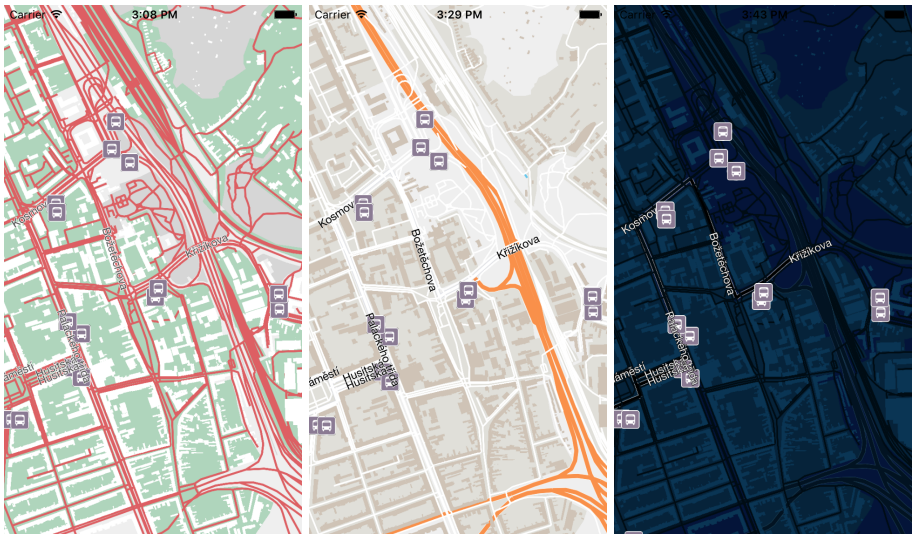
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Library for OSM Rendering on Smartphones

David Vaďura



Abstract
The aim of this work was to create a mobile map rendering library, using GPU acceleration to render vector map directly on end-user's device. That includes ability to draw line and polygon features, add POI icons and labels, while also enabling the user to change the appearance. The library must also manage map data: acquire the tiles on-demand, cache them, and display them at the appropriate time. The proposed architecture defines a generic data source to make tile loading possible from various sources, both online and offline. A hybrid approach to map rendering was adopted. Base features are rendered to a texture. Layers which contain text and icons are rendered online on top of the base features, these can be rotated or scaled when the viewport changes. A C++ library was created as the result of this work. It works on both iOS and Android, and it is possible to port to other platforms (the largest requirement is the support of OpenGL ES). What makes my work distinct from other map libraries are the clearly defined features (such as offline map rendering) that no other library fully meets. This library will be easy to use for individuals/companies developing mobile applications for multiple platforms.

Keywords: Map — Rendering — OpenGLES

Supplementary Material: *N/A*

*xvadur02@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Maps are part of many mobile applications and making them fast and beautiful takes a lot of effort. There are many libraries that render maps, while only a few of them are also multi-platform and work with vector data. If we keep adding requirements like support for offline rendering or commercial use, we quickly realize that there is almost no choice of libraries with all these features when developing mobile applications.

The goal of this work is to create a map rendering library with the following requirements:

- Render vector map on mobile phones at a rea-

sonable speed to provide smooth controls to the user.
- Work with both online and offline sources of data.
- Provide complex styling options defined in a style file. The appearance (color, ouline, line width, etc.) of every feature can be modified.

Let us compare these requirements with existing solutions on the market. Probably the most common and quickest approach are the Google Maps [1] and MapKit [2]. These are the libraries native to the two target platforms: Android and iOS. They both come with simple API, proprietary maps and great support. They also support custom annotations on top of the built-in map. The biggest issue with them is the online-only map data, which is not suitable for many applications. If the corresponding application requires internet connection to work, both of these would be a good choices However, if the developer needs more features, like custom maps or offline rendering, they do not yet have these features in the year 2017.

At the time of writing of this paper, the greatest alternative is, without doubt, Mapbox GL Native [3]. This is the native counterpart to the Mapbox GL JS - Javascript library for the web. It supports both Android and iOS, and the code itself is open-source as well, which proves helpful to the development. The default way to use it is together with Mapbox owned map data. The company provides multiple plans for both free and commercial apps, with limits on map views/users and other features. Unfortunately, there is very limited choice between the plans and the pricing is quite expensive for medium sized applications.

The most interesting from the rest is tangrames [4], which is again, as with Mapbox, a port of Javascript-first library from company Mapzen. The library is still early in the development (at least on mobile), so many of the required features (for example offline mapping) are yet to be added. It might become a good alternative to Mapbox in the future, but the current feature set is more limited.

Another part in the research was spent to compare the sources of map data for the developed library. OpenStreetMap (OSM) stores its own data in the XML format, and it also contains a huge amount of data unnecessary for the rendering. For that reason it is not used directly, but other providers compile and maintain copies of OSM specifically for the purpose of vector map rendering. One of them is the aforementioned company Mapzen, which currently provides an online data source free of charge. The other project, which I used while working on my library, is OpenMapTiles.org. It provides downloads of packages of cities, countries, or the whole planet.

2. Tile Loading and Processing

The typical internal representation of computer map is with tiles. A tile is a small slice of the resulting map, often with the size of 256×256 pixels. To optimise loading and displaying of such map, its tiles are organised into zoom levels (going from 0 to 19 in the case of OpenStreetMap). The number of tiles grows with the zoom level. The consequence of this is that the amount of details can change as the user zooms the map.

There are many formats that are actively used to store vector map data. As mentioned before, OSM uses XML but that is for obvious reasons very inefficient for the mobile platform. Other text-based formats like GeoJSON are also very popular, but I chose the binary format Mapbox Vector Tile (MVT). It is generally smaller than text-based formats and there is no problem with using a binary format on mobile (unlike web, where GeoJSON is more appropriate). It is not a format per se, but a specification of the data stored inside protocol buffers [5]. To speed up the parsing even more, I did not use the Google Protobuf library, but a minimal implementation protozero [6]. One of the features of the Protobuf library is automatic generation of classes in the target language based on a definition file (.proto). The deserialization methods in the generated code however proved too slow for realtime applications, because they also do a lot of unnecessary work (validation of each field for example). For that reason I implemented the deserialization manually using protozero.

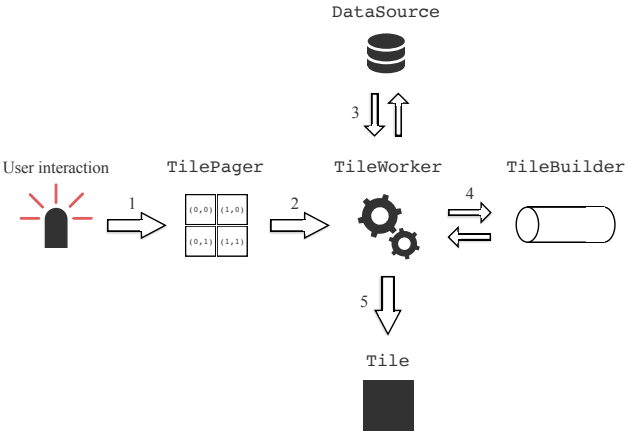


Figure 1. This figure illustrates the process of loading and processing data into Tiles. The step numbers correspond to the list in this chapter.

The proposed architecture divides the processing pipeline into a sequence of steps, as illustrated in Fig-

ure 1:

1. Every time the viewport changes (caused most of the time by user interaction) `TilePager` is given the bounds of the visible part of the screen and the current scale. From these, `TilePager` determines indices of tiles that should be visible. After that follows an iteration over all these indices and a check if they are already available in the cache. In the case of a cache miss, the tiles are queued up in the `TileWorkerQueue`.
2. `TileWorkerQueue` starts with a predefined number of background threads. These consume the tasks from the queue and manage the loading and processing of the tiles. After a thread acquires a tile that needs to be processed, first of all it requests data from its `DataSource`.
3. `DataSource` downloads/reads tile data. The tiles are most often situated on online Web Map Service (WMS), or inside an offline database.
4. Tile data is parsed with an appropriate parser. The result of this process is an instance of `Tile`, which is ready for rendering. The process of parsing is tightly connected with the *symbolizing* illustrated in Figure 4, which transforms the OSM geometry to triangles that can be pushed to the GPU.
5. Tile is moved to the `GPUWorker` thread, which renders its static geometry to a texture. After this step, the tile is ready to be displayed on the screen.

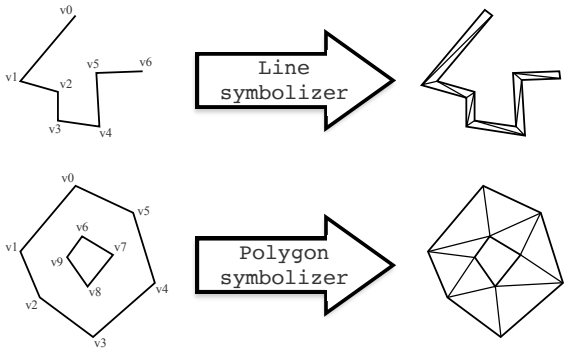


Figure 2. Lines and polygons from the OSM data go through a process called symbolizing. The geometry is tessellated there so that it can be rendered with OpenGL.

To avoid loading the same tile multiple times when user scrolls the map, they are stored in a cache. For maps, a simple Least Recently Used (LRU) cache can be enough. The key in my implementation is `TileID`, which is composed of the *x*, *y* and *zoom* coordinates of the tile.

3. Styling the Map

One of the advantages of vector maps is the opportunity to change the appearance on the fly, without modifying the source data at all. By appearance I mean even choosing which features to show and which not, allowing the user to filter their visibility online.

Styling for the library is defined in a style file in JSON format. It consists of multiple objects, the most important being "layers" array containing description of every single layer (Figure 3) which gets shown in the resulting map.

Each layer must contain several required attributes and any number of other optional ones. Required are:

- **Source-layer** is the name of the layer from OSM data.
- **Type** describes types of geometry of this layer. It can be one of the following: line, polygon, icon, label. Based on this attribute an instance of subclass of the base `StyleLayer` object is created, e.g. `LineLayer` or `PolygonLayer`.

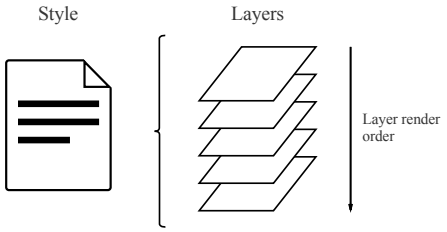


Figure 3. The style file contains an array of layers, which are rendered in the order they appear in the file. The only exception to this are icons and text, that are always rendered above the other layers.

When building Tiles from the raw data, the `Style` is used to filter features that should go into it. The result is saving time and memory by skipping all features that will not appear on the screen. The rest goes into the symbolizing step, where it is converted to OpenGL geometry.

The other step where `Style` comes in is the rendering itself. To optimise the process, features belonging to the same layer are all grouped into one "bucket", basically a list of features of the same type. When it comes to rendering them, it happens by iterating over the buckets and rendering all of their contents. This means that the shader uniforms for each layer have to be set only once before rendering the first feature from the bucket and it results in fewer state changes.

4. Rendering Map Features with OpenGL

The rendering happens in two phases:

1. The first phase is executed right after a tile is done loading. Layers containing geometry other than text and icons are rendered on a background thread using a shared OpenGL ES context into texture.
2. The second phase is the actual rendering to the framebuffer. The pre-rendered textures are rendered first and text and icons appear above them. To reduce stress on the device, redraw only happens if the map has changed in some way (new tiles are ready, user moves the map, etc.).

Moving polygon and line rendering to the background removes the limit of 16 ms that are allocated for each frame (if we want the map to update at 60 frames per second). This removes a lot of the limitations on the amount of details in the map. The elapsed time only delays the moment the tile appears on the screen for the first time.

The text and icons, on the other hand, must be displayed in realtime, because (1) text could become hard to read after scaling; (2) some labels and icons must rotate to stay parallel with the bottom of the screen. As a result of this approach, text and icons can not appear under any other layer of the map. However, I have not seen a map that would require this, yet.

Vector maps usually require only four types of geometry: lines, polygons, icons, and text. Each of them obviously needs a different approach and algorithms.

4.1 Polygon Features

Polygon rendering is the pretty straightforward. OSM can contain polygons with holes, so an algorithm supporting hole removal is necessary. To tesselate the polygons I used the earcut algorithm, more specifically Mapbox developed library earcut [7].

4.2 POI

Even easier to render are the icons that are used for Points of Interest (POI). All icons are packed into one/few textures (depending on their size) to minimise the number of state changes. To represent a POI on the map, I used texture mapped quads.

4.3 Line Features

Drawing lines with OpenGL is most of the time much more complicated then it appears to be. Even though the OpenGL standard defines a geometry type GL_LINES, it is almost never the right way to render lines. The biggest limitation is the line width, which is not specified in the standard. As a result, the maximum width is hardware-specific and often too small (around 10 pixels). Since the users of this library would expect to be able to set any line width, we had to find another way to draw lines.

To circumvent these limitations I render lines using triangles. Although this is obviously more expensive (another step of tessellating is needed and each line segment results in at least two triangles), it gives me more control over the result. Different line joins can be chosen (miter, round, bevel), as well as caps (round, square) if the tessellation supports it.

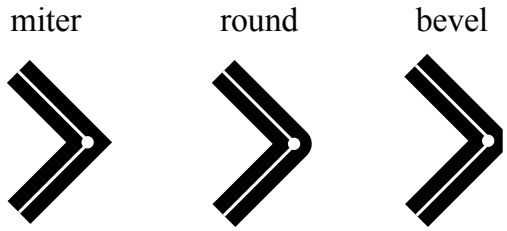


Figure 4. This illustration shows three most used line join types.

With the limited time I did not implement neither joins or caps. The minimal requirement for line rendering was support of dash patterns, which was implemented in a GLSL shader. The other way would require to split lines into short segments in the symbolizer, which would be slower as it would have to be done on the CPU. The shader, on the other hand, is pretty simple and it only requires one extra vertex attribute – distance along the line. This is the sum of the distances along the line up to the corresponding vertex. The texture that is based on the pattern is illustrated in Figure 5.

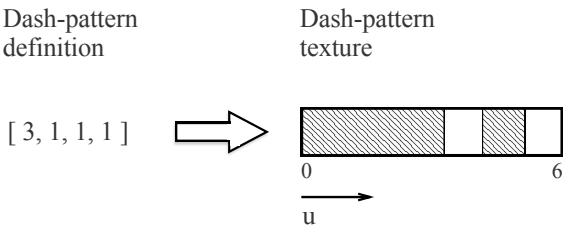


Figure 5. Dash-pattern is defined as an array, each element describing the length of the corresponding section. The picture illustrates how the pattern is converted into an OpenGL texture.

With the use of interpolation across vertices, this value represents distance of each fragment from the beginning of the linestring as shown in Figure 6. It is used as the x coordinate to sample inside a 1D texture that represents the required dash-pattern. Inside this texture, white colour represents line, while black represents the gaps between. This texture is constructed as a part of the style parsing process and passed as an uniform every time the line is drawn.

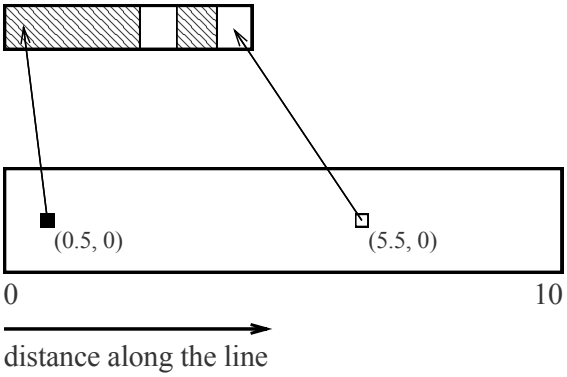


Figure 6. When a line should be drawn with a dash pattern, alpha value of each pixel is sampled from the pattern texture. The x position in the texture equals the distance of the point from the line beginning.

4.4 Labels

Because maps usually contain a large amount of text, high quality text rendering was required. The most often used technique, which is fast but does not give very satisfying results, is pre-rendering glyphs and packing them into one texture. The drawback to this approach is that the text gets blurry when up-scaled or rotated. An example of such a library is FreeType [8].

Alternative approach that I used in my work is rendering fonts using Signed Distance Fields (SDF) developed by Valve [9]. I have written a GLSL shader, also allowing to set outer halo of glyphs (used to better distinguish letters from the layers below).

5. Conclusion

The goal of this work was to define and develop a multi-platform map rendering library for smartphones with multiple required features, namely vector data processing and offline usage. In this paper we summarised the used architecture. It defines the basic pipeline for processing the map data and describes the required algorithms for rendering different features that are part of the OpenStreetMap.

The library is currently running on both platforms it is targeting (Android and iOS). It can work with both online and offline MVT tiles. The rendering and basic styling is ready to be used and further expanded. The performance is also excellent most of the time, hitting 60 fps on Galaxy S4. Despite that, the code is still going through a lot of testing and debugging.

The library obviously needs many improvements and new features that would be used by production apps. Our future plans are to clean-up and optimise the existing code before implementing new features. I intend to keep working on this project and release a working library within this year.

Acknowledgements

I would like to thank my supervisor Herout Adam, prof. Ing., Ph.D., for his help.

I would also like to thank to Vladislav Skoumal, Ing., with whom I consulted my work regularly.

References

[1] Google Inc. Google Maps. https://developers.google.com/maps/. Accessed April 23, 2017.

[2] Apple Inc. MapKit. https://developer.apple.com/reference/mapkit. Accessed April 23, 2017.

[3] Mapbox. Mapbox GL Native. https://github.com/mapbox/mapbox-gl-native. Accessed April 23, 2017.

[4] Mapzen. Tangram-es. https://github.com/tangrams/tangram-es. Accessed April 23, 2017.

[5] Google Inc. Protocol Buffers. https://developers.google.com/protocol-buffers/. Accessed April 23, 2017.

[6] Mapbox. protozero. https://github.com/mapbox/protozero. Accessed April 23, 2017.

[7] Mapbox. earcut.hpp. https://github.com/mapbox/earcut.hpp. Accessed April 23, 2017.

[8] David Turner, Robert Wilhelm, Werner Lemberg, and contributors. FreeType. https://www.freetype.org. Accessed April 23, 2017.

[9] Chris Green of Valve. Improved alpha-tested magnification for vector textures and special effects. 2007. SIGGRAPH Course on Advanced Real-Time Rendering in 3D Graphics and Games. http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf.

[10] James M. Van Verth and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications*. A K Peters/CRC Press, 3 edition, 2015. ISBN 1482250926.

[11] Kevin Brothaler. *OpenGL ES 2 for Android: A Quick-Start Guide (Pragmatic Programmers)*. Pragmatic Bookshelf, 2013. ISBN 1937785343.