Distance maps of transportation networks
web-based, interactive rendering and stylization

Conception and prototyping of an interactive rendering technique for distance maps based on OpenStreetMap graph data utilizing glTF tiles and WebGL.

Alexander Schoedon, B.Sc
Geoinformation and Visualization, M.Sc
Institute of Geography, Potsdam University
schoedon@uni-potsdam.de

ABSTRACT
Different approaches for styling and rendering maps in web applications exist. Widely used architectures are raster or vector services that provide web mapping apps with geodata in client/server models. Raster data is efficiently prerendered server-side using a static, predefined layout. Vector data can be rendered client-side using dynamic layouts with CPU-consuming JavaScript postprocessing algorithms.

In the light of powerful, dedicated graphics hardware being even available on mobile devices nowadays, this paper suggests new techniques for client-side rendering of web maps with complex geometries on GPUs. This technique allows to display complex geodata and maintain a dynamic, interactive layout while preserving real-time rendering performance and low response times.

A proof-of-concept prototype implementation is attached to this paper to allow first insights in performance and feasibility.

Categories and Subject Descriptors
E.1.3 [**Data**]: Data Structures – *Graphs and networks*; G.2.2 [**Mathematics of Computing**]: Discrete Mathematics – *Graph Theory*; H.3.5 [**Information Systems**]: Information Storage and Retrieval – *Online Information Services*; I.3.6 [**Computing Methodologies**]: Computer Graphics – *Methodology and Techniques*

General Terms
Experimentation, Performance

Keywords
Computergraphics, Geodata, Webmapping, Tiling, Distance Maps, Graph Network, WebGL, glTF

1. INTRODUCTION
Rendering transportation networks in web-based applications remains a performance critical task due to the complexity of the underlying geodata. An OpenStreetMap graph dataset snapshot of the Berlin/Brandenburg region as of October 2015 contains around 900,000 edges [22]. Using classic approaches to render such a huge dataset in a web browser either leaves users with a predefined, static layout (raster data) or a notable computation-intense rendering process (vector data).

Figure 1: A possible distance map visualization.

This paper proposes a new technology of web-based transportation network visualization with the application of line-based distance maps (compare figure 1).

1.1 Known issues
The two aforementioned solutions of styling and rendering web maps are widely established and have proven effective. But both approaches have certain drawbacks.

Data transmitted in prerendered **raster** data formats (e.g. png, jpg) does not require any client-side processing and can be compressed and cached easily. This is used by major web mapping services like Google or Bing maps. The disadvantage for interactive mapping solutions is the lack of possibilities for users to dynamically interact with the map and retrieve custom layouts at runtime without requesting a full map tile reload. Web services using this technology solve this with tiny vector overlays displaying additional user-styled information. But it is not possible to interact with the map data itself.

Geodata transmitted in **vector** formats (e.g. json, gml) opposes the raster tile approach and allows client-side stylization and rendering as the geographic raw data suddenly

becomes available for the browser. But this advantage of options utilizing the geodata in the client comes with a major drawback in performance. Both the processing of the data and the rendering for the user are solved with CPU-consuming JavaScript algorithms[1].

1.2 Challenge

The challenge of this work is twofold interesting. On the one hand it is important to enable rendering using GPU-based techniques like WebGL [26]. This allows dynamic, interactive and user-defined layouts to be rendered directly on the client's device. But on the other hand it is a must to completely eliminate the client-side postprocessing of the geodata as this becomes a major performance bottleneck with increasing data complexity.

The solution presented in this paper is a geometry-based approach rather than known vector- or raster-based solutions. It maintains the goal to allow real-time rendering with outstanding performance and very low response times for the client.

2. RELATED WORK

A general motivation on web-based distance map applications in the context of geospatial analytics offers Hollburg et al. (2012) [**?**].

Glander et al. (2010) researches on visualization of distance maps with focus on polygon-based approaches [**?**]. Vaaraniemi et al. (2011) as well as Trapp et al. (2015) develop and evaluate solutions for transport network visualization utilizing modern computer graphics [**?**][**?**].

Both Coughlin (2013) and Trevett (2013) deliver outstanding motivations on why we need a standardized data format close to hardware devices for 3D applications on the web [**?**][**?**].

Altmaier et al. (2003) was among the first to outline issues in web-based geovisualization applications [**?**]. Klimke et al. (2011) proposed a camera path specification for geovisualization services on the web and mobile devices [**?**].

3. IMPLEMENTATION

The implementation is solved by an evolutionary prototyping process. The presented prototypes are therefore not compareable but rather help to outline issues with existing solutions and pave the way to the – for this paper – final solution utilizing glTF geometry tiles. Four prototypes are implemented:

Pt. #1 Leaflet with native lines (JavaScript, GeoJSON)

Pt. #2 Leaflet with canvas overlay (WebGL, GeoJSON)

Pt. #3 Leaflet with viewport query (WebGL, GeoJSON)

Pt. #4 Leaflet with geometry tiling (WebGL, glTF)

[1]Some more recent solutions offer GPU-based rendering but fail with supplying convenient solutions of pre- or postprocessing of the vector data.

Prototype #1 is not interesting for this work as it neither uses WebGL nor glTF and is only implemented to have a classic solution available. In addition, the performance of the first prototype is very poor, even for small distance maps (5 minutes travel time). It can be discarded directly.

Prototype #2 is interesting both for the successfull WebGL integration in Leaflet and the issues arising with GeoJSON postprocessing. See section 3.2 for more details.

Prototype #3 is utilizing viewport-based server requests on GeoJSON data. This approach is consuming too much bandwidth and server-capacities. Therefore, it is discarded in favour of a tile-based approach developed in the next step.

Prototype #4 is implementing a geometry tile service explicitly utilizing the glTF data exchange format. See section 3.3 for more details.

3.1 Design decisions

Prior to implementation details, some design decisions on frameworks, data formats and geopraphic projections have to be discussed.

3.1.1 Webmapping frameworks

This project focuses on the JavaScript webmapping framework *Leaflet-JS* [15]. Better alternatives with a more advanced WebGL-integration are available (e.g. *OpenLayers 3*) but a high compatibility with the existing *Route360-JS* API developed by Motion Intelligence GmbH [24] is a requirement for this project. Therefore, Leaflet will be used.

There is no native support for WebGL in Leaflet, yet[2]. Along with this implementation, a handful of Leaflet plugins will be developed which support a future WebGL integration [19][18] (see section 3.3.3).

3.1.2 Data formats

Due to the data-intense applications in webmapping services, most implementations follow a client/server model. It's important to research and evaluate options on exchange formats for the underlying geodata.

The following formats are considered worth for comparison.

- In computer graphics, the COLLADA digital asset exchange format (`.dae`) is used for modelling purposes and exchange of editable 3D models.
- The new OpenGL transfer fromat (`.gltf`) is currently being drafted by the Khronos Group [13] and promises to be a file format more close to the hardware requirements.
- In geoinformation sciences, GeoJSON is a JavaScript object notation (`.json`) which is extended by geographic features with geometries and properties.

[2]While writing this paper, a spanish software engineer, Iván Sánchez Ortega, started working on a Leaflet.GL plugin architecture [17] which was not taken into consideration as it is still experimental and was not available when this project implementation was started.

- The Geography Markup Language (`.gml`) is an XML grammar for expressing geographical features.

Formats not taken into account are KML, Google's equivalent to GML, and TopoJSON, another JSON format with merged geometry fields. In addition, raster data formats are ignored as they do not store any extractable geometry information.

Table 1 compares the stated formats and evaluates both, their space-complexity and postprocessing requirements. The space-complexity is important to evaluate the required bandwidth for the application. The postprocessing is the aforementioned bottleneck in performance of transforming geographic data into close-to-hardware array buffers for the GPUs.

Format	Space-Complexity	Client-Postprocessing
.dae.gz	`++`	required, decompress
.bgltf	`+o`	not required
.gltf.gz	`+o`	decompress only
.dae	`oo`	required
.gltf	`o-`	not required
.json	`o-`	required
.gml	`--`	required

Table 1: Comparison of data formats

Concerning the space requirements, both glTF and COLLADA perform above average. Base of the comparison are the Cesium Milk Truck and Cesium Man by Analytical Graphics Inc [9]. The binary version of glTF (`.bgltf`) is even smaller than a gzipped version (`.gltf.gz`). Classic geodata formats fail in terms of space-complexity since both, JSON and XML formats are quite bloated.

Concerning the client-side postprocessing requirements, only the OpenGL transfer format allows to store array buffers which eleminates any javascript processing other than requesting and reading the data. This is an obvious knockout criteria for the other candidates and therefore glTF will be considered the best choice for this application. Details on the postprocessing issues will be discussed in section 3.2.3.

3.1.3 Geographic projections

To minimize the processing of data and allow easy GL-transformations, it's important to reduce reprojections and avoid spherical units (e.g. degree, lat/lon).

Leaflet uses the `EPSG:4326` standard projection which is the world geodetic system 1984 (WGS84) and uses a latitude/longitude coordinate format. This means, all programming interfaces of Leaflet return values in degree.

Internally, Leaflet uses `EPSG:3857`, the web mercator projection, a metric system going back to Gerard Mercator's flat world map in 1569 [25]. It uses northing and easting as a measure of distance in meters from the equator and the prime meridian. This is already an advantage over WGS84 as it does not require computing-intense spheric transformations in calculations. Unfortunately, Leaflet does not allow access to the internal metric system without automatically triggering one or more reprojections from/to WGS84 [23].

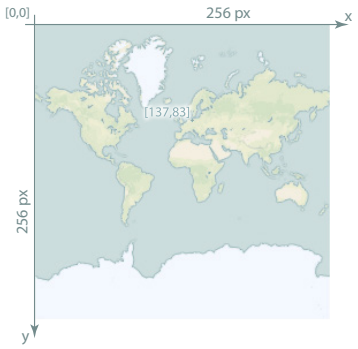

Figure 2: The world in 256 square pixels.

To simplify the geographic coordinates, all geographic references will be projected on a single tile of size 256*256 pixel with it's origin in the north-west corner (see figure 2). In this reference system, the Brandenburg Gate in Berlin would be at pixel coordinate `[137.51253, 83.9612]`. This allows enough precision, moves as close as we can get to hardware-coordinates and is still device-independent.

3.2 Prototype #2 with GeoJSON

The 2nd Prototype is implementing Sumbera's Leaflet Canvas Overlay [14]. It is using a static local GeoJSON dataset with precalculated travel times for cars starting at the Brandenburg Gate in Berlin. The data is unfiltered but truncated at 15 minutes travel time (900 seconds) to limit file size and processing amount. The file is 8.8 MiB in size [10].

3.2.1 Architecture

Leaflet provides a 2D `L.map()` canvas on a `<div id="map" />` HTML DOM element. This can be used for basic web mapping tools like raster base tiles or simple vector items. The extended class `L.canvasOverlay()` provides Leaflet with a 3D overlay which handles redrawing of the canvas and therefore allows basic WebGL context integration (figure 3).

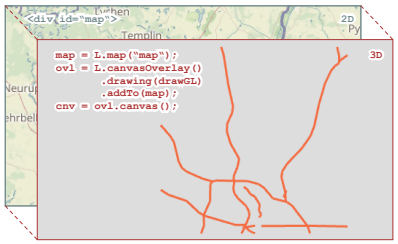

Figure 3: 3D overlay on 2D map canvas.

On each map interaction, a `drawGL()` call will be triggered by the overlay. This function reads the underlying GeoJSON and extracts all geographic features. For each feature in the featureset, the algorithm checks the travel time properties and decides whether the feature is visible (below selected threshold) or not.

Each visible feature will be converted to typed array buffers for vertices and colours. Each coordinate of the geometry has to be mapped to pixel coordinates (section 3.1.3).

3.2.2 *Viewport math*

To display geographic coordinates on a GPU-rendered map, two major steps have to be performed.

First, the geographic projection has to be transformed into a pixel projection. Therefore, the coordinate origin has to be moved to the topleft corner by translating it along the x-axis and the negative y-axis for half an equator. The units will be scaled from 40.075.016 meters to 256 pixels.

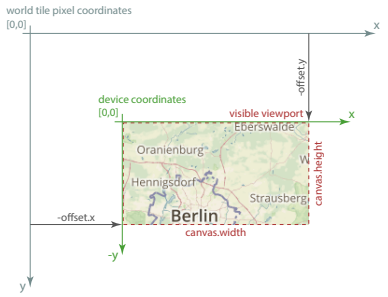

Figure 4: Viewport coordinates overview.

Finally, this pixel projection has to be made visible in the current viewport of the client's device. The device coordinate origin will be translated to the topleft corner of the visible viewport. The model view will be scaled using the current zoom level scale factor and visible canvas size. The topleft corner of the map can be retrieved from Leaflet and used as an offset for the model view to translate to the displayed geometries (figure 4).

3.2.3 *Client issues*

This implementation shows two issues.

A blocking performance issue is the conversion from geodata to array buffers. For the mentioned dataset of only 15 minutes travel time this results in an iteration over 29.864 geographic features. In addition, the coordinates have to be iterated as well, leaving this example with 129.034 spheric conversions for 64.517 coordinates.

Another problem is the unability to access the internal `EPSG:3857` metric coordinates from Leaflet. All positions have to be transformed using compute-intense spherical (degree) rather than optimized metric conversions [23].

3.3 Prototype #4 with glTF

A solution which eleminates client-side data processing is required. The 4th prototype implements a new geometry-tiling approach [11].

Map tiles are a common way to divide map data into portions of similar size. A tile at zoom level 0 shows the full world. Zoom level divides the world in 4 tiles, zoom level 2 in 16 tiles, and so on.

3.3.1 *Geometry tiling*

Table 2 highlights the advantages of geometry tiling. The layout stays dynamic and can be adjusted at runtime by custom user inputs similar to vector tiling approaches.

The data processing is moved from client-side to server-side similar to raster tiling approaches. The result is high runtime preformance with low latencies and the resulting geometries can be cached client-side.

The rendering happens on dedicated graphic devices and ensures outstanding performance even for complex geodata sets.

Tile	Layout	Processing	Rendering
Raster	static	Server	Server
Vector	dynamic	Client/CPU	Client/GPU
Geometry	dynamic	Server	Client/GPU

Table 2: Tiling options

Figure 5 shows a screenshot of a modified version of the 4th prototype. It displays the geometry tile `[2200,1343]` at zoom level 12 rendered with WebGL using random colours.

Figure 5: Geometry tile rendered with WebGL.

3.3.2 *Architecture*

The current implementation requires a client/server architecture which offers both, a tiling service supplying the precalculated geometry tiles and a routing service supplying travel times on-request at runtime (figure 6).

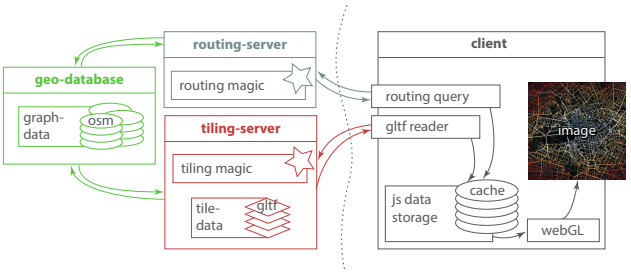

Figure 6: Architecture: tiling, routing, rendering.

The tiling server regulary fetches OpenStreetMap snapshots and loads them into a PostGIS database. A tiling algorithm developed by Silbersiepe (2015) filters the geodata based on attributes and zoom level and pregenerates tiles for at least 20 zoom levels [27].

A routing server operated by Motion Intelligence GmbH provides travel times based on the selected transportation type

(e.g. car, bike, public transport) and a starting location. This data can be requested multiple times without the need to update the underlying geometry tiles again.

3.3.3 Tile data structure

Two Leaflet plugins are being developed to manage the tiling data logic [18].

- `L.TileBuffer` is the actual class for all geometry tiles. Each instance has the properties of `x`, `y` and `zoom`. In addition it stores the three required typed array buffers to render the tile with WebGL: a `Float32Array` for vertices, a `Uint16Array` for indices and a `Float32Array` for colours. This information is enough to render the complete tile in a single draw call.
- `L.TileBufferCollection` is a class which implements the basic tile caching logic. It has a `zoom` and a `size` property. In addition, it holds a collection of `L.TileBuffer` objects for the current zoom level. As soon as the client requests a redraw of the scene, the collection can be rendered based on zoom and position of the visible tiles.

On each load event of a tile, Leaflet will request the geometry (vertices and indices) from the tiling server and the travel times (colours) from the routing server. These three arrays are recieved in glTF format. They are used to create `L.TileBuffer` objects (figure 7).

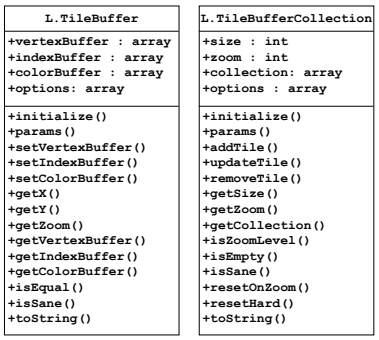

Figure 7: Two Leaflet plugins for geometry tiling.

3.3.4 Implementation status

This work focuses on frontend development only.

The backend is operated by Motion Intelligence GmbH. The geodatabase with preprocessed OpenStreetMap graph data is available. The routing server API is available, too, but lacks of a working colour buffer (glTF) implementation for transportation networks. The tiling server is available but, unfortunately, is not serving glTF geometry tiles, yet.

The frontend implementation is feature-complete. The JavaScript tile cache and the WebGL rendering logic for geometry tiles is fully implemented with prototype #4. The only piece missing is the glTF interface for the backend communication.

4. CONCLUSIONS

A possible solution for a new approach of efficiently rendering and styling interactive web maps was demonstrated with the application focus on distance maps. Possible limitations were discovered and countermeasures developed.

Geometry tiles utilizing preprocessed glTF data seem to be a promising way to render maps on both desktop and mobile devices. Future work should focus on developing a working glTF processing backend with tiling and routing server. On top of this, a fully working client/server infrastructure should be evaluated regarding it's performances. Results should be compared with classical raster or vector solutions.

APPENDIX

A. RELATED LINKS

[9] Cesium Sample Models
https://github.com/AnalyticalGraphicsInc/cesium/tree/c300529011714035cb254b493bf22b62ccf9da3d/Apps/SampleData/models

[10] Distance Maps Prototype #2
http://donschoe.github.io/leaflet-distance-maps-pt2/
https://github.com/donSchoe/leaflet-distance-maps-pt2

[11] Distance Maps Prototype #4
http://donschoe.github.io/leaflet-distance-maps-pt4/
https://github.com/donSchoe/leaflet-distance-maps-pt4

[12] GeoGL: The author's public blog
http://geogl.ghost.io

[13] glTF draft specification
https://github.com/KhronosGroup/glTF/blob/master/specification

[14] Leaflet canvas overlay
http://bl.ocks.org/Sumbera/11114288
https://github.com/donSchoe/L.CanvasOverlay

[15] Leaflet documentation
http://leafletjs.com/reference.html

[16] Leaflet heatmaps
https://github.com/Leaflet/Leaflet.heat

[17] Leaflet GL
https://github.com/IvanSanchez/Leaflet.gl

[18] Leaflet tile buffer
https://github.com/donSchoe/L.TileBuffer

[19] Leaflet WebGL
https://github.com/donSchoe/Leaflet.WebGL

[20] Mapbox Raster Tiles
https://www.mapbox.com/developers/api/maps/

[21] Mapzen Vector Tiles
https://mapzen.com/projects/vector-tiles/

[22] OpenStreetMap
https://www.openstreetmap.org

[23] Q: How to access Leaflet `EPSG:3857` coordinates?
http://gis.stackexchange.com/q/153849/13692

[24] Route360 JS API
https://developers.route360.net

[25] Time: Mercator World Map 1569
http://ideas.time.com/2013/11/21/a-history-of-the-world-in-twelve-maps/slide/gerard-mercator-world-map-1569/

[26] WebGL specification
https://khronos.org/registry/webgl/specs/latest

B. UNPUBLISHED

[27] Silbersiepe, J. Web-based rendering of street networks. Spatial Analytics. Hasso-Plattner-Institute. 2015.

C. IMAGE CREDITS

All tables in this paper are own work (tables 1, 2). All figures are own graphics (figures 2, 3, 4, 6, 7) or screenshots of own work (figures 1, 5).

Background raster tiles are provided by Mapbox [20]. Vector data is provided by Motion Intelligence [24] and Mapzen [21].

D. ACKNOWLEDGMENTS

The author would like to thank Iván Sánchez Ortega, Stanislav Sumbera and Vladimir Agafonkin for the inspiring pioneer work on Leaflet-WebGL integrations [17][14][16]. In addition, this work would not be possible without routing data provided by Henning Hollburg of Motion Intelligence GmbH (Route360 JS API) [24].

E. READ MORE

Read more on the experiences with Leaflet and WebGL on the author's blog 'GeoGL' [12].