UNIVERSITY OF GOTHENBURG

Fast Map Rendering for Mobile Devices

Yauheniya Renhart

Master Thesis work in Software Engineering and Management

Report No. 2010:070
ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2009

Abstract

Despite rapid development of mobile technologies mobile graphics still has performance and quality problems, which are critical for map applications. A number of graphics APIs, libraries and optimization techniques exist nowadays. The research question of this work is what graphics solution to choose for mobile map applications for both Java ME and Android platforms. The research objectives are identification of important qualities of map rendering as well as graphics APIs, libraries and drawing techniques evaluation. Basing on APIs evaluation recommendations for graphics solutions are formulated. The method included investigation of APIs specifications and articles and a series of experiments conducted on several mobile phones. The findings from this research give evidence that standard graphics API for Java ME has the best performance for the majority of phones. On the other hand a newer Mobile 3D Graphics API is faster only for few hardware accelerated phones. The research demonstrated that OpenGL ES for Android can be more efficient than Custom 2D API but not in all cases. Thus it is recommended to use OpenGL ES API for map applications only under certain conditions, while Custom 2D API is always applicable. It is concluded from the research that the simplest and oldest solutions can still be the most efficient. However in some cases a better performance can be achieved with newer APIs that take advantage of hardware acceleration.

Preface

This master thesis research was conducted in Idevio AB company in Goteborg from February to June 2010. The research aim was to find efficient methods for map rendering on mobile devices, the results of which could be used to improve one of Idevio products – RaveGeo Map Client, its mobile version. The map client together with Open Street Map and TeleAtlas data was used extensively in the research as an example of map application.

I would like to thank Idevio staff who helped me with technical issues as well as with finding mobile devices to test on. Especially I want to thank Johan Persson, technical lead in Idevio, who provided competent guidance throughout the project. I would also like to thank Urban Nulden, my supervisor at IT University and the Head of the Department of Applied Information Technology, for his valuable advice and support during the study. Finally, I want to thank my fellow students who inspired me with ideas for this thesis work.

Contents

Abbreviations

2D	Two-dimensional
3D	Three- dimensional
API	Application programming interface
CPU	Central processing unit
GPU	Graphics processing unit
JAVA ME	Java Platform, Micro Edition
LCDUI	Limited Connected Device User Interface
M2G	Mobile 2D Graphics
M3G	Mobile 3D Graphics
OSM	Open Street Map
PDA	Personal Digital Assistant
SVG	Scalable Vector Graphics
VBO	Vertex Buffer Object
XML	eXtensible Markup Language

1 Introduction

1.1 Background

Mobile phones usage has grown rapidly in the last 10 years. The number of mobile cellular subscribers increased from 500 million in 2000 up to 4.6 billion in the end of 2009 according to International Telecommunication Union. The purpose of mobile phones has also changed – it is no longer just a device to make phone calls. With development of display technology showing complex graphics on mobile screen became feasible. Among other many new possibilities map navigation with mobile phone became a reality.

Nowadays mobile mapping services are used intensively. Google Maps for mobile is probably the most popular one, but there is huge amount of other mapping engines. It is possible to search for specific location, to calculate route, to find out your position, select the best restaurant nearby, check the bus timetable and as much as one can think of. Except of end-products a variety of mapping APIs exist for developers to easily create their own services. In this way the development grows further and mobile map navigation became a part of everyday life for many of us. However it is still not an exceptional case when after installing mobile map application the user finds out that it is unfortunately too slow. A number of reasons can cause performance problems of course, but mainly there can be two bottlenecks. One is geographical data retrieving and processing and the other is drawing the map on the screen. Both of them occur due to small mobile devices limitations, such as power supply, memory capacity and CPU computational power (Biuk-Aghai, 2005). The drawing map speed directly depends on graphics possibilities of mobile devices.

With display improvements mobile graphics also has greatly improved. It is long time since vector graphics is preferred to raster as more efficient and thus is used in most mobile map services. Mobile graphics APIs gave push to mobile graphics research and development. Together with rendering speed graphics quality is important concern for map applications. The quality of mobile graphics is of course limited by small display which will never become big enough. The great variety of phones with different display sizes and resolutions introduces another challenge for creating good quality graphics.

As we can see despite the rapid development of mobile graphics both performance and graphics quality problems still exist. Around 10 years ago there was not much choice for developer, there were not many ways for displaying graphical content. Today the situation can be described as opposite. Though the performance and quality problems are not fully solved, the number of existing APIs, graphical libraries and techniques provides a variety of solutions that developers and software designers are free to choose from. However it is often not an easy decision. Different APIs are better for different purpose. Which of them are faster, which produce smother graphics is not well-known. In addition the devices are not at all the same. Different graphics processing, computational capacity, screen resolution makes it difficult to come up with universal solution in terms of both performance and graphics quality. Capin, et. al., (2008) states that particular technique can be efficient for one device but inefficient for another. For this reason it is a real challenge to find universal solution and select the most appropriate API or technique from the wide variety. Hopkins (2007) describes the problem of choice as “API overload”. It can be extended to “graphics solution overload” with also graphical libraries and techniques. This problem comes up with development of map applications or any other graphics applications. Due to lack of objective analysis and evaluation of available solutions developers have to make critical decision without having proper information, without enough time to get it. Thus software applications can be of lower quality than they could with all the technology development, both software

and hardware. This unfortunate outcome can of course be avoided if technology is chosen more thoroughly, if all the required qualities are taken into consideration, if different approaches are analyzed in terms of these qualities and any third-party APIs or libraries are used as intended by authors.

1.2 Research Focus

Creating fast good quality mobile map applications requires, among other things, making correct decision when choosing graphics technology. Map drawing can become a major challenge in performance tuning of map application (Biuk-Aghai, 2005). That is why the choice of technology is critical for both performance and graphics quality. A kind of analysis of possible graphics solutions for mobile devices I expect to provide as a result of this research. Provided recommendations should help software developers and architects to make choice and select the best approach.

While these recommendations would probably be good for any mobile graphics applications this research is focused on map applications. Specific criteria targeted at map applications will be defined for graphics solutions evaluation. I will mainly focus on performance of graphics rendering, but will consider other qualities as well. The resulting quality criteria will also be useful for controlling quality of already existing map applications. It will help software quality managers to better focus their quality review on specific attributes as well as to know if they can be potentially improved. After defining quality criteria for evaluation it should be decided what to evaluate. This will require research on what graphics APIs, libraries and techniques exist today and how they can be applied for map rendering. It is intended to investigate different approach to speed up drawing, probably to try their combination, modification in order to improve the result. As a result of graphics displaying methods evaluation one or more is expected to be proposed for the best efficiency of map rendering.

By mobile devices not only mobile phones are meant, but also other devices such as e.g. PDA. While these more advanced devices are also used widely with map applications in this research I will focus on mobile phones. This research is intended to analyze problems caused by small device limitations and it is mainly the case with mobile phones. Moreover PDAs are not as wide-spread as phones. I am going to focus on graphics solution investigation for mobile phones with Java platform (Java ME) and Android mobile phones. Java is supported by almost every mobile phone nowadays, thus it seems reasonable to investigate that. Android popularity is also growing fast. In this way I pick the most common environment for mobile map applications and make the research as objective as possible.

1.3 Overall Research Aim and Objectives

The overall goal of this research is to find efficient methods for map rendering on mobile devices. Drawing performance will be considered as the most important issue, but other qualities will also be investigated. It is necessary to identify which of them are essential for map rendering and how good rates can be achieved. Searching efficient ways of map rendering will involve exploring and evaluating existing graphics mobile APIs, libraries and techniques. The evaluation will be performed by the means of APIs specifications and other articles review and testing. Finally one or more approaches for fast and good quality map rendering are expected to be recommended.

To summarize the above the research objectives follow:

- Identify important qualities of map rendering
- Evaluate available Java ME and Android graphics APIs and libraries by means of literature review and testing

- Investigate techniques that can speed up drawing
- Propose one or more efficient alternatives for map rendering on mobile devices

The listed objectives cannot be considered as independent, but as closely interrelated with each other. The first step or objective is to identify what to search for in the research, what is meant by “efficient” map rendering. Without this definition it is not exactly clear how to meet the second objective – evaluation of graphics APIs and libraries. The second objective gives understanding of currently available solutions, provides their evaluation based on quality attributes defined earlier. However third party APIs and libraries don't clarify the whole range of possible graphics solutions, don't exhaust all the methods of drawing speed and quality improvement. For this reason the third objective is needed which will reflect on other techniques and the possibilities that they provide. After getting a full comprehension of different methods of map rendering it will become possible to formulate recommendation. The last objective as a result of all prior is to propose the final one or more efficient alternatives for map rendering.

This research will contribute to the development of mobile graphics specifically for mobile map applications. While it will not provide a new drawing library or new drawing algorithms it will give an evaluation of currently used graphics solutions based on others opinions but also on self prepared tests. The focus on mobile map applications will help software engineers and quality managers working with mapping services to make correct technology decision, perform adequate quality control and as a result produce better products. The next chapter “Theoretical Background” will provide an overview of today’s trends in mobile graphics in general and rendering maps in particular being based on articles, specifications and other literature review.

2 Theoretical Background

Quick development of mobile graphics technology resulted in a number of available approaches that can be chosen for drawing graphics. Different APIs and libraries emerged both for Java ME and Android platforms. Various techniques are used to speed up drawing and to improve graphics quality. In this section the current situation in mobile graphics world will be analyzed. I will give an overview on how graphics performance problems are solved nowadays, what is graphics quality, what is important specifically for map applications. Criteria for evaluating graphics solutions for map applications will be proposed. A study of available APIs, libraries and drawing techniques will be included as well. This section will give understanding of what today's mobile graphics developers have to choose from when designing their applications.

2.1 Mobile Graphics Speed and Quality

When it is spoken about mobile graphics it should be always taken into account that mobile devices are not at all same with desktop computers and thus need different approach. It is especially true when concerning graphics performance and quality. Mobile devices have a number of limitations such as computational power, memory capacity which affects graphics performance. Small device screen and its resolution are of course strong limitations for graphics quality.

The problem of bounded CPU is partly overcome with GPU – separate processing unit for rendering graphics (Capin, et.al., 2008). The use of GPU is also called hardware acceleration and can improve graphics performance significantly (Pulli et. al., 2007; Akenine-Möller, Ström, 2008; Capin et.al., 2008). Hardware implementation is also efficient in terms of power consumption (Akenine-Möller, Ström, 2008), which is so important for mobile devices. To obtain performance advantage from hardware acceleration it is necessary to access 3D hardware. Several open standard APIs were introduced for that purpose. Unfortunately GPUs are still not applied in so many phones. What is worse is that different approaches should sometimes be applied depending on hardware acceleration. This can make multi-platform development harder. A number of special recommendations for better performance are provided by Imagination Technologies, GPU manufacturer (Beets, 2005). For example Beets explains that it is important to minimize the number of API calls minimizing this way CPU load created by API Overhead. By API Overhead he means communication between API and hardware accelerator overhead. Also it is described how CPU/GPU parallelism can be destroyed which affects performance badly. All this means that developing for hardware accelerated devices requires caution and understanding of how the APIs are intended to be used. The special approach will be discussed in more details in Research Method section.

It is not only drawing speed that matters, but graphics quality is essential as well. One of the important characteristics of graphics is how smooth it is. Any graphical shape consists of pixels, which can be though as small squares. If for example line angle is not horizontal or vertical then instead of smooth line a staircase of pixels is produces. By blending the foreground and background colors it is possible to make shapes look smoother. This technique is called *antialiasing* (Pulli et. al., 2007).

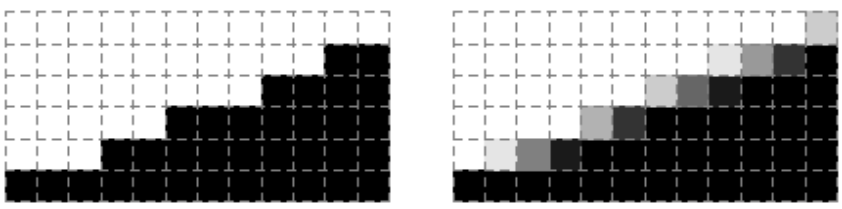

Figure 1: Line is approximated to a staircase of pixels. Then it is blended to look smoother (Pulli et. al., 2007).

Anti-aliasing can also be understood as increasing perceived resolution of the screen (Nokia, 2005). The technique seems to be useful especially for map applications, as it makes the map more readable. Unfortunately anti-aliasing algorithms are not simple and in most cases affect drawing speed. Moreover the technique implementation should take into account physical display characteristics. Thus most graphics APIs do not define an exact algorithm (Pulli et. al., 2007).

Transparency is another quality attribute for graphics. With the use of alpha channel except of red, green and blue it is possible to set graphics transparency level. Sometimes it can be really important to be able to draw not only fully opaque shapes. In the case of map applications it is useful for drawing different areas on top of each other.

Colors, of course, is something that gets user's attention as soon as he or she looks at the graphics. Most responsibility for colors is on hardware, for example there is no way to draw colorful picture on monochromic display. However there is a software technique that can be used to improve color perceptions. It is called *dithering*. Dithering is a kind of noise or pattern which is used for displaying missing color as a composite of existing colors. In this way it increases display color depth (Pulli et. al., 2007). The image below demonstrates this effect.

Figure 2: Dithering (Pulli et. al., 2007)

Though useful around ten yeas ago these techniques are rather obsolete nowadays. The reason for that is development of color displays with more than 65000 colors, which is normally more than enough. If we look at these techniques in terms of map applications dithering may actually make the map readability worse, and true colors are not important in most cases. Another technique used to improve graphics quality is *fog*. It is used in 3D graphics for showing far-away objects, making them look more natural. For example far away mountains seem to be grayish or bluish. This technique may be useful for those map applications that display 3D view.

There is an obvious trade-off between graphics speed and its quality. Every quality improving technique requires additional computation. Thus it is important to know which quality attributes should be fulfilled and which can be ignored in the sake of performance.

2.2 Criteria for Choosing Map Applications Graphics Solution

Prior to analyzing available graphics solutions the focus of the analysis should be identified. The solutions performance is one important issue as it is also the goal of this research: to find fast map rendering methods. Portability is another essential quality to consider as graphics solution is intended for widely used map applications, the users of which have diverse phones. In connection with graphics quality for rendering maps anti-aliasing and transparency support will be taken into account. Finally, it also matters how much effort is required for implementation. The complexity of development may influence performance as well as graphics quality due to possible errors. That is why it is useful to know how easy it is to draw map with the chosen graphics solution. In the case of graphics APIs and libraries it depends on what features are supported. Specifically for map applications the following features would make development easier and probably enable better product production:

1. Support for thick lines

   Mobile phones used to have small screens, probably for this reason initial drawing APIs do not support lines that are wider than 1 pixel. In today's situation however this functionality is not sufficient. Drawing maps requires for example drawing roads which are thick lines. If the feature is not supported quite complex algorithm should be implemented for constructing a line from triangles or polygons.

2. Support for polygons

   Some APIs do not support drawing polygons, but only triangles. However polygons are really critical for drawing different areas on a map. Triangulation algorithms can be used to create polygon from triangles, but this of course requires additional effort.

3. Support for drawing text

   It is obvious that text has to be drawn on any map. With APIs which do not support this feature it may be quite complicated to draw text with just lines.

As a result of this discussion the following criteria were selected for graphics solutions analysis:

- Performance
- Portability
- Graphics quality: Anti-aliasing, Transparency
- Functionality: thick lines, polygons, text

2.3 Overview of Graphics APIs and Libraries

Since Java ME was introduced back in 2000 mobile graphics has been continuously developed. In the beginning there was nothing for developer to choose from but later on the variety of graphics APIs and libraries grew. Both 2D and 3D graphics became widely spread. Nowadays it requires some effort to decide which API is most suitable for specific task. Android platform does not support that many APIs and libraries, however for 2D graphics there is still a choice from two options. For most map applications 2D graphics is used. However 3D graphics can also be utilized for map rendering as long as it is not worse in speed and quality.

The image below (see Figure3) illustrates today's graphics APIs and libraries for Java ME and Android and their dependencies on one another. The image does not present all the APIs and libraries that exist today, but those that are easily available, popular today and will be investigated in this research. Those APIs that are for desktops or only for C/C++ are not of interest in this work but mentioned only as a base for Java ME and Android APIs. If to look only at Java ME and Android parts of the image, then each square in those parts is a graphics solution that will be evaluated in this research. If two squares are included in one this means that the solutions are not independent and can only be used together. Later on I will gradually go into details for each API / library.

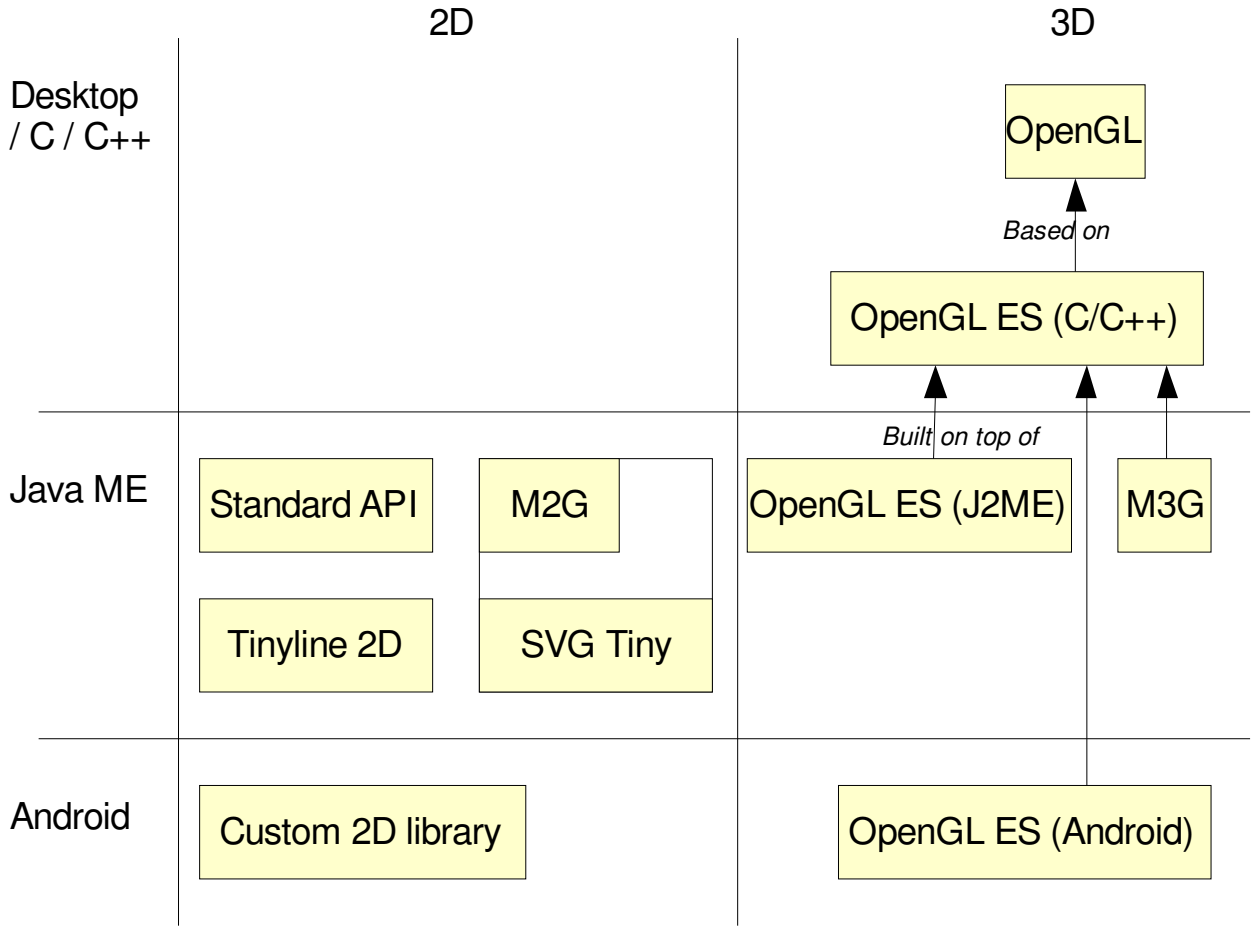

Figure 3: 2D and 3D graphics API for Java ME and Android mobile platforms.

I will start with an overview of Java ME 2D graphics APIs and libraries. Hopkins (2007) provides a brief description of Java ME APIs that existed in 2007. One of the simplest, oldest and probably still most commonly used is a standard Graphics API which is part of LCDUI package and provided with Java ME. It's details can be found in Java ME specification (Sun Microsystems, 2010).

For 2D rendering it is quite common to use Scalable Vector Graphics (SVG) or SVG Tiny (W3C, 2008) intended for mobile phones. It is a specification for XML-based file format, which is used to describe 2D graphics elements. As the name suggests the main idea of this format is that the graphics is scalable, which means that it is possible to zoom in and out without recreating graphical element. SVG has a number of other advantages (Hopkins, 2007) however in this research it only matters how it is utilized for graphics rendering performance and quality. XML specification by itself cannot provide any solution. This is where Scalable 2D Vector Graphics API comes in. The API is specified as JSR 226 in the Java Community Process (Eskelinen, 2005) and is also called Mobile 2D Graphics (M2G). Being based on SVG this API provides capabilities of rendering SVG graphics. Powers (2005) demonstrates a simple usage of the API with both loading predefined graphics and creating new. As the diagram at Figure 3 suggests M2G API and SVG Tiny specification make up one 2D graphics solution for Java ME.

With development of mobile graphics standard APIs some drawing libraries were also developed. One of them is a commercial Tinyline 2D library for Java ME (Girow, 2010). It has much reacher

functionality in comparison with mentioned above 2D Java ME APIs. In short three options for mobile 2D graphics for Java ME are presented in Figure 3:

- Standard Graphics API in LCDUI package: old and simple
- M2G with SVG Tiny: scalable graphics
- Tinyline 2D: commercial library with reach functionality

Android platform has only one common 2D - only API: a custom 2D graphics library based on Google's Skia open source graphics engine (Skia 2D Graphics Library, 2010). The details of the API can be found in Android specification (Android APIs, 2010).

Now an overview of 3D graphics APIs and libraries will follow. OpenGL is probably the most common standard ever for 3D graphics(Khronos Group, 2010). It is not intended for mobile devices however. OpenGL ES API on the other hand is a compact version for OpenGL adopted for embedded systems. The design goal of this API was to minimize the use of resources which are limited on mobile devices (Pulli, 2006). OpenGL ES is intended for mobile devices but it is still not intended for Java applications.

Mobile 3D Graphics API (M3G) was the first 3D API for Java ME (Quasau, 2004). It is built on top of OpenGL ES API, being though more high-level with just a subset of features of OpenGL ES. It is indeed very popular nowadays. Already in 2006 over 100 handset models had M3G support (Pulli, 2006). Today most of newer phones include it. The specification for Mobile 3D Graphics is defined in JSR 184 (Aarnio, 2003).

Later OpenGL ES became also available for Java. Java binding for OpenGL ES is better aligned with OpenGL ES than M3G providing its full functionality. Java binding for OpenGL ES API is specified as JSR 239 (Riggs, 2006) and unfortunately is much less spread than M3G or SVG. For this reason I will not investigate it in this research taking also into account that it cannot be significantly faster than M3G as they have the same base.

Another OpenGL ES extension is an Android version built on top of OpenGL ES. Only OpenGL ES 1.0 version is currently supported by Android. Thus OpenGL ES 1.0 specification can be used to get more details (Blythe, 2004). The API is supported by all Android phones and is the main tool for Android 3D graphics. See figure 3 for whole OpenGL based hierarchy of APIs.

This was a general overview of the APIs and libraries, the details will be provided in subsequent sections. I will focus on performance, which is the main concern, but also graphics quality, feature set and portability of each API.

2.4 Graphics APIs for Java ME

As it was described above several graphics APIs were developed for Java ME. This section will include detailed investigation of 2D APIs, Standard API and Mobile 2D Graphics (M2G), and Mobile 3D Graphics API.

2.4.1 2D APIs

The main advantage of Standard Graphics API is that it is 100% portable. Being part of a standard package it is supported by every phone that supports Java ME. It existed since beginning of Java ME back in 2000 and was the only choice for developers of that time (Hopkins, 2007). However it can look different on different devices depending on screen size or other characteristics. It has rather limited

functionality, which was probably the main reason for new APIs to emerge. For example it doesn't support thick lines or polygons making it necessary to implement complex algorithms of polygon triangulation and drawing lines with triangles. On the other hand it supports drawing text which is so important for drawing maps. The API includes no quality improving features, such as anti-aliasing, transparency is only supported for immutable images. It uses simple drawing model and probably is not very well optimized.

Another 2D API, JSR 226 or Scalable 2D Vector Graphics API is widely spread these days, especially for SVG applications. The API is rather simple and is closely aligned with SVG Tiny specification. Its primary goal was to provide possibility to manipulate SVG content, not a full drawing toolkit (Powers, 2005). Even though this research has not so much to do with SVG – it is focused not on how graphics is stored but on how it is rendered – it is still worth to look at the SVG rendering API. Though it usually is not fast to parse XML document, the graphics rendering part could be good in performance or quality or both.

Nokia expert group, that was working on defining the API, describes advantages of scalable graphics as being adaptable to different devices, easily zoomed without loose of quality, efficient due to use of gzip compression and some others (Nokia, 2006). However nothing is said about rendering performance. Graphics quality is dependent on API implementation. As for API functionality even though it is not intended to be a drawing library it has quite a variety of features from SVG Tiny specification. Polygons, thick lines, ellipses, drawing text are supported. Transparency however is only supported for images and not graphical elements.

As for performance of the API I will have to try it out and see.

2.4.2 Mobile 3D Graphics API (M3G)

The first 3D API for Java appeared with high demand for 3D graphics. It became very popular and is supported by many phones. The good performance was one of the important issues of its design. First of all it is based on well optimized C/C++ API, meaning that Java functions call the correspondent functions written in native code. Pulli, one of the co-authors of M3G specification, and others explain that the decision to use native implementation was made due to efficiency considerations (2007). Java has always been slower than C/C++ languages. Another performance advantage of M3G is that it can make use of 3D graphics hardware acceleration just as OpenGL ES. To minimize Java to native code to hardware communication and perform as much as possible in native code the *retained* mode was introduced. It is opposite to *immediate* mode, which can also be called direct, meaning that each graphical object is drawn separately. This approach is chosen in OpenGL ES. But for Mobile 3D graphics a more efficient retained mode is recommended (Pulli et.al., 2007). In this mode first the scene of objects is prepared and then rendered. In this way there is far less API communication overhead and techniques for faster drawing provided by hardware can be utilized. All this performance enhancements give hope for this API to be really useful. However it should be kept in mind that although the API itself is supported by so many phones not all of them also have hardware acceleration. As stated in Nokia guide (2006) the API “is targeted at devices that typically have very little processing power and memory, and no hardware support for 3D graphics.” This gives another hope that hardware support is not that important and probably software acceleration improves performance as well. But still there is no good evidence for it. Moreover, map applications usually require only 2D graphics and the efficiency of using complicated 3D API for 2D graphics is not clear. All this makes it obvious that reviewing articles and guides is not enough to conclude about M3G performance.

Mobile 3D Graphics API is also quite complex with many possibilities for performance enhancements. That is why it is important to use the API correctly as recommended for better optimization. As it was already mentioned retained mode should be preferred to immediate. Synchronization between 2D and 3 D graphics can cause efficiency problems so it is better to have a pure 3D application (Pulli, et.al., 2007). Some of the methods for optimizing 3D hardware accelerated graphics were already described above (Beets, 2005). As M3G uses hardware acceleration proposed methods can and probably should also be used with it.

As for graphics quality some contradiction can be noticed. On the one hand the API includes quality improving features such as anti-aliasing, dithering and full color. On the other hand it has very limited functionality for actual drawing. Basically only triangles are supported, not polygons or even lines or text is possible to draw. Everything consists from *triangle strips*, which are groups of triangles with common borders drawn one by one. This makes it difficult to implement simple things, but also the quality may go down depending on implementation. For example 1px. wide line may look not that nice if constructed from triangles. Anti-aliasing would probably solve the problem but it is supported on very few phones, at least was like that three years ago (Pulli et.al., 2007). Dithering and full color functionality is said not to be useful any longer with new displays. Nowadays dithering usually takes place automatically and true color is used in most implementations regardless if is turned on or not.

2.5 Android Graphics APIs

For Android platform there are two standard graphics APIs available: a custom 2D graphics library and OpenGL ES API for both 2D and 3D graphics (Graphics, 2010). Unlike the case with Java ME phones both these standards are supported by all android phones. Moreover when Android appeared mobile technology had already developed enough for sophisticated 3D games and other graphics applications. That is why both APIs have a much higher range of features than for example Standard Graphics API for Java ME.

A custom 2D graphics library is based on Google's Skia open source graphics engine (Skia 2D Graphics Library, 2010). It supports the features that are useful for map rendering such as thick lines, polygons, drawing text. Anti-aliasing technique and transparency are also included in the API. As for performance it is not possible to say anything without actual testing. What is known is that it does not take advantage of hardware acceleration. However the use of OpenGL ES (see below) was implemented as an experimental feature. By setting OpenGL ES context all drawing calls can be redirect to OpenGL ES (Android APIs, 2010). This means that performance may be close or same to OpenGL ES performance, which uses hardware acceleration. But as I mentioned it is only an experimental feature, not yet supported and thus it will not be investigated in this work.

Android's OpenGL ES API is similar to Mobile 3D Graphics for Java ME in that they are both built on top of OpenGL ES for C/C++. Most Android phones currently support only OpenGL ES 1.0 version, the details for which can be found in its specification (Blythe, 2004). Even though OpenGL ES supports more functionality than M3G it was also intended to make the API as compact as possible (Pulli et. al., 2007). From performance point of view the API supports fixed-point arithmetics, which can make the calls more efficient, as well as hardware acceleration is used if it is available (Pulli et. al., 2007). The good news is that unlike with Java ME most Android phones support hardware acceleration. Basing on smart phone processor guide (Available at: http://www.techautos.com/2010/03/14/smartphone-processor-guide) and the list of all Android phones (Available at: http://www.androphones.com/all-android-phones.php) where the processor is specified

for each phone it was possible to conclude that at least 19 out of 25 phones have GPU. Moreover in OpenGL ES API it is possible to find out programmatically if the phone supports hardware acceleration. This gives opportunity to select efficient solution for each phone individually. Another performance optimization was introduced in OpenGL ES 1.1 – Vertex Buffer Objects (VBO). Pulli et. al., (2007) describes the feature as moving away from pure immediate mode. VBOs allow to store the data in high-performance memory and thus to reduce copying data from CPU to GPU (Vertex Buffer Objects, 2010). Unfortunately VBOs are only supported by OpenGL ES 1.1 version which is not available in all Android phones.

As it is still unclear how much the performance depends on hardware acceleration and if it is better than Custom 2D Graphics library certain testing has to be performed. If OpenGL ES with hardware acceleration really improves performance than it is possible to get efficient map rendering on most Android phones. Same as for M3G special concerns for hardware accelerated development should be taken into account described by Beets (2005).

As for functionality of OpenGL ES API it is better than M3G, but unfortunately worse than Custom 2D graphics library. Anti-aliasing is supported but not by each implementation, which means that not every phone will display anti-aliased graphics. Transparency on the other hand is fully supported. Same as M3G the API is based on triangle strips. All polygons must be built from triangles. However lines and even thick lines are supported which can make both implementation easier and graphics quality better. Drawing text same as for all OpenGL ES implementations is not supported. The details of APIs functionality were taken from its specification (Blythe, 2004).

2.6 Graphics Libraries

Simultaneously with standard APIs non-standard software only mobile graphics libraries were developed. Their developers tried to solve the problems of lack of functionality, performance, portability and graphics quality. One of such libraries is JGL – Graphics 3D library for Java (Bing-Yu Chen, 2006). It was created as software implementation of OpenGL ES API. The library's developers, Bing-Yu Chen and Cheng-Han Tu (2005), explain that the main purpose was portability as the library does not require device's support of the API. Unfortunately for some reasons this library has not been updated since 2006 (Bing-Yu Chen, 2006). Moreover according to its authors' article it has some unresolved performance issues (Cheng-Han Tu, Bing-Yu Chen, 2005). For this reason I will not investigate it further.

Another, longer living and probably more successful, library is commercial TinyLine 2D (Girow, 2010). It was first created in 2002 and continues to be updated. TinyLine SVG, another version of the library is a common solution for map applications that use SVG graphics. Powers (2005) mentions the library as Mobile 2D Graphics implementation. Dong Li and others (2007) use it in their research on mobile SVG map service. Harun (2009) recommends using TinyLine as a high performance library. From the first sight at the library's API the most attractive is the library's functionality. It is 2D and thus is quite easy to understand. As it was already mentioned 3D graphics is not needed by most map applications. The library supports drawing thick lines, polygons and text in different directions. Some additional features like different styles for lines can be found useful as well. Graphics quality can be improved with the help of anti-aliasing. Transparency is another useful function of the library. The question which is left opened is drawing performance, although the author states that "it is written in optimized Java" (Girow, 2010). Additionally it can be noted that fixed-point number mathematics is used which should influence performance in a good way. On the other hand on TinyLine's forum some optimization tips is given by administrator (Girow, 2010). Most of them propose to avoid using some of

library's functionality, e.g. transparency and thick lines. This means that efficiency problems exist and additional investigation is needed.

2.7 Rendering Optimization Techniques

Until now I described some graphics APIs and libraries focusing among other things on their performance. But where does the good performance come from? Is that possible to improve it without using any of third-party APIs? To answer these questions I will investigate and analyze rendering optimization techniques. I will also see which of the techniques are used by which standard APIs and libraries.

If to look at graphics rendering process it appears that one of time consuming stages that require a lot of calculations is rasterization stage. This is when the application should determine which pixels should be painted in which colors (Pulli et. al., 2007). It is not a primitive process and thus has to be optimized for better performance. Valdin (2006) addresses rasterization stage optimization as *algorithmic* optimization. He presents briefly a couple of high-speed algorithms for graphics drawing. Among them are Bresenham's Algorithm for line drawing and an even faster Bresenham's Run-Length Slice Line-Drawing Algorithm. Unfortunately the information of the algorithms used is usually not given by graphics APIs creators.

While algorithms above attempt to draw primitives in shorter time other techniques like *culling* and *clipping* enable graphics engine not to draw certain shapes or their parts at all. This is possible due to the fact that not all primitives are visible at all times, some of them may be hidden behind others, some parts may be out of canvas area. *Back-face culling* technique is used for 3D rendering and is based on the assumption that back side of a real solid object is not visible (Pulli et. al., 2007). Capin (2008) explains *culling* as "selecting from a group" meaning selecting those objects that are not hidden by others. He presents several algorithms on how culling can be implemented with GPU. Avoiding drawing graphics outside the screen is called *clipping*. The easiest way to implement it is to only draw the whole figure but only if any of its points belong to canvas area. However, it is more efficient to draw only that part of the figure that is visible. A number of algorithm exist that can optimize clipping, such as Cohen-Sutherland Line-Clipping Algorithm and Sutherland-Hodgman's Polygon-Clipping Algorithm (Valdin, 2006). Culling and clipping techniques are used by OpenGL (and thus OpenGL ES) and most Mobile 3D Graphics API implementations (Pulli et. al., 2007).

Culling technique however is not feasible in an *immediate mode*. Immediate mode means that the objects are displayed right after the drawing function is called. Therefore it is obvious that it is not possible to optimize by not drawing an object which has already been drawn. *Retained mode* is a technique that was first introduced to improve performance of original OpenGL (Pulli et. al., 2007). In this mode the graphical objects are first stored in a scene graph and then rendered altogether. This mode reduces communication between APIs and hardware, which is especially crucial for hardware accelerated devices. Additionally it is giving the library more control over drawing giving it an opportunity for optimization, e.g. applying culling. Mobile 3D Graphics was designed to be a retained-mode API from the beginning. But it was not the same with OpenGL, which was immediate-mode only from start (Pulli et. al., 2007). It was using *display lists* – another optimization technique. Display list is just a group of compiled function calls stored for subsequent execution (Khronos Group, 2010). While display lists improve graphics performance compared with immediate-mode they don't give possibility to optimize drawing. Moreover they were found inflexible from the application point of view and thus removed in OpenGL ES (Pulli et. al., 2007). However, as it was already mentioned, another technique was introduced in OpenGL ES 1.1 – *Vertex Buffer Objects* (Vertex Buffer Objects, 2010). Using them is

not the same as display lists or retained mode – each API call is still executed separately. However it may reduce significantly the data transferring between CPU and GPU. The vertex and index data for a shape is copied once to GPU and can be reused later on. This technique seems to make sense only if the same object is displayed in different frames, although it is still possible to use VBOs the contents of which changes with every rendering. According to Khronos Group (2010) VBOs can give a real performance boost for 3D graphics applications.

One more way for graphics rendering optimization is using *fixed-point* arithmetic instead of *float-point*. The idea behind this technique is that integer operations are run much faster than float operations. It is possible to emulate float operations with integer numbers by moving decimal point 4 or 5 steps to the right. For example store 3,14159265 as 314159 (Pulli et. al., 2007). Some accuracy will be lost but performance will be gained. This technique is especially popular with 2D graphics libraries. For example TinyLine 2D is using it (Girow, 2010). For 3D libraries however the precision of fixed-point mathematics may be not always enough. This is one of the reasons that M3G and OpenGL are floating-point APIs. However until recently most phones didn't support floating point operations and thus OpenGL ES supports profiles for both fixed and float point arithmetic (Pulli, 2006). Under normal circumstances fixed-point arithmetic can improve performance greatly, though it is not always the case. Beets (2005) describes a Floating Point co-processor such as the ARM Vector Floating Point (VFP) unit which is used nowadays by some devices. Beets (2005) also presents the results from performance testing with the use of floating and fixed point arithmetic. Some of his results are displayed in a Table 1 below.

Algorithm	Floating Point Unit	Frames per second
Floating Point	Software	72
Fixed Point	Software	304
Floating Point	Hardware VFP	415
Optimized Algorithm Floating Point	Hardware VFP	>1000

Table 1: Beets (2005) performance tests results for fixed-point and floating-point arithmetic.

An obvious conclusion can be made from the test results. Fixed-point arithmetic is really faster than floating point arithmetic for software only implementations. However processor with floating point unit does better with floating points, improving performance dramatically with also optimized algorithm. It can be summarized that using floating point can increase accuracy and performance for limited hardware. This solution is not very portable and fixed-point arithmetic is thus more preferable.

The techniques above are very low-level and can mostly be applied at rasterization stage. Unfortunately this makes it not feasible to test and investigate them further in this research. Applying them would require developing a new drawing library which is out of scope of this work. On the other hand some higher level graphics optimization ways can be applied.

If the graphics library does not provide enough functionality, such as drawing polygons, thick lines or text then it is up to developer to implement necessary algorithms for performing these functions. The algorithms used for that influence drawing performance. *Double buffering* is a high-level technique used for graphics rendering on the screen. Double buffering means that the graphics is first rendered into a back buffer, then notifies the system when it is complete and only after that it is rendered on the screen (Pulli et. al., 2007). This technique helps to avoid flickering and gives user perception of a better performance. However it does not truly improve the drawing speed or graphics quality discussed in this work. Another high-level technique is *caching data*. If the same frame needs to be redrawn then there is

no need to read and process the data second time. The data structures for drawing can be prepared once and used afterwards. The drawback of this technique is that for many applications the scene is changed often and is not redrawn. In this case new data needs to be prepared on every rendering.

2.8 Conclusion

As a result of literature review section I can make some conclusions about available graphics solutions. Basing on the needs of most map rendering applications criteria for APIs evaluation were chosen. Below is a table (Table 2) that summarizes the Java ME graphics APIs, libraries and their important characteristics.

Characteristic	Standard Graphics API	SVG	Mobile 3D Graphics API	TinyLine 2D library
Performance	Not known to be optimized	Unknown	Known to be optimized in software and hardware	Uknown, software only optimization
Portability	All Java ME phones	Most Java ME phones	Most Java ME phones	All Java ME phones
Anti-aliasing	-	Implementation dependent	Implementation dependent	+
Transparency	-	-	+	+
Thick Lines	-	+	-	+
Polygons	-	+	-	+
Text	+	+	-	+

Table 2: Java ME Graphics APIs' characteristics

As we can see Mobile 3D Graphics API seems to have the best potential for performence. On the other hand it does not support the functionality needed. TinyLine 2D is good in all the criteria except that it is uknown for performance.

The same comparison was performed for Android APIs. It is summorized in Table 3 below.

Characteristic	Custom 2D Graphics API	OpenGL ES 1.0
Performance	Software only optimized	Known to be optimized in software and hardware
Portability	All Android phones	All Android phones
Antialiasing	+	Implementation dependent
Transparency	+	+
Thick Lines	+	+
Polygons	+	-
Text	+	-

Table 3: Android Graphics APIs' characteristics

While OpenGL ES seems to have good performance it does not support some useful for map rendering features. On the contrary Custom 2D Graphics API satisfies all functionality needs, but it is unknown about its performance. Hopefully the performance testing that will follow later will clarify this issue.

Along with available APIs different rendering optimization techniques were analyzed. The techniques themselves are very low-level and thus not feasible to further analyze in the scope of this thesis work. However it is interesting to see which of the techniques are used by which mentioned earlier APIs. The table 4 shows the correspondence between three rendering techniques and most of selected for further analysis APIs and libraries. It also includes information if the API can take advantage of hardware acceleration.

Technique	Standard Graphics API	Mobile 3D Graphics API	TinyLine 2D library	Android Custom 2D library	Android OpenGL ES
Culling	-	+	-	-	Face-culling only
Retained Mode	-	+	-	-	-
Vertex Buffer Objects	-	-	-	-	+ (only for 1.1 version)
Fixed-Point or Integer Arithmetic	+ (Integer only)	-	+ (Fixed-Point)	-	+ (Fixed-Point)
Hardware Acceleration	-	+	-	-	+

Table 4: Drawing optimization techniques used by mobile graphics APIs and libraries

In terms of using optimization techniques Mobile 3D Graphics API seems to be most optimized together with OpenGL ES for Android. On the other hand Standard Graphics API and TinyLine 2D library utilize integer only arithmetic which is known to be around 4 times faster than float arithmetic (Beets, 2005) and is also more portable if compared for example with hardware acceleration. As it is not possible to get true performance results of the APIs and their techniques from the literature further investigation in the form of testing is required. The next section "Research Method" will describe what kinds of tests will be run and how the results will be analysed.

3 Research Method

Graphics APIs and libraries will be evaluated in terms of both drawing speed and rendering quality. Other qualities like portability and functionality will also be considered. As for graphics performance testing is used as the main research method along with literature review. An approach to drawing speed testing as well as its results analysis will be described below.

3.1 Drawing Performance Evaluation

By measuring time of performing graphics operations we can evaluate and compare graphics APIs performance. Tests or benchmarks as they are also called can be *synthetic* and *application*. Synthetic tests measure speed of drawing of one or a couple of graphics components. It can be a line, a polygon or a group of shapes. The term "synthetic" is opposite to real, meaning that the graphics produced in synthetic tests is never used in real applications. These graphics is only created for testing purpose. The application tests, on the contrary, run realistic programs and measure time of drawing more complex scenes (Valdin, 2006). For example it can be a game scene or a map as in my case.

Synthetic tests have a number of advantages. First of all they are much easier to implement than application tests. As they are less complex it is easier to control the application flow and find out what causes performance problems if any. Moreover with synthetic testing it is possible to figure out ways to optimize drawing, for example displaying same shape but with different settings provided by API. Due to these reasons I will start with synthetic tests in this research. By running synthetic tests first I can find out early which APIs are worth further investigation. On the other hand synthetic tests will not provide the real drawing time in real applications.

After synthetic testing it is good to make some application tests to see how the APIs will perform in real software. To implement them a map client of Idevio (Idevio AB, 2010) will be used with necessary modifications needed for specific APIs usage. By drawing same area of the map and measuring time it is possible see which APIs are better in terms of performance in a real mapping application.

3.2 Testing Framework

Before running any tests, synthetic or application, it needs to be defined how to run them, what exactly to measure, where to write the result etc. First of all it is important to run tests on actual devices and not any kind of emulators. The rendering speed is strongly dependent on the device, for example as for Mobile 3D Graphics the performance difference can be really high if graphics 3D hardware acceleration is supported. Quite a lot of tests have to be run on several mobile devices. The devices may be not so easily accessible. Thus it is also important to make the tests run easily and fast. For this reason it is not a good idea to have each test as separate program, but rather a kind of testing framework is needed. In addition to already mentioned requirements the tests must be fair, meaning that they should measure what is expected and give adequate result. Every test must run under similar conditions, so that the results are only test-dependent. Otherwise they cannot be considered as correct. It should also be easy to add new tests as it is not possible to define all the tests needed in the beginning. Similarly new APIs, libraries or techniques may be discovered, so it should not be a problem to test them with the same framework.

Finally I came up with the following requirements for testing framework:

- should be possible to run tests on mobile devices and get results for each device
- it must be easy to run the tests
- it must be easy to add new tests or APIs / libraries
- tests must be fair or correct

As the tests have to be run on mobile devices I looked at well-known JMUnit framework (Silva, 2008). It gives the possibility to run a bunch of tests on a mobile device and see the results on the screen. Apart from that it supports special kind of tests for performance testing. This seemed quite appropriate for me. However like other xUnit frameworks it only shows which tests passed and which failed. Even performance tests can only show if the processing time exceeded previously defined limit or not. In my case, however, I need exact time spent for operations to be able to compare them afterwards. So I decided to create myself a small testing framework specifically for my needs.

As I expect to test graphics performance for both J2ME and Android platforms two versions of testing framework is implemented so that it can be run on both platforms. J2ME version can be run on any java enabled mobile phone while Android version should work for any Android phone. With polymorphism and abstraction I managed to make it easy to add new tests or APIs. The logic of what to draw (tests) and how to draw (drawing implementations) is well separated. That is why it is easy to add new tests or new APIs / libraries and switch between them.

The results of the tests are presented on the device screen. It is possible to run all kinds of test / API combinations at a time however due to memory issues tests can be run under unequal conditions. Thus it is better to create a separate distribution (jad, jar, apk files) for each combination that will be run separately on a mobile device. To exclude some occasional incorrect results each test is run more than once, the number is adjustable. The displayed results contain average, maximum and minimum times of the tests as well as standard deviation. The standard deviation provides information on how tests results differ from each other. Knowing that makes it possible to verify tests correctness.

Same testing framework can be used to test different graphics APIs, libraries as well as some optimization techniques. It will be easy to see if one method or another helps to improve rendering performance or not. The results of the tests are expected to be easy to analyze as they are very straight forward: time for each operation will be measured in milliseconds. By comparing the numbers I will be able to define assume which graphics solution has better performance. In case of unsatisfactory results some investigation will still be needed, such as: if I use the API correctly, if the tests are appropriate, would that work better on other phones or under other conditions. The strategy for tests results analysis will be described in more details in the subsequent sections. Selecting appropriate set of tests to run is another issue that has to be considered thoroughly. The next section will give details of what tests I will run and for which graphics APIs / libraries.

3.3 Synthetic tests

Running synthetic tests is not the same as testing a software product. The main difference is probably that there are no clear indicators for results analysis. These tests will not fail or pass but will produce necessary data for comparison of different graphics solutions. Another important difference is that the tests are *synthetic*, the situations reproduced in them will never happen in real applications. However despite these differences some common practices of test case design can still be applied. While expected result is not clearly defined the strategy for its analysis should be specified. The result for

every synthetic test is presented as a number of milliseconds which indicates how much time it took to perform test case operations. As the goal of testing is to compare performance of different solutions the results analysis is quite straight-forward and requires comparison of time. However some time limit has to be selected which will be considered as inappropriate time. If the test result gets up to this limit this would mean that additional tests have to be run to reduce the chance of API misuse or inadequate test cases. This limit will be assumed as *200 milliseconds*. It was defined by running several tests in advance to see average result time. The details of what should be done if the test took longer will be defined for each test separately. A principle of selecting boundary values for test cases (Galin, 2004) will be partly applied in that the tests should be able of measuring the time of the "worst" case. Since the "worst" case is not known a rather complex shapes will be selected, which probably are not reproducible in real applications. Next I will provide the details of synthetic tests to run for different APIs and libraries.

According to above review of existing APIs and libraries the following were selected for further investigation:

- Standard Graphics API (J2ME)
- Mobile 2D Graphics API (M2G) (J2ME)
- Mobile 3D Graphics API (M3G) (J2ME)
- TinyLine 2D library (J2ME)
- Custom Graphics API (Android)
- OpenGL ES API (Android)

The chosen synthetic tests are intended to include some basic functionality, the features commonly used in map applications and those functions that are specific for particular API.
The following are tests I find important to run for all APIs or libraries:

- draw rectangle
  The most basic test, draws one simple shape.
  Result analysis: This test is really simple. That is why if the time exceeds previously defined limit of 200 milliseconds it can be concluded that something in the test is wrong. Probably the API is not fully supported by the phone or the phone has no free memory. The reasons will have to be investigated

- draw long polyline
  Shows how the API is capable of drawing a more complex shape. I can take 1000 points for example to see how quickly that big but single object is rendered. As explained above the selection of 1000 points is based on testing boundary values principle.
  Result analysis: If the test takes inappropriate time, more than 200 milliseconds, than additional tests with realistic line length will be performed. Of course it is difficult to define what realistic line length is for map applications in general. That is why in my case RaveGeo Map Client (Idevio AB, 2010) will be taken as an example. By logging out the operations that are performed while the map is rendered it will become possible to get an average line length. As for Mobile 3D Graphics and OpenGL ES APIs additional tests will be run based on specific

approach for hardware accelerated graphics (see 3.3.1 Hardware Accelerated Graphics Tests below).

- draw a group of objects
  One of the common use cases in map rendering is drawing several objects at a time, e.g. several areas. That is why it is useful to see how API is capable of drawing a number of objects. In this test 400 rectangles will be drawn on the screen. Same as for polyline this number is chosen according to boundary values principle, which in my case means to select a rather high number. This test is also good for testing Mobile 3D Graphics retained mode, which is its one of the main optimization features. The rectangles will be drawn in retained mode as well as one by one. This will let me know how retained mode influences the performance and how other APIs that don't have this mode are capable of drawing the same scene.
  Result analysis: If the time is more than 200 milliseconds an attempt of drawing less rectangles should be taken. As for Mobile 3D Graphics and OpenGL ES APIs additional tests will be run based on specific approach for hardware accelerated graphics (see 3.3.1 Hardware Accelerated Graphics Tests below).

The following tests are more specific for different APIs:

- draw thick polyline
  This test only applies to those APIs that support drawing thick polylines. As it was mentioned earlier it is an important feature for map rendering, as in most cases thick polylines appear very often on the map. The length of the line will be1000 points. The width will be defined as a close to average line width used in RaveGeo Map Client.
  Result analysis: The approach to result analysis will be same as for polyline test. Drawing a shorter more realistic line will be tested in the case of time limit exceeding.

- draw triangle strip
  The idea of this test comes from Mobile 3D Graphics API. As it was mentioned earlier it is fully based on triangle strips. That is why it is interesting to see if it is better than others in drawing them. I will also use rather complicated shape of 1000 points.
  Result analysis: In case of exceeding 200 milliseconds limit a more realistic size for a shape will be picked. As for some APIs the thick lines are constructed from triangle strips the realistic size of a triangle strip will be counted based on realistic line length. The triangle strip should be at least twice as long as the line it constructs. That is why the realistic size of a triangle strip will be considered twice as high as of a polyline.

The tests will be run both with and without anti-aliasing for Android APIs and only with anti-aliasing where it is supported for Java ME platform.

3.3.1 Testing Hardware Accelerated Graphics

Hardware accelerated graphics requires a special approach for developing. According to Beets (2005) there is a number of ways to create performance problems by misusing the APIs that rely on GPU. Among the chosen APIs two take advantage from hardware acceleration: Mobile 3D Graphics and OpenGL ES. That is why in case of performance problems with these APIs a number of specially designed tests will have to be run. This will verify that the API was used correctly or will reveal the cause of the problem. The list of the tests follows:

- draw previously created polyline
  Both M3G and OpenGL ES APIs first construct an object (polyline) and then render it. This test measures the speed of actual rendering while the object is created in advance. This will allow to see where the bottleneck is.
  Result analysis: If the time is considerably less than the original polyline test this will mean that the problem is not with rendering but with object creation and is probably in the test code.

- draw polyline with back-face culling
  Culling is an optimization technique which can be enabled or disabled. For drawing polyline it should be disabled as polyline is not a 3D object and some parts of it are back-face that should be visible. Enabling culling will make the test to draw only half of the line.
  Result analysis: The expected result for this test is twice quicker than original polyline test as only half of the line is drawn. However if the difference will be higher than two times than it can be concluded that there is some additional overhead when culling is disabled. In this case the line should be drawn in a different way so that back-face can be hidden.

- draw polyline with disabled depth buffer
  Depth buffer is also called z-buffer and defines a finite number of z-coordinates that can be assigned to 3D points (Pulli et. al., 2007). In 2D graphics however in some cases it is appropriate to draw everything at the same z-coordinate and thus depth buffer can be disabled.
  Result analysis: If the result is considerably better than of the original polyline test then it should be investigated how disabling depth buffer may influence real map applications.

- draw a group of objects close to each other
  Beets (2005) gives several recommendations on how to create well optimized hardware accelerated graphics. On of the recommendation is to keep objects close to each other or to sort them by screen locality. This is what will be performed in this tests: 400 rectangles instead of being submitted in random order will be submitted in the order of their coordinates.
  Result analysis: If the time for this test is considerably less than the time of the original polyline test than it should investigated how feasible it is to keep objects close in real map application and considered when comparing APIs.

- draw polyline with different number of calls
  This test may be especially useful for OpenGL ES as it does not have retained mode. By drawing the same line with different number of calls, for example 1000 points in one call, 200 points in 5 calls etc., it will be possible to see how significant the effect of the number of calls is. This will demonstrate if the lack of retained mode is an important drawback of OpenGL ES.
  Result analysis: If the rendering time does not depend on the number of calls this means that there will be no problem with too many API calls and thus retained mode is not needed.

- Check if hardware acceleration is used
  The test is only possible for OpenGL ES as it gives opportunity to get the name of graphics renderer. It provides information of mobile device's hardware acceleration. Even if the phone supports hardware acceleration it might for some reason not use it for particular program.
  Result analysis: If the name of the graphics renderer is "Android PixelFlinger 1.0" (the same as of phone emulator) then software only acceleration is used. In this case if the phone is known to

support hardware acceleration the problem should be investigated.

3.4 Application tests

As application tests require more effort for their implementation they will not be run for each API or library. Only several APIs will be selected basing on synthetic tests results. For implementing application tests RaveGeo Map Client (Idevio AB, 2010) will be used. It is an API which renders the map data. As probably most map applications it is based on layers which can be displayed altogether or separately. Four types of layers are supported: polygons layer, lines layer, symbols layer and labels layer. Each type of layer is capable of drawing one type of map elements: polygons, lines, images and text respectively. Depending on selected APIs / libraries for application testing one or more types of layers will be displayed in tests. This is due to limitations of different APIs. While it is possible, for example, to draw polygons or text with any API it might be complex to implement it with those APIs that don't support the feature.

In the RaveGeo Map Client all drawing functionality is encapsulated in one class which makes it easy to modify the drawing algorithms. On the other hand there is some complexity with measuring the performance time as the map is drawn in different threads while the data is asynchronously loading. To make the measuring possible and correct the loading / drawing algorithm was changed for the tests. The data is first loaded and the loading speed does not affect the test results. After loading the map is rendered in one thread several times, same as with synthetic tests. Finally, the results appear on the screen. The average, the minimum, the maximum time in milliseconds and the standard deviation are displayed over the map. In this way it is possible to see both the quality of the rendered graphics and the performance.

In the application tests more than one map database will be used. The databases differ in the number of geographical objects, their type and size, which in turn cause differences in the drawing calls that are performed for rendering. For this reason it considered to be important to run the tests with at least two different databases to get adequate results. Open Street Map and TeleAtlas map data will be used for the testing.

Unlike in synthetic tests in application tests it is not easy to know which operations are actually performed and how many times. Simple logging will be used to get the details of performed calls. This will help to figure out where the performance bottleneck is and try to optimize the drawing algorithm.

3.5 Application level optimization

Running application tests is expected to show how an API / library is capable of drawing a real map. First simple tests will be run, where each drawing operation calls the API. However these tests might not be enough to make final conclusions. Additional optimization attempts will be performed and tested. This should be a high-level optimization which is possible to make in the scope of this work on top of selected APIs / libraries.

For any kind of optimization it is important to know where the problem lies. Pulli, et. al., (2007) proposed a method to define what the drawing speed is limited by. It requires several tests to be run. The method is general for 3D applications and includes working with lights, texturing and blending. It also takes into account those applications that redraw the frame often, e.g. games. However in my application tests none of lights, texturing, blending or animation will be used, thus these techniques cannot be the reason for performance problems. Figure 4 shows a simplified tuning method based on

the one that Pulli, et.al., (2007) proposed, which will be used in this work.

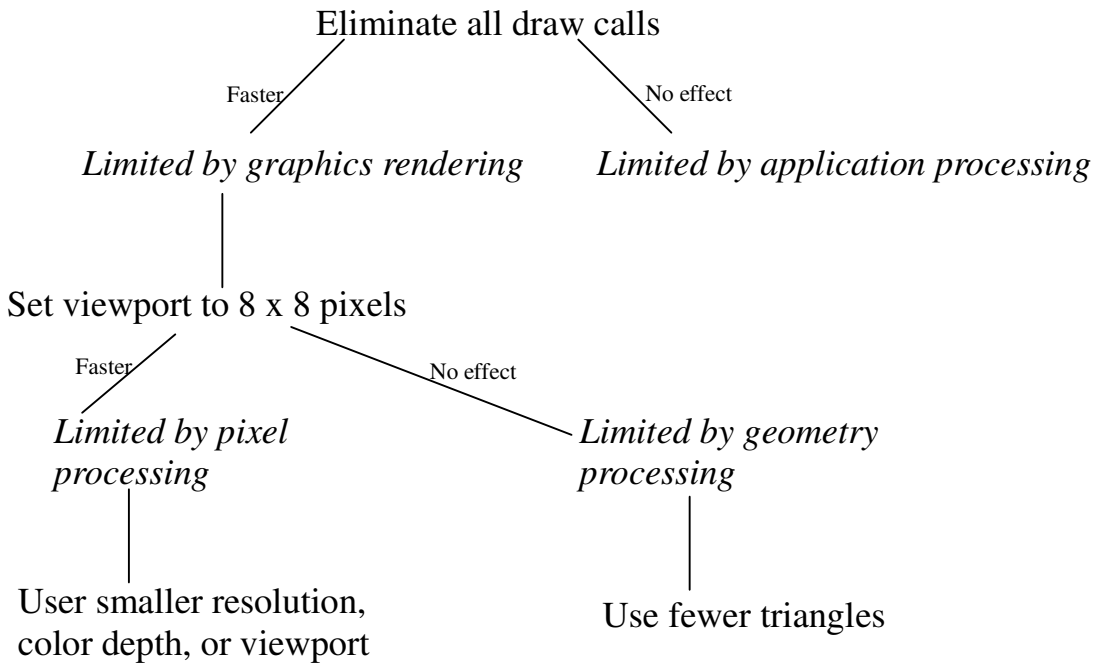

Figure 4: OpenGL ES performance tuning method. Based on the method proposed by Pulli, et. al., (2007).

In this method the first step is to define if the performance problems are at all due to graphics processing. Unlike in synthetic tests in application tests that will be run not only drawing operations are performed but also other application logic. After that setting a very small viewport (or screen) demonstrates if the problem is with pixel processing or geometry processing. Once I know what the performance is limited by I will also know what to optimize. If the problem is with pixel processing than there is not much possibility for optimization, except of making smaller color depth and resolution. If this is the case care should be taken of the quality of resulting graphics. If the rendering is limited by geometry processing, then the problem can be either in too complex shapes or too many of them resulting in a lot of API calls. Beets (2005), ARM (2007) give a number of optimization recommendations, related to geometry processing. Most of them involve the geometry submission order, geometry complexity and the number of API calls. The problem with map applications, however, is that it may be difficult to control which drawing calls are sent and in which order. The drawing calls sequence depends on map data structure, which might not be easy to modify. For example RaveGeo Map Client (Idevio AB, 2010) gets the data from the server and does not have any control over it. That is why optimization has to be done on the client side, i.e. after the drawing call has been received. The optimization methods described here are mostly based on the attempt to avoid drawing unnecessary shapes, to minimize the number of shapes and the number of API calls. The number of API calls may be especially important for OpenGL ES API as it does not support neither retained mode nor display lists. The image below (see Figure 5) demonstrates high-level communication between map engine components and where the optimization is possible. The map client reads the map data, which cannot be modified in the scope of this research. Thus too many or too complex shapes may be read. It is only the client side where optimization can be applied. As shown on the image the Map Client defines what API calls to make or how to group geographical data. This is what I will attempt to optimize. Once the API is called no changes can be done to the drawing algorithm as it is internal API implementation. For

this reason I refer to these optimization methods as high-level.

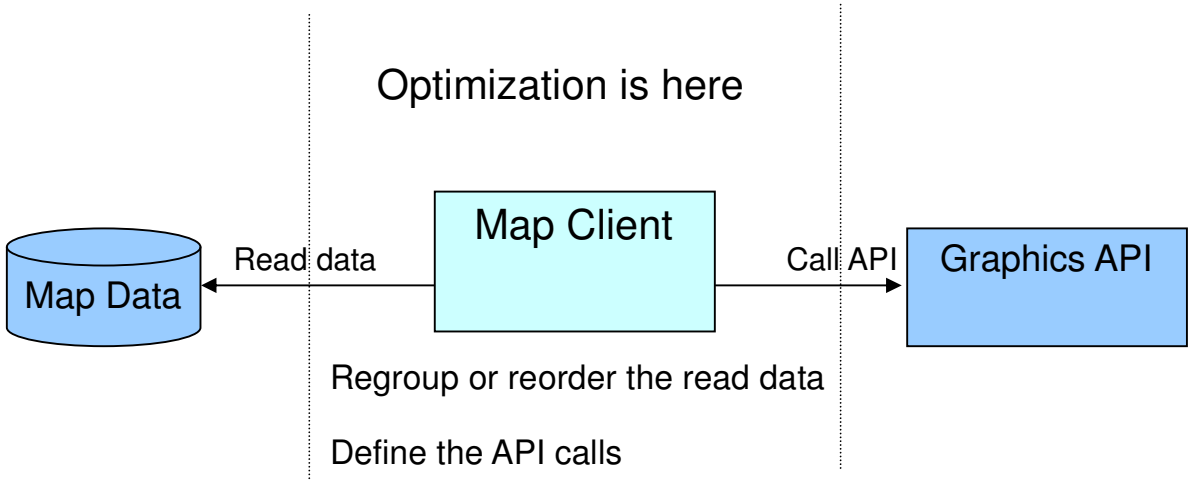

Figure 5: High-level optimization for a map client.

In the case of geometry rendering performance bottleneck the following optimization methods will be tested:

1. High-level clipping

   Clipping technique was presented earlier in Rendering Optimization Techniques section. Clipping is used to avoid drawing outside the viewport or screen. While some APIs use the technique it might still be useful to do a simple clipping before calling the API. For example, if the whole line lies outside the viewport than there is no need to make an API call. A simple algorithm for clipping will be used: only those line segments that have any points inside the viewport will be submitted to the API.

The following three methods are intended to minimize the number of API calls. They are intended for OpenGL ES API. The idea is to combine the lines to get fewer calls.

2. Concatenating shapes

   RaveGeo Map Client logging showed that some lines have common vertexes thus it seems logical to combine these lines in one. However it doesn't seem to be too many of these cases in RaveGeo Map Client and it is unlikely that the situation is common in other map applications.

3. Adding degenerate triangles

   Adding degenerate triangles in a triangle strip is a common technique to combine several disconnected strips (Pulli et. al., 2007). To connect two triangle strips two degenerate triangles are added each of which has 2 same vertexes. In this way the shapes are displayed as disconnected but are in fact one triangle strip that can be drawn in one API call. Unfortunately, this approach does not work for line strips. There is no way to draw lines with different width in one line strip, nor is it possible to draw several line strips in one call. For this reason this approach can only work if lines are constructed from triangles. This method may improve performance, but it should be considered that the number of elements in the strip will get higher. Before trying this optimization method it will be good to compare the line drawing efficiency

with line strips and triangle strips. This can be done with synthetic test similar to those described above.

4. Adding transparent lines

   While it is not possible to draw lines with different width in one line strip it is possible to draw lines with different colors. The idea of this optimization method is to connect separate line strips with transparent lines. Following this approach I can get as few calls as often the line width changes. This method requires drawing additional unnecessary transparent lines, but can reduce the number of calls significantly.

The following optimization method is OpenGL ES specific.

5. OpenGL ES state

   In the OpenGL ES development guide provided by ARM (ARM, 2010) it is noted that for better efficiency it is important not to make redundant OpenGL ES state changes. Moreover Pulli, et. al., (2007) explains that it makes sense to group the objects depending on their state. OpenGL ES state keeps, for example, the current line width and color (Blythe, 2004). In the map applications it may often happen, that same color or line width is set several times for each geometry. Thus it is better to check if the state really changed before making a call to the API.

6. Caching data

   Finally, I will try to cache the geometry data to see how OpenGL ES is capable of drawing graphics when all the data is ready. The technique was described in Rendering Optimization Techniques section. This test will redraw the same frame several times; starting from the second frame there will be no need to construct the geometry structure again. This method will let us know what takes most of the time: drawing geometry or preparing data for it. Moreover it can be used in a real map application when the same map area is repainted.

The optimization methods above are oriented at map applications. While RaveGeo Map Client will be used as an example of map application, the stated issues may be common for other map clients. If different map clients use the data from one source they would probably run into the same situations of too many drawing calls, separated lines or redundant state changes. These kinds of optimization may change rendering efficiency both to the better and to the worse. Thus all of them will be tested and the results will be discussed in Results Analysis section.

4 Results Analysis

The review of different graphics APIs and libraries specifications, articles with opinion of other researchers and developers and, finally, the conducted performance testing gave good comprehension of available graphics solutions for map rendering on mobile devices. This section will first provide the results of described above performance testing. The results analysis will follow with relation to previous findings in literature review.

4.1 Java ME Performance Tests Results

In the stage of literature review several Java ME graphics APIs and libraries were selected for further investigation: Standard Graphics API, Mobile 2D Graphics API (SVG), Mobile 3D Graphics API (M3G) and Tinyline 2D library. In this section I will provide my findings in terms of performance of these APIs. Three phones were selected for testing: Ericsson K800, Nokia 6120 Classic and Nokia 97 mini. First of all the three common synthetic tests were performed. The results are presented in the Table 5. For phone/API/test combinations where the time is missing the test was either not conducted due to low results of other tests or it hanged the phone.

	Ericsson K800			Nokia 6120 Classic			Nokia 97 mini		
API/ library	Rectangle	Polyline	Group of objects	Rectangle	Polyline	Group of objects	Rectangle	Polyline	Group of objects
Standard	1	27	8	0	133	12	0	119	8
SVG	Not supported			89	-	-	127	-	-
M3G	7	434	1251	7	80	1949	15	128	1060
Tinyline	22	6339	-	-	-	-	50	11564	-

Table 5: Synthetic tests results for Java ME APIs / libraries

The results of these first tests were rather surprising. While articles and specifications said nothing about Mobile 2D Graphics (SVG) performance and Tinyline 2D library's, it was clearly stated that Mobile 3D Graphics can be really efficient (Pulli et. al., 2007). Tinyline 2D also appeared to be unexpectedly slow with drawing long polyline of 1000 points. As it can be seen from the Table 5 the old Standard Graphics API for Java ME is considerably faster than others at least on the selected phones.

According to strategy defined in Research Method section more tests are required in case of results are worse than 200 milliseconds. First of all the low performance of M3G was investigated. Due to relative complexity of the API the performance bottleneck could be in the wrong API usage. The results of additional testing of Mobile 3D Graphics are presented in Table 6.

N	Test	Ericsson K800	Nokia 97 mini
	Original polyline test	434	128
1	Polyline previously created	430	
2	Back-face culling on	242	
3	Depth buffer disabled	399	75
	Original group of objects test	1251	1060
4	Close objects	1330	1052

Table 6: Synthetic tests results for M3G API

As we can see from the first test result in Table 6 the creation of polyline object didn't influence drawing performance. This means that the problem is in actual drawing. The second test "Back-face culling on" ran in approximately twice shorter time, but it also displayed only half of the line. The third test which disabled the depth buffer is the only one which changed the result. Thus it can be stated that for 2D graphics it is important to disable the depth buffer as it plays no role but affects performance. Grouping the objects close to each other also didn't change the situation as can be concluded from test 4. As Mobile 3D Graphics is based on triangle strips another test seemed important to run – drawing a triangle strip. This test was performed for Mobile 3D Graphics API and Standard Graphics API which can only draw triangles one by one. The length was chosen to be 1000 points, but also a shorter one which is more realistic. To figure out a more realistic length I logged out the length of lines that are drawn by RaveGeo Map Client. In the table below (see Table 7) the results for triangle strip tests are displayed.

	Ericsson K800		Nokia 6120 Classic	Nokia 97 mini
API/ library	Triangle Strip (1000)	Triangle Strip (30)	Triangle Strip (1000)	Triangle Strip (1000)
Standard	56	5	51	40
M3G	762	26	86	186

Table 7: Synthetic tests results for triangle strip functionality for Java ME platform.

The results show that drawing triangle strips is also not faster with Mobile 3D Graphics than with Standard Graphics API on selected phones.

The obvious reason for the problem could be the absence of hardware acceleration. Indeed, none of the selected for testing phones is hardware accelerated. In the articles however it was mostly spoken of GPU high performance advantages (Pulli et. al., 2007; Akenine-Möller, Ström, 2008; Capin et.al., 2008). A search for hardware accelerated Java ME phones unfortunately showed that there is not that many of them. For example according to Nokia's document on hardware accelerated graphics optimization (2007) the list of phones with GPU is very short: it was four phones at that time. This shows that although M3G performance may be good on hardware accelerated phones this graphics solution would have really low portability being suitable for just a few phones.

The next step was to look closer at Tinyline 2D. It showed extremely low results with drawing 1000 points line. However that many points is probably not drawn in a real application. Previously realistic line length was defined as 15 points. In the same way as a realistic line width was defined to be around 10 pixels. As one of the good advantages of Tinyline is its functionality, thick lines feature, for

example, it is important to test thick lines. It could be that the library was slow with long normal line but was quicker with short thick lines in comparison with Standard Graphics API constructing thick lines from triangles. The two tests were run additionally to compare Tinyline 2D with Standard Graphics API: drawing a thick line of 15 points and drawing a thick line of 50 points, both with width of 10 pixels. The tests were run only on Sony Ericsson K800. The results are shown in the table below (see Table 8).

	Thick line (15 points)	Thick line (50 points)
Standard Graphics API	5	66
Tinyline 2D library	12	180

Table 8: Synthetic tests results for thick line functionality for Java ME platform.

The results of the tests show that Standard Graphics API is in any case faster than Tinyline 2D. Most likely when Tinyline 2D was developed the focus was on its functionality and not performance.

Due to synthetic tests results which showed that Standard Graphics API is considerably faster than any other no application tests were conducted for Java ME platform. From both literature review and testing it can be concluded that Mobile 3D Graphics API can be fast on hardware accelerated phones. However for most of the Java ME phones, that don't have GPU, Standard Graphics API still remains to be the most efficient solution.

4.2 Android Synthetic Tests Results

Currently Android developers have mainly two options for rendering graphics on mobile devices. One is custom 2D graphics library, which is most common and easy to use, and the other is OpenGL ES, which is a 3D solution. OpenGL ES can take advantage of hardware acceleration if it is available and thus a good performance can be achieved according to literature study results. However performance tests of both APIs were also conducted to gain an adequate result. Two Android phones were used for testing: Samsung Galaxy 7500 and Google Nexus One phone. The three basic synthetic tests were run: one rectangle, long polyline and group of objects. Apart from them drawing thick line speed was measured. Anti-aliasing was taken into account as it may influence the performance significantly. However Samsung Galaxy 7500 does not support anti-aliasing for OpenGL ES, that is why the results for test with anti-aliasing and without are the same. The results for synthetic tests are presented in the table below (see Table 9).

Phone	Google Nexus One		Samsung Galaxy 7500	
Test / API	Custom 2D API	OpenGL ES	Custom 2D API	OpenGL ES
Rectangle	1	0	2	4
Polyline not anti-aliased	27	1	21	19
Polyline anti-aliased	35	1	58	20(no anti-alias)
Thick line not anti-aliased	30	1	59	21
Thick line anti-aliased	122	2	332	22(no anti-alias)
Group of objects	10	38	49	151

Table 9: Synthetic tests results for Android Graphics APIs

Tests results show that OpenGL ES can be much more efficient than Custom 2D library in some cases. It is true for Google Nexus One: 1-2 milliseconds for all kinds of lines with OpenGL ES in comparison with 27-35 milliseconds of the same drawing performed by Custom 2D library. On Samsung Galaxy OpenGL ES showed good results for thick lines drawing which is also important. Unfortunately we don't see the same advantage of OpenGL ES in a "Group of objects" test. It is actually slower than non-hardware accelerated Custom 2D library. As it was mentioned earlier OpenGL ES does not support retained mode thus drawing a big group of objects (400) results in API calls overhead. Beets (2005) explains that it is a bad practice to make too many OpenGL ES API calls. Same recommendation gives producer of another GPU – Mali GPU (ARM, 2007). In their development guide it is stated: *"When you call glDrawArrays or glDrawElements, the graphics driver collects all current OpenGL ES states, textures and vertex attribute data. The driver processes these to generate appropriate commands for the graphics hardware to perform the specified draw operation."* This sounds rather scaring and time-consuming. In order to find out how the number of calls influences the performance additional tests were run. For each test a line of 5000 points is rendered but with different number of calls. Google Nexus 1 phone was used for this test. The results follow below in Table 10:

Test \ API	Custom 2D API	OpenGL ES API
5 calls	101	2
50 calls	109	6
100 calls	105	8
200 calls	111	29
500 calls	120	37

Table 10: Number of calls dependency on Google Nexus One phone.

It can be concluded from the test results that it is not important for Custom 2D API how many calls were performed as long as the geometry is the same. However for OpenGL ES API a significant difference is noticed. This means that minimizing the number of API calls is really crucial for OpenGL ES applications. One last synthetic test to run was the test which checks hardware acceleration usage. The names of graphics renderers are "Fimg" and "Adreno" for Samsung Galaxy 7500 and Google Nexus One respectively. If the phones used only software acceleration the names would be "Android PixelFlinger 1.0" similar to the one that Android emulator has. This means that both phones do use hardware acceleration with OpenGL ES. In general synthetic tests results for Android platform confirm the previous finding made in literature review.

4.3 Android Application Tests Results

Since synthetic tests don't show the real situation some application tests were also run. As it was described in the Research Method section RaveGeo Map Client was used for making application tests. As OpenGL ES does not support drawing text or polygons application tests were run for just one map layer – lines layer. While it is still not the map that most users view (a full map with all layers) nevertheless it can be displayed in real applications and can give a better understanding of how fast the drawing is performed. Two map databases were used: Open Street Map and TeleAtlas map databases. Drawing line method was modified to use OpenGL ES API. The results for the first application test with the line layer are presented in Table 11 below. Figure 6 below demonstrates the screenshots from test running where the difference between two databases can be realized.

Phone	Google Nexus One		Samsung Galaxy 7500	
Test \ API	Custom 2D API	OpenGL ES	Custom 2D API	OpenGL ES
Open Street Map	64	70	151	219
TeleAtlas	204	309	834	924

Table 11: Application test results for Android Graphics APIs

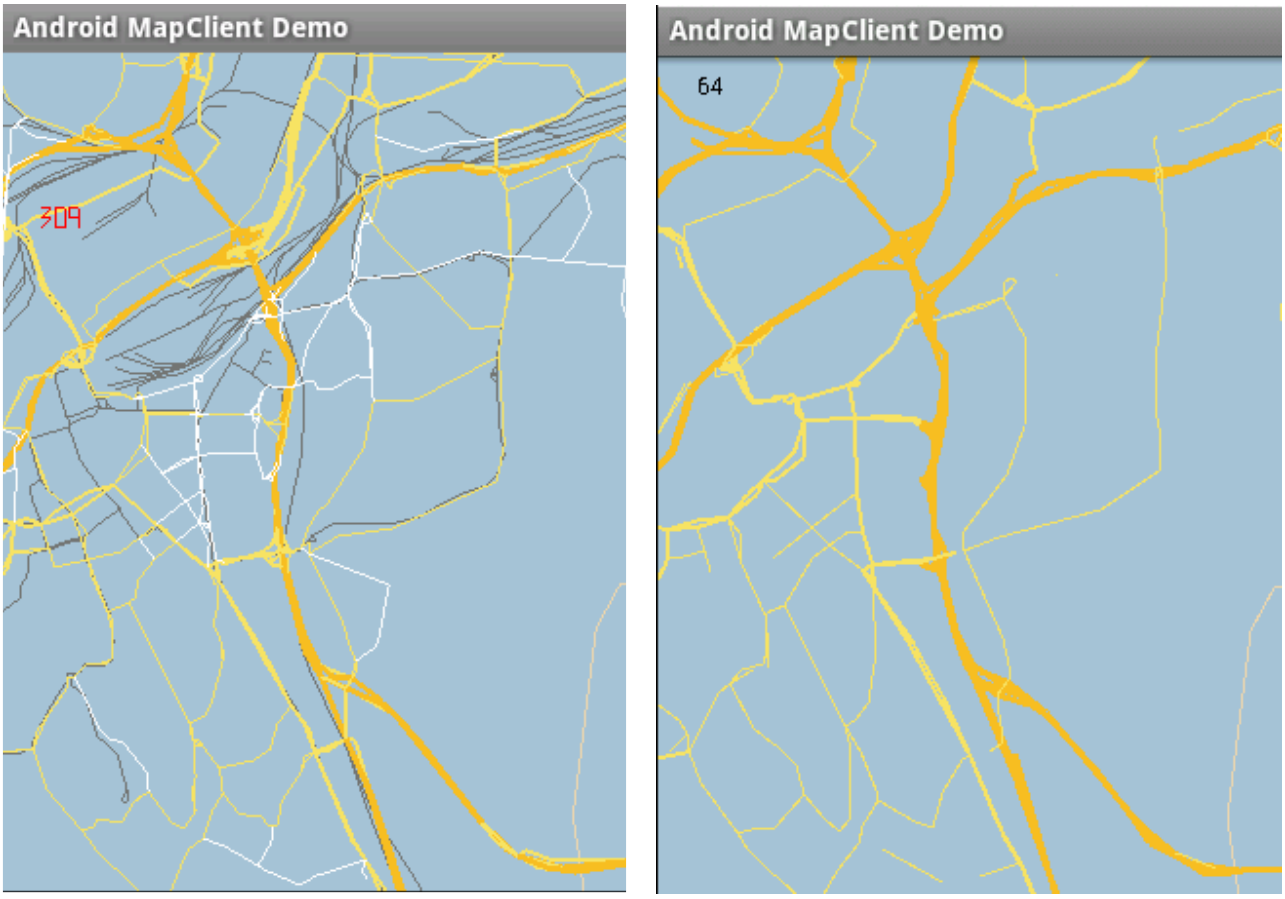

Figure 6: Application tests for Android Graphics APIs. Left: TeleAtlas data with OpenGL ES. Right: Open Street Map data with Custom 2D API.

Despite the very good performance of OpenGL ES in drawing some shapes it didn't manage to draw the whole map faster than Custom 2D API. Next the reasons for that were investigated and the attempts to optimize OpenGL ES drawing were taken.

4.3.1 Optimization Tests Results

The first step of OpenGL ES optimization was to define what the performance was limited by. See Figure 5 above which demonstrates how this can be done. First of all the drawing calls were disabled to see what portion of the processing time was spent on other than drawing operations. After disabling the drawing calls I got the following processing time for Open Street Map:

Google Nexus One: 42
Samsung Galaxy 7500: 110

As we can see high portion of time is spent not on drawing calls, but on application logic: around half of the time. However another half is still taken by the drawing operations and thus it makes sense to optimize them. Next another test was run with a very small viewport to see if the performance bottleneck was with pixel processing or with geometry processing. With the purpose not to depend on the application processing only the time spent for drawing was measured. Samsung Galaxy 7500 phone was used for this test. First the drawing only time with normal viewport was measured and then with the smaller one. The results were the following:

Normal viewport: 89
Small viewport: 92

As the processing time does not depend on the viewport size it can be stated that the graphics performance bottleneck is in geometry processing. This means that the geometry submission could be optimized. One of the reasons for geometry processing performance problem could be the number of drawing calls as synthetic test (Group of objects) already demonstrated this problem. From synthetic tests results we can see that the number of calls does not influence Custom 2D API, so reducing the number of calls may only affect OpenGL ES implementation.

Before trying to reduce the number of calls, however, I tried to avoid changing OpenGL ES state when it is not necessary, as proposed by ARM (ARM, 2007). In the map client that I used it related to lines color and width. This was tested on Google Nexus One phone with Open Street Map data. An average result is 67 milliseconds, which does not make much difference in comparison with the first version which gave 70 milliseconds. In any case avoiding state changes is a good practice and it unlikely can make things worse, thus it was applied in all subsequent tests.

Next I tested optimization ideas to reduce the number of calls that were described in Application Level Optimization section. Trying to concatenate lines that have same ending point seemed to be most straight-forward, but it didn't lead to significant calls decrease. The results from the test didn't change the drawing performance as well.

Pulli, et. al, (2007) described a common method for connecting triangle strips. This can be done by adding two degenerate triangles, each of which has two same vertexes. Such degenerated triangles are not displayed at all. This is how it is possible to draw several disconnected triangle strips in one API call. Unfortunately, the same techniques could not be used for lines as it is not possible to change the width for different segments. (Otherwise could be changed to zero.) However some APIs, e.g. Mobile 3D Graphics, don't support lines at all and everything is created from triangles. I tried to create lines from triangles instead of using line strip with the purpose to combine them in on or more API calls. First I ran synthetic tests to see how this change of algorithm influences drawing performance. The time for drawing normal and thick lines, constructed from triangles, with 1000 points are displayed in

Table 12.

Test \ Phone	Google Nexus One	Samsung Galaxy 7500
Polyline from triangles	31	61
Thick polyine from tirangles	33	71
Normal polyline	1	19
Normal thick polyline	1	21

Table 12: OpenGL ES tests results for drawing lines constructed from triangle strips in comparison with lines constructed from line strips.

It is clear from the table that line strips are better optimized than triangle strips. Another point is that drawing a triangle strip contains around four times more elements than a line strip. This comes from an algorithm for creating lines from triangles that was used. The performance difference is dramatic especially for Google Nexus One phone. However application tests were also run to see if the smaller number of API calls could be more important than the time of drawing each single line. As putting everything in one huge triangle strip could cause memory issues the triangle strips were split with 1000 points lines and 500 points lines. The optimization method was tested on both available phones for Open Street Map. The results are available in Table 13.

Test \ Phone	Google Nexus One	Samsung Galaxy 7500
Original result	70	219
Lines from triangles, same number of calls	101	254
1000 points in a triangle strip	175	480
500 points in a triangle strip	156	455

Table 13: OpenGL ES tests results for drawing map, where lines are constructed from triangle strips in comparison with lines constructed from line strips.

While the number of API calls drop dramatically the efficiency changed to the worse. It could mean that smaller number of API calls does not improve efficiency, but this contradicts with synthetic tests results. That is why it is more likely that switching to triangle strips instead of line strips affects performance that much. In the previous synthetic tests we could see a high difference between drawing lines from line strips and from triangle strips. What seems surprising is that drawing all lines with triangles but keeping the same number of calls is faster than combining triangle strips in one with degenerate triangles. This can mean that too many triangles in a triangle strip also cause efficiency problems.

The next step I took was trying to combine the lines with transparent segments. This reduced the number of API calls significantly. However, the results for both phones didn't prove this method to improve performance. For OSM map data the results were the following:
Google Nexus One: 75
Samsung Galaxy 7500: 402

This demonstrates that although the number of calls fall down the time of the drawing took even longer for Samsung phone. It is likely that the transparency functionality is time consuming and thus it is not efficient to draw transparent lines. The same test was run with disabled blending, i.e. the separate lines where connected with non-transparent lines. The results are as follows:

Google Nexus One: 84
Samsung Galaxy 7500: 363

The result means than transparency is not that critical at least not for Google Nexus One phone. Same as with triangle tests I come to the conclusion that the number of calls is not as important as the geometry complexity and size. The geometry grouping in the map client could take time as well. The next step was to try to reduce the size of the lines. As a map application cannot really control what drawing calls it gets it may get lines that fully or partly are outside the viewport. High-level clipping described in Application Level Optimization section was used to avoid submitting such lines to OpenGL ES. It was tested on Samsung phone for Open Street Map data. The number of calls didn't decrease as a result but the length of some lines became smaller. The resulting time was 199 milliseconds, which is less than original 219 milliseconds. However the difference is not significant, which can mean that there was not that many lines outside the viewport.

The final planned test is caching the data. It is good to see how fast the drawing is performed when the geometry structures are ready. It was tested on Samsung Galaxy 7500 phone for Open Street Map data. The results were the following:

1st Frame: 266
Average of subsequent frames: 54

This finally improves the performance greatly. While there is no need to construct data structures for the frames starting with second, there is also no need to perform much of application logic which can also take significant time. I ran another test for Open Street Map which included no optimizations but measured only the drawing time without application logic. This was run for both OpenGL ES API and Custom 2D library. Here are the results:

OpenGL ES: 82
Custom 2D API: 64

82 milliseconds for OpenGL ES is slower than 54 milliseconds when the data is prepared. The results of the caching test show that not only making many gl calls are important but also preparing necessary objects. This also explains why in previous tests the results were getting worse after trying to group geometries. Combining lines coordinates in one array and wrapping them in a structure suitable for OpenGL ES calls afterwards takes time. We can see that OpenGL ES could perform better than Custom 2D API (54 milliseconds and 64 milliseconds respectively) only when the data for OpenGL ES was prepared in advance.

4.3.2 Tests Results Summary

Running application tests with the use of RaveGeo Map Client (Idevio AB, 2010) that draws line layers showed that Custom 2D API is more efficient than OpenGL ES for both phones. Some optimization tests were also run to find out the reasons for slow performance of OpenGL ES and improve it. It was found that most of the time in the application test spent not for drawing. This, however, doesn't mean

that there is no point to optimize drawing. Firstly, it may not be an issue with other map applications. Secondly, RaveGeo Map Client was modified to make it possible to measure the time, while in reality it runs in a different way. The viewport test showed that the problem is lying in the geometry processing. Table 14 shows the results for a number of tests that were run to optimize geometry submission.

Lines concatenation	No effect or worse
Lines constructed from triangles, several triangle strips are combined in one	
Line strips are combined in one with transparent lines	
Line strips are combined in one with non-transparent lines	
High-level clipping	Slightly improved performance
Caching the data	Much better, starting from the second frame. Application logic overhead is also removed.

Table 14: Results for OpenGL ES application tests to optimized geometry submission.

Among all the optimization tests only caching the data gave significant improvement. This was faster due to two reasons:

- data structures for OpenGL ES were constructed in advance
- much of application logic such as reading preloaded geographical data was not needed to be run

Measuring drawing time only, which is not dependent on additional application logic gave the results, presented in the diagram below (see Figure 7).

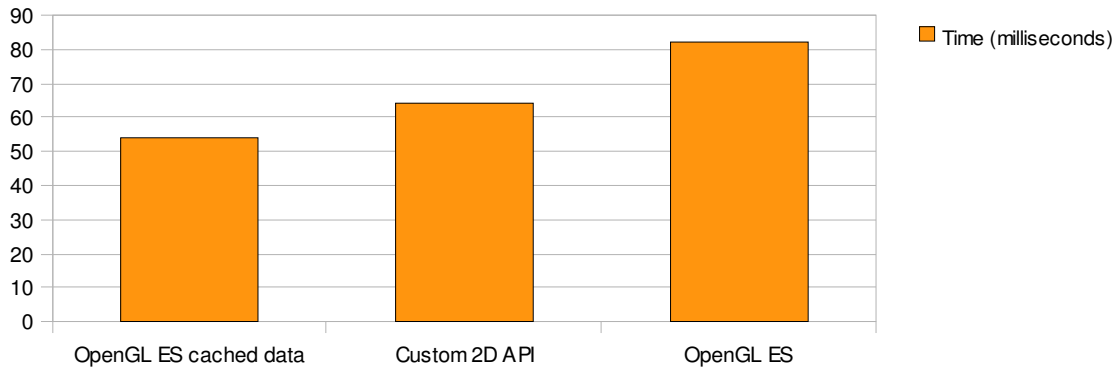

Figure 7: Line layers drawing time on Android.

The time difference is not really noticeable, however with less number of calls OpenGL ES could perform better as showed synthetic tests. Finally, I can conclude that under certain conditions OpenGL ES can be more efficient than Custom 2D API. These conditions are caching data were possible as well as reducing the number of geometries. Reducing the number of geometries would cause less OpenGL ES API calls and also less overhead related to preparing data structures. To conclude Android testing results it can be stated that OpenGL ES performance heavily depends on submitted geometry structure.

While the API is really good in drawing separate objects, which fully agrees with literature review findings, it may not be the case for map rendering. The articles and specifications also mostly refer to 3D applications with animated graphics, e.g. games, where the application has control over the order and structure of submitted geometries. As it is not the same for map rendering the results of the application testing may seem to contradict with the literature review.

Performance testing, both synthetic and application level, showed the efficiency of mobile graphics APIs in terms of drawing speed. In Discussion section a summary of graphics APIs focusing on performance, portability, graphics quality and functionality will be provided. Finally recommendations for using them will be formulated.

5 Discussion

The discussion will be related to the overall aim of the research, which was to find efficient methods for map rendering, and specific research objectives that are listed below:

- Identify important qualities of map rendering
- Evaluate available Java ME and Android graphics APIs and libraries by means of literature review and testing
- Investigate techniques that can speed up drawing
- Propose one or more efficient alternatives for map rendering on mobile devices

This section will summarize the findings of this research work and provide conclusions for each research objective. Basing on the conclusions it will then give recommendations related to mobile map rendering solutions to software architects, developers and quality managers.

5.1 Findings Summary and Conclusions

While drawing performance was considered to be the most significant quality of map rendering other qualities were taking into account as well. Their identification was carried out in the scope of the first objective which was to identify important qualities of map rendering. Literature review provided understanding of what graphics quality consists of and what quality attributes are important for mobile graphics in general. Basing on that, as well as RaveGeo Map Client (Idevio AB, 2010) functionality, important qualities for particularly map rendering were selected. As a conclusion for first objective these qualities were defined to be anti-aliasing and transparency. Anti-aliasing makes the graphics smoother, while displaying transparent objects on the map can make the graphics more compact. Both of these qualities improve map readability.

The second objective, which was to evaluate graphics APIs and libraries, required most of the research work. First a criterion for evaluation was selected focusing on map applications. As a result four important issues made up this criterion:

- Performance
- Portability
- Graphics quality: anti-aliasing, transparency, basing on the first objectives conclusion
- Functionality: thick lines, polygons, text

The last three issues were found out from APIs and libraries specifications, as well as other literature. As for APIs performance along with literature review extensive testing on mobile phones was conducted as a research method. Some of the tests were performed with the use of Idevio's mobile map client (Idevio AB, 2010). The testing results appeared to be unexpected and in a way contradicting with literature review. However, this does not mean that the articles or specifications provide wrong information, but that map applications may use the APIs in specific way and that the APIs perform differently on various phones.

First I will summarize graphics APIs and libraries evaluation for Java ME platform. The following graphics APIs and libraries were investigated:

- Standard Graphics API
- Mobile 2D Graphics API (M2G)
- Mobile 3D Graphics API (M3G)
- TinyLine 2D library

The diagram below (see Figure 8) shows approximate comparison of investigated Java ME APIs against four selected criteria.

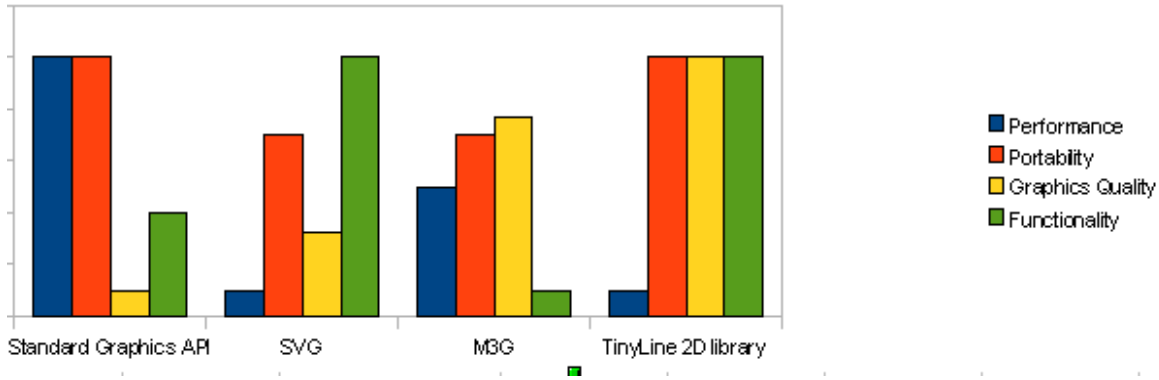

Figure 8: Java ME graphics APIs and libraries evaluation

The values in the diagram should not be considered absolute but are only for comparison purpose. Moreover criteria can be only compared between different APIs but not between each other. The values for different criteria were not weighted according to criteria importance. Also the performance of M3G specified in the diagram does not correspond to any mobile device. It was discovered from both testing and literature review that M3G API has extremely different performance on different phones. While it can be really fast on hardware accelerated phones it is not the case with other devices where it falls significantly. This means that on some phones, though nowadays really few, M3G can outperform Standard Graphics API in drawing speed, but on others it is opposite. Portability for Standard Graphics API and TinyLine 2D library is 100%, which means that they would work on every Java ME phone. On the other hand it is not the case with SVG and M3G APIs. Graphics quality values were constructed from the transparency and anti-aliasing support, same as functionality from drawing thick lines, polygons and text support.

Next the results from Android graphics APIs evaluation will be presented. Nowadays Android developers have a choice of two graphics APIs:

- Custom Graphics API
- OpenGL ES API

The diagram on Figure 9 shows approximate comparison of investigated Android graphics APIs against four selected criteria. The diagram is built on the same principle as the one for Java ME APIs.

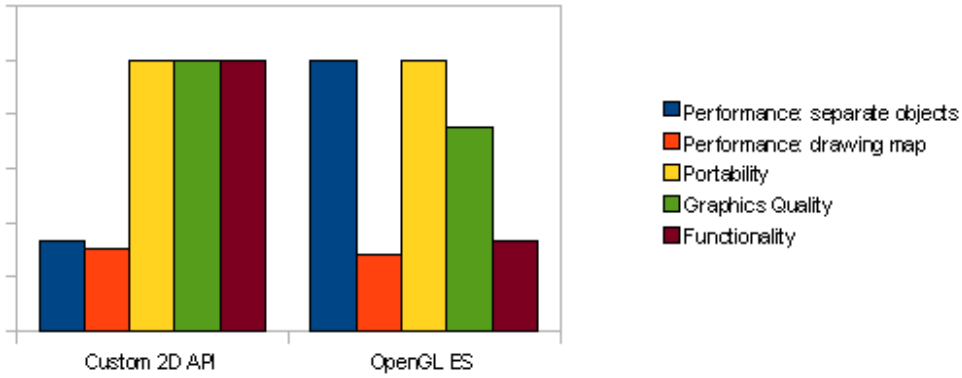

Figure 9: Android graphics APIs and libraries evaluation

Both APIs are portable throughout all Android phones. Graphics quality and functionality are better supported for Custom 2D API. However, it is not that clear with performance. While OpenGL ES API can draw separate objects, e.g. lines, really fast, a number of problems arise when it comes to drawing the whole map. The following are the issues that negatively influence OpenGL ES performance for map rendering:

– too many geometries, resulting in high number of API calls

– time-consuming data construction prior to API calls

It was also found that trying to combine separate geometries results in an overhead of constructing geometry data.

Caching the data can improve the drawing performance of OpenGL ES so, that it is slightly faster than Custom 2D API. Finally, regarding Android APIs performance it was concluded that OpenGL ES can be more efficient for map rendering than Custom 2D API, but under certain conditions. Caching the data, optimizing data construction and grouping geometries before sending them to map client can provide high performance of OpenGL ES graphics.

The third objective was to investigate techniques that can speed up drawing. The literature review gave knowledge of common techniques, yet it was concluded that most of them are too low-level to be feasible to implement in most map engines. At least it was not intended to make study of those low-level optimizations in this work. Some high-level techniques, however, were tested with OpenGL ES API on Android phones: high-level clipping and data caching. It was discovered that caching the data can improve performance significantly for both APIs as it eliminates reading the data when the same area needs to be repainted. It can increase the drawing speed up to three times for the second and subsequent drawings of the same map area.

My last objective was to propose efficient alternatives for map rendering on mobile devices. The recommendations on what graphics solutions to use for map engines will be provided in the next section.

5.2 Recommendations

Basing on the research results it is possible to formulate recommendations on how to achieve better map rendering performance on mobile devices. The most critical decision is probably the choice of graphics APIs or libraries for the given platform. This kind of recommendations will be based on the conclusions made from the second objective result, which was to evaluate available APIs and libraries. The recommendation for selecting drawing APIs for both platforms, Java ME and Android, are

summarized in the diagram at Figure 10. More detailed description will follow.

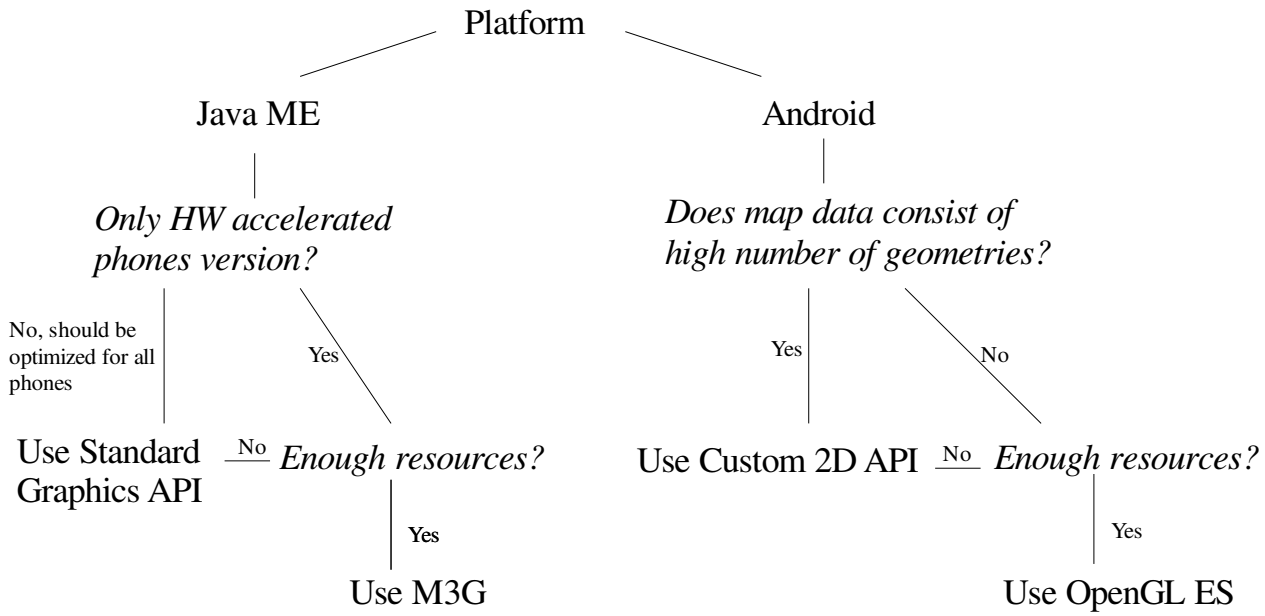

Figure 10: Recommendations for selecting mobile graphics API for map rendering.

First Java ME map rendering alternatives will be proposed. From the diagram at Figure 8 which summarizes Java ME APIs evaluation it may still seem unclear which graphics solution is better. However, as drawing performance was assumed to be the most important characteristic it is only Standard Graphics API and Mobile 3D Graphics that can be considered. As currently there is only a few Java ME hardware accelerated phones on the market using M3G would contradict portability requirement. That is why while making decision for Java ME platform the first question should be what degree of portability is required. In this research portability was considered to be an important issue thus it is proposed to use Standard Graphics API from LCDUI package as the best combination of performance and portability. If, however, software designer or developer would like to create a special hardware accelerated version than it makes sense to use Mobile 3D Graphics. In this case the limited functionality of M3G should be taken into account as everything it supports is triangles, not even lines. This can add high degree of development complexity as well as cause additional graphics quality and performance problems. Also M3G API is more complex than Standard Graphics API, using it not as it was intended may affect drawing efficiency. For this reasons it is advised to use this API if enough time resources are available as well as skillful stuff.

For Android platform the situation is not the same as with Java ME. Both Android APIs are available for all Android phones. Most of Android phones support hardware acceleration which OpenGL ES relies on. The research results reveal that OpenGL ES can be significantly faster than Custom 2D API in some cases. For map applications though geometry processing can cause performance decrease. One of the major problems that affect drawing performance is the number of API calls. That is why the first question to answer is how the geographical data is stored. If there are too many, separate, but small geometries, than it is likely that Custom 2D API will be more efficient. Around 200 separate objects for all roads of one frame can be considered as too high number of geometries. If the data structure can be modified than the number of objects should be minimized. For relatively low number of objects

OpenGL ES can be used efficiently. In this case the following should be taken into account:

- the number of API calls should be minimized
- the geometry data preparation should be optimized
- were possible the geometry data should be cached and objects should be reused

Also, same as with Mobile 3D Graphics API it requires more effort to use OpenGL ES than Custom 2D API. The obvious benefit of Custom 2D API is the ease of development. The geometry submission and number of calls don't affect performance, no preparation before the actual drawing call is needed. The map client needs not to care about what kind of map data it gets. Moreover the reach functionality allows saving time on implementation of required features, such as drawing text or polygons. According to the conclusions of the third objective caching data can speed up drawing significantly. This is only the case when the same map area is redrawn nevertheless it is worth using the technique were possible.

The final recommendation is given to those who are responsible for software quality, e.g. quality managers, and involved in mobile mapping projects. Basing on the conclusion from the first objective, it is recommended to pay attention at defined important graphics quality attributes such as anti-aliasing and transparency. This will allow seeing if the graphics quality could be potentially improved. If the map seems not to be smooth, for example, than probably it is because of lack of anti-aliasing.

6 Conclusion

The goal of this research was to find efficient methods for map rendering on mobile devices for both Java ME and Android platforms. The first objective of the research was to define important graphics qualities for map application. Basing on the literature review and also RaveGeo Map Client (Idevio AB, 2010) functionality two important qualities were selected: anti-aliasing and transparency. After that mobile graphics APIs and libraries were investigated by the means of articles and APIs specifications review as well as extensive performance testing. APIs evaluation was the second objective of the report. The APIs were evaluated according to four selected criteria: performance as the most important, portability, graphics quality and functionality. To fulfill the third research objective techniques that can speed up drawing were analyzed. The forth and last objective was to formulate the recommendations for graphics solutions for mobile map rendering.

The recommendations were based on conclusions from previous objectives. For Java ME platform using Standard Graphics API from LCDUI package is recommended, although Mobile 3D Graphics API can be used for special versions for hardware accelerated devices. This is based on APIs evaluation from which it was concluded that Standard Graphics API still remains the best combination of performance and portability, although Mobile 3D Graphics API can be more efficient for hardware accelerated phones. As for Android platform two graphics APIs were investigated: Custom 2D API and OpenGL ES. The evaluation of these APIs was not very straight-forward. It was concluded that Custom 2D API is more efficient in many cases although under certain conditions OpenGL ES can perform faster with its hardware accelerated graphics. It is recommended to use OpenGL ES API if the geographical data structured in a way that the number of objects is minimal. It was also concluded that complexity of APIs and their functionality should be taken into account as both Mobile 3D Graphics and OpenGL ES require more development effort than their alternatives. The third objective which was to explore techniques to optimize drawing revealed that caching data where possible can significantly improve drawing performance. Thus this technique was also recommended for map rendering on mobile devices.

Regarding given recommendations it is interesting to see how they are applicable for RaveGeo Map Client that was used as map application example. For Java ME platform Standard Graphics API is used at the moment, thick lines and polygons are implemented with triangulation algorithms. According to recommendation given this actually is the best solution for most of the phones. That is why the performance can only be improved for a few hardware accelerated phones. As for Android platform the situation is slightly different. The current solution used for RaveGeo Map Client is Custom 2D API. With the data that is used today it is in fact the fastest solution. As the client uses map data from different sources, that are also updated often, it is difficult to control the number of geometries. Custom 2D API works with similar speed regardless the data structure and thus no unexpected performance fall would happen in case of database changes. However, data caching could be applied. As Android virtual machine is not very well optimized, reading the data even if it is already loaded is slow. That is why avoiding it when possible can give noticeable performance advantage.

In the next section the limitations of this work will be addressed and recommendations for future research will be proposed.

6.1 Limitations and Future Work

The research had a number of limitations that will be summarized in this section. Mainly in connection with these limitations some recommendations on possible future research will be given.

The most important limitation was rather small number of mobile devices that it was possible to test on. Those tests that were run showed that the results can differ significantly for different devices. Another limitation was that for application tests RaveGeo Map Client was used as an example of map application. However, it is obvious that some issues that are true for this application may be not same for others. Some other common potential problems could be missed if they were not the case for RaveGeo Map Client. This is why it could be good to make a more thorough study of mapping engines to know how they usually work and can be optimized.

In the scope of this work more research could be conducted on how to optimize OpenGL ES for Android phones. For example using Vertex Buffer Objects could be investigated which I didn't have sufficient time for in the scope of this work.

With the fast development of technologies, APIs modifications, new phones releasing the situation with mobile graphics solutions may change. This is another limitation that the research was conducted in the fast changing world of mobile technology. For example if more Java ME phones support hardware acceleration it may make sense to better investigate Mobile 3D Graphics performance. OpenGL ES 2.0 version is already available on some Android phones. This version has some essential changes and thus may be good to see how they affect map rendering efficiency.

Finally, the research was highly focused on map applications, while other mobile graphics applications were not considered. A more general evaluation of graphics APIs and recommendations for choosing one of them would provide a better picture of available graphics solutions.

7 References

Aarnio, T, 2003. *JSR 184: Mobile 3D Graphics API for J2ME.* Java Community Process.

Akenine-Möller, T, Ström, J, 2008. *Graphics Processing Units for Handhelds.* Proceedings of the IEEE.

Android APIs, 2010. In Android APIs specification. Available at: http://developer.android.com/reference [Accessed 31st March 2010]

ARM, 2007. *Mali GPU OpenGL ES Application Development Guide.* Mali GPU Application Development guides.

Beets, K, 2005. *Developing 3D Applications for PowerVR MBX Accelerated ARM Platforms.* Imagination Technologies Ltd.

Bing-Yu Chen, R., 2006. *JGL. 3D Graphics Library for Java.* Available at: http://www.cmlab.csie.ntu.edu.tw/~robin/jGL/ [Accessed 11th March 2010]

Biuk-Aghai, R, 2005. *Performance Tuning in the MacauMap Mobile Map Application.* Proceedings of the Third International Conference on Information Technology and Applications.

Blythe, D, 2004. *OpenGL ES Common/Common-Lite Profile Specification. Version 1.0.02.* Khronos Group.

Capin T., Pulli K., Akenine-Möller T., 2008. *The State of the Art in Mobile Graphics Research*. IEEE Computer Graphics and Applications.

Cheng-Han Tu, Bing-Yu Chen, R., 2005. *The Architecture of a J2ME-based OpenGL ES 3D Library.* Ninth International Conference on Computer Aided Design and Computer Graphics (CAD/CG 2005). IEEE.

Dong Li, Yongxiong Zhang, Bingjun Yu, Ning Gu and Yuhui Peng, 2007. *Research on Mobile SVG Map Service Based on Java Mobile Phone.* Asia-Pacific Service Computing Conference, The 2nd. IEEE.

Eskelinen, J, 2005. *JSR 226: Scalable 2D Vector Graphics API for J2ME* . Java Community Process.

Graphics, 2010. In Android Development Guide. Available at: http://developer.android.com/guide/topics/graphics [Accessed 31st March 2010]

Galin, D, 2004. *Software Quality Assurance. From theory to implementation.* Essex, England: Pearson Education Limited.

Girow, A, 2010. *Tinyline 2D.* Available at: http://www.tinyline.com/2d [Accessed 11th March 2010]

Harun, H. Jailani, N. Bakar, M.A. Zakaria, M.S. Abdullah, S., 2009. *A Generic Framework for Developing Map-Based Mobile Application.* International Conference on Electrical Engineering and Informatics 5-7 August 2009, Selangor, Malaysia. IEEE.

Hopkins, June 2007. *The Java ME GUI APIs at a Glance.* Sun Developer Network. Available at: http://developers.sun.com/mobility/midp/articles/guiapis [Accessed 1st February 2010]

Idevio AB, 2010. *RaveGeo Map Client*. Available at: http://www.idevio.com/v2/index.php/ravegeo-mapclient.html [Accessed 2nd May 2010]

Khronos Group, 2010. *Open GL*. Available at: http://www.opengl.org [Accessed 1st February 2010]

Nokia, 2005. *Mobile 3D Graphics API. Technical Specification.* Version 1.1.

Nokia, 2006. *MIDP: Scalable 2D Vector Graphics API Developer's Guide*. Version 1.1.

Nokia, 2007. *Best Practices for HW-Accelerated Graphics Optimization.* Version 1.0.

Powers, M, August 2005. *Getting Started with Mobile 2D Graphics for J2ME.* Sun Developer Network. Available at: http://developers.sun.com/mobility/midp/articles/s2dvg [Accessed 1st February 2010]

Pulli, K, 2006. *New APIs for Mobile Graphics.* Electronic Imaging: Multimedia on Mobile Devices II.

Pulli K., Aarnio T., Miettinen V., Roimela K., Vaarala J, 2007. *Mobile 3D Graphics with OpenGL ES and M3G.* Morgan Kaufmann.

Riggs, R, 2006. *JSR 239: JavaTM Binding for the OpenGL® ES API* . Java Community Process.

Silva, B, 2008. *JMUnit.* Available at: http://jmunit.sourceforge.net [Accessed 20th February 2010]

Skia 2D Graphics Library, 2010. Google open source projects. Avaialble at: http://code.google.com/p/skia [Accessed 31st March 2010]

Sun Microsystems, 2010. *Java ME.* Available at: http://java.sun.com/javame [Accessed 1st February 2010]

Valdin, I, 2006. *Graphics optimization for J2ME compatible mobile phones.* IEEE Tenth International Symposium

Vertex Buffer Objects, 2010. In OpenGL wiki pages. Khronos Group. Available at:

http://www.opengl.org/wiki/Vertex_Buffer_Object [Accessed 15th March 2010]

W3C (World Wide Web Consortium), 2008. *Scalable Vector Graphics (SVG) Tiny 1.2 Specification.* Available at: http://www.w3.org/TR/SVGTiny12/ [Accessed 1st February 2010]

Qusau, M, 2004. *Getting Started With the Mobile 3D Graphics API for J2ME.* Sun Developer Network. Available at: http://developers.sun.com/mobility/apis/articles/3dgraphics [Accessed 1st February 2010]