University of Pécs
Faculty of Sciences
Doctoral School of Earth Sciences

Creating the foundations of a universal client-side Web GIS system

Author:
Gábor Farkas

Advisor:
Dr. Titusz Bugya

Pécs, 2020.

Table of Contents

1. Introduction

Web GIS is a relatively new field in geoinformatics, following the evolution of the Web. Its main purpose is to combine new technologies originating from the development of Internet-based technologies with traditional spatial data visualization and analysis techniques. Results from this field can be considered as porting GIS applications to the Web, although the development and research process behind a Web GIS application is more complex than that. Since the environment behaves differently, both new problems and opportunities arise in the process. By overcoming those problems, new techniques are getting developed. Some of the techniques are Web-specific, while others can be generalized to be useful even in desktop applications.

While the field of Web GIS is still young compared to other, mature fields in geoinformatics, it has its roots dating back to the early days of the Internet. Web mapping, the predecessor of Web GIS, made the first attempts to use the Internet as a medium for propagating spatial information to the masses. The first web mapping application, the Xerox PARC Map Viewer was released in 1993 (Haklay et al., 2008), not long after the Internet was released for civil use (Leiner et al., 1997). Since then, geoinformatics still keeps pace with technological advances of the Internet.

Like every Internet-based application, a Web GIS has a decentralized, server-client architecture (Figure 1). There is a server listening to requests, serving files, executing local programs, and sending responses. Through server-side programs, the server has access to central resources (e.g. databases, files). Connecting to the server, there are an arbitrary number of clients. Clients send requests, update the application's state according to responses, and run code on client machines. According to the amount of logic outsourced to the client, there are thin clients and thick clients. In a thin client architecture most of the logic is running on the server, while in a thick client architecture, the clients' computing resources (e.g. CPU time, memory) are used to save server resources. In a thick client, the amount of logic outsourced to the client can vary.

In the early days of Internet GIS, choosing between thin and thick client architecture needed different considerations. Using native thick clients was unfeasible even after new JavaScript standards made it possible. Companies seeing perspective in using thick clients had to use plug-ins (e.g. ActiveX, Java, Flash), users needed to install on their computers (Peng, 1999). Those solutions slowly faded away, since their environments lost support over time with only Adobe Flash remaining, coming

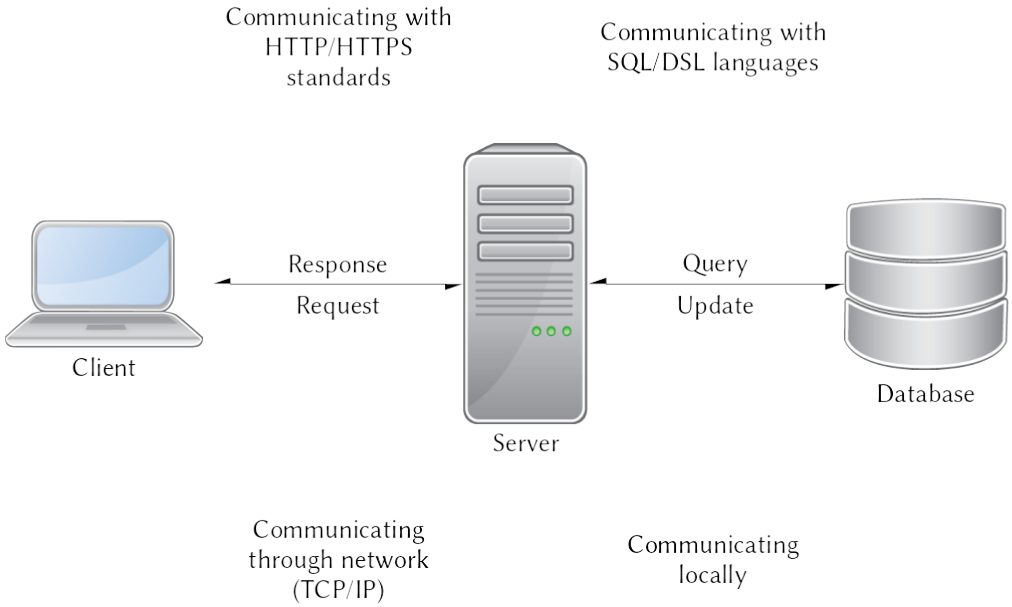


Figure 1: Schematic representation of a basic server-client architecture used by Web applications. HTTP (Hypertext Transfer Protocol) and HTTPS (Hypertext Transfer Protocol Secure) standardize the data format sent between clients and servers. SQL (Structured Query Language) is a language standard for relational databases, while different DSLs (Domain Specific Languages) are used to communicate with different object databases. TCP/IP (Transmission Control Protocol over Internet Protocol) standardizes the communication process between computers through a network.

to the end of its lifetime in 2020 (Adobe Corporate Communications, 2017). On the other hand, server-side applications did not suffer such casualties. Server-side solutions developed in thin client architectures persisted, evolved, and now form a mature basis for spatial servers.

With the advent of Web 2.0 (O'Reilly, 2007), web mapping and Web GIS replaced Internet GIS. Starting with Google Maps in 2005 (Farkas, 2015), a new trend emerged, porting spatial algorithms to the client side using JavaScript. Since JavaScript is a weakly typed language with an interpreter and just in time (JIT) compiler built into browsers, JavaScript programs are slower than compiled code, and more dependent on coding style (Gong et al., 2015). Consequently, it is not always feasible to automatically convert software and libraries written in other languages to JavaScript. Careful optimization and alternative techniques are often needed to achieve optimal performance. This made client-side Web GIS development a new territory with both new problems and opportunities.

Modern web solutions are developed as applications rather than documents, and

browsers are considered as complete environments rather than document viewer tools (Taivalsaari et al., 2011). This trend has created a focus on client-side Web GIS frameworks and libraries (Ramsey, 2007). Since there is demand for decentralized, web-based spatial solutions, there are several companies offering services for spatial data visualization, analysis, and distribution (e.g. CARTO, Mapbox, ArcGIS Online). Albeit most of the services offer a more or less complete GIS environment, their applications are rather server heavy.

There are also groups of developers working on pure client-side solutions for several aspects of a GIS system. Since JavaScript is capable of handling binary data starting from the ECMAScript 2015 specification (Ecma International, 2015), there are libraries for interacting with various popular data exchange formats (e.g. Shapefile, Geopackage, GeoTIFF). There are libraries for visualizing arbitrarily large vector layers with hardware acceleration (e.g. Mapbox GL JS, Tangram, Kepler.gl). There are libraries even for analyzing vector data (e.g. JSTS, Turf). There is no library, however, being capable of abstracting those diverse technologies, and giving developers a framework for creating complete client-side Web GIS applications.

By creating such a basis, truly scalable Web GIS applications can be built easily. Each application can define the client's weight conditionally, executing only expensive calculations on the server. There are many other possibilities with such freedom. For example, creating serverless Web GIS applications, or having only one code base to maintain for both data acquisition, and analysis, based on the type of the device.

Such a system could be best utilized by mobile devices. Evidently, in the past decade, the number of smart mobile devices increased dramatically. These microcomputers are now strong enough to support advanced computational tasks, like 3D video games or office applications. Many applications are built on the most popular operating systems (e.g. Android, iOS), however, those applications must be maintained for multiple platforms. On the other hand, many applications target the Web and are provided as services, available through a modern browser on any device. With such a universal Web GIS application, professionals can be fully platform-independent, as the web browser is the environment, and it can run on any device with a functional browser. The same code base could support fieldwork, analysis, and visualization, offering different capabilities on different devices. Modular development has the advantage of creating software components, which can be used conditionally. For example, such a Web GIS in surveying mode should not load geostatistical modules, saving battery life.

2. Purpose of the Study

The purpose of this study is to find a client-side web mapping library capable enough to be the basis of a Web GIS with some extensions. In order to make an objective decision, current technologies are compared in a competitive analysis. With the most capable library chosen, its weaknesses are outlined with a list of basic GIS features. Those features are chosen based on several pieces of literature. The outlined weaknesses are then reduced to a set of severe problems, which must be mitigated before the library can be considered a solid Web GIS foundation. Finally, solutions are presented to each element of the subset, creating a basic, but functional Web GIS library.

The steps of this study in a list format are the following:

1. Composing a list of basic GIS features based on literature.
2. Selecting capable candidates from current web mapping technologies.
3. Comparing candidates with a competitive analysis.
4. Choosing the most capable candidate for being a basis for extensions and fixes.
5. Outlining the most severe shortcomings of the chosen library.
6. Implementing fixes for those flaws, ending up with a functional Web GIS foundation.

3. Literature Review

Standing on the shoulders of giants, as the saying goes. Maintaining the objectivity of analysis is not a trivial task. This is exceptionally true in the IT segment, where a developer has some proven stacks of technologies with enough experience to use in various solutions. Still, objectivity can be increased by careful interpretation of existing literature. Certainly, there are some areas where subjectivity is not avoidable, or even feasible due to technical limitations. Such an area is outlining basic GIS features. For example, the capability of connecting to a database is considered basic by most experts. However, it is not possible on the client-side without using a server-side component. This study strives for maintaining the highest level of objectivity by carefully synthesizing classical and novel literature. Still, it leaves some space for subjectivity.

For this chapter to make sense, some of the results should be disclosed prematurely. The most capable web mapping library from a GIS perspective has been found out to be OpenLayers. It had two serious flaws preventing it from being a Web GIS basis out of the box. One of them was a partial hardware accelerated vector rendering engine, incapable of rendering lines and polygons. The other was its lack of raster support.

3.1. From Internet mapping to Web GIS

The Internet is a massively scalable network, which interconnects billions of computers (Ericsson, 2017). Such a system is unimaginable without standardizing. While technically the World Wide Web is only a subset of the Internet, it is its most significant part with an estimated 4 billion human users (Miniwatts Marketing Group, 2019). The Web could not have achieve such statistics without an extraordinarily large number of standards, and software (e.g. browsers, server applications) complying with them. Web standards are maintained by the World Wide Web Consortium (W3C), an organization developing new standards with the help of academic institutions, companies, and the public. Both W3C and the Web started with a single proposal for three fundamental Web technologies: Uniform Resource Identifier (URI), Hypertext Transfer Protocol (HTTP), and Hypertext Markup Language (HTML) (Berners-Lee, 1989).

Since adequate standardization is not only essential for maintaining a scalable system but also feasible in decentralized, inter-organizational development, the spatial industry also followed this trend. Two major organizations have risen

up to maintain standards and organize projects. The Open Geospatial Consortium (OGC) has taken on developing and maintaining standards (OGC, 2019), just like the W3C in case of the Web. The Open Source Geospatial Foundation (OSGeo) complements this work by incubating and supporting open-source geospatial software (OSGeo, 2019). They also organize events helping developers from different projects to connect and collaborate. Unlike W3C, these two organizations do not restrict themselves to Web technologies and standards, but both of them are major contributors to the vision of a fully geo-enabled Web.

Early developers of Internet mapping applications did not adhere to geospatial standards, as it took some time for these geospatial organizations to take a foothold in the field of the Internet (Reed et al., 2015). They simply followed early Internet standards to make their applications work and paved the road for later participants (Peterson, 1999). A typical Internet map from this era had a thin client drawing styled maps as images, and a server generating new maps on every user interaction (Haklay et al., 2008). Thick clients were time-intensive to develop, therefore scarce, and they often served a special purpose, like interfacing with a locally installed desktop GIS software (Plewe, 1997).

The first geospatial Web standard, created by OGC's Web Mapping Special Interest Group, was drafted in 1998 (Reed et al., 2015), and released in 2000 (Doyle, 2000). Web Map Server (WMS) has standardized the rendering and communication pipeline of Internet maps. It differentiates between three client weights (thin, medium, thick) according to the amount of data styled on the client (Farkas, 2017a). While in modern software design there are other, more important optimization factors (e.g. analysis), back then data visualization was a serious bottleneck, resulting in such a classification. Most of the standard describes the way a WMS-enabled server and client should communicate. It uses a Representational State Transfer (REST) approach (Feng et al., 2009), using URIs to describe the state of the required map. According to URI parameters, the server is able to generate a map and return it to the client as an image. This architectural style defined later OGC Web standards, collectively known as OGC Web Services (OWS). It also influenced many other spatial standards, like OSGeo's Tile Map Service (TMS) specification (OSGeo, 2012).

It can be argued where is the border between Internet mapping and web mapping. When did we start creating new and trendy web maps instead of old fashioned Internet maps? The answer is the lack of sharp borders, but a long transition ending at around the release of Google Maps in 2005 in the Web 2.0 era. While it was one of the first dynamic web mapping applications which did not have to refresh the page between successive map states, its significance is due to more technical reasons. Google used several novel optimization methods in order to provide a satisfactory user experience.

The most argued optimization in the cartographic community was introducing a map projection, called Web Mercator, Spherical Mercator, or Google Mercator. It is a spherical, pseudo-conformal projection, using ellipsoidal (WGS 84) coordinates with a spherical projection (Battersby et al., 2014). It has both technical

and cartographic issues (on a global scale), leading to a very late adoption by the European Petrol Survey Group (EPGS) by public pressure (Nielsen, 2008). While this controversial projection received several criticisms from professionals, it became the de facto standard projection of web mapping (Stefanakis, 2017). Its popularity was initially due to the great performance of spherical calculations compared to ellipsoidal ones. As using the projection became a well-supported solution by both clients and servers, it still has not been given up in favor of better projections.

Due to the Mercator projection's properties, and Web Mercator's spherical nature, Earth in its full extent is projected to a perfect square. This property was favorable for the other great optimization novelty of Google: tiling up the map. Since on demand generation of maps was an expensive task, performance of traditional WMS services were limited (P. Yang et al., 2007). With tiling, Google mitigated this problem by cutting out the on demand rendering part from the pipeline.

Tiling is similar to building overview pyramids in case of rasters to increase rendering performance on smaller scales (C. Yang et al., 2011). In this process, a discrete number of scales are predefined and called zoom levels. The first zoom level contains the whole data extent as a single tile (traditionally a 256×256 pixels image). The next zoom level doubles the scale, and partitions that tile into four new tiles with more cartographic details on them. Upon a static tile generation process, each tile is stored locally, and served to the client according to a tiling pattern. As a result, performance is greatly increased on the expense of disk space, as every zoom level z has 4^z tiles to store with a total number of tiles of $\sum_{n=1}^{z} 4^n$.

Tiling is only useful, when a client knows how to compose a seamless map from individual tiles. Google Maps had a client-side which knew the tiling pattern, and could request new tiles according to user interactions (e.g. panning, zooming) without reloading the page. For this, one of the most significant Web 2.0 features, Asynchronous JavaScript and XML (AJAX) was used (Paulson, 2005). AJAX made possible to create rich and dynamic web applications instead of static web pages. These web mapping clients, capable of calculating positions, and drawing maps dynamically with on demand requests are called slippy maps (Batty et al., 2010). Slippy maps quickly became the new standard in web mapping.

Finally, Google Maps was not a simple product of web mapping. It offered an Application Programming Interface (API) with which developers could make their own applications based on Google Maps. They could overlay their own layers, and add custom functionality according to their customers' requests.

Starting a web mapping revolution, Google Maps influenced many new products and standards. Open-source APIs emerged (e.g. OpenLayers), and techniques were standardized. Tiling was such a prominent technique, it was reverse engineered in a few years (Liu et al., 2007), and got standardized by both OSGeo and OGC. OSGeo's WMS Tile Caching (WMS-C) got superseded by both OSGeo's TMS and OGC's Web Map Tile Service (WMTS). WMTS is the most capable one, letting users thoroughly customize the service, for example allowing them to define other projections than Web Mercator for their data (Masó et al., 2010). Both of them have standardized tile layouts (Figure 2), which are either supported out of the box,

or can be easily implemented in most web mapping libraries.

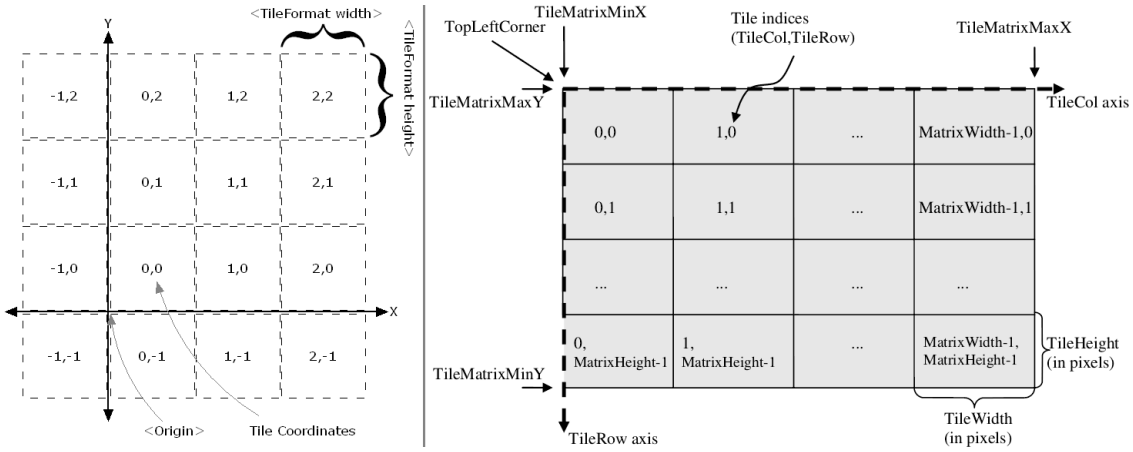


Figure 2: Tile layout of TMS (left) and WMTS (right). TMS has a loosely defined origin allowing for negative tile indices. WMTS indexing always starts with the top-left tile (Farkas, 2015).

In the meantime, two other important OGC specifications were released: Web Feature Service (WFS), and Web Coverage Service (WCS). WFS (Vretanos, 2002) standardized sending vector data over a network using a geospatial data exchange standard, Geographic Markup Language (GML). WCS (Evans, 2002) standardized sending raw raster data through the Internet. Initially, neither of them made an impact on web mapping, since client applications favored using images over styling raw vector data, while client-side raster management was not feasible.

3.2. Geospatial visualization on the Web

In the Web 2.0 era, development focus slowly shifted to client-side applications. Client computers became stronger and browsers became more capable. First, the main focus was on rendering spatial data on the client-side. The traditional way of rendering was injecting Document Object Model (DOM) elements (Wood et al., 2000) into the web page on user interactions. DOM elements are created by web clients (e.g. browsers) from HTML elements (Program 1). After loading a web page, the DOM structure (DOM tree, as it is a tree structure) represents the state of the page. It can be modified with JavaScript, making the web page more dynamic. Image overlays and tiles used image elements, while vector data were visualized with Scalable Vector Graphics (SVG) elements.

3.2.1. Early days of vector data in web clients

SVG allowed for styling raw vector data on the client-side, although performance was a bottleneck (Jayathilake et al., 2011). Making things worse, downloading and parsing verbose GML outputs from WFS services put additional overhead on web maps (Peng & Zhang, 2004). One of the necessary breakthroughs came with a

```
<html>
    <head>
        <title>An example page</title>
    </head>
    <body>
        <div>
            <img src="https://i.pinimg.com/originals/e6/63/12/
                e663126a9bf25fcddbe8abdceab2b4ce.jpg">
        </div>
    </body>
</html>
```

Program 1: A short HTML example. After parsing, each element is saved into the DOM tree as a DOM element. Every attribute like visibility, styling, or the element's place in the hierarchy is stored in the element's DOM object.

new, Web-based geospatial data exchange standard. In 2008, the initial release of GeoJSON (Butler et al., 2016), a spatial extension of the concise JavaScript Object Notation (JSON) reduced vector data sizes served over the Internet.

While first there were only a handful of web mapping libraries to build upon, around 2010, their numbers increased. Decentralized, collaborative open-source development became easier with platforms built around Version Control Systems (VCS). Some of those platforms (e.g. GitHub, Bitbucket) have supported free hosting and management for open-source projects. From the numerous projects, two different designs can be distinguished. There were general data visualization libraries capable of handling spatial data (e.g. D3, Processing.js, Raphaël), and web mapping libraries specialized in visualizing spatial data (e.g. OpenLayers 2, Leaflet). Among these libraries, one project, D3 has been optimized very carefully (Bostock et al., 2011), still, its DOM-based rendering puts a hard limit on the maximum number of feasibly displayable features.

The other innovation required for faster vector visualization was provided by the HTML5 specification (Hickson et al., 2014). Part of the specification was the Canvas rendering context (Cabanier et al., 2015). The Canvas specification introduced a drawing API for drawing textures, texts, points, line strings, and polygons on the web page. By using the API in web mapping libraries, developers could abandon inserting and manipulating SVG elements in the DOM tree, making the application more scalable (Figure 3).

First Canvas did not provide better performance than traditional SVG methods, although its merits coming from the possibility of manipulating rasterized results on a per-pixel basis were clearly understood (Sauerwein, 2010). Later, as the Canvas technology matured, its vector rendering performance has overcome SVG rendering. This was mostly due to the ability of partially hardware accelerating the rendering pipeline (Curran & George, 2012). By outsourcing some of the rendering tasks to the GPU, browsers could achieve much greater performance than with traditional

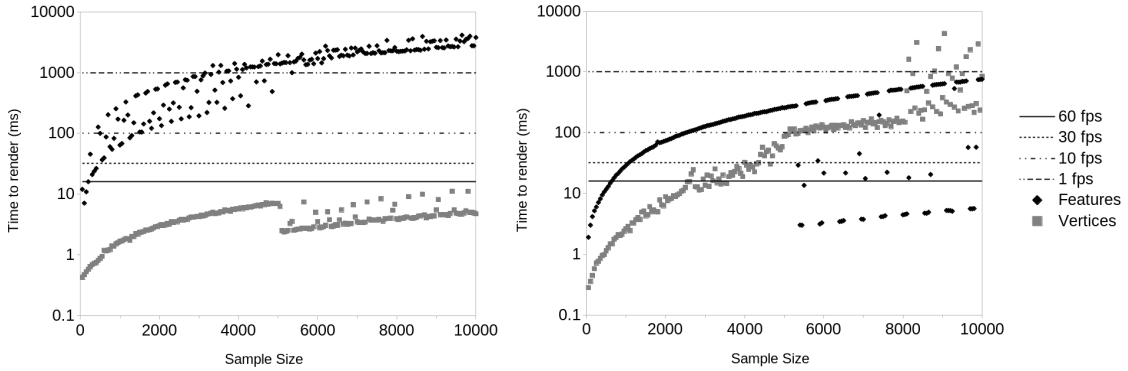


Figure 3: DOM and Canvas performance with different line string samples. OpenLayers 2 (left) with a DOM vector renderer (SVG) is less sensitive to the number of vertices in a single line string, however, with an increasing number of features, it quickly becomes unresponsive. OpenLayers 3 (right) using a Canvas renderer has better overall scalability, but it is more sensitive to the number of vertices in a single feature (Farkas, 2019).

DOM elements.

3.2.2. Hardware accelerated vector rendering

The last step in bringing web-based data visualization performance closer to desktop applications was the introduction of WebGL (Leung & Salga, 2010). This specification initially designed by Khronos Group (Marrin, 2011), the organization behind leading hardware acceleration technologies like OpenGL and Vulkan. WebGL 1.0 is a subset of the OpenGL ES 2.0 API. It allows utilizing the GPU from web applications directly through different client-side OpenGL Shading Language (GLSL) scripts complementing the applications' JavaScript code. While this technology only allows a narrow subset of OpenGL functionality to be used, developers could have still achieved proper hardware acceleration, starting a new era of 3D web applications.

The appearance of WebGL influenced web mapping in several ways. By the end of the long adaptation process, a new type of web mapping application was born. Virtual globes are special 3D applications, visualizing spatial data on a spherical or ellipsoidal surface (Christen et al., 2012). By leveraging the capabilities of WebGL, web-based virtual globe APIs were developed, offering 3D or 4D spatial visualizations (Konde & Saran, 2017). While there were other, general 3D modeling APIs adaptable to geospatial visualization (Resch et al., 2014), virtual globes were closer to web mapping. They had built-in definitions about geospatial concepts like geographic coordinates or projections (Gede, 2015).

In the case of 2D web mapping applications, WebGL made faster vector visualization possible, which allowed a larger number of visible features in a map. As a result, some web mapping applications (e.g. Google Maps, Mapbox) started experimenting with vector maps. This approach stopped using precomposed and

prerendered image tiles as base maps, every content was raw vector data styled on the client-side. WebGL, on the other hand, was still not sufficient for visualizing arbitrary sized layers, therefore further optimizations were required.

These optimizations targeted vector data exchange formats used on the Web. One of these approaches was a compressed GeoJSON format, called TopoJSON (Bostock & Metcalf, 2013). TopoJSON uses a reversible quantization technique to have integer coordinates, and stores topology to reduce redundancy. It can achieve an 80% reduction in file size, although both quantizing and calculating geometries need computing. The other prominent technique was using Protocol Buffers, a binary format for structured data created and open-sourced by Google (G. Kaur & Fuad, 2010). While this format was first used by Google to reduce data exchange in its applications, it could be extended to transfer spatial data efficiently (Steiniger & Hunter, 2017). Data in Protocol Buffers is very fast to transfer and to parse, although the format is binary, therefore limited to browsers with ECMAScript 2015 (JavaScript 6) support.

Another important innovation was the concept of vector tiling (Antoniou et al., 2009). The idea of vector tiling has been utilized by both Google and Mapbox using their own methods (Ingensand et al., 2016). While Google's vector tiling was created as a closed standard, Mapbox created an open-source standard called Mapbox Vector Tile (MVT). This specification has slowly become a widely supported way of using vector tiles (Springmeyer, 2015).

Vector tiling uses the same concept as regular image tiling. Vector data are split according to a tile layout, the tiles are stored on the server. On the other hand, there are some significant technical discrepancies between the two techniques. In case of vector tiling, the same vector layers get clipped to a grid, but usually with different generalization levels. These generalizations are called Level of Detail (LOD) levels instead of zoom levels (Gaffuri, 2012). Since vector-only maps can provide smoother transitions between LODs, vector tile renderers (e.g. Google Maps, Mapbox GL JS) are using more zoom levels, changing the feeling of vector-based web maps.

A further peculiarity of using this technique is, polygon rendering needs to be prepared for vector tiles. Since polygons can span over multiple tiles, they often need to be cut at tile borders. Consequently, if a renderer strives for correct visualization, it needs to distinguish tile borders from polygon boundaries (Persson, 2004).

Since vector tiles store raw vector data (e.g. geometries and attributes) rather than pixel color values, they can have various, significantly different formats. Some of the most popular formats are GeoJSON, TopoJSON, and MVT. From these, MVT is the most efficient, as it uses Protocol Buffers for encoding spatial data. On the other hand, it is optimized for rendering (Eugene et al., 2017), therefore MVT is better for base maps, where individual features are not queried.

Currently there is only one application blending most of these methods together in order to give a solution to end users: Google Maps (Figure 4). This state of the art application successfully demonstrates, how the technologies mentioned beforehand can be utilized to create an efficient 2D/3D web mapping application with a virtual globe and some GIS capabilities.

Figure 4: Screenshot of the current Google Maps application. It uses vector data where it can, has smooth transitions between traditional zoom levels, and renders sharp features and labels on every zoom level. It is a WebGL-based 3D application including a virtual globe, but showing larger scales in 2D by default (Google, 2019).

3.2.3. Rasters on the Web

Raster handling went a different way in web mapping than in desktop GIS due to various factors. In desktop GIS, raster was an essential data structure, since it was the only feasible solution for handling spatial data in early architectures (Lim, 2008). The spread of vector visualization and vector processing did not render the raster model superfluous, only changed its function. Ever since, raster data has been used for representing continuous phenomena, while vector data has been used for discrete entities (Bugya & Farkas, 2018).

During the maturation of web mapping, there were already well-established desktop programs for raster management capable of running on the server-side of a Web application. GDAL (Warmerdam, 2008) is still the de facto standard for headless (no Graphical User Interface, GUI) raster processing and conversion in open-source solutions. It can be used as a low-level library in software, and a Command Line Interface (CLI) tool in scripted processes. GRASS (Neteler et al., 2012) is a complete GIS capable of scientific raster analyses in both interactive and headless mode. There are also other lower-level tools – like GeoTools (Turton, 2008) – built around these technologies which could be used in geospatial server applications.

With Web 2.0, HTML5, and the start of modern web mapping, server-side applications were perfectly capable of manipulating rasters and providing image results to clients. This was feasible, since arbitrary sized raw rasters could not be handled by browsers. One of the main restrictions was the supported image formats. Browsers still only support a limited number of formats (e.g. PNG, JPEG, GIF, BMP), which are appropriate for Web use. TIFFs are not considered appropriate,

since it is not a streaming format (i.e. the whole file needs to be downloaded before displaying), and file sizes can be rather big (Firefox Contributors, 2002).

If TIFFs were supported, browsers still could not handle GeoTIFF files (Ritter & Ruth, 1997). They do not have a concept about geospatial metadata, could not apply a representation model on non-color values (e.g. elevation, precipitation, temperature), and could not use burned-in palettes for categorical data. While ASCII formats, such as ArcInfo ASCII Grid (Yu & Custer, 2006), could still be handled by early client-side applications, ASCII file sizes can be very large in case of rasters. On the contrary, image results received from a geospatial server have been small, and easy to use.

Consequently, raster became synonymous with image in web mapping applications (Król, 2018). Until ECMAScript 2015, there were no significant changes in client-side raster management. After JavaScript was capable of operating on binary data, virtual globes were the first to take advantage of the new possibilities by using raster formats. Desktop virtual globes already relied on rasters for terrain rendering (Cozzi & Ring, 2011). Since web mapping applications already used tiled data, these formats just needed to be ported to JavaScript. During the evolution of Web virtual globes, these formats have undergone some optimizations, though. First, the standard Heightmap format (similar to Digital Elevation Models, DEMs in GIS) got a TMS service (Di Staso et al., 2016). Later, it has been optimized to store data relative to the terrain's complexity, resulting in the quantized mesh format (Krämer & Gutbell, 2015).

With binary enabled JavaScript, numerous other libraries were created for handling binary data exchange formats. From these, geotiff.js can be used for parsing raw GeoTIFF files and gather raw raster data from them (Schindler, 2016). There are still no web mapping libraries utilizing this data abstraction tool for overlaying client-side rasters out of the box. On the other hand, the NASA Web World Wind virtual globe has implemented this capability (NASA, 2018), showing that client-side raster visualization is slowly spreading in the field of Web-based geospatial visualization.

3.3. Web GIS clients

Web mapping and Web GIS are often interchangeably used terms in Web-based geospatial application development. There are some identifiable distinctions in the literature, though. Web mapping is more widely used for client-side applications, while Web GIS is more often used for complete systems consisting of a server and a client-side. The use of Web GIS for such an application is understandable. A GIS separates from a spatial data visualization application by true GIS features (Thrall & Thrall, 1999), like spatial operations. In a Web GIS, the server-side possesses GIS capabilities, no matter how thin the client is.

Clients alone are less frequently specified as Web GIS. Clients with GIS functionality are more often considered as a part of a whole Web GIS system. Consequently, Web GIS clients do not have a well-recognized denomination. A popular categoriza-

tion differentiates between thin, medium, and thick clients (Doyle, 2000), but only considers the amount of data rendered in the client. When Web GIS clients did appear in the literature, some called them GIS clients (Plewe, 1997), some called them massive clients (Farkas, 2017a). Disputing the term to be used is a cavil, although it is important to acknowledge this rising client category in Web GIS.

If massive clients are considered as Web GIS clients with GIS functionality, every modern Web GIS client could fit in, as most of them are capable of at least some coordinate transformations. To reduce the ambiguity of the category, only those clients are considered, which have spatial data analysis capabilities. Still, the variability in massive clients is high, as there are many such GIS features.

The high number of possibilities for creating a massive client is only problematic when they are compared. In such cases, however, the assessments' creators should declare a finite set of features they are interested in. For this study, those are universal GIS features. That is, the bare minimum GIS functionality with which a massive client can be called a serverless universal GIS.

3.3.1. A universal GIS

Geographic Information Systems have evolved from multiple other systems. Among multiple definitions in classic literature, GIS has been defined as the intersection of four preceding systems: computer cartography, database management (DBMS), computer-aided design (CAD), and remote sensing (Maguire, 1991). While a basic GIS can omit remote sensing capabilities, the other three categories are essential parts for spatial data visualization, geometry manipulation, and attribute data management.

These non-GIS-specific features form the inner core of a GIS. The structure of spatial data in a GIS comes from computer cartography. Irrespective of the given system's internal data structure, spatial data are organized on individual layers. Every layer is part of the visualized locality, representing one coherent characteristic (Tomlin, 2017). The most determinative feature of computer cartography inherited by GIS is the representation model. A GIS must be capable of creating different types of maps from spatial data by applying different visual variables on raw data (Roth, 2017).

From the DBMS domain, GIS has inherited an internal relational data structure used for attribute management. Since vector features can hold as many attributes as they see fit, the relational architecture keeps those values consistent. While in most cases the end-user only sees an attribute table with a field calculator, this design ensures a fast bidirectional connection between geometries and attributes. Moreover, by being compatible with relational DBMS (RDBMS) software, GIS software can be integrated with spatial databases allowing for secure collaborative work. The GIS software does not have to ensure its data integrity in parallel use, since RDBMS software already follow the atomicity, consistency, isolation, and durability (ACID) principles (Haerder & Reuter, 1983).

CAD functionalities form the basics of vector geometry management in GIS.

Those are mostly related to interactions, like geometry selection, affine transformations (e.g. shift, rotate, scale), or updating attribute data on a per feature basis. There are also low-level rendering capabilities related to CAD software, like geometry styling or hardware accelerated vector visualization.

While remote sensing capabilities are not necessarily mandatory features in a basic universal GIS, there are some features from this domain which can hold interest in such a system. These are image processing algorithms, making basic raster analysis possible. If some less complex algorithms (e.g. map algebra, reclassification, convolution) are implemented, users can execute fairly complex raster analyses by involving them in a longer workflow, possibly also including some GIS-specific analysis features.

By using up mature techniques from other information systems, the core of a GIS was already given. This made creating GIS software easier and therefore faster but enforced some inherent conceptual limitations on them. There were debates about the long term weaknesses of using a relatively low complexity concept for modeling geographical phenomena (Chrisman, 1987). There were even proposed solutions for avoiding such limitations (Goodchild et al., 2007). However, the aforementioned design principles have remained being the basis of GIS, and new challenges have been solved by continuously building on them up to date.

Building on these features, there are numerous GIS-specific features of interest in a universal GIS client. One of the main groups is data abstraction. A universal GIS must accept spatial data in common formats, and it must be able to produce exports in them for interoperability. While there are a large number of spatial data exchange formats, there is no point in handling all of them in a basic GIS. It should be enough to support the most popular ones, like Shapefile, GeoJSON, KML, GeoTIFF, and ArcInfo ASCII Grid (Orlik & Orlikova, 2014).

It can be argued if the support of coordinate reference systems (CRS) should be considered as a non-GIS-specific (computer cartography) or a GIS-specific feature. Proper projection handling involves on-the-fly transformation of both raster and vector data to a common projection in modern clients. Since on-the-fly transformation involves reprojecting geometries and warping rasters, CRS handling is considered a GIS feature in this study.

The last group of features in this category is GIS-specific spatial analysis capabilities. While a basic GIS client does not need to have excessive amounts of analysis features, it should be able to execute basic analysis on features and rasters. From a conclusive list of universal GIS features (Albrecht, 1998), a basic client should be able to make measurements, interpolate, search based on spatial relationships, and execute basic geoprocesses on vector layers. Those basic geoprocesses include buffering, dissolving, point-in-polygon (PIP) operations (Thrall & Thrall, 1999), and spatial set operations in continuous space (i.e. intersection, union, difference, symmetrical difference on geometries). Furthermore, there should be an option for converting between vector and raster data types without interpolating (Meaden & Chi, 1996).

If a client possesses these capabilities, it can firmly be called a universal GIS

client. However, a universal Web GIS client must have some Web related capabilities, since it belongs in a special niche. These capabilities are mostly related to services (e.g. WMS, WFS, WCS). Otherwise, without the ability to communicate with a standard spatial server, a Web GIS client would be limited. Since spatial databases can hold a huge amount of data (Agrawal & Gupta, 2014), and Web clients cannot connect to databases yet, currently this is the only way to use RDBMSs in a Web GIS client.

By categorizing these features, and extending or refining them where necessary, a comprehensive list can be created (Figure 5). Since – as in most of such comparisons – some subjectivity is involved, it is important to note that this is not the only way to compare massive clients for their universality. This is merely a possibility based on past literature.

Figure 5: Features of a basic universal massive Web GIS client grouped by functionality. Triangles, crosses, and diamonds denote GIS-specific features, non-GIS-specific features, and web mapping specific features respectively (Farkas, 2017a).

4. Materials and Methods

The study consists of several, methodically different steps, therefore multiple methods have been involved. The first step was choosing the right basis for a universal massive client. Since a list of spatial data visualization libraries were to be compared, this step mostly relies on theory. The list of GIS features used by this study (Figure 5) served as the basis of comparison, that is the list of GIS features compared amongst the chosen libraries.

While a competitive analysis is useful, it is not enough to cover every characteristic of the compared libraries. There are some aspects, which are hard to grasp, but affect developers significantly. Some of them, like the level of documentation, number of tutorials, or developer activity can be quantified or qualified. However, the complexity of a piece of software from users' perspective is very hard to determine. The standard method for assessing such a characteristic is creating expert surveys (Roth et al., 2014). On the other hand, there are some static software metrics which can shed some light on the problem without the long process of surveying.

4.1. Scoring libraries on GIS features

During the competitive analysis, candidate libraries were scored based on their support of each examined feature. Several features consist of subfeatures, which need to be supported in order to achieve a perfect score. If a library fully implemented a feature, it got a full score of 1.

Partial scores were given in two cases. There were cases when a library's support were only partial due to the negligence of one or more subfeatures. In other cases, the library did not implement a feature at all, but there was a third party extension implementing that feature. In both of those cases, the library were given a partial score of 0.5.

It worth noting why third party implementations reduced the score of a library. Plugins are often developed by a small group of developers in need of the given feature, independent from the core development of a library. Therefore, the development process of the two software differ. While core features are maintained with the evolution of a library, extensions need to be upgraded with every major release introducing changes in the library's API. Consequently, by supporting a feature in an extension, there is no guarantee for a plugin to support the next version. If it has support, there can be delays between a new release of the core library, and the upgrade. This reduces reliability with every plugin introduced into the Web GIS

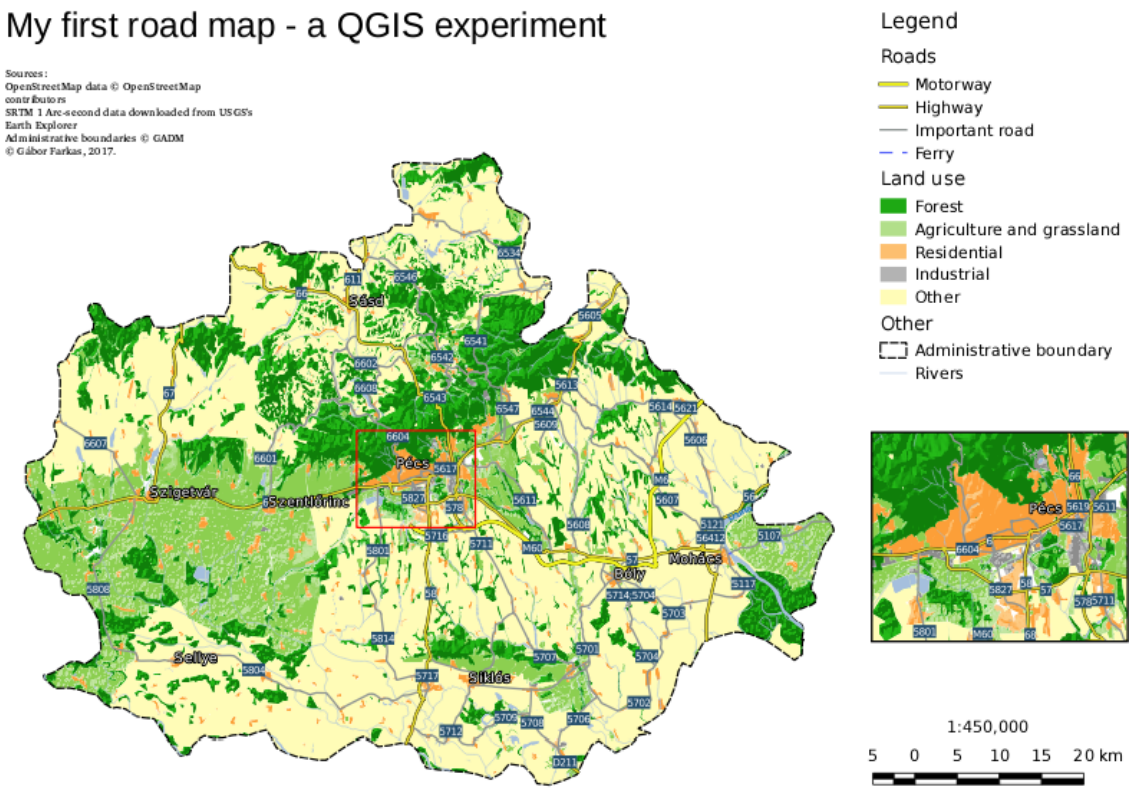


Figure 6: A hillshade representation of a DEM blended over a land use layer in QGIS (Farkas, 2017b).

solution. In the worst case, an extension's development ceases, tying users to a single release of the core library.

If a library did not support a feature at all, it got a score of 0. There were several cases, when this was inevitable (e.g. connecting to spatial databases). Those features were included with considerations for future development of Web technologies. This way, if at some point browsers will be able connect to databases, the evaluation's frame does not need to be revised.

Final scores were provided by averaging support points, resulting in a 100% coverage in cases when a library offers core support for every examined feature. Since the final score does not give a complete picture about the strengths and peculiarities of candidates, subcategories (e.g. rendering, format handling, representation) were also evaluated.

There were seven examined categories amongst candidate libraries (Figure 5). Rendering consists of non GIS specific features responsible for drawing entities, and overlaying map contents efficiently. Rendering geometries, rasters, and images are trivial, while hardware acceleration is necessary for drawing large number of features with feasible performance. Blending layers is also included, as it is a typical feature in computer graphics, and while not supported in every GIS, has cartographic merits (Figure 6).

Format handling was divided into four subcategories. From various vector formats, ESRI Shapefile as a popular binary format, Keyhole Markup Language (KML) as a popular ASCII format, and GeoJSON as a popular Web format were selected. Now the Geopackage format might be a better interest than Shapefile, however, in the time of the original survey, this trend change was not clear. From geospatial Web services, WFS, and Transactional WFS (WFS-T) were selected, since they are one of the best ways to communicate with a server side geospatial database. With WFS-T, users can not only read from, but also write back changes to the database.

From raster formats, the most popular binary format of GeoTIFF was chosen, along with one of the most popular ASCII formats: ArcInfo ASCII Grid. WCS capabilities were also checked, as it is a standard way to request raster data on the Web. Images were split between traditional image formats and tile services. Traditional image formats included WMS, since it usually communicates maps in JPEG and PNG in a standardized way. Tile services included some of the most popular standards along with commercial services (WMTS, TMS, OSM's slippy map format, Google Maps, ArcGIS REST API, Bing Maps).

Database related features consisted of two subcategories: connection and functionality. In the connection category, the ability of connecting to popular spatial servers (PostGIS, SpatiaLite, MySQL) was assessed. While it is known that browsers cannot connect to server side databases directly, this subcategory still servers a purpose as space for further development. For example, since the original survey, a JavaScript library was released for handling SQLite database files (Zakai, 2019), which form the basis for SpatiaLite files.

Database functionalities do not have to rely on external databases, since JavaScript is capable of storing data in an SQL-like structure called Indexed Database (IndexedDB). While it is conceptually similar to RDBMSs, it can store large amounts of data in an object oriented format. From this group, a simple implementation would yield a point, while querying capabilities along with a spatial query language were also examined. These functionalities were checked since an internal database has a key role in not only querying, but also processing data efficiently (Revesz, 2008).

The data group had a significant amount of features, since most of the GIS analysis features formed an integral part of it. Preprocessing was mainly considered as different spatial procedures applied to vector data before visualizing or analyzing. Proper attribute management (read attributes, build attribute table) were surveyed as well as geometry related procedures. Geometry management features were low level functionality, like the support of Z and M (measure) coordinates, geometry validation, single type vector layers, spatial indexing, generalization, and on the fly transformation.

Data conversion had three different features. Besides the typical vector to raster, and raster to vector conversions, interpolation methods were also considered in this subcategory. Just like the former two, interpolation had to come with successful type conversion (i.e. vector to raster). Specialized methods with an interpolation

aspect (e.g. creating heatmaps) were not considered.

In the subgroup of data manipulation, the most essential features were collected in order to manage layers, and edit vector layers. From layer management, adding–, removing–, and changing the order of layers held the most significance for usability, while typed layers (e.g. vector, raster, image, and subtypes) are also important for programmability. Apart from that, one should be able to edit vector layers (geometries and attributes). Attribute data should also be updated with bulk processes, preferably through some kind of field calculator mechanism.

Under data analysis functionalities, it was striven to group some of the lower level analysis algorithms, on which one can build advanced GIS tools. Basic geoprocessing have low level vector analysis techniques, like creating buffer zones, dissolving geometries, and clipping. Topological analysis is not trivial, as it groups basic techniques acting on topological relationships, rather than algorithms operating on topological data structures. This category surveyed the existence of intersecting, uniting, and calculating the differences between input geometries.

The rest of the subgroup consisted of raster analysis techniques. Besides to the ability of modifying images and rasters, a basic GIS was considered to also need classification and raster algebra functionalities. While convolution (moving window in GRASS GIS, focal statistics in ArcMap) could be considered as nonessential, it is easy to support, and one can utilize it for different specialized tasks (e.g. terrain analysis). Finally, writing Web Processing Service (WPS) requests is a service type functionality. It was deemed important, as it can be used to send requests for spatial operations on layers to a capable spatial server (Schut, 2007).

Projections support assessed some of the most important projection related capabilities. Transforming vector layers and warping raster layers are capabilities directly relying on proper projection support. If a library can do these operations, it can handle projections. It also matters how many different projections a library can recognize and use. Under custom projections, the two most popular and essential projections were assessed: WGS84 Plate Carrée (EPSG:4326), and Web Mercator (EPSG:3857). There are many more existing projections, which are essential in GIS workflows. Therefore, the ability to use arbitrary, but well-known projections was also assessed. If a library can recognize every projection in the EPSG database out of the box, it was considered as full support.

Interactivity is a core concept in any library capable of spatial visualization. For this, the most significant interactions were collected in the interaction group. A library must support modifying its view, which consists of panning, zooming, and rotating the map. It is beneficial, if it can show mouse coordinates, and measure distances on the map. The capability of changing time is essential for spatiotemporal data. Finally, users should be given interactions for convenient geometry editing, which are drawing, modifying (i.e. at least translating and rotating), selecting, querying, and snapping.

The representation category consists of two subgroups. First, styling capabilities were considered. A library should give options to style both vector and raster data, and build representation models based on user provided definitions. The end

product of a workflow is usually some kind of thematic map. This feature examined the capability to create the two most basic thematic representations: choropleth and proportional symbol maps.

When a user produces a map from results, it is often a requirement to add cartographic elements. Regardless if the product is a digital map or an interactive map, there are some basic cartographic elements which should be supported by an ideal candidate. Scale bars and legends are the most usual elements, while graticules and overview maps (or inset maps) are also common. While the necessity of a north arrow is debatable, there are some cases when it should be provided for easier orientation. Finally, users should be able to add static text boxes to the map, which are not bound to the map, but to its container. This feature is convenient for static, printable products.

4.2. Static software metrics

Static software metrics are properties or indices of a software derived from the source code. They are used to to make assumptions to software quality, complexity, or maintainability by looking at the source code from the outside. Some of the most important, classical metrics are Lines of Code (LOC), cyclomatic complexity, and Halstead's software science metrics (Fenton & Neil, 1999).

Since LOC is very crude, there are some more popular alternatives, which can be acquired just as easily. One of them is Logical LOC (LLOC), which considers every statement a separate line. This excludes not only line breaks and comments, but also groups control flow statements spanning over multiple lines for better readability (Nguyen et al., 2007). Since LLOC is language independent, it can be used for comparing client libraries.

Halstead went forward by calculating distinct and total operators and operands in the source code. He defined indices (e.g. volume, vocabulary, length, level, effort) calculated from those inputs with simple equations (K. Kaur et al., 2009). Some of the software science metrics were quite ambitious relative to the simplicity of input variables. Consequently, Halstead used some assumptions related to human efficiency, such as the human brain uses a binary search mechanism, or a human brain can make 18 elementary discriminations (Stroud number) per second (Shepperd & Ince, 1994). Despite the criticism, software science metrics have been useful in different applications (Jones, 2001). However, since they were evaluated on small FORTRAN and COBOL programs, and they ignore many important new aspects of software development (Shepperd & Ince, 1994), they are not that useful in case of assessing JavaScript libraries.

Cyclomatic complexity is another universal metric with a different approach. It is graph-theoretic, thus it creates a graph from the source code. In the formula (Equation 1) it calculates with blocks of code (n vertices), which are delimited by control flow statements, like if-else clauses and for loops (e edges). Connected subroutines (p components) are also counted (McCabe, 1976). In the end, it treats the code as a control flow, assuming the complexity of the code is proportional to

the complexity of the generated graph. Cyclomatic complexity alone is hard to compare between different libraries, due to its direct proportion with the size of the project (LLOC already accounts for that). However, normalizing it with the number of functions F can add a useful point to the total complexity of a library.

$$v(G) = e - n + p \tag{1}$$

In order to approximate the total complexity of a library, an evasive characteristic was targeted: the learning curve. It is very hard – if not impossible – to give a general mathematical function for calculating the learning curve of a piece of software, which will apply to every developer. On the other hand, it can be assumed, if enough aspects of complexity are covered, a formula from static software metrics can make a crude approximation. If the results are correct by the order of magnitude, it can be used as a filter for narrowing down the subjects in an assessment. Then a more precise, survey-based analysis can be done with a better focus.

The final metric, Approximate Learning Curve for JavaScript ($\mathrm{ALC_{JS}}$) was tailored for JavaScript libraries specifically (Farkas, 2017a). The formula (Equation 2) includes an additional part, which was not discussed before. Seemingly, the number of LLOC per exposed functions EF have the greatest impact on the learning curve of a project (Fowler et al., 1999). Consequently, the number of exposed functions was also measured.

$$ALC_{JS} = \log_{10} LLOC \times \log_2(\frac{v(G)}{F} \times \frac{LLOC}{EF}) \tag{2}$$

The formula has two equally weighted parts multiplied together. The left gives a score considering the size of the library. The right side creates a score based on the library's complexity, measuring per function size and cyclomatic complexity. Size is considered as an external property (user's aspect), while cyclomatic complexity is considered as an internal property (developer's aspect). The two logarithmic transformations are used for equalizing weights. The base ten transformation is used for calculating the order of magnitude of a library's size. The base two transformation is an empirical value to counteract the slight imbalance towards the right side of the equation.

Static software metrics were measured using complexity-report, an open source tool written for Node.js, and designed for measuring JavaScript code. There was only one problematic metric which could not been recorded with complexity-report: the number of exposed functions. Exposed functions were considered as classes and static functions available for users from a library's namespace. Contrary to class based languages, JavaScript uses prototypes, and constructors look like regular functions. This makes counting exposed functions hard from the outside. Consequently, a small JavaScript routine was created (Program 2), which was able to easily count functions directly accessible from the namespace.

While static software metrics are useful for giving an initial picture about the complexity of a library, there are some other characteristics contributing to the overall usability. Some of the more important ones are documentation, community,

```
function measure(obj, visited) {
  visited = visited || [];
  var count = 0;
  if (obj !== window) {
    visited.push(obj);
    for (var i in obj) {
      if (visited.indexOf(obj[i]) === -1) {
        visited.push(obj[i]);
        if (typeof obj[i] === 'object') {
          count += measure(obj[i], visited);
        } else if (typeof obj[i] === 'function') {
          count++;
        }
      }
    }
  }
  return count;
};
```

Program 2: A JavaScript routine counting accessible functions from a namespace. The result is a static software metric, as the same measurement can be done from the outside with appropriate filtering. The routine deliberately skips methods, since they belong to their respective classes from the perspective of users. Therefore, duplicate methods from class inheritance do not need to be handled.

Category	Documentation score	Ratio of answered questions
Poor	0	0 – 0.25
Decent	0 – 0.5	0.25 – 0.5
Good	0.5 – 1	0.5 – 0.75
Very good	1 –	0.75 – 1

Table 1: Rules of applying ordinal values to raw documentation and community support scores. Apart from the Poor category, the intervals are exclusive of the first value, and inclusive of the second.

and support (Ramsey, 2007; Steiniger & Hunter, 2013; Poorazizi & Hunter, 2015). Those characteristics are hard to evaluate, since there are numerous different ways to measure them, and they are usually evaluated using an ordinal scale. In order to make the measurements reproducible, a method was designed based on numeric attributes.

A documentation score (Equation 3) was calculated from the number of API documentation A (basically its existence), the number of tutorials T, and the number of examples E. Since the API documentation is essential for users, its existence does not improve the score, but its absence results in a score of 0. The number of tutorials and examples are both scaled according to their importance. Since tutorials are more comprehensive, harder to create, and help users accommodate themselves to the library faster, they receive a larger weight than examples. Examples are short solutions for specific functionalities or problems, therefore they get a smaller weight. In the end, the final scores were converted to ordinal values (Table 1).

$$Score = A \times (T/10 + E/100) \tag{3}$$

Community and support are soft metrics, which are hard to evaluate. However, as the assessed web mapping libraries are open source products, it is possible to use public Version Control Systems (VCS) statistics to acquire a decent picture about the developer community. In this study, two such metrics were collected. The first one is the number of contributors and major contributors in the project. Major contributors are considered as developers, who added more than 1000 lines to the source code. The second one is a release frequency, which can be calculated from the number of releases n, and the days passed between the first D_{FR} and the last release D_{LR} (Equation 4).

$$RF = \frac{D_{LR} - D_{FR}}{n - 1} \tag{4}$$

Support data were collected from two sources. Since most of the compared projects are using GitHub as a VCS, GitHub's issue system could be leveraged to collect some information about developer support. In this metric, the number of open issues were collected in contrast to the number of total issues of a library. For community support, two popular forums were involved. Stack Overflow and GIS

Metric	Description
Feature matrix	Support of basic GIS features which can act as indicators for the massiveness of the library.
Size	Physical size of the production version of the library.
LLOC	Logical lines of code in the production version of the library.
CC/F	Cyclomatic complexity of the library normalized with its number of functions.
Exposed functions	Functions which can be directly invoked from the namespace of the library.
ALC_{JS}	Approximate learning curve derived from static metrics.
Documentation	Documentation quality derived from the library's API, tutorials, and examples.
Contributors	The number of contributors and major contributors of the library.
Release frequency	Average number of days between releases.
Open issues	Number of open issues and their ratio to total issues in the library's VCS.
Community support	Quality of community support derived from number of answered questions on Stack Exchange forums related to the library.

Table 2: Summary of the metrics used for evaluating candidate libraries, and choosing the best one for a general massive Web GIS client (Farkas, 2017a).

Stack Exchange are question and answer (Q&A) sites, where users can post problems encountered with a specific technology. To these questions, both developers and more experienced users can post solutions. Once a proper solution is found, the problem is marked as answered. Apart from the possibility of gaining quick support, these forums offer statistics regarding questions asked and answered, grouped by technology. The ratio of answered questions was used to assess the quality of community support on an ordinal scale (Table 1).

These metrics (Table 2) were used to evalutate the libraries of interest two different times. The first comparison was done in 2016 (Farkas, 2017a), showing a possible methodology for such an assessment. The second one was created in 2019, after the applied improvements, measuring the impact of the research. This way, it can be seen, if the selected technology was a good choice. Furthermore, a second data point is created for monitoring the development process of client side web mapping technologies.

4.3. Benchmarking

In order to evaluate the performance of extensions written for the chosen library, they were benchmarked for both performance and memory footprint. Different methods were used for benchmarking hardware accelerated rendering, and raster

management. The common device used by every benchmark was a Dell Inspiron 7567 laptop with a Core i7-7700HQ CPU, 8 GB DDR4 RAM, an NVIDIA GeForce GTX 1050 Ti dedicated GPU (through the propertiary NVIDIA driver), an Intel HD Graphics 630 integrated GPU, and a 15.6-inch display with a resolution of 1920×1080 pixels. Since client side applications needed to be measured, a Web browser was used for all of the benchmarks. It was a 64-bit Chromium with hardware accelerated Canvas, running on a Debian 9 OS.

Since hardware accelerated rendering is a general technique, another, weaker device was also introduced for those tests. It was a Lenovo A536 smartphone with a Quad-core 1.3 GHz Cortex-A7 CPU, 1 GB RAM, an ARM Mali-400 MP2 GPU, and a 5-inch display with a resolution of 480×854 pixels. The browser used on the smartphone was a 32-bit Chrome running on Android 4.4.2, accessed through remote debugging. From the available browsers, Chrome and Chromium were chosen due to their similar code bases, and their excellent developer tools.

Hardware accelerated rendering benchmarks were carried out using a specific application (Farkas, 2018b) written for the measurements. It used the high precision Performance Timeline API (Grigorik et al., 2016) by measuring several consecutive redraws and writing the drawing time to the browser's console. In the end, outliers were filtered out, and the rest of the data points were averaged for a representative result. Outliers were common mostly due to initial overhead and unexpected load from external processes.

Since a web mapping library can cache rendered data to further accelerate drawing speed during animations and user interactions (e.g. pan, zoom), animations and complete redraws were measured separately. Complete redraws were measured by repositioning the map back and forth on the X axis automatically (Program 3). There were 10 successive measurements for each zoom level. Due to the low amount of measurements – which is a result of low performance with large datasets – if a measurement produced outliers, it was repeated.

Animation measurement used a similar approach, although it performed a panning animation on a predefined path instead of moving the map back and forth. The program measured elapsed time between frames, making the number of data points a function of frame rate. Since frame rates can be low under heavy load, the received data points from a single measurement varied between 5 and 100. If a measurement produced fewer than 10 stable data points, it was repeated, and the new results were added to previous ones.

Furthermore, using the developer tools of Chromium, detailed performance tests were conducted (Figure 7). These benchmarks measured the ratio of time spent on different phases of the rendering pipeline. With this method, possible bottlenecks could be identified giving insight into approximate performance gain without them.

Due to technical limitations explained later, raster rendering was benchmarked using only the laptop setup with its dedicated NVIDIA graphics card. In this case, the difference between animation frames and complete redraws is inherited from image and polygon rendering. Consequently, only complete redraws were measured using the performance tool of Chromium's developer tools. On the other hand,

```
button.addEventListener('click', function() {
    var last = 0;
    var direction = 1;
    var observer = new PerformanceObserver(function(list) {
        var curr = list.getEntries()[0].duration - time;
        console.log(curr - last);
        last = curr;
    });
    observer.observe({entryTypes: ['mark', 'measure']});
    _this.getMap().on('postcompose', function() {
        performance.measure('draw');
    });
    c = 1;
    var moveFunc = function() {
        if (c < 10) {
            _this.getMap().once('postrender', moveFunc);
        }
        var center = _this.getMap().getView().getCenter();
        var zoomMod = Math.max(10 - _this.getMap().getView().getZoom(),
            0.1);
        _this.getMap().getView().setCenter([center[0] + 10000 * zoomMod *
            direction, center[1]]);
        c++;
        direction *= -1;
    }
    performance.mark('draw');
    var time = performance.now();
    moveFunc();
});
```

Program 3: Part of the OpenLayers control created for measuring complete redraws. Using OpenLayers' event mechanism, it waits for a compose task to finish, and calls the moving function again, until the required number of redraws are reached.

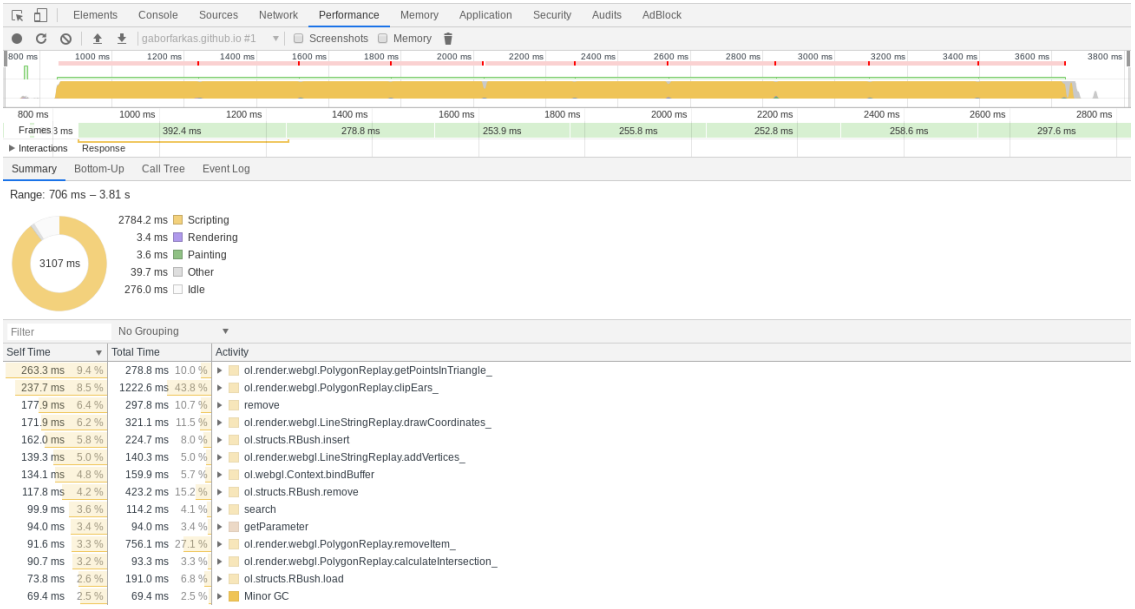


Figure 7: A performance benchmark showing the ratio of time spent on different browser tasks, and the most expensive function calls during the measured time interval.

as raster layers have higher memory demand than vectors, the memory footprint of different techniques was also measured. For this, a heap snapshot was created for every case, and the most demanding data structures were identified along with the total memory usage. Unfortunately, heap snapshots tend to require exponentially more memory than the measured application itself. In cases when loading heap snapshots crashed the browser, only the application's total memory usage was recorded.

4.4. Sample data

In case of every benchmark, a set of sample data had to be used. Three sets of sample data were used, carefully tailored to the respective benchmark. Two layer groups have been created for hardware accelerated rendering measurements. One is a thematic map showing a real world web mapping example, while the second one is a state of a GIS workflow, representing a typical GIS load.

The thematic map (Figure 8) consists of a choropleth polygon layer, a line layer with constant styling, and a thematic point layer. The thematics are population density for country polygons, population for settlement points, and major rivers represented as simple lines. It also has labels showing capitals. The world map shows data from the Natural Earth 1:50 000 000 dataset, loaded as GeoJSON, and styled on the client side. The layers were filtered in a desktop application in order to avoid lower performance from superfluous client side filtering.

The GIS group has three large, unfiltered layers in the extent of Hungary with raw styling. In this case, the problem was rather a simple visualization of data

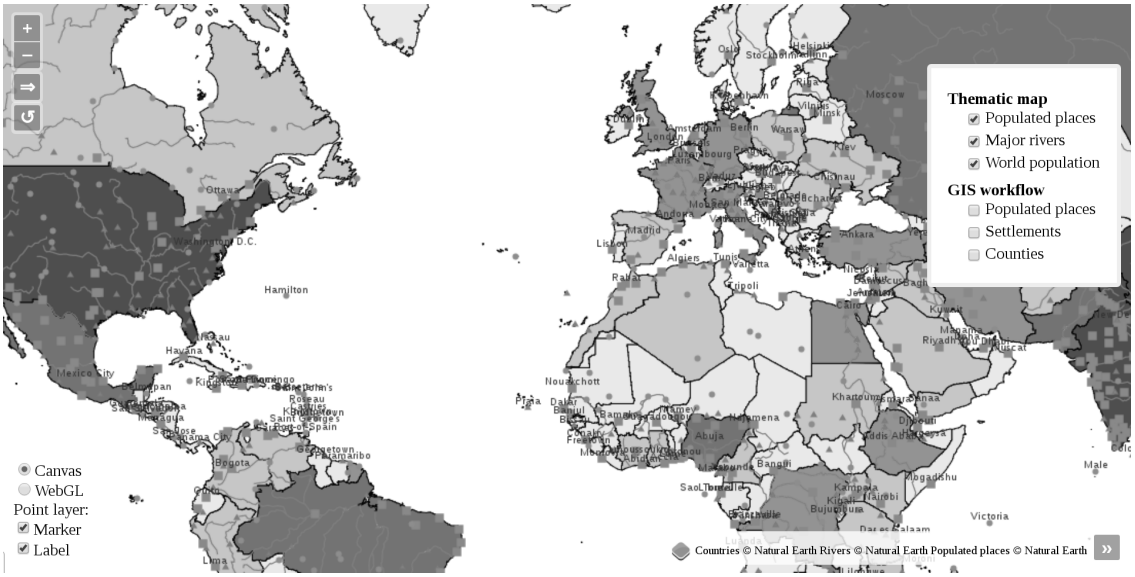


Figure 8: Thematic web map used for benchmarking a real world example of a web mapping application (Farkas, 2019).

produced in a GIS workflow, than a cartographic representation. Two polygon layers are raw OpenStreetMap (OSM) data, representing administrative areas on a settlement, and a county level. The third one is a point layer from the GeoNames Free Gazetteer Data dataset, containing settlements and other points of interests. The GIS workflow represented with the three layers is the following (Farkas, 2019):

1. Join the attributes of the OSM settlement layer to the GeoNames layer using the common field.

2. Extract settlements from the GeoNames layer by selecting features where the common field is not null.

3. Aggregate population of the settlements residing in each zone using the features of the OSM county layer as zones.

4. Calculate population density from the aggregated population data and the areas of county polygons.

Layers in the two groups have a fixed number of features and vertices (Table 3) affecting drawing speed. However, these properties are not guaranteed to be constant in a web map. Web mapping libraries can generalize vector layers on small scales. On large scales, they can filter out features outside the extent of the map by using a spatial index, such as an R-tree variant (Beckmann et al., 1990). In order to cover the effects of such optimization techniques, measurements were repeated on different zoom levels (Table 4).

The benchmarking application have only the most necessary functionality for measuring rendering performance. On the upper left side, there are two custom

	Type	Features	Vertices
Thematic group			
Countries	Polygon	241	99 566
Rivers	LineString	460	25 629
Settlements	Point	1249	1249
Capitals	Label	200	200
Total	–	2150	126 644
GIS group			
Settlements	Polygon	3170	428 803
Counties	Polygon	20	61 770
Places	Point	9839	9839
Total	–	13 029	500 412

Table 3: Feature and vertex counts of benchmarked layers (Farkas, 2019).

Zoom level	Scale	Center	Features	Vertices	
1	1:295 829 355	0; 0	2150	37 753	Thematic group
2	1:147 914 678	0; 0	2150	57 284	
3	1:73 957 339	0; 2 200 000	2118	78 940	
4	1:36 978 669	7 300 000; 5 500 000	1210	60 882	
5	1:18 489 335	3 400 000; 6 200 000	513	43 820	
8	1:2 311 167	2 200 000; 5 970 000	13 029	243 724	GIS group
9	1:1 155 583	2 100 000; 5 970 000	10 458	279 139	
10	1:577 792	2 050 000; 5 900 000	3541	128 998	
11	1:288 896	1 980 000; 5 920 000	1144	55 362	
12	1:144 448	1 980 000; 5 920 000	359	22 097	

Table 4: Zoom level dependent attributes of benchmarked layer groups. Center coordinates are in the map's projection (EPSG:3857). They were empirically defined to have optimal amount of features and vertices on every zoom level (Farkas, 2019).

controls for starting a measurement. Below the zoom controls, the upper button measures complete redraws, while the lower one measures animation speed. The output can be seen in the browser's console, and the web page must be reloaded between successive benchmarks. On the upper right side, a layer chooser can be expanded by clicking on the OpenLayers logo. In it, the user can choose between individual layers and layer groups. A layer group can be enabled by enabling every layer in the group. On the lower left side, there are checkboxes for markers and labels. Since the thematic group has a single layer for both of them, these checkboxes can be used to toggle them. Finally, the user can choose between the two rendering engines (Canvas and WebGL). Selecting an engine should be the first step, as changing it recreates the whole map.

In order to measure raster management techniques efficiently, two different sample rasters were used (Figure 9). One of them is a small Digital Elevation Model (DEM) from GRASS GIS's Spearfish60 example dataset. The other one is a multiband raster. It contains a multispectral Landsat 8 imagery of Baranya county using the red (R), green (G), blue (B), and near-infrared (NIR) spectral channels. The geographic resolution of RGB and NIR channels differ, therefore the NIR band with 60 meters resolution was resampled to match the other bands with 30 meters resolution.

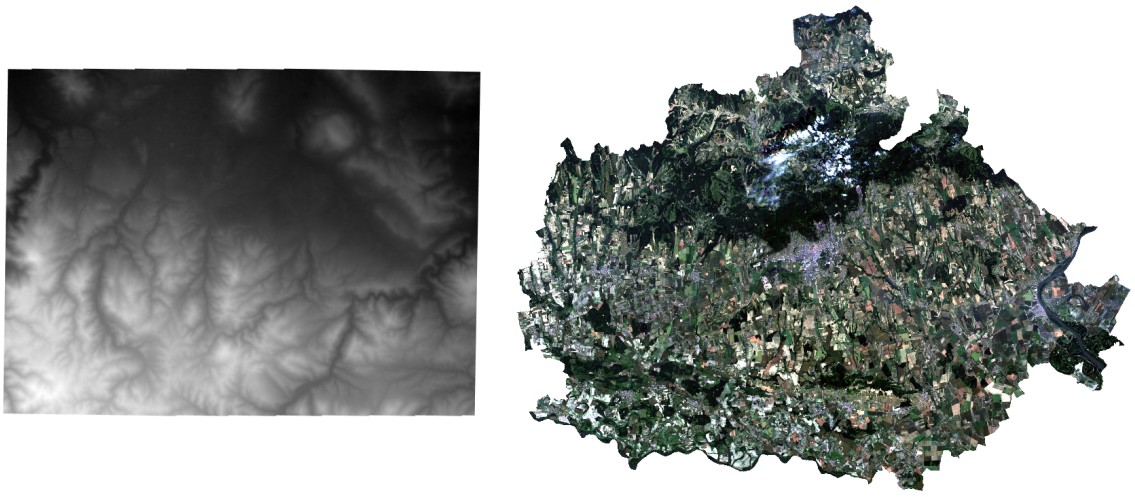


Figure 9: Sample rasters used for visualization and benchmarking. Both of the rasters are reprojected on the fly to the Web Mercator (EPSG:3857) projection. The Spearfish60 DEM (left) has a monochrome greyscale style, while the Baranya imagery (right) has an RGB style created from three corresponding bands (Farkas, 2020).

Overall, three variations of the two sample rasters were used (Table 5). The hexagonal variation of the Spearfish60 DEM is not a properly resampled hexagonal coverage, but original rectangular cells treated as hexagons. This causes geographically slightly distorted results, however, the visualization capabilities of a hexagonal raster can still be observed.

	Width	Height	Resolution	Format	Projection
Spearfish60	634	477	30 m × 30 m	Arc/Info ASCII Grid	EPSG:26713
Spearfish60 (hexagonal)	634	477	17 m	HexASCII	EPSG:26713
Baranya imagery	3172	2492	30 m × 30 m	GeoTIFF	EPSG:23700

Table 5: General attributes of benchmarked raster layers (Farkas, 2020). The resolution seems higher in case of the hexagonal variant, however the HexASCII format defines resolution as the length of a hexagonal cell's size.

5. The Ideal Candidate

In order to choose the best basis for a universal Web GIS client, numerous data visualization libraries were selected. From those libraries (Table 6), however, only a few passed the initial filters, making them subjects of further comparison. Initial filtering narrowed down possibilities to a group of ideal candidates by sorting out unsuitable libraries for a universal Web GIS client. Using this filtering system, each library was classified. The classification represents the strongest reason against a library (if there are multiple reasons), or its possible candidacy.

The strongest reason against a library was its proprietary nature. While proprietary libraries can be used freely, they have restrictive license conditions, and their source code are not disclosed. They would act as a black box in the system, which is unfeasible for several reasons. API changes and their general life cycle is unpredictable, and their features are fixed, therefore they can be hardly extended if they do not support an essential feature. These characteristics make those libraries unreliable as a basis of a universal Web GIS client. This criterion sorted out ArcGIS API for JavaScript and Google Maps JavaScript API, two prominent libraries, which could have been ideal candidates.

The second filter excluded abandoned libraries. Products without active development are also unreliable, since they get increasingly outdated as Web technology evolves. There are two reasons a library got an abandoned classification. Either its development has been officially ended (e.g. Processing.js), or there has not been any activity in its VCS repository for more than a year. For this reason, several popular, and back in the day cutting-edge libraries were excluded, like ka-Map, Modest Maps, or Polymaps. OpenLayers 2 is a special case, though. Since it was not abandoned in the time of the original survey (Appendix 2), it is treated as a candidate despite its currently abandoned status.

Libraries marked as general are data visualization tools capable of creating rich and interactive plots from raw data (Bostock et al., 2011). All of the listed data visualization libraries can create maps, although too general ones have only limited geospatial concepts. While they can be extended with web mapping features, it would require considerably more work than selecting a library with adequate geospatial capabilities.

Specific libraries, on the other hand, are proper web mapping libraries, although they are specifically tailored to a single use case. For example, CARTO.js is designed to work with CARTO tools, while Mapbox JS and Mapbox GL JS can be used efficiently in the Mapbox ecosystem. Kepler.js is a great spatial data visualization

Name	Version	Type	License	Dependency	Last release	Last activity	Classification
ArcGIS API for JavaScript*	4.11	Web mapping	Commercial	–	< 6 months	Unknown	Proprietary
Bing Maps AJAX Control*	8.0	Web mapping	Commercial	–	> Year	Unknown	Proprietary
CARTO.js	4.1.11	Web mapping	BSD 3-Clause	Leaflet	< 6 months	< Month	Specific
Cesium	1.57	Virtual globe	Apache 2.0	–	< Week	< Week	Candidate
D3	5.9.2	Data visualization	BSD 3-Clause	–	< 6 months	< Week	General
deck.gl	7.0.4	Web mapping	MIT	–	< Week	< Week	Candidate**
Google Maps JavaScript API*	3.36	Web mapping	Commercial	–	< 6 months	Unknown	Proprietary
HERE Maps API for JavaScript*	3.0.17.0	Web mapping	Commercial	–	< 6 months	Unknown	Proprietary
ka-Map	1.0	Web mapping	MIT	–	> Year	Unknown	Abandoned
Kartograph	0.8.2	Web mapping	GNU LGPL	Raphaël	> Year	> Year	Abandoned
kepler.gl	1.0.0-2	Data visualization	MIT	deck.gl	< Week	< Week	Specific
Leaflet	1.4.0	Web mapping	BSD 2-Clause	–	< 6 months	< Day	Candidate
Mapbox JS*	3.2.0	Web mapping	BSD 3-Clause	Leaflet	< 6 months	< Month	Specific
Mapbox GL JS*	0.54.0	Web mapping	BSD 3-Clause	–	< Month	< Day	Specific
MapQuery	0.1	Web mapping	MIT	OpenLayers 2	> Year	> Year	Abandoned
MapQuest.js*	1.3.2	Web mapping	Commercial	Leaflet	> Year	Unknown	Proprietary
MapQuest-GL.js*	0.4.0	Web mapping	Commercial	–	> Year	Unknown	Proprietary
Modest Maps	3.3.6	Web mapping	BSD	–	> Year	> Year	Abandoned
NASA Web World Wind	0.9.0	Virtual globe	Apache 2.0	–	> Year	< Month	Candidate
OpenLayers 2	2.13.1	Web mapping	BSD 2-Clause	–	> Year	> Year	Abandoned
OpenLayers	5.3.0	Web mapping	BSD 2-Clause	–	< 6 months	< Week	Candidate
OpenScales	2.2	Web mapping	GNU LGPL	–	> Year	> Year	Abandoned
OpenStreetMap iD	2.14.3	Web mapping	ISC	D3	< 6 months	< Day	Specific
OpenWebGlobe	Unknown	Virtual globe	MIT	–	No release	> Year	Abandoned
Polymaps	2.5.1	Web mapping	BSD 3-Clause	–	> Year	> Year	Abandoned
Processing.js	1.6.6	Data visualization	MIT	–	> Year	< 6 months	Abandoned
Raphaël	2.2.8	Data visualization	MIT	–	< 6 months	< 6 months	General
Tangram	0.18.1	Web mapping	MIT	Leaflet	< Month	< Day	Extension
WebGL Earth	2.4.2	Virtual globe	GNU GPLv3	Cesium	> Year	< 6 months	Extension

*Requires an API key.

**Could have been a candidate.

Table 6: List of considered JavaScript libraries capable of spatial data visualization. Extended version of the original list (Farkas, 2017a). Relative times are compared to the date of revision: 4th May, 2019.

Name	Change	Old value
CARTO.js	Name	CartoDB.js
deck.gl	New library	–
kepler.gl	New library	–
MapQuest.js	Name	MapQuest JavaScript Maps API
MapQuest-GL.js	New library	–
OpenLayers 2	Classification	Candidate
OpenLayers	Name	OpenLayers 3
Processing.js	Classification	General
Tangram	New library	–

Table 7: Important changes in libraries between the two surveys taken in 2016 and 2019. The column "Change" represents the column in the two tables (Table 6 and Appendix 2), where the change happened. For MapQuest.js, the whole line changed, since MapQuest discontinued its old service, and replaced it with a new one.

tool, although it is designed for visualizing millions of points in multiple ways for exploratory data analysis (Behrens, 1997).

Finally, libraries marked as extensions were filtered out. These libraries can be considered as plugins rather than standalone libraries, extending the capabilities of their main dependency. Tangram is a 3D rendering engine build for Leaflet, and therefore a Web GIS client using Tangram must be built using Leaflet's core classes and methods. WebGL Earth is a facade (Shalloway & Trott, 2002) for Cesium, making it easier to use for simple virtual globes. Unfortunately, it does not expose core Cesium functionalities, limiting the number of usable features.

The candidates according to the revised survey are Cesium, deck.gl, Leaflet, NASA Web World Wind, and OpenLayers. However, the original list (Appendix 2) did not take deck.gl into account. Furthermore, OpenLayers 2 was not abandoned back then (Table 7). Since this part of the study established the choice of a basis for a universal Web GIS client, and that choice was made in 2016, instead of deck.gl, OpenLayers 2 is listed as a candidate. This makes the final list of candidates Cesium, Leaflet, NASA Web World Wind, OpenLayers 2, and OpenLayers.

It is important to note, that two of the candidates are virtual globes. However, both Cesium and NASA Web World Wind have strong geospatial foundations, capable of rendering 2D maps, and have numerous GIS features required for a Web GIS client. OpenLayers, the successor of OpenLayers 2, is a mature, robust, and feature-rich library, which makes it an excellent candidate. While OpenLayers 2 has became abandoned, it reached a complete status, making it an appropriate reference for the current status of OpenLayers. Finally, while Leaflet is a lightweight web mapping library, its superior extensibility has given space for a vast number of third party plugins. By considering those extensions, Leaflet can be an ideal basis for a Web GIS client.

Feature group	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Rendering	80%	40%	60%	40%	60%
Formats	65%	62%	53%	82%	76%
Database	0%	8%	0%	17%	0%
Data	32%	30%	18%	34%	44%
Projection	63%	50%	75%	63%	88%
Interaction	33%	50%	33%	83%	72%
Representation	22%	44%	33%	56%	56%
Average	41%	41%	34%	54%	56%

Table 8: GIS feature coverage of candidate libraries (Farkas, 2017a). Detailed results can be seen in Appendix 1.

5.1. Competitive analysis

The overall results of the competitive analysis (Table 8) did not show extraordinary differences between supported features in candidate libraries. Both OpenLayers 2 and OpenLayers outperformed the rest of the candidates, which is mainly due to their development philosophy. OpenLayers libraries have always been created with GIS considerations in mind. Their internal structure resembles desktop GIS software', allowing users to create rich web maps with GIS capabilities.

Leaflet and Cesium both achieved second place, not far behind OpenLayers libraries. Leaflet scored lower due to its lightweight nature. The core library only supports a handful of features, while the rest of the GIS functionalities are provided by third-party extensions. This characteristic makes Leaflet easy to learn, but also a capable basis, when mastered. Some dangers are originating from this phenomenon, targeting reliability, though. As mentioned before, plugins are often maintained by a small number of developers independent of core development. Therefore, there is no guarantee for a plugin to be available for the next release of the core library, and there can be delays between the release of a new Leaflet version, and the release of updated extensions.

Cesium scoring as high as Leaflet was outstanding for a virtual globe. As it can be seen in the category scores, it lacks web mapping libraries' superior 2D interactivity and cartographic representation capabilities, but it makes up for them with its powerful rendering engine, capable of both 2D and 3D spatial visualizations.

While NASA Web World Wind had the least implemented features, its total coverage was still promising. The library was in an initial, rudimentary state, but still had decent support in most of the categories. It could be sorted out at this point compared to other examined technologies, nevertheless, its evolution should be monitored in the future. As it is a Web port of the mature and popular desktop virtual globe NASA World Wind, it will likely catch up to other candidates eventually.

In the rendering category, Cesium outperformed every other candidate due to its capable rendering engine. It could do anything besides raster visualization, which

Format	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Vector	50%	60%	50%	90%	90%
Raster	17%	17%	17%	17%	17%
Image	100%	100%	100%	100%	100%
Tile service	67%	32%	50%	100%	83%
Average	65%	62%	53%	82%	76%

Table 9: Data exchange format support of candidate libraries (Farkas, 2017a). Detailed results can be seen in Appendix 1.

is a feature lacking from every library. NASA Web World Wind and OpenLayers also scored high, although for different reasons. NASA Web World Wind has a fully implemented WebGL renderer, capable of hardware accelerating 2D vector visualization. OpenLayers, on the other hand, had partial WebGL support (it could render images and vector points), and partial support for blending different layers. That is, blending WebGL layers could be achieved with some programming, not directly supported by the API. On the other hand, its less performant HTML5 Canvas engine supported blending layers from the API.

Considering format handling (Table 9), OpenLayers 2 had the most implemented features, although all of the libraries showed decent coverage. Image format handling showed the greatest coverage, since candidates did not only supported popular image formats (PNG, JPEG), but all of them could use WMS outputs of spatial servers. Specific tile formats, on the other hand, showed more variable support. Some of the popular services like WMTS, OpenStreetMap's slippy map format, Bing Maps, and TMS could be used with most of the libraries. Others, like ArcGIS REST API, or Google Maps had lower coverage. While the low support of Google Maps is controversial, since it is one of the most widely used base maps, it is due to its license conditions, forcing libraries to abandon core support. Where it is supported, it is either a legacy version, or a third party plugin not strictly obeying to license conditions.

In the case of vector formats, GeoJSON and KML had the greatest support. It is worth noting, that the binary ESRI Shapefile format was supported by NASA Web World Wind out of the box. Since there are universal libraries for parsing Shapefiles, other libraries could still make a use of it. Communication with WFS servers was only supported by web mapping libraries. It is unfortunate, as using WFS-T is one of the few ways of saving spatial data from a Web client into a database. Other ways include using proprietary services and creating custom server-side logic.

Raster format support was uniform amongst the candidates. Since they do not have any form of raster management, they neither could parse raster files. The only exception was GeoTIFF. Similarly to ESRI Shapefile, there is a universal library for parsing raw GeoTIFF data and converting it to a binary array. This way, every library could be able to overlay images coming from GeoTIFF files with some work.

All of the libraries performed poorly in regards database-related functionality (Table 10). The connection subcategory is understandable, since browsers still

Database	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Connection	0%	0%	0%	0%	0%
Functionality	0%	17%	0%	33%	0%
Average	0%	8%	0%	17%	0%

Table 10: Database realted feature support of the candidate libraries (Farkas, 2017a). Detailed results can be seen in Appendix 1.

Data	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Pre-process	50%	25%	25%	19%	63%
Conversion	0%	0%	0%	0%	0%
Manipulation	50%	67%	25%	83%	67%
Analysis	13%	19%	13%	25%	25%
Average	34%	32%	20%	34%	46%

Table 11: Data management feature support of candidate libraries (Farkas, 2017a). Detailed results can be seen in Appendix 1.

cannot connect to databases directly. In order to establish such a connection, a server-side component must be present mediating between clients and the database. Such a component can be a WFS service (WFS-T for writing back data), which is DBMS independent by design, and was already checked with data exchange formats.

The database related functionality could be able to score a few points for some of the libraries, though. These functionalities are not dependent on DBMS components but implemented locally. By using the IndexedDB API (Mehta et al., 2015), libraries could have implemented a data storing and retrieval mechanism for reducing memory footprint. By not having such DBMS functionality, along with a query language for users to interact, candidates could be only given scores based on other means of querying and filtering vector data. Since Leaflet had a way to filter features for display, it partially fulfilled this requirement. Furthermore, OpenLayers 2 had both querying and filtering capabilities.

Features related to data management (Table 11) had similar average coverages, but diverse subcategory coverages. In preprocessing data, OpenLayers and Cesium excelled with their numerous optimization techniques. These techniques are mostly related to vector data, like geometry simplification according to scale, or handling per-vertex Z and M (measurement) coordinates. They could also transform vector layers to the map's projection, although in OpenLayers, these transformations were permanent rather than on the fly. On the other hand, OpenLayers used a spatial index on vector data, which was unique amongst candidate libraries. Unfortunately, geometry validation and automatic attribute table generation were not implemented in any of the libraries. NASA Web World Wind was especially weak in this subcategory, since it discarded feature attributes, whose support is essential in GIS applications.

Data conversion had a uniform, 0% coverage amongst libraries. The lack of raster to vector conversion support is understandable, since raster management was

not supported either by any of them. On the other hand, traces were found of other functionalities in some libraries. For example, web mapping libraries could generate heat maps from point data, which touches both interpolation and vector to raster conversion. Furthermore, OpenLayers was able to generate and cache textures created from vector data for better performance. However, these functionality were not considered even for a partial fulfillment, since those are specialized functionality related to data visualization.

In terms of data manipulation techniques, candidates performed much better. Layer management capabilities (i.e. adding and removing layers, changing layer order) were mostly supported. Updating feature attributes and geometries – which are also essential for analysis – were mostly implemented, although virtual globes did not allow to modify geometries in place. Similarly to attribute tables, a field calculator tool was neither included in any of the libraries.

Storing the type of layers was only done by OpenLayers 2. This is an interesting feature, since one could argue, every layer object has an inherent type, as they are instantiated from their constructors. However, in JavaScript, the `typeof` operator returns `object` for every class. Consequently, if layers do not store their types, users must do `instanceof` comparisons with every possible layer type in the library. Alternatively, users can access the `Function.name` attribute of constructors, which is not necessarily returning correct values due to minifying processes used for compressing libraries (Vasilescu et al., 2017).

From data analysis features, no significant support could be observed. Basic geoprocessing and topological analysis could be implemented in every candidate using general libraries, such as JavaScript Topology Suite (JSTS) and Turf (Schernthanner et al., 2017; Kulawiak et al., 2019). From the other features, OpenLayers had support for modifying image layers on the fly according to a user-defined function working on a per-pixel or per-image basis. This technique could be used for applying a convolution matrix, however, the matrix's logic must have been programmed by the user, and the result would not have been permanent. A more important feature, writing WPS requests, was supported by all of the web mapping libraries. With this functionality, a wide variety of spatial analyses can be conducted using a spatial server with WPS capabilities.

Considering projection support, all of the libraries performed well. They could transform vector layers between at least WGS84 and Web Mercator. Virtual globes could additionally warp image layers. Additionally, OpenLayers could also do this, making it unique amongst web mapping libraries. Other projections could also be handled by the candidates. While virtual globes were restricted to a set of hard-coded projections, web mapping libraries could use any with the JavaScript port of PROJ.4.

The number of supported interactions strongly differed between web mapping libraries and virtual globes. Web mapping libraries supported interactive vector manipulation (e.g. drawing, modifying, snapping features), while virtual globes resorted to visualizing features. Interactive tools for querying and selecting features were also dominantly present in web mapping libraries, just like interactive

Representation	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Styling	67%	67%	67%	67%	83%
Cartographic elements	0%	33%	17%	50%	42%
Average	22%	44%	33%	56%	56%

Table 12: Representation feature support of candidate libraries (Farkas, 2017a). Detailed results can be seen in Appendix 1.

Library	Size (KB)	LLOC	CC/F	EF	ALC_{JS}
Cesium	11 420	292 500	2.08	911	51.27
Leaflet	162	3639	2.00	200	18.47
NASA Web World Wind	1452	13 037	2.50	187	30.64
OpenLayers 2	872	23 702	2.82	207	36.46
OpenLayers	499	21 451	2.36	223	33.90

Table 13: Static software metrics of candidate libraries (Farkas, 2017a). CC/F stands for per function cyclomatic complexity.

measurements.

Regarding the map state, most of the libraries supported the basic interactions of panning, zooming, and rotating the view. Leaflet and OpenLayers 2 were the only exceptions, not having rotating capabilities. From the compared libraries, only Cesium had core support for changing the temporal dimension, allowing users to visualize spatiotemporal datasets out of the box. On the other hand, OpenLayers 2 and Leaflet both had third party extensions for the same purpose.

The representation capabilities of candidates (Table 12) showed expected results. Styling vector layers and creating thematic maps were supported by all of the libraries. Styling raster, on the other hand, was only partially supported by OpenLayers, as it could manipulate image layers on the fly. Cartographic elements were mostly supported by web mapping libraries. Creating scale bars, graticules, and overview maps were achievable with all of the web mapping libraries, while legends and arbitrary text boxes were not supported by any of the candidates. As an interesting fact, NASA Web World Wind offered a built-in north arrow, inherited from its desktop version's user interface (UI) design.

5.2. Metrical results

Upon investigating further using static software metrics (Table 13), no surprising results could be found. Sizes were not related directly to LLOC values, due to the different compression tools used by different developer teams, and the number of assets included. Such assets were typically Cascading Stylesheet (CSS) files and images bundled with production versions of the libraries. LLOC values, per function cyclomatic complexity, and the number of exposed functions were able to describe several aspects of candidate libraries. Furthermore, ALC_{JS} values matched experienced difficulties with the candidates.

Property	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Documentation	Good	Good	Decent	Very good	Very good
Community	Good	Very good	Poor	Good	Good
Contributors	89 (29)	236 (8)	12 (5)	98 (16)	154 (27)
Open issues	409 (25%)	225 (7%)	40 (56%)	383 (64%)	414 (21%)
RF	28	68	N/A	32	24

Table 14: Non-code metrics of candidate libraries (Farkas, 2017a). *RF* stands for release frequency, and shows the average number of days between two successive release.

Leaflet is the smallest library, although it has as many exposed functions as other web mapping libraries. Its complexity is low, making it easy to learn and develop. Its slight learning curve makes it very popular for beginners and developers not familiar with GIS. NASA Web World Wind is the smallest amongst the two virtual globes. Its learning curve is significantly steeper, since it is a virtual globe, having more complex dynamics. It is still easy to use for visualizing 3D phenomena. Its low number of exposed functions represents its maturity, as it was in an early development stage when the survey was conducted.

OpenLayers libraries and Cesium could be called the big players in the group. They are harder to learn into, even harder to master and develop. They are mature, big, robust libraries, with a wide variety of functionality to offer. Cesium is the largest project, having a codebase comparable to mature desktop solutions. On the other hand, it has a very low per function complexity, showing that developers put a lot of effort into creating clean, transparent code. Its steep learning curve is in ratio with the complexity of a virtual globe capable of visualizing spatiotemporal 3D phenomena.

Other, non-code metrics (Table 14) shed some light on the development difficulty with candidates. In the table, major contributors are shown between parenthesis next to the total number of contributors in a project. Next to the number of open issues, their ratio to the total number of issues is shown in a similar fashion.

Most of the candidates had sufficient documentation, with which users can start to build applications quickly. While Cesium, OpenLayers 2, and OpenLayers excelled in examples, Leaflet had very thorough tutorials for basic use cases. OpenLayers 2 had an outstanding number of examples (210), while OpenLayers had the most tutorials (23). As an exception, NASA Web World Wind only had a very few of both tutorials and examples, making it harder to learn into.

Community metrics showed a similar picture. In this category, Leaflet excelled with not only the highest ratio of answered questions (80%), but also with the highest total number of questions on StackExchange forums (5447). Unfortunately, NASA Web World Wind did not have a measurable community, probably due to being young and unadvertised. The raw numbers behind ordinal values look promising, although they should be taken with care. By putting them in contrast with the popular open-source and community-developed data visualization library D3, all of them seem very low (Figure 10).

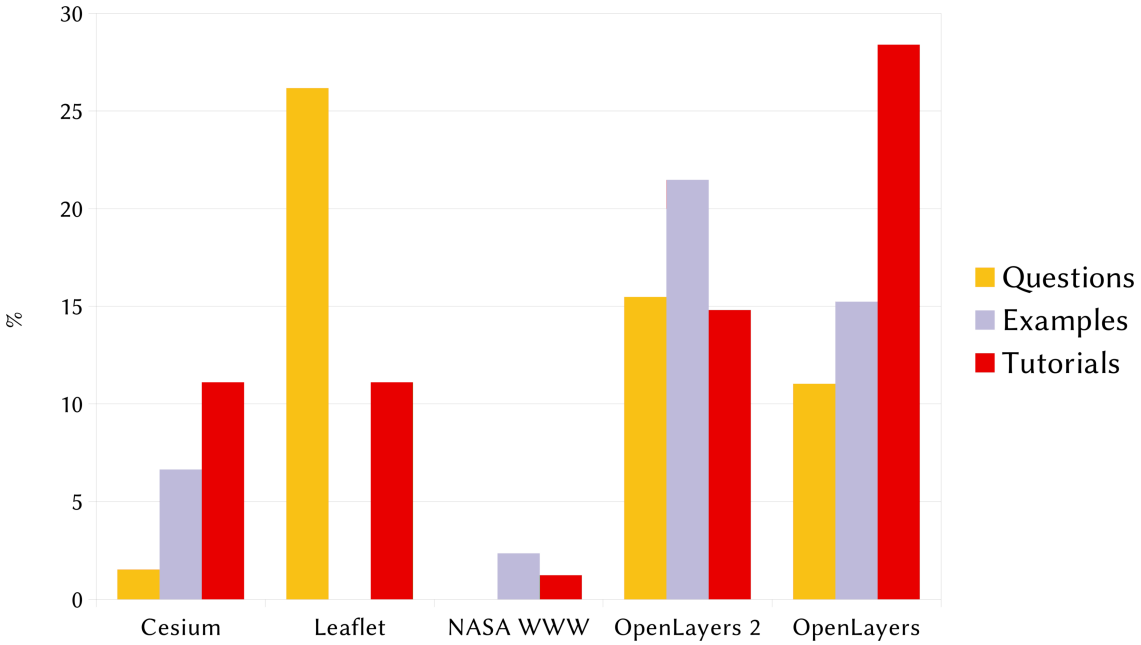


Figure 10: The number of tutorials, examples, and answered questions on StackExchange forums of the five candidates. All of the numbers are normalized with D3's statistics, and represented in percentage values.

Developer statistics showed that while Leaflet had the most contributors, it also had the smaller core developer group. In this category, Cesium and OpenLayers excelled with a high number of major contributors, implying their long term stability. Seemingly, the number of core developers was slightly related to the size of the project. The number of open issues also favored the aforementioned libraries, since they had low ratios to the total number of issues. On the other hand, Leaflet highly outperformed all the other candidates in terms of open issues. The candidates' release frequency were also showing stability, except for NASA Web World Wind, whose releases could not be tracked, since it was using a private development system.

5.2.1. Approximate learning curve for JavaScript

A metric is only useful when it shows meaningful results, properly approximating the truth. This is especially true for complex metrics, making regressions between statistical values and soft phenomena. The perceived learning curve of a library is a very soft phenomenon, since it varies between individuals, and other, hardly enumerable characteristics of the given technology. When such an empirical phenomenon is tried to be approximated numerically, statistics allows for manipulating input values to give correct results for a sample. In order to avoid such a pitfall, ALC_{JS} was validated against a larger sample with known learning difficulties.

First, four new JavaScript libraries and a basic function (Program 4) were introduced, and ranked by their difficulties to use according to the author's experience

Library	Rank	Label
Cesium	1	Very Hard
OpenLayers 2	2	Hard
OpenLayers	3	Hard
Google Maps API	4	Hard
NASA Web World Wind	5	Moderate
D3	6	Moderate
Leaflet	7	Easy
jQuery	8	Easy
GeoJSON Lite	9	Easy
addNums	10	Basic

Table 15: Ranking of libraries used for validation according to their experienced learning curves. Libraries were also grouped in ordinal categories. Members of a single category did not have significant differences in difficulty to use.

(Table 15). Some of the libraries, like Google Maps API and D3 were chosen for a later validation phase. Others, like jQuery and GeoJSON Lite, were included to sufficiently represent the lower spectrum of learning curves, since web mapping libraries and virtual globes are usually harder to use.

```
var addNums = function(a, b) {
    return a + b;
};
```

Program 4: A simple JavaScript function treated as a library for validating $\mathrm{ALC}_{\mathrm{JS}}$.

From the newly introduced and previously unmentioned technologies, Google Maps API is a closed source JavaScript library for creating web mapping applications on top of the publicly available Google Maps application. Due to its popularity, tutorials, and examples, it is very easy to get started. However, its capabilities extend far beyond the surface, making it hard to sufficiently understand for creating rich applications.

The jQuery library is a popular framework, well-known for its steep learning curve and user-friendliness (Lindley, 2009). Its main purpose is to facade some of the more advanced functionalities of JavaScript, making it easier to use for beginners. It is also popular amongst professionals, since it makes developing web pages faster when the overhead is acceptable. While the library primarily targets general use cases, like sending AJAX requests, selecting DOM elements, or animating content, there are many specialized use cases. Even simple web maps can be created with it (Król & Szomorowa, 2015).

GeoJSON Lite is a very simple library written for parsing, serializing, and validating GeoJSON data. It has an internal feature and geometry class system, which is used by its random feature generator. It can create random points, lines, and polygons, fill them up with random attributes and output the resulting GeoJSON

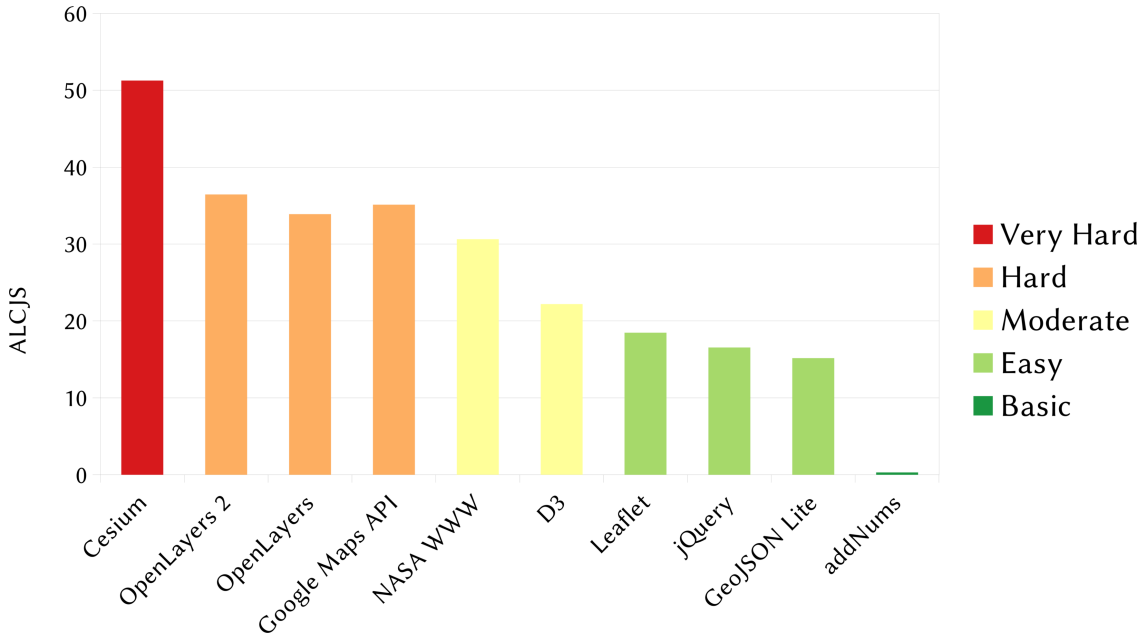


Figure 11: ALC_{JS} values of libraries used for validation ordered by their empirical ranking. Colors indicate different ordinal difficulty categories.

for later use. While the functionalities or the library are limited, users still need to understand how parametrization and different methods work.

By correlating ALC_{JS} with the empirical ranking of libraries (Figure 11), the metric follows experienced difficulty feasibly. The `addNums` function got a score of 0.3, little above 0, which cannot be reached due to its logarithmic components' asymptotic nature (Equation 2). The only difference between ALC_{JS} scores and ranks was between OpenLayers and Google Maps API, where the difference was substantial. Ordinal categories matched ALC_{JS} scores even better. Basic, easy, hard, and very hard categories were very easy to differentiate by their scores. Libraries with moderate difficulty showed a smooth transition between easy and hard ones.

In order to incorporate an expert survey, a study was used, which recorded the development process of a web mapping application with OpenLayers 2, Leaflet, D3, and Google Maps API (Roth et al., 2014). That study recorded developer moods during four independent development processes aiming towards the same goal. Moods were categorized as positive, neutral, and negative, and summarized in the end.

When comparing ALC_{JS} scores with experienced moods, positive and negative trends mostly follow the libraries' score. The only difference is with Google Maps API and D3. This exception can be considered a phenomenon rather than an error, though. Since D3 is a general data visualization library, while the rest of the technologies are web mapping libraries, D3 requires more work in order to create a web mapping application. Furthermore, most of its examples cover different intuitive charting solutions, making it harder to find existing solutions for web

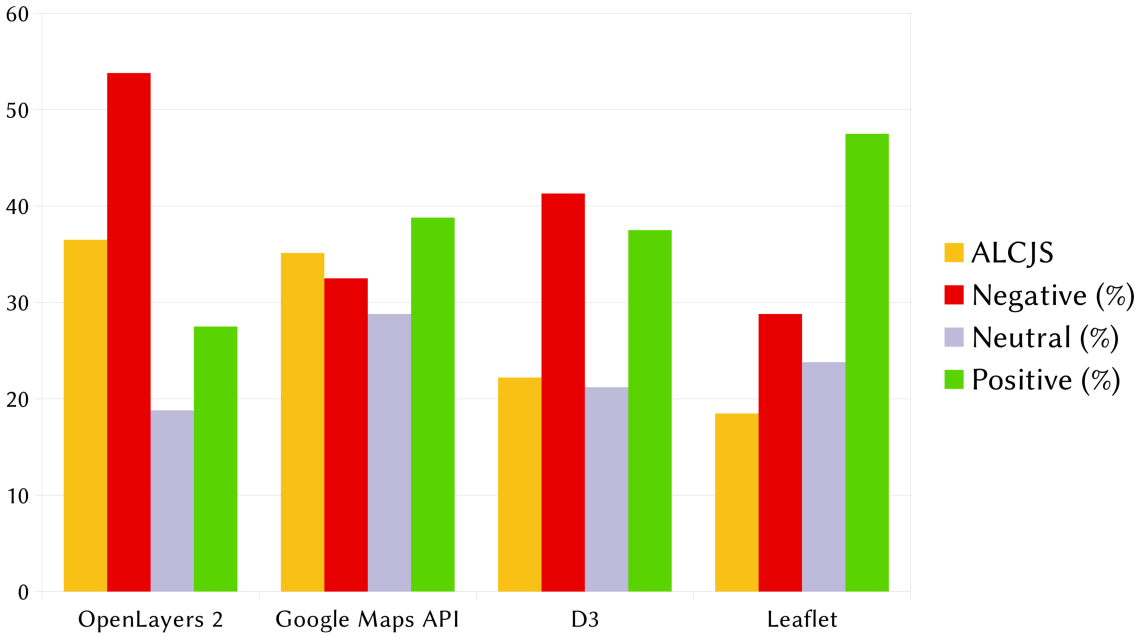


Figure 12: Categorized and summarized moods from the diary study created by Roth et al. (2014) along with the libraries' ALC_{JS} scores. ALC_{JS} scores do not match the percentage scale of moods, therefore values should not be directly compared. Only trends should be observed.

mapping problems.

While the validation is not conclusive, it shows that ALC_{JS} has merits, and can be used for approximating learning difficulties. ALC_{JS} scores should be interpreted along with other, non-code metrics, and should be used only to exclude outliers. That is, libraries which could not be learned easily. Furthermore, the metric could be refined with other object-oriented metrics, or generalized for other languages. It is certain, ALC_{JS} should be validated more thoroughly with an expert survey including more libraries. Before then, it should not be used alone for making any decisions.

5.3. Selecting a candidate

According to candidates' feature coverage and metrical results, they could be narrowed down to two technologies: Cesium and OpenLayers. While OpenLayers has a slighter learning curve, it held little importance, since the author was familiar with both of the libraries. On the other hand, the larger community behind OpenLayers had a minor influence on the decision.

Since the two libraries were in more or less a tie, the supported features were reinvestigated, and crucial features were identified. Those features were considered crucial, which were absolutely necessary for a general Web GIS application to work. Three such deficiencies were identified in the libraries; the lack of general projection support in Cesium, the lack of full hardware acceleration in OpenLayers, and the lack of raster management in both of them.

Estimations were made for implementing the two unique lack of features. In Cesium, there was a skeleton (an empty class) for custom projections in the API. However, it seemed like projections were tightly coupled to the rendering engine. Cesium offered a fixed number of hard-coded projections and could change the map view between a 3D representation, and a 2D representation in the given projection with a smooth transition. The projection classes seemed to be quite different, making it hard to generalize. While such an implementation was considered possible, the required time could not be estimated due to possible pitfalls.

OpenLayers already had a WebGL engine for drawing hardware accelerated point and image layers. Since the basic structure of the engine was given, it seemed an easier way to implement line, polygon, circle, and text rendering. It was considered, that some essential modifications will be needed in core classes due to possible pitfalls, still, a rough estimation could be created for implementing the missing rendering mechanisms in a year. Mainly for this reason, OpenLayers was chosen as the basis for a universal Web GIS.

This way, the next chapters of the thesis concentrate on the implementation of crucial features in OpenLayers. First, hardware accelerated rendering methodologies are presented, then some raster management techniques. These implementations aimed to provide basic mechanisms for a Web GIS application, therefore they can be further optimized.

5.4. The structure of OpenLayers

In order for this thesis to make complete sense, the basic structure of OpenLayers needs to be presented. The library strives for a GIS-like structure, still it includes the typical web mapping components. Just like in any other web mapping libraries, there is a basic set of classes responsible for creating a map. The central class of the library is `ol.Map`, and the rest of the basic classes can connect to it with a 1:1 or a 1:N relationship. OpenLayers' base object structure is peculiar, as every important class is the child of the `ol.Object` class, which inherits from the `ol.Observable` class (Figure 13). As the name suggests, this internal base class – carefully hidden from the everyday user – takes care of event handling. This is a very strong point of OpenLayers, as it successfully implements a DOM style event mechanism, providing an easy to use interface for asynchronous coding.

While the map object is responsible for holding the whole application together, OpenLayers is fully modular. Its different components are provided as parameters and can be changed at will. For example, the map takes an `ol.View` instance, which is responsible for storing the current state of the view (e.g. center, zoom level, projection). If changed, the map continues to run with the new view object.

A very important group of classes in OpenLayers are layers. Their structure is twofold. Layer classes only define how their content gets handled internally. There is a different class for image layers, tile layers, and vector layers, as they need different internal logic. A tile layer, for example, needs to know about tile layouts, needs the capability to cache tiles, and needs to be able to provide appropriate

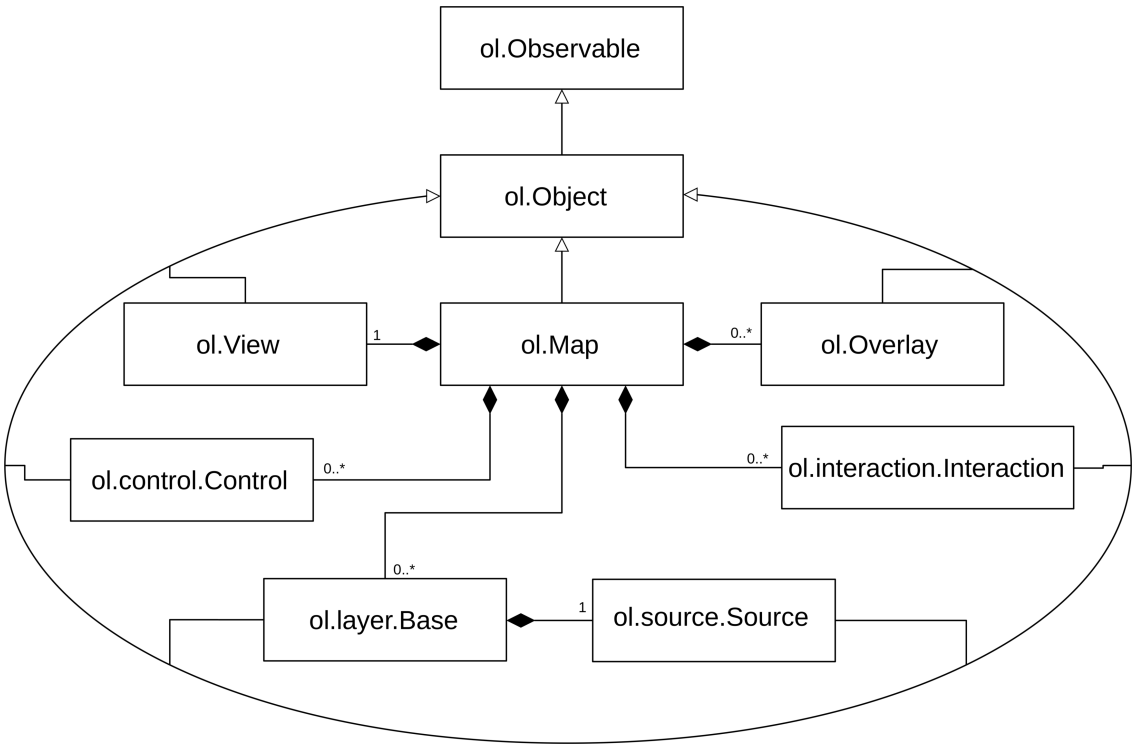


Figure 13: UML diagram of base constructors in the OpenLayers library. Every class inherits from the `ol.Observable` class, which provides the same event mechanism for every OpenLayers object. This means, events are used consistently by different classes of the library (Farkas, 2016).

tiles to the renderer. Data management is outsourced to another group of classes: the sources. Source objects are provided to layers as parameters, and take care of downloading, parsing, and abstracting different data sources. These source classes are specialized for every different format OpenLayers can handle. There is a source for OpenStreetMap, for ArcGIS REST API servers, for WMS servers, for vector datasets, and many other types of input.

Styles form another group of classes in OpenLayers. Style objects can define vector styling rules. There are different style classes for polygon fills, polygon and line string strokes, point symbolizers, and labels. These styles are provided only for vector layers, which provides them to the renderer on demand.

For an efficient raster management implementation, not only raster related base classes are essential. While they must be present to hold matrix data, and make different operations, they are insufficient by themselves. For displaying rasters, a raster layer class, raster source classes (e.g. for GeoTIFF and ArcInfo ASCII Grid), and raster styles are also necessary. At least three styles are essential; a monochrome for 8-bit styling, a pseudocolor for categorized and graduated smybology and an RGB for composites.

Another important aspect of OpenLayers from the perspective of this thesis is rendering. There are two rendering engines in the library, one for HTML5 Canvas rendering, and one for hardware accelerated WebGL rendering. Both of the

renderers are created with the same structure, although they are not interoperable (at least before version 6.0). The user must choose to use one of them exclusively. From a developer standpoint, they are easier to develop, validate, and compare, since they use the same mechanisms (Figure 14).

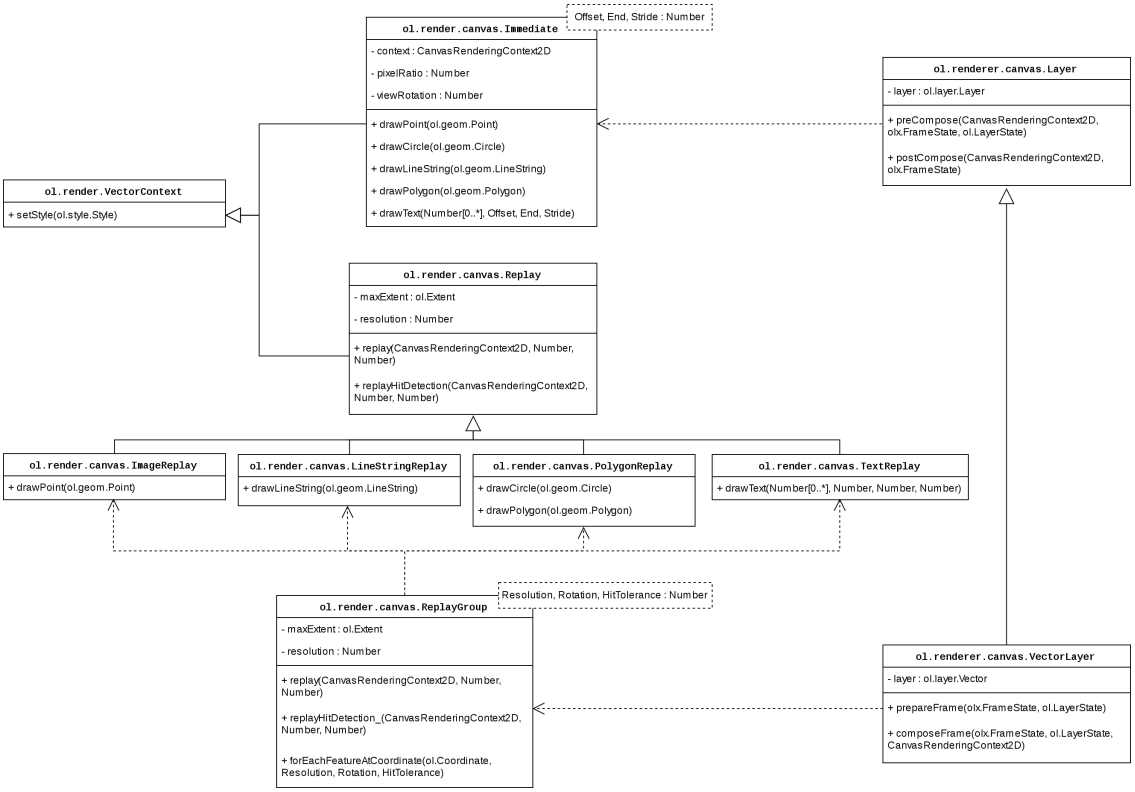


Figure 14: UML diagram of the internal mechanism of OpenLayers' HTML5 Canvas rendering engine. Similar classes build up the WebGL renderer. There are some minor, technical differences, although the same principles apply (Farkas, 2019). A larger image can be found in Appendix 3.

Layers and renderers are connected in the map object. On instantiation, the map creates high-level renderer objects, responsible for creating an initial view. Then it assigns renderer classes to layer types. Every different layer type has a renderer, which is responsible for drawing entities on the map. Entities can be images, tiles, or individual vector features.

There are two rendering modes in both of the engines. The immediate mode draws every layer synchronously and immediately on the map canvas. Since it cannot cache and does not have any other optimization possibilities, it is only used on rare occasions. The other mode is called replay. In replaying mode, the renderer collects styled entities, sorts them based on their Z level values, creates different caches, and draws with a better performance. Similarly to CSS, Z level can be used to override the default drawing order of different elements.

Caching is one of the most significant differences in the two rendering engines. The HTML5 Canvas technology is based on instructions. There are different drawing methods offered by the built-in API for different drawing tasks, similarly to the

Logo language. For example, there is an instruction to start a line, to add vertices, to generate a stroke, and to fill a closed shape. Images can also be inserted with other instructions. Consequently, the Canvas renderer can only cache these instructions, providing a slightly better performance, when it only has to replay previous instructions. On the other hand, the WebGL renderer uses buffers, sent to the GPU, and processed by it according to GLSL programs. When buffers are cached, the replay mechanism can provide a significant performance boost over regenerating them.

The two phases accomplishing different tasks in OpenLayers are preparing–, and composing a frame. In the prepare phase, the required preprocessing steps are done, including caching. In the compose phase, the prepared data are drawn on the map canvas. During a complete redraw, both of the phases are executed once, while during animation or interaction, only composing is called repeatedly. It is very important to distinguish between these two cases, since an animation frame is created faster, but cannot introduce new vector features, which were previously outside of the view's extent.

In order to implement a complete WebGL engine, several gaps need to be filled. The WebGL rendering engine of OpenLayers only supported image, tile, and point rendering. Since point rendering is done by the same class as image rendering, there were only two classes implemented. On the top of the existing structure, a line string replay, a polygon replay, and a text replay needed to be included. Also, a circle replay turned out to be another requirement, as in WebGL circles should not be treated as special polygons.

In addition, for raster rendering, raster related renderer classes were also needed. These classes are responsible for applying styles on raw raster data (i.e. matrices), and either creating textures from styled rasters, or rendering cells as polygons. They were also used for caching, which included pyramid building, loading cells into R-trees, and caching buffers.

6. Hardware Accelerated Vector Rendering

In order to implement the first big requirement for a general Web GIS client, current support needed to be analyzed, and the inner workings of hardware acceleration was to be understood. Before this modification, OpenLayers supported image and point rendering via WebGL. This support manifested as an image renderer, since points were first rendered through the Canvas API, cached, and overlaid on the map canvas as textures. That is, OpenLayers was able to render textures with its hardware accelerated renderer.

For rendering other types of cartographic elements, line strings, polygons, and labels needed to be supported. Since the implementation was targeting basic support, only the most necessary rendering techniques were covered. On the other hand, feasible representation (e.g. smooth rendering, no visual artifacts, ability to style features) was a criterion. Consequently, both line string rendering and polygon rendering needed their respective best practices, well outlined in computer graphics literature. For drawing labels, the existing texture renderer could be exploited.

WebGL – just like OpenGL – cannot render complex geometries natively. It offers a low-level API, which can be programmed to bring a 2D or a 3D scene to life. It has a limited number of entities, called primitives, which are handled automatically. There is a primitive for points (`gl.POINTS`), capable of drawing squares representing points of arbitrary radii. Line primitives (`gl.LINES`, `gl.LINE_LOOP`, `gl.LINE_STRIP`) are slightly different methods for drawing segments. While `gl.LINES` can be used with segments of arbitrary width, it does not support segment connections. The others can draw line strings, although with a fixed width of 1 pixel. Finally, there are triangle primitives (`gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, `gl.TRIANGLE_FAN`) capable of drawing triangles on the screen. These fixed and limited primitives act as building blocks for any computer graphics application powered by WebGL.

By using one of these primitive types, developers can provide an array of coordinates (amongst other numeric parameters) for the API to draw. WebGL utilizes the GPU for rendering, therefore input data need to go through a pipeline including different components. While WebGL is based on OpenGL for Embedded Systems (OpenGL ES), its rendering pipeline is still complex. There are many phases (Ganovelli et al., 2014) from which two are of special interest, since they define what happens with input data, and are programmable. These are the vertex shader and the fragment shader.

Shaders are GLSL programs telling the GPU how it should handle input coor-

dinates and parameters. In the vertex shader, every vertex is handled separately. They must be transformed to Normalized Device Coordinates (NDC), a coordinate system used by this phase. NDC spans from -1 and 1 in every axis, independently from the size of the canvas. The fragment shader is after the rasterization phase and is run for every pixel in the resulting layer. In this shader, the program must assign a color to each pixel based on the developer's rules. In each pass, the exact row and column of the processed pixel is known, therefore this phase uses screen coordinates. The two shader programs can communicate through special variables (in case of triangles), which need to be defined for every vertex, and are interpolated between vertices for every pixel.

In the case of textures, the pipeline is similar to vector rendering. Since the fragment shader defines each pixel's color, drawing texture is also done there. Textures need a vector bounding box, where they will be displayed. This rectangle (also known as a quad) consists of two triangles and created in the vertex shader. In the JavaScript program, a texture is bound to each quad, and only then, the fragment shader can apply the bound texture to each pixel of the quad.

With some of the most important aspects of WebGL summarized, there are few slightly more advanced techniques to mention. OpenGL based rendering engines use at least two different coordinate reference systems (CRSs). One of them is the internal CRS of the application. In the case of a cartographic or GIS application, it is defined by the map's projection. The other is the NDC, to which every vertex needs to be transformed. In more complex cases, there can be other CRSs as well, like screen coordinates or sheet coordinates. Coordinate transformations are usually done by multiplying an affine 3D transformation matrix with each coordinate represented as a vector. The matrix needs to be provided only once per scene, and the GPU can calculate primitives from raw inputs with great performance.

The other technique is avoiding rare problems occurring from floating-point rounding. Since most of the GPUs can only handle 32-bit floating-point numbers, there are some cases when 64-bit input values are rounded. In cartographic applications using different map projections, this is not even such a rare occasion. Several global projections use meters as units, limiting the maximum precision of coordinates in a 32-bit representation. The rounding error is in the order of centimeters, which is negligible on a global scale. However, since users can zoom into maps, these applications can be used on scales where these errors become significant. Apart from the professional significance of such errors, if not handled properly, these unwanted roundings generate serious visual artifacts (Cozzi & Ring, 2011). On larger zoom levels, a phenomenon called jittering can be observed, where during zooming, geometries start to visibly shake.

There are two different techniques for avoiding such errors. One of them is coordinate packing (Cozzi & Ring, 2011), which is more popular in modern hardware accelerated web mapping applications. This method packs a single 64-bit floating-point value in two 32-bit values and passes them to the GPU. Then the GPU can operate on those values, avoiding rounding errors. The other method is called relative to eye (RTE) rendering. This floating origin method (Thorne, 2005) is an

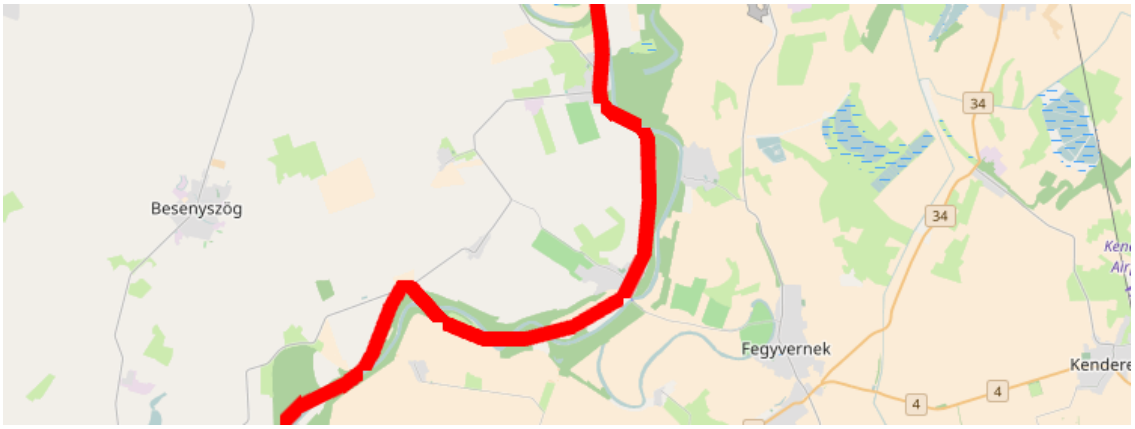


Figure 15: Part of the river Tisza rendered with `gl.LINES`. While the line could be styled with a color and a width, individual line segments are not connected properly, resulting in visual artifacts.

older solution for such problems, where every input coordinate is transformed to represent relative distances from the center of the view. This way, on scales where jittering start to cause problems, coordinate values become smaller, making more place for precision digits. Since OpenLayers already used RTE to avoid jittering effects, the implementation maintained that choice, and was built on it.

Finally, antialiasing is a technique for smoothing out jagged edges produced by slant geometries. To visualize this, a black 1-pixel line should be imagined on a white background. Since displays have square pixels, only straight (axis-oriented) geometries can be displayed as straight edges. When the geometry is skewed, diagonal pixels need to be colored black. This makes the edges look jagged, which can be smoothed out by gradually blending such an edge into the background. By increasing the alpha value on the edges, or mixing them with the background color, a smooth transition is produced. This technique is called antialiasing. While most web mapping applications use their antialiasing algorithms, for the sake of simplicity, this implementation relied on browsers' built-in multisample antialiasing (MSAA) technique (Liktor & Dachsbacher, 2013).

6.1. Line strings as triangles

Drawing lines is hard (DesLauriers, 2015). It could be easy if one could rely on built-in line rendering functionality. However, since web mapping applications need line string layers to be styled properly, neither of those functionalities are feasible. The `gl.LINES` mode has the best approximation for the requirements of line string layers, although line connections and line endings cannot be handled with it (Figure 15).

This shortcoming raises the demand for a more convoluted drawing methodology. For a correct representation of geospatial line string data, segments must be triangulated, in which way, both line endings and line connections can be maintained. Furthermore, both line endings and connections can be customized in a

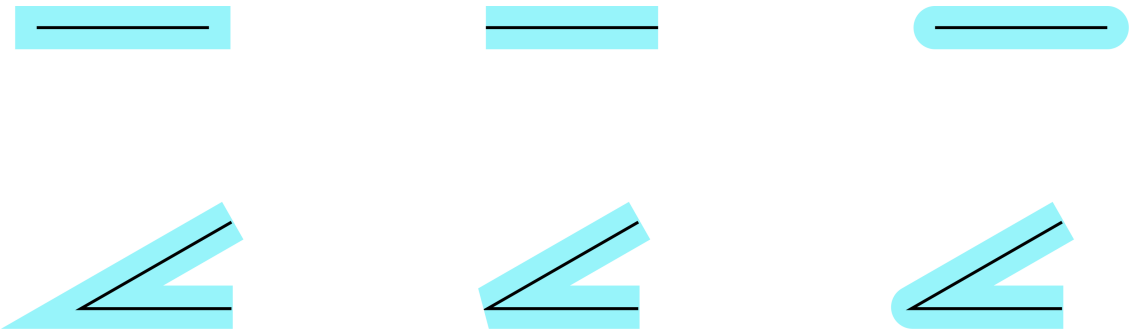


Figure 16: Relationship of base lines (black) with their respective line caps (top row) and line joins (bottom row). Line caps can be square (left), butt (center), and round (right). Line joins can be miter (left), bevel (center), and round (right).

capable application (Hearn & Baker, 2004). Typical line endings (caps) can be square, butt, and round, while typical line joins can be miter, bevel, and round (Figure 16). In order to handle all of these cases, line strings must be triangulated beforehand.

Line string triangulation has several methods, although most of them root back to the same set of calculations. Every segment needs to be cut into at least two triangles. If there are caps, they need to be added to the end of the line string as two additional triangles. All of the necessary points for these steps can be calculated by offsetting the endpoints of segments. A slightly more complicated calculation is the join point (both upper and lower) of two segments. Every join needs a lower point, while a miter join also needs an upper point. These join points must take the orientations of the two meeting segments into account. If every point is calculated, a two-segment line (V_1, V_2, V_3) can be triangulated with 8 points (Figure 17).

The triangulation can be done in the CPU with relative ease. By transforming offset distances to CRS units, every vertex of every line triangle can be given to the GPU for rendering. On the other hand, this method would slow down the engine, since a huge amount of calculations need to be done for every redraw. Consequently, vertex position calculations needed to be ported to the GPU. This way, the GPU can calculate every vertex position in parallel, providing decent performance. In order to achieve this, a universal vertex shader program was created, which could calculate all of the required offsets.

The problem with such an approach is only one vertex shader operates on every input coordinate. That is, there is a simple program, which needs to be able to decide whether it has to create a simple perpendicular offset with half of the line's width, or it has to create a miter. It needs to be able to decide the direction of the given offset. It also needs to be able to apply rounding, if required. All of those requirements need to be fulfilled before the GPU can process every vertex in a line segment asynchronously.

Such a program needs at least 6 parameters to work. First, it needs the coordinates of the current point in the line string. It also needs the coordinates of the previous, and next points, or it will not be able to calculate line joins. When

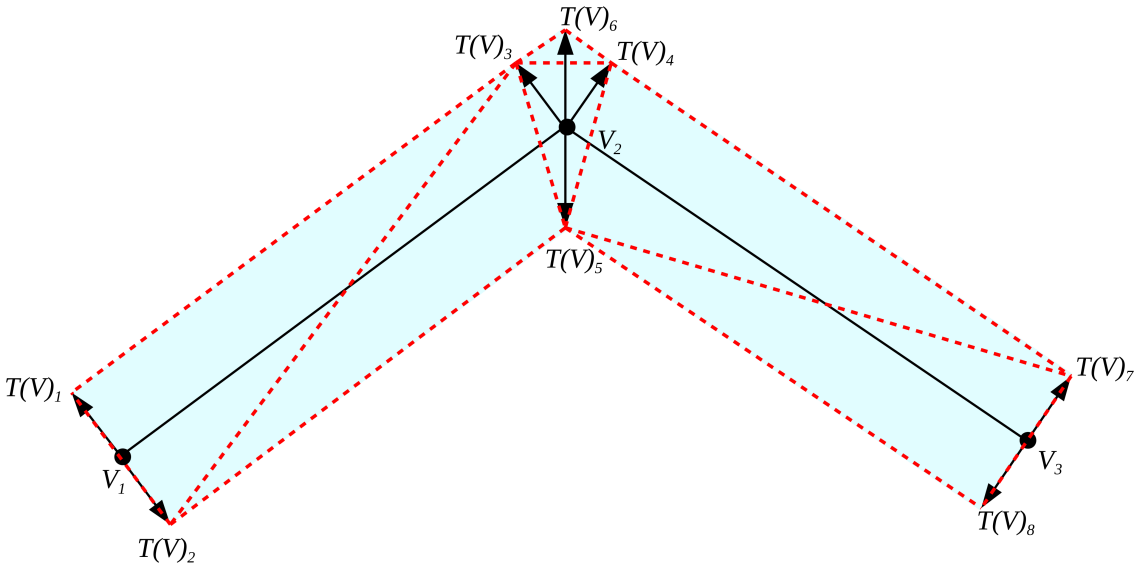


Figure 17: Triangulation scheme of two connected line segments V_1, V_2, V_3 with a miter line join and no line caps. Arrows show the magnitudes and directions of offsets needed to require triangulation points $T(V)_n$. Dashed lines represent triangles resulting from connecting triangulation points in the right order.

processing a starting or ending point, it still needs the location of the neighboring vertex. It also needs an instruction (e.g. offset, miter, square cap) to apply the correct offsetting procedure. Finally, it needs a direction, and if rounding is a requirement. As the application uses WebGL 1.0, which is only capable of receiving 32-bit floating points from the CPU, using so many variables for such simple parameters would have been a waste of resources. In order to optimize communication between the CPU and the GPU, a basic encoding was used to compress an instruction i, a direction d, and a rounding factor r into a single parameter $p = i \times d \times r$. For assuring lossless decoding, instructions were prime numbers, direction was either 1 or -1, and rounding was 2 if required, and 1 if not.

With the required parameters outlined and provided, a simple vertex shader could be built by following a fairly simple pseudocode (Program 5). With this method, the input needs to contain three pairs of coordinates, and an encoded parameter for every vertex in the triangulation. Providing the parameters for every vertex in the visualized polyline is not enough, as WebGL shaders cannot emit new vertices. Since there is no geometry shader in WebGL, every triangulation vertex needs to be provided, increasing the input's redundancy. For example, a three vertex line string needs at least two sets of parameters with the first pair of coordinates, four with the second, and two with the third. However, in the seemingly redundant parameter sets, the directions and instructions are different.

While the simple approach works most of the time, there are some edge cases which made the shader more complex. The first edge case is when bevel line joins are required. This can be handled easily, since miters form a separate triangle ($\triangle T(V)_3 T(V)_4 T(V)_6$ in Figure 17). The superfluous triangle can be discarded by

```
Require: p_0, p_1, p_2, direction, instruction, width
i ← instruction
d ← direction
w ← width
u ← p_1 − p_0
v ← p2 − p1
if i = offset then
    // Offset p_1 along rotated û or v̂ with half width
    p_1 ← p_1 + w ÷ 2 × ⟨−û_y, û_x⟩ × d
else if i = miter then
    // Calculate tangent vector
    k ← û + v̂
    // Calculate miter length
    l ← w ÷ 2 ÷ k̂ · ⟨−v̂_y, v̂_x⟩
    // Offset p_1 along rotated tangent with l
    p_1 ← p_1 + ⟨−k̂_y, k̂_x⟩ × l × d
end if
```

Program 5: Simple pseudocode for calculating offsets required for line string triangulation (Farkas, 2019).

either excluding it from the input array in the CPU or degenerating it in the GPU. The most convenient GPU approach is making $T(V)_6$ equal to $T(V)_3$ or $T(V)_4$.

A more problematic edge case is when triangles overlap. Such a phenomenon can occur when a lines string has self-intersections. Visual artifacts can only be observed from this edge case, when opacity is set lower than 100%. When two semi-transparent triangles meet, their overlapping parts are less transparent. This edge case was solved with adequate WebGL parametrization in JavaScript. A depth test was used for filtering out already colored pixels from the whole layer, therefore every pixel got colored only once.

Triangles can also overlap when the angle of two line segments is too sharp compared to the segments' width. In such a case, a more troubling artifact can occur. Due to rounding problems, the bottom miter can be displaced, producing sharp wedges, which cannot be controlled by limiting the maximum length of miters. This problem had to be solved with additional logic in the vertex shader. Intersections were searched between the two meeting line edges ($\overline{T(V)_2T(V)_5} \cap \overline{T(V)_7T(V)_8}$ or $\overline{T(V)_7T(V)_8} \cap \overline{T(V)_1T(V)_2}$ in Figure 17). If an intersection point could be found, the bottom miter was snapped to that point.

Finally, the problem of rounding had to be solved. In order to simplify the problem, and make the solution as fast as possible, rounding was done in the fragment shader (Program 6). When rounding is required, top miter is drawn. Then, the coordinate of the line join is passed to the fragment shader. In the fragment shader, the join coordinates (V_2 in Figure 17) are transformed to screen coordinates. Finally, the distance between the currently processed pixel, and the line join is measured.

If the distance is longer than half of the line's width, the pixel is discarded.

```
varying float v_round;
varying vec2 v_roundVertex;
varying float v_halfWidth;

uniform float u_opacity;
uniform vec4 u_color;
uniform vec2 u_size;
uniform float u_pixelRatio;

void main(void) {
  if (v_round > 0.0) {
    vec2 windowCoords = vec2(
        (v_roundVertex.x + 1.0) /2.0 * u_size.x * u_pixelRatio,
        (v_roundVertex.y + 1.0) /2.0 * u_size.y * u_pixelRatio);
    if (length(windowCoords - gl_FragCoord.xy) > v_halfWidth *
        u_pixelRatio) {
      discard;
    }
  }
  gl_FragColor = u_color;
  float alpha = u_color.a * u_opacity;
  if (alpha == 0.0) {
    discard;
  }
  gl_FragColor.a = alpha;
}
```

Program 6: Fragment shader for rounding miters and square caps. Note that the device pixel ratio is included in the calculations, otherwise, results would be incorrect for high DPI (e.g. retina) displays. These displays have such high pixel densities, that they need to form logical pixels from several physical pixels, while the GPU uses physical pixels only.

By handling these edge cases, most of the visual artifacts in computers with modern GPUs were mitigated. On the other hand, many more such techniques must be used to provide a backward-compatible solution, especially for old integrated video processors.

6.2. Breaking up polygons

Rendering polygons is not as hard as rendering line strings. If they are partitioned to triangles, the GPU can draw fills right away, while strokes can be rendered with the line string renderer. The hard part of drawing polygons is breaking them up to triangles. While simple polygons can be partitioned with low complexity,

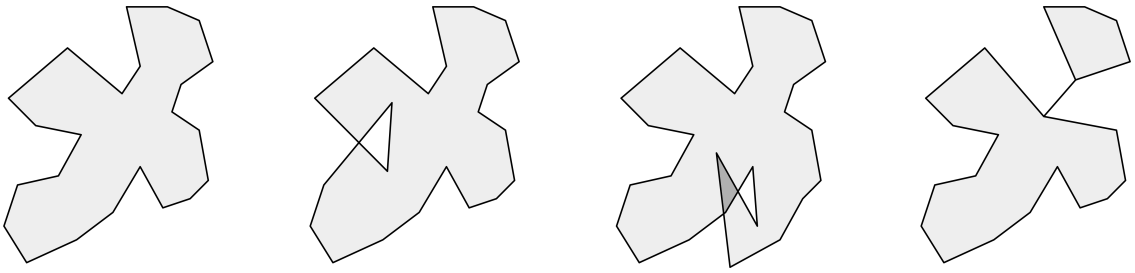


Figure 18: A simple polygon (left), and three polygons with different deficiencies. From left to right a self-intersecting polygon, a self-overlapping polygon, and a degenerate polygon.

topological errors (e.g. self-intersections, self-overlaps, collinear segments) are much harder to properly handle (Figure 18).

Polygon triangulation rarely raises a problem in most computer graphics applications, due to several mature libraries capable of breaking up polygons robustly, with great performance. However, those libraries and frameworks (e.g. GLU tessellator, cairo, Qt) are written for desktop applications. In the Web ecosystem, there are fewer solutions, and some of the direct ports are not optimal due to different overheads.

The theoretical background of triangulation is thoroughly studied and developed, and is the same for Web applications and desktop applications. There are several different techniques for the same goal. Delaunay triangulation (Held, 2001) creates the best quality decompositions, although it is significantly slower than other techniques. Delaunay triangulation creates a triangulation $T(P)$ from polygon P, where every triangle in $T(P)$ is a Delaunay triangle. Circumcircles of Delaunay triangles $\triangle P_i P_j P_k$ cannot contain any other points from P than P_i, P_j, or P_k (Lee & Schachter, 1980). It is best suited for professional applications other than visualization, like creating triangulated irregular network (TIN) models (Peucker et al., 1978).

There are two popular algorithms for fast triangulation. One of them is called trapezoidal decomposition, which partitions the polygon to trapezoids with horizontal sweeping lines going through each vertex, and triangulates the resulting trapezoids (Seidel, 1991). The other technique is called ear clipping, which simply connects the vertices of a polygon P. It cuts sequential vertices P_i, P_{i+1}, P_{i+2}, if they form a valid triangle (ear), and continues the process until the polygon is triangulated (Meisters, 1975). Since the rules of clipping ears can significantly affect the outcome, there are several variants of this technique.

There are two JavaScript libraries capable of fast triangulation. Libtess is a direct port of GLU, which uses trapezoidal decomposition. It is precise, can handle polygons with topological errors, but rather slow in its JavaScript form (Table 16). The other one is earcut (Agafonkin, 2015), which uses ear clipping for fast triangulation. It can be used for real-time triangulation, although it cannot handle topological errors. It adapts the Fast Industrial-Strength Triangulation (FIST) algorithm (Held, 2001). Since for OpenLayers both performance and precision were

Sample data	Vertices	earcut	libtess
OSM building	15	795 935	50 640
dude shape	94	35 658	10 339
holed dude shape	104	28 319	8883
complex OSM water	2523	543	77.54
huge OSM water	5667	95	29.30

Table 16: Performance of libtess and earcut with various sample data (Agafonkin, 2015). Performance is in operations per second, therefore higher values mean better performance.

criteria, another FIST variant was created, which can handle topological errors.

The first rule of ear clipping, is every vertex of polygon P can be categorized as convex or reflex. Only triangles with convex middle vertices can form ears. A vertex P_i is convex, when $\angle P_{i-1}P_iP_{i+1}$ is less than 180°. Otherwise, the vertex is categorized as reflex. This rule applies to every ear, however there needs to be more rules for proper triangulation. FIST proposes two sets of rules CE1 and CE2 (Held, 2001). While CE2 is faster to check and easier to implement, it only applies to simple polygons. CE1, on the other hand, is universal, and can be used for polygons with topological error. By adapting the two sets of rules (Farkas, 2019), in the implementation $\triangle P_{i-1}P_iP_{i+1}$ forms an ear, iff:

1. P is simple
2. P_i is convex
3. there are no vertices of P in $\triangle P_{i-1}P_iP_{i+1}$ (except for P_{i-1}, P_i, P_{i+1})

or

1. P_i is convex
2. $\overline{P_{i-1}P_{i+1}}$ does not intersect with any segment of P (except in P_{i-1}, P_{i+1})
3. $\overline{P_{i-1}P_{i+1}}$ is completely inside P

The implementation uses a doubly linked list as its main data structure. The linked list's nodes are segments, due to the need of checking intersections. As a secondary data structure, it uses an R-Tree, indexing every segment for increased performance. Since simple polygons should be triangulated as fast as possible, it uses a penalizing approach 7. The more deficiencies the polygon has, more precise and slower techniques are applied. First, the topology of the polygon is checked. If the polygon is simple, it uses the first set of rules for fast triangulation. If not, it starts triangulating using the second set of rules. If a self-intersection is found, it places a Steiner point in the intersection, and continues clipping ears. When no ears are remaining, it checks if the polygon became simple. In such a case, the remaining part can be triangulated, however, the order of points must be changed.

If the polygon is still not simple, the algorithm cuts the polygon into two parts, and starts two new processes with them. The clipped ears are put into an array, which can be handed to the GPU.

```
Require: Polygon P with each vertex classified as convex or reflex based
    on convexity with respect to neighboring vertices
while |P| > 3
    if P is simple
        if there are valid ears in P then
            T ← Clipped ears using fast rules
        else if P can be reclassified then
            P ← Reclassified P
        else
            // P can have touching segments
            P ← P with resolved self-intersections
        end if
    else
        if there are valid ears in P then
            T ← Clipped ears using precise rules
        else if P can be reclassified then
            P ← Reclassified P
        else if There are self-intersections then
            P ← P with resolved self-intersections
        else if P is simple again then
            P ← P with inverted orientation
        else
            Split P into two parts and triangulate
        end if
    end if
end while
T ← T+ last ear in P
```

Program 7: Pseudocode of the ear clipping algorithm used for triangulating polygons (Farkas, 2019). Since the classification of vertices (convex, reflex) can change with every clipped ear, the algorithm first tries to reclassify the polygon after it runs out of ears.

Polygons with holes are handled by the de facto standard way of bridging. This technique creates a single polygon, by merging the holes into the main ring without changing its orientation (Figure 19). During the process, existing vertices are linked together to form a single degenerate polygon, hence the name bridging. While the implementation can triangulate polygons with topological errors reliably, it is not free from errors. There are degrees of deficiency when the algorithm cannot create correct triangulations. In such cases, the process does not hang, it still creates a visualization, although with some triangles misplaced or excluded. However, according to local tests on various datasets, the implementation creates correct results with real-life data on machines with modern GPU.

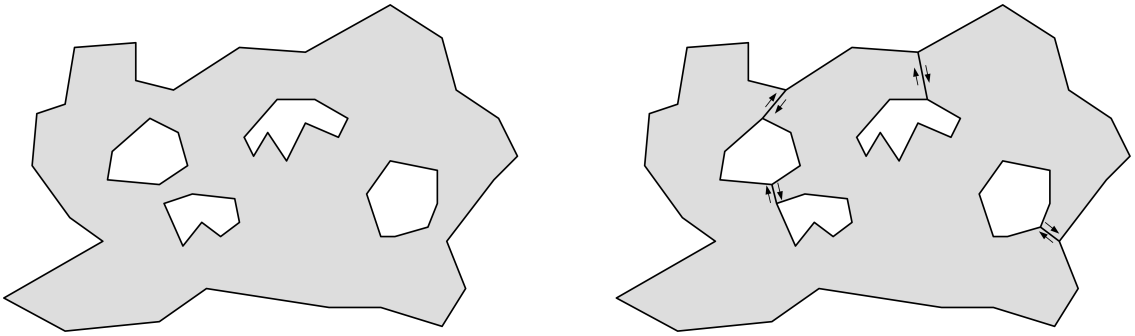


Figure 19: A polygon with holes (left). The holes are merged into the main ring in a spatial order (e.g. left to right), forming a single degenerate polygon without holes (right) as a result.

6.3. Drawing other features

There were two additional functionalities, which needed to be implemented: drawing circles and drawing labels. While in the Canvas engine circles can be handled by the polygon renderer, in WebGL, they need to be handled separately. Circles can be drawn as a set of triangles, although the result will be only an approximation of a circular arc. In order to draw smooth circles, a better solution was implemented.

Just like polygons, circles have fill and stroke styles. The stroke line's baseline is sitting on the circle's edge, therefore the stroke only expands the circle by half of the stroke's width. In this approach, the vertex shader draws a square from two triangles. The corner points of the square are calculated in the vertex shader. They are placed in a way, that the resulting square completely encloses the styled circle.

Styling is done in the fragment shader, similarly to rounding line joins and line caps. Based on the distance of every pixel from the circle's center point, it gets colored or discarded. If the pixel is within the fill radius, it gets a fill color. If it is outside of the fill radius, but inside of the stroke radius, it gets a stroke color. On the border of the fill and stroke radii, colors from the two styles get blended for a smooth transition. Outside of the stroke radius, pixels get discarded. While this approach is simple and reliable (Figure 20), it has one significant limitation. It is hard to extend for dashed lines, which was not implemented by any of the renderers.

Label rendering was like image rendering for the drawing part. A label texture is generated first and bound to the WebGL context. In the vertex shader, a rectangle is drawn in the form of two triangles in the place of the label. Finally, the label texture is placed in the rectangle. While drawing labels is simple, creating label textures required some considerations.

Similarly to the image renderer, labels are generated and cached with the HTML5 Canvas API, on internal `canvas` elements. By avoiding this dependency on the Canvas API, one could make faster and memory-efficient solutions, however, every font would need to be served. In order to keep the ability to use any browser-supported font, this dependency was deemed necessary by the implementation. In the preparation phase, not individual labels are cached, since that would be a waste of memory.

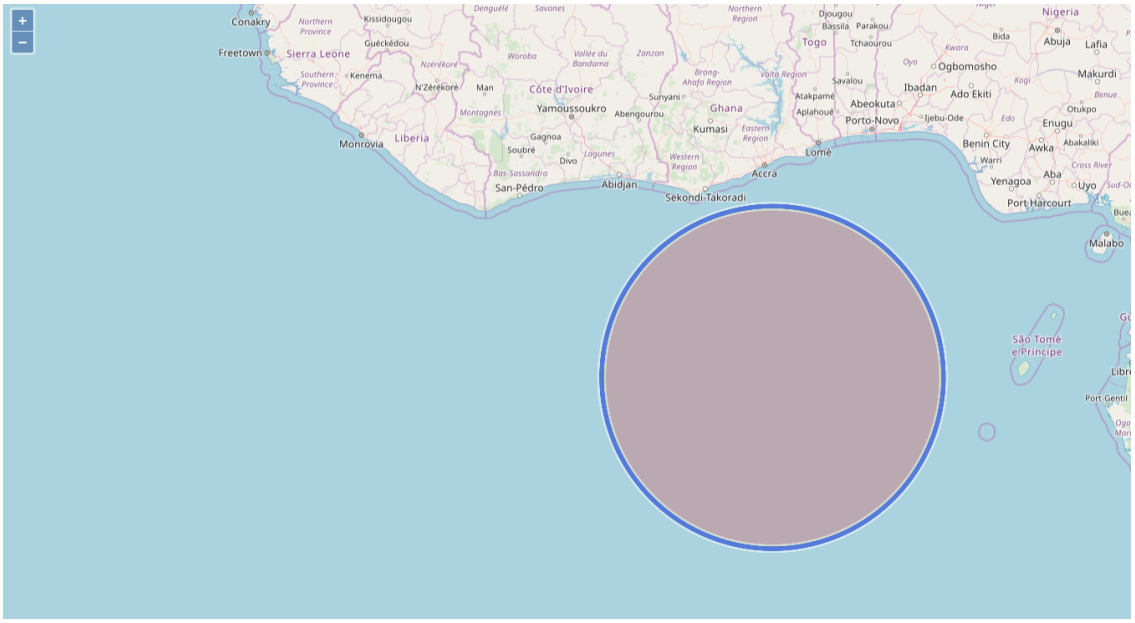


Figure 20: An opaque circle with a complex stroke style drawn by the WebGL circle renderer. The stroke has a white line of 10-pixel width under a 5 pixels wide blue line. The red fill has a 20% opacity, while every stroke line has a 50% opacity.

Figure 21: Example of a glyph atlas used by Mapbox (Käfer, 2014).

A glyph atlas (Figure 21) is used to store individual characters with the same styling in a single cache. Then, when labels are rendered, those characters are put next to each other, their overall width and height are measured, and the label texture is put onto the map. A possible improvement would be using signed distance fields (SDF) for improved rendering quality (Felzenszwalb & Huttenlocher, 2012).

6.4. Benchmarking the renderer

OpenLayers having two similar rendering engines for Canvas and WebGL offers a great opportunity to assess the new WebGL implementation. By benchmarking both of the engines with the same datasets using different GPUs, a good picture can be obtained about strengths, limitations, and room for improvement. Since raw frames per second (FPS) values are meaningful mostly to the trained eyes, some

Zoom level	1	2	3	4	5
NVIDIA GPU					
WebGL animating	56.35	56.82	57.53	56.00	54.19
Canvas animating	29.24	23.64	22.45	32.59	46.37
WebGL drawing	14.75	10.92	10.26	12.18	15.37
Canvas drawing	22.94	18.59	14.23	24.33	33.00
Intel GPU					
WebGL animating	58.99	55.34	52.59	59.01	59.72
Canvas animating	26.07	20.21	15.17	16.77	22.83
WebGL drawing	10.91	8.37	7.17	7.72	8.03
Canvas drawing	13.57	12.92	10.28	10.71	14.66
ARM Mali GPU					
WebGL animating	12.67	12.08	10.66	16.51	21.83
Canvas animating	1.19	1.22	1.87	3.69	5.84
WebGL drawing	1.25	0.94	0.82	1.45	2.27
Canvas drawing	1.13	0.96	1.32	2.51	4.05

Table 17: Rendering performance (FPS) of the thematic layer group. Lags are emphasized by underlining, and severe lags by framing (Farkas, 2019).

thresholds were also highlighted in the results. Cases were categorized as lagging, when the FPS went below 16, the threshold value for perceiving subsequent frames as a continuous animation (Mengeringhausen & Witherell, 1962). Severe lags were considered under 5 FPS, where the application is hardly usable.

The first set of measurements targeted the thematic layer group (Table 3). Results (Table 17) indicate that the Canvas renderer can handle a load of a typical, not optimized vector-based web map on decent GPUs. On the other hand, the WebGL renderer has a better overall performance. The animating speed of the WebGL renderer was exceptionally high on both the integrated Intel and the dedicated NVIDIA GPU. Those values perturb around the maximum 60 FPS value of a display with a refresh rate of 60 Hz. Around 60 FPS, results showed more instability. While a 1 FPS difference was significant around the 20 – 30 FPS interval, above 50, a 4 FPS difference could mean only a minor disturbance during a benchmark.

General trends of OpenLayers' rendering mechanism can be identified by looking at the Canvas results, and the WebGL drawing results. The first zoom level has the best generalization. Since fewer vertices need to be drawn (Table 4), both drawing and animating are faster. Then, as the scale grows, rendering performance lowers. However, on the 4th zoom level, a leap can be observed. This is when the map's canvas cannot hold the whole extent anymore, and spatial indexing starts to limit the number of visualized features.

Overall user experience is mostly affected by the animating speed. Those FPS values are observed during user interactions, like zooming, panning, and rotating. Drawing speed is only observed when the map comes to a standstill. Therefore, if animating speed is high enough to create continuous animations, and drawing

speed stops the rendering pipeline for half a second, it is perceived as better than if the map lags during interactions.

This is the main reason why a naive, not thoroughly optimized WebGL engine still has a great impact. By caching buffers (triangulated vertices), it can provide a significant performance boost during animations. On the contrary, as the Canvas engine can only cache drawing instructions, the performance boost is not as notable. On the other hand, by considering animating speed the main contributor to decent user experience, a typical vector web map can be visualized feasibly on stronger GPUs with both of the renderers.

The only scenario that resulted in severe lags was using the ARM Mali GPU in a handheld device. Since it is a weak GPU in an obsolete smartphone, it can represent low end builds. While the application was lagging using the Canvas renderer, the WebGL engine could still animate the map with only slight lags. Therefore, using a WebGL engine has a very important benefit of making an application less dependent on the age or computing power of the device.

By investigating further (Figure 22), trends of individual components could be identified. The Intel GPU results were chosen for this, since the NVIDIA GPU was strong enough to blur some of the more subtle differences, hiding some details. The most significant components in rendering are polygons. The total FPS follows polygon rendering performance very closely, while it is affected by other features less dominantly. This is due to the need for triangulation. Both Canvas and WebGL engines need to triangulate drawn polygons, although, with the Canvas API, this step is hidden from users.

There are some additional differences worth noting in the individual profiles of rendering. From Canvas polygon drawing and WebGL polygon drawing performances, the difference of a not thoroughly optimized JavaScript–, and a mature desktop triangulation process can be observed. Canvas triangulation always results in correct decomposition and is significantly faster than its WebGL counterpart. This is not only due to the levels of optimization, additional overhead from the JavaScript pipeline also contributes. On the other hand, triangulation results created for the WebGL engine can be cached. The other difference is between animating and drawing lines with the Canvas engine. For polygons, caching instructions do not provide a significant boost (due to the need of triangulation). On the other hand, for lines, caching results in a stable 60 FPS, just like in case of animating with the WebGL engine.

Since additional trends were sought after, the NVIDIA and Intel GPUs were put under heavy load in the GIS scenario (Table 18). Due to ARM Mali's staggering under a load of a lighter thematic composition, that GPU was excluded from these measurements. The weight of a GIS workflow (Table 4) was tolling on the Intel GPU. Even WebGL animating performance has dropped from the optimal 60 FPS range. This was mostly due to the large number of triangles needed to be drawn in case of real-world polygon data, however, the large number of points also contributed.

This is such a case when the Canvas renderer becomes infeasible. Since this is a

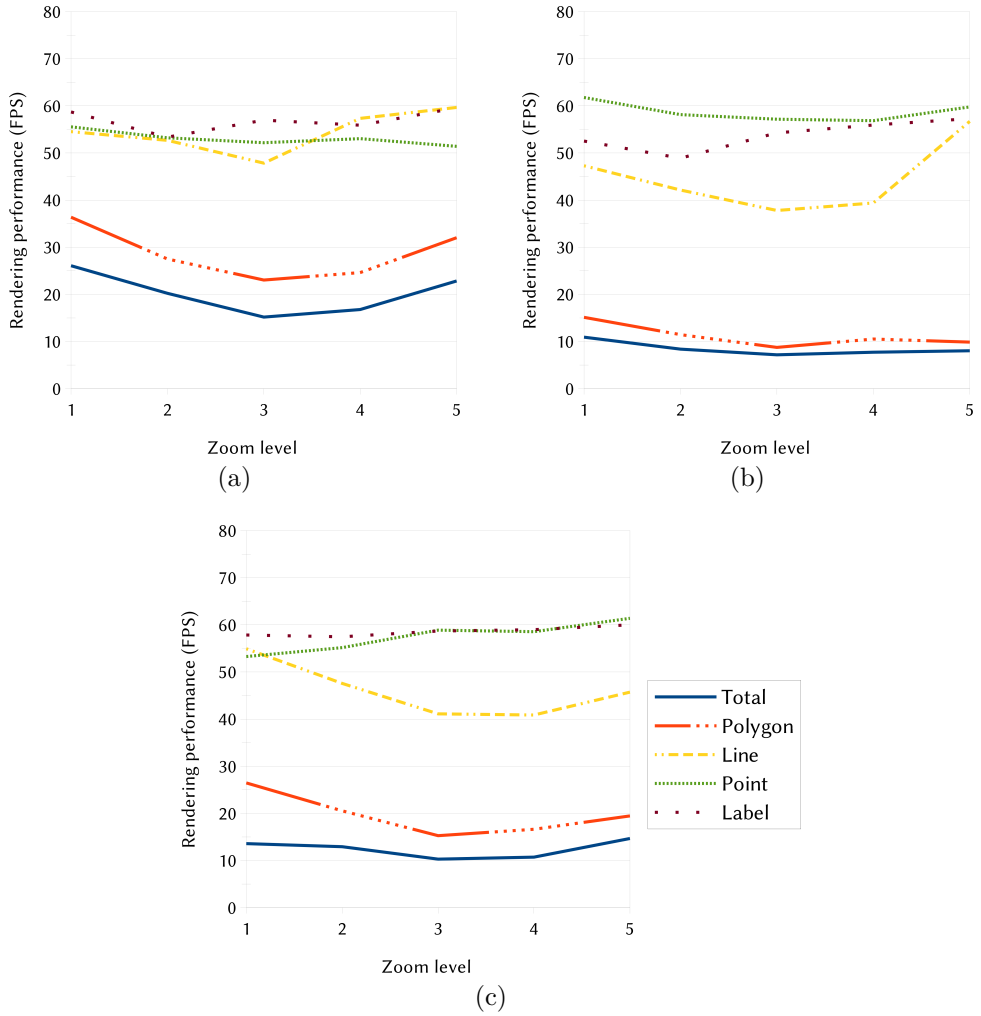


Figure 22: Detailed results of the thematic layer group's rendering tests on the Intel GPU. Canvas animating performance (a), WebGL drawing performance (b), Canvas drawing performance (c) (Farkas, 2019).

possible rendering load from a small size analysis, visualizing arbitrary data requires a WebGL renderer. Canvas animating and drawing performance was low even on the NVIDIA GPU. Before the R-Tree could exclude some of the geometries, the application suffered from heavy lags. On the other hand, WebGL drawing performance was also bad. While the whole application felt smooth, the few milliseconds stops could be observed between interactions. For example, after a pan, the map could not be interacted with instantly. Between two successive interactions, a few milliseconds have to be passed, resulting in a suboptimal experience.

By looking at the point rendering performance only (Figure 23), the trends of texture rendering can be seen. Considering points alone, some of the results exceeded the 60 FPS limit, which is due to the efficiency of rendering points as textures. Every point needs only two triangles (a quad), and if every point has the

Zoom level	8	9	10	11	12
NVIDIA GPU					
WebGL animating	59.05	57.28	57.76	57.76	58.69
Canvas animating	3.43	4.03	8.34	15.54	28.00
WebGL drawing	4.1	3.53	7.29	13.01	28.99
Canvas drawing	2.67	2.90	7.18	13.31	23.54
Intel GPU					
WebGL animating	17.33	24.83	44.85	51.64	56.30
Canvas animating	2.75	2.41	6.14	11.00	18.85
WebGL drawing	2.97	3.05	5.59	9.91	16.08
Canvas drawing	2.04	2.13	5.29	8.96	14.49

Table 18: Rendering performance (FPS) of the GIS layer group. Lags are emphasized by underlining, and severe lags by framing (Farkas, 2019).

same symbology, only one texture is needed. On the other hand, by having many textures on lower zoom levels (Table 3), both engines produced lower FPS values. Even the NVIDIA GPU had poor performance during the drawing process of the Canvas engine. Increased performance on larger zoom levels followed the number of visible features in the point layer closely (9839; 7878; 2575; 801 and 242 vertices in zoom levels 8, 9, 10, 11 and 12 respectively).

In order to gain additional, more specific insight into the WebGL engine, both the thematic and the GIS groups were profiled using the NVIDIA GPU. From various predefined zoom levels and extents (Table 4), the one with the heaviest load was chosen. Consequently, the thematic group was profiled on zoom level 3, while the GIS group on zoom level 8. Profiling was type-specific. Since possible bottlenecks and optimization possibilities were sought, different geometry renderers were profiled individually.

From the profiles, several categories collected and aggregated by the browser are shown. Scripting is the time spent on executing JavaScript routines. Rendering represents the time spent on styling DOM elements. Painting holds image rendering processes. Other has categories, which could not be categorized into the previous groups. Time spent on the GPU for example counted into the group of other processes. Furthermore, as OpenLayers clearly, and rather accurately separates cacheable and not cacheable parts, the phases of preparation (cacheable) and composition (non-cacheable) were also measured. Those ratios are of the total drawing time, not time spent on scripting.

In the case of drawing points and line strings (Table 19), most of the time is spent on scripting. Rendering and painting are negligible compared to other phases. Other processes take some time, which is probably due to time spent on the GPU. Since Canvas elements are partially hardware accelerated, the Canvas renderer is also using the GPU, albeit it is hidden from the library. Preparation and composition phases are mostly related to the engine. The Canvas renderer can cache less data, making the composition phase's ratio larger. The WebGL engine

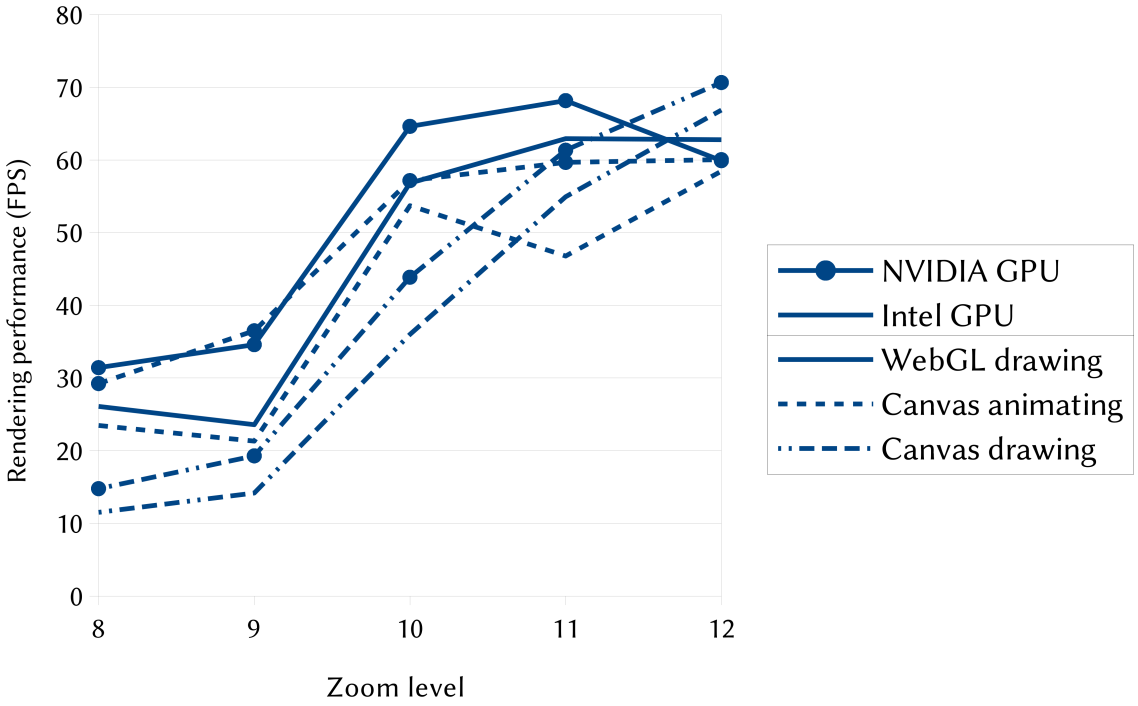


Figure 23: Rendering performance of the GIS point layer under various circumstances (Farkas, 2019).

can cache more data, although the Canvas engine's line string renderer has similarly effective caching capabilities.

From the four different branches, there was only one function call with outstanding self-time. The WebGL engine's line string renderer spent 13.25% of its total time on the `addVertices` function. This function extends the buffer with new triangulation parameters. Since it operates on a single, large, dynamic array, that call can be optimized by calculating and allocating the whole buffer before triangulating a line string layer. Using a typed array would speed up the process even more, although it would limit the usability of the library to browsers with ECMAScript 2015 support.

	Point		Line string	
	Canvas	WebGL	Canvas	WebGL
Drawing time (ms)	19.1	9.66	46.65	21.2
Scripting	79.01%	89.86%	87.16%	78.49%
Rendering	0.42%	1.14%	0.24%	0.61%
Painting	2.57%	3.31%	0.71%	2.36%
Other	18.22%	5.69%	11.83%	18.49%
Preparation	40.31%	62.01%	66.05%	58.77%
Composition	26.65%	7.76%	17.43%	11.79%

Table 19: Ratio of different calls when rendering points and lines (Farkas, 2019).

	Label		Polygon		
	Canvas	WebGL	Canvas	WebGL	WebGL-c[a]
Drawing time (ms)	11.47	21.16	37.27	283.84	54.03
Scripting	78.55%	63.52%	59.67%	99.63%	98.04%
Rendering	0.87%	0.8%	0.32%	0.04%	0.22%
Painting	4.01%	3.07%	0.78%	0.1%	0.54%
Other	17%	32.8%	39.31%	0.22%	1.17%
Preparation	46.21%	49.76%	32.28%	95.59%	76.81%
Composition	11.33%	3.59%	23.1%	3.34%	17.53%

[a]Cached. In this scenario, triangulation was disabled in the application.

Table 20: Ratio of different calls when rendering labels and polygons (Farkas, 2019).

Breaking down label and polygon rendering (Table 20), the dominance of JavaScript routines in the pipeline can be reassured. While both of the renderers have a large portion of scripting, WebGL's 99.63% is exceptionally dominant. By looking at the disproportionate drawing time, it can be seen, that triangulation takes very much time. It is better optimized in the Canvas polygon renderer, which has a large portion of other calls. Probably both triangulation and drawing are aggregated there. By looking at the WebGL polygon renderer with triangulation turned off, it is still slower than the Canvas engine. On the other hand, the ratios of different groups are no longer disproportional.

By looking at individual calls, several outliers could be identified in the WebGL label and polygon renderers. The label renderer spends 16.07% of its total time in the `getTextSize` function. This function is responsible for creating internal temporary labels using the Canvas renderer in order to obtain the width and height of the required label container. Considering that this function is also responsible for placing new glyphs in the atlas, and caching their widths, it has little space for further optimization. The polygon renderer on the other hand, spends 80.96% of its total time on triangulating functions. This could be avoided by storing triangulation data on polygons and only recalculating them when their geometries change. Furthermore, `drawPolygonCoordinates` could be observed with a 10.85% portion of elapsed time. It is worth noting, since the polygon layer had the most vertices, and raw coordinates need to be transformed due to RTE in OpenLayers. Therefore, a further possible optimization would be emulating 64-bit calculations on the GPU rather than using RTE for solving precision problems.

Measuring the GIS group (Table 21) added a few additional insights. This time, WebGL polygon rendering was measured without triangulation, since involving it would have not been progressive. According to the results, without triangulation, the WebGL renderer has a better time complexity than the Canvas polygon renderer. There is a turning point, where even a naive WebGL engine becomes more efficient, and the GIS group is beyond that point. As an insignificant, but interesting observation, the 71.76% ratio of other processes in the Canvas polygon renderer corroborates the assumption of the triangulation process is grouped there.

	Point		Polygon	
	Canvas	WebGL	Canvas	WebGL-c[a]
Drawing time (ms)	84.48	36.28	337.79	248.76
Scripting	78.61%	95.78%	28.04%	97.16%
Rendering	0.09%	0.55%	0.05%	0.05%
Painting	0.37%	0.83%	0.15%	0.10%
Other	20.98%	2.78%	71.76%	2.67%
Preparation	49.49%	86.99%	14.56%	68.28%
Composition	26.31%	4.82%	12.90%	27.57%

[a]Cached. In this scenario, triangulation was disabled in the application.

Table 21: Ratio of different calls when rendering the GIS group (Farkas, 2019).

7. Implementing Raster Management

For efficient raster management, not only several new classes were created, but the whole process was reconsidered. Traditional raster data by definition are matrices mapped to rectangular cells in a grid. Their design can be traced back to old computer systems, where computing power was limited, and it was necessary to have a direct mapping to square grids of different displays (e.g. plotter, monitor). While computers evolved, the definition of the raster model did not change. It is still useful, since it can visualize spatially continuous phenomena, without the need for interpretation (Bugya & Farkas, 2018).

With the exponential growth of computing power and the evolution of analysis techniques, new demands have arisen. Some of those demands could be addressed by extending the raster model. The necessity of cells being square-shaped has been lifted, and rectangular cells can be used in modern GIS software. 3D rasters (voxels) is another case when the traditional model could be extended, and voxel datasets can be utilized in some of the systems with 3D capabilities (e.g. GRASS GIS). However, it is still not possible to use different patterns, like triangular or hexagonal tessellations.

7.1. Rasters and coverages

It is interesting to look into the inner workings of the GIS raster model. By looking at a typical raster management pipeline, it can be observed, that rasters have a weakly coupled data and representation model. The data model is the underlying matrix with numeric values, each value representing a single cell in a spatial grid. This data structure – with the general rules of displaying cells – ensures the integrity of the model. No operation can be applied, which creates overlaps or gaps between cells, the topology of cells is ensured by the common understanding of raster values are mapped to grids. By using this understanding, it can be stated, that the raster model's integrity is provided by the data model.

The other part of the model is the representation model. In this phase, raster cells are rendered on the map as colored grid cells. Colors are assigned to cells according to rules and their raw numeric values. Furthermore, if rendering is considered as part of the representation model, – as common practice dictates – colors derived from raster values are mapped to a rectangular, axis-oriented grid. Grid cells can have different sizes on different axes, however, those sizes must be constant for individual cells in a single layer.

Data model	
Advantages	Continuous coverage
	Small size, good compressibility
	Simple data structure
	Good for parallel computing
	Easy to overlay when aligned
Disadvantages	Quadratic growth in raster size
Representation model	
Advantages	Easy to create textures
	Fast rendering (textures, pyramids)
	Easy to resample and interpolate
	Georeferencing is unequivocal
Disadvantages	Hard to reproject
	Rotation needs resampling
	Precision depends on latitude
	Sampling bias

Table 22: Characteristics of the raster data model without being exhaustive (Bugya & Farkas, 2018).

7.1.1. Characteristics of the raster model

By collecting raster characteristics based on their level of conceptuality (Bugya & Farkas, 2018), several advantages and some limitations can be distinguished, and tied to data and representation models (Table 22). One of the greatest advantages is continuous spatial coverage, as discussed before. Small size and good compressibility are also related to the data model, since those are originating from the underlying matrix data structure. Since rasters do not hold voluminous spatial information like vectors, most of the data are numeric values of a single type. Rasters only need a spatial tie point and a transformation matrix for creating discrete grids from image-like data (Ritter & Ruth, 1997). Consequently, lossless image compression methods can be applied to raster layers, resulting in small, binary files.

The simple data structure of rasters allows for easy and fast processing. Local cell coordinates can be converted to spatial coordinates, making visualization, subsetting, and extending easy. By breaking up large raster files to smaller tiles, they can be processed in parallel, solving edge rows and columns after processing is finished for neighboring tiles. When several raster layers are aligned (have the same resolution and extent), there is no need for spatial operations to calculate on multiple layers. Layers below each other can be aggregated with simple matrix operations, making operations faster than spatially processing each pair of cells.

The only disadvantage of having such a data model is its quadratic growth. For example, when a cell is added to the bottom of the layer, a whole row needs to be created. In a worse case, if the resolution of the layer is doubled, every cell partitions to four new cells. Therefore, the size of the raster quadruples. This

can be problematic in the case of real-world analyses. If two raster layers need to be aggregated, and two have different resolutions (e.g. optical and thermal bands of satellite imagery), the one with the lower resolution has to be resampled prior to processing. The size of that band changes polynomially, while its amount of information stays the same.

The representation model also holds several advantages. Its most significant one is the easiness to create textures from raster layers. This mostly applies to rasters with square cells, since they can be directly mapped to image textures with square pixels. Styled textures then can be cached for fast visualization. While rectangular cells need to be resampled to squares, it is still faster than drawing cells individually. Furthermore, textures are perfect candidates for building pyramids. Since squares can easily be partitioned to four half-sized, congruent squares, multiple pyramids can be built for even faster visualization on small scales.

Pyramid building (also called mipmapping in computer graphics) is a technique for creating lower-resolution versions of the same image. In GIS it is mostly used for accelerating rendering, since drawing large rasters on scales where individual cells cannot be observed can take unnecessarily large amounts of computing power. It is also important for feasible visualization because applying filters (e.g. bilinear, trilinear, cubic) on pyramids can smooth out sharp edges from naive resampling methods (e.g. nearest neighbor). Similarly to pyramid generation, applying other raster processing techniques (e.g. slope generation) is convenient on rectangular grids, as cell relationships are well-defined.

While there are some clear advantages of the traditional raster representation model, it also has several limitations. Raster layers are typically hard to reproject (warp), as the reprojected layer needs to have a rectangular grid (Figure 24). It does not matter if the size of individual cells would ideally change with latitude in a projection, the warped raster must comply with this constraint by interpolating cell values. This introduces variable precision among grid cells in some use cases, since the grid is regular in a projected CRS.

The same principle applies to rotating raster layers. While they can be rotated with the map for visualization purposes, when the grid is rotated and exported, cells need to be resampled. This limitation originates from rasters being axis-oriented by definition. Since they only store a tie point to a CRS (origin), the grid is generated by the renderer from the origin (lower-left corner), and the spatial dimensions of cells. While this limitation could have been easily mitigated by storing an additional rotation parameter in raster files, this feature did not meet high enough demand to be a common practice.

Sampling bias is one of the most canonical critiques of the raster model. In many use cases, each cell in a raster grid represents its center. With squares, the most ideal shapes in the traditional raster model, this introduces a slight bias in statistical calculations (Birch et al., 2007). In a square-shaped cell, edge distances from the center vary. Diagonals are approximately 40% farther than vertical or horizontal lines, resulting in a slight distortion. Samples towards to cell corners have a weaker relationship to the cell's center. Therefore, if samples are aggregated

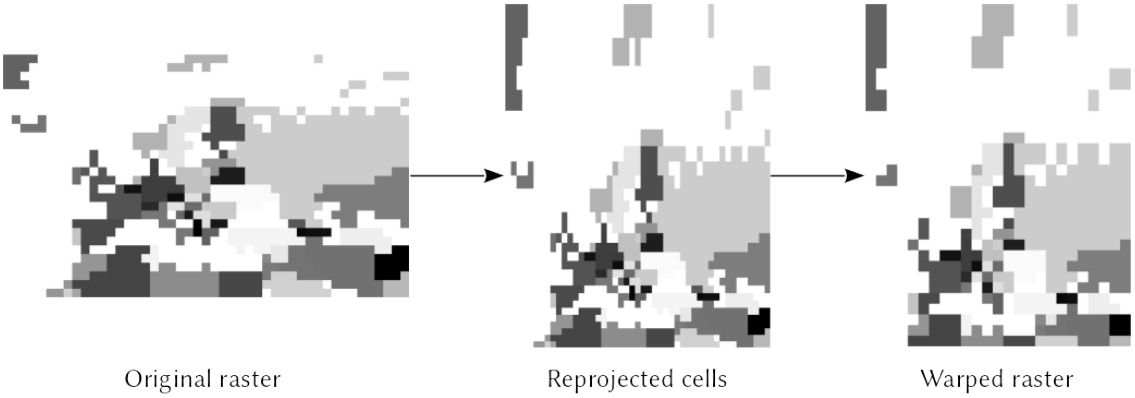


Figure 24: A raster warping process example on a Europe dataset with a resolution of 2 degrees. While some GIS applications (e.g. QGIS) have sophisticated algorithms to create the second visualization with on-the-fly image reprojection, a warped raster must have identical cells (Farkas, 2020).

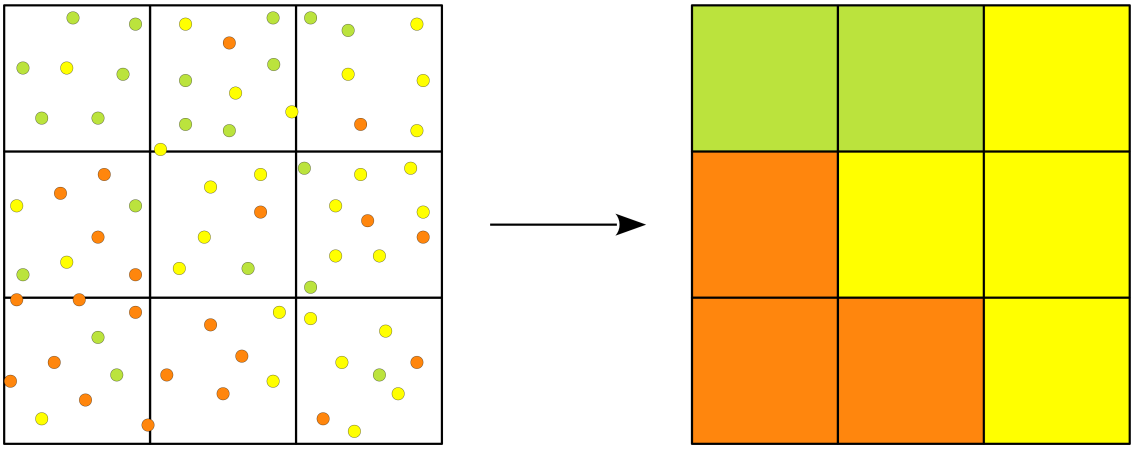


Figure 25: An example of sampling individual observations, and aggregating them to a raster grid. The aggregating method is mode, assigning the value of the majority of observations to each cell.

in cells (Figure 25), raster values will be biased due to the number of observations outside of the cell's inscribed circle (Figure 26). This is a typical argument for using hexagonal grids, since in a hexagon, biased area is only 9.31% compared to 21.46% in case of squares.

7.1.2. Treating rasters as vectors

It can be observed, most of the raster model's advantages are coming from its data model, while most of its limitations are of the representation model (Bugya & Farkas, 2018). If a new, more permissive representation model could be built, most of the model's disadvantages could would be mitigated. Since there are no mature and standard ways to represent rasters on the Web, an attempt was made to create such a new model. This model is called the coverage model, owing its name to OGC's WCS, which transmits raster data with similar considerations in mind.

Figure 26: Sampling bias with a square, and a hexagonal cell. Observations in lighter areas (inscribed circles) can be aggregated without concerns, since they equally represent the center of the cell. Observations in darker areas, however, are contributing to the overall bias of sampling.

The coverage model keeps the data model of rasters (i.e. uses matrices), but is renders that data as vectors. It treats every cell as a single polygon without a stroke style. Using this technique, the rendering process is slower, but each cell can be scaled, rotated, and projected easily. Furthermore, it allows for additional grid patterns. Its only requirement is to have an unequivocal mapping between matrix elements and coverage cells. The mapping is called a pattern, which allows translating and rotating successive elements. The pattern is valid if it is circular. The last transformation must result in the relative orientation and position of the first cell.

A coverage pattern has two major components. There are two arrays of transformations (offsets and rotations) for rows and columns. Row transformations are applied to successive cells in a row, while column transformations are applied to the first cell in each row. There can be an arbitrary number of transformations in each array, as long as the pattern is circular. Otherwise, the coverage will break its continuous tessellation eventually. Transformations in an array are applied to cells according to their position in the data matrix. For example, if there are two row transformations and five cells in a row, the first cell gets no transform, the second cell gets the first transform, the third cell gets the second transform, the fourth cell gets the first transform again, and the fifth cell gets the second transform again. Transformations are additive, which means, they are always applied to the previous cell, not the first one. Column transformations are applied to the first cell of the previous row.

The three regular tessellations: triangular, rectangular, hexagonal (Carr et al., 1992) form special cases in the coverage model. Since there was a high demand for creating such sampling grids, these tessellations are supported by major GIS software in the form of vector tools (Ramakrishna et al., 2013). Just like in those tools, a pattern can be automatically generated based on some simple parameters. Rectangular and hexagonal grids, the two most popular tessellations have the least degrees of freedom (Figure 27). A rectangular pattern has two offsets, a cell width

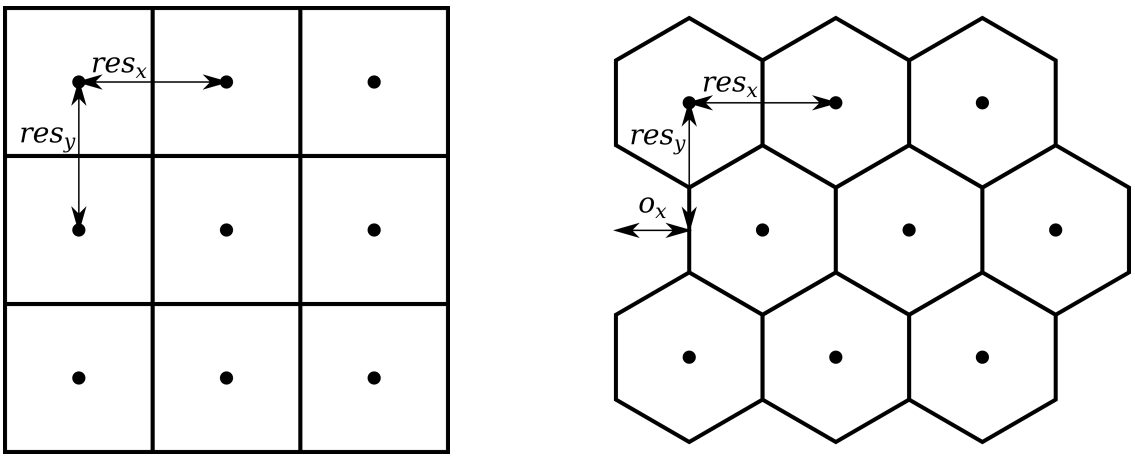


Figure 27: A rectangular and a hexagonal coverage pattern. Resolution (res_x, res_y) defines cell offsets in rows and columns, while an additional offset parameter (o_x) defines the offset of every second row in a hexagonal coverage (Bugya & Farkas, 2018).

for rows, and a cell height for columns. A hexagonal pattern needs an additional offset, since every even or odd row is offset in the pattern. This offset, however, can be calculated from the resolution of a cell, and therefore, only a sign is needed as a parameter. The sign decides if the coverage is odd or even (odd rows or even rows are offset).

Since patterns need to be circular, the hexagonal column pattern needs two transformations. If going from bottom to top in an even coverage, the first transformation shifts the next cell on the vertical axis by the res_y. It also translates the cell on the horizontal axis by o_x, which is $res_x \div 2$. The next transformation in the column pattern also shifts the cell on the vertical axis by res_y, but it translates it on the horizontal axis by $-o_x$. While the orientation of cells (flat-topped or point-topped) could add an additional degree of freedom, one can be transformed into the other by rotating and offsetting the whole grid.

The third regular tessellation, triangular pattern has the most degrees of freedom. As cells must change their orientations repeatedly in the pattern, it has more than one valid mappings to the underlying matrix (Figure 28). In a single mapping, the first triangle can face up or down, and there are also even and odd versions of the same coverage, similarly to hexagons. Due to the complexity of triangular patterns, they are less popular than rectangular and hexagonal ones (Birch et al., 2007), although they are good examples for coverage patterns (Program 8).

The coverage model also permits custom cell shapes. Those cells must be defined in a normalized coordinate system spanning over the bounding box of the cell (Figure 29). The resolution of the coverage, in this case, represents the width and height of the bounding box. The centroid of the cell is not the centroid of the polygon, but the centroid of its bounding box. By defining custom coverages, it is the user's responsibility to provide a pattern which results in continuous coverage.

From the examples above, it is obvious, that many patters do not have a well-defined lower left point, contrary to rectangular coverages. Since storing only an

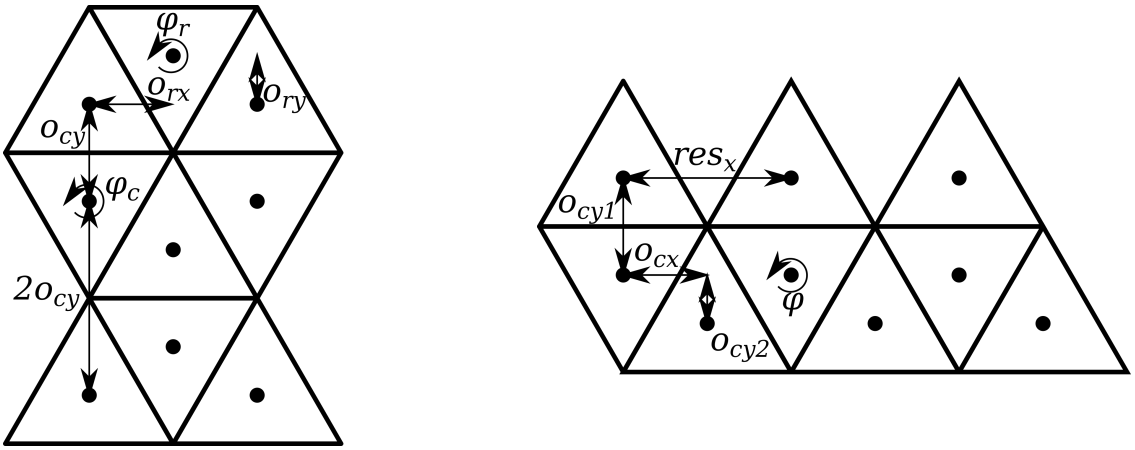
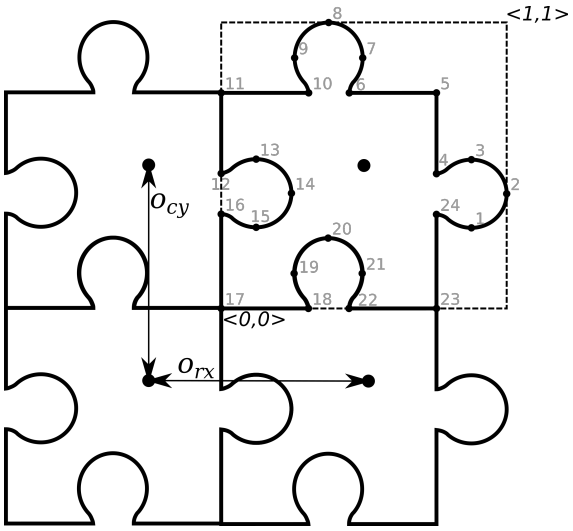


Figure 28: Two possible triangular coverage patterns laying out a 3 × 3 matrix. The first one needs offsets and rotations in both the row and the column transforms. The second one needs less parameters, but column transformations are more complex. (Bugya & Farkas, 2018).

```
{
    origin: [46.07, 18.21],
    row_pattern: [
        {
            rotation: 180deg,
            offset: [0.5, 1/3]
        },
        {
            rotation: 180deg,
            offset: [0.5, -1/3]
        }
    ],
    col_pattern: [
        {
            rotation: 180deg,
            offset: [0, 4/3]
        },
        {
            rotation: 180deg,
            offset: [0, 2/3]
        }
    ]
}
```

Program 8: An example of a pattern definition using a JSON style syntax tailored for a triangular coverage made of regular triangles. Offsets are unitless, and represent ratios to the resolution of a cell. Horizontal offsets are related to the width of cells, while vertical offsets are related to the height of cells (Bugya & Farkas, 2018).



ID	X	Y
1	0.9	0.3
2	1	0.4
3	0.9	0.5
4	0.95	0.47
5	0.75	0.75
6	0.45	0.75
7	0.5	0.88
8	0.37	1
9	0.25	0.88
10	0.3	0.75

Figure 29: An arbitrary pattern described with a minimal number of attributes. Coordinates of the shape are stored in a normalized coordinate system, therefore it can be linearly scaled with the resolution of the coverage. As this pattern is based on a rectangular coverage, only one offset is needed for cells in the same row (o_{rx}), and one for new rows (o_{cy}) (Bugya & Farkas, 2018).

origin in a raster is beneficial, the coverage model defines the origin as the center of the lower-left cell's bounding box. This makes the origin point independent from both the pattern and the resolution. While this technique was allowed by old specifications, like the ArcInfo ASCII Grid format (Yu & Custer, 2006), it never became popular due to the nature of traditional rasters. Furthermore, by using centroids as reference points, offsets can be applied to centroids, and only rotations need to consider whole cells. As cells are rotated around their bounding boxes' centroids, if users provide cell shapes which do not share their centroid with their bounding boxes' centroids, they have to correct rotational displacements in the offsets.

With the coverage model, rasters are treated more naturally, as edge cases of the vector model. They can be optimized based on the regularity of the pattern (Table 23). While the pattern is rectangular, it maintains every advantage of the traditional raster model and is burdened with all of its limitations. Hexagonal coverages lift some of the disadvantages, while they can be optimized very well to provide fast processing (Her, 1995). With the regularity of the pattern decreasing, there are fewer advantages and more limitations, ending up with a limited, but assuredly continuous vector layer in the end.

7.2. Traditional raster management

The first step in implementing a working raster management pipeline was creating the basis for both raster and coverage models. Those classes and functions are mostly related to traditional raster management, since the coverage model only changes the representation model of rasters. Base classes include containers for

Common charactersitics	
Advantages	Easy to reproject
	Easy to rotate
	Variable cell size
Disadvantages	Slower than textures
Rectangular coverage	
Advantages	Easily convertible to textures
	Easy to interpolate and resample
	Georeferencing is unequivocal
Disadvantages	Sampling bias
Hexagonal coverage	
Advantages	Easy to interpolate
	Less sampling bias
	More neighbors, better coverage
Disadvantages	Hard to resample
	Slower than rectangular
	Hard to create textures
	No lower left corner
Custom coverage	
Advantages	Pattern can be tailored to use case
Disadvantages	Performance depends on pattern
	Hard to interpolate and resample
	Hard to create textures
	Georeferencing is not trivial

Table 23: Characteristics of the coverage representation model using different patterns (Bugya & Farkas, 2018).

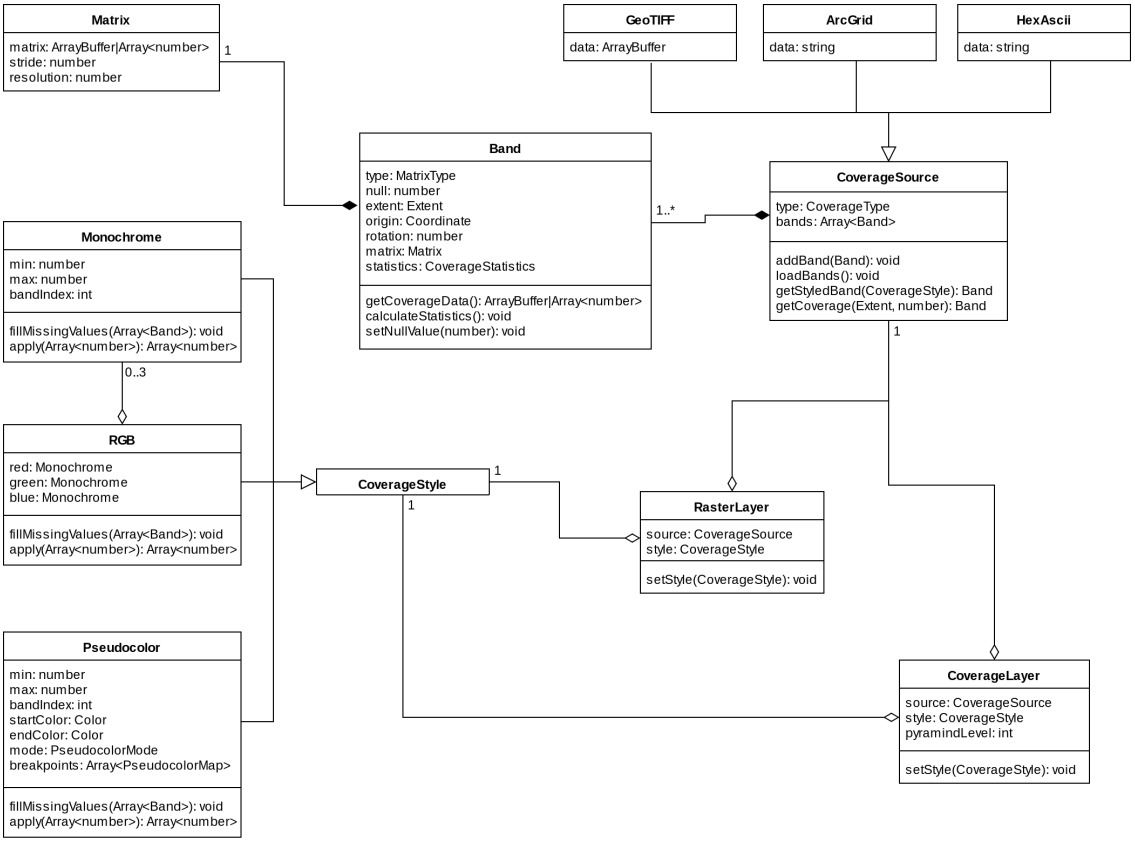


Figure 30: UML diagram of the most important classes related to the OpenLayers raster management implementation (Farkas, 2020). A larger version can be found in Appendix 4.

raster data, methods for styling, layers and sources for integrating rasters into the OpenLayers ecosystem, and renderers for visualizing the layers (Figure 30).

The utility functions absent from Figure 30 are related to aligning raster bands. While the current implementation does not deal with raster processing, there are use cases for aligning in general raster management. RGB images are often created by composing different bands in a single layer, or from multiple layers. Each raster layer or band is associated with a single channel, and as a result, an RGB visualization is created. Since there is no guarantee of different components of the RGB composition share the same extent and resolution, they might need preprocessing before composition.

Utility functions have the single purpose of aligning bands if they differ. If bands have a different row or column numbers, they extend smaller layers with new rows and columns filled with null values. If they have different resolutions, there is a resampling function, which resamples layers to the most common resolution from the inputs. Since resampling often needs interpolation, a simple nearest neighbor technique is implemented. It works on grid centroids and assigns a new value to every resampled grid centroid from the nearest original grid centroid. Finally, if the layers still do not share the same extent, there is a sub-cell offset between them. In

this case, a function translates every other grid to perfectly match the first one.

7.2.1. Base classes

There are two core classes for raster containers. The first one is the matrix, which stores raw raster data in a binary form, if possible. In order to maintain backward compatibility, it is capable to store raw data in a simple array. Since matrices are represented as arrays, the number of rows and columns also needs to be stored. The number of columns is stored as the stride of the matrix, from which the number of rows can be calculated. In JavaScript, binary arrays are represented as array buffers. They do not have types and such, therefore, they can be read by using a view. The view gives a type to the resulting typed array with a specified element size (e.g. 16-bit, 32-bit). Consequently, the numeric data type of the matrix also needs to be stored.

Since the matrix is a low-level concept, not intended to be used directly by users, there is a higher-level container, called a band. Bands act as interfaces storing most of the metadata associated with a single matrix. They know about the type of the underlying matrix stored in a typeless array buffer. They also know about the band's extent, origin, rotation, and can query the stride or raw matrix values from the underlying matrix class. As additional functionality, bands calculate some basic descriptive statistics from raw values. This is necessary for styling, since applying a default style on values with an arbitrary range requires a minimum and maximum value at a minimum. It also stores variance, sum, and count values for the underlying matrix.

Instantiations of the band class are needed to store individual bands in a source object. In order to reduce the number of required classes, a single source class was created for both raster and coverage representations. It is an abstract class, intended to be extended by different practical realizations. For traditional rasters, a GeoTIFF and an ArcGrid class was created, inheriting from the coverage source. While the ArcGrid class reads ASCII files, and parses them with a custom parser, the GeoTIFF class uses geotiff.js for parsing binary GeoTIFF files. Both of the classes create bands from parsed data.

Like in every layer type in OpenLayers, source objects need to be associated with layer objects. The raster layer class is a lightweight interface communicating between the source object and the renderer. It stores the style of the raster layer, and has a setter method for modifying it on the fly. Additionally, it only acts as a container for the source object.

Coverage styles were created in the scheme of existing vector styles. There are three styles for slightly different purposes. The monochrome style is for continuous greyscale representations. The pseudocolor style is for continuous or categorized color representations. Finally, the RGB style is for creating color composites from three different bands. These style classes are very simple, capable of storing parameters related to styling and applying styles on raw matrices. They also have a `fillMissingValues` convenience method, which fills missing parameters based on

band statistics.

The monochrome style is the simplest of the three, it only needs a minimum and a maximum value. Between those limits, it linearly interpolates every matrix value on a byte range. It clamps values outside of those limits to 0 and 255, represented with black and white, respectively. The RGB style is not much more complex than the monochrome. It needs three monochrome styles, one for every channel. When styling a raw matrix, it uses the same method as the monochrome style, but with three bands. The only major difference between them is null handling. The monochrome style sets the cell to fully transparent if it finds a null value. The RGB style only does this, if all of the channels have null values in a cell. Otherwise, it sets the null channel's value to 0.

The pseudocolor style is the most complex of the three, since it has more parameters and two different modes. In the interpolated mode, it linearly interpolates between two colors based on matrix values. As a result, a continuous transition is created between multiple discrete colors. In categorized mode, it assigns a single color to every cell between interval limits. For both modes, a start and end color are needed, while additional breakpoints can be inserted for more categories or transitions. During applying the style, the pseudocolor class creates intervals from limits, their colors, and breakpoints, and applies colors to matrix elements based on its mode.

7.2.2. The raster renderer

The last piece required for visualizing rasters is the renderer. Since an image layer renderer was already present in the source code, and the traditional raster model uses textures for visualizing rasters, there was no need to create an additional renderer. The raster layer simply extends the image layer, therefore they can use the same renderer in OpenLayers.

The image renderer uses the layer object to interface with its source object, containing data needed to visualize. First, it needs a `getImage` method on the image layer's source, which provides an OpenLayers image object. The image object is a container for an image with an event mechanism. In order to fulfill this criterion, the `CoverageSource` class can return a styled image from its raw raster data. For the styling, it receives the layer's style object, every time it gets updated. With the style object, it can style the raw data, and create an image from that.

Image creation involves three steps. First, an empty canvas is created with dimensions matching matrix dimensions from raw raster data. Next, the canvas is filled with pixels according to the cell colors of the styled raster band, received from the coverage source. In the end, the canvas represents the styled matrix, where every pixel corresponds to a single cell. Since cells do not have to be squares even in a traditional raster, the ratio of the raster layer's width and height are calculated. If the ratio is other than 1, the third step is executed. A new canvas is created according to the raster's width and height ratio, and the original canvas' content is stretched on it. This results in a raster texture with rectangular, non-square cells

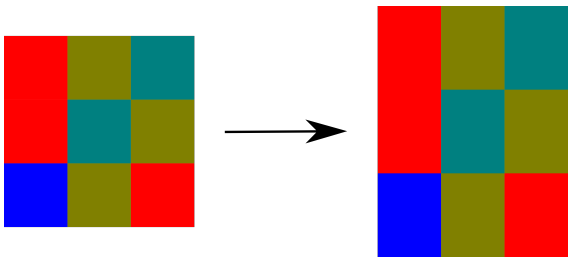


Figure 31: Creating traditional rasters of non-square cells is done by stretching an image with square cells. The performance of stretching is great, however, as everything – including interpolation – is done by the browser, it might introduce visual artifacts with categorized data (Farkas, 2020).

(Figure 31).

When the renderer gets the image, it caches it until the image gets outdated. To sign the renderer, that a new image should be queried, OpenLayers' revision counter was used. When a change occurs in the raster layer, which needs regenerating the texture, the revision counter is incremented. There is a special case in image rendering, which needed to be incorporated. OpenLayers has an image reprojection pipeline, which can transform images from a projection to another with triangulation. In order to support on-the-fly raster reprojection, this pipeline is used when the projection of the map and the raster layer differ.

To summarize the pipeline, users instantiate a raster layer with a GeoTIFF or an ArcGrid source. The source parses raw matrix data, and creates one or more bands from it. General statistical indices are calculated and stored in the bands. On rendering, the renderer asks the source through the layer to provide a styled image. The source applies a user-defined or a default style to the bands, creates an image from the styled raster, and reprojects the image, if necessary. It returns the image to the renderer, which caches it for better performance. Finally, the renderer stretches the raster image to its extent, resulting in a rendered raster layer. There is only one important step in raster management not discussed before. Pyramid building was not implemented for texture-based rasters, since both of the rendering engines have mipmapping capabilities.

7.3. Handling coverages

The coverage renderer has significantly more custom logic than the raster renderer. It uses the same set of base classes (e.g. layers, sources, styles), however, it has some peculiarities, only available as coverages. The most interesting one is a hexagonal coverage format as a source class. HexASCII (de Sousa & Leitão, 2017) is an ArcInfo ASCII Grid adaption for hexagonal rasters (Figure 32). It stores the matrix in ASCII format and some metadata in a header. The header provides necessary information for positioning and laying down the grid. These are the number of rows, the number of columns, the no data value, the center coordinates of the lower-left cell, and the length of a cell's side. Since it only allows regular hexagons, these

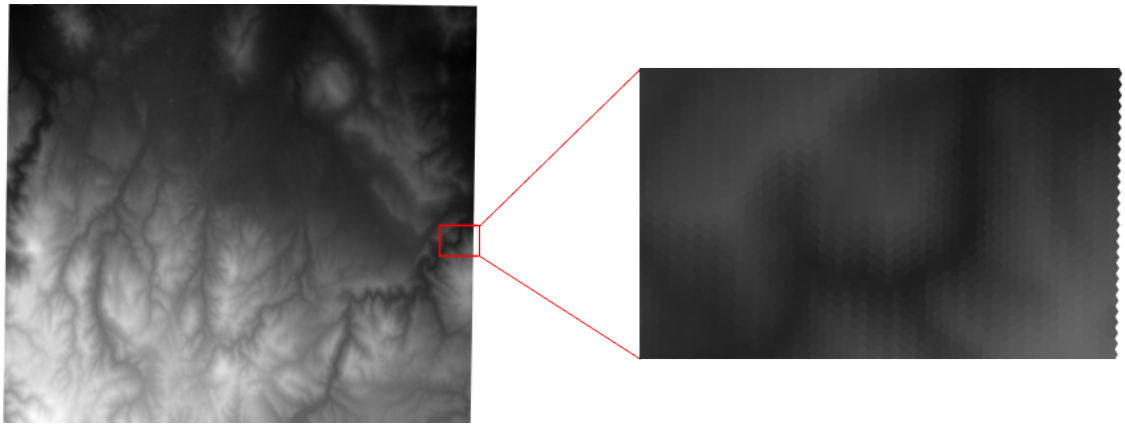


Figure 32: The Spearfish60 DEM rendered as a HexASCII coverage (Farkas, 2020).

parameters are sufficient for creating a hexagonal grid.

Besides the new source, three additional classes were created for the coverage renderer. A dedicated coverage layer was created, since renderers and layers are coupled in OpenLayers. It is only a simple container capable of storing the layer's style and some other basic properties. Additionally, two renderer classes were built. One for the Canvas engine, and one for the WebGL engine in order to find out if the Canvas engine is capable of visualizing rasters as vectors with a dedicated and optimized renderer.

The two coverage renderers have several shared capabilities. During the rendering process, both pyramids and R-Trees are used in an interconnected way. When a coverage layer is loaded, and a renderer is instantiated for it, a spatial index is built for every cell, on every pyramid level. By default, there are a maximum of 10 levels. In the end, every pyramid level contains an R-Tree with every cell indexed. From the R-Tree, the renderers can query cells in the map's viewport, and do not have to render every cell on larger scales.

Cells are generated in a utility function called `createGrid`. The function takes a styled band and some essential metadata (e.g. pattern, projection) as arguments. It not only creates a grid based on the coverage pattern, but it also generates an R-Tree, and places every cell in it. The R-Tree indexes cells with their centroids, speeding up calculations. There are two supported edge cases: rectangular and hexagonal. If the coverage is one of those types, the grid is generated automatically (Program 9). If not, it is treated as a custom coverage (Figure 33), and a cell shape is needed along with a coverage pattern.

In the R-Tree, color is also stored besides cell coordinates. Since the color has three 8-bit components, it is compressed to a single 32-bit value. This not only lowers memory footprint but also makes sorting cells based on colors easy. From this point, the two renderers behave differently. The canvas renderer sorts cells in the map's extent, since then, a single fill and stroke command are enough for a single color. This optimization affects performance mostly due to the need for cosmetic strokes. The canvas renderer cannot render adjacent cells seamlessly, when they are

Require: Cell shape S, Pattern P, First centroid C_0, Resolution R, R-Tree T
$S_R \leftarrow S$ scaled by R
$S_T \leftarrow$ triangulated S_R
do
    $\varphi \leftarrow$ current cell's rotation from P
    $S_i \leftarrow$ translate S_T by C_i
    $S_i \leftarrow$ rotate S_i around C_i by φ
    Add S_i to T
    $C_i \leftarrow C_i$ translated by next cell's offset from P
until there are no cells remaining

Program 9: Pseudocode for generating cells with custom coverages (Bugya & Farkas, 2018).

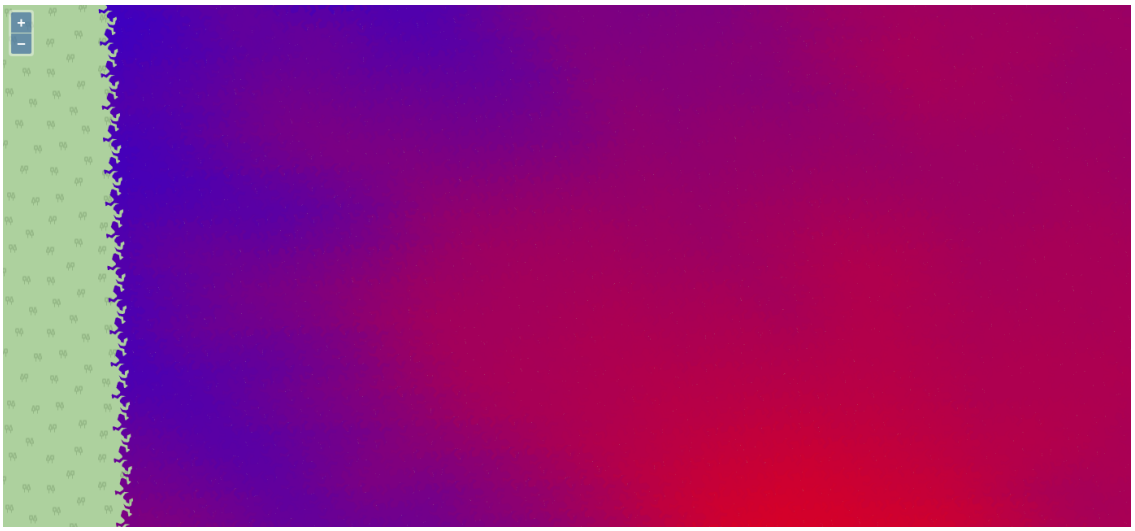


Figure 33: The Spearfish60 DEM visualized as a custom coverage using Escher geckos as cell shapes. The coverage uses a pseudocolor style interpolated from blue to red.

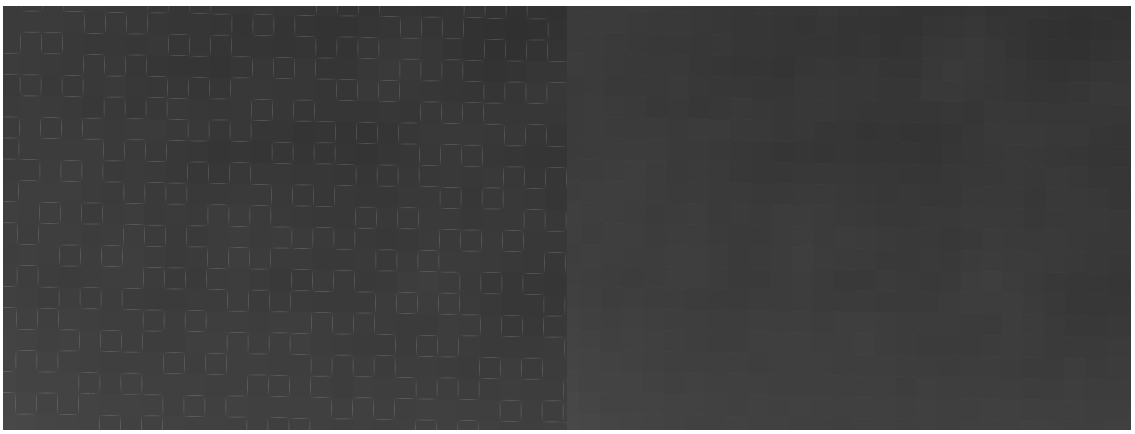


Figure 34: The same scene before (left) and after (right) applying a 1 pixel wide cosmetic stroke using the Canvas engine (Farkas, 2018).

slanted, and it can only be solved painlessly with a few pixels stroke (Figure 34).

The WebGL renderer uses a custom GLSL program with a custom replay. It was created solely to draw coverages, however, there were not much space for optimization. The replay assigns colors to vertices rather than using uniform variables. This speeds up drawing on the cost of additional memory consumption. The GLSL program's only optimization is decoding compressed colors in the GPU.

7.3.1. Hexagonal pyramids

The main problem with supporting hexagonal coverages was with building pyramids. Most of the functionality related to rectangular matrices could be adapted with relative ease, but hexagonal pyramid building has some peculiarities. While the method is simple for averaging hexagonal cell values for a next level (Figure 35), the coverage either shrinks or expands in successive levels.

The method extends on different approaches used in discrete global systems. These partitioning schemes concentrate on apertures, the ratio of cell area between successive levels (Sahr, 2013). From those schemes (Figure 36), some of them (e.g. aperture 3, aperture 7) require rotating cells in the next level, changing the orientation of successive levels. Since those are harder to compute and visually less aesthetic, they are not appropriate for building pyramids. Aperture 4 has the required rotational properties, and Sahr's aperture 4 is similar to the method used by the implementation. Both of them create new cells from 4 of the previous level, although the other aperture 4 partitioning uses more cells to construct a new one. This results in more averaging, and less precision, although the center of the previous level's central cell coincides with the centroid of the new cell. This property is more important for discrete global grids, and because of creating pyramids is for visualization purposes only, the new aperture 4 method (Figure 35) was used with visually more precise results.

Unfortunately, all of the aperture 4 solutions suffer from the same problem.

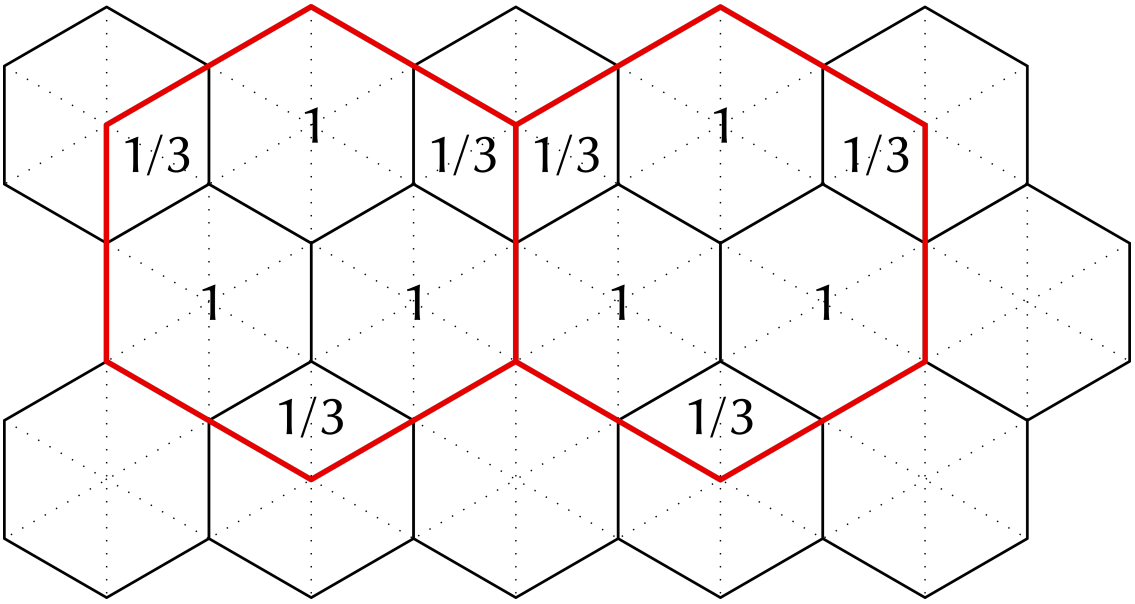
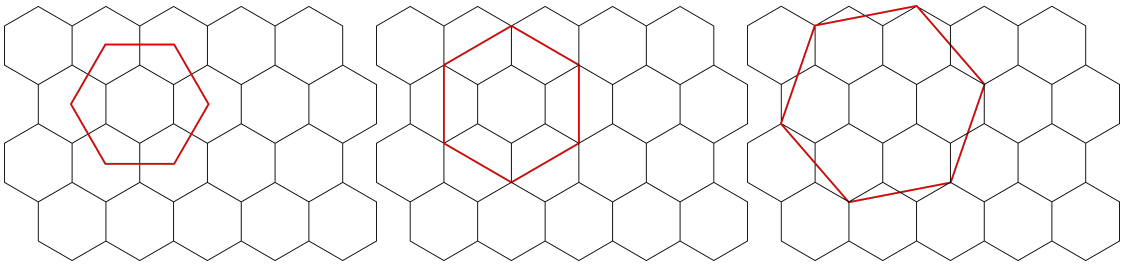
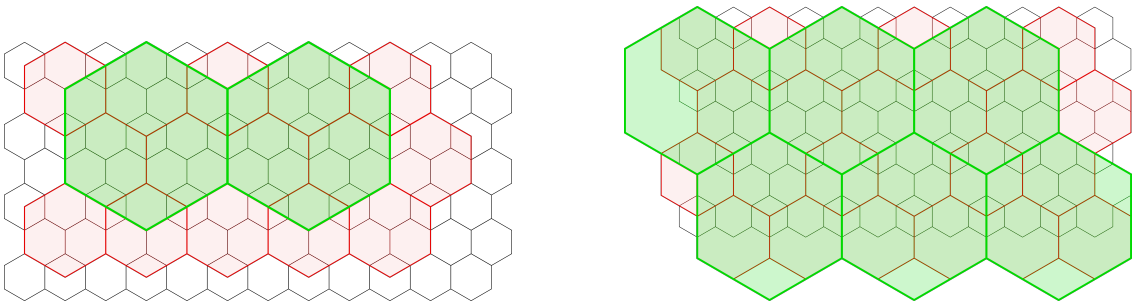


Figure 35: Method used for averaging cells for the next level of a hexagonal pyramids. Cells contributing to new cells in the generalized coverage are highlighted, and their weights are shown (Farkas, 2020).



(a) (b) (c)

Figure 36: A classical aperture 3 (a), aperture 4 (b), and aperture 7 (c) partitioning scheme. They are used for discrete global hexagonal grids, for which centroids of two cells in different resolutions must coincide. This is not true for pyramids.



(a) (b)

Figure 37: Example for a shrinking (a), and an expanding (b) hexagonal coverage. Both of the phenomena can only occur, when the pyramid building algorithm strives to only cut or extend the previous level (Farkas, 2020).

They are perfect for global grids, where they have a periodicity. On the other hand, coverage layers have edges, which need to be handled. Rectangular coverages do not have the same problem, since if they have an odd number of rows or columns, the pyramid building algorithm only has to decide between shrinking and extension once. After then, successive pyramid levels will be even. This is not the case with hexagonal coverages. Since building a new cell involves at least 6 cells from the previous level, there will always be cells, which cannot form a new hexagon entirely. Based on the implementation's choice of approach, successive levels will either shrink or expand (Figure 37).

No matter which approach is chosen, the resulting pyramids will be problematic. Independent of the technique, the number of cells will reach a minimum value, if the number of levels is not constrained. In case of shrinking, this value is defined by the ratio of rows and columns in a coverage. Expanding has a constant value of 1. When converging to these values, a pure shrinking approach can cut 60% of the original layer, while a pure expanding approach can add 50% more area to it. This is – in theory – geographically problematic, since on lower scales, the layer will span over significantly different areas, than on higher scales, distorting the visualized phenomenon. Fortunately, on scales where the shrinking or expanding causes a problem, the layer the differences in covered areas by successive pyramid levels can be hardly observed, not alone interpreted.

From the two effects, expanding might seem less severe than shrinking, however, a shrank layer keeps its numerical precision. Every cell is direct derivatives from cells in the previous level. This cannot be told about an expanded layer. The expansion effect is created by an algorithm, which takes cells beyond the original extent of the layer into account. Those cells can only have one value: null. By including more and more null cells into the equation, successive levels become less precise.

There are multiple possibilities to mitigate this problem. One can create an algorithm, which builds every level from the original grid. Alternatively, the algorithm could calculate if shrinking or expanding is the better approach for an edge

column or row in the next pyramid level. If the number of available cells are above a threshold limit, the algorithm expands the coverage, otherwise, it shrinks it. The current implementation, however, does not use any of these more sophisticated methods. It uses a pure shrinking approach for the best precision.

7.4. Benchmarking the pipelines

Some of the benchmarks were done during development to see which set of optimizations should be used for a feasible user experience. Similarly to vector rendering, due to OpenLayers' rendering mechanism, animating speed was more important than drawing speed. Furthermore, in this case, memory footprint also had significance. Since most of the optimizations applicable for coverage management increases performance by increasing its memory footprint, it was important to make some measurements about memory consumption.

7.4.1. Applied optimizations on coverage rendering

Using a spatial index for storing cells had the largest impact on performance, although it was not planned in the first implementation. The impact of this step was not thoroughly investigated, since before, both of the engines struggled with drawing coverages, and after, the WebGL engine became usable (Table 24). According to a single measurement, the Canvas engine took a boost of around 24%. On the other hand, the need for applying cosmetic strokes appeared. Since the Canvas renderer performed very poorly even with a small coverage, the extra 868.7 ms added to animating time made it completely unfeasible to use. Consequently, the Canvas engine was excluded from further measurements, and the development concentrated on hardware accelerated coverage rendering.

In the WebGL renderer, the first optimization was using vertex attributes for colors. While uniform attributes do not need as much memory as including a color for every vertex, the drawing speed becomes very slow with frequent color changing. With colors as vertex attributes, the memory cost quadrupled, however animating speed became faster by an order of magnitude. This is an acceptable exchange, since such a performance boost was needed to have a usable WebGL engine. With around 20 FPS during animations, drawing speed was less of a concern.

The final optimization was using pyramids to speed up rendering on smaller scales. This step mostly affected drawing speed, since during animations, vertex buffers are cached and replayed. By using pyramids, drawing speed could depend on the zoom level when the whole layer is in the map's extent, not only when only its portion can be seen. The animating performance was also increased, although FPS values over 20 were hardly differentiated.

	Redraw	Animate
Canvas engine		
Time	2202.1 ms	1678.6 ms
Performance	0.5 fps	1.7 fps
Memory	112.5 MiB	
WebGL engine (colors as uniforms)		
Time	3009.6 ms	410.2 ms
Performance	0.3 fps	2.4 fps
Memory	30.8 MiB	
WebGL engine (colors as vertex attributes)		
Time	2299.1 ms	48.1 ms
Performance	0.4 fps	20.8 fps
Memory	112.0 MiB	

Table 24: Performance and memory metrics of the two coverage renderers rendering the Spearfish60 DEM as a rectangular coverage on zoom level 12. All of the cases are using an R-Tree as a first optimization step, while the WebGL engine uses different number of vertex attributes as a second one (Farkas, 2018).

	Bands	Cells	Time	Memory
Parsing				
Spearfish60	1	302 418	376 ms	1.2 MiB
Baranya imagery	4	31 618 496	3435 ms	60.3 MiB
Styling				
Spearfish60	1	302 418	27 ms	–
Baranya imagery	3	23 713 872	779 ms	–

Table 25: Parsing and styling metrics of the two examined raster layers. Hexagonal coverages have the same metrics as rectangular ones, since the difference between them in these phases is a single attribute value (Farkas, 2020).

7.4.2. Rasters versus coverages

As discussed before, the whole raster management process have many common parts. Parsing raster layers, creating abstracted data representations, and applying styles are all common features of both of the renderers. It is only after styling, when the two branches (i.e. rasters and coverages) separate, and form their own rendering pipelines. While common steps are independent from the rendering engine, they are integral parts of the whole pipeline, therefore they were measured (Table 25).

According to results, those steps are optimized well enough. By using binary buffers for storing raw matrix data, the memory footprint could be decreased to a decent level compared to the number of stored cells. Parsing takes only a few seconds in case of large binary raster files, which is only a one time cost. Styling can occur multiple times, although it is relatively fast, and less frequent than redraws or animation frames. The memory footprint of styled rasters were not measured,

	Prepare time	Draw time	Memory
Raster layer			
Spearfish60	37 ms	3 ms	87 KiB
Baranya imagery	343 ms	8 ms	77.5 KiB
Coverage layer			
Spearfish60	2579 ms	1 – 1032 ms	152.9 MiB
Spearfish60 (hexagonal)	3001 ms	1 – 1747 ms	170.4 MiB

Table 26: Rendering metrics of raster and coverage layers. In case of coverage layers, the draw time is a function of the pyramid level, and the number of visible cells. The range limits are empirical best and worst case values (Farkas, 2020).

since they are temporary. Once the spatial index is created, the styled raster is dereferenced and garbage collected.

The next two steps, dependent on the type of the layer are preparing and drawing. Since the Canvas engine was previously excluded from measurements, only hardware accelerated raster and coverage rendering was assessed (Table 26). It is worth noting, raster layers can be used with the Canvas renderer effectively. Since they are texture-based layers, they can be overlaid to the map with the same performance as OpenStreetMap tiles. However, measuring the differences between Canvas and WebGL raster rendering would have added no additional value, since image rendering performance was already explored. Furthermore, differences between animation frames and complete redraws would have neither contributed much, since every case provided smooth animations.

The preparation phase in case of rasters and coverages is not the same as in the case of rendering. It groups steps related to creating and caching visualized data, even between complete redraws. Preparation time is not significant in the case of traditional rasters. Since there are two steps after styling (creating an image and optionally reprojecting it), and both of them are well optimized, even large rasters can be prepared quickly. Rendering is even faster, and the memory footprint of the cached image is minimal. The two steps combined, however, is not fast enough for creating continuous animations from large rasters by restyling the raster at every frame. It can be done with smaller layers, like the Spearfish60 DEM, though.

In the case of coverages, only the Spearfish60 DEM was measured, both as a rectangular, and a hexagonal coverage. Unfortunately, there were some memory-heavy steps in the pipeline, which caused the system to run out of available memory in case of the Baranya imagery. This indicates the inadequate scalability of the implementation, and its long way ahead to become more than a mere prototype.

Much difference could not be observed between rectangular and hexagonal coverages. Preparation times are higher than in the case of traditional rasters, due to the extensive process of creating pyramids with spatial indexing. Most of the time is spent on transforming (rotating, offsetting, reprojecting) individual cells. Those data structures also cause a large memory footprint compared to cached textures. On the other hand, drawing speeds are in most cases tolerable. The worst-case sce-

Level	Cells	Time	Memory	Heap memory
1	292 220	1032 ms	111.8 MiB	365 (+172) MiB
2	73 181	265 ms	31.0 MiB	193 (+17) MiB
3	18 230	96 ms	7.6 MiB	176 (+5) MiB
4	4 468	16 ms	1.9 MiB	171 (+6) MiB
5	1 026	18 ms	478.2 KiB	165 (+1) MiB
6	169	5 ms	114.1 KiB	164 (+0) MiB
7	63	3 ms	27.4 KiB	164 (+0) MiB
8	12	1 ms	5.7 KiB	164 (+0) MiB
9	2	1 ms	1.2 KiB	164 (+0) MiB

Table 27: Rendering metrics of different pyramid levels in the Spearfish60 rectangular coverage (Farkas, 2020). Heap memory was recorded, since on higher zoom levels, the browser ran out of memory during creating exact memory snapshots.

narios were measured when neither pyramids nor spatial indices could apply their beneficial effects on the scene.

By looking at the detailed metrics of different pyramid levels (Table 27), detailed versions not only need gradually more time to draw but also consume polynomially more memory. This is problematic, since the measured layer is a small one compared to typical real-world data. From the total heap memory of the application on different levels, the amount of memory consumed for rendering is put between parentheses. This is one of the most problematic parts, since that excess memory consumption comes from OpenLayers' rendering design, and not from the implementation's lack of scalability. It can be only reduced by emulating 64-bit floating operations on the GPU instead of using RTE rendering.

Learning from the lesson, there are many ways of optimizing the second part of coverage rendering further. First of all, the spatial index should be avoided, if possible. In the case of rectangular and hexagonal patterns, map coordinates can be transformed to row and column numbers in the matrix. In the case of custom coverages, however, there is no easy way to avoid building a spatial index.

As an alternative approach, spatial indexing could be applied with a better memory footprint. If every entry stores only cells' center coordinates along with a color, rectangular coverages could have a decreased memory footprint by 70%. This is the worst case, thus cells with more vertices would benefit even better. In this approach, however, the GPU must be able to create cell coordinates from a single center coordinate, otherwise, performance would significantly suffer. A possible solution would be creating custom vertex shaders for rectangular and hexagonal coverages, or as GLSL programs are provided to the WebGL API as strings, they could be generated according to the pattern.

8. Evolution and Impact

On top of the main results, the capabilities of OpenLayers were reassessed. This final assessment has two purposes; to give the thesis a frame, and to examine the future adequacy of the candidate library of being a complex and universal Web GIS basis. For this, the evolution of the library was taken into account over the past few years, more precisely, from the first assessment.

In the second half of 2019, a new major OpenLayers release went into production from beta testing. With OpenLayers 6, many changes were introduced. Some of them are directly related to this study, since they affect the library's rendering mechanism and the possibilities of hardware acceleration. While the new version offers more improvements for developers through API changes, the modified rendering mechanism alone is a prominent new feature.

From the full assessment methodology from Chapter 4, only GIS capabilities were reexamined. As other methods are focusing on choosing the right candidate library, they were not deemed necessary for reevaluation, as the candidate has been already chosen. The scoring scheme has not been changed, therefore the library got a score of 1 for a full implementation, a score of 0.5 for a partial implementation, or third-party plugin support, and a score of 0 for no support. The two implementations discussed in previous chapters were considered third-party extensions.

8.1. Changes in supported features

In the past years, the GIS capabilities of OpenLayers have risen by 4% to 60% of the examined features (Table 28). This increase is mostly due to the raster management implementation, discussed in Chapter 7. There are also minor changes in representation, and a major, but nontrivial improvement in rendering.

The rendering engine of OpenLayers went through numerous revisions between the two examined versions. While some of these revisions cannot be identified from the feature coverage table, the library achieved its most significant improvements in this area. The 10% increment in the library's scoring is from the added capability of rendering raw raster data on the client side, however, its WebGL engine took most of the changes during a lengthy development process with many iterations and revisions.

Not far after the extended WebGL renderer discussed in this study has been completed, it got integrated into OpenLayers. It became an experimental feature, since fresh out of the oven, it was only tested with unit tests and basic rendering

Feature group	OpenLayers (3)	OpenLayers (6)
Rendering	60%	70%
Formats	76%	82%
Database	0%	0%
Data	44%	50%
Projection	88%	88%
Interaction	72%	72%
Representation	56%	56%
Average	56%	60%

Table 28: GIS feature coverage of the latest release of OpenLayers (6.0.1) compared to the one from the original study (3.17.1). Detailed results can be seen in Appendix 5.

tests. With only one maintainer (the developer of the code), bug fixes were slow, and there were little to no new features added. As the engine had only basic rendering capabilities, and there were no prospects of extending it to the Canvas renderer's level, it had been discarded with version 6.

The improvement in this step is not obvious. The developers did not drop the WebGL renderer without compensations. They restructured the whole rendering system, making it the most versatile one among similar libraries. Previously, the rendering engine was tied to the view, therefore, every layer was forced to use a single technology. With the new structure, rendering technology is tied to a layer, therefore, users can choose their preferred engine on a per-layer basis.

One of the advantages of such design is now users are allowed to use hardware accelerated rendering with large layers in a GIS workflow, and can choose the Canvas renderer with more capabilities for cartographic use in the same map. They can process large layers, filter the number of features, generalize geometries with WebGL, and create maps from the smaller results with the Canvas engine. From a GIS perspective, this allows for integrating an editor and a cartographic composer in the same application seamlessly.

Furthermore, per-layer renderers allow for integrating various third-party mapping technologies in the same application. Users can include D3 visualizations and hardware accelerated layers in the same map. This versatility allows for arbitrary extensions by interfacing with OpenLayers, which can be very useful in a universal Web GIS application.

Regarding format handling (Table 29), the scoring of OpenLayers also increased due to the raster manager extension. By implementing various data exchange formats, two format requirements (GeoTIFF and Arc/Info ASCII Grid) have been fulfilled. As the implementation was designed to be used with various Web services, WCS was also implemented, which can be used with both GeoTIFF and Arc/Info ASCII Grid, completing a third criterion.

Changes in the Data category (Table 30) were also due to new raster capabilities. The raster manager can expose its raw matrix content, which can be manipulated with simple algorithms. Therefore, trivial raster manipulation processes, like

Format	OpenLayers (3)	OpenLayers (6)
Vector	90%	90%
Raster	17%	50%
Image	100%	100%
Tile service	83%	83%
Average	76%	82%

Table 29: Data exchange format support of the latest release of OpenLayers (6.0.1) compared to the one from the original study (3.17.1). Detailed results can be seen in Appendix 5.

Data	OpenLayers (3)	OpenLayers (6)
Pre-process	63%	63%
Conversion	0%	0%
Manipulation	67%	67%
Analysis	25%	44%
Average	44%	50%

Table 30: Data management feature support of the latest release of OpenLayers (6.0.1) compared to the one from the original study (3.17.1). Detailed results can be seen in Appendix 5.

raster modification, raster algebra, and classification were considered partially implemented. Convolution was not included in the supported processing features, as it would need larger amount of custom code to implement without helper functions.

There were two changed functionalities affecting examined features, which did not change the score of the library. Styling rasters was partially implemented in the original version by accessing raw image pixel values, and the new raw raster styling was also considered a partial fulfillment. The other changed capability was the overview map, which became more stable in the new version. However, as the new version cannot synchronize itself with map layers, it was still considered a partial support.

While the overall improvement was only 4%, OpenLayers became more capable as a basis of a universal Web GIS application. By using the raster module, and hardware accelerated vector rendering, one can now build an effective GIS application, which can work in a cloud architecture, and does not require any installed GIS software or any particular platform. Such applications can adapt themselves to the used platform. Apart from the usual responsive Web design, for example, they can limit their processing capabilities on handheld devices during a field survey for extended battery life. They can also use the client's machine for calculations below server scale for a more effective server-client balance. Finally, these improvements do not only affect universal Web GIS applications, as specialized software can also benefit from both raster and coverage support (e.g. using hexagonal rasters).

9. Conclusions

This thesis has explored the possibility of building a universal Web GIS software using existing components. Since no feasible solution was found for this goal, some of the most crucial features were implemented in the most appropriate library, OpenLayers. The features deemed necessary for a universal client, but missing from OpenLayers were hardware accelerated vector rendering and raster management. With those features implemented, there is now a feasible stack for creating better Web GIS clients.

Since the thesis consists of several diverse parts, a more elaborate set of conclusions seems appropriate, one for each part. During the search for the best foundation for such a client, several approaches were examined. The analysis remained metrical, there was no expert survey included. From the results, it seems like a set of static software metrics along with some softer ones related to documentation, community, and other characteristics can be enough for assessing different JavaScript libraries.

Furthermore, a new metric, Approximate Learning Curve for JavaScript was created and used. By correlating to other, more interactive and survey-based results along with personal experience, it seems like the metric can do what it was designed for. It can roughly approximate the learning curve of a JavaScript library. It cannot be used for distinguishing between similarly complex libraries, although it seems appropriate for detecting outliers (e.g. too complex, too simple).

While many libraries were assessed, only a few were selected, mostly according to their GIS functionalities. There are many capable spatial data visualization libraries, both in the 2D, and the 3D areas, however only a few were created with cartographic and GIS aspects in mind. Most of the libraries can be used for various data analysis workflows, charting, or creating simple static maps, but they would require an excessive amount of work to form them into a universal Web GIS.

It might seem a little surprising, that two virtual globes made it into the group of candidates. It is, however, a plausible ratio, since a virtual globe requires at least as many spatial functionality as a web mapping library, although with better-optimized solutions. The two candidate virtual globes also had some GIS functionality, with Cesium being the more capable. It would not be much of an exaggeration to state that Cesium is the most capable of all candidates. However, it has serious shortcomings from the aspect of GIS, like its lack of proper projection support.

Creating a hardware accelerated rendering engine is not a trivial task. There are many pitfalls, if the solution needs to be both fast and general. It is presumed,

that some of the major obstacles were not even encountered. However, a basic renderer could be created, which can outperform the current one with large, arbitrary datasets. While the WebGL implementation might be not as suitable for cartographic purposes as the Canvas renderer, it can be used for GIS workflows and basic spatial visualizations.

Due to the extended capabilities of the Canvas renderer, it should be used, if the visualized layers can be optimized. It is suitable for vector tiles when each zoom level can be sufficiently generalized on the server-side, but the client still has the opportunity to render and style vector graphics. For small maps in other data exchange formats (around 2000 features or 60 000 vertices), the Canvas renderer is sufficient, the results are smoother, and there are more styling options than in the current WebGL renderer. In cases when the Canvas renderer is not feasible anymore (e.g. big data, animated vectors), the WebGL renderer offers a usable alternative.

Presumably, the most significant part of this thesis is revisiting raster management. Desktop solutions rely on mature libraries with a long history, and very few bugs. These libraries (e.g. GDAL) can do such great work in raster processing, swapping them for a new component in order to support a modern concept does not seem feasible. On the other hand, Web technologies are still young, and due to different constraints, most traditional problems require new solutions. This makes the Web a great boilerplate for testing out new concepts. If a technique, an approach, or an application works out well, it might be of greater interest for implementing in other environments.

An example of such a concept is the coverage model. While the raster model's popularity is understandable due to its advantages – especially for working with spatially continuous data – it has severe limitations. While demands for alternatives were not strong enough for revisiting such a stable concept, recently, the increasing popularity of hexagonal rasters spawned efforts for extending on the traditional raster concept. For example, core developers of GRASS GIS found Luís Moreira de Sousa's idea of a hexagonal raster implementation with a warm welcome.

Since it is now possible to create a new raster concept due to increased demand, investigating the feasible degree of generalization is the correct approach. Since then, the extended model will be more stable, and it is less likely that it will need further revisions in the near future. The coverage model offers such an investigation by generalizing the raster model to the level of vectors. On the other hand, it still keeps rectangular and hexagonal coverages as edge cases, which can be better optimized than custom ones.

Preliminary results showed that while the coverage model is not feasible to use for real-world data, it can be shaped into a working library. The proof of concept demonstrated, hexagonal coverages can be handled without using verbose vector data structures, and maintaining complex topological relationships. Furthermore, coverages should not replace rasters, as they will never be as fast as textures. The coverage model should complement the raster model, offering a hybrid solution for professionals working with more complex coverages. On mobile devices and embedded systems, where processing power and memory is limited, or battery discharge

time is a relevant factor, rasters will be a better solution in the foreseeable future, than rectangular coverages.

The coverage model's lack of scalability is mostly a matter of optimization, which points at future works. Both the raster management and the WebGL renderer implementations should be optimized better. The coverage pipeline should have a more efficient caching mechanism, while the vector renderer pipeline should have better graphical quality. When the coverage renderer can stand as a feasible proof of concept for professional use, its practicality can be seen.

As a more imminent future task, both of the extensions will be recreated as modules. The main reason behind this is OpenLayers' revised rendering pipeline. With OpenLayers 6, the library will have layer renderers, allowing users to mix Canvas and WebGL powered layers. On this, the WebGL renderer will be recreated as a third-party module, along with the raster management parts. The two separate modules will only be improved after they are successfully reimplemented. This is the main reason behind concrete examples of using the implementations are mostly excluded from this work. Still, the coupled version of the raster manager can be reached at GitHub (Farkas, 2018a). The vector implementation was part of OpenLayers until version 6.

Acknowledgements

First, I would like to thank my advisor, Dr. Titusz Bugya, for helping me in the past years. His tutoring, ideas, and remarks significantly influenced the quality of the whole research behind this thesis, for the better. He always tried to point me in a better direction, with more outcomes, generality, applicability, and future. He never let me stuck in a loophole, and made serious efforts to guide me to the best path for an aspiring researcher.

I would like to thank Dr. Zoltán Dövényi, former head of the Doctoral School of Earth Sciences for his help during my PhD studies. He never ceased to inspire me by helping me reach grants, standing up for me, or just having a good conversation with me. I also thank Mónika Kovács, the coordinator of the same school, for taking the toll of administration of me, and for kindly helping me with all of my problems and questions during my studies.

I own serious gratitude to Jørn Watvedt and András Hervai for having me working on Cadify. During my time on the project, I gained valuable first-hand experience on how a modern CAD software relates to a GIS by programming CAD solutions using SolidWorks' API.

The months of writing this thesis burdened me down to a great extent. I am grateful to Dr. István Geresdi, director of the Institute of Geography, and to Dr. Péter Gyenizse, head of the Department of Cartography and Geoinformatics, for letting me move away from some of my academic and institutional duties for this time. I am also grateful to all of those who took over some of my works and responsibilities in the period of writing this thesis.

Implementing raster management was supported by the ÚNKP-17-3-I New National Excellence Program of the Ministry of Human Capacities, Hungary. Some publications on the research covered by this thesis have been supported by the European Union, co-financed by the European Social Fund Grant no.: EFOP-3.6.1.-16-2016-00004 entitled by Comprehensive Development for Implementing Smart Specialization Strategies at the University of Pécs.

References

Agrawal, S., & Gupta, R. D. (2014). Development and Comparison of Open Source Based Web GIS Frameworks on WAMP and Apache Tomcat Web Servers. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XL*(4), 1–5.

Albrecht, J. (1998). Universal analytical GIS operations – a task-oriented systematization of data structure-independent GIS functionality. *Geographic information research: Transatlantic perspectives*, 577–591.

Antoniou, V., Morley, J., & Haklay, M. M. (2009). Tiled vectors: A method for vector transmission over the web. In *International symposium on web and wireless geographical information systems* (pp. 56–71).

Battersby, S. E., Finn, M. P., Usery, E. L., & Yamamoto, K. H. (2014). Implications of web Mercator and its use in online mapping. *Cartographica: The International Journal for Geographic Information and Geovisualization*, *49*(2), 85–101.

Batty, M., Hudson-Smith, A., Milton, R., & Crooks, A. (2010). Map mashups, Web 2.0 and the GIS revolution. *Annals of GIS*, *16*(1), 1–13.

Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). The R*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, *19*(2), 322–331.

Behrens, J. T. (1997). Principles and procedures of exploratory data analysis. *Psychological Methods*, *2*(2), 131–160.

Berners-Lee, T. J. (1989). *Information management: A proposal* (Tech. Rep. No. CERN-DD-89-001-OC). CERN.

Birch, C. P., Oom, S. P., & Beecham, J. A. (2007). Rectangular and hexagonal grids used for observation, experiment and simulation in ecology. *Ecological Modelling*, *206*(3), 347–359.

Bostock, M., Ogievetsky, V., & Heer, J. (2011). D^3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, *17*(12), 2301–2309.

Bugya, T., & Farkas, G. (2018). An Alternative Raster Display Model. In C. Grueau, R. Laurini, & L. Ragia (Eds.), *Proceedings of the 4th international conference on geographical information systems theory, applications and management (gistam 2018)* (pp. 262–268).

Butler, H., Gillies, S., & Schaub, T. (2016). *The GeoJSON Format* (Tech. Rep. No. RFC 2946). IETF.

Cabanier, R., Mann, J., Munro, J., Wiltzius, T., & Hickson, I. (2015). *HTML Canvas 2D Context* (Tech. Rep.). W3C.

Carr, D. B., Olsen, A. R., & White, D. (1992). Hexagon mosaic maps for display of univariate and bivariate geographical data. *Cartography and Geographic Information Systems*, *19*(4), 228–236.

Chrisman, N. (1987). Fundamental principles of geographic information systems. In *Proceedings of auto-carto* (Vol. 8, pp. 32–41).

Christen, M., Nebiker, S., & Loesch, B. (2012). Web-based large-scale 3D-geovisualisation using WebGL: the OpenWebGlobe project. *International Journal of 3-D Information Modeling (IJ3DIM)*, *1*(3), 16–25.

Cozzi, P., & Ring, K. (2011). *3D engine design for virtual globes*. CRC Press.

Curran, K., & George, C. (2012). The future of web and mobile game development. *International Journal of Cloud Computing and Services Science*, *1*(1), 25–34.

de Sousa, L. M., & Leitão, J. P. (2017). HexASCII: A file format for cartographical hexagonal rasters. *Transactions in GIS*, *22*(1), 217–232.

Di Staso, U., Soave, M., Giori, A., Prandi, F., & De Amicis, R. (2016). Heterogeneous-resolution and multi-source terrain builder for cesiumjs webgl virtual globe. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, *10*(1), 129-135.

Doyle, A. (2000). *OpenGIS Web Map Server Interface Implementation Specification Revision 1.0.0* (Tech. Rep. No. 00-028). OGC.

Ecma International. (2015). *ECMAScript® 2015 Language Specification* (Tech. Rep. No. ECMA-262). ECMA.

Eugene, G. Y., Di, L., Rahman, S., Lin, L., Zhang, C., Hu, L., ... Yang, G. (2017). Performance improvement on a Web Geospatial service for the remote sensing flood-induced crop loss assessment web application using vector tiling. In *2017 6th international conference on agro-geoinformatics* (pp. 1–6).

Evans, J. D. (2002). *OWS1 Web Coverage Service (WCS) Version 0.7* (Tech. Rep. No. OGC 02-058). OGC.

Farkas, G. (2015). *Comparison of Web Mapping Libraries for Building WebGIS Clients* (Unpublished master's thesis). University of Pécs, Pécs, Hungary.

Farkas, G. (2016). *Mastering OpenLayers 3*. Packt Publishing, Birmingham, UK.

Farkas, G. (2017a). Applicability of open-source web mapping libraries for building massive Web GIS clients. *Journal of Geographical Systems*, *19*(3), 273–295.

Farkas, G. (2017b). *Practical GIS*. Packt Publishing, Birmingham, UK.

Farkas, G. (2018). Towards visualizing coverage data on the Web. In *Az elmélet és a gyakorlat találkozása a térinformatikában ix.: Theory meets practice in gis.* (pp. 107–113).

Farkas, G. (2019). Hardware-Accelerating 2D Web Maps: A Case Study. *Cartographica*, *54*(4), 245–260.

Farkas, G. (2020). Possibilities of using raster data in client side Web maps. *Transactions in GIS*, *24*(1), 72–84.

Felzenszwalb, P. F., & Huttenlocher, D. P. (2012). Distance transforms of sampled functions. *Theory of computing*, *8*(1), 415–428.

Feng, X., Shen, J., & Fan, Y. (2009). REST: An alternative to RPC for Web services architecture. In *2009 first international conference on future information networks* (pp. 7–10).

Fenton, N. E., & Neil, M. (1999). Software metrics: successes, failures and new directions. *Journal of Systems and Software*, *47*(2–3), 149–157.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Gaffuri, J. (2012). Toward web mapping with vector data. In *International conference on geographic information science* (pp. 87–101).

Ganovelli, F., Corsini, M., Pattanaik, S., & Di Benedetto, M. (2014). *Introduction to computer graphics: a practical learning approach*. CRC Press.

Gede, M. (2015). Novel Globe Publishing Techniques Using WebGL. *e-Perimetron*, *10*(2), 87–93.

Gong, L., Pradel, M., & Sen, K. (2015). JITProf: pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 357–368).

Goodchild, M. F., Yuan, M., & Cova, T. J. (2007). Towards a general theory of geographic representation in GIS. *International journal of geographical information science*, *21*(3), 239–260.

Grigorik, I., Mann, J., & Wang, Z. (2016). *Performance timeline level 2* (Candidate Recommendation). W3C.

Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, *15*(4), 287–317.

Haklay, M., Singleton, A., & Parker, C. (2008). Web Mapping 2.0: The Neogeography of the GeoWeb. *Geography Compass*, *2*(6), 2011–2039.

Hearn, D., & Baker, M. P. (2004). *Computer graphics with OpenGL*. Pearson Prentice Hall.

Held, M. (2001). FIST: Fast industrial-strength triangulation of polygons. *Algorithmica*, *30*(4), 563–596.

Her, I. (1995). Geometric transformations on the hexagonal grid. *IEEE Transactions on Image Processing*, *4*(9), 1213–1222.

Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E. D., O'Connor, T., & Pfeiffer, S. (2014). *HTML5* (Tech. Rep.). W3C.

Ingensand, J., Nappez, M., Moullet, C., Gasser, L., Ertz, O., & Composto, S. (2016). Implementation of Tiled Vector Services: A Case Study. In *Sdw@ giscience* (pp. 26–34).

Jayathilake, D., Perera, S., Bandara, S., Wanniarachchi, H., & Herath, L. (2011). A technical insight into community Geographic Information Systems for smartphones. In *2011 ieee international conference on computer applications and industrial electronics (iccaie)* (pp. 379–384).

Jones, E. L. (2001). Metrics based plagarism monitoring. *Journal of Computing Sciences in Colleges*, *16*(4), 253–261.

Kaur, G., & Fuad, M. M. (2010). An evaluation of protocol buffer. In *Proceedings of the ieee southeastcon 2010 (southeastcon)* (pp. 459–462).

Kaur, K., Minhas, K., Mehan, N., & Kakkar, N. (2009). Static and dynamic complexity analysis of software metrics. *World Academy of Science, Engineering and Technology*, *56*, 159–161.

Konde, A., & Saran, S. (2017). Web enabled spatio-temporal semantic analysis of traffic noise using CityGML. *J Geomatics*, *11*(2), 248–259.

Krämer, M., & Gutbell, R. (2015). A case study on 3D geospatial applications in the web using state-of-the-art WebGL frameworks. In *Proceedings of the 20th international conference on 3d web technology* (pp. 189–197).

Król, K. (2018). Comparative analysis of the performance of selected raster map viewers. *Geomatics, Landmanagement and Landscape*, *2*, 23-32.

Król, K., & Szomorowa, L. (2015). The possibilities of using chosen jQuery JavaScript components in creating interactive maps. *Geomatics, Landmanagement and Landscape*, *2*, 45–54.

Kulawiak, M., Dawidowicz, A., & Pacholczyk, M. E. (2019). Analysis of server-side and client-side Web-GIS data processing methods on the example of JTS and JSTS using open data from OSM and geoportal. *Computers & Geosciences*, *129*, 26–37.

Lee, D.-T., & Schachter, B. J. (1980). Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences*, *9*(3), 219–242.

Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., ... Wolff, S. S. (1997). The past and future history of the Internet. *Communications of the ACM*, *40*(2), 102–109.

Leung, C., & Salga, A. (2010). Enabling webgl. In *Proceedings of the 19th international conference on world wide web* (pp. 1369–1370).

Liktor, G., & Dachsbacher, C. (2013). Decoupled Deferred Shading on the GPU. In W. Engel (Ed.), *Gpu pro 4: Advanced rendering techniques* (pp. 81–98). CRC Press.

Lim, H. (2008). Raster Data. In S. Shekhar & H. Xiong (Eds.), *Encyclopedia of gis* (pp. 949–955). Springer, NY, USA.

Lindley, C. (2009). *jQuery Cookbook: Solutions & Examples for jQuery Developers*. O'Reilly Media, Inc.

Liu, Z., Pierce, M. E., Fox, G. C., & Devadasan, N. (2007). Implementing a caching and tiling map server: a web 2.0 case study. In *2007 international symposium on collaborative technologies and systems* (pp. 247–256).

Maguire, D. J. (1991). An overview and definition of GIS. *Geographical information systems: Principles and applications*, *1*, 9–20.

Marrin, C. (2011). *WebGL Specification Version 1.0* (Tech. Rep.). Khronos Group.

Masó, J., Pomakis, K., & Julià, N. (2010). *OpenGIS® Web Map Tile Service Implementation Standard Version 1.0.0* (Tech. Rep. No. OGC 07-057r7). OGC.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320.

Meaden, G. J., & Chi, T. D. (1996). *Geographical information systems Applications to marine fisheries*. Food and Agriculture Organization of the United Nations, Rome.

Mehta, N., Sicking, J., Graff, E., Popescu, A., Orlow, J., & Bell, J. (2015). *Indexed Database API* (Tech. Rep.). W3C.

Meisters, G. H. (1975). Polygons have ears. *The American Mathermatical Monthly*, *82*(6), 648–651.

Mengeringhausen, H. C., & Witherell, W. R. (1962). A nonstandard use of 16mm to meet the 8mm print cost challenge. *Journal of the SMPTE*, *71*(8), 566–568.

Neteler, M., Bowman, M. H., Landa, M., & Metz, M. (2012). GRASS GIS: A multi-purpose open source GIS. *Environmental Modelling & Software*, *31*, 124–130.

Nguyen, V., Deeds-Rubin, S., Tan, T., & Boehm, B. (2007). *A SLOC Counting Standard* (Tech. Rep. No. USC-CSSE-2007-737). Center for Systems and Software Engineering.

O'Reilly, T. (2007). What is Web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, *65*(1), 17–37.

Orlik, A., & Orlikova, L. (2014). Current Trends in Formats and Coordinate Transformations of Geospatial Data – Based on MyGeoData Converter. *Central European Journal of Geosciences*, *6*(3), 354–362.

Paulson, L. D. (2005). Building rich web applications with Ajax. *Computer*, *38*(10), 14–17.

Peng, Z. R. (1999). An assessment framework for the development of Internet GIS. *Environment and Planning B: Planning and Design*, *26*(1), 117–132.

Peng, Z. R., & Zhang, C. (2004). The roles of geography markup language (GML), scalable vector graphics (SVG), and Web feature service (WFS) specifications in the development of Internet geographic information systems (GIS). *Journal of Geographical Systems*, *6*(2), 95–116.

Persson, J. (2004). Streaming of compressed multi-resolution geographic vector data. In *Proceedings of the 12th international conference on geoinformatics geospatial information research: Bridging the pacific and atlantic* (pp. 765–772).

Peterson, M. P. (1999). Trends in Internet Map Use – A Second Look. In *Proceedings of the 19th international cartographic conference* (pp. 571–580).

Peucker, T. K., Fowler, R. J., Little, J. J., Mark, D. M., & Carto, A. (1978). The triangulated irregular network. In *American society of photogrammetry proceedings of the digital terrain models symposium* (pp. 96–103).

Plewe, B. (1997). *GIS online: Information retrieval, mapping, and the Internet*. OnWord Press.

Poorazizi, M. E., & Hunter, A. J. (2015). Evaluation of Web Processing Service Frameworks. *OSGEO Journal*, *14*, 29–42.

Ramakrishna, A., Chang, Y., & Maheswaran, R. (2013). An interactive web based spatio-temporal visualization system. In G. Bebis et al. (Eds.), *Advances in visual computing: 9th international symposium, isvc 2013, part ii.* (pp. 673–680).

Ramsey, P. (2007). *The State of Open Source GIS* (Tech. Rep.). Refractions Research Inc.

Reed, C., Buehler, K., & McKee, L. (2015). OGC consensus: How successful standards are made. *ISPRS International Journal of Geo-Information*, *4*(3), 1693–1706.

Resch, B., Wohlfahrt, R., & Wosniok, C. (2014). Web-based 4D visualization of marine geo-data using WebGL. *Cartography and Geographic Information Science*, *41*(3), 235–247.

Revesz, P. (2008). Constraint Databases, Spatial. In S. Shekhar & H. Xiong (Eds.), *Encyclopedia of GIS* (pp. 157–160). Springer, NY, USA.

Ritter, N., & Ruth, M. (1997). The GeoTiff data interchange standard for raster geographic images. *International Journal of Remote Sensing*, *18*(7), 1637-1647.

Roth, R. E. (2017). Visual variables. In D. Richardson, N. Castree, M. F. Goodchild, A. Kobayashi, W. Liu, & R. A. Marston (Eds.), *International encyclopedia of geography: People, the earth, environment and technology* (pp. 1–11).

Roth, R. E., Donohue, R., Sack, C., Wallace, T., & Buckingham, T. (2014). A Process for Keeping Pace with Evolving Web Mapping Technologies. *Cartographic Perspectives*, *0*(78), 25–52.

Sahr, K. (2013). On the Optimal Representation of Vector Location using Fixed-Width Multi-Precision Quantizers. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XL-4*(W2), 1–8.

Sauerwein, T. (2010). *Evaluation of HTML5 for its Use in the Web Mapping Client OpenLayers* (Thesis). Fachhochschule Kaiserlautern, University of Applied Sciences.

Schernthanner, H., Steppan, S., Kuntzsch, C., Borg, E., & Asche, H. (2017). Automated web-based geoprocessing of rental prices. In *International conference on computational science and its applications* (pp. 512–524).

Schut, P. (2007). *OpenGIS® Web Processing Service Version 1.0.0* (Tech. Rep. No. OGC 05-007r7). OGC.

Seidel, R. (1991). A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry*, *1*(1), 51–64.

Shalloway, A., & Trott, J. R. (2002). *Design Patterns Explained A New Perspective on Object-Oriented Design*. Addison-Wesley Professional.

Shepperd, M., & Ince, D. C. (1994). A critique of three metrics. *Journal of Systems and Software*, *26*(3), 197–210.

Stefanakis, E. (2017). Web mercator and raster tile maps: two cornerstones of online map service providers. *Geomatica*, *71*(2), 100–109.

Steiniger, S., & Hunter, A. J. (2013). The 2012 free and open source GIS software map – A guide to facilitate research, development, and adoption. *Computers, Environment and Urban Systems*, *39*, 136–150.

Steiniger, S., & Hunter, A. J. (2017). Data structure: spatial data on the web. In D. Richardson, N. Castree, M. F. Goodchild, A. Kobayashi, W. Liu, & R. A. Marston (Eds.), *International encyclopedia of geography: People, the earth, environment and technology* (pp. 1–12).

Taivalsaari, A., Mikkonen, T., Anttonen, M., & Salminen, A. (2011). The death of binary software: End user software moves to the web. In *Creating, connecting and collaborating through computing (c5)* (pp. 17–23).

Thorne, C. (2005). Using a floating origin to improve fidelity and performance of large, distributed virtual worlds. In *2005 international conference on cyberworlds (cw'05)* (pp. 263–270).

Thrall, S. E., & Thrall, G. I. (1999). Desktop GIS software. In P. A. Longley, M. F. Goodchild, D. J. Maguire, & D. W. Rhind (Eds.), *Geographical Information Systems Abridged* (pp. 331–345). John Wiley & Sons, Inc.

Tomlin, C. D. (2017). Cartographic modeling. In D. Richardson, N. Castree, M. F. Goodchild, A. Kobayashi, W. Liu, & R. A. Marston (Eds.), *International encyclopedia of geography: People, the earth, environment and technology* (pp. 1–6).

Turton, I. (2008). GeoTools. In G. B. Hall & M. G. Leahy (Eds.), *Open source approaches in spatial data handling. advances in geographic information science, vol 2.* (pp. 153–169). Springer.

Vasilescu, B., Casalnuovo, C., & Devanbu, P. (2017). Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 683–693).

Vretanos, P. A. (2002). *Web Feature Service Implementation Specification Version 1.0.0* (Tech. Rep. No. OGC 02-058). OGC.

Warmerdam, F. (2008). The Geospatial Data Abstraction Library. In G. B. Hall & M. G. Leahy (Eds.), *Open source approaches in spatial data handling. advances in geographic information science, vol 2.* (pp. 87–104). Springer.

Wood, L., Hors, A. L., Apparao, V., Byrne, S., Champion, M., Isaacs, S., ... Wilson, C. (2000). *Document Object Model (DOM) Level 1 Specification (Second Edition)* (Tech. Rep.). W3C.

Yang, C., Wu, H., Huang, Q., Li, Z., Li, J., Li, W., ... Sun, M. (2011). WebGIS performance issues and solutions. In S. Li, S. Dragicevic, & B. Veenendaal (Eds.), *Advances in web-based gis, mapping services and applications* (pp. 121–138). CRC Press.

Yang, P., Cao, Y., & Evans, J. (2007). Web map server performance and client design principles. *GIScience & Remote Sensing*, *44*(4), 320–333.

Online References

Adobe Corporate Communications. (2017). *Flash & The Future of Interactive Content.* https://theblog.adobe.com/adobe-flash-update/. (Accessed: 2019-04-06)

Agafonkin, V. (2015). *Earcut: The fastest and smallest JavaScript polygon triangulation library for your WebGL apps.* https://github.com/mapbox/earcut. (Accessed: 2019-05-24)

Bostock, M., & Metcalf, C. (2013). *The TopoJSON Format Specification.* https://github.com/topojson/topojson-specification/blob/master/README.md. (Accessed: 2019-04-16)

DesLauriers, M. (2015). *Drawing Lines is Hard.* https://mattdesl.svbtle.com/drawing-lines-is-hard. (Accessed: 2019-05-21)

Ericsson. (2017). *Internet of Things forecast.* https://www.ericsson.com/en/mobility-report/internet-of-things-forecast. (Accessed: 2019-04-09)

Farkas, G. (2018a). *GaborFarkas/ol3 at raster_base.* https://github.com/GaborFarkas/ol3/tree/raster_base. (Accessed: 2019-06-06)

Farkas, G. (2018b). *OpenLayers WebGL rendering benchmark tool.* https://gaborfarkas.github.io/rendering_pub/profile/. (Accessed: 2019-04-30)

Firefox Contributors. (2002). *[RFE] TIFF Support?* https://bugzilla.mozilla.org/show_bug.cgi?id=160261. (Accessed: 2019-04-20)

Google. (2019). *Google Maps.* https://www.google.com/maps. (Accessed: 2019-04-16)

Käfer, K. (2014). *Drawing Text with Signed Distance Fields in Mapbox GL.* https://blog.mapbox.com/drawing-text-with-signed-distance-fields-in-mapbox-gl-b0933af6f817. (Accessed: 2019-05-25)

Miniwatts Marketing Group. (2019). *World Internet Users and 2019 Population Stats.* https://www.internetworldstats.com/stats.htm. (Accessed: 2019-04-09)

NASA. (2018). *WorldWind GeoTiff.* https://files.worldwind.arc.nasa.gov/artifactory/apps/web/examples/GeoTiffExample.html. (Accessed: 2019-04-20)

Nielsen, M. (2008). *Spherical/Web Mercator: EPSG code 3785.* https://www.sharpgis.net/post/2008/05/15/SphericalWeb-Mercator-EPSG-code-3785. (Accessed: 2019-04-11)

OGC. (2019). *About OGC.* http://www.opengeospatial.org/about. (Accessed: 2019-04-10)

OSGeo. (2012). *Tile Map Service Specification.* http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification. (Accessed: 2019-04-10)

OSGeo. (2019). *About OSGeo.* https://www.osgeo.org/about/. (Accessed: 2019-04-10)

Schindler, F. (2016). *geotiff.js and plotty.js - Visualizing Scientific Raster Data in the Browser.* https://av.tib.eu/media/20373. (Accessed: 2019-04-20)

Springmeyer, D. (2015). *Mapbox Vector Tile Specification adopted by Esri.* https://blog.mapbox.com/mapbox-vector-tile-specification-adopted-by-esri-14138105872f. (Accessed: 2019-04-19)

Yu, E., & Custer, A. (2006). *ArcInfo ASCII Grid format.* http://old.geotools.org/ArcInfo-ASCII-Grid-format_5250.html. (Accessed: 2019-04-20)

Zakai, A. (2019). *sql.js Releases.* https://github.com/kripken/sql.js/releases. (Accessed: 2019-05-18)

A. Appendix

Category	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Rendering					
Hardware acceleration	1	0	1	0	0.5
Render geometry	1	1	1	1	1
Render raster	0	0	0	0	0
Render image	1	1	1	1	1
Blend layers	1	0	0	0	0.5
Formats – Vector					
ESRI Shapefile	0.5	0.5	1	0.5	0.5
KML	1	0.5	0.5	1	1
GeoJSON	1	1	1	1	1
WFS	0	0.5	0	1	1
Write transaction	0	0.5	0	1	1
Formats – Raster					
GeoTiff	0.5	0.5	0.5	0.5	0.5
Arc/Info ASCII GRID	0	0	0	0	0
WCS	0	0	0	0	0
Formats – Image					
JPEG	1	1	1	1	1
PNG	1	1	1	1	1
WMS	1	1	1	1	1
Formats –Image – Tile service					
WMTS	1	0.5	1	1	1
TMS	1	1	0	1	0.5
OpenStreetMap slippy map	1	1	1	1	1
Google Maps	0	0.5	0	1	0.5
ArcGIS REST API	1	0.5	0	1	1
Bing Maps	1	0.5	1	1	1
Database – Connection					
PostGIS	0	0	0	0	0
SpatiaLite	0	0	0	0	0
MySQL	0	0	0	0	0

Appendix 1: Detailed support table of candidate libraries for the competitive analysis (Farkas, 2017a). 1 means core support, 0.5 means partial support or support by a third party extension, and 0 means no support.

Category	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Database – Functionality					
Using DBMS	0	0	0	0	0
Query/filter	0	0.5	0	1	0
Query language	0	0	0	0	0
Data – Pre-process					
On the fly transformation	1	0	1	0	0.5
Read attribute data	1	1	0	1	1
Z, and M coordinates	1	0	0.5	0	1
Geometry types	0.5	0.5	0.5	0.5	0.5
Spatial indexing	0	0	0	0	1
Geometry validation	0	0	0	0	0
Geometry simplification	0.5	0.5	0	0	1
Attribute table	0	0	0	0	0
Data – Conversion					
Interpolate	0	0	0	0	0
Raster to vector	0	0	0	0	0
Vector to raster	0	0	0	0	0
Data – Manipulation					
Update attribute data	1	1	0	1	1
Update geometry	0	1	0	1	1
Field calculator	0	0	0	0	0
Add/remove layer	1	1	1	1	1
Change layer order	1	1	0.5	1	1
Typed layers	0	0	0	1	0
Data – Analysis					
Basic geoprocessing	0.5	0.5	0.5	0.5	0.5
Topological analysis	0.5	0.5	0.5	0.5	0.5
Modify image	0	0	0	0	0.5
Modify raster	0	0	0	0	0
Raster algebra	0	0	0	0	0
Classification	0	0	0	0	0
Convolution	0	0	0	0	0
Write WPS request	0	0.5	0	1	0.5
Projection					
Transform vector	1	1	1	1	1
Warp raster	1	0	1	0	1
Well-known projections	0.5	0.5	0.5	0.5	0.5
Custom projections	0	0.5	0.5	1	1

Appendix 1 continued ...

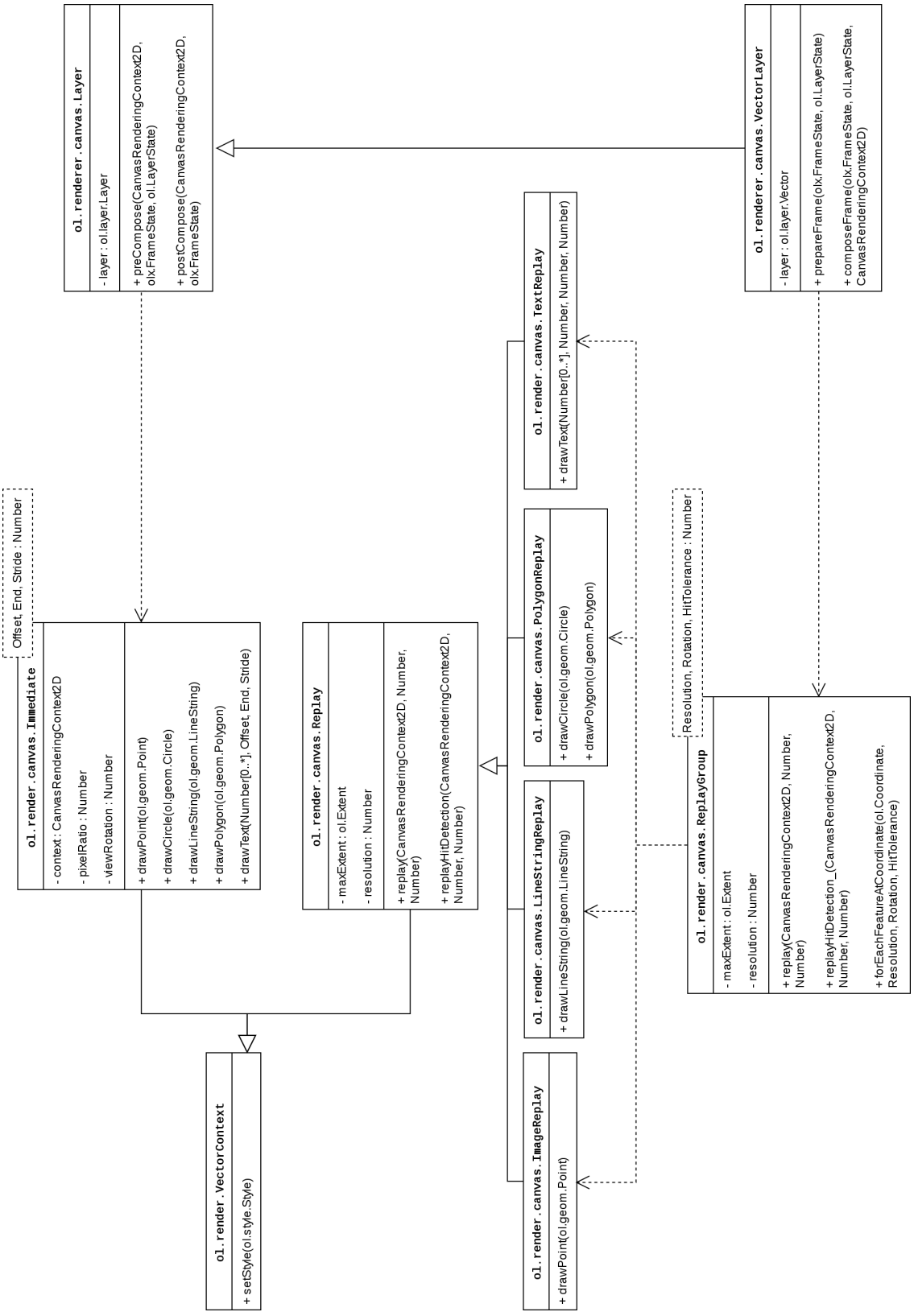
Category	Cesium	Leaflet	NASA WWW	OpenLayers 2	OpenLayers
Interaction					
Draw features	0	0.5	0	1	1
Modify features	0	0.5	0	1	1
Snap points	0	0.5	0	1	1
Modify view	1	0.5	1	0.5	1
Select features	1	0.5	0.5	1	1
Query	0	0.5	0.5	0.5	0.5
Measure	0	0.5	0	1	0
Change time	1	0.5	0	0.5	0
Show mouse coordinates	0	0.5	1	1	1
Representation – Styling					
Style vector	1	1	1	1	1
Style raster	0	0	0	0	0.5
Thematic maps	1	1	1	1	1
Representation – Cartographic elements					
Scale bar	0	1	0	1	1
North arrow	0	0	1	0	0
Legend	0	0	0	0	0
Graticule	0	0.5	0	1	1
Text box	0	0	0	0	0
Overview map	0	0.5	0	1	0.5

Appendix 1 continued ...

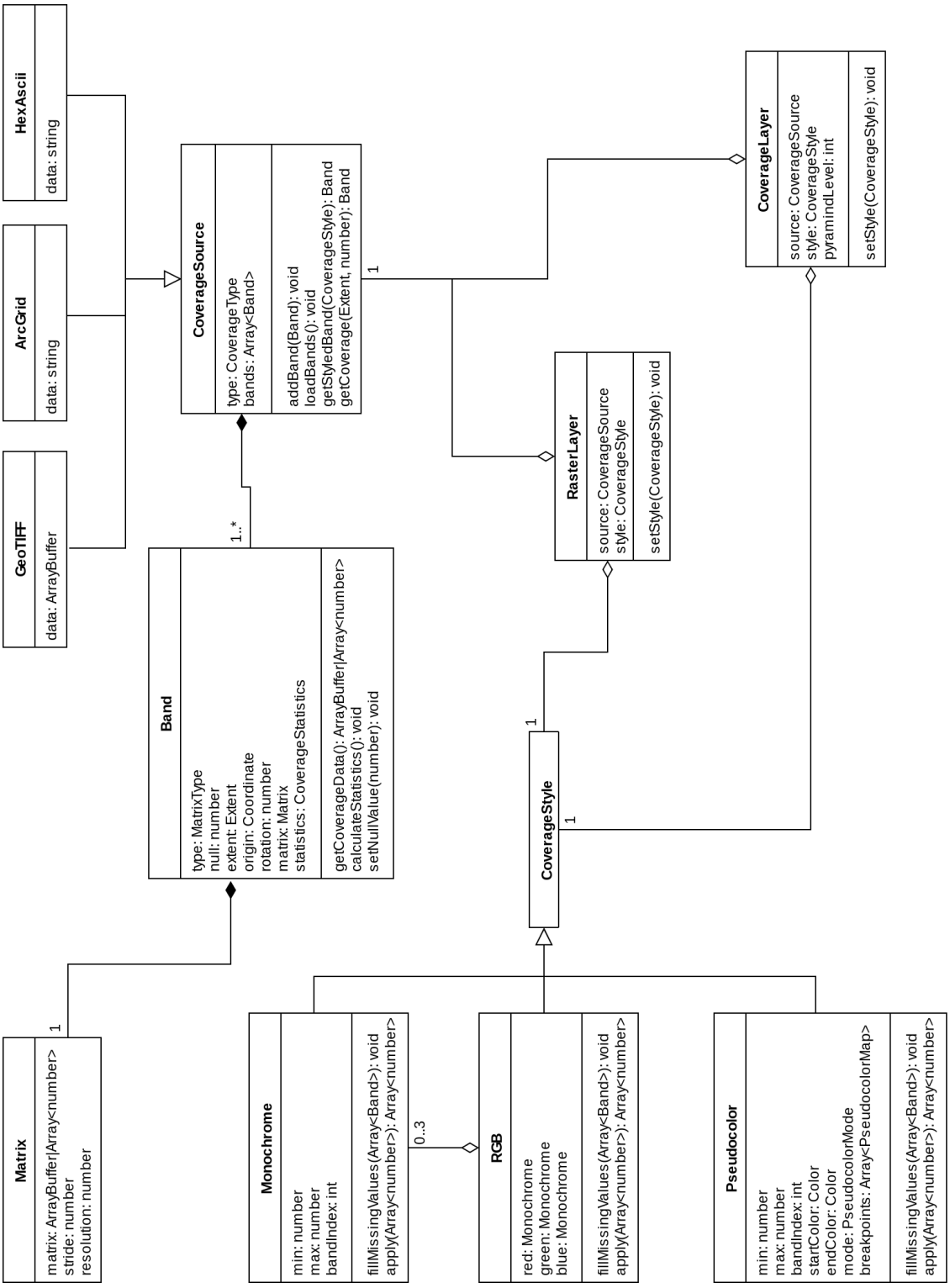
Name	Version	Type	License	Dependency	Last release	Last activity	Classification
ArcGIS API for JavaScript*	4.0	Web mapping	Commercial	–	< 6 months	Unknown	Proprietary
Bing Maps AJAX Control*	7.0	Web mapping	Commercial	–	> Year	Unknown	Proprietary
CartoDB.js	3.15.10	Web mapping	BSD 3-Clause	Leaflet	< Month	< Month	Specific
Cesium	1.23	Virtual globe	Apache 2.0	–	< Month	< Week	Candidate
D3	4.1.1	Data visualization	BSD 3-Clause	–	< Month	< Week	General
Google Maps JavaScript API*	3.24	Web mapping	Commercial	–	< 6 months	Unknown	Proprietary
HERE Maps API for JavaScript*	3.0.12.4	Web mapping	Commercial	–	< Year	Unknown	Proprietary
ka-Map	1.0	Web mapping	MIT	–	> Year	Unknown	Abandoned
Kartograph	0.8.2	Web mapping	GNU LGPL	Raphaël	> Year	> Year	Abandoned
Leaflet	1.0.0-rc2	Web mapping	BSD 2-Clause	–	< Month	< Week	Candidate
Mapbox JS*	2.4.0	Web mapping	BSD 3-Clause	Leaflet	< 6 months	< Month	Specific
Mapbox GL JS*	0.21.0	Web mapping	BSD 3-Clause	–	< Month	< Day	Specific
MapQuery	0.1	Web mapping	MIT	OpenLayers 2	> Year	> Year	Abandoned
MapQuest JavaScript Maps API*	7.2	Web mapping	Commercial	–	> Year	Unknown	Proprietary
Modest Maps	3.3.6	Web mapping	BSD	–	> Year	> Year	Abandoned
NASA Web World Wind	0.0.1	Virtual globe	NOSA	–	> 6 months	< Week	Candidate
OpenLayers 2	2.13.1	Web mapping	BSD 2-Clause	–	> Year	< Week	Candidate
OpenLayers 3	3.17.1	Web mapping	BSD 2-Clause	–	< Month	< Week	Candidate
OpenScales	2.2	Web mapping	GNU LGPL	–	> Year	< 6 months	Other
OpenStreetMap iD	1.9.7	Web mapping	ISC	D3	< Month	< Day	Specific
OpenWebGlobe	Unknown	Virtual globe	MIT	–	No release	< Year	Abandoned
Polymaps	2.5.1	Web mapping	BSD 3-Clause	–	> Year	> Year	Abandoned
Processing.js	1.6.0	Data visualization	MIT	–	< Month	< Month	General
Raphaël	2.2.0	Data visualization	MIT	–	< 6 months	< Month	General
WebGL Earth	2.4.2	Virtual globe	GNU GPLv3	Cesium	< Month	< Month	Extension

*Requires an API key.

Appendix 2: List of considered JavaScript libraries capable of spatial data visualization. Original list (Farkas, 2017a). Relative times are compared to the date of survey: 28th July, 2016.



Appendix 3: Larger version of the UML diagram in Figure 14.



Appendix 4: Larger version of the UML diagram in Figure 30.

Category	OpenLayers (3)	OpenLayers (6)	Changed
Rendering			
Hardware acceleration	0.5	0.5	*
Render geometry	1	1	
Render raster	0	0.5	*
Render image	1	1	
Blend layers	0.5	0.5	
Formats – Vector			
ESRI Shapefile	0.5	0.5	
KML	1	1	
GeoJSON	1	1	
WFS	1	1	
Write transaction	1	1	
Formats – Raster			
GeoTiff	0.5	0.5	*
Arc/Info ASCII GRID	0	0.5	*
WCS	0	0.5	*
Formats – Image			
JPEG	1	1	
PNG	1	1	
WMS	1	1	
Formats –Image – Tile service			
WMTS	1	1	
TMS	0.5	0.5	
OpenStreetMap slippy map	1	1	
Google Maps	0.5	0.5	
ArcGIS REST API	1	1	
Bing Maps	1	1	
Database – Connection			
PostGIS	0	0	
SpatiaLite	0	0	
MySQL	0	0	

Appendix 5: Detailed support table of the latest release of OpenLayers (6.0.1) compared to the one from the original study (3.17.1). 1 means core support, 0.5 means partial support or support by a third party extension, and 0 means no support.

Category	OpenLayers (3)	OpenLayers (6)	Changed
Database – Functionality			
Using DBMS	0	0	
Query/filter	0	0	
Query language	0	0	
Data – Pre-process			
On the fly transformation	0.5	0.5	
Read attribute data	1	1	
Z, and M coordinates	1	1	
Geometry types	0.5	0.5	
Spatial indexing	1	1	
Geometry validation	0	0	
Geometry simplification	1	1	
Attribute table	0	0	
Data – Conversion			
Interpolate	0	0	
Raster to vector	0	0	
Vector to raster	0	0	
Data – Manipulation			
Update attribute data	1	1	
Update geometry	1	1	
Field calculator	0	0	
Add/remove layer	1	1	
Change layer order	1	1	
Typed layers	0	0	
Data – Analysis			
Basic geoprocessing	0.5	0.5	
Topological analysis	0.5	0.5	
Modify image	0.5	0.5	
Modify raster	0	0.5	*
Raster algebra	0	0.5	*
Classification	0	0.5	*
Convolution	0	0	
Write WPS request	0.5	0.5	
Projection			
Transform vector	1	1	
Warp raster	1	1	
Well-known projections	0.5	0.5	
Custom projections	1	1	

Appendix 5 continued . . .

Category	OpenLayers (3)	OpenLayers (6)	Changed
Interaction			
Draw features	1	1	
Modify features	1	1	
Snap points	1	1	
Modify view	1	1	
Select features	1	1	
Query	0.5	0.5	
Measure	1	1	
Change time	0	0	
Show mouse coordinates	1	1	
Representation – Styling			
Style vector	1	1	
Style raster	0.5	0.5	*
Thematic maps	1	1	
Representation – Cartographic elements			
Scale bar	1	1	
North arrow	0	0	
Legend	0	0	
Graticule	1	1	
Text box	0	0	
Overview map	0.5	0.5	*

Appendix 5 continued ...