GPU accelerated rendering of vector based maps on iOS

by

Jonas Bromö and Alexander Qvick Faxå

LIU-IDA/LITH-EX-A--14/023--SE

2014-05-30
Final thesis

GPU accelerated rendering of vector based maps on iOS

by

Jonas Bromö and Alexander Qvick Faxå

LIU-IDA/LITH-EX-A--14/023--SE

2014-05-30

Supervisor: Anders Fröberg (IDA), Thibault Durand (IT-Bolaget Per & Per AB)
Examiner: Erik Berglund
Abstract

Digital maps can be represented as either raster (bitmap images) or vector data. Vector maps are often preferable as they can be stored more efficiently and rendered irrespective of screen resolution. Vector map rendering on demand can be a computationally intensive task and has to be implemented in an efficient manner to ensure good performance and a satisfied end-user, especially on mobile devices with limited computational resources.

This thesis discusses different ways of utilizing the on-chip GPU to improve the vector map rendering performance of an existing iOS app. It describes an implementation that uses OpenGL ES 2.0 to achieve the same end-result as the old CPU-based implementation using the same underlying map infrastructure. By using the OpenGL based map renderer as well as implementing other performance optimizations, the authors were able to achieve an almost fivefold increase in rendering performance on an iPad Air.
AGG  Anti-Grain Geometry. Open source graphics library with a software renderer that supports Anti-Aliasing and Subpixel Accuracy. Mapnik uses this renderer in MapDemo.

app  A self-contained program or piece of software designed to fulfill a particular purpose, often referred to in the context of mobile devices. Short for application.

FreeType  A software font rasterization engine capable of rendering text as bitmap images.

glyph  In the context of typography, a graphical representation of a readable character. For example, the latin lower case letter ”a” is represented as different glyphs in the fonts Arial and Helvetica.

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

iOS  Operating system for mobile devices developed by Apple Inc. Currently used by the iPhone, iPad, iPod Touch and Apple TV.

Jailbreaking  In the context of iOS, the process of removing limitations in the operating system to enable access or functionality originally restricted by Apple.

MapDemo  iOS application developed by IT-Bolaget Per & Per that has the ability to render a custom vector based map using Mapnik. The rendering performance of MapDemo is used as a baseline in this thesis.

Mapnik  Free toolkit for developing mapping applications, see section 1.1.6

Map Kit  iOS framework for embedding maps and geo-functionality into iOS apps, see section 1.1.5
**Retina display** Marketing term used by Apple Inc. for describing LCD displays of their products that have a pixel density so high that a human cannot make out individual pixels from a normal viewing distance.

**texel** Also known as texture element, is the most basic unit of computer graphics textures. As a bitmap image consists of a grid of pixels, a texture consists of a grid of texels.

**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

1 Introduction  
1.1 Background ................................................. 1  
1.1.1 IT-Bolaget Per & Per .................................. 1  
1.1.2 Map rendering alternatives ......................... 2  
1.1.3 Map styling .............................................. 3  
1.1.4 Hardware constraints ................................. 4  
1.1.5 Apple Map Kit .......................................... 4  
1.1.6 Mapnik .................................................. 4  
1.2 Problem description ...................................... 5  
1.3 Goal .......................................................... 6  
1.4 Motivation .................................................. 6  
1.5 Approach .................................................... 6  
1.6 Scope and limitations .................................... 7  
1.7 Related work ............................................... 7  

2 Product analysis ................................................. 8  
2.1 Indexing the world by tiles .............................. 8  
2.2 Tile sizes in Map Kit ..................................... 9  
2.3 From vector data files to pixels .......................... 9  
2.4 Mapnik rendering .......................................... 11  
2.5 Rendering performance bottleneck ..................... 12  

3 Graphics rendering on iOS .................................... 13  
3.1 Possible graphics technologies .......................... 13  
3.2 Tile-Based Deferred Rendering ........................... 14  
3.3 Text rendering ............................................. 14  
3.4 OpenGL ES drawing primitives ......................... 15  
3.4.1 Triangles ............................................... 15  
3.5 Creating OpenGL compatible geometry ................ 16  
3.5.1 Lines .................................................. 17  
3.5.2 Polygons ............................................... 17  
3.5.3 Text .................................................. 20
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The same map styled in two different ways.</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>MapDemo with a custom map as a Map Kit tile overlay.</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Tiles on multiple zoom levels</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Simple and complex polygons</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Tile rendering flow</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Drawing lines with GL_LINES and GL_TRIANGLES</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Glyphs packed in texture atlas</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Mipmaps of the glyph texture atlas</td>
<td>27</td>
</tr>
<tr>
<td>4.5</td>
<td>One level batch rendering of tiles</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>The number of points and characters in the map data used to render lines,</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>polygons and text for the benchmark regions</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Rendering times for our naive OpenGL implementations compared to MapDemo.</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Rendering times with and without batched tile rendering</td>
<td>32</td>
</tr>
<tr>
<td>5.4</td>
<td>Rendering times with and without preprocessing</td>
<td>33</td>
</tr>
<tr>
<td>5.5</td>
<td>Rendering of preprocessed data compared to stencil buffer rendering</td>
<td>34</td>
</tr>
<tr>
<td>5.6</td>
<td>Text quality comparison</td>
<td>35</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparison between MapDemo and our final implementation</td>
<td>36</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Today, digital maps are used in many applications on mobile devices. This thesis discusses techniques for improving the rendering performance for a certain family of digital maps, vector based maps, using the on-chip graphics processor that many modern mobile devices are equipped with.

Map rendering is the process of transforming geographical data stored on file into a bitmap image that can be displayed on the screen. This transformation can be a computationally heavy task depending on the source data and the desired output. Bad rendering performance, in terms of speed and image quality, might affect the user negatively. This makes it interesting to look into how the limited hardware of mobile devices can be used as efficiently as possible to maximize map rendering performance.

1.1 Background

This section contains only the information necessary to describe the reasons for this thesis and the problem that is the core of it. For a more in-depth description and analysis of the different parts that make up the system that is to be improved, please refer to chapter 2.

1.1.1 IT-Bolaget Per & Per

IT-Bolaget Per & Per is a small IT consulting firm located in Mjärdevi Science Park, Linköping, Sweden. They are specialized in mobile and integrated solutions as well as enterprise IT services and they make sure to keep up with the latest technologies. IT-Bolaget Per & Per are early adopters of new products and services within their focus area.
Many of IT-Bolaget Per & Per’s current projects are centered around Geographic Information Systems (GIS), computer systems dealing with geographical data.

**Mobile map framework**

IT-Bolaget Per & Per uses different kinds of geographical map data as a base in many of their IT solutions. They have developed their own map framework for iOS devices that is capable of drawing custom maps depending on the needs of their customers. The framework supports both raster and vector based maps.

Many of the mobile applications developed by IT-Bolaget Per & Per are designed to be used far away from urban areas. Because of this, the framework is implemented in such a way that the map data can be either downloaded from a central server or stored locally to be used when cellular reception is bad or when data traffic is to be minimized.

The mobile map framework is built on top of the Apple Map Kit framework, see section 1.1.5. For displaying raster based map data, IT-Bolaget Per & Per’s map framework utilizes Map Kit functionality directly. For displaying vector based map data, the map framework uses a third-party open-source toolkit called Mapnik, see section 1.1.6, as a rendering engine to transform vector data into raster images that are then displayed using Map Kit.

MapDemo is an iOS app developed by IT-Bolaget Per & Per that, among other things, uses their map framework to render a custom map on device from vector data.

**1.1.2 Map rendering alternatives**

There are different ways of rendering maps in mobile applications. The simplest way is to bundle the entire map with the application before the app is distributed. This solution is not very storage efficient since more map data than what the user is interested in often has to be downloaded and stored together with the app. Instead, a common practice is to serve the part of the map that the user wants to see, on demand, from a central server.

Maps can be stored either as raster or vector data. Raster data means that the map consists of bitmap images that are ready to be displayed. The advantage of this approach is that no additional processing of the map data has to be done before it is displayed. The disadvantage is that raster data often requires more storage than vector data and that raster data is resolution dependent.
Vector data on the other hand is more flexible than raster data since it only contains the geometrical information needed to render an image. This means that the data can be stored compactly and rendered to images of different sizes in native resolution.

There are two main ways to render maps stored as vector data into pixels for display on mobile devices. One way is to let the server take care of the rendering and transmit processed raster images to the client. The second way is to let the client render the map itself.

Client-side rendering has multiple advantages. Vector data can be stored much more efficiently than raster data, which means that less data has to be downloaded to the mobile device, regardless of it being downloaded on demand or for offline usage. At the same time, the server needs only to act as a file server without any vector computational abilities if rendering is offloaded to the client.

As we will see later, the main reason for choosing to render server-side is performance.

This thesis focuses on client-side rendering of vector map data.

### 1.1.3 Map styling

Styling a map is the process of deciding what should be visible on the map and what the final result should look like. This could mean hiding small buildings on low zoom levels or setting the width and color of a road. For raster based maps, the styling has to be decided on before the map is created as the result cannot be changed.

This is not the case for vector maps as the styling can be completely separated from the map geometry, even into entirely different files. The files with style information contain rules that the map renderer has to obey and the end-result can be completely changed by just changing the style rules. Figure 1.1 shows two different ways of styling the same map geometry.

The style rules are applied at rendering time. Which style rule to choose might depend on the current visible region as it is common that the same map feature is styled differently depending on zoom level.

Different users might not be interested in seeing the same map as user requirements and needs vary, sometimes for the same user between different use cases. [18] gives (in Swedish) an in-depth look at the process of styling vector maps on mobile devices.

It is important to note that the look of the map can be changed dynamically without any need for external communication if and only if rendering is done on the client.
1.1.4 Hardware constraints

A mobile device has limited resources (processor, memory and battery) in comparison to a regular computer but might still have a high-resolution display. This makes it important for developers of graphic heavy applications, such as vector map renderers, to implement their pixel processing efficiently in order to ensure a fast and fluent experience for the end user.

1.1.5 Apple Map Kit

Custom iOS map renderers can be built on top of Map Kit, the general purpose map framework provided by Apple for all iOS developers to use. Map Kit is designed to be easily integrated into any iOS app, especially when using Apple’s own maps. A third-party developer with custom map data can still take advantage of Map Kit’s other features such as interaction with the map, location services, compass integration etc. as Map Kit provides support for annotating the map and drawing custom images on top of, or instead of, the map provided by Apple. Displaying custom map images using Map Kit can be done through a feature called tile based overlays. These overlays divide the map into a number of squares that can be processed and handled independently. For every geographical region that is to be displayed, Map Kit will request a bitmap image for each one of the tiles, given the tile’s size and coordinates on the map. An example of a custom tile overlay can be seen in figure 1.2.

1.1.6 Mapnik

Mapnik is a third-party open-source toolkit that can render vector based map data into raster images with predefined styling. OpenStreetMap, a
popular free map service with user editable maps, uses Mapnik to render the map on their website[14]. Mapnik is written in C++ and can be compiled for different platforms and architectures. There is an official iOS build of Mapnik.

1.2 Problem description

IT-Bolaget Per & Per has discovered that the rendering of bitmap tiles from Mapnik vector map data is very CPU-heavy on iOS devices. They do not get the desired rendering performance, especially on older hardware, which negatively affects the user experience.
The problem that this thesis is to tackle is to identify the bottlenecks of the current rendering pipeline and look into what can be done to improve the rendering performance. We are especially interested in knowing whether the on-chip GPU can be used not only for drawing the bitmap tiles but also for rendering vector map data with improved performance. If this turns out to be a feasible thing to do, IT-Bolaget Per & Per wants to know what needs to be changed in their current implementation to accommodate for such a change of work assignment.

1.3 Goal

A common goal has been decided on between us, the client and the examiner of this thesis:

- Investigate technologies that, based on the current map system framework or not, uses the GPU of the mobile device for vector map rendering in a way that optimizes the performance.
- Based on the findings, implement and demonstrate a prototype that uses the GPU for rendering, if this is possible to do in a satisfactory manner, or improve the current CPU-based solution if not.

The most important criteria to see whether the problem could be solved or not is to see how much the rendering times can be improved. In the best of worlds, the rendering and drawing of vector tiles can be made more or less instant. This would lead to a dramatically improved user experience.

There is also a possibility that we, with this thesis, discover that there is not much to be done as far as improving performance goes. This can still be interesting knowledge-wise for IT-Bolaget Per & Per if the limitations and causes have been identified.

1.4 Motivation

Customer satisfaction is very important for all kinds of user-facing software. IT-Bolaget Per & Per considers user interface fluency to be a big part of the overall experience of their mobile applications and they are interested in improving it whenever possible.

1.5 Approach

The problem is to be approached by looking into whether any best practices exists for rendering vector images on GPUs in general, and on iOS devices in
particular. Which technologies are available for us to use?

The current product needs to be examined in detail and benchmarked in order to get a good understanding of the current system and its bottlenecks. This will also be important in order to establish a performance baseline for comparisons with alternative technologies.

We are also planning on looking into research about handling vector map data. Can structuring and processing the data differently improve performance?

1.6 Scope and limitations

We have restricted us to work with iOS in this thesis. This still means that portability is an important topic that needs to be evaluated for a potential solution but any implementation is only to be done for iOS.

Another limitation is that we are to focus only on the rendering aspect of the map application. Other possible performance optimizations, such as image caching and improved file I/O operations might be implemented alongside a new rendering engine but will not be discussed in detail in this thesis.

We have also agreed to limit implementations to solutions based on Map Kit, Apple’s map framework. This is because of time constraints since evaluating many different map infrastructure solutions is not interesting from a pure rendering point of view and would require a lot of work beside the main topic of this thesis.

1.7 Related work

There are many open-source and commercial software applications and frameworks available for rendering maps, both for desktop and mobile platforms. IT-Bolaget Per & Per has either considered or tried many of these products but has still to find a solution that meets their requirements on performance and flexibility.

However, neither we nor the client are aware of any open source or commercial GPU-accelerated map rendering software for iOS.
Chapter 2

Product analysis

In this chapter we describe the current map rendering solution that is to be improved and identifies its performance bottlenecks. We explain how vector primitives stored on file are translated into pixels on screen and the infrastructure of an app that uses Map Kit tile overlays, such as MapDemo to display maps.

2.1 Indexing the world by tiles

As described in section 1.1.5, Map Kit lets the user display a custom map through a tile based overlay. For tile based overlays, the entire world is divided into square tiles that are laid out on a two-dimensional grid. Map Kit also uses a third dimension that represents the level of detail, also known as the zoom level. Zoom level 0 contains the entire world in one tile and has the lowest level of detail. For every subsequent zoom level, the number of tiles quadruple as every tile is divided into four.

There are different ways of projecting the three-dimensional surface of the earth onto a two-dimensional map. Map Kit uses what is called a Mercator projection which ensures that longitudinal lines that converge at the poles become parallel vertical lines in the projection. By using this projection, every geographical location on every zoom level is present in one and only one tile specified by x-, y- and z-coordinates.
2.2 Tile sizes in Map Kit

By default, the tile size of Map Kit tile overlays is 256x256 pixels irrespective of the screen resolution of the device, e.g. Retina or non-Retina displays, see MKTileOverlay in [2]. This causes Map Kit to behave differently for iOS devices with Retina displays and devices with non-Retina displays. Instead of requesting tiles of 512x512 pixels in size on Retina display devices, Map Kit request four times the number of 256x256 tiles. In order for the same geographical region to be displayed, these tiles are on one zoom level higher compared to a non-Retina display device.

The Map Kit framework reference[2] documents a tileSize property of tile overlays which could be used to change the size of the requested tiles and thus alter the behavior described above. Interestingly, both our and IT-Bolaget Per & Per’s tests have shown that changing this property has no effect on how Map Kit requests tiles. As such, any implementation using Map Kit tile overlays is restricted to 256x256 pixel tiles.

2.3 From vector data files to pixels

As described in 1.1.5, when using a custom tile overlay on top of the map provided by Apple, Map Kit will request bitmap image data, tile by tile, for the geographical region that is currently visible.

As we have seen, one of the main advantages of using vector based map data is that the data is more or less resolution independent. The same data can be rendered on different zoom levels in different image sizes and never look pixelated. A distributor of map data can provide data for all or some chosen zoom levels, meaning there is a one-to-one mapping between a tile and a map data file on such zoom levels. All the information needed to render a tile is found in a single file. For other zoom levels, the information needed to render a tile is found in the closest lower zoom level that has a corresponding file. Since one tile on a lower zoom level corresponds to the same geographical region as multiple tiles on higher zoom levels, the same file might be used for rendering multiple tiles simultaneously.

For example, let say that we have map data files for zoom level 8. Now, if we want to render a tile A on zoom level 9, tile A’s geographical region is included in a tile B on zoom level 8 so that file is used when rendering A. At the same time, another tile C on zoom level 9 might also correspond to B on zoom level 8 since B covers a bigger geographical region than A and C. This means that the same file on zoom level 8 will be used for multiple tiles on zoom level 9. Figure 2.1 illustrates this example.

This imposes a potential performance problem for zoom levels without files
as the same file will be read and the same data processed for multiple tiles while only a portion of the features of the file will be visualized for every tile. Devices with Retina display suffer even more from the bottleneck described above since, as described in section 2.2, Map Kit will request four times the number of tiles on a Retina display device compared to a non-Retina display device.

![Figure 2.1: Tiles on multiple zoom levels. Illustration of the example discussed in section 2.3.](image)

The map data files are stored on a central server but can also be cached locally after they have been downloaded to the client. Every file contains a number of features for that tile, each belonging to one of a number of global layers. Layers most often correspond to a certain kind of map object such that all features of a layer should be rendered in the same way. For example, we can have one layer with lakes and one layer with railroads.

Every feature is of one of the following forms:

- Lines. Roads, power lines, railroads etc. are represented as lines. Line features consist of a number of line segments whose coordinates are specified in the file.

- Polygons. Forests, lakes, buildings etc. are represented as polygons. Polygons are closed paths consisting of a number of line segments whose coordinates are specified in the file.

- Points. Text and special map symbols are represented as points. Points contain character, coordinate and font information in the file.

Since the tile vector files only contain geometrical information of features and nothing about their visual representation, one or multiple additional files containing styling rules are included with the application. These files state, for example, that polygon features of the lake layer should be filled
with a certain shade of blue or that names of cities should be rendered with a certain typeface.

Both the geometrical and visual information described above is given as input to the Mapnik renderer together with the dimensions, in pixels, of the requested image. The tile can then be rendered into a bitmap image that is given back to Map Kit to be displayed.

### 2.4 Mapnik rendering

Mapnik supports a variety of rendering backends, AGG (Anti-Grain Geometry) being Mapnik’s primary renderer. A Mapnik renderer implements a number of methods that tell Mapnik how to process different kinds of geometry and provides Mapnik with a rasterizer object that is able to convert the geometry into pixels in a bitmap image.

Mapnik processes the geometry by applying a number of vertex conversions that transform the geometry in one way or another. Which vertex conversions to apply is decided on by the renderer, some are required to produce a correct result while some are optional and might increase performance or the visual quality of the image depending on the capabilities of the rasterizer. Some of the vertex conversions Mapnik provides include:

- **Transform.** I.e. projecting the geometry onto the coordinate system of the bitmap image.
- **Clip.** Clipping the geometry to only include what will be visible in the current bitmap image.
- **Simplify.** Reducing the number of points in the geometry.
- **Smooth.** Approximating the geometry to give smoother shapes.
- **Dash.** I.e. make a line dashed.
- **Stroke.** Convert a line of a certain width into a polygon.

The clip, smooth, dash and stroke conversions use functionality provided by AGG whereas the transform and simplify conversions use other functionality provided by Mapnik.

As described in section 2.3, only lines, polygons and points (text) occur in the map used by MapDemo. After a line or polygon feature has passed the vertex conversions, a resulting path is passed on to the rasterizer. Paths consist of a series of vertices (x-, y-coordinates) and are either rasterized as filled in the case of polygons or non-filled in the case of lines. The AGG rasterization process also uses Anti-Aliasing and Subpixel Accuracy techniques to improve the visual quality of the resulting bitmap image. Text is handled a bit differently from lines and polygons, Mapnik uses FreeType
for text rasterization and text is represented as point features in the data. Point features contain position, font and a number placement and layout parameters such as displacement, alignment, rotation and color.

### 2.5 Rendering performance bottleneck

The number of features in a tile varies a lot but can easily exceed several thousand in some maps. Every one of these features for every tile needs to be translated into pixels and colored in the right way according to the style rules. Since the feature processing and rendering is solely done on the CPU on the device by the Mapnik iOS library, rendering performance is limited by the number of features the CPU can process in a given time frame.

While this is happening, the GPU sits idle waiting for a processed image to display. The question we ask ourselves in this thesis is: can we do something smarter that takes advantage of the combined performance of the CPU and the GPU in order to render tiles faster? This would mean that the map would appear faster and power consumption could possibly decrease if the CPU would not have to work under high load for as long.
Chapter 3

Graphics rendering on iOS

In this chapter we investigate what technologies can be used on iOS to increase the performance of IT-Bolaget Per & Per current map rendering solution.

3.1 Possible graphics technologies

Apple provides several frameworks for creating and interacting with media on iOS devices but only two frameworks for rendering graphics. These are Core Graphics and OpenGL ES, OpenGL ES being the only alternative that gives the developer control over the graphics hardware directly. This is because function calls to OpenGL are designed to be translatable to GPU commands.[12].

The Apple graphics technologies hierarchy can be found in [5]. It says that Core Graphics, also known as Quartz, is the native drawing API for iOS apps. Core Graphics supports custom 2D vector- and bitmap-based rendering but is not as fast as OpenGL ES.

The same source also states that OpenGL ES is a technology that is implemented closer to the graphics hardware and that it defines a platform-neutral API for rendering graphics on the GPU.

Rendering vector map data into bitmaps can be done using any of the two mentioned frameworks or a third party library such as Mapnik.

OpenGL ES can be used to produce bitmap tiles from vector data by rendering to an off-screen framebuffer object. The bitmap images that are
extracted from the framebuffer can be used as tile overlays to Map Kit. To be able to call any OpenGL ES functions the app needs to create an OpenGL ES context that manages the rendering state. By creating multiple contexts it is possible to have multiple threads rendering simultaneously without interfering with each other. However, it is important to restrict each context to a single thread [12].

The CPU and GPU share the same main memory on iOS devices [12]. This reduces the overhead of copying rendering results from the GPU to the CPU.

### 3.2 Tile-Based Deferred Rendering

Tile-Based Deferred Rendering (TBDR) is a technique used by the GPUs in all iOS devices that allows the GPU to access memory very efficiently and discard drawing commands to areas outside of the current framebuffer object. This means that vertices of OpenGL geometry primitives that are sent to the GPU do not necessarily have to lie within the visible area that has been specified by the developer, they will simply be discarded if they do not.

TBDR also allows for hidden surface removal which can significantly reduce the calculations needed by the GPU. TBDR works best when rendering large scenes and may lose much of its efficiency when rendering smaller scenes. [6]

### 3.3 Text rendering

In addition to lines and polygons, text is an essential part of the map rendering process. There are mainly two possible approaches to display text on the map when using Map Kit: either drawing the text on top of the map by using Map Kit functionality or by rendering the text into the bitmap tiles themselves.

Map Kit provides a concept of displaying content that is defined by a single coordinate point, called annotations [8]. Annotations can be added and removed from the map as required and can be customized to take advantage of the text rendering abilities of iOS. An advantage of displaying text using annotations is that the text is kept separate from the bitmap tiles. This allows for the text to preserve its size and rotation while zooming and rotating the map, which may be desirable in some apps. Refer to section 6.3.1 for a discussion about using annotations.
If instead rendering the text into the bitmap tiles, it is possible to use OpenGL ES and thus fully control the utilization of the GPU. OpenGL ES itself does not provide any text rendering functionality but text rendering using OpenGL is a well-researched area in computer graphics and there are techniques to achieve fast text rendering using OpenGL, see section 3.5.3.

3.4 OpenGL ES drawing primitives

OpenGL ES supports rendering of primitives such as points, lines and triangles but not arbitrary geometrical objects. As we will see, the triangle is the most interesting OpenGL ES primitive for our purposes. The different ways of specifying triangles in OpenGL ES are described below.

3.4.1 Triangles

The triangle is the simplest and most useful two-dimensional geometrical shape in computer graphics. Because of its flexibility when it comes to geometry construction, most graphics hardware is constructed to be very good at rendering triangles. Triangles have the advantage that many of them can be connected in order to create more complex shapes. In OpenGL, triangles can be drawn by submitting a list of vertices to the GPU. How the list is interpreted depends on the current drawing mode. There are three modes for drawing triangles:

*GL_TRIANGLES* is the simplest. The triangles are all specified explicitly and independent of each other in sequence. If the vertex list ABCDEF is submitted, two triangles are created: ABC and DEF.

*GL_TRIANGLE_FAN* makes use of the first vertex as a common vertex for all triangles. The first triangle is created using the first three vertices, then for every subsequent vertex, a triangle is created between the common vertex, the previous vertex and the current vertex. If the vertex list ABCDEF is submitted, four triangles are created: ABC, ACD, ADE and AEF.

*GL_TRIANGLE_STRIP* is similar to GL_TRIANGLE_FAN but keeps track of the previous two vertices instead of one previous vertex and the common vertex. For every vertex submitted, a triangle is created between it and the two preceding vertices. If the vertex list ABCDEF is submitted, four triangles are created: ABC, BCD, CDE and DEF.

Triangle strips and triangle fans have an advantage over simple lists of triangles in that they can specify a set of (connected) triangles using far less data.
than if every triangle would be described individually by its three vertices. In the examples above, four triangles were described instead of two using the same amount of data.

**Degenerate triangles**

The downside of using triangle fans or strips is that all the resulting triangles will be connected to each other. This is a good thing in many cases, e.g. when specifying more complex polygons as a set of triangles or when dealing with connected line segments represented as triangles.

Sometimes you want to submit different features to OpenGL that do not have anything to do with each other and should not be visually connected. This is easy to do with multiple draw commands but since every draw command sent to OpenGL comes with a performance overhead, it is not always the best option.

Rendering multiple disconnected shapes in one call can be accomplished for triangle strips by using *degenerate triangles*, zero-area triangles that are created by submitting the same vertex two times for the same triangle. By using degenerate triangles and triangle strips, non-visible jumps can be introduced in the geometry. Degenerate triangles does not work the in the same way for triangle fans since all triangles in a triangle fan share a common vertex. However, by submitting the common vertex multiple times it is possible to achieve gaps between the triangles in a triangle fan.

Submitting multiple disconnected shapes at once can also be accomplished by using the OpenGL *primitive restart* feature in later versions of OpenGL. For mobile devices supporting OpenGL ES, primitive restart was introduced in version 3.0.

### 3.5 Creating OpenGL compatible geometry

We have seen in section 2.3 that vector-based maps contain lines, polygons and text. The line and polygon geometry can be specified arbitrarily by the map maker. This does not imply any restriction that prohibits the creator of the map to include geometry that is not drawable directly with OpenGL. As such, there must be a way to process the map geometry data into OpenGL primitives for OpenGL to be a viable rendering alternative.

Luckily, geometry reconstruction is a problem often encountered in OpenGL applications with many solutions with different advantages and disadvantages depending on the result requirements and data input format. This section discusses what we can do for the different kinds of geometry that we need to be able to handle.
3.5.1 Lines

A line is given as a list of line segments with specified endpoints and an associated line width. The width of the line is specified in the style sheet and can thus be changed from within the app. In other words, the width is not a static property.

The style sheet can also specify that the line should be dashed using dashes of a certain length and with a certain gap between dashes.

A graphical representation of a line segment is a two-dimensional rectangle. OpenGL can draw such a shape using either one of the OpenGL line primitives with the line endpoints as input (the width can be specified with the `glLineWidth()` function call in OpenGL ES 2.0) or as two connected triangles.

If the triangle approach is used, it is preferable to use `GL_TRIANGLE_STRIP` as many segments can be specified in one strip using as little data as possible.

When it comes to dashed lines, there is no native support in OpenGL ES 2.0 to draw dashed lines. Desktop OpenGL includes support for dashed lines through the `glLineStipple()` function call but it is not part of OpenGL ES on iOS. The dash effect can be achieved by breaking up the connected line segments into smaller disconnected segments or by applying a striped texture with transparent segments on top of the line. Both of these approaches require computing the endpoints of the dashes on the fly, either for constructing new geometry or for specifying texture coordinates.

3.5.2 Polygons

Since OpenGL provides no functionality to draw arbitrary polygons directly\[13\], it has to be done in some other way. There are two different ways, described below, to do this live in OpenGL. It is also possible to preprocess polygon geometry so that OpenGL compatible features are available when needed. The same end-result can be achieved with all of these approaches.

Note that these methods are designed to work with single-colored polygons, meaning every graphics fragment has the property of either being inside or outside the polygon. For multi-colored polygons or polygons with gradients, other techniques, e.g. working with textures, have to be used.

CPU triangulation

A well-researched technique to draw arbitrary polygons is triangulation, subdividing the polygons into triangles, on the CPU before the new triangle
geometry is sent to OpenGL to be drawn with one of OpenGL’s triangle primitives.

In computer graphics, polygons can be categorized as either simple or complex, where the latter means that the polygon is self-intersecting, has internal holes and/or overlapping sides as seen in figure 3.1.

![Simple and complex polygons](image)

**Figure 3.1:** Simple and complex polygons.

Complex polygons are hard to triangulate since most efficient triangulation libraries or published algorithms assume that the input polygons are simple. Unfortunately, this is not always the case for vector map data and our solution has to be robust enough to handle any simple or complex polygon.

The General Polygon Clipper library (GPC) from the University of Manchester can triangulate arbitrary polygons as well as output the triangulations triangle strips\(^4\). We have seen that the triangle strip is a supported primitive by OpenGL. Because GPC is written in C, it is also highly flexible and can be used on multiple platforms. This made GPC our first choice for triangulating and clipping polygons.

It is worth noting that there are algorithms that produce better triangulations in terms of the number of triangles and the minimum angle of any triangle in the triangulations (in order to avoid very thin triangles). Shewchuk \(^{20}\) described such an algorithm in 1996 but to our knowledge it has not been implemented in a library as robust, small, flexible and easily integrable as GPC.

**OpenGL stencil buffer rendering**

The other way to render complex polygons is to make use of OpenGL’s stencil buffer. Rueda et al. describes a way to do this in \(^{16}\). The idea is to take the polygon and construct a triangle fan using the centroid of the
polygon as the common vertex. The fan is rendered in the stencil buffer, with depth and color buffer writes disabled, counting the number of times a fragment is written into. It turns out that exactly the fragments enclosed by the polygon outline will be written to an odd number of times. This means that we, after drawing the triangles to the stencil buffer, can enable color and depth buffer writes, draw a big rectangle than encloses the polygon and use the fragments of the stencil buffer with an odd number of writes as a mask. The result is a correctly rendered polygon.

This algorithm can be simplified a little by taking advantage of the fact that we are only interested in knowing whether the resulting number of writes to a fragment is odd or even. If the stencil buffer is cleared before drawing each polygon and the OpenGL stencil buffer operation invert is used every time a fragment is written to (effectively toggling the value between zero and non-zero), the stencil buffer will have a non-zero value for a fragment if and only if it is interior to the polygon after rendering.

It turns out that there is nothing magical with the centroid of the polygon in this algorithm. [21], a well respected book in the computer graphics community, describes the same algorithm using the first point of the polygon as the common vertex in the triangle fan. The advantage of using the centroid is that it keeps the generated triangles relatively small, resulting in fewer stencil buffer writes.

A big difference between the two proposed solutions is that polygon triangulation is a CPU intensive task whereas the complexity of the stencil buffer approach lies in rendering geometry in a temporary buffer as well as drawing to the depth and color buffers with a stencil mask. These are GPU heavy operations.

Preprocessing polygon geometry

One possibility to avoid the need for triangulating map features on demand, at rendering time, when using the triangulation approach is to preprocess the data into triangulated geometry. This is something that can be done either on the central server or on device after the map data has been downloaded to local storage. In both cases, there is possibly a performance advantage in bypassing the triangulation step at rendering time and sending the map geometry information to the GPU directly.

The preprocessing process is highly parallelizable regardless of where it is done. This is because no tile depends on any other. All tiles can be processed independently.

As preprocessing map data only consists of performing some of the work that will have to be done anyways at an earlier stage of the rendering process, the end-result does not differ.
3.5.3 Text

As OpenGL itself does not provide any text rendering functionality, rendering text in OpenGL can be achieved by either converting text into geometry or textures. A common technique is to use a font rasterizer to create glyph images that are used to texture simple quads. This technique is applicable with several graphics rendering APIs, including OpenGL. A widely used font rasterization software suitable for this purpose is FreeType, it is used by Mapnik in combination with the AGG renderer. Using this technique, the glyph images can either be created in advance or, as in Mapnik, on demand. The latter approach can result in a higher quality result as it allows for the font rasterization software to render in native resolution and apply anti-aliasing and hinting techniques that account for the size and orientation of the glyphs. When creating the glyph images in advance, it is often needed to scale and rotate them to achieve the desired size and orientation, which can introduce visible artifacts as a result of interpolation. However, as rasterizing fonts is computationally heavy and creating the glyph images in advance does reduce the amount of work needed to display text, it might be preferable in practice.

The article [19] describes a method, which is a variant of the above mentioned technique, to render text where glyph images are rendered and packed into a texture atlas. The method includes an algorithm to efficiently pack the glyph images in the texture atlas, making it possible to store glyphs of multiple fonts and sizes in a single texture atlas. This minimizes the number of textures and the texture size needed to store the glyphs. The glyph images are rendered into the texture atlas as font masks, containing only alpha information. The proposed technique is able to achieve high quality text rendering under certain circumstances where the text does not need to be scaled or transformed afterwards. **Signed distance field fonts** are suggested as a better option for such cases.

A text rendering technique using signed distance field rendering has been presented in a paper by Valve [17]. While the previously described font rasterization technique renders font masks, indicating if a texel in the texture is inside or outside the glyph, a signed distance field texture contains a distance value for each texel, indicating how far each texel is from its nearest texel inside the glyph. The idea is that such signed distance field textures are better suited for linear interpolation and thus able to produce a higher quality result when scaled or transformed by the graphics rendering API.
Chapter 4

Implementation details

This chapter describes the different approaches to utilizing the GPU together with Map Kit that we investigated and implemented in order to see if rendering performance could be improved.

As seen in chapter 3, using OpenGL is currently the only way to fully control and utilize the on-chip GPU when rendering image data on iOS. OpenGL is also supported on many other platforms than iOS, an advantage for larger versatile applications. This made OpenGL our technology of choice to focus our implementation efforts on. We decided to implement our solutions using OpenGL ES 2.0 since the more recent version, OpenGL ES 3.0, is not available on all iOS devices that IT-Bolaget Per & Per want to support.

As described in section 1.1.5, Map Kit supports tile based bitmap overlays to display custom map data. When using tile overlays is the only alternative that allows the custom map data to be rendered with OpenGL. Therefore, the approach we have chosen is to render bitmap images using OpenGL and use them as tile overlays with Map Kit.

4.1 Parallel rendering using OpenGL

Map Kit requests multiple tiles at once in an asynchronous manner to allow for good utilization of the multi-core architecture on modern iOS devices. Therefore, we chose to make the tile rendering able to run in parallel as well. As described in chapter 3, OpenGL is able to render concurrently into separate OpenGL contexts as long as each context is always accessed from the same thread. To ensure this we implemented a renderer manager to handle the creation and usage of our OpenGL renderers. Each OpenGL
renderer manages its own OpenGL context and the renderer manager makes sure that each renderer runs in its own thread.

Because of Map Kit's asynchronous architecture, the drawing of a tile onto the map does not immediately follow the Map Kit tile request. Therefore, each tile that is rendered as a result of a tile request is put into an interim storage until Map Kit is ready to draw the tile onto the map. The flow is illustrated in figure 4.1.

![Tile rendering flow](image)

4.2 Styling and rendering tiles

We have seen in section 1.1.3 that the process of styling map features consists of traversing all visible features for a tile and applying the active style rules according to a style sheet. As described in section 2.4, Mapnik already implements this functionality.

We decided on implementing two different approaches, 4.2.1 and 4.2.2, to see if Mapnik could still be used efficiently when rendering with OpenGL or if we would benefit from implementing our own feature processing and styling functionality.

4.2.1 An OpenGL rendering backend in Mapnik

As Mapnik supports several rendering backends, it is written in a way that makes it possible to implement a custom renderer. We took advantage of this and implemented an OpenGL renderer to see if the app could benefit from utilizing the GPU instead of AGG's software renderer. The main advantage of this approach is that we can rely on Mapnik for processing and styling the
map data, as in MapDemo. To implement an OpenGL renderer in Mapnik we looked at how the AGG renderer was implemented and changed the necessary parts in order to capture and prepare the geometry to be able to render it with OpenGL instead of AGG.

4.2.2 A custom renderer without Mapnik

In order to have the most control over the rendering process, we implemented the approach of letting Mapnik parse the visual style information from file but handle parsing and processing of the feature geometry into OpenGL vertex data ourselves. This means that we can avoid performing unnecessary work that Mapnik might do to prepare data for the software renderer and instead prepare the feature geometry for OpenGL directly.

4.3 Approaches to creating OpenGL compatible geometry

As described in section 2.4, the map data used for MapDemo contain lines, polygons and points (text). We have previously seen in section 3.4 that the set of geometrical objects that is drawable with OpenGL is limited and smaller than the set of geometry that our map contains. The map data has to be prepared before it is submitted to OpenGL.

As there is no obvious way to do this that is fast, memory efficient and works for all kinds of geometry, we tried out different possible alternatives for different feature types and compared them for performance and aesthetics.

4.3.1 Lines

For rendering the lines on the map, we tried both using OpenGL’s native line primitive (GL_LINES) as well as rendering every line segment as two triangles.

We concluded from a purely visual inspection that only the triangle approach could deliver an end-results that matched the quality of the software renderer on iOS. The GL_LINES approach had problems producing a satisfactory geometry. As can be seen in figure 4.2 the problem was particularly visible at the line caps for very short lines as OpenGL drew them with horizontal or vertical cuts, even when the lines did not follow these directions.

The triangle geometry is constructed by taking the endpoints of every line segment together with the width of the line and creating two triangles with
Figure 4.2: Drawing very short lines with GL_TRIANGLES (left) and GL_LINES (right) with a visible line cap problem in the latter case. The direction of the short dashed lines is the same as for the long lines, i.e. their width is longer than their length.

sharing hypotenuses. The triangles from multiple connected line segments can be easily connected in a single triangle strip.

Dashed lines were accomplished by using the approach of computing new line geometry taking into account the rules from the active style sheet. Multiple disconnected line segments were sent to OpenGL in one function call using triangle strips with degenerate triangles as described in section 3.4.1.

4.3.2 Polygons

Since there are two completely different approaches to rendering polygons, see section 3.5.2 that utilize the device resources in different ways, we implemented both of them in order to do a comparison. Comparing CPU triangulation to the OpenGL stencil buffer approach can be done solely from a performance point of view as the end-result looks the same.

The GPC library was used for triangulation whenever applicable. It outputs triangle strips that can be forwarded directly to OpenGL.

In order to get an understanding of the magnitude of the rendering time that could be saved by not having to triangulate on demand, we implemented an on device preprocessing step as described in section 3.5.2 that iterated through all local tiles and converted the general polygons into triangle strip geometry before the user was able to see the map.

Unfortunately, one has to be careful not to destroy the original geometry. As discussed in section 1.1.1 dynamic map styling is a key advantage of vector based maps. This means that the style rules might change after the data has been processed and triangulated. For example, if a user all of a
sudden wants to stroke the outline of all lakes with a different color than the fill color of the lakes, we will run into problems if we have overwritten the original geometry with a set of triangles instead of keeping the original path that describes the outline.

As such, it is necessary to keep both the original as well as the triangulated geometry. This leads to having to store more data which might or might not be a feasible option depending on the performance advantages and the size of the additional data.

When implementing the OpenGL stencil buffer version of polygon rendering, we tried both the approaches described in section 3.5.2 for which common vertex to use; the polygon centroid or the first vertex of the polygon. Using the centroid might reduce the total area of the resulting triangles in the triangle fan and reduce the number of write operations to the stencil buffer. However, for convex polygons, using the centroid and the first vertex will result in a triangle fan covering the same area. Thus, computing the centroid of the polygon might introduce unnecessary overhead.

Multiple polygons can be submitted to OpenGL using multiple draw commands or in one command using the technique with degenerate triangles described in section 3.4.1. The latter is used when triangulating as the GPU will color the same area in both cases and the small overhead of just executing one command is preferable over a lot of communication between the CPU and the GPU. The decision is not as easy for the stencil buffer approach. Using the same vertex as the common vertex for many polygons within a tile will most likely result in rendering big triangles in the empty area between polygons (as all triangles for all polygons will share a common vertex) to the stencil buffer an even number of times resulting in a lot of unnecessary stencil buffer write operations.

4.3.3 Text

The text rendering technique described in the article is implemented in the project Freetype GL. We decided to use this implementation out of simplicity as it provides an easy-to-use interface and has all the functionality we need. As the Freetype GL library is primarily written for desktop OpenGL we had to make some minor changes to make it comply with OpenGL ES.

The Freetype GL library is able to pack multiple fonts into a single texture atlas. We take advantage of this by letting the library render and pack glyphs of all of the required fonts into a single texture atlas. This is done once during application launch to maximize performance. Freetype GL is not able to choose the font size that maximizes the usage of the texture atlas for a given set of glyphs. This means its desirable to chose a font size such
that all glyphs fit into the texture atlas while achieving good texture usage. See figure 4.3 for the resulting texture atlas in our implementation.

Figure 4.3: All glyphs of the required fonts for IT-Bolaget Per & Per’s map packed into a single gray scale texture atlas.

Because the text size is decided on by the style sheet, the application may render glyphs of many different sizes. While it is possible to render multiple glyphs of multiple font sizes, we chose to render each set of glyphs for the required fonts in one font size only, then rely on scaling to achieve the required text sizes. This is because the application may require more text sizes than would be practical to have rendered in one or more texture atlases due to resolution or memory limitations. While scaling the glyph textures may reduce the text quality significantly due to texture interpolation, it is possible to reduce such decrease in quality by letting OpenGL generate mipmap textures for the texture atlas. Mipmapping is a technique first described by Williams [22] in 1983 as a way to improve image quality when downscaling images. Mipmaps are a set of down-sampled versions of a texture that accompany the original texture. The technique is supported natively in OpenGL ES 2.0 on iOS. Mipmaps can be generated automatically by OpenGL and are created recursively by halving the texture size each time, as can be seen in figure 4.4. When rendering, OpenGL automatically chooses either the original texture or the mipmap texture that best matches the target resolution, thus achieving a high quality result.

When the texture atlas is generated, a mapping between glyphs, font information and texture coordinates is created. This mapping is used when rendering text to texture simple triangle geometry with sub-regions of the texture atlas corresponding to the glyphs and font to be rendered.
4.4 Batch rendering of tiles

As described in section 2.3, one major bottleneck in MapDemo is that the rendering on some zoom levels caused many tiles to read and process the same vector data redundantly.

We implemented batch rendering of tiles, meaning rendering multiple tiles at once, whenever possible to eliminate this bottleneck. Instead of letting multiple tiles on a zoom level without vector data read and process the same vector data file from a lower zoom level, we batch render a larger tile corresponding to a zoom level closer to or at the zoom level of the file. The idea is illustrated in figure 4.5.

Ideally one would always render all tiles covered by the same vector data file in one operation as it would eliminate the need to read and process the same data more than once when rendering all tiles for a certain region of the
map. In practice, this may require rendering images of a higher resolution than a specific device can handle, i.e. when there is a large difference in zoom level between the tiles to be rendered and the vector data files. The maximum resolution of an image that can be rendered is limited by the maximum size of an OpenGL ES render buffer, which is device specific. It is also impractical to use too large render buffers as larger buffers limit the number of renderers that can be used due to their memory consumption. As the CPU and GPU share system memory, it is desirable to limit the GPU memory utilization in order to achieve good overall app performance and user experience.

We have chosen to always batch render a power of two number of tiles out of simplicity. A batch rendering level of one corresponds to a virtual zoom level one step above the current zoom level. A batch rendering level of $n$ produces $2^n$ tiles at once. Our implementation limits batch rendering up to a certain number of tiles at once, a maximum batch rendering level. As described in section 2.2, Map Kit requests and renders 256x256 pixel tiles and, for example, for a batch rendering level of two, the rendering resolution is doubled two times corresponding to an image size of 1024x1024 pixels and sixteen tiles at once.

After the batch has been rendered, the tiles are split up into 256x256 pixel tiles before being drawn on the map by Map Kit.
Chapter 5

Result

5.1 Benchmarking setup

In order to compare the rendering performance of different implementations throughout the thesis work, we decided on constructing a benchmark test suite to make the testing consistent and reliable.

Our test suite consisted of rendering six different map regions with a varying number of map features and tiles on different zoom levels. We chose six regions that covered a broad range of map data, i.e. regions with a noticeable difference in the number of features of different types. The amount of data for each of the different feature types, i.e. lines, polygons and text, for the six benchmark regions is presented in figure 5.1. For each region, the table shows the total number of points in the map data that are used to render lines and polygons respectively as well as the total number of text characters used to render labels and symbols.

<table>
<thead>
<tr>
<th>Region</th>
<th>Line points</th>
<th>Polygon points</th>
<th>Text characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34992</td>
<td>473170</td>
<td>2230</td>
</tr>
<tr>
<td>2</td>
<td>68192</td>
<td>384397</td>
<td>1134</td>
</tr>
<tr>
<td>3</td>
<td>26751</td>
<td>142644</td>
<td>535</td>
</tr>
<tr>
<td>4</td>
<td>30819</td>
<td>142644</td>
<td>558</td>
</tr>
<tr>
<td>5</td>
<td>52168</td>
<td>98659</td>
<td>45103</td>
</tr>
<tr>
<td>6</td>
<td>26581</td>
<td>16632</td>
<td>2002</td>
</tr>
</tbody>
</table>

Figure 5.1: The number of points and characters in the map data used to render lines, polygons and text for the benchmark regions.

Only the actual rendering of tiles from files on disk to bitmap images was timed, not drawing these images to the screen as that is done by iOS and...
Map Kit irrespective of how the rendering is implemented. All map data was stored locally on device before the benchmark was started and the same iPad Air device was used for all tests. Every region was rendered five times for all implementations and an average rendering time was computed per region per implementation for comparisons.

The iPad Air was chosen as the testing device as it is the most modern Apple iPad at the time of writing and currently the main device of interest for IT-Bolaget Per & Per.

5.2 Naive OpenGL rendering

As we saw in chapter 4, our first approach was to process the same data as MapDemo, at rendering time, and perform triangulation on the CPU. The triangles were rendered using OpenGL instead of a software renderer like AGG as in MapDemo.

We implemented this in two slightly different ways: by processing the map features ourselves (section 4.2.2, called CustomProcess below) but also by letting Mapnik process the map features (section 4.2.1, called MapnikProcess below). The reason behind this was that we did not know which one of these two methods was preferable from a performance and flexibility point of view and we wanted to compare them. The benchmarking results can be seen in figure 5.2.

As seen in the diagram, the rendering performance differs quite a lot between different regions and the three implementations. What is interesting in this diagram is the difference between MapnikProcess and CustomProcess. MapDemo is included here as a baseline and the difference between MapDemo and the other two should not be interpreted as a definitive answer to the question whether rendering performance can be improved by rendering the map with OpenGL. The results from our implementations are from naive and non-optimized renderers.

What is interesting to note is that CustomProcess depends less on third-party functionality than MapnikProcess, meaning there would be more room for fine-tuning the data flow in CustomProcess in later stages. This, together with the favorable results, inspired us to focus our development efforts on the CustomProcess approach to rendering.

From analyzing the individual performance of the OpenGL based rendering alternatives, it became clear that triangulating line and polygon features on demand was a performance bottleneck. This motivated us to look into what could be done to eliminate this step as it is something that MapDemo’s software renderer does not need.
5.3 Batch rendering

Our batch rendering implementation described in section 4.4 did increase overall rendering performance since less data is read and processed redundantly. The benchmarking results can be seen in figure 5.3.

Another advantage of batch rendering is that less renderer operations are required to render any full view of tiles. This decreases the need for running many renderers in parallel to maximize performance. Since the RAM used when rendering, e.g. for storing OpenGL buffers, is proportional to the number of renderers used, it is beneficial to keep this number low.

5.4 Pre-triangulating polygon features

As described in section 3.5.2 and 5.2, there is possibly a big performance gain in preprocessing the map features into OpenGL triangle geometry. The benchmark results can be seen in figure 5.4, which compares the rendering performance with and without preprocessing.
For our sample region of about 18,500 tiles, this approach resulted in having to store about 20% more data on device. IT-Bolaget Per & Per considers this increase to be acceptable if there are clear performance advantages over all other online processing and rendering approaches since their customers usually do not fill up their on-device storage. Speed is prioritized over efficient data storage as long as it does not mean storing excessive amounts of data.

As seen in the diagram, the performance advantage of preprocessing the map data is substantial and justifies the additional storage needed.

When deciding on whether this would be a viable alternative in a customer product, IT-Bolaget Per & Per would have to decide if their customers could accept waiting for the device to preprocess all map data before using the app or if it would be better to preprocess the data server-side. The latter would lead to more data for the client to download as discussed in section 3.5.2.
5.5 Stencil buffer polygon rendering

The second approach to rendering polygons that we implemented was using the OpenGL stencil buffer. This algorithm differs greatly from triangulating polygons on the CPU but yields the same end-result assuming that the CPU triangulation algorithm handles arbitrary polygons correctly. As noted before, our triangulation library of choice, GPC, does this in satisfactory manner.

This means that the only interesting comparison aspect is performance. As we discovered in section 5.4 pre-triangulating all polygon features is a feasible option when taking the CPU triangulation approach. Figure 5.5 depicts a performance comparison between these two implementations.

The result might seem non-intuitive at first. If all polygon features are pre-triangulated before rendering, how can it be slower to draw the triangles directly instead of taking a detour through the stencil buffer? We believe that the answer lies in the amount of data that needs to be processed.

As mentioned in section 4.3.2, we need to store both the triangulated version and the non-triangulated version of every polygon after pre-processing the
Figure 5.5: Rendering of preprocessed data compared to stencil buffer rendering.

data in order to be able to provide custom polygon styling. This means that more data has to be traversed and styled. It is also important to remember that drawing triangles using the stencil buffer is a GPU heavy operation, meaning the CPU can process different data simultaneously.

5.6 Final implementation

The final software is based on the early CustomProcess approach. It uses OpenGL to render vector map data into a framebuffer object that is converted into a bitmap image and transmitted to Map Kit for display. It implements batch rendering for fast processing of multiple tiles originating from the same source data file. The styling of the map is dynamic and can be changed on device.

Just as MapDemo, our implementation supports rendering lines, polygons and text. Lines are converted into triangle strips for maximum quality and flexibility. Polygons are rendered using the stencil buffer approach for good rendering performance, for eliminating the need of preprocessing and
additional data storage and for not having to depend on a third-party triangulation library. Fonts are rendered into a texture atlas on startup using Freetype GL and text is drawn by texturing simple triangle geometry.

5.6.1 Quality

In a side by side comparison between MapDemo and our final implementation, there are very few visible differences in the end-result. OpenGL is able to draw the same geometry with the same style rules applied.

The only notable visible difference is the rendering of text where MapDemo manages to deliver a slightly better looking end-result. This is expected since Mapnik renders text on the fly in the requested resolution whereas our final implementation makes use of a texture atlas that requires the labels to be scaled and rotated into the requested size and orientation. The quality loss of downscaling is reduced by the use of Mipmaps but the end-result can never be as good as rendering directly in native resolution. However, we consider the differences to be so small that a side-by-side comparison is needed to spot them, see figure 5.6.

![Figure 5.6: Comparing two text labels rendered by MapDemo (left) and our implementation (right).](image)

5.6.2 Performance

The main goal of this thesis was to investigate technologies that utilized the GPU in order to optimize the rendering performance of an existing CPU-based application. A performance comparison between our final implementation and the original application can be seen in figure 5.7.

As can be seen in the diagram, our final implementation performs substantially better than MapDemo on all benchmarking regions with an average rendering performance improvement of 4.7x. This is a very noticeable difference when using the product as our final implementation renders the map very fast for all regions.
Figure 5.7: Comparison between MapDemo and our final implementation.
Chapter 6

Discussion

6.1 Method

Many of the results presented in this thesis originate from a trial-and-error approach. By parallelizing the work and trying out different ways to achieve the same end-result, we have been able to see how theoretical results work in practice on an iOS device and compared them to each other in terms of quality and performance. We believe that this way of working has been beneficial for the final implementation, early and novel results have been discarded along the way. The area of focus has changed constantly throughout the thesis project as we did not predict many of the challenges we later encountered.

The work that we have done and what resulted in the final implementation is substantial and contains many different parts. It is hard to develop software of this magnitude using a traditional research/design/implement/evaluate approach and an iterative work cycle as proposed in the now industry famous Agile Manifesto, [15], has shown to lead to great results at IT-Bolaget Per & Per in the past. We have worked with continuous deliveries, customer meetings and tried to have working software as often as possible throughout the months that led up to the final implementation.

As with all academic projects, we have had to limit us to the time and resources available for our disposal. With more time, we would have wanted to investigate more approaches to mobile vector maps rendering, especially solutions not based on Apple Map Kit and the limitations it comes with. This is discussed in section 6.3.

Another area that we wish we had more time to work with is the text rendering aspect of the software. There is a lot of interesting research in this area and we have only been able to implement a small subset of what is
possible. As a result of this, the rendered text is the part of the final result that has the lowest quality in comparison to the old CPU-based rendering approach.

Our final implementation still depends on Mapnik for parsing and applying style rules as the style sheet used is written in a Mapnik specific format. This is a bit unfortunate as we decided to not use any other Mapnik functionality in the final software. In general, we do not think it is a good idea to depend on a large third-party library and only use a minor part of its sub-functionality because of legacy input data. It might have been a good idea to try to get rid of all Mapnik dependencies after we decided to focus on writing our own feature parser and renderer in order to maximize software flexibility.

6.2 Result

The final implementation presented in chapter 4 shows that the goals of this thesis have been met. The performance optimizations that we have outlined are substantial and the software is flexible enough to support the features that were used in the previous product while still being expandable.

It is worth mentioning that the positive performance improvements shown is the result of optimizing many smaller parts of the software as well as streamlining the data flow. Some of these are unique to the implemented GPU-based renderer but some, e.g. the concept of batch rendering, could be applied on the old solution without any major modifications. This makes it hard to assess the importance of OpenGL in the final result.

We have tried implementing batch rendering and other non-GPU specific performance improvements in MapDemo in order to only compare our OpenGL renderer against AGG. The benchmarking results were still heavily in our favor. However, this does not give a definite answer as we are comparing two different components in separate environments, dependent on different data flows, but taking these results into account together with the arguments outlined in this thesis, we have reasons to believe that an OpenGL approach to rendering vector maps is preferable.

6.2.1 Impact at IT-Bolaget Per & Per

Our implementation has the potential of being very important and act as a cornerstone in future mobile iOS applications by IT-Bolaget Per & Per.

The software has been implemented with future use in mind and is designed to be easily integrated into IT-Bolaget Per & Per’s mobile map framework.
As the solution is heavily based on OpenGL, a platform independent technology, we believe that many of the ideas presented could be implemented for other platforms as well, both mobile, desktop and web based.

6.3 Future studies

The set of technologies that we have looked into during this thesis is just a small portion of everything that is possible to use for rendering vector based maps on mobile devices. It is a fast-changing area with new ideas and products being presented regularly.

We advice IT-Bolaget Per & Per to keep their eyes open and continue to look into how vector maps can be rendered for maximal performance and flexibility using new tools and technologies. In this section, we give some starting points to consider in the future.

6.3.1 Better text handling

For the chosen text rendering implementation, described in section 4.3.3, where a texture atlas is rendered once during application launch, a simple improvement would be to pre-render the texture and instead bundle it with the application. This would reduce the time needed to start and initialize the application while achieving the same results.

More generally, handling text in a different way is something we think would be interesting to look into further. The ability to rotate the map is a common feature in many mapping applications, e.g. Google Maps and Apple Maps. Rotation is currently disabled in both MapDemo and our implementation even though Map Kit supports it. It is disabled mainly because we consider the ability to render text horizontally, irrespective of map rotation, to be a requirement for allowing such functionality. This is something that is not possible to achieve with the current approach as text is rendered into the bitmap tiles, just like in MapDemo. The text would rotate with the map if rotation was enabled.

As with rotation, text in our implementation scale with the map when zooming. This is possibly an unwanted behavior, one that many other mapping applications do not have.

Because of the concept of tiles and having the text rendered into the bitmap tiles directly, achieving rotation or scale independent text would require each tile to be re-rendered continuously as the user rotates or zooms the map. This would be very computationally heavy and thus not a feasible option.
In order to allow for rotation and scale independent text rendering, text would need to be handled separately from the tiles. As described in section 3.3, Map Kit provide a way to achieve this with annotations. However, using annotations for the amount of text that can occur in a high detail map did perform very poorly during testing. The main problem is that Map Kit’s annotations API is not designed to handle frequent additions and removals of large amounts of annotations as required when zooming and panning the map.

Because of this, we believe that the above mentioned text behavior is not possible to achieve using Map Kit as the underlaying map framework. A possible future solution would be to develop a custom map framework that replaces Map Kit’s functionality in order to handle text and tiles separately.

### 6.3.2 Beyond the tile concept of Map Kit

All the work that we have done and all performance comparisons are based on the iOS core framework Map Kit and its tile overlays. There are several advantages of using Map Kit, e.g. the easiness of integrating it into an existing iOS app and making the user feel at home since many other iOS apps are based on the same framework.

We believe that the main benefit of using the concept of tiles and rendering vector data into bitmap images is that it is possible to render very detailed maps and still achieve high frame rates and good responsiveness when interacting with the map.

However, we cannot know if it is the best solution for IT-Bolaget Per & Per and their specific map data. It would have been interesting to look into the possibilities of implementing a renderer that was not tile-based. As vector map data has the advantage of being resolution independent, it would in theory be possible to create a solution that does not have the concept of zoom levels but instead provides dynamic panning and zooming. As both Apple and Google, two of the main developers of mobile maps applications, have shown that this kind of technology is possible in their mobile map applications, it probably is an interesting area for IT-Bolaget Per & Per to consider in the future.

### 6.3.3 OpenCL and other GPU programming frameworks

As we have seen, native GPU programming on iOS is currently limited to OpenGL even though, for example, OpenCL is supported by the hardware as shown by [11]. Unfortunately, using other technologies than OpenGL and taking full control of the hardware requires Jailbreaking the device which is
not an option for IT-Bolaget Per & Per that develops and sells commercial software to customers.

If Apple decides to make OpenCL or other GPU computing frameworks available for third-party developers to use in future versions of iOS, it would be something worth considering for processing the vector map data.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Jonas Bromö och Alexander Qvick Faxå