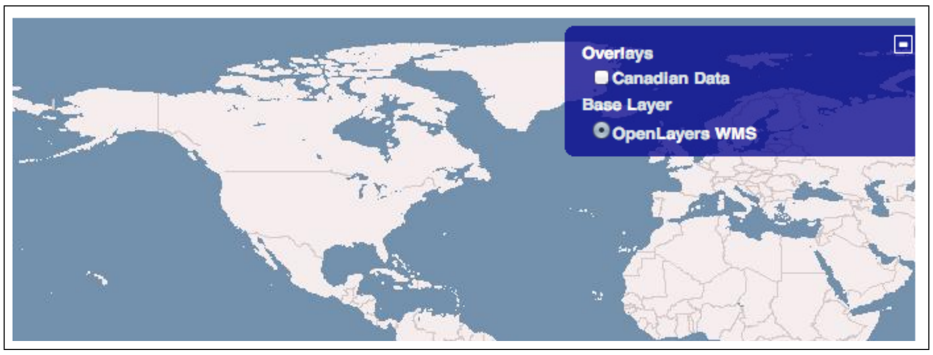


Figure 2.6: OpenLayers map with a inactive layer

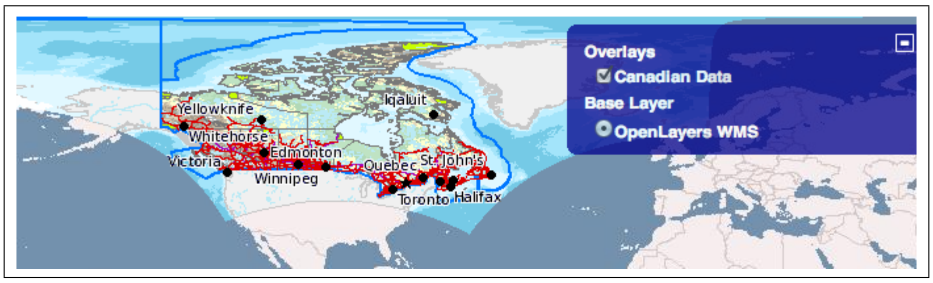


Figure 2.7: OpenLayers map with active layer

OpenLayers also provides users with a Graticule; this element creates a grid in the map referencing latitudes and longitudes. With a regular size, it adjusts its values according to the zoom level. In order to use the Graticule one uses the following JavaScript:

```javascript
var grid = new OpenLayers.Control.Graticule({
␣␣␣␣numPoints: 2,
␣␣␣␣labelled: true
});
map.addControl(grid);
```

Listing 2.13: Creating a Graticule control

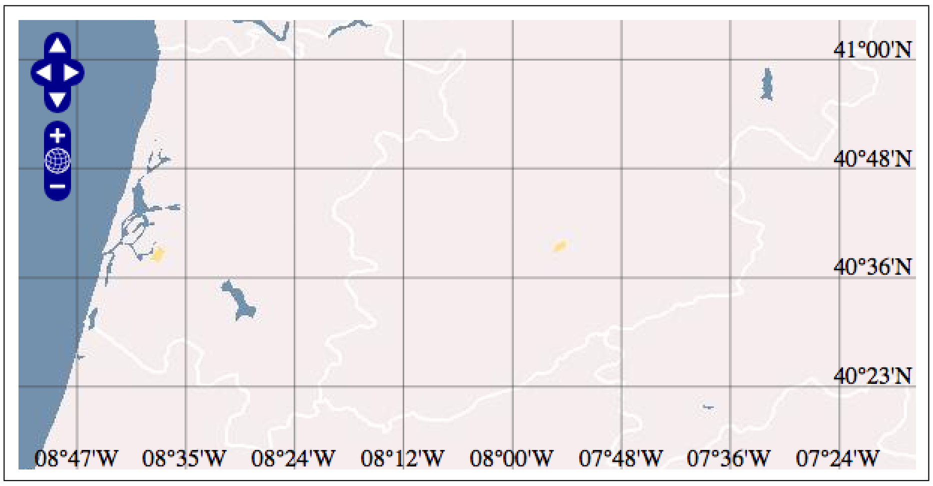


Figure 2.8: OpenLayers map with Graticule overlay

OpenLayers allows users to create text and points of interest. This is achieved by creating a TAB separated text file, in which one includes the point latitude and longitude, the title, the description and the icon to be displayed.

```javascript
var textPOI = new OpenLayers.Layer.Text( "text", {location: "./textfile.
    txt"} );
map.addLayer(textPOI);
```

Listing 2.14: Creating a Point of Interest with text

```
point␣␣␣title␣␣␣description␣icon
10,20 ␣␣my orange title␣my orange description
2,4␣my aqua title␣␣␣my aqua description
42,-71␣␣my purple title␣my purple description<br/>is great.␣http://www.
    openlayers.org/api/img/zoom-world-mini.png
```

Listing 2.15: Textfile needed for creating a Point of Interest with text

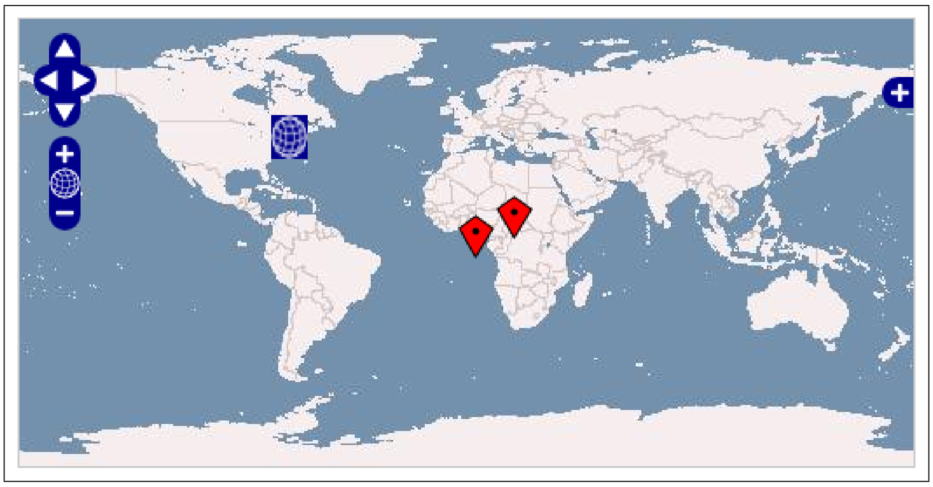
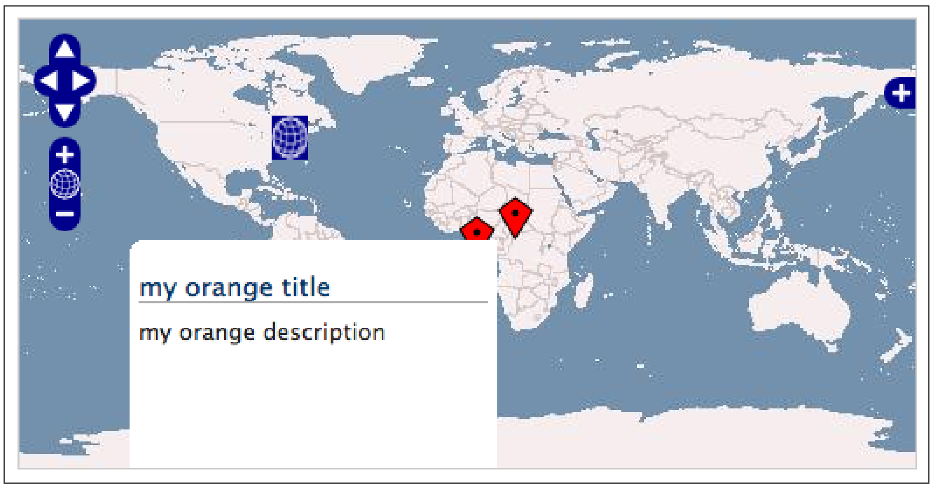


Figure 2.9: OpenLayers map with Points of Interest



my orange title

my orange description

Figure 2.10: OpenLayers map with Points of Interest and related information box

Vector shapes such as polygons, lines or points can be drawn in a map. The code required for drawing in a map is more complex that the previous examples. This complexity is due to the fact that OpenLayers requires code to create the layers for the shapes, the controllers which allow users to switch the drawing tool and also mouse handling functions. For the sake of simplicity, the code will not be listed, and only an image with the results will be shown.

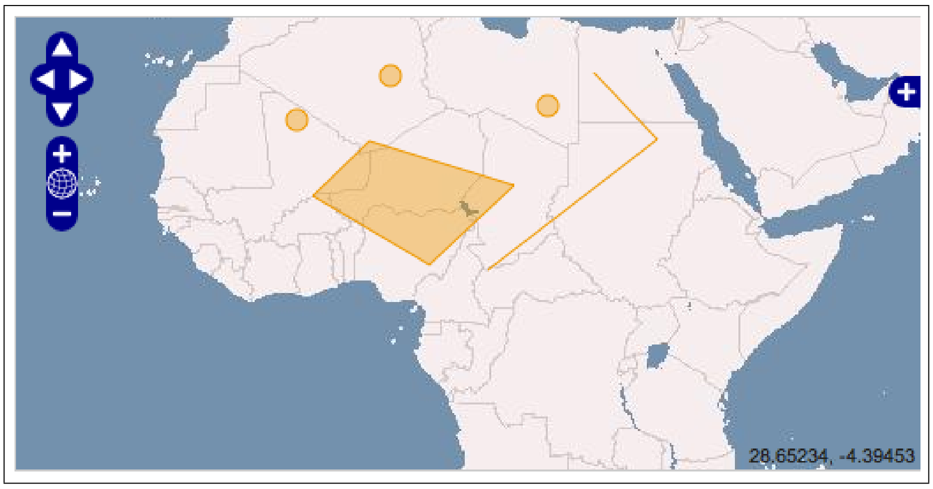


Figure 2.11: OpenLayers map with vector shapes

2.5 WebGL Earth

WebGLEarth is a project that provides users with a three-dimensional globe in the browser or mobile devices. According to the authors, there is support for the following functions:

- rotation and zoom of the globe
- camera tilt
- free movement in space
- support for existing maps like OpenStreetMap or Bing
- support for different layers or overlays like OpenLayers
- support for markers
- support for custom textures such as images from other planets

As requisites, WebGLEarth needs a browser that supports HTML5 canvas object, the WebGL extension and JavaScript[1]. Creating a new scene in WebGL Earth is simple, just like in OpenLayers as shown in 2.4.1.

```
1  <!DOCTYPE HTML>
2  <html>
3  <head>
4  <script src="http://www.webglearth.com/api.js"></script>
5  <script>
6  ␣function initialize() {
7  ␣␣var options = { zoom: 3, position: [47.19537,8.524404], proxyHost:
       'http://data.webglearth.com/cgi-bin/corsproxy.fcgi?url=' };
8  ␣␣var earth = new WebGLEarth('earth_div', options);
9  ␣}
10
11 </head>
12 <body onload="initialize()">
13 ␣<h1>WebGL Earth API: Hello World</h1>
14 ␣<div id="earth_div" style="width:600px;height:400px;border:1px solid
       gray; padding:2px;"></div>
15 </body>
16 </html>
```

Listing 2.16: Creating a WebGL Earth globe

In line four from listing 2.16, the library is included in the HTML page. Line seven the options object is created, providing desired values for the globe and the next line the globe is created, using as arguments a string defining the associated `div` element and the options object. This options object can have the following variables:

```
map - (WebGLEarth.Maps.OSM)
zoom : number
position : [lat,lng]
altitude : number
panning : boolean
tilting : boolean
zooming : boolean
atmosphere : boolean
proxyHost : string
```

Listing 2.17: Options object for WebGL Earth

From the names of the variables one can quickly infer what their purpose is. `map` allows users to choose which map tiles will be used as textures, `zoom` is the zoom level for the tiles, `position` is a vector containing latitude and longitude for the camera, `altitude` is the camera altitude in meters. `panning`, `tilting`, `zooming` and `atmosphere` are boolean values which enable or disable the function with the same name. `proxyHost` allows users to define a proxy.

WebGL Earth despite using 3D, is only a globe and the maps are displayed as textures, as can be seen in image 2.12. When zooming close to the surface, there is no sense of height. Since there are no digital elevation models, the notion of the globe disappears when the camera gets close to the surface, the visualization appears two dimensional, just like the one fond in OpenLayers and image 2.13.

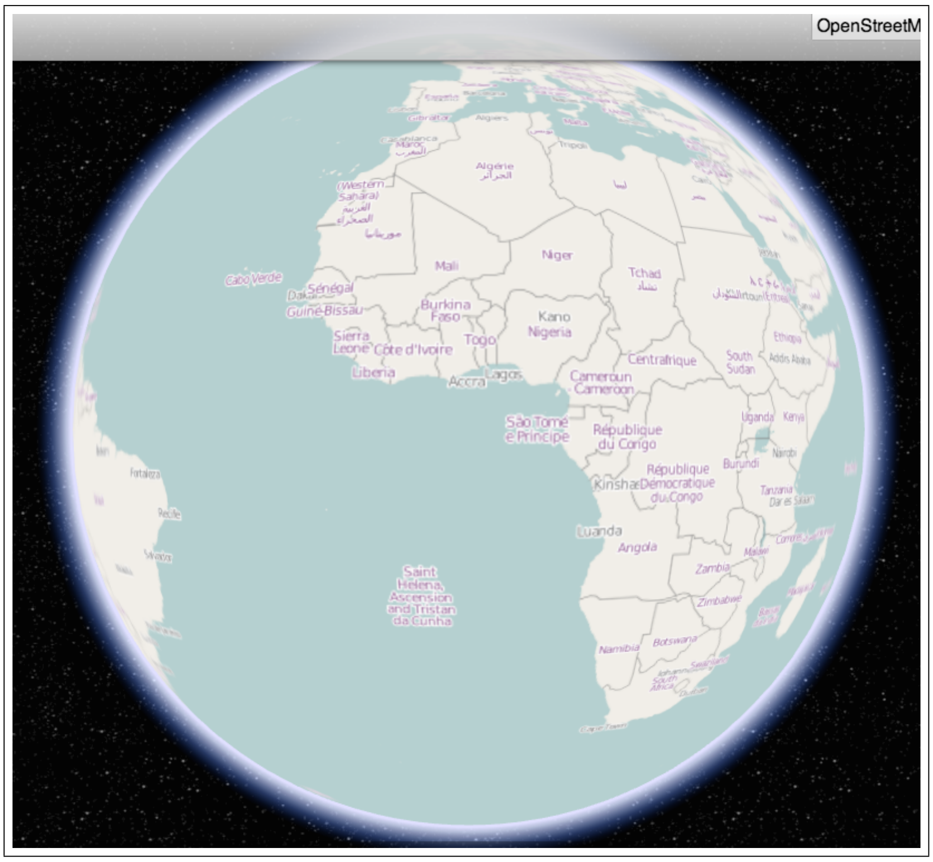


Figure 2.12: WebGL Earth globe

Figure 2.13: WebGL Earth globe zoomed in

2.6 Web 3D Service

W3DS is a service for three-dimensional geodata such as digital elevation models, city and building models, vegetation and street furniture.

The purpose of W3DS is much like that of WMS, yet while the later supplies attributes and semantic information as well as images, W3DS purpose is to provide scene graphs consisting of a tree like structure of nodes, groups, transforms, shapes, materials, and geometries. In order to obtain attributes like those found on WMS, users rely on the `GetFeatureInfo` provided by W3DS. The data format supported by W3DS is X3D, since this is a World Wide Web Consortium (W3C) standard. The level of detail for the provided geometry goes from highly detailed 3D models to prototype-like structures, enabling a controlled performance and keeping the geometry detail to the minimum needed.

The W3DS API has five different operations; `GetCapabilities`, `GetScene`, `GetTile`, `GetLayerInfo`, `GetFeatureInfo`. From those, only the first two are mandatory, being the remaining optional.

All operations have parameters, and the following are mandatory and common to every request:

SERVICE Service identifier (W3DS).

REQUEST Request identifier (`GetCapabilities` or `GetScene`).

VERSION The version of the operation to be used.

2.6.1 GetCapabilities

This operation allows users to know which resources are available in the server. The reply is a XML which contains metadata about the server and the owner. Also in the reply is information of available operations and a description of all data present in the server. This description usually contains a list of layers available and, for each layer, a list of their properties, like the coordinate system, the size, styles, if it is tiled or if there are different Level of Detail (LOD).

2.6.2 GetScene

Using the GetScene request, users can obtain 3D data from the server. As parameters for the request, the following are mandatory; `CRS` which is the coordinate system being used, `BoundingBox` which defines the rectangle encompassing the dataset, `Format` provides the server with the required output format for the 3D data, most commonly the X3D file format .`Layer` informs the server of which layers are to be included in the dataset.

2.6.3 GetTile

Since X3DOM allows adding geometry data during runtime, and the data in the server can be quite large, there is sectioned data. These are called tiles and cannot be accessed by the GetScene request.
The GetTile request allows users to select specific tiles contained in the server and the request has the following mandatory parameters; `CRS`, `Layer` and `Format` just like the GetScene request as well as `TileLevel` which defines the desired LOD for the tile, `TileRow` which is the row of the tile and `TileCol` is the column.

2.6.4 GetLayerInfo

Layers have information about their attributes, and through the use of this request, it is possible to obtain a XML document with the attributes of a specified layer.

2.6.5 GetFeatureInfo

This request allows one to obtain information about features of a element contained in a layer. Making a `GetFeatureInfo` request requires four mandatory parameters; `CRS`, `Layers`, `Format` and `Coordinates`. `CRS` is the coordinate system used, `Layers` is a list of layers the user is requesting information from, `Format` is the reply format and `Coordinates` as the name implies is a pair of coordinates used to search for the features.

Chapter 3

Implementation

3.1 JavaScript Object and Initial Requirements

Before starting the development stage, one must define the requirements the library must meet. In order to use the library, users should have a HTML5 compatible browser, as well as the X3DOM library and jQuery.

X3DOM as previously stated in section 2.3, is the bridge between the browser and X3D. jQuery is a JavaScript library that intends to simplify HTML document traversing and facilitate AJAX operations.

As for the OL3 library, it must fulfill the following requisites in order to be a fully functional prototype.

- asynchronous communication with the server
- have the ability to make a GetCapabilities request and obtain the information contained in the response
- implement a Bounding Box
- have a controlable camera and be aware of the camera's position and rotation
- be able to calculate the viewing frustrum
- the ability to differentiate layers

- capable of managing tiles of needed geometry

3.1.1 Object Oriented Programming in JavaScript and the OL3 Object

When developing in JavaScript one must retain the following premises; in JavaScript everything is an object and the use of name spacing avoids collision with other libraries.

Since it is a prototype-based language, to create new objects one just clones other objects. Using `function someFunction() {};` is exactly the same as `var someFunction = function(){};`. This newly created function is also an object and can be added as a property of some other pre-existing object.

When using code with other JavaScript libraries it is recommended the use of namespaces. This will allow for lower naming collision with other libraries and create unique groups that can be better organized and managed.

The namespace creation function guarantees that a single object with that name is created, so when one tries to create a namespace that already exists, an error will be thrown. The namespace chosen was OL3.

The selected namespace is then created and a Class function is added to the object. This function intends to implement a construct similar to those present in Java or C++. All the other classes implemented as a part of the OL3 object are then created using the OL3 namespace and implement this Class definition. This will guarantee a more organized and easier to maintain structure.

```
OL3.Class = function(
    // Class definition object: mandatory
    __proto__
    ) {
        var Class = __proto__.hasOwnProperty("initialize") ?
        // use it ...
        __proto__.initialize :
        // otherwise create one and assign it
        (__proto__.initialize = function () {})
```

```
   ;
   Class.prototype = __proto__;
   return Class;
};
```

Listing 3.1: Class implementation

The Class function of the OL3 object checks if a newly created object has a `initialize` function; if so it will use it as a constructor, otherwise it will create an empty one.

To define a new class, one would do the following:

```
OL3.Person = new OL3.Class(
{
initialize : function (name)
{
this.name = name;
}
sayName : function()
{
alert(this.name);
}
});
```

Listing 3.2: New class definition

To create a new instance of the Person class, all one needs to do is assign a variable with the Person object.

```
var Human = new OL3.Person("John Doe");
```

Listing 3.3: Instantiating a new Person Object

Having the namespace and the Class function implemented, the next step was the creation of objects that allowed connection to the server, obtaining information about the data present in the server and objects that represented the main structures needed; the Scene, the Camera and the Layer.

3.1.2 Asynchronous Communication with the Web Server

JavaScript supports asynchronous communications with web servers. Gathering information and 3D data from a server is a common operation and so an object to perform these tasks with a web server was created. This object, named OL3.XMLHTTPRequest, uses the XMLHttpRequest (XHR) API to send Hypertext Transfer Protocol (HTTP) requests to the server and load the data contained in the response back to the scripts.

Any object which requires data to be loaded from the web server has a function called `getReply` that is used by the OL3.XMLHTTPRequest object when the data has been loaded successfully. The object's `getReply` function will act to what has been returned from the server and then, accordingly to the required action, will parse and use the information, being this about a layer or 3D data contained in the web server.

The OL3.XMLHTTPRequest implementation relies on the jQuery library since it simplifies the use of XHR.

```
 1 var jqxhr = $.ajax( urlRequest )
 2     .done
 3     (
 4         function()
 5         {
 6             var reply = jqxhr.responseText;
 7             parent.getReply(reply);
 8         }
 9     )
10     .fail
11     (
12         function()
13         {
14             throw new Error("Error obtaining data from server");
15         }
16     );
```

Listing 3.4: Example of XMLHttpRequest using jQuery

jQuery makes the implementation quite fast and simple, as can be seen in the listing 3.4. This listing represents more than 50% of the code required to create the OL3.XMLHTTPRequest object.

In line 6, the variable `reply` contains the data from the web server, and in line 7, the `getReply` method from the object which made a request is being supplied with the results. These are always XML files, which the object will parse and create the necessary structures.

3.1.3 GetCapabilities

The OL3 implements a GetCapabilities request. By sending this request, a client can obtain information about a W3DS server, such as available data, supported formats among others.

Using the following Uniform Resource Locator (URL), one can make a capabilities request to a W3DS server.[7]
`http://3dwebgis.di.uminho.pt/geoserver3D/w3ds?VERSION=0.4.0&SERVICE=w3ds&REQUEST=GetCapabilities`

The request is made using AJAX, and relies on the OL3.XMLHTTPRequest class. The reply is a XML file containing all the information relative to the server. The information a GetCapabilities reply gives consists of the following items; services supported, formats supported, spatial reference systems, list of map layers, SLD/Styles and vendor specific codes.

```xml
<ows:Operation name="GetScene">
    <ows:DCP>
        <ows:HTTP>
            <ows:Get xlink:href="http://3dwebgis.di.uminho.pt/geoserver3D/
    w3ds?">
                <ows:Constraint name="GetEncoding">
                    <ows:AllowedValues>
                        <ows:Value>KVP</ows:Value>
                    </ows:AllowedValues>
                </ows:Constraint>
            </ows:Get>
        </ows:HTTP>
    </ows:DCP>
</ows:Operation>
```

Listing 3.5: Example of a GetCapabilities reply describing a get scene request

Each time a layer is created, there is a new OL3.Capabilities object created which is part of the layer object. Using a string containing the W3DS URL and a string defining a name for the layer, the OL3.Capabilities object will parse the reply XML obtained from the W3DS. From the data received, the object will try to check if a layer with the name the user provided exists. If found, the layer properties will be parsed and returned to the new layer. If no information is found for the layer name requested, the new layer will not be created.

Since the OL3.Capabilities object will communicate with the server, this object possesses a `getReply` function.

This function will check if the reply from the W3DS server is a valid one or an exception has occurred. In case the response is valid, then it proceeds to find requested layer. When the layer is found, the OL3.Capabilities parsers will obtain all the properties of the layer, such as the Coordinate Reference System (CRS), the styles, the different LODs and if the layer is tiled or not. These properties will then be sent back to the layer object and a new layer is available to the user.

3.1.4 Bounding Box

> "The bounding box is the computationally simplest of all linear bounding containers, and the one most frequently used in many applications." [9]

Inheriting from the geographic information metadata standard **ISO 19115 Metadata Standard**, this object represents the maximum extents of a two dimensional object in a 2D coordinate system, being represented by a four value pair of $min(x)$, $max(x)$, $min(y)$ and $max(y)$. These values must fully enclose the object and the rectangle faces must be aligned with the axes of a Cartesian coordinate system.

The Bounding Box class is a structure that contains the two delimiting points of a layer, these being the lower left corner and the upper right corner. In geospatial data, these serve as an approximation for the areal coverage of the feature.[4]

The OL3 bounding box object contains an array with the four values delimiting the rectangle.

lowercorner1 this value is the equivalent to $min(x)$

lowercorner2 this value is the equivalent to $min(z)$

uppercorner1 this value is the equivalent to $max(x)$

uppercorner2 this value is the equivalent to $max(z)$

Besides the rectangle values, there is a CRS variable, so that there is a reference to the coordinate system being used.

This being a library which allows navigation in 3D, one would assume that the bounding box would be in 3D, with x, y and z axis, yet the values returned by the W3DS only contain two axis (x and z).

Following is a description of the 3D bounding box implementation compared with the W3DS implementation.

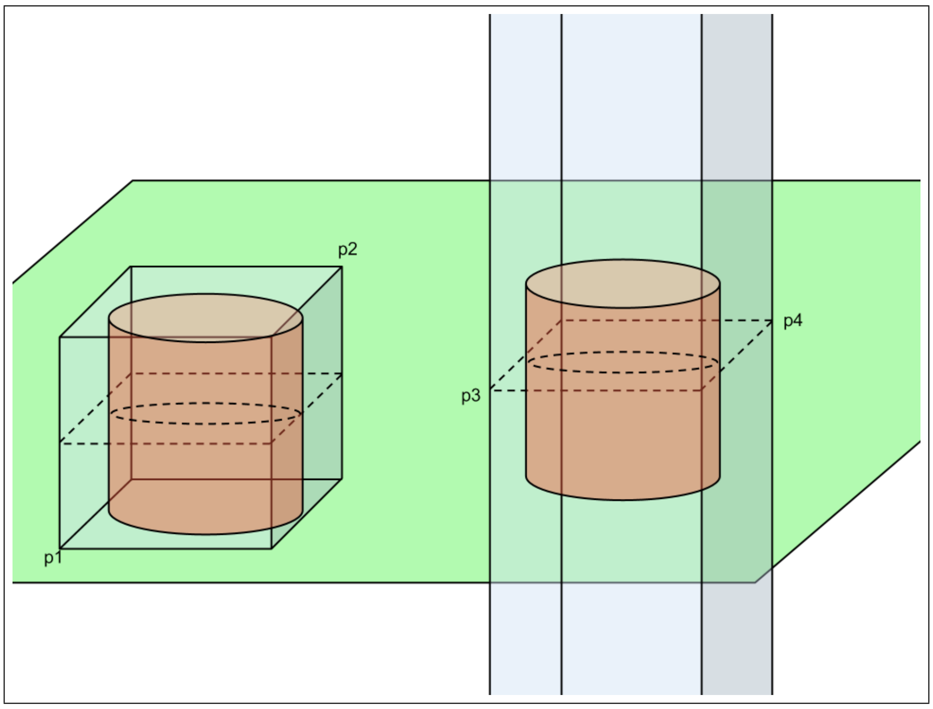


Figure 3.1: 3D Bounding box (left) and current implementation (right)

As can be seen in image 3.1, on the left side there is a 3D bounding box and on the right is the current implementation. The light green shape represents the $y = 0$ plane which intersects two cylinders. The 3D bounding box containing the left cylinder is a blue cube which can be defined with points $p1_{x,y,z}$ and $p2_{x,y,z}$. Using the current implementation, shown on the right, there is only $p3_{x,z}$ and $p4_{x,z}$. With these vales one can only define a rectangle, that when changed into 3D would represent all objects contained inside a parallelogram extending into $-\infty$ and $+\infty$ in the y axis.

Since geographical data is quite uniform vertically, the need for a 3D bounding box is not that relevant and is open for discussion. W3DS is only the source for 3D data used by OL3, the bounding box implementation in the server is irrelevant, and OL3 must use that implementation.

3.2 OL3 Structures

3.2.1 Scene

The Scene is a top-level object and acts as a container which allows children objects to interact with each other. It contains a Camera object and a Layer objects array. The Camera object will allow the different objects within the scene to access the camera properties and the Layer array will contain all layers associated with the scene.

The scene initialization is quite similar to the one used in OpenLayers, as demonstrated in 2.8. The user creates a `div` element in the HTML document which will contain the scene, and then he must initialize the scene by using the following function:

```
var scene = new OL3.Scene(bb, options);
```

Listing 3.6: Creating a Scene

As seen in the example listing 3.6, one creates an scene object as a variable. The `bb` argument provided is a Bounding Box, which is mandatory, but yet of no use. As for the `options` variable, it is an object which contains properties for the scene. This last object is not mandatory, for the Scene object has default values, as shown in listing 3.7.

```
var default_args =
  {
    'div' :"#map",
    'width' :450,
    'height':450,
    'stat' :false,
    'log' :false
  };
```

Listing 3.7: Scene default variables

The default arguments represent the following:

div This string is the `id` of the `div` element in which the user intends to place the scene.

width / height These variables represent the width and height of the rendering window for the X3DOM object, which will then be placed inside the `div` element.

stat / log These two boolean variables are here to allow users to turn on debug information regarding the X3DOM object.

When a scene is created, the HTML DOM is changed and X3D nodes are inserted. These nodes are an empty X3D scene without any geometry nodes, and the scene-graph resembles the one described in 11.

Inside the scene node there is a transform node, whose id is named "entrada".The node serves as an entry point where all 3D elements will be added. Through this method one can identify clearly which elements of the scene-graph are part of the initial scene, those outside the transform node, and the ones added later, inside the transform node.

3.2.2 Camera Implementation

In order to allow interaction for the scene navigation, several implementations of camera movement were studied. The main purpose of this study was to allow the end-user to move the camera in the scene and the library always being aware of the camera position and rotation. Through this constant monitoring, it was also intended to have a notion of what geometry was seen by the camera.

Three different implementations were studied; one which the user could move the camera with buttons present in the HTML, the second approach was standard X3DOM camera movement with the mouse and calculating the viewing frustum, and finally using the same method as the previous, yet instead of calculating the viewing frustum, rays are emitted from the four corners of the camera and then calculating where they would intersect the ground plane. A more detailed description is given below.

HTML buttons

This approach uses four transform nodes just below the scene root node and HTML button elements to control transformations applied to the scene-graph.

The transform nodes are a translation node, to change the scene translation in the x, y and z axis, as well as three rotation nodes, one for each axis of rotation. All the 3D elements will be placed inside these nodes, in order to be affected by their transformation. By changing the values of the transform nodes, one can change the scene in relation to the camera position.

This is the approach with the easier implementation, yet there is a minor setback, as there is no simple method to calculate what geometry is within the camera view port.

Viewing Frustum

The viewing frustum[5] is a volume in the modeling world where all the visible objects reside, although some occlusions may occur. This volume has the shape of a truncated pyramid when using a perspective projection, where the apex resides in the camera position and the base in the far clipping plane. The near clipping plane truncates the top of the pyramid, thus calling the volume a frustum.

The camera controls in this implementation rely on mouse interactions with the canvas element. All changes to the scene and the camera are handled by X3DOM, therefore to have the camera position and rotation as well as what geometry is visible, one must obtain those values from X3DOM.

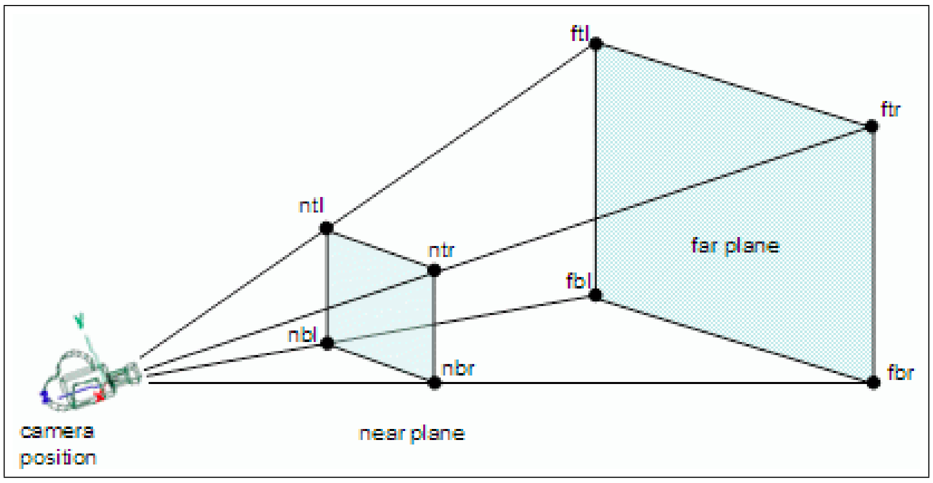


Figure 3.2: Camera viewing frustum - obtained in Lighthouse3d.com

Obtaining the position and rotation of the camera object is quite straight forward. When a user changes the camera position the view port element triggers an event and three objects are returned; the first is an array with the three position values. The second is the camera rotation, in which there is an array with the rotation amount for each axis, as well as a rotation value. The third value is a matrix which is used to calculate the viewing frustum.

From the matrix returned by moving the camera, the up vector is obtained from the values contained inside, as well as the right and look vectors. In addition to those vectors, the near clipping distance and the far clipping distance are required; these values are hard coded in X3DOM and are 0.1 and 100000. These two values represent the distance from the camera to the near plane and far plane, and these are the planes which contain the visible objects. Using the near and far clipping distances it is possible to calculate the center point of the clipping planes.

The far, fc, and near clipping planes center nc is obtained using the following two formulas:

$$fc_{x,y,z} = p_{x,y,z} + d_{x,y,z} \times farDist$$

$$nc_{x,y,z} = p_{x,y,z} + d_{x,y,z} \times nearDist$$

Where p is the camera position, d is a normalized vector with the direction of the camera's viewing ray, $farDist$ is the far clipping distance and $nearDist$ is the near clipping distance.

Using the field of view, which is obtained from the X3DOM viewpoint object, the aspect ratio and the clipping distances, one can obtain the the width and the height of the near and far planes.

Height of near plane

$$Hnear = 2 \times \tan(fov/2) \times nearDist$$

Where fov is the field of view and $nearDist$ is the near clipping distance.

Width of near plane

$$Wnear = Hnear \times ratio$$

Using the height of the near plane $Hnear$ calculated previously, multiplying it with the value of the aspect ratio $ratio$ one obtains the width of the near plane.

Height of far plane

$$Hfar = 2 \times \tan(fov/2) \times farDist$$

With the field of view fov and the far clipping plane distance, $farDist$, the height of the far plane is obtained.

Width of far plane

$$Wfar = Hfar \times ratio$$

As for the width of the far clipping plane, it can be calculated using the height of the far plane, $Hfar$, and the aspect ratio $ratio$.

From the values calculated with the previous formulas, the only remaining step to obtain the viewing frustum is to know the values of the four points defining the corners of the near and far clipping planes.

To determine the far clipping plane corners, one must use the following functions:

$$ftl_{x,y,z} = fc_{x,y,z} + (up_{x,y,z} \times Hfar/2) - (right_{x,y,z} \times Wfar/2)$$

$$ftr_{x,y,z} = fc_{x,y,z} + (up_{x,y,z} \times Hfar/2) + (right_{x,y,z} \times Wfar/2)$$

$$fbl_{x,y,z} = fc_{x,y,z} - (up_{x,y,z} \times Hfar/2) - (right_{x,y,z} \times Wfar/2)$$

$$fbr_{x,y,z} = fc_{x,y,z} - (up_{x,y,z} \times Hfar/2) + (right_{x,y,z} \times Wfar/2)$$

Where ftl is the far top left corner, ftr is the far top right corner, fbl the far bottom left corner and fbr the far bottom right corner of the plane. fc is the far clipping plane center, up is the up vector and $right$ the right vector. The $Wfar$ stands for width of the far clipping plane and $Hfar$ the height of the far clipping plane.

The same functions can be used to determine the corners of the near clipping plane, ntl, ntr, nbl and nbr. One still uses the up and right vectors, up and $right$, but instead of the width and height of the far clipping plane, the width and height of the near clipping plane are used, $Hnear$ and $Wnear$.

$$ntl_{x,y,z} = nc_{x,y,z} + (up_{x,y,z} \times Hnear/2) - (right_{x,y,z} \times Wnear/2)$$

$$ntr_{x,y,z} = nc_{x,y,z} + (up_{x,y,z} \times Hnear/2) + (right_{x,y,z} \times Wnear/2)$$

$$nbl_{x,y,z} = nc_{x,y,z} - (up_{x,y,z} \times Hnear/2) - (right_{x,y,z} \times Wnear/2)$$

$$nbr_{x,y,z} = nc_{x,y,z} - (up_{x,y,z} \times Hnear/2) + (right_{x,y,z} \times Wnear/2)$$

Ray Emission

Using the previously described approach of the viewing frustum as a starting point, there is another solution to obtaining the four edges defining the sides of the truncated pyramid. X3DOM allows users to emit rays at specific points in the view port. These rays will return a normalized vector with a direction in the form of a X3DOM Line object, comprised of a starting position and a direction. If one emits four rays, one at each view port corner, the result would be the direction of the four edges defining the pyramid sides of the viewing frustum.

To emit a ray at a given position the following method is used:

```
var line = viewarea.calcViewRay(x, y);
```

Listing 3.8: Emitting a ray

In this approach there is no need to calculate the viewing frustum and the clipping planes, only the edges of the pyramid. This is done because of the intersection with the $y = 0$ plane. This intersection is detailed in the following subsection.

$y = 0$ plane intersection

To load only the needed geometry from the server and keep the amount data to a minimum, one must find what is seen by the camera and therefore required to be loaded. Since the far clipping distance is 100000, loading all data between the camera and the far clipping plane might be unnecessary. Since the navigation in made most of the time from a bird's eye perspective, the approach was intersecting the viewing frustum with the $y = 0$ plane.

The first step for this approach is to understand the math behind that operation[10]. That being done, the next step is to implement the appropriate functions. An explanation for both follows.

Given the plane defined by x_1, x_2, x_3 and a line passing through x_4 and x_5, the intersecting point can be solved by the following simultaneous equations for x, y, z and t

$$0 = \begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} \tag{3.1}$$

$$x = x_4 + (x_5 - x_4)t \tag{3.2}$$

$$y = y_4 + (y_5 - y_4)t \tag{3.3}$$

$$z = z_4 + (z_5 - z_4)t \tag{3.4}$$

where t is:

$$t = -\frac{\begin{vmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 0 \\ x_1 & x_2 & x_3 & x_5 - x_4 \\ y_1 & y_2 & y_3 & y_5 - y_4 \\ z_1 & z_2 & z_3 & z_5 - z_4 \end{vmatrix}}, \tag{3.5}$$

When moving from the theory to implementation, most implementations found were in Java or C++, and required the creation of specific classes. One implementation in GNU Octave was found[8] which did not required creating classes or complex structures and a port was done to JavaScript.

```
linePlaneIntersect: function(lineP1, lineP2, quad)
{
    var t = null;
    var p = null;
    var lineDir = {};
    lineDir.x = lineP1.x - lineP2.x;
    lineDir.y = lineP1.y - lineP2.y;
    lineDir.z = lineP1.z - lineP2.z;
    var numerator = Utils.dot(quad.normal, lineDir);
    if(Math.abs(numerator) > 0.000001)
    {
        t = (Utils.dot(quad.normal,Utils.subtract2Points(quad.point1,
    lineP1) ) ) / numerator;
        p = new OL3.Point(parseFloat(lineP1.x) + parseFloat( lineDir.x*t),
    parseFloat(lineP1.y) + parseFloat( lineDir.y*t), parseFloat(lineP1.z)
    + parseFloat( lineDir.z*t));
        return p;
    } else
    {
        return null;
    }
},
```

Listing 3.9: Intersection of a plane with a line function

The function `linePlaneIntersect` takes as arguments two points crossed by the line `lineP1` and `lineP2` and a plane `quad`. This plane object besides four points contains a normal, which will be used to calculate the intersection. If the dot product of the plane's normal and the `lineDir` is of significant value, as seen in line 9 of listing 3.9, then there is an intersection and that point, `p`, is calculated. Besides calculating the point of intersection, the function also provides `t`, which can be used to check if the intersection is contained between the two points `lineP1` and `lineP2`.

Using either the Viewing Frustum approach described in 3.2.2 or the Ray emission method described in 3.2.2 one can obtain the four lines that contain all objects visible by the camera. Intersecting those lines with the $y = 0$ plane, the four points that create the bounding box of the visible objects are calculated. Using this bounding box it's just a matter of asking the server for geometry inside those points to obtain all the data needed.

Comparison between the different approaches

The HTML buttons method as a user interface does what is required, as it allows a controlled navigation. Although a controlled navigation is good for inexperienced users, as it avoids getting lost, it doesn't work well with large amounts of data as the library cannot calculate the bounds of camera visibility.

When using the viewing frustum the navigation is smoother yet more unrestrained, therefore inexperienced users can get lost navigating the scene. This approach however, does allow the library to calculate the visible objects, making this optimal for large amounts of data that need to be gathered from the server when needed. This implementation had a error, as there were issues when changing the camera and obtaining the points of intersection in the $y = 0$ plane. These where swapped on some axis and when swiveling the camera the geometry requested from the server was wrong. Due to this error the Ray emission was found as a viable solution.

The ray emission approach allows users to experience the same smooth and unrestrained navigation as the viewing frustum implementation, but since the amount of functions required to obtain the four points is considerably smaller and the library produces no errors calculating the intersection points when the camera swivels, this is the current method used by the Camera class.

3.2.3 Layer

Just like in OpenLayers different geographical information has different layers, so OL3 borrows the layers concept in order to separate different geometry information.

As an example, if one should have terrain data, buildings and sewage lines in a scene, one would separate the terrain, the buildings and the sewage in different layers. This separation will benefit the library as a more adequate method of controlling the geometry information in the scene graph and to allow easier access and perform changes in specific geometry groups. This separation also allows users to request only what they wish to load, since they could be interested in only parts of the geometry contained in the W3DS server, say for example only the buildings and sewage lines, but no terrain.

Each time a Layer is created, the Scene object adds the new Layer to an array. This is done so users or other objects can access any layer contained the scene.

```
var buildings = new OL3.Layer(parent,urlEntry, nameLayerEntry);
```

Listing 3.10: Creating a new Layer

Creating a Layer is a straight-forward process, using the constructor and supplying a scene which will contain the Layer, `parent`, a URL for the W3DS server, `urlEntry`, and the last parameter is the name of the layer as contained in the web server, `nameLayerEntry`.

When instantiating a new Layer, the object will use the Ol3.Capabilities class to check if it is a valid layer. This is done so that the layer can have a valid W3DS URL and a valid name. This valid name implies that the name of the layer must be the same as a layer name in the W3DS server being used.

Doing so ensures that the layer will have available data in the server to be shown to the users.

If there is no such layer in the W3DS server, the Layer object will not be added to the scene, avoiding later requests for non-existent geometry data from the server.

Obtaining geometry data from a server

A Layer is responsible for obtaining geometry data from the server and displaying it to the users. In the server one can find two types of layers; the first being untiled models and the second tiled geometry. When creating a layer, at the time the capabilities file is parsed, there is information whether the layer is tiled or not. This is reflected in a property of the layer object, and therefore, when making requests for geometry one needs to differentiate the requests. If the layer is not tiled the GetScene request is used, as described in page 32, otherwise the GetTile request is used as seen in page 33.

Making a request to the W3DS server for geometry is made just like the Capabilities request; a OL3.XMLHTTPRequest is created providing the server with the required data, and then the server returns the X3D file asynchronously. As soon as the server makes the data available to the layer object, the X3D file is parsed and the nodes are inserted in the DOM. Although the geometry comes in the same format, X3D, the approaches to deciding which data is requested from the server are different.

Loading a untiled model, the layer makes a request to the server for the geometry and when the data is received, the X3D nodes are parsed and then inserted into the scene-graph entry point of the HTML DOM.

Tiled geometry requires a different approach than the one used for untiled models. Tiled geometry is used for large amounts of data, which makes it cumbersome to load in a single request. To avoid this setback, this data is divided in square tiles. The W3DS server allows users to request tiles according to a grid position. This grid depends on the bounding box of the geometry and the level of detail desired. When creating a layer, the tile sizes are obtained from the GetCapabilities response, and then the layer object will create a

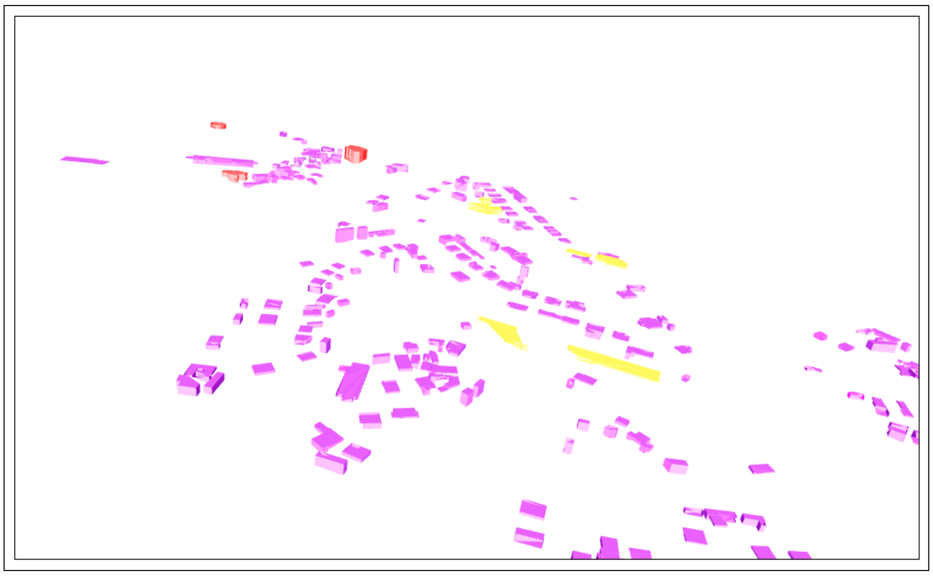


Figure 3.3: Loaded geometry for a untiled model using a GetScene request

structure to contain the grid information. This structure is an array whose length is dependent of the different LODs available. In each item of the array is stored a two-dimensional array of tiles, reflecting a grid of tiles present in the server for the respective LOD.

Tiles and the Grid

Tiled geometry is obtained using a GetTile request from a W3DS server. OL3 layers can either have untiled or tiled geometry. Layers with tiled geometry possess a grid array, where grids are created and stored according to the number of LODs present in the server for the specified layer.

For each level of detail, there is a specific tile size and using the tile size and the layer's bounding box, the amount of rows and columns composing the grid are calculated.

```
this.numberColumns = Math.floor(width/tilesize);
this.numberRows = Math.floor(height/tilesize);
```

Listing 3.11: Calculating the grid's columns and rows

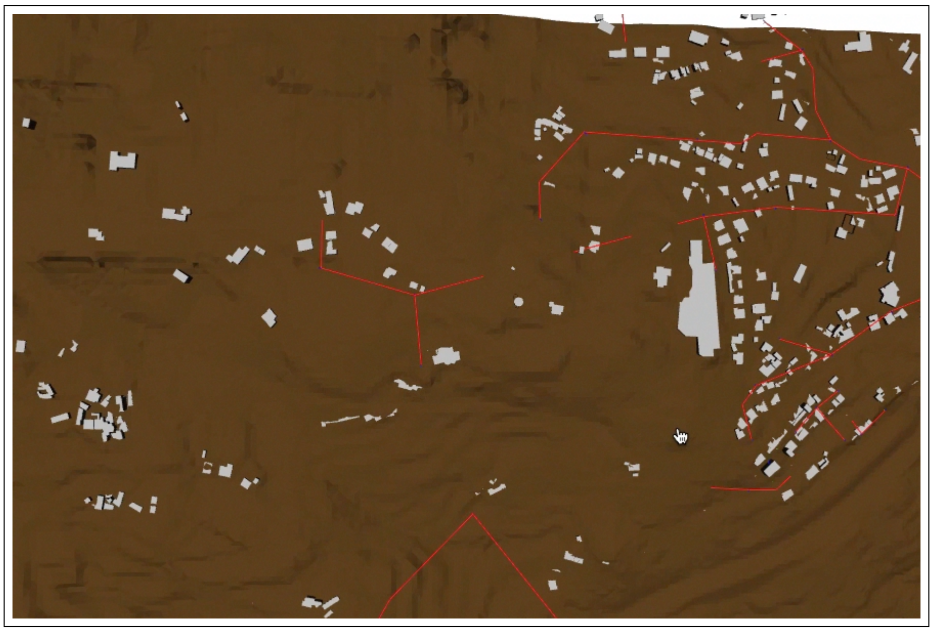


Figure 3.4: Loaded geometry for a untiled model using a GetScene request and tiled geometry using a GetTile request

The used W3DS server had exponential tile sizes, and the grid array structure can be represented graphically as seen in figure 3.5. The smaller the square is, the bigger the LOD.

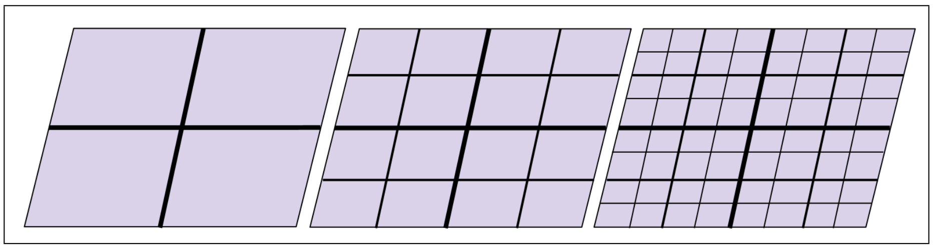


Figure 3.5: Grid representation

Each grid contained in the array is itself a two-dimensional array, with its size given by the number of columns and rows present in the grid. Each element of that array is a tile.

The Tile object is a OL3 class that allows the library to check if the tile has been requested from the server or if the tile is loaded in the scene. These will help reduce the amount of data loaded as no repeated geometry will be requested. It is also possible to check the tile's lower corner, the row, column and also the midpoint. Using both the grid and the tiles, loading the necessary geometry from the server is simplified. With the method described in page 49, the library will build a temporary grid to decide which geometry is required.

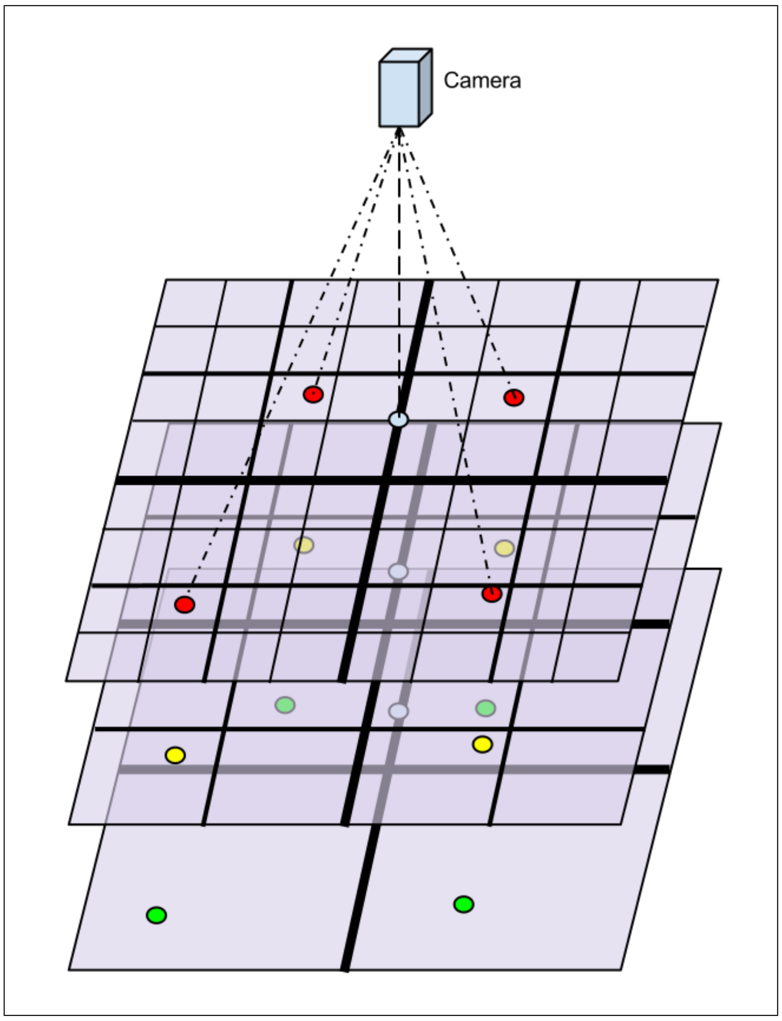


Figure 3.6: Temporary Grid construction

Starting with figure 3.6 as a reference, the first grid is the one with a higher LOD and the last grid is the one with the lowest LOD.

The temporary grid is built in the following way; first the camera position and the intersection points of the viewing frustum with the $y = 0$ plane are obtained. The camera position intersects the grids in the point represented by the blue circle and the viewing frustum for the first grid intersects the plane in the four red circles, the second grid is intersected in the yellow circles and the third grid in the green circles. The points coordinates are the same for the different grids, but the column and row that contains them can differ.

After obtaining each grid's rows and columns for the four points and using the bounding box principle, the lower and upper corners of the points in the grid are calculated, yet instead of coordinates, the row and column numbers are used. The next step is calculating the 2D distance between the middle of each tile contained in the box and the camera position. If the distance is smaller than a threshold the tile is selected, else it's discarded. This step is repeated for all tiles in all the desired LOD grids.

Taking into account the exponential nature of the grids, using just the previous method could create gaps or tiles of different LOD overlapping in the loaded geometry. Two additional steps are required to ensure no such errors occur. Each time a tile is required and not part of the grid with the lowest LOD, three more are added to the required tiles, forming a 2×2 square that aligns perfectly with the corresponding tile in the previous LOD grid. As stated previously, the grid with the lower LOD doesn't need this step, because there are no larger tiles to align to. Gaps in the grid are now solved and the only remaining action is to avoid overlapping. To do so, the groups of tiles with higher LODs are checked against the next larger grid and if there is an overlap, the tile from the larger grid is discarded. This is repeated for all grids except for the one with the lowest LOD since there are no overlaps possible.

There is now a temporary grid with different tile sizes built and all that remains is to check which tiles are not yet loaded and request them from the W3DS server using a GetTile request.

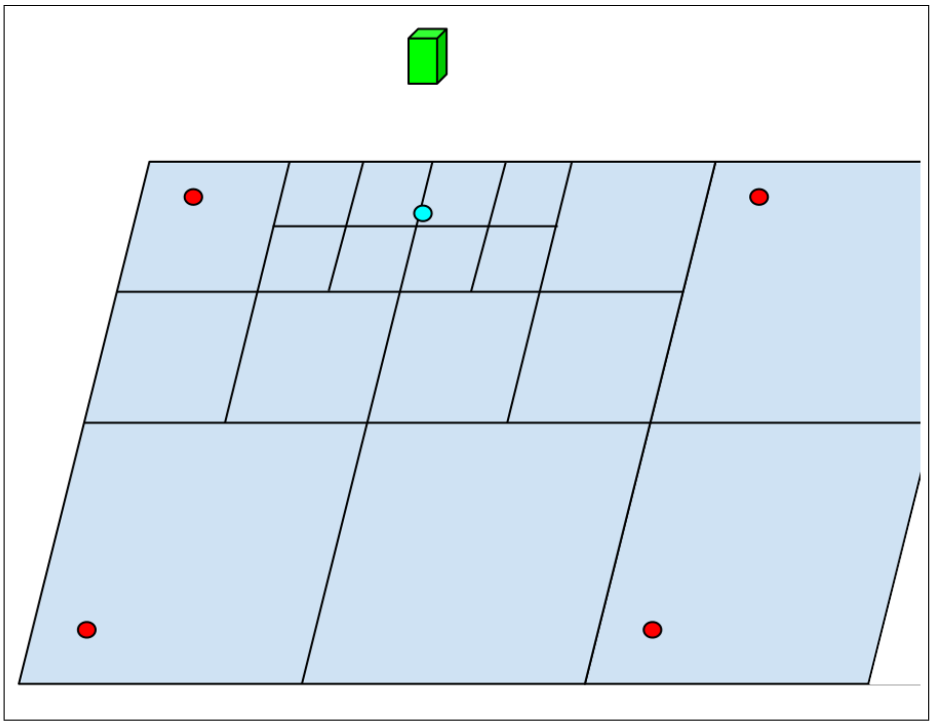


Figure 3.7: Temporary Grid layout

Image 3.7 represent the completed temporary grid. The green cube represents the camera and the blue dot on the grid is the camera position in the grid. The red dots represent the four points where the camera viewing frustum intersects the $y = 0$ plane. The tiles are loaded in sets of four and the grid is now built with several LOD and no gaps or overlaps. Tiles that fall outside the viewable area are not loaded, therefore no extra data was used from the W3DS server.

Chapter 4

OL3 in Action

In the previous chapter we described the implementation of the OL3 library, focusing on specific algorithms that were needed to handle the specific requirement of 3D data.

In this chapter, we will show how to get the existing code, how to set up and run a simple web page with a very basic map and finally, we show how to contribute to the OL3 implementation by sending pull requests.

4.1 Manage code development using Git

When several developers are working on the same project, code management tools are required to manage all contributions. These tools allow multiple users contributing to the code without overwriting, by accident, other contributions made by the different elements involved in the project. Also there is the possibility to track the complete history of the project's development, and what changes were made during the development process and by whom. The history capabilities of code management tools are a good provider of information to be included in the release notes, informing users of the changes made since the previous version.

We are using the Bitbucket online git repository to store our project.

4.1.1 Complementary code management tools

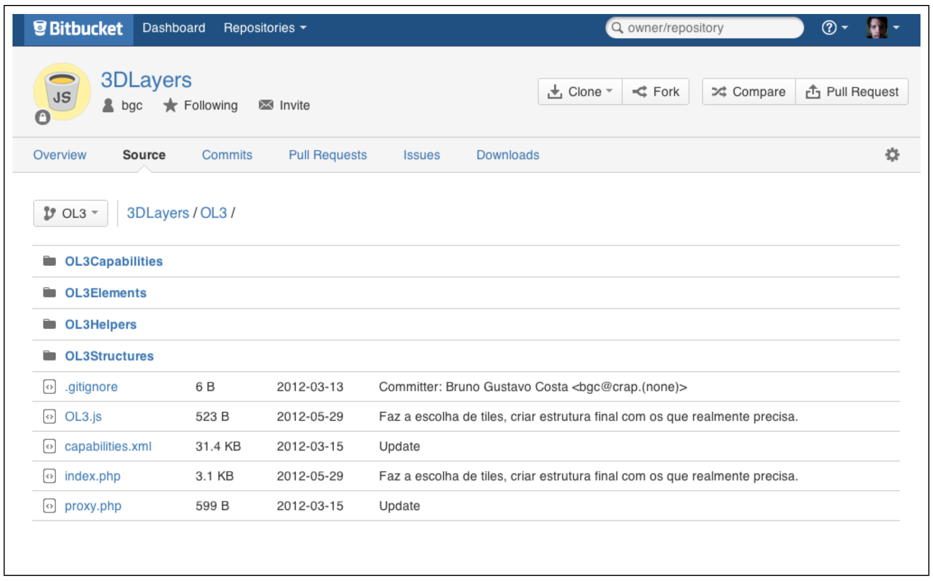
Besides code versioning and management, the online repository offers are complementary tools like viewing source code, commit history as well as issue tracking and wiki.

Issue tracking is a powerful tool for developers, since all involved in the project as well as users can submit issues found either when developing or using the software. These issues can then be ranked according to their importance, marking more serious errors as more important issues and therefore giving them more prominence over others. Issues can also be delegated to specific users or teams, avoiding redundant work by other developers.

The wiki provided by the repository allows the developers to create guides to help users getting started using the library, by providing tutorials, as well as create guides to explain other developers how they can help in the development, by explaining how the library is structured and how to create additional functions and objects.

4.2 How to get OL3 code

Downloading the existing code is quite simple. Everyone can get the code and the most simple way to do it, is to download the code using the web interface of the repository.



OL3Capabilities			
OL3Elements			
OL3Helpers			
OL3Structures			
.gitignore	6 B	2012-03-13	Committer: Bruno Gustavo Costa <bgc@crap.(none)>
OL3.js	523 B	2012-05-29	Faz a escolha de tiles, criar estrutura final com os que realmente precisa.
capabilities.xml	31.4 KB	2012-03-15	Update
index.php	3.1 KB	2012-05-29	Faz a escolha de tiles, criar estrutura final com os que realmente precisa.
proxy.php	599 B	2012-03-15	Update

Figure 4.1: Web interface of the OL3 repository

The code can also be downloaded using git, on the command line. In order to do so, one must have git installed in the computer and run the following command in the terminal:

```
git clone https://bgc@bitbucket.org/bgc/3dlayers.git
```

Listing 4.1: Obtaining the library source code

After making changes, the user can then commit the code, using the following command.

```
git commit -a
```

Listing 4.2: Making a commit

To send the source to the repository, the user must "push" the source. This is achieved by doing the following.

```
git push
```

Listing 4.3: Making a commit

IDEs, like Eclipse have plugins that support remote git repositories. In this kind of setup, a new project can be created with the downloaded code. Basically, these plugins are interfaces on top of git.

4.3 How to create a very basic map

OL3 has used OpenLayers as a guide, due to its maturity. Creating a simple map in OL3 is therefore quite similar to creating one in OpenLayers.

The first step is including the needed external libraries, namely jQuery and X3DOM.

```
 <script type="text/javascript" src="../jq/jquery-1.6.2.js"></script>
<script type="text/javascript" src="../x3/x3dom.js"></script>
<link type="text/css" rel="stylesheet" href="../x3/x3dom.css">
```

Listing 4.4: Including the libraries

The next step is including the OL3 library files. This is done with the following HTML.

```
<script type="text/javascript" src="OL3Helpers/namespace.js"></script>
<script type="text/javascript" src="OL3.js"></script>
<script type="text/javascript" src="OL3Structures/Point.js"></script>
<script type="text/javascript" src="OL3Helpers/Utils.js"></script>
<script type="text/javascript" src="OL3Structures/Vector.js"></script>
<script type="text/javascript" src="OL3Structures/Line.js"></script>
<script type="text/javascript" src="OL3Structures/Quad.js"></script>
<script type="text/javascript" src="OL3Structures/GridPoint.js"></script>
<script type="text/javascript" src="OL3Helpers/XMLHTTPRequest.js"></script
    >
<script type="text/javascript" src="OL3Structures/Tile.js"></script>
<script type="text/javascript" src="OL3Structures/Grid.js"></script>
<script type="text/javascript" src="OL3Structures/Camera.js"></script>
<script type="text/javascript" src="OL3Structures/BBox.js"></script>
<script type="text/javascript" src="OL3Structures/SimpleGrid.js"></script>
<script type="text/javascript" src="OL3Capabilities/Capabilites.js"></
    script>
<script type="text/javascript" src="OL3Elements/Layer.js"></script>
<script type="text/javascript" src="OL3Elements/Scene.js"></script>
```

Listing 4.5: Including the OL3 library files

Still in the HTML head, one must include the following JavaScript code:

```
function viewFunc(evt)
{
```

```
3      var vFpos = evt.position;
4      var vFrot = evt.orientation;
5      var vFmat = evt.matrix;
6      cena.camera.viewFunc(evt);
7  }
8  var cena;
9  var camada;
10 var bb;
11 function init()
12 {
13     bb = new OL3.BBox();
14     bb.setBounds([0,1,20,40]);
15     cena = new OL3.Scene(bb);
16     /*camada = new OL3.Layer(cena,"http://3dwebgis.di.uminho" +
17     ".pt/geoserver3D/w3ds?version=0.4&SERVICE=W3DS&REQUEST=",
18     "geoserver3D:dem_tiled_3d");*/
19     camada = new OL3.Layer(cena,
20     "http://localhost/3dlayers/OL3/geoserver" +
21     "-GetCapabilities_dual" +
22     ".xml",
23     "geoserver3D:dem_tiled_3d");
24         }
25 $(document).ready(function(){init();});
```

Listing 4.6: Setting up a scene

```
<body>
<div id="map"></div>
<script type="text/javascript">
function addIt()
{
    document.getElementById('vp_global').addEventListener('
    viewpointChanged', viewFunc, false);
}
</script>
</body>
```

Listing 4.7: The HTML body for using a scene

The `viewFunc` defined in lines 1 - 7 in listing 4.6 is created so that when the camera moves the camera information is obtained and sent to the OL3 camera object. This function is later added to the X3DOM viewpoint as shown in the function `addIt()` in listing 4.7.

Next variables are declared, `cena` for the scene, `camada` for the layer and `bb` for a bounding box. The bounding box is mandatory, yet at the current stage it's not used, so any values can be supplied as the corners. The `init()` function, as shown in listing 4.6 lines 11 - 24, will instantiate the object that represent the scene. The bounding box is created with fictional values and a new scene is created with the fictional bounding box and the default parameters, as seen in listing 3.7. After the creation of a scene, it is populated with a layer. The layer has the parameters parent, which is the containing scene, the url for the W3DS server being used and the layer name being requested from the W3DS server. The last line in the listing is to execute the `init()` function when the HTML document is ready.

If the layer exists in the server and is available to the user, then the scene will be drawn in the `<div id="map">` element.

4.4 How to contribute to the OL3 development

Using git to create a local copy of the library, one can contribute to the development. There are two possibilities; improving class functionalities and adding new classes.

Improving class functionalities can be made either by changing pre-existing functions in order to make them better or by adding new functions to objects, thus enabling them to perform more actions.

```
 exampleFunction : function()
{
  //print object's properties to console
  console.log(this.properties);
},
```

Listing 4.8: Function syntax

The correct syntax for adding functions to a class is presented in the previous listing. The function name comes first, followed by `: function()`, where additional parameters can be supplied in between the parenthesis. Inside the brackets is the code.

Creating a new class to be used by OL3, one uses the following:

```
OL3.Person = new OL3.Class(
{
    name : "",
    age : 0,
    sex: "",

    initialize : function (name, age, sex)
        {
        this.name = name;
        this.age = age;
        this.sex = sex;
    },
    getName: function()
    {
        console.log(this.name);
    },
    getAge: function()
    {
        console.log(this.age);
    },
    getSex: function()
    {
        console.log(this.sex);
    },
    greet: function()
    {
        console.log("hello, my name is "+this.name);
    }
});
```

Listing 4.9: Class syntax

Any OL3 class uses the OL3.Class template. This template uses the `initialize()` function as a constructor, and variables are either declared before the constructor by name or inside the constructor by name preceded by `this.`. Functions are created used the example given in listing 4.8. All variables and functions declared outside the constructor follow a structure, `name : value` , or `name: function(){}` , the comma acts in place of the semicolon, separating the various expression that are part of the object.

Chapter 5

Conclusions and Future Work

Over the course of the research for the thesis and the library development, a analysis of the current technologies and available libraries for scene navigation was made. The new capabilities of HTML5 and WebGL allow for the first time a plug in free and browser independent transition from the two dimensional web to a three-dimensional visualization.

Scene navigation libraries can benefit from this three dimension ability, therefore creating a more rich user experience as well as a more detailed data set. Ol3 intends to be an open source solution that makes use of the new available specifications and break free from the two dimensional restraints experienced until now. Using JavaScript, X3DOM, X3D and WebGL, this library provides the necessary tools to create a scene and obtain geometry from W3DS servers, this being either full models or tiled geometry.

5.1 Research

The research part was a study in the emerging specifications that allow the development of the library as well as existing open source libraries for scene navigation. The specifications studied where WebGL, X3D, X3DOM and W3DS.

WebGL, as shown in section 2.1, is an API that allows the use of 3D graphics in the browser. Its compatibility with DOM languages such as JavaScript, makes this a powerful tool to create plug in free and browser independent 3D visualizations.

X3D, which was discussed in section 2.2, is an open standards file format for the representation of 3D scene graphs, and according to the HTML5 specification is the method to declare 3D scenes. Inheriting from VRML and extending its capabilities, this file format extends its predecessor and allows users to use the more familiar XML syntax.

Through the use of components and profiles, as seen in subsection 2.2.1, it is possible for content creators to clearly define which features are required from X3D, therefore creating a more concise experience and visualization in different platforms. One setback in X3D is that no method exists to update or synchronize the X3D elements which allows only a single import of the scene and making it static. This is solved with the use of X3DOM

X3DOM creates a bridge between HTML5 and X3D, overcoming the update and synchronization issues between the two. Through this library, it's possible to create dynamic X3D scenes in the DOM, where users can change elements and the changes are reflected in the visualization. This scene manipulation is made through DOM scripting languages like JavaScript, leaving the role of X3D as a scene graph definition and rendering.

W3DS is a service that provides three-dimensional geodata, and allied with JavaScript and X3DOM, it is possible to create dynamic scenes which are loaded from one or several W3DS servers asynchronously and only when needed. This allows content creators to develop large amounts of data and store it in the server, as there is no need to load all the data set. Querying the service for details about the data and obtaining the actual data is quite simple, as the server implements several requests, as shown in section 2.6. These tools are the foundation for the development of the OL3 library.

Besides the study made about the technologies and specifications, a study about existing open source libraries for scene navigation was made. The targets of this study were OpenLayers and WebGL Earth.

OpenLayers is an open source JavaScript library which allows the visualization of geographical data in browsers. OpenLayers is quite simple to use, and, in an HTML document, one only needs to include the library in the document header and create a div element for the map. OpenLayers displays map images as tiles, and has a asynchronous loader in order to load the required tiles.

OpenLayers is a quite mature library, with a diverse set of tools and map support, yet it relies on a two-dimensional representation of the world. It is a good reference for tools and asynchronous data loading, but the navigation lacks a dimension.

WebGL Earth is a open source JavaScript library that allows users to create and navigate a globe using a browser. Still in the early stages, it implements a three-dimensional camera navigation system as well as loading data asynchronously, yet the scene only offers a globe with modifiable textures. There is no sense of height and the textures are obtained from WMS servers or can be defined by users. Although a nice step in breaking free from the restraints of two dimensional web, the data is still two dimensional and limited set of tools.

5.2 Implementation

After the research was made and an understanding of the specifications and libraries required to create OL3 was achieved, the next step was the implementation.

The starting point the was implementation of a name space to avoid collision with other JavaScript libraries and creating a construct so there is a similarity with classes as present in Java and C++. After a structure was defined, the next phase was setting up asynchronous communication with the W3DS server, in order to obtain information and data from the Web Service. The main information provider from the W3DS server is the `GetCapabilities`, where one can find all information about the W3DS server and the data contained within. The final requirement was the creation of a bounding box class. Bounding boxes are present in WMS servers and W3DS also uses these objects. Due to their importance in obtaining data and limiting geodata extents, OL3 needs this class implemented before heading into other objects.

The Scene class defines the main OL3 object, which represents a 3D scene. All layers and the camera are a part of the Scene and use it as an access point to all required data from other objects included in the Scene. This class is also responsible for accessing the HTML DOM and creating the necessary elements for the X3DOM view port to be rendered in. The Scene class was implemented successfully and handles all required functions.

The Camera was problematic, as there were three different approaches implemented. The need for constant information about camera position and rotation as well as which coordinates delimited the viewing frustum made this implementation difficult. The calculation of the viewing frustum used a large amount of functions and the final result had some issues when swiveling the camera and had to be abandoned due to the remaining time left for the library development was not enough to resolve them. The next approach was the one described in subsection 3.2.2, Emitting Rays, and proven more accurate and with less lines of code needed. This class is implemented successfully and the basic functions required are present.

The Layer class is a complex one. Different types of layers in the W3DS server, tiled and untiled, create a need of versatility in handling requests and data management. The Layer class went through many versions, and is yet to be completed. In the initial phases, untiled layers where handled just like tiled layers, in order to keep track of loaded geometry, yet at the current state, there is no method to handle scene requests. The scene request method was dropped as to implement a better tiled layer management, using the camera properties and the grid object, as seen in subsection 3.2.3. This new tiled layer method is nearly finished as there is a grid calculation for needed tiles and there are no requests for already loaded tiles.

One issue left open for the layer object and regarding tile management is that older tiles which are not needed are still kept in the scene and not discarded. Also there is an issue regarding the overlap of tiles. Since older tiles are not discarded when a new tile of different LOD is required it will overlap the older tile present. This is an important issue that needs special attention in order to make the Layer class more close to a final version.

Other issue left to resolve is handling of the untiled layers, which can be either as a tiled layer with a fixed size and no LOD or a completely different approach using small bounding boxes.

One final requirement for the Layer class is HTML controls for visibility and implement functions to toggle the Layer visible or invisible. This is yet to be implemented, but a viable solution has been studied.

5.3 Future Work

For future work in OL3, the most important elements in the list is closing the issues the library currently has and were referenced in the previous section.

After completing the pending tasks, one option for future work is incorporating the Underscore JavaScript library to handle all tasks that require array iteration. Since tiled layers can have many arrays and each time the camera moves there is a need to iterate them in order to manage the tiles, using Underscore can make this task faster.

To make the library more useful another option for future work would be the creation of another type of Layer. This new object would be designed to handle buildings and allow a 3D visualization of routing. Just like OpenLayers there are implementations of two dimensional indoor routing available, yet these could benefit from a three-dimensional representation as not all navigation is made in two dimensions.

5.4 Conclusion

The library has the main features implemented, despite the issues reported in the Layer class. The most difficult task was the Camera implementation due to the errors encountered, yet the solution found was able to close the issue and required much less functions than the one that had been previously proposed.

The library does need more work, in order to be usable by the general public and to be considered complete and functional, but the development made until now is promising.

Bibliography

[1] http://www.webglearth.org/api @ www.webglearth.org. [Accessed September 16th, 2011].

[2] WebGL Specification - Editor's Draft. `https://www.khronos.org/registry/webgl/specs/latest/`. [Accessed November 3rd, 2011].

[3] Brutzman, D., and Daly, L. *Extensible 3d graphics for web authors.* 2007.

[4] Caldwell, D. Unlocking the Mysteries of the Bounding Box. *ALA Map and Geography Round Table. . . .* (2005).

[5] Fernandes, A. R. View Frustum Culling @ www.lighthouse3d.com. `http://www.lighthouse3d.com/tutorials/view-frustum-culling/`. [Accessed May 17th, 2012].

[6] Jung, Y., Behr, J., and Graf, H. X3DOM AS CARRIER OF THE VIRTUAL HERITAGE. *Integration The Vlsi Journal* (2001), 1–8.

[7] Rocha, J. G., and Oliveira, N. C. W3DS Open Source Implementation. [Accessed May 31st, 2012], 2012.

[8] Samyn, K. Line plane intersection - GNU Octave - a knol by Koen Samyn. [Accessed June 5th, 2012].

[9] Sunday, D. Bounding Containers for Polygons, Polyhedra, and Point Sets (2D & 3D).

[10] WESSTEIN, E. W. Line Plane Intersection. `http://mathworld.wolfram.com/Line-PlaneIntersection.html`. [Accessed June 7th, 2012].