University of Pécs

Faculty of Sciences

Doctoral School of Earth Sciences

Creating the foundations of a universal client side Web GIS system

Main theses of the PhD dissertation

Author:
Gábor Farkas

Advisor:
Dr. Titusz Bugya

Pécs, 2020.

Name and address of doctoral school:

University of Pécs
Faculty of Sciences
Doctoral School of Earth Sciences
H-7624 Pécs, Ifjúság street 6.

Head of doctoral school:

Prof. Dr. István Geresdi
full professor
Faculty of Sciences
Institute of Geography and Earth Sciences
Department of Geology and Meteorology

Name of discipline:

Physical geography and land evaluation

Head of discipline:

Prof. Dr. Dénes Lóczy
full professor
Faculty of Sciences
Institute of Geography and Earth Sciences
Department of Physical and Environmental Geography

Advisor:

Dr. Titusz Bugya
assistant professor
Faculty of Sciences
Institute of Geography and Earth Sciences
Department of Cartography and Geoinformatics

1. Introduction

Web GIS is a relatively new field in geoinformatics, following the evolution of the Web. Its main purpose is combining new technologies coming 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 on 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.

With the advent of Web 2.0 (O'Reilly, 2007), web mapping and Web GIS replaced old-fashioned Internet GIS. Starting with Google Maps in 2005 (Farkas, 2015), a new trend emerged, porting spatial algorithms to the client-side using JavaScript. JavaScript programs are slower than compiled code, and more dependent on coding style (Gong et al., 2015), therefore, 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.

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). There are groups of developers working on pure client-side solutions for several aspects of a GIS system. There are libraries for interacting with various popular data exchange formats (e.g. Shapefile, Geopackage, GeoTIFF) and also 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 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. 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.

For defining a universal Web GIS system, one has to go back to Geographic Information System definitions. GISs 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. From computer cartography comes the structure of spatial data in a GIS. 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.

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.

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. 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. 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.

2. Purpose of 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 basis. 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 basis.

3. Materials and Methods

3.1. Competitive analysis and software metrics

The study consists of several, methodically different steps, therefore there are multiple methods. The first step was choosing the right basis for a universal massive client. Since a list of spatial data visualization libraries needed to be compared, this step mostly relies on theory. The list of GIS features used by this study (Figure **??**) 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.

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.

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.

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.

In order to approximate the total complexity of a library, an evasive characteristic was targeted: the learning curve. It can be assumed, if enough aspects of complexity are covered, a formula from static software metrics can make a crude approximation. Approximate Learning Curve for JavaScript (ALC$_{\mathrm{JS}}$) was tailored for JavaScript libraries specifically (Farkas, 2017). The formula (Equation 1) includes Logical Lines of Code (LLOC), cyclomatic complexity $v(G)$, and the number of LLOC per exposed functions EF, as they have great impact on the learning curve of a project (Fowler et al., 1999).

$$ALC_{JS} = \log_{10} LLOC \times \log_2(\frac{v(G)}{F} \times \frac{LLOC}{EF}) \tag{1}$$

There are some other characteristics contributing to the overall usability. Some of the more important ones are documentation, community, 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 2) 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. As tutorials are more comprehensive, harder to create, and help users accommodate themselves to the library faster, they receive a larger weight than examples. In the end, the final scores were converted to ordinal values (Table 1).

$$Score = A \times (T/10 + E/100) \tag{2}$$

Community and support are soft metrics, which are hard to evaluate. 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 3).

$$RF = \frac{D_{LR} - D_{FR}}{n - 1} \tag{3}$$

Support data were collected from two sources. Since most of the compared projects are using GitHub as a Version Control System, 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 (Stack Overflow and GIS Stack Exchange) were involved. The ratio of answered questions was used to assess the quality of community support on an ordinal scale (Table 1).

3.2. 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 and a 64-bit Chromium with hardware accelerated Canvas, running on a Debian 9 OS. For some of the hardware accelerated rendering tests a a Lenovo A536 smartphone was also introduced with a 32-bit Chrome running on Android 4.4.2, accessed through remote debugging.

Hardware accelerated rendering benchmarks were carried out using a specific application 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.

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. 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. 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.

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 (Figure 1), while the second one is a state of a GIS workflow, representing a typical GIS load.

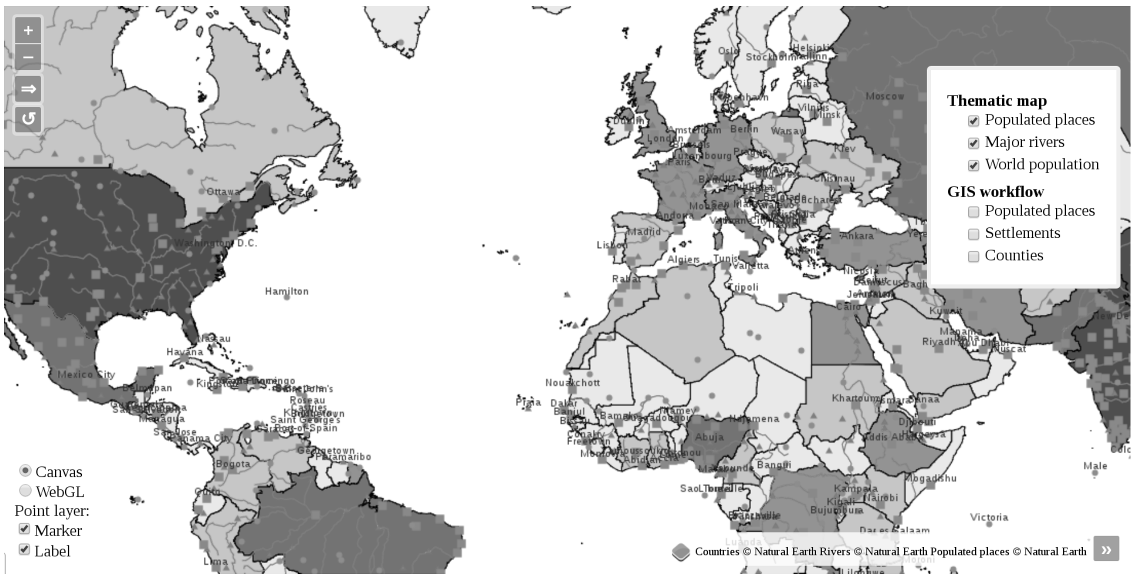

Figure 1: Thematic web map used for benchmarking a real world example of a web mapping application (Farkas, 2019).

In order to measure raster management techniques efficiently, two different sample rasters were used (Figure 2). 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.

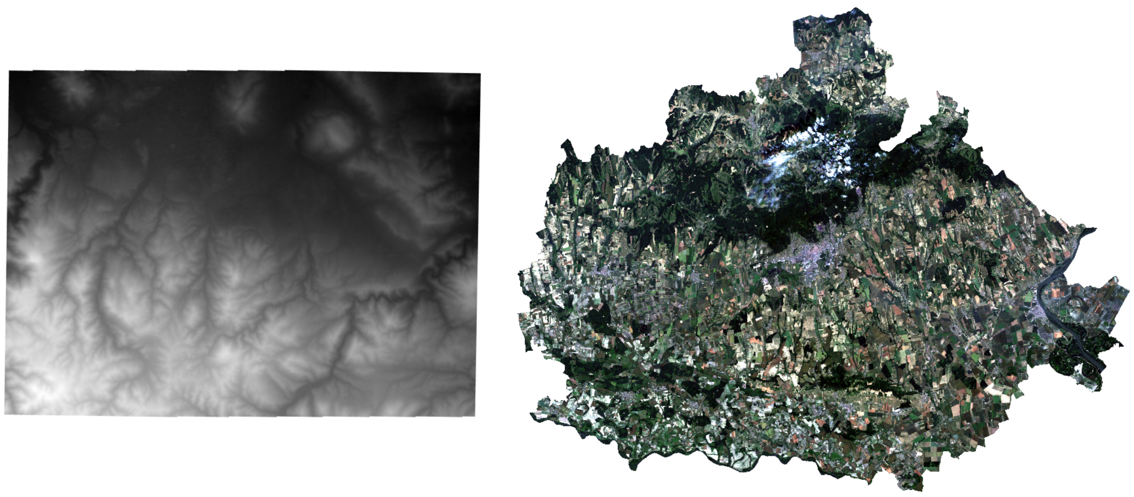

Figure 2: Sample rasters used for visualization and benchmarking. Both of the rasters are reprojected on the fly to the Web Mercator (EPSG:3857) projection. The Spearfish60 DEM (left) has a monochrome greyscale style, while the Baranya imagery (right) has an RGB style created from three corresponding bands (Farkas, 2020).

4. Results

4.1. Choosing a candidate

In order to choose the best basis for a universal Web GIS client, numerous data visualization libraries were selected. From those libraries, 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, namely 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.

The overall results of the competitive analysis (Table 2) 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.

Upon investigating further using static software metrics (Table 3), no surprising results could be found. LLOC values, per function cyclomatic complexity, and the number of exposed functions were able to describe several aspects of candidate libraries. Furthermore, $\mathrm{ALC}_{\mathrm{JS}}$ values matched experienced difficulties with the candidates.

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

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 2: GIS feature coverage of candidate libraries (Farkas, 2017).

Library	Size (KB)	LLOC	CC/F	EF	$\mathrm{ALC_{JS}}$
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 3: Static software metrics of candidate libraries (Farkas, 2017). *CC/F* stands for per function cyclomatic complexity.

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

develop. 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. 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. 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 4) 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.

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. 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.

According to candidates' feature coverage and metrical results, they could be narrowed down to two technologies: Cesium and OpenLayers. Since the two libraries were in more or less a tie, the supported features were reinvestigated, and crucial features were identified. 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. 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.

4.2. Hardware accelerated vector rendering

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. 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. 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.

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. 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. A

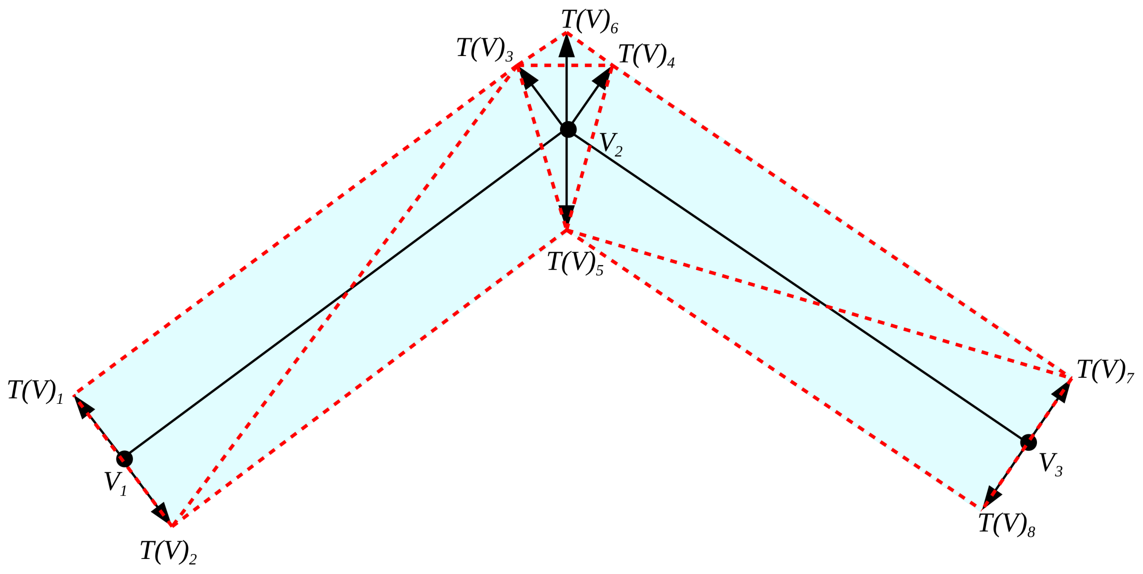

Figure 3: Triangulation scheme of two connected line segments V_1, V_2, V_3 with a miter line join and no line caps. Arrows show the magnitudes and directions of offsets needed to require triangulation points $T(V)_n$. Dashed lines represent triangles resulting from connecting triangulation points in the right order.

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 3).

The final program calculates every point in the GPU. 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. The encoded parameter contains an instruction i (e.g. offset, miter, square cap), a direction d, and a rounding factor r compressed into a single number $p = i \times d \times r$, which can be decoded in the GPU.

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. 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 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

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	$\underline{14.75}$	$\underline{10.92}$	$\underline{10.26}$	$\underline{12.18}$	$\underline{15.37}$
Canvas drawing	22.94	18.59	$\underline{14.23}$	24.33	33.00
Intel GPU					
WebGL animating	58.99	55.34	52.59	59.01	59.72
Canvas animating	26.07	20.21	$\underline{15.17}$	16.77	22.83
WebGL drawing	$\underline{10.91}$	$\underline{8.37}$	$\underline{7.17}$	$\underline{7.72}$	$\underline{8.03}$
Canvas drawing	$\underline{13.57}$	$\underline{12.92}$	$\underline{10.28}$	$\underline{10.71}$	$\underline{14.66}$
ARM Mali GPU					
WebGL animating	$\underline{12.67}$	$\underline{12.08}$	$\underline{10.66}$	16.51	21.83
Canvas animating	$\boxed{1.19}$	$\boxed{1.22}$	$\boxed{1.87}$	$\boxed{3.69}$	$\underline{5.84}$
WebGL drawing	$\boxed{1.25}$	$\boxed{0.94}$	$\boxed{0.82}$	$\boxed{1.45}$	$\boxed{2.27}$
Canvas drawing	$\boxed{1.13}$	$\boxed{0.96}$	$\boxed{1.32}$	$\boxed{2.51}$	$\boxed{4.05}$

Table 5: Rendering performance (FPS) of the thematic layer group. Lags are emphasized by underlining, and severe lags by framing (Farkas, 2019).

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. The more deficiencies the polygon has, more precise and slower techniques are applied.

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 final implementation a glyph atlas 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.

The first set of measurements targeted the thematic layer group. Results (Table 5) 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 is more optimal. 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.

Overall user experience is mostly affected by the animating speed. Those FPS values are experienced during user interactions, like zooming, panning, and rotating. Drawing speed is only experienced when the map comes to a standstill. Therefore,

	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%

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

Table 6: Ratio of different calls when rendering the GIS group (Farkas, 2019).

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 of 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 speedup is not as notable.

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.

Measuring the GIS group (Table 6) added a few additional insights. This time, WebGL polygon rendering was measured without triangulation, since involving it would have not been added anything new to the picture. According to the results, without triangulation, the WebGL renderer has a better time complexity than the Canvas polygon renderer. It means, there is a turning point, where even a naive WebGL engine becomes more efficient, and the GIS group is beyond that point.

4.3. 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. 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. 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.

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.

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.

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. 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.

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

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. For traditional rasters, a GeoTIFF and an ArcGrid class was created, inheriting from the coverage source.

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 4). 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.

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

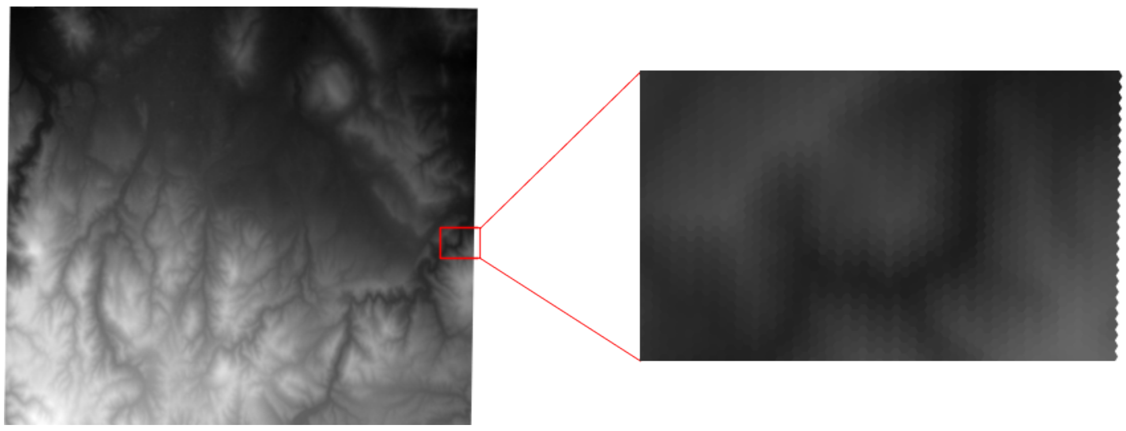

Figure 4: The Spearfish60 DEM rendered as a HexASCII coverage (Farkas, 2020).

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.

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 7).

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. 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.

The preparation phase in case of rasters and coverages groups steps related to creating and caching visualized data, even between complete redraws. Preparation time is not significant in the case of traditional rasters (Table 8). 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.

By looking at the detailed metrics of different pyramid levels (Table 9), 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

	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 7: 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).

	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 8: 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).

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 9: 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.

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.

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.

5. 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.

During investigating the best basis 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. 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).

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.

The Canvas renderer is suitable with 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), it 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. 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.

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.

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.

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).

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

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

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

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.

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

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).

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

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

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

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

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.

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

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

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.

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.

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).

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).