Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

An in-depth analysis of dynamically rendered vector-based maps with WebGL using Mapbox GL JS

by

Oskar Eriksson & Emil Rydkvist

LIU-IDA/LITH-EX-A--15/046--SE

2015-08-26

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

An in-depth analysis of dynamically rendered vector-based maps with WebGL using Mapbox GL JS

Department of Computer and Information Science

Linköpings universitet

Oskar Eriksson and Emil Rydkvist

2015-08-26

Abstract

The regular way of displaying maps in a web browser is by downloading raster images from a server and lay them side by side to make up a map. If any information on the map is changed, new images has to be downloaded, it cannot be done on the client. The introduction of WebGL opens up a whole new world of delivering advanced graphics content to the end user in a web browser. Utilizing this technology for displaying maps means only the source data is sent to the web browser where the map gets rendered using the device's GPU. This adds a number of benefits such as the ability of changing map appearance on the client, add new features to the map and often less data transfer. It however sets higher expectations of the client device's hardware as it needs to render the map at a high enough frame rate to not appear slow and unresponsive. This thesis investigates a framework for client side map rendering in a web browser, Mapbox GL JS, with focus on performance. It shows how map source data can be generated as well as its corresponding style rules are constructed with performance in mind. It provides benchmarking results of different map data sets with different detail intensity and shows that a device with good GPU performance is needed for an acceptable user experience. It also shows that lowering the amount of rendered detail does not necessarily result in better performance.

Glossary

frame rate The rate of which the screen produces new frames. In this thesis frame rate is expressed and measured in frames per second (FPS).

GPU Graphics processing unit. A GPU is a specialized processor designed to offload specific tasks from a computers main processor (CPU) such as rendering computer graphics.

Mapbox GL JS An open-source client side map renderer for web platforms. Developed by Mapbox [17].

zoom level In the context of digital maps, the size of the geographical area that can fit on a fixed size display. A lower zoom level corresponds to a bigger area than a higher zoom level.

Contents

Chapter 1

Introduction

Digital maps are a common visualization tool in web applications. An advantage of digital maps compared to traditional maps is the possibility for users to interact with the map. Using this technique it makes it possible to make advanced map interfaces used as a tool for business relying on maps as a mean of information.

Traditionally, digital maps are rendered on the server and sent as images to the client, however new web technology opens up possibilities for dynamic rendering of maps in a web browser. This leads to more ways of interacting with the map and changing its appearance depending on the situation, without the need of an extensive server application.

When the amount of data that needs to be displayed becomes large, it is crucial to render the map in an efficient way. If the rendering performance is bad, in form of speed or image quality, the user experience might be poor. By using the graphics processor of the device, the rendering performance can be optimized. This thesis examines the open-source framework Mapbox GL JS and investigates whether it is suited to dynamically render vector based maps in a browser. The definition of dynamic rendering is described in chapter 2.3.

1.1 Motivation

IT-Bolaget Per & Per has developed a framework for displaying data heavy vector based maps in mobile applications. It currently supports devices running iOS. The framework is based on OpenGL, a cross-language API for rendering 2D and 3D graphics. The current framework shares data formats and other software components with Mapbox's map components. This is a contributing factor to the interest to investigate Mapbox GL JS. IT-Bolaget Per & Per sees the possibility of taking this type of map rendering to the browser in order to create responsive and user-friendly web-based map applications.

1.2 Problem description

Developing a framework for rendering vector based maps in a web browser is a non-trivial and resource demanding task. It could therefore be beneficial to use an existing open source implementation like Mapbox GL JS, and if needed modify it to suit the specific application needs. It is also crucial that the end product is performing well, which could be a problem when using a map data set that has lots of features and geometries.

1.3 Goal

- With focus on perceived performance, investigate whether the framework Mapbox GL JS is suited to dynamically render data heavy vector based maps in a browser using the GPU.
- Analyze how the input vector data and its corresponding style rules affects perceived performance.
- Present possible solutions and theories how vector data and style rules can be optimized to gain better perceived performance.

A prototype that uses Mapbox GL JS will be developed and presented. The prototype will be used to determine if Mapbox GL JS is a suitable solution to render large amounts of vector data in the form of a digital map. The most important criteria to see if Mapbox GL JS is a suitable solution for the problem is performance in form of frame rate.

1.4 Approach

When an user utilizes an application built on Mapbox GL JS, there are many aspects of the implementation that may affect the the user experience negatively. The most notable ways they appear to the user is by the application having low frame rate or contents of the map are loading slow, and thus not appear at all. Our focus is to analyze how the map data set together with its style rules affects the performance of the application and the end user experience.

A prototype was developed to showcase how IT-Bolaget Per & Per's current vector tile map data and associated stylesheets perform in Mapbox GL JS. Optimizations and alternative solutions were made where possible and where we saw it had the biggest impact on performance.

An in-depth analysis that compares different implementations, map datasets and use-cases was made. This can then serve as a good foundation to stand on when IT-Bolaget Per & Per is choosing directions.

1.5 Scope and limitations

The main scope for this master thesis is performance in client rendered vector tile based maps. We have chosen to limit this thesis to investigate map data sets and their corresponding styles impact on performance. Limitations will be made in terms of which browsers and devices that will be tested. All tests were performed on Macbook Pro's with different hardware in the browser Goggle Chrome.

Chapter 2

Background

2.1 IT-Bolaget Per & Per

IT-Bolaget Per & Per is a consulting firm located in Mjärdevi Science Park, Linköping, Sweden. They are active in several business areas, where their main focus at the moment is on developing user friendly mobile experiences for the forest industry where map oriented applications are the main area of interest. These application are used by IT-Bolaget Per & Per's customer to display and interact with geographical data in several different formats, depending on the current use case. IT-Bolaget Per & Per strives to push limits and try out new technologies, where this master thesis is a part of that mission.

2.2 Digital maps

Digital maps can be found in many different devices, where web browsers and mobile apps for smartphones or tablets are common. The traditional way of storing and viewing a digital map is by dividing it into square images and then only view the specific images for that area that is being shown at the moment. If the user shall be able to zoom the map, there has to be images optimized for each zoom level so that the map doesn't appear blurry or if the map should contain more or less details depending on zoom level. Map portions saved like this are called raster tiles. However, doing this is not very ideal in terms of storage and limitations in customization, so finding another solution can be of interest in modern application. [1] [28]

Storing the map data in vector format is a wide used technique nowadays that makes it possible to affect and filter which data to store and send to the client. It also means that unlimited amount of data can be assigned to the map, so it doesn't necessarily mean that the file size of the map decreases just because the map is

stored in vector format. [32]

2.3 Dynamic rendering

In some web map applications the map becomes blurry for a moment when zooming in. This is a result from zooming in on existing images before images for the new zoom level has been delivered. The map shown in 2.1a is a result of this. Figure 2.1b shows a map of the same geographic area. This time with correct images for the specific zoom level. By rendering the map dynamically from vector data, these blurry maps never appear when changing zoom level. For a more in-depth description of vector data, web maps and zoom levels, please refer to chapter 3.

(a) A map where images haven't finished rendering appears blurry

(b) When the images of a map are rendered, the map appears sharp

Figure 2.1: Two maps of the same geographic area. Images retrieved from [27].

This thesis investigates the possibilities of rendering a map in real-time in a web browser. This means that on every change in the map, for example when the user pans or zooms, the whole map will be re-rendered. When doing this it is of utter importance to render the maps in a good frame rate. Thanks to recent development of open standards for graphics rendering on the web, where WebGL is the most adopted, this is possible. However, deliver WebGL content to a web browser does not come without complications. Even if most modern browsers support WebGL there are many other factors to take into account, such as hardware support, performance and differences in browsers implementation. Devices that support WebGL browsers range from low end hardware to high end computers. A challenge is to deliver solutions with acceptable frame rates to all these devices.

2.4 Evaluation of perceived performance

The definition of good perceived performance is highly subtle where one person can think that an application is performing good whereas another person thinks the same application is near to useless due to its poor performance. It's also highly dependent on the expectations and previous experiences. One can have the feeling that an application is performing good but after using another similar application that is performing way better, the first one is often rejected and seen as bad performing.

The common technique of measuring performance is counting how many frames that is rendered per second, mostly referred to as frame rate or FPS. This can give a good idea of how the application is performing but it's not without difficulties. An application can have a good frame rate overall, but small dips regularly or irregularly will make the application be perceived as unresponsive and bad performing. For example, in a map application, certain locations on the map can have higher density of geometries to render to the screen. This can lead to decreased frame rate when panning across that specific location, while the app performs good otherwise.

Also, as in our case when tiles are downloaded, processed and rendered on the fly, the time from when the application request the tile from the server until it's ready to render is of utter importance. If tiles are rendered fast, but it takes long time before it can even begin to render the tile onto the screen, the map application will be perceived as slow and unresponsive.

This thesis will concentrate on measuring performance in different use cases and using different implementations and data, and then provide possible solutions to the issues found. It is later up to IT-Bolaget Per & Per and their clients to decide whether the application performs good enough to be taken into further development.

Chapter 3

Theory

3.1 Vector Based Geographical Information

There are several methods and standards for describing geographical information as vector based geometries. Very often it's also desired to be able to store data that is related to a geometry. These metadata can later be used for any purposes. One common use-case is to be able to style different geometries according to their metadata properties. This chapter will give a brief introduction of the formats relevant for this thesis.

3.1.1 Map projection and tiles

In this thesis we're only working with two dimensional maps. Mapping the three dimensional earth surface onto a two dimensional surface can be done in many ways and there are many standards. A common technique, also used in Mapbox's applications, is using the Spherical Mercator projection. Normally, on earths surface, the length between two longitude or latitude lines differ depending on where you are. The Spherical Mercator projection maps out the spherical earth surface onto an cylinder which then can be rolled out to a two dimensional surface. This results in latitude and longitude lines that are parallel. The benefit of this is that geometry vertex locations will not be dependent on where they are [16]. Having this projection makes it possible to divide the map into squared tiles and use them in map applications.

3.1.2 Shapefile

A Shapefile is a file containing non-topological geographical information and attribute information. The geometries can be points, lines and area features and are stored as one or a set of vector coordinates [10].

3.1.3 GeoJSON

GeoJSON is a JSON syntax developed for describing geographical data. A GeoJSON object can be a geometry, a feature or a collection of features. A geometry could for example be a polygon or a line. A feature is an object with a geometry, but also has custom properties that can be any JSON compatible string, number or object [4]. For example a feature can describe a building where the geometry is a polygon and the properties contain information about the building itself. The GeoJSON can be used as map source data in Mapbox GL JS however we are using it for generating Mapnik vector tiles, as described in chapter 4.2.3.

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": { "type": "Point", "coordinates": [102.0, 0.5] },
      "properties": {
        "type": "building",
        "inhabitants": 3
      }
    },
    { "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0] ]
      }
    }
  ]
}
```

Figure 3.1: Example of GeoJSON FeatureCollection

3.2 Vector Tiles

The map is divided into square tiles according to Google's tile coordinate system [30], where each tile consists of the geometry data that is to be shown for that area. This occurs for all zoom levels that are present in the current application. The tiles used to make up a map can be served in an image format to Mapbox GL JS. Tiles can also be rendered from a file containing vector based data, Protocol buffers [20]. The vector data describes all the geometries, such as city names and height curves, as a corresponding image tile would do. Vector based data is flexible and can be styled for different use cases before it is rendered as a tile. The same data can be used to render tiles with different resolutions for different zoom levels without

looking blurred or pixelated. Map data can be provided for all or some zoom levels. Each data file corresponds to a geographic area on a specific zoom level. On the zoom levels where data is provided, a file contains all the information needed to render a single tile.

The information in the files are organized into layers and features according to the Mapbox Vector Tile specification [19]. Layers corresponds to certain objects on the map. Each tile can have multiple layers, for example a building layer and a road layer. Every layer contains one or multiple features with geometrical information that describes that specific layer. For example a feature in the building layer can contain a geometry of the type polygon as well as coordinates to where the building is located.

Every feature is described as one of the three types:

- Linestrings. Streets, roads, railroads etc. are rendered as lines. Each linestring contains coordinates of start and end points of each segment of the linestring.
- Polygons. Buildings, parks, lakes etc are rendered as polygons. Each polygon contains coordinates of start and end points of each segment of the closed path that is a polygon.
- Points. Text and symbols. Each point contains coordinates as well as information of what characters are to be rendered.

These vector data files only describes the geometrical properties of features. In order to render the final map, the style files described in chapter 3.4, needs to be included.

3.3 Rendering of Tile Based Web Maps

There are generally speaking two ways to render vector based maps, server or client rendered. Server rendering means that the map information is interpreted by an server application, converted into bitmap images and then sent to the client for display. Each image represents a specific area on the map. It is also specific to the current level of zooming that the user has chosen to display the map. If the user zooms or in any way changes the information to be shown on the map, new images needs to be generated on the server and sent to the client.

Client rendering is when the map data is sent in vector format to the client and lets it handle the rendering and presentation to the user. It means that the data being sent can be picked and optimized depending on situation, which in many cases leads to less bandwidth and storage space on the client being used.

There are advantages and disadvantages with both techniques. However as the possibilities of doing advanced graphics rendering in a web browser continues to

grow, rendering the maps on the client can become the better alternative in more and more applications.

3.3.1 Simplification of Geometries

Line and polygon geometries in a map consists of multiple line segments. Line segments are joined together by points. The amount of detail needed for geometries may vary from map to map. When zooming out, the size of geometries decreases, meaning some geometries can be less detailed without affecting the perceived visual appearance. If that is the case, the geometry can be simplified, resulting in a geometry with less points.

A widely used curve simplification algorithm is the Douglas-Peucker algorithm [8], which is also used in this thesis. Given a curve made up by line segments and a distance threshold ϵ, the algorithm finds a similar curve with fewer points. The algorithm attempts to approximate a sequence of points by a line segment between the first and last point of the sequence. This line segment is illustrated as line *a* in step 1 of Figure 3.2. The point farthest from the line is found, point *c*. If the distance, *b*, to the point is below the threshold ϵ, the approximation is accepted. This means that the point can be discarded and the original curve is replaced with the the new line. If the distance is greater than the threshold, the algorithm is recursively applied to the two sub-sequences before and after the chosen point [12], in this case point *c*. New lines are drawn from the first to the last point in each sub-sequence and a new point that is farthest away from the line is found. The distance between the line and the point is compared with the threshold and the point is either kept or discarded. This is repeated until all points have been examined and a new, approximated, line is found. A result from the algorithm is shown in step 4, where the grey points are the discarded ones.

Great-Circle Distance

The unit of the threshold ϵ is the same as unit of the distances between points. For a Cartesian coordinate system a point is defined by x and y coordinates, making it easy to calculate distances. In the case with maps, the coordinates of the points are given in longitude and latitude degrees. However, the great-circle distance between two points on earth's surface can be calculated using *haversine's* formula described in 3.1. The great circle distance is the shortest distance between two points on the surface of a sphere [31].

$$a = sin^2(\Delta\varphi/2) + cos\varphi_1 \cdot cos\varphi_2 \cdot sin^2(\Delta\lambda/2) \tag{3.1}$$

Where φ is latitude and λ is longitude. The result a is the square of half the chord length between the points (φ_1, λ_1) and (φ_2, λ_2) [5]. a is then used to calculate the

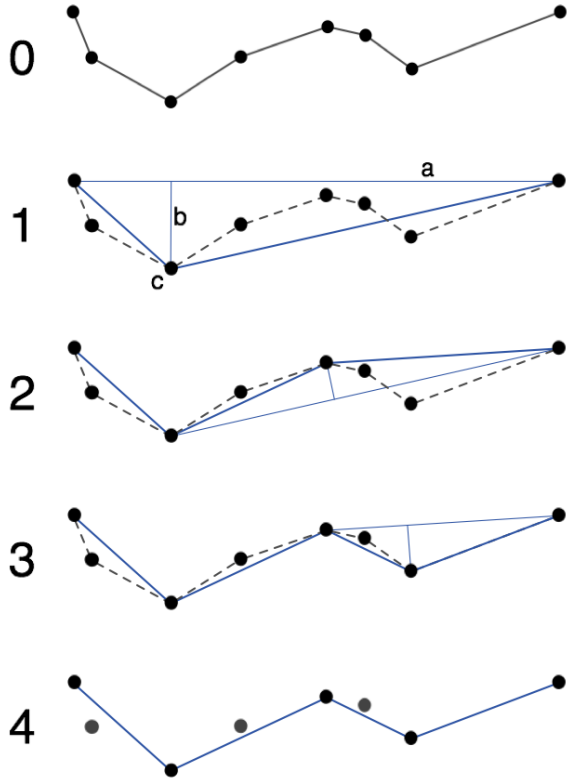

Figure 3.2: Curve being simplified with Douglas-Peucker algorithm

angular distance:

$$c = 2 \cdot atan2(\sqrt{a}, \sqrt{(1-a)}) \tag{3.2}$$

The distance in meters are then given by equation 3.3. Where $R = 6.371 \cdot 10^3$ is the radius of the earth in meters.

$$d = R \cdot c \tag{3.3}$$

With the distance given in meters it is easier to relate to real life distances, making it easier to choose an acceptable value for the simplification threshold ϵ. The latitude degree of the coordinates affect the conversion. Meaning that the result in meters vary depending on where on earth the points are, even if the distance in degrees between the points are the same.

3.4 Map Styling

How the final result of a map should look like on the screen is determined in the map styling process. This includes what will be visible on the map and how objects will be displayed. For example setting the size and color of buildings or deciding to hide them on low zoom levels. The styling of raster based maps needs to be done before the map is rendered, as the resulting raster image cannot be changed.

For vector based maps the style is separated from the map geometry. The style information and the information of the map geometry can be placed into different files. The style-files includes rules that the map renderer obeys. This makes the vector based maps very flexible. By changing the rules in the style-files, the resulting map can look completely different. This is useful since different use cases might have certain needs which requires the map to be styled in a specific way. In Figure 3.3 the same map geometry has been styled in two different ways. The style rules are applied at rendering time, meaning that the style rule to choose can depend on the current visible region. This makes it possible to style the same map feature differently at run-time depending on zoom level and other properties. Styling vector based maps like this can be done regardless if the rendering process is done on the server or client.

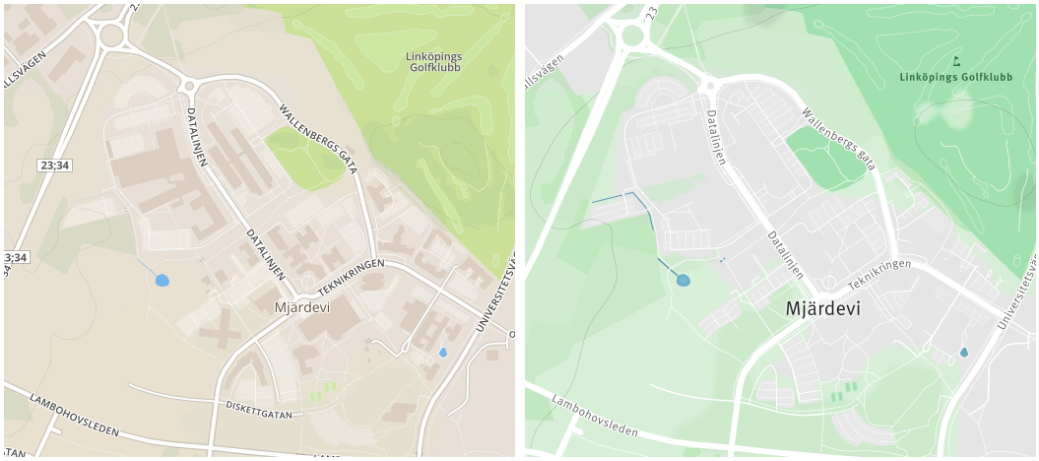

Figure 3.3: Two maps of the same map geometry styled with different stylesheets.

There are several languages for describing the style of a map, the three relevant languages in this thesis are CartoCSS, Mapnik XML and Mapbox GL JSON.

3.4.1 CartoCSS

CartoCSS is a styling language developed by Mapbox and is used in their map editing applications Tilemill and Mapbox Studio. It has similar syntax as regular CSS and is designed to be easy to learn for web developers that are already familiar with it. It is engineered to target the Mapnik renderer and be compiled to Mapnik XML. [6]

The below example is a basic example of one geometry that is being styled using the CartoCSS language.

```
#building {
    ::school {
        polygon-fill: #f4eff9;
        line-color: #ff0000;
        line-width: 0.1;
    }
}
```

3.4.2 Mapnik XML

A Mapnik XML stylesheet consists of a set of layers. These refer to the same layers defined in the vector tile, which are often a type of geometry, for example buildings. Each layer consists of styles, which are specific to different types of geometries in the same layer. For the building example there could for example be one style for schools and another style for industries. Further on, one style consists of a set or rules that is used to, for example, filter on zoom level or other settings. Each rule can contain three types of properties:

- Filter. All rules of a style are examined to check if it has a filter specified and if it matches the objects currently being rendered. The filter compares a feature's attributes against specified rules.
- Scale denominators. Specifies on which zoom levels a rule is activated. There are min and max scale denominators in order to specify maximum and minimum zoom level for a rule.
- Symbolizer. Specifies how a geometry is being rendered.

For more detail on Mapnik's style rules, please refer to its XML configuration reference [21].

Figure 3.4 illustrates a map style in Mapnik XML format.

```
<Layer name="building">
  <StyleName>school</StyleName>
</Layer>

<Style name="school">
  <Rule>
    <PolygonSymbolizer fill="#f2eff9"/>
    <LineSymbolizer stroke="#ff0000" stroke-width="0.1"/>
  </Rule>
</Style>
```

Figure 3.4: Mapnik XML

3.4.3 Mapbox GL JSON

Mapbox GL JSON is an pure JSON structure used for styling maps in Mapbox GL JS. Since both Mapnik XML and Mapbox GL JSON is meant to style Mapnik vector tiles, they share many characteristics. Figure 3.5 describes a map styling in Mapbox GL JSON format. It corresponds to the same map style as the Mapnik XML shown in Figure 3.4. The *type* property describes what type of geometry that is being rendered for that specific layer. Below are the three layer types that is being focused in this thesis:

- Fill. Buildings, parks, lakes and other polygon objects are of the type fill. These are areas that can be filled with solid color or a pattern. It is also possible to give fill objects an outline.
- Line. Streets, roads, borders etc are of the type line. However, due to the limitations of OpenGL's *GL_Lines* as well as the fact that OpenGL's antialiasing is not reliably present for all devices, Mapbox GL render lines as polygons. [9].
- Symbol. The type used when rendering texts and icons, such as city names or points of interest.

In addition to the type property, layers have two other properties that determine how data from that layer is rendered; *layout* and *paint*.

Layout specifies for example where on the map geometries are placed and when they are visible or not. The value of a layout property is calculated and interpolated once per tile, at each integer zoom level [7]. Layouts are applied early in the rendering process. They define how data is passed to the GPU for the current layer.

By sharing layout properties between layers, the overall processing time can be decreased. This also allows multiple layers to share GPU memory and other layout associated resources. [18].

Paint properties are used change the visual appearance of the map. For example line widths, colors and text sizes. The value of a paint property is calculated and interpolated at every frame when interacting with the map [7]. The value of a paint property can be a *stops* object, which contain multiple pairs of zoom level and value. It describes how that specific paint property is being rendered at different zoom levels. For example the width of a line can change dynamically when zooming. Paints are applied at a later stage in the rendering process. Layers that shares layout properties can still have independent paint properties. Layers can have class-specific paint properties, making it possible to change the appearance of the map in an easy way [18].

Layers may contain the optional min-zoom and max-zoom properties. They describe between which zoom levels the layer should be rendered at. If the corresponding data layer includes data at unnecessary zoom levels for the current style, min- and max-zoom should be used. This is a way to minimize the number of layers being processed during the loading of a tile.

```
"layers": [
  {
    "id": "school",
    "type": "fill",
    "source-layer": "building",
    "paint": {
      "fill-color": "#f2eff9"
    }
  },
  {
    "id": "school",
    "type": "line",
    "source-layer": "building",
    "paint": {
      "line-color": "#ff0000",
      "line-width": 0.1
    }
  }
]
```

Figure 3.5: Mapbox JSON

3.5 Maps in Mapbox GL JS

When interacting with a map rendered by Mapbox GL JS, the experience is like panning across a large, continuous image. As an image of the entire world at street level would be too large to download or hold in memory at once, the map consists of multiple small square images called tiles [13]. By placing tiles side-by-side an illusion of a large continuous image is created. It is possible to view different geographic areas in the same contiguous space by panning across the map. Instead of changing between maps with different detail and zoom level, the user can navigate with a single continuous system. By zooming in and out in this system, the amount of detail on the map varies. Higher zoom levels generally result in a more detailed map.

3.5.1 Pyramid coordinates

A tiles position in the world is described by a 3-dimensional coordinate system. Each tile has a z, x and y coordinate. The z coordinate describes the tiles zoom level while the x and y coordinates describes its position within a square grid for zoom level z. Zoom level 0 consists of only one tile that covers the whole globe. This tile has the lowest level of detail. When increasing the zoom level by one, each tile is divided into 4 new tiles [13].

3.5.2 Tiles in Mapbox GL JS

With the exponential relationship between z and number of tiles in the pyramid coordinate system, the amount of tiles needed to create the same map of a geographic area becomes large at higher zoom levels. By only serving the tiles needed to show a map of chosen geographic location the amount of tiles becomes manageable when it comes to storage and bandwidth. Once a tile is downloaded, the browser can store it in its cache and use it again when the same tile is needed. By loading the tiles progressively, starting at the center of the screen, the user can pan or zoom to a specific location even if the tiles at the edges of the screen haven't been loaded. [13]

3.5.3 Protocol buffers

Protocol buffers are a way of serializing structured data, and is maintained by Google. It is language and platform neutral, which means that protocol buffers can be used on any platform and in any language. However to serialize and deserialize data there needs to be specific implementation for that environment, but the data being sent is consistent. One of its main characteristics is that the structure of the data has to be defined in a schema in beforehand, and both the sender and

receiver needs to have the schema. However it is possible to change the schema and data from the sender's point of view, later on when the receiver is parsing the incoming data, those attributes that it does not know of are skipped.

Protocol buffers are often compared to JSON or XML. Both JSON and XML are a human readable and verbose whereas protocol buffers are in non human-readable binary format and better optimized for machine reading. This results in the protocol buffer files being 3-10 times smaller and 20-100 times faster parsing. [11]

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
</person>
```

This small XML snippet is at least 69 bytes, if you remove whitespaces, and would take 5,000-10,000 nanoseconds to parse. The equivalent protocol buffer file with the same information would be around 28 bytes in size and take 100-200 nanoseconds to parse. [11].

The schema used to serialize and parse protocol buffer files are defined in .proto-files and are called messages. Each protocol buffer message contains a series of name-value pairs that describe the dataset.

```
message Person {
  required string name = 1;
  optional boolean alive = 2;
  repeated PhoneNumber phone = 3;

message PhoneNumber {
    required string number = 1;
  }
}
```

This is a minimal example of a message type Person, where each field has a name, a type, an indicator whether it's required, not required, repeated and an unique identifier. The identifier is used to identify fields when in binary format and should not be changed once the message type has been used. The message format supports hierarchical structure as seen in the example where the PhoneNumber message type is defined inside Person and one Person can have a list of PhoneNumber thanks to the "repeated" keyword.

Chapter 4

Method

4.1 Converting stylesheets

In order to use vector tiles in Mapbox GL JS, the corresponding source stylesheet has to be in the Mapbox GL JSON format. In their current solution they edit the stylesheet using CartoCSS which then is compiled into Mapnik XML. Keeping two separate source stylesheets using two different languages is not to be considered, therefore another solution has to be made up. We chose to develop a converter that simply takes a Mapnik XML stylesheet as input and outputs a stylesheet with the same rules but in the Mapbox GL JSON format. The Mapbox organization is providing lots of their software as open source through their Github account, therefore we chose to make our converter open source as well. We wrote the converter in Javascript using Node.js.

The stylesheet converter implemented in this thesis consists of 3 steps: A preprocess step, an interpretation step and a postprocess step.

4.1.1 Preprocess

The converter takes a Mapnik XML-file as input. As described in 3.4.2, the XML-file consists of layers and styles, both of which are structured with XML-tags. The first step is to translate the file to JSON syntax. This was done using the node package *xml2json* [26]. The result is a JSON object containing layer and style keys. The corresponding attributes and elements from the XML will be the values of the layer and style keys. Each style is then merged into its corresponding layer. Each Style contain a set of rules that needs to be translated to Mapbox GL JSON.

4.1.2 Interpretation

The result from the preprocessing step will be a JSON file consisting of layers. Each layer has its *style* object and each style has one or multiple *rules*. These rules are in Mapnik XML format and needs to be interpreted and translated to Mapbox GL JSON format. Rules describe when its corresponding layer should be visible on the map. They also describe the visual appearance of the geometries of the layers. As described in 3.4.2, a Mapnik XML rule may include *scale denominators*, *filters* and *symbolizers*. Scale denominators are translated to zoom levels according to OpenStreetMap's scale denominators [23]. These values are used to set the *min-zoom* and *max-zoom* properties of a layer. The different types of symbolizers from Mapnik XML is converted to Mapbox JSON's *fill*, *line* or *symbol* properties according to table 4.1. There are other types of symbolizers than the ones in the table. However, those are not yet implemented in the converter. For the most of the parameters in a Mapnik symbolizer, there is a corresponding Mapbox GL style property. For example a LineSymbolizer's *stroke-width* in Mapnik is translated to *line-width* for a layer of the type line in Mapbox. If a layer is of the type fill or line, its style properties are translated and placed in a paint object. When translated, layers of the type symbol will have a layout object as well as a paint object.

Table 4.1: Table on how the most common Mapnik XML rules are translated to Mapbox JSON.

Mapnik XML	Mapbox JSON
TextSymbolizer	symbol
LineSymbolizer	line
PolygonSymbolizer	fill
PolygonPatternSymbolizer	fill

A style in Mapnik XML can have multiple rules with varying style properties per zoom level. For example different stroke-widths depending on the scale denominators. When this is the case, the values of the rule properties are translated and placed in a Mapbox' *stops* object.

Filters exists in both Mapnik XML and Mapbox GL JSON and are similar. The structure of filters were changed to suit the syntax of Mapbox GL JSON. All rule elements of Mapnik XML and their properties can be found in Mapnik's XML configuration reference [22]. For more details on Mapbox GL JSON style properties, please refer to Mapbox GL Style reference [18].

When the translation is done every rule from the Mapnik XML will correspond to a layer in the Mapbox GL JSON. Each layer will have a type and paint property as well as an optional layout property. All layers are given a *source-layer* property

with a value retrieved from the name attribute in the Mapnik XML. This describes which layer from the vector tile the style layer corresponds to.

A Mapbox GL JSON is required to contain a source object with an URL to the data. A version property is also required. If the map has text and icons, URLs to the fonts and image-sprites needs to be included. These properties cannot be retrieved from the Mapnik XML file. Therefore, a setting file with information of these properties were made for each style that were to be converted.

4.1.3 Postprocess

After the Mapnik XML has been converted to Mapbox GL JSON, a postprocess step is executed. During this step, all layers are examined in order to check if multiple layers has the same source-layers and filters. If this is the case, the layers are merged into one layer. When two layers are merged, many of their properties are identical. However, properties such as min-zoom, max-zoom and stops may differ. The values of min-zoom and max-zoom are set to the minimum respectively maximum value from the layers that are being merged. The stops are merged and sorted by zoom levels. All redundant values are removed.

The result from the postprocess is a more compact stylesheet with fewer layers.

4.2 Generating Mapbox vector tiles

To be able to have full control of our map data we had to have a way of generating vector tiles from source data. We chose to use a dataset from Lantmäteriet called Fastighetskartan. [14] It has a fairly high amount of detail, which would serve good for testing and optimizing performance. However this map is provided in Shapefile format, which has to be processed and converted into vector tiles before it can be used in Mapbox GL JS. We needed a tool to quickly be able to generate vector tiles from the given source data with the option to choose a specific geographical area and have full control of the data and be able to change it according to our needs. Therefore we decided to develop an own tool for this purpose. It was made in Javascript with Node JS.

4.2.1 Using a PostGIS database

To process the provided shapefiles from Lantmäteriet, we chose to use a PostGIS database. Importing map data in that format is a straight forward process, and PostGIS' interface makes it possible to query only the information you want. The shapefiles was imported to the PostGIS database using the GUI utility pgShapeloader, which converts the files into SQL queries and pushes them in to the database using

the command line utility shp2pgsql [15]. Worth noting is that the shapefiles are divided into layers. There is one shapefile for each layer. This will result in the PostGIS having multiple tables, one per layer.

4.2.2 Knowing what to query

We wanted to be able to choose a specific geographical area and generate tiles only to cover that. To do that we used a tool available on Mapbox' Github repository called tile-cover [29]. It takes a GeoJSON geometry as argument and returns the coordinates, in xyz-format, of the tiles needed to cover that area. By doing that for each zoom level we get a complete list of tiles that we need to query the PostGIS database for.

Figure 4.1: Covering an area with a polygon geometry

However only having the xyz-coordinates won't suffice, we need to have the longitude and latitude bounds for each tile. This is because PostGIS does not have the ability to know For that we're using an utility called *node-sphericalmercator*, also available in Mapbox' Github repository [25]. Knowing the xyz-coordinates of a tile, it has the ability to return the west/east longitude bounds and the north/south latitude bounds. When those bounds has been retrieved, we can use it to query the PostGIS database.

4.2.3 Querying from PostGIS

Retrieving data from a PostGIS database can be done in several ways and its output can be in multiple different formats, however we chose to go with GeoJSON since it's widely used within Mapbox' utilities and tools. Using PostGIS' spatial SQL syntax we were able use query functions as ST_AsGeoJSON and ST_MakeEnvelope to get the result in GeoJSON.

```
SELECT st_asgeojson(geom), ANDAMAL_1 FROM by_0980 WHERE
    by_0980.geom && ST_MakeEnvelope(18.369140625,
    57.68066002977235, 18.4130859375, 57.70414723434193)
```

The above code snippet describes how to query one geometry type from one layer in one tile. The SELECT keyword *ANDAMAL_1* in this example describes which attribute of the layer *by_0980* that should be queried. For example a layer on the map could consist of buildings, then the attribute of such building could describe what type of building it is. The *ST_MakeEnvelope* function makes the PostGIS query only select geometries that are within these bounds. If a geometry crosses the boundings, PostGIS will automatically clip those geometries and make them fit within the bounding box. A query like this then has to be run for each desired layer. The result from each query is then saved as a GeoJSON collection according to the GeoJSON specificiation [4].

4.2.4 Saving the tile to file

To use the tiles in Mapbox GL JS they are saved as Mapnik compatible Protocol Buffers as explained in the Mapbox vector tiles overview [4]. To accomplish this, the *node-mapnik* utility available on the Mapnik repository [24], was used. This makes it possible to convert the GeoJSON collections into Mapnik vector tile compatible Protocol Buffers.

4.2.5 Simplification of geometries

Different map sources provide different detail levels of geometries. For example a map made for in-car navigation may not need to have that high amount of detail, it does not matter that much if the geometries are somewhat alike the real world objects. As long as it doesn't prevent the user from using the map for its intended use it's fine. However when for example having a map that is used for measuring property bounds or structure details of buildings the details of geometries will matter on a higher degree. In this thesis we used a map source with a very high detail, Fastighetskartan from Lantmäteriet [14]. According to Lantmäteriet this map is suitable for viewing on a scale range of 1:5000 - 1:20000 which translates to a zoom level between 15 and 17 in our map application. [3]. Having a optimized map source for each zoom level is to prefer for an optimal user experience and actual map usage, however having to deal with multiple sources would take time for us and not necessarily be of any use when testing performance. We therefore chose to use the same map source across multiple zoom levels. Doing that is however not entirely problem-free. When zooming out the map there are more geometries that fit on the screen, which means there are more data for the system to handle when

rendering. If the system doesn't have performance enough to handle this it will turn into an unresponsive and not that good user experience. Minimizing the amount of detail, the amount of vertices in map geometries, could lead to better performance. To try this hypothesis we used a simplification function which is provided in the PostGIS toolset, *st_simplify*. It makes it possible to simplify any geometry when querying the database, with a specified treshold value. The query would look the same as it does in section 4.2.3 but with the addition of running the *st_simplify* funtion on the geometry before returning it as GeoJSON. The second argument in *st_simplify* specifies how forgiving the simplification algorithm should be, how this value is affecting the geometries is described in the theory section 3.3.1.

```
SELECT st_asgeojson(st_simplify(geom, 0.00001)), ANDAMAL_1
    FROM by_0980....
```

In this case the value 0.00001 is given in latitude and longitude degrees which corresponds to approximately 1.11 meters in latitude direction and 0.60 meters in longitude direction at a latitude position of 57.6 degrees. We also simplified geometries with a degree of 0.00005, which corresponds to 5.57 and 2.99 meters in latitude respectively longitude directions.

4.2.6 On the fly or pregeneration of tiles

The PostGIS database is divided into tables, where each of them represent a layer from the map source. As we're constructing the map data set tile by tile we need to query the database once for each layer, for all tiles. We developed our tile generation utility to have the ability to both pregenerate tiles and save to disk, as well as generating them on the fly as a web server. When we're querying the PostGIS database for a tile it is beneficial to do do all the queries asynchronously, especially if generating on the fly and the client browser is waiting for a response. To solve this we used a node module called *async* [2]. It makes it possible to run multiple scripts asynchronously, in our case database queries, and continue to run the rest of the program only all tasks have completed. After all queries are done we have all the layers for that tile as GeoJSON and can then generate a Mapbox vector tile from them. To either save to disk or return to the requesting web client.

4.3 Perceived performance

Perceived performance was measured in frame rate, which is a common measurement in graphical applications. We also wanted to mimic the use pattern of a regular user but still have tests that are repeatable and as consistent as possible. This would give a result that is close to reality with high reliability and validity. For that Mapbox GL

JS' build in functions was used for panning across the map, if configured correctly they can mimic how a user would interact with the map.

Multiple test cases were created where the camera follows a predefined path over the map. The path is solely panning across the map on the same zoom level, but it crosses geographical areas that has different amount of geometries and detail. Each test case follows the same path and crosses the exact same tiles every time and the duration of the run is also the same for each test case. However different map data sets were used to see the difference in data. Using the tile generator developed we could generate different versions of the map data set with different simplifications and constructions, which then were used in the tests. The tests were run mainly with Fastighetskartan as the map data set, as vector tiles generated by our tile generator. However OpenStreetMap map data set, as provided by Mapbox, were used as well to get results to compare with.

During the time the path runs, at a predefined interval, the current frame rate is recorded and saved. The number of tiles and how many points that are in that specific frame and time is also saved. It is also known which types of points that are present at each saving, such as if it belongs to a line, a polygon or is just a point. With this information it's possible to know what geometries that was displayed on the screen at a specific time during the test case. Which means it can be used together with the recorded frame rate data for later analysis.

Chapter 5

Result

5.1 Generating tiles

We built an utility for generating vector tiles according to the Mapbox Vector Tile specification, as described in section 4.2. Its generated vector tiles can be either saved to disk or served to a web client where the tile generator acts as a web server. Both uses the same generation procedure, only the last step when the tile is done is different.

A test was carried out to measure how long it takes to generate vector tiles for a certain area. Gotland was chosen in this case, which has an area of 3183.7 km^2, which corresponds to 215 tiles on zoom level 12 and 776 tiles on zoom level 13. The test was run with two different simplifications for each zoom level to show the difference in time. The simplification thresholds is specified in latitude/longitude degrees and in this case 0.00001° corresponds to somewhere around 1 meter. How this is calculated can be read about in the theory section 3.3.1.

Simplifying the geometries in a map reduces their amount of points. The average amount of points per tile can be seen in tables 5.1 and 5.2. Reducing the amount of points also reduces the amount of visual detail, which can be seen in figure 5.1. Simplifying with a threshold value of 0.00001° makes no or very little visual difference but an simplification of 0.00005° skews the geometries.

Table 5.1: Generating tiles for Gotland on zoom level 12, 215 tiles

Simplification (degrees)	Total time (minutes)	Time per tile (seconds)	Avg. points per tile
None	6	1.7	147909
0.00001	4.1	1.15	96693 (35% less)
0.00005	2.5	0.7	45492 (69% less)

Table 5.2: Generating tiles for Gotland on zoom level 13, 776 tiles

Simplification (degrees)	Total time (minutes)	Time per tile (seconds)	Avg. points per tile
None	17	1.3	121257
0.00001	13.5	1	75920 (37% less)
0.00005	5.56	0.43	33604 (72% less)

(a) No simplification (b) 0.00001° simplification (c) 0.00005° simplification

Figure 5.1: Three 100x100 meter areas with different simplification. Copyright Lantmäteriet I2014/00578.

5.2 Benchmarking Setup

All test cases were performed the same amount of times and an average of each test case was carried out in order to avoid deviations and give a more accurate result. The tests were executed on different devices in order to investigate how hardware affects the performance. This way it was possible to come to conclusions on what type of hardware that would serve for rendering the data heavy vector maps. Table 5.3 includes all devices and their hardware that was used for performance testing. The same path was used for all tests. The map was displayed with the same screen resolution for all test devices.

5.3 Perceived Performance

The graphs in Figure 5.2 and 5.3 visualizes the result from logging frame rate and number of points for the same path with different map data sets. The map data set used in Figure 5.2 is Fastighetskartan by Lantmäteriet without any simplification. In Figure 5.3 Fastighetskartan by Lantmäteriet with all geometries simplified with 0.00005 °. Both tests were performed on a MacBook Pro early 2011 with dedicated

Table 5.3: Devices and their hardware that was used for testing performance.

Device	GPU	CPU	RAM
MacBook Pro early 2011 dedicated GPU	AMD Radeon HD 6750M 1024 MB	2.2 GHz Intel Core i7	8GB 1333MHz DDR3
MacBook Pro early 2011 integrated GPU	Intel HD Graphics 3000 512 MB	2.2 GHz Intel Core i7	8GB 1333MHz DDR3
MacBook Pro late 2013 dedicated GPU	Nvidia Geforce GT 750M 2048MB	2.3 GHz Intel Core i7 (Turbo Boost up to 3.5GHz)	16GB 1600MHz DDR3
Mac Pro Late 2013	AMD FirePro D500 3072 MB	3.5 GHz 6-Core Intel Xeon ES (Turbo Boost up to 3.9GHz)	32 GB 1867 MHz DDR3

GPU.

The same test was performed on a MacBook Pro late 2013 with dedicated GPU. The result from this path is shown in Figure 5.4.

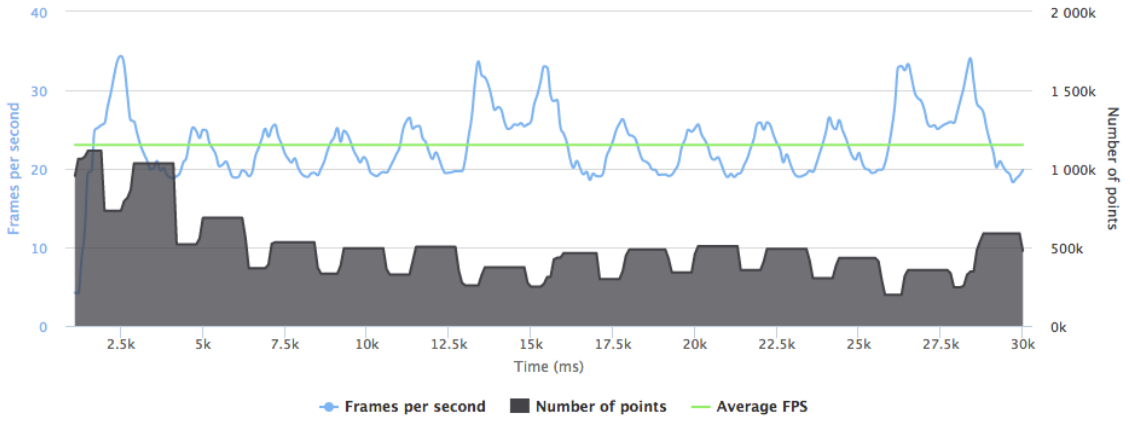

Figure 5.2: Fps and number of points vs time from running a predefined path over Gotland. Fastighetskartan map data set, not simplified. The test was performed on a MacBook Pro early 2011 with dedicated GPU.

The average frame rate for the entire path with different map data is shown in Figure 5.5. In addition to the two Fastighetskartan map data sets, the chart shows the average frame rate of two OpenStreetMap map data sets. The map data set denoted as *OSM 1* has a lower detail level than *Fastighetskartan* and *OSM 2* has

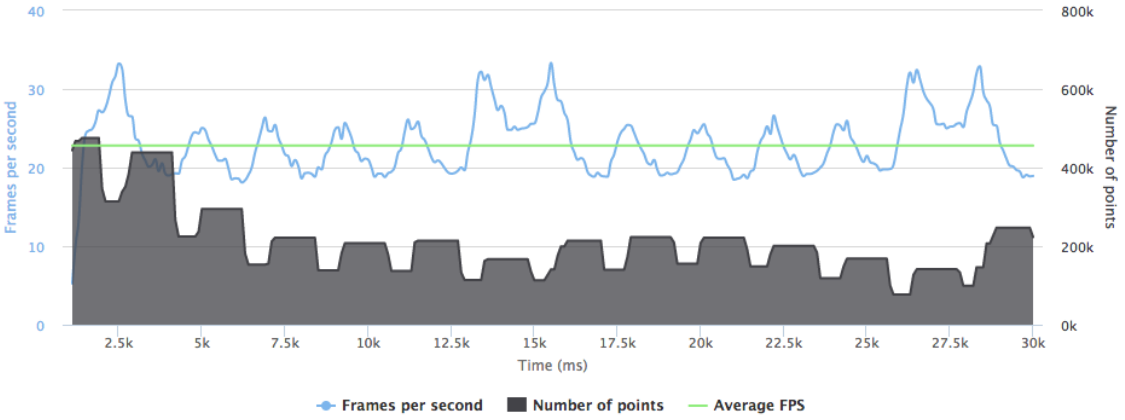

Figure 5.3: Frame rate and number of points vs time from running a predefined path over Gotland. Fastighetskartan map data set, simplified with 0.00005°. The test was performed on a MacBook Pro early 2011 with dedicated GPU.

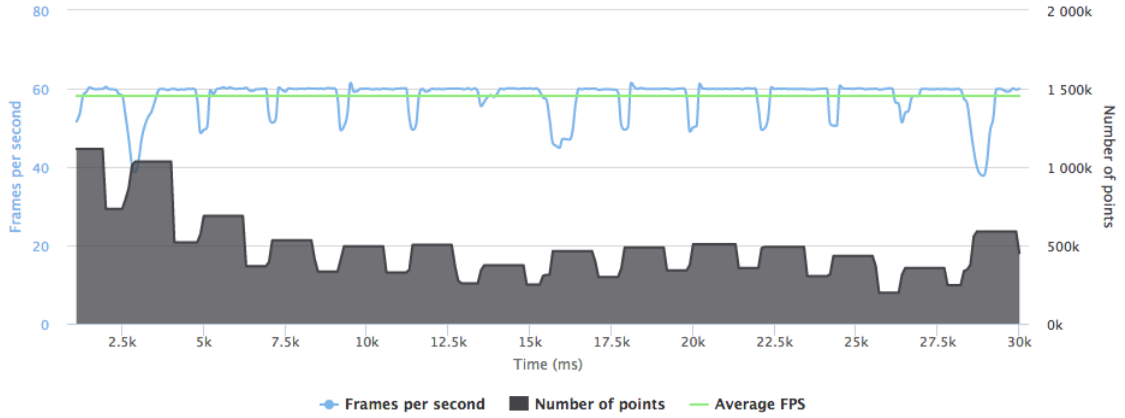

Figure 5.4: Frame rate and number of points vs time from running a predefined path over Gotland. Fastighetskartan map data set, not simplified. The test was performed on a MacBook Pro late 2013 with dedicated GPU.

lot less details. Figure 5.6 is the result from the same tests on a MacBook Pro early 2011 with integrated GPU. The result of the tests on a MacBook Pro late 2013 is shown in 5.7.

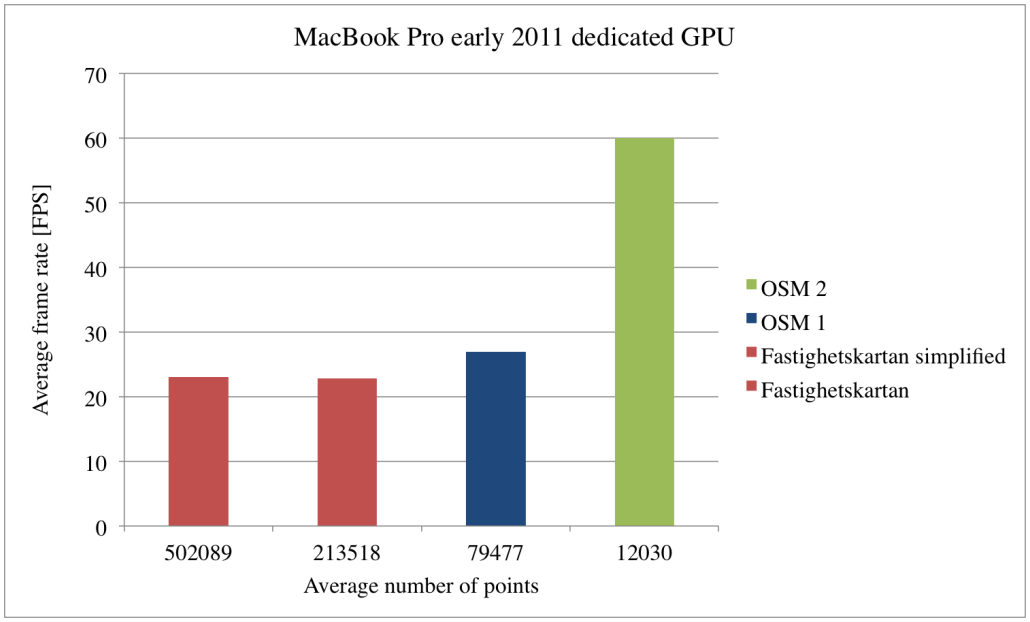

Figure 5.5: Macbook Pro early 2011 with **dedicated** GPU. Average frame rate of the same path with four map data sets with varying number of points.

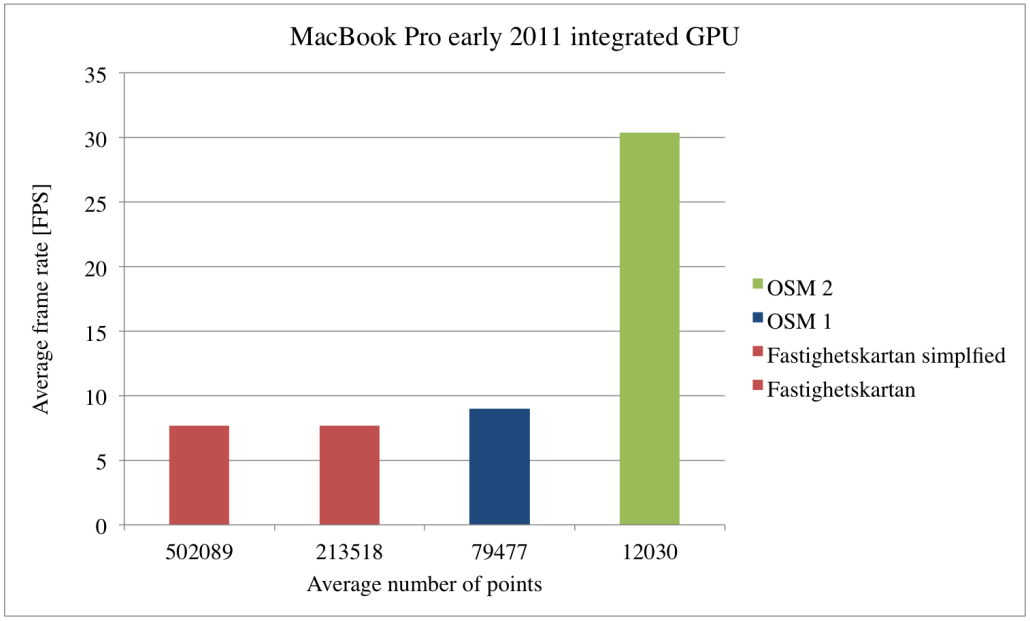

Figure 5.6: Macbook Pro early 2011 with **integrated** GPU. Average frame rate of the same path with four map data sets with varying number of points.

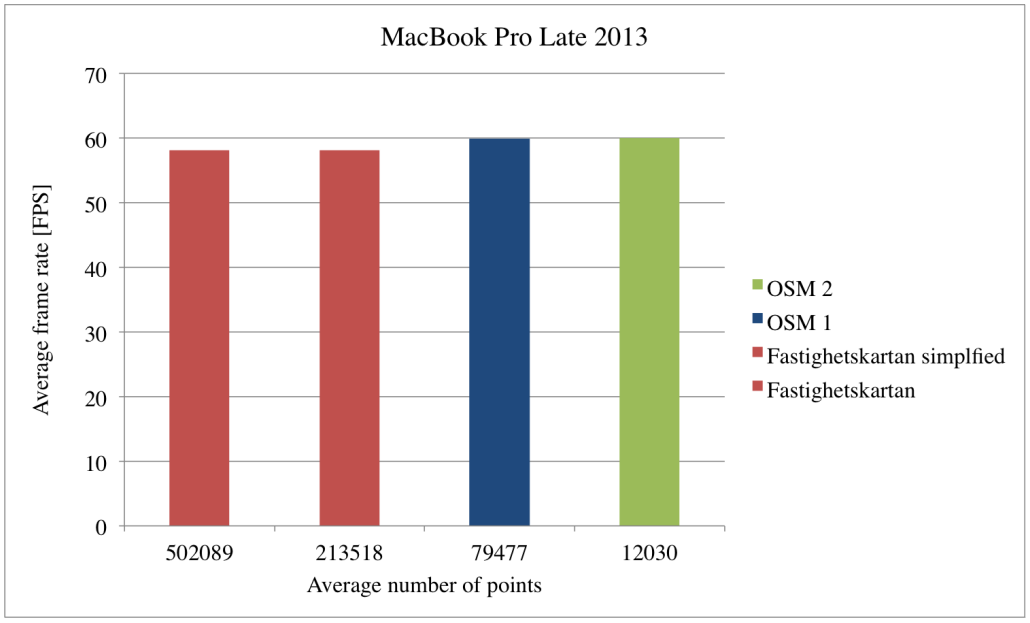

Figure 5.7: Macbook Pro late 2013 with **dedicated** GPU. Average frame rate of the same path with four map data sets with varying number of points.

Chapter 6

Discussion

6.1 Generating tiles

A requirement to start testing performance in Mapbox GL JS was to have a relevant data set. Mapbox provides extensive support for OpenStreetMap with a lot of styles to test with, however OpenStreetMap is neither detailed nor data-heavy. The scope for this thesis was to see how Mapbox GL JS performs with data-heavy and very detailed map data sets, where OpenStreetMap wasn't the best fit. So to do this we chose to use Fastighetskartan by Lantmäteriet. This data set is very detailed and would be a good fit to test the capabilities Mapbox GL JS has, performance wise.

The tile generator that was developed turned out to be a good tool for doing rapid testing with different layer combinations and simplifications. It has the ability to only generate tiles for a specific area, which comes handy since performance testing can be done as good on small areas as big geographical areas. The important factor is the amount of detail on the area that tiles are generated for, which differs a lot between cities and less inhabited areas. However generating the tiles is not that fast. As seen in the result tables 5.1 and 5.2 it can take up to two seconds for one tile to be generated. This might be an acceptable time if generating tiles and saving them for later usage, but serving tiles on the fly requires an fast generation time. Having the user wait for ten seconds or more for the tiles for an area to load is not acceptable. If using the tile generator as a web server and serving the tiles on the fly, further optimizations has to be made. However developing an utility like this does not directly lead towards the goal of this thesis to investigate Mapbox GL JS performance. It can rather be seen as a must have tool to be able to reach the goal.

6.2 Stylesheet converter

The data set used, Fastighetskartan by Lantmäteriet, is very detailed and would be a good choice for testing performance, however this means we would have to provide our own stylesheet. For this we had two options, make a stylesheet from scratch or develop a converter to re-use the existing XML stylesheets, where we decided to go with the latter option. The stylesheet converter in chapter 4.1 demonstrates that it is possible to convert Mapnik XML to Mapbox GL JSON, that can be used to style a Mapbox GL JS map made up of vector tiles. However the stylesheets converted during this thesis does not include all Mapnik XML style properties. The properties not used in our original XML stylesheets are not implemented in the converter, making it non-general. To make it general and support all Mapnik XML properties would require a great workload and it's not even possible to get every detail right since all Mapnik XML properties doesn't have a Mapbox GL JSON counterpart. The same applies vice versa, all properties and structural possibilities implemented in the Mapbox GL JSON specification isn't possible to do in Mapnik XML.

The amount of time it takes to create stylesheet from scratch for the map data set used in this thesis is very long. So despite the time it took to develop the converter, being able to convert a complete style saves significant amount of time, especially since the converter can be used for several different stylesheets. It is therefore a powerful tool when making prototypes such as the ones in this thesis.

6.3 Mapbox GL JS performance

6.3.1 Method

The performance of a vector map rendered in a browser with Mapbox GL JS depends on many factors. Mapbox GL JS is still in development and has not yet been officially released by the Mapbox team. Features are added daily to their Github repository. This means that the analysis and observations related to performance made in this thesis may be of lower significance in the future. Future changes of the Mapbox GL JS implementation and its surrounding standards and utilities may affect the circumstances under which this thesis were made, thus its result may become obsolete.

The focus of this thesis was to investigate how optimizing map data and its corresponding style rules affects the performance experienced by the user. However, this approach have both a positive and negative side. The map data and stylesheets are the things that differ from use case to use case, therefore it's of great value knowing how to optimize them for best performance. The map data set used mainly in this thesis, Fastighetskartan, is just one of many data sets available where all

have different structure, properties and detail levels. We have been in contact with developers on the Mapbox team, where they expressed the importance of using an optimized data set and stylesheet for optimal performance. This has partly been a motivation for us when choosing a scope for this thesis.

On the other hand it could have been beneficial to further investigate the Mapbox GL JS implementation to find design choices that could have been done in other ways to improve performance. During our contact with developers on Mapbox we also understood that Mapbox GL JS is intended to provide very high visual fidelity. This approach could result in a map application that performs bad on certain devices, which could be more devastating than having a map that does have as high visual qualities as possible. This is an area which could have been interesting to investigate further, such as different methods of lowering visual quality to gain better performance.

To measure performance in terms of frame rate we used an FPS meter method based on the *requestAnimationFrame* feature present in most modern browsers. A problem with this approach is the delay or offset the FPS meter has, since the last 16 frames are used to calculate a mean. The time each frame takes to render differ greatly between frames so to be able to get usable data from it, a mean has to be calculated.

An issue worth noticing is that Mapbox GL JS does use on-demand rendering, which means it only draws a new frame when it's needed. This means it only draws a new frame when for example the user is interacting with the map or new content is loaded into it. Many applications, like games and other WebGL applications, uses a method where it tries to render as fast as the web browser allows it to. This is also what the most information available about measuring WebGL performance suggest. Using an approach like in Mapbox GL JS to only render content when it changes may affect the FPS calculations in certain situations and result in faulty measurements.

This thesis focuses on measuring frame rate, but other attempts on measuring performance were made as well. We wanted to be able to measure how long it takes for the GPU to render certain types of geometry. For example we had a hypothesis that lines, such as roads and contours, takes longer to render than polygons. To test that hypothesis we would need to know how long an specific geometry takes to render on the GPU. However when making a draw call to the GPU using WebGL there are no way of knowing that. We are only able to see how long time a frame takes to render, which ultimately is very similar to measuring frame rate.

6.3.2 Result

From the graphs in figures 5.2 and 5.3 it is possible to see that the resulting frame rate follows the same pattern, despite that the total number of points has been

reduced with an average of 57% in the second graph. Both results in an average frame rate of 23 FPS. Since it is the same geographical area that is being rendered at the same time for both graphs, the relation of the number of points are very similar. In both graphs it is apparent that the frame rate decreases when the number of points increases rapidly. This is more evident in 5.4 where the frame rate is stable at 60 FPS and drops when peaks occurs in the number of points. These peaks are the result of tiles not being able to load fast enough when the screen is panning across the map. Some parts of the screen will therefore be white without content for some time, resulting in the drops in number of tiles. Our hypothesis from the beginning was that a reduced number of points in geometries would increase performance, which we discovered it does not necessarily do.

From the results in section 5.3 it is possible to see that hardware has a significant affect on the perceived performance. From figure 5.6 one can see that a MacBook Pro early 2011 with integrated GPU had an average frame rate below 10 during the test for all but the very simple OpenStreetMap data set. From this we can conclude that a dedicated GPU is necessity in order to achieve an acceptable perceived performance for the more detailed data sets.

Something worth noting is that the OSM2 data set resulted in a high average frame rate for all devices. The fact that the data set consists of a significant amount less points is not solely the reason for this result. With this data set large portions of the view port are not covered by geometries but instead of a background color. This means that geometries does not need to be rendered at a large part of the screen.

The early hypothesis we made was that an stylesheet optimization will affect performance. This turned out to be true, but more work is needed to give an complete answer to this. During early development of our Mapnik XML to Mapbox GL JSON converter we noticed that higher number of style rule layers in the Mapbox GL JSON stylesheet will affect performance negatively. Combining and reusing style rules and using the available options described on Mapbox's style specification [18] will result in better performance. However as we got our Mapbox GL stylesheets from our converter, rather than constructing them by hand, we do not have any benchmarks proving this. An important thing worth mentioning is that we used a Mapnik XML stylesheet where we made no effort in checking if the converted Mapbox GL JSON contains any unnecessary rules. The original stylesheet is very large and so is the converted Mapbox GL stylesheet, where it would be an demanding task to manually check if there were any abnormalities from Mapbox's style reference. It might be possible to make a Mapnik XML to Mapbox GL JSON converter that does this checking, but it would probably require quite an effort. It will, according to our observatiosn, probably always be better to construct a stylesheet by hand.

Chapter 7

Conclusion

This thesis investigates how Mapbox GL JS is performing when rendering data heavy maps. This was done by carrying out different tests and benchmarks on different devices. Its results were then analysed and discussed. Our problem description was defined as follows.

- With focus on perceived performance, investigate whether the framework Mapbox GL JS is suited to dynamically render data heavy vector based maps in a browser using the GPU.
- Analyze how the input vector data and its corresponding style rules affects perceived performance.
- Present possible solutions and theories how vector data and style rules can be optimized to gain better perceived performance.

We can see in our benchmarks that rendering data heavy maps in Mapbox GL JS sets certain requirements on the hardware, certainly the GPU. A Macbook Pro from 2011 is barely performing good enough for a pleasant user experience. This is of course highly subjective but we think it is not good enough. We have seen that reducing the amount of detail of the geometries in a data set will not affect performance. To make better conclusions on how the construction of the stylesheet affects performance, it would be beneficial to have a hand made stylesheet to compare with. However we can see that the OpenStreetMap data set together with Mapbox's provided stylesheet does not perform significantly better than Fastighetskartan together with our converted stylesheet.

Bibliography

[1] Gennady L. Andrienko and Natalia V. Andrienko. "Interactive maps for visual data exploration". In: *International Journal of Geographical Information Science* 13.4 (1999), pp. 355–374. URL: http://geoanalytics.net/and/papers/ijgis99.pdf.

[2] *Async.js*. Accessed: 2015-05-26. URL: https://github.com/caolan/async.

[3] Aileen Buckley. *How can you tell what map scales are shown for online maps?* Accessed: 2015-05-24. URL: http://blogs.esri.com/esri/arcgis/2009/03/19/how-can-you-tell-what-map-scales-are-shown-for-online-maps/.

[4] H. Butler et al. *The GeoJSON Format Specification*. Accessed: 2015-05-21. URL: http://geojson.org/geojson-spec.html.

[5] *Calculate distance, bearing and more between Latitude/Longitude points*. Accessed: 2015-05-25. URL: http://www.movable-type.co.uk/scripts/latlong.html.

[6] *CartoCSS on Github*. Accessed: 2015-03-18. URL: https://github.com/mapbox/carto.

[7] *Cartographic design in GL maps*. Accessed: 2015-03-18. URL: https://2015.foss4g-na.org/sites/default/files/slides/cartography-for-gl.pdf.

[8] D. Douglas and T. Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". In: *Canadian Cartographer* 10 (1973), pp. 112–122.

[9] *Drawing Antialiased Lines with OpenGL*. Accessed: 2015-03-12. URL: https://www.mapbox.com/blog/drawing-antialiased-lines/.

[10] *ESRI Shapefile Technical Description*. Accessed: 2015-05-21. 1998. URL: https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf.

[11] *Google Developers - Protocol Buffers*. Accessed: 2015-03-11. URL: https://developers.google.com/protocol-buffers/.

[12] P. Heckbert and M. Garland. *Survey of Polygonal Surface Simplification Algorithms*. URL: http://www.cs.cmu.edu/afs/cs/user/garland/www/Papers/simp.pdf.

[13] *How does web maps work?* Accessed: 2015-03-12. URL: https://www.mapbox.com/guides/how-web-maps-work/.

[14] Lantmäteriet. *GSD-Fastighetskartan, vektor*. Accessed: 2015-05-24. URL: http://www.lantmateriet.se/sv/Kartor-och-geografisk-information/Kartor/Fastighetskartan/GSD-Fastighetskartan-vektor-/.

[15] *Loading data into PostGIS with pgShapeloader*. Accessed: 2015-05-27. URL: http://suite.opengeo.org/4.1/dataadmin/pgGettingStarted/pgshapeloader.html.

[16] *Location and Maps Programming Guide: Displaying Maps*. Accessed: 2015-05-21. URL: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/MapKit/MapKit.html.

[17] *Mapbox GL JS*. https://www.mapbox.com/. Accessed: 2015-02-11.

[18] *Mapbox GL Style Reference*. Accessed: 2015-02-07. URL: https://www.mapbox.com/mapbox-gl-style-spec/.

[19] *Mapbox Vector Tile Specification*. Accessed: 2015-03-16. URL: https://github.com/mapbox/vector-tile-spec/tree/master/1.0.1.

[20] *Mapbox Vector Tiles*. Accessed: 2015-03-12. URL: https://www.mapbox.com/developers/vector-tiles/.

[21] *Mapnik XML Config Reference*. Accessed: 2015-05-18. URL: https://github.com/mapnik/mapnik/wiki/XMLConfigReference.

[22] *Mapnik XML Config Reference - rules*. Accessed: 2015-05-18. URL: https://github.com/mapnik/mapnik/wiki/XMLConfigReference#rule.

[23] *Min Scale Denominator*. Accessed: 2015-05-18. URL: http://wiki.openstreetmap.org/wiki/MinScaleDenominator.

[24] *node-napnik*. Accessed: 2015-05-27. URL: https://github.com/mapnik/node-mapnik.

[25] *node-sphericalmercator*. Accessed: 2015-05-27. URL: https://github.com/mapbox/node-sphericalmercator.

[26] *node-xml2json*. Accessed: 2015-02-12. URL: https://github.com/Leonidas-from-XIV/node-xml2js.

[27] *Open Street Maps*. Accessed: 2015-03-17. URL: http://www.openstreetmap.org/.

[28] Sharon Oviatt. "Multimodal Interactive Maps: Designing for Human Performance". In: *Hum.-Comput. Interact.* 12.1 (Mar. 1997), pp. 93–129. ISSN: 0737-0024. DOI: 10.1207/s15327051hci1201\&2_4. URL: http://dx.doi.org/10.1207/s15327051hci1201\&2_4.

[29] *tile-cover*. Accessed: 2015-05-27. URL: https://github.com/mapbox/tile-cover.

[30] *Tiles à la Google Maps: Coordinates, Tile Bounds and Projection*. Accessed: 2015-05-27. URL: http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/.

[31] *Wolfram MathWorld - Great Circle*. Accessed: 2015-05-25. URL: http://mathworld.wolfram.com/GreatCircle.html.

[32] Bisheng Yang. "A Multi-resolution Model of Vector Map Data for Rapid Transmission over the Internet". In: *Comput. Geosci.* 31.5 (June 2005), pp. 569–578. ISSN: 0098-3004. DOI: 10.1016/j.cageo.2004.11.011. URL: http://dx.doi.org/10.1016/j.cageo.2004.11.011.