Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Rendering Interactive Maps on Mobile Devices Using Graphics Hardware

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Lukas Rössler
Matrikelnummer 0625652

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 28.09.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Rendering Interactive Maps on Mobile Devices Using Graphics Hardware

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Lukas Rössler
Registration Number 0625652

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Vienna, 28.09.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Lukas Rössler
Ebenstraße 23, 3204 Kirchberg an der Pielach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank my supervisor, Prof. Michael Wimmer, for his support and especially for his objective criticism during the creation of this thesis; without either, this work would not be what it is today.

Another large thanks goes to the people at Ulmon, who provided me with hardware to implement and test my implementation on, ordered large quantities of coffee and were always there to help me out when I was mentally stuck.

Last, but not least, I want to thank my family, especially my parents, for enabling my academic studies, both financially and by providing any possible support. Of course, neither my studies nor this thesis would have been possible without the help of my fiancée, who not just tolerated many evenings and weekends of work, but also did her best to encourage me to continue working when the "To Do" list only seemed to get longer and longer.

Abstract

Mapping and navigation applications on mobile devices such as smart phones or tablets are increasingly popular. Modern maps are often rendered directly from vector data. Since the performance of a previous CPU-based map renderer was unsatisfactory, a hardware accelerated map rendering prototype for mobile devices based on OpenGL ES 2.0 was created. A novel hybrid rendering architecture is introduced to combine the advantages of tile-based and true real time rendering solutions. The architecture consists of a tile server that renders base map tile images and a client to display them. The new third component, the post-processor, draws dynamic map features such as icons and text above the tiles in real time, enabling a 3D fly-over mode. All components run inside the same process directly on the device. For the rendering of lines, an important map feature, a new rending algorithm was developed, enabling to draw lines of arbitrary width with one of three different line cap styles. Additionally the line can be stippled with a user-defined pattern where each line dash is rendered with the selected cap style. Antialiasing of the line is supported with an arbitrary circularly symmetric filter kernel of user-definable radius. To accelerate icon rendering, a texture atlas is used to store the icons, and a simple but effective packing algorithm has been developed to generate the atlas online.

Kurzfassung

Karten- und Navigationsprogramme auf Mobilgeräten wie Smartphones oder Tablets sind bereits weit verbreitet. Moderne Applikationen rendern die Karten oft direkt von vektorisierten Daten. Aufgrund unzufriedenstellender Performance eines früheren, CPU-basierten Renderers, wurde ein hardwarebeschleunigter Map-Rendering Prototyp für Mobilegeräte auf der Basis von OpenGL ES 2.0 erstellt. Eine neuartige, hybride Architektur, die die Vorteile von Tile-basierten- und Echtzeit-Renderern vereint, wird vorgestellt. Diese Architektur besteht aus einem Tile Server, der Tiles für die Basiskarte rendert, und einem Client, der diese darstellt. Die dritte, neue, Komponente der Architektur, der Post-Processor, ist in der Lage dynamische Daten wie Icons oder Text in Echtzeit über die Basiskarte zu legen, was einen 3D Modus ermöglicht. Alle drei Komponenten werden in einem Prozess direkt auf dem Endgerät ausgeführt. Um Linien, welche bei der Kartendarstellung unumgänglich sind, darzustellen, wurde ein neuer Rendering-Algorithmus entwickelt, der es möglich macht, Linien von beliebiger Breite zu rendern und mit einem von drei Linienabschlusstypen zu versehen. Zusätzlich kann die Linie strichliert werden, wobei jeder Teilstrich mit dem aktuell gewählten Linienabschlusstyp gezeichnet wird. Antialiasing von Linien wird mit Hilfe eines beliebigen rotationssymmetrischen Filters von wählbarem Radius ermöglicht. Um das Rendern von Icons zu beschleunigen wird ein Texture Atlas verwendet und es wurde ein einfacher aber effektiver Algorithmus zur Erstellung und Organisation des Atlas entwickelt.

Contents

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Problem Statement and Aim of this Work 2
1.3 Contributions 3
1.4 Structure 3

2 Related Work **5**
2.1 Graphics APIs 5
2.2 Map Rendering 11
2.3 Out-of-Core Rendering 19
2.4 Texture Atlas Packing 23
2.5 Line Rendering 23
2.6 Polygon Rendering 29

3 Implementation & Own Contribution **33**
3.1 The Hybrid Rendering Architecture 33
3.2 Server 35
3.3 Client-Server Communication 47
3.4 Client 49
3.5 Post-Processing 54
3.6 Development Environment 55

4 Results **57**
4.1 Performance 57
4.2 Quality 62
4.3 Features 65

5 Conclusions **71**
5.1 Challenges 71
5.2 Findings 72
5.3 Future Work 72

Bibliography **75**

CHAPTER 1

Introduction

1.1 Background and Motivation

In recent times, the broad availability of Global Navigation Satellite System (GNSS)[1]-equipped handheld devices has paved the way for mobile mapping applications. Previously, GNSS-supported navigation has only been possible with dedicated devices, mostly used in automotive transportation. In contrast to those dedicated devices, many smart phone owners have their device with them most of the time, so navigation can also be used while walking or using public transportation, as a replacement for a printed map or a city guide. To display a map on a mobile device, a raster image of the visible portion of the map is needed; mapping applications differ in the way this image data is aquired. There are two approaches to solve this task:

- Pre-rendered raster images, organized in a tile pyramid (see Section 2.2.1 for details)
- Vector maps

Pre-rendered images are usually stored online on a tile server and downloaded by the mapping application, requiring an active internet connection. Because of the limited storage capacities of mobile devices, storing the images locally on the device is usually not feasible. An advantage of this approach is that displaying the downloaded tile images is a quick and easy task, but the fact that an internet connection is necessary has a disadvantage: a mobile data connection might not always be available or it might be very expensive, for instance when traveling to a foreign country. To avoid this, applications that use vector maps can store the map data locally, because a vector map needs only a fraction of the storage needed to store pre-rendered images. The raster images needed to display the map on the screen are then created directly on the mobile device. A disadvantage of this method is that generating images from the vector map is a difficult task – especially when the computational capabilities of mobile devices are considered. At the same time, new possibilities arise: the map style can be changed (for instance to a

[1]e.g. GPS, GLONASS

night-mode), features can be displayed or hidden dynamically, and the map can be rotated and tilted while the text labels stay upright.

Another strength of many modern handheld devices like smart phones or tablets is graphics processing. Mobile gaming is an ever-increasing industry and the graphics hardware present in current generation smart phones can be compared to what was state-of-the-art on desktop computers less than ten years ago[2]. However, graphics processors are not designed to render 2D vector graphics; they are good at rendering 3D scenes with complex objects and sophisticated lighting.

Given the processing power of those graphics processors and the task of map rendering, the obvious next step is to trick them into doing what they were not made for: vector graphics. To do so, the problem of drawing vector graphics has to be transformed into a rendering problem that can take advantage of the structure and strengths of a mobile graphics processor.

This thesis was written in cooperation with a Viennese company, Ulmon GmbH, who developed a CPU vector rendering library for their iOS applications; the implementation, this work is based upon, will eventually be used in many of their products. Ulmon GmbH products use OpenStreetMap data in their mapping products, and since the proposed implementation was designed to be compatible with their architecture, this data source was also used for this thesis.

1.2 Problem Statement and Aim of this Work

The main task of the presented work is to create a prototype of a hardware accelerated rendering library to showcase that drawing vector graphics is possible using mobile graphics processors. Outside of the scope of this work, the prototype will be tuned and extended and eventually released. To use the graphics processor, standardized APIs are implemented in the graphics driver; the most widely used one being OpenGL ES, a subset of OpenGL on desktop computers, optimized for mobile graphics. Due to its wide availability, OpenGL ES 2.0 was chosen; the newest version, OpenGL ES 3.0, is not yet supported by any devices.

Graphics processors are optimized to handle geometry that consists of a large number of triangles. Unfortunately, geometry may be specified in a much less restricted way in vector graphics. Common geometry types are paths, consisting of line segments and/or curves; and shapes, which are often implemented as a closed path. A path can be decorated with various paints, which may have gradients, icons, dash patterns and so on. The overall challenge of this work is to translate the vector geometry into triangle-based geometry, so that the graphics processor can render it efficiently.

The main questions researched in this thesis are in what ways the traditional tile-based structure of interactive map services can be modified to enable features like map rotation and a 3D mode and whether and how it is possible to simulate a vector rendering engine like Apple's Quartz 2D using the tool set provided by a 3D graphics API like OpenGL ES. The new hybrid architecture is presented as a way to answer the first part of this question; for the second part, a number of different methods has been developed, which are described in the next section.

[2]Comparing the texture fill rate – the PowerVR SGX543MP4 used in the Apple iPad 3 and iPhone 5 achives 1900MT/s; a 2005 NVidia GeForce FX Go 5700 is able to do 1800MT/s

1.3 Contributions

In this work, methods that cover various topics of computer science and real time rendering are presented, some of which extend previous work to enable new features. Those methods are quickly outlined in this section.

- HYBRID RENDERING ARCHITECTURE: To combine the advantages of tile-based and true realtime renderers, an architecture that splits a map into a static base map and a set of dynamic features is presented. The base map is rendered tile-based and the dynamic features are then overlayed to provide good visibility and readability in all map rotations and scales. No such architecture was previously presented in literature.

- MULTI-TILES: To avoid redundant loading operations, the novel concept of multi-tiles is introduced: A multi-tile is a virtual tile covering a square area of actual tiles. Multi-tiles always have power-of-two dimensions and addresses divisible by the side length of the square they cover so that each tile has a unique multi-tile it is assigned to.

- LINE RENDERING: An algorithm to render line strips of arbitrary width is developed. Independently from the line geometry, the line can be decorated with three different line cap styles and a user-defined stipple pattern can be applied to the line. Each dash is then rendered with the selected line cap style; additionally the line is fully antialiased with an arbitrary circularly symmetric filter. While rendering antialiased lines was previously possible, rendering of line strips and correct handling of line caps for every single dash is a new contribution.

- TILE SELECTION: To determine the tiles that are visible in a camera with perspective projection, the view frustum of the camera is projected to the map and a polygon rasterization algorithm is used to determine all tiles touched by the resulting trapezium. This method specifically exploits the fact that the map is organized in tiles to get an exact set of the covered tiles in contrast to the approximate solution of previous clipmap approaches.

- TEXTURE ATLAS PACKING: To speed up rendering of icons, the icon images are cached in a texture atlas. A novel HARMONIC-inspired [15] algorithm is presented that packs the icons into the texture atlas using a quadtree structure. While not providing better storage efficiency than existing online packing algorithms, the method is extremely simple and performs similar to HARMONIC because the icons are small and all similar in size.

1.4 Structure

The presented work is structured into 5 chapters: In Chapter 1, "Introduction", a short overview on the background of this thesis and on the motives for creating a hardware-accelerated map renderer is given, novel contributions are outlined and the structure of the thesis is currently being presented.

Chapter 2, "Related Work", gives details about the technological foundations this work is based upon, introduces Graphics APIs, gives a primer on geographical projections and map

rendering and then presents previous scientific work related to the contributions of this thesis: 2D packing algorithms, line antialiasing methods and several line dashing implementations are described and an overview on packing algorithms is given. Polygon tessellation and rasterization algorithms are introduced in the end of this chapter.

In the next Chapter, “Implementation & Own Contribution” (3), the rendering library that has been implemented as a basis for this thesis is presented in detail, starting with the development environment, the data format and an overview on the general architecture. Then, the four parts of the implementation are described, and relevant methods and algorithms are put in relation with the scientific work presented in Chapter 2.

In Chapter 4, “Results”, the performance of the renderer is compared to the previous implementation in terms of speed, features and quality. It is shown that the proposed rendering library is able to render tiles much faster while providing more features and some minor quality improvements. Afterwards, the architecture of the system is compared to out-of-core rendering, and the line drawing algorithm is analyzed with regard to previous line rendering methods.

Finally, Chapter 5, “Conclusions”, sums up the presented work and the most challenging parts of the implementation. Some lessons that have been learned during the implementation are briefly presented and, as a last part, remaining tasks to make the renderer a release candidate are named and ideas on how they could be implemented are discussed.

CHAPTER 2

Related Work

The aim of this chapter is to provide an overview over the technologies and techniques used, as well as to present other scientific work related to the topic of this thesis. First, the technological foundations will be introduced.

2.1 Graphics APIs

In this section, the graphics API used in the proposed implementation, OpenGL ES 2.0, will be described along with a brief description of its "parental" API, OpenGL, and a short part about OpenVG, an alternative API. How to use OpenGL ES 2.0 with EGL (or EAGL on the iOS platform) is outlined at the end of this section.

2.1.1 OpenGL

OpenGL is a platform-independent application programming interface (API) for 3D graphics. On desktop computers, two major 3D APIs exist, DirectX and OpenGL. DirectX is only available on Microsoft Windows and Microsoft's game consoles, while OpenGL can be used on many different platforms, including Windows, Linux, MacOS X and various other UNIX-flavored operating systems. OpenGL was introduced by Silicon Graphics Inc. (SGI) in 1992 and is currently managed by the Khronos Group [42]. Since then, OpenGL has evolved to its current 4.3 version. The OpenGL API is implemented by hardware vendors according to the OpenGL specification; applications that use OpenGL link against a vendor-supplied library.

OpenGL is designed as a state machine and most of the API-calls modify the state in some way. The state is stored in the OpenGL context, that means if, for instance, a feature is enabled with a `glEnable` call, it stays enabled until it is disabled with a `glDisable` call or the OpenGL context is destroyed. When a context is created, it is initialized to a default state.

Applications that use OpenGL include some computer games like the Quake and Doom series, graphical user interfaces (e.g. on MacOS X) as well as digital content creation tools (Maya, Softimage|XSI) [23].

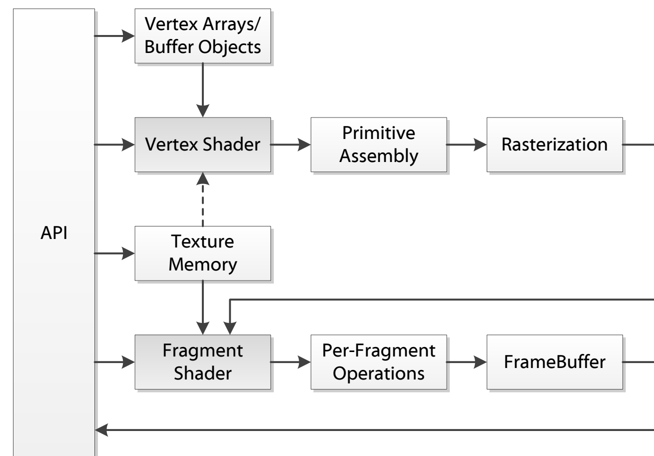


Figure 2.1: OpenGL ES 2.0 Graphics Pipeline (drawing from [23])

2.1.2 OpenGL ES

The wide acceptance of the OpenGL API and the availability of graphics processors in handheld devices made the introduction of a special OpenGL version for those devices reasonable. OpenGL ES was created by streamlining the OpenGL API, duplicate functionality was removed and the constraints of mobile devices, i.e. CPU power, memory and memory bandwidth, were taken into account. While OpenGL ES is a subset of OpenGL, compatibility to OpenGL was preserved with the goal that any application that uses only the reduced subset of OpenGL can also run on OpenGL ES [23].

Four API versions have so far been released by the Khronos Group: OpenGL ES 1.0 and 1.1 are derived from OpenGL 1.3 and 1.5, respectively, and implement a fixed-function pipeline; OpenGL ES 2.0 was written against OpenGL 2.0 and introduced a programmable pipeline. In contrast to OpenGL 2.0, where both the fixed-function- and the programmable pipeline are available, OpenGL ES 2.0 does not have a fixed-function pipeline anymore and is therefore incompatible to OpenGL ES 1.x, though the fixed-function pipeline can be simulated using shader programs. OpenGL ES 3.0 was recently introduced at SIGGRAPH 2012 and is derived from the OpenGL 3.3 and 4.2 specifications [13].

This thesis uses OpenGL ES 2.0 due to its wide hardware support at the time of writing.

2.1.3 OpenGL ES 2.0

The OpenGL ES 2.0 rendering pipeline can be seen in Figure 2.1. The programmable stages of the pipeline are shaded in gray; the dashed arrow denotes an optional feature, which does not have to be supported by a hardware vendor. The different stages of the pipeline will now be discussed briefly in the order they occur. However, this section is only intended to give a quick

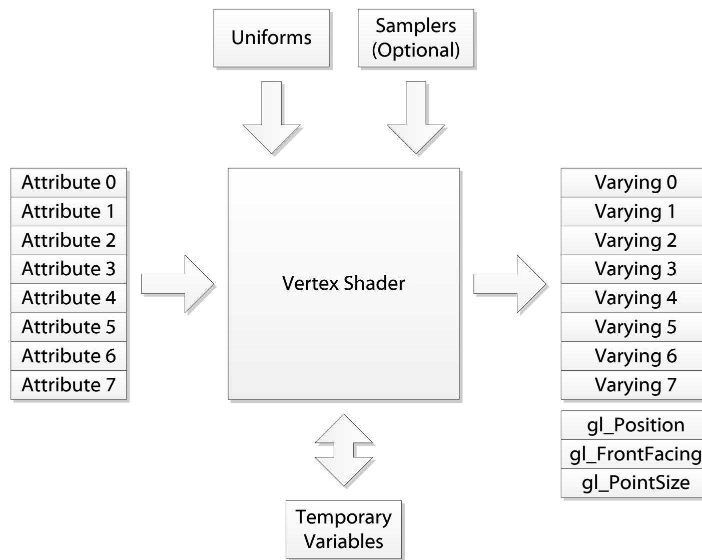


Figure 2.2: OpenGL ES 2.0 Vertex Shader (drawing from [23])

overview, many details have therefore been omitted. For more details, please refer to Munshi et al. [23].

Vertex Shader

In computer graphics, a point, either on its own or as a part of a larger primitive like a line or a triangle, is called vertex. A vertex has a position in 3D space and can be accompanied by attributes, e.g. a normal vector, texture coordinates or a vertex color. A single vertex with all its attributes is the input to the vertex shader, the first programmable stage of the rendering pipeline. Since there is no fixed-function pipeline in OpenGL ES 2.0, it is mandatory that the application uploads, compiles, links and activates a vertex shader program, written in the C-like OpenGL Shading Language (GLSL). When a draw call is sent by the application, the vertex shader program is run for every vertex of the active vertex array and the attribute variables of the program are initialized with the vertex attributes of the current vertex. A vertex shader usually needs to have at least one attribute variable: the position of the vertex.

Additionally, uniform variables can be passed to a vertex shader to represent constant data, like the position of the light source in a scene. As shown in Figure 2.2, a vertex shader might also be able to access texture memory using samplers (special uniform variables representing textures), but this feature is not enforced by the specification.

The output of a vertex shader is again a vertex, but it does not have to consist of the same properties as before the execution of the vertex shader. There is no fixed mapping from input-vertex-properties stored in attribute variables to output-vertex-properties stored in varying vari-

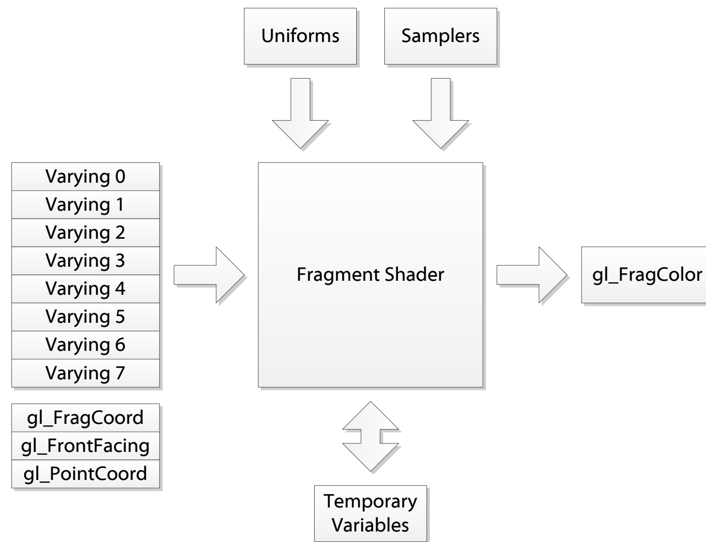


Figure 2.3: OpenGL ES 2.0 Fragment Shader (drawing from [23])

ables. These variables are later interpolated for each generated fragment in the rasterization stage.

Primitive Assembly

An OpenGL ES primitive is either a line, a point sprite or a triangle. Depending on which primitive mode is currently in use, the primitive assembly stage takes one, two or three vertices (i.e. the varying variables written by one/two/three execution(s) of the vertex shader) and constructs a primitive. The primitive is now tested against the view frustum; if it is completely outside of the frustum, it is discarded, if it is partially outside, it is clipped. The next step is to transform the primitive into screen coordinates and optionally cull it based on its orientation (backface-culling). If the primitive is not culled, it is passed to the rasterization stage.

Rasterization

In this stage primitives are transformed into fragments. A fragment can be seen as a data structure consisting of a pixel position on the screen, a depth value and auxiliary attributes defined by the varying variables of the vertex shader. Each primitive is rastered into a number of fragments and each attribute (written into the varying variables by the vertex shader) is interpolated accordingly. The rasterization itself depends on the primitive type, but the result of the rasterization stage is always a set of fragments for each primitive.

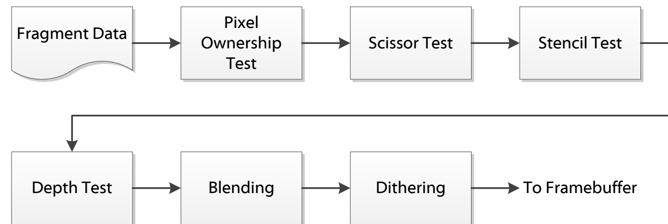


Figure 2.4: OpenGL ES 2.0 Per-Fragment Operations (drawing from [23])

Fragment Shader

The second programmable part of the rendering pipeline is the fragment shader (Figure 2.3) – as for the vertex shader, an OpenGL ES 2.0 application cannot run without a fragment shader program in place. The program is executed once for each fragment the rasterization stage creates. The high-level input of a fragment shader is a fragment, encoded in a number of varying variables. Like the vertex shader, the fragment shader may have uniform variables and samplers, but in contrast to the vertex shader stage, the availability of samplers in the fragment shader stage is guaranteed by the specification. The output of the fragment shader is a single four-dimensional vector – the final color of the fragment. Note that there can be an arbitrary number of fragments per pixel, because each primitive that is rasterized may touch any pixel on the screen. The final color depends on the ownership-, scissor-, stencil- and depth tests, as well as on blending and dithering; stages that are described in the next section. Generation of too many fragments, for instance due to excessive overdraw, can impact performance and must be avoided.

Per-Fragment Operations

After the fragment shader, a number of per-fragment operations take place, as can be seen in Figure 2.4: the pixel ownership test determines, whether the pixel in question currently belongs to the OpenGL context or if it is hidden by, for instance, another window. This is the only test that cannot be manipulated by the application, all following stages can (at least) be turned on and off. In OpenGL it is possible to set “scissors”, a rectangular mask to prohibit framebuffer writes outside of the mask. The scissor test queries the mask and discards the fragment if the scissor test fails. The stencil and depth tests read the stencil and depth buffer, respectively, the function used to decide whether to discard the fragment can be defined by the application. Blending uses a function to mix the color of the current fragment with the color of the framebuffer at the fragment’s position; again the function can be set by the user. Finally, dithering can be used to improve image quality on displays with a reduced color space by simulating unavailable colors with patterns of available colors.

When the per-fragment operations stage is completed and the fragment has not been discarded by any of the tests, the fragment color, depth and stencil values are written to the color, depth and stencil buffer, respectively. However, none of those buffers is required to exist and

Figure 2.5: The OpenVG Tiger reference image

writing to them can be controlled by write masks (e.g. for disabling depth writes for a certain object).

Finally, the contents of the framebuffer may be read back into the application by another set of API calls, however the depth and stencil buffers cannot be read back.

2.1.4 OpenVG

Another API managed by the Khronos Group is OpenVG, the Open Vector Graphics library. The first stable version, 1.0, was released in 2005 [37] and has evolved into version 1.1, released in 2008 [14] since then. The OpenVG API is very similar to OpenGL ES 1.x, but instead of 3D graphics, OpenVG focuses on 2D vector graphics. Rost and Rice [37] mention SVG viewers, portable mapping and e-book readers as target applications, which make OpenVG seem optimal for the task of displaying hardware accelerated vector maps on mobile devices. The fact that OpenGL ES 2.0 was preferred over OpenVG in this thesis has several reasons.

- Hardware support for OpenVG is currently very rare. None of the Apple iOS devices that have to be supported, due to the cooperation with Ulmon GmbH, have hardware accelerated OpenVG. There are, however, mobile Graphic Processing Units (GPUs) with OpenVG support, like the Qualcomm Adreno GPU that is used in many Android devices [34]. The Android OS does not provide any support for OpenVG though.

- Software Renderers are not fast enough yet. Lee et al. [16] provided a reference implementation of OpenVG in 2007 which rendered the Tiger reference image (Figure 2.5) in 3.6 seconds on a fairly modern desktop computer. Although being about twice as fast as the previous reference implementation, this is not suitable for realtime performance on mobile devices.

- Implementing the OpenVG API on top of OpenGL ES 2.0 was considered not feasible as only a small subset of OpenVG would be actually used and the complexity of another layer of middleware was to be avoided. However, some of the algorithms presented in this work could very well be used to provide such an implementation.

2.1.5 EGL

To issue OpenGL commands, a valid OpenGL context is needed. With regular OpenGL, the context is created using platform-dependent APIs like WGL on Microsoft Windows or GLX for the X Windowing System. With OpenGL ES, there is a Khronos API, EGL, to handle this. EGL is capable of querying the available display devices and initializing them. Rendering surfaces can be created according to the display's capabilities and can be shared among multiple Khronos APIs (e.g. OpenVG and OpenGL ES). Last but not least, OpenGL contexts can be created and attached to a rendering surface.

Rendering with multiple OpenGL contexts

Since OpenGL is a state machine, it is not thread safe, so rendering must always occur single threaded. To overcome this limitation, it is possible to create multiple OpenGL contexts (each with its own state) and set the desired context to be *current* on the current thread. By itself, this is not very useful, because the two OpenGL contexts are not related in any way and rendering to the same rendering surface will lead to undefined behavior. It is, however, possible to make multiple OpenGL contexts *shared*, meaning that they share the same "address space" so that all data that one context creates is available in all other contexts. A common use case for this is to have a single rendering thread that draws to the back buffer and one or multiple threads that create content. Most of the time, this is done by uploading textures that the main rendering thread can then use. That way, costly operations that would otherwise lead to low frame rates in the main loop can be done in the background, which is exactly the approach that the proposed system uses.

EAGL

Since most of the implementation for this thesis was done on the Apple iOS platform, the EAGL API had to be used instead of EGL. EAGL is Apple's adaption of EGL, implemented in Objective-C. Unfortunately, it is not possible to create OpenGL contexts in C/C++ code on iOS, so this has to be done outside of the rendering library.

2.2 Map Rendering

This section will introduce the basics of coordinate systems and map projections before describing the OpenStreetMap project that is used as a data source for this work. Next, two OpenStreetMap renderers will be presented as an example of how OpenStreetMap data can be converted to a map. As a last topic, the CPU-based rendering library to be replaced by this implementation is briefly outlined.

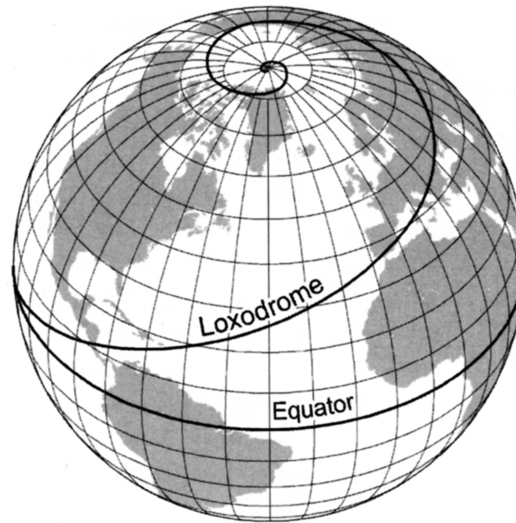


Figure 2.6: A loxodrome visualized on the globe (drawing from [20])

2.2.1 Geographic Coordinate Systems

A geographic coordinate system is used to uniquely address any point on the surface of the earth. Because the earth is not a perfect sphere, its shape has to be approximated mathematically. The most common approximation is given in the World Geodetic System 1984 (WGS84) [22]; the WGS84 contains a reference ellipsoid with the main axes pointing at the poles and at the prime meridian. The size and shape of the ellipsoid are defined by the semi major axis length of $a = 6378137.0$m and a reciprocal of flattening $1/f = 298.257223563$.

The most widely used addressing scheme is the latitude/longitude (lat/lon) scheme. As there are multiple reference ellipsoids, a position given by latitude and longitude is only unique if the used reference ellipsoid is known. This thesis exclusively uses the WGS84 reference ellipsoid because it is widely accepted; it is, for instance, used by the GPS system. The latitude (ϕ) of a point is the angle between the equatorial plane and the normal of the reference ellipsoid passing through the point and therefore ranges from $-90°$ at the south pole to $+90°$ at the north pole. Any point with a latitude of $0°$ lies on the equator. The longitude (λ) of a point is the angle between the plane containing the prime meridian and the normal of the reference ellipsoid passing through the point; it therefore ranges from $-180°$ to $+180°$. Points with a longitude of $-180°$ lie on the same meridian as points with a longitude of $+180°$. The point with $(\phi, \lambda) = (0°, 0°)$ lies in the Atlantic ocean, about 625km south of Ghana.

The Mercator Projection

Since the earth is (approximately) ellipsoidal, it is not possible to unroll its surface to a flat plane without distortion. Methods that describe this unrolling process are called map projections. An old and popular map projection is the Mercator projection.

To understand why this particular projection was much needed at its time, we have to look

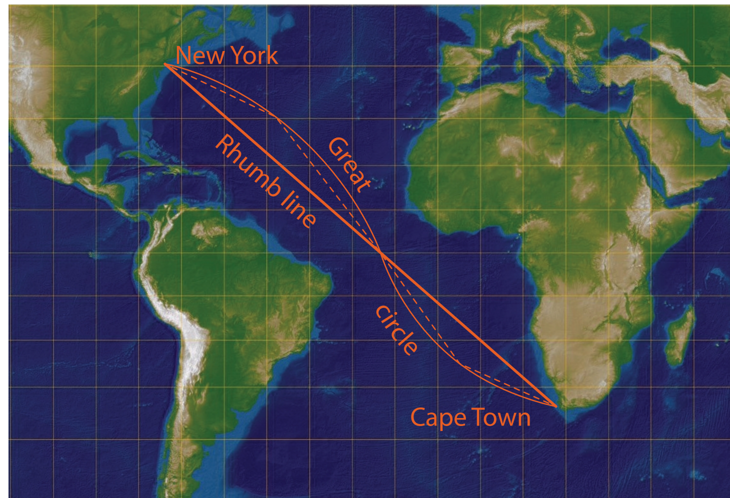


Figure 2.7: A loxodrome and a great circle visualized on a Mercator projected map. Note that the great circle is the shorter route (image from [36]).

at how navigation in the sixteenth century worked. Sailors were equipped with not much else but a compass and had no means of determining their current location on the sea. The easiest way of navigating under such circumstances is to have a fixed course (e.g. 270° for west) and constantly sail this direction. Such a direction, on a vessel, is called a "bearing". A ship that cruises with a fixed bearing crosses each meridian at exactly the same angle; the course is then called a loxodrome, or a rhumb line[1][20]. It must however be noted that this course is not a straight line, and not the shortest possible course either, which is denoted by the *great circle*, so called because great circles are the largest circles one can draw on a sphere; the shortest possible path between to points on a sphere is always denoted by the great circle passing through both points. If a loxodrome is plotted onto a globe, it becomes a spiral towards one of the earth's poles (visualized in Figure 2.6), let alone two exceptions: a loxodrome with a bearing of exactly east or west will never reach a pole and a loxodrome with a bearing of exactly north or south will reach the pole on the shortest possible path. Only in those two special cases the path along the loxodrome will be equally long as the path along the great circle; for all other cases the loxodrome is a detour. This fact can easily be understood when a loxodrome and a great circle, both passing through the same two points, are plotted onto a globe – as already mentioned, the loxodrome will end up as a spiral to one of the poles, connecting the two points with an arc, while the great circle will connect the points on the shortest possible way.

A special feature of the Mercator projection is that loxodromes can be drawn as straight lines. While the great circle is in fact the shorter course, the loxodrome is much easier to sail because of its constant bearing. To combine both the short way and the ease of navigation, modern sailors often approximate the great circle with a series of loxodromes. Figure 2.7 shows both a loxodrome and a great circle plotted on a Mercator projected map – the loxodrome looks

[1] *Loxodrome (rhumb line):* "A line that intersects all meridians at the same angle" [20].

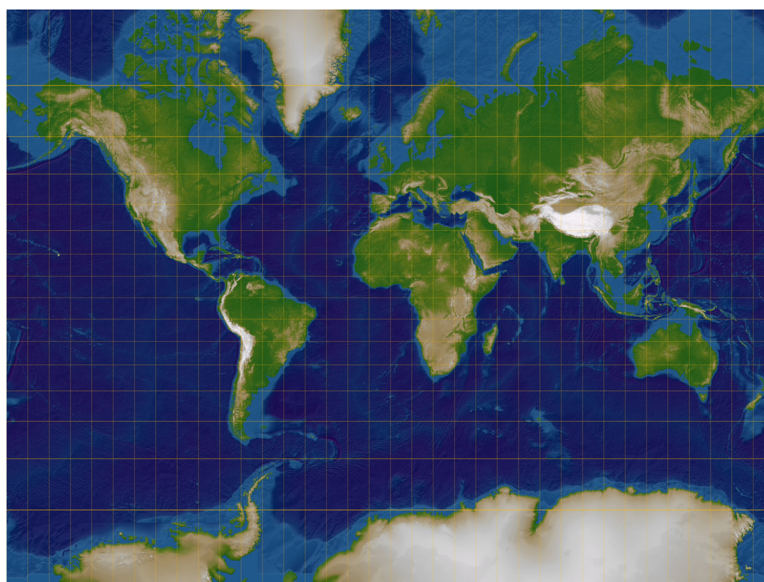


Figure 2.8: A world map in Mercator projection with a ten-degree gratitude (image from [36]).

"deceptively shorter than either the great-circle route or its multi-rhumb approximation" [20]. Following the definition of a loxodrome (it passes all meridians at the same angle), it can be seen that meridians must be parallel straight lines on a Mercator projection; it can also be understood that the poles are extended into infinity. Figure 2.8 shows a world map in Mercator projection, overlayed with a grid of a constant distance of ten degrees between adjacent lines (a ten-degree graticule). The top-most parallel that is shown is $\lambda = 70°$. The severe area-enlargement towards the poles can easily be explained: the circumference of the earth at the $60°$ parallel is about half of the circumference along the equator, but since all meridians must be parallel in a Mercator projection, distances along the $60°$ parallel must be drawn twice as large as distances along the equator. This leads to misconceptions, for instance Greenland looks larger than China even though it is in fact about 4.4 times smaller. The Mercator projection has been heavily criticized because many developing countries in the tropics look small compared to industrial nations in the temperate zone [20].

The "Google Projection"

The Google projection, or "spherical Mercator projection" is a special case of the Mercator projection and has become popular because both Google Maps and Bing Maps use it in their online services. It uses lat/lon coordinates according to the WGS84 reference ellipsoid, but projects the coordinates *as if* the reference ellipsoid was a sphere with the semi-major axis length of the WGS84 reference ellipsoid as radius. This simplification is done to decrease computational complexity. A typical world map that uses the Google projection has its origin at $(\phi, \lambda) = (0°, 0°)$ with the x-axis pointing east and the y-axis pointing north. Due to the aforementioned simplification, the conversion between latitude/longitude and map units (meters) is very simple with

spherical trigonometry [1]:

$$
\begin{aligned}
x &= lon * 20037508.34/180.0 \\
y &= \log\left(\tan\left((90 + lat) * \pi/360\right)\right) / (\pi/180) \\
y &= y * 20037508.34/180.0
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
lon &= (x/20037508.34) * 180 \\
lat &= (y/20037508.34) * 180 \\
lat &= 180/\pi * \left(2 * \arctan\left(\exp\left(lat * \pi/180\right)\right) - \pi/2\right)
\end{aligned}
\tag{2.2}
$$

Since the poles cannot be displayed in a Mercator-projected map, the maximum plotted latitude has to be defined – for simplicity, both Google Maps and Bing Maps use $\phi = 85.05113°$, because at this value the map becomes exactly square.

The Tile Pyramid

Traditionally, online mapping services use pre-rendered images to display maps. To reduce the amount of transferred data, the map is split into tiles (square images of equal size) that are seamlessly displayed next to each other. Additionally, zoomlevels are introduced because the number of tiles would otherwise quickly increase when zooming out. The tiling system that is used by all major online map providers (Google Maps, Bing Maps, Yahoo Maps, OpenStreetMap, OpenAerialMap) [29] covers the whole world in a single tile at level 0 and quadruples the number of tiles in each level while the resolution of a tile image stays the same at all times. A specific tile can then be addressed with an integer triple $(x, y, zoomlevel)$, with $(0, 0, zoomlevel)$ being the north-west tile of $zoomlevel$. That way a tile pyramid (Figure 2.9) results, where each level consists of $2^{level} \times 2^{level}$ tiles, which can be exploited to easily find the parent or the children of a given tile (even across multiple levels) by simply bit-shifting the tile coordinates. The number of zoomlevels differs, in this thesis the most zoomed-in level is level 18.

2.2.2 OpenStreetMap

Founded in 2004, the OpenStreetMap project aims to provide "a set of map data that's free to use, editable, and licensed under new copyright schemes" [11]. Like Wikipedia, OpenStreetMap is community driven; every registered user is allowed to edit and extend the map. Much of the map is generated by uploading GPX (GPS eXchange) tracks, but out-of-copyright or public-domain maps have also been imported. Another data source are aerial images: map features are then manually tracked by users; the map of Baghdad, Iraq, was created that way, resulting in the most detailed online map of the city in 2008 [11]. Yahoo Maps and Bing Maps have both allowed the OpenStreetMap project to access their aerial images. Rendered tile images and the underlying data are available through public APIs, with a creative commons license making even commercial usage possible.

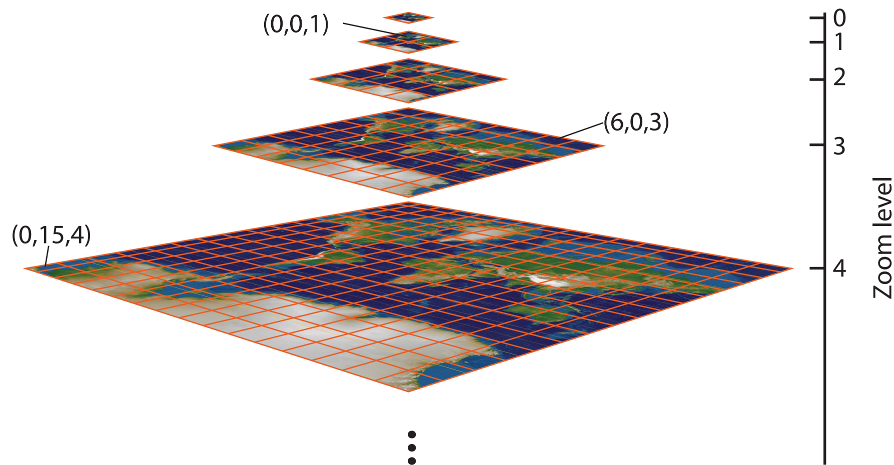


Figure 2.9: The first five zoomlevels of the tile pyramid. Tile addresses of the form $(x, y, zoomlevel)$ are given for three sample tiles (image from [36]).

Data Format

The underlying data format of the OpenStreetMap consists of points (nodes) that contain a time stamp, latitude/longitude and user information [11]. Apart from points, two additional feature types exist:

- **Lines** (ways), consisting of a list of references to nodes. Open ways denote linear features.
- **Polygons** (areal features), stored as a single, closed way.

Each node and way can additionally store an arbitrary number of key=value pairs. While both the key and the value are allowed to be anything, there is a set of keys and values that have become common; most users stick to these attributes.

The data can be downloaded in various ways and formats, most notably the planet.osm format containing all the data of the OpenStreetMap project, encoded in an XML variant. A planet.osm is released weekly and, at the time of writing, it is about 22GB of compressed data (250GB uncompressed) [26].

2.2.3 Slippy Maps

"Slippy Map" is a term used to describe interactive map interfaces like Google Maps [11]. It is "slippy" because a seemingly endless map can be slipped into view from being hidden outside of the screen (like a huge printed map that slides into the view of a camera mounted above a table). Slippy maps are usually implemented using a thin client, for instance, a web application that downloads tile images from a map server; with the tiles being organized as a tile pyramid as described in Section 2.2.1. OpenStreetMap uses such a slippy map for its primary web interface,

driven by the open source AJAX library OpenLayers [11]. As usual, tiles are stored online on a tile server; the programs used to render these tiles are described in the following section.

2.2.4 OpenStreetMap Renderers

There are various map renderers for OpenStreetMap data, designed and used as tile renderers. While the proposed implementation has a different scope (it is not an actual tile renderer, but a real-time renderer), its design is similar to some existing renderers, especially to Mapnik. The two most important of those renderers will now be described in a few words.

Osmarender

Osmarender is a set of XSL[2] transformations to transform OpenStreetMap data, given in the XML-like format of the planet.osm files, to the SVG[3] file format. Osmarender can therefore not be used as a standalone application; an XSL processor is needed to execute the transformations. There is an Osmarender compatible implementation, *or/p*[4], written in Perl. It does not need an XSL processor, because the transformations are done directly in Perl and can for this reason be used as a standalone application. The Tiles@home project, a distributed tile rendering service, used or/p to render tiles for the OpenStreetMap project. It must however be noted that the output of both Osmarender and or/p is not a raster image but a vector image – to produce raster tile images, an additional SVG renderer is needed.

Mapnik

Mapnik is the main tile renderer used for the OpenStreetMap website. The architecture of the proposed implementation mostly follows the Mapnik architecture, so the most important structures are now briefly described. Mapnik uses a special stylesheet to store the map style, consisting of a set of *rules*, each with one or several *symbolizers*. *Rule groups* are ordered lists of rules. Each item on the map is represented by a *feature*, with each feature having exactly one rule group attached. When a symbolizer is executed on a feature, it reads information from the feature and renders it according to its own specification. To render a feature, all rules from the feature's rule group are read and each rule is checked whether it must be applied on the current zoomlevel. If the rule applies, each symbolizer of the rule is executed on the feature. This process is illustrated in Algorithm 2.1.

For instance, if a street must have a casing in a different color than the fill color, the street is first drawn in the casing color, then, a bit narrower, in the filling color. This duplicated drawing can be encoded entirely in the stylesheet with the architecture described.

There are several types of symbolizers, each using a different set of attributes from a feature:

- LINE SYMBOLIZER: draws a line, probably dashed.
- LINE PATTERN SYMBOLIZER: draws a line with a repeated picture on it.

[2]Extensible Stylesheet Language
[3]Scalable Vector Graphics
[4]**o**sma**r**ender/**p**erl

input : A list of features to render and a stylesheet
output: A map image.

```
1 foreach Feature f do
2     foreach Rule r of the RuleGroup rg of f do
3         if appliesOnCurrentLevel(r) then
4             foreach Symbolizer s of r do
5                 execute(s,f); //Implementation depends on type of s
6             end
7         end
8     end
9 end
```

Algorithm 2.1: Simplified Mapnik rendering algorithm

- POLYGON SYMBOLIZER: draws a color-filled polygon.
- POLYGON PATTERN SYMBOLIZER: draws a polygon with a repeated picture on it (e.g. a tree to encode woods).
- POINT SYMBOLIZER: draws a point, either single colored or as a picture.
- TEXT SYMBOLIZER: renders text, either along a line, next to a point or in the middle of a polygon.
- SHIELD SYMBOLIZER: renders shields, i.e. small text items inside an image border. Used mostly for highways.

Mapnik is able to use two different rendering libraries to do the actual drawing: AGG[5] and cairo[6], both of which are CPU-based vector drawing libraries.

2.2.5 The Ulmon GmbH libMapRenderer

As already mentioned, the implementation this work is based upon was written as a replacement for another rendering library: *libMapRenderer*. The legacy engine is a pure tile-based renderer that creates tile images directly on the mobile device; it was originally written for the iOS platform, however, a not-feature-complete port to Android exists. A thin client uses a slippy map view to display the tiles and manages a rendering queue that is used by the server thread. The determination of the visible tiles is trivial, since only a non-rotating 2D view of the map is supported: the upper left and lower right tiles are calculated and the rectangle spanned by those two tiles contains all visible tiles. On iOS, classes to do this are provided by the operating system, on Android the mapsforge[7] library is used. The general architecture is very similar to Mapnik; the same notions of features, rules and symbolizers are used to render the map tiles. Like in

[5] Anti-Grain Geometry, `http://www.antigrain.com`
[6] `http://cairographics.org/`
[7] `http://code.google.com/p/mapsforge/`

Mapnik, the actual drawing is outsourced to the Quartz 2D vector drawing toolkit or the AGG library on iOS and Android, respectively, that means that all rendering is done on the CPU, the GPU remains unused.

2.2.6 Other Popular Mapping Applications

Both smart phone operating systems Android and iOS have their own, vector based mapping application, Google Maps and Apple Maps, respectively. The older of the two, Google Maps, was originally released in 2007 as "Google Maps for Mobile", a Java application compatible to any Java ME enabled mobile device. At that time, the application could not display vector maps but downloaded static tile images from a tile server. Since the release of Android in 2008, Google Maps was continuously extended, voice commands, traffic reports and support for Google StreetView™ were added in 2009. Vector based rendering was introduced in 2010. Along with vector rendering, support for rendering transparent three-dimensional buildings was introduced. By 2012, Google Maps applications are available on almost every smart phone platform, including Blackberry, iOS, Nokia S60 and Windows Phone.

Apple Maps, released in September 2012, replaced Google Maps on iOS devices. The most prominent feature, the satellite-image-enhanced 3D mode, was not well received by many customers, as it contained lots of errors; for instance, the Brooklyn Bridge in New York looks as if it was collapsed. Apple Maps is also criticized for lack of details and errors in the map data.

2.3 Out-of-Core Rendering

Out-of-core rendering describes the process of rendering large amounts of data, more specifically data that is too large to fit into graphics memory, or not even into the main memory of a computer. In this context, virtual texturing will be described, as it handles rendering very large textures, a problem very similar to map rendering. A map can be seen as a single, very large texture covering the planet. The following introduction to mipmaps, a fundamental concept in computer graphics, will leads directly to the clipmap, the main data structure used in virtual texturing.

2.3.1 Mipmaps

Mipmaps where introduced in 1983 by Williams [52]. To achieve good visual quality of textured 3D objects, it is not always sufficient to evaluate a single texel[8]; often the texture must be evaluated over a window of view-dependent shape. Evaluating such windows in realtime is expensive, so they can be approximated using a prefiltered set of images, the mipmap. A mipmap is calculated by repeated down sampling of the source image to half its size, resulting in a pyramid with the source image as a base and a 1×1 pixel image at its top.

At runtime, a floating point level in the mipmap pyramid is calculated for each texture lookup – such that the screen-pixel-to-texel-ratio is 1:1. Since the mipmap pyramid has only discrete levels, the level closest to the calculated level is then selected for the texture lookup. Alternatively, when doing trilinear filtering, the two adjacent levels are selected, evaluated and then

[8] Texture Element, a pixel in a texture image.

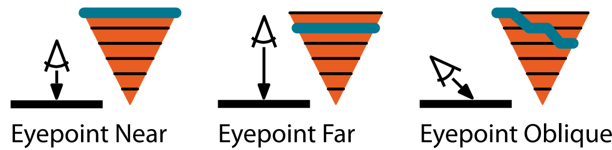


Figure 2.10: Texel Access within a Mipmap (drawing from [48])

interpolated using the fractional part of the calculated level. This reduces a banding artifact at the mipmap level change, especially visible when looking at planes at an acute angle. A view down a long corridor with textured walls, floor and ceiling is often used to illustrate this artifact.

2.3.2 Clipmaps

A clipmap, as introduced by Tanner et al. in 1998 [48], is similar to a mipmap, except that only a small portion of the whole mipmap actually resides in graphics memory. Tanner et al. aimed at using a texture showing the whole world in satellite photos at 1m/pixel resolution, an image being approximately $2^{26} \times 2^{26}$ pixels large. To do so, they built special hardware for the Silicon Graphics Inc. InfiniteReality™ machine proposed by Montrym et al. [21]. The usage of such a huge texture is possible because of the observation that only a small part of a very large mipmap will actually be used. Considering a mipmap with a 32768^2 texel base image on a 1024^2 pixel screen, at most 1024^2 texels will be accessed from a mipmap level before the adjacent level is used (except for trilinear filtering, where up to 2048^2 texels will be read). Figure 2.10 illustrates the lookup inside a mipmap: consider the center diagram where all texture lookups hit exactly level 1. Continuing our example, 1024^2 texels will be read from level 1. If the viewpoint moves a little closer to the plane and trilinear filtering is activated, samples will be blended from the levels 0 and 1, and the 1024^2 texels from level 1 map to 2048^2 texels in level 0.

A clipmap is defined as a mipmap where each level is clipped to a specified extent around a specified center. Mipmap levels which are smaller than the specified extent are fully included in the clipmap, resulting in an obelisk shape (Figure 2.11). The clipped levels form the clipmap stack, the levels that are included fully, form the clipmap pyramid.

The clip size is defined statically based on the window size, the clip center has to be recalculated in each frame. A proper selection of the clip center is crucial for an efficient usage of the clipmap; Tanner et al. [48] state that the selection is application-dependent, fly-over scenarios with a narrow field of view need to set the clip center far away at the distance whereas applications where quick turns may happen (such as head-tracked virtual reality) should set the clip center at the viewers foot position. After a new clip center has been calculated, the clipmap stack needs to be updated (imagine the stack to become slanted) – the clipmap pyramid, however, is kept at all times as a fallback if new data cannot be streamed into video memory fast enough. Tanner et al. used a toroidal addressing scheme to be able to update only a small part of each clipmap stack level. This addressing scheme calculates the actual texture coordinates from the

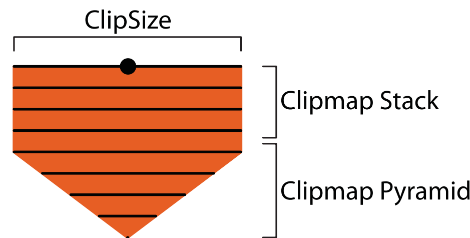


Figure 2.11: A clipmap with its clipmap stack and clipmap pyramid (drawing from [48])

virtual texture coordinates by adding the current clip center and taking the modulo clip size from the resulting value. As an offline preprocessing step, the whole mipmap is split into a tile pyramid as described earlier in this chapter (Section 2.2.1). The tiles are then read into main memory (as a second-level cache) and streamed into video memory when the clipmap stack needs to be updated.

Almost ten years later, Crawfis et al. [8] stated that the necessary hardware support for clipmaps was not present in a majority of the graphics cards and presented a clipmap implementation on consumer GPUs. Two methods for selecting the proper clip level at each pixel as well as a variety of clipping algorithms to clip the texture coordinates to the clip map levels were discussed, and an overview on the update cycle was given, which differs in details for the different shader programs. Performance evaluation resulted in frame rates between 100 and 200 frames per second on state-of-the-art (2007) consumer level graphics hardware. The scene used in the performance tests is a fly through scene over the San Francisco Bay, textured at $0.\overline{3}$m/texel resolution.

In parallel to Crawfis et al., Seoane et al. [41] implemented a similar system by using only the fixed-function pipeline; however a vertex shader can be used to calculate the texture coordinates more efficiently. For each clipmap stack level, a single texture is used, each of which contains mipmaps. While this causes redundant data to be held in graphics memory, it enables trilinear filtering without doing it manually in a fragment shader. Beside the wasted memory due to mipmaps for each clipmap stack level, the algorithm requires the actual geometry data to be subdivided to match the texture tiles, an approach both Tanner et al. [48] and Crawfis et al. [8] seek to avoid to decouple geometry from the texture. For certain applications, subdivided geometry may be acceptable though, and Seoane et al. achieve good performance using their algorithm.

A virtual texturing system similar to one used in a commercial game[9] is described by Barret [4], called Sparse Virtual Textures. This approach decouples the geometry from the texture; the geometry can be completely arbitrary, and the texture coordinates stored with the geometry reference the virtual texture as if it was not virtual. Barret introduces a novel way to determine the texture tiles needed using a two pass approach: in the first pass, the whole scene is rendered, but the fragment shader's only output is the tile address needed at each pixel. To calculate the

[9] MegaTexturing in id Software's *Rage*

mipmap level of the virtual texture, the mipmap calculation of the hardware has to be simulated. This can either be done by calculating the derivatives of the texture coordinates in screen space or by querying the hardware for the mipmap level it would have selected in a texture lookup. This is possible by using the OpenGL extension `ARB_texture_query_lod` [17]. An additional advantage of directly querying the hardware for the mipmap level is that the mipmap selection algorithm is not standardized by OpenGL, so different hardware vendors might use different algorithms. The result of the first pass is read back to the CPU to update the tile cache, which is implemented as a texture atlas, storing all tiles side by side. A texture atlas is a large texture, usually used to minimize state changes (such as texture switches) by combining many small textures into a single larger one. This tackles an important problem in modern rendering applications, as state changes often severely slow down an application. Another popular method of reducing state changes is to combine many small draw calls, such as very small objects, into a single, larger, draw call. The notion of a clipmap is not present anymore in Barret's approach; instead the – in some aspects much simpler – analogy to virtual memory is used. A page table texture, consisting of a single texel for each virtual texture tile, is used to store a pointer into the texture atlas, to a tile that currently has data for the virtual tile. This tile in the texture atlas may be the virtual tile itself, but it may also be any of its parents, if the virtual tile is not yet loaded – that way it can be assured that a texture lookup will always yield correct data, even if it might be from a coarser level; the coarsest level of the virtual mipmap (a single tile) is always held in memory, similar to the clipmap pyramid in [48]. The page table itself also has mipmaps, each storing pointers into the texture atlas for the virtual tiles of the corresponding mipmap level of the virtual texture. In the second pass, the result from the first pass along with the page table is used to transform the virtual texture coordinates into physical texture coordinates inside the texture atlas and a normal texture lookup can be performed to yield the color of the fragment. Tri-linear filtering can be accomplished manually in the fragment shader by querying both mip levels of the virtual texture and combining them according to the fractional part of the mip level (retrieved from the first pass).

Another modern virtual texturing system is described by Andersson and Göransson [2] for the WebGL platform. Their implementation is very similar to the one described by Barret [4], but since the texture tiles are streamed over the internet, loading a specific tile takes even longer and an efficient management of the tiles is needed.

There are multiple approaches to optimize usage of the slow tile storage media; using the main memory as a second-level cache as well as pre-fetching tiles is crucial for the performance of a virtual texturing system. Although a tile that has not yet been fetched can easily be replaced by a lower resolution tile, this is not desirable as it leads to a visible loss of image quality [48]. There are multiple strategies for determining tiles that are likely to be needed in the upcoming frames, Neu [24] investigates heuristics based on current camera motion; but also trivial approaches, like increasing the size of the viewport in the first pass (such that there is a "prefetch border" around the screen) work well under certain circumstances [2].

2.4 Texture Atlas Packing

In the previous section, it was mentioned that it is more efficient to pack many small textures into a single larger texture than using separate textures. This combined texture, the texture atlas, can be generated in various ways which can be divided into offline and online algorithms. The process of inserting items (images) into a texture atlas is a rectangular bin packing problem. Offline algorithms, as the term suggests, are run as a preprocessing step; an important criterion is that all items which must be included in the texture atlas are known beforehand. Online algorithms are designed to insert one item at a time, without knowledge on any future items and without ever moving an item which has already been placed [19]. In the proposed work, a simple online packing algorithm is used to create a texture atlas holding all the icons used in map rendering; this is why offline packing algorithms will not be discussed in this section. The problem can be narrowed down further, because in our case, the available space is bounded (to the size of the texture atlas) and we only have to work on square items in 2D.

The first bounded space online bin packing algorithms (for the 1D bin packing problem), HARMONIC$_\mathrm{M}$, was presented by Lee and Lee [15] in 1985. The basic idea is to classify incoming items with sizes in the interval $(0, 1]$ by their size and pack them into one of M bins. The interval of a bin k is defined as $I_k = (1/(k+1), 1/k]$, $1 \leq k < M$; the last bin takes the interval $I_M = (0, 1/M]$, hence the name of the algorithm. HARMONIC$_\mathrm{M}$ was extended in the same paper [15] to REFINED HARMONIC with a better performance ratio compared to the optimal solution. The algorithm was later refined to MODIFIED HARMONIC and MODIFIED HARMONIC 2 in 1989 by Ramanan et al. [35] and to HARMONIC++ in 2002 by Seiden [40].

While the algorithms described aim at solving the NP-hard [19] problem of bin usage optimization, an optimal solution is not important for our task since we deal with very small square icons (resolutions in the current stylesheet range from 3×3 to 24×24) and are able to create a texture atlas with more than enough space to hold them, even if the icons are not perfectly tightly packed. Nevertheless, the principle of classifying items into harmonic bins according to their size is used in the proposed algorithm.

2.5 Line Rendering

Line rendering is an important part when rendering maps; without line rendering, streets and roads cannot be displayed, and without streets and roads, not much information remains on the map. Lines are often subject to aliasing, so methods of rendering antialiased lines are discussed at the beginning of this section; how different stylistic attributes of lines can be drawn is reviewed thereafter. OpenGL ES 2.0 offers the following rendering modes for lines:

- `GL_LINE_STRIP`: renders a continuous line along all the points that are passed to the draw call.
- `GL_LINE_LOOP`: like `GL_LINE_STRIP`, but an additional line is drawn from the last to the first point.
- `GL_LINES`: every odd point is considered a start point and every even point is considered an end point, so separate line segments are drawn.

Figure 2.12: A line, antialiased (at the top) and aliased (at the bottom)

Additionally, OpenGL ES 2.0 is able to rasterize lines with adjustable width, through the `glLineWidth` call.

2.5.1 Antialiasing

In signal processing, *aliasing* is an effect that occurs when analog signals are sampled at a rate lower than twice the highest frequency of the signal (Nyquist-Shannon sampling theorem, [25, 44]). When a band limited signal is sampled at twice (or more) its maximum frequency, it can be restored from its samples in a lossless way (i.e. no aliasing happens). When rendering lines, the transition from background color to line color and back is a square wave, which is a not-band-limited function and would therefore need an infinite sampling frequency. Since the sampling frequency of a display is limited by its pixel size, line rendering is often accomplished by sampling at the pixel centers and drawing either the background color or the line color, depending on whether the pixel center lies on the line or not. This type of sampling generates staircase effects as illustrated in Figure 2.12 which are especially visible when the line is animated, because the steps seem to move along the line then [18]. To avoid this artifact, the square wave has to be filtered with a low-pass filter to remove all frequencies higher or equal to half the sampling frequency (i.e. the pixel density); the resulting function can then be reconstructed perfectly according to the Nyquist-Shannon sampling theorem. Of course, the sharp edge of the mathematically defined line is then replaced by a smooth curve where the line color has to be blended with the background color, which is done using transparent pixels.

In regular (desktop) OpenGL, the `glHint` API exposes a way to let the OpenGL implementation take care of line antialiasing: `GL_LINE_SMOOTH_HINT`. The implementation is, however, not required to follow the hint. In OpenGL ES 2.0 this hint is not included, line antialiasing is therefore not supported at all. Nevertheless, antialiased line rendering is crucial for producing a visually appealing image.

One of the first line antialiasing methods was presented by Gupta and Sproull in 1981 [10]. In this paper, a conic filter is used to determine the intensity value for each pixel, which corresponds to the volume of the intersection of the filter cone with the line. Gupta and Sproull discovered that for circularly symmetric filters of radius 1, the pixel intensity can be written as a function only dependent on the distance of the pixel from the line center p and the width of the line t:

$$I = F(p, t) \tag{2.3}$$

This function F may encode any circularly symmetric filter. For any desired line width, a table can be precalculated, containing the intensity values for a number of distances. The drawing

algorithm itself is based on the line drawing algorithm by Bresenham [5]; the inner loop is extended to calculate the distance from the line center and perform the lookup to retrieve the correct intensity value for up to three pixels in each pixel row (a restriction for simplicity reasons, Gupta and Sproull state that the algorithm my be trivially extended to take more pixels into account).

To handle end points, intensities for six end pixels are precalculated for 16 different slopes and another lookup is performed at runtime. That way, different end point styles can be accomplished by using different end point tables.

Similar to [10], Turkowski [50] proposed a function to map from point-line-distance and line width to intensity. For handling end points, a point-segment-distance is suggested instead of the point-line-distance. While a mathematical line is infinite and the point-line-distance yields wrong results near the end points, segments only exist between the start and the end point of a line. Usage of the segment-line-distance results in round ends of thick lines and smoothed corners of line strips.

To calculate the point-segment-distance analytically, two cases must be distinguished: if the point is inside the infinitely wide line segment, (i.e. inside both half-planes defined by orthogonal lines through the end points), the point-segment-distance is equal to the point-line-distance. If the point is outside, the point-segment-distance is the Euclidean distance to the nearest end point. Alternatively, the point-segment-distance can be calculated by coordinate system transformations, in particular by rotating and translating the line segment such that one of the end points corresponds to the origin and the other end point lies on the positive x-axis. By transforming the point in question using this transformation, only its x coordinate has to be evaluated to determine whether the line-distance or the segment-distance has to be evaluated. Also the evaluation of the line-distance becomes much more efficient as the absolute value of the y-coordinate then corresponds to the line-distance.

Similar to Gupta and Sproull [10] and Turkowski [50], McNamara et al. [18] describe a prefiltering technique using a lookup table to map the pixel-line-distance to an intensity value for the pixel. The table can either be built into hardware or calculated once at runtime by filtering a prototypical line at several distances from the line center. McNamara et al. use 32 distances and store a 5-bit intensity value for each of them. To calculate the point-line-distance, a combination of four edge functions E_i, winding clockwise around the mathematical line, is used:

$$E(x, y) = (x - x_0) * \Delta y - (y - y_0) * \Delta x \tag{2.4}$$

Where (x_0, y_0) denotes the start-, (x_1, y_1) denotes the end point of the directed edge and $\Delta x = x_1 - x_0$ and $\Delta y = y_1 - y_0$ denote the direction of the edge. $E(x, y)$ is positive for points right of the edge, negative for points left of the edge and zero for points on the edge. Its absolute value is the distance of the point to the edge, scaled by the distance between (x_0, y_0) and (x_1, y_1). The correctly scaled distance function is therefore given as

$$D(x, y) = \frac{E(x, y)}{\sqrt{\Delta x^2 + \Delta y^2}} \tag{2.5}$$

which is then scaled according to the line width w and the filter radius r. A sample line, surrounded by its four distance functions, can be seen in Figure 2.13. It can be seen that the scaled

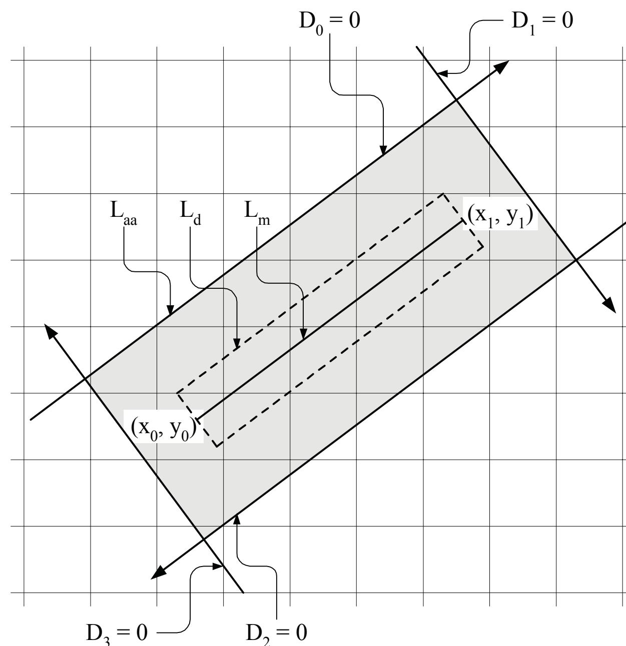
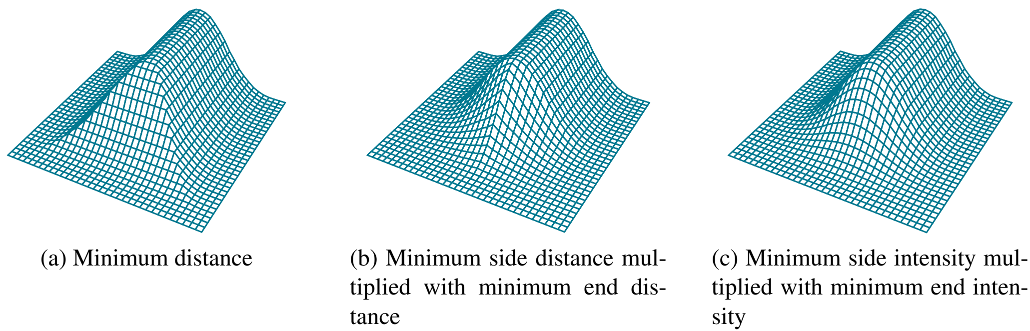


Figure 2.13: Four signed distance functions. The area where all of them are positive is called the antialiased line (L_{aa}). The mathematical line L_m and the desired line L_d are also shown. (plot from [18])

distance function does not depend on the actual values of w and r, but only on their ratio. In practice, it is sufficient to group the different ratios into ranges and use the same distance function for an entire range by limiting the ratio r/w to the interval $[0.5, 2]$. For each pixel, four distance functions can now be evaluated (a signed distance to each of the line edges); pixels with only positive distances will receive an intensity greater than 0, pixels with one or several negative distances (i.e. pixels outside of the rectangle defined by the line edges) are assigned an intensity of 0. Which of the four distances is used to lookup the intensity in the table directly influences the appearance of the line's end points (see Figure 2.14 for a comparison of the different line endings). The simplest solution of using the smallest distance results in a line ending that looks like a chisel with noticeable corners, which is not desirable. The slightly more complicated approach of using the product of the minimum of the distances orthogonal to the line and the minimum of the distances along the line gives a smoother line ending, but introduces a sharp cut in the middle of the line, which starts to be visible at a line width of 3 pixels, according to McNamara et al. [18]. The computationally most expensive approach of evaluating the intensities for all four distances and using the product of the minimum intensity perpendicular to the line and the minimum intensity along the line gives the best result, indistinguishable from the ground truth (where the line is actually convolved with the correct filter) [18].

A sample fragment-shader implementation of the method of McNamara et al. along with line and polygon antialiasing examples can be found in the article of Chan and Durand [6].

Qin et al. [33] propose a system to render antialiased glyphs directly from a vector repre-



(a) Minimum distance

(b) Minimum side distance multiplied with minimum end distance

(c) Minimum side intensity multiplied with minimum end intensity

Figure 2.14: Different line ending intensities (plots from [18])

sentation. A signed distance function is used to calculate the minimal distance of a point to the nearest point of the shape that is being rendered. To accelerate the calculation of this function, a Voronoi diagram overlayed with a sampling grid is used. The Voronoi diagram can be precalculated and the sampling grid is non uniform to take the complexity of a shape into account. To use anisotropic antialiasing, the signed distance function is then transformed from texture space to screen space using partial derivatives of the texture coordinates. The transformed signed distance function can then be used for antialiasing, using a cubic smooth transition function:

$$smoothstep(g) = 1/2 + g_c * (3/2 - 2g_c^2) \tag{2.6}$$

with g_c being g clamped to the interval $[-1, 1]$ [33].

Similar to the proposed approach (described in Section 3.2.4), Bærentzen et al. [3] render long, narrow rectangles instead of lines as in [6, 18], because of the line width restriction of both OpenGL and DirectX when using the line primitive. The vertex positions are calculated entirely in the vertex shader by using only the end points of the line and the line width. The fragment shader then calculates the distance from the fragment center to the line segment; how exactly this is done remains to speculate, but it can be assumed that a method similar to the one described in [50] is used.

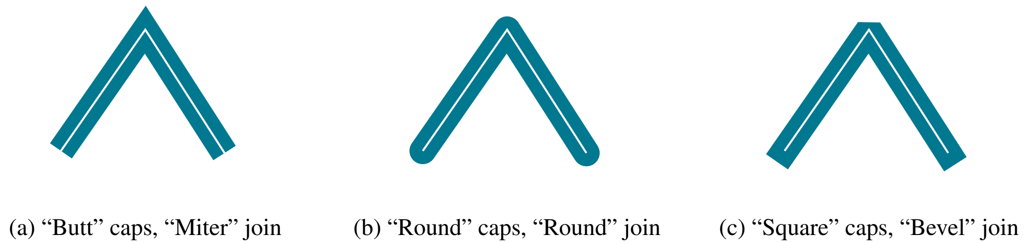
Not unlike [33], the intensity is directly calculated from the distance instead of using a lookup table with prefiltered values – the function used by Bærentzen et al. is the GLSL function $exp2(x) = 2^x$:

$$I = exp2(-2d^2) \tag{2.7}$$

with d being the pixel-line-segment-distance.

2.5.2 Join and Cap Styles

In vector graphics, three line cap styles and three line join styles have been established, which are illustrated in Figure 2.15. For acute angles, miter joins get very long, so the ratio of the line width to miter length, i.e. the distance from the tip to the inner corner, is usually limited; if the limit is exceeded, the miter join is clamped to a bevel join.



(a) “Butt” caps, “Miter” join (b) “Round” caps, “Round” join (c) “Square” caps, “Bevel” join

Figure 2.15: Different line caps and joins, the original path is shown in white.

Rendering antialiased line joins and line caps with OpenGL has not been covered explicitly in literature. However, all of the previously discussed papers about line rendering handle them partially. Gupta and Sproull [10] can render all three different line caps by modifying their lookup tables, and their approach could be extended to include lookup tables for each possible angle at a join. This would of course not be feasible, as the amount of data in the lookup tables would be huge and yet inaccurate. Turkowski [50] mentions that different line caps and joins can be drawn by modifying the point-segment-distance-calculation at line end points and illustrates how to render round caps and round joins. Although they do not mention it, the system proposed by McNamara et al. [18] is able to render both butt and round caps by modifying the intensity calculation from the four edge distances. What is called a “chisel” tip in the paper is in fact an approximation of a butt cap. McNamara et al. only render straight lines without any joins, so no join styles are discusssed. Chan and Durand [6] accomplish round joins by simply drawing each line separately with round caps. If the end point of the first line is equal to the start point of the second line this results in a correct round join. Qin et al. [33] render only closed shapes, so no end points occur, however miter and round joins can be rendered with their system by using different distance measures. A special “pseudo-distance” is defined for rendering miter joins; the pseudo distance is the “distance to the closest point on the infinite line containing the closest segment” [33]. If the “true distance” is used, i.e. end points are taken into account, round joins are rendered. Bærentzen et al. [3] render round caps by using the point-segment-distance for the intensity calculation, no other cap or join styles are mentioned. This approach was extended in this work to handle all three cap styles and miter style joins (Section 3.2.4). Finally, a previously unmentioned patent by Persson [28] shows how to tessellate lines into triangle strips while taking all join styles into account, which is used as a basis for the line tessellation in the presented work.

2.5.3 Dashing

Drawing dashed/stippled lines is an important feature when rendering OpenStreetMap data, because many commonly known map features (e.g. railroads) need to be displayed as such. Up to OpenGL 3.0, there was a `glLineStipple` API-call to specify a dash pattern, consisting of a 16-bit pattern and a scale parameter. This call has never been part of the OpenGL ES API, so its behavior has to be mimicked using the available features.

Seetharamaiah et al. [39] propose an algorithm to be implemented in graphics hardware to backup the `glLineStipple` API for unordered line rasterization by calculating the position of the current location relative to the line's end points. The stipple pattern, initial step count (the line does not have to start with the start of the stipple pattern) and the current location are then used by the fragment generator to decide whether a fragment should be created at the current position, or not. Of course, this binary decision is likely to introduce aliasing at the dash borders; to avoid this, the edges at the dash borders have to be low-pass filtered like the edges of the line itself, as covered in Section 2.5.1.

Poddar [31] introduced a method to render stippled lines by encoding the stipple pattern into a texture map. Intended to be implemented in a display driver, this is a software-only solution for backing up the `glLineStipple` call and enabling dashed line rendering without special purpose graphics hardware. A simplified version of this idea is used in the proposed line renderer (Section 3.2.4).

2.6 Polygon Rendering

Beside lines and points, polygons are the third important data type in map rendering. Polygons are used to visualize features like houses, but also larger areas like woods, national parks or even coast lines. To render polygons, they must be triangulated because modern graphics hardware is optimized to work on triangles and OpenGL ES 2.0 does not support a polygon geometric primitive. The second aspect discussed in this section is direct polygon rasterization, but not because it is needed to render polygons – as already mentioned, they are triangulated anyway – but because a polygon rasterization algorithm is used to determine the currently visible tiles in Section 3.4.2; i.e. the projected view frustum is rasterized into tiles, not into pixels.

2.6.1 Triangulation

Since polygons cannot be rendered directly, they have to be transformed into triangles prior to rendering. The tesselator used in the proposed implementation (the GLU Tesselator [7, 43]) is very robust; there are, however, algorithms that produce better triangulations (fewer triangles, no degenerated triangles with one angle close to $180°$, Delaunay-Triangulations), like Shewchuk's *Triangle* algorithm [45].

For the use case of map rendering, however, the robustness and speed of the GLU Tesselator is preferred over other higher-quality, but more complicated algorithms; furthermore the algorithm can process degenerated polygons as well as polygons with holes, both of which may occur in map data. For hole-handling, five different winding rules are supported. Although the input to the algorithm consists of 3D points, the algorithm essentially works in 2D; all input points are projected to a plane, and the projected points are then processed. As a first step, a line-sweep algorithm is used to partition the polygon into x-monotone regions, defined as follows: “Any vertical line intersects an x-monotone region in at most one interval” [43, in the file `alg-outline`]. The x-monotone regions can then be triangulated using a simplified version of the algorithm proposed by Preparata and Shamos [32]. As a last step, the resulting triangles

from all x-monotone regions are grouped together into a set of triangle strips and triangle fans if possible, otherwise separate triangles are returned.

2.6.2 Rasterization

As already mentioned, polygon rasterization converts a polygon into a set of pixels that are covered by the polygon and is therefore a very elementary process in rendering. Pavlidis [27] compared approaches to tackle this problem in 1981, characterizing them into two groups: polygon-based and pixel-based. Polygon-based algorithms work on a list of edges that must be sorted according to their maximum y-coordinate. A set of active edges intersecting the current scanline is maintained, sorted by their intersection point with the scanline. Rasterization can then be done by filling all pixels between the first and the second, the third and the fourth intersection and so on. Such an algorithm is described by Hearn and Baker [12, Chapter 4-10]; being built for arbitrary (even concave) polygons, some preprocessing is necessary apart from sorting the edges: for instance, certain edges must be shortened to avoid duplicate vertices. Pixel-based algorithms, on the other hand, do not require any kind of preprocessing, it is even possible to rasterize any kind of contour given a known seed point inside the contour. Starting from the seed point, the colored region "grows" until it is limited by the contour boundaries. Parity-based algorithms are an extension to pixel-based algorithms, where no seed point is required anymore.

A set of parity-based algorithms is described by Dunlavey [9], reaching maximum memory efficiency by trading storage for speed; no other storage but the framebuffer itself is needed. Each of the presented algorithms is optimized for a different kind of polygon, though there is no solution that handles every kind equally well.

Pixel-based algorithms work well for single-colored polygons, where each pixel does only have a binary property: being inside or outside of the shape. Shaded polygons, where each pixel (or fragment) is assigned a set of attributes like color or depth with different values based on the location of the pixel in the polygon, require a scanline algorithm (called polygon-based by Pavlidis [27]) to work. Such an algorithm is described by Pineda [30], a parallel scanline filling algorithm which uses linear edge functions as a mathematical parity test. Edge functions are defined to have the value zero on the edge and being positive on one, and negative on the other side of the edge, the same edge functions which were used by McNamara et al. [18]. Once the edge functions are defined for each edge of the polygon, it can be traversed in various ways. During traversal, the attributes of each fragment are interpolated; even though the traversal does not have to be done continuously, but by a set of independent interpolators, each processing a block of pixels. This is possible due to the linear nature of the edge function. A third useful property of the algorithm is that the sub-pixel accuracy of vertices can be fully preserved.

Finally, Rueda et al. [38] describe a very simple algorithm to directly render polygons on modern graphics hardware. This is done by creating a trivial tessellation of the polygon by connecting the end points of each edge with the centroid of the polygon. The resulting triangles are then rendered into a special "presence" buffer, which is only able to store a single bit for each pixel. Each write access to a pixel inverts its value. Since all edges (including edges of holes) are used to create the tessellation, fragments outside of the polygon are touched an even number of times (for instance twice for simple holes) while fragments inside the polygon are touched an odd number of times. The presence buffer (implementable in hardware using an

OpenGL stencil buffer) records exactly that property by design. A downside of the algorithm is that shading (i.e. interpolating certain values inside the polygon) cannot be combined easily with the algorithm, since the presence buffer only records if a pixel is touched by the polygon or not. Also, the algorithm is not efficient regarding the fragment fill rate, because for non-convex polygons or polygons with holes, many more fragments are rasterized than are actually contained in the polygon.

CHAPTER 3

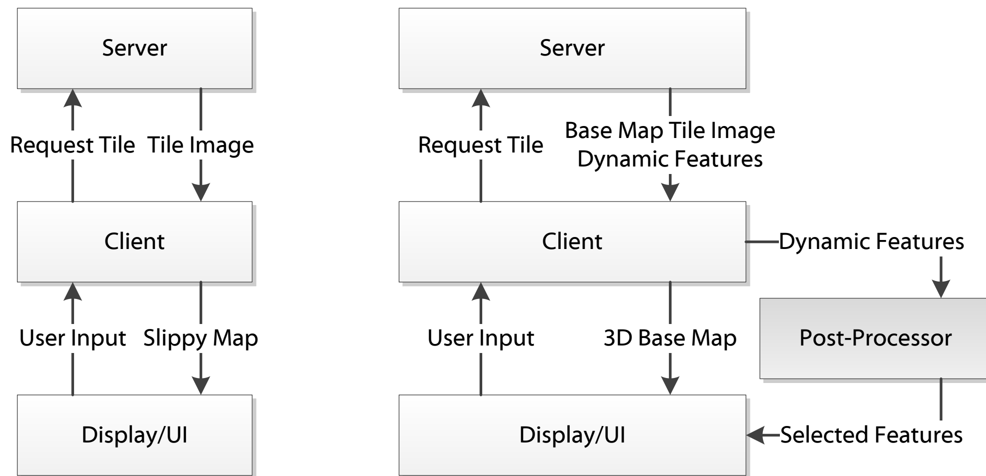
Implementation & Own Contribution

In this chapter, the rendering library implemented as a base for this thesis is described. Along with the overall description of the architecture, certain algorithms and methods will be described in detail and compared to previous solutions to the same problems.

3.1 The Hybrid Rendering Architecture

Since the problem of displaying a detailed map for the whole world is similar to the virtual texturing problem (described in Section 2.3.2), the solutions are also somewhat similar. In contrast to virtual texturing, the texture does not exist yet when the program is run, instead it is created on-the-fly, which is comparable to streaming texture tiles from very slow media and therefore makes the tile-loading-bottleneck even more severe. The original intention to render the visible portion of the map completely from scratch in each animation frame was quickly discarded because of the amount of data and the limited capabilities of the target devices. Instead, the architecture of Ulmon GmbHs libMapRenderer was extended to a hybrid between a tile-based renderer and a full realtime renderer, *libMapRendererGL*. The Mapnik styling architecture, used in the previous renderer, is also used in the proposed implementation; all data structures and symbolizers described in Section 2.2.4 about Mapnik are also present in the new rendering library. Map features can be divided coarsely into two groups: static and dynamic features. All static features together form the "base map", a map containing all street data, railroads, houses, woods, etc – basically everything but labels and points of interest (POIs). Like the old renderer, the proposed system implements a server-client architecture where both the server and the client run on the mobile device, but while the server in libMapRenderer produced final tile images, the server does now only work on static features, producing a base map. A side-by-side comparison of the old and the new architecture can be seen in Figure 3.1.

A 3D slippy map (Section 2.2.3) client is used to display the base map, and after the client is finished, the post-processer renders dynamic features directly to the screen, drawing over the result of the client (the base map). Dynamic features, so called because their position and



(a) A tile-based Rendering Architecture

(b) The proposed Architecture

Figure 3.1: A comparison of a standard tile-based Architecture with the new Hybrid Architecture.

orientation on the map is not fixed, include road names, POI icons and -labels and general map labels like country names. In the previous implementation, rotating the map was not possible, because the dynamic features were baked into the tile images, and rotating would have caused them to be upside-down eventually. In the new renderer, since dynamic features are excluded from the map tiles, the map can be rotated arbitrarily while dynamic features stay upright and readable at all times. Additionally, tilting the map into a 3D mode is possible without creating the impression of only skewing a map image, because POI icons and text stay parallel to the screen as if they were standing out of the map.

Technically, static and dynamic features can be distinguished by the symbolizer they are drawn with. Line, line pattern, polygon and polygon pattern symbolizers produce static data, point symbolizers have to be split according to the icon they render. Point symbolizers that draw trees (e.g. in an alley), mountain peaks, springs, railroad crossings or other low-level icons are treated as static data, while point symbolizers that draw POIs like post offices, restaurants, hotels or airports are considered dynamic.

As in the previous renderer, client and server are implemented as threads inside the same process, so they share the same address space. To handle communication, special data structures (described in Section 3.3) have been implemented; these include a rendering queue that is filled by the client and worked on by the server and a tile cache that manages all tiles that are currently available in the system. The rendering queue is sorted by priority, so the server can always work off the top of the queue. The tile cache keeps track about the last usage of each tile and uses this to implement a Least Recently Used (LRU) cache cleaning strategy. Since all rendering is implemented using OpenGL ES 2.0, a tile image is stored as a texture; as a tile image store, a

texture atlas could not be used because of the texture size restrictions on the target devices; the maximum texture size on current-generation iOS hardware is 2048×2048, which is not enough to store a sufficient number of tiles (see Section 3.2.3 for details on the tile size). To enable multi-threaded rendering, both the server and the client need to have their own OpenGL context, which are created to share the same address space as described in Section 2.1.5 (The post-processor is run in the same thread as the client, so it does not need its own OpenGL context). That way, the server can create a new texture when it renders a tile, and as soon as it is finished, the client can access the texture by simply using the same OpenGL texture handle the server used for accessing the texture. On a side note, the server must call the `glFinish` function before the client can use the texture, to make sure all rendering to the texture is finished.

In the following sections, the individual parts of the rendering library are described, beginning with the tile server.

3.2 Server

The servers task is to query the rendering queue for a tile to render, load its data from the file system, render all static data into the tile image and make the tile available to the client. From the rendering queue, the server gets a tile address (triplets containing x- and y-coordinates and a $zoomlevel$) of the next tile to render. The system is designed to work with multiple servers that process tiles from the same queue concurrently – of course each server needs to have its own OpenGL context and all contexts in the system must be shared. It must however be noted that the GPU is usually only capable of processing one command at a time, so even when rendering with multiple contexts, the actual processing of the commands happens in serial fashion.

The server never renders directly to the screen, instead, it creates its own frame buffer object (FBO) at startup. Optionally, the server can perform multi-sample antialiasing (MSAA) on the tile images, if it is supported by the hardware - the OpenGL Extension `APPLE_framebuffer_multisample` must be supported for this. As already mentioned in Section 3.1, each tile image is stored in a separate texture, so the server creates a new texture for each tile it renders and binds it to its FBO. Because the renderer (both libMapRenderer and the new implementation) support multiple installed maps (multiple ulmbin files, see Section 3.2.1), each installed map has to be checked whether it covers the tile that is currently rendered. Usually, none or only a single map covers a given tile; however, there are special cases where multiple maps contribute to a single tile image, for instance if a tile of a very high zoomlevel is rendered such that the maps of two cities that are near each other are inside the tile pyramid spanned by the children of the current tile.

When the list of maps that must be processed for the current tile has been determined, the server must load the data from each map (Section 3.2.2). The data is then split into static and dynamic data as previously described, and the static data is rendered (Section 3.2.4). All dynamic data is stored together with the tile image to be processed by the post-processor later. As a last step, mipmaps are created of the tile image to provide trilinear filtering (described in Section 3.4.3); they are generated by using the `glGenerateMipmap` API call.

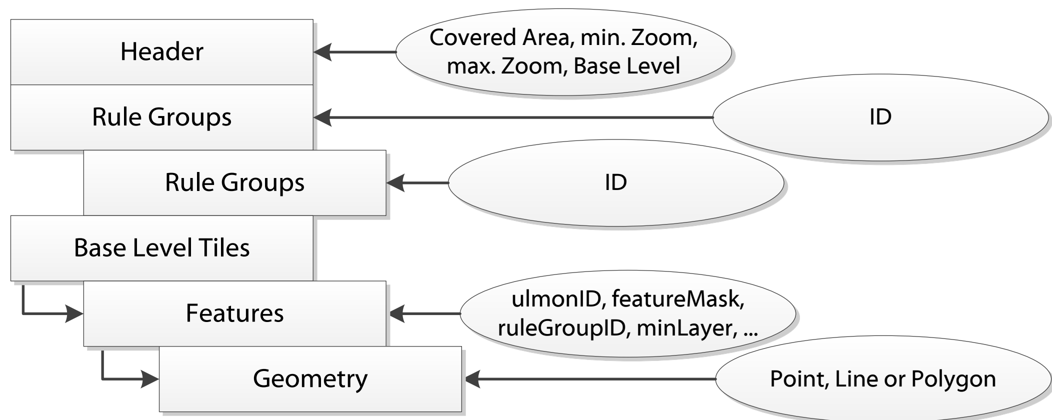


Figure 3.2: A simple map of the ulmbin file format

3.2.1 Data Format

The map renderer does not work directly on OpenStreetMap data – the data is first converted to a proprietary format defined by Ulmon GmbH. The format is closely related to the Mapnik rendering architecture (see Section 2.2.4) and it is split into two parts: the style information and the actual map data. Rules and symbolizers are stored in an SQLite-database; the *symbolizers* table contains all attributes of all symbolizer types and a discrimination column to identify the symbolizer type. Furthermore, each symbolizer has an ID as a primary key and a foreign key identifying the rule it belongs to. The *rules* table holds all rules of the stylesheet, consisting of an interval of zoomlevels the rule applies to, a layer ID and, to identify the rule, a unique ID. Layers in this context are similar to the layers found in many image processing applications, they are used to specify the order in which the rules must be used to ensure correct overdraw. Both the rules and the symbolizers are read into an in-memory data structure at application startup and are kept in memory until the program is terminated.

In contrast to the style information, the map data is stored in a special binary format, called *ulmbin*, whose format is quickly outlined in Figure 3.2. A map, in this context, denotes a single ulmbin file, containing map data for a certain rectangular region. To make positioning of a map in the global tile pyramid (see Section 2.2.1) easier, the map extents are cut to tile borders, usually at zoomlevel 14. Metadata, stored in the ulmbin header, contains the location and size of the rectangle covered by the map, given in lat/lon coordinates, the minimum and maximum zoomlevel and base zoomlevel of the map, which is used to split the map features into tiles. The base level is usually relatively high, a common selection is level 13 (level 18 is the most zoomed-in level). Such a base level tile contains the map features for all tiles inside the pyramid with the base level tile as tip, so if the features of a level 18 tile are needed, its parent on the base level must be calculated and the correct features have to be extracted from the base level tile. Next, a number of *rule groups* is read from the ulmbin file. As previously mentioned, a rule

group is defined as an ordered list of rules, in this case defined by their IDs. The rule groups are specific for a single map, they are read once when the map is loaded and kept in memory until the map is unloaded (usually when the application exits). The rule IDs are replaced by pointers to the actual rule objects from the global rule data structure to accelerate accessing them.

A feature is defined by a globally unique `ulmonID` and references a rule group from the current ulmbin file. Another important field is the so called `featureMask`, a 16-bit unsigned integer that is used to refine the granularity of the base level. Each base level tile has four children in the level right below the base level, and 16 children in the next level. The `featureMask` stores 16 independent bits, each used as a boolean value to define in which of those 16 children the current feature is used. The usage of the mask is detailed in Section 3.2.2.

Apart from the `ulmonID` and the `featureMask`, each feature stores a rule group ID, which is used to process the feature later, different strings like a name and a house number in multiple languages, and a `minLayer`. The `minLayer` denotes the lowest layer this feature is drawn in and is later used to correctly sort the features. Last, the most important property of a feature is stored, the actual geometry. A geometry can be one of the three types *point*, *line* and *polygon*. In any case it consists of one or several two dimensional positions, given in spherical Mercator projected centimeters as described in Section 2.2.1, relative to the south-west corner of the base level parent tile. This relative addressing scheme makes it possible to save storage space by using variable-length encoded integers, integers that reserve a single bit of each byte as a "continue flag" specifying whether another byte has to be read or not. Features are stored in sorted order, from the highest zoomlevel to the lowest, i.e. features that only occur on the most detailed level 18 are stored last.

3.2.2 Data Loading

Loading, in this context, means conversion of map features from their packed form in an ulmbin file to in-memory data structures that are ready to be rendered. Also, some features that cannot be part of the current tile because of their geographical position are filtered to speed up rendering later. In an ulmbin file, map features are stored in blocks for each tile on the base level. If data for a tile on another level is needed, the base level parent or the base level children of the tile need to be determined, since a block of feature data can only be loaded wholly. The tile coordinates of the current tile $(x, y, zoom)$ must therefore be "projected" to the base level. To do so, the zoomlevel difference to the base level is calculated:

$$d = zoom - zoom_{base} \tag{3.1}$$

If d is positive, the current tile is below the base level and has a single base level parent; the coordinates of this parent tile can then be retrieved by shifting the coordinates of the requested tile d bits to the right.

$$\begin{aligned} x_{base} &= x \gg d \\ y_{base} &= y \gg d \end{aligned} \tag{3.2}$$

Otherwise, if $d < 0$, the current tile is above the base level, and its children on the base level are inside a square region. The side length of the square can be calculated by bit shifting 2 with

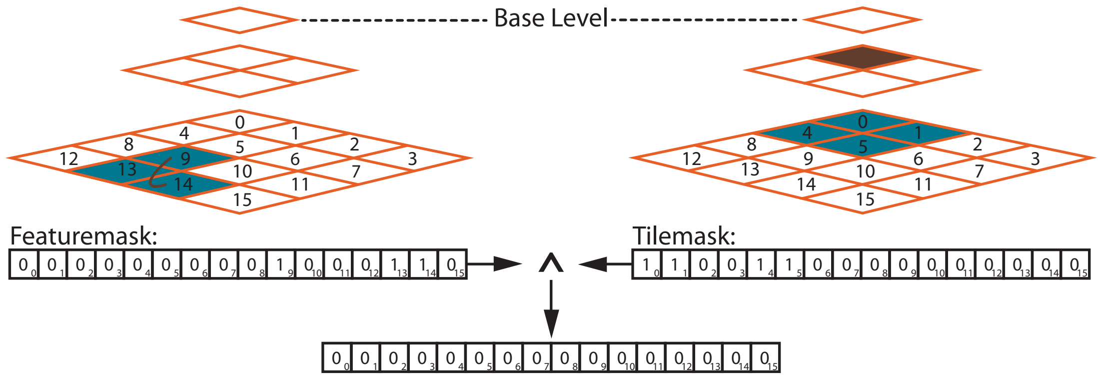


Figure 3.3: A linear feature (drawn in brown) can be discarded because the logical conjunction of the `featuremask` and the `tilemask` has no high bits. The tile that is currently rendered is highlighted in brown; it is one level below the base level.

the positive difference to the left; the coordinates of the north-west tile of the area are calculated likewise.

$$
\begin{aligned}
l &= 2 \ll (-d) \\
x_{basemin} &= x \ll (-d) \\
y_{basemin} &= y \ll (-d) \\
x_{basemax} &= x_{basemin} + l - 1 \\
y_{basemax} &= y_{basemin} + l - 1
\end{aligned}
\tag{3.3}
$$

In Section 3.2.1, the `featureMask` has been introduced as a property of a feature. Figure 3.3 illustrates how this mask can be used to determine whether a feature has to be loaded on the current tile or not. This only applies for tiles below the base level, i.e. if $d \leq 0$, this check is skipped. To execute the test, a 16-bit `tileMask` has to be calculated, which contains one bit for each of the 16 children of the base level parent on level $zoom_{base} + 2$. In Figure 3.3, the current tile is one level below the base level ($d = 1$), so four of the 16 tiles are children of the current tile and are marked by a high bit in the mask. If the current tile would be on a lower level ($d \geq 2$), the mask would only contain a single high bit.

For the case where the requested tile is above the base level ($d < 0$), the following steps are repeated for each base level tile. As a next step, the data block for the base level parent is read into memory. To minimize the effect of storage latency, the data block is kept in memory, until a tile with a different base level parent is requested. Since the map is often viewed on low levels, this saves a lot of storage operations; for instance, a level 13 (base) tile covers 32×32 tiles on the most detailed level 18, so 1024 tiles on that level have the same base level parent tile. Next, each feature is read until all features up to the requested zoomlevel are processed – if a level 17 tile is requested, there is no need to read features that belong only to level 18; and since the features are stored in exactly that order, the processing can simply stop early in that case. For each feature, the `featureMask` is read and the logical conjunction of the `tileMask` and

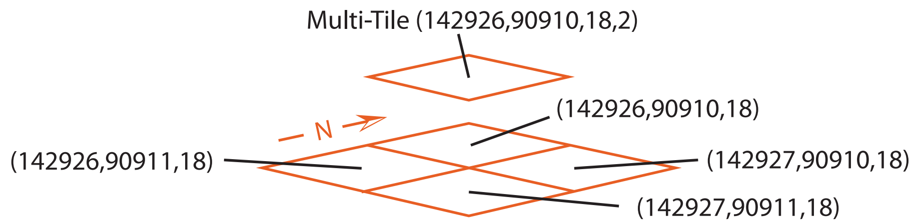


Figure 3.4: A multi-tile. The `multiTileFactor` is given as the last parameter of the multi-tile's address. Note that the north-western child tile of the multi-tile has the same address as the multi tile.

the `featureMask` is calculated; if it has a numerical value of 0, i.e. if it has no high bits, the feature in question can be discarded without any further processing because it is outside of the current tile as illustrated for a sample linear feature in Figure 3.3.

Features that are not discarded because of their `featureMask` are read into memory; after that the points of the feature are tested for intersection with the requested tile – if the feature is found to be outside, it is discarded. This test must be done differently for each of the three feature types point, line and polygon, and is not implemented as an exact test but as a conservative heuristic. All features that pass this last test will end up in the final tile in one or another way – static data will be rendered in the server, dynamic data in the post-processor.

3.2.3 Multi-Tiles

Especially on the most detailed levels 17 and 18, the loading concept described in the previous section suffers from multiple feature-loads, because on those levels, most features cover more than a single tile. Additionally, the conservative inside test might cause features to be processed that are not visible on the requested tile. Therefore, a novel concept is introduced in the renderer: multi-tiles, which try to overcome this problem by loading and rendering multiple tiles at once. The basic principle of multi-tiles is illustrated in Figure 3.4.

Multi-tiles are virtual tiles that cover a square area of actual tiles. The side length of this area, the `multiTileFactor`, is configurable at compile time; however, 2×2 multi-tiles have proven to be a good compromise between memory usage and rendering speed. As a restriction, the `multiTileFactor` must be a power-of-two number. Multi-tile sizes up to 4×4 tiles have been tested, but while rendering such large tiles is still about as fast as rendering a single tile, the memory usage of such large tiles is not suitable for mobile devices. Like actual tiles, multi-tiles are defined by a triplet $(x, y, zoom)$, but in contrast to actual tiles, the x- and y-coordinates are always divisible by the `multiTileFactor`. That way, the whole world can be covered without overlapping. The multi-tile covering a given actual tile can therefore easily be calculated by subtracting the remainder of the division of the coordinate by the `multiTileFactor` from the coordinate:

$$\begin{aligned} x_{multi} &= x - (x \,\%\, \texttt{multiTileFactor}) \\ y_{multi} &= y - (y \,\%\, \texttt{multiTileFactor}) \end{aligned} \tag{3.4}$$

The calculation of the base level parent must be changed to correctly handle multi-tiles, which is done by calculating the *virtual zoomlevel* of the multi-tile (the zoomlevel at which the multi-tile consists of a single tile). The virtual zoomlevel is one level above the actual zoomlevel for 2×2 multi-tiles; for the general case it is calculated using the *logarithmus dualis*, which has to yield an integer since the `multiTileFactor` can only be a power-of-two number.

$$zoom_{virtual} = zoom - \log(\texttt{multiTileFactor}) / \log(2) \tag{3.5}$$

The coordinates of the multi-tile are then projected to $zoom_{virtual}$ and the rest of the base level parent calculation can be completed as described using the projected coordinates.

Like the base level parent, the `tileMask` is calculated differently for multi-tiles – a feature can only be discarded if it is not part of *any* of the actual tiles covered by the multi-tile. This can be accomplished by combining the `tileMask`s of all actual tiles using a *logical or* operator. For instance, a tile one level below the base level ($d = 1$, see Section 3.2.2) would result in a `tileMask` with 4 high and 12 low bits. In contrast to that, a 2×2 multi-tile will combine 4 such `tileMask`s and will always yield only high bits.

The usual resolution of a tile image is 256×256 pixels; these dimensions are used, for instance, by Google Maps and Bing Maps. Naturally, for multi-tile rendering, this resolution has to be multiplied by the `multiTileFactor`. Another factor to consider for the tile image resolution is the pixel density of the screen. 256×256 pixel tiles work well for standard density screens with a dots per inch (DPI) value of about 130 to 170. Some new devices, however, feature high density screens (for instance Apple's Retina display technology) with doubled DPI ratings that require the tile resolution to be doubled as well. Considering such a high density screen and a `multiTileFactor` of 2, the tile image resolution grows to 1024×1024 pixels, which is why no texture atlas can be used for storing the tile images, as mentioned in Section 3.1.

3.2.4 Tile Rendering

As already mentioned, the features from all base level tiles are filtered using geometry data and their `featureMask` and read into memory in the data loading step. After that, all features are first sorted using their `minLayer` to ensure proper overdraw and then iterated over. For each feature, the rule group is looked up from the active map and for each rule of the rule group, a loop over all symbolizers of the rule is run. Now, the symbolizers are filtered according to their type: line, line pattern, polygon, polygon pattern and some point symbolizers produce static data while (most) point, text and shield symbolizers produce data that has to stay upright (dynamic data). Each symbolizer can be executed on a feature, meaning that it uses the data available in the feature to render according to its own specification.

Symbolizers that produce static data are instantly executed and render directly into the tile texture (via the server's FBO); dynamic symbolizers, however, cannot be executed in the server because their data has to be drawn over the final slippy map in the client. To accomplish this, pairs of a symbolizer and a feature are created and stored together with the reference to the tile texture in a *Tile* datastructure.

Since a feature may be processed by any number of symbolizers producing both static and dynamic data, the memory used by a feature may be still needed after the processing of the fea-

ture in the server has been finished. Therefore the memory management for features is handled using a reference counting mechanism that deletes the feature as soon as the last symbolizer-feature pair is deleted. This way, if a feature produces only static data, it is deleted once the server finishes rendering the tile; if a feature has to be used later (in the post-processing stage), it is kept in memory until the tile it belongs to is no longer needed and removed.

In the following sections, the static symbolizers are presented. More specifically, it is described what happens when a line, point or polygon symbolizer is executed on a feature.

Line Rendering

Like Persson [28], lines in the proposed system are drawn using the OpenGL triangle strip primitive. This is done to overcome limitations of the line primitive as used by McNamara et al. [18] and Chan and Durand [6], mainly the limitation that OpenGL implementations may limit the maximum line width, and also to compensate for shortcomings of the quad primitive as used by Bærentzen et al. [3]. Apart from the fact that the quad primitive is not available in OpenGL ES 2.0, it is also not suitable to render line strips, which is much needed in the case of street data, because rendering each straight element on its own would decrease rendering performance. Unlike all of the previous publications, a special focus is set on line caps – all three cap styles (see Figure 2.15) can be rendered by only setting a uniform flag in the shader. As for line joins, with the presented tessellation, only miter style joins can be rendered, however, this is acceptable for the map rendering use case, because the current stylesheet only uses this join style. An extension to support bevel style joins is possible and is described in Section 5.3. Unfortunately, since OpenGL ES 2.0 does not have a geometry shader, it is not possible to create new vertices directly on the GPU, instead, for each vertex needed by the tessellation, exactly one vertex has to be sent to the vertex shader, where it can be moved to the correct position. Therefore the start and the end vertex have to be sent four times, other vertices twice, each with an additional vertex attribute specifying the extrusion direction. Figure 3.5 gives an example of how a two-element line strip is tessellated.

To render different line cap styles, the line ends are tessellated in fine granularity, whereas the line segments consist of only two triangles. In total, $2n + 4$ vertices are needed to render a line strip consisting of n points. The order in which the vertices are generated is denoted by the vertex index (0 to 9 in Figure 3.5).

Antialiasing, as illustrated in Figure 3.6, is done by filtering the mathematical line with a filter kernel encoded by a function to map from the pixel-line-segment-distance d_l, the line width w and the filter radius r to an intensity value I_l, according to the finding of Gupta and Sproull [10] that such a function can be used to encode a circularly symmetric filter. Since lines are not only drawn white on black, this intensity can be used as transparency to blend the pixel with the background. Instead of pre-calculating lookup tables for different line widths, an analytic function is used to model the filter kernel, as done, for instance, by Qin et al. [33]. Since only fragments inside the tessellated line are rasterized and some fragments outside of but near the mathematical line (up to a distance of r) receive intensity values higher than zero, the line width is increased by the filter diameter, resulting in the actual line width $w_a = w + 2r$. This is done once at application startup, when the symbolizers are loaded. The Euclidean pixel-line-segment-distance d_l (given in pixels) is calculated from the texture coordinates (s, t) that are set

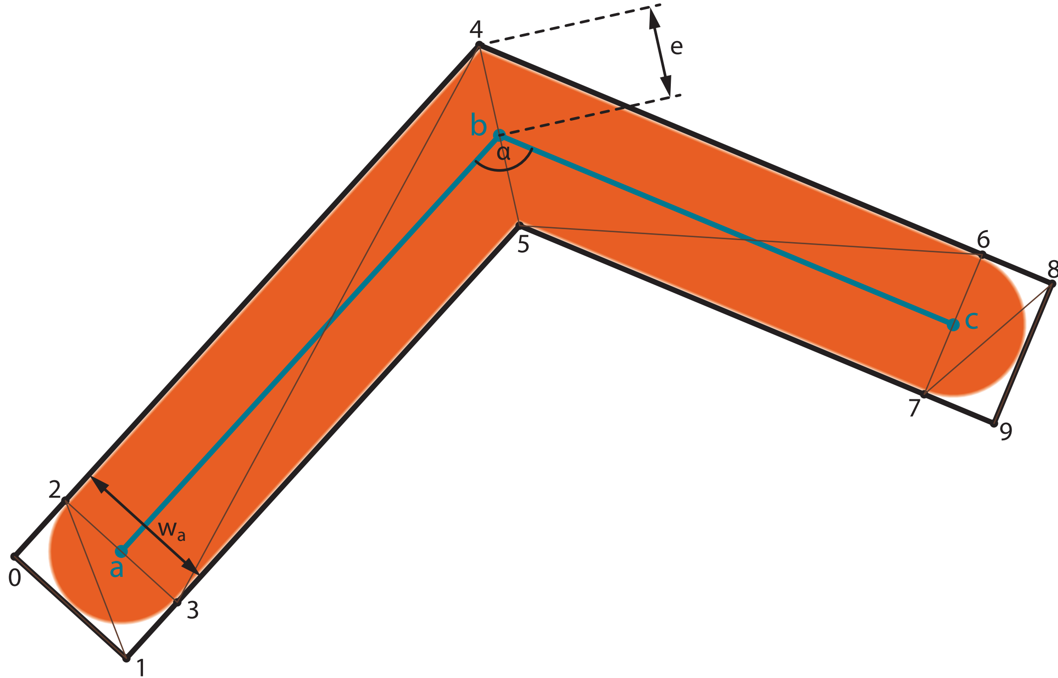


Figure 3.5: Tessellation of a line strip consisting of the three points a, b and c into ten vertices and eight triangles, using a miter join and round line caps. The extended width w_a, the extrusion length e and the angle α are plotted.

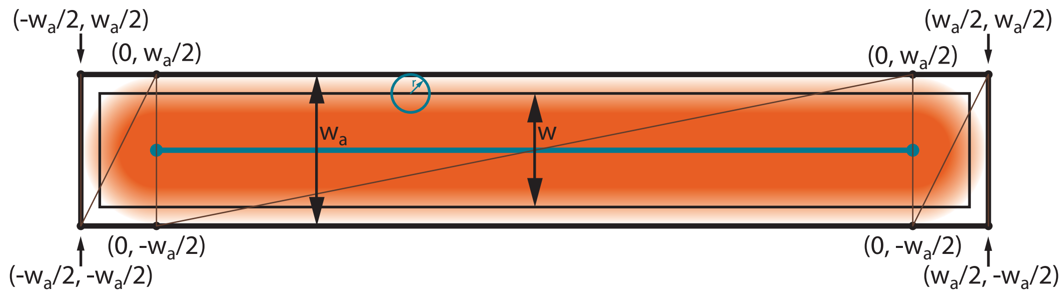


Figure 3.6: The texture coordinates used for the distance calculation. A very large filter was used for illustrational purposes here.

in the vertex shader, which is discussed later in this section.

$$d_l = \sqrt{s^2 + t^2} \tag{3.6}$$

Like in the algorithm of Qin et al. [33], the function used as a smooth falloff is the GLSL `smoothstep` function; the intensity is calculated from w_a, d_l and r as follows:

$$I_l = 1.0 - \text{smoothstep}\left(w_a/2 - 2r, w_a/2, d_l\right) \tag{3.7}$$

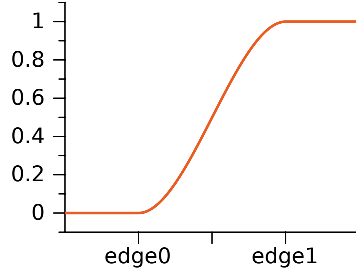


Figure 3.7: $\mathrm{smoothstep}\,(edge0, edge1, x)$

By using Equations 3.6 and 3.7 in the fragment shader, the appearance of the line can be controlled entirely by the vertex shader by setting different texture coordinates. Vertices that are extruded directly to the left or right of the line points (vertices 2 to 7 in Figure 3.5) are given texture coordinates of $(0.0, w_a/2)$ for left points or $(0.0, -w_a/2)$ for right points.

For the line cap vertices, the cap style must be taken into account – for square or butt caps, those vertices are assigned the same texture coordinates as the inner vertices; for round caps, however, the texture coordinate vector to each of the two corners at a line end must have a length equal to the euclidean distance from the line end point to the corner, the s-coordinate is therefore set to $-w_a/2$ at the line start point and $w_a/2$ at the line end point while the t-coordinate stays the same, $w_a/2$ for points right of the line, $-w_a/2$ for points left of the line.

The final discrimination between butt and square caps is accomplished by extruding the cap vertices differently. For square caps, the vertices are positioned as in Figure 3.5, but for butt caps, they are only extruded in normal direction of the line segment, not in collinear direction – that way vertex 0 is at the same position as vertex 2 and vertex 1 is at the same position as vertex 3. This results in degenerate triangles for butt caps, but they can safely be ignored. Of course, the same effect could have been achieved by omitting the first two and the last two vertices, but this would have had to be done in the application code, which was not desirable, because the line geometry was to be separated from the line style.

As can be seen in Figure 3.5, the amount of extrusion (the distance from the extruded vertices to the line, e) depends on the line width w and the angle α of the two line segments at a join. To simplify calculation and remove a conditional path in the shader, the angle α at the start and end point is assumed to be $180°$. The actual extrusion e is then calculated using the following formula:

$$e = w_a / \left(2 * \sin\left(\alpha/2\right)\right) \tag{3.8}$$

For the start and end points, e is equal to $w_a/2$ because $\alpha = 180°$ for those points.

The last important stylistic feature of a line is the dash pattern. The method presented is inspired by Poddar [31], who used a texture to store the dash pattern, but extends Poddar's approach by storing a distance value instead of only binary information to be able to use the antialiasing approach presented previously. In the stylesheet, the dash pattern is given as a series of numbers, denoting pixel lengths of “on” and “off” intervals, the first number being an “on” interval. During the initialization of a symbolizer, this dash pattern is converted into a single-

channel 1D texture, with each pixel containing the distance from the current pixel center to the nearest on/off switch (which is located at a pixel border, so all distances have a fractional part of 0.5). To distinguish “on” and “off” intervals, the distance is stored with a positive sign in “on” intervals and a negative sign in “off” intervals. Because of the limited texture data types in OpenGL ES 2.0 (no floating point textures, a maximum of 8 bit per channel and no signed data types), a value of 127.5 is added to the distance before storing it in the texture, making it a positive 8-bit integer, if the maximum dash length is set to 256 pixels. The repetition mode of the dash texture is set to repeat on the s-axis (along the line) and to clamp on the t-axis, so that the pattern repeats automatically if the line length exceeds the pattern length. The filtering mode of the texture is set to linear filtering. When rendering, each line point is given the total length of the line up to the current point (in world coordinates) as an attribute. The vertex shader converts this length to pixels and passes it to the fragment shader as a varying variable, to be interpolated for each fragment of the line. The fragment shader uses this interpolated length as a texture coordinate for a lookup in the dash pattern texture, converts the resulting value from the $[0, 1]$ interval to the $[0, 255]$ interval and subtracts 127.5. The resulting value d_d is, due to linear filtering, a sub-pixel-exact distance along the line to the nearest on/off switch, whose sign determines if the current fragment is inside an “on” or an “off” interval. This distance can be used to create the correct line cap for each dash: Equations 3.9, 3.10 and 3.11 are used to calculate the intensities for round, square and butt caps for line dashes, respectively.

$$I_d = 1.0 - \text{smoothstep}\left(w_a/2 - 2r, w_a/2, \text{sgn}\left(-d_d\right) * \sqrt{d_d^2 + t^2}\right) \tag{3.9}$$

$$I_d = 1.0 - \text{smoothstep}\left(w_a/2 - 2r, w_a/2, -d_d\right) \tag{3.10}$$

$$I_d = \text{smoothstep}\left(-r, r, d_d\right) \tag{3.11}$$

As before, r denotes the radius of the filter kernel used for antialiasings. Note that for round caps (Equation 3.9) the Euclidean distance is multiplied by the sign of $-d_d$. This is done to fill the “on” areas; otherwise only a circle around each on/off switch would be drawn. The same correction must be applied for square caps (Equation 3.10), it is sufficient to simply negate d_d here. Inspired by the “best” method of McNamara et al. [18], the final intensity of the fragment (that is written to the alpha channel of `gl_FragColor`) is calculated as follows:

$$I = I_l * I_d \tag{3.12}$$

A sample line, rendered with the proposed method can be seen in Figure 3.8

Point Rendering

In the context of map rendering, a point feature is used to display an icon and/or text at a specific point on the map. Point features include POIs as well as landmarks such as peaks, springs or lighthouses. Usually a point feature is drawn by a point and a text symbolizer, producing an icon and an accompanying text. In this section, point symbolizers are explained, which are used to

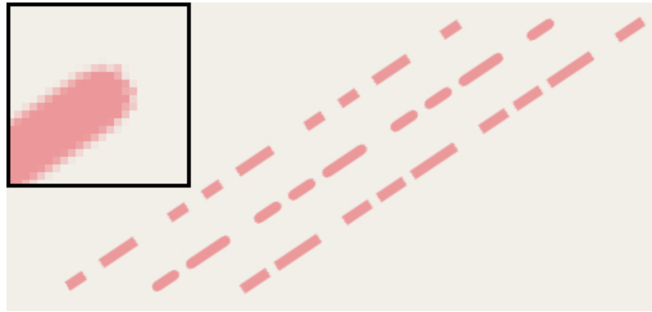


Figure 3.8: A dashed line with a width of 10 pixels, rendered with the three different cap styles butt, round and square, from left to right. The dash pattern in use here is $16 - 16 - 32 - 32 - 16 - 16$, i.e. short on - short off - long on - long off - short on - short off. A round cap has been enlarged 6 times to illustrate how transparent pixels are used for antialiasing.

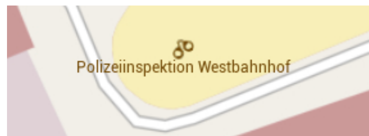


Figure 3.9: A point icon with a text label (a police station near the Viennese “Westbahnhof”).

produce icons. Text rendering is not discussed here (refer to Section 3.5.1 for details about this). Point symbolizers can be split into two groups: some basic icons are rendered directly into the map image, others are overlayed in the post-processing stage, either way, the rendering process itself stays the same.

To render points, the OpenGL point sprite primitive is used, which is a very simple way to render textured quads at a given position with a configurable size. Only a single position, the position where the icon should be placed, is passed to the OpenGL draw call. In the vertex shader, the `gl_PointSize` predefined output variable has to be set to the side length of the icon in pixels. A limitation of this approach is that only square icons can be used. The GPU then automatically creates four vertices, each shifted by half the `gl_PointSize` into one of the four corners and sets the predefined `gl_PointCoord` input variable for each fragment. `gl_PointCoord` contains the position of the fragment inside the generated quad, with $(0, 0)$ being the lower left corner and $(1, 1)$ being the upper right corner. These coordinates can then be used as texture coordinates for a lookup in the icon texture. Figure 3.9 illustrates how such icons are used in a map.

As described earlier, in Section 2.3.2, state changes like texture switches can severely slow down rendering in OpenGL. Since there are many different point symbolizers with different icons, creating a texture for each of those icons and constantly switching them would introduce such a bottleneck, so all icons are loaded into a large texture atlas instead when the symbolizers are loaded. The insertion into the texture atlas is done by a simple bin packing algorithm: Following the idea of Lee and Lee [15], the texture atlas of size $2^{M-1} \times 2^{M-1}$, $M > 1$ is

backed by a quadtree structure, resulting in M different bin sizes. That way, the texture atlas holds 2^{2k} bins in level $k < M$, but of course each bin on level k covers four bins on level $k+1$. When inserting an icon, the necessary bin size (the smallest possible $s = 2^l$ that is larger than the icon size) is calculated, and the quadtree is traversed until an empty bin of size s is found. If a bin is found, the icon is inserted, if not, it is rejected and cannot be stored in the texture atlas. Algorithm 3.1 illustrates this routine.

input : A quadtree, accessible by its root node root, and an icon to be inserted.
output: Either true or false, indicating whether icon could be inserted or not.

```
1  if root.filled ∨ (root.binsize == icon.s ∧ root.empty) then return false;
2  if root.binsize == icon.s then
3  |   copyToTex(root.pixeloffset, icon.bitmap);
4  |   root.filled ← true;
5  |   return true;
6  else
7  |   done ← false;
8  |   for i ← 0; i < 4 ∧ ¬done; i ← i + 1 do //Try children recursively
9  |   |   done ← insert(root.child[i], icon);
10 |   end
11 |   if done then root.empty ← false;
12 |   return done;
13 end
```

Algorithm 3.1: The `insert` function of the texture atlas

The extents of the icon inside the texture atlas (in relative coordinates within the interval $[0, 1] \times [0, 1]$) are stored with the symbolizer. When rendering, they are passed to the vertex shader as additional attributes, which hands it through to the fragment shader. This is only possible because no interpolation is done for the OpenGL points primitive, since every fragment is only influenced by a single vertex. Calculating the correct texture coordinates inside the texture atlas can then be done using linear interpolation between the given extents with the `gl_PointCoord` coordinates. To avoid even more texture switches, the icon atlas texture stays bound to a reserved texture unit during the whole lifetime of the application.

Polygon Rendering

Polygons are drawn by polygon and polygon pattern symbolizers. Before polygons can be rendered, they have to be tessellated into triangular structures, because OpenGL ES 2.0 has no polygon primitive type (neither does any recent version of OpenGL on desktop computers). The proposed implementation uses the GLU tessellator [7] to do this task at runtime, as it perfectly matches the needs of this application. It is extremely robust and performs easy tessellations very fast; most polygons encountered in map data are rather simple, many of them consisting only of a few vertices (e.g. houses). Additionally, the output of the GLU tessellator is perfectly compat-

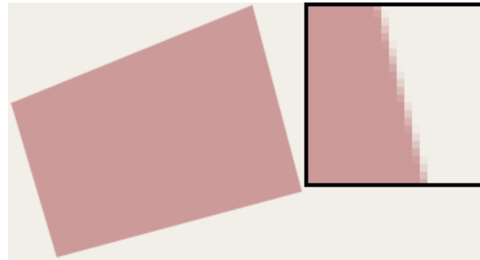


Figure 3.10: A polygon, rendered with the proposed antialiasing method, including a six times enlarged section of an edge.

ible to OpenGL rendering, because it only emits one or several of the OpenGL primitive types triangle, triangle strip and triangle fan.

Once the polygons are given as triangles, rendering them is trivial. Depending on the symbolizer, polygons are either filled with a solid color (passed as a vertex attribute) or textured with a repeated icon (polygon pattern symbolizer, used for woods, graveyards etc.). The icons for the polygon pattern symbolizers are stored in the same texture atlas as the icons for point symbolizers; the texture coordinates needed for the repeated application are calculated in the vertex shader depending on the vertex position. The fragment shader then uses the fractional part of the texture coordinates to do the same linear interpolation and texture atlas lookup as for point rendering.

Antialiasing for polygons is implemented roughly as described by Chan and Durand [6] and originally proposed by Snyder et al. [47], i.e. the polygon edges are overdrawn with anti-aliased lines; of course the proposed line antialiasing method (described in Section 2.5.1) is used for this. Figure 3.10 shows a polygon, antialiased with the method described. As an alternative to this, MSAA can be enabled in the server, as described in the beginning of Section 3.2.

3.3 Client-Server Communication

Both the client and the server are implemented to run within the same address space, i.e. they can access the same data structures. Therefore communication was implemented using shared structures together with access synchronization. To make data structures globally available while maintaining an object-oriented architecture, the singleton pattern is used extensively throughout the library. Every resource is controlled by a singleton manager which provides an interface for accessing the resource and handles inter-thread synchronization using the POSIX [49] thread library to achieve platform independence. In this section, the most important data structures will be covered and their usage by the server and client will be explained.

3.3.1 The Tile

A tile is a structure holding all information of a single multi-tile (see Section 3.2.3 for details on multi-tiles). This includes the tile address $(x, y, zoomlevel)$, the `multiTileFactor` of this

tile (which is currently the same for all tiles in the system, but individual values may be used in the future), the tile image texture handle and the dynamic features for this tile. Although the division into static and dynamic features is done in the loading phase, dynamic features become available for the client slightly earlier than the tile image containing the base map, which is useful because the client can use a low resolution tile image from a parent tile while the correct tile image is not yet available. If the dynamic features of the tile are already available though, the client can display them above the low resolution map, reducing the impression of slow rendering and/or unavailable data, and creating a smoother loading process.

To realize this, a tile can be in one of the following states:

- `INITIAL`: The tile has been created, but no data has been loaded yet. There is no tile image and no post-processing data.
- `INPROGRESS`: The server is currently working on the tile, data loading is in progress. There is no tile image and no post-processing data yet.
- `LOADED`: Data for the tile has been loaded but not yet rendered. There is no tile image, but post-processing data is already available.
- `FINISHED`: The tile image has been rendered and the server's OpenGL context has been flushed. All data is available.
- `NOTRENDERABLE`: There is no data for this tile, i.e. none of the installed maps covers it. It will be ignored during rendering.

In the main loop of the client, all visible tiles are requested from the tile cache in each frame. There is no such thing as a cache miss; if the requested tile is not already present in the cache, it is created during the request. The client then processes the tile according to its state and renders either nothing, or only the post-processing data, or both.

3.3.2 The Rendering Queue

The rendering queue is the task list for the server. It holds all tiles that have been requested by the client, but have not yet been processed by the server. To avoid filling up the queue with unimportant tiles (for instance during a fast pan on the map), the queue must be sorted in a way that favors tiles close to the current view point. Additionally, the client may request a tile multiple times before it is processed (because the client tries to access needed tiles in each frame), so the data structure must handle multiple inserts of the same item gracefully. Summing up, the rendering queue must fulfill the following requirements:

1. Uniqueness: No tile can be in the rendering queue twice at the same time.
2. Priority Sorted: Items in the rendering queue must be sorted by a configurable priority.
3. Updatable Priorities: The priority of an item must not be fixed; priorities may be updated in batches. Correct order must be reestablished once the batch update is complete.

The data structure backing up the rendering queue is made of two simple parallel data structures to meet the given requirements: a set and a priority-sorted queue. When pushing an element into the rendering queue, the set is searched for the element; if it is not found, the element is inserted into both structures. The opposite action, popping an element from the rendering queue, finds the element with the highest priority in the priority-sorted queue, removes it from both containers and returns it.

To calculate the priority used in the rendering queue, the distance of the tile from the *center tile*, the tile at the screen center, is calculated. As a distance measurement, the Manhattan distance is used; if the tile is on a different zoomlevel, a penalty of 5 is added for each level, so that tiles on the same level as the middle tile are prioritized. The priority used for the rendering queue is updated once in each animation frame for all items in the queue, so it is not necessary to support on-the-fly updates of the priority. Instead, when the priority is updated, all items are popped from the rendering queue, their priority is recalculated and they are inserted into the rendering queue again. While this seems to be a naive approach, there is another advantage apart from simplicity: tiles with a distance larger than a configurable threshold can be removed instantly by not inserting them into the queue again, thus keeping the queue from growing too large. Tiles that are removed from the rendering queue this way are also removed from the tile cache.

3.3.3 The Tile Cache

The tile cache is the central data structure of the app, it manages all known tiles, keeps them in memory as long as they are necessary and manages memory usage. It has to provide fast lookup of a tile given its address, keep a time stamp of the last access with each tile and be able to remove tiles that have not been accessed for some time to reduce the memory footprint. As already mentioned, a tile is created as soon as the client requests it for the first time. This is done by the tile cache, and the newly created tile is instantly added to both the cache and the rendering queue. Since the client requests all tiles it needs in each animation frame, fast lookup of tiles is the most crucial feature. This is accomplished with a map data structure, with the tile address as key, and a structure containing the tile itself and a time stamp as value. The time stamp stores the time when the tile was last accessed and is updated each time the client requests a tile. This time stamp is used by the LRU replacement strategy that is run once in each animation frame and removes tiles until the total memory consumption of the cache falls below a threshold. Memory consumption is estimated in a crude way, depending on the tile state and the number of post-processing features stored in the tile. For instance, tiles in `INITIAL` state do not contribute to the memory consumption calculation at all.

3.4 Client

The second largest part of libMapRendererGL, the client, is to be described in this section. The central responsibility of the client is to display the map in a slippy map fashion (Section 2.2.3) and react to user interaction such as zooming or panning of the map. In addition to this, being not as obvious but equally important, the client schedules tiles for rendering (as described in Section

3.3.3) and renders the post-processing features, which contain all the text and are therefore vital for the look and feel of the final map.

3.4.1 Coordinate System & Camera Setup

Everything described up to now was exclusively done in 2D, because the map data is given in two dimensions (on the spherical Mercator projected map). Now, since the rendering library supports a 3D mapping mode, a three-dimensional camera system with a perspective projection is used. The world coordinate system the client uses is a right-handed coordinate system with the map itself located in the x-z-plane. The y-axis points up ("into the sky" in contrast to "to earth's center"). The unit used in the world coordinate system is centimeters, because the map data is given in this unit, so no coordinate transformation needs to be done. When a tile image is rendered, a textured square (consisting of two triangles) is drawn in the x-z-plane, with its size being the actual size of the tile in centimeters, calculated using the width of the world map in Google projection (637813700 is the radius of the underlying sphere in cm, see Section 2.2.1), divided by the number of tiles on the current level:

$$w = \frac{2 * 637813700 * \pi}{2^{zoomlevevl}} \tag{3.13}$$

The view point is (for a 2D map view) located above the x-z-plane and its view vector is parallel to the y-axis, looking into negative y. The camera position and direction is defined by a look-at-point (on the x-z-plane), a distance from the look-at-point and two angles (yaw and pitch). The yaw angle controls the rotation of the camera around the y-axis (or any axis parallel to the y-axis); the pitch angle controls the "amount of 3D" of the map view, as known from various navigation devices. For yaw rotations, it is necessary to support rotation around an axis parallel to the y-axis, passing through the x-z-plane at a configurable point. Often, this point is the look-at-point, but user interaction on a touch screen device (which this library is targeted at) usually also enables the user to rotate around any point visible on the screen. As an additional constraint, the camera must always stay upright in 3D mode.

Zooming in or out is done by simply moving the view point closer or further away, the tile size on the screen then changes automatically due to the perspective projection. The zoomlevel of the tiles that are displayed is calculated using the view point distance from the x-z-plane, the field-of-view of the camera, the display resolution and the tile resolution – it is selected such that tile images are never enlarged above a configurable image-pixel to screen-pixel ratio.

3.4.2 Tile Selection

Virtual texturing, as done by Tanner et al. [48], applies the "virtual map texture" all at once to a large object representing the earth's surface. In our case, each tile texture is stored separately and applied to a quad of the size of a tile; this avoids some complexity with calculating the correct texture coordinates, as neither a toroidal addressing scheme, nor a texture atlas and indirection texture [2, 17] is needed. To determine the visible tiles, it is sufficient to intersect the view frustum with the x-z-plane (resulting in a trapezium or even a rectangle for the 2D case) and finding the tiles that are touched by the trapezium, because all the geometry lies in this plane.

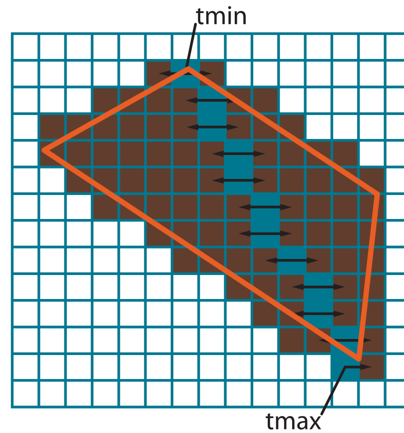


Figure 3.11: A tile rasterizer is used to determine the tiles visible on the screen. Blue tiles are selected by Bresenham's line drawing algorithm [5], brown tiles are found by the east-west search.

The intersection of the view frustum with the x-z-plane is calculated by extracting the four edges of the view frustum from the projection matrix and performing simple line-plane intersection. Since the tiles form a regular grid on the map plane, a polygon rasterization algorithm can be used to determine the tiles, as mentioned in Section 2.6. To avoid the complexity and the necessary preprocessing of a scanline rasterization algorithm as described by Hearn and Baker [12], an algorithm inspired by the "smart" one proposed by Pineda [30] is implemented. A further downside of a scanline-based algorithm is that tiles which touch an edge of the polygon with only a single corner might be missed if tile centers are used for scanline positions; however if scanlines were placed on tile borders, special handling of the first and/or last scanline would be necessary.

The following calculations happen entirely in tile space, a space constructed such that a tile in the current zoomlevel has a side length of 1; the polygon is therefore transformed into that space. An illustration of the polygon in tile space along with the tiles that are selected by the algorithm can be seen in in Figure 3.11. The highest and the lowest tile, t_{min} and t_{max}, respectively, are then calculated. The (vertical) line through both of those points is then rasterized in tile space using line drawing algorithm introduced by Bresenham [5]; to avoid selecting multiple tiles with the same y-coordinate, the algorithm has been modified to skip such steps and jump to the next scanline immediately (the tiles which are skipped here are selected by the east-west search later anyway). All tiles that are selected by Bresenham's line drawing algorithm are touched by the polygon, because the line from t_{min} to t_{max} is either a diagonal or one edge of the trapezium. To find the other visible tiles, in each step of the line drawing (once on each scanline) an east-west search is performed until a tile that is completely outside of the polygon is found. This algorithm is outlined in pseudo code in Algorithm 3.2. The `overlaps` function encodes an intersection test, where the four corners of the current tile are tested to be inside the trapezium. Only if none of them is inside, the reverse test is executed, whether one of the four corners of the trapezium is inside the current tile. If both of those tests fail, the function returns false, otherwise true. Note

that the exit variable `done` in Algorithm 3.2 would not be necessary, if line 20 was a conditional jump instead; both loops could then be implemented as infinite loops.

```
input  : Intersection of the view frustum with the x-z-plane in world space, corners[4]
output: A list L of tiles visible on the screen.
1  highest ← 0; lowest ← 0;
2  for i ← 1 to 3 do
3  |  if corners[highest].y < corners[i].y then highest ← i;
4  |  if corners[lowest].y > corners[i].y then lowest ← i;
5  end
6  start ← getTileAddress(corners[highest]);
7  end ← getTileAddress(corners[lowest]);
8  x0 ← start.x; x1 ← end.x; y0 ← start.y; y1 ← end.y;
9  dx ← abs(x1-x0); dy ← −abs(y1-y0);
10 sx ← (x0 < x1 ? 1 : −1); sy ← (y0 < y1 ? 1 : −1);
11 err ← dx + dy; done ← false;
12 repeat
13 |  for t ← (x0, y0); true; t ← (t.x + 1, y0) do //Search east
14 |  |  if overlaps(corners,t) then add(L, t); else break;
15 |  end
16 |  for t ← (x0 − 1, y0); true; t ← (t.x − 1, y0) do //Search west
17 |  |  if overlaps(corners,t) then add(L, t); else break;
18 |  end
19 |  repeat //Skip tiles on same scan line
20 |  |  done ← (x0 == x1 ∧ y0 == y1);
21 |  |  err2 ← 2 ∗ err;
22 |  |  if err2 > dy then err ← err + dy; x0 ← x0 + sx;
23 |  |  if err2 < dx then err ← err + dx; y0 ← y0 + sy; break;
24 |  until done;
25 until done;
```

Algorithm 3.2: Tile Selection using polygon rasterization

Although the fact that the textured geometry is flat enables usage of the original clipmap algorithm by Tanner et al. [48], the approach described here is better suited for the map rendering problem. This is because the clipmap data structure requires to hold rendered images of the highest levels (the clipmap pyramid, see Figure 2.11) in memory all the time. While suitable for a precalculated mipmap, this is not possible in our case because of the data format: typically only certain maps are installed on a device, there is no map of the whole world that could be used to generate the clipmap pyramid tiles. Even if the fact that only certain, small regions of the clipmap pyramid are actually filled, was acceptable ("unmapped regions" could simply be displayed black), rendering the clipmap pyramid would take too long in case the user installed more than a few maps. Of each installed map, all the base level tiles would have to be read and rendered, a process not possible in a realtime application.

As for the tile selection, the proposed method is preferred over both the clipmap [48] and the virtual texturing approach [4]. In the clipmap approach, the selection is determined by the cache center and the clip size, no exact determination of the visible area is performed. When rendering, the whole geometry (in the simplest case a single large quad covering the whole world) is drawn and textured with the virtual texture. Since the cache center should have been set by the application beforehand, the visible parts of the texture should then reside in graphics memory in sufficient resolution, parts outside of the clipmap stack are only available in low resolution, but they are most likely clipped by the rendering pipeline anyway. The virtual texturing approach, designed to be used with arbitrary geometry, needs an additional rendering pass whose result has to be loaded back into main memory to be analyzed. Since the map is planar, the second rendering pass and the bottleneck of reading back its result can easily be avoided with the proposed algorithm. A drawback of this method is that even for a tilted view, there is only a single zoomlevel for all visible tiles, so tiles far away from the camera are likely to be at a too-detailed zoomlevel and tiles near the view point may become blurred because they are enlarged too much. This is a problem that is not easily fixed, because if tiles of different zoomlevels are rendered next to each other, certain features like minor roads, that only exist on the more-detailed level, will cause the tile borders to be visible, which is an undesirable artifact.

3.4.3 Main Loop

Two aspects of the client, the world coordinate setup and the tile selection algorithm, have already been discussed; in this section, an overview on the main loop of the application, large parts of which are handled in the client, will be given. The main loop cannot be handled entirely in the platform-independent rendering library because user interaction must be done by platform-dependent code. Therefore, libMapRendererGL exposes an API including different ways to forward user input to the library and a `frame` function that must be called in each animation frame after all user interaction calls. This function can be coarsely split into two parts: the update part and the render part.

Update

In this part, all user interaction that has occurred since the last update call is processed; if no user interaction has been sent, nothing needs to be done here. Otherwise, the camera is updated and the tile selection algorithm is run. Additionally to what was outlined in Algorithm 3.2, each visible tile is instantly requested from the cache, so that the server can start rendering it in case it is not yet available, and for tiles that are already in a `LOADED` or `FINISHED` state, their dynamic data is passed to the post-processor (Section 3.5). If a visible tile is not available, a request for its parent tile is issued to the tile cache, causing its time stamp to be updated if it already exists (and therefore protecting it from deletion). If the parent tile is not available yet, it will be added to the rendering queue, but with a low priority. The tile cache update procedure and the reorganization of the rendering queue is also invoked at this stage; it is safe to do so now because all tiles needed in the current frame have already been requested, so their timestamps have been updated and they have been added to the rendering queue, if necessary.

Render

The render part is then fairly trivial, the list is iterated over and each tile image is rendered as a textured quad on the x-z-plane. If a tile is still not available (although it has already been added to the rendering queue in the update stage), the client tries to replace it with its direct children or any of its parental tiles. Compared to virtual texturing algorithms [4, 48] this feature is new, as virtual texturing can only replace tiles with their parents. This tile replacement is the only exception from the rule stated in Section 3.4.2 that only tiles of the same zoomlevel can be visible on the screen. In 3D mode, this rule has several disadvantages, because tiles near the screen are enlarged while tiles far away from the screen are downsized; both situations may lead to unpleasant artifacts. To minimize those artifacts, the textures are rendered with the best texture filters available on OpenGL ES 2.0, bilinear filtering for magnification and trilinear filtering for minification. Additionally, the switch to the next, more detailed zoomlevel is done earlier than necessary so avoid extreme magnification at the cost of even more minification; this is acceptable because the minification happens "far away" from the user, in areas that are of low interest anyway. With the projection matrix used in the proposed renderer, at most about 20 multi-tiles are visible at once; they can easily be rendered in realtime without any performance loss.

As a last step of the rendering stage, the post-processor is invoked to render the accumulated features of all visible tiles over the final map image. Regardless of the current camera angle, (for instance in 3D-mode), those features are always rendered fronto-parallel, so that they appear undistorted on the screen; and they stay so even if the map is tilted or rotated. After this, the `frame` function returns and control is given back to the application code.

3.5 Post-Processing

The post-processor is responsible for handling the dynamic data of all tiles that are currently visible on the screen. It is implemented in a third thread beside the server and the client (main) thread, but has no OpenGL context because the actual rendering is done in the client thread. When symbolizer-feature pairs are sent to the post-processor (in the update stage of the client's main loop), it first filters them for duplicates. This is necessary because features close to tile borders or features that span multiple tiles like streets (the label of the street name is a dynamic feature) might be stored in multiple neighboring tiles. Instead of recreating this duplicate-free list of symbolizer-feature pairs in each frame, it is only updated. New pairs are inserted and features that belong only to tiles that are no longer visible are removed, to reduce the amount of calculation necessary in a single frame. As soon as a feature is added to the post-processor, the post-processor adds it to its internal list of visible dynamic features and starts processing it, if necessary; text features, for instance, need to be rendered into a bitmap in main memory before they can be uploaded to the GPU to be rendered. When a feature is ready to render (i.e. when the preprocessing is finished), it stays in this state until it is removed from the post-processor because it is no longer visible, so the result of the preprocessing is cached throughout the life time of the feature.

Internally, the post-processor needs to decide which items will be rendered and at which

position. While the position of a feature is sometimes fixed (e.g. for point features), it may also be variable, a prominent example being text – it may be placed above or below an accompanying icon, or at any place along a road. The aim of the post-processor is to detect and avoid collisions by either moving or skipping items. Together with the Ulmon GmbH, it was decided that implementing the whole post-processor would exceed the scope of this thesis, so only two parts were done: the rendering of icons (points) and text labels of point and polygon features; text labeling of linear features (road names) was not implemented, though the architecture of the post-processor is designed in a modular way such that new types of features can be added at any time. How icons are rendered has already been described in Section 3.2.4; text rendering is covered in the following section.

3.5.1 Text Rendering

Text rendering consists of multiple stages; a font must be loaded, the required glyphs must be rendered and uploaded to graphics memory, and then the text must be rendered using OpenGL. Since map rendering requires text in many different languages and scripts, almost the whole unicode character set must be usable – this is accomplished by loading multiple fonts containing different characters, and testing them one after the other when a glyph is requested. To load fonts and render glyphs, the FreeType 2[1] library is used, which also provides a glyph cache to speed up glyph retrieval. When a string of text is about to be rendered, first all required glyphs are loaded from the font file and rendered into an in-memory bitmap. The whole string is then represented by a single bitmap, which is uploaded to the graphics memory as a texture and displayed using a small textured quad. The texture ID assigned to a specific string of text is stored with the dynamic feature, so that the post-processor can simply render it in the future frames without having to preprocess it again. Since each glyph is requested in the correct rotation and scale, FreeType is able to render glyphs antialiased, which results in very high-quality text display; a downside of this is, however, that a glyph must not be rotated or scaled after it has been rendered with FreeType – it has to be rendered pixel-exact on the screen to preserve the high quality. This is because FreeType not only renders the glyph antialiased, it also provides *hinting*, a process described in the FreeType documentation [51] as "grid-fitting", to reduce the artifacts from scaling the glyph outlines to the desired size. Without hinting, straight lines, for instance in the glyphs for the letters T and E, might have different widths. Hinting information can either be stored with the font or applied automatically by FreeType, however, when the glyph is not displayed pixel-exact on the screen, the hinting is lost, as can be seen in Figure 3.12. The misaligned line has been offset by half a pixel in each direction and displayed with bilinear filtering. Note that the enlarged, misaligned letter e is blurry and seems to be cut off.

3.6 Development Environment

The proposed rendering library, libMapRendererGL, is designed in an object-oriented way and implemented in C++. It exposes an API written in the C language to ensure compatiblity to both

[1] `http://www.freetype.org/freetype2/`

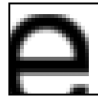


Figure 3.12: Hinted text rendering, both pixel-aligned and misaligned (Image from [46]).

Objective-C (the primary programming language on the iOS platform) and the Java Native Interface (JNI) needed to use the library on Android. The library has been developed on Mac OS X in the XCode Integrated Development Environment (IDE), which is also used to compile for the iOS platform. Compilation for Android is accomplished using the Android Native Development Kit (NDK) along with build scripts for the standard GNU[2] build system.

[2] GNU's Not Unix

CHAPTER 4

Results

In this chapter, the proposed implementation will be evaluated and compared to previous map rendering solutions. Since the rendering library was written to replace a CPU renderer, libMap-Renderer, most comparisons are done against this library. Specific parts, especially about line rendering, are compared to scientific work presented in Chapter 2. The results are split into three categories: performance, quality and features.

4.1 Performance

Comparing libMapRendererGL to libMapRenderer is difficult because of their different architecture. While the old implementation renders all the data into map images which are then simply displayed, the new library renders only static data into the tiles, all dynamic features are rendered as a post-processing step in each animation frame. This fact makes speed comparison particularly hard, since neither the rendering time per tile nor the refresh rate in the main loop can be used as a measure. This is because in both measurement areas, the two rendering libraries fulfill different tasks; the tile time would be unfairly short in the proposed implementation because less is rendered (only the base map) while the refresh rate would be unfairly short in the legacy implementation, since no post-processing is done there - only displaying tile images runs as fast as the display allows it (in both libraries). Nevertheless, two tests are executed, described in the following sections.

4.1.1 Comparison of the Server-Performance

To measure the performance of the tile server, the old renderer has to be modified to exclude all dynamic features, such that both implementations render only base map tiles, just as the server in the proposed implementation. As mentioned in Section 3.1, static and dynamic features can be distinguished by the symbolizer they are drawn with, so certain types of symbolizers are excluded from libMapRenderer. This includes text and shield symbolizers and a majority of the point symbolizers; leaving only polygon, line and some point symbolizers with primitive icons

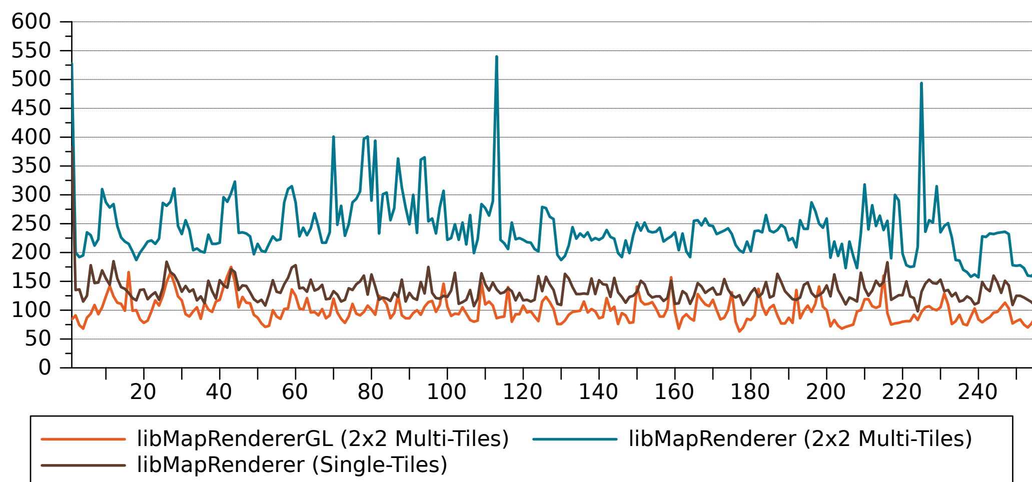


Figure 4.1: Total times in milliseconds for 256 multi-tiles in the center of Vienna, lower is better.

in place. Additionally, the old library has to be extended to render multi-tiles of the same size (2×2) as libMapRendererGL. All numbers and figures in this section are created with such a modified libMapRenderer, except those marked as single-tile, where the additional symbolizers are excluded, but instead of 2×2 multi-tiles, only the north-west of each multi-tile is rendered. As a test device, an Apple iPad (1st generation) is used.

To minimize file system read latency, a 32×32 tile (16×16 multi-tile) area at the most detailed level 18 in the center of Vienna was selected as a test map – this area corresponds to a single base level tile $(4468, 2840, 13)$ and loading from slow storage into memory occurs only at the very first tile because both implementations cache the base level tile block in memory. A plot of the time it takes to create each individual multi-tile can be seen in Figure 4.1.

Obviously, the new implementation outperforms libMapRenderer on every single rendered tile if multi-tiles are used; on average, it renders a tile in 99.80ms, which is 2.42 times faster than 241.43ms, the average time of libMapRenderer. If no multi-tiles are used in the old library, rendering is noticeably quicker; however, with an average rendering time of 134.80ms, libMapRendererGL is still 1.35 times faster, although the old library renders only a quarter of the area.

If the rendering time is split into preprocessing and rendering, it becomes apparent that libMapRendererGL (Figure 4.2) spends most of the time (80.38%) on preprocessing; less than one fifth is used for the actual rendering. In the legacy implementation (Figure 4.3), the total time is almost equally distributed between preprocessing and rendering; on average, 47.27% are spent on preprocessing. The actual loading from the ulmbin file happens only in the very first tile in this case, because the memory block for the base level tile remains cached in main memory then. The two implementations do roughly the same computations in the preprocessing procedure (creation of the relevant memory structures, unpacking of the variable-length integers, calculation of polygon centers for icon placement, etc.), except that the OpenGL ES 2.0 library additionally duplicates line vertices for the line tessellation and triangulates each polygon with

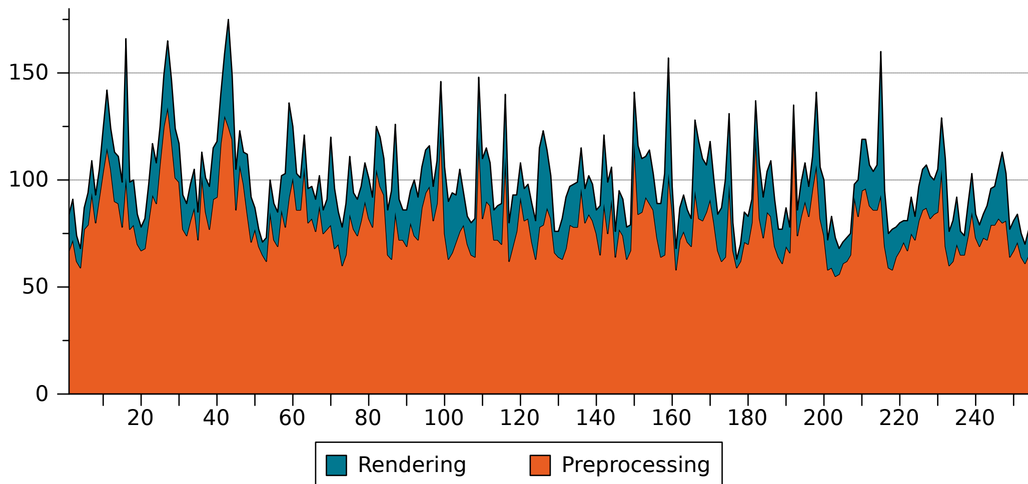


Figure 4.2: The multi-tile time of the OpenGL rendering library split into preprocessing and rendering.

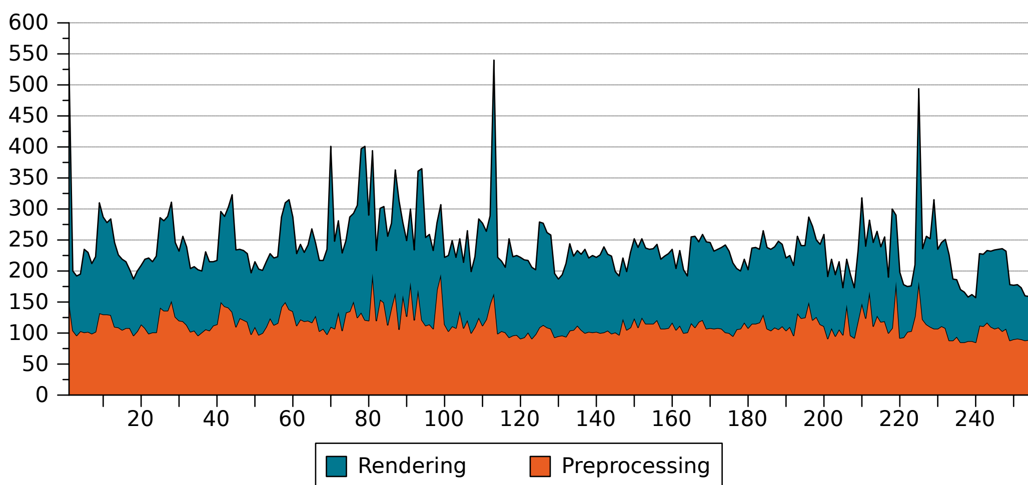


Figure 4.3: The multi-tile time of the CPU rendering library split into preprocessing and rendering.

the GLU tessellator.

Even though the new implementation has these additional two tasks in the preprocessing phase, it performs better than the legacy library. The preprocessing times for each tile can be seen in Figure 4.4; preprocessing in libMapRenderer takes 114.13ms on average, in libMapRendererGL only 80.22ms, which is 1.42 times faster. Since the preprocessing procedure takes place entirely on the CPU, this performance improvement is not due to usage of different hardware to do the same task, but because of much more efficient usage of data structures and a

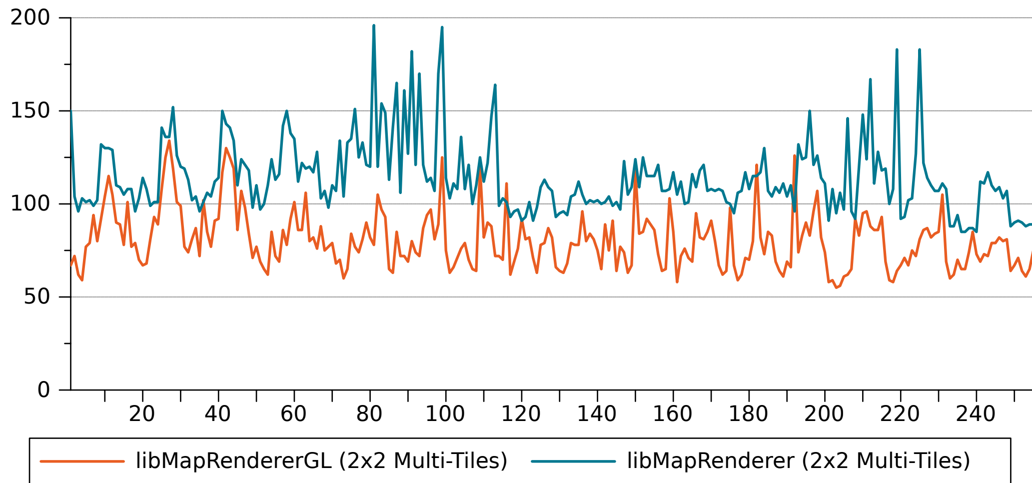


Figure 4.4: Preprocessing times in milliseconds for the test map, lower is better.

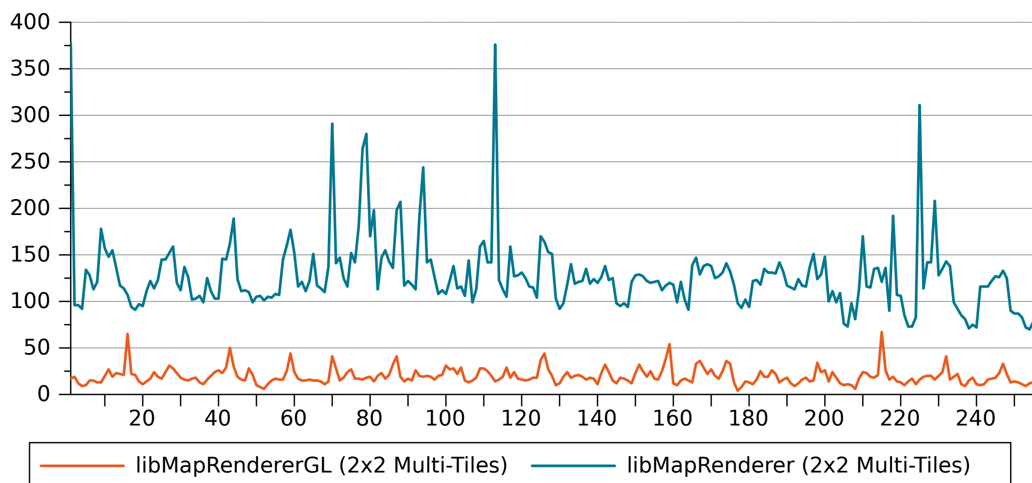


Figure 4.5: Rendering times in milliseconds for the test map, lower is better.

complete redesign of the preprocessing phase.

As the preprocessing procedure, the rendering phase was redesigned from scratch; which, together with the usage of the graphics processor for rendering, resulted in a massive speed improvement: the new library renders a tile 6.50 times faster on average. A comparison of the rendering times can be evaluated in Figure 4.5. On average, libMapRendererGL renders a multi-tile in 19.58ms, while the Quartz 2D rendering code of the old implementation needs 114.13ms for the same task. The rendering time per multi-tile is also much more consistent than in the legacy implementation, where certain tiles introduce large spikes in the rendering time.

Summing up, performance improvement was accomplished by completely re-implementing

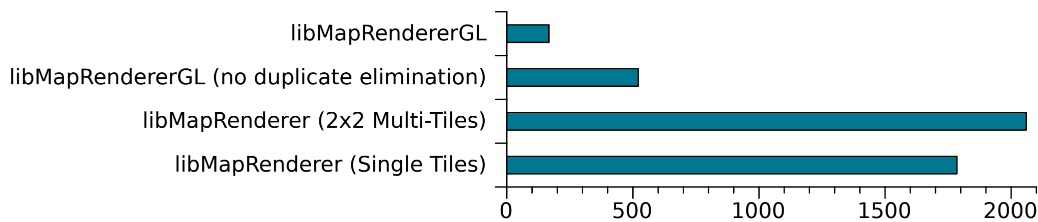


Figure 4.6: Post-Processor speed in milliseconds, lower is better

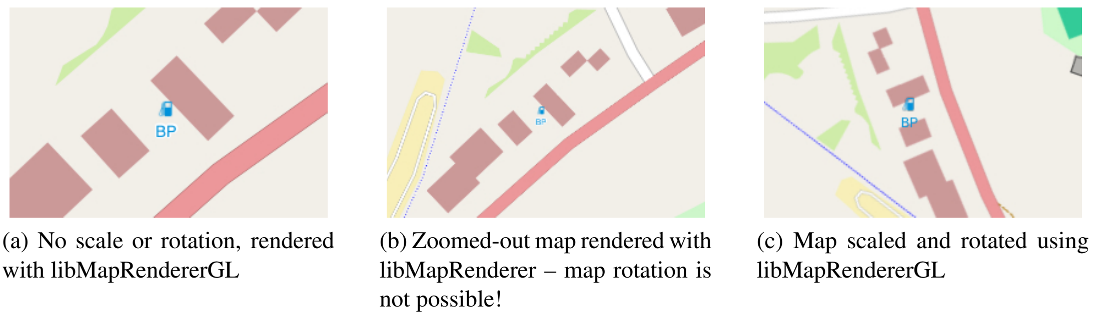
the whole rendering library with a special focus on clean design and fast execution on the one hand, and usage of the graphics hardware for the actual rendering on the other hand, which further reduces the CPU load. Not a single line of code was taken from the legacy implementation without modification; in fact, only two utility functions have been re-used at all. As a result, the prototype that has been implemented for this thesis consists of 153 source code files, 17929 lines of code and 5734 lines of comments, ignoring whitespace such as blank lines (counted with CLOC[1]).

4.1.2 Comparison of the Post-Processor Performance

Comparing the post-processor performance is even more complicated than comparing the two servers because the legacy implementation has no post-processor. Nevertheless, it is possible to measure the performance of the proposed post-processor: A 4×4 multi-tile (8×8 tile) area in the center of Vienna is rendered and the time it takes libMapRenderer to render only the dynamic features for this area is measured; all static features are excluded. Of course, those features are rendered into tile images, not directly on the screen as done in libMapRendererGL. In the new renderer, the same 16 multi-tiles are loaded without rendering tile images, once all of them are loaded, their dynamic features are added to the post-processor and the time it takes the post-processor to render them is measured. Under normal circumstances, the post-processor would never have to process so much data at once because the dynamic data junks of the multi-tiles would be added one after the other, in the order the server finishes loading them; furthermore, the post-processor would normally run in parallel to the rendering thread, not synchronized with it as in this test case. Anyway, in a test with the parameters described, both libraries fulfill the same task so the measured times can be compared. As before, both single tiles and 2×2 multi-tiles have been used in the old renderer - results of the test can be seen in Figure 4.6.

It can clearly be seen that libMapRendererGL is able to render the dynamic features much faster. Part of this is due to the hybrid rendering architecture – in the old implementation, features near tile borders are drawn on several tiles (and are then clipped), while the hybrid architecture avoids this. When the elimination of such duplicates is deactivated, the time needed in the post-processor rises by a factor of 3.12, from 123ms to 521ms. A large part of the time the legacy renderer spends on these features seems to be used on setting up required data structures. This can be seen in the difference between single tile and multi-tile rendering: although only a quarter

[1]Count Lines Of Code, `http://cloc.sourceforge.net`



(a) No scale or rotation, rendered with libMapRendererGL

(b) Zoomed-out map rendered with libMapRenderer – map rotation is not possible!

(c) Map scaled and rotated using libMapRendererGL

Figure 4.7: A labeled icon rendered with both renderers. Note that when rotating and/or zooming the map, both the icon and the text stay the same size and orientation in the proposed implementation.

of the area is processed when using single tiles, rendering time decreases only from 2060ms to 1785ms.

Almost all the time used by the post-processor in libMapRendererGL is spent on preprocessing the dynamic features; for instance, glyphs are rendered with FreeType and all glyphs of a label are blended together into a bitmap in main memory. As previously mentioned, this would normally happen in parallel to rendering the tiles in the client. The actual rendering step where icons and labels are drawn takes less than 1ms for the scene described.

4.2 Quality

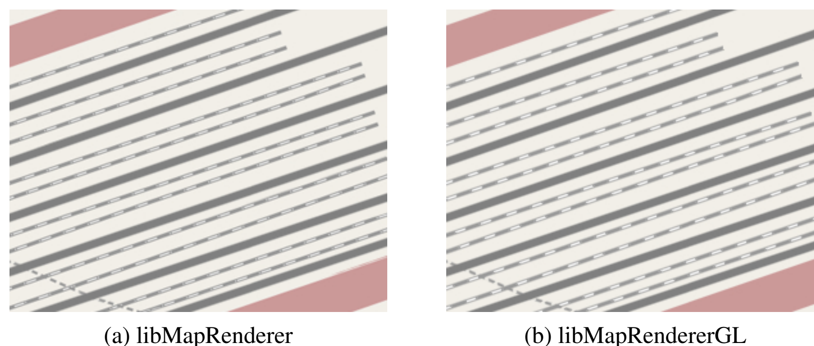
In this section, the rendering quality of libMapRendererGL is first compared to the previous solution, libMapRenderer, regarding different aspects of rendering. Later some comparisons with scientific work, presented in Chapter 2, are performed.

4.2.1 Comparison to libMapRenderer

The three major feature types, points, lines and polygons, are here compared to the previous rendering library. Additionally, text rendering, limited to horizontal text, is evaluated.

Point Rendering

In the old rendering library, point rendering is very simple. The contents of the icon bitmap are simply blended into the final map image. However, the fact that the icon is baked into the tile image has several disadvantages: for once, the icon is not presented pixel-exact on the screen, so some kind of filter is applied to the icon as soon as the map is zoomed in or out. While this filtering might not lead to severe artifacts, the icon is at least blurred when it is enlarged. Furthermore, when zooming out, the icons may be scaled down to a point where they are no longer properly recognizable. In the presented work, most icons are drawn in the post-processing stage; they are always displayed fronto-parallel and in their native resolution, so no resizing or



(a) libMapRenderer (b) libMapRendererGL

Figure 4.8: A train station rendered with both libraries. Note how the white dashes are much better visible with the proposed implementation because the legacy renderer performs MSAA on the final image, resulting in a slight blur.

filtering of icons ever happens. Icons that are drawn in the server (and are therefore baked into the tile image), include only very simple images like colored squares used, for instance, for single trees; those icons are hand-selected to ensure that all higher-level icons are drawn in the post-processor.

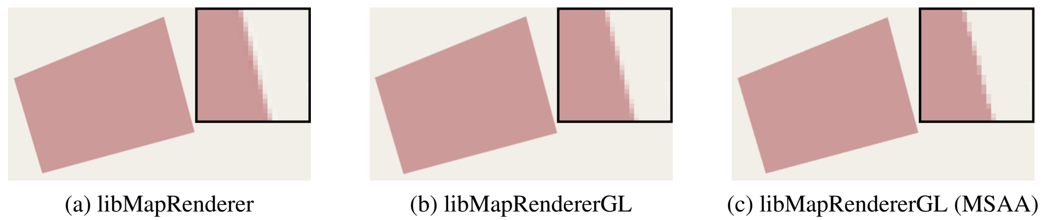
A second disadvantage of baking icons into tile images is that most icons are not rotationally invariant, so the map cannot be rotated while maintaining proper upright display of the icons. This problem is also avoided with the post-processing approach, as can be seen in Figure 4.7.

Line Rendering

Since Quartz 2D performs "correct" line rendering, i.e. each line dash is rendered as a separate line segment and antialiasing is performed by convolution with a filter kernel, better results than reached with the legacy library are not possible. The goal was therefore to speed up line rendering with hardware acceleration while maintaining high-quality lines. The speed up was already shown in Section 4.1, the quality of the rendered lines shall be evaluated here.

As described in Section 3.2.4, line antialiasing is performed using the correct pixel-line-segment distance, which is then converted to an intensity value for that pixel. Gupta and Sproull [10] stated that for any circularly symmetric filter kernel, this method yields the same result as a convolution of the line with the filter kernel. The filter currently implemented is the cubic `smoothstep` function, but it can easily be replaced by any mathematical function encoding a circularly symmetric filter kernel.

The proposed dashing algorithm is able to produce dashes with correct line caps as stated in Section 3.2.4. The shape of the cap is calculated analytically in the shader, and antialiasing is done using the same method as used for general line antialiasing. Therefore dashes are also rendered equally well in the proposed system as in libMapRenderer; very fine dashes look even better when rendered with the proposed method, because Quartz 2D performs MSAA on the final image, which blurs fine features noticeably. A comparsion can be seen in Figure 4.8.



(a) libMapRenderer (b) libMapRendererGL (c) libMapRendererGL (MSAA)

Figure 4.9: A polygon rendered with both libraries. In the enlarged section, it can clearly be seen, that MSAA produces a “blockier” result than the implemented antialiasing method.

Polygon Rendering

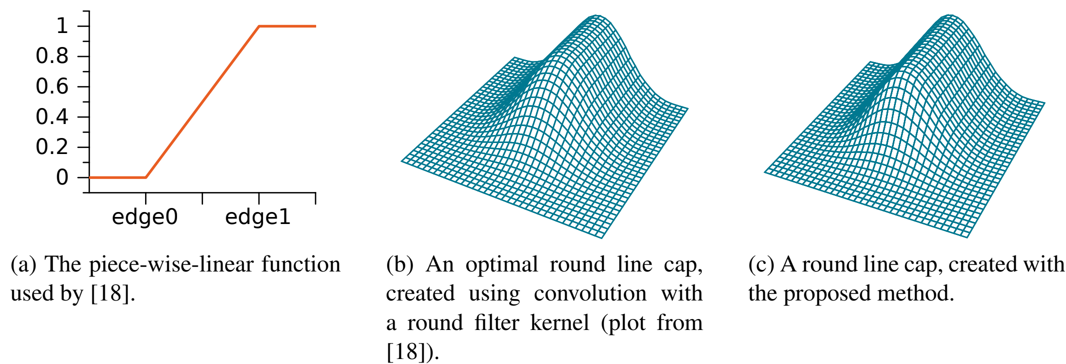
Similar to line rendering, the polygon rendering of Quartz 2D can only be used as reference solution rather than trying to improve it. Polygon antialiasing is done with a bit of a hack in the proposed work, as the polygon edges are simply overdrawn with lines of the correct color. Results of this method, compared to a polygon rendered with the previous renderer, can be seen in Figure 4.9. The additional rendering pass involved with this method slows down rendering by about 15%, as an alternative, MSAA, which imposes only about 3% slow down, can be turned on in the server; however, since the whole image is then blurred, this makes fine features like dashes less clear (see also Figure 4.8). Additionally, the implemented method produces a smoother result than MSAA.

Text Rendering

In the legacy rendering library, text is baked directly into the tile images, just as everything else. This leads to the same problems as with point rendering (described earlier in this section). For text, both rotation and down-scaling are even more severe. Note that, while the old renderer cannot reproduce text in a pixel-exact way on the screen, the artifact mentioned in Section 3.5.1, is not severe because Quartz 2D performs MSAA on the whole tile image. A downside of the proposed implementation is that text halos, which increase readability in the legacy implementation, cannot be rendered; a comparison of text rendering with map rotation and scaling can be seen in the point rendering section, Figure 4.7.

4.2.2 Comparison to McNamara et al.

McNamara et al. [18] state that the optimal solution to the line antialiasing problem is the convolution of the line with a filter kernel. Figure 4.10 shows a comparison of the intensity values of a convolution and the proposed solution. While the actual values are not equal, because McNamara et al. use a piece-wise-linear function (Figure 4.10a) to map distance to intensity while the proposed implementation uses `smoothstep` to do the same task, there is no visible difference between the plots. McNamara et al. achieve such a good solution only with their most complicated method that requires intensity calculation for all four boundary edges; the proposed method needs only a single intensity calculation because the true pixel-line-segment distance is known.



(a) The piece-wise-linear function used by [18].

(b) An optimal round line cap, created using convolution with a round filter kernel (plot from [18]).

(c) A round line cap, created with the proposed method.

Figure 4.10: Line cap comparison with the optimal solution

4.3 Features

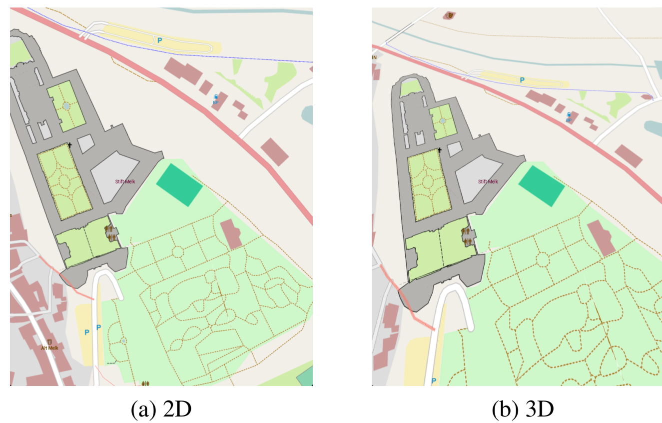
In this section, the features that are new in comparison to the previous rendering library are introduced. With regard to libMapRenderer, most new features arise from the novel hybrid rendering architecture. Later in this section, the new possibilities of the line rendering algorithm are related to previous scientific work.

4.3.1 Features of the Hybrid Architecture

The old library, libMapRendererGL was designed from scratch to offer certain features that the old rendering library could not provide. The most prominent of these features is a 3D mode, to enable a fly-over-like map view. In addition to simply changing the view point, the proposed implementation is capable of rendering icons and text parallel to the screen, so that they always stay undistorted no matter how tilted the current map view is.

In fact, icons and text not only stay upright when tilting the map, they also do so when rotating in both the 2D and 3D modes. This makes it possible to have a “compass mode” where the map automatically rotates to reflect the current orientation of the device, if it has a built-in electronic compass. Figure 4.11 illustrates the 3D view and also shows some icons and labels that are rendered fronto-parallel.

The post-processing architecture also fixes another problem of the legacy implementation: due to rendering everything (including text) into the tile images, certain text items like road names, which are placed dynamically along a road, are not always placed at the exact same position on neighboring tiles. This results in a visible artifact when the text does not completely fit in a single tile and crosses a tile border; the text is then clipped at the border. An example of such an artifact can be seen in Figure 4.12. The new rendering library circumvents this problem by rendering only static features (that have a fixed position) into tile images. If static features cross tile borders, they are rendered at the same position by all tiles involved and can therefore be tiled seamlessly. Dynamic features (whose position might not be fixed but determined by the post-processor) are rendered in the post-processing step, not into the tile images. This does not only save redundant rendering calls, but also eliminates the bug described.



(a) 2D (b) 3D

Figure 4.11: Melk Abbey, Austria, in both 2D and 3D Modes. Note how the icons and the text stay upright and parallel to the screen.

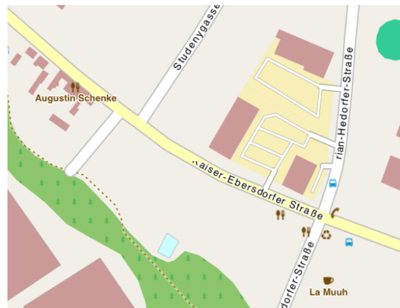


Figure 4.12: Text being cut at tile borders in the legacy rendering library.

A possible future enhancement is the overlay of routing data. This data (generated either online or locally on the device) can be integrated into the post-processing architecture easily to enable a navigation mode.

Beside the advantages, the new implementation also has some disadvantages when compared to libMapRenderer. Line rendering, in particular, lacks some features in comparison to real vector drawing libraries like Quartz 2D. Of the three possible line joins, only the miter join is currently implemented, and while the bevel join is a possible extension, the round join is very hard to include in the current implementation. Additionally, since line vertices have to be duplicated on the CPU, the memory consumption for a line geometry is raised. Without the duplication, a line strip consisting of n points needs $2 * n$ floats; with duplication, $2 * (2 * n + 2)$ floats are needed to store the coordinates, a single byte is used to store the extrusion direction for each vertex, a float per vertex stores the angle at each join and for dashing an additional float per vertex (the line length up to this vertex) is needed. Summing up, a line strip of n points needs $4 * (2 * n + 2)$ floats plus $(2 * n + 2)$ bytes of memory.

The overall experience of the mapping application powered by the proposed rendering li-

brary is changed to a smoother, more modern user experience. The fact that many features are obviously "not inside" the map, but "stand on" the map (in 3D mode) draws attention to the icons, especially during an animation phase, because they then move differently from base map image. Previously, moving the map on the screen created the impression of moving a large paper map around (the original slippy map idea); the new renderer, however, causes an impression of a live-generated, interactive map.

Platform Independency

Although iOS is the primary target platform for the rendering library, platform independency is an important new feature of the proposed implementation. The goal was to be able to compile and run the whole code on each platform while keeping the platform-dependent code to a minimum. The following requirements must be fulfilled to successfully compile libMapRendererGL:

- OpenGL ES 2.0 (headers and library)
- POSIX Threads
- The C++ Standard Template Library
- A C++ compiler with support for RTTI[2]

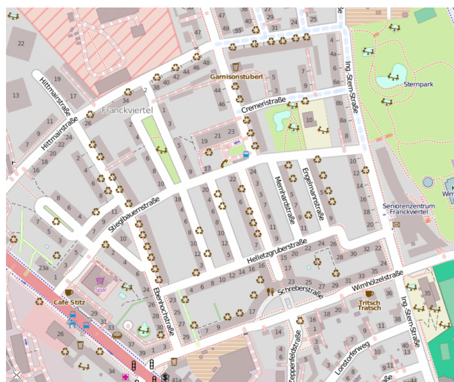
Apart from minor code additions needed for each supported platform (mainly to locate the correct OpenGL headers), the whole rendering library will compile on any platform providing these features.

While the previous rendering library was also ported from iOS to Android, the port needed severe modification and a lot of additional code in many places. Also, some features are missing from the Android port, because the actual rendering was done by a 3rd-party library on both platforms – AGG on Android and Quartz 2D on iOS. Since libMapRendererGL handles all rendering itself, the results will be the same on each platform.

4.3.2 Novel Features of the Line Rendering Algorithm

To our knowledge, the proposed line rendering algorithm is the first method to combine antialiased line rendering of line strips with arbitrary width, all three different line cap styles and dashing (including the correct line cap style for each dash). While the methods of McNamara et al. [18] and Chan and Durand [6] produce visually appealing lines and round line caps, other types of caps cannot be rendered, and no line strips (and therefore also no line joins) are supported, although Chan and Durand simulate round line joins by drawing two line segments with round line caps. Persson [28] tessellates lines and renders them as triangle strips; however, cap styles and join styles must be encoded in the geometry of the line, while the presented method strictly separates line geometry from the line style. Bærentzen et al. [3] calculate the pixel-line-segment distance in a similar way as in the proposed work, producing equal round line caps, but

[2]Runtime Type Information



(a) Suburbs of Linz with a lot of dustbins.

(b) A courtyard in Vienna including every single tree.

Figure 4.13: Overly crowded areas in the OpenStreetMap

line strips can only be rendered as in [6], by rendering each segment separately. Dashing is not supported with this method.

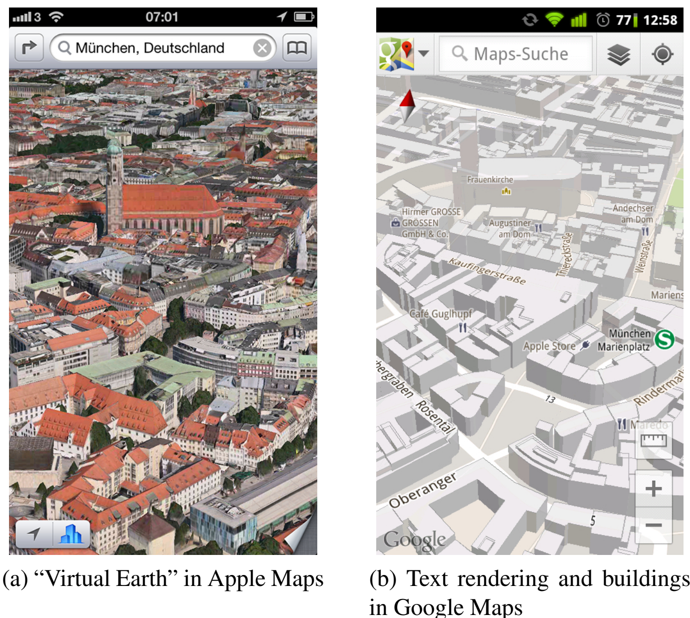
Seetharamaiah et al. [39] are able to decorate lines with a user-defined stipple pattern, however, since their implementation discards fragments in an "off" interval, no antialiasing or correct caps for line dashes are done. The same limitations apply to the algorithm of Poddar [31],7 who invented the method of storing the dash pattern in a texture, which was extended to store not only binary but distance information in the proposed algorithm.

4.3.3 Features compared to popular Map Rendering Applications

The most obvious difference of the proposed renderer and popular mapping applications like Google Maps and Apple Maps (apart from the style) is the feature richness of the OpenStreetMap data. Both mentioned mapping applications retrieve their data from commercial map data providers like NAVTEQ[3] and TomTom[4], while all data in the OpenStreetMap stems either from out-of-copyright maps or is manually added by members of the community. As a result, data from the OpenStreetMap project is usually more up-to-date than those from commercial providers. Also, as already mentioned, OpenStreetMap data often contains much more information about POIs. This has both a good and a bad side; on the one hand there are almost no possible POIs that are not mapped in one or the other way, but on the other hand the map may seem crowded – for instance in some areas of the Austrian city Linz where every single dustbin is shown, or in some courtyards in Vienna, where all the trees are displayed on the map. Figure 4.13 shows screenshots from `openstreetmap.org` for both of those areas.

Like the presented renderer, both Google Maps and Apple Maps use vector data to render the map locally on the device, but both of them download the data live from the internet. Of course, no details about this are known, but it can be assumed, that the data is specifically tailored to the

[3] `http://www.navteq.com`
[4] `http://www.tomtom.com`



(a) "Virtual Earth" in Apple Maps (b) Text rendering and buildings in Google Maps

Figure 4.14: Google Maps and Apple Maps

needs of the device and the current settings of the app, so that no further preprocessing needs to be done prior to rendering. Google Maps provides the possibility to store sections of the map on the device and the download size of the map also suggests the focus on quick rendering as opposed to small file size: a map of Vienna, when downloaded from Google Maps, needs more than 100 megabytes of space, while the ulmbin map of roughly the same area is exactly 8.8 megabytes in size.

As for the architecture of the proposed library, Google Maps has inspired many ideas: they seem to have some kind of hybrid architecture too, because sometimes a tile-based structure is clearly visible, but many features are obviously rendered over this structure. Both applications provide a 3D view and are able to draw three-dimensional buildings in the most detailed zoomlevels, which is planned as a possible future extension of the proposed library. On Apple Maps, the buildings are even textured with satellite and aerial imagery, obviously aiming at a "virtual earth" experience (Figure 4.14a), although this technology seems to be in its infancy, considering the amount of bad press it received almost instantly after its launch.

As for text rendering, Google Maps seems to take a slightly different approach than the one presented here. Text items are not always rendered fronto-parallel, but seem to "pop out" of the map sometimes, an example of this can be seen in Figure 4.14b. However this leads to a rather unpleasant visual discontinuity, which is why the presented approach was favored.

CHAPTER 5

Conclusions

In this work, an OpenGL ES 2.0 map rendering engine for mobile platforms has been presented. The renderer uses a custom vector data format based on data from the OpenStreetMap project to store maps, and dynamically renders them using a special stylesheet. In this section, the major challenges encountered during the implementation are briefly discussed and the most important findings are stated. Finally, missing features and ideas on how to implement them are listed.

5.1 Challenges

The first and probably most important challenge was to design a fast, scalable and memory efficient architecture. When considering the amount of data it soon became obvious that a full realtime rendering engine would not be feasible with the restricted capabilities of mobile devices. As some of the required features (e.g. map rotation) can not be implemented in a tile-based system, because text and icons would be upside-down eventually, the hybrid rendering approach that has been described in the previous sections was designed. To avoid user interface "hickups", tile rendering cannot be done in the main thread, a thread for the tile server had to be created instead.

An optimization to the data loading of the previous rendering library has been introduced: a concept called multi-tiles, where each multi-tile covers a square area of actual tiles. Redundant feature loading can be avoided this way; the size of the multi-tile is only constrained by memory limitations, multi-tile sizes up to 4×4 tiles have been tested and could be rendered in almost the same time a single tile needed to render, while covering 16 times the area of a normal tile.

Communication between the server thread and the main thread (where the client is run) can be done without a specific communication protocol because they share the same address space. The design and implementation of the data structures to provide a thread-safe rendering queue and tile cache was another challenge.

Inside the server, line rendering is the most complex part. A line tessellation algorithm was implemented to enable rendering of line strips with arbitrary style. The line renderer is able to

draw all three different line cap styles (round, butt and square caps) and a line can be decorated with a dash pattern. Each dash is drawn using the correct cap style, which has not been covered in realtime rendering literature previously.

To determine the tiles that are visible on the screen, a tile selection algorithm is used. This algorithm exploits the flat geometry of the map and uses a polygon rasterization algorithm to calculate the tiles touched by the projected view frustum. While this problem is similar to the problem solved in the clipmap approach by Tanner et al. [48], an exact tile selection was needed instead of the course estimation done in clipmapping.

To render the large amounts of icons needed for OpenStreetMap data, all icons are packed into a texture atlas at application startup. A simple but fast online square packing algorithm is presented to place the icons in the texture atlas. The algorithm takes advantage of the fact that the icons are square, of similar sizes and small compared to the size of the texture atlas.

The last challenge that needs to be solved, is the post-processing part. An efficient data structure to handle duplicate items (from neighboring tiles) and dynamic addition and removal of features was implemented. Additionally a message passing system was designed to handle communication between the client and the post-processor while not slowing down either of the two.

5.2 Findings

An important lesson to learn from the implementation work for this thesis is that while using the GPU for rendering significantly speeds up tile generation, the CPU is still the limiting factor most of the time. Even though the features of map data do not scale well with GPU architecture (many small features in contrast to large chunks of geometry, which would be best for graphics hardware), only a fraction of the total time is spent on rendering.

During development, the importance of minimizing state changes for OpenGL performance was confirmed, for instance, the usage of a texture atlas for the point icons improved the rendering speed drastically.

5.3 Future Work

The aim of this work was to create a prototype to showcase the most important features of an OpenGL map rendering library. In the future, this prototype will be extended and revised to be eventually released as a replacement for libMapRenderer. There are several research topics that need to be investigated; inclusion of more than a single line join and fast text rendering being only two of those.

As stated in Section 4.1, more than 80% of the time spent on one tile is spent on preprocessing. This time could be reduced drastically by improving the data format to require less computation during loading. A tradeoff between file size and loading speed will have to be found. The data format could also be optimized such that everything that has to be drawn on the same layer (for instance all road casings) can easily be grouped together into a single draw call; related to this, the stylesheet can be optimized to reduce the number of layers (and thus, the overdraw which would again speed up rendering).

Round and bevel joins have not been implemented in the proposed system because the mapping stylesheet in use does not enforce them. Bevel joins could be realized using a similar trick as is used for the butt line caps though: two additional vertices would have to be inserted at each join; for miter joins this would result in two duplicate vertices, for bevel joins only the inner vertex would be doubled. If and how round joins can be included in the line renderer remains to be researched.

The dilemma about text rendering has already been described in Section 3.5.1, the fastest method to solve that dilemma is still to be found. One possible solution is to use a texture atlas for all horizontal glyphs, the proposed packing algorithm (Section 3.2.4) is perfectly suited for this. For non-horizontal text, a texture atlas is not suitable because of the huge number of combinations of glyph, glyph size and angle. Tests will have to be done to clarify if letting the font loading library, FreeType, do the rendering of these glyphs in realtime is feasible, or if some kind of cache is needed, and if the visual artifacts introduced by not displaying glyphs in a pixel-exact way are tolerable. Especially MSAA could assist in avoiding those artifacts.

The post-processing stage in general is another important evaluation area: an efficient screen space collision detection is needed, and it must be investigated, how results of the collision detection can be transferred to the next frame, even when the camera position changed. For road names, which are rendered approximately along roads, a way to keep the rendered strings is needed, even when the camera moves; for camera rotations the road names must always be recalculated, but the ability to re-use at least certain glyphs would be desirable.

Bibliography

[1]Alastair Aitchison. The Google Maps / Bing Maps Spherical Mercator Projection, 2011. URL http://goo.gl/rgggZ.

[2]Sven Andersson and Jhonny Göransson. *Virtual Texturing with WebGL*. Master's thesis, Chalmers University of Technology, 2012.

[3]Jakob Andreas Bærentzen, Steen Lund Nielsen, Mikkel Gjøl, and Bent D. Larsen. Two methods for antialiased wireframe drawing with hidden line removal. In *Proceedings of the 24th Spring Conference on Computer Graphics*, pages 171–177, Budmerice, Slovakia, 2010. ACM Press.

[4]Sean Barret. Sparse Virtual Textures. Game Developer Conference, 2008. URL http://goo.gl/gr8GC.

[5]Jack E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[6]Eric Chan and Frédo Durand. Fast Prefiltered Lines. In *GPU Gems 2*, chapter 22. Addison-Wesley, 2005.

[7]Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, and Kevin P. Smith. The OpenGL Graphics System Utility Library (Version 1.3). Technical Report November, SGI Inc., 1998.

[8]Roger Crawfis, Eric Noble, Michael Ford, Frederic Kuck, and Eric Wagner. Clipmapping on the GPU. Technical report, Ohio State University, 2007.

[9]Michael R. Dunlavey. Efficient polygon-filling algorithms for raster displays. *ACM Transactions on Graphics*, 2(4):264–273, October 1983.

[10]Satish Gupta and Robert F. Sproull. Filtering Edges for Gray-Scale Displays. *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, 15(3):1–5, 1981.

[11]Mordechai Haklay and Patrick Weber. OpenStreetMap: User-Generated street Maps. *IEEE Pervasice Computing*, 7(4):12–18, 2008.

[12]Donald Hearn and Pauline M. Baker. *Computer Graphics with OpenGL*. Pearson Prentice Hall, 3rd edition, 2004. ISBN 0-13-120238-3.

[13]Khronos Group. Khronos Releases OpenGL ES 3.0 Specification to Bring Mobile 3D Graphics to the Next Level, 2012. URL `http://goo.gl/qVilb`.

[14]Khronos Group. OpenVG, 2012. URL `http://goo.gl/LiHH3`.

[15]C. C. Lee and D. T. Lee. A simple on-line bin-packing algorithm. *Journal of the ACM*, 32 (3):562–572, July 1985.

[16]Sang-Yun Lee, Sunghwan Kim, Jihoon Chung, and Byung-Uk Choi. Salable Vector Graphics (OpenVG) for Creating Animation Image in Embedded Systems. In *Knowledge-Based Intelligent Information and Engineering Systems*, volume 4693, pages 99–108. Springer Berlin / Heidelberg, 2007.

[17]Albert Julian Mayer. *Virtual Texturing*. Master's thesis, Vienna University of Technology, 2010.

[18]Robert McNamara, Joel McCormack, and Norman P. Jouppi. Prefiltered antialiased lines using half-plane distance functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 77–85, New York, New York, USA, 2000. ACM Press.

[19]F. K. Miyazawa and Y. Wakabayashi. Parametric on-line algorithms for packing rectangles and boxes. *European Journal of Operational Research*, 150(2):281–292, October 2003.

[20]Mark Stephen Monmonier. *Rhumb Lines and Map Wars - A Social History of the Mercator Projection*. University Of Chicago Press, 2004. ISBN 0226534316.

[21]John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinite-Reality: A Real-Time Graphics System. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 293–302. ACM Press, 1997.

[22]William M. Mularie. Department of Defense World Geodetic System 1984. Technical report, National Imagery and Mapping Agency, 2000.

[23]Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL ES 2.0 Programming Guide*. Addison Wesley, 2009. ISBN 978-0-321-50279-7.

[24]Andreas Neu. *Virtual Texturing*. Bachelor's thesis, RWTH Aachen University, 2010.

[25]Harry Nyquist. Certain Topics in Telegraph Transmission Theory. *Transactions of the American Institute of Electrical Engineers*, 47(2):617–644, 1928.

[26]OpenStreetMap. OpenStreetMap Wiki, 2012. URL `http://goo.gl/DGbJH`.

[27]Theo Pavlidis. Contour filling in raster graphics. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '81, pages 29–36, Dallas, Texas, August 1981. ACM.

[28]Per Persson. Method of and arrangement for rendering a path. European Patent Application EP2107528, 2009.

[29]Klokan Petr P řidal. Tiles à la Google Maps: Coordinates, Tile Bounds and Projection, 2008. URL `http://goo.gl/ubux6`.

[30]Juan Pineda. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 17–20, New York, NY, USA, August 1988. ACM Press.

[31]Bimal Poddar. Line stipple pattern emulation through texture mapping. United States Patent US 7,280,114 B2, 2007.

[32]Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer New York, 1st edition, 1985.

[33]Zheng Qin, Michael D. McCool, and Craig S. Kaplan. Real-time texture-mapped vector glyphs. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 125–132, Redwood City, California, 2006. ACM Press.

[34]Qualcomm Inc. Adreno ™ Graphics Processing Units, 2012. URL `http://goo.gl/IVwPG`.

[35]Prakash Ramanan, Donna J. Brown, C. C. Lee, and D. T. Lee. On-line bin packing in linear time. *Journal of Algorithms*, 10(3):305–326, September 1989.

[36]Lars H. Rohwedder. Normal Square Mercator projection, 2006. URL `http://goo.gl/0swk7`.

[37]Randi Rost and Dan Rice. OpenVG. In *ACM SIGGRAPH 2006 Courses*, Boston, Massachusetts, 2006. ACM Press.

[38]A.J. Rueda, R.J. Segura, F.R. Feito, and J. Ruiz de Miras. Rasterizing complex polygons without tessellations. *Graphical Models*, 66(3):127–132, May 2004.

[39]Avinash Seetharamaiah, Bimal Poddar, and William B. Sadler. System and method for rasterization order independent line stipple. United States Patent Application US 2007/0146366 A1, 2007.

[40]Steven S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, September 2002.

[41]Antonio Seoane, Javier Taibo, Luis Hernández, Rubén López, and Alberto Jaspe. Hardware-Independent Clipmapping. In *The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2007.

[42]SGI Inc. OpenGL, 1992. URL http://goo.gl/5A6wH.

[43]SGI Inc. OpenGL ® Sample Implementation, 2000. URL http://goo.gl/U3QZ4.

[44]Claude Elwood Shannon. Communication in the Presence of Noise. *Proceedings of the Institute of Radio Engineers (IRE)*, 37(1):10–21, 1947.

[45]Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. *Applied Computational Geometry: Towards Geometric Engineering*, 1148:203–222, 1996.

[46]Guus Sliepen. Modern OpenGL Text Rendering, 2011. URL http://goo.gl/0BzC6.

[47]John Snyder, Pedro V. Sander, Hugues Hoppe, and Steven Gortler. Discontinuity Edge Overdraw. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 167–174, New York, New York, USA, 2001. ACM Press.

[48]Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap: A Virtual Mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158. ACM Press, 1998.

[49]The Austin Common Standards Revision Group. IEEE 1003.1-2008 Standard for Information Technology - Portable Operating System Interface (POSIX®), 2008. URL http://goo.gl/kBjOE.

[50]Kenneth Turkowski. Anti-Aliasing through the Use of Coordinate Transformations. *ACM Transactions on Graphics (TOG)*, 1(3):215–234, 1982.

[51]David Turner. Hinting and bitmap rendering. FreeType Glyph Conventions, 2000. URL http://goo.gl/v3GFJ.

[52]Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, Detroit, Michigan, United States, 1983. ACM Press.