Towards visualizing coverage data on the Web

Gábor Farkas

Lecturer, University of Pécs, Faculty of Sciences, Department of Cartography and Geoinformatics, gfarkas@gamma.ttk.pte.hu

Abstract: This paper presents a Web-based implementation of the coverage model, which was proposed to extend the traditional raster model. While the coverage model is able to overcome the most severe disadvantages of the raster model (e.g. reprojection, sampling bias), its vector-based display model needs more computational power. In order to create a proof of concept, the web mapping library OpenLayers was used as a basis for the implementation. After several failed attempts, the final program was able to render the Spearfish60 DEM from GRASS GIS's example data feasibly. While both the rendering speed (20.8 fps) and the memory footprint (112 MiB) can be improved in the future, the results are promising for the first viable implementation.

Introduction

Raster is one of the two most significant data models used in GIS. It can be described as numeric values placed in a regular grid with rectangular cells. Each cell contains a single value, represented by a color according to coloring rules, creating a continuous coverage without gaps or overlaps between neighboring cells. In the 1960s, the raster model was created with paper maps (pen plotting), and specificities of the programming language FORTRAN in mind (Lim H. 2008). Later, the model was generalized as much as it was possible without making essential changes to the underlying concept.

Since the raster model was created considering the most trivial visualization of the underlying matrix, it only left limited space for further improvements. On the other hand, demand rose for special rasters – in particular hexagonal (Jurasinski G. – Beierkuhnlein C. 2006; Birch et al. 2007) – which the traditional model could note cope with. As a result, these special needs got addressed using the most feasible method regarding effort: by laying down regular hexagonal vector grids widely known as honeycombs. While this solved spatial bias (de Sousa L.M. – Leitão J.P. 2018), one of the most canonical problems of traditional rasters, this approach sacrifices every benefit of the raster model to do so.

Attempting to overcome the disadvantages of the raster model, and extend it further in the same time, a new coverage model (Bugya T. – Farkas G. 2018) was proposed. This model keeps the raster data model (matrices), while replaces its representation model with a vector based one. Consequently, with a trivial mapping between the data matrix and the pattern, one can define seamless coverages covering an extent even with multiple shapes. Due to the semi-vector nature of this model, the texture-based disadvantages of traditional raster (e.g. hard reprojection, hard rotation, unable to use hexagons) are mitigated. However, as cells are rendered as polygons, the performance compared to using textures should be considered.

While the performance penalty from using polygons instead of textures might not be a problem in case of a desktop application, it is more interesting to see how this new model affects a Web-based application. Due to security reasons, browsers have limited access to disk space and memory. Moreover, as JavaScript is an interpreted runtime programming language, it is outperformed by compiled programming languages utilized by desktop applications (Crockford 2008). Finally, browsers can use multiple rendering techniques and provide different Application Programming Interfaces (APIs) for them. These techniques named Document Object Model (DOM), HTML Canvas 2D Context (Canvas), and WebGL, perform differently (Lienert et al. 2012). For performance reasons DOM can be rarely found in modern Web GIS applications, however Canvas and WebGL are popular technologies.

Materials and methods

In this paper, an early implementation of the coverage model is explored, mostly in terms of performance. For the basis of the implementation, the open source web mapping library OpenLayers was chosen. Since OpenLayers has a nicely abstracted code base, the implementation was easy and straightforward. Furthermore, the library can use two different rendering engines, a Canvas and the WebGL based one. They are both hardware-accelerated renderers, although WebGL offers more space for optimizations. Consequently, the effect of using coverage model in Web applications could be assessed more subtly, by implementing it with both of the supported rendering engines.

Data and device

The benchmarks were carried out on a Dell Inspiron 7567 machine. It has an Intel Core i7-7700HQ CPU, 8 GB DDR4 RAM, and a 15.6-inch display with native resolution of 1920x1080 pixels. The workstation's integrated GPU was used for the tests, which is an Intel HD Graphics 630. The browser used was a 64-bit Chromium 64.0 built on a Debian 9.3 64-bit operating system. Due to length and time constraints, other browsers were not considered in this study. Performance was measured using Chromium's Developer Tools.

From the full set of different coverages, the rectangular ones were evaluated. These look the same as traditional rasters, but rendered using rectangular polygons as cells. The data used was the popular sfdem raster layer from GRASS GIS's Spearfish60 example dataset. It is a Digital Elevation Model (DEM) with 30x30 meters resolution in Universal Transverse Mercator (UTM) projection. Since performance can be a function of zoom level (scale), the largest zoom level was chosen which could still contain the whole layer.

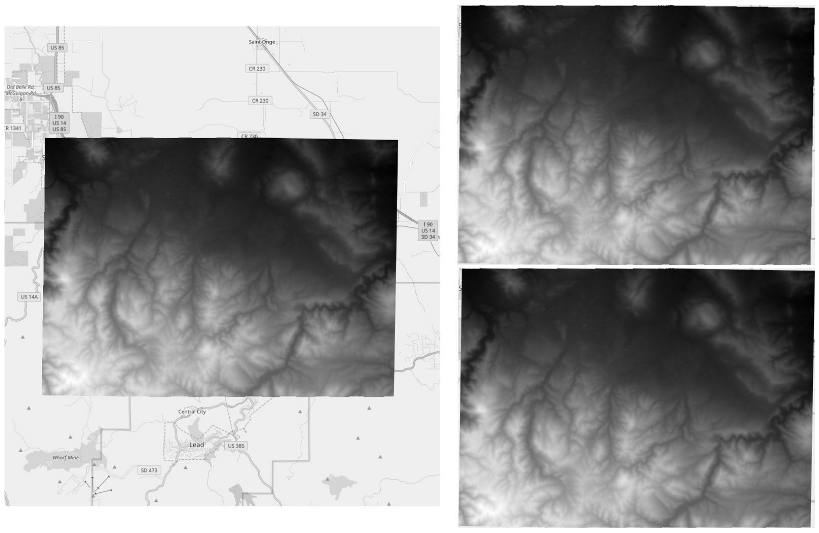

Fig. 1. Spearfish60 DEM as traditional raster reprojected to Web Mercator (EPSG:3857) with some OpenStreetMap context (left). The same layer displayed as a coverage with the Canvas renderer (upper right), and with the WebGL renderer (lower right)

Implementation

After having a working traditional raster renderer for visual comparison (*Fig. 1.*), the coverage renderer was firstly implemented for the Canvas engine. For this renderer, the original vector rendering engine was slightly modified, but no severe changes were done. Besides a custom ol/renderer/canvas/CoverageLayer class, only common, renderer independent classes were developed and used. In this case, rendering cells became the responsibility of the existing polygon renderer.

The WebGL implementation needed more changes, and renderer related classes. Considering WebGL's customizability, this engine was more carefully tailored to the coverage model. That not only resulted in an ol/renderer/webgl/CoverageLayer class, but also a new vector rendering class: ol/render/webgl/CoverageReplay. The latter class can take a list of cells as input, and can draw them on the map canvas efficiently.

Due to conditions beyond the author's control, the fate of the implementation is yet to be decided. Currently the code can be reached at the author's OpenLayers fork on GitHub (Internet1). Later, it will either get merged in the OpenLayers code base, or released as an independent module in a dedicated GitHub repository.

Results

The implementation process involved constant monitoring for both of the rendering engines, helping to track down bottlenecks and applying improvements. However, this was only done to the extent of feasibly creating a first viable product, therefore there is still some space for further optimizations.

Canvas implementation: the first steps

Since the first version of the coverage renderer was built for OpenLayers' Canvas rendering engine, some general, coverage related, but renderer independent development had to be done in this phase. Converting a raster file to a coverage layer has three major steps:

1. Parse and style raster data (usual raster processing)
2. Create a vector grid with styled cell values assigned. Reproject if needed.
3. Draw the vector grid to the map canvas.

While the first step is part of common raster processing and styling, the other two needed additional coding. In the implementation, the vector grid was created according to a pattern (e.g. rectangular, hexagonal, custom), reprojected in place, and cached. Drawing on the other hand – as pixel positions change with panning and zooming – had to be called frequently. As a result, the major bottleneck in the process could be identified as drawing speed (*Table 1*).

Thanks to OpenLayers' optimized rendering pipeline, there are two kind of rendering processes. One of them is the complete redraw of a frame, which occurs when the map comes to a stop. The other one is a replaying mechanism for speeding up rendering during an animation or interaction. During a replay, OpenLayers caches everything it can from the previous complete redraw, therefore new features are not introduced to the scene.

While the one time cost of creating grids can be called optimal, redrawing the whole layer was very expensive. Moreover, as only about 24% of total CPU time could be spared by caching, the overall performance of the Canvas engine was unacceptable for the coverage model. Making performance even worse, additional measures had to be applied on cells with edges not parallel with axes (*Fig. 2.*). In these cases, the renderer occasionally left pixel-wide gaps between cells, which could be resolved by using a cosmetic stroke on the cells. While this method required minimal additional code, stroking added 868.7 ms drawing time with little space for further optimizations.

Table 1. Performance of different phases of rendering the Spearfish60 DEM as a coverage using the Canvas engine

	Create grid	**Redraw**	**Animate**
Time (ms)	769.3	2202.1	1678.6
Frames per second	1.3	0.5	1.7

Table 2. Memory footprint of different permanent structures created during rendering Spearfish60 DEM as a coverage with the Canvas engine

	Cached by renderer (animation)	**Grid without R-Tree**	**Grid with R-Tree**
Memory (MiB)	140.0	69.3	112.5

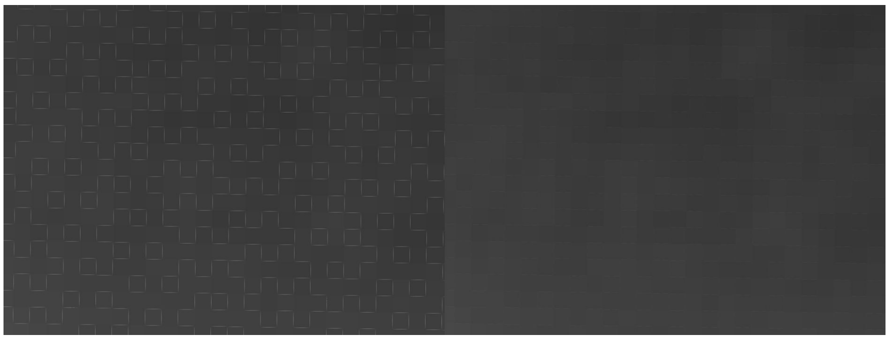
Fig. 2. The same scene before (left) and after (right) applying a cosmetic stroke using the Canvas engine

Although the Canvas experiment could not be considered successful, a renderer independent problem was still identified and resolved: constant drawing times. One of the advantages of the vector model is the ability of reducing processing time by filtering out irrelevant features using a spatial filter. This advantage could be utilized for the coverage model by indexing cell geometries using a spatial index, like an R-Tree. While loading already cached cells into an R-Tree caused minimal overhead (290 ms), it only affected the one time cost of creating grids. However, consumed memory was considered (*Table 2*), as originally only the raw coordinates of grid cells were stored in memory. While using an R-Tree doubled the required memory, it was still lower than the renderer's memory footprint. Furthermore, as performance on higher zoom levels increased drastically, using a spatial index was considered beneficial.

Making the model faster using WebGL

In case of the WebGL engine, storing geometries without spatial indexing was not considered due to the benefits of using an R-Tree. While this implementation needed more custom classes, there were less considerations to make regarding performance. In the first phase – like in case of vector geometries – cell coordinates were passed to the GPU as vertex attributes, and colors as uniform values. That is, when a color changed between drawing two successive cells, the drawing process was halted, and the new color value was passed to the GPU. On the other hand, vertices were handed over in a single pass, using an array.

While the overall performance – mostly due to faster animations – was better than using the Canvas renderer (*Table 3*), it was still unfeasible for general use. Since the rendering pipeline was already tailored to the coverage model, time complexity could only be reduced easily on the expense of space complexity. Consequently, the drawing procedure was modified to hand over colors in a single pass along with cell coordinates to the GPU (*Table 3*).

Passing colors as vertex attributes had the largest impact on the first implementation, resulting in the first viable product. Complete redraws still took a lot of time, however, it had only minor effect on user experience, as during interactions

Table 3. Performance of complete redraws and animations with two different methods using the WebGL engine on the Spearfish60 DEM.

	Colors as uniforms		**Colors as vertex attributes**	
	Redraw	**Animate**	**Redraw**	**Animate**
Time (ms)	3009.6	410.2	2299.1	48.1
Frames per second	0.3	2.4	0.4	20.8

(e.g. pan, zoom) no lags could be observed. In order to have a full picture about the changes, memory footprints were also examined using both of the rendering methods (*Table 4*). Although supplying colors as vertex attributes multiplied the memory footprint of rendering, it was more advantageous than using the Canvas renderer.

Future improvements

There is still space for improving the coverage model in Web mapping applications further, although these are not trivial, and require further experiments. By using pyramids, and precalculating the overview matrices of coverages, the Canvas engine could be improved, and it might reach feasible performance. However, pyramids are only trivial for rectangular coverages. For other types, a general method has to be created, which can approximate overviews accurately.

Furthermore, the WebGL coverage renderer can be enhanced by porting grid calculations to the GPU. This method would not only increase performance, but also reduce memory consumption. On the other hand, only a subset of cell shapes could be supported this way, with their own hard coded rendering routines.

Conclusion

In this paper, the first Web application supporting the coverage model was presented. While the vector based coverage model offers clear benefits over the traditional raster model, its feasibility is not trivial. By exploring various rendering methods using two different engines of the web mapping library OpenLayers, a first viable program was created. This program is still far from optimal, although it showed promising results. Using the hardware-accelerated and thoroughly customizable WebGL rendering engine, 20.8 frames per second (fps) could be reached with the Spearfish60 DEM raster, treated as a coverage. Its memory footprint is quite large (112 MiB for caching the whole layer), however, it can be improved with nontrivial optimization techniques, like storing pyramids, or porting some of the calculations to the GPU.

Table 4. Memory footprints of cached data during a replay with the two WebGL rendering methods using the same dataset.

	Colors as uniforms	**Colors as vertex attributes**
Memory (MiB)	30.8	112.0

Acknowledgements

This study was supported by the ÚNKP-17-3-I New National Excellence Program of the Ministry of Human Capacities, Hungary.

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(3), pp. 347–359.

Bugya T. – Farkas G. (2018): An Alternative Raster Display Model. Proceedings of the 4th International Conference on Geographical Information Systems Theory, Applications and Management (GISTAM 2018), pp. 262–268.

Crockford, D. (2008). JavaScript: The Good Parts. O'Reilly Media, Inc.

de Sousa, L. M. – Leitão, J. P. (2018): HexASCII: a file format for cartographical hexagonal rasters.Transaction in GIS, 22(1), pp. 217–232.

Jurasinski, G. – Beierkuhnlein, C. (2006). Spatial patterns of biodiversity-assessing vegetation using hexagonal grids. Biology and environment: proceedings of the Royal Irish Academy, pp. 401–411.

Lienert, C. – Jenny, B. – Schnabel, O. – Hurni, L. (2012): Current Trends in Vector-Based Internet Mapping: A Technical Review. Online maps with apis and webservices, pp. 23–36.

Lim, H. (2008): Raster Data. Encyclopedia of GIS, pp. 949–955.

Internet resources

Internet 1 – https://github.com/GaborFarkas/ol3/tree/raster_base, Downloaded: March 2018.