University of Pécs
Faculty of Sciences
Institute of Geography

Comparison of Web Mapping Libraries for Building WebGIS Clients

Written by: Gábor Farkas Advisor: Titusz Bugya

2015.

Table of Contents

I. Introduction

1.1 Introduction to Study

WebGIS (Web based Geographic Information System) is a quite frequently used term nowadays in the GIS world. It originates from the 1990s, with the mass uptake of the Internet. Cartographers, geographers, and spatial analysts began to experiment with this new technology, and how to utilize it to share spatial information with the public in a sophisticated manner. Until then, spatial data was only available in a local scope. “GIS analysts would access data from powerful PCs that were often connected to a central file server. Specialized software was required to view or manipulate the data, effectively narrowing the audience that could benefit from the data” (S. Quinn, J. A. Dutton 2014).

In the recent twenty years with the rapid advancement in Web technologies, spatial data became available in a more global context. Dedicated map servers provide spatial data to the world originating from big commercial companies (e.g. ESRI, Google) to open-source organizations (e.g. OpenStreetMap, Mapbox). This enormous growth in freely available spatial data, and the promising technological advancement led to today's GIS phenomenon, where Web based GIS applications slowly surpass offline GIS solutions. This tendency is best described by how even ESRI — “which has a 43 percent share in the GIS market” (ESRI 2015.) — makes increasing investments in its online services.

To completely understand this study, one must have some basic experience in GIS, and Web programming. The readers of this study are assumed to be familiar with using APIs, have some basic understanding about Web standards and technologies, and have some experience in JavaScript programming. The study is focused on the various open-source WebGIS technologies, therefore basic geographic and GIS concepts (e.g. projections), as well as closed source products, are out of the scope of the study.

1.2 Actuality of Study

The recent ascendancy of Web 2.0 caused an increased demand for dynamic Web content. People realized, they can include Web applications in their workflow. Developers started to create frameworks to support dynamic contents, and automation. Contents became integrated into RDBMS systems instead of serving static files directly from the file system. These tendencies made an impact on the GIS world, too. Different frameworks have been developed with the sole purpose of helping GIS professionals build web mapping systems and ease publishing process.

Initially, WebGIS systems only supported visualization. Users could browse interactive maps, and see their content in details. As technology advanced, these WebGIS applications gained more capabilities. They could draw vector features, query layers, even make coordinate transformations. As browsers became more powerful, they got capable of resolving such complex computational problems, priorly only desktop applications could do. These days, with proper applications, and a strong back end, they can be an equivalent to modern desktop GIS applications. “The pieces of a complete WebGIS application are out there, we just have to assemble it” (T. Bugya ex. verb. 2015).

1.3 Relevance of Study

The first of the WebGIS maps in the 1990s were "served out by newborn versions of software such as Map Server and Esri ArcIMS" (S. Quinn, J. A. Dutton 2014). They were slow, pixelated, but they were interactive. One could look at them as the pioneers of WebGIS. After a decade later, the revolution of the modern WebGIS started by Google. In February, 2005, they published Google Maps, which, as described by E. Schütze (2007), got reverse engineered immediately. As a response, Google dropped the plan for a commercial product, and published its first API of Google Maps. In later 2005, MetaCarta created OpenLayers in collaboration with one of the reverse engineers, Phil Lindsay.

In the last decade, dozens of web mapping libraries and frameworks got developed. Most of these initiatives are deprecated by now. Some of them are officially abandoned, while some of them just doesn't have any recent development. G. Carrillo (2012) compared 44 open source WebGIS applications, from which 19 were already abandoned. The comparison can be seen in Figure 1. Nowadays, the most successful WebGIS projects emerged from the masses, and most of them got incubated by the Open Source Geospatial Foundation (OSGeo) as an official software project. OSGeo projects are reliable softwares with constant development, and a stable contributor network.

However, the most widely used WebGIS applications are not mandatorily OSGeo projects. For example, in the scope of web mapping libraries, the two main open source competitors are OpenLayers and Leaflet. Only OpenLayers is an OSGeo project, but many favor Lefalet over OpenLayers. Contrary to OpenLayers, Leaflet is a lightweight library with extended visualization capabilities. It was developed to be small, modular, and easily deployable. To sum up, in WebGIS, — similarly to this study — one shouldn't take only OSGeo projects into consideration, but every popular alternative based on public demand.

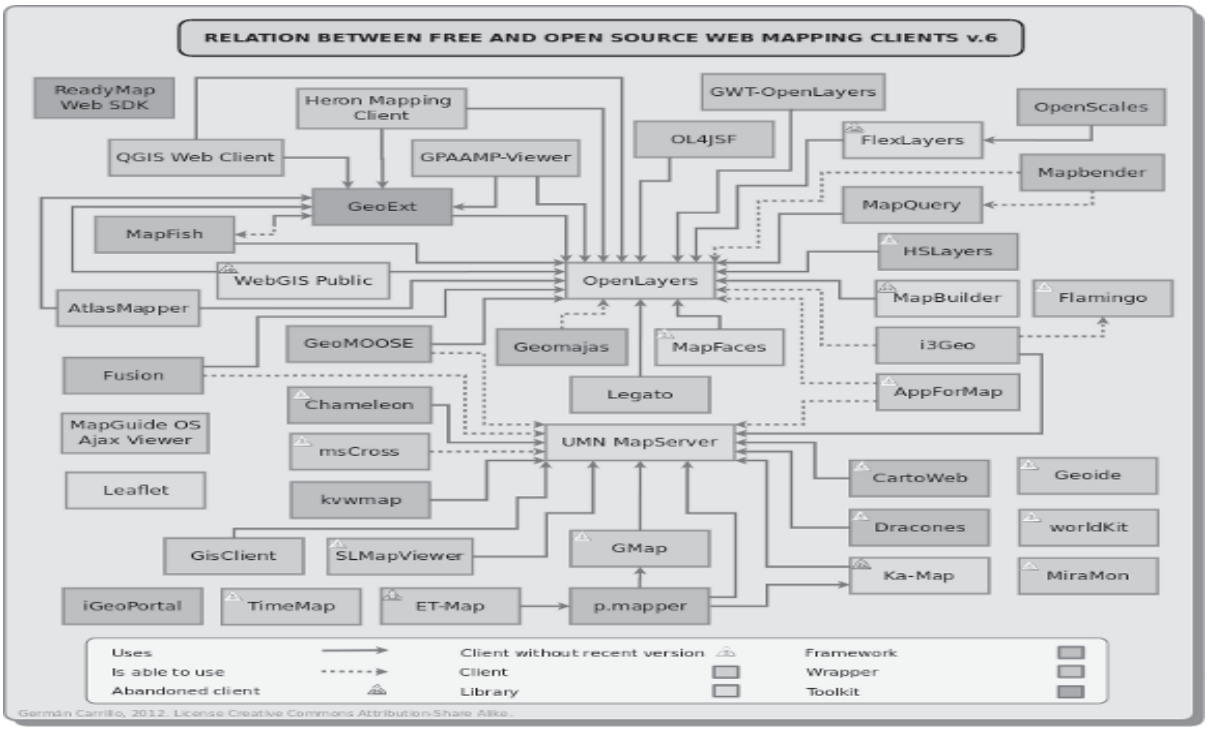

Figure 1: Relation between WebGIS Clients. Copyright 2012 by G. Carrillo. Reprinted with permission.

With the raising popularity of open source mapping technology, the Open Geospatial Consotrium (OGC) came up with WebGIS standards, which tend to be unify the web mapping formats. This standardization led to similar data-exchange methods in OGC compliant map servers, on which applications could be built. Now, every determinative open source WebGIS application use these standards.

One should not overlook the tendencies in the cloud computing market, when deliberating on the importance of WebGIS. Cisco (2014) states in its forecast, the installed workloads (physical or virtual machines) will share a 78% - 22% proportion in favor of cloud data centers by 2018. In 2013, this ratio was 53% - 47% to cloud data centers over traditional ones. These statistics can lead to the assumption, that companies start to realize, it is more economical to invest in a cloud service, then building up, and maintain their own IT network. Cloud services can be found in the WebGIS world, too. Some of them (e.g. CartoDB, MapBox) are open source applications, one just have to pay for the hosting, and the related services.

To sum up the previous thoughts, and cut to the point, the public and business demand in decentralized workflows, and global self-representation became evident over the past decade. This phenomenon makes an impact on the GIS market, too. With the advancement in Web technologies, the great open source web mapping projects, the OGC standards, and the high demand for publishing interactive spatial data, WebGIS became a subject of importance. It should be studied and taught (as it is in some Hungarian institutes, too), developed, and built upon. If “our advanced economies increasingly oriented around services” (H. Chesbrough 2010), WebGIS is a way to go.

II. Review of the Literature

2.1 OGC Services

“The Open Geospatial Consortium (OGC) is an international industry consortium of 510 companies, government agencies and universities participating in a consensus process to develop publicly available interface standards” (OGC 2015). OGC is a complex association, which has four main programs, from which the Standards Program is in the study's range of interest. The Standards Program is responsible for making and adopting existent standards for encodings and interfaces, maintaining and periodically revising them, and writing proper technical documentations about them.

OGC's Web standards are commonly referred as OGC Web Services (OWS). These services are commonly implemented into open source map servers. However, on the front end, only three of them have great significance. Because the browser can only display color-encoded images, Web Coverage Service (WCS) driven data is out of interest. WCS is an interface for exchanging attribute-encoded raster data (e.g. DEMs) usually in GeoTiff format. They need to be paletted, and served as a Web Map Service (WMS) layer by the back end server. Also, some kind of Web Processing Service (WPS) implementation can be found in most of the map servers, but in front end libraries, only OpenLayers 2 has native support for it, yet.

There are three crucial OWS standards, which need to be supported in every decent web mapping application. These are the WMS, the Web Map Tile Service (WMTS), and the Web Feature Service (WFS). OWS implementations have one common property. All of them must support a capabilities request, which returns “metadata about the capabilities provided by any server that implements an OWS interface Implementation Specification” (OGC 2010). Along with a non OGC standard format of XYZ (not to be confused with the similarly named raster format), these essential formats get described in more details below.

2.1.1 Web Map Service

OGC's WMS is a basic standard for displaying raster data in WebGIS applications. It comes in two versions, 1.1.1 and 1.3.0. There are some differences in accessing a map with different WMS versions (switched up lat/lng order, coordinate system denomination). A WMS compliant server can answer up to four WMS requests. The two mandatory requests are GetCapabilities and GetMap. The GetCapabilities request returns the metadata of the server's WMS capabilities, as discussed above. “As a response for a GetMap request, the server generates a map for a given area. [...] We get an image as a result” (T. B. Nguyen 2014). This request is more complicated, because it has seven request specific parameters, and further six optional ones, as it can be seen in Figure 2.

Request parameter	Mandatory/optional	Description
VERSION=1.3.0	M	Request version.
REQUEST=GetMap	M	Request name.
LAYERS=layer_list	M	Comma-separated list of one or more map layers.
STYLES=style_list	M	Comma-separated list of one rendering style per requested layer.
CRS=namespace:identifier	M	Coordinate reference system.
BBOX=minx,miny,maxx,maxy	M	Bounding box corners (lower left, upper right) in CRS units.
WIDTH=output_width	M	Width in pixels of map picture.
HEIGHT=output_height	M	Height in pixels of map picture.
FORMAT=output_format	M	Output format of map.
TRANSPARENT=TRUE|FALSE	O	Background transparency of map (default=FALSE).
BGCOLOR=color_value	O	Hexadecimal red-green-blue colour value for the background color (default=0xFFFFFF).
EXCEPTIONS=exception_format	O	The format in which exceptions are to be reported by the WMS (default=XML).
TIME=time	O	Time value of layer desired.
ELEVATION=elevation	O	Elevation of layer desired.
Other sample dimension(s)	O	Value of other dimensions as appropriate.

Figure 2: The parameters of a GetMap request. Copyright 2006 by Open Geospatial Consortium Inc. Reprinted with permission.

An ordinary GetMap request consists of the mandatory options and is similar to the following URL:

```
http://www.agt.bme.hu/cgi-bin/mapserv?
map=/var/www/html/gis/wms/eu_dem/eu_dem.map&SERVICE=WMS&
      VERSION=1.1.1&
      REQUEST=GetMap&
      LAYERS=mo_eov_szines&
      SRS=EPSG:23700&
      BBOX=400000,45000,950000,380000&
      WIDTH=450&
      HEIGHT=265&
      FORMAT=image/png
```

Figure 3 describes the server's response to the previous GetMap request. There are two important notes to be mentioned about the request. The version number is 1.1.1, therefore the projection needs to be denominated with the key SRS. In 1.3.0, the correct key is CRS. It has to be provided by its Europian Petroleum Survey Group (EPSG) code. Also, the bounding box needs to be defined in the requested coordinate system (in this case HD72/EOV), in a counter-clockwise direction, from West to North.

The optional requests of a WMS implementation are GetFeatureInfo and GetLegendGraphics. The GetFeatureInfo request returns available feature informations from the source data on given coordinates calculated by the provided pixel coordinates. If the source is vector data, it returns features. If it is WCS data, the response will contain the pixel value from the original dataset, while if it is an image, the returned data consists of RGB band values for the given pixel.

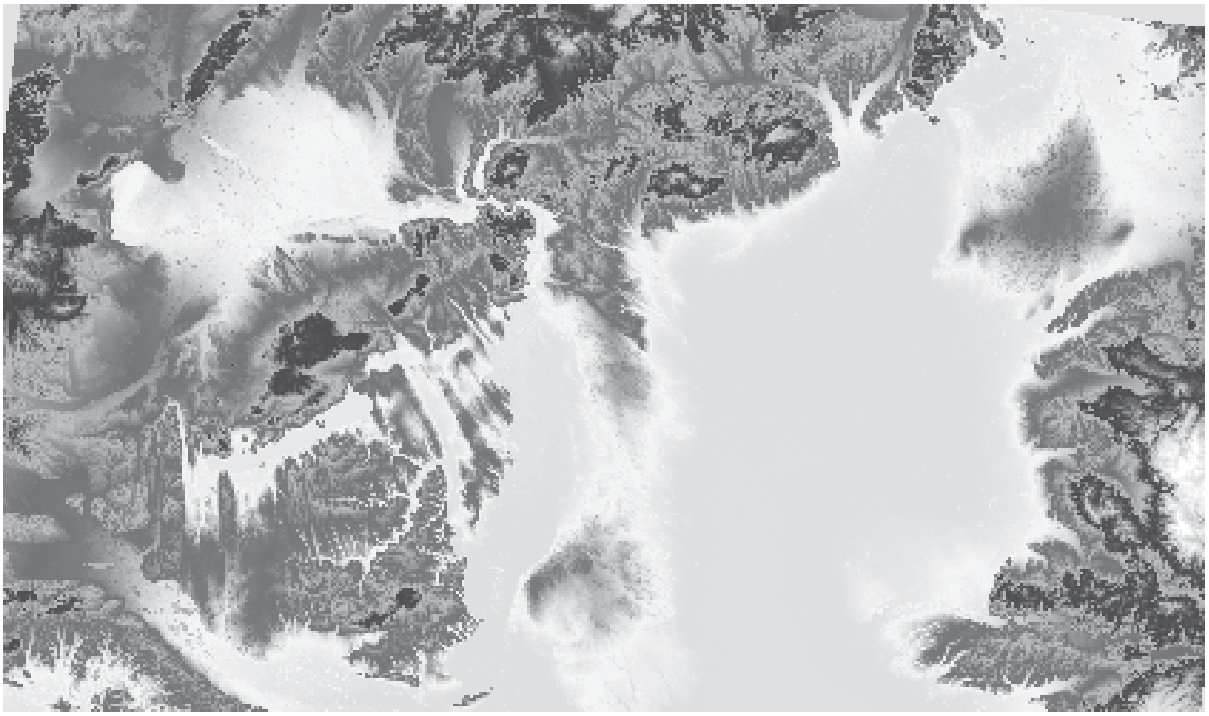

Figure 3: Response from a MapServer with the sample request.

The GetLegendGraphics is a version 1.1.1 capability. The 1.3.0 specification no longer confers it, but it remained in map servers. The GetLegendGraphics request returns an image with the layer legend, but only, if the WMS implementation of the map server is SLD (Styled Layer Descriptor) compatible.

2.1.2 Web Feature Service

OGC's WFS has a different, more complicated structure than WMS, because it returns spatial data in an XML encoded format. This way, clients can render the response as vector data. This allows applications to do spatial operations, queries, filters, and "define the styles of each feature on the client side" (M. Gede 2014).

WFS has four versions, from which two (1.1.0, 2.0) are widely used (despite the fact, 2.0 is depreciated, and 2.0.2 is promoted to use by OGC). There are two important request in WFS besides of GetCapabilities. The DescribeFeatureType request returns the metadata associated with a feature type (e.g. layer) provided by the user. The GetFeature request returns all of the features, which are evaluated true from the request parameters. It has three mandatory arguments from which a request looks similar to the following:

```
http://demo.opengeo.org/geoserver/wfs?
      VERSION=2.0&
      REQUEST=GetFeature&
      TYPENAMES=states
```

The request above returns all of the features from the states layer in the default GML3 format. There are several more request types and optional parameters, so the study only covers a limited set of those options. The SRSNAME parameter defines the coordinate system in which the features are requested. The COUNT (MAXFEATURES in version 1.1.0) limits the number of the output features.

The BBOX property only requests features from a user defined bounding box. The FILTER property can be used to define custom spatial or/and attribute filters, which get evaluated on the server. The BBOX and FILTER parameters are exclusive in version 1.1.0. “In lieu of a FEATUREID or FILTER, a client may specify a bounding box” (OGC 2005). In version 2.0, one or more BBOX parameter can be nested in a filter. The FEATUREID is a version 1.1.0 specific parameter, which requests a single feature from the server with the provided unique id.

With the PROPERTYNAME parameter, one can restrict the list of attribute names to receive with the features. Finally, the OUTPUTFORMAT takes care of the formatting of the result. The default format is a versioned GML in both of the WFS versions, while the rest of the formats are left to be absolutely vendor specific. This can be important, when a client has to process the result. The browser can parse an XML document, but GeoJSON is a shorter, more laconic format for exchanging spatial data with their attributes. GeoJSON is a standardized JSON object, which “is faster and uses fewer resources than its XML counterpart” (N. Nurseitov et. al. 2009).

2.1.3 Web Map Tile Service

OGC's WMTS is similar to the WMS service, although it provides a more scalable method to serve rester data. It is designed to server pre-generated tiles from the server, from which the most frequently used ones are usually stored in the memory to reduce latency. WMTS's predecessors were OSGeo's two specifications, WMS-C (Web Map Service - Cached), and TMS (Tile Map Service). OGC responded to the demand for a tile service with WMTS. Currently, TMS and WMTS are the mostly used tile service specifications in WebGIS.

TMS can be considered as a simplified WMTS. TMS has less capabilities, therefore needs less configuration. It has one major difference from WMTS however, as it can be seen in Figure 4, TMS tile indices are following the rules of the Cartesian coordinate system. WMTS tile indices on the other hand, are starting from the upper-left corner of the tiled source, and cannot have negative values.

The service has three request, and much parameters, from which only a few are in the interest of the study. One of them is the GetTile request, which similarly to other OGC services, returns a tile, when properly argumented. The other request is GetFeatureInfo, which as in WMS, returns the attributes of a given pixel. This is a further addition compared to TMS. The following sample URL requests one single tile from a WMTS:

```
http://demo.opengeo.org/geoserver/gwc/service/wmts?
REQUEST=GetTile&
VERSION=1.0.0&
LAYER=ne:NE1_HR_LC_SR_W_DR&
STYLE=&
FORMAT=image/png&
TILEMATRIXSET=EPSG:900913&
TILEMATRIX=EPSG:900913:6&
TILEROW=22&
TILECOL=35
```

These properties are the mandatory ones in a WMTS GetTile request. The only version is 1.0.0, consequently the value of the VERSION parameter is axiomatic. As tiles are pre-generated images, they can be present in different projections, and styles for every layer. In this case, there aren't any styles besides the default one, but as OGC (2010) states, the STYLE parameter is a mandatory one, it is included in the example as an empty argument. The FORMAT parameter specifies the output format, which ordinarily is an image.

The TILEMATRIXSET parameter defines a set of tiles in a layer, which is usually defined by the set's projection. Every set of tiles contain tiles of different scales. Tiles pertaining to a given scale are gathered in a particular TILEMATRIX. In a certain scale, every tile can be accessed by its indices with the TILEROW and TILECOLUMN parameters.

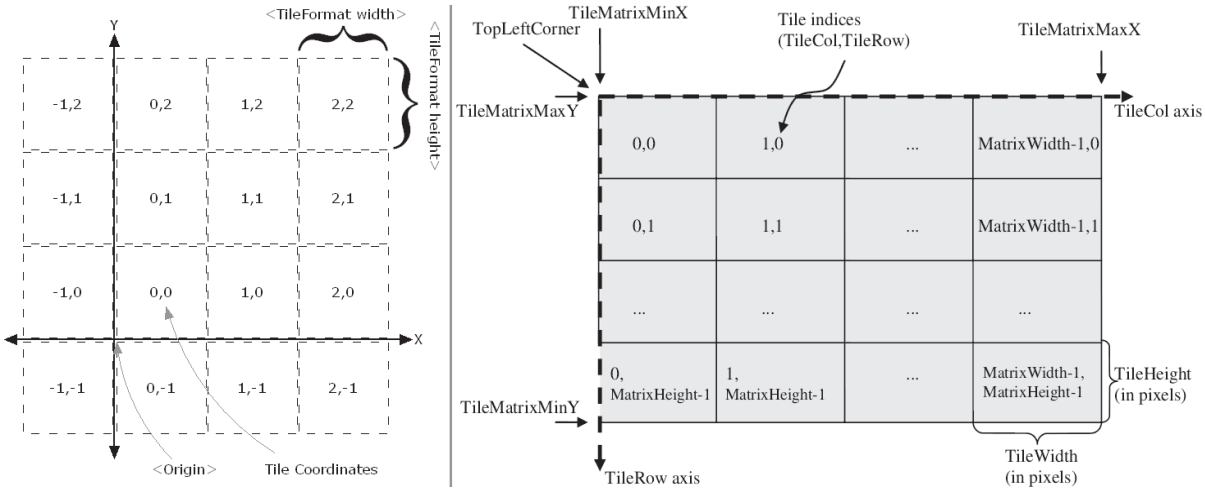

Figure 4: TMS tile space (left) and WMTS tile space (right). Copyright 2006 by P. Ramsey, and 2010 by Open Geospatial Consortium Inc. Adapted with permission.

2.2 Other Web Mapping Formats

Obviously, there are other web mapping formats, than the ones covered by OGC specifications. This section goes through some of the now well known, but not thoroughly trivial web mapping formats, as well as some of the cutting edge specifications, which can increase a web mapping application's performance and capabilities, if used wisely.

2.2.1 XYZ Format

The XYZ is a is a tiling scheme, similar to TMS. The name of the format comes from OpenLayer's nomination, but originally it is a tiling scheme for OpenStreetMap's Slippy Map application (which is also used by MapQuest, Stamen, CatroDB, just to mention a few), hence it is officially called Slippy map tilenames. The XYZ refers to the path of the tiles created by this scheme. “Each zoom level is a directory, each column is a subdirectory, and each tile in that column is a file” (OpenStreetMap Contributors 2015).

This format does not have any other capabilities, than serving tiles. In reality, tiles does not have to be really served, as they are directly accessible. The format has one serious restriction: as it was developed to serve OpenStreetMap's needs, it only supports EPSG:3857. The tile path scheme for the standard OpenStreetMap layer is http://[abc].tile.openstreetmap.org/z/x/y.png, where [abc] is one of the three subservers (for load balancing), z stands for the zoom level (scale), x represents a column, and y.png is the resource image named after the row number. A simple request looks similar to the following:
`http://a.tile.openstreetmap.org/15/18041/11647.png`

2.2.2 UTFGrid

UTFGrid or hit grid is a specification made by Mapbox. The concept of the standard is to create a JSON based ASCII art raster format, where every pixel is represented by an ASCII character. Every character is mapped to a set of attributes, so the application can look up attribute information without contacting the server, or loading vector data into memory.

As Mapbox (2013) states in the UTFGrid specification, coming from the Unicode character handling in JSON, it can handle a maximum of 65501 different keys (65535 - 32 control characters - " - \). Each UTFGrid tile represents a 64x64 grid of ASCII art, and attribute data linked to every different ASCII key. This way, older browsers and low capacity devices can also give a good performance, when attribute look up is needed.

The tiles can be served as a TMS, or with XYZ tile scheme. Tile generators are available as open source third party softwares, or as a native feature in Mapnik from version 2.0.0. One drawback can be highlighted: the UTFGrid tiles need to be pre-generated and stored on the server, similar to regular raster tiles, therefore they can increase performance on the cost of disk space.

2.2.3 TileJSON and Vector Tile

TileJSON is also a specification from Mapbox. It is a JSON based markup language, which describes TMS or XYZ layer configurations. With TileJSON files, one can make a tile based layer configuration portable, and reuse it anytime even in a different environment.

TileJSON can handle any TMS or XYZ compatible resource. It can be effectively used to configure the display of raster and vector tiles, associate boundaries, default center, legend, zoom levels, attribution or even an UTFGrid to them.

Vector tile is a relatively new specification from Mapbox, which stores vector data in a tiled JSON format. It is an experimental technology, but it shows promising results. The intend behind this format is to serve vector data with greater performance, than WFS. For now it can only support Web Mercator (EPSG:3857) tile generation (Mapnik 3.x delivers this feature), and they have contradictory utilizations. “Some vector tile sources are clipped so that all geometry is bounded in the tiles, potentially chopping features in half. Other vector tile sources serve unclipped geometry so that a whole lake may be returned even if only a small part of it intersects the tile” (OpenStreetMap Contributors 2014).

2.2.4 TopoJSON

TopoJSON is a vector storage format originally created by Mike Bostock, who is behind the famous Data Driven Documents (D3) library. TopoJSON is similar to the GeoJSON standard, but it eliminates redundancy with using shared arcs. It also quantizes coordinates with a scale, and an offset, therefore every coordinate can be stored as an integer.

"As a result, TopoJSON is substantially more compact than GeoJSON. The above shapefile of U.S. counties is 2.2M as a GeoJSON file, but only 436K as a boundary mesh, a reduction of 80.4% even without simplification" (M. Bostock 2012). With this format, clients can be loaded with translating and rendering vector data from TopoJSON. This eventuates in a better load optimization, especially if the server sends out vector tiles. Serving TopoJSON in a WFS context is to be implemented by map servers.

2.3 Web Mapping Libraries

There spectrum of web mapping applications is really wide. From servers to clients on a horizontal level, from libraries to frameworks on a vertical scale, there are a variety of open source applications to choose from. P. Ramsey (2007) defined three classes to loosely categorize web mapping applications. These were the frameworks, the toolkits, and the servers. The toolkits are modular softwares, which can be easily integrated into applications, while the frameworks are easily deployable applications, which can be customized, rather than integrated.

To go further with this classification, the study differentiates toolkits and libraries. The main difference between the two term is, that libraries are the foundation-stones of building web mapping clients, while toolkits either extend their capabilities, or merge them with other libraries, thereby creating a cross-product, which can be used to build rich web mapping applications.

The scope of the study only involves web mapping libraries from amongst many different web mapping technologies. R. Donohue et. al. (2012) made a comparison between web mapping applications. Figure 5 shows their result, which presents 35 web mapping applications in the function of their capabilities. The darkest shade implies the feature is included in the corresponding application with a native support, while the lightest means, the feature is impossible to implement in the application (i. e. needs irrational amount of coding).

They highlighted D3, GeoMOOSE, Google Maps API, MapServer, OpenLayers and OpenScales as web mapping applications suggested to use. D3 is the richest library of them, but it is designed to create beautiful and interactive static maps. It can be integrated into other libraries, like Leaflet, but it cannot be used to create a WebGIS application. GeoMOOSE is a framework built on OpenLayers. It is a great platform, but it is not really customizable. Google Maps API is unfortunately not open source. MapServer is a server application, and OpenScales is Flash based, which is a technology slowly replaced by JavaScript.

For further analysis, the study compares the most promising web mapping libraries to build a WebGIS application on. These libraries are OpenLayers 2, OpenLayers 3, and Mapbox JS. Mapbox JS extends Leaflet with essential functions, and as its creator is now a member of the Mapbox team, their joined efforts gives the library an even greater potential.

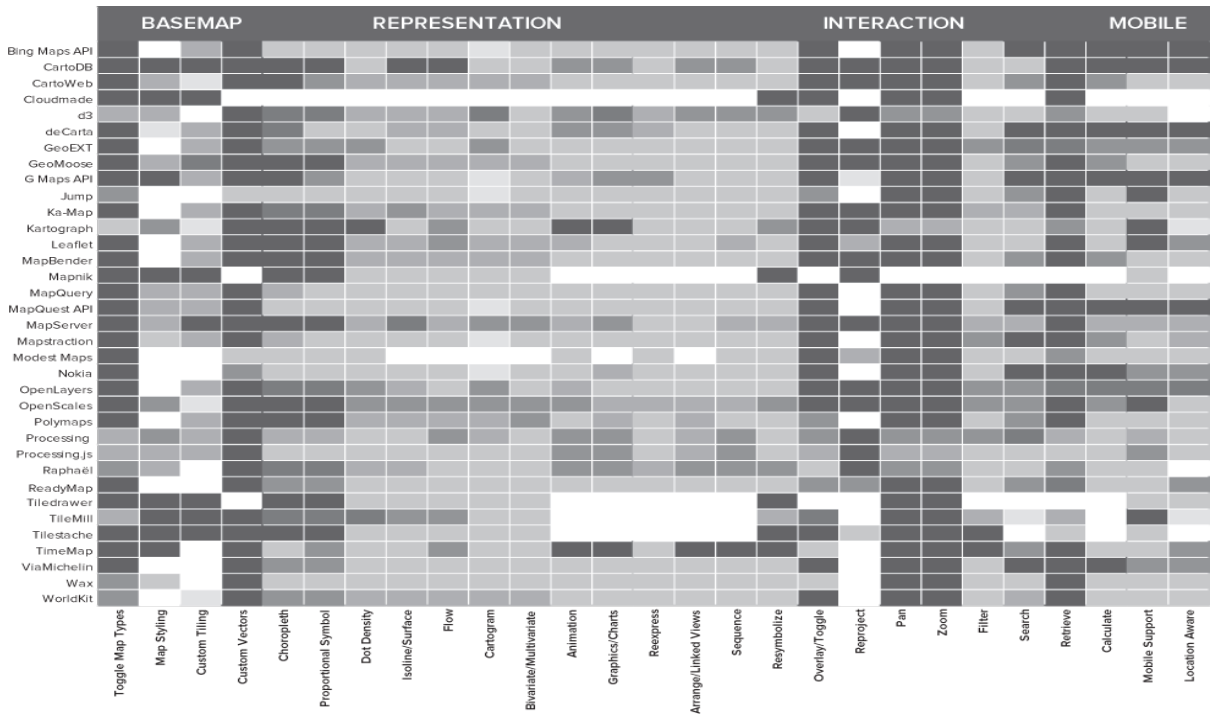

Figure 5: Comparison of web mapping applications. Copyright 2012 by R. DONOHUE et. al. Reprinted with permission.

2.3.1 OpenLayers 2

OpenLayes 2 was created in 2005 with the fund of MetaCarta Labs. The first release was in 2006, and in 2007 it became an OSGeo project. The library is written in JavaScript, therefore it uses an object-oriented approach. It is a stable, closed project with great capabilities. It won't undergo any change in its functionality, thus it can be utilized as a stable basis for a web mapping application. However, this can be considered as a fundamental drawback, too. Because it is not in development, it cannot keep pace with the evolution of Web technologies.

OpenLayers 2 has a structured build with different objects representing different functionalities. The main objects can be found in every web mapping library. They are the map object, the layer object, and the control object. The library's great flexibility lies in the subclasses. There are a wide variety of layer formats, support for displaying, editing, and saving vector data with WFS, custom projection support, and numerous different controls.

The library uses DOM (Document Object Model) elements to render data to an interactive map. There are three types of renderer OpenLayers 2 can use: Canvas, SVG, and VML. They affect the rendering mechanism of vector features. The default vector renderer is SVG.

Figure 6 shows the relation of some of the more important classes, especially layer classes and their relationship in an UML diagram. It is important to mention, the layer options are stored as separate subclasses in OpenLayers 2. The Events class is also very important, as there are numerous events added to most of the class, and some of them are very useful, if not vital.

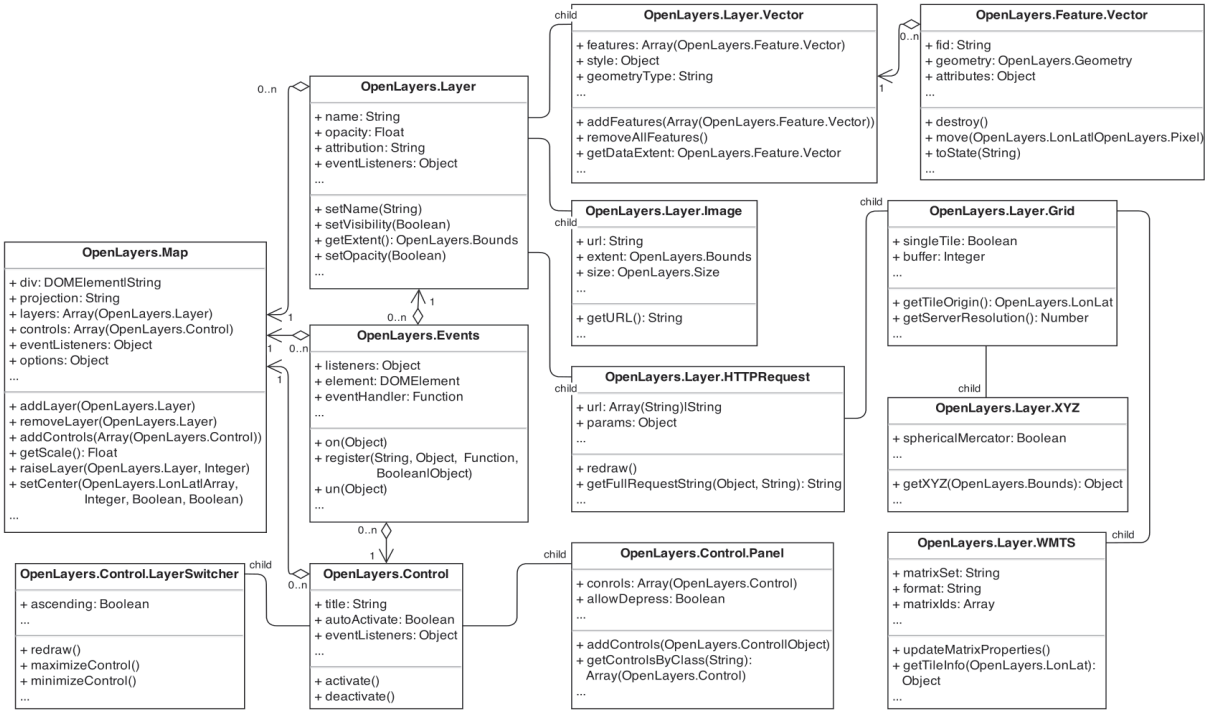

Figure 6: Partial UML diagram of OpenLayers 2 based on the OpenLayers 2 API. Copyright 2013 by OpenLayers Contributors. Adapted with permission.

A basic workflow with OpenLayers 2 starts with creating one or more map object. Every map object can possess zero or more layer objects and control objects, albeit one or more is recommended. A layer can be a raster layer of some source (tile or image), or a vector layer. Vector layers can be filled with features described in the internal format. There are different format objects, which can be used to read vector data from the corresponding source format.

The OpenLayers 2 API can be found at http://dev.openlayers.org/releases/OpenLayers-2.13.1/doc/apidocs/, and is encouraged to look up and use, if something is not clear from the study's examples.

2.3.2 OpenLayers 3

The first release of OpenLayers 3 dates back to 2014. It is a fresh project with a better and more well-considered structure. It is under constant development, newer versions are coming out frequently. Nevertheless, the library is in an early development status, therefore migrating a complete WebGIS application takes serious considerations, and many custom functions to rewrite some OpenLayers 2 functionality.

The library has a new code base, and is a “fundamental redesign of the OpenLayers web mapping library. [...] OL3 has been rewritten from the ground up to use modern design patterns” (OpenLayers Contributors 2014). The new code is made with Google's Closure Library. This way “the developers can concentrate on mapping functionality, and be sure that the underlying software is well-tested and cross-browser” (OpenLayers Contributors 2014).

The primary renderer in OpenLayers 3 is Canvas. The library loads all of its elements on one Canvas element, optimizing rendering speed and extending functionality. The library also supports WebGL technology, which uses hardware acceleration to display content. WebGL is also used by Cesium, a 3D globe renderer, which is a milestone for the OpenLayers 3 project to support natively.

As the UML of a basic workflow shows in Figure 7, with the structural improvements, the library became more complicated, but at the same time, it got easier to use. Events for example, can be set on any child of the Observable class with a simple “on” function. Furthermore, there aren't as much emphasis on the controls, as in OpenLayers 2, because the interactive controls got migrated into a separate class. Finally, maybe the most conspicuous novelty from the diagram is worth mentioning. There are only a very few public property in OpenLayers 3. The properties are compressed with the Closure Compiler, hence they can be hardly accessed. Instead, one can use an object's appropriate methods to extract and set properties.

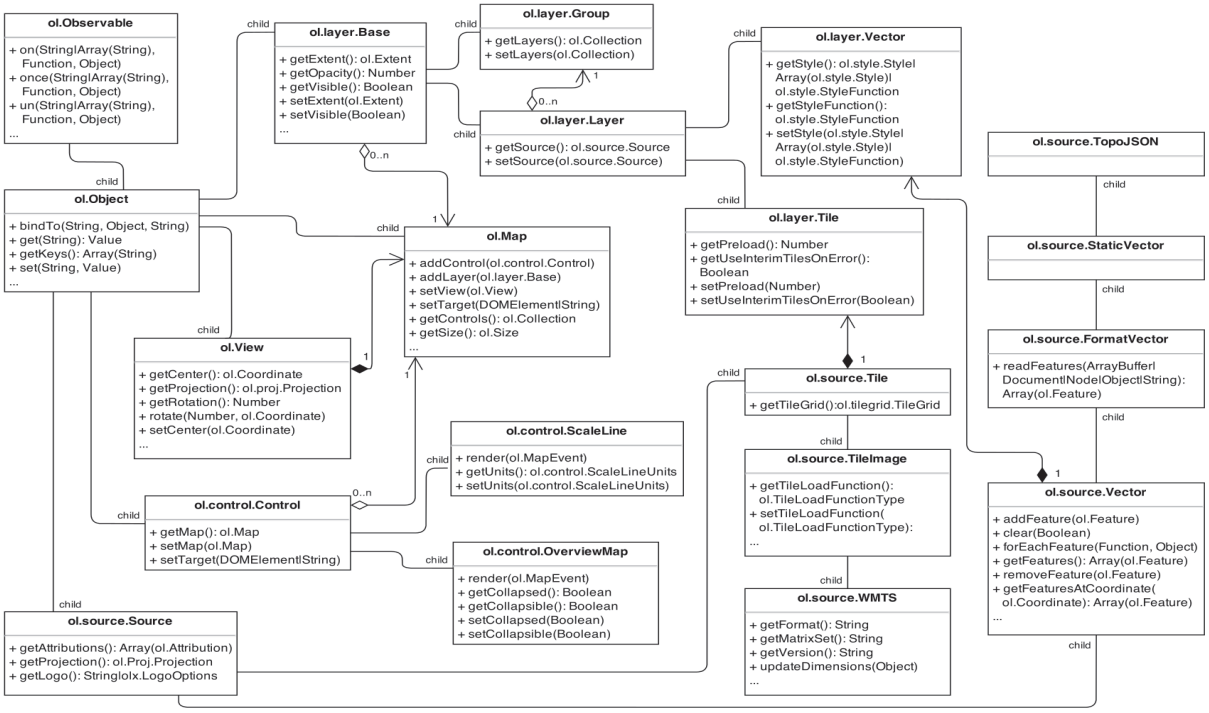

Figure 7: Partial UML diagram of OpenLayers 3 based on the OpenLayers 3 API. Copyright 2015 by OpenLayers Contributors. Adapted with permission.

A basic workflow in OpenLayers 3 also starts with creating one or more map object. Every map object need an own view object, which defines the extent, projection, center, etc. Maps can have layers, controls, and interactions. Layers can be aggregated in layer groups. Every layer need a source, which are specialized classes for different layer types. For example, a TopoJSON source can read a TopoJSON file, and translate its content for the library.

The OpenLayers 3 API can be found at http://openlayers.org/en/v3.4.0/apidoc/, and is encouraged to look up and use, if something is not clear from the study's examples.

2.3.3 Mapbox JS

Mapbox JS is built upon the popular lightweight web mapping library, Leaflet. Leaflet was created by Vladimir Agafonkin in 2010. As the subtitle states on its home page, it is a library for mobile-friendly maps. It is small, compact, and fast. The library doesn't have such capabilities as OpenLayers, but it wasn't intended to be a robust base of a complex WebGIS application in the first place. However, owing to its rapid development by Agafonkin, and a decent group of contributors, it became a potential participant for such an application.

“Leaflet is designed with simplicity, performance and usability in mind. It works efficiently across all major desktop and mobile platforms” (V. Agafonkin 2015). The library can be separated to two important parts, the core, and the plugins. The core is a small and stable library developed mosty by Agafonkin. The library is extendible via third-party plugins, which are based, and compatible with the core library, and were created by various contributors.

The plugins can be accessed from the contributors' git repository, but the best ones are featured on the home page of the library. The main disadvantage of plugin based extensions, is the possibility of incompatible plugins. The extensions must be compatible with the core library, but cross-compatibility is not obligatory. This can be a major issue, if one would like to use multiple plugins, but they don't cooperate with each other.

Mapbox JS acts as an adhesive agent in some of Leaflet's loose mechanisms. It extends some of the core classes to make it more suitable for a WebGIS application. It also has a rich API with examples, and a list of selected plugins, which can be used without conflict.

The map object is the central class of a Mapbox JS application. It contains layers, controls, and options, like the center or the default zoom level of the map. There are three type of layers in the API. The tile layer loads tiled maps, the grid layer loads UTFGrid files, while the feature layer groups vector data in a layer.

Feature layers in Leaflet and Mapbox JS are handled slackly by default. Every feature added to the map is put into a separate layer object, therefore the eachLayer query returns all of the features along with the layer groups. This behavior makes harder to create dynamic WebGIS applications. Most of the map contents are displayed as DOM elements, while vector data are rendered as SVG items.

The Mapbox JS API can be found at https://www.mapbox.com/mapbox.js/api/v2.1.6/, and is encouraged to look up and use, if something is not clear from the study's examples.

III. Purpose of Study

3.1 Purpose of Study

The purpose of the study is to give a comprehensive comparison of the web mapping libraries discussed above. The OpenLayers series and Mapbox JS have the highest potential to be chosen as a front end of a WebGIS application, but the choice is not trivial. The study attempts to rate these libraries by their strengths and weaknesses, and offer basic considerations when to choose a particular library. The study also gives examples for the capabilities of the libraries.

An initial consideration must be mentioned here: the study doesn't intend to downgrade any of the discussed libraries. It notes the weaknesses of them in the aspect of building a WebGIS application, but it tries to emphasize their strengths, and give realistic cases, when one should be preferred over the other ones. This is the main rationale behind choosing only the presented libraries.

A list of the objectives in a well interpretable form:

-Comparison of OpenLayers 2, OpenLayers 3, and Mapbox JS.
-Demonstration of various features with examples.
-Classification of the libraries based on their capabilities.
-Considerations for deploying the right library in different cases.

IV. Discussion

4.1 Introduction

There are several ways to compare web mapping libraries. This study rates them by their capabilities in essential features to act as a front end of a WebGIS application. The categories represent feature collections. Most of the examined functionality are based on the popular open source desktop GIS software, QGIS. However, web technology has some limitations, thus features impossible to implement effectively are not included.

Due to the lengthy nature of prettified code blocks, complete examples are not included in-text, but as a digital supplement appended to the study. The example codes are organized by libraries and section numbers. When the study references an example, it should be searched by the section number under the preferred library name in the examples folder. If a feature cannot be implemented by one or more library, it is explicitly stated in the appropriate subsection.

As the study also has a tutorial aspect, it briefly describes the main considerations of implementing various features. It contains code snippets to illustrate those considerations, and help understand the basic concepts behind web mapping libraries. The code snippets are just abbreviations of the complete examples, therefore inspecting the utter code is strongly recommended. A hosted alternative of the examples can be found at https://gaborfarkas.github.io/thesis.

4.2 Research Methodology

The methods of comparison consists of two different parts. One of them is a capability test, which tends to determine if a library is capable of achieving a specified objective. The other part is a simple benchmarking, which compares the speed of the libraries in various aspects.

The capability test comprises five main classes. These classes represent the key aspects of building a WebGIS application. The subclasses of these main categories are detailed in the beginning of each related subsection. The classes are Data I/O (input and output), Layer management, Attribute and Vector management, Map controls, and Composing controls. These five categories involve the most essential parts of a WebGIS graphical user interface.

The rating of achieving the objectives (i.e. implementing the given feature) are very similar to the rating method of R. Donohue et. al. The study uses a complex grading system. There are three main values, which refer to the applicability of a feature. If a library natively supports a feature, it gets a supported flag. Only those features are considered supported, whose deployment can be achieved with the library's functions. If a feature can be utilized with a plugin or some extra coding (preferably with a well-known workaround), the library gets a deployable flag. If a feature cannot be realized by a library, it gets a not supported flag. There is also a special, broken flag given to libraries, which tend to support the feature, but cannot achieve it in this version. It is considered as not supported, but additionally indicates, the feature was, or will be correctly supported by other versions of the library.

Each supported feature is graded forth with the number of lines needed to achieve them. This rating method is subjective, as it depends of a person's coding style and knowledge. However, as the codes are written by one person, the grades can proportionately describe the complexity of implementing a feature with a library.

The benchmarking is done with the Benchmark.js JavaScript library. There are separate benchmarks for the subclasses, because they aren't guaranteed being able to implemented in all of the libraries. As a result, comparing the speed of a whole class results in unreliable outcome in favor of the library, which can implement the least of the features.

Only natively supported features go under benchmarking. as they're optimized by their creators, thus can represent the speed of a library. Functions created by me, or any other author of a well-known method are not guaranteed to run at their optimum speed, therefore they're opted out of benchmarking. It must be stated, there are a lot of asynchronous calls in the libraries. It depends on the libraries' structure if they force the application t o wait until the calls are completed. This means, functions with asynchronous calls, like fetching feature data from a remote server, aren't guaranteed to yield reliable benchmarking results.

Benchmark.js is described in details by Google developer M. Hossain (2012). First it enters in an analysis state, when it determines the number of cycles needed to get statistically significant result (it tries to reach a minimum of 99% certainty). After the number of the required iterations has been calculated, it enters in a sampling phase. When the sampling phase is over, it can give the result in operations/second along with the percent of uncertainty, and the number of runs sampled.

The benchmarks were done on a 64-bit Debian Wheezy (version 7.8), with Mozilla Firefox 37.0. The relevant parts of the hardware specification are the Intel(R) Core(TM) i3-2350M processor, and the 6 GB RAM with a configured clock speed of 1067 MHz. Benchmark.js used W3C's High Resolution Time API, which is supported by all of the major browsers. The compared libraries are OpenLayers 2.13.1, OpenLayers 3.4.0, and Mapbox JS 2.1.6.

4.3 Data I/O

The input and output capabilities of a library is a key aspect in choosing a library for a WebGIS application. It defines the possibilities of the server side configuration. The more formats a library can handle, the more extensible an application can be. The most important formats to handle in WebGIS are tile formats, single image formats, vector formats, and parsing metadata.

Tile formats can come in various forms. The most typical tile formats are the TMS, and XYZ, while supporting WMTS, TileJSON, or UTFGrid can also mean more scalability, therefore they will be checked. Image formats are mainly for opening and/or georeferencing raster data on the client side. Vector formats are very important, as they represent true interaction between the user and the source data. There are several vector formats, but only one of them is crucial for a library to support. However, it is important for a library to be able to give vector output in a standardized format. This way users can commit changes, edit, update, or delete features on the server side from the GUI. Metadata are served with various OGC web services in XML format, and contain additional properties concerning the published layers.

4.3.1 Tile Formats

From the various tile formats the study compares the most popular, and useful ones, which can be effectively used in a WebGIS application. The subjects of comparison are the XYZ, WMTS, TMS, UTFGrid, and TileJSON tile formats. The XYZ, WTMS, and TMS formats can be used to render raster base layers with a good performance. UTFGrid is useful when attribution data is needed, but rendering vector data, or sending requests to the server for extra information is considered superfluous, (e.g. in mobile applications). TileJSON is a great abstraction format, which can be used for saving projects, or wrap layers with their metadata included on server side and send it with a single response.

As Figure 8 shows, most of the features are supported or deployable in the current versions of the libraries. Mapbox JS offers the most convenient way to deploy these features. Programming them in OpenLayers 2 is still easy, but implementing minimal TileJSON support needs some extra lines of code. OpenLayers 3 is the most tedious in the aspect of code length, but it offers peerless scalability in customizing tile sources.

	XYZ	TMS	WMTS	UTFGrid	TileJSON
OpenLayers 2	15	12	20	34	33
OpenLayers 3	25	21	33	29	15
Mapbox JS	10	9	16	NaN	7

: Supported
: Deployable
: Broken

Figure 8: Support diagram of tile layers in OpenLayers 2, OpenLayers 3, and Mapbox JS.

OpenLayers 2 offers native support for all of the tested features, except TileJSON. Implementing the first three formats is an easy tasks, they can be parameterized in the map object. The XYZ format takes some template URLs as input. The templates have to mark the place of the variables with x, y, and z in brackets, preceded by the commonly used variable sign: $.

```
http://otile1.mqcdn.com/tiles/1.0.0/map/${z}/${x}/${y}.png
```

The TMS format takes an URL, and some layer options as input. Unlike other formats, the URL must end with a forward slash. The options must include the name of the layer, and the type of the response image. In case of using GeoServer's GeoWebCache, the layer name contains extra information, like the SRS code, and the image type.

```
"http://demo.opengeo.org/geoserver/gwc/service/tms/", {
    layername: 'nasa:bluemarble@EPSG:900913@png',
    type:'png'
})
```

Setting up a WMTS format should not be more complicated, than constructing a WMTS request manually. OpenLayers 2 needs to know the URL, the layer name, the martixSet, the array of matrixIds (which is the name of the matrixSet, followed by the zoom levels, delimited with a colon in most of the cases), the format, and the requested style (which can be null, but should be included regardless).

Implementing a UTFGrid requires a layer object, and a control. The layer object contains the information required to connect to the source, while the control handles the interaction with the layer. Constructing the layer object is similar to the XYZ format, as it needs an URL template. An optional resolution parameter can also be defined to smooth the experience of the interaction. The control has to know the type of the interaction, which can be click, hover, or move, and the callback function, which processes the actually looked up attribute information. The library passes an object to this function, containing information from all of the active UTFGrid layers (if not specified differently) in an associative array.

```
for (var idx in infoLookup) {
    info = infoLookup[idx];
    if (info && info.data) {
        output.innerHTML = infoLookup[idx].data.admin;
    }
}
```

TileJSON is not supported by OpenLayers 2, but it can be extended to provide basic support. As TileJSON files can store XYZ and TMS schemes, but mostly used for storing XYZ tile sources, the example parser only handles XYZ format. It grabs the file through an AJAX request, changes the URL templates to match the library's URL scheme, then fills a new XYZ layer with elementary properties provided by the TileJSON file.

```
for (var i=0;i<tileObj.tiles.length;i++) {
    tileObj.tiles[i] = tileObj.tiles[i].replace(/{/g, '${');
}
if (tileObj.scheme === 'xyz') {
    var XYZ = new OpenLayers.Layer.XYZ(tileName, tileObj.tiles, {
        attribution: tileObj.attribution
    });
    var maxZoom = tileObj.maxzoom || 31;
    var minZoom = tileObj.minzoom || 0;
    XYZ.maxResolution = map.getResolutionForZoom(parseInt(minZoom));
    XYZ.numZoomLevels = maxZoom - minZoom + 1;
} else {
    throw new Error('No valid XYZ tilesource provided.');
}
```

OpenLayers 3 has more scalable methods to define layers, thus it is more complicated to parameterize a layer correctly. Formats, which tend to be easy to deploy, are truly have convenient methods, but applying layers from other sources can seem to be hard at first glance. The XYZ and the UTFGrid formats are the most straightforward to deploy, as the library only needs URLs to construct layers with such sources. The TileJSON format needs an URL pointing to a TileJSON file, while the XYZ source expects an array of URL templates as input. The templates are simplified compared to the OpenLayers 2 URL schemes.

```
http://otile1.mqcdn.com/tiles/1.0.0/map/{z}/{x}/{y}.png
```

There is no dedicated TMS source in OpenLayers 3, but the XYZ format can be extended to match the tiling scheme of the TMS, if it's published in a Web Mercator (EPSG:3857) projection. The custom tiling scheme can be defined in the tileUrlFunction property, which receives an array with the XYZ tile grid's z, x, and y indices for every tile of the grid in the current extent, and expects an URL calculated from them as a result.

```
//Original code by Ian McIntosh
//(https://groups.google.com/forum/#!msg/ol3-dev/8Co4JhwioGQ/nkTcljlUsDsJ)
tileUrlFunction: function(coordinate) {
    var z = coordinate[0];
    var x = coordinate[1];
    var y = (1 << z) - coordinate[2] - 1;
    return 'http://demo.opengeo.org/geoserver/gwc/service/tms/1.0.0/nasa:'+
           'bluemarble@EPSG:900913@png/' + z + '/' + x + '/' + y + '.png';
}
```

Defining a WMTS layer can be tricky, but it reveals some information about the library's tile handling. Along with the usual parameters, the WMTS source needs a tile grid object with the matrixIds, the origin, and the array of resolutions. The origin is the northwest corner of the layer, while the resolutions array can be calculated from the width of the tiles in map units, and the zoom level, as every zoom level doubles the previous one's resolution. Calculating the required parameters for a layer, which covers the whole extent of a projection can be achieved as the following:

```
var projection = ol.proj.get('EPSG:3857');
var size = ol.extent.getWidth(projection.getExtent()) / 256;
for (var i=0;i<31;i++) {
    matrixIds.push('EPSG:900913:' + i);
    resolutions.push(size / Math.pow(2, i))
}
```

The UTFGrid is similar to OpenLayers 2's definition, however instead of a specific control, it requires an event listening to the map object. The event can invoke the layer source's descriptive forDataAtCoordinateAndResolution method, which looks up the layer's attributes at a given location and resolution. The UTFGrid layer only requires a TileJSON file, which has to contain the path templates to the UTFGrid tiles.

Mapbox JS theoretically supports most of the studied tile formats, however in practice, the support of UTFGrid layers is broken with TileJSON files, therefore creating UTFGrid layers can only be done with a registered account, and a map id. WMTS is the second exception, as it is supported by a third party plugin.

Defining naturally supported layers are the easiest of the compared libraries in Mapbox JS. They only require an URL or an URL template, optionally with a set of subdomains, if there are more, than one. The only difference between an XYZ and a TMS layer, is the TMS layer needs a tms parameter set to true. Constructing a TileJSON layer is even more simple, as it only needs an URL pointing to a TileJSON file.

```
L.tileLayer("http://otile{s}.mqcdn.com/tiles/1.0.0/map/{z}/{x}/{y}.png", {
    subdomains: '1234',
    attribution: "&copy; MapQuest, OpenStreetMap contributors"
})
```

Accessing a WMTS source with the Leaflet WMTS plugin needs some extra configuration, but is still easier, than in OpenLayers. Setting up the array of matrixIds can be confusing when applied to a custom source. Every member of the array is an object, which needs to have an identifier property with a unique matrix id for the particular zoom level, and a topLeftCorner property, which is the origin of the layer wrapped in a L.LatLng object. When the origin of the layer equals the northwest corner of the projection extent, the first coordinate should be lowered, to avoid a wrong tile index addressing bug.

```
for (var i=0;i<31;i++) {
    matrixIds.push({
        identifier: 'EPSG:900913:' + i,
        topLeftCorner: new L.LatLng(20037508.3427,-20037508.3428)});
}
```

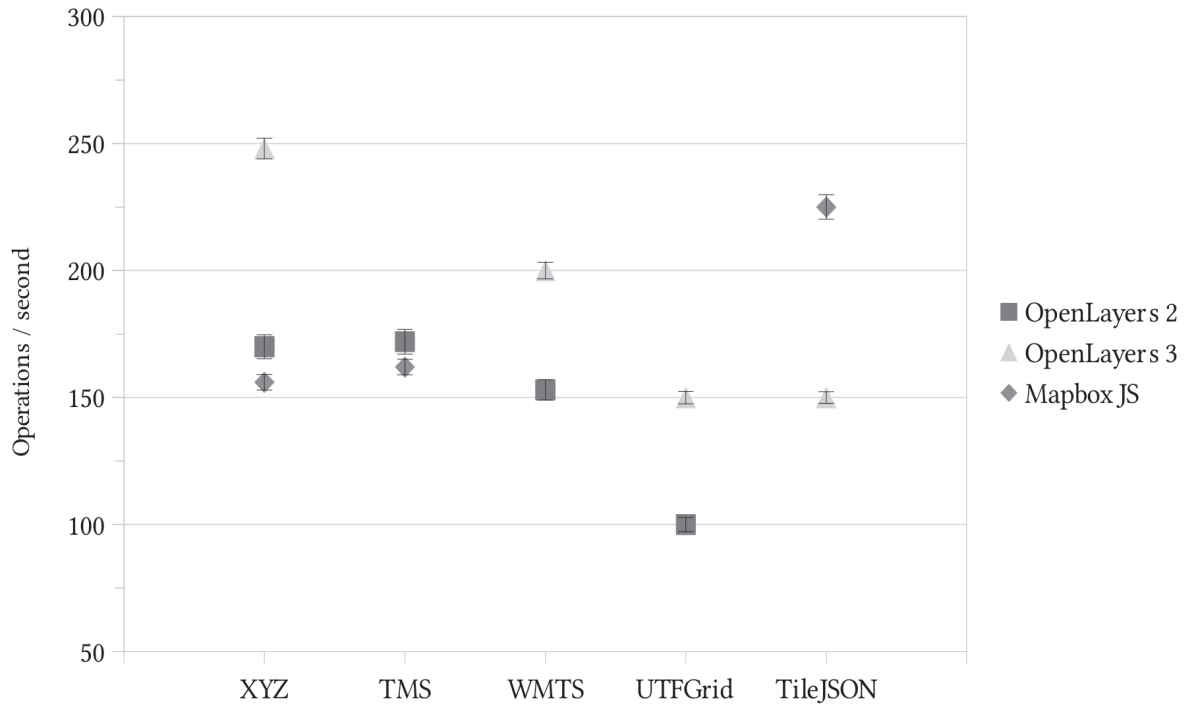

Figure 9: Benchmark diagram of tile layers in OpenLayers 2, OpenLayers 3, and Mapbox JS.

The result of the benchmarking can be seen in Figure 9. OpenLayers 3, and Mapbox JS exceed in the aspect of speed. Mapbox JS shows outstanding performance in processing the Mapbox standard TileJSON format, and is expected to produce similar results with UTFGrid, if it will be fixed. OpenLayers 3 shows its strength in deploying layers with OGC web services, and other traditional open source tile formats.

4.3.2 Image Formats

Image formats have two important benefits in a WebGIS application. The first one is unequivocal: to georeference and process images provided by users on the front end. The second one is not as trivial, as it is related to the technical solutions of web mapping services. Tile services are mostly used with static data. They get rendered periodically, and until the next rendering, the same dataset is served. If the data changes dynamically, using a WMS should be considered. A WMS generates and serves the raster images in the time of the request, therefore it can be used to serve up-to-date data at the expense of speed, and cacheability.

The default behavior in OpenLayers 2, and Mapbox JS with WMS, is splicing up the view by a regular grid, and requesting a separate image for each member of the grid. This behavior can increase user experience, but can also result in strange graphical errors.

When the WMS server renders labels, or other graphical elements on the top of the base layer, it generates them for every requested image. If the images are requested in a tiled format, the labels get replicated by the number of relevant tiles, while graphical elements get cutted on tile edges. This phenomenon is caused by the fact, that the server does not know, if the requested images belong to the same layer in the front end application. To avoid such problems, the libraries should be able to request single images from the WMS server covering the actual view extent.

Figure 10 details, the three examined features can be deployed in both of the libraries. As usual, Mapbox JS offers the most convenient way to implement them, but for image WMS sources, it needs a plugins. OpenLayers 2 has also easy methods to code the features, and OpenLayers 3 is the most tedious, but has plenty of customization options.

	WMS Tile	WMS Image	Static Image
OpenLayers 2	13	16	13
OpenLayers 3	19	19	17
Mapbox JS	12	12	9

: Supported
: Deployable

Figure 10 : Support diagram of image layers in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Constructing a tiled, and a single image WMS layer in OpenLayers 2 is almost identical. The single image layer only needs one extra parameter, singleTile set to true. It also needs some basic information about the server, precisely the URL of the server, and the name of the layer. The first argument must be the display name of the layer in the application.

```
new OpenLayers.Layer.WMS(
    'Global Imagery',
    'http://demo.opengeo.org/geoserver/wms',
    {
        layers: 'bluemarble'
    },{
        singleTile: true
})
```

Creating an image layer in the library doesn't require any object with properties and values, just four mandatory arguments. The layer constructor needs to know the display name of the layer, the URL to the data source, the extent, and the size of the image. The latter two need to be wrapped in the corresponding OpenLayers objects.

```
new OpenLayers.Layer.Image(
    'City Lights',
    '../../../res/image/4_m_citylights_lg.gif',
    new OpenLayers.Bounds(-180, -88.759, 180, 88.759),
    new OpenLayers.Size(580, 288)
)
```

OpenLayers 3's layer structure requires to define single image and tiled WMS layers differently. There is a separate WMS source for image layers (ImageWMS) and for tile layers (TileWMS). The parameterizing of the sources are identical, it just has to be considered, which of them should be applied. The constructor uses key-value pairs as input, and the server specific parameters have to be nested as an object in the params property.

```
new ol.source.ImageWMS({
    url: 'http://demo.opengeo.org/geoserver/wms',
    params: {
        layers: 'bluemarble',
        format: 'image/png'
    }
})
```

Defining an image layer is more simple, as only three parameters are needed: the URL to the image, the image extent as an array, and the image size as an array. Obviously, the parameters have to be declared in a KVP manner. In both of the OpenLayers library, extents have to be defined in the order: west, south, east, north. Sizes have to follow the wide, height order.

In Mapbox JS, only tiled WMS layers are supported natively. However, with a plugin, single image WMS layers can be achieved, too. With the plugin, creating the layers are identical, just the constructor differs. A tiled WMS layer can be constructed with L.tileLayer.wms, while the single image WMS layer's constructor is L.nonTiledLayer.wms. As parameters, they accept the usual parameters.

```
L.tileLayer.wms("http://demo.opengeo.org/geoserver/wms", {
    layers: 'bluemarble',
    format: 'image/png',
    transparent: true,
})
```

Image layers can be created with L.imageOverlay with the library. The name of the constructor suggests, it is a different layer type, and indeed it is handled differently. Unlike OpenLayers, Mapbox JS needs only two argument to create such a layer. The first one is the URL to the image, while the second one is an array with two coordinate pairs. The array must contain the southwest and the northeast coordinates of the image.

```
L.imageOverlay('../../../res/image/4_m_citylights_lg.gif',
    [[-89.5, -180], [89.5, 180]])
```

Figure 11 shows the benchmarking results of creating image layers. Owing to the default canvas renderer, and highly optimized code, OpenLayers 3 gives the best performance. The other two library shows moderate speed, with a lower performance in creating tiled WMS layers, and a slightly higher in the case of single image WMS layers. In these libraries, single image WMS should be preferred over tiled WMS. It grants a better performance, generates less traffic, and eliminates the possible graphical issues created by WMS servers.

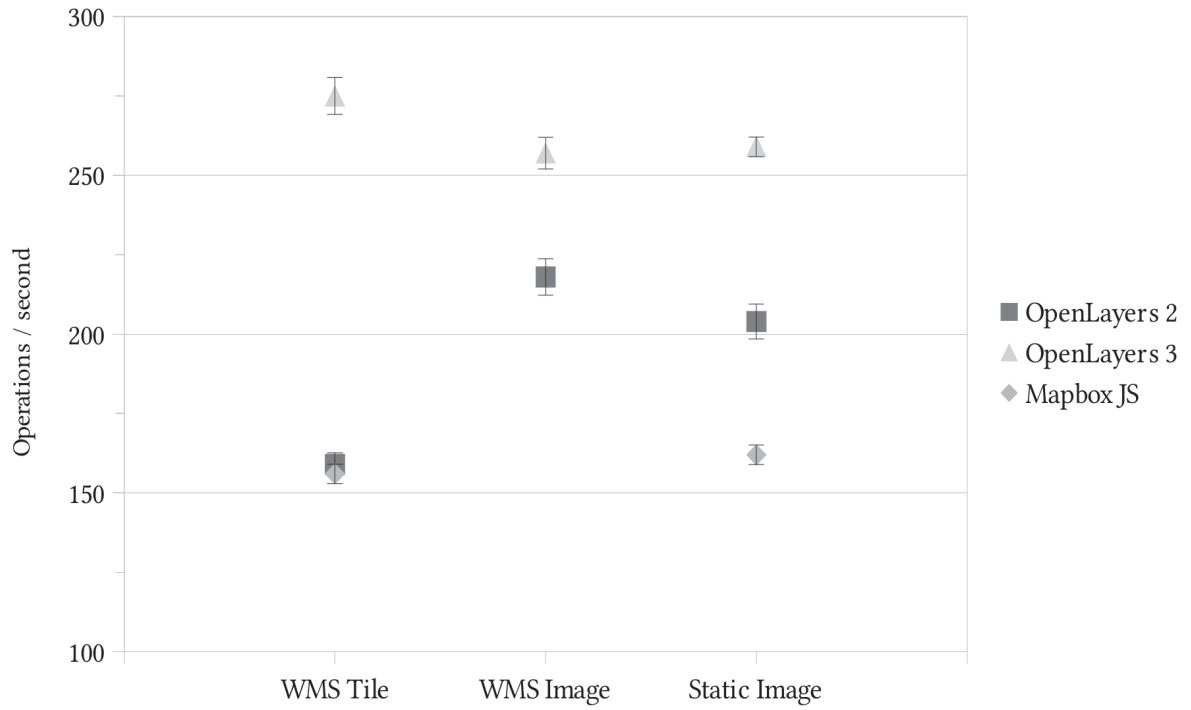

Figure 11: Benchmark diagram of image layers in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.3.3 Vector Formats as Input

Vector formats are a necessity in a WebGIS application, as they can be read, modified, and saved by the user. There are various ways to fetch features from a server, and some of them have such specialties, their support defines, if a library is suitable for the front end of an application.

The examined formats are WFS, GeoJSON, TopoJSON, GPX, KML, OSM, CSV, and Vector tiles. They are just a fracture of the vector formats used by geospatial applications, but they are very useful, and their support can be crucial. As a side note, all of these formats, are ASCII based, none of them use a binary encoding. While binary formats can store huge datasets in a compact size, web technology always preferred human readable data, as "concepts like bytes, bits, memory access and even raw file access is utterly alien to JavaScript" (J. L. Aasenden 2015).

WFS makes direct contact to the server with carefully parameterized URLs. They can be used to fetch features, and attributes, to save changes, update, or delete features on the server side. For making modifications on the server side, a special service, WFS-T (Web Feature Service - Transaction) has been developed, which can make transactions in GML format. This is the most direct communication method between a WebGIS client and server, therefore WFS support is recommended in the WebGIS application.

GeoJSON and TopoJSON are JSON (JavaScript Object Notation) based formats, therefore JavaScript has native support for parsing, and converting them back to string. Browsers can process them quickly, thus these type of formats offer the best performance. Web mapping libraries tend to natively support at least GeoJSON, and as server side GIS applications can also process them, their support ensures the capability of effective communication between client and server.

The rest of the studied formats are mostly used in special circumstances. GPX (GPS Exchange Format) can be used for field survey applications, when the users have to upload their measurements directly to the application, modify them if needed, then send the data to the server. KML and OSM are XML (Extensible Markup Language) based formats primarily used by Google Maps, and OpenStreetMaps. CSV (Comma Separated Values) are used by publishing dynamically changing point data, as the format can only store a single pair of coordinates per record effectively. Vector tiles are for storing and publishing static vector data in a tiled format, to grant accessibility and possibility for adaptation, but also offer good performance.

As the support diagram presents in Figure 12, GeoJSON is the only natively supported vector input format by all three libraries. OpenLayers 2's lack of support of TopoJSON and Vector tiles are due to the finished development of the library. These are newer formats, which got integrated into OpenLayers 3. OpenLayers 3 on the other side, does not support CSV input at this version, but it is not guaranteed, it will not be implemented in the future. Mapbox JS only supports GeoJSON natively, which is attributable to the lightweight nature of the library. There is a plugin called Omnivore, however, which can be used to convert some of the compared data formats into GeoJSON.

	WFS	GeoJSON	TopoJSON	GPX	KML	CSV	OSM	Vector Tiles
OpenLayers 2	15	15	NaN	15	15	15	15	NaN
OpenLayers 3	30	16	16	16	16	NaN	NaN	21
Mapbox JS	NaN	6	6	6	6	10	17	NaN

: Supported
: Deployable
: Broken
: Not supported

Figure 12: Support diagram of vector input formats in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, a WFS connection requires a simple vector layer, and a WFS protocol. Procotols are used to fetch data asynchronously in OpenLayers 2, and the HTTP is the most commonly used one. The minimal configuration of the protocol must contains the URL to the server, and the featureType, which is basically the layer name. It is recommended to add the featureNS property, too. In a WFS server, there can be multiple layers present on the same layer name, and their namespace property separates them. To be absolutely sure, the right layer will be requested, the namespace parameter should be provided with the full namespace, not just the abbreviation. Vector layers also have a strategy parameter, which contains one or more loading strategy. Layers in Web Mercator projection cannot be fetched with BBOX strategy in OpenLayers 2, therefore the example uses Fixed, which loads all features on initialization.

```
new OpenLayers.Layer.Vector('OSM Admin 5678', {
    strategies: [new OpenLayers.Strategy.Fixed()],
    protocol: new OpenLayers.Protocol.WFS({
        url: "http://demo.opengeo.org/geoserver/wfs",
        featureType: "admin_5678"
    })
})
```

The rest of the formats can be loaded with a HTTP protocol, with the appropriate format object. The format objects can be parameterized, but it is not necessary. GeoJSON format has a very useful ignoreExtraDims parameter, which ignores all of the higher dimensions in the geometries, as OpenLayers 2 can only handle X and Y.

```
protocol: new OpenLayers.Protocol.HTTP({
    url: '../../../res/vector/ne_roads.geojson',
    format: new OpenLayers.Format.GeoJSON()
})
```

CSV data sources has to be well formatted to be accepted as a layer. OpenLayers 2 only accepts tabulator as field delimiter, and two mandatory fields, lat, and lon also have to be provided. The CSV (Text) format has a constraint, it plots the input data as a marker layer.

Establishing a WFS connection in OpenLayers 3 requires a loader function. The loader function receives the current extent, resolution, and projection object as input, and executes the function with every change in the view, thus the function has to include a method to process the received features, and add them to the layer. As the loader function already contains the feature type, and namespace, defining them separately only applies a local filter for the received features. Currently WFS only works with Web Mercator using GeoServer as the WFS server, as GeoServer publishes WGS 84 data in reverse coordinate order, and OpenLayers 3's GML parser cannot be configured to use reverse coordinate ordering.

```
loader: function(extent, res, proj) {
    var source = this;
    var url = 'http://demo.opengeo.org/geoserver/wfs?request=GetFeature&' +
              'version=1.1.0&typename=osm:admin_5678&srsname=' +
              proj.getCode() + '&bbox=' + extent.join(',');
    var ajax = new XMLHttpRequest();
    ajax.open('GET', url);
    ajax.onreadystatechange = function() {
        if (ajax.status === 200 && ajax.readyState === 4) {
            source.addFeatures(source.readFeatures(ajax.responseText));
        }
    };
    ajax.send();
}
```

Most of the other supported formats can be used with a StaticVector source object, which can be parameterized with an URL, and a format object, and has to be wrapped in a vector layer object. The Tile vector format is the only exception, which needs some additional configuration. The source object has to be TileVector, while the parameters must include a format, a projection, an URL, and a tile grid. The tile grid defines the tiling scheme, and usually is XYZ. The URL follows a template, while the format parses the grid files. The tile grid's maxZoom property is mandatory for creating the layer.

```
source: new ol.source.TileVector({
    format: new ol.format.TopoJSON(),
    projection: 'EPSG:3857',
    url: 'http://{a-c}.tile.openstreetmap.us/vectiles-land-usages/' +
         '{z}/{x}/{y}.topojson',
    tileGrid: new ol.tilegrid.XYZ({
        maxZoom: 19
    })
})
```

Mapbox JS offers an extremely simple way to use supported feature formats. A file with the only natively supported format, GeoJSON can be loaded to a Mapbox feature layer with a single loadURL function. Other deployable formats can be loaded with the Omnivore plugin.

```
map.featureLayer.loadURL('../../../res/vector/ne_roads.geojson');
omnivore.topojson('../../../res/vector/ne_roads.topojson').addTo(map);
```

Only the CSV format takes some parameters, in which the user can define the delimiter character, the longitude column name, and the latitude column name.

```
omnivore.csv('../../../res/vector/ne_capitals.csv', {
    latfield: 'lat',
    lonfield: 'lon',
    delimiter: '\t'
}).addTo(map);
```

As the benchmarking result shows in Figure 13, OpenLayers 3 is the fastest, Mapbox JS shows good performance, while OpenLayers 2 is the least efficient in loading vector layers from outer sources. CSV didn't get benchmarked, as it can only accept point data, therefore the sample dataset differs from the others. This could be used to denote the proportional differences in the loading speed of OpenLayers 2, and Mapbox JS, however Mapbox JS is opted out of the benchmarking, because CSV is supported through a plugin.

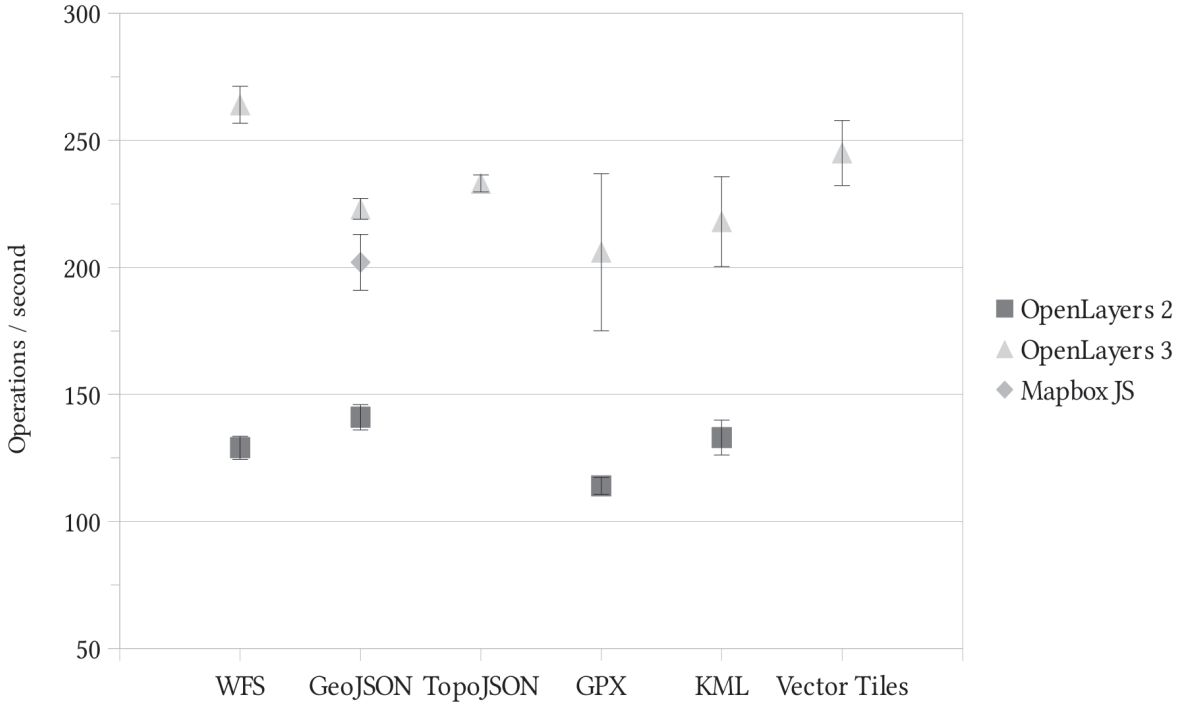

Figure 13: Benchmark diagram of vector input formats in OpenLayers 2, OpenLayers 3, and Mapbox JS.

The margin of error in benchmarking OpenLayers 3's GPX format was huge due to the huge amount of memory needed and allocated by the browser. The browser only didn't crash, when a small sample was studied, but with such number of cycles, the benchmarking could not show up reliable results. This format is advised to be used to transfer only small datasets, if preprocessing is undesirable.

4.3.4 Vector Formats as Output

Output vector formats define the capability of an application to effectively communicate vector data back to the server. Conceivably, WFS-T is the most widely used format to achieve such tasks, but a well prepared server can also receive information in more compact formats. WFS uses GML as a context for sending vector data, and GML tends to be lengthy.

The study compares the capability of writing GeoJSON, TopoJSON, GPX, KML, and GML data from the internal vector format. GeoJSON and TopoJSON are especially good formats for data exchange, as they can effectively send bigger datasets in a processable way. Server side applications also know how to handle at least GeoJSON.

GPX and KML can be useful to write in special cases. GPX can be used to upload data directly to GPS systems, and ease field surveys, while KML can be used from visualizing data on Google products, like Google Earth to acting as an exchange format for 3D data processing and conversion, like Collada in SketchUp.

The support diagram in Figure 14 shows the TopoJSON format is not supported currently by any of the compared libraries. Due to its resource demanding nature it is understandable, but expected to get support for in the future, at least in OpenLayers 3. The other formats are supported in OpenLayers 2, and OpenLayers 3, while Mapbox JS only offers GeoJSON export, which should be considered acceptable for a simple WebGIS application.

	GeoJSON	TopoJSON	GPX	KML	GML
OpenLayers 2	3	NaN	4	4	4
OpenLayers 3	3	NaN	4	4	7
Mapbox JS	2	NaN	NaN	NaN	NaN

: Supported
: Not supported

Figure 14: Support diagram of vector output formats in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Serializing features in OpenLayers 2 is very simple. It needs a specific format object, and its write method can simply generate a string from an array of features. To write every feature of a layer to the specified output, the features array of the layer has to be passed to the format object.

```
var layer = map.layers[0];
var parser = new OpenLayers.Format.GeoJSON();
parser.write(layer.features);
```

OpenLayers 3 also offers similarly convenient methods for serializing vector data. Only GML is the exception, where the feature type, and the namespace is also needed as further parameters. The writeFeature method generates the output, and as usual, it expects an array of features as input.

```
var features = map.getLayers().getArray()[0].getSource().getFeatures();
var parser = new ol.format.GML({
    featureType: 'hunroads',
    featureNS: 'www.naturalearthdata.com'
});
parser.writeFeatures(features);
```

Mapbox JS uses a different approach to serialize feature data to GeoJSON. Every feature has a separate layer object, while they're grouped in feature groups. All the feature layers, and feature groups has a toGeoJSON method, which writes out their content in GeoJSON format. The main limitation is, there is no convenient way to access the correct vector layer, if there are more, than one. If there is only one vector layer, it can be accessed through the map object's featureLayer property.

```
var geojsonObj = map.featureLayer.toGeoJSON();
JSON.stringify(geojsonObj);
```

The benchmarking results in Figure 15 indicates, GeoJSON serializing is faster, than stringifying other formats by categories in all of the libraries. OpenLayers 2 gives a better performance in generating output for the more widely used formats, while OpenLayers 3 also offers good performance.

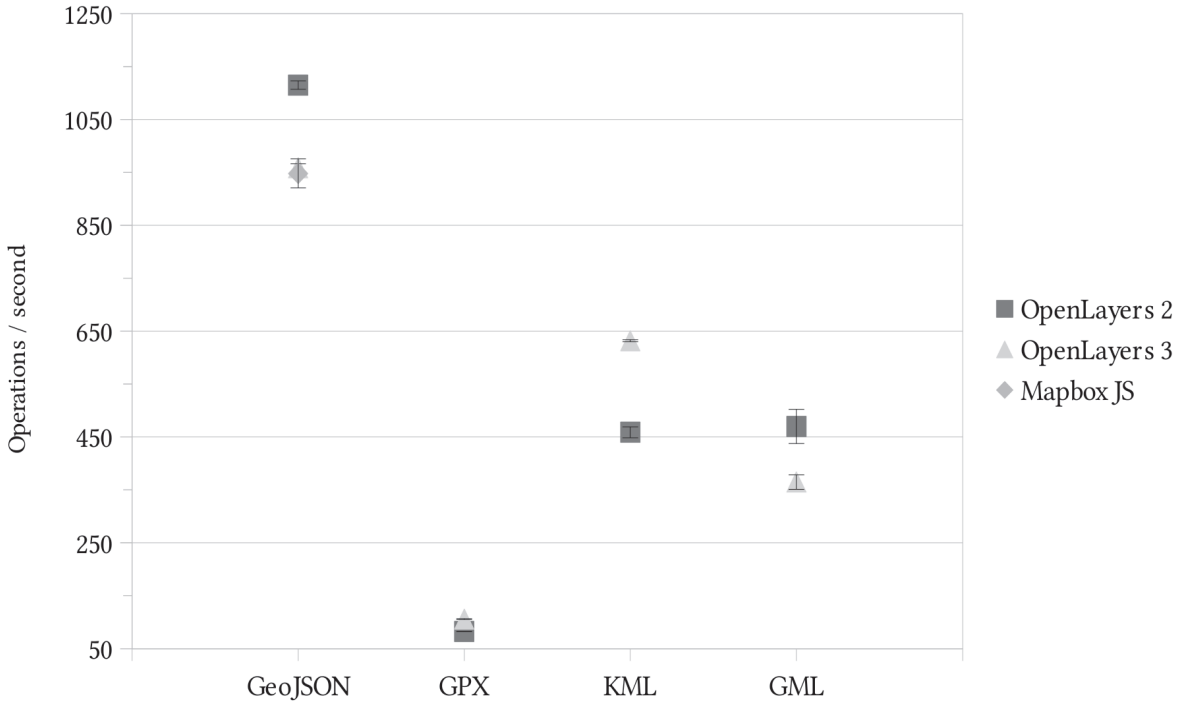

Figure 15: Benchmark diagram of vector output formats in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.3.5 Metadata

Separately processable metadata are bound with OWS standards, like WMS, WFS, WMTS, WCS, and WPS. They are served as XML documents, and can be accessed with the common request in the implementations, GetCapabilities. They contain valuable information regarding to mainly raster layers. There are some in other ways inaccessible properties of the layers shipped with this request, like the layer's extent, legend, or projection.

The support diagram in Figure 16 tells, MapboxJS doesn't have any convenience methods to parse GetCapabilities strings. It is still possible to process the response, however a simple parser have to be written to filter out the useful information. OpenLayers 2 has a huge support for every variety of capabilities response, but only three of them is analyzed by the study. OpenLayers 3 also offers two capabilities formats.

	WMS	WMTS	WFS
OpenLayers 2	10	10	10
OpenLayers 3	10	10	NaN
Mapbox JS	NaN	NaN	NaN

: Supported
: Not supported

Figure 16: Support diagram of reading OWS capabilities in OpenLayers 2, OpenLayers 3, and Mapbox JS.

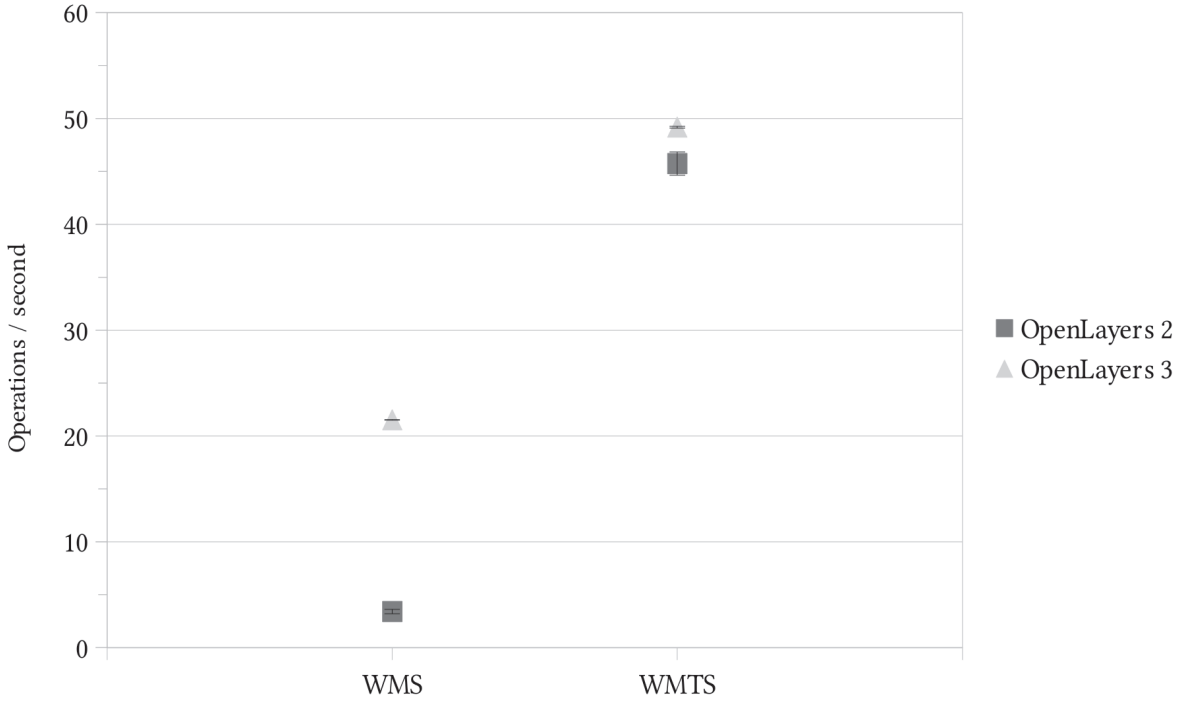

Figure 17: Benchmark diagram of reading OWS capabilities in OpenLayers 2, and OpenLayers 3.

The behavior of these parsers are completely identical to the other format objects, thus creating them only needs the appropriate constructor name. The formats can only read data, as expected. OpenLayers 2 can parse WFS capabilities data, but only up to version 1.1.0. Benchmarking results in Figure 17 indicate, the performance of the WMS parser has been gradually improved in OpenLayers 3, while the WMTS parser is nearly on the same level of speed. The WFS parser hasn't been benchmarked, as it is only implemented in OpenLayers 2, and the parsed data are not identical.

4.4 Layer Management

The way a library handles layers is another key concept in its capability of being the basis of a WebGIS application. The library needs to organize layers in a way, they remain easily accessible after initialization. After a layer is constructed, the user should be able to make modifications to it, like digitizing new features to a vector layer, or changing basic properties (e. g. layer name, projection, layer order). The three compared libraries have a more or less well-organized layer structure, however deeper analysis is needed in order to define their possible fields of application.

This section compares the libraries in three main categories. General layer operations are essential to be supported in order to make a usable application. They cover the most basic layer functions, like adding, or removing layers. Setting and modifying layer attributes is the second category. An applicable library must offer methods to change basic attributes, like layer opacity, or visibility, while it should be able to set constraints to a layer, too, like restricting layer extent, or the geometry type the layer accepts.

Layer events is the final category. Events make a web page dynamic with functions, only executed after a specific user input. Therefore the number of events a web mapping library supports define its dynamism. The more events listeners can be attached to, the more specialized functionality can be implemented. Events can be used to improve user experience (e.g. communicating the user if a layer has been loaded properly), or to build crucial functionality, like requiring attribute data for newly created features in a convenient form.

4.4.1 General Operations

General operations include adding layers, removing layers, and changing layer order. Layer order can be changed by altering the Z index of a layer object, which refers to the absolute position of a layer in a layer stack. Changing layer order has two operations, getting a layer's Z index, and changing it by a given value. The capability of setting a layer's Z index should be enough to change layer order, but for finer layer control, the application should be able to query the actual Z index and change it by a delta value.

As the support diagram in Figure 18 shows, most of the general operations are available in all of the libraries. OpenLayers 2 offers some simple functions to achieve the tasks, while in OpenLayers 3 Z index management is not trivial. The layer stack can be accessed by chaining two functions, which eventually return an array of layers. The layer order is defined by the order of the order of the elements in the layer collection. Altering the stack can be achieved by calling the collection object's setAt function.

	Add layer	Remove layer	Get Z index	Set Z index
OpenLayers 2	1	1	1	1
OpenLayers 3	1	1	2	2
Mapbox JS	1	1	NaN	1

: Supported
: Not supported

Figure 18 : Support diagram of general layer operations in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Mapbox JS uses an unusual approach to organize layers. The map contains panes for every layer type. There is a pane for tiles, overlays, markers, and popups. Every pane is handled separately by the library, and often have a local scope. This way, the Z index can only be interpreted for similar kind of layers. Tile layers are on the bottom of the stack, while vectors are on the top by default. This can be changed, however then tile layers will cover out vectors entirely, as because layers of the same type are organized in separate containers. Creating a layer stack with alternate layers of different kind is alas impossible.

In OpenLayers 2, there are a wide variety of convenience functions for achieving layer manipulation. For general operations, one can use the consequently named functions on the map object with layer objects, and optionally integers as arguments. For some of the functions, there are plural equivalents, which expect an array of objects as input. For changing layer order, one can also use the raiseLayer function, which requires a delta value besides of the layer object.

```
map.addLayer(layerObject);
map.removeLayer(layerObject);
map.getLayerIndex(layerObject);
map.setLayerIndex(layerObject, 0);
```

OpenLayers 3 offers the same methods for adding and removing layers, however for changing Z index, it requires some JavaScript array manipulation, as mentioned above. Getting the Z index is simple, the function only have to check the index of the layer in the array. If the method returns -1, the layer is not present in the composition. Setting the Z index can be done by native array manipulation, but the library offers a convenient method, which can be called on any collection object. The insertAt method needs an index, and an element of the underlying array as input, but in order to avoid duplicated layers, the element needs to be removed prior to inserting.

```
var layerArray = map.getLayers().getArray();
var layerIndex = layerArray.indexOf(inputLayer);
map.getLayers().remove(inputLayer);
map.getLayers().insertAt(0, inputLayer);
```

There is nothing special about Mapbox JS's supported methods, they're simple to implement, but changing the Z index of a layer requires some explanation. Due to the uncommon layer handling in Mapbox JS, setting the Z index only has a local impact. Furthermore, unlike OpenLayers libraries, Mapbox JS doesn't store the current position of a layer on object level, thus getting the Z index is not implementable. There are two universal methods in the library to change a layer's position without further calculations. The bringToFront and bringToBack methods can alter the layer's location in the stack, but still in a local scope by the difference of the layer's current index and the layer stack's extrema.

```
layerObject.setZIndex(0);
```

The benchmarking results, as shown in Figure 19, couldn't be done on all of the test cases. The result was limited by OpenLayers 2's inability in completing the test of the layer removal function. The benchmarking yielded varied results. OpenLayers 2 shows great performance in layer ordering, while in adding a new layer, it greatly falls behind the other libraries. Mapbox JS shows the least difference in various cases. If not outstanding, it still delivers a decent overall performance.

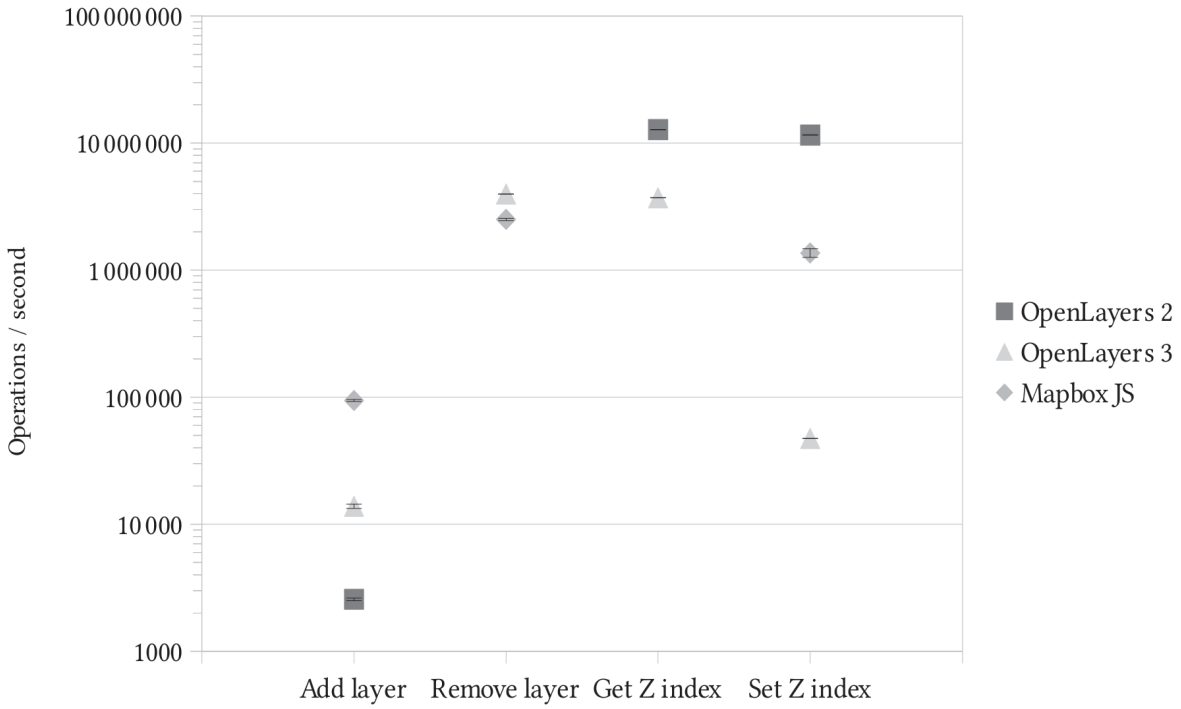

Figure 19: Benchmark diagram of general layer operations in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.4.2 Layer Attributes

Layer objects should be able to support storing attributes, and changing them appropriately. The most important attributes for a WebGIS application are visibility, opacity, extent, resolution constraints, and geometry type constraints. Some of these attributes are only need to be stored for every layer object, while others should be actively used by the renderer. The latter type should have native support by the library in order to work properly.

The extent property should be stored in every layer for the WebGIS application, as it enables zooming to the layer's extent. It should be stored in every library's native bound format, thus it can be used directly with the library's view setting method.

Resolution is a map constraint, which defines the maximum and minimum zoom level the layer can be displayed on. It is a necessity in non scalable (i.e. raster) layers, especially with tiled sources. The rationale behind this functionality, is if a layer is accessed through an OWS server, the layer's server side resolution bounds are respected by the client side application, thus display errors are avoided.

Visibility and opacity settings are mostly required for publishing maps. These controls, however, can ease one's workflow, by letting the user modifying the visibility of a layer, this way helping the analyst visually interpret spatial correlations.

Constraining geometry type in the client application can be useful for further, server side processing. Web mapping formats usually support geometry collections, although server side algorithms expect only one type of geometry in a layer. To overcome this contradiction, and avoid creating an overhead by sanitizing layers on the server side prior to processing, the WebGIS GUI should be able to effectively define the geometry type a vector layer can accept.

The support diagram in Figure 20 indicates, most of the features are supported by the libraries on some level. OpenLayers 2 supports all of the test cases, while OpenLayers 3 and Mapbox JS can't restrict the geometry type in a vector layer. This can be implemented manually, however other core functions, like adding features would ignore the custom property, thus every related functionality have to be extended.

	Extent	Resolution	Visibility	Opacity	Geometry
OpenLayers 2	1	2	1	1	1
OpenLayers 3	1	2	1	1	NaN
Mapbox JS	1	2	5	5	NaN

: Supported
: Deployable
: Not supported

Figure 20: Support diagram of layer attributes in OpenLayers 2, OpenLayers 3, and Mapbox JS.

With OpenLayers 2, one can easily define the geometry type and resolution constraints of a layer. They are stored in the layer object under the appropriate property names. The geometry type must be a string referencing the class name of a geometry. Layer extent can also be defined, but with a bounds object, which can be constructed from four coordinates given as arguments. The coordinate order is West, South, East, North. Visibility and opacity can be modified with the layer's setOpacity and setVisibility methods. The method for changing opacity expects a floating point value between 0 and 1, while the other one needs to be provided with a boolean value.

```
layerObject.maxExtent = new OpenLayers.Bounds(-180,-90,180,90);
layerObject.maxResolution = 0.5;
layerObject.minResolution = 2000;
layerObject.setVisibility(0);
layerObject.geometryType = 'OpenLayers.Geometry.Point';
```

OpenLayers 3 can be configured similarly, but instead of modifying object properties directly, they need to be set with setter methods. There is no dedicated bounds class in OpenLayers 3, therefore the extent needs to be defined as a simple array. The array follows the same coordinate order. Setting the other properties follows the same pattern, although the method for toggling a layer's visibility is setVisible in this library.

```
layerObject.setExtent([-180,-90,180,90]);
layerObject.setMinResolution(0.5);
layerObject.setMaxResolution(2000);
layerObject.setVisible(0);
```

Mapbox JS follows a different way to store properties and set constraints. The extent property needs to be manually defined, and it is recommended to fill it with the library's native bounds class. The constructor expects two arrays as input, the first filled with the Southern, and the Western coordinates, while the second filled with the Northern, and the Eastern ones in respective order. The resolution constraints can be defined in the form of zoom levels, which do only makes sense, when they really needed: in case of OWS layer sources.

```
layerObject.extent = L.latLngBounds([-90,-180], [90,180]);
layerObject.options.maxZoom = 18;
layerObject.options.minZoom = 0;
```

Setting the opacity and visibility of the layers don't have universal methods. Raster layers have a setOpacity method, in all other cases, similarly to changing the Z index, these functionality need to be implemented on the DOM level, instead of object level. As raster and vector layers don't share their structure, the input layer's type need to be checked in order to assign the correct styles to their DOM elements. Luckily, vector layers have a setStyle function, which can be used to change a whole layer's style in a bulk method.

```
if (layerObj instanceof L.Path || layerObj instanceof L.FeatureGroup) {
    layerObj.setStyle({opacity: 0});
} else if (layerObj instanceof L.TileLayer) {
    lyr.getContainer().style.display = 'none';
}
```

The benchmarking diagram in Figure 21 represents the difference in performance between OpenLayers 2's direct accessing structure, and OpenLayers 3's object interface structure. OpenLayers 2 sets and accesses its objects' properties directly, which is very fast, but it can lead to malfunctioning behavior. Contrary to this structure, OpenLayers 3 does not expose its objects' properties, and only lets them set and accessed via predefined methods. This way the procedure is slower, but the code becomes more secure, compact, and reliable.

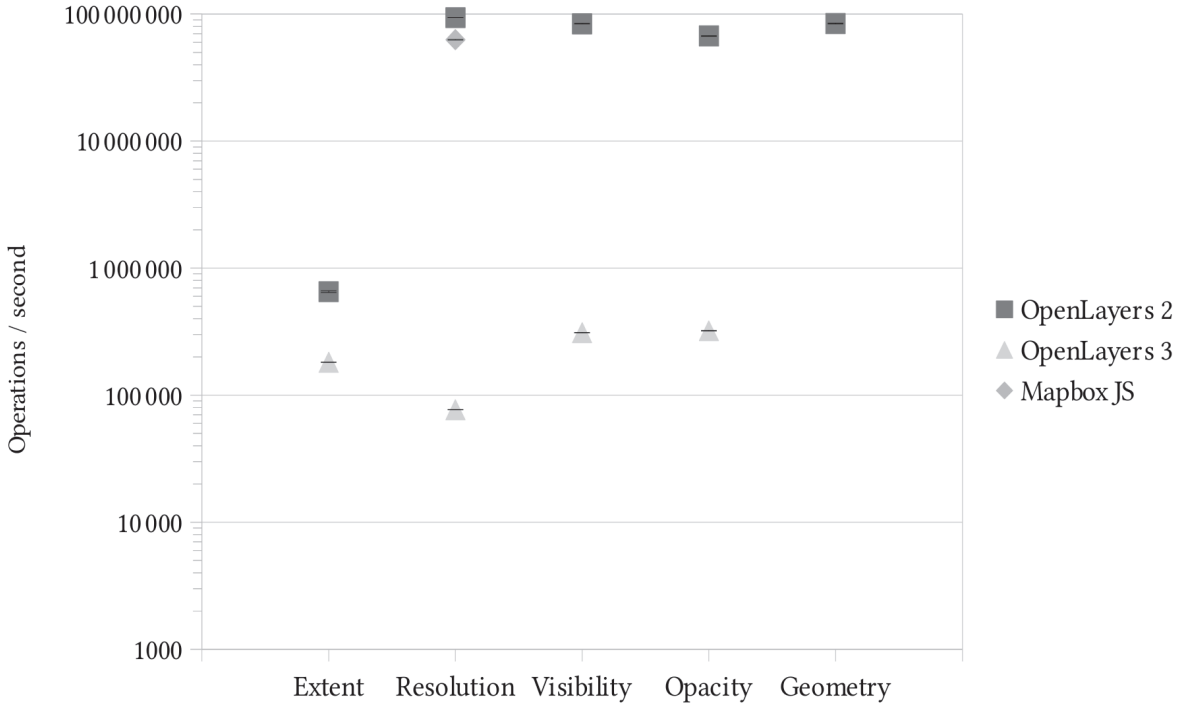

Figure 21: Benchmark diagram of layer attributes in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.4.3 Layer Events

The richness of events associated to layers can gradually improve the programmability of a library, and the dynamism of the application product. The libraries support a great number of events. Some of them are fired from the map object, while others are dispatched from other parts of the application. The study does not differentiate events fired by separate objects, but gathers the most essentials, which can be associated to layers. Luckily all of the analyzed libraries support creating custom events, if one is not supported natively.

The importance of browser events in a library cannot be emphasized enough. Events can be considered as extension cords to internal functions. When the user calls a function with an event dispatcher, it fires the event at some point. Along with other attributes, the event sends the object from which the event has been dispatched, wrapped in an object's target property. If an event listener is registered to the particular event, the listener function is called in the moment the event is fired. This way, any function which fires an event can be extended with custom functionality.

The study compares some of the most essential events in order to make a responsive WebGIS application. Adding layers, removing layers, changing layer order, and properties should be propagated, as custom widgets and GUI elements need to be updated (e.g. legend) after a change has occurred. Usefulness of the other analyzed events has been discussed above.

The support diagram in Figure 22 reflects the statements above. If an event is not natively supported by a library, it can be implemented as a custom event. Custom events need to be fired manually in the appropriate function. They support the object the event get dispatched from by default, however, they can be extended with additional data. When firing an event manually, supplement of additional information should be thoroughly considered.

	Add/Remove	Order	Property	Load	Features
OpenLayers 2	3	6	6	3	3
OpenLayers 3	3	3	6	6	3
Mapbox JS	3	3	3	3	3

: Supported
: Deployable

Figure 22: Support diagram of layer events in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, listeners to events related to adding, removing, changing order of, and changing property of layers have to be registered to the map object. There are multiple methods to register events in the library, but every one of them can be accessed through the events property of an object supporting events. The most convenient method is to use the events object's on method, which expects an object as an input with the event types as keys, and functions as values.

```
var event = function(){/*Do something*/};
map.events.on({addlayer: event});
```

Events related to changes in the state of a layer can be accessed through the changelayer event type. Every time a layer's name, position, opacity, parameters, visibility, or attribution changes, the event fires with the appropriate kind of change stored in the event object's property attribute.

```
var event = function(eventObject){
    if (eventObject.property === 'order') {
        /*Do something*/
    }
};
map.events.on({'changelayer': event});
```

Listeners to the other events have to be registered to every layer individually. Every layer object has a loadstart and loadend event, which can be listened to, while vector layers have additional events related to features, like featureadded and featureremoved.

```
var event = function(){/*Do something*/};
layerObject.events.on({loadstart: event});
layerObject.events.on({featureadded: event});
```

For the sake of completeness, OpenLayers 2 supports custom events, which can be fired by any object with a valid events object. Firing a custom event is very simple, one just have to call the event object's triggerEvent method with the event type and the additional supplements as arguments.

```
var event = function(){/*Do something*/};
map.events.on({customEvent: event)};
map.events.triggerEvent('customEvent', {additionalProperty: 'value'});
```

In OpenLayers 3, events fired on adding and removing layers can be listened on the layers object. Registering an event can be done with the object's on method. The method expects two arguments, the event type, and a function to execute. Every object has an on method in the library, which is an observable object, therefore descendant of the ol.Observable class.

```
var event = function(){/*Do something*/};
map.getLayers().on('add', event);
map.getLayers().on('remove', event);
```

To use custom events, one can use the on method on any event capable object, while fire events with the dispatchEvent method. This method does not accept any further arguments other than the event type, and provides a layer object with the propagator of the event in the target property.

```
var event = function() {/*Do something*/};
map.getLayers().on('changeorder', event);
map.getLayers().dispatchEvent('changeorder');
```

Listening to property changes can be done by registering an event listener to the layer object's propertychange event. Most of the observable objects have this type of event. The listener gets a key property in the event object as an additional value, which contains the property name, which induced the event.

```
var event = function(eventObject) {
    if (eventObject.key === 'myProperty') {
        /*Do something*/
    }
};
layerObject.on('propertychange', event);
```

The rest of the events are propagated by layer sources. Every source object have a change event, and a getState function, however, as stated by É. Lemoine (2014), it only works properly with static vector sources currently. The addfeature and removefeature events are working as intended, however they are only fired by vector sources.

```
var event = function(){
    if (layerObject.getSource().getState() === 'ready') {
        /*Do something*/
    }
};
layerObject.getSource().on('change', event);
layerObject.getSource().on('addfeature', event);
```

Mapbox JS also uses the on method to register event listeners with two arguments, the event type, and the executable function. Events fired by adding and removing layers can be listened on the map object. The type names of these events are layeradd, and layerremove.

```
var event = function(){/*Do something*/};
map.on('layeradd', event);
```

The event firing when the layer order changes can be implemented as a custom event, along with the property change event. Custom events can be fired by using a capable object's fireEvent or fire function. Both of the functions do the same. They require a type name as a first argument, while accept an object with key-value pairs as a second one.

```
var event = function() {/*Do something*/};
map.on('changeorder', event);
map.fireEvent('changeorder', {additionalProperty: 'value'});
```

Layer loading events are only partially implemented. They can only be listened on tile layers with the event types loading, and load. Feature related events can be listened on vector layers. The usual way to handle vector layers in Mapbox JS, is grouping them into feature groups, while letting the library add all of the features as separate layers. Adding and removing features can be listened as adding and removing layers on feature groups, but this way additional functionality should be implemented to avoid firing the map object's layerremove event with removing a feature.

```
var event = function(){/*Do something*/};
tileObject.on('loading', event);
tileObject.on('load', event);
vectorObject.on('layeradd', event);
```

The benchmark diagram in Figure 23 shows expected result. OpenLayers 2 is very agile in registering events, Mapbox JS offers a moderate performance, while OpenLayers 3 is the least efficient, but the most stable of the compared libraries. The unusually slow event registering on the map object is the only outlier in the result.

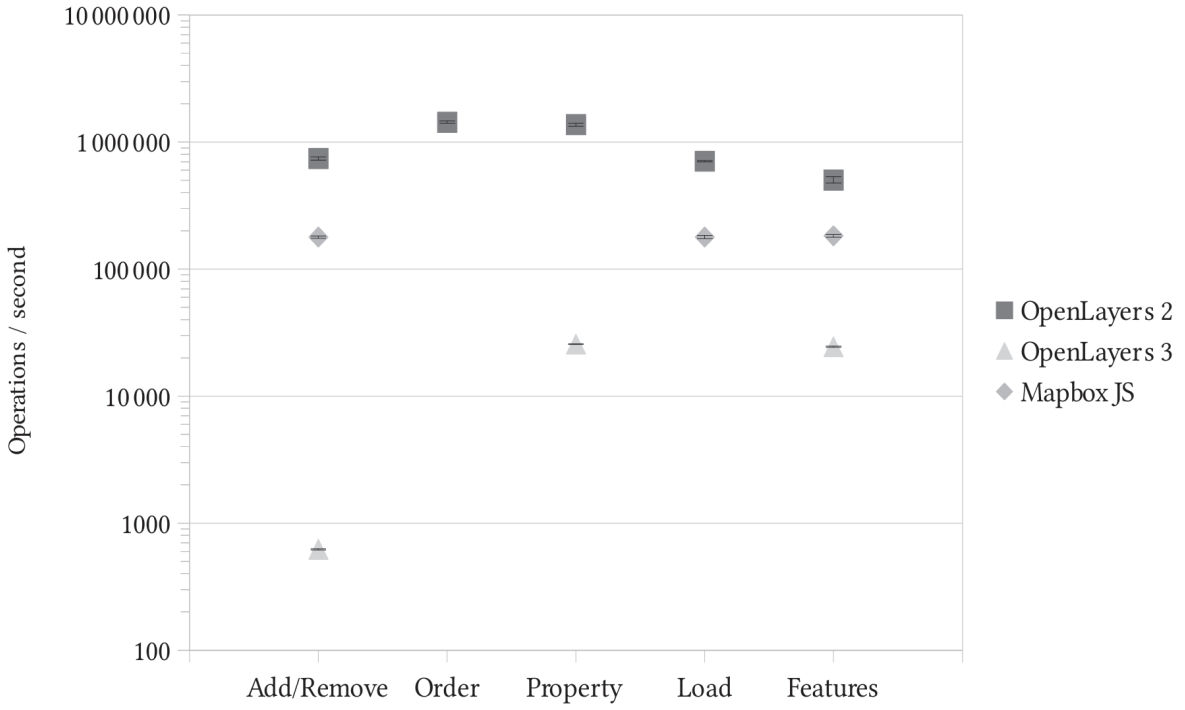

Figure 23: Benchmark diagram of layer events in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.5 Feature Management

There are two main types of data a spatial analyst has to deal with: raster and vector. Vector data tend to be faster to make calculations with. Image processing algorithms are hard to implement on the client side, as they usually work with large datasets, and do resource-intensive calculations. On the other side, topological calculations can be done in a browser with a decent performance. This makes feature management a key concept in choosing a web mapping library for a WebGIS application. Basic spatial operations, like solving PIP (Point in Polygon) problems, buffering features, or checking for intersections can, and should be done on the client side for load balancing. Naturally, this is only true for smaller datasets.

Attribute handling is another problematic concept in web mapping libraries. Attribute data for vectors are traditionally stored in relational databases. They have a fixed number of columns, containing a value for every feature in the dataset. Every column can contain only one type of value, while occasionally, additional rules are set for forbidding null values, or storing only unique values in the records. These are basic assumptions, a desktop GIS can work with to increase performance, and a WebGIS can't.

Web mapping libraries don't store headers to constrain the data type an attribute record can store. Furthermore, both internal data structure, and data exchange formats only store values which are not null by default to grant better performance, and optimize server load. Due to web mapping libraries' inconsistent attribute handling, additional filters should be implemented in order to make a reliable WebGIS application.

The study analyses feature management in four subsections. Supported geometry types determine the input capabilities of a library. Support for simple geometries are relatively easy to implement, therefore the main question is related to the support of multipart geometries. Basic spatial operations are also compared. They consist of calculating data extent, solving PIP problems, recognizing intersections, and calculating a feature's basic spatial properties: centroid, length, and area. Calculating data extent can be used for zooming to a vector layer, and narrowing down the subjects of spatial operations.

Attribute management compares the libraries' capability of getting, and setting attribute data. It also checks for consistent data handling, which is broken down to requesting headers, and validating input data based on headers. The last category is building an attribute table, which in this case an object, with records for every header element, regardless if a feature has a value for the given attribute.

Queries are closely related to attribute management. The study analyses three cases, attribute queries, spatial queries, and nested queries. These are functions, which evaluate an expression on a layer's features, and return an array with the list of features, which have passed the test. Libraries should be able to use queries as hard filters, filtering out every feature on which the query evaluates as false. Hard filters have to completely remove the filtered out features, thus no further operations are applied on them. On the other hand, the function must filter out features in a way, they remain recoverable, as data loss is unacceptable in a reliable WebGIS application.

4.5.1 Geometry Types

The study compares four basic geometry types. Point, line, polygon, and multipart polygon input were used to analyze the capabilities of the libraries. These are basic, 2D feature types, 3D capabilities are not discussed by the study. However, a notation should be set, there are JavaScript libraries designed to plot 3D data. One of them, Cesium JS, can be integrated into OpenLayers 3. The integration library is currently in development. The current release can be used with OpenLayers 3.3.0.

As the support diagram in Figure 24 presents, the studied geometry types are natively supported by all of the libraries. From the code, one can take the glance at the inner structure of geometry handling in the libraries. Mapbox JS has a very simple structure, as it stores coordinates in arrays. OpenLayers 3 uses a similar method to store geometry. It uses a different classes for every geometry type, while the coordinates are still stored as arrays. OpenLayers 2 has the most complicated structure, as every part of a geometry is an instance of the appropriate class. Multipart polygons are made of polygons, which contain linear rings, which consist of points. This way, every single part of the geometry has a great number of convenience methods to use, but writing them is tedious, while processing them is slower, and they use more memory.

	Point	Line	Polygon	Multipart
OpenLayers 2	3	8	10	20
OpenLayers 3	3	3	3	3
Mapbox JS	1	1	1	1

: Supported

Figure 24: Support diagram of geometry types in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, the complexity of a geometry can radically increase the length of its declaration. While the point class takes only two coordinates, every other geometry can only be described with subordinate classes. Every geometry constructor, except of points, accepts an array of subordinate constructors. For multipart features, the array have to contain their singlepart counterparts. Line constructors take points, while polygon constructors accept linear rings, which also consist of points. In a polygon, the first linear ring describes the polygon, while the subsequent ones mark the interior boundaries (holes in it).

OpenLayers 3 and Mapbox JS require an array of coordinates as input (even for point geometries). The more complex a geometry is, the more dimensions the input array has. This results as a more compact input, but it also becomes more confusing to code it, especially with high dimension numbers. The required number of dimensions follows the pattern of OpenLayers 2's inner structure. The only difference is in Mapbox JS, as it accepts one dimension less for describing polygons without holes.

The benchmark diagram in Figure 25 shows partially unexpected results. Despite the fact, that OpenLayers 2 constructs a geometry object for every point, it still achieves a greater performance, than Mapbox JS. This phenomenon might be due to the nature of Mapbox JS, creating a separate layer for every feature, while in OpenLayers 2, the layer object is already initiated when the feature is added by the benchmarked function. OpenLayers 3 can completely exploit the advantages of its simpler geometry structure, offering a great performance in adding features to a layer.

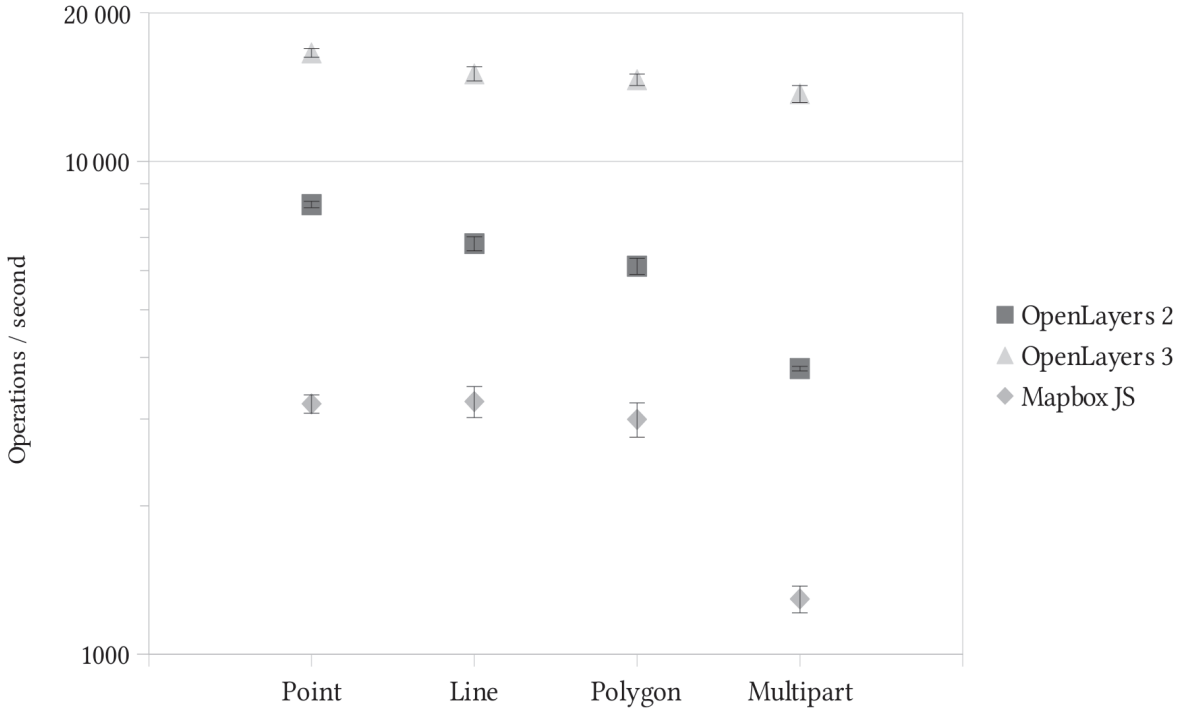

Figure 25: Benchmark diagram of geometry types in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.5.2 Spatial Operations

As mentioned above, spatial operations on smaller vector data could, and should be done on the client side in order to balance load, and increase performance. There are a wide palette of useful spatial algorithms used by analysts to gather additional information from vector datasets. Most of these algorithms are verbose, therefore implementing all of them in a web mapping library would drastically increase its size.

Luckily, there are some JavaScript libraries dedicated to perform topological operations on features, like Turf JS, JSTS, or GeoScript JS. They can be fed with GeoJSON or WKT representations of features, and produce a similar output, which can be read by the web mapping libraries. Most of these algorithms, like buffer, union, or spatial join should be handled by an individual topological suite, therefore these capabilities are only loaded, when needed, otherwise not consuming memory space unnecessarily.

Some of the spatial algorithms, however, should be available as core features. These are operations, which don't produce an output layer, but used to test features in a rapid manner. They usually used to select or filter features by their locations, and the emphasis is on performance.

As spatial operations only have to fulfill spatial queries with decent performance in the libraries, full implementations of computational geometry algorithms are not necessary. Most of the cases, they only have to produce a binary value as a test result, or calculate a simple spatial parameter of a feature.

From the test cases the extent calculates the data bounds of a layer. The PIP test checks if a point is in a polygon, or not. This feature can be used for many purposes, but most of the time, it is used to determine if the vertices or the centroid of a feature is in a user defined selection box. The same ascertainment applies to the intersect operator, too, however it can be applied to non point geometries easier, but yields worse performance. Feature length and area are used for measurments, and measurment related analysis.

The support diagram in Figure 26 indicates, only OpenLayers 2 has all of the analyzed capabilities. OpenLayers 3 supports most of them natively, however measuring a shape's length requires some extra lines of code. Mapbox JS supports PIP analysis via the PIP plugin. Furthermore, both length, and area measurements have to be reimplemented. The examples use the Pythagoras theorem for length calculations, and the geoalgorithm created by D. SUNDAY (2012) for calculating the area of an irregular polygon. In all of the libraries, taking geodesic measurements are possible, in OpenLayers 2, every geometry object has convenience functions for it. Intersections in OpenLayers 2 are detected with a minimalistic implementation of the Bentley–Ottman algorithm.

	Extent	PIP	Intersect	Centroid	Length	Area
OpenLayers 2	1	1	1	1	1	1
OpenLayers 3	1	2	NaN	NaN	14	2
Mapbox JS	1	6	NaN	NaN	16	15

: Supported
: Deployable
: Not supported

Figure 26: Support diagram of spatial operations in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, the available spatial operations can be easily invoked. Calculating the data extent requires a vector layer, while the other functions have to be called by geometry objects. All of the spatial functions can be used by any geometry object. Intersection and PIP analyzes require an additional geometry as input. The area and length properties are measured on the map's coordinate system, and returned in the appropriate unit, while the getGeodesicLength, and getGeodesicArea functions return measurements on a spheroid in meters.

```
layerObject.getDataExtent();
layerObject.features[0].geometry.intersects(
    new OpenLayers.Geometry.Point(0,0));
```

OpenLayers 3 offers similar methods for gathering the data extent of a vector layer, and calculating a feature's area. However, gathering spatial properties of shapes are not consequent, thus gathering area is only possible for polygon features. There is a native getLength function in the library, but as it only can be called from line strings, and polygons also have perimeter, the study defined a custom getLength function for polygon geometries. The function also have a caveat, as it does not deal with interior rings, just with the outer boundary of a simple polygon.

```
var coordArr = this.getCoordinates()[0];
var length = 0;
for (var i=1;i<coordArr.length;i++) {
    length += Math.sqrt(Math.pow(coordArr[i-1][0] - coordArr[i][0],2) +
              Math.pow(coordArr[i-1][1] - coordArr[i][1],2));
}
```

The solution for intersection problems in OpenLayers 3 is limited to extents. Every geometry has an intersectsExtent method, which compares the geometry's extent with an input extent. This method can be effectively used for solving PIP problems, as any point's extent is the point itself. The method uses a hidden containsXY function, if the library considers the conditions as a PIP problem, which is not exported, thus gets obfuscated in the final, compressed library. The containsXY method uses an implementation of the Jordan curve theorem, therefore the intersectsExtent method does the same, when used with a point and a polygon geometry, and gives false result when the point is in a complex polygon's self-intersection.

```
var geom = lyr.getSource().getFeatures()[0].getGeometry();
geom.intersectsExtent(new ol.geom.Point([0,0]).getExtent());
```

Mapbox JS has limited capabilities in the field of spatial operations. Natively it can calculate the data extent of a layer, and measure geodesic distances between points. Every other method is supported with some sort of external code. Solving the PIP problem can be done with the PIP plugin, which requires two inputs: a layer, and a point. As a result, it returns an array with the polygons the point is inside. There is a third, optional boolean argument, which stops the search on the first valid intersection.

```
leafletPip.pointInLayer(L.latLng(0,0), map.featureLayer);
```

Calculating length and area in a flat plane can be done with the Draw plugin, or manually implementing the algorithms. As they're easy to write, the study implemented them manually, and will deal with the Draw plugin later. The length measurement is achieved as in OpenLayers 3, but for calculating a polygon's area, the study used a triangulation algorithm originating from Dan Sunday.

```
var coordArr = featureObject.getLatLngs() || [];
var area = 0.0;
coordArr.push(coordArr[0], coordArr[1]);
for (var i=1;i<coordArr.length-1;i++) {
    area += coordArr[i].lng * (coordArr[i+1].lat - coordArr[i-1].lat);
}
area /= 2.0;
```

The benchmarking diagram in Figure 27 shows, OpenLayers 2 delivers the best performance in spatial operations. The other libraries' capabilities are limited, however, OpenLayers 3 also gives good results in the natively supported features. Mapbox JS's extent calculations are also fast enough to not influence user experience. The similar test results of OpenLayers 2's intersection and PIP analysis worth a notion. This phenomenon is caused by a filter in the main code, which recognizes PIP problems when intersect is called, and evaluates the much cheaper containsPoint method on them. Therefore, the two operations are almost the same this case.

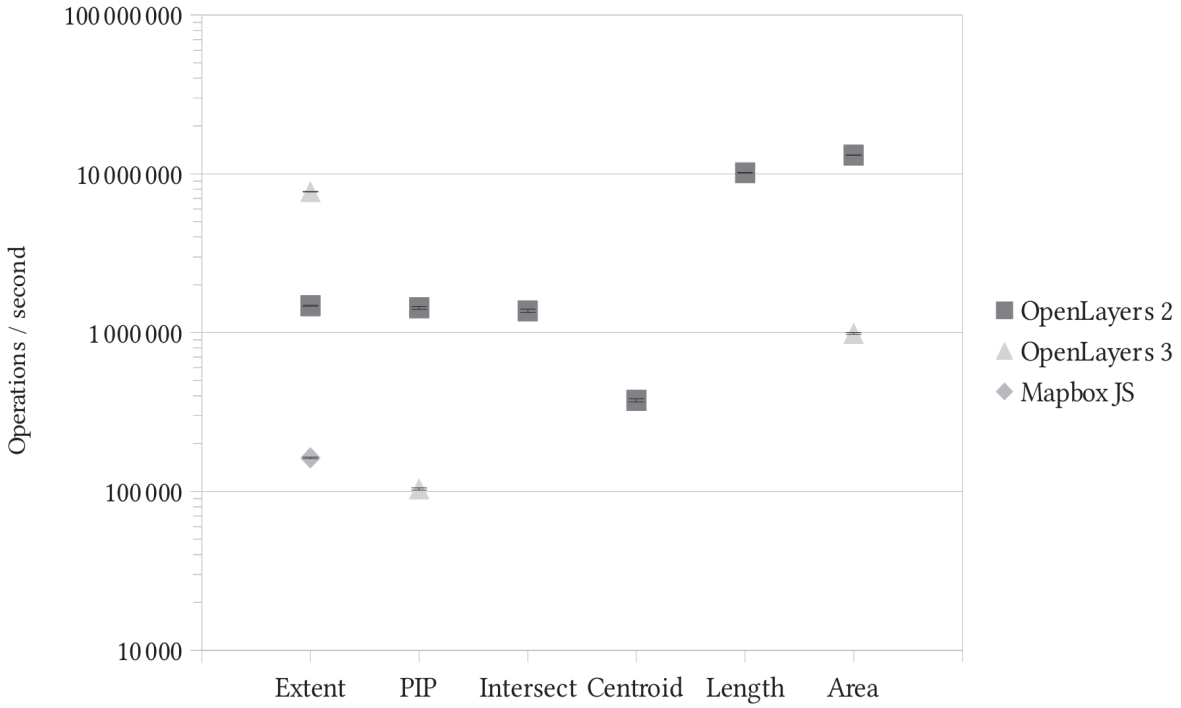

Figure 27: Benchmark diagram of spatial operations in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.5.3 Attribute Management

The study splits attribute management into five parts in order to detect the libraries' capability of handling attributes in a consistent, reliable way. Handling attribute data can be an important aspect, when a WebGIS application is used for statistical analysis, or when digitizing features with a list of attributes just to mention a few cases. Storing and getting attributes are natively supported in all of the libraries. They can accomplish this tasks in a convenient way, however setting attributes can be slightly more difficult. Apart from OpenLayers 3, this method has to be implemented manually. In order to set the attributes manually, the properties object of the features need to be accessed, and the input data has to be inserted in an iterative way.

Headers and validation are needed for the libraries to handle attributes reliably, and to be responsive. If every attribute has a stored header object with an attribute type, the validating function can easily test, or even sanitize the input data, and occasionally inform the user if it encounters an invalid input. A responsive application, which sanitize input attributes, and prompts invalid input, can make users feel greater control over their works, therefore the application delivers better user experience.

Drawing an attribute table is a basic GIS feature. Attribute tables offer a simple, but comprehensive overview of a vector layer's attribute data. Based on the implementation style, an attribute table can act as a simple output, or an input as well. In production use, gathering the required attributes should be linked with the drawing function for an increment in performance. However, the example function only gathers attributes in an object, as the drawing process should not be handled by any library, as it would drastically decrease their scalability. The function gathers null values along with attribute values to preserve the schema of a relational database.

The support diagram in Figure 28 indicates, only accessing a feature's attributes is universally supported in the libraries. There is a convenience method for setting attributes in OpenLayers 3, and the custom implementations in the other libraries are following its input scheme. The rest of the test functions have to be coded in the form of extensions, but they can be achieved without greater difficulties.

	Get attributes	Set attributes	Headers	Validation	Attribute table
OpenLayers 2	1	6	9	14	13
OpenLayers 3	3	1	10	14	13
Mapbox JS	1	6	9	14	12

: Supported
: Deployable
: Not supported

Figure 28: Support diagram of attribute management in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Getting the attributes associated to a feature in OpenLayers 2 is very easy, as every feature object has an attributes property. This property is also an object, which only contains the attribute names and values in a key-value pair format, thus making it iterative.

Setting attributes can be done with a function, which expects a feature object, and an attributes object as an input. The function then simply iterates through the attributes object, and sets every attribute provided there to the feature object.

```
function setAttributes(featureObject, attributesObject) {
    for (var i in attributesObject) {
        featureObject.attributes[i] = attributesObject[i];
    }
}
```

Gathering headers is working in a similar way, but it executes two iterations. The first one cycles through the input layer's features, while the second one looks up every attribute name in a given feature. An array is declared at the beginning of the function, which gets filled with unique attribute names.

```
var headers = [];
for (var i=0;i<layerObject.features.length;i++) {
    for (var j in layerObject.features[i].attributes) {
        if (headers.indexOf(j) === -1) {
            headers.push(j);
        }
    }
}
```

The validating function assumes, the input layer has a header object with attribute names and types in it, paired up as key-value pairs. The type can be int or float, every other value is handled as a string. Similarly to setting attributes, the function iterates through the input attributes, but prior to updating the input feature object, it looks up the attribute name in the layer's header object. If the attribute should be a number, it runs the value through the appropriate JavaScript filter. In any other case, it converts the value to string. Obviously, the function is a minimalistic implementation, as it could include communication methods with the user instead of assuming one's intentions.

```
function setValidatedAttributes(featureObject, attributesObject) {
    for (var i in attributesObject) {
        if (featureObject.layer.headers[i] === 'int') {
            var attribute = parseInt(attributesObject[i], 10);
        } else if (featureObject.layer.headers[i] === 'float') {
            var attribute = parseFloat(attributesObject[i]);
        } else {
            var attribute = String(attributesObject[i]);
        }
        feature.attributes[i] = attribute;
    }
}
```

The attribute table creating function iterates through every feature to collect values. First, it gathers the attribute names from the layer's header object. With the assumed attribute names, it tries to fill the rows of the table with attribute values from each feature. If a feature can't provide value for a given attribute, the function fills its place with a null value. Every feature is represented by its ID in the resulting object, or if it does not have one, the index of the current cycle.

```
var attrTable = {};
for (var i=0;i<layer.features.length;i++) {
    var tableEntry = attrTable[feature.fid || i] = {};
    for (var j in layer.headers) {
      tableEntry[j] = feature.attributes[j] ? feature.attributes[j] : null;
    }
}
```

In OpenLayers 3, accessing a feature's attributes is possible via the built-in getProperties method. The method returns every property bound to the feature, including the geometry. The geometry, this way, have to be ignored for further processing. However, the filtering need to be applied with further considerations, as invalid features without geometry can be added to a layer. Setting attributes can be achieved with the setProperties method, which can be called from a feature object, and requires an attribute object as an input, like in the OpenLayers 2 example.

```
var attributesObject = featureObject.getProperties();
delete attributesObject.geometry;
```

The header requesting example uses the library's capability of returning just the property names in an array. The method is called getKeys. The function iterates through every feature, requests the attribute names from the feature, and creates the union of the existing array, and the returned one, while dumping duplicated values. For this task, JavaScript's native array manipulation methods were used.

```
for (var i=0;i<featuresObject.length;i++) {
    var keys = featuresObject[i].getKeys();
    keys.splice(0,1); //Removing geometry as a key
    headers = headers.concat(keys.filter(function (member) {
        return headers.indexOf(member) === -1;
    }));
}
```

Validating input, and drawing attribute table use the same function, as in OpenLayers 2. However, some additional considerations should be taken into account. The header object should be bound to the source object, not the layer, as the layer can change its source, and doing so, the underlying data completely changes. In OpenLayers 3, if a feature has an ID, it can be accessed via its getId method. Finally, the attribute table should be called from the source object, not the layer.

Mapbox JS works similar to OpenLayers 2. There are some differences, which have to be discussed. Mapbox JS stores every feature attribute in a feature's properties object. Furthermore, as the library creates an individual layer for every feature, and groups them in a feature group, representing the vector layer, every feature has to be accessed via a feature layer's feature property. Every feature layer should contain only one feature, making attribute management automatable. In production, the algorithms should be able to recognize multipart features, as they are also stored in feature groups, containing the parts in separate layer objects.

For iterating through a layer's features, the layer's eachLayer function can be used. Using the method needs to make an assumption for layer structure. Input layers must be feature groups, and should contain a header object. As features can be single- or multiparts, a production implementation should be able to effectively distinguish them.

```
var attrTable = {};
layerObject.eachLayer(function(featureLayer) {
    var tableEntry = attrTable[featureLayer._leaflet_id] = {};
    [...]
});
```

Benchmarking attribute management didn't led to any new significant result. Only getting attributes could be compared among libraries. As getting and setting features cannot work with the same test data (as get features takes no input), the two cases cannot be effectively compared in OpenLayers 3. For only one feature, the study doesn't publish a diagram, just the raw data.

OpenLayers 2 performed 750.837.084 operations, while Mapbox JS achieved 106.672.191 operations per second in getting attributes. These great values are due to the direct query of the parent object. OpenLayers 3 offers 429.502 operations per second on the same field, while 115.855 operations per second on setting attributes. As OpenLayers 3 communicates with its inner structure via helper functions, the decreased performance comes with more inner filters, greater stability, and scalability. The margin of error was below 1% in most of the cases, except for OpenLayers 3's getting attribute function, where it was 3.59%.

4.5.4 Queries

Queries are bound strictly to attribute management and spatial operations. They are testing functions, evaluating an expression on a feature, and returning a boolean value. The expression can be an attribute, or a spatial test. Queries can be used to select or filter out features in a layer, decreasing the set, which will farther operations process.

There are four cases the study analyses in the subject of queries. Attribute queries are attribute related expressions. Spatial queries tend to check for the placement of a feature's geometry compared to the input feature's geometry. They can involve intersection, PIP, within, or containment tests, just to name a few. The study only tests intersection query, as it is implemented in at least one of the libraries, and can be effectively used for filtering features regardless of geometry type. Nested queries analyses the libraries' capability of applying different filters connected with logical operators.

Filtering layers represent a functionality to apply a hard filter on a layer with an expression. The filter effectively removes all of the features from a layer, on which the expression evaluates as false, and stores them in a temporary store. If the filter changes, the filter should automatically restore the layer to its original state, and apply the new filter to it. The filter must operate without data loss, as in a reliable WebGIS application data loss is unacceptable.

The support diagram in Figure 29 shows, except of spatial filters, the libraries can be prepared for being capable of filtering with some supplemental code. As from spatial operations, PIP tests can be applied to geometries in all of the libraries, they offer a limited support for spatial queries. However, for precise calculations in case of geometry types with more than zero dimension, they are not sufficient. For example, OpenLayers 3's intersectsExtent method can only be applied to a geometry and another geometry's extent. If the input geometry is not a point, there are several cases, when its extent intersects the other geometry, while in reality, there is no intersection between them.

	Attribute	Spatial	Nested	Filter layer
OpenLayers 2	14	13	23	6
OpenLayers 3	38	NaN	38	48
Mapbox JS	37	NaN	37	3

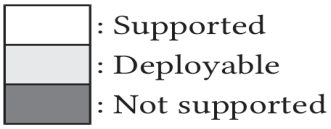

Figure 29: Support diagram of queries in OpenLayers 2, OpenLayers 3, and Mapbox JS.

OpenLayers 2 offers a set of convenient, but verbose methods for evaluating filters on features with its Filter class. The class has attribute, spatial, and logical filters, making them ideal for nesting expressions with great ease. A constructed filter has to be parameterized according to its type. Comparison filters take an attribute name, value or values based on the comparison type, and an operator. Spatial filters need a geometry object as a basis of comparison, while logical filters accept other filters. All of them have an evaluate method, which needs to be fed with a feature, and evaluates it based on the context.

```
var filter = new OpenLayers.Filter.Logical({
    type: OpenLayers.Filter.Logical.OR,
    filters: [
        new OpenLayers.Filter.Spatial({
            type: OpenLayers.Filter.Spatial.INTERSECTS,
            value: geometryObject
        }),
        new OpenLayers.Filter.Comparison({
            type: OpenLayers.Filter.Comparison.EQUAL_TO,
            property: 'name',
            value: 'Hungary'
        })
    ]
});
filter.evaluate(featureObject);
```

To apply a hard filter to an entire layer, the layer needs to be constructed with a Filter strategy. The strategy should remain accessible, as its setFilter needs to be called in order to update the layer's active filter. The filtered out features get stored in the strategy object, when a filter is applied.

Unfortunately, OpenLayers 3 does not have any filtering capability implemented, yet. The manual extensions provided by this study follows OpenLayers 2's filtering pattern, with decreased verbosity. The attribute filter requires a feature, an attribute name, an attribute value, and an operator. This is a minimalistic implementation, which only handles a limited set of comparisons. The evaluating process is simple, it checks the type of the operator, then compares the provided attribute value with the feature's based on it.

```
function filterFeature(feature, property, value, operator) {
    var bool = false;
    if (feature.getProperties()[property]) {
        var featProp = feature.getProperties()[property];
        switch (operator) {
            case "==":
                bool = (featProp == value);
                break;
            [...]
        }
    }
    return bool;
}
```

Nesting these filters only needs several function calls and logical operators between them. Applying a hard filter to a layer also needs a custom implementation. The source object needs to store the filtered out features, and add them back to the layer, when a new filter is set. For this purpose, a filtered property is declared in the source object.

```
function filterLayer(layer, property, value, operator) {
    var featArr = [];
    var source = layer.getSource();
    source.filtered = source.filtered || [];
    if (source.filtered.length > 0) {
        source.addFeatures(source.filtered);
        source.filtered = [];
    }
    var features = source.getFeatures();
    for (var i=0;i<features.length;i++) {
        if (! filterFeature(features[i], property, value, operator)) {
            featArr.push(features[i]);
            source.removeFeature(features[i]);
        }
    }
    source.filtered = featArr;
}
```

In Mapbox JS, the custom implementations are nearly the same as in OpenLayers 3. The only difference is, they need to check the feature's properties object to evaluate the expression. The library natively supports hard filters on a layer. The filter takes a function as an input, which should return a boolean value. The function gets every feature as input, therefore any custom test can be called on them.

```
map.featureLayer.setFilter(function(featureObject) {
    return (featureObject.properties.name == 'Hungary');
});
```

The benchmark diagram in Figure 30 is gapped, however it emphasizes an important fact. As the spatial filter in OpenLayers 2 tests polygon intersections, the real intersects method has been called. Compared to the attribute filter, and the nested filter, which is the simultaneous evaluation of the spatial, and the attribute filter, the real expense of checking for intersections can be seen. It drastically decreases performance, and should be called with caution, especially on large datasets.

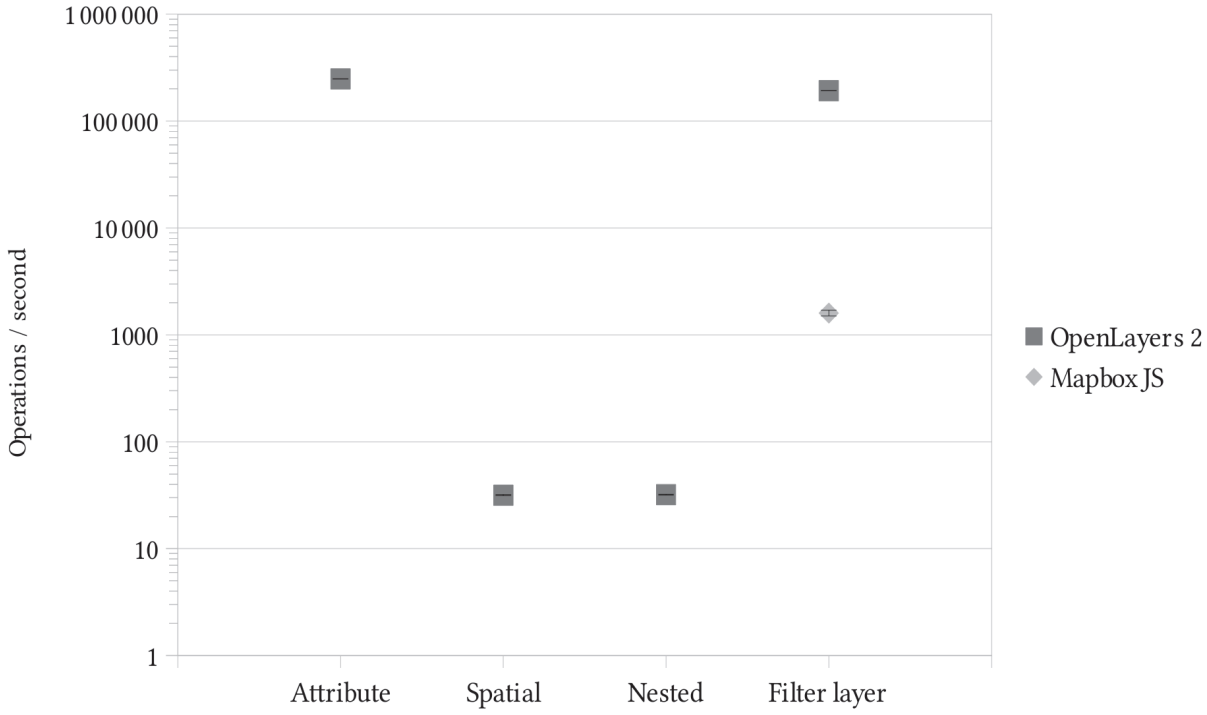

Figure 30: Benchmark diagram of queries in OpenLayers 2, and Mapbox JS.

4.6 Map Controls

Map controls act as a medium, or interface between the user, and the WebGIS application. Through map controls, the user should be able to change the view, rotate the map, draw, or modify features, or just measure distances. There is a wide range of map controls users need in order to use the application smoothly, and be productive. No matter how smart an application is, it is only as good, as it is capable of communicating with the user. This makes map controls a key feature in the analysis.

The study classifies map controls into four categories. Projection controls define the capability of a library to handle different projections. As a versatile WebGIS application should handle different projections, and on the fly projecting, this is a very important feature. The nature of web based raster layer formats constrains the possibilities of playing with projections on the client side. Web mapping libraries traditionally don't even have raster warping capabilities. On the other hand, handling vector layers with different projections, therefore bringing them to a common denominator should be supported.

View controls are fairly simple, and expected to be mostly supported by all of the libraries. The users should be able to reposition the map with a pair of coordinates. Rotating the view became a popular demand recently, with the implementation of a similar control in desktop GIS applications.

Interactions are feature based controls, which ought to handle the user's intentions for drawing, selecting, or modifying features, helping digitization, or measuring with points, lines, or polygons. Any web mapping library capable to offer handlers for such actions, is able to support these controls. Miscellaneous controls are hard to categorize in a general class. The study considers popups, layer switcher, and the ability of building custom toolbars as the most valuable non-classifiable features.

4.6.1 Projection

Projecting (plotting coordinates measured on an ellipsoid to a flat plane) is the heart of every GIS application. Most of the desktop GIS applications have support for a huge amount of map projections. Most of the existing coordinate reference systems concerning the Earth, are collected, and maintained in the EPSG Geodetic Parameter Dataset. With the dataset, referring to a projection as its EPSG code became a common practice.

Two projections are used widely in web mapping applications. Google's Web Mercator (EPSG:3857 or EPSG:900913) is a semi-conformal cylindrical projection. It originally used a sphere, then the WGS84 ellipsoid as its datum, but as S. E. Battersby et. al. (2014) states, it has still projected WGS84 coordinates as if they were on a sphere. It uses meters as units, and well-known for its lack of area preservations, especially near the poles. The other commonly used projection (EPSG:4326) is a direct plotting of latitude and longitude coordinates. It also uses the WGS84 ellipsoid as its datum. It is an equidistant cylindrical projection, also called as the Plate Carrée projection. Due to the direct plotting of coordinates to a flat plane, it uses degrees as units.

The first class of the analysis covers the support of the two default web mapping projections. Further projection support is compared under the custom projection class. Supporting various projections can increase the usability of a WebGIS application, as vector data don't have to be preprocessed. Fortunately, the PROJ.4 library supports all of the commonly used projections, and its JavaScript adaption, Proj4js can be integrated in all of the compared libraries.

The third case the study tests is the libraries' capability of changing projections in a single session. It is commonly known as on the fly transformation, and can be used to change the projection dynamically, if the workflow defines one as more appropriate. Finally, the study checks the ability of transforming vectors automatically between projections. If a library is capable of achieving such results, it is presumable, it can transform single pair of coordinates.

The support diagram in Figure 31 shows, the tested features are mostly implementable in most of the libraries. Mapbox JS does not support dynamic projection changes, but custom projections can be applied to a map with external plugins. The basic library, which must be included with all of the libraries is Proj4js, however Mapbox JS also needs an integrating library for custom projection support, called Proj4Leaflet.

	Default	Custom	Change	Reproject
OpenLayers 2	13	16	34	25
OpenLayers 3	19	20	24	17
Mapbox JS	12	18	NaN	6

: Supported
: Deployable
: Not supported

Figure 31: Support diagram of projections in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Defining projections in OpenLayers 2 can be done at object construction. The projection parameter has to contain the EPSG code of the desired projection definition. Web Mercator should be referred as EPSG:900913, as the library does not recognize EPSG:3857 in all of the cases, while server side applications should be able to recognize the two projection codes as aliases.

Including the Proj4js library (OpenLayers 2 only supports Proj4js version 1.1.0 or lesser natively), results in the capability of requesting other definitions from locally stored definition files, or from the library itself. The Proj4js library does not store the projection definitions, it just converts them into projection objects, which can be used by OpenLayers 2. Once a projection has been defined with Proj4js, OpenLayers 2 accepts it like one of the default projections.

```
Proj4js.defs['EPSG:23700'] = '+proj=somerc +lat_0=47.14439372222222
+lon_0=19.04857177777778 +k_0=0.99993 +x_0=650000 +y_0=200000 +ellps=GRS67
+towgs84=52.17,-71.82,-14.9,0,0,0,0 +units=m +no_defs';
map = new OpenLayers.Map('map', {
    projection: 'EPSG:23700',
});
```

There are several important changes to notice, when Proj4js is included. The whole projection system of OpenLayers 2 gets altered, as it won't gather projection information from its default set of projections, but from the Proj4js definitions dictionary. EPSG:3857 cannot be used as an alias to Web Mercator, if it is not defined with its PROJ.4 definition. Also, with version 1.1.0, transforming WGS84 coordinates near the projection bounds is faulty. These problems should be handled correctly, thus appropriate considerations must be set.

As usual, the study uses a minimalistic approach to define a function capable of switching between projections in the same session. There are only a few key concepts, which should be emphasized. The map's projection is defined with a string, containing the EPSG code of the current projection. The projection's units and bounds can be stored in the map object, therefore if they are present, they need to be changed. All of the layers contain a projection object, which contrary to the map object, contain the current projection in an object form. Layers also can have units, and bounds defined.

In order to change the projection, one has to change the map's projection parameter, its units, and its maxExtent parameters. As raster layers cannot be warped, their projection also need to be changed. For increased stability, the application should be aware of the valid projections a raster layer's source server supports for the given layer. After all of the parameters are set, the library has to recalculate the resolutions bound to zoom levels. Vector layers on the other hand, should not be altered, but transformed to the new projection. This can be automatized by calling their refresh method.

```
for (var i=0;i<map.layers.length;i++) {
    if (!( map.layers[i] instanceof OpenLayers.Layer.Vector)) {
        map.layers[i].projection = newProj;
        map.layers[i].units = map.layers[i].units ? newProj.proj.units :
            null;
        map.layers[i].maxExtent = map.layers[i].maxExtent ?
            map.layers[i].maxExtent.transform(oldProj, newProj) : null;
        map.layers[i].initResolutions();
    } else {
        map.layers[i].refresh();
    }
}
```

Transforming vector layers in OpenLayers 2 is simple, but has a significant caveat, which should be handled with considerations. As the map only stores the string representation of the current projection, the object is derived from the base layer's projection parameter, which is the first layer, if no dedicated base layers are present. Therefore, in order to transform a vector layer, the composition should have a base layer defined in the target projection, or else the map will be rendered in the vector layer's projection.

OpenLayers 3 offers easy to implement, convenient methods for projection handling. Its rethought and wonderfully designed inner structure allows developers to easily define and change between projections. The projection has to be declared in the view object, only vector layers support additional projection definitions. To apply a projection to the map, the projection parameter of the view object has to contain the string representation of the projection's EPSG code.

Defining custom projections can be done with the Proj4js library, similarly to OpenLayers 2. OpenLayers 3 supports version 2.2 and greater of Proj4js. With the custom projection defined with its PROJ.4 definition in Proj4js, OpenLayers 3 can refer to it with its name of definition (usually its EPSG code). Changing between projections can be done by applying a new view object to the map with its setView method.

Vector layer sources offer two projection related properties, the projection, and the defaultProjection. In order to transform a vector layer automatically, both of the parameters need to be set. The projection must contain the map's projection (destination projection), while the defaultProjection contains the projection of the input data. This behavior requests for some additional coding when changing projection.

Mapbox JS has much simpler mechanics in projections. The default projections can be defined as a map parameter (crs), and dedicated projection objects has to be used. The default one is Web Mercator, for other projections, the object has to be provided. On initialization, a layout is created based on the map's projection. This increases the performance of adding further layers, but decreases scalability, as changing between projections is not possible.

Defining a custom projection needs two third party softwares, the Proj4js library, and the Proj4Leaflet plugin. Every natively not supported projection has to be constructed with Proj4Leaflet, from its PROJ.4 definition, and a set of resolutions. Finally, the constructed object can be used to add support for the projection. If a constructed projection is used, the layers must have their continuousWorld parameter set to true in order to show up in the composition. The resolution array can be defined properly in an experimental way, however the powers of 2 is a great starting point, as tiled sources often use such metrics.

```
var eov = new L.Proj.CRS('EPSG:23700', '+proj=somerc
+lat_0=47.14439372222222 +lon_0=19.04857177777778 +k_0=0.99993 +x_0=650000
+y_0=200000 +ellps=GRS67 +towgs84=52.17,-71.82,-14.9,0,0,0,0 +units=m
+no_defs',
    {
        resolutions: [8192,4096,2048,1024,512,256,128,64,32,16,8,4]
    }
);
```

Mapbox JS always assumes a vector layer is in plain WGS84 longitudes and latitudes. Therefore, on initialization, it tries to transform the contents to the map projection from EPSG:4326 projection. To override this behavior, and add layers in other projections, Proj4Leaflet has to be included, and the layer has to be constructed with its L.Proj.GeoJSON constructor.

The study did not benchmark projection support, hence they are provided as constructor parameters. The benchmarking would have only compared the performance of creating map objects, and adding layers. Changing projection could have been compared, if OpenLayers 2 had a native support for it, but as a custom implementation is provided, it opted out from testing.

4.6.2 View

In terms of map view, the libraries' capability of setting the view, getting the extent, setting an extent as map view, and rotating has been studied. Setting the view programmatically is a basic expectation from a library. Getting the extent can be used to mark special regions, like computational regions, or bookmark regions manually. Fitting a bound to the view can be used to zoom to a layer's extent, while rotating has much applicability from digitizing to composing.

There is a map metric, which is commonly used in desktop GIS applications, but cannot be correctly implemented in WebGIS environments. Scales can only be estimated by any web mapping library, therefore calculating it precisely is impossible. As P. Robins (2010) details on his website, scale is calculated from map resolution and display resolution. Map resolution is a given attribute for every layer or map, but as long as JavaScript cannot access the screen's physical dimensions, it can only make assumptions for the scale.

According to the support diagram in Figure 32, except for rotating the map, the libraries offer convenient methods for view management. As the development of OpenLayers 2 is finished, it won't grant support to map rotation in the future, and as stated by V. Agafonkin (2015), it won't be implemented in Mapbox JS either in the near future.

	Set view	Get extent	Fit bounds	Rotate
OpenLayers 2	1	1	1	NaN
OpenLayers 3	2	1	1	1
Mapbox JS	1	1	1	NaN

: Supported
: Not supported

Figure 32: Support diagram of view controls in OpenLayers 2, OpenLayers 3, and Mapbox JS.

OpenLayers 2 can change its view with easily parameterizable, short methods. The extent of the current view can be accessed via the map's getExtent function. Changing the view can be done by the setCenter function, which requires an array of coordinates, and a zoom level. Fitting an extent to the view is achieved by the zoomToExtent method, which only requires an array of four coordinates, and tries to find the most appropriate view from which the provided extent can be seen.

In OpenLayers 3, view related properties should be accessed from the view object. For setting the view, there are two separate methods, setCenter, and setZoom. They have to be called separately, not in a combined form, like in the other libraries. Getting the extent, and fitting bounds are not trivial in the library. To get the current extent, one has to calculate it from the size of the map with the view object's calculateExtent method. To fit bounds, not just the bounds array has to be sent to the appropriate method, but the size of the map along with it. Rotating the view is very simple, but the degree of rotation has to be set in radians.

```
map.getView().calculateExtent(map.getSize());
map.getView().fitExtent([1692000,5669000,2672000,6275000], map.getSize());
```

Mapbox JS's view controls work similar to OpenLayers 2's, with the difference, the array of bounds has to be provided as a two dimensional array. The array has to contain two arrays with the coordinates of the transversal corner points of the extent in latitudes and longitudes.

As it can be seen on the benchmarking diagram in Figure 33, Mapbox JS shows great performance in general map controls. However, when more complex calculations are needed, OpenLayers libraries take the lead. Despite its non-trivial, and sometimes quite inconvenient methods, OpenLayers 3 seems to handle view controls in the most efficient way.

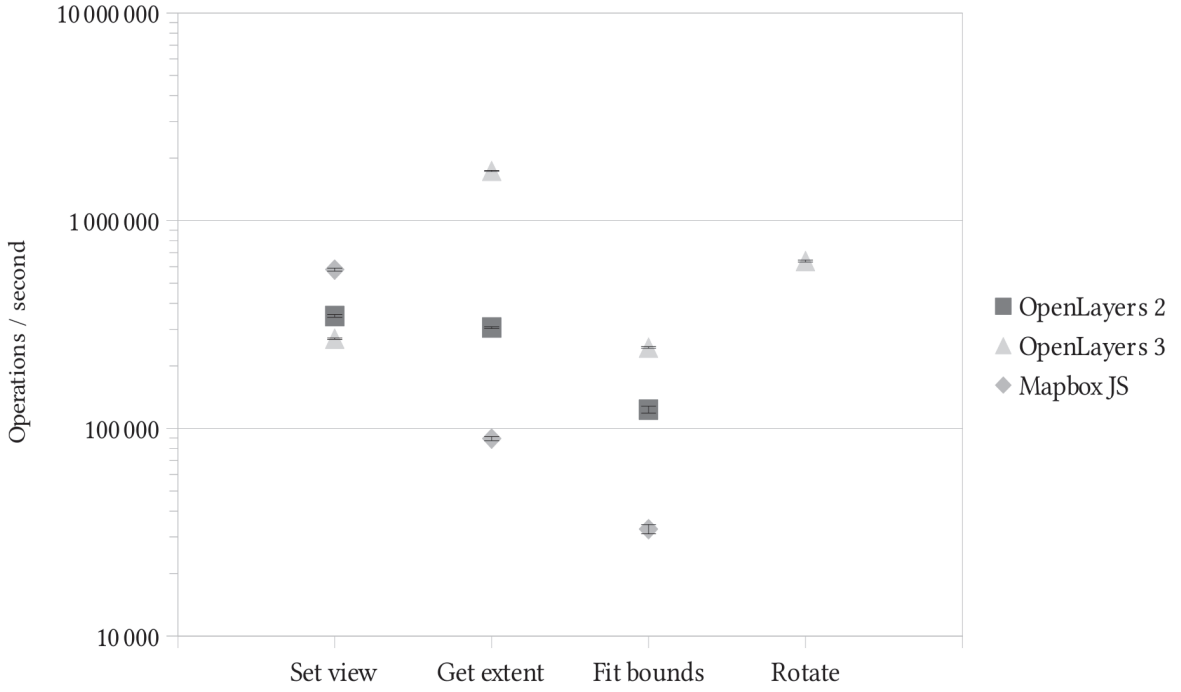

Figure 33: Benchmark diagram of view controls in OpenLayers 2, OpenLayers 3, and Mapbox JS.

4.6.3 Interactions

Interactions can be considered as a set of GUI methods the user can communicate with the WebGIS application. Basic interactions, like panning, or zooming, are not compared by the study, as all of the decent web mapping libraries support them. Only more advanced interactions, like drawing features, or measuring distances are in the scope of this study.

The study discusses interactions in five categories. Drawing interaction is the capability of drawing features dynamically based on user input. Feature selection is the ability of selecting features by clicking, or drawing a bounding box. Libraries should be able to support selecting with irregular polygons. Modifying features is an interaction based on selecting a single feature, and drawing. It should be able to modify the position of vertices, and add additional ones in a minimalistic design.

Snapping is an advanced interaction, which usually can be implemented with a great expense. It has to search for the closest vertex in a set of features, usually all of the features in one or more layers. Snapping is a very popular demand in applications designed for digitizing. Measuring capabilities are the last category. It is based on the drawing interaction, ads should be able to measure the input feature's length or area based on the type of the tool. It also should be able to produce geodesic measurements, at least on a sphere. A full implementation should be able to also measure distances on an ellipsoid.

The support diagram in Figure 34 shows, most of the interactions are easily implementable. However, selecting and snapping tools are not available for all of the libraries. This study is written in an unfortunate time in term of comparing the snapping interaction, as it is now fully implemented in OpenLayers 3, too, and will be shipped in the next stable release, 3.5.0.

Mapbox JS shows limited support for advanced interactions. This is due to the fact, it actually does not support any of the interactions natively. There is a great plugin, Draw, which implements basic drawing interactions, and handlers. Most of the draw related controls in Mapbox JS, like the measuring control, are extensions of Draw, and also depend on it.

	Draw	Select	Modify	Snap	Measure
OpenLayers 2	5	5	3	5	3
OpenLayers 3	5	4	4	NaN	9
Mapbox JS	8	NaN	8	28	1

: Supported
: Deployable
: Not supported

Figure 34: Support diagram of interactions in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, interactions are categorized as regular controls. They can be added only with a few mandatory parameters, but most of them are very scalable, therefore it can handle much more optional arguments, which alter the behavior of the control. The basic parameters are usually the layer object, on which the interaction operates, and the type of the handler. The handler type defines the type of geometry the user can draw on the map canvas. Once the interaction is added to the map, it can be activated and deactivated programmatically, therefore they only need to be added once.

```
var interactionObject = new OpenLayers.Control.DrawFeature(map.layers[1],
    OpenLayers.Handler.Polygon
);
map.addControl(interactionObject);
interactionObject.activate();
```

OpenLayers 3 has a dedicated class for interactions, called interaction. They are not as consequent, as their OpenLayers 2 representatives, but can be parameterized logically. The draw interaction needs a layer source, as it will save drawn features in it. It also requires a type string, which can be Point, LineString, Polygon, or their multipart variants.

```
var interactionObject = new ol.interaction.Draw({
    source: map.getLayers().getArray()[1].getSource(),
    type: 'Polygon'
});
map.addInteraction(interactionObject);
```

The selecting interaction needs an array of layers the interaction can select features from. For the modification interaction, a collection of features needs to be provided. As the getFeatures method of source objects return features in an array, they need to be wrapped in a collection object prior to adding them to the interaction. This way, an arbitrary set of features can be defined as targets of modifications.

```
var layerObject = map.getLayers().getArray()[1];
var interactionObject = new ol.interaction.Modify({
    features: new ol.Collection(layerObject.getSource().getFeatures())
});
map.addInteraction(interactionObject);
```

The library does not have a dedicated interaction for measurements, but rather offers official examples for implementing one. The official example is beautiful, but extremely verbose, therefore a minimalistic custom implementation is presented in the study. If an implementation requires to show the measured geometries on the map, the code needs to be extended by adding a temporary layer to the map. The temporary layer's source object then accepts the measured geometries, and keeps them.

```
var interactionObject = new ol.interaction.Draw({
    source: null,
    type: 'LineString'
});
interactionObject.on('drawend', function(evt) {
    var length = evt.feature.getGeometry().getLength();
    console.log(length/1000 + ' km');
});
map.addInteraction(interactionObject);
```

Some of the compared interactions can be implemented in Mapbox JS via the Draw plugin. Draw extends the control object, and includes both drawing, and modifying capabilities in a simple form. It offers a complete toolbar, which can be used for digitizing. Snapping and measuring interactions can be implemented via extensions of Draw. The measuring interaction is simple, and straightforward, but snapping is more complicated.

```
interactionObject = new L.Control.Draw({
    edit: {
        featureGroup: map.featureLayer
    }
}).addTo(map);
```

```
map.on('draw:created', function(eventObject) {
    map.featureLayer.addLayer(eventObject.layer);
});
```

For snapping, two additional plugins are required: Snap, and GeometryUtil. Furthermore, if one takes the time to analyze the hosted example for snapping, it becomes clear soon enough, that the snapping interaction has some caveats, which require serious considerations prior to implementing in a production environment.

The benchmarking diagram in Figure 35 indicates, there isn't much relevant difference in the performance of adding interactions in the two OpenLayers libraries. Except for snap, the interactions need to register event listeners on the map, initialize their inner structure, and add their target datasets. Their construction performance is, therefore low. The modifying interaction in OpenLayers 2 is faster, but not in a significant way, concerning this is a one time expense. Snapping tool can be initialized very quickly, as it only acts as an agent during drawing or modifying, therefore it only does expensive calculations, when one of those interactions are invoked.

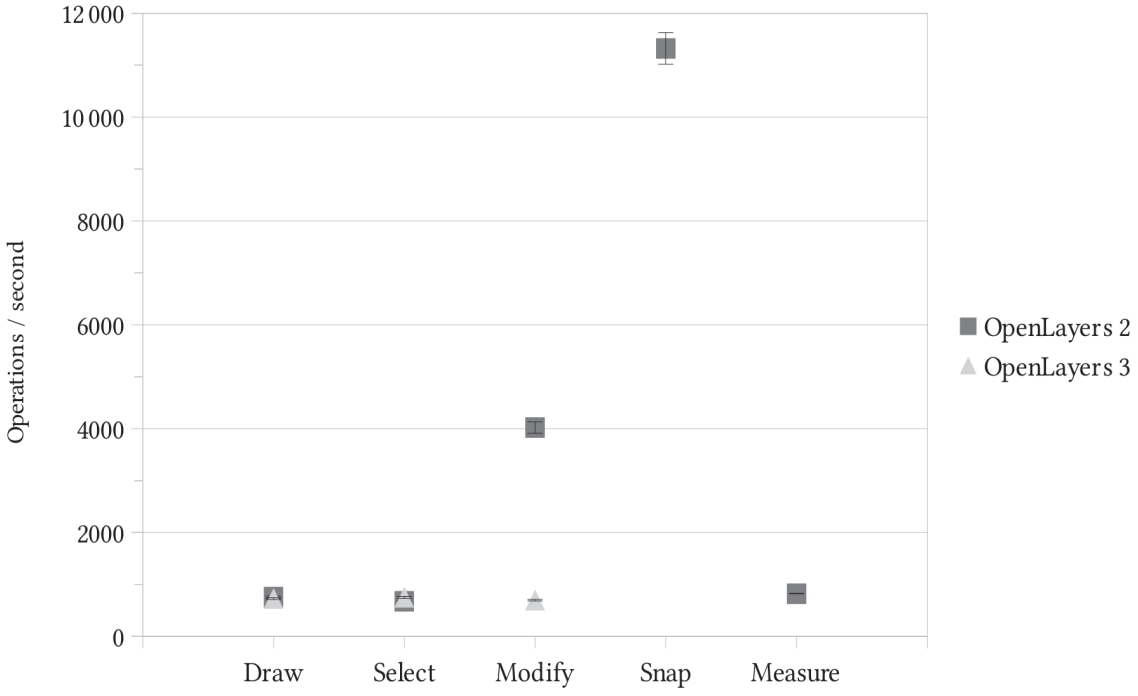

Figure 35: Benchmark diagram of interactions in OpenLayers 2, and OpenLayers 3.

4.6.4 Miscellaneous

The rest of the studied map controls are distinct, but important features, which could not be categorized in a definite category, therefore the term miscellaneous was chosen. The category contains three important features: popups, layer switcher, and toolbars. All of them have a special role in a WebGIS application, and should be supported by the libraries.

Popups are geographically bound overlays. There are various use cases of popups, mostly related to identification and supplying additional information about a point or range of interest. A WebGIS application could also use them for identifying features, while another valid use case would be requesting users to fill the attribute table of newly digitized features.

Layer switcher is a special control. It cannot replace a well-structured layer tree, but can add a lightweight solution to switch between layers, when a layer tree is not necessary. It also offers thoroughly tested, and optimized methods to handle layers in a library, which can be used to build a layer tree control.

Toolbars analyses the capability of a library to offer methods, which can be used for creating custom controls in the style of the library. A WebGIS application can store its set of controls outside the map, especially if dozens of controls needed. However, if just a few controls are satisfactory, implementing them in the style of the library should be easier, and might grant a better user experience.

As it can be seen in Figure 36, all of the libraries support these miscellaneous map controls on some level. OpenLayers 3 does not have a native support for layer switcher, but due to the popular demand, a plugin were created for it, and can be easily included, and implemented. Creating custom tools can seem to be lengthy in all of the libraries, but they are much shorter, than defining the buttons for the controls manually, then styling it accordingly.

	Popup	Layer switcher	Toolbar
OpenLayers 2	7	2	9
OpenLayers 3	8	2	17
Mapbox JS	4	4	12

: Supported
: Deployable

Figure 36: Support diagram of miscellaneous controls in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, popups are handled separately from layers. They can be found in the map's popups property. Creating a popup needs at least four arguments to display correctly. The first argument is the name of the popup, as each of them can be conferred with a unique name. The second parameter is the position of the overlay, while the third one is its size. The fourth parameter is the HTML content, and the fifth optional argument is a boolean value, defining whether the popup should include a close button in it. Popups can be added to, and removed from the composition by dedicated methods called from the map object.

The layer switcher is a simple control, which can be added to the map, and afterwards, it automatically updates itself with every change in the layer stack. The layer switcher can be used to change between base layers, and alter the visibility of overlays.

Toolbars can be created in OpenLayers 2 with the panel control. The panel groups several controls, displays them on the map with custom CSS styling for every containing element, and handles them in respect to the other elements. For example, if a control is activated in a panel, every other control of the same type gets deactivated automatically. The most important property of the panel in a WebGIS application, is the allowDepress. It allows users to deactivate controls in place, not only by activating another control. For the contained controls to show up in the map, at least the button class has to be styled in CSS in the scope of the panel class.

```
var controlObject = new OpenLayers.Control.Panel({
    allowDepress: true
});
controlObject.addControls([
    new OpenLayers.Control.Button({
        title: 'My custom control'
    })
]);
```

OpenLayers 3 offers a more scalable method for creating overlays. Popups are created with the overlay constructor, and can be added to the map as overlay objects. The constructor needs two parameters, the DOM element of the popup, and its geographic location. The content is purely described by the DOM element, no additional wrappers are applied on them.

The layer switcher can be accessed with Matt Walker's plugin. It offers a convenient method for constructing and adding it as a control. The layers which are presented in the control must have a title parameter, and optionally a type parameter. If the type parameter is set as base, the control treats the given layer as a base layer. The control's working principle is identical to OpenLayers 2's layer switcher.

To bind a custom control to a button styled as a native OpenLayers 3 control button, the control needs a DOM element defined. The DOM element must be a button wrapped in a div element. The div element's class name must contain the class ol-control, and its position must be defined relative to the map. If these conditions are met, the library takes care of the rest of the styling.

Overlays in Mapbox JS are handled as individual layers. They are added to the overlay pane, and modified with layer commands. To create a popup, one have to construct an empty popup object, set its geographical position, add content, and open it on the map. It can be done with four commands. However, as Leaflet, the basis of Mapbox JS is started out as a lightweight map viewer application, it offers convenience methods for binding popups directly to features.

The layer switcher in Mapox JS is a static control. It has to be filled with a group of base layers and overlays, providing the name for the layers. Layers can be added to, and removed from the control manually, therefore to achieve results, like in the OpenLayers libraries, layer events should be listened to, and appropriate layer switcher methods should be executed on them.

Adding custom controls is very easy, however a few concepts must be understood in order to get proper result. The appearance of the control, similarly to OpenLayers 3, can be defined in the custom control itself. Extentding L.Control can require various parameters based on the complexity of the custom control, however for styling, only the options and the onAdd properties are important. The options parameter contains information about the position of the control. The onAdd property is a function, which has to return a DOM element. For decent styling, an anchor element must be wrapped in a div element, and the div's class name must contain the class leaflet-bar.

```
var customControl = L.Control.extend({
    options: {
        position: 'topright'
    },
    onAdd: function (map) {
        var container = L.DomUtil.create('div', 'leaflet-bar');
        container.title = 'My custom control';
        L.DomUtil.create('a', 'custom-control', container);
        return container;
    }
});
```

The benchmark diagram in Figure 37 presents, OpenLayers 3 offers great performance in creating popups. It does not wrap overlays, or process them in any way, just renders a div element on the map canvas in a given location.

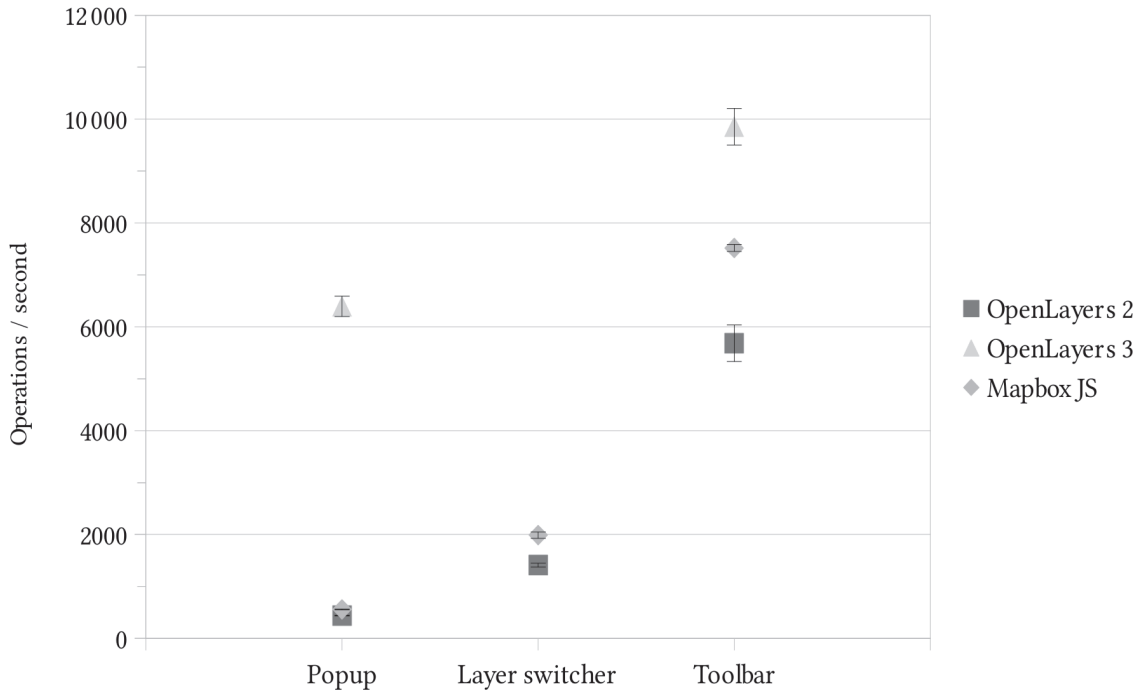

Figure 37: Benchmark diagram of miscellaneous controls in OpenLayers 2, OpenLayers 3, and Mapbox JS.

The other feature with greater differences, is the capability of creating custom tools. OpenLayers 2's low performance is due to the double construction, as it has to set up a panel control, then add a button control to it. Mapbox JS only creates one control, but processes it on the way, based on the control's parameters. OpenLayers 3 on the other side, just grabs the control object with its already styled DOM element, and renders it on the map.

4.7 Composing Controls

After a successful study, the map products usually get published illustrating the research paper. The raw data get styled, often cartographic elements are applied, then the results of a spatial analysis are shown in maps created with cartographic post-production. This process in the field of WebGIS is principally the same, but the methods are different. There are serious technical boundaries of exporting maps in arbitrary resolutions from a WebGIS client. Therefore, the study concentrates on composing controls, which make interactive maps more convenient, and unequivocal for the audience.

As for published maps, not only the exactness of the data is important, but they should have a demanding appeal, and communicate underlying information in an intelligible form. These requirements can be accomplished by proper styling, and using composing controls. Both of them should be supported by a decent library, thus they are considered as the last key concept in choosing a web mapping library for a WebGIS application.

The study splits composing controls into three further subcategories. Vector styles determine the capability of a web mapping library in applying different styles to features, altering only their appeal, not the features themselves. In the subcategory, it is also studied, if a library is capable of rendering labels systematically, based on feature attributes.

In the second part, special thematic layers are discussed. Thematic layers are used to illustrate spatial distribution of an analyzed phenomenon. There are many kind of thematic layers, but the study only compares the most typically used ones, which are likely to be implemented in the web mapping libraries. They are usually achieved by special styling based on the values of a specific attribute.

The last studied class is the set of cartographic elements. Cartographic elements are controls designed to communicate additional information about the map in a clear manner. The study involves the most important elements, which are likely to be used in web compositions, not only in paper maps. These elements do not replace the need for cartographic post-production in order to compose beautiful maps. They can be rather used for helping the user navigating, and understanding the content of the map.

The ability of printing, or exporting maps on arbitrary resolutions, is a contradictory feature not discussed by the study. The traditional way of exporting maps, are via requests to the map server, which renders a composition in a specified resolution, with a specified set of layers. The drawback of this method is, servers only render layer content, any other elements have to be added by the user. The other, canvas based method, is exporting the canvas content to a picture. This can be done by OpenLayers 3, however as the controls are rendered outside the canvas, this method also results in a plain map. Still in these modern days, the most universal method is taking a screen shot from the map region.

4.7.1 Vector Styles

Styling features can be used to make a map product more appealing, or even more exact. Styling can not only used for making the look of the map more original, but it can also emphasize, or weight features, if proper styling is used. For example, in a general road network map, highways should always be more dominant, than inferior roads, but in a hike map, paths should be emphasized over highways.

Custom styles can be applied to single features, however the study only analyzes consistent styling, where an entire layer is styled with a set of rules, or a function. Points, lines, and polygons have different styling properties due to the dimensional differences. Also, the capability of rendering labels has been tested. Labels are used to simplify the interpretation of a map by signing important features with their names, or other attributes.

The support diagram in Figure 38 indicates, basic vector styling is supported by all of the libraries. Mapbox JS offers the shortest methods for styling vectors. Feature groups (i.e. layers) can be styled collectively, however every feature can also be styled separately. In the tested version of the library, loading markers (i.e. points) from a local source needs an access key, therefore they were loaded manually.

	Point	Line	Polygon	Labels
OpenLayers 2	20	19	19	20
OpenLayers 3	24	22	21	26
Mapbox JS	22	12	12	24

: Supported

Figure 38: Support diagram of vector styles in OpenLayers 2, OpenLayers 3, and Mapbox JS.

In OpenLayers 2, styles can be defined either on layer construction, or by setting the layer's style or styleMap property later. The set of valid style options are defined by the Symbolizer class. There are different symbolizers for points, lines, polygons, and labels. The full list of accepted parameters can be found in the API documentation.

There are two ways to define styles in the library. The style parameter of the layer accepts an object with symbolizers, and values. For a minimal custom configuration a stroke color, and a fill color should be set for a layer consisting of geometries (i.e. not labels). For point layers, an additional radius parameter should be added, as points are symbolized with circles, if not with an external graphic. For dashing a line, the library offers minimal support. The library supports five predefined dash styles, namely dot, dash, dashdot, longdash, and longdashdot.

```
style: {
    fillColor: '#ffffff',
    strokeColor: '#000000',
    pointRadius: 5
}
```

The other way is using a style map. Style maps are constructed objects, accepting a single style, or multiple style objects as key-value pairs. The key in this case should be the name of one or more rendering intents. Rendering intents are default styles for different cases (e.g. default, selected, temporary). The other use case of a style map is its ability to evaluate feature attributes. The attributes can be added as template strings, with the template style used in the XYZ sources. On runtime, they get substituted appropriately.

```
styleMap: new OpenLayers.StyleMap(
    new OpenLayers.Style({
        label: "${NAME}"
    })
)
```

In OpenLayers 3, the available styling options are presented in the Style class. There are individual style objects for circles, strokes, fills, and text among others. Every valid parameter is shown under the appropriate style object in the API documentation.

Styles can be defined with the style parameter at layer construction, or with the vector layer's setStyle method later. There are two ways to define styles. Styles can be supplied directly as nested style objects to the layer, or a style function can be defined. A style function is a simple function with a feature, and a resolution as input parameters. This way different styles can be defined for different cases based on the map resolution, or the feature attributes. The function has to return an array of style objects.

```
style: function(feature, resolution) {
    return [new ol.style.Style({
        text: new ol.style.Text({
            text: feature.get('NAME'),
            fill: new ol.style.Fill({
                color: '#000000'
            })
        })
    })];
}
```

In Mapbox JS, styles can be defined only with objects containing style properties, and values. The valid options are defined by Leaflet's Path class's options section in the API documentation. However, this method can only be used to style circle markers, lines, and polygons. For displaying labels, a different approach is needed.

Points are naturally rendered as markers by the library. Markers have a special, styling property, named icon, which can be used to bind an icon or any other DOM element to the features. This method differs from regular styling, as it does not use the setStyle method. In the GeoJSON parser, a pointToLayer function can be defined, which casts the input features in the defined form. The function can access the feature objects, therefore it can be used to render them as div elements, containing an attribute value. The main caveat of this function, is for displaying a set of features along with their labels requires the features to be duplicated.

```
pointToLayer: function(feature, latlng) {
    return L.marker(latlng, {
        icon: L.divIcon({
            html: feature.properties.NAME,
            iconSize: [100, 20],
            className: 'my-custom-label-class'
        })
    })
}
```

Benchmarking was not done for this subcategory, due to the nature of styling. Styling can be done on layer construction, and later with presumably different performance. The different styling methods, like style objects, style maps, and style functions would also have made benchmarking more difficult, as the correct styling method is case dependent.

4.7.2 Thematic Layers

As mentioned above, thematic maps are used to present spatial distribution of a phenomenon by classifying features based on an attribute. The two most widely used thematic maps are chloropleth maps, and maps with proportional symbols. The two can be used to achieve the same result, but based on different input data. Chloropleth maps can be used to classify polygon features by changing their fill color, while proportional symbol maps can be used to classify point features by changing the size of the symbols.

Two more advanced thematic map types are compared by the study. Chart maps can be done easily by some of the most popular desktop GIS environments, however in WebGIS implementing such feature is far from trivial. Most of the web mapping libraries work with standard image formats for displaying raster data, and SVG format for displaying vector data. Adding support for creating SVG charts in a web mapping library can be a tremendous amount of work. However, using canvas, and with some core support for custom canvas layers can make the work foreseeable.

The mass uptake of using heat maps for showing spatial distribution came recently. The ability of generating heat maps has a long tradition in both desktop GIS environments, and map servers. Live rendering support for heat maps are scarce in desktop GIS applications, but in web mapping, this was an early demand, and all the decent web mapping libraries support it in some form.

As the support diagram in Figure 39 shows, chloropleth, and proportional symbol maps are supported by all of the libraries. This is due to their basic styling requirements. If a library can style features based on their attributes, these thematic layers are implementable. Diagram overlays are only deployable in OpenLayers 3, as it is completely canvas based, and offers a designated source type for custom canvas layers. Heat maps are supported by the libraries, but only OpenLayers 3 offers native support. In OpenLayers 2, there is a custom class created by one of the core developers, but it never made it to a release. There are several heat map plugins for Mapbox JS. The study uses the one released by the author of Leaflet.

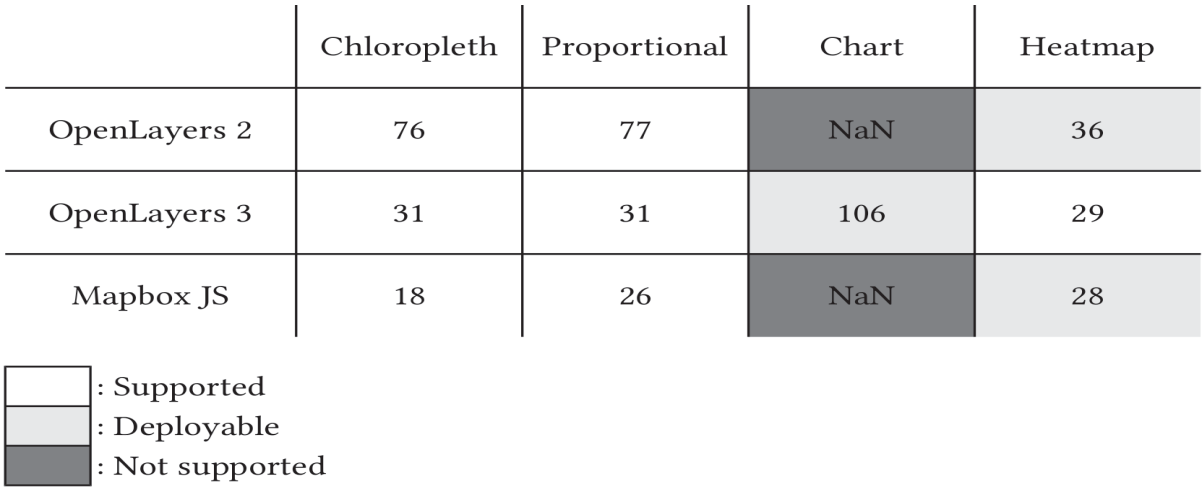

Figure 39: Support diagram of thematic layers in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Chloropleth and proportional symbol effects can be achieved in OpenLayers 2 via special style maps. Style maps can contain an array of rules. Rules consist of a filter, and symbolizer options for the case, the feature gets evaluated as true. Every symbolizer set outside the array of rules get applied regardless of the evaluation's result. For a complete classification, the set of rules must cover every case, setting a default case outside of the rules array does not work.

```
new OpenLayers.Rule({
    filter: new OpenLayers.Filter.Comparison({
        type: '>',
        property: 'pop_est',
        value: 112094336
    }),
    symbolizer: {
        fillColor: '#bd0026'
    }
})
```

For creating heat maps, a special renderer has been developed. The renderer takes a vector layer as an input, and renders it on a canvas element. To generate heat maps, the value Heatmap has to be defined in the renderers property of the layer, then a style map has to be applied.

The style map has to contain the radius of a single heat point, and an attribute column containing intensity values. The intensity values must be between 0 and 1, therefore the attributes need to be normalized, if they scale in another range. For the transformation, the style map offers a context property, in which an arbitrary attribute can be calculated from existing ones.

```
for (var i=0;i<map.layers[0].features.length;i++) {
    var properties = map.layers[0].features[i].attributes;
    max = max < properties['POP_MAX'] ? properties['POP_MAX'] : max;
}
new OpenLayers.Style({
    pointRadius: 20,
    weight: '${weight}'
    }, {
    context: {
        weight: function(feature) {
            return feature.attributes['POP_MAX']/max;
        }
    }
})
```

In OpenLayers 3, style functions can be used to make chloropleth and proportional symbol maps. In a style function, feature objects can be accessed, therefore proper styling can be achieved based on a feature attribute. For classifying attributes, nested conditional operators can be used as a simple solution.

```
var prop = feature.get('pop_est');
var color = prop > 112094336 ? '#bd0026' :
            prop > 84147425 ? '#f03b20' :
            prop > 56200515 ? '#fd8d3c' :
            prop > 28253604 ? '#fecc5c' : '#ffffb2';
```

Creating diagram overlays are not trivial, but they are possible to implement with the ImageCanvas source. This source type can use a custom function, which gets a set of useful information about the map, and the current view, and expected to return a canvas object. The example function draws pie charts based on multiple attributes. According to A. Santiago (2015), the author of the original code the example is adapted from, the main function calculates the difference of the map, and the canvas extent, then draws every element separately wedge by wedge, keeping track of the already drawn arcs.

The example extends this core functionality by wrapping it in a custom layer object. The custom layer needs a vector source specified as an input with point or singlepart polygon features in it. Instead of drawing pie diagrams on random locations, the function reads the coordinates of the points, or the label point coordinates of the polygons, and render a diagram on them with the values of the specified attribute names. For this example, the map object needs to be exposed. For using it in a production environment, the code must be extended along further considerations.

For generating heat maps with the library, it offers a dedicated layer type. The heat map layer accepts a vector source, and an attribute name, containing intensity values among other parameters. The intensity values must be between 0 and 1 similarly to OpenLayers 2. The main difference is, no styling or context can be used, therefore if the input data does not have an intensity attribute, it needs to be calculated for every feature.

```
for (var i=0;i<source.getFeatures().length;i++) {
    var prop = source.getFeatures()[i].get('POP_MAX');
    source.getFeatures()[i].setProperties({weight: prop/max});
}
```

Creating chloropleth and proportional symbol maps in Mapbox JS needs another approach, than in OpenLayers. As every feature need to be styled differently based on a classification, the correct symbolizer has to be applied to every feature separately. As Mapbox JS does not offer any convenience method for styling based on an attribute, the layer's eachLayer function has to be called. In the function the correct symbolizer can be evaluated with nested conditional operators, like in OpenLayers 3.

Creating heat maps with the plugin Leaflet.heat, requires some preprocessing. The plugin accepts an array of features. Every array member has to be a further array with a maximum of three elements. The first element is the latitude of the point, the second is the longitude, while the third is the intensity, and it is optional. The drawback of the plugin is, this array needs to be created manually. However it also has a significant advantage. The heat map layer constructor accepts a max property, which is the maximum intensity of the data series, therefore the data does not have to be in a normalized form.

```
geojsonLayer.eachLayer(function(layer) {
    var latlng = layer.getLatLng();
    var prop = layer.feature.properties;
    latLngs.push([latlng.lat, latlng.lng, prop['POP_MAX']]);
});
L.heatLayer(latLngs, {max: max}).addTo(map);
```

Benchmarking was also not done for this subcategory, as only chloropleth, and proportional symbol layers are supported natively by more than one library. As these layers are created by appropriate styling methods, the same rule applies, as for the previous subcategory, styling vectors.

4.7.3 Cartographic Elements

Cartographic elements are often used to give a final polish to a paper map. They can make a map more understandable, or can ease the navigation on it. In the field of web mapping, these elements are equally important, but the emphasis is on usability over appeal. The set of cartographic elements a web mapping library should support is much more narrow, than the set used by paper maps.

Contrary to paper maps, interactive maps can only use up the container element to draw maps, and related objects. Obviously, cartographic elements can be placed outside the map view, but if not handled carefully, it can easily break the integrity of the map, resulting in a worse user experience. To avoid this, cartographic elements should be constrained to the minimum.

The study compares the libraries' support for the most essential elements. These are the legend, the scale bar, the north arrow, the graticule, the overview map, and the attribution. The map legend should be short, and informative. In a WebGIS application, the map legend should be the part of the layer tree, but in an interactive web map, it should be provided as the part of the map. It should not contain too much information, distracting users from the map, but just enough to make interpreting the map easier.

Scale bars, contrary to absolute scale values, are calculated from geodesic distances on the map. In most of the projections, scale depends on the latitude values of the map extent. On larger scales, the range of the latitudes are wide enough to show inaccurate values. Furthermore, the calculation methods can also influence accuracy. While OpenLayers 3 calculates the scale on the center of the map view, giving a smaller margin of error in both directions, Mapbox JS calculates it in the bottom, giving an increasing error margin to the top.

North arrow is a contradictory element. There were several debates about its necessity in all maps. As C. D. Morais (2011) argues, North arrow is not necessary in every case, but its abandonment should be based on considerations. If the map is oriented to North, and this orientation is obvious, or the intended audience is familiar with the orientation of the map, it can be abandoned. However, for any other case, it should be supported by a library capable of rotating the map (in this case OpenLayers 3). Other libraries should also support North arrows, as the audience will probably not be familiar with the operation principles of the used web mapping library.

A graticule can effectively support orientation, or measurements on a map. When using projections, in which meridians converge to the pole, a graticule can be especially useful for reading absolute directions from specific points. Overview maps should also be supported, as it can also support orientation, especially on maps with a small, restricted extent. Attribution is the last tested feature, which helps overcome legal term issues, and can easily process and display attribution data linked to one or more layers.

The support diagram in Figure 40 indicates, the vast majority of the studied features are implementable with all of the libraries. Some of them need a third party extension. There are two exciting data in OpenLayers 3. The implementation of a North arrow requires exactly zero lines of extra code, as it is natively supported by the rotation control. Following the principle mentioned above, it is hidden, if the map's orientation is North. However, with every change, the arrow symbol of the control rotates to North. The other unfortunate fact is, the overview map is broken. It is constructed with a fixed Web Mercator projection, therefore it renders a blank view in other projections.

In OpenLayers 2, the legend control needs a manual extension. It is not supported natively, but can be implemented easily. The example uses a custom legend control, which requests a WMS legend image. It has a draw function, which is responsible for drawing the legend element.

The control checks for every layer in the stack, if it has a showLegend property set to true. It accesses the URL of the WMS layer, for GeoServer compatibility, checks if the URL is already parameterized, then it sends a legend request to the server. Finally, it wraps the response image in a DOM element, and displays the result. The style of the legend is defined with CSS rules, by default, in the ol-legend class.

	Legend	Scale bar	North arrow	Graticule	Overview	Attribution
OpenLayers 2	33	2	15	2	2	2
OpenLayers 3	50	2	0	2	NaN	2
Mapbox JS	2	1	12	1	6	1

: Supported
: Deployable
: Broken

Figure 40: Support diagram of cartographic elements in OpenLayers 2, OpenLayers 3, and Mapbox JS.

Scale bars can be created easily by the application, only one parameter should be set. If the control is constructed with the geodesic parameter set to true, the scale bar displays more accurate values. The North arrow, similarly to the legend control requires extending the control class. It also only needs a draw function, which wraps an image on the provided path in a DOM element, and renders it on the map.

```
class: 'ol-northarrow',
imagePath: '../../../res/image/narrow.png',
draw: function() {
    this.div = document.createElement('div');
    var narrowImg = document.createElement('img');
    narrowImg.src = this.imagePath;
    this.div.appendChild(narrowImg);
    this.div.className = this.class + ' olControlNoSelect';
    return this.div;
}
```

The rest of the elements can be applied on the map with simple control constructors. The attribution control does not require any extra parameter, but the layers must contain attribution information for the control to display information.

In OpenLayers 3, the legend control also needs to be manually implemented. The custom implementation in the example follows the pattern described above. However, in the library, the custom control cannot access the map at construction, therefore the requests have to be called in the setMap function. Also, for updating capability, the control draws items in a separate function. For further optimization, the iteration also should be wrapped in a dedicated function.

The rest of the controls can be invoked by regular control constructors. For creating a North arrow, similar to the custom implementations in the rest of the libraries, a custom control has been created. The only addition to the code is, the control keeps track of the map's rotation, and rotates the North arrow accordingly.

In Mapbox JS, legend control is natively supported. A default legend control is added to the map at construction. The object has an addLegend method, by which HTML or DOM elements describing a single legend can be added.

For creating a North arrow, an extension is needed, similarly to the other libraries. The drawing function must be included in the onAdd function, which is expected to return a DOM element. Creating a graticule can be done with the Leaflet.Graticule plugin. The constructor creates a feature group, thus he graticule needs to be treated as a layer object.

Overview maps can be added with the Leaflet-MiniMap plugin. The constructor needs a set of layer objects, different than the layers already added. If the overview layer needs to be the same as the map layer, it has to be reconstructed. The rest of the controls are natively supported, and can be added as regular controls to the map with the same considerations as in the OpenLayers libraries.

As it can be seen in Figure 41, the benchmarking results show similar performance in most of the features. The North arrow was not benchmarked, as it is only supported natively by OpenLayers 3, and as it is a default control, the result would have been effectively infinity. The margin of error in OpenLayers 3 is very high, but it does not influence the final result. Constructing attribution control in OpenLayers 2 is significantly faster, than in the other libraries, but the difference in reality is not observable by the user.

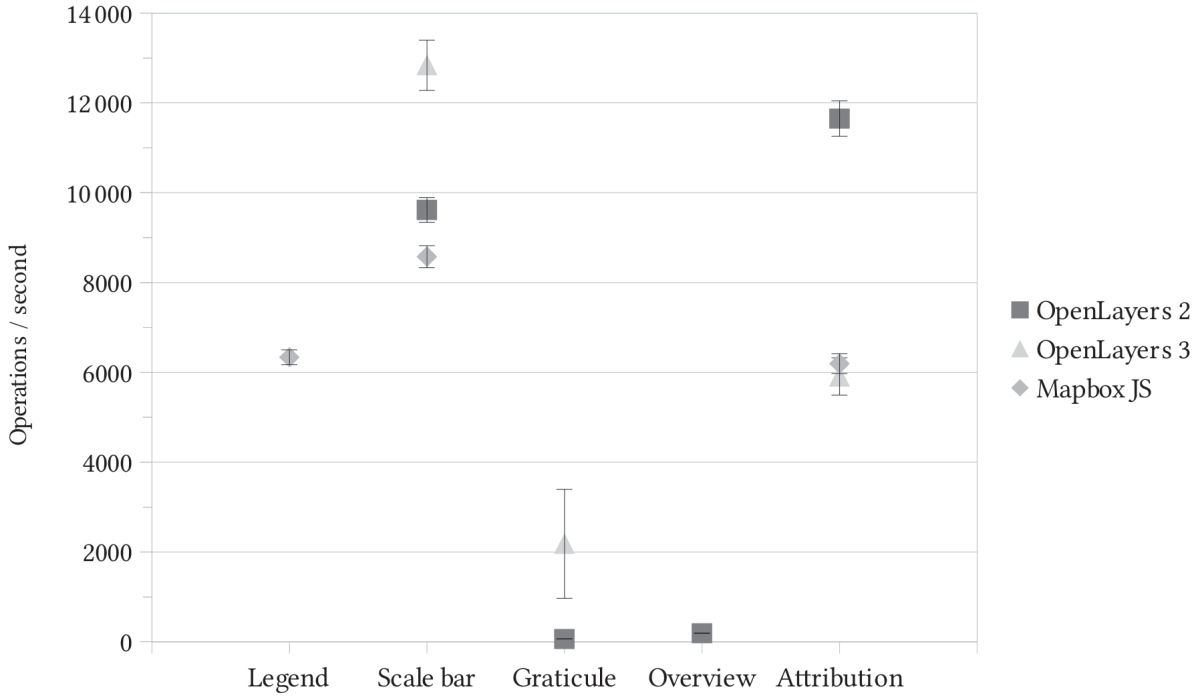

Figure 41: Benchmark diagram of cartographic elements in OpenLayers 2, OpenLayers 3, and Mapbox JS.

V. Conclusions

5.1 Summary

Summing up the analyses shows, all of the compared libraries are capable of being the basis of a WebGIS application. They have their typical strengths and weaknesses, and require a well considered setup in order to work effectively. The heavy lifting should always be done by the server, however in a healthy, and balanced setup, the client side should be able to do lesser computational operations beside visualization related tasks, and feature management.

The final statistics are calculated from the support diagrams, and tend to be as simple as possible. Due to this simplicity, the results reflect the reality. The first analysis is a summed up form of the libraries' support for various features. It summarizes the the number of natively supported, deployable, broken, and not supported features. The second analysis studies the average number of lines required to deploy a set of features with a library. It also contains the total average lines needed in order to deploy every feature appearing in the study.

The summarized support diagram in Figure 42 shows, the largest proportion of the studied features are supported, or deployable by all of the libraries. OpenLayers 2 is the most stable library, as it does not have any broken feature, which cannot be implemented in some way. The percentage of the not supported features are only about 6%, while most of the analyzed features are natively supported. Only about 11.5% of the total number of features need third party extensions.

	Supported	Deployable	Broken	Not supported	Overall
OpenLayers 2	72	10	0	5	87
OpenLayers 3	62	15	2	8	87
Mapbox JS	38	30	1	18	87

Figure 42: Summarized support diagram in OpenLayers 2, OpenLayers 3, and Mapbox JS.

These are great results, however the library is out of development. The not supported features are related to the cutting-edge technologies, including the newest formats, map rotation, and drawing canvas based custom layers. Due to the vast amount of natively supported features, the library is quite big. It takes 925.9 KB of disk space without plugins, and 932.9 KB with the heat map plugin, and the same amount of additional network traffic for every request. This is the largest of the compared libraries.

OpenLayers 3 has the second best results. There are only two broken features, which cannot be implemented in a WebGIS application, the OSM input support, and the overview map support. They are easily negligible, and will be fixed in the future.

Only about 9% of the total number of studied features are not supported by the library. These features include the CSV format input, some of the metadata reading capabilities, and mostly the lack in spatial operations. The support for spatial operations are important, and call for an extension. Probably, the library will grant core support for them in the future, but until then, third party topological libraries can easily overcome this problem, with some overhead.

About 17% of the features need third party extensions. As OpenLayers 3 is a compressed library, compiled with Closure Compiler, extending it is not as simple, as extending OpenLayers 2. However, writing additional features needed for a WebGIS client is possible, and with the rapid development of the library, OpenLayers 3 can be the most robust, and versatile basis of a WebGIS application.

The size of the library is 477.2 KB without plugins, and 493.9 KB with the layer switcher extension. The base code is much larger, than OpenLayers 2's, but with highly optimized compression methods, the compressed version is gradually smaller, therefore every request generates about 50% less traffic just in the term of loading the core web mapping library.

Mapbox JS shows a poor native support rate for a WebGIS application, however the key features are supported, or can be implemented by third party extensions. There is only one broken feature, the UTFGrid support. This feature, however, can be accessed with Mapbox JS 2.1.6. via a Mapbox account and an ID to a TileJSON data source.

About 20% of the compared features are not supported by the library. There are some key features among them, like the WFS support, the metadata reading capability, the spatial operations, or the rotation control. Some of these features, however, can be neglected in most of the causal WebGIS applications, while others can be implemented with other libraries, like Turf, or D3.

The rate of the supported, and deployable features are nearly equivalent, about 43.5%, and 34.5% respectively. As the library has a small, and stable core, and is plugin based, most of the deployable features are available via various third party plugins. Due to the philosophy behind Leaflet, the basis of Mapbox JS, the core library is only 278.6 KB in size. The extensions needed for deploying most of the studied features takes more than the size of the core library in disk place, extending the library's total size to 474.4 KB.

Figure 43 represents the average number of lines of code required to completely implement a set of features. The values were calculated by summing up the lines of code needed for deploy every supported, or deployable feature in a category, and dividing the result by the number of supported, or deployable features. The libraries did not get a penalty for not supporting a feature, and the partial results were not weighted in any way. This results in a set of undistorted values, representing the verbosity of a library.

The rationale behind not giving penalties or any kind of weighting is, if an essential feature is not supported by a library, the library will not be chosen as the basis of the client application. Considering the amount of code, and the coding style needed to use a web mapping library should not be the first consideration in any case.

	Data I/O	Layer Management	Feature Management	Map Controls	Composing Controls	Overall
OpenLayers 2	13.11	2.13	8.46	8.30	30.61	12.65
OpenLayers 3	15.32	2.32	14.32	8.94	27.90	13.76
Mapbox JS	8.00	2.42	11.14	7.73	15.11	8.88

Figure 43: Average verbosity in OpenLayers 2, OpenLayers 3, and Mapbox JS.

OpenLayers 2 has a total average verbosity of 12.65 lines/feature. It has a fairly convenient coding style, most of the features can be implemented with less then 10 lines of code. For establishing connection to a server, or any other data source, it offers separate layer classes for every different source. This behavior minimizes the amount of code needed, every layer construction's length roughly equals to the number of parameters needed.

For layer, and feature management, the library offers numerous convenience methods. A few features in feature management, like creating more complex geometries, or nesting filters have higher space requirements. Defining map controls also do not take a high amount of coding, except for changing the map's projection.

Implementing composing controls require the highest average amount of coding of the compared libraries. This phenomenon is caused by two significant outliers, the chloropleth, and the proportional symbol maps. These thematic layers need as many nested rules, as many categories they have, and only one rule takes as much as 10 lines of code.

The outliers are due to custom implementations, and nested classes, which can be very extensive. As the possibility of nesting in filters, and styling requires object constructors of the same length, it can gradually increase the verbosity of the final code.

OpenLayers 3 has the largest average verbosity with 13.76 lines/feature. The difference between the average verbosity of OpenLayers 2, and OpenLayers 3 is negligible. The minor increment in OpenLayers 3 is due to the better structured core library, which is more scalable, and therefore needs more parameters to define a feature.

The increased amount of coding required for implementing data exchange is caused by the separation of layer, and source objects. There are a few layer types, and various source classes. In every layer, a source must be also constructed, and correctly parameterized. In terms of layer management, the library offers the best verbosity rate, due to the library's convenience methods to manipulate layer properties.

In order to implement feature management options, the library needs about 70% more lines of code, than OpenLayers 2. This is caused by the lack of support in filters. The need for custom implementations generate serious outliers in both OpenLayers 3, and Mapbox JS. Map controls can be implemented with nearly the same verbosity, as in OpenLayers 2, while in composing controls, the lengthy implementation needed for creating a diagram overlay is balanced by the succinct nature of style functions.

Mapbox JS offers least of the studied features, but it also the most concise of the compared libraries. It has an average verbosity of 8.88 lines/feature, and it is due to the philosophy behind Leaflet, the core library behind Mapbox JS. Leaflet is built with a main purpose to be a lightweight, mobile-friendly web mapping library, which can be used as simply as possible.

The only outliers in the statistical data, are the filtering implementations. As the library only supports layer-wise filters, and feature filters need to be manually implemented, those extensions are almost as lengthy, as in OpenLayers 3. The snapping control is also more verbose, than in OpenLayers 2, but it is due to the implementation style of the third party plugin, Leaflet.Snap.

5.2 Key Findings

For an inclusive, and comprehensive WebGIS client application, OpenLayers 2 is still the most adequate web mapping library. It offers a stable core, with an extensive support for features needed for a well-optimized application. It offers a moderate performance, and needs some considerations before deploying, but great applications can be built on it with minimal additional coding.

OpenLayers 3's main strength lies in the WebGL renderer, which offers 3D capabilities with a Cesium JS integration, and a great performance. The rendering speed is also improved by drawing every element of the layer stack on a single canvas. It is a thoughtful library, with a well-considered, and logical structure.

As OpenLayers 2 gets increasingly outdated with the advancements in web technologies, OpenLayers 3 will surpass OpenLayers 2 in the near future. It does not have such a comprehensive set of features, like its predecessor, but if an existing application can neglect client side spatial operations, or can replace it with a topological library, the migration should be advocated. For new projects, OpenLayers 3 should always precede OpenLayers 2 with the same considerations.

Mapbox JS is also capable of being the basis of a great WebGIS application, however the application's configuration must met some criteria. The library is lightweight, fast, and easy to implement with, but it lacks support for some key features. In a WebGIS application built with Mapbox JS, the server must execute all of the spatial operations. Furthermore, the library's only completely supported vector data exchange format is GeoJSON, therefore the server side processing application must be able the process data received in that particular format. PostGIS for example is a great GIS application for this task, as it can be seen in CartoDB.

The other important issue to consider, is Mapbox JS's layer handling. It can easily support one vector layer, but if multiple, separate layers are needed, a custom solution must be implemented in order to handle the data correctly. Raster layers on the other hand are supported without caveats, therefore the library can be used for a digitizing application, with some considerations made about drawing, and snapping interactions.

Finally, Mapbox JS is built upon Leaflet. It extends the core Leaflet library with some very useful features, but if they are negligible in the client application, Leaflet should be used instead. There are no differences between the coding style, and the structure of the two libraries, but as Mapbox JS is a wrapper library, the core Leaflet library is smaller in size.

5.3 Limitations

The most important fact to understand in order to understand the limitations of the study is, benchmarking is not unit testing. The benchmarks comparing the performance of the libraries are done in a single set of circumstances for every benchmarked feature. The circumstances are the same for all of the libraries, therefore the results can be used for illustrating the proportional differences between the libraries' performance in various cases.

To produce statistically significant benchmarking results, unit tests are needed in order to find out the complexity of a function. Without knowing the complexity, the results can be misleading. For example, if function A has a performance of 100 operations/second, while function B achieves the same result on a sample dataset on a frequency of 200 operations/second, function B has better performance. However, if function A is linear in terms of complexity, while B is quadratic, the functions' rate of performance changes with the size of the input data. With double sized data, the functions finish at the same time, while with even larger datasets, function A becomes faster.

The study completely understands the limitations behind benchmarking without knowing the complexity of a function, therefore the results are not evaluated in the summary. The benchmarked performance data still can be used to broadly illustrate the performance differences between the libraries, but in order to produce significant results, proper unit tests have to be made. Unfortunately, doing complete unit tests is out of the scope of this study in terms of both length, and time.

Another important limitation is the length of custom implementations. These are minimalistic extensions in one particular coding style. In a production environment, these code snippets should be optimized, and extended with at least additional conditional tests. Due to their lack in optimization, these implementations act as outliers in the final statistics, therefore they should be handled with care. The most important consideration is, with proper optimization, these extensions should be more concise, while in a production environment, they will be much more verbose with a great probability.

5.4 Further Research Directions

One of the possible, and rational future direction is to make proper unit tests, and benchmark the performance of the libraries in terms of various features. A benchmarking with unit tests would be as long, as this particular study itself. The unit tests should find out the differences between the libraries' performance, the approximate complexity of various functions, and optionally should make suggestions for faster, and more optimized algorithms, if necessary.

As two of the three compared libraries are in constant development, this comparison is only reckoned a snapshot. While writing these lines, OpenLayers 3.5.0 has came out, and Leaflet's version 1.0 is about to be released in the near future. The requirements for a successful WebGIS application are nearly static, but the libraries change rapidly. Periodically, the study should be revised, and a fresh snapshot should be made, which compares the actually competent web mapping libraries.

Another direction, which is planned to be made, and released in the future, is the comparison of other parts of a WebGIS application. Server side open source web mapping, and GIS softwares are the subjectives of these researches. Sequential studies should compare map servers, tile servers, tile producing, and caching softwares, and GIS softwares capable of being integrated in a WebGIS application. These GIS softwares need to be able to receive commands from command line, and produce an output based on the provided parameters.

These parts should be studied in an organized way, from servers to processing softwares. The study should emphasize WPS comparisons, and alternatives. Finally, a complete WebGIS software should be made based on the results. The final product should be able to grant the capabilities of a complete desktop GIS environment, in which the processing is made, and the data is stored on a server, and the decentralization of the workflow can be effectively achieved.

Finally, the study only covers open source technologies. There are other, proprietary, and closed source web mapping libraries, which are very popular, and can be used in a special WebGIS environment. ArcGIS API for JavaScript or Google Maps API are also great web mapping libraries, which can be compared similarly. Additional considerations must be done, though, as they are licensed differently, and the product built on them must not violate the term of services of these libraries.

Acknowledgements

I would like to give acknowledgement for every person, who affected the outcome of this study in some way. I would like to thank Professor József Tóth, for founding the Institute of Geography in the University of Pécs. Without the Institute, I would hardly been able to make this research. I would also like to thank Dr. Titusz Bugya for all the support, but in the same time, the free hand he gave to me, to shape the research in a way, as I thought it fit.

I would like to thank for my family all the support they gave. I would like to particularly thank my couple, and darling Orsolya Hosszú the extraordinary amount of support, love, and sympathy she gave to me under the time period, the study was written.

Last, but not least, I would like to thank Vladimir Agafonkin, every OpenLayers Contributor, Leaflet Contributor, and Mapbox JS Contributor, for making such great web mapping libraries. They are all spectacular in their way, and gave a wonderful research basis for me. I hope, the development of these web mapping libraries, and optionally new, emerging technologies will continue, they will keep pace with the advancement in web technologies, and will give a solid research, and development basis for a long time.

References

Aasenden, J. L. 2015: Binary data, taking JavaScript to the next step with Smart Mobile Studio. – https://jonlennartaasenden.wordpress.com/2015/02/27/binary-data-taking-javascript-to-the-next-step-with-smart-mobile-studio/ (12 February 2015).

Agafonkin, V. 2015: Leaflet - a JavaScript library for mobile-friendly maps. – http://leafletjs.com/index.html (26 January 2015).

Agafonkin, V. 2015: rotation of map and contents to x degress. · Issue #268 · Leaflet/Leaflet. – https://github.com/Leaflet/Leaflet/issues/268 (10 April 2015).

Battersby, S. E. – Finn, M. P. – Usery, E. L. – Yamamoto, K. H. 2014: Implications of Web Mercator and Its Use in Online Mapping. – Cartographica 49:2. p. 86.

Bostock, M. 2012: Home · mbostock/topojson Wiki. – https://github.com/mbostock/topojson/wiki (22 January 2015).

Carrillo, G. 2012: Web mapping client comparison v.6. – http://geotux.tuxfamily.org/index.php/en/geo-blogs/item/291-comparacion-clientes-web-v6 (7 January 2015).

Cisco 2014: Cisco Global Cloud Index: Forecast and Methodology, 2013–2018. – Cisco Systems, Inc., San Jose, CA. p. 10.

Chesbrough, H. 2010: Open Services Innovation: Rethinking Your Business to Grow and Compete in a New Era. – John Wiley & Sons, San Francisco, CA. p. 2.

Donohue, R. – Wallace, T. – Sack, C. – Roth, R. – Buckingham, T. 2012: Keeping pace with emerging web mapping technologies. – North American Cartographic Information Society (NACIS), Portland. 44 p.

Environmental Systems Research Institute 2015: Independent Report Highlights Esri as Leader in Global GIS Market. – http://www.esri.com/esri-news/releases/15-1qtr/independent-report-highlights-esri-as-leader-in-global-gis-market (5 January 2015).

Gede, M. 2014: Open Source rendszerek a térinformatikai gyakorlatban – Interaktív webtérképek készítése OpenLayers és MapServer használatával. – http://tarsadalominformatika.elte.hu/tananyagok/opensource/lecke4_lap2.html (14 January 2015).

Hossain, M. 2012: benchmark.js: how it works. – http://monsur.hossa.in/2012/12/11/benchmarkjs.html (1 February 2015).

Lemoine, É. 2014: Layer loadstart & loadend events in OpenLayers 3. – https://gis.stackexchange.com/questions/123149/layer-loadstart-loadend-events-in-openlayers-3 (2 March 2015).

Mapbox 2013: UTFGrid. – https://github.com/mapbox/utfgrid-spec/blob/master/1.3/utfgrid.md (21 January 2015).

Morais, C. D. 2011: To North Arrow or Not to North Arrow. – http://www.gislounge.com/to-north-arrow-or-not-to-north-arrow/ (1 May 2015).

Nguyen, T. B. 2014: GeoServer. – http://tarsadalominformatika.elte.hu/tananyagok/geoserver/lecke6_lap2.html (14 January 2015).

Nurseitov, N. – Paulson, M. – Reynolds, R. – Izurieta, C. 2009: Comparison of JSON and XML Data Interchange Formats: A Case Study. – Montana State University, Bozeman. 6 p.

OpenLayers Contributors 2014: OpenLayers 3 – Introduction. – http://openlayers.org/en/v3.4.0/doc/tutorials/introduction.html (26 January 2015).

OpenStreetMap Contributors 2015: Slippy map tilenames. – https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames (20 January 2015).

OpenStreetMap Contributors 2014: Vector tiles. – https://wiki.openstreetmap.org/wiki/Vector_tiles (21 January 2015).

Open Geospatial Consortium 2015: About OGC. – http://www.opengeospatial.org/ogc (10 January 2015).

Open Geospatial Consortium 2010: OGC Web Services Common Standard. – Open Geospatial Consortium, Wayland, MA. p. 12.

Open Geospatial Consortium 2006: OpenGIS® Web Map Server Implementation Specification. – Open Geospatial Consortium, Wayland, MA. p. 33.

Open Geospatial Consortium 2010: OpenGIS® Web Map Tile Service Implementation Standard. – Open Geospatial Consortium, Wayland, MA. pp. 9—19.

Open Geospatial Consortium 2005: Web Feature Service Implementation Specification. – Open Geospatial Consortium, Wayland, MA. p. 100.

Quinn, S. – Dutton, J. A. 2014: The history and importance of web mapping. – https://www.e-education.psu.edu/geog585/node/643 (5 January 2015).

Ramsey, P. 2007: The State of Open Source GIS. – Refractions Research Inc., Victoria, BC. p. 43.

Ramsey, P. 2006: Tms.png (541×562). – http://wiki.osgeo.org/images/e/e7/Tms.png (20 January 2015).

Robins, P. 2010: Using OpenLayers: OL Concepts. – http://www.peterrobins.co.uk/it/olconcepts.html (2 April 2015).

SANTIAGO, A. 2015: The Book of OpenLayers3 - Code samples. – https://acanimal.github.io/thebookofopenlayers3/chapter03_04_imagecanvas.html (28 April 2015).

SCHÜTZE, E. 2007: Current state of technology and potential of Smart Map Browsing in web browsers. – Bremen University of Applied Sciences, Osnabrück. p. 47.

SUNDAY, D. 2012: Area of Triangles and Polygons. – http://geomalgorithms.com/a01-_area.html (23 March 2015).