DOI: 10.1111/tgis.12588

RESEARCH ARTICLE

Transactions in GIS   WILEY

Possibilities of using raster data in client-side web maps

Gábor Farkas

Department of Cartography and Geoinformatics, University of Pécs, Pécs, Hungary

Correspondence
Gábor Farkas, Department of Cartography and Geoinformatics, University of Pécs, Ifjúság street 6, Pécs H-7624, Hungary.
Email: gfarkas@gamma.ttk.pte.hu

Funding information
Emberi Eroforrások Minisztériuma, Grant/Award Number: ÚNKP-17-3-I; European Social Fund, Grant/Award Number: EFOP-3.6.1.-16-2016-00004

Abstract

Companies and individual developers have recently put serious effort into improving web mapping libraries. A significant front in this development is hardware-accelerated vector graphics. Owing to those efforts, and the continuously evolving World Wide Web, users can visualize large vector layers, and even animate them. On the other hand, this rapid development cannot be observed with raster data. There are some data abstraction libraries for reading raster files, although web mapping libraries do not use them to offer raster capabilities. Since there are no mature raster management pipelines on the web, this study explores two inherently different techniques for handling raster data. One of them uses the traditional, texture-based method. The other is a hybrid technique rendering raster layers as vectors, overcoming some limitations of the raster model. While the traditional technique gives a smooth user experience, the hybrid method shows promising results for rendering hexagonal coverages.

1 | INTRODUCTION

Rapid development of technology in recent decades has not only introduced new possibilities in GIS and cartography, but also transformed them. The main focus has shifted from traditional mapping and analysis of spatial data to new, digitally better-supported approaches. For example, experts can now produce three-dimensional models from digital images, have to deal with different types of big data, and are finding ways to apply machine learning to spatial problems. A shifting focus can also be seen in the case of web mapping and web GIS. Since the first static internet maps were created and rendered by a server (Haklay, Singleton, & Parker, 2008; Peterson, 1999), this field has undergone a complete transformation. Now more and more logic is migrated to the client side, changing a thin client (Doyle, 2000) from the only feasible solution to an optional choice. Clients take advantage of the strength of personal computers and handheld devices. They handle projections and coordinate transformations, set up grids for tiled maps, handle user interactions, style raw vector data, and also analyze vector data, if necessary (Di Pasquale, Fresta, Maiellaro, Padula, & Scala, 2012).

While most of the functionality of a simple desktop GIS can be found in web mapping libraries (Farkas, 2017) or in other forms (e.g., standalone JavaScript libraries, third-party modules), client-side raster management is a different matter. Traditionally, raster data were processed, warped, and rendered on the server side, and were served as images to the clients (Gawne-Cain & Holcroft, 2000). Such an architecture was necessary due to the relatively high computational demands of large rasters and, in general, the high computational complexity of raster algorithms. The pipeline was later optimized to create cacheable image tiles (Wendlik, Karut, & Behr, 2011) in a single projection. In practice, tiled layers often contained complete, production-ready maps consisting of raster and vector data, which also contributed to the popularity of using spatial servers exclusively for raster processing.

Handling raster layers on the server side was not solely a consideration of performance. General client-side web applications (i.e., browsers) were limited in functionality and extendibility. First, they could only support a handful of image formats (e.g., PNG, JPEG, BMP, GIF), optimized for fast data transmission. Therefore, making them automatically create visualizations from raw raster data (such as GeoTIFF) was not possible. Second, their only general, not browser-specific scripting language, JavaScript, was not capable of operating on binary data. Since the more popular, space-efficient raster formats had binary encoding, this was also an issue. While sending moderate raster layers in ASCII format was not feasible due to generally low consumer bandwidth, this problem could be mitigated by using Java applets and ActiveX controls (Peng, 1999). Overall, numerous synergistic issues and the satisfactory results achievable by converting rasters to RGB images mutually led to the huge delay in dealing with client-side raster processing.

Most of the limitations preventing rasters from becoming a part of a web GIS architecture have recently been lifted (Taivalsaari, Mikkonen, Anttonen, & Salminen, 2011). We have strong client machines, hardware acceleration built into browsers, and high bandwidth in developed countries. Furthermore, ECMAScript 2015 (also known as JavaScript 6) introduced support for processing raw binary data in browsers (Ecma International, 2015). With a simple step holding extraordinary importance, it opened the door for many client-side JavaScript libraries targeting binary spatial data exchange formats (e.g., shapefile, GeoPackage, GeoTIFF). By utilizing one of those libraries (Schindler, 2016, geotiff.js) it is now possible to efficiently parse raster layers for client-side application.

In traditional GIS, raster processing is older than vector processing. Consequently, there is mature code in desktop environments, on which desktop GIS software can rely. Those libraries (Warmerdam, 2008) and algorithms (Neteler, Bowman, Landa, & Metz, 2012) form a solid and optimized basis for using raster data, and therefore they are not subjects for reconsideration. On the other hand, web-based environments do not share the same history, and their raster processing is still in its infancy. This is not necessarily detrimental, as it leaves space for experimenting with novel approaches for modern demand.

2 | MATERIALS AND METHODS

The subject of this study was developed in an incremental manner in order to measure performance with different techniques. All stages targeted the OpenLayers web mapping library. OpenLayers was chosen due to the author's expertise with the library, and the already established code parts from previous studies. Code development took a long time, during which OpenLayers changed significantly. As a result, the code accessible from the author's

GitHub repository[1] can be compiled with an older version of OpenLayers. The complete raster module will be rewritten later to match the new OpenLayers architecture, and will be released as a third-party module.

The first technique implemented is a traditional raster model, using textures for displaying raster images. While traditional raster handling is fast and adequate for rectangular rasters, it imposes serious limitations on the underlying data (Bugya & Farkas, 2018). Apart from the advantages of using vector cells in the representation model, the coverage model has one clear advantage. It allows users to work with cell shapes other than a rectangle. Furthermore, they can define a pattern, if a shape with inappropriate symmetrical properties is needed (Figure 1).

In order to keep the implementation practical, besides the rectangular coverage for comparison, only the hexagonal coverage was realized, due to its proven merits (Birch, Oom, & Beecham, 2007; de Sousa, Nery, Sousa, & Matos, 2006). Specifically, the HexASCII format (de Sousa & Leitão, 2017) was chosen. Since it is a fully specified file format, the implementation can serve as a testing ground for further studies and experiments.

In order to throw light on the nature of different techniques, they were benchmarked. The benchmarking was performed on a Dell Inspiron 7567 with an Intel Core i7-7700HQ CPU, 8 GB DDR4 RAM, an NVIDIA GeForce GTX 1050 Ti dedicated GPU (through the proprietary NVIDIA driver), and a 15.6-inch display with a resolution of 1,920×1,080 pixels. The browser hosting the test application was 64-bit Chromium, running on a Debian 9 OS. Benchmarks were performed in terms of both memory footprint and performance. Two sample rasters were chosen for benchmarks; the well-known Spearfish60 DEM from GRASS GIS sample data sets, and the four-band (red, green, blue, near-infrared) Landsat 8 imagery of Baranya county, Hungary (Figure 2).

3 | RASTER PROCESSING WITH JAVASCRIPT

The target web mapping library, OpenLayers, had both an HTML5 Canvas-based (partially hardware accelerated), and a WebGL-based (fully hardware accelerated) rendering engine at the time of this study. It was already established (Farkas, 2018) that the now abandoned WebGL renderer is the only one capable of handling raster layers as vectors. Consequently, while the implementation targets both of the engines, only WebGL results were examined.

Since there are few, if any, examples of a general web-based raster module, all of the supporting classes had to be written from scratch. Classes and functions independent of the representation model include internal raster storing logic, basic utility functions (e.g., resampling, interpolating, subsetting), styling logic, and data abstraction logic for handling data exchange formats. Those classes (Figure 3) are coupled tightly to the core application programming interface, in order to keep the amount of code at a minimum.

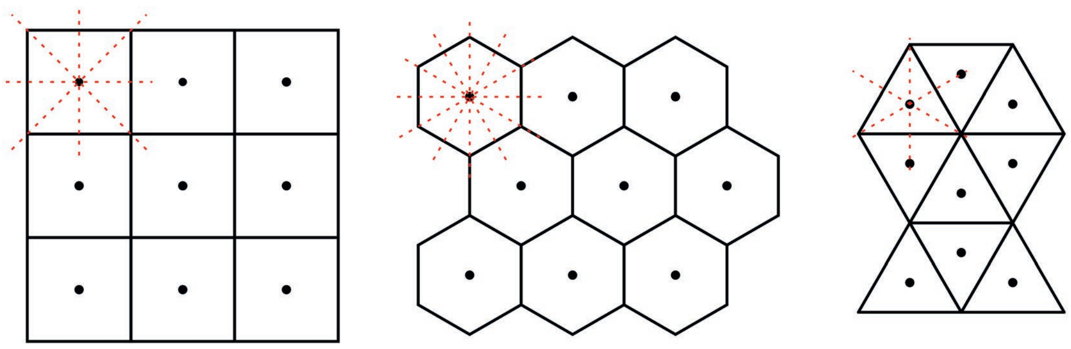

FIGURE 1 The three regular Euclidean tiling patterns (rectangular, hexagonal, triangular) with their symmetry axes highlighted. The first two can create a seamless tiling without modifying cell geometries because they have at least two congruent symmetry axes. Triangular cells need to be rotated

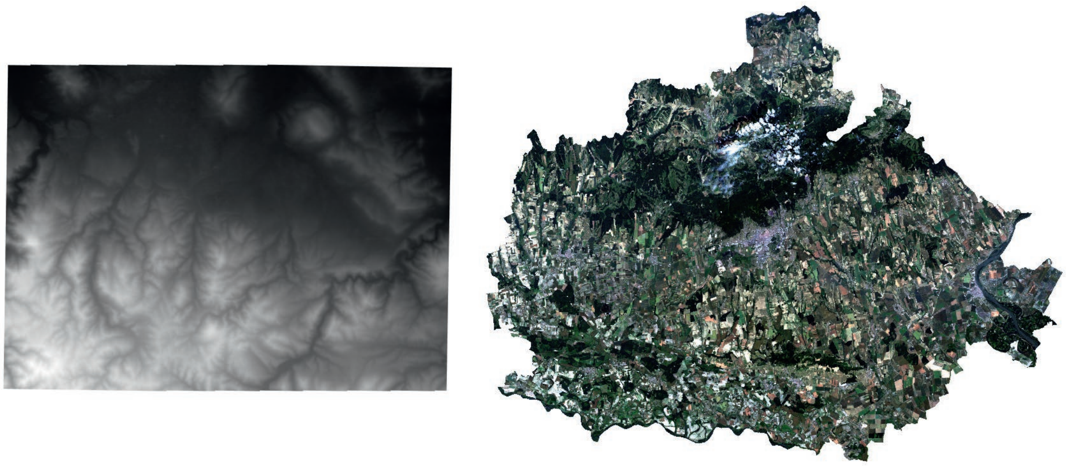

FIGURE 2 Sample rasters used for visualization and benchmarking. Both of the rasters are visualized in OpenLayers, and reprojected on the fly to the Web Mercator (EPSG:3857) projection. The Spearfish60 DEM (left) has a monochrome grayscale style, while the Baranya imagery (right) has an RGB style created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

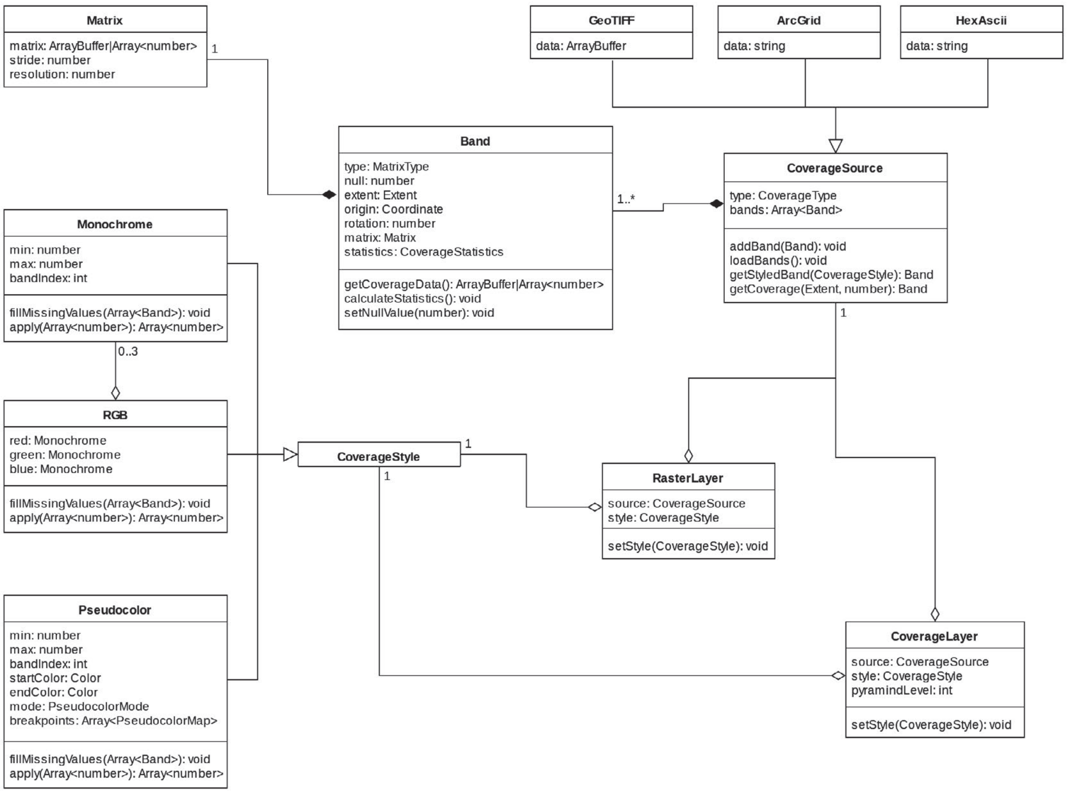

FIGURE 3 Generalized UML diagram of the raster module's base classes with their most important fields and methods

The raster processing pipeline starts with the abstraction layer. Currently one binary and two ASCII data exchange formats are supported. ASCII formats are parsed with string manipulation techniques, while GeoTIFF files use a third-party library, geotiff.js. Bands extracted from raster files are stored separately in their own `Band` classes along with some metadata. These include no data value, number of columns (stride), descriptive statistics, and an origin coordinate. The origin of the band in this case is the center of the lower-left cell. Raw cell values are stored in `Matrix` classes, which have one-to-one relationships with their `Band` instances. Matrices are stored in binary array buffers, if possible.

Following OpenLayers conventions, a `CoverageSource` class is used to store raster data, and two layer classes (`RasterLayer` and `CoverageLayer`) are used to interface with the rendering engines. While it would have been logical for the layer classes to create the styled representation models, it was more convenient to implement this part of the logic in the source class. Consequently, the source can return both an image of the styled matrix, and the raw styled matrix with interlaced color (RGBA) values.

Applying styles on raw data is accomplished with the style classes (`Monochrome`, `Pseudocolor`, `RGB`). The methods are applied to every band in a source. They know which bands are used, and map raw matrix values to colors according to user-defined attributes (e.g., minimum, maximum). If such an attribute is missing, they are using the given band's statistics to fill in the gap. Problems can arise if styling uses multiple bands. In such cases, further measures must be taken to keep the input bands aligned. There are utility functions for resampling bands with different resolutions, and subsetting bands with different extents. In order to keep the original data intact and reusable, original matrix values are not modified. If a user applies a new style, the bands must go through the styling pipeline again, resulting in a new styled band or image.

3.1 | Using traditional rasters

Visualizing rasters as images exploits browser capabilities, thus uses internal mechanisms for generating textures based on representation models. It also leverages the OpenLayers image reprojection algorithm for raster warping. Since browsers can handle resolution changes with mipmapping, there is no need for pyramid building or any additional optimization techniques.

Traditional rasters can be used with the `RasterLayer` class, which uses the built-in `WebGLImageLayerRenderer` renderer class. The renderer class expects an image with a spatial extent, which gets overlaid to the map. In this pipeline, the source class is requested to create an image from the styled band, reproject it, and give it to the renderer. As a result, drawing performance is high, memory footprint is minimal, but only rectangular cells can be used. Non-square cells are created by stretching the image with the ratio of a cell's dimensions (Figure 4).

3.2 | Implementing coverages

The other technique implemented is visualizing rasters as vectors, which has been named the coverage model (Bugya & Farkas, 2018). Coverages have the same data structure; they store raw data in matrices, and other attributes in bands. Only the representation model differs. Instead of drawing styled bands as images, coverage cells are represented with polygons. By using this technique, users have the opportunity to use different cell shapes (e.g., hexagons) without worrying about topology and data integrity (Figure 5).

Another benefit is the exact transformation of cells. One disadvantage of using images is that they need to be axis-oriented. Unlike images, coverage cells can be rotated and scaled on a cell-by-cell basis without interpolating the results. This is exceptionally useful during reprojection. While a traditional raster image needs to keep a constant resolution after warping, coverage cells can have different sizes (Figure 6). This allows for reprojecting a raster to a different projection on the fly for visualization without losing accuracy or duplicating layers. Furthermore, if the source data changes (e.g., due to an analysis process), the reprojected layer instantly adapts.

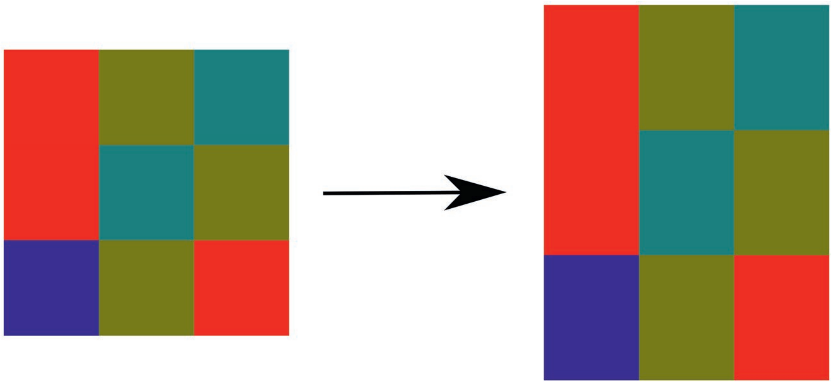

FIGURE 4 Traditional rasters of non-square cells are created by stretching an image with square cells. The performance of stretching is great; however, as everything (including interpolation) is performed by the browser, it might introduce visual artifacts with categorized data

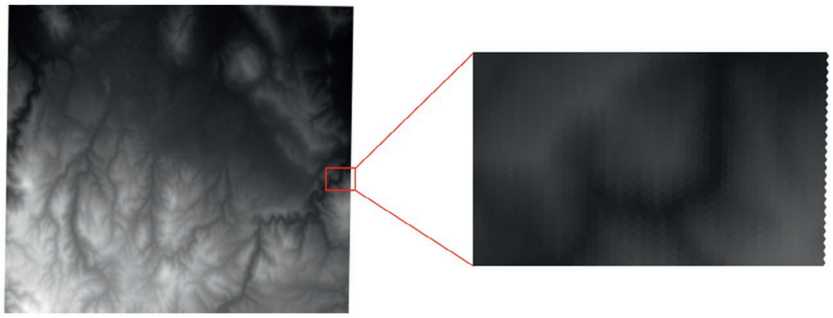

FIGURE 5 Spearfish60 DEM visualized as a hexagonal coverage. The original raster data is interpreted as a coverage with hexagonal cells, therefore the result is skewed

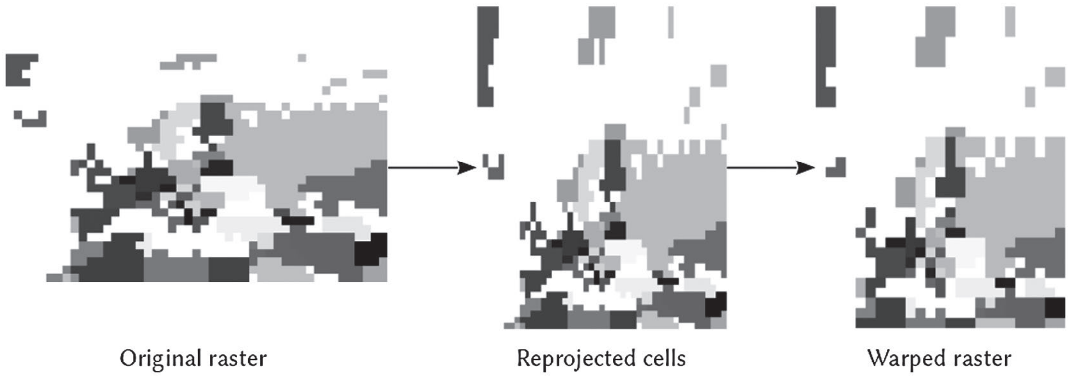

FIGURE 6 A raster warping process example on a Europe data set with a resolution of 2 degrees. While some GIS applications (e.g., QGIS) have sophisticated algorithms to approximate the second visualization with raster images, the coverage model is exact and does not need such extensions

The only drawback of the coverage renderer's first implementation was its poor performance. As a first optimization, spatial indexing was introduced, and cells were cached. While it made drawing faster on greater zoom levels, it did not improve worst-case performance. On the top of spatial indexing, the current implementation also uses pyramids. Pyramids are generated for both rectangular and hexagonal coverages, making drawing times acceptable.

Coverages use the `CoverageLayer` class, which supplies styled bands to the custom `WebGLCoverageLayerRenderer` class. The renderer class uses utility functions to build cell geometries by laying out a grid according to grid type (e.g., rectangular, hexagonal), and reprojecting each cell. Cells are stored in an R-Tree. This process is repeated for every pyramid level where, first, a generalized styled band is created from the last level. Algorithmically there is one significant difference between rectangular and hexagonal processing: creating pyramids. The implementation does not use traditional hexagon aperture techniques (Sahr, 2013), but partitions each cell into six triangles, and weights them in order to calculate downsampled cell values (Figure 7). As a consequence, cell orientations do not change, the number of cells is reduced quadratically at each level, and irregular hexagons with equal internal angles are also supported. Such a technique yields better approximations, although it is restricted to raster layers, since, unlike in discrete global grids, cells in different levels do not need to share their centroids.

3.3 | The shrink–expand phenomenon

While the pyramid generating algorithm described above makes using hexagonal coverages simple, it has one significant disadvantage. Similar to rectangular rasters, a cell in a hexagonal pyramid has an area of four cells of the previous level. For traditional rasters, however, edge effects can only be observed in layers with an odd number of rows or columns. These edge effects need to be handled only once, since then the number of columns and rows in the layer will be even, therefore further pyramid levels can be built without a remainder. The pyramid generating algorithm can decide whether to extend an odd dimension (expand), or cut it (shrink). Whichever technique the implementation chooses, the maximum shrinkage or expansion will be only one row or column.

In a hexagonal coverage, however, aggregating cells is harder. Every cell in a pyramid consists of three whole cells, and three partial cells from the previous level (Figure 7). Consequently, it does not matter if a pyramid cell has an area of four original cells; every pyramid cell needs six cells to be calculated accurately. As a result, if the implementation prefers to calculate every cell accurately, the layer's total area can shrink by 60% on the last pyramid level (Figure 8).

If an implementation chooses the opposite technique for edge handling, and extends the previous level, the layer will expand with every new pyramid level. Without breaking up the original pattern of the layer (e.g., every second row is offset) this can cause a 50% expansion in total area on the last level. Although the amount of

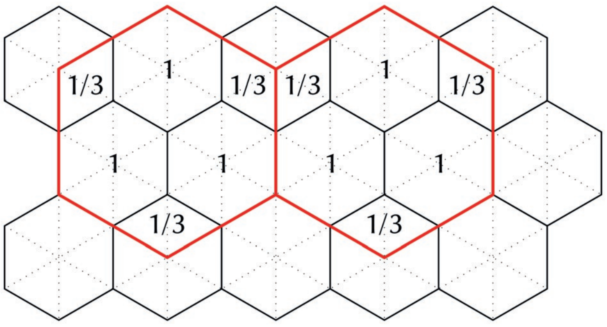

FIGURE 7 Pyramid creation scheme with hexagonal coverages. Similarly to rectangular pyramids, new cells have double size. New cells are created from the triangular decomposition of old ones, and their values are weighted averages of original cell values

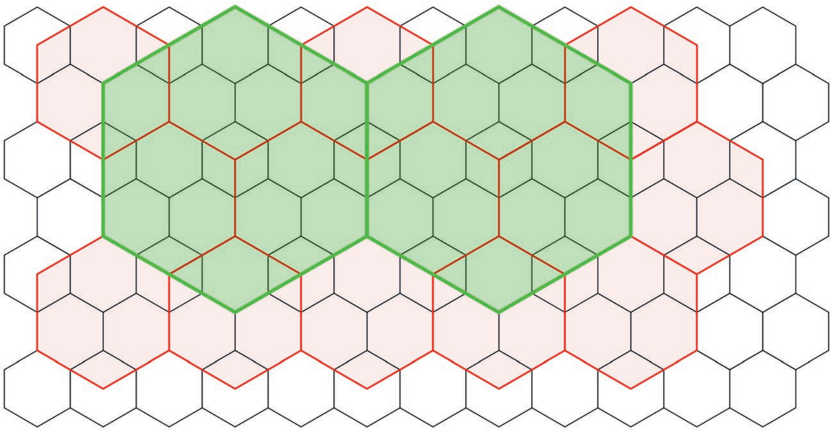

FIGURE 8 Shrinking pyramids can be observed when the implementation aims for maximum accuracy, and only creates new cells when their values can be calculated precisely from the previous level

expansion seems more feasible than the amount of shrinkage in the other case, the accuracy of cell values also decreases significantly due to the increasing number of null values (Figure 9).

Due to space and time constraints, it is outside the scope of this article to mathematically describe this phenomenon or to try to solve the problem caused by it. The OpenLayers implementation presented strives for accuracy, therefore it uses the shrinking method when building hexagonal pyramids.

4 | DISCUSSION

Since the implementation is naive and unoptimized, its usability in real-world scenarios is currently limited. The three variants of the two testing data sets (Table 1) could not be compared directly in every test case. While coverage layers did achieve acceptable performance with the Spearfish60 DEM, the coverage renderer could not handle the Baranya imagery at all. Furthermore, traditional raster methods cannot be applied on hexagonal coverages, which excludes the hexagonal Spearfish60 DEM from some test cases.

The rendering pipeline can be separated into three coherent, individually assessable, distinct parts. The first part is the abstraction layer, reading raster files. It converts the structure of a data exchange format into the module's internal raster structure. The second part is styling, which is independent of the rendering engine and the type of visualization (i.e., raster or coverage). The third part is drawing, where the raster is overlaid on the canvas.

While parsing a raster file is a one-time cost, it can affect user experience if it blocks the event loop of the browser. In such a case, the application seems frozen for the loading period. This can be avoided by parsing asynchronously. Parsing time (Table 2) is not only affected by raster size. The binary format of GeoTIFF files requires decoding, which places additional overhead on total performance.

Styling is one of the most important processes in client-side raster visualizations. Users can apply different representation models on layers without the need to create new styling rules in spatial server applications. The time cost of applying a new style is negligible with small rasters. It can, however, expand considerably with increasing styling complexity (e.g., creating an RGB style from three bands), or by increasing the number of cells. For the Baranya imagery, styling performance (Figure 2) would no longer allow for a continuous animation, although for most GIS use cases, it is not disturbing.

When layers are rendered as textures, the application gives a very smooth experience. Navigation in both test layers is seamless and their memory footprints are low (Table 3). A small script was written to change the styling

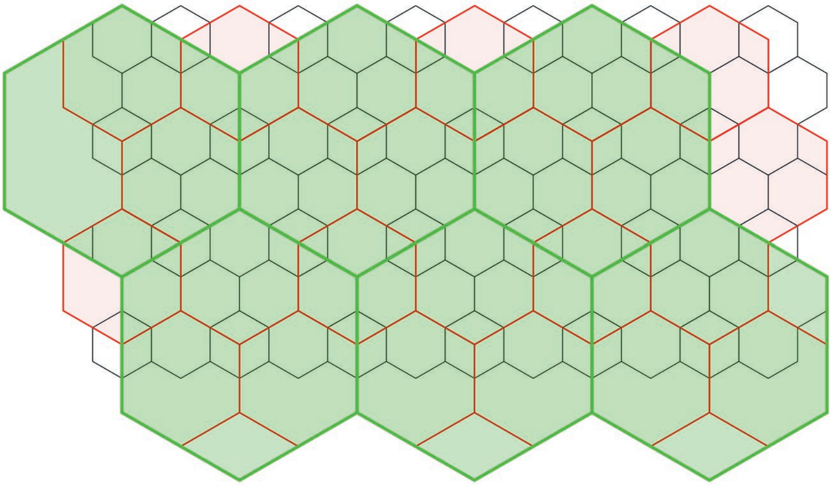

FIGURE 9 Expanding pyramids can be observed when the implementation tries to minimize the total area loss by extending the previous level. Expansion is not as severe as shrinkage, but accuracy is continuously reduced by introducing more cells with null values at every level

TABLE 1 General attributes of the raster layers examined

	Width	Height	Resolution	Format	Projection
Spearfish60	634	477	30 m × 30 m	Arc/Info ASCII Grid	EPSG:26713
Spearfish60 (hexagonal)	634	477	17 m[a]	HexASCII	EPSG:26713
Baranya imagery	3,172	2,492	30 m × 30 m	GeoTIFF	EPSG:23700

[a]HexASCII defines resolution as the length of a regular hexagon's single side.

TABLE 2 Parsing and styling metrics of the two raster layers examined. Hexagonal coverages have the same metrics as rectangular ones, since the difference between them in these phases is a single attribute value

	Bands	Cells	Time (ms)	Memory
Parsing				
Spearfish60	1	302,418	376	1.2 MB
Baranya imagery	4	31,618,496	3,435	60.3 MB
Styling				
Spearfish60	1	302,418	27	–
Baranya imagery	3	23,713,872	779	–

of the Baranya imagery between RGB and pseudocolor every 5 s (Figure 10). During restyling, the lag is almost unnoticeable. Preparing the layers (creating cached textures) increases the loading time of larger rasters. In such cases, most of that time is spent on reprojecting the image, extending the styling process to around a second.

Visualizing raster layers as vectors yields variable results. Coverage layers are disadvantageous to their raster counterparts in every metric (Table 3). The preparation time, which mostly consists of creating pyramid grids, takes several seconds, making the application unresponsive for a short period after every restyling. The memory footprints of pyramids are huge compared to textures, which is mostly due to spatial indexing and storing

TABLE 3 Rendering metrics of raster and coverage layers. For 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

	Preparation time (ms)	Draw time (ms)	Memory
Raster layer			
Spearfish60	37	3	87 kB
Baranya imagery	343	8	77.5 kB
Coverage layer			
Spearfish60	2,579	1–1,032	152.9 MB
Spearfish60 (hexagonal)	3,001	1–1,747	170.4 MB

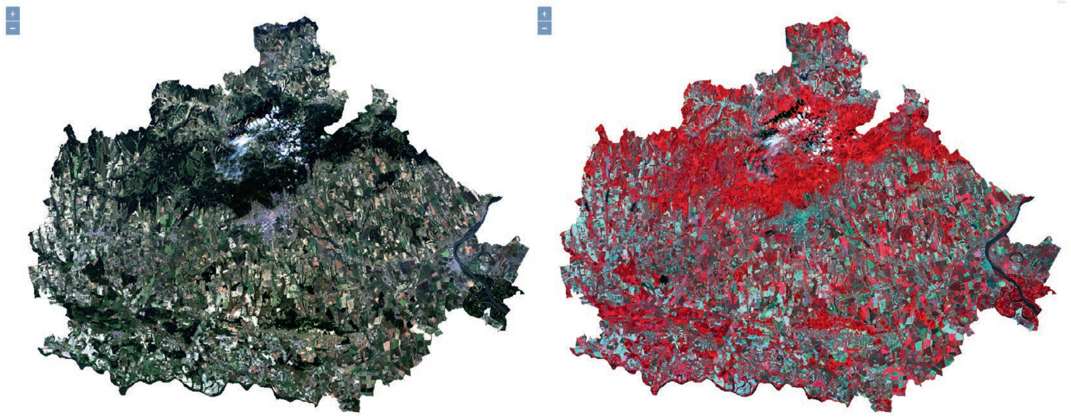

FIGURE 10 Two different stylings of the Baranya imagery using different sets of bands. The RGB styling (left) uses corresponding bands, while the pseudocolor styling (right) uses the band combination NIR, R, G, for the RGB channels

triangulated cells almost ready to draw. This can be improved with some optimization, such as storing tiles instead of cells.

The other problem is with drawing performance, which increases quadratically with almost every pyramid level (Table 4). At the largest zoom levels, even a small layer can cause massive lags. On the other hand, overall experience is still tolerable. This is caused by the OpenLayers rendering technique. It caches vertex and index buffers between complete redraws, therefore animations and interactions (e.g., panning, zooming) have improved performance. As a result, the user has a smooth experience during an interaction, and the application lags only at complete redraws.

However, the buffer caching of OpenLayers causes another problem. Cached buffers increase the memory footprint of the application. Increased memory usage is a function of drawn vertices, thus the more cells that can be seen, the more additional memory the application uses. At larger zoom levels this can be so high that memory profiling tools do not work without running out of system memory. In order to approximate the amount of memory reserved for cached data, total heap memory was recorded for each pyramid level (Table 4). Differences between adjacent levels were also calculated for easier interpretation.

While coverage layers cannot compete with texture-based techniques, it is worth considering whether they should be compared directly at all. The coverage model mitigates inherent, core limitations of the raster model, allowing for visualizing hexagonal coverages without needing to create a vulnerable, more complex, and less efficient honeycomb vector grid. Its poor performance is its most serious limitation, although it is more a trait of

TABLE 4 Rendering metrics of different pyramid levels in the Spearfish60 rectangular coverage

Level	Cells	Time (ms)	Memory	Heap memory (MB)
1	292,220	1,032	111.8 MB	365 (+172)
2	73,181	265	31.0 MB	193 (+17)
3	18,230	96	7.6 MB	176 (+5)
4	4,468	16	1.9 MB	171 (+6)
5	1,026	18	478.2 kB	165 (+1)
6	169	5	114.1 kB	164 (+0)
7	63	3	27.4 kB	164 (+0)
8	12	1	5.7 kB	164 (+0)
9	2	1	1.2 kB	164 (+0)

the current implementation than of the model itself. It is believed that tailoring the model to its environment with careful optimizations will allow real-world hexagonal coverages to be visualized efficiently. Such an optimization could be storing only cell centroids in memory, and outsourcing grid generation to the GPU.

Aside from optimization and invested developer time, raster size will always be a limitation for both raster and coverage layers. Spatial databases can store excessive amounts of raster data (e.g., global climate data sets). For spatial big data, the Web Coverage Service (WCS) standard offers a ready solution, and by using it, users can extract bounded samples of a large layer. Since the implementation can work with WCS enabled server applications (e.g., GeoServer), small subsets can be obtained from any served raster, moving the whole big data problem to the server side.

5 | CONCLUSIONS

In this article some possibilities of client-side raster visualization on the web were assessed. Two distinct techniques were implemented in the OpenLayers web mapping library as a proof of concept. The feasibility of using a vector-based raster model (coverage model) versus using traditional, texture-based raster visualization was explored.

Due to the ability to interpret binary data in JavaScript, client-side raster management is not only possible, but also efficient on the web. Furthermore, there is a stable library for reading GeoTIFF files, one of the most widely used raster formats. Since this library can be used in any data abstraction pipeline, it is easy to implement GeoTIFF support in any web mapping library. While the implementation discussed has been created for OpenLayers, it can be adapted to any other software with relative ease, requiring only adequate proficiency from the adapting developer in the target environment. Readability has always been a concern during the development process, and as the raster manager becomes a standalone module, it will contain even less OpenLayers-specific code.

Traditional raster visualization has proven to be effective even without optimizations. Applying a new style can slow down the application a bit, but with asynchronous calls (threading), visible lags can be avoided. On the other hand, coverage visualization has more severe limitations due to poor performance and large memory footprint. However, these limitations can be mitigated with some optimizations. Currently a GPU is only used for drawing cells, while it could generate grids at the expense of limiting users to rectangular and hexagonal grids. Memory efficiency also would be enhanced by that, while a further optimization could be indexing group of cells (tiles) instead of individual cells.

Sadly, no matter how well a coverage implementation is optimized, it cannot compete with a texture-based raster implementation in terms of either performance or memory consumption. That said, with careful optimization, coverages can extend the raster model. The coverage model offers capabilities beyond the limits of traditional rasters, which can be useful in several disciplines. It can be used to introduce hexagonal coverages into GIS software, or even custom patterns with continuous coverage and guaranteed data integrity, where necessary. When the efficacy of coverage visualization achieves an optimal level, it could even make traditional raster visualization an optimization step for limited systems (e.g., embedded systems, battery-operated devices).

It should also be borne in mind that, in a desktop application, the same algorithm can produce better metrics than on the web. Furthermore, desktop graphics libraries (e.g., OpenGL) have better capabilities than their web-based counterparts. Consequently, implementing the coverage model in a desktop application should increase performance significantly.

ACKNOWLEDGEMENTS

The author would like to thank Titusz Bugya for sharing his thoughts on breaking up hexagons to triangles. The study was supported by the ÚNKP-17-3-I. New National Excellence Program of the Ministry of Human Capacities, Hungary. The project has been supported by the European Union, co-financed by European Social Fund Grant no. EFOP-3.6.1.-16-2016-00004, "Comprehensive Development for Implementing Smart Specialization Strategies at the University of Pécs."

ORCID

Gábor Farkas https://orcid.org/0000-0002-9871-3556

NOTE

[1]The code can be found at https://github.com/GaborFarkas/ol3/tree/raster_base

REFERENCES

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*, 347–359.

Bugya, T., & Farkas, G. (2018). An alternative raster display model. In *Proceedings of the Fourth International Conference on Geographical Information Systems Theory, Applications and Management, Funchal, Portugal* (pp. 262–268). Piscataway, NJ: IEEE.

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

de Sousa, L. M., Nery, F., Sousa, R., & Matos, J. (2006). Assessing the accuracy of hexagonal versus square tilled grids in preserving DEM surface flow directions. In M. Caetano & M. Painho (Eds.), *Proceedings of the Seventh International Symposium on Spatial Accuracy Assessment in Natural Resources and Environmental Sciences, Lisbon, Portugal* (pp. 191–200).

Di Pasquale, D., Fresta, G., Maiellaro, N., Padula, M., & Scala, P. L. (2012). New frontiers for WebGIS platforms generation. In I. Maurtua (Ed.), *Human machine interaction: Getting closer* (pp. 85–114). London, UK: Intech Open.

Doyle, A. (2000). *OpenGIS web map server interface implementation specification revision 1.0.0* (Technical Report No. 00-028). Wayland, MA: Open Geospatial Consortium.

Ecma International. (2015). *ECMAScript R 2015 language specification* (Technical Report No. ECMA-262). Geneva, Switzerland: Author.

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 V. Molnár (Ed.), *Az elmélet és a gyakorlat találkozása a térinformatikában IX: Theory meets practice in GIS* (pp. 107–113). Debrecen, Hungary: Debrecen University Press.

Gawne-Cain, A., & Holcroft, C. (2000). An introduction to OpenGIS. *GI News*, *September*, 54–57.

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

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

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

Peterson, M. P. (1999). Trends in internet map use: A second look. In *Proceedings of the 19th International Cartographic Conference, ICA, Ottawa, Canada* (pp. 571–580). Victoria, BC, Canada: Department of Geography, University of Victoria.

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.

Schindler, F. (2016). *geotiff.js and plotty.js: Visualizing scientific raster data in the browser.* Retrieved from https://av.tib.eu/media/20373

Taivalsaari, A., Mikkonen, T., Anttonen, M., & Salminen, A. (2011). The death of binary software: End user software moves to the web. In *Proceedings of the Ninth International Conference on Creating, Connecting and Collaborating through Computing, Kyoto, Japan* (pp. 17–23). Piscataway, NJ: IEEE.

Warmerdam, F. (2008). The geospatial data abstraction library. In G. B. Hall & M. G. Leahy (Eds.), *Open source approaches in spatial data handling* (pp. 87–104). Berlin, Germany: Springer.

Wendlik, V., Karut, I., & Behr, F. (2011). Tiling concepts and tile indexing in Internet mapping APIs. In F. Behr, M. Ngigi, A. P. Pradeepkumar, & M. Zimmermann (Eds.), *Geoinformation for a better world: Applied geoinformatics for society and environment* (pp. 116–121). Nairobi, Kenya: AGSE.

How to cite this article: Farkas G. Possibilities of using raster data in client-side web maps. *Transactions in GIS*. 2020;24:72–84. https://doi.org/10.1111/tgis.12588