Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Bratlie, Cecilia Norevik
Oueslati, Eimen
Skålerud, Nils Petter

Rendering Vector-Based Geographical Maps with Qt

Bachelor's thesis in Bachelor of Programming, Bachelor of Engineering in Computer Science
Supervisor: Palomar, Rafael
Co-supervisor: Su, Xiang

May 2024

NTNU
Norwegian University of
Science and Technology

Bratlie, Cecilia Norevik
Oueslati, Eimen
Skålerud, Nils Petter

Rendering Vector-Based Geographical Maps with Qt

Bachelor's thesis in Bachelor of Programming, Bachelor of Engineering in Computer Science
Supervisor: Palomar, Rafael
Co-supervisor: Su, Xiang
May 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Geographical maps are an important tool when navigating unfamiliar areas and terrain, and they power logistics worldwide by modeling the shape, size, and location of features, cities, and countries. The Qt framework currently does not support visualizing maps using vector graphics. Vector graphics have several benefits in terms of visual fidelity and visualizing maps, including scaling of imagery without loss of visual quality. The Qt Group wishes to investigate how to integrate this technology into their software ecosystem. Providing vector-based visualization of maps may make the Qt framework a more competitive product. This thesis establishes a proof-of-concept software that demonstrates how to render maps with the Qt framework and vector graphics.

Sammendrag

Geografiske kart er et sentralt verktøy ved utforsking av ukjente områder og terreng, og de er avgjørende for logistikk over hele verden ved å modellere formen, størrelsen og lokasjonen til landskapsformasjoner, byer og land. Per i dag støtter ikke Qt-rammeverket visualisering av kart ved bruk av vektorgrafikk. Vektorgrafikk har flere fordeler, inkludert muligheten til å skalere bilder uten tap av bildekvalitet. *The Qt Group* ønsker å utforske hvordan denne teknologien kan innlemmes i deres programvaremiljø. Å kunne vise vektorbaserte kart kan bidra til å gjøre Qt-rammeverket til et mer konkurransedyktig produkt. Denne avhandlingen etablerer en konseptutprøvende programvare som demonstrerer hvordan kart kan framstilles med Qt-rammeverket og vektorgrafikk.

Preface

Two supervisors have guided the work on this thesis. We greatly appreciate Associate Professor Xiang Su's supervision and support of the project during its first two months. He helped us come up with the original project plan and encouraged us to perform software testing on the implemented software. After that, Associate Professor Rafael Palomar assumed the role of supervisor. We would like to thank him for his help and guidance during the final months of the project. His knowledge of C++, software testing, and reporting was invaluable to us during the project's final months, and we are incredibly grateful.

The Qt Group, the product owner, has been a joy to work with. Everyone from Qt has been supportive, kind, and helpful to us, providing input, support, and even cake (!) when working with us. We would like to especially thank Matthias Rauter, our contact person at Qt, for his support while working on the project.

Contents

List of Figures

List of Tables

Listings

Acronyms

CI Continuous Integration. 32, 62, 64, 65, 74

FR Functional Requirement. 26

GUI Graphical User Interface. 1, 2, 27, 33, 37, 38, 48, 77

MSVC Microsoft Visual C++. 58, 60

NFR Non-Functional Requirement. 27

NTNU Norwegian University of Science and Technology. 4, 76

PBF Protocolbuffer Binary Format. 41

TDD Test-Driven Development. 4

UN United Nations. 74

Glossary

aspect-ratio The mathematical relation between width and height in a rectangle. Expressed as a scalar width $\div$ height. 42, 43, 136

hardware-acceleration Accelerating a workload by using dedicated hardware. This generally allows higher throughput and improved power-efficiency [1]. In this paper, it is mostly used in the context of graphics where dedicated Graphics Processing Units are employed to improve graphics performance. 13

Merlin A component of the software developed in this thesis. This component is responsible for performing automated graphical output tests. 64

pixel map Digital imagery encoded as a table of color-values. Also commonly referred to as “bitmaps”. Individual cells of this table may be referred to as a “pixel”. This kind of image-encoding you will find when using digital photography, and is also employed in image-file-formats such as PNG or JPEG. 11

Protobuf Google Protocol Buffers. 47, 58

tile-normalized coordinate Two-dimensional coordinates, denoted by the symbol $\mathbf{TN} \in [0,1]^2$. The components 0 and 1 map to the directions X and Y, respectively. In the X direction, 0 maps to the left edge of the map-tile and 1 maps to the right edge. In the Y direction, 0 maps to the top edge of the map-tile and 1 maps to the bottom edge. 134

tile-position-triplet A triplet $\mathbf{T}$ containing components describing a tile's position within the Web Mercator projection. The components include zoom-level $\mathbf{z} \in \{0, 1, \ldots, 16\}$ and the zero-based indexed coordinates $(\mathbf{x}, \mathbf{y}) \in \{0, 1, ..., 2^{\mathbf{z}}-1\}^2$. The formal definition is described in Figure E.2. 135

TileLoader A component of the software developed in this thesis. This component is responsible for loading tiles on background-threads as well as storing and caching them. 31–34, 36, 49, 59

viewport Rectangle that is projected onto the world map based on the displays view configuration. These components include center coordinates and zoom level. The viewport forms a relation between a segment on the world map and what is displayed on the display. See section 5.2 for more. 13, 42, 43, 136

world-normalized coordinate Two-dimensional coordinates, denoted by the symbol $\mathbf{WN} \in [0,1]^2$. The components 0 and 1 map to the directions X and Y, respectively. In direction X, 0 maps to the left edge of the world-map and 1 maps to the right edge. In direction Y, 0 maps to the top edge of the world-map and 1 maps to the bottom edge. 134, 137

zoom-level A scalar property of the viewport. Determines the size of the area of the world map that is projected into the viewport. 13, 42, 43, 135, 136

Chapter 1

Introduction

1.1 Problem Domain, Delimitation, and Task Definition

1.1.1 Problem Domain

The Qt Framework [2], developed by *The Qt Group*, is a popular tool for creating computer applications with a Graphical User Interfaces (GUIs). In many applications, it is desirable to include functionality to display and interact with geographical maps as part of the user experience. In this thesis, The Qt Group and their representative will also be referred to as the "product owner" or simply "Qt".

The Qt Group has asked us to investigate how to integrate vector-based map functionality into their software ecosystem to render a world map, as opposed to utilizing traditional raster source data to generate maps (see section 2.1 for more details).

The problem domain is in the intersection between being able to load and parse vector data and then subsequently rendering the map to a display using the Qt framework. This problem domain involves computer graphics, parsing JSON data based on complex schemas, networking, and multithreading.

1.1.2 Problem Delimitation

The problem domain is vast, while the team's time and pre-existing knowledge of the topic are limited. The scope of the project has been intentionally narrowed to enable the team to achieve a few very specific goals, as outlined in section 1.4.

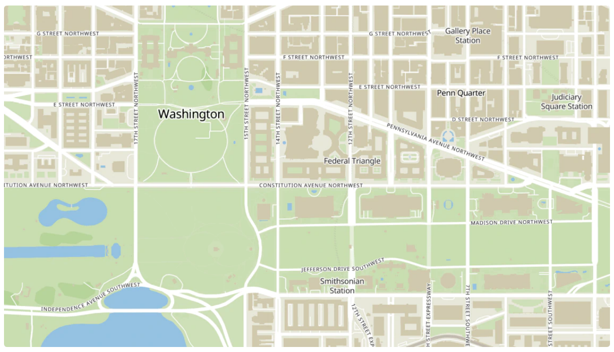


Figure 1.1: Map-style Basic-V2 example. Image pulled from MapTiler [4]

This thesis deliberately excludes alternative map display technologies, other GUI-toolkits, and features such as 3D graphics and rotation of the map. It also does not cover adaptations for web or mobile platforms. The narrow focus allows for a detailed exploration of map functionality in selected scenarios. The delimitations are fully described in section 3.2.

1.1.3 Task Definition

The team is to develop an application that can display a geographical map, and the rendered map must be generated using vector data. The map data has to be parsed and rendered in real-time and allow live user interaction (for instance zooming in and out, and moving around the map).

The application will render geographical map data supplied by the MapTiler service [3]. Specifically, it should be able to display the MapTiler map style called *Basic-V2* [4] (see Figure 1.1). The rendered map must include elements such as terrain, roads, street and place names.

1.2 Motivation

The product owner wishes to investigate how they might improve the support for map functionality in the Qt framework. At the time of writing, the Qt framework only supports raster-based map rendering. The use of raster-based rendering comes with several downsides, some of which are elaborated upon in section 2.1. Including this type of functionality may help the Qt framework become a more competitive product. Through this thesis, the product owner wishes to gain insight into what challenges must be solved in order to integrate vector-based rendering of geographical maps into their software ecosystem.

1.3 Target Audience

The thesis has been written as a report to be read by someone who has a basic understanding of programming and software engineering, such as software developers or software engineering students. The thesis pays great attention to design choices and describes the implementation details required to understand, reproduce, and further extend this work.

Assumptions About the Reader

This thesis focuses on many technical aspects of application development and assumes the reader has good foundational knowledge of computer science. The reader should have some background knowledge about the following topics:

- **Mathematics:** Basic calculus, including reading basic mathematical notation.
- **Basic Networking:** Knowing what HTTP requests are, and that HTTP responses may return both a status code and data when HTTP requests are made.
- **Foundations of Computer Graphics**
- **Computer Programming:** Understanding of core programming concepts such as data structures and classes, algorithms, and software design patterns. Multithreading and thread synchronization techniques are also relevant.

Additionally, having a good understanding of the C++ programming language (or a similar programming language like C or Java) and the Qt framework may be beneficial.

1.4 Thesis Goals

This section lists the thesis goals established by the team. To what extent the goals were reached by the end of the project, will be reflected upon in chapter 9.

Result Goals

- Deliver a proof-of-concept application that showcases vector graphics rendering of geographical maps using the Qt framework.
- Establish a project that the team can include in their professional portfolio.
- Provide useful research material for The Qt Group to implement improved support for map-functionality in the Qt Framework. The developed solution is *not* expected to be production-ready.

Effect Goals

- Demonstrate the benefits of vector-based rendering for geographical maps within the Qt framework to the broader developer community. This includes showcasing scalability and enhanced user interaction capabilities compared to raster-based rendering.
- Cultivate a collaboration between the Norwegian University of Science and Technology (NTNU) and The Qt Group. This could lead to more student projects, internships, and research initiatives to help bridge the gap between academic learning and industry needs.

Learning Goals

- Improve programming and coding skills, in particular learning how to use C++ with Qt.
- Learn how to test C++ applications.
- Improve competence in Scrum-based development.
- Improve competence on Test-Driven Development (TDD) through real-world practice.
- Learn more about best practices when developing in a team
- Learn from the development culture at Qt and integrate this into the work and adhering to Qt's development methodologies.

1.5 Team Background and Skills

In this thesis, ”the team” will refer to the authors of this report; the group of the three students presented below.

Cecila Norevik Bratlie
Bachelor of Programming (BPROG)
Cecilia had taken three classes using C or C++, but she considers herself to be more skilled in programming languages like Golang, Rust, or Kotlin. She has no experience with Qt from before. She will be responsible for networking components in the application and project quality assurance.

Eimen Oueslati
Bachelor of Engineering in Computer Science (BIDATA)
Eimen is more proficient in Java and did not know C++ ahead of the project. However, he was fairly familiar with object-oriented languages. He has also worked with Golang, MatLab, and Python. He has no experience with Qt prior to the project's start. He will be in charge of application data parsing and support project quality assurance.

Nils Petter Skålerud
Bachelor of Programming (BPROG)
Nils is very experienced with C++ and had some prior knowledge of Qt and their framework before the project began. He has also developed his own graphics engine in C++ and is very knowledgeable when it comes to graphics and rendering. Ahead of the project, Nils had experience working with advanced C++ coding practices, multi-threading, and computer graphics, though not in the context of the Qt ecosystem. Given his extensive skill and experience as a C++ developer, he will serve as both the project team leader and Scrum master.

1.5.1 Team Motivation

Nils Petter had initially sought out The Qt Group for a thesis, and the team agreed that this thesis was the one they were the most motivated to work on. The team was interested in working with C++, and not many other thesis options allowed the use of C++. The team was also interested in being able to visit the Qt offices to gain insights about professional software development environments.

The team members had previously worked together and agreed to cooperate on the thesis since the previous experience was great. When discussing the assignment with others, in particular lecturers and others associated with the Department of Computer Science, it was estimated the thesis work would be quite challenging for any student group. With Cecilia and Eimen having limited C++ knowledge at the start of the project, the team was aware that they would have to learn a lot throughout the development process.

1.6 Other Roles

Supervisors

The project had two supervisors during the development process. The initial supervisor, Xiang Su, oversaw the project for two months. Subsequently, Rafael Palomar assumed the role of project supervisor and supervised it until its completion in May, 2024. The team held occasional meetings With the first supervisor. With the second supervisor, weekly meetings were scheduled with a provided agenda. These meetings facilitated discussions on current progress and addressed any challenges encountered since the last meeting.

Qt Representative

The representative from The Qt Group was Matthias Rauter, who has been the primary contact person for the thesis. The Qt representative's role has involved introducing the team to the problem domain, suggesting improvements to the system, and providing feedback on the development of the thesis project.

1.7 Project Work Framework

During the project development, the team utilized Scrum as the working methodology. During development, the team visited the Qt offices often weekly to consult with them on current progress, hear their wishes for the project, and ask for guidance when needed.

1.7.1 Team Meetings

The team usually had team meetings once or twice daily. Exactly how they were organized, is outlined further in the project plan (see Appendix A) and chapter 6, and finally reflected upon in section 9.5.

1.7.2 Supervision Meetings

Throughout the duration of the project, the team has had weekly meetings with the supervisor. These meetings would often include the team giving a progress update, and also receiving feedback from the supervisor on how to academically improve the project. During the latter half of development, the focus of these meetings shifted gradually towards the report part of the project.

1.7.3 Product Owner Meetings

The product owner frequently participated in meetings with the team, especially during the first months of the project. Meetings were initially held weekly and would include the team giving progress updates and receiving feedback from the product owner to make sure the project was adhering to requirements and meeting the product owner's expectations. Meetings were often held physically in Nydalen, Oslo, at Qt's offices. Later in development, the team conducted short, intermittent discussions with the product owner via Teams to ensure that the project progressed as Qt expected.

On several occasions, the team consulted with employees at The Qt Group to gather input on new features or issues that arose during development. These consultations never included the employees writing code; instead, they provided high-level feedback on how to approach the problems in question.

1.8 Tools and Technologies

The team used multiple tools and technologies to facilitate the planning, execution, and management of the project. *Monday.com* and *Google Drive* have been used for project management and planning. *Discord* has been used for internal team communication. *Microsoft Teams* has been used for communication with supervisor and Product Owner. *Overleaf* was

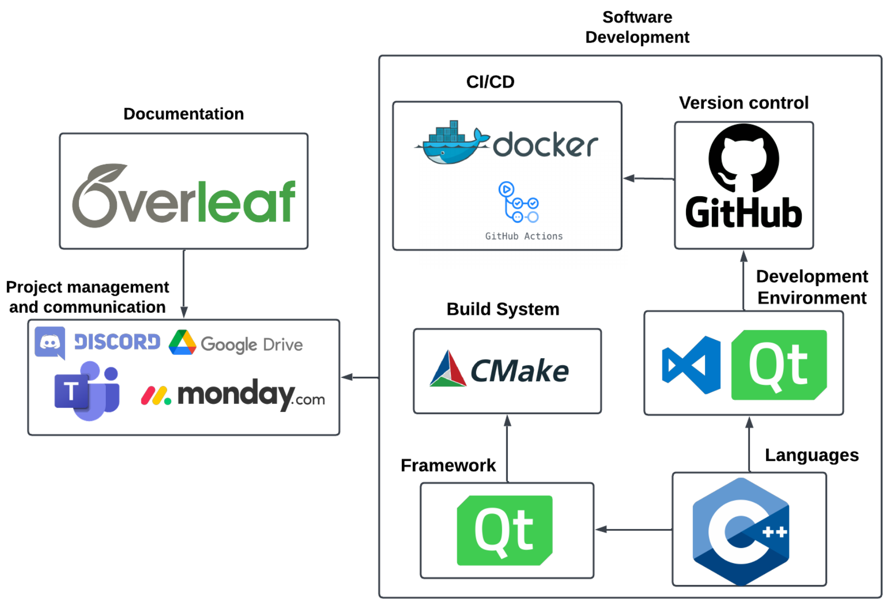


Figure 1.2: Tools used in the project

the primary tool used for writing the thesis. *GitHub* was used to host the source code of the developed system, as well as managing the Scrum issue board during development. For project execution and software development, a multitude of tools have been used: Visual Studio Code, Qt Creator, C++, CMake, the Qt framework, and Docker. A visual overview of the tools utilized is shown in Figure 1.2.

1.9 Thesis Structure

This report is divided into 9 chapters. **Chapter 1** gives a non-technical overview of the thesis project. This includes outlining the task, project goals, thesis target audience, how work was organized, and the structure of the thesis report document. **Chapter 2** provides details on technologies and related problem domains that the reader should be aware of to understand the implementation. **Chapter 3** contains requirements, expectations, and delimitations for the final product. **Chapter 4** outlines the technical and non-technical design aspects of the final system, including the rationale for design decisions. **Chapter 5** explains the software implementation in further detail, stating how application components function and integrate with each other. **Chapter 6** discusses how the development process has been structured. Efforts the team has made to improve the quality and testing of the final solution are covered by **chapter 7**. This includes topics such as unit testing, component testing, acceptance testing, and quality assurance measures. **Chapter 8** explains how to build and run the project, listing tools that are necessary for this to be successful. **Chapter 9** discusses the approaches mentioned in chapter 5, as well as alternative approaches. It also includes a discussion on why certain approaches were chosen over the alternatives.

The thesis refers to a repository containing the developed software source code described by the thesis. This repository is distributed alongside this thesis report and is available online, see Bratlie *et al.* [5]. The full list of references used in the thesis, followed by appendices, can be found at the end of the document.

Chapter 2

Background

This chapter establishes the prerequisite knowledge and terms required to understand the thesis and software implementation. Additionally, it describes challenges and conventions within the problem domain that are relevant to the system presented in the thesis.

2.1 Raster-Based versus Vector-Based Rendering

The process of outputting visual imagery to a display can involve a multitude of complex approaches. This section discusses two common approaches relevant to this thesis work, *raster imagery* and *vector graphics*. These terms describe how the source image data is encoded when stored. Both approaches present advantages and disadvantages, and this section will only discuss those that are the most relevant to the project. Further reading material can be found in GOMEZ GRAPHICS [6].

This section outlines how utilizing vector-based graphics may lead to improved visual fidelity and correctness of a displayed image. These benefits serve as a motivation for the product owner to investigate integrating support for vector-based maps into the Qt framework.

2.1.1 Raster Images

Raster-based imagery implies the source image is encoded as matrices of color values[1]. In the context of this project, each individual map tile (defined in subsection 2.2.2) is a static

[1]Single/multiple channels depending on the type of image

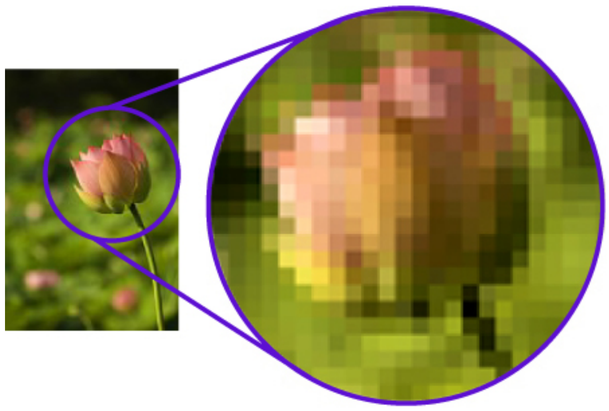


Figure 2.1: Raster-image illustration. Image taken from [6].

image encoded as a matrix of colors. The matrix has a fixed width and height. This type of encoding is often referred to as *pixel maps* [7]. In this thesis, the term *raster images* refers to this encoding. An illustration of how computer imagery can be broken down into a matrix of colors is shown in Figure 2.1.

2.1.2 Vector Graphics

The term vector graphics refers to the process of producing computer imagery based on geometrical shapes, such as lines, polygons, and curves. These shapes are defined in terms of the Cartesian coordinate system [8].

Modern displays are commonly raster-based. Consequently, vector graphics must be projected and rasterized onto the display in order to be displayed correctly. This thesis will only cover polygons, lines, and points for the source data, as that is what MapTiler provides (see section 2.2).

2.1.3 Benefits and Disadvantages

This section discusses two benefits that come with vector-based graphics, which are highly relevant to the motivation of this thesis. These benefits are specifically related to *scalability* and *layered rendering*.

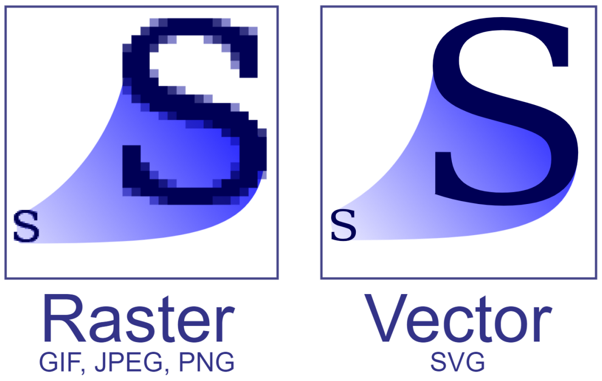


Figure 2.2: Magnifying vector and raster-graphics. Image taken retrieved from Yug, modifications by Cfaerber et al. [11]

Scalability

As a consequence of being geometrically defined, vector-based graphics can scale up and down to any size with no impact on the visual fidelity [9]. Scaling can be applied to the geometrical shapes before they are rasterized onto the display.

In contrast, raster images exhibit visual artifacts, which refer to unintended distortions, when magnified. This type of visual artifact is commonly known as *pixelation* [10] and can be recognized by the image visibly turning into a matrix of colored squares. This effect can be seen in Figure 2.2.

Layered Rendering

Vector graphics allow for grouping of data based on what the data represents. A core benefit of this structure is being able to transform shapes differently based on categorization or hide categories altogether. One relevant example is how terrain shapes can be distinguished from text and symbol shapes when using vector data. With this approach, a map-rendering implementation may rotate the world map without also rotating the orientation of text and symbols. Different scaling may be applied between text and terrain, allowing text and symbol sizes to remain constant relative to the display. This may be a desired property when rendering geographical maps, since text and symbols can stay the same while scaling other elements.

Raster-based imagery does not provide this functionality. Raster-based imagery inherently binds all elements together. A transformation applied to the map – like rotation or scaling – affects all elements of the image uniformly, including text and symbols. Consequently, text may become visually distorted or unreadable when the map is transformed (such as when rotated to be upside down). This property disallows hiding an element, as there is no additional raster data that stores the background data separately.

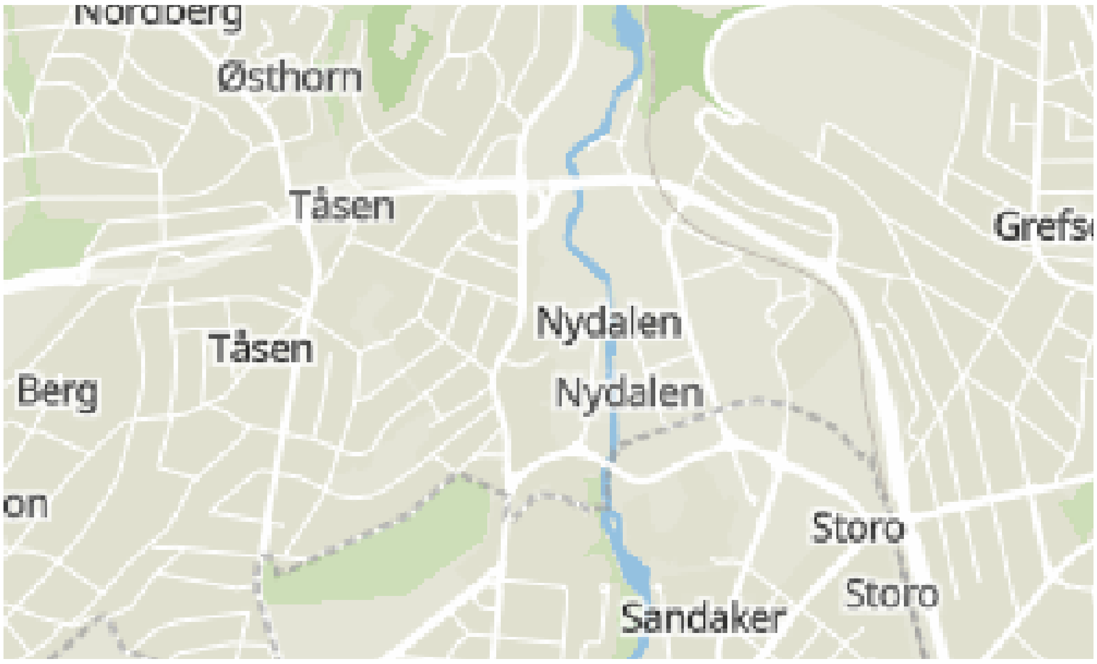
2.1.4 Other Considerations

There are additional factors to consider when implementing either approach. While vector-based graphics introduce improvements to visual fidelity and flexibility, their implementation is more complex and computationally intensive compared to using raster maps. A common approach is to use hardware-acceleration in order to generate vector imagery at a sufficiently high rate to be used in interactive applications.

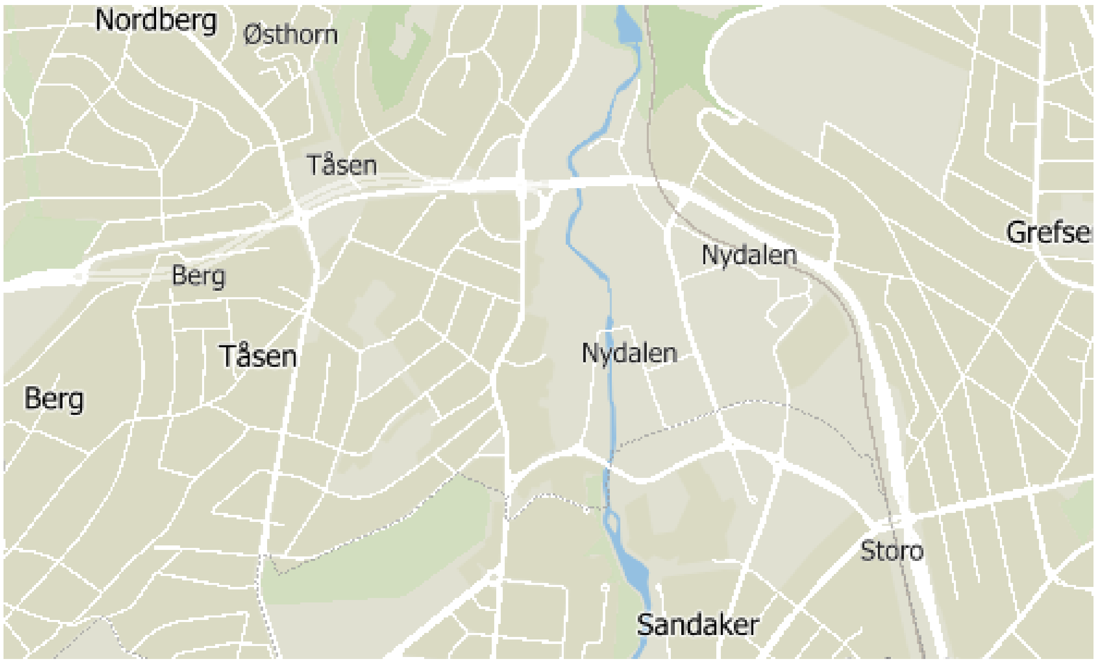
2.1.5 Demonstration

Screenshots of both approaches being utilized within the final software of this project are shown in Figure 2.3. This figure uses a non-discrete viewport zoom-level (see section 5.2 for additional viewport information), near the boundary where one of two discrete map detail levels is chosen. Consequently, each tile is magnified as much as possible. These images have been intentionally selected and magnified to showcase the image-quality problems of raster-based rendering in a worst-case scenario.

The limitation of text scaling is apparent in this figure. Figure 2.3b has the correct text-size displayed. Meanwhile, the text size is visibly magnified in Figure 2.3a. In the raster-based screenshot, the text also has perceptible pixelation.



(a) Raster



(b) Vector

Figure 2.3: Raster vs vector image quality Differences

2.2 MapTiler

At the time of writing, The Qt Group does not provide services that host geographical map data that can be displayed by the system presented in this thesis. For this reason, the map software needs to rely on a third-party service to load the geographical data. Qt has requested that the team design their software to fetch this information from the third-party MapTiler service [3] (this delimitation is described further in section 3.2). MapTiler supplies both geographical map data and specifications on how the map should be visualized. The remainder of section 2.2 gives an overview of how MapTiler supplies the geographical data that is necessary to render the application's map.

2.2.1 Web Mercator Projection

The world is a three-dimensional sphere, therefore it is important to use a projection formula to project each part of the world onto a two-dimensional plane that can be visualized. MapTiler utilizes a variant of the Mercator projection [12], which is the most common projection utilized in end-user applications.

MapTiler uses the Web Mercator projection. Web Mercator is a variant of the Mercator map projection with the adjustment that the world map is cropped to be perfectly square. Consequently, the northernmost and southernmost parts of the world are not included in the map. An example of the Web Mercator projection is found in Figure 2.4. The projection can be understood in this figure by inspecting how geometry is placed onto the two-dimensional plane. In particular, a common trait (and disadvantage) of Mercator is how shapes become increasingly bigger when moving towards the northern and southern edges of the map.

2.2.2 Spatial Partitioning

MapTiler splits a square two-dimensional world map into multiple smaller, quadratic pieces, which will be referred to as "tiles". The way the world map is divided forms a quad-tree [13]. This allows the world map to be divided up into different levels according to the level of detail the application wants to display. A visualization can be found in Figure 2.5. A rendering implementation can therefore select a level of detail that maps adequately to the zoom that is currently being used to display the map. In essence, a lower zoom will result in fewer tiles, where each tile has a lower amount of detail and covers a larger area

Figure 2.4: MapTiler Basic-V2 world map. Image supplied by MapTiler.

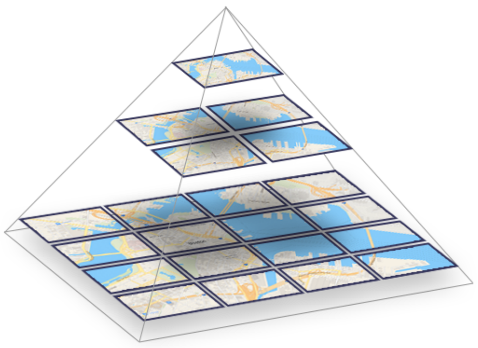


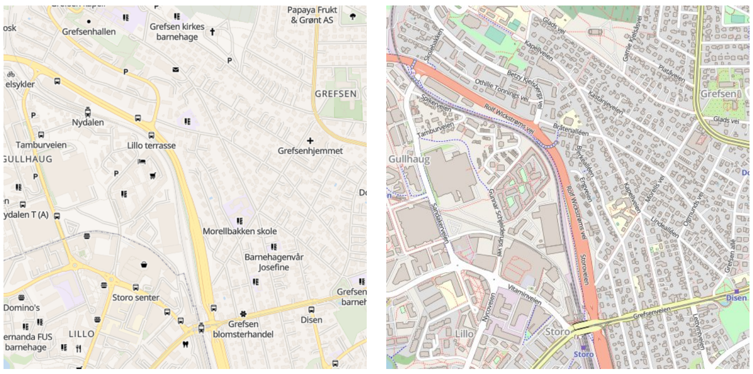
Figure 2.5: MapTiler spatial partitioning. Image pulled from MapTiler [14]

of the world map. A higher zoom will result in more tiles, where each tile will have a higher amount of details and covers a smaller portion of the map.

This has the advantage of allowing the user's device to only store the tiles that it needs. Additionally, a rendering implementation does not need to display details that would otherwise be imperceptible. A disadvantage of this is the added complexity that stems from stitching together multiple tiles to create the illusion of one seamless, coherent world map. If the tiles are not perfectly aligned, it can lead to visual artifacts like gaps in the final map.

2.2.3 Data Overview

This section describes some of the data exposed by MapTiler to allow third-party applications (such as the system presented in this thesis) to render geographical maps.



(a) Bright-V2 (b) OpenStreetMaps

Figure 2.6: MapTiler style examples

MapTiler provides different map "styles", where each unique style contains geographical data that will be visualized differently. An example of different styles can be found in Figure 2.6, which shows the difference in style between the Bright-V2 and the OpenStreetMaps styles. Note how the figure shows two images of the same area, with different styling and set of elements.

The *style sheet* determines how a map should be styled, while a *tile sheet* is what contains tile data. Multiple style sheets may reference the same tile sheet(s). Consequently, multiple styles may use the same set of vector tile data while being visualized differently. How the implemented application handles style sheets and their connection to tile sheets, is elaborated upon further in section 5.1.

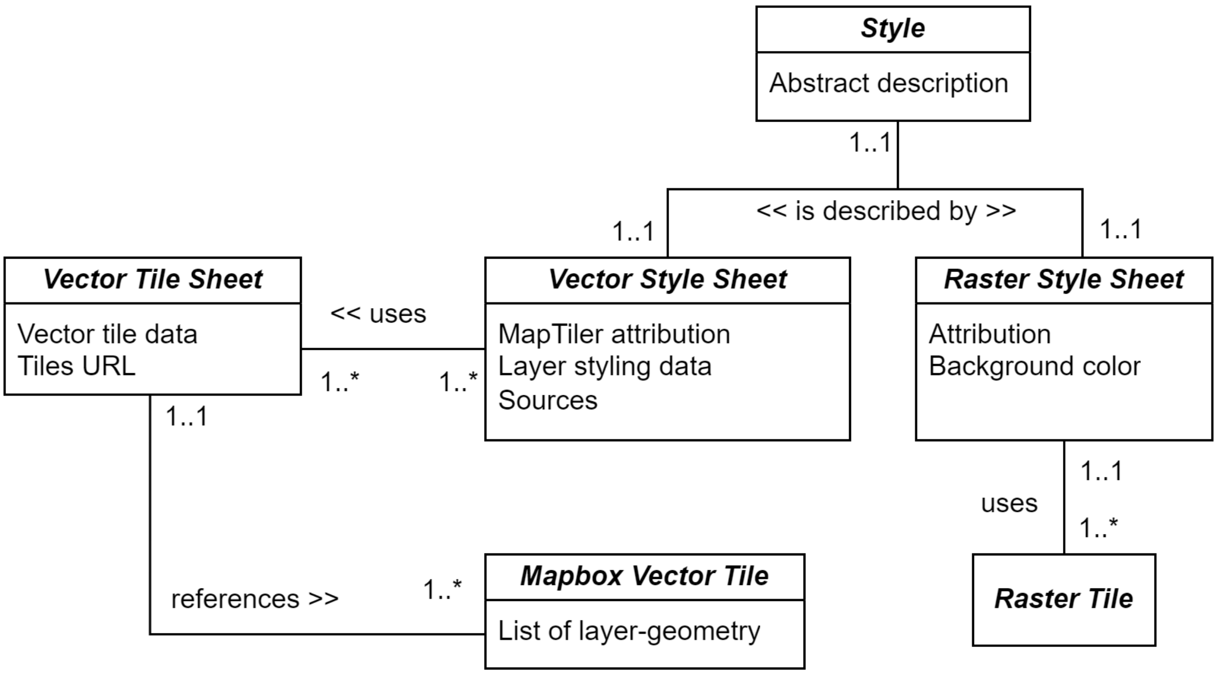


Figure 2.7: MapTiler data domain model

Figure 2.7 illustrates how MapTiler structures data. This representation is for illustration purposes only specific to the thesis. The entities in this figure can be described as the following:

- **Style** — A named description of the visual style of a geographical map representation along with the data associated.
- **Vector Style Sheet** — Describes the appearance of a style when rendered with vector graphics. This includes what elements are included (such as specific roads, text, and symbols) and also how they are styled (color, outline, size). The vector style sheet references exactly one tile sheet. The style sheet is exported as a JSON file and adheres to the MapTiler GL Style Specification [15].
- **Vector Tile Sheet** — References a collection of vector tiles and also contains relevant information to that collection. This will include – but is not limited to – attribution. This data allows third-party applications to credit the correct actor for gathering the data.
- **Mapbox Vector Tile** — Binary data stream that contains vector-based geographical data within a single tile. Data is encoded according to the MapBox Vector Tile specification [16].

- **Raster Style Sheet** — Describes minimal information related to displaying raster-tiles.
- **Raster Tile** — Raster image describing a single tile. Commonly encoded as a PNG or JPEG image file.

2.3 Mapbox Vector Tiles

The Mapbox Vector Tile specification [16] is the standard used by MapTiler for encoding the geographic data contained in a single tile. It does not contain information on how to style the data when visualized. A basic understanding of the standard, specifically how the data is structured and encoded, is necessary to understand the implementation of the application. It is critical that the software presented is able to decode this encoding scheme in order to successfully present the map to the user.

2.3.1 Protobuf

Vector Tiles Encoding Format

Vector tiles are a space-efficient format for storing geographic vector data. They are a specific application of *Google Protocol buffers*. "Protocol buffers are language-neutral, platform-neutral extensible mechanisms for serializing[2] structured data" [17].

Protocol Buffers Benefits

Protocol Buffers are a fast and efficient encoding format for structured data. They work by invoking a protocol buffer compiler to generate language-specific classes from a `.proto` file definition. Protocol buffers provide numerous advantages including efficient data storage, rapid parsing, support across multiple programming languages, and enhanced functionality through automatically generated classes. Many projects including gRPC, Google Cloud, Envoy Proxy, Twitter, and Apache Mesos use them. The benefits of protocol buffers include:

- **Cross-language Compatibility**: Since generating a class for a specific message is reliant on the definition residing in the `.proto` file, the protocol buffers compiler can generate classes for a lengthy list of programming languages including C++, C#, Java, PHP, Python, and Ruby. This makes protocol buffers language-neutral, meaning that two programs created with different programming languages can communicate using

[2]Serializing data means transforming it into a format that can be stored or transmitted more efficiently

protocol buffers (together with gRPC) given that the programs are written in one of the supported languages.

- **Extensibility**: Protocol buffers enable you to update the proto definition without the need to update the code, providing you follow the appropriate practices when creating message definitions in proto files. This means that not only do protocol buffers offer backward compatibility capabilities, but also forward compatibility. Old code will be able to read newer version messages ignoring newly added attributes, and new code can read old messages assigning default values to new fields.

Protocol Buffers Limitations

Protocol buffers might not be the best fit for some situations as they suffer from some limitations. They are best suited for small data chunks that can be internally loaded into memory and might cause problems for larger data volumes. Protocol buffers are also not well supported in non-object-oriented programming languages. They are also not self-describing, meaning that they can not be fully interpreted without their corresponding .proto files. Another disadvantage of protocol buffers is that they are not human-readable. Unlike JSON and XML, protocol buffers have a binary representation which makes them very fast to serialize and deserialize at the cost of readability.

Protocol Buffers Functionality

To use protocol buffers, a message schema must first be defined in a `.proto` file. Following the protocol buffers documentation examples [17], a “Person” message can be created. Listing 2.1 defines the "Person" message with three fields: name, id, and email. The message fields must specify if the field is optional or repeated, the data type of the fields, the name of the field, and a field number.

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

Listing 2.1: Person Protocol buffer Message Example.

After defining the message schema, the protocol buffer compiler should be invoked to generate corresponding classes in the chosen language. The generated classes can then be used to serialize and deserialize data.

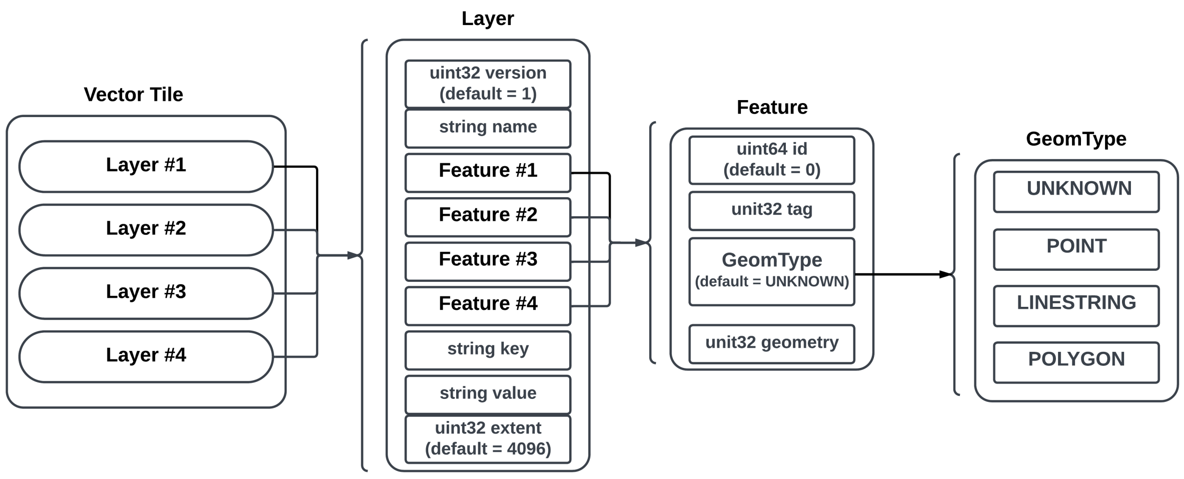


Figure 2.8: Vector tile structure diagram

2.3.2 Vector Tile Structure

The Mapbox Vector Tile specification [16] describes the components of vector tiles and their internal structure. It includes instructions on decoding vector tiles from Protobuf files into geometry that can be rendered on the screen. The vector tile Protobuf message is defined in Appendix F. The internal structure of the data inside a vector tile follows the proto message definition. A vector tile contains two major components, layers and features. An illustration of the vector tile structure is shown in Figure 2.8.

Tile Layers

Each tile must contain at least one layer. A layer encapsulates geometric features and their corresponding metadata. Layers usually contain features of the same type, either polygons, lines, or points. The features in layers are usually aggregated by type. For example, a polygon layer could contain all the features of buildings, or a line layer could contain all the features of roads. A layer is composed of the following components:

- **Version:** Defaults to 1. This is the version number of the layer and it should contain the major version number. This is used by decoders to determine if they can decode layers of this version. Decoders may skip layers with unknown versions.
- **Name:** This serves as the id of the layer. Each layer name must be unique within the parent tile.
- **Features:** A list of geometric features. Each layer must contain at least one feature.

- **Keys:** A 0-indexed list that stores the keys of the metadata of features.
- **Values:** A 0-indexed list that stores the values of the metadata of features.
- **Extent:** Defaults to 4096. A number that represents the extent of the coordinate system. For example an extent of 100 means that each coordinate unit within the tile refers to 1/100th of its square dimensions. In the case of an extent that is equal to 100, the point that has the coordinates (0, 0) would be on the top left of the tile, and a point that has the coordinates (100, 100) would be on the bottom right of the tile.

Layer Features

A feature represents a geometric entity within a layer. A feature can be of type polygon, line, or point. They encapsulate geometry information in addition to optional metadata that can be used for styling the features. An illustration of the structure of a layer is shown in Figure 2.9. Features are composed of the following components:

- **Type**: This property is an enumerator that specifies the type of the feature and it can take one of the following values: POLYGON, LINESTRING, POINT, UNKNOWN. This property is used to determine how to decode and render the features.
- **Geometry**: A list of 32-bit integers encoding the commands and coordinates used to render the feature.
- **Tags**: An optional list of integer pairs referring to the zero-indexed lists of keys and values inside the parent layer. These tags are used by the renderer in conjunction with the style sheet to style the feature.
- **ID**: An optional integer, unique for every feature within a layer, that can be used to identify the feature.

Geometry Encoding

Geometry data is encoded as a 32-bit integer array. Each integer in the list represents either a command-type or a parameter for a command. The geometry is defined in a screen coordinate system, such that the top-left corner is the origin of the coordinate system. The X-axis points positively to the right and the Y-axis points positively downwards. A command integer is an integer that encodes a command ID and a command count. A command ID describes an operation. The supported commands are:

- **MoveTo**: This command has the ID 1 and moves the painter to the specified coordinates. This command takes two parameters that represent the X and Y values of the point that the painter must move to.

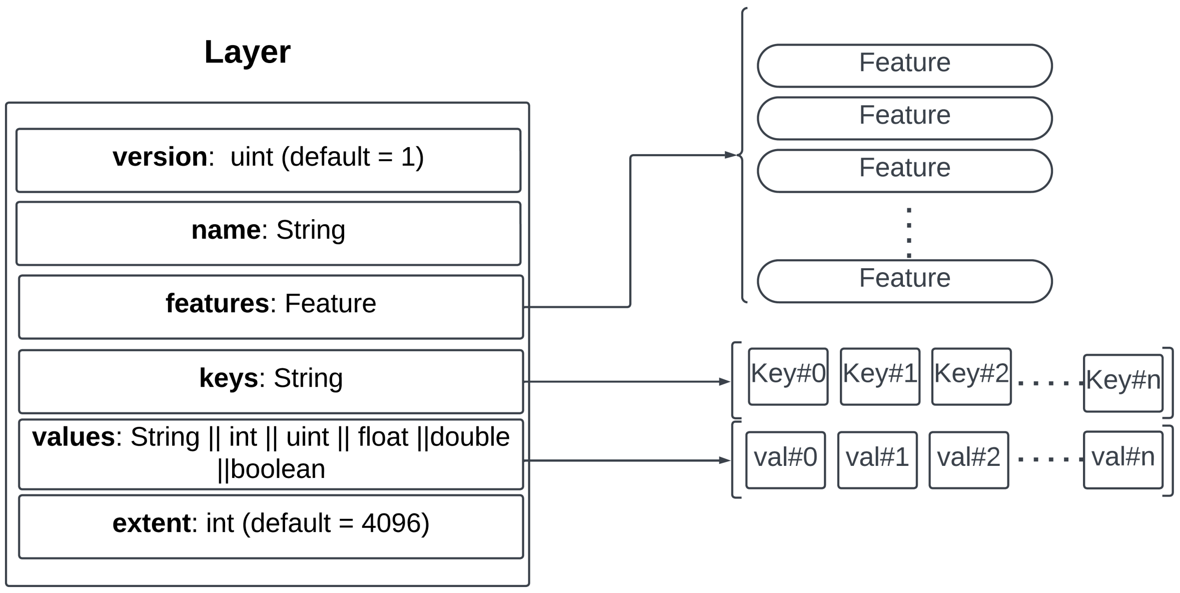


Figure 2.9: Layer structure diagram

```
int commandInteger = (commandID & 0x7) || (commandCount << 3);
int commandId = CommandInteger & 0x7;
int commandCount = commandInteger >> 3;
```

Listing 2.2: Relation between command integer, command ID, and command count

- **LineTo**: This command has the ID 2 and draws a line from the current position of the painter to the specified coordinates. This command takes two parameters which represent the x and y values of the end point of the line to be drawn. This command changes the painter's position to be at the end of the line that was just drawn.
- **ClosePath**: This command has the ID 7 and closes the current path. This means that a line will be drawn from the current painter position to the start position of the current path. This command does not change the position of the painter.

The command count specifies how many times the command should be executed. The command ID is encoded as the three least significant bits of the integer, and the command count is encoded as the remaining 29 bits. The relation between the command integer, the command ID, and the command count is shown in Listing 2.2.

Geometry coordinates are encoded using zigzag encoding [18]. In this encoding, positive

```
int value = (ParameterInteger >> 1) ^ -(ParameterInteger & 1);
```

Listing 2.3: Decoding parameter integer

and negative integers are both encoded as positive integers. Some command integers can take parameters. The number of parameters needed is equal to twice the commandCount. Decoding parameter integers can be done using the conversion shown in Listing 2.3.

Geometry Decoding Example

This example illustrates how to decode a tile of size 10x10 with a 2 cell buffer where the following array encodes a square:

```
[9, 4, 4, 26, 12, 0, 0, 12, 11, 0, 15]
```

The geometry array can be decoded and rendered with the following steps:

- **Step 1**: Decode the first command integer "9", which translates to command ID = 1 and command count = 1, this corresponds to MoveTo being executed once. The next two parameter integers are "4" and "4" since the command count is equal to 1. Then, the parameter integers are decoded, giving the coordinate pair (2,2). This pair represents the coordinates for where to execute the operation. When executing the command, the painter moves to the point (2,2).
- **Step 2**: Decode the next command integer "26", which translates to command ID = 2 and command count = 3. This corresponds to performing LineTo 3 times. The next 6 parameter integers are "12", "0", "0", "12", "11", and "0" since the previous command count was equal to 3. The parameter integers are decoded to get the following coordinate pairs: (0,6), (6,0), and (-6, 0). Then, the commands are run to get the three sides of the square. Each coordinate pair is relative to the current pen position one and *not* to the origin of the tile.
- **Step 3**: Decode the last command integer "15", which translates to command ID = 7 and command count = 1. This corresponds to ClosePath 1 time. ClosePath does not require any parameters and closes the path.

Following these steps will lead to the results presented in Figure 2.10.

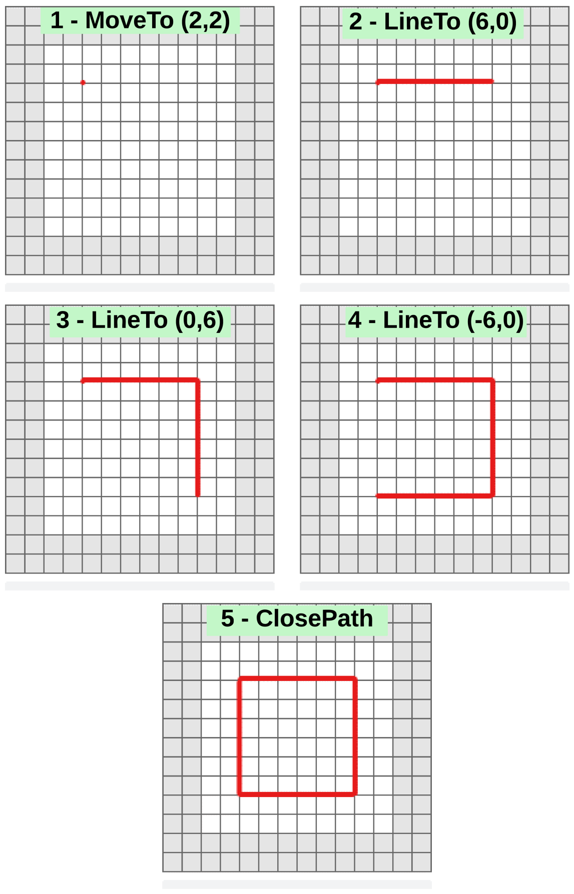


Figure 2.10: Rendering a square using vector graphics

Feature Attributes

Feature attributes are referred to as metadata throughout the project. The features' metadata provides additional information about features that can be used for dynamic styling. The chosen tileset for our project uses the MapTiler Planet schema for metadata [19]. The metadata is encoded as a tags list of integer pairs referencing the zero-based indices of the keys and values list of the parent tile. The number of elements in the tags list must be even. An example of how the metadata is encoded can be found in Appendix G.

Chapter 3

Requirements Specifications

3.1 Product Owner Requirements

The project requirements were formulated by Qt. These requirements were adjusted throughout the development process after discussions with the team.

3.1.1 Functional Requirements

The Functional Requirements (FRs) specified by Qt are as follows:

— **FR 1:** The system shall render a map to the display, with the source data being vector-based.

— **FR 2:** The system shall be able to render any section of the world map.

— **FR 3:** MapBox Vector Tile Format [16] shall be the format of choice for the vector source data to be used by the system.

— **FR 4:** The map rendered by the system shall include roads, place names, and basic terrains.

— **FR 5:** The system shall support as least the BasicV2 [4] vector map style from MapTiler.

— **FR 6:** The system must render the map in real-time. Real-time rendering involves producing the graphics "just in time". The imagery is generated during software runtime, not rendered ahead of time.

The product owner did not expect the final program to be a production-ready application. The development team was offered a high degree of freedom to develop and explore different approaches given that the final system satisfies the requirements listed above.

3.1.2 Non-Functional Requirements

The Non-Functional Requirements (NFRs) for the system are as follows:

- **NFR 1:** The system shall be suitable for Qt to use as a reference for their continued map software development.
- **NFR 2:** The Qt framework and tools shall be actively used in the software.
- **NFR 3:** The system shall use software-based rendering with QPainter [20].
- **NFR 4:** The system solution must be well documented, including how to parse relevant information and the process to display it correctly. The thesis report serves as this documentation.
- **NFR 5:** The software will be licensed under the MIT license [21].

3.2 Problem Delimitation

This project explicitly focuses on map functionality within the context of the Qt ecosystem, while using an external API called MapTiler to download the geographic data from the web. There are several topics that won't be covered, this list is non-exhaustive and covers only the most relevant topics:

- This thesis will not discuss alternative solutions or technologies for displaying maps.
- This thesis will not be discussing the implementation of map-functionality in the context of GUI-toolkits beyond Qt.
- This thesis will not discuss the implementation of map data provided by services other than MapTiler.
- The thesis will only focus on displaying maps using the Web Mercator projection [12] (see subsection 2.2.1). Other types of map projections are not discussed.
- This thesis will only prioritize displaying simple map elements that are useful for navigating urban areas. Map elements that are useful for research purposes, or elements that help navigate the wilderness, will not be discussed.
- This thesis will only discuss rendering maps as a flat two-dimensional plane. Three-dimensional graphics will not be discussed.
- Rotation of the map will not be discussed. The map will always be displayed with north facing the top of the display.

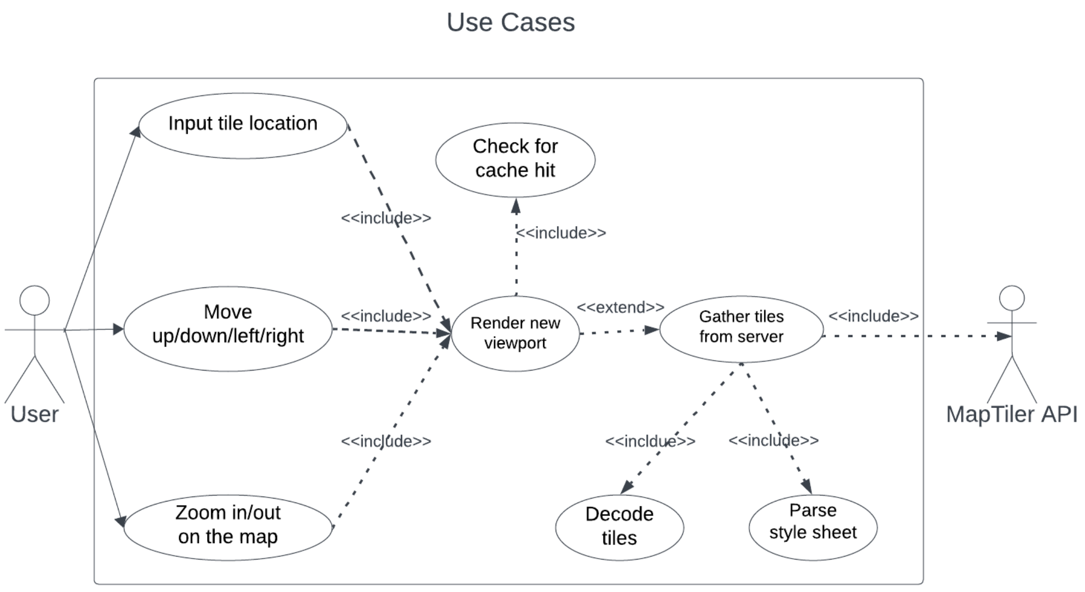


Figure 3.1: Use case diagram

- The solution will only be developed for Windows and Linux desktop platforms[1]. Web and mobile platforms are not discussed.

3.3 Use Cases

As mentioned in section 3.1, this project is not meant to be a full-fledged application. With that being said this project is meant to be a reference, it is a starting point for a production-level application that displays maps using vector graphics and the Mapbox standard [16]. Figure 3.1 is an example of how a final application might be used:

3.3.1 Use Case Descriptions

Case	**Description**
Use Case 1	Input tile location or coordinate
Actor	User

[1]Due to the cross-platform nature of the Qt framework, the team expects it to be possible to run the software on mobile without significant effort.

Goal	Input the coordinates to move to
Description	The user needs to input the longitude and latitude coordinates and a zoom level to which the map widget will snap.

Use Case 2	Move around the map
Actor	User
Goal	Move the center of the map to one of the four directions
Description	Mover the map up, down, right, or left. This triggers a new rendering pipeline for the new tiles that should be rendered on the viewport. This might also trigger new tile requests if the tiles are not in the cache.

Use Case 3	Zoom in/out
Actor	User
Goal	Increase or decrease zoom level
Description	Zoom in or out which might trigger new tile requests if the new viewport zoom level warrants a change in the map zoom level.

Chapter 4

Design

4.1 Fundamental Design Choices

The overall design decisions for the project were made to satisfy the requirements listed in chapter 3. As the development team was offered a high degree of freedom in choosing the tools, technologies, and architecture for the system, a list was created in the early stages of the project development to delineate these components. The following list describes some of the choices that were made:

- The solution must be coded in the C++17 standard.
- The Qt framework and Qt tools must be used actively to develop the software.
- The source map data must be supplied by the MapTiler service [3].
- The final map must include basic elements that can represent urban areas. This includes the following:
  - Roads, including road names
  - Rivers, water
  - Building outlines
  - Basic fill-colored terrain
  - Place-names, landmarks
- The solution must include interactive controls, including panning and zooming.
- The solution must include testing such as unit testing, automated testing, and integration testing.

A data flow diagram of the system can be found in Figure 4.1.

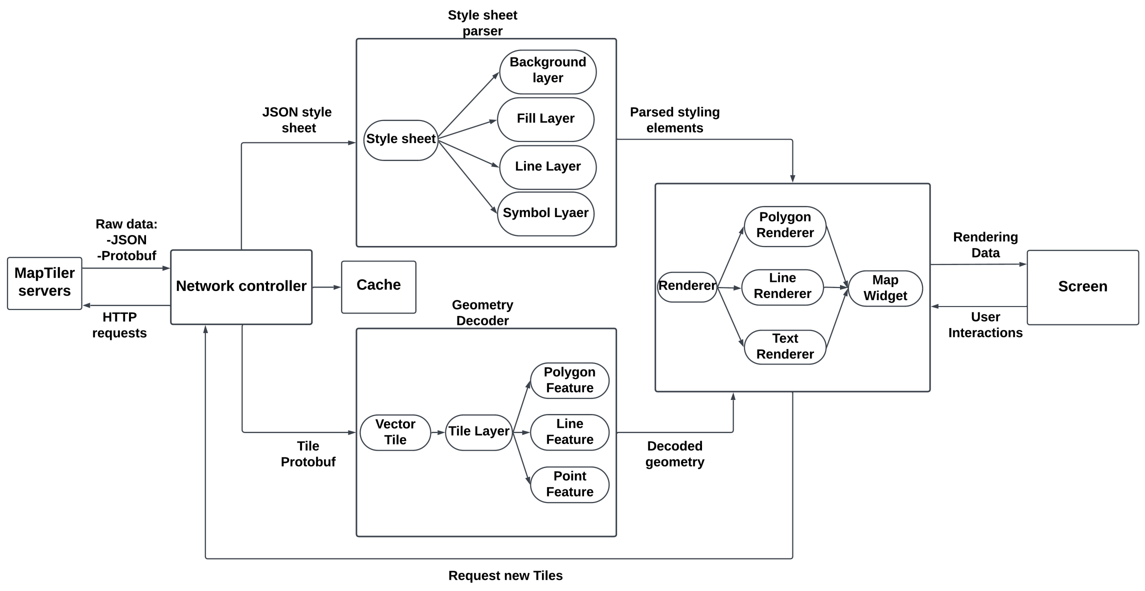


Figure 4.1: Data flow diagram

4.2 Architectural Design

The architecture of the software was designed to be modular and scalable, with an emphasis on testability. The architecture facilitates extensibility by decoupling components and defining clear boundaries between subsystems, while also promoting code reusability. The implemented architecture is a take on the *server-client pattern* [22, pp. 180–182], where the TileLoader acts as the server and the MapWidget acts as the client. An overview of the system is illustrated in Figure 4.2. A detailed description of each component can be found in section 4.3.

Although the architecture is simple, the internal implementation of each component is quite complex.

4.2.1 Testability as a Core Design Principle

When designing the architecture of the project, testability was a consideration from the beginning. The project emphasized modular testing, allowing detailed scrutiny of each component's functionality in isolation.

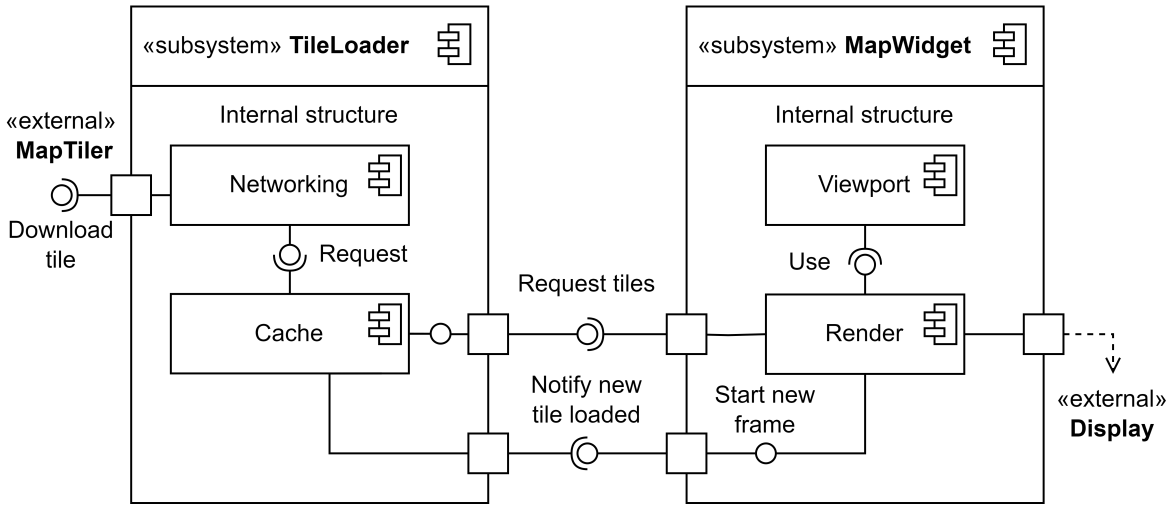


Figure 4.2: Architectural component diagram

Each component was designed to be testable. Components had to be highly configurable while limiting their reliance on other components. As an example, the TileLoader component was designed to function with and without internet connectivity. The TileLoader can rely solely on a local disk cache to load tiles. It can accept different cache directory paths on both Unix-based systems and Windows at runtime to locate this cache and can be configured to selectively load vector tiles only. This flexibility is essential when testing specific functionalities in both online and offline environments.

Running tests on individual components is not possible if all components are tightly built into the main application package. It is necessary to separate testable code away from a single executable package in order to test each component in isolation. Therefore, the architecture separates functionality into separate packages. All common functionality is defined as one library-type package "`maplib`". Tests and the application can then be defined as executable-packages that are linked to maplib. This allows for testing of individual components correctly. Packages are implemented using CMake targets in the implementation. An overview of these packages can be seen in Figure 4.3.

The primary application normally requires a windowing system present on the operating system in order to run, which is not available inside headless test environments. Splitting the packages as outlined makes it possible to run the tests in a headless environment, such as during Continuous Integration (CI) (see section 7.4).

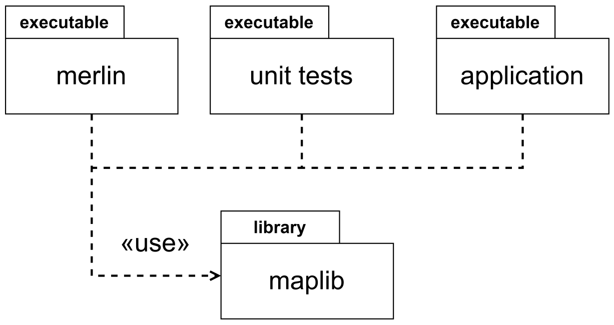


Figure 4.3: Package diagram

4.3 System Components

This section describes the software solution at a conceptual level. It describes the design of system software components, and how the components were designed to interact.

4.3.1 TileLoader Component

The TileLoader is responsible for loading individual tiles from the MapTiler web service and caching them locally. The TileLoader was designed to:

- Accept a set of tile-coordinates to prepare for displaying.
- Store ready tiles in memory, in a form that can be read and displayed.
- Remove tiles from memory when they are no longer actively being used.
- Process requests asynchronously and in parallel in relation to user interface operations.
- Cache tiles locally on disk.
- Notify user interface components whenever a new, relevant tile is ready for display.

The TileLoader is designed to process tiles in an *asynchronous and parallelized* manner in relation to the GUI logic. As shown in section 7.2, processing tiles is too time-consuming to leverage a smooth user experience. Consequently, it is important to run the tile-processing workload in the background in a way that does not interfere with the responsiveness of the GUI. A downside to this approach is that the rendering will output frames where not all tiles are present, as they are still loading. Instead, the display will be refreshed with new tiles as they are loaded.

Cache

Due to the high latency of downloading tiles from the web[1], the TileLoader stores a cache of the most recently used tiles. The purpose of this cache is two-fold; to reduce the amount of traffic to the MapTiler service, and to decrease loading times by loading the tile directly from local files. MapTiler restricts how many web requests a given MapTiler account can perform in a short time-span. Because the application is interactive and requests tiles when they become visible, it may request anywhere from dozens to hundreds of tiles to MapTiler with each use.

The caching mechanism contains two levels. Tiles will be cached on disk. A subset of these tiles will also be cached in memory while the application runs. Keeping recently used tiles in memory is convenient for rendering the tiles repeatedly and also allows the tiles to instantly appear if they were previously used.

It was initially planned to evict tiles from memory over time to minimize memory usage, but the eviction policy was not implemented due to time constraints and a lack of prioritization. This is discussed further in section 9.3 and section 9.4.

Request Process

The most important function exposed by the TileLoader is the `Bach::TileLoader::requestTiles` method. This is the function that will take a list of tile-coordinates, and immediately return the tiles that are ready while queuing up missing tiles for background processing. A visual overview of this procedure is shown in Figure 4.4.

4.3.2 Rendering Component

Rendering is a critical component of the project and is responsible for drawing map data correctly to the display. This includes rendering polygons, lines, and text with styling that is similar to that found in the raster-based maps. This component has the following requirements:

[1]Compared to loading from disk.

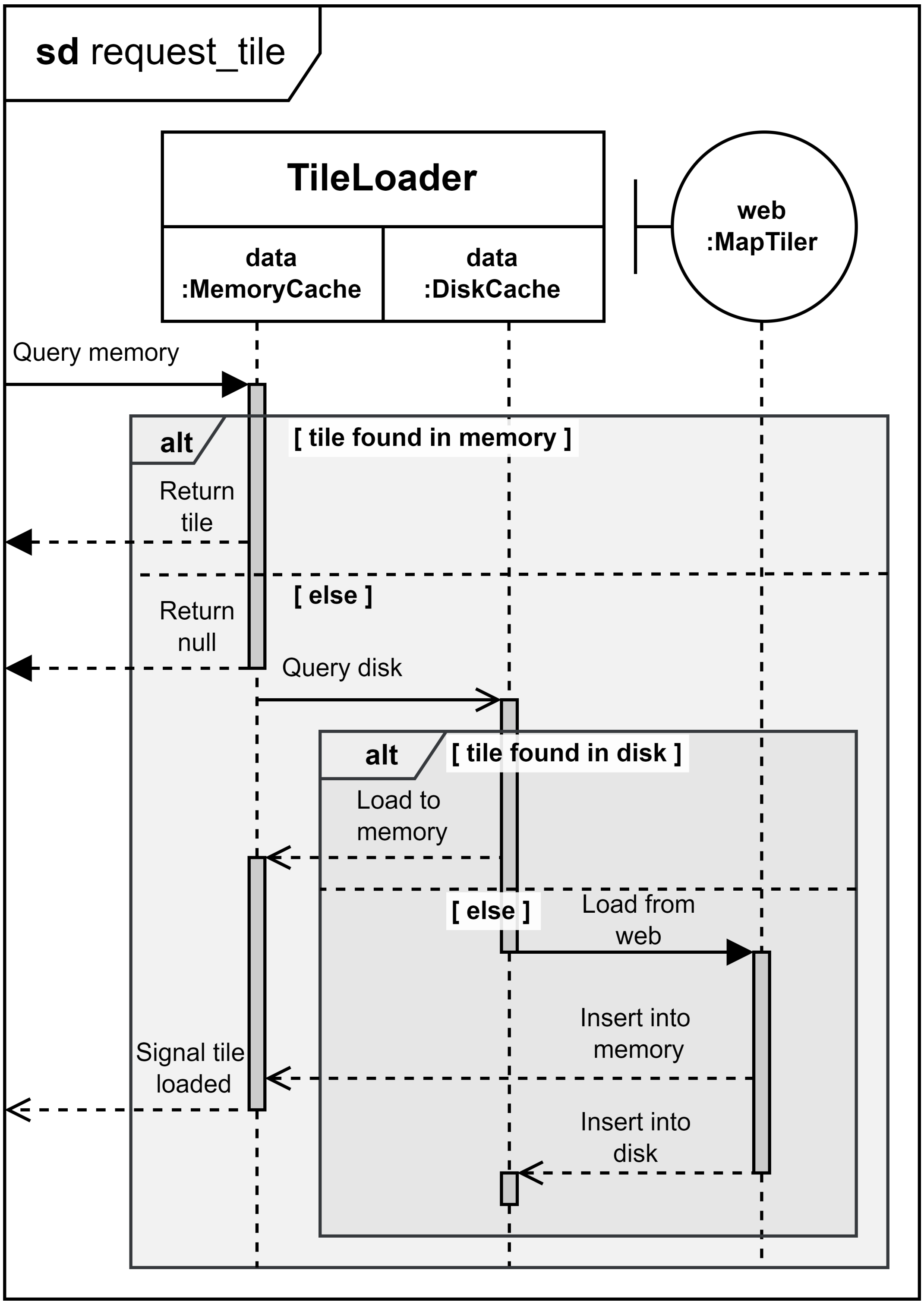


Figure 4.4: Request tile sequence diagram

- Request tiles from the TileLoader.
- Display tiles inside the viewport.
- Display multiple tiles on-screen as one seamless map.
- Render without a complete set of tiles.
- Use the QPainter class [20] for drawing primitives such as polygons, lines, and text.
- Rendered tiles must include elements such as terrain, roads, and place names.
- Rendered tiles must be styled according to the MapTiler style sheet specifications. Such as color, line width, text size, text color, and text outlines.
- Support usage of unit tests.
- Filter out specific displayed elements based on configuration, such as hiding all roads.

The rendering component needs to be able to render without a complete set of tiles to fill the viewport. This is because the TileLoader does not load tiles immediately, and instead streams them into memory over time. The rendering component will render nothing in place of missing tiles, and instead render new frames as new tiles are loaded.

4.3.3 Networking Component

The networking component has been designed for network interactions to get different vector tile types semi-automatically from the web. This meant hard-coding the initial required steps of the network interactions, but automating subsequent network calls where possible. One goal for the network component design was to make it easy to add, update, or swap between style sheet, source, and tile sheet types. In addition, errors must be handled gracefully, and software problems are logged to the developer while an appropriate error message may be displayed to the end user. The network error system is complex enough to handle critical errors, but simple enough to make it easy for the product owner to re-use, modify, or replace when they take over the project, making the software more sustainable. It would be straightforward to hardcode the style sheet and tile sheets for the Basic-V2 map style. This would be simple to implement since one could write a few lines of code, hardcoding the desired requests to hardcoded URLs. It would also be fairly cheap, since that implementation is fast, and hence would be cheaper than a more involved implementation.

A downside of this approach is that it could make future development for Qt more tedious since the entire system would need a rework or re-implementation to change between map types. The software should be flexible enough to handle that situation out of the box. Implementing support for grabbing tile data semi-automatically also means that is easy to swap between and check different map types. If one rebuilds the project and calls for a different style sheet type or source type (given that they exist in the MapTiler API), the networking components can grab all relevant data and handle potential errors correctly. If a future developer wants to call data that's stored in a different format, the interactions would of course have to be re-implemented. But for as long as map data has the same structure as MapTiler's JSON-encoded data, the software will be able to grab that data correctly and handle errors.

4.4 Graphical User Interface Design

The product owner did not provide the team with requirements for the graphical interface. Qt offered to design the interface and provide all icons and other graphical elements, but the team declined since it wouldn't be allowed to be delivered as a part of the thesis.

At the beginning of the project, a document was made detailing some requirements the team had for the graphical interface. The design was inspired by maps on MapTiler's website and Google Maps.

One goal of the graphical interface design was to make interactions available via clickable buttons. In addition, it should support map panning with the mouse buttons and map zooming with the mouse wheel. This would offer users multiple ways of interacting with the application. Furthermore, the interface should use a color scheme that matches Qt's color scheme to stylistically fit in with other Qt applications. Qt often uses green, grey, black, and white in designs, as well as having rectangular buttons with rounded corners.

4.4.1 Initial GUI Design Plans

The user interface had to support zooming and panning and correspondingly signal to the rendering component when it needed to redraw the map. The original user interface enabled these operations by using finger gestures, cursor actions, and on-screen controls. The original GUI was mocked up as shown in Figure 4.5.

4.4.2 Final GUI

Due to time constraints, the final version of the GUI design is different from what the team envisioned during the planning phase. However, the final GUI supports all the planned and required functionality. An illustration of the final application, with its GUI, is shown in Figure 4.6.

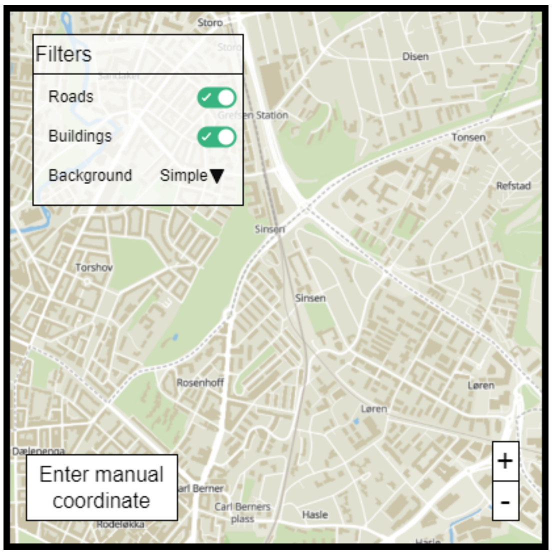


Figure 4.5: Initial GUI mockup

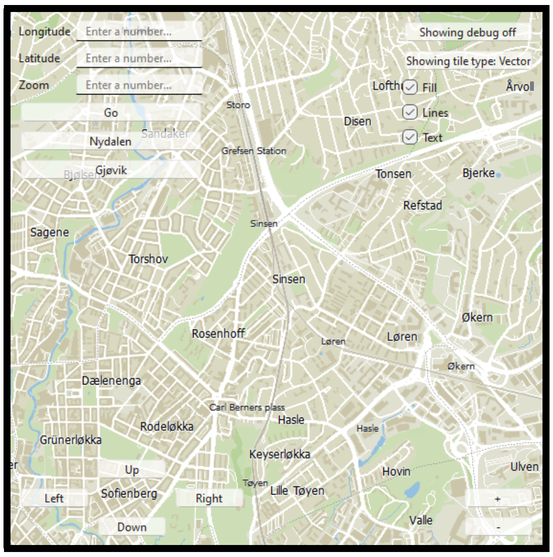


Figure 4.6: Final GUI design

Chapter 5

Implementation and Production

This section describes how the software was implemented along with descriptions of challenges the team faced in doing so. For the sake of brevity, this section is intentionally non-exhaustive and only particularly interesting implementation parts have been documented here.

5.1 Network Implementation

As previously outlined, the application relies on downloading map data from the external MapTiler API. Even though the final version of the network implementation is relatively simple, it performs a critical role in the system. The network component must download tiles automatically from the web at runtime unless they've been previously cached.

When downloading tiles from MapTiler, multiple requests have to be made in a row to get data. If the requests return data successfully, the data must be parsed, and then additional requests must be made based on the parsed results. The flowchart in Figure 5.1 models these steps [1].

First, the map style sheet type must be selected and then requested from MapTiler. This style sheet is encoded in JSON. If the first request is successful and returns valid data, one of the style sheet fields should be called "sources" (as long as the API follows the current structure), see Listing 5.1. This is where a *source type* must be selected, where multiple vector and raster source types may be available. Each source type has an associated link

[1]Flowcharts usually flow down to the right. This figure flows down to the right, goes up to the right, and flows down again to fit on the page.

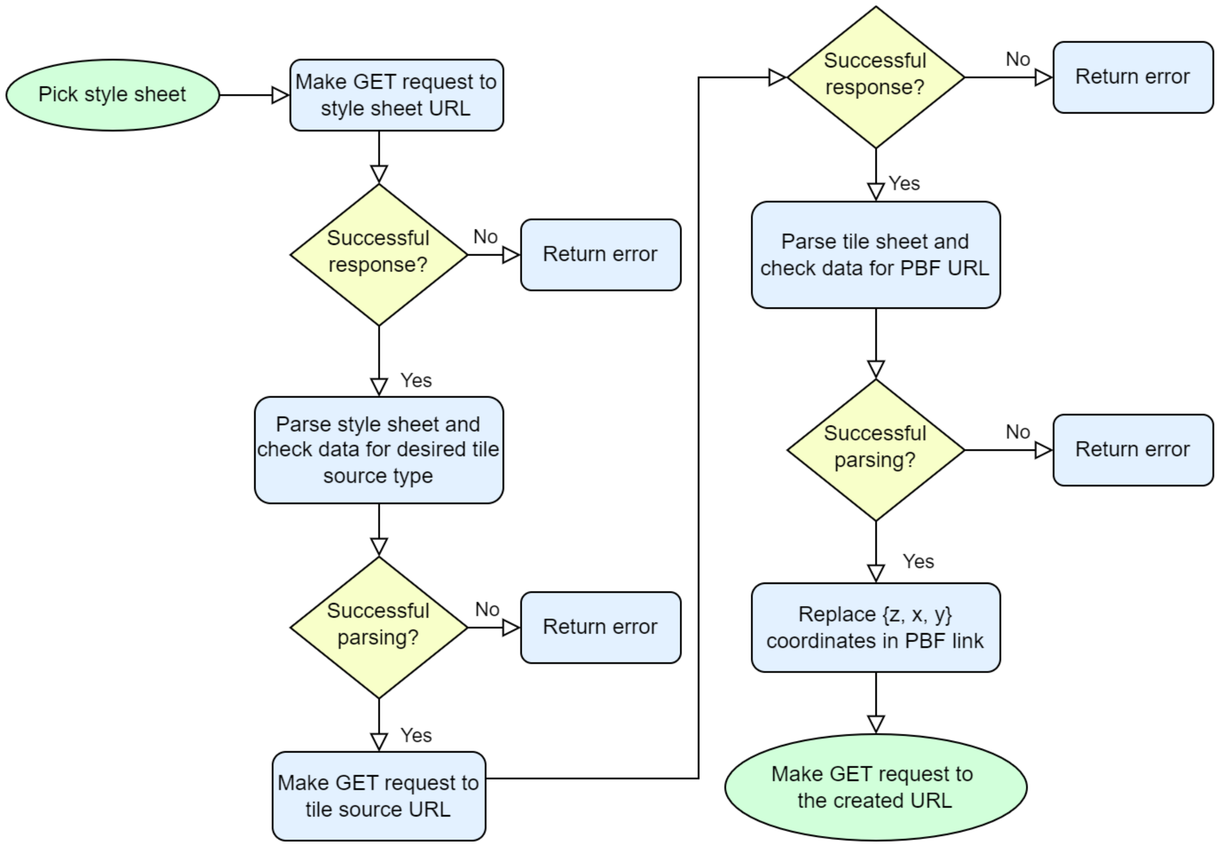

Pick style sheet
Make GET request to style sheet URL
Successful response?
No
Return error
Yes
Parse style sheet and check data for desired tile source type
Successful parsing?
No
Return error
Yes
Make GET request to tile source URL
Successful response?
No
Return error
Yes
Parse tile sheet and check data for PBF URL
Successful parsing?
No
Return error
Yes
Replace {z, x, y} coordinates in PBF link
Make GET request to the created URL

Figure 5.1: Flowchart modeling the networking component

```
{
    "version": 8,
    "id": "basic-v2",
    "name": "Basic",
    "sources": {
        "maptiler_planet": {
            "url":
             ↪  "https://api.maptiler.com/tiles/v3/tiles.json?key=myKey",
            "type": "vector"
        }, ... /* SNIP */
```

Listing 5.1: MapTiler style sheet example

```
    "tiles": [
        "https://api.maptiler.com/tiles/v3/{z}/{x}/{y}.pbf?key=myKey"
    ]
```

Listing 5.2: MapTiler tile sheet example

under the “url” field, which is where the corresponding *tile sheet* is hosted. For this project, the “maptiler_planet” source type is used.

To get the corresponding tile sheet, a `GET` request must be made to the URL of “maptiler_planet”. If the request is successful, the tile sheet is downloaded. This contains a final link at the bottom in the “tiles” field (Listing 5.2). The vector tiles are encoded as Protocol-buffer Binary Format (PBF) files on MapTiler, which is why that final URL is referred to as `PBF link` or `PBF URL` in the thesis.

To finally download a vector tile from MapTiler, the application replaces z, x, and y with zoom level, longitude, and latitude in the PBF link and makes `GET` requests to download vector tiles.

Which PBF link one gets in the end, depends on the selected style sheet and source type. Sometimes, *different style sheets* have the *same source value* but ultimately have *different vector tile sheets*. Additionally, there is no guarantee that tiles will always be hosted at the exact same location on the web. The network component has been implemented to be able to get vector tiles automatically for as long as the expected JSON fields (“sources”, “maptiler_planet”, and “url”) can be parsed.

The functionalities that interact with the MapTiler API primarily handle HTTP requests with (potential) errors and results. The networking component also parses HTTP responses. The `ResultType` enum class was implemented to support this and contains HTTP and parsing successes or errors.

The `HttpResponse` and `ParsedLink` structs both utilize the `ResultType`. The `HttpResponse` stores network response data as a byte array (this will contain map data), and the success or error as a `ResultType`. The `ParsedLink` stores a parsed URL string, and a `ResultType`. The implementation is simple but stores data together with a (potential) error. This could help the developer determine if there are problems with either the downloaded data itself, parsing as a part of the networking, issues with the MapTiler API, and bugs in other components. This feature has helped debug code throughout development, as there have been times where MapTiler responded successfully, but data would be parsed incorrectly by other components further downstream.

5.2 The Viewport

In order to position and scale tiles correctly on-screen, the team designed an abstract concept to mathematically model how smaller portions of the world-map are projected onto the display[2]. This concept was named the *"viewport"* and can be thought of as the camera for the map. The viewport is defined as an axis-aligned[3] rectangle that is projected onto the world map based on the displays "view" configuration. This configuration includes center coordinates relative to the world-map and a *zoom-level* property. The viewport forms a relation between a segment on the world map and what is output to the display.

The viewport shares the aspect-ratio of the display. Because the display can be rectangular (such as a smartphone), the viewport can also be rectangular. By allowing the viewport to share the same aspect-ratio as that of the display, we can avoid graphical stretching of shapes when projecting from the viewport to the display. A demonstration of the viewport can be seen in Figure 5.2.

The viewport includes a scalar *zoom-level* property that determines the size of the area covered on the world map. This property defines the length of the viewport's as 2^{-Z}, which is then interpreted as a factor of the world map. Consequently, a higher zoom-level will

[2]While the team designed this concept for this project, the team expects that other map-applications include a very similar concept.

[3]Aligned relative to the display

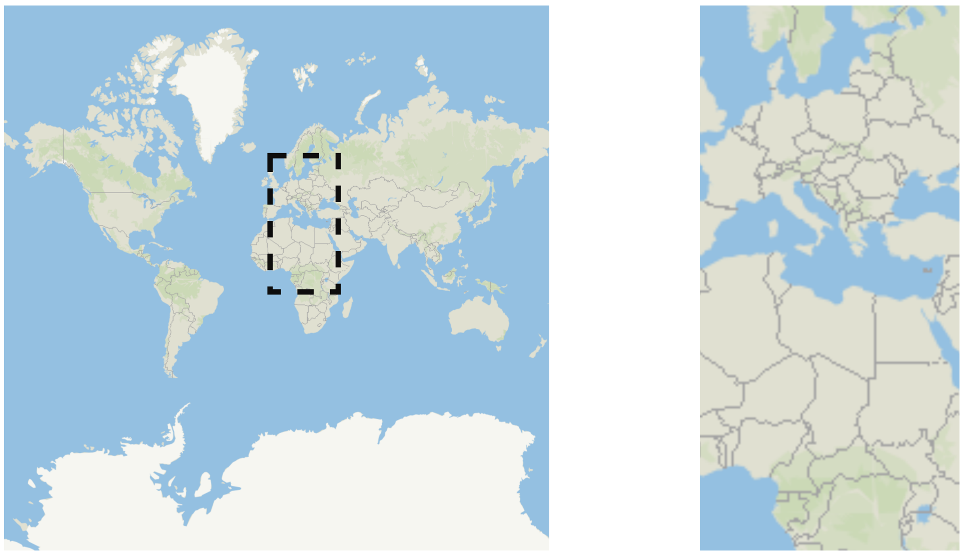


Figure 5.2: Viewport example. Viewport visualized as a black dashed rectangular line-segment. (a) shows the viewport in the context of the world map. (b) shows how the map would be displayed on-screen. Figure is for demonstration purposes only, and does not reflect the output of the application.

magnify the world map on the display. A viewport zoom-level of 0 will make the viewport be equal size to the world map, while each integer step will make the map four times as large[4] in relation to the viewport. The calculated length is then mapped to the longest side of the viewport's rectangle, while the shorter side takes the aspect-ratio of the viewport into account.

There is no relation described by MapTiler between the viewport's zoom-level and the level of detail of the world map. It is highly beneficial to establish such a relation in order to display the optimal amount of detail on the world map, depending on the viewport's zoom-level. The relation that has been developed for the system presented is outlined in Appendix E.

5.3 Style Sheet Parsing

The styles sheet is a JSON document containing essential information used for styling tiles' layers [15]. The styling information is encoded into JSON objects inside the `Layers` list in

[4]Since the relation is applied in both X and Y direction

the style sheet document. Layers generally contain two sub-properties that influence how the layer is rendered, a layout property and a paint property. The layout property includes properties such as line cap style, text font, text letter spacing, and symbol spacing, while the paint property includes properties such as background color, fill color, fill opacity, and line width. The style sheet divides the layer types into nine layer style types, of which the application utilizes the following four:

- **Background**: This layer style describes the styling properties of the map background. This layer does not correspond to any map features, it represents the default layer where there is nothing to render. The properties that are used from this layer type are:
  - **Background-color**: The color to be used for the background.
  - **Background opacity**: The opacity of the background.
- **Fill layer style type**: This layer style is exclusive to features of type polygon. The properties used from this layer type are:
  - **fill-antialias**: Determines whether anti-aliasing is used to render this layer.
  - **fill-color**: The color used to fill the polygons.
  - **fill-opacity**: The opacity of the fill layer.
- **Line layer style type**: this layer style corresponds to features of the type line. The properties used from this layer type are:
  - **line-cap**: Determine the line's cap style. The cap style can be either butt, round, or square[5].
  - **line-color**: The color used for the pen to draw the line.
  - **line-dasharray**: An array for the length of the alternating dashes and gaps that form the dash pattern.
  - **line-join**: Determine the style used when joining lines. The join style can be bevel, round, or miter[6].
  - **line-opacity**: The opacity of the line.
  - **line-width**: The width of the line.
- **Symbol layer style type**: This layer style corresponds to features of type point. However, this also corresponds to road names which are features of type `line`. The prop-

[5]Details on line cap styles supported by the Qt framework can be found in `https://doc.qt.io/qt-6/qpen.html#cap-style`

[6]Details on line join styles can be found in `https://doc.qt.io/qt-6/qpen.html#join-style`

erties used from this layer type are:
  - **text-color**: The color of the text.
  - **text-field**: The text to be rendered.
  - **text-font**: The for stach to use for text rendering.
  - **text-halo-color**: The color of the text outline.
  - **text-halo-width**: The width the the text outline.
  - **text-letter-spacing**: The space between individual letters. This is used only for curved text as the letter spacing is automatically determined from the font and text size for normal text.
  - **text-max-angle**: The maximum allowed angle difference between two adjacent characters in curved text.
  - **text-max-width**: The maximum allowed width for text wrapping.
  - **text-opacity**: The opacity of the text.
  - **text-size**: font size.
  - **text-transform**: Whether the text should be all uppercase, all lowercase, or not altered.

Depending on its type, each layer style is parsed into one of the following classes: `BackgroundStyle`, `FillLayerStyle`, `LineLayerStyle`, `SymbolLayerStyle`, or `NotImplementedStyle`. All these classes are subclasses of the `AbstractLayerStyle` class, which encapsulates all the data that is common for all layer style types. The shared style properties are:

- **ID**: The id of the layer style.
- **source-layer**: This references the id of the layers in the actual vector data. This is what is used to tie each layer style to its corresponding layer in the vector tile.
- **source**: The name of the source tile set.
- **minzoom**: The minimum zoom level for the layer.
- **maxzoom**: The maximum zoom level for the layer.
- **visibility**: Determines whether this layer should be rendered or not.
- **filter**: used to exclude certain features.

Since styling values can be of different value types, the layer style classes use `QVariant`[7] to store these values. Note that not all the styling values are raw values, sometimes they can be expressed as expressions that need to be resolved.

[7]The Qvariant class is a union for the most common Qt classes, see: `https://doc.qt.io/qt-6/qvariant.html`

5.3.1 Expression Parsing

Some style properties are written as expressions that need to be resolved in order to get the actual values. The expressions have their own specification [23]. Expressions are expressed as JSON lists in the style sheet. The most common expression list pattern is to have a string specifying the expression operation as the first element, and then the two operands to execute the expression on. To solve this problem, a basic interpreter that would resolve all the expressions currently supported by the application was implemented. The `Evaluator` static class is responsible for evaluating and resolving expressions. The class is made static because it doesn't need to hold any state. All the supported expression strings are saved in a map. Each expression string is saved as a key in the map, with the value being a pointer to the function that resolves that expression. The static function `Evaluator::resolveExpression` is responsible for resolving expressions by calling the appropriate function. Note that expressions can be nested, and to solve this, indirect recursion was used, where all the functions that deal with expressions can call `Evaluator::resolveExpression` if they encounter a nested expression. The supported expressions in the current application are:

- **get**: Takes one operand. It retrieves a property value from a feature's metadata.
- **has**: Takes one operand. It checks if a feature has a specific property in its metadata.
- **in**: Takes two operands, an input value, and a list of values. It checks if the input value exists in the list.
- **!=**: Takes two operands. Check for the inequality of the operands.
- **==**: Takes two operands. Check for the equality of the operands.
- **>**: Takes two operands. Check if the first operand is greater than the second.
- **all**: Takes a list of expressions, and checks if all the expressions evaluate to true.
- **case**: Takes a list of inputs and expression pairs, and returns the first input whose corresponding expression evaluates to true.
- **coalesce**: Takes a list of expressions and returns the result of the first expression that does not evaluate to null.
- **match**: Takes a label and a list of input and output value pairs. Returns the output whose input matches the label.
- **interpolate**: Takes an input value and a list of numbers. Performs a linear interpolation of the input value on the list of numbers.

5.4 Vector Tiles Parsing

After receiving the vector tile data from the Maptiler API, the data needs to be deserialized to extract meaningful information that can be used to render the vector data. This is done in two stages. The first stage is to deserialize the protocol buffer message, and then decode and parse the message content. The deserialization and parsing are done from the `Bach::tileFromByteArray` function.

5.4.1 Deserializing the Protocol Buffer Message

This is the first step to extract the information from the received network response. This is done through a deserialization method using the `QProtobufSerializer`[8] class object that deserializes the Protobuf message into the class generated by the protoc compiler (mentioned in subsection 2.3.1). Note that this step is a major cause of the poor performance of the application, this is due to a bug in the internal function for deserializing protocol buffer messages from Qt. A bug report has been issued for this.

5.4.2 Parsing and Decoding the Tile Data and Geometry

After the protocol buffer message has been deserialized, the class data has to be accessed to extract relevant information. This step starts by iterating over the list of layers inside the tile and creating `TileLayer` objects for each layer. The next step is to iterate over all the features for each layer and check its type. Checking the type of the feature is necessary because decoding geometry is done differently for each feature type. The loop will call the metadata parsing function `populateFeatureMetaData` and then decode the feature's geometry array to a `QPainterPath`[9] object using either `polygonFeatureFromProto`, `lineFeatureFromProto`, `textLineFeatureFromProto`, or `pointFeatureFromProto` depending on the feature type. The geometry decoding follows the MapBox Vector Tile specification [16]. The type of a feature is stored as an enumerator. Features of type `polygon` encode the geometry of countries, oceans, buildings, etc. Features of type `line` encode the geometry of borders, rivers, roads, etc. Features of type `point` encode the geometry of text such as continents, country

[8]See: `https://doc.qt.io/qt-6/qprotobufserializer.html`
[9]See: `https://doc.qt.io/qt-6/qpainterpath.html`

names, cities, place names, etc. One notable exception is that road names, and curved text in general are encoded as line features. Any other feature types are ignored.

5.5 Parallel Loading of Tiles

As per the requirements in chapter 3, it was important to load tiles in an asynchronous manner so as to not let slow tile-loading interfere with app responsiveness. As discussed in section 7.2, parsing tiles will halt the application if performed on the same thread as the GUI logic.

In order to address this, the team decided to employ background threads. Using background threads significantly improved performance and application responsiveness, by offloading long operations to independent execution units. This comes with the drawback that some parts of the visible map may not be present at all times. Instead, tiles will be loaded over time and placed into view as they finish loading.

QThreadPool

The implementation uses the *QThreadPool* class [24] from the Qt framework to manage background worker threads. This class implements the pattern within multi-threaded computing commonly known as *"thread-pool"* [25]. This allows the implementation to move away from the philosophy of managing threads, and instead manage asynchronous "tasks". It also simplifies the code by letting QThreadPool manage the lifetime of each thread.

When utilizing a thread-pool, tasks can be sent to a fixed collection of threads. Each thread will then extract a new task from an internal list as they finish executing tasks. This is useful in the case where many tiles are loaded. The tiles that are queued first will load with minimal interruption. This allows the CPU to allocate more time to process the tiles that are queued first, consequently, they are likely to finish processing earlier. This is in contrast to the alternative where there are many individual threads active, where all threads compete equally for time on the CPU. In such a scenario, the effect is that each tile takes longer to process, and will be finished in a less predictable order.

The performance impact of scaling from a single background thread to multiple has been measured and is discussed in section 7.3. Following the favorable results, the implementation configures the thread-pool to use the ideal number of threads based on the CPU being used[10].

5.5.1 Request Process

Any time a collection of tiles is requested from the TileLoader, one of two things will happen. If the tile in question is already loaded in memory, the tile is returned immediately. If the tile is *not* found in memory, it will be submitted as a job to the thread-pool for background loading. By letting the request function not block execution and return as early as possible, the function can be used inside rendering code without performance problems. Any kind of thread-blocking will lead to lower responsiveness and negatively impact the rendering performance. Consequently, it becomes important to return from the request function as soon as possible. The code for this process can be found in the function `TileLoader::requestTiles`, inside the file `lib/TileLoader.cpp`

5.5.2 Per-Task Operations

From then on, each task is responsible for loading a single tile (either from disk or web) and then inserting it correctly into the cache. A task is assigned exactly one tile-coordinate that it should load. The task must then determine whether the tile being loaded is already present on the disk cache. If the tile is not already present on disk, the byte stream is requested through a network request. If the tile is present on disk, it can be loaded from file. Both of these approaches will result in a byte stream, modeled by the QByteArray class [26], containing the serialized MapBox Vector Tile data [16].

Once we've loaded the necessary QByteArray, what remains is to parse it into a VectorTile object, write it to the disk cache, and then finally insert it into the memory storage.

Because there are multiple threads attempting to put data into a single, shared memory storage, thread-synchronization is used to prevent race-conditions [27]. A mutex lock [28] is employed to cover all shared data, which each thread must lock before memory storage can be modified.

Finally, whenever a new tile has been inserted into memory, it needs to trigger the signal callback to tell other systems that a new tile is now available. In the implementation, the only system that gets notified is the MapWidget object.

[10]Which is the default behavior at the time of writing

5.6 Rendering Map Data

5.6.1 Rendering Polygons

Rendering polygons is a straightforward task when using `QPainter` [20]. All the rendering code does is pass the path containing the polygon and apply the style parameters to the painter object, then call `QPainter::drawPath` [29] method.

5.6.2 Rendering Lines

The process for rendering lines is the exact same as rendering polygons, except lines have additional styling properties(see section 5.3.)

5.6.3 Rendering Text

Text features are a special type of feature in the sense that the geometry only describes the position of the text and not the text content itself. The text content is in the feature metadata (attributes). Text features usually contain the text content in more than one language. To determine which text is required, the "text-field" property in the feature's corresponding layer style is used. Text is the last component to be rendered on screen, as it should be rendered on top of everything else. Most text features, such as continent names, country names, city names, and place names are point features. However, some text features, such as road names, are line features since this type of text is supposed to be rendered along a straight or curved line.

Place Names

Most tiles will include a very dense set of coordinates on the world map for displaying text. Consequently, each piece of text will be positioned close enough together to overlap. This effect is in some cases quite extreme and will cause the text to overlap until it becomes unreadable. It is necessary to implement a collision detection algorithm for the text, to determine what text should be shown or hidden. Consequently, all text features need to go through a pre-processing step. In this step, all text features are sorted based on a property “rank”, which determines how important a text feature is. Text with a higher rank has higher rendering priority in case of collision. The text features are then iterated and parsed into a struct that contains the text's content, position, font, color, and outline data. After text has been processed, it is added to a list to be rendered. Only text that has passed the collision detection filter is added to the list. This list is then looped through to render all its text elements. After that, everything else is rendered in the current viewport. The first iteration of text rendering was done using the `QPainter::drawText` method [30]. However, this method did not provide any utility for text outlines. To solve this, `QPainterPath` [31] was used for the text, and an additional outline was added. This solution produced correct results, but it was very slow since the text had to be converted into a path with the `QPainterPath::addText` method [32]. The final software uses the `QTextLayout::draw` method [33], which provides a way to add text outlines without too big of an overhead. Originally, text was rendered using the `QPainter::drawPath` method [29]. Note that the `QPainter`'s drawing method does not support text outline and text wrapping at the same time.

This was solved by switching to `QTextLayout` and creating a custom function for text wrapping. A special case of normal text is a long text, which includes "United States of America", "People's Republic of China", and "Democratic Republic of the Congo". Text that would exceed the maximum allowed width specified by the "text-max-width" property would need to be split into several lines. The split text is treated as if it were multiple separate instances of text. The text positioning is done through the following formula: $y = (I - L/2) * H$ where I is the zero-based index of the text in the text array, L is the length of the array, and H is the height of the text given the current font size. All the text elements in the array have the same X coordinate.

Rendering Curved text

Curved text is a special case of text where the rendering must follow a line path. Since the Qt framework does not provide such functionality, a custom solution must be implemented. The process for curved text rendering goes through the following steps:

- **Stage 1: Process the Feature Geometry**
  Curved text features are of type `Line`. However, they are processed differently from normal line features. Line features can be composed of multiple lines, which complicates determining exactly where the text should be rendered. To solve this, a separate function was implemented for decoding the line geometry. This function returns only the longest line segment in the list of line segments.

- **Stage 2: Process the Text**

  The first step in this stage is to collect the text that will be rendered. The code then checks if the text can fit within the line. Instances of text that do not fit are discarded. The text must then be processed character by character to determine the position and rotation of each character using the following algorithm (`CurrentLength` is set to 0 at the start of the loop):

  1. Get the coordinates of the point along the path at `CurrentLength`
  2. Get the angle of the path at `CurrentLength`.
  3. Check if the difference between this angle and the previous angle exceeds the maximum allowed angle difference. If it exceeds this value, return from the function.
  4. Increase `CurrentLength` by the width of the character + letter spacing.

  After being processed, the curved text is stored in a struct containing the font, text color, opacity, outline details, and a list of structs each containing a single character with its position and rotation angle.

- **Stage 3: Render the text**

  Rendering curved text is also done using the `QTextLayout::draw` method [33]. Note that all the text for the entire viewport is rendered in one loop. All the elements in the text list are iterated over, and for each element, individual character structs are looped over. Finally, they are rendered using their corresponding position and rotation angle.

Chapter 6

Development Process

In this chapter, the development process and Scrum as a project management framework will be discussed. The project plan (see Appendix A) outlines the original plan, and the team adhered to it to a large degree. This chapter outlines the overarching development methodology. Later, in section 9.5, the implementation of the development plan is discussed.

6.1 Development Plan

The project can be split into three phases: (1) The planning phase; (2) the implementation phase; and (3) the reporting phase. Note that the documentation process leading to the final report was performed during the entire development process.

The planning phase was dedicated to those activities preceding the implementation phase, such as establishing contact with the supervisor and product owner, acquiring the key competencies, learning the key technologies (like C++ and the Qt framework), familiarizing with the development environment, planning the project development (establishing team guidelines, procedures, and rules), establishing the software requirements and system design. While some parts of the software were developed during this time, the major goal of this period was to prepare for the following development phases.

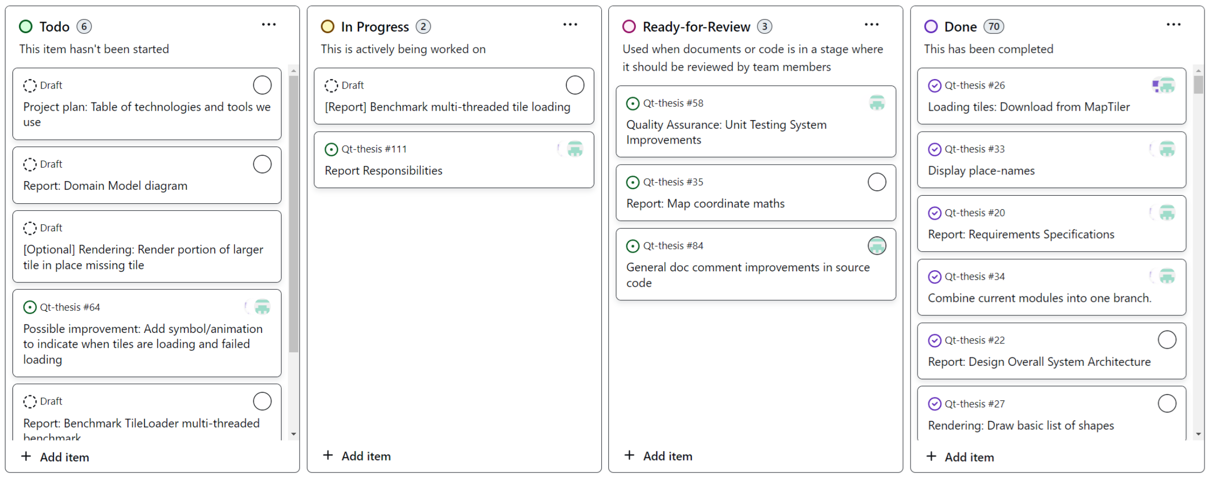
The dedicated development phase consisted of developing the software itself and implementing features listed in the project plan while maintaining the code base and improving quality along the way. During this time, there was also a change of supervisor for the project, so a week was spent familiarising the new supervisor with the project.

During the final phase, the dedicated reporting period, most of the team's time was dedicated to writing the final project report and making minor improvements to existing software, without implementing major new features. Additional new features wouldn't be implemented unless the product owner required it, to avoid delivering unplanned, unfinished, untested, or poorly documented code.

Minor improvements included cleaning up code and improving its readability by changing its style, renaming variables to be more descriptive, adding and improving code documentation, fixing bugs, improving error handling to ensure software behaves as expected, and updating unit tests.

6.2 Scrum

During the development process, a customized version of Scrum was used, Figure 6.1 shows the Scrum board used to organize the project tasks. Each sprint would last for two weeks and have sprint goals. Each sprint would be planned during a Scrum planning meeting at the beginning of the sprint. The work and workflow would then be reviewed halfway through the sprint during a Scrum review meeting. On the final day of the sprint (or the same day as the next sprint would start) there would be a Scrum retrospective where the workflow was assessed and adjusted. In addition, the team had stand-up meetings multiple times per week, usually 4 days a week. These meetings would last between 15-60 minutes and completed and planned daily work would be discussed. These stand-ups ensured consistent teamwork and progress and allowed the team to address any problems. Eimen and Cecilia were required to notify the team if they had any suggestions to improve project quality, and this was primarily done during standup meetings.


Todo 6
This item hasn't been started

Draft
Project plan: Table of technologies and tools we use

Draft
Report: Domain Model diagram

Draft
[Optional] Rendering: Render portion of larger tile in place missing tile

Qt-thesis #64
Possible improvement: Add symbol/animation to indicate when tiles are loading and failed loading

Draft
Report: Benchmark TileLoader multi-threaded benchmark

+ Add item

In Progress 2
This is actively being worked on

Draft
[Report] Benchmark multi-threaded tile loading

Qt-thesis #111
Report Responsibilities

+ Add item

Ready-for-Review 3
Used when documents or code is in a stage where it should be reviewed by team members

Qt-thesis #58
Quality Assurance: Unit Testing System Improvements

Qt-thesis #35
Report: Map coordinate maths

Qt-thesis #84
General doc comment improvements in source code

+ Add item

Done 70
This has been completed

Qt-thesis #26
Loading tiles: Download from MapTiler

Qt-thesis #33
Display place-names

Qt-thesis #20
Report: Requirements Specifications

Qt-thesis #34
Combine current modules into one branch.

Qt-thesis #22
Report: Design Overall System Architecture

Qt-thesis #27
Rendering: Draw basic list of shapes

+ Add item

Figure 6.1: Github project ScrumBoard screenshot

The team has two programming and one computer science student. Two of us were also retaking courses this semester, leading to the team having different schedules. Therefore, Scrum usage had to be customized. The differences in study programs meant at least one person was busy with other projects for a few hours to a couple of weeks, and the team accommodated this when planning work.

Scrum was used since it's an agile workflow. It allowed for planning work for each sprint, adjusting the work during the sprint, and moving development activities to different times based on reviews and retrospectives. This made the workflow flexible.

Chapter 7

Testing and Quality Assurance

During this project, the team learned how to perform and implement different kinds of tests: unit tests for C++ with Qt using the Qt Test framework, rendering component tests, and an acceptance test with the product owner to check that the software was acceptable to Qt. The tests were performed to learn new testing techniques, test the quality of the software in multiple ways, and finally use the results to improve software quality.

7.1 Unit Testing

Unit tests were developed using the Qt Test module[1], which is part of the Qt framework. Unit testing serves as a basis for testing individual components of the project's software [22, p. 232]. Implementing unit tests is beneficial to improve software quality, locate bugs, and provide documentation [34].

No one in the team was familiar with C++ unit testing frameworks ahead of development. It is possible to develop unit tests with Doctest[2], Catch2[3], or Qt Test. Doctest and Catch2 seem to be popular unit testing frameworks for C++, but Qt Test supports the Qt framework out of the box, has a lot of documentation available online, and the product owner could help if there were problems. Qt Test was chosen as the unit test framework for these reasons.

[1]Qt Test documentation: https://doc.qt.io/qt-6/qttest-index.html

[2]Doctest can be found here: `https://github.com/doctest/doctest/blob/master/doc/markdown/tutorial.md`

[3]Catch2: `https://github.com/catchorg/Catch2`

When using Qt Test, CMake can add test resources to each test file and build test executables. Each Qt Test executable can contain multiple unit tests inside. Each executable built with Qt Test will set up its own "test" main function environment from where implemented tests are performed. The unit tests were organized so that `.cpp` files in the `lib` folder had their own Qt Test unit test executable in the `unit-tests` folder.

Tests can be run from the terminal with the `ctest` command, or they can be run directly from Qt Creator.

Qt has a code coverage tool called Squish Coco, which relies on another tool called the Code Coverage Browser. Squish Coco generates code coverage reports and the Code Coverage browser displays which parts of the code have been unit-tested. Since the team had decided to use Qt Test, using Coco to generate code coverage reports was explored. There are multiple steps that must be completed to get Squish Coco to work on Windows, and the team was unable to set it up due to build complications. The project supervisor recommended making as many unit tests as possible and not spending additional time getting code coverage tools to work. This was also discussed with the product owner, who agreed.

7.2 Benchmark: Tile Parsing

Benchmark tests are used to "... determine current performance and can be used to improve application performance" [35]. The purpose of the tile parsing benchmark is to determine whether parsing tiles is so slow that it warrants being performed as an asynchronous background task.

During the early stages of development, the team and product owner observed the parsing of tiles using the Qt gRPC module to be slow. One example of this was when the team tried loading an entire map with 16 tiles in the debug mode. After 10 minutes had passed, only 4 tiles had finished loading and rendered to the screen. The following discusses how it was confirmed that the performance issue is caused by the Qt gRPC module and not related to the tile-loading code developed by the team. How the results support the decision to make the application multi-threaded, is also discussed.

This benchmark can be run locally by running the executable `tile_parsing_benchmark`.

7.2.1 Benchmarking Test Methodology

The benchmark test loads all the tiles in zoom level 2, which sums to 16 tiles in total. Each file is loaded from disk into memory before any test runs to avoid disk IO interfering with the results. This test is executed 10 times to get a sample pool. Then, the average load time per file is calculated to estimate how much time is spent loading a tile.

This test was run locally on Windows 11, with a Ryzen 3700X CPU and 32GB of memory. The test was not run on a clean installation of the operating system, but an attempt was made to minimize the amount of other processes running on the system by terminating non-critical system processes. The Microsoft Visual C++ (MSVC) compiler was used. The test was performed in two configurations, once with the compiler set to the debug profile and once with the release profile.

7.2.2 Expected Results

The vector data that the software loads follows the MapBox Vector Tile v1.0 standard [16], which is in turn a standard built on the Protobuf technology. Protobuf is designed to be an efficient data-interchange format with fast deserialization speeds. Comparable, exact measurements on Protobuf have proven hard to find. The team hopes to see near-instant parsing speeds when dealing with file sizes that range from 10 KiB to 1 MiB.

7.2.3 Test Results and Discussion

The test results are shown in Figure 7.1. The results show that loading a single tile, on average, takes 8684.52 ms for the debug profile, and 1534.64 ms for the release profile.

The test only loads high-level tiles, which the team has observed to be somewhat slower to parse than tiles in higher zoom levels.

In addition, the test benchmarks the function VectorTile::fromByteArray(const QByteArray&). This function both performs the Protobuf parsing and parses the intermediary struct into a structure that can then be displayed. An improvement could be modifying the test to use only the internal implementation that performs the Protobuf parsing section.

The benchmark shows that no profile is fast enough to leverage a responsive user experience without offloading work to some background execution. A satisfactory result would involve being able to load ~16 tiles quickly enough to not produce visible stuttering. It

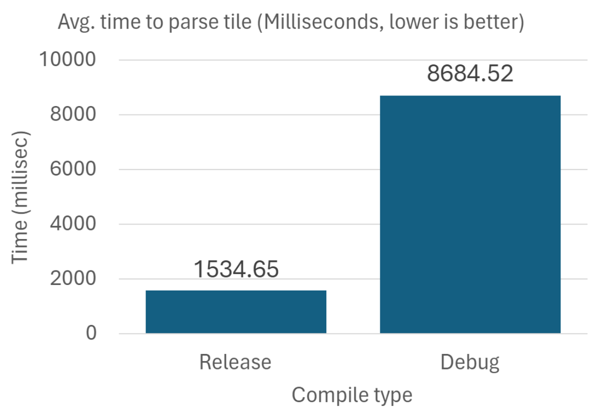


Figure 7.1: Tile parsing benchmark results

must be possible to load multiple tiles without the user interface becoming unresponsive. Therefore, it is beneficial to offload the tile-parsing workload to other threads, which will not interfere with application responsiveness.

7.3 Benchmark: Multithreaded TileLoader

The purpose of this benchmark is to determine whether using multiple background worker threads yields a performance improvement – compared to using a single background worker thread – when loading multiple vector tiles. The total time it takes to load multiple tiles will be inspected under different configurations.

The test compares performance by requesting multiple sets of tiles to the TileLoader. How many background threads the TileLoader is allowed to use will also be adjusted, and the performance will be tested for each case. The benchmark can be run locally by running the executable `tileloader_threaded_benchmark`.

7.3.1 Methodology

The test pre-loads tile files into a temporary folder, which can then be read by the TileLoader. The TileLoader is configured to not pull any tiles from the web. The TileLoader has been configured to only load vector tiles[4]. A new instance of the TileLoader is created between each iteration to not allow the TileLoader to cache data between runs.

[4]Meaning no raster-based tiles are loaded.

The selection of tiles has been chosen based on observations made by the team during normal use of the application. The team has observed that the number of tiles being processed when moving the viewport around varies anywhere from 0 to $\sim$16. The range of tiles being loaded was decided to be from 1 to 32. The selection of which specific tiles to load was determined by attempting to emulate a heavy load. Therefore, the tiles are chosen by listing all tiles in zoom-level 3 (64 tiles in total, using BasicV2 style), sorting the tiles by file size in descending order, and selecting the first 1 to 32 tiles.

The test was run locally on Windows 11, with a Ryzen 3700X CPU and 32GB of memory with a Kingston A2000 SSD. The test was not run on a clean installation of the operating system, but an attempt was made to minimize the number of other processes running on the system by terminating processes not critical to the operating system. The build uses the MSVC compiler with the ”Release” CMake profile. The test will run in configurations of a single thread and 16 threads, as that is the optimal thread count chosen by QThreadPool when used on this machine.

The benchmark initially compared loading tiles from disk and tiles pre-loaded in memory. Using the hardware mentioned, the performance difference was negligible[5]. For simplicity, this benchmark will only load from disk. This finding indicates that the workload is not IO-bound.

Each configuration was tested 5 times and an average was calculated for each configuration.

7.3.2 Expected Results

Inspection of source code shows that parsing an individual tile can be done largely in isolation. The team expects to be able to parallelize the process very effectively, given enough tiles to saturate all CPU cores. Consequently, enough tiles must be loaded in parallel to saturate all CPU cores on the system and maximize performance improvements. In theory, there should be an inverse linear relationship between time reduction and core-count ($100\% - 100\% \div$ core-count). Following this theory, the team would expect to see a 50% time reduction when using 2 cores, 75% time reduction with 4 cores, and 87.5% when using 8 cores. The team recognizes that the software and the test are subject to real-world inaccuracies and imperfect implementation. Therefore, the team expects significant performance improvement but far less than the theory would suggest.

[5] <1% difference in favor of loading from disk

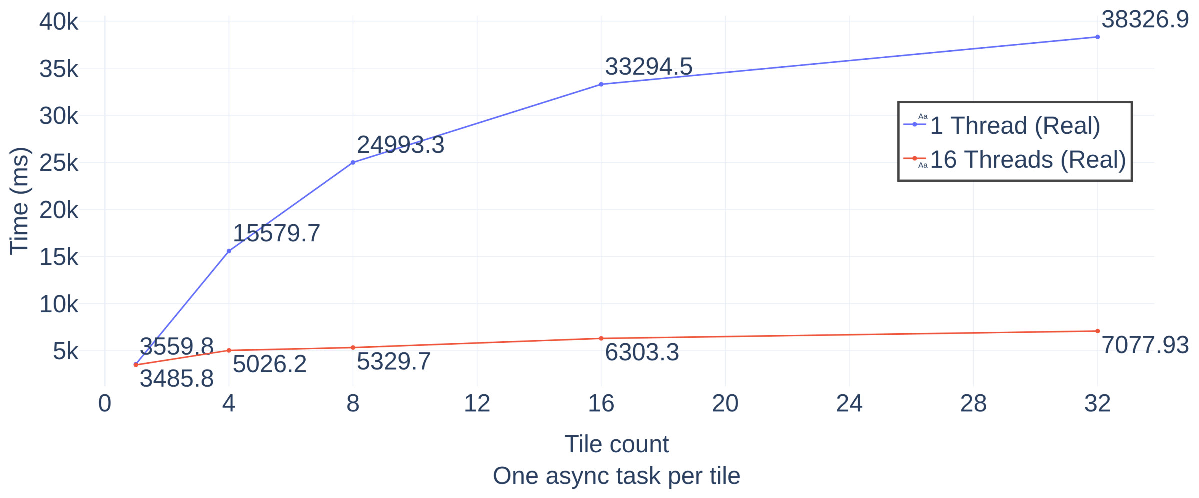


Figure 7.2: TileLoader threaded benchmark results. The horizontal axis represents the amount of tiles loaded in this data point. The vertical axis represents how much time it takes to load all the given tiles in milliseconds, where lower is better.

The team expects some variability in the results. The test data consists of unequal tiles. Each individual tile will have different contents, and this will impact the time it takes to parse the different tiles. How large this will impact the variability of the results is unclear ahead of the test.

The team expects that adding tiles while keeping the tile count less than or equal to the CPU core count, should introduce little to no increase in processing time. This is due to the test methodology being bottlenecked by the tile that takes the longest time to process, in this specific scenario.

7.3.3 Results and Discussion

The test results have been plotted in Figure 7.2. The results show that multithreading improves performance, but only when loading multiple tiles simultaneously. The results show a trend that adding more tiles introduces diminishing additions to the overall processing time. This is inherent due to our method for extracting tiles from zoom-level 3, where the first tiles are expected to load more slowly than the latter tiles due to the tiles with fewer data being loaded last. At tilecount = 8 there was a 78.7% reduction in processing time, which was close to the expectation set by the team (87.5%).

As expected, new tiles can be loaded with nearly no impact on load times as long as the number of tiles is less than or equal to the number of cores.

QThreadPool will create the background threads on-demand by default, and the current implementation relies on this behavior. Instantiating new threads can be costly, and because they are instantiated during testing, test results are less accurate. An improvement to the test could include instantiating threads ahead of time to more closely mimic or represent a `TileLoader` object that has already been alive for some time.

The test shows a clear pattern of increased throughput of tile processing when using multiple threads. This supports the team's decision to employ multiple background worker threads when processing tiles.

7.4 Continuous Integration

In March 2024, unit tests were correctly deployed on GitHub. The system was set up so that unit tests would be built from the `dev` and `main` branches and had to pass before other branches could be merged into them.

CI is used to ensure that code base updates do not break the previously developed system and that the system behaves as expected.

The CI setup uses GitHub Actions [36] and Docker [37]. GitHub Actions workflows build the project and then run unit tests and Merlin graphics test (see subsection 7.5.3). If these tests fail, the pull requests are blocked until the code or tests have been updated. The most important workflow is called `main.yml` and can be found in the repository under the path `.github/workflows/main.yml`.

As the required version of Qt (v6.7) is not provided in any of the available GitHub Actions images at the time of writing[6], Qt must be compiled from source in order for the project to build and run correctly. This compilation process usually takes at least 20 minutes. With a minimal setup, this compilation process would have to be executed all over again every time the workflow is initiated. The consequence of this is that the continuous integration test would have very poor responsiveness.

To remedy this, a Docker image based on Ubuntu 22.04 was created. A `Dockerfile` script clones, builds, and installs Qt. Basing the continuous integration on this Docker image makes it possible to omit compiling Qt every time the Actions workflow compiles the project.

[6]At the time of writing when using Ubuntu 22.04, only Qt v6.2.4 was available.

This significantly improves the responsiveness of the continuous integration, allowing it to complete significantly faster.

7.5 Testing of Graphical Output

Component rendering tests were implemented to increase robustness during development, by having that tests that can automatically detect rendering regressions. A large portion of the project entails rendering, and implementing rendering tests was therefore deemed to be critical to ensure robustness during development, even if it was not required by the product owner (see chapter 3).

If the incorrect rendering code is merged, the corresponding rendering tests should fail. The team therefore wanted to implement tests that could capture the expected results of the rendering software at a point in time where the team deemed the displayed map to be correct. New code would generate new rendering results, and these would be compared to the baseline. Note that the team is not qualified to determine when maps are technically correct, but a baseline would be established to be able to graphically compare old and new rendering.

7.5.1 Ideal Data-Flow

The rendering test subsystem was designed to be executed based on a collection of test cases. Each test included details about the viewport, the list of tiles to render, and how the map should be rendered (e.g. what elements to include in the rendering). The system should then be able to produce a set of output images based on the test cases. The team assumed that a collection of image files to be sufficient for these outputs, one image per test case.

It would then be possible to generate these output images at a specific point in time for future comparisons with images generated by new(er) rendering code. This set of outputs will be referred to as a *baseline*, which will form the basis of future graphics rendering tests. Whenever the behavior of the rendering functionality is updated and intentionally produces different images, the baseline would have to be updated.

Graphics tests were performed on the baseline using newer code. The newer code would attempt to generate the same test cases and compare the newly generated output to the equivalent output found in the baseline. If the tests discover that the results are unacceptable, it is indicative of either a regression in the rendering functionality or that the baseline has to be rebuilt to reflect new rendering behavior.

Ideally, there should exist some mechanism to inspect the failed test cases, so that it is easier to isolate what the issue might be. The rendering tests should be simple to run both locally, and remotely inside the CI workflow so that everyone in the team can run tests and inspect the results easily.

One expected issue with comparing images is accounting for minor differences in rendering behavior across platforms and devices [38]. The team expects the tests to record minor imperceptible differences between outputs due to internal implementation details. It would be beneficial to have a threshold mechanism that allows for minor differences to be evaluated as acceptable by the tests.

7.5.2 Lancelot

Qt recommended using one of their internal tools, “Lancelot”, for image comparison. Lancelot is used to catch regressions in Qt’s existing rendering code. Lancelot seemed to cover most of the rendering test requirements, it would establish a baseline, save the results, and then newer code could be tested against the baseline. However, the use of Lancelot was dropped due to its complexity and poor documentation. No team member was able to understand how to proceed with the setup. After careful consideration, the team collectively decided to create their own simple solution as an alternative to Lancelot. This solution was dubbed Merlin.

7.5.3 Merlin

Merlin handles automated testing of graphical output. Merlin is able to read a predetermined JSON file containing test cases and output the expected rendered images as image files.

The team wanted to keep Merlin as small as possible and not involve too many systems. For the image comparison evaluation, the team agreed that using third-party tools would be more reliable than what the team could produce on their own. The solution was to use an external tool called *ImageMagick* [39] to handle the image comparison evaluation, this also implements the functionality for acceptable differential thresholds.

Detailed instructions on how to operate Merlin, as well as defining test cases, are found in the file `tests/merlin/README.md` in the source code repository [5].

7.5.4 Output

If all test cases were found acceptable, then the `merlin_rendering_output_tests` program will return a success result code. However, in the case that a test case is deemed unacceptable, the executable will generate a folder `failure_report` which will contain the specific test cases that failed, along with the expected baseline and a differential image. This report is also output as an artifact during CI.

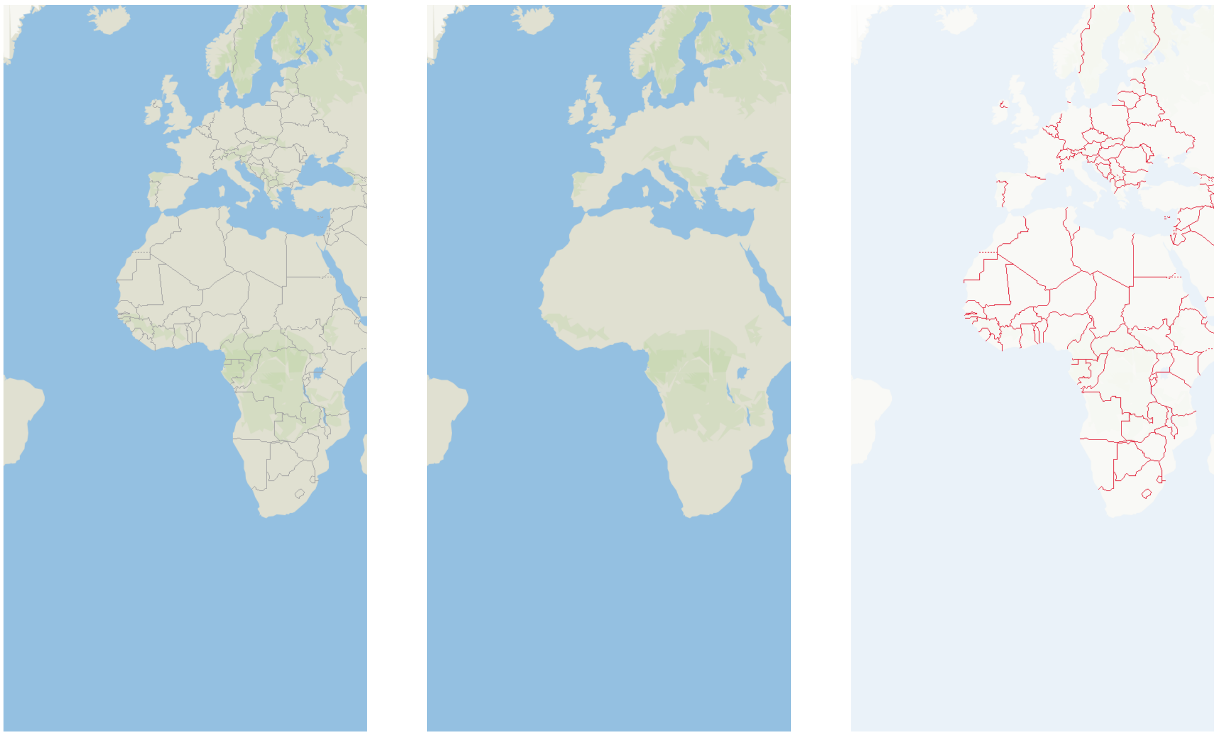
This type of error report is demonstrated in Figure 7.3. This figure demonstrates a regression, where lines are not being rendered. The intended correct behavior is to have lines rendered during tests that enable them. In such a scenario, red highlights will be rendered in all areas (pixels) where the produced image differs from the baseline past the threshold.

Figure 7.3a shows an image of what is considered to be the correct output, as described by the baseline. Figure 7.3b shows an image of what is outputted by the current state of the software, in this case with an intentional regression. Figure 7.3c shows the differential between the previous two outputs. Any parts that deviate from the baseline are highlighted in red. In this case, this means all the lines that are missing in the rendering are highlighted in red. A critical limitation of this tool is that it is unable to reliably determine the correctness of text.

7.6 Acceptance Test

Acceptance testing is the final phase of software testing [40], and it "... ensure[s] that systems are high-quality and meet the needs of their users" [41]. If a product passes an acceptance test, it means the product is adequate and is approved by its end user [42, 43]. On May 14th, 2024, the product owner performed an acceptance test to approve the final delivered software.

The test had 10 test cases with tasks to perform, and the test design is inspired by technologies [41]. If the tester deemed the test result to be acceptable, "Pass" is filled into the table, otherwise "Fail" is filled in. The test cases were based on the software requirements. In the end, all tests passed, and the product owner approved the software. The full test can be found in Appendix K, while a brief overview is available in table 7.1. The acceptance test covers all project requirements, except NFR 4. As mentioned in subsection 3.1.2, writing and delivering the thesis itself is what meets this requirement.



(a) Baseline (b) Generated (c) Diff

Figure 7.3: Merlin report example

Table 7.1: Acceptance test summary

Test	**Task**	**Requirements**	**Result**
01	Download and run the program successfully.	FR 1	Pass
02	Run the program and check that vector tiles are loading.	FR 1	Pass
03	Run the program and wait for the map to finish loading.	FR 2, FR 6	Pass
04	Zoom in until text, roads, rivers, and buildings are rendering.	FR 4	Pass
05	Confirm that `.mvt` files are loaded and cached.	FR 3	Pass
06	Confirm that Basic V2 Map Type is used.	FR 5	Pass
07	Confirm that Qt can continue development of codebase.	NFR 1	Pass
08	Check that Qt functionality is used in the software.	NFR 2	Pass
09	Confirm that `QPainter` has been used in the codebase.	NFR 3	Pass
10	Confirm that the project uses an MIT license.	NFR 5	Pass

7.7 Quality Assurance

7.7.1 External and Internal Standards

One development goal was to follow Qt's best practices[7], while also adhering to C++ best practices[8]. Code documentation, in terms of comments, was written using the QDoc standard (a documentation style and tool developed by Qt)[9], that is similar to Doxygen. If QDoc is to be used to generate official documentation, the documentation comments should be on the `.cpp` files, not `.h` files. This is why most of the project source code has documentation comments on `.cpp` files.

Furthermore, the team established a set of rules for the use of version control systems. In this project, Git was employed. Each commit had to follow the internal Git Guidelines document (see the Project Plan in Appendix A). This was done to improve the quality of each commit by explaining briefly and precisely the commit content, making the Git workspace more professional. Each commit had to describe what had been implemented or changed and mention any relevant issues in the Issue Tracker. Commits like this made it easier to track what was implemented when, tie issues to commits, and roll back when mistakes were made.

Additionally, rules were introduced for how to interact with the `main` and `dev` branches. The `dev` branch was a staging post before pulling major features to `main`. No one could pull directly to `main`, and all pulls to `main` had to be done via merge requests. When making merge requests to `main` and `dev`, at least one other person had to review the code and approve the pull request. If any problems or errors were caught at this stage, all problems had to be addressed before merging. This added a manual check before every pull to `main`, meaning no one could make accidental or unplanned changes to the project's most important branch. The team was allowed to make changes directly to `dev` for minor changes, but the group still reviewed each other's work for larger pull requests into `dev` to maintain good software quality by performing regular code reviews.

All team meetings, supervision meetings, and meetings with Qt were documented following meeting template documents made by the team (see. The meeting documents helped organize and document the workflow consistently. Scrum structured the workflow, while the

[7]Qt best practices can be found here: `https://doc.qt.io/qt-6/best-practices.html`
[8]C++ core guidelines: `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`
[9]Writing documentation with QDoc: `https://doc.qt.io/qt-6/qdoc-guide-writing.html`

meeting documentation shows what was worked on when, and the GitHub Issue Tracker[10] with its Scrum Board[11] were used to track the critical development and reporting tasks. Using the Issue Tracker and Scrum board allowed the team to track development and split tasks between team members.

[10]Issue Tracker: `https://github.com/cecilianor/Qt-thesis/issues`
[11]Scrum Board: `https://github.com/users/cecilianor/projects/2`

Chapter 8

Installation

This chapter covers building and running the source code. The exact instructions can be found in the repository associated with this thesis [5], in the `BUILD_INSTRUCTIONS.md` and `HOW_TO_RUN.md` files. Details on how to use the Merlin subsystem can be found in the file `tests/merlin/README.md`. Any other related details will be listed in the `README.md` file located in the root directory of the repository.

8.1 Software Dependencies and Tools

The software dependencies and tools used in this project, as well as their versions, are listed in Table 8.1.

The project strictly relies on Qt v6.7.0 (and up) due to fixes applied to the Qt GRPC module. At the time of writing, this version is very new and therefore is not available through package managers such as `apt-get` on Ubuntu 22.04. This means the build process on Linux includes building Qt from source, which requires numerous additional steps to build the project. The team was unable to build Qt from source on Windows due to not being able to resolve the third-party library requirements of the Qt GRPC module.

The fixes necessary to the code present in v6.7.0 involve updating Qt Protobuf API functionality, specifically to identify the type of decoded Protobuf values. If one tries to compile the project for Qt v6.6.3, there are several compiler errors, most of which are shown in Listing 8.1. These bugs were fixed in v6.7.0.

Table 8.1: Software dependencies and tools

Tool	Version	Usage
Windows 11	22H2	Primary OS used for development
Ubuntu	22.04	Used for Dockerising the project (for GitHub CI/CD)
git	2.39.1	Source control. Downloading source files.
Qt	6.7.0	Rendering, networking, Protobuf parsing, containers, threading, helper utilities
GCC	11.4.0	C++ compiler for Ubuntu
MSVC	v143	C++ compiler for Windows
CMake	3.22.1	Generating build files
Ninja	1.10.2	Running compilation and linker step on build files
vcpkg	2024-01-11	Package manager for Windows
Protobuf	3.21.12	Dependency for QtGrpc module
Image Magick	6.9.11	Image comparison during rendering output tests

8.2 Application Prequisites

A prerequisite to run the main application executable is to supply it with a private MapTiler API key. This allows the application to successfully get map data from MapTiler. If the reader wishes to run the supplied code, they must get a (free) MapTiler API key before running the application.

A user can either set the key as an environment variable or supply the key in a text file and place it in the same folder as the executable. Exact instructions are found in the file `HOW_TO_RUN.md` (see the source code associated with this thesis [5]).

```
Qt-thesis\lib\VectorTiles.cpp(346): error C2039: 'hasStringValue': is not a member of
    'vector_tile::Tile_QtProtobufNested::Value'
Qt-thesis\build\Desktop_Qt_6_6_3_MSVC2019_64bit-Debug\vector_tile.qpb.h(80): note: see
    declaration of 'vector_tile::Tile_QtProtobufNested::Value'
Error continues...
```

Listing 8.1: Qt v6.6 compiler errors

Chapter 9

Discussions and Conclusions

9.1 Discussions

Developing the map application has shown that it is possible to generate and render a geographical map using vector-based data. The learning curve has been steep due to the team having to acquire many new skills, and this has led to multiple aspects of development being both interesting and challenging.

When implementing the network component, two major issues had to be addressed. First, the networking component had to be able to store and handle errors gracefully, while bearing in mind that the product owner wanted its error-handling system to be simple. This was solved by storing data and its potential error together in a struct. This solution was simple to use and update.

Qt's QNetworkAccessManager class blocks rendering when both network handling and rendering run on the same thread. When the rendering component would wait for the network access manager to complete execution, the application would appear to be unresponsive. Multithreading solved this problem. It also improved application execution times since the loading of individual tiles could be moved to separate threads and be executed in paralle l(see section 7.3).

After a few months of development, support for tile caching to disk was implemented. This allowed for fewer network calls to be made during program execution.

The vector tile decoder is another crucial component of the application. Its implementation was straightforward for the most part (see section 5.4). A challenge with implementing this component is that road names, and curved text in general, are tied to a point layer style while they contain line geometry. To solve this, all instances of curved text had to be isolated from other line features and processed separately.

Another aspect of the project that was challenging is text rendering. Outlining text and allowing text wrapping is crucial to render text correctly. The issue is that Qt does not have functionality that supports both features simultaneously. To solve this issue, two approaches were considered. The first approach involves using the QPainter class[1]. When QPainter is used, the text and its position are passed to the `QPainter::drawText` method[2], and the painter object will take care of rendering the text. The advantage of this approach is that it has a simple implementation and the painter object automatically takes care of text wrapping. The disadvantage is that the method does not support text outlines. Alternatively, `QPainter::drawPath`[3] can be used to manually add text outlines after converting the text to a painter path object using `QPainterPath::addText` method[4]. However, this method does not offer automatic text wrapping. This approach also lowers the rendering quality and negatively affects performance. The latter method was used during the early stages of text rendering, then the team consulted with Qt and was directed to use the `QTextLayout`[5] class. This class offers text outlines, as well as being faster than directly using the painter, and offers better rendering quality. However, it does not offer any functionality for text wrapping, so this functionality has to be implemented by the team. The team decided to use the `QTextLayout` class with a custom function for text wrapping. Using the automatic text outline and getting better quality and performance outweigh the disadvantages of missing the text wrapping functionality.

The project also uncovered bugs in Qt's underlying software, which have been rectified in parallel with the map software development. One of these bugs was the main cause of the performance issues in the application, and is related to parsing the protocol buffers into C++ classes using `QProtobufSerializer`.

Developing the software on Windows has proven to be challenging, and the team recommends using a Unix-based operating system like Linux for future development.

[1]See: `https://doc.qt.io/qt-6/qpainter.html`
[2]See: `https://doc.qt.io/qt-6/qpainter.html#drawText`
[3]See: `https://doc.qt.io/qt-6/qpainter.html#drawPath`
[4]See: `https://doc.qt.io/qt-6/qpainterpath.html#addText`
[5]See: `https://doc.qt.io/qt-6/qtextlayout.html`

9.1.1 Test Result Discussions

The system has been tested extensively using multiple different test methods. The most important one may be the acceptance test, where Qt accepts the software for final delivery, and confirms that the project meets *all requirements*. The passing unit tests confirm that the unit-tested functionality behaves as expected. The CI has supported development by ensuring that new functionality would integrate with older functionality without causing any software bugs.

The benchmark tests have shown that utilizing background threads is a necessity to ensure application responsiveness. The tests in section 7.2 show that the processing of tiles is slow to a problematic degree in both debug and release builds of the software. The benchmark tests in section 7.3 have shown that there is a significant performance improvement from utilizing multiple threads when loading tiles in parallel.

The introduction of multithreading greatly increased the responsiveness of the application but is still not within acceptable targets for production-ready software in either debug or release builds. It should be noted that even if the application performance in the release profile was sufficient, the debug profile performance is still problematic. Developers often need to develop and do iterative testing using the debug profile. If loading tiles is too slow, it will negatively affect the iteration times of the developers and impact overall productivity. As an example, while the product owner accepted the final version of the system, they pointed out that the software was so slow they thought it had stopped responding when running the application. The current performance bottleneck for processing tiles is confirmed by the product owner to be a performance bug within the Qt framework.

The developed Merlin tool has allowed for visual regression testing, and this would be impossible to test with unit tests.

9.1.2 Sustainability

The work in this thesis may contribute to the following United Nations (UN) Sustainability Goals [44].

- **Goal 04** – Quality Education
- **Goal 08** – Decent Work and Economic Growth
- **Goal 10** – Reduced inequalities
- **Goal 13** – Climate Action

The Qt framework itself and this thesis are distributed as open-source. Therefore all implementation details are available for the public to inspect and utilize. This information is available freely to everyone on the internet. Everyone with internet access, regardless of background, can access this information and may learn from studying the source code. This contributes to goals 4 and 10.

By making the software available for public use in this way, third parties may then go on to improve their own software products and in turn, create more competitive products for the software industry. This may facilitate economic growth in terms of bringing more competition to the market, and also through creating job positions that will have to maintain this functionality. This contributes to goal 8.

A feature of the software that may make the project more economically and environmentally sustainable, is the usage of data caching to disk. The project uses tiles that are grabbed dynamically from MapTiler. By reducing the amount of network requests, the cost to operate the software will be lowered as well as reducing the load on networking hardware. This in turn allows the hardware to live longer. This contributes to goals 8 and 13.

9.2 Usage of Artificial Intelligence

Little production code has been generated using artificial intelligence tools. The team experienced tools like ChatGPT to provide insufficient code advice and outdated information on how to utilize the modern parts of the Qt framework and hence advises against using AI tools to generate C++ code where Qt is used. Artificial intelligence tools were used for other things for the project.

About 10-20 % of unit tests were implemented with the help of AI tools. ChatGPT and Copilot provided guidance to correct unit tests. If the tools suggested tests were incorrect, the source code and corresponding tests would be inspected manually to ensure correctness.

AI tools were also used to guide the team when *containerizing* the project (Docker). ChatGPT was used to help solve problems with LaTeXand Overleaf to spend less time formatting the report, leaving more time for other project activities. Finally, both ChatGPT and Copilot were asked to help set up Squish Coco (Unsuccessfully, see chapter 7).

The team and supervisor believe the development of the project has been supported by a balanced use of AI tools. They were used with the aim of improving the learning and reducing the load on repetitive tasks and contributions that add little value, thus enabling the team to focus on more meaningful contributions to the project.

9.3 Criticisms of the Thesis

Before proposing this project, the team consulted with NTNU's teaching staff and Qt's representative, who anticipated this project proposal to be quite challenging. The initial work progressed well, and development was challenging, but not *too* challenging. As the work progressed, it gradually got more difficult.

Some problems in the thesis likely stem from the combination of a lack of background knowledge and skills at the start of the project, and the very steep learning curve. The team generally had limited knowledge of C++, Qt (with compilation of Qt from source), CMake, and test development for C++, and these technologies had to be acquired or learned for the project. Reaching sufficient skill levels was more demanding and took longer than anticipated.

Even if the final software meets its requirements, it would benefit from having additional work put into the caching mechanism, implementing the cache eviction policy, improving text rendering, and in particular improving the rendering of street names. Utilizing coverage reports could help locate untested code and support additional unit test development, ensuring a larger portion of the codebase could have been unit-tested thoroughly.

9.4 Future Work

This section highlights some of the improvements the team would like to have included if given more time to work on the project. These elements could contribute to creating a production-level application. An in-depth discussion of some possible improvements can be found in Appendix L.

A missing feature that is crucial to make the map application production-ready is the cache eviction policy. The implemented software caches tiles in memory. However, these tiles are stored in memory for the entirety of the program's lifetime. An eviction policy is necessary to limit memory usage. There are two simple approaches to this problem. The first is to implement a timeout-based eviction policy, where tiles that remain in memory for a given duration are deleted, and the timer is refreshed when the tile is accessed. The second approach is to implement a space-based eviction policy. In this approach, the cache has a maximum capacity. The capacity can be calculated either by tile count or utilized memory space, and whenever the cache is full, the application deletes the least used tiles.

The current application utilizes software accelerated rendering with QPainter [20]. A major improvement that should be considered for future work would be to implement hardware-accelerated graphics rendering for the application. This can be done by using QRhi [45].

Another feature that could be improved is the graphical interface. The current GUI is sufficient for a proof-of-concept application. However, for a production-level application, a complete redesign of the application's graphical interface must take place. This should be done after analyzing the target audience of the application and assessing the graphical interface of already existing map applications, as well as referencing the best practices for GUI design.

9.5 Assessment of Team Work

As previously mentioned in the thesis, the team consisted of two programming students and one computer science student. This provided the team with a broad range of skills and capabilities that could benefit the project. Figure 9.1 shows the teamwork distribution across thirteen categories, and Figure 9.2 lists the team's time distribution.

Most of the project time was spent coding (404 hours), writing and proofreading the thesis (382 and 111 hours), and having team meetings (296 hours), with a total work time of 1852 hours (see Figure 9.1 and Figure 9.2). Team meetings would usually consist of assigning work, and they would also include joint work sessions to design, program, or test the system.

Having frequent short team stand-up meetings ensured a consistent workflow, and weekly meetings with the supervisor ensured that the thesis was progressing and stayed on track. The project plan was followed to a large extent, but some adjustments were made during development, for example when the first viable prototype was completed during sprint 3 instead of the planned sprint 4. This allowed the team to spend additional time improving code quality, implementing additional unit tests, and researching how to render texts and cache data. Using Scrum was crucial to the success of the project, as it allowed for adjustments and improvements of the software during the development phase. It also helped mitigate any inconsistencies in the workflow of the project. In addition, the team rules listed

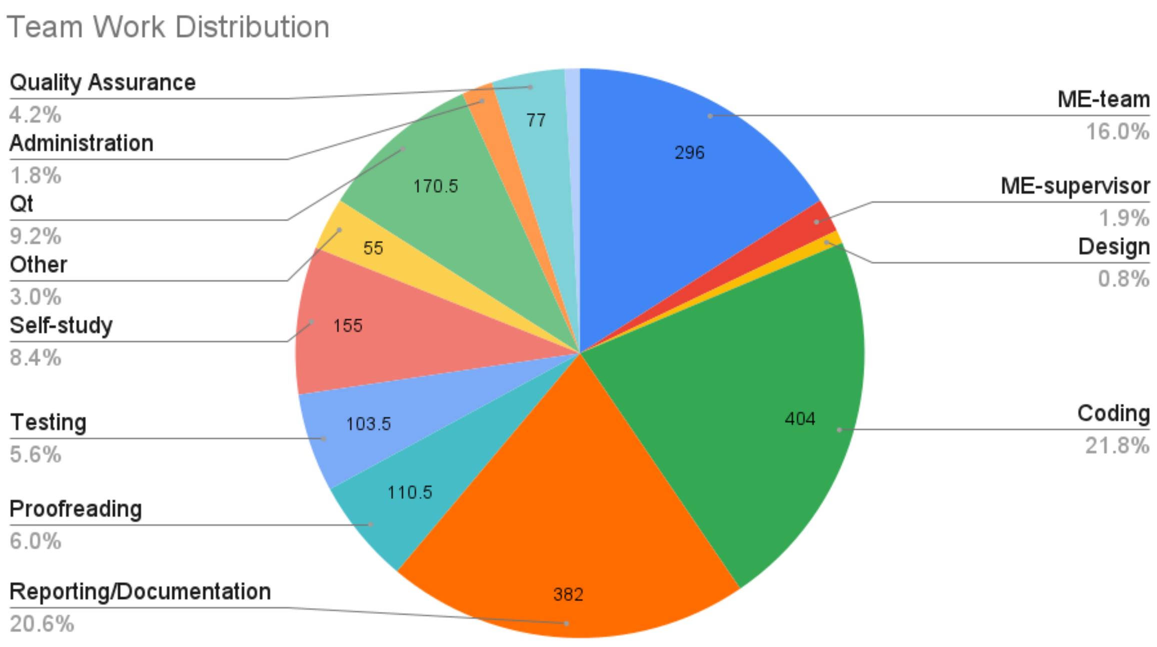


Figure 9.1: Team work distribution

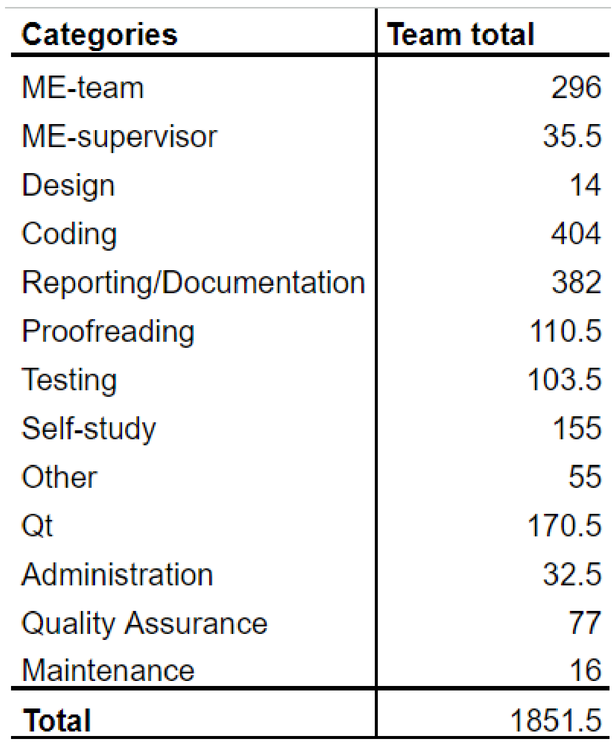
in Appendix A helped keep the project on track.

The team is very proud to have developed software that fulfills the requirements laid out by the product owner. The team believes the result goals and the learning goals outlined in section 1.4 have been achieved. While the effect goals are hard to evaluate at the time of writing, the team hopes they are achieved in time as well.

9.6 Final Conclusion

Both the development team and the product owner are pleased with the thesis outcomes. Matthias Rauter, Qt's representative, has given positive feedback on the project's results. Knowing that Qt is pleased with the product demonstrates that the project has been a success.

Working on the thesis has deepened our understanding of software development in the context of C++ and the Qt framework, and we are all more confident in our software development skills. In addition, we've gained a new set of skills, specifically in terms of using C++ and the Qt framework to develop a graphical application, unit testing C++ with Qt, setting up CI/CD environments, performing threading experiments, and ultimately learning to work as a team with different strengths, motivations, and approaches to programming. Furthermore, we have gained insight into the challenges that developers face in professional settings, from analyzing and fixing software bugs to setting up testing infrastructure and maintaining a large codebase.



Categories	Team total
ME-team	296
ME-supervisor	35.5
Design	14
Coding	404
Reporting/Documentation	382
Proofreading	110.5
Testing	103.5
Self-study	155
Other	55
Qt	170.5
Administration	32.5
Quality Assurance	77
Maintenance	16
Total	1851.5

Figure 9.2: Hourly statistics for the team

Bibliography

[1] HEAVY.AI. “What is hardware acceleration? definition and FAQs.” Accessed: 2024-02-28. (2022), [Online]. Available: `https://www.heavy.ai/technical-glossary/hardware-acceleration`.

[2] The Qt Group. “Qt Framework.” Accessed 2024-04-22. (2024), [Online]. Available: `https://www.qt.io/product/framework`.

[3] MapTiler. “Maptiler.” Accessed 2024-01-11. (2023), [Online]. Available: `https://www.maptiler.com/`.

[4] MapTiler. “Basic | lightweight basemap for overlaying own geodata.” Accessed 2024-04-22. (2024), [Online]. Available: `https://www.maptiler.com/maps/basic/`.

[5] C. N. Bratlie, E. Oueslati, and N. P. Skålerud. “Qt bachelor thesis repository.” Accessed 2024-05-20. Commit hash 47fc45b. (2024), [Online]. Available: `https://github.com/cecilianor/QT-thesis`.

[6] GOMEZ GRAPHICS. “Raster vs vector.” Accessed 2024-05-12. (2024), [Online]. Available: `https://vector-conversions.com/vectorizing/raster_vs_vector.html`.

[7] J. D. Foley, *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1995, p. 13.

[8] R. Descartes and P. J. Olscamp, “Discourse on method, optics, geometry, and meteorology,” 1965. [Online]. Available: `https://api.semanticscholar.org/CorpusID:60649903`.

[9] B. Lutkevich. “Vector graphics.” Accessed 2024-05-11. (2021), [Online]. Available: `https://www.techtarget.com/whatis/definition/vector-graphics`.

[10] T. Bachmann, *Perception of pixelated images*. Academic Press, 2016.

[11] Yug, modifications by Cfaerber et al. “Demonstration of differences between bitmap and svg images.” Accessed 2024-05-14. (2006), [Online]. Available: https://commons.wikimedia.org/wiki/File:Bitmap_VS_SVG.svg.

[12] S. E. Battersby, M. P. Finn, E. L. Usery, and K. H. Yamamoto, “Implications of web mercator and its use in online mapping,” *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 49, no. 2, pp. 85–101, 2014.

[13] J. Kang. “An interactive explanation of quadtrees.” Accessed 2024-04-11. (2014), [Online]. Available: https://jimkang.com/quadtreevis/.

[14] MapTiler. “Coordinates, tile bounds and projection.” Accessed 2024-04-22. (2024), [Online]. Available: https://www.maptiler.com/google-maps-coordinates-tile-bounds-projection.

[15] MapTiler. “Gl style specification.” Accessed 2024-01-11. (2024), [Online]. Available: https://docs.maptiler.com/gl-style-specification/.

[16] V. Agafonkin, J. Firebaugh, E. Fischer, K. Käfer, C. Loyd, T. MacWright, A. Pavlenko, D. Springmeyer, and B. Thompson. “MapBox Vector Tile Specification v1.0.0.” Accessed 2024-01-11, MapBox. (2014), [Online]. Available: https://github.com/mapbox/vector-tile-spec/blob/master/2.1/README.md.

[17] *Protocol buffers documentation*, Accessed 2024-02-06. [Online]. Available: https://protobuf.dev.

[18] *Encoding*, Accessed 2024-04-23. [Online]. Available: https://protobuf.dev/programming-guides/encoding/#signed-ints.

[19] “Maptiler planet schema.” Accessed 2024-04-23. (2024), [Online]. Available: https://docs.maptiler.com/schema/planet/.

[20] The Qt Group. “QPainter class.” Accessed 2024-01-11. (2024), [Online]. Available: https://doc.qt.io/qt-6/qpainter.html.

[21] Massachusetts Institute of Technology. “MIT license.” Accessed 2024-01-11. (2024), [Online]. Available: https://mit-license.org/.

[22] I. Sommerville, *SOFTWARE ENGINEERING*, 10th ed. Pearson Education, 2016.

[23] MapTiler. “GL Style Specification, Expressions.” Accessed 2024-01-11. (2024), [Online]. Available: https://docs.maptiler.com/gl-style-specification/expressions/.

[24] The Qt Group. “QThreadPool class.” Accessed 2024-04-02. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qthreadpool.html`.

[25] R. Kriemann, “Implementation and usage of a thread pool based on posix threads,” *Max-Planck-Institue for Mathematics in the Sciences, Inselstr*, pp. 22–26, 2004.

[26] The Qt Group. “QByteArray class.” Accessed 2024-04-24. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qbytearray.html`.

[27] R. H. Netzer and B. P. Miller, “What are race conditions? some issues and formalizations,” *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[28] B. B. Brandenburg and J. H. Anderson, “Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks,” in *Proceedings of the ninth ACM international conference on Embedded software*, 2011, pp. 69–78.

[29] The Qt Group. “QPainter class.” Accessed 2024-04-24. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qpainter.html#drawPath`.

[30] The Qt Group. “QPainter class.” Accessed 2024-04-24. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qpainter.html#drawText`.

[31] The Qt Group. “QPainterPath class.” Accessed 2024-01-11. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qpainterpath.html`.

[32] The Qt Group. “QPainterPath class.” Accessed 2024-04-24. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qpainterpath.html#addText`.

[33] The Qt Group. “QTextLayout class.” Accessed 2024-04-24. (2024), [Online]. Available: `https://doc.qt.io/qt-6/qtextlayout.html#draw`.

[34] C. Solutions. “5 benefits of unit testing and why you should care.” Accessed 2024-05-21. (2022), [Online]. Available: `https://cswsolutions.com/blog/posts/2022/december/5-benefits-of-unit-testing-and-why-you-should-care/`.

[35] I. B. Machines. “Benchmark testing.” Accessed 2024-05-14. (2021), [Online]. Available: `https://www.ibm.com/docs/en/db2/10.5?topic=methodology-benchmark-testing`.

[36] GitHub. “GitHub Actions.” Accessed 2024-03-25. (2024), [Online]. Available: `https://github.com/features/actions`.

[37] Docker. “Docker.” Accessed 2024-03-25. (2024), [Online]. Available: https://www.docker.com/why-docker/.

[38] Y. H. Yee and A. Newman, “A perceptual metric for production testing,” in *ACM SIGGRAPH 2004 Sketches*, 2004, p. 121.

[39] ImageMagick Studio LLC. “ImageMagick: Create, Edit, Compose, or Convert Digital Images.” Accessed 2024-04-12. (2024), [Online]. Available: https://imagemagick.org/index.php.

[40] G. for Geeks. “Acceptance testing – software testing.” Accessed 2024-05-18. (2024), [Online]. Available: https://www.geeksforgeeks.org/acceptance-testing-software-testing/.

[41] T. technologies. “Acceptance Testing : What, Why, Types & How to Do?” Accessed 2024-05-13. (2024), [Online]. Available: https://testsigma.com/guides/acceptance-testing/.

[42] A. Alliance. “What is acceptance testing?” Accessed 2024-05-18. (2024), [Online]. Available: https://www.agilealliance.org/glossary/acceptance-testing/.

[43] A. S. Gillis. “Acceptance testing.” Accessed 2024-05-18. (2024), [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test.

[44] United Nations. “The 17 goals.” Accessed 2024-04-24. (2024), [Online]. Available: https://sdgs.un.org/goals.

[45] The Qt Group. “QRhi class.” Accessed 2024-01-11. (2024), [Online]. Available: https://doc.qt.io/qt-6/qrhi.html.

[46] P. O’Malley. “Write acceptance tests.” Accessed 2024-05-13. (), [Online]. Available: https://openclassrooms.com/en/courses/4544621-learn-about-agile-project-management-and-scrum/5081076-write-acceptance-tests.

Appendix A

Original Project Plan

Note that the original project plan had two appendices that themselves were called appendix A and B. The thesis appendix B will follow after these appendices.

Project Plan

Cecilia Norevik Bratlie, Eimen Oueslati, Nils Petter Skålerud

January 2024

Contents

List of Figures

List of Tables

1 Goals and Restrictions

1.1 Background

The Qt Group (product owner) is a software company whose primary product is the Qt framework for C++. This framework primarily provides development tools to make applications or graphical user interfaces (GUI). Including map functionality is essential for many types of applications, and this is something the Qt Group would like to have developed further.

Qt has some existing solutions for map functionality, but these are based on raster-based images and maps. The raster image format, also known as bitmap images, are pixel-based images. Raster images are constructed from pixels, small dots that hold distinct color and tonal details (University of Michigan (2023)). Qt wants us to develop software using vector data to render and display maps. Vector maps are formed by calculating and drawing paths between points using mathematical algorithms rather than pixel values.

Previously, Qt used an open-source version of *MapBox v1* (Agafonkin et al. (2014)), to develop its map software. When MapBox version 2 was released, it was closed-source, and Qt needed an alternative to further develop its map-oriented functionalities. An alternative to MapBox is *MapTiler*, and this is the one that will be used for the project. MapTiler provides free APIs that can serve different maps, map types, map tiles, and represent different kinds of terrain, buildings, and landmarks on a map. MapTiler supplies map-data, divided into tiles, in the MapBox Vector Tile v1 format.

Qt already has access to software to generate and process raster-based maps. They now want us to create a proof-of-concept on using Qt framework for creating vector-based map rendering and functionality.

1.2 Project Goals

1.2.1 Result Goals

- Create a program or programs that can be used and developed further by ourselves and others in the future.
- Develop a good project to show future employers.
- Achieve a grade of B or higher.

1.2.2 Effect Goals

- Implement a proof of concept that Qt's current tools can create vector-based map rendering. This includes:
  - Downloading map style sheets and vector-tile data using HTTP requests.
  - Cache map vector tile data.
  - Parsing styling information from the MapTiler stylesheet, such as fill-color, line-widths etc.
  - Implementing map panning.
  - Implementing map zooming.
- Implement vector map rendering functionality with the *QPainter* tool (The Qt Group (2024a)).
- The final source code should be usable to the Qt Group for future development.
- Include good documentation that follows C++ standards, Qt conventions, and our internal guidelines to facilitate continued development after project completion.

1.2.3 Learning Goals

- Learn more about how to complete a project in a professional environment.
- Learn how to use the Qt framework when developing C++ applications.
- Achieve a practical understanding of how to develop application-tools with map-functionality.
- Learn more about how to properly apply Scrum as a development framework.
- Learn how to write an academic report as a larger team based on developing the app.

1.3 Restrictions

Qt has informed us that we may shape our thesis to a large extent as long as we focus on rendering of vector-based map data in the MapBox tile format. We are free to use any programming language or framework, but Qt would like to continue development of the software after the project is completed. Since Qt primarily uses C++ with the Qt framework, we would like to use them as well to ease Qt's future development. In light of this, our team and Qt have collaborated to formulate the following restrictions for the project:

- The solution must be coded in the C++17 programming language.
- Use the Qt framework and tools actively in the software.

- Use software-based rendering with QPainter (The Qt Group, 2024a).
- Render a basic map that includes roads, basic terrain, and landmarks. These are represented by lines, fill, and symbols in the map style sheets.
- The source map data must based on the MapBox vector tile data-format (Agafonkin et al., 2014), supplied by the MapTiler service.
- The source data and displayed map need to be in vector-format, not relying on using raster images.

1.4 License

The project will be licensed under the MIT license (Massachusetts Institute of Technology, 2024). Qt has given us some freedom in terms of choosing license for our project, but we were strongly recommended to use MIT. This allows Qt to do what they want with the final software, while letting the group retain ownership. If Qt or anyone else will keep using or developing the software, they must credit the group as the original authors.

2 Scope

2.1 Problem domain

Relevant technologies
The problem and solution will include, but is not limited to, the following technologies:

- C++17 programming language
- Qt6 framework (The Qt Group (2024b))
- MapBox vector tile data specification (Agafonkin et al. (2014))
- Vector-based rendering
- Networking: HTTP, Protobuf, JSON
- Unit and component testing in a C++ context

2.2 Delimitation

End User Experience
The developed sofware will serve as a proof-of-concept map application for Qt. As such, the end user experience is not a primary concern for the project. As such we will not conduct end user tests.

Runtime Performance
The emphasis of this project does not lie in achieving optimal runtime performance. Qt has clarified that they expect the performance of the software to be poor based on the given restrictions. The application will use software-based rendering[1], utilizing QPainter, not hardware-accelerated rendering.

As such, the project will selectively adopt advanced optimization techniques, only employing it when strictly necessary for the usability of the application. These kinds of optimizations include multi-threading and custom allocation strategies. The rationale for this decision is to avoid complicating both code and architecture, as these kinds of optimization may often lead to higher risk of race conditions and use-after-free bugs. For this project, the benefit of increased performance does not justify the added complexity.

Nevertheless, the project will have to include concurrency and caching in some limited form, as described in subsection 2.3.

[1]Software-based rendering involves executing drawing algorithms on the CPU rather than leveraging GPU capabilities, which, while simpler to integrate, typically yields suboptimal runtime performance.

QML
Integration of the Qt Modeling Language (QML) will not be a component of this project. While QML represents a forward-looking focus of Qt to streamline application development processes, its integration complexity is too demanding for this project.

Map Features
Our project will focus on functionality that is useful to users in urban areas, as elaborated in subsection 2.3. Consequently, we will not implement nautical charts or topographic or geological maps, as these maps contain and display data differently than a typical street map.

Data Sources
The finished project will read map and tile data downloaded from MapTiler, supplied in the Mapbox version 1 data format.

Platform Compatibility
While Qt is cross-platform and should theoretically build and run for most platforms with little effort, our project will focus solely on Windows x86 and Android-based mobile platforms. This is due to the groups experience with said platforms and also lack of devices that run Linux, MacOS, or iOS.

2.3 Problem Statement

We will make a proof-of-concept application that renders maps based on vector data.

The following project description will focus on adhering to the restrictions (see subsection 1.3). Additionally, a few other features have been added based on what the group is enthusiastic to develop, or based on what we believe will strengthen our thesis.

Core functionality
The final solution will heavily prioritize the primary functionality: Loading and rendering maps.

Loading maps
The application will load map-data supplied from MapTiler. The map-data will be in the MapBox vector tile data format. The loading will be done in an asynchronous, concurrent manner relative to user interaction and rendering logic, in order to allow smooth operation of the application while new tiles are being loaded.

The rationale for making tile loading asynchronous is based on our expectations for the speed at which tiles can be downloaded and processed. In the case of loading taking time in a synchronous context, the user interaction will halt until all the tiles are processed. We expect this to result in severe stuttering during the user interactions.

The asynchronous loading has certain effects on the user experience, such as tiles appearing on the display in a deferred manner. An example includes the user panning the view, the map is panned smoothly but loaded tiles will appear over time.

In order to limit how many HTTP requests we make to MapTiler, a caching of the loaded map-tiles will be implemented. This caching will ensure that previously loaded tiles are stored persistently and loaded from the device the application is running on.

Rendering of maps

The displayed map will feature elements such as fill-colored background terrain, roads, road names, landmarks (like restaurants and parks), and satellite imagery. The rendering will be able to display multiple map tiles when necessary to fill the screen, and also be able to display any tile at any location at any zoom level. Furthermore, the map-rendering functionality will be able to filter in and out certain elements from the displayed map itself, such as roads, rivers, buildings, background.

Additionally, the rendering will try to adhere to the stylesheet (supplied by MapTiler) of the map as much as possible. This includes fill-colors, outlines and other graphical effects. At the point of writing this project plan, it is hard to determine exactly the complexity of this task. The team expects that fill-colors and outlines will be possible to implement, while each subsequent styling type will be evaluated for implementation during the course of development.

The rendering functionality will be implemented using the QPainter class from the Qt framework.

The displayed map will focus on layers and elements that are useful in a urban navigation use-case.

The displayed map will also be able to switch the background to raster-based satellite imagery.

User Interface

The user interface will be designed to work on both desktop and mobile, with the user interface designed to be mobile-first.

The user interface will include interactivity controls, such as panning and zooming the viewed map. This will be done by having on-screen controls, and also by using the Windows mouse cursor or through touch-input on mobile devices.

The UI will include (but is not limited to) the following functionality:

- Increase and decrease zoom-level.

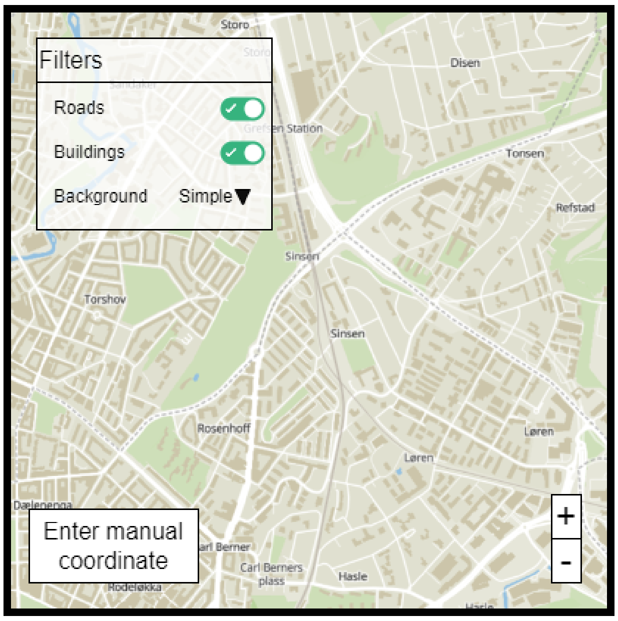


Figure 1: Product mockup

- Pan the view in north, west, east and south directions.
- Enable and disable the rendering of specific layers in the map.
- Manually enter coordinates and zoom level.
- Change background from simple to satellite photo.

Platforms

The application will run on Windows. Though the details of Android deployment is unclear at the time of writing, we expect that it will also run on Android smartphones.

Mockup

A rough mockup of the final product can be found in Figure 1. This figure visualizes what components might exist in the final solution, but does not necessarily represent the final design.

Analysis of continued development

An important aspect of our thesis will be focused on analysing the viability of continued development of our solution. In this analysis, we will be discussing details regarding how useful it is for someone to continue our work after the project is over, and also we will be analysing what parts would have to be modified and what features would be the immediate next step. Such an analysis can provide good value to Qt and also serve as a valuable learning exercise for the group.

This analysis would include topics such as how to transition rendering into hardware-acceleration, and a discussion surrounding how to rewrite parts to conform better to Qts internal code.

2.3.1 Prototype

One big milestone for our project is a prototype. This prototype is estimated to be finished halfway through the development process and will ensure that we implement the most core functionalities of our final project. It is is critical that this prototype is able to load and display a single tile.

The prototype may be include some more functionality if it's useful simple to include and is useful towards the final project requirements.

The prototype will include the following functionality

- Download and parse the style JSON from MapTiler
- Download, parse and process a single tile from MapTiler
- Display the given tile:
    - Display all polygon and line features from the vector tile data
- Location and zoom-level will be hardcoded.

3 Project organization

3.1 Roles and responsibilities

- **The Qt Company, Matthias Rauter:** Product owner.
- **Nils:** Team leader, Scrum master, and Latex/Overleaf supervisor.
- **Eimen:** Network and quality assurance responsible.
- **Cecilia:** Responsible for quality assurance and Git.

3.2 Routines and Group Rules

See appendix A for the project's group contract where routines and group rules are described.

4 Planning, Follow-Up, and Reporting

4.1 Project Management Methodology: Scrum

We will follow the Scrum methodology to structure the workflow. This section describes our Scrum plan (scrum.org (2024)) for the project. Scrum was chosen not only to emulate a professional work environment, but also for its agility in allowing us to adjust our work and workflow when necessary. By default, Scrum is not designed to prioritize documentation. The goal is always to have a working product or program. In our adaptation of Scrum, we have included specific tasks in the product backlog that are focused on writing various sections of the thesis, ensuring that our academic requirements are integrated into the agile workflow. The work will be split across 2-week sprints. Every sprint starts and ends on every other Wednesday. In the middle of these sprints, we will have a sprint review.

4.2 Plan for Status Meetings and Decision Points

4.2.1 Sprint Planning

Sprint planning will happen on every other week, marking the beginning of a sprint. In this meeting we will establish the goals and tasks for the coming sprint. Under sprint planning, the scrum master will prepare a goal for that particular sprint. The team will discuss how much time they have available for this sprint, and what can realistically be done for the sprint. The team will pick tasks from the Product Backlog and divide it up into smaller subtasks. They will then define the criteria for this task being "done" (for example when a task is completed and has been written about in the report), and estimate how much time they will need.

4.2.2 Sprint Review

Sprint reviews happen in the middle of a sprint. This is where the team demonstrates what they've been working on, what challenges they faced and how they solved them. The purpose of this meeting is to adjust the current sprint goals if necessary, and also to fill the entire team on pieces of the project that they haven't directly touched themselves. This helps us stick to our goal of everyone being responsible for every piece of the code. This meeting happens at mid-point of a sprint, one week into the sprint.

4.2.3 Sprint Retrospective

The retrospective happens at the end of a sprint, very close to the sprint planning meeting. It should function as a short summary of what has been done and not been finished. This is where we discuss tasks that need to be extended into an upcoming sprint. We also discuss any problems the team has with the organization of the work, such as re-evaluating the structure of meetings.

5 Quality Assurance

5.1 Routines, Standards, Documentation, Tools

Routines and Standards

Everybody in the group is responsible for ensuring good project quality, but Eimen and Cecilia take extra responsibility for code and text quality assurance. We have agreed that we need to document what we're working on every week in a timesheet and write technical notes when decisions are made. We also agree to follow an internal standard for how to use Git.

Additionally, we will work on the report during the entire semester (and we've already started at the time of handing in the project plan). We will also do our best to follow C++ standards and Qt standards in our code base.

Documentation

The Scrum framework dictates that one shouldn't focus too much on documentation, and it's having a working program or project is the most important when using Scrum. Since the report is the primary way that the project will be assessed, we will prioritize documentation to a great extent. Critical documentation activities are:

- Writing and completing the final report and all other mandatory deliverables for the project.
- Taking notes from all meetings with the team, supervisor, and Qt in a standardised way.
- Writing technical notes and documents when making decisions, researching topics that are relevant to the project, and documenting problems, proposed solutions, and implemented solutions.
- Use documentation comments in source code, like QDoc, which is Qt's documentation comment standard.
- Use regular comments to explain source code.
- Use Git issues and commits do document what was or is to be done when. Commits shall link to applicable issues to make it easy to access relevant content directly from the issues themselves or commit history.
- Document usage of AI tools both in text and code. Larger code snippets will have it marked in the doc comments if and where AI has been used to generate code. It must

be specified what and how AI has been used in all facets of the project (for imagery, figures, tables, coding help, as a learning tool, or writing aid).

Latex

The project plan and final report are written in LaTex, using Overleaf. All the project code is in the Qts GitLab instance and our personal GitHub instance, the latter can be found here: https://github.com/cecilianor/QT-thesis.

Git

Git is used to store project source code files. It is also going to be used to write and back up technical documents that don't warrant their own reports but are still used for the project and/or final report. Git issues are used to organise work, and the issues also make up the project Scrum board. Issues may be split into 'sub-issues' that refer back to the 'parent' issue. This allows us to divide the workload between team members and see who is working on what part of larger issues.

Team Git guidelines have been formalized in a Git Guidelines document that has been appended to the project plan. It covers how the Git issues must be made (content and format), how to write semantic Git commits, and how to format Markdown files on Git.

Planned AI Usage

AI tools like ChatGPT may be used for guidance during the project. It can be used to find inspiration when writing text or code, or it can be used as a teacher to teach us new topics. The team doesn't have to document using AI tools for these purposes along the way, but they should still be mentioned in the "Reflections about AI Usage" section in the final report.

AI may also be used to write or proofread text, generate or comment code, or generating imagery for the project. If AI has been used to do any of these things, it **must** be clearly documented *where*, and *how* it has been used. It must be stated in the text itself or in a comment above the code: 'The following paragraph/code has been generated using ChatGPT.'

Cecilia monitors how AI tools are used and ask members about AI usage at least every 2 sprints and review AI generated content. We are all responsible for not overusing or misusing AI tools. If we think AI must be used to generate *an entire solution*, it may indicate that our skill level is too low, that the project scope is too high, or that we need to approach the problem differently. These issues must be addressed when they come up, and we'll discuss how to handle them during team meetings. The software and report must be made by *us*, AI is only supposed to be a supporting tool.

5.2 Plans for Inspections and Testing

5.2.1 Inspections

The QA heads will inspect text and code multiple times throughout the semester to ensure the project, the process, and the report quality are as good as possible. QAs must notify the team if they think something needs improvement.

5.2.2 Testing

Unit tests and component tests will be performed to uncover mistakes and errors, check that the software behaves as intended, and to enhance overall software quality. Acceptability tests will also be performed with the product owner to ensure Qt gets a satisfactory software.

Maximizing the usability of the end software is not a project goal. Due to the nature of the project being mostly oriented as a proof of concept, this project will not include user tests. Qt has informed us that code coverage is not a major concern either, and we believe this can be too time-consuming to add in addition to other planned test. Therefore, the group will not use code coverage tools.

Unit testing - Data Parsing

Unit tests will be implemented early on in development to test data parsing, one of the first project tasks. The tests will be performed on functions that handle parsing of third-party data. This means validating that MapTiler data is correctly parsed into the intended C++ structures and functions. This category of testing includes taking a 'snapshot' of tile data provided by MapTiler, and basing tests on the snapshot. This also introduces some level of mocking by relying on a local copy of the MapTiler data.

A part of the unit tests will also include validating loaded values, e.g whether the number of roads is correct or that it includes expected style data. An example of this is to compare the number of layers that are parsed and loaded against what is stored in the snapshot. This ensures that parsing executed without errors and that the result is valid.

Component Tests - Comparing Display Output

Component tests will be run on the application's rendering functionality. This entails testing when the rendering functionality is able to output a baseline image that is free of rendering artifacts, and that it contains all the visual elements it should. At this point, we can implement a rendering component test that will run the same data set as the ground truth data and compare the output of newer software versions to the ground truth. The baseline is expected to be re-determined a few times during the course of development as the feature-set changes. This type of test could prove vital to ensuring that the rendering-code does not suffer regressions. For this type of test, Qt's rendering testing tool *Lancelot* can be used.

A major concern with this approach is whether the test results will be reliable. Comparing pixels of a display output is sensitive to minor imperceptible differences. These minor differences might show up as false negatives in test results. To mitigate this, one could experiment with a threshold to allow some differences. Another concern is keeping the baseline up-to-date whenever the functional-requirements change.

Performance Test - Parsing and Rendering (Optional)
Useful performance benchmarks could be to check motion fluidity and the response time needed to display a map. These can be measured in Frames Per Second (FPS) and seconds, respectively. For our project, performance is not a key metric for success and therefore will only happen if there is un-allocated development time near the end of the project.

Benchmarking the response time would include measuring how much time it takes to download, parse, process and display map-tiles. To test the motion fluidity, we would measure how long it takes to render a completely new frame at a position on the map.

Acceptance test with Qt
Near the end of development, we will hold an acceptance test with Qt where we demonstrate our application and take feedback from Qt to map what is missing from our solution. This will include reviewing the code itself as well as the user-facing experience. This will serve as the final point for Qt to submit their feedback to the project.

5.3 Project Level Risk Analysis Overview

This section covers the identified project risks, their likelihood, their consequences level, and their corresponding mitigation strategy. The likelihood of a risk happening is assessed to be either unlikely, likely, or highly likely. The risk levels are assessed to be one of the following:

- **Insignificant:** The risk will not require any changes and will not impact the project much, if at all.
- **Unproblematic:** This risk should not cause any major delays or require large adjustments to the project's plan.
- **Problematic:** If a risk of this level occurs, some adjustments to the project's scope might be necessary. This might cause some delays and might require contacting the thesis supervisor and/or course coordinator.
- **Critical:** Risks of this level will cause major changes to the project plan and ultimately the end product of the project. Robust mitigation strategies must be put in place for

these risks.

The description of the risks and their mitigation strategies are described in Table 1.

Table 1: Risk analysis table

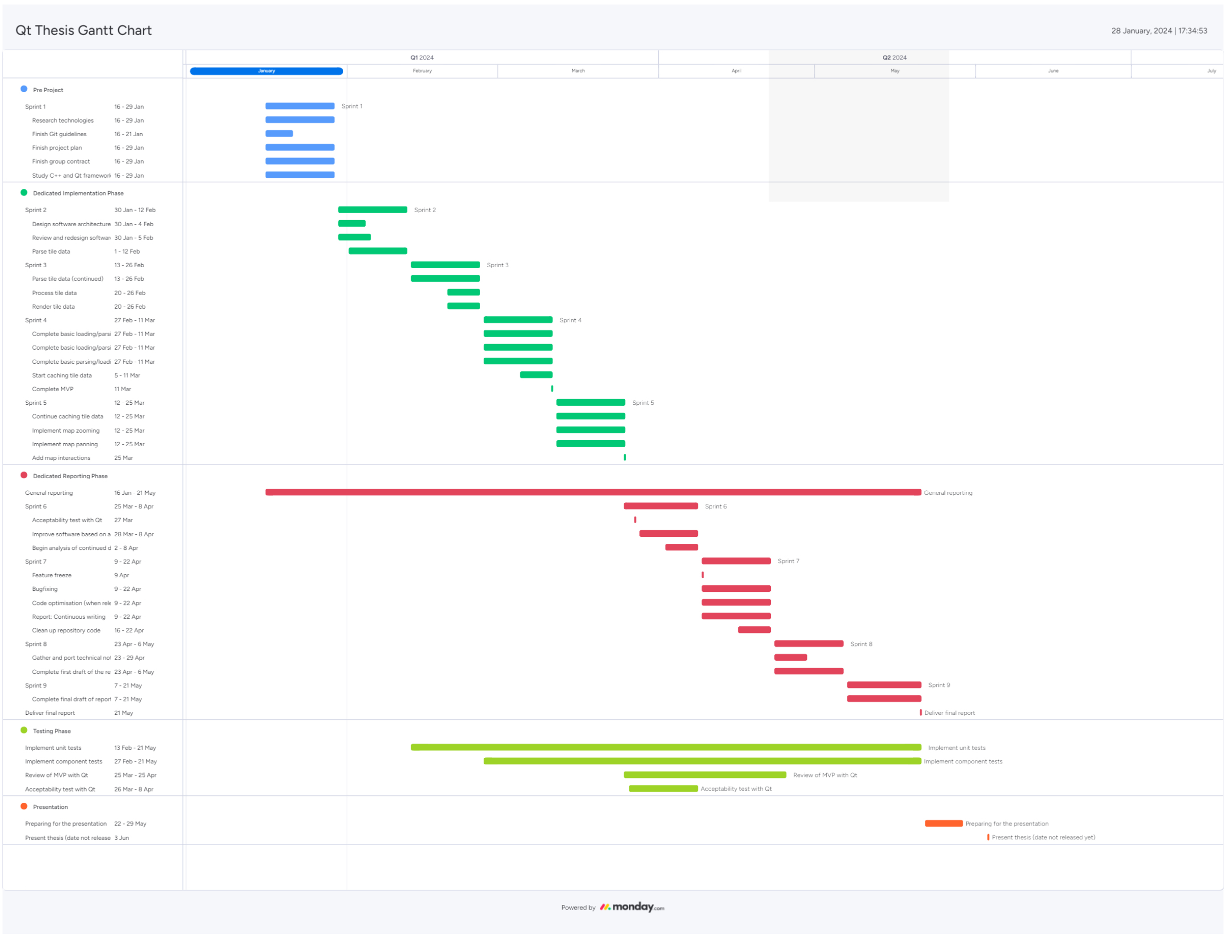
Nr.	Description	Mitigation strategy
1	**Risk:** Team member quits the course **Likelihood:** Unlikely **Consequence:** Critical	Seeing that the occurrence likelihood of this scenario is minuscule, No strict mitigation strategy has been set in place. However, in the worst-case scenario where this scenario happens, the remaining team members should adjust the scope of the project to account for the missing team member.
2	**Risk:** Team member leaves the project for over two weeks **Likelihood:** Unlikely **Consequence:** Problematic	The Supervisor and/or course coordinator will be contacted to plan the workflow going forward. We may have to adjust the project's scope and redistribute the tasks among the remaining team members. If the team member disappears without contact, the measures will apply after a week instead of two.
3	**Risk:** Loss of contact with Qt **Likelihood:** Unlikely **Consequence:** Critical	We will try to reach the contact person at Qt through email or Microsoft Teams. However, as a preventative measure, a clear definition of the project's requirements and scope will be set at the early stages of the project to avoid any major delays or disruptions to the workflow in this scenario. In addition, the thesis supervisor will be contacted to get guidance on how to proceed going forward.

Table 1: Risk analysis table

Nr.	Description	Mitigation strategy
4	**Risk:** Loss of contact with the supervisor **Likelihood:** Unlikely **Consequence:** Problematic	We will try to reach the thesis supervisor by email or Skype. The course coordinator will be contacted to solve any problems concerning communication with the supervisor
5	**Risk:** Too great or narrow project scope **Likelihood:** Likely **Consequence:** Critical	The scope of the project will be adjusted accordingly. A scrum development model has been adopted to quickly address any change in the requirements or scope of the project
6	**Risk:** Inconsistent workflow **Likelihood:** Unlikely **Consequence:** Problematic	A meeting should be held between the team members to address any inconsistencies in the workflow. Having consistent team communication and Scrum meetings with stand-ups, reviews, and retrospectives helps mitigate this risk since we have to communicate and work consistently.
7	**Risk:** Information loss **Likelihood:** Unlikely **Consequence:** Critical	Seeing as this scenario will have a major impact on the project, all project files are being stored in cloud solutions to avoid data loss (GitHub, Overleaf, Google Drive). In the case of local file loss on our end, the specific file is to be pulled from the cloud solution. Cecilia will back up repository code and Overleaf text files weekly on hard-drive as well in case files are lost from any of the cloud solutions.

Table 1: Risk analysis table

Nr.	Description	Mitigation strategy
8	**Risk:** Disagreement amongst group members **Likelihood:** Unlikely **Consequence:** Insignificant - Problematic	Disagreements amongst the group members will be solved during meetings. If no solution is reached, the group leader will have the final say in the matter. We believe the consequences can be insignificant to problematic depending on how long the disagreement persists, and if it can be resolved quickly or not.
9	**Risk:** Team member falls ill **Likelihood:** Highly likely **Consequence:** Unproblematic	Seeing as we have no control over this particular scenario and its high occurrence likelihood, we may redistribute the workload for that particular week.



● Pre Project	
Sprint 1	16 - 29 Jan
Research technologies	16 - 29 Jan
Finish Git guidelines	16 - 21 Jan
Finish project plan	16 - 29 Jan
Finish group contract	16 - 29 Jan
Study C++ and Qt framework	16 - 29 Jan
● Dedicated Implementation Phase	
Sprint 2	30 Jan - 12 Feb
Design software architecture	30 Jan - 4 Feb
Review and redesign softwar	30 Jan - 5 Feb
Parse tile data	1 - 12 Feb
Sprint 3	13 - 26 Feb
Parse tile data (continued)	13 - 26 Feb
Process tile data	20 - 26 Feb
Render tile data	20 - 26 Feb
Sprint 4	27 Feb - 11 Mar
Complete basic loading/parsi	27 Feb - 11 Mar
Complete basic loading/parsi	27 Feb - 11 Mar
Complete basic parsing/loadi	27 Feb - 11 Mar
Start caching tile data	5 - 11 Mar
Complete MVP	11 Mar
Sprint 5	12 - 25 Mar
Continue caching tile data	12 - 25 Mar
Implement map zooming	12 - 25 Mar
Implement map panning	12 - 25 Mar
Add map interactions	25 Mar
● Dedicated Reporting Phase	
General reporting	16 Jan - 21 May
Sprint 6	25 Mar - 8 Apr
Acceptability test with Qt	27 Mar
Improve software based on a	28 Mar - 8 Apr
Begin analysis of continued d	2 - 8 Apr
Sprint 7	9 - 22 Apr
Feature freeze	9 Apr
Bugfixing	9 - 22 Apr
Code optimisation (when rele	9 - 22 Apr
Report: Continuous writing	9 - 22 Apr
Clean up repository code	16 - 22 Apr
Sprint 8	23 Apr - 6 May
Gather and port technical no	23 - 29 Apr
Complete first draft of the re	23 Apr - 6 May
Sprint 9	7 - 21 May
Complete final draft of report	7 - 21 May
Deliver final report	21 May
● Testing Phase	
Implement unit tests	13 Feb - 21 May
Implement component tests	27 Feb - 21 May
Review of MVP with Qt	25 Mar - 25 Apr
Acceptability test with Qt	26 Mar - 8 Apr
● Presentation	
Preparing for the presentation	22 - 29 May
Present thesis (date not release	3 Jun

Figure 2: Project Gantt Chart

6 Plan for Execution

6.1 Gantt Chart

The Gantt chart for the project is described in Figure 2.

6.2 Milestones

For this project we have prepared a few milestones to ensure that the groups work stays focused and directed as much as possible. These milestones, as well as their deadlines are listed in Table 2.

Table 2: Milestones

Date	Milestone
March 10th	Minimum viable prototype (MVP)
April 21st	Feature freeze
May 5th	Complete code freeze
May 21st	Delivery of final report

6.3 Sprint schedule

Each sprint will have a clearly defined goal (as described in in subsection 4.2). The list of sprints for the project duration, as well as planned sprint goals can be found in Table 3.

6.4 Product backlog

The product backlog (Table 4) is the basis for the Scrum board, and it's used and updated during sprint planning. The product backlog will be updated throughout development as more details of the implementation becomes clearer.

The team has decided that tasks in the product backlog aren't complete until they've been written about in a technical note or the final report. The text can be a draft and doesn't have to make it into the final report, but it must describe what, how, and why something was done the way it was. If the writer isn't sure if a task is complete, it's marked as Ready-for-review in the issue board, and one (or more) team members will review the work.

Table 4 contains the initial product backlog with rough estimates of the time required to finish tasks. A 'low' estimate implies up to 2 days of work, 'medium' implies between 2-4 days of work, and a 'high' estimate implies more than 4 days of work. The list is in loose chronological order, but tasks can be done in parallel.

Table 3: Sprint schedule

Sprint	**Week**	**Goals**
1	3-4	Get acquainted with Qt framework and complete project plan.
2	5-6	Design overall software architecture. Load, parse, process, display (initial) tile data.
3	7-8	Load, parse, process, display tile data (continued). Start implementing unit tests and component tests when functionality is ready.
4	9-10	Complete MVP. Automate unit tests for tile data processing. Continue updating unit and component tests. Start data caching.
5	11-12	Implement map interactivity: panning and zooming. Load multiple tiles. Continue with data caching.
6	13-14	Software improvement based on acceptability test. Start of dedicated reporting phase. Begin analysis of continued development.
7	15-16	Feature freeze. Fix bugs, optimise code, clean up repository.
8	17-18	Gather all technical notes and work on final report. Fix bugs.
9	19-20	Finish the final report.

Table 4: Product backlog

Task	**Estimated time spent**
Deliver project plan	High
Begin final report	Low
Design initial UI	Medium
Design overall architecture	High
Domain Model diagram	Low
Architecture diagram	Low
Data flow diagram	Medium
Setup initial development environment	Low
Parse single map tile	High
Simple HTTP commands	Low
Parse style JSON	Low
Parse tile PBF	Low
Generate layer vertices	Low
Create initial unit tests	Medium
Downloading map-data	Low
Parsing data-structures	Low
Validating parsed data	Low
Set up automated testing environment	High
Automate unit tests	Medium
Automate component tests	Medium
Acceptability tests with Qt	Medium
Test if initial software architecture is acceptable	Low
Test if initial GUI is acceptable	Low
Test if prototype is acceptable	Low
Test if final software is acceptable	Low
Render single tile	Medium
Output polygons with QPainter	Low
Finish prototype	High
Component testing: Rendering	High
Loading multiple tiles	Medium
Caching tile HTTP responses	Medium
Rendering: Panning/zooming support	Medium
Dynamically loading tiles	Medium

Task	**Estimated time spent**
Rendering: Satellite imagery	Medium
Interactive controls	Medium
Panning	Low
Zooming	Low
Tracking user-location (Optional)	Low
Report: Analysis of continued development	High
Finish report	High

References

Agafonkin, V., Firebaugh, J., Fischer, E., Käfer, K., Loyd, C., MacWright, T., Pavlenko, A., Springmeyer, D., & Thompson, B. (2014). *MapBox Vector Tile Specification v1.0.0* [Accessed 2024-01-11]. MapBox. https://github.com/mapbox/vector-tile-spec/tree/c9be969f24c4e8d8f3ea13b232757b31184d2f13/1.0.0

Massachusetts Institute of Technology. (2024). MIT license [Accessed 2024-01-11]. https://mit-license.org/

scrum.org. (2024). What is scrum? Retrieved January 16, 2024, from https://www.scrum.org/learning-series/what-is-scrum/

The Qt Group. (2024a). QPainter class [Accessed 2024-01-11]. https://doc.qt.io/qt-6/qpainter.html

The Qt Group. (2024b). Qt6 reference pages [Accessed 2024-01-11]. https://doc.qt.io/qt-6/reference-overview.html

University of Michigan. (2023). Raster images [Accessed 2024-02-01]. https://guides.lib.umich.edu/c.php?g=282942&p=1885352

A Group Contract

Bachelor Qt - Group Contract

Nils Petter Skålerud, Eimen Oueslati, Cecilia Norevik Bratlie

January 2024

1 Roles and Responsibilities

- Group Leader: Nils Petter Skålerud
  - Make sure everyone works the agreed-upon amount of time.
  - Make sure everyone works on and completes their assigned work. If there is a problem or someone needs help with completing a task, the team members will help each other complete it. If there is a problem with this, the group leader has the final say on what to do and who is going to complete the task.
  - Manage the issue-board and make sure it stays correct and up-to-date.
- Communication Responsible: Nils Petter Skålerud
  - Issue meeting summons and propose and write meeting agendas.
  - Schedule meetings with the thesis supervisor.
  - Schedule meetings with Qt.
  - Responsible for communication with Qt regarding project specifications, design, and implementation.
- Quality Assurance Responsible: Cecilia Norevik Bratlie and Eimen Oueslati
  - Proofread documents to ensure they are free of typographical errors.
  - Monitor the quality of code and ensure that best practices and coding conventions are being followed.
  - Ensure that all the diagrams, charts, and figures are correct (with labels and titles as well).
  - Manage file directories and files to make it simple and intuitive to navigate to and between them.
- Documentation Responsible: Everyone

1

A Group Contract

  - Chech that code is properly commented before merging it to the main branch.
  - Make sure all the required documents exist.
  - Ensure that documents contain required content.
  - Make sure that all the documents are correctly structured and follow internal standards and guidelines.
  - Ensure that Overleaf documents and source code on Git is backed up on an external hard-drive once a week.

- Secretary: Everyone (on rotation)
  - Write reports of internal and external meetings.
  - Take notes during meetings.

2 Expectations of the Workflow

- Group meetings should be short, preferably 30 minutes or less.
- Although each member is assigned to work on certain issues or tasks, the workflow is agile. This means we re-assess how the work is going, and tasks can be re-assigned to other team members later.
- Aim for a B or higher as the final grade.
- Working 30-35 hours work per week is expected, but everyone must work at least 20 hours every week.
- Everyone has at least one subject parallel to the bachelor assignment, so we will prioritise attending classes for those subjects. Some of us may occasionally miss the weekly required 20 hours of work because of that. This is okay when the team is notified about it, and that we make up for the missing work later.
- Everyone must document their work hours in the time schedule
- If someone is sick for a full week, they are allowed to write down 30 hours in the time schedule.
- If a member is sick, they are not expected to work until they feel better. If someone is gone for two weeks or more, the project supervisor will be notified.
- Everyone is expected to show up to meetings at least once a week.
- If a member is not able to attend a meeting, they should notify the group at least 24 hours in advance.

A Group Contract

- A member is expected to give notice to the group if they will be more than five minutes late to meetings.
- The group will have slots in the schedule for the group to work on things simultaneously (online or physical), each member is expected to participate in the majority of these.

If a team member fails to meet expectations, they will be reported to the supervisor. The group will discuss what to do from there. Repercussions can include getting dropped from the project.

Minor deviations from the expectations are tolerated.

3 Distribution of the Workload

- Every team member is expected to do roughly an equal amount of work. Cecilia and Nils are expexted to do a bit more than Eimen since their thesis is worth 22.5 ECTs, while Eimen's thesis is worth 20 ECTs.
- The one who has implemented a feature, is assigned to writing about it in the report unless the team agrees to assign the work to someone else.

4 Member Contact Information

- Name: Nils Petter Skålerud
  E-mail: nilspska@stud.ntnu.no
  Phone: +47 948 17 868
  Student ID: 510627
- Name: Cecilia Norevik Bratlie
  E-mail 1 : cecilinb@stud.ntnu.no (primary)
  E-mail 2 : cecilia_nor@hotmail.com (secondary)
  Phone: +47 412 67 982
  Student ID: 562936
- Name: Eimen Oueslati
  E-mail: eimeno@stud.ntnu.no
  Phone: +47 412 52 604
  Student ID: 559465

A Group Contract

We hereby confirm that we have all participated actively in writing the group contract, we understand its contents, and we will follow it during the entire duration of the project.

Gjøvik, January 29, 2024

[illegible]	[illegible]	[illegible]
Nils Petter Skålerud	Eimen Oueslati	Cecilia Norevik Bratlie

4

B Git Guidelines

Git Guidelines

Cecilia Norevik Bratlie

January 2024

Contents

B Git Guidelines

1 Introduction

The following document describes how Git will be used during the QT bachelor thesis project. The goals of this document are to:

- Streamline how the Git workspace is used and developed by all team members.
- Improve the overall quality of the project by making commits as consistent as possible between team members.
- Use Git in a professional way by following good practises for Git commits and Markdown text.
- Make it easier to understand developed code at a glance, without having to read the source code itself (at least initially). If something is unclear or wrong with the code, good commits can help locate errors, and roll back to the correct previous version if necessary.
- Document commits in a semantic way. This means using tags like ‘feature‘, ‘fix‘ or others, and providing a commit description.
- Make it easier for QT to take over development after the thesis is done, since they can go back in the logs and see how the system was developed.

B Git Guidelines

2 Git Guidelines

2.1 Git Issues

Git issues must be linked to the project's Scrum board. Give the issue a descriptive title, and add additional information in the comment section. An issue called "Complete Project Plan" without any further description is incomplete and must be updated. Some textual description or (check)list of tasks specifies what needs to be done to complete the issue. Issues can contain a larger set of tasks that are then assigned to smaller 'sub' or 'child' issues. The 'children' must refer back to the original 'parent' issue. This is done to ensure that once the child issues are complete, the parent issue is updated accordingly. It's also good practise to refer to related issues.

Example from Issue #5
The following issue has a title and a description of the work, and it is acceptable during the project. Note how it refers back to a parent issue at the bottom:

```
The Gantt chart must be created to hand in as a part of the
project plan deliverable. It must contain project specific
activities with milestones and deadlines for when project
decisions must be made.

Updates #3 once completed.
```

2.2 Git Commit Message Structure

A Git commit message should be structured as follows (Commits.org, n.d., Seguin and Hevery, 2019):

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

The type is the keyword describing the commit in a semantic way. The two most important 'type' keywords are 'fix' and 'feat', but others also exist (see Seguin and Hevery, 2019). The type and subject must be provided on the first line.

3

B Git Guidelines

For the sake of consistency in the team, leave the subject line non-capitalized. There is no period (.) at the end of the subject line or footer lines, but the body line has that. The subject line must not have more than 50-70 characters, and it's written in the imperative tense (fix), not the present (fix/fixes/fixing) or past tense (fixed).

The (<scope>) is optional and can provide additional contextual information about the item that's being updated, like an element, language, parser, and so on. The 'scope' always refers to a noun.

The <body> describes the commit content in further detail and is optional to use. Note that further paragraphs in the body start after a blank line.

Finally, you can add a <footer> that starts on a new line. The footer refers to issues if the commit updates or is related on an issue. Footers have their own keywords, and the issues are referred to with # followed by the issue number (like #2).

B Git Guidelines

2.2.1 Commit Examples

Example of a non-semantic commit:

```
updated bug...
```

This kind of commit should not be written and pushed to Git. It's lacking a keyword and doesn't describe clearly communicate what the commit actually updates or fixes, or if it's related to an issue.

Example of a semantic commit (retrieved from Commits.org, n.d.):

```
fix: prevent racing of requests

Introduce a request id and a reference to latest request.
Dismiss incoming responses other than from latest request.

Remove timeouts which were used to mitigate the racing
issue but are obsolete now.

Reviewed-by: Z
Refs: #123
```

This commit uses semantic keywords and descriptions. It's clear what the commit changes, and the commit refers to an issue. Note that the 'Reviewed-by' section in the footer is optional for us. Only use it if someone else reviews your commit before it's committed.

2.3 Subject Line Keywords Descriptions

The following descriptions are based on Commits.org, n.d. and Seguin and Hevery, 2019. The 'type' keywords to use in the subject line are:

- **fix**: The commit fixes a bug in the code.
- **feat**: The commit adds a new feature to the code.
- **docs**: Updates to written documentation, including fixing typos in text or code.
- **style**: Changes to the *styling of code*, like adding or removing white-spaces, adding semicolons or colons, or reformatting the code.

B Git Guidelines

- **build**: Changes to the build system or external dependencies, like if you change or add dependencies to a project.
- **ci**: Used if the commit changes CI configuration files and scripts.
- **refactor**: Changes to code that don't add features or fix bugs.
- **perf**: Code edits that increase performance.
- **test**: Adding or updating tests.
- **chore**: Used for changes where other keywords don't apply.

2.4 Footer Keywords Descriptions

In the footer, one refers to issue numbers. Git accepts the following keywords in the footer (Collins, 2022):

- Close, Closes, Closed: Closes an issue.
- Fix, Fixes, Fixed: Fixes an issue.
- Resolve, Resolves, Resolved: Resolves an issue, so that the issue can now be reviewed before closing it.

Git will update issues in the Git system when these keywords are used, meaning that commits can update, resolve or close issues directly. Prefer using the imperative tense here.

2.5 Markdown Guidelines for GitHub and GitLab Wiki

When writing documents on Git, headers should be used. Headers start with #, ##, ..., ###### followed by a space and mark headers from level <h1> to <h6>. Emphasis is marked with *italics*, strong emphasis should be marked with **bold**. Refer to the following document or talk to Cecilia if something is unclear.

B Git Guidelines

References

Collins, M. (2022). *Conventional commits: A better way.* Retrieved January 16, 2024, from https://medium.com/neudesic-innovation/conventional-commits-a-better-way-78d6785c2e08

Commits.org, C. (n.d.). *Conventional commits.* Retrieved January 16, 2024, from https://www.conventionalcommits.org/en/v1.0.0/

Seguin, A., & Hevery, M. (2019). *Contributing to angular.* Retrieved January 16, 2024, from https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#-commit-message-guidelines

Appendix B

Product Backlog

Table B.1: Product backlog

Task	**Estimated time spent**
Deliver project plan	High
Begin final report	Low
Design initial UI	Medium
Design overall architecture	High
• Domain Model diagram	Low
• Architecture diagram	Low
• Data flow diagram	Medium
Setup initial development environment	Low
• Setting up Qt Protobuf module for Windows	Low
Parse single map tile	High
• Simple HTTP commands	Low
• Parse style JSON	Low
• Parse tile PBF	Low
• Produce QPainterPath from layer data	Low
Render single tile	Medium
• Output polygons with QPainter	Low
• Include fill-color from stylesheet	Low
Finish prototype	High

Task	**Estimated time spent**
• Combine functionalities into one branch	Medium
Create initial unit tests	Medium
• Downloading map-data	Low
• Parsing data-structures	Low
• Validating parsed data	Low
• Component testing: Rendering with Lancelot	High
Set up automated testing environment	High
• Automate unit tests	Medium
• Automate component tests	Medium
Acceptability tests with Qt	Medium
• Test if initial software architecture is acceptable	Low
• Test if initial GUI is acceptable	Low
• Test if prototype is acceptable	Low
• Test if final software is acceptable	Low
Loading multiple tiles	Medium
• Caching tile HTTP responses	Medium
Rendering: Panning/zooming support	Medium
• Dynamically loading tiles	Medium
• Displaying multiple tiles at correct size/position	Medium
Rendering: Satellite imagery	Medium
Interactive controls	Medium
• Panning	Low
• Zooming	Low
Tracking user-location (Optional)	Low
Report: Analysis of continued development	High
Finish report	High

Appendix C

Sprint Schedule

Table C.1: Sprint schedule

Sprint	Week	Goals
1	3-4	Get acquainted with Qt framework and complete project plan.
2	5-6	Design overall software architecture. Load, parse, process, display (initial) tile data.
3	7-8	Load, parse, process, display tile data (continued). Start implementing unit tests and component tests when functionality is ready.
4	9-10	Complete MVP. Automate unit tests for tile data processing. Continue updating unit and component tests. Start data caching.
5	11-12	Text rendering. Implement PNG comparison. Bugfixes. Build instructions and testing continued.
6	13-14	Software improvement based on acceptability test. Start of dedicated reporting phase. Begin analysis of continued development.
7	15-16	Feature freeze. Fix bugs, optimise code, clean up repository. Continued work on report.
8	17-18	Gather all technical notes and work on final report. Fix bugs.
9	19-20	Finish the final report.

Appendix D

Meeting Examples

<> Meeting

Date:	Month, Day, 2024
Time:	00:00-00:00
Present:	Eimen, Nils, Cecilia, (+others)
Secretary:	

Agenda

-

Notes

Topic 1

Topic 2

Standup Meeting

Date:	April 2nd, 2024
Time:	10:00-10:30
Present:	Nils, Cecilia, Eimen
Secretary:	Nils

Agenda

- Standup
- Other discussion points

Standup

Nils:

- Has worked on:
    - Raster tiles
        - Had issues merging commits from dev when trying to create PR to dev-branch. Ended up cherry picking files and making a manual commit to dev-branch.
        - Made sure to keep load-rastertiles branch intact, but it might need to be rollbacked in case some functionality was lost.
    - Would like to merge the text rendering as well, but that has to be sorted later.
    - Loaded MapTiler key from environment variable

- Will work on:
    - Lancelot testing
    - Text rendering merging?
    - Nils can write a benchmark that tests the performance differences between running one background thread versus many for parsing. Cecilia: Would be really good to do this.

Cecilia:

- Has worked on:
    - Raster tiles
    - Has implemented unit testing for the rendering maths:
        - Nils: Will need to double check that this was merged into the dev-branch
        - Had some problems with losing thread-safety and getting thread-errors when trying to refactor lambda functions and gave up temporarily.
    - Been working a lot on the report:
        - NTNU has a template that they highly recommend we use
    - Got feedback from Rafael about unit tests
    - Question: Will we have symbols in our app?
- Will work on:
    - Help Nils and Eimen

        - Port report to the new template
        - Text Matthias about acceptance test next Tuesday

Eimen:

- Has worked on:
    - Cleaned up the text branch
- Will work on:
    - Text rendering

Regarding symbols

We picked a map type that doesn't have symbols! Eimen confirms it takes a lot of time to parse expressions. Nils thinks we don't have time. We initially wanted this feature, but didn't have time. We can write this down as a criticism in the report. Look into how it works, and write about how we would approach it. Suggest for future work?

It is also problematic that we would have to support a new map-style in order to properly test symbols.

We can also mention that symbols would require changes to how we cache.

Immediate tasks

- Check that Cecilia's rendering and unit test changes were merged into dev-branch
- Merge place-names

Supervision Meeting

Date:	April 12th, 2024
Time:	10:00-10:30
Present:	Nils, Eimen, Cecilia, Rafael
Secretary:	Everyone

Agenda

- Progress update
- Acceptance
- References in report to specific Qt classes
- Questions from team members

Notes

01 Progress update

This week is feature freeze, no new major features will be added (with some small exceptions). We will now focus on report writing, only coding will be bugfixing or minor implementations that will make the software look and perform as well as possible.

- Merlin (home-made automated rendering output testing system).
- Curved text rendering
- Dashed lines
- Map drag and drop panning

Nils will make a PR to Rafaels Docker script. Projects own Docker file is currently not up to date due to needing a new required dependency (imagemagick).

We will try to implement a simple cache eviction policy next week.

02 Acceptance

We performed an acceptance test. Let’s discuss how it was done and the results.
Nils:

- Confirmed that we are meeting the software requirements. One piece of the requirements was missing in the list of requirements, so that needs to be addressed. (Done)

- Key findings:
    - Product owner agreed to participate.
    - Largely met the requirements, and he is very happy with the software so far

- Matthias’ 3 wishes:
    1) Mouse input (important)
    2) QTextLayout (important)
    3) Add street names (bonus)

How to display these results in the report?

03 References in report to specific Qt

Nils: I’ve been quite a bit of references in the report, pointing to the documentation of central classes/tools we use in Qt. Examples include QThreadPool, QNetworkAccessManager, QRhi, Vulkan, DirectX, Metal … Is this common, or should we focus more on articles?

Could use footnotes for more technical documents

04 Other Discussion Points

Merlin

- It’s possible to store test image data on GitHub
- Rafael will send some resources to Nils on Teams

Text Rendering

Eimen Showcases Text Rendering to Rafael

Qt Meeting

Date: January 8th, 2024
Time: 12:00 - 13:00
Present: Eimen, Nils, Cecilia, Matthias (Qt representative)
Secretary: Cecilia

Project plan link: https://www.overleaf.com/read/mwgmcysdtwrw#03b5a7

Agenda

- What have we worked on so far?
- Our ideas for the project
- Restrictions from Qt?
- Clarifications?
- Coding standards in Qt? Any formatting tools?

Notes

What have we worked on so far?

- Report and documentation.

Our Ideas for the Project

Copy-paste in the functional requirements we discussed, so Matthias can see them?
Can be found here:
https://docs.google.com/document/d/1mB74R2aKfUNuBTwOrPnKRWzKpg4RZcwdUGOqb9cEaOc/edit?usp=sharing

Our ideas:

- Lines, polygons
- Qt version must be matched against a particular Android SDK version.
- Platforms: Desktop primarily, but we will design the UI as mobile-first to make it easier to port to Android or other OS's
- Do some styling of the map.
- We are not focusing on navigation in the map.

MVP:

- Get core functionalities in place early

Restrictions/Limitations from Qt?

- Write that software-rendering is a restrictions.
- Compare raster and vector results form MapTiler.

Clarifications?

- Anything Matthias or we want to know, clarify, ask.

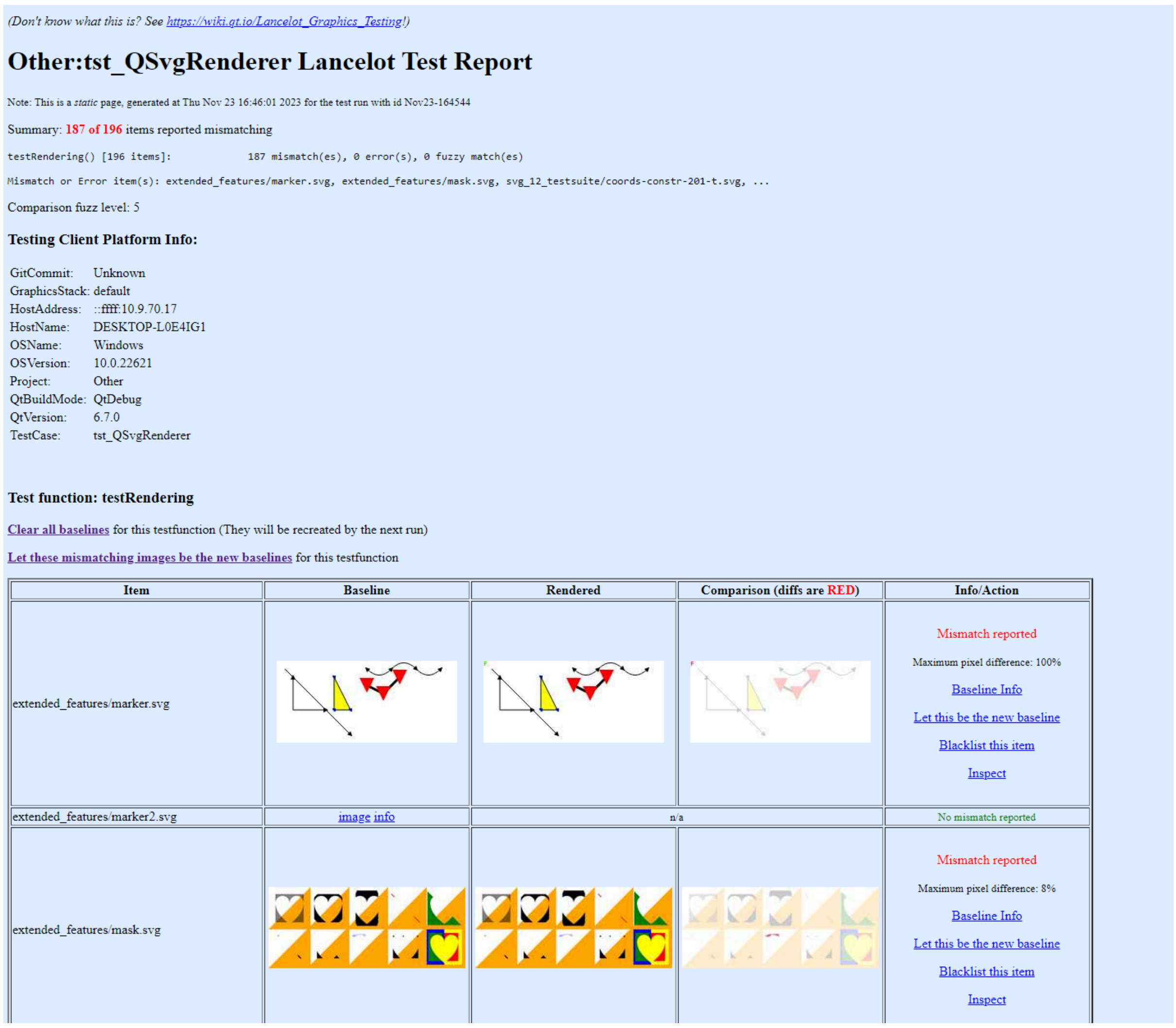
Qt Code Standards, Qt Formatting Tools

- Matthias will ask someone at QT about this and update the team when he knows.
- There are standards.
- Note: It's not very strict in Qt, though.
- Qt does NOT require us to follow a specific format as long as the C++ code is valid and compiles.
- Compare raster and vector results form MapTiler.

Automated Testing and other Tests (?)

- It's possible to set up tests for data parsing.
- Qt has auto-tests that test basic functionality and ensure that nothing crashes, like SVG tests (crashes and security), graphics tests (test output of a picture, is a baseline test where the same image is always used), Matthias has shown us how the result of one of these tests can look. The Lancelot can be set up outside of the Qt CI.

pipeline:



(Don't know what this is? See https://wiki.qt.io/Lancelot_Graphics_Testing!)

Other:tst_QSvgRenderer Lancelot Test Report

Note: This is a *static* page, generated at Thu Nov 23 16:46:01 2023 for the test run with id Nov23-164544

Summary: 187 of 196 items reported mismatching

```
testRendering() [196 items]:            187 mismatch(es), 0 error(s), 0 fuzzy match(es)

Mismatch or Error item(s): extended_features/marker.svg, extended_features/mask.svg, svg_12_testsuite/coords-constr-201-t.svg, ...
```

Comparison fuzz level: 5

Testing Client Platform Info:

GitCommit:	Unknown
GraphicsStack:	default
HostAddress:	::ffff:10.9.70.17
HostName:	DESKTOP-L0E4IG1
OSName:	Windows
OSVersion:	10.0.22621
Project:	Other
QtBuildMode:	QtDebug
QtVersion:	6.7.0
TestCase:	tst_QSvgRenderer

Test function: testRendering

Clear all baselines for this testfunction (They will be recreated by the next run)

Let these mismatching images be the new baselines for this testfunction

Item	Baseline	Rendered	Comparison (diffs are RED)	Info/Action
extended_features/marker.svg				Mismatch reported Maximum pixel difference: 100% Baseline Info Let this be the new baseline Blacklist this item Inspect
extended_features/marker2.svg	image info	n/a		No mismatch reported
extended_features/mask.svg				Mismatch reported Maximum pixel difference: 8% Baseline Info Let this be the new baseline Blacklist this item Inspect

Tips from Matthias

- Always write Qt, not QT!!
- Map layers: Line layer, polygon layer, symbol layer
- Do we add satellite raster imagery? This is optional, but shouldn't be too hard to add, maybe take 1-2 days.
- Use designers to create mockups: Karolina
- Most important part: Split the assignment into separate tasks or parts:
  - Downloading data
  - Parsing data
  - User interface
- Tell Qt how feasible we think it is to continue development of this project. Matthias would really like this. Make sure we document what we are covering or not in the software we're developing. This is good for both Qt and the report thesis.
-

Appendix E

Math Formulas

E.1 World-Normalized Coordinates

Two terms are central to this project; *world-normalized coordinates*s and *tile-normalized coordinates*s. World-normalized coordinates is a coordinate-space that is used commonly internally in our implementation. This coordinate-space was created by us for this project, a similar one may or may not exist in other map implementations as well. The purpose of this coordinate-space is to simplify any math in the implementation, by making it less confusing which coordinate space we are working in. The world-normalized coordinate is defined as having its origin $(0,0)$ in the top-left corner of the world map. The X axis moves positively to the right. The Y axis moves positively *downwards*. Each axis is normalized respective to the world map itself. This means that in the X direction, a value of 0 maps to the world maps left edge, while a value of 1 maps to the right edge. in the Y direction, a value of 0 maps to the world maps top edge, while a value of 1 maps to the bottom edge. Because we can assume the world-map to be perfectly square, the world-normalized coordinate-space is guaranteed to be uniform along both axes.

E.2 Converting Longitude and Latitude to World-Normalized Coordinates

Given longitude $\lambda \in [-\pi, \pi]$ and latitude $\phi \in [-0.5\pi, 0.5\pi]$, we can calculate our world-normalized coordinates by first applying the Mercator formula and then normalizing the range. The value $\phi_{\max}$ is defined as such to make the aspect ratio of the Mercator projection be equal to 1, thus transforming the Mercator projection into Web Mercator. This process is defined in Figure E.1.

$$\begin{aligned}
\text{normalize}(value, min, max) &= \frac{\text{value} - \text{min}}{\text{max} - \text{min}} \\
\phi_{\max} &\simeq 1.4837270864 \\
x &= \lambda \\
y &= \ln\left(\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right)\right) \\
\mathbf{WN}_x &= normalize(x, -\pi, \pi) \\
\mathbf{WN}_y &= normalize(y, -\phi_{\max}, \phi_{\max})
\end{aligned}$$

Figure E.1: Converting longitude and latitude to world-normalized coordinates

E.3 Indexing of Tiles

We may index into the correct tiles by taking the maps zoom-level into account. In this thesis, we define the “map zoom-level” as the integer which defines how many times the world-map has been divided. This is used to index into the correct level of detail in the quad-tree. This value has an upper bound determined either by the style-sheet. In this thesis, we only discuss the map-style *Basic*, which at the time of writing has 16 levels (0 to 15) in total. In our mathematical notation, we define this value as $\mathbf{Z} \in \{0, 1...15\}$.

We can then describe the number of tiles in each direction in the XY-plane as 2^{zoom}. Consequently, tiles are laid out in a two-dimensional grid, with tiles having indexed coordinates in the set $\{0, 1...2^{zoom} - 1\}$.

E.4 Tile-Positional Triplet

In this paper, the term *tile-position-triplet* refers to an individual tile’s position within the map’s zoom-level, along with the zoom-level that tile belongs to. It is denoted by the symbol $\mathbf{T}$ with components $(\mathbf{x}, \mathbf{y}, \mathbf{z})$. The formal definition, along with the constraints of the components, is described in Figure E.2

$$\begin{aligned} &z \in \{0, 1, \ldots, 16\} \\ &x \in \{0, 1, \ldots, 2^z - 1\} \\ &y \in \{0, 1, \ldots, 2^z - 1\} \\ &\mathbf{T} = (\mathbf{x}, \mathbf{y}, \mathbf{z}) \end{aligned}$$

Figure E.2: Tile-position-triplet definition

E.5 Viewport Size

Given the viewport's zoom-level $\mathbf{Z} \in [0...16]$ and the viewport's aspect-ratio. We can calculate a viewport's width $\mathbf{V_{width}}$ and height $\mathbf{V_{height}}$ as a factor of the world map's size by employing the formula described in Figure E.3.

$$\begin{aligned} \mathbf{V_{width}} &= 2^{-Z} \cdot \min\left(1, {V_{\text{aspect}}}^{-1}\right) \\ \mathbf{V_{height}} &= 2^{-Z} \cdot \max(1, V_{\text{aspect}}) \end{aligned}$$

Figure E.3: Calculating viewport size as a factor of the world map

E.6 Calculating Visible Tiles

We can calculate the set of visible tiles by projecting four of the viewport's edges — left, top, bottom and right — onto the world map. From there we can translate these into the same coordinate-space as the indexed coordinates of the tiles. By applying a floor and ceiling operation to these results, respectively, we can form the boundaries of a set of tiles in X and Y directions. The final set $\{\mathbf{T}\}$ can be expressed as the cartesian product of these sets as described in Figure E.4.

This approach is limited to axis-aligned viewports. A rotating viewport would require a different approach.

Given:

Zoom-level of map **Z**

Viewport center coordinates in world-normalized coordinates**WN**

Viewport width in world-normalized coordinate-space $\mathbf{V}_{\text{width}}$

Viewport height in world-normalized coordinate-space $\mathbf{V}_{\text{height}}$

Let:

$$l = \left\lfloor 2^Z \cdot \left(\mathrm{WN}_x - \frac{V_{width}}{2} \right) \right\rfloor$$

$$r = \left\lceil 2^Z \cdot \left(\mathrm{WN}_x + \frac{V_{width}}{2} \right) \right\rceil$$

$$t = \left\lfloor 2^Z \cdot \left(\mathrm{WN}_y - \frac{V_{height}}{2} \right) \right\rfloor$$

$$b = \left\lceil 2^Z \cdot \left(\mathrm{WN}_y + \frac{V_{height}}{2} \right) \right\rceil$$

$$A = \{(x_i, y_i), \ldots\} = \{l, l+1, \ldots, r\} \times \{\mathrm{t}, \mathrm{t}+1, \ldots, \mathrm{b}\}$$

$$\{\mathbf{T}\} = \{(x_i, y_i, Z) | (x_i, y_i) \in A\}$$

Figure E.4: Calculating set of tiles within viewport

E.6.1 Selecting Map Level of Detail

There is, technically, no relation between the viewport zoom-level and the map zoom-level. However, it is important to define one so that the application will render a level of detail that is adequate for the degree of viewport zoom. Failure to do so can have a consequence of the map containing either too little or too much detail, which can impact both user experience and runtime performance. It should be noted that since the viewport zoom is part of continuous set of values, while the map zoom is in set of integer values, it's impossible to keep tiles at a fixed on-screen size while the viewport-zoom changes.

One option is to form a relation where the viewport is rounded and clamped to the nearest integer. This will provide a near $1:1$ relation where each tile will have an on-screen that is close to being equal to the screens size. Such a relation can be described as

At the request of the product owner, we developed a simple formula that allows to us to fit our map-zoom level as to make each tile close to a specific size on-screen. In our case, we wanted to make on-screen tiles have a size of 512 pixels and then choose the closest map zoom-level to achieve this. Our formula is outlined in Figure E.5.

Given:

V_{width}	the width of the display (in pixels)
V_{height}	the height of the display (in pixels)
V_{zoom}	the viewport's zoom level
desiredsize	the desired tile size (in pixels)

$$\begin{aligned}\text{targetmapzoom} &= \text{round}\left(V_{\text{zoom}} - \log_2\left(\frac{\text{desiredsize}}{\max\left(V_{\text{width}}, V_{\text{height}}\right)}\right)\right)\\ \text{result} &= \text{clamp}\left(\text{targetmapzoom}, 0, 15\right)\end{aligned}$$

Figure E.5: Approximating Tile On-screen Size

Appendix F

Vector Tile Protobuf Structure

```
option optimize_for = LITE_RUNTIME;

message Tile {

        // GeomType is described in section 4.3.4 of the specification
        enum GeomType {
             UNKNOWN = 0;
             POINT = 1;
             LINESTRING = 2;
             POLYGON = 3;
        }

        // Variant type encoding
        // The use of values is described in section 4.1 of the specification
        message Value {
                // Exactly one of these values must be present in a valid message
                optional string string_value = 1;
                optional float float_value = 2;
                optional double double_value = 3;
                optional int64 int_value = 4;
                optional uint64 uint_value = 5;
                optional sint64 sint_value = 6;
                optional bool bool_value = 7;

                extensions 8 to max;
        }

        // Features are described in section 4.2 of the specification
        message Feature {
                optional uint64 id = 1 [ default = 0 ];

                // Tags of this feature are encoded as repeated pairs of
                // integers.
                // A detailed description of tags is located in sections
                // 4.2 and 4.4 of the specification
                repeated uint32 tags = 2 [ packed = true ];

                // The type of geometry stored in this feature.
                optional GeomType type = 3 [ default = UNKNOWN ];

                // Contains a stream of commands and parameters (vertices).
                // A detailed description on geometry encoding is located in
                // section 4.3 of the specification.
                repeated uint32 geometry = 4 [ packed = true ];
        }

        // Layers are described in section 4.1 of the specification
        message Layer {
                // Any compliant implementation must first read the version
                // number encoded in this message and choose the correct
                // implementation for this version number before proceeding to
                // decode other parts of this message.
                required uint32 version = 15 [ default = 1 ];

                required string name = 1;

                // The actual features in this tile.
                repeated Feature features = 2;

                // Dictionary encoding for keys
                repeated string keys = 3;

                // Dictionary encoding for values
                repeated Value values = 4;

                // Although this is an "optional" field it is required by the ↩
    specification.
```

```
                // See https://github.com/mapbox/vector-tile-spec/issues/47
                optional uint32 extent = 5 [ default = 4096 ];

                extensions 16 to max;
        }

        repeated Layer layers = 3;

        extensions 16 to 8191;
}
```

Appendix G

Feature Attributes Encoding Example

This example taken from the vector tile specification repository [16]: For example, a GeoJSON feature like:

```json
{
    "type": "FeatureCollection",
    "features": [
        {
            "geometry": {
                "type": "Point",
                "coordinates": [
                    -8247861.1000836585,
                    4970241.327215323
                ]
            },
            "type": "Feature",
            "properties": {
                "hello": "world",
                "h": "world",
                "count": 1.23
            }
        },
        {
            "geometry": {
                "type": "Point",
                "coordinates": [
                    -8247861.1000836585,
                    4970241.327215323
                ]
            },
            "type": "Feature",
            "properties": {
                "hello": "again",
                "count": 2
            }
        }
    ]
}
```

Listing G.1: Feature Metadata JSON Encoding.Taken From Github Repository [16]

Could be structured like this:

```
layers {
  version: 2
  name: "points"
  features: {
    id: 1
    tags: 0
    tags: 0
    tags: 1
    tags: 0
    tags: 2
    tags: 1
    type: Point
    geometry: 9
    geometry: 2410
    geometry: 3080
  }
  features {
    id: 2
    tags: 0
    tags: 2
    tags: 2
    tags: 3
    type: Point
    geometry: 9
    geometry: 2410
    geometry: 3080
  }
  keys: "hello"
  keys: "h"
  keys: "count"
  values: {
    string_value: "world"
  }
  values: {
    double_value: 1.23
  }
  values: {
    string_value: "again"
  }
  values: {
    int_value: 2
  }
  extent: 4096
}
```

Listing G.2: Feature Metadata Protobuf Encoding. Taken From Github Repository [16]

Appendix H

Unit Test Example

Below follows a code snippet that shows how the thesis unit tests look. This test was checked for logical errors by posting it to ChatGPT. Please note that the formatting is styled differently from how it is in the actual code so that it will fit on the page here.

```
// Test the getTilesLink function with an unknown source type
void UnitTesting::
getTilesLink_unknown_source_type_returns_unknown_source_type_error()
{
    QString unknownType = ("random_string");
    // Create a valid JSON style sheet with a different source type
    QJsonObject sourcesObject;
    QJsonObject sourceTypeObject;
    sourceTypeObject["url"] = "https://example.com/tiles";
    sourcesObject["another_source_type"] = sourceTypeObject;
    QJsonObject jsonObject;
    jsonObject["sources"] = sourcesObject;
    QJsonDocument styleSheet(jsonObject);

    // Call the function with the style sheet and an unknown source type
    ParsedLink parsedLink = Bach::getTilesheetUrlFromStyleSheet(styleSheet,
     ↪   unknownType);

    // Verify that the result type is unknown source type
    QCOMPARE(parsedLink.resultType, ResultType::UnknownSourceType);
}
```

Listing H.1: Unit test where ChatGPT helped

Appendix I

Software Review with Product Owner, April 2024

On April 9th, 2024, a semi-structured acceptance test was performed where Matthias Rauter tested the software. Some tasks and questions were prepared beforehand, but feedback could also be provided freely as the conversation went along. The interviewer could ask follow-up questions and inquire further about any topic. That way, the interviewer could ensure that desired input was gathered, while Qt's contact person could provide feedback on current and missing features if he liked to. The prepared questions, with the team's interview notes, can be found in Appendix J.

Software Review Setup

Eleven test questions were prepared before performing the test. Before starting the test, Matthias was notified that he could stop the test at any point, and he confirmed he wanted to participate voluntarily and was okay with having data gathered for the thesis. He was notified that any personal or sensitive information (for example about family, religion, or sexuality) would not be recorded or used.

The software itself was configured to run without the participant having to input any additional information (like the MapTiler key) to start the program. The screen was shared via a projector to show performed software interactions to the team while the test was ongoing.

Results

During the test, Matthias was asked to freely explore the application. The following was pointed out: dotted lines were missing, that one could change (map) projection in the future, that `QTextLayout` could be used to improve application performance, that mouse controls and street names should be added, that one could add support for having the map either stop or wrap once the user goes outside the map boundaries, that it could be nice to hide the map controls automatically after some time had passed, and decrease the size of the "+" and "-" buttons. He further stated that the application didn't have to support map symbols.

Matthias stated multiple times that the team should be happy and proud of the result.

At the end of the conversation, the contact person was asked to pick the three features he thought were critical to improve or implement before final delivery, to which the following four were mentioned:

1. Use `QTextLayout` to render text instead of the current solution to improve application performance.
2. Add mouse input.
3. Add street names.
4. List standards or (map) features the software supports.

The contact person stated that he was very happy with the application, but that the team should try to add `QTextLayout` (1) and mouse controls (2) before the final delivery. He stated that (3) and (4) would be nice to implement as well, but that these were not critical. Additionally, 1-4 were not required to be completed. The software was approved by Qt.

Feature Changes and Implementations Based on Software Review

Since the test, the team implemented support for `QTextLayout`, added panning and zooming mouse interactions, and street names are now displayed in the application. A list of map features that the software supports, has been added to the project's GitHub.

Based on the acceptance test, the team decided not to add additional support for parsing map types other than Basic V2. Using a different map type is required if the software must render symbols. Since the test revealed that supporting symbols was no longer required, the team decided to not implement symbol support.

Conclusion

The results of the acceptance test show that Qt was satisfied with the application as it was in April 2024. The product owner gave input on features that they saw hadn't been implemented, like dashed-lines support, mouse interactions (panning and zooming), and missing street names. Qt's contact person wanted the team to try using `QTextLayout` for text rendering and implement mouse interactions before the final delivery. Adding in street names and listing standards or features the software supports were also mentioned as nice additions. None of these features were required to be completed for the final delivery. Finally, it was confirmed that supporting symbols in the software was no longer a requirement of the software.

Appendix J

Software Review Questions, April 2024

Num	Task/Instruction	Question(s)
1	Revise the requirements that we agreed upon, as outlined in section 3.1 in the report.	Can you confirm that the requirements are correct? Is anything incorrect or missing?
2	Let the product owner play with the application for 1-2 minutes.	Could you tell us your first impressions? What do you think about the application? Are any elements missing or being rendered incorrectly?
3	Have the product owner zoom, pan and filter out elements (fill, line, text). Start at the initial zoom level and go down to at least zoom level 10.	How is the loading time (to you)? Are there any rendering issues when zooming or loading? Is the responsiveness and performance acceptable to you? If not, could you please elaborate?
4	Switch between the raster tiles and vector tile maps.	What are your first impressions of the two modes? Does anything stick out to you as interesting, nice, bad, or something else? There are slight color variations between the two map types. What do you think about that? Is there anything you can see here you'd like to have changed?
5	Have product owner input pre-prepared coordinates for New York manually.	How is that experience? Can you describe it? Does the system behave like you expect it to?
6	Have product owner input a self-chosen set of coordinates manually.	How is that experience? Can you describe it? Does the system behave like you expect it to?

7	Have product owner navigate to Nydalen or Gjøvik with the buttons on the left in the GUI.	How is that experience? Can you describe it? Does the system behave like you expect it to?
8	Test the buttons on the menu in the upper right corner.	Do they behave like you'd expect? Is anything missing there?
9	Inspect the text.	What do you think about the text? Follow-up if product-owner doesn't mention this un-prompted: Does text look acceptable? Is it critical to include curved text for road names?
10	Play with the application and try to break it.	Can you play with the application and try to break it?
11	Ask for final feedback and start post-test discussion.	We have gone through all of the questions. Is there anything you would like to discuss further? Do you have any questions for us? What would you like us prioritize for the final improvements of the software?

Table J.1: Acceptance Test Questions and Feedback

Post-Test Discussion

After asking these questions, the post-test discussion with Matthias started.

Notes Taken By Nils During the Test

- Product Owner agreed to the test.
- We largely met the requirements set out by the Product Owner
- Product Owner was generally very happy with the result of the project. (Seems to have exceeded their expectations?)
- Product Owner tried to repeatedly press the panning buttons. Then had to wait for a while before app became responsive again and would then have panned far.
- "The raster ones have slightly more features. They seem to have lines smaller than one pixel."
- Product Owner said he liked the rendering filters (fill, lines, text).
- Product Owner said he liked the raster-tile comparison functionality (more than expected, it was initially not requested).
- Product Owner pointed out these possible features or attributes in the software:
- Product Owner asked how it would look when loading other style sheets.
- Product Owner tried to write in coordinates by only typing in zoom and no coordinates but this didn't work. Also wanted to load coordinates by pressing enter (instead of manually clicking Go button).
- Product Owner would like in the documentation, a matrix of features and styles supported.
- Regarding filtering buttons in top-left: "For debugging and example application, it's super useful. For a front-end product you would keep this away."
- "This is really good responsiveness with no text. With text is almost acceptable responsiveness if zoomed in."
- Said he wants to see other map types, but he clarified that it wasn't for this project
- Initially Product Owner did not notice a slight variation in color between vector and raster-rendering. "That indicates we're missing a feature. That might be a bug we can solve. Some customers might use color to identify parts of the map in the future". He stated that this wasn't a problem for him, but that it may be an issue to some end-users of graphical applications, and that this is sometimes reported by them.
    - Cursor movement (panning and zooming) not implemented. For zooming, it doesn't need to support pinching, only scrollwheel.
    - Product Owner suggested that the final feature before hand-ind should be curved text.

  - World map is not repeating when panning all the way to the side. Could also instead implement limits on how much you can pan outside world map.
  - Product Owner noticed that tiles have mistaken boundary problems sometimes (line placement bug)
  - Product Owner noticed that dashed lines are not rendered correctly.
  - Product Owner noticed that lines that are supposed to be less than 1 pixel wide, ended up becoming exactly one pixel wide when anti-aliasing is not enabled.
  - Product Owner mentioned debug information could have more info, but also that parts of it could or should be hidden for thesis presentation?
  - Could potentially have an improved approach to render-filters where we list each individual layer of the loaded style sheet.
  - Auto-hide controls, suggested after 3 seconds. He suggested that they pop into their own menu that can be opened again, or that they can become partially transparent after the suggested time.
  - Add background to controls.
  - Product Owner pointed out that text gets cut off around tile boundaries. (Problem with using tile-local text collision rather than global).
  - Mentioned he would have liked the + and - symbols to be smaller than they were.
  - Product Owner would have liked the coordinates-UI to update when moving the viewport around.
  - Product Owner mentioned text rendering is ’unproportionally’ slow. He stated the class QTextLayout should improve it.
- Cecilia: Final question, can you break it? Product Owner could write 190 into longitude. Viewport can be zoomed out too much. Checks memory usage, Product Owner stated it was acceptable, "Still less than Chrome".
- Product Owner disconnected the internet and introduced some terminal errors, but it did not impact the actual visuals or user experience.
- Discussion after test: Product owner mentioned potentially changing or using other projections, like azimuth. Something called Proj4 was also mentioned in this context.

Notes Taken By Cecilia During the Test

When Cecilia had finished asking Matthias the listed questions, she asked him to pick three features he thought needed to be implemented in the project’s final weeks. He wanted the following to be improved:

- Use QTextLayout instead of the current solution.
- Add mouse input.
- Add street names.
- He finally mentioned: It would be amazing to list standards or (map) features the software supports.

He stated that he was very happy, but that we should try to add at least QTextLayout and mouse controls.

Software Review Summary

The overall impression from the Product Owner seemed to be good, since he stated multiple times we should be proud of our work, and that the program looks good. He approved the product in its current state, but asked us to improve text rendering and adding mouse controls if we could make it in time for the final delivery of the thesis.

Appendix K

Final Acceptance Test

This test contains 10 test cases (see the next page). Each case explains the pre-requisites (”Given”). The ”When” fields list what the tester needs to do, and the ”Then” fields lists what should happen next for the test to be successful.

If the product owner approves of the results, ”yes” will be filled in for each ”Approved by product owner” section and ”no” otherwise. The test design is inspired by [46] and [41].

Scenario: 01	Download and run the program successfully.
Corresponding requirements	FR 1
Given	Having access to the Qt thesis GitHub and a MapTiler key. This can be found at `https://github.com/cecilianor/Qt-thesis`.
When	Tester clones the `main` branch from the repository AND follows the GitHub build instruction to build and run the code.
Then	The program starts and renders the map.
Approved by product owner	yes

Scenario: 02	Run the program and check that vector tiles are loading.
Corresponding requirements	FR 1
Given	The program is running.
When	Clicking the ”Showing tile type” button to swap between raster and vector map type AND then zooming in and out with mouse wheel or W and S keys.
Then	The map swaps between vector and raster source type. When zooming in and out, the vector image will always be sharp, the raster image will be blurry when between whole number/integer zoom levels, like 2.5. This will show visually that vector data has been used.
Approved by product owner	yes

Scenario: 03	Run the program and wait for the map to finish loading.
Corresponding requirements	FR 2, FR 6
Given	The program is running.
When	The program runs for the first time AND the machine has network access.
Then	Vector map tiles start loading automatically and will be rendered one at a time, in real-time, until all tiles in the viewport have been rendered.
Approved by product owner	yes

Scenario: 04	Zoom in until text, roads, rivers, and buildings are rendering.
Corresponding requirements	FR 4
Given	The program is running.
When	Using W and S keys to zoom in/out OR using mouse wheel to zoom in to zoom level 14 OR press the ”Gjøvik” button and zoom in from there. Pressing the debug button will show the zoom level for the current tiles.
Then	Place and road names, roads, rivers, and buildings are displayed.
Approved by product owner	yes

Scenario: 05	Confirm that `.mvt` files are loaded and cached.
Corresponding requirements	FR 3
Given	The tester has run the program for the first time AND downloaded at least one vector map tile.
When	The tester accesses their generic cache location AND the tile has been written to the cache location.
Then	The tile data is stored in the cache as an `.mvt` file (MapBox Vector Tile format)
Approved by product owner	yes

Scenario: 06	Confirm that Basic V2 Map Type is used.
Corresponding requirements	FR 5
Given	The tester has view access to the GitHub repository AND has opened the `main.cpp` file in the app folder.
When	Locating line 42.
Then	The code calls to use the correct Basic V2 map type.
Approved by product owner	yes

Scenario: 07	Confirm that Qt can continue development of codebase.
Corresponding requirements	NFR 1
Given	The tester has view access to the GitHub repository and all project source files.
When	Inspecting source code files.
Then	The tester assesses the codebase to determine if it’s suitable for Qt’s continued map software development.
Approved by product owner	yes

Scenario: 08	Check that Qt functionality is used in the software.
Corresponding requirements	NFR 2
Given	The tester has view access to the GitHub repository and all project source files.
When	The tester inspects a source file, for example `MapWidget.cpp` in the app folder.
Then	The tester can confirm that the Qt framework is used by the software.
Approved by product owner	yes

Scenario: 9	Confirm that `QPainter` has been used in the codebase.
Corresponding requirements	NFR 3
Given	The tester has view access to the GitHub repository and all project source files.
When	Inspecting header files and functions in the files `Rendering.h` and `Rendering.cpp` in the lib folder.
Then	The tester can confirm that `QPainter` has been used in the software.
Approved by product owner	yes

Scenario: 10	Confirm that the project uses an MIT license.
Corresponding requirements	NFR 5
Given	The tester has view access to the GitHub repository and all project source files.
When	The tester opens `https://github.com/cecilianor/Qt-thesis/tree/main`.
Then	An MIT license can be located when scrolling down to the Readme and License section.
Approved by product owner	yes

Appendix L

Continued Development Report

L.1 Introduction

This report has been written as a part of the delivery of a bachelor thesis from 2024. This thesis work entailed parsing and then rendering both raster and vector-based map data for the Qt group, the project's product owner.

The report briefly covers the project context, scope, and implemented features. Afterwards, it presents some issues that were encountered during development, followed by a reflection on what could be improved, and finally suggesting what Qt could do to continue developing the map software after they take it over after the deliverance of the bachelor thesis.

L.2 Improving Rendering Performance

Given more time, the group would look into methods to improve the responsiveness and motion fluidity of the application. The project uses software acceleration (a side-effect of using the QPainter tool) for the rendering, which means the actual drawing algorithms are ran executed on the CPU. Graphics rendering is a massively parallelized operation that can be greatly accelerated by using a GPU (Graphics Processing Unit). Compared to a CPU, a GPU excels at performing rendering tasks both in terms of absolute performance and also in terms of power-efficiency. By offloading the rendering workload to the GPU, we expect to gain considerable performance improvements in terms of how fast individual frames can be produced. This in turn means we can output more Frames Per Second (FPS) which means an improvement in perceived motion fluidity and responsiveness.

L.2.1 Challenges

While the group is not experienced in hardware-accelerated rendering in the context of the Qt framework, we can estimate certain tasks that would be required to facilitate the transition.

In order to facilitate hardware-accelerated rendering, one must use a graphics API (Vulkan, DirectX, Metal). We would expect this aspect to be implemented using Qt QRhi[45] class, which provides a high-level abstraction over multiple graphics APIs.

L.2.2 Moving Data to the GPU

The code works a lot with geometry. Fortunately, all of the data are a combination of points and edges (no curves), which maps nicely onto what a GPU wants to work with. In our solution, lines and polygons are represented by the QPainterPath[31] object. This object does not work within the context of hardware-accelerated rendering.

For polygons, we expect that this data will have to be encoded as a list of vertices and indices in order to be rendered using the GPU.

Lines are encoded as lists of points, with an external value describing the thickness. In order to be rendered, lines would therefore have to be triangulated into proper meshes.

L.3 Stylesheet Parsing

Given more time on this project, there are certain limitations we would like to address when parsing style sheets.

Improved Style Sheet parsing

Our current application is highly tailored to the stylesheet that was recommended by Qt, the `basic-v2` [4]. For the future development of the project, an improvement would be to make the application accept any style sheet, even those that are not provided by Maptiler -as long as they adhere to the style specification [15]. The subclasses of the `AbstractLayerStyle` only parse and use the properties that we need for the `basic-v2` style, which is only a small subset of all the properties allowed. An ideal solution would check for the presence of all possible properties and be able to parse and use any subset of them. Furthermore, the `Evaluator` class if only able to process the expressions that are used in the `basic-v2` style sheet. An improvement of the class would be to add support for all the possible expressions in the specification [23]. The `Evaluator` class is designed to be extensible so that adding support for all the other expressions, even future expressions, would be seamless and would not require any change in the already existing code. The only thing that would be needed is to add the static functions that would handle the new expressions to the class and add their entry in the map member variable.

L.3.1 Error Handling

In the current implementation, it's always assumed that loaded style sheets come from MapTiler and are compliant with the MapTiler style-sheet specification[15]. An ideal solution would allow for a style sheet to come from any source, and be used in the application. A challenge is here that a style sheet is not guaranteed to follow the MapTiler style-sheet specification.

An example of style sheets containing invalid values is expressions [23]. The "linear" operator, denoting a linear interpolation, requires exactly 2 operands. This should be denoted as a JSON array with length 3, where the first element is the operator and the remaining elements count as the operands. It's possible to write a style-sheet JSON that is a valid JSON but is not a valid style-sheet, by writing a JSON array that does not have the correct length.

In our project, we do not handle such an error.

L.3.2 Layer Filtering

The group wanted to implement more advanced functionality for showing and hiding different elements of the map. An approach we had planned out but had no time for, included being able to show/hide specific layers defined by the style sheet. This approach would include an additional step during the parsing of the style-sheet, to grab the list of layer-names. We then would create a map structure that maps specific layers to a boolean value, representing whether the layer should be hidden or shown. This would then be configurable in a UI and passed into the rendering functionality.

L.3.3 Improved Text Rendering

Road names

Tex rendering was one of the more complicated problems to solve during development, especially curved text rendering. Since Qt does not support any functionality for curved text using QPainter [20], we had to make our custom solution for curved text rendering. This solution has some shortcomings. Some areas that would be a good subject for future development within text rendering would be:

- **Improve speed**: Our solution is not as efficient as it should be for a real-time rendering application. Some optimizations to curved text rendering might be a good idea for a future development task. However, it is important to note that all rendering operations are software-accelerated, which makes rendering much slower. Implementing hardware acceleration might solve this problem.
- **Improve individual character positioning and rotation**: Another improvement that could be made is the position and rotation of each character within the text. This means making sure that individual characters are rendered exactly where they should be and with the correct rotation. Our current solution does an approximation of where the character should be drawn.
- **Improve non-lating text rendering**: Our current solution does not consider non-latin languages including Arabic, Hebrew, and some Asian languages. This causes text written in these languages not to be correct, as some of these languages are written from right to left, or individual characters change shape depending on their position within the text. This is also a good task to take on for future development regarding text rendering.

L.3.4 Rendering Fallback for Missing Tiles

Due to the nature of our asynchronously loaded tiles, whenever the user zooms into an area where tiles are not yet loaded, the user will be met with only the background color. When the user has an internet connection, tiles should get loaded in over time and the user should see proper information. However, this type of user experience can potentially be improved in the case where the user has no internet connection, or has higher-level tiles loaded. A solution for improving this would involve identifying visible tiles that are missing from cache, searching for their higher-level counterpart and then rendering a subset of that larger tile in place of the missing tile. This way, we allow the user to see something more than the background color,

L.4 Symbols

Icons

The map style that we chose for our application does not include any icons. However, a complete solution should support icons, and this might be a good task that developers who want to use our work or build upon it should work on. To include this functionality the following operations must be performed:

- **Parse URL to request sprites**: The first step is to grab the sprite URL from the style sheet for the specific map style being used. The link must provide two files, an index file and an image file. For the openstreetmap style sheet, we can parse the following field:

```
"sprite":
↪  "https://api.maptiler.com/maps/openstreetmap/sprite"
```

Listing L.1: OpenStreetmap style sheet sprite URL field

- **Request the index file**: The index file can be requested by appending ".json" to the sprite URL. The index file contains information that describes the position and size of individual icons within the sprite PNG. The index file can optionally specify the format of the icons.
- **Request the image file**: The image file can be requested by appending ".png" to the sprite URL. The image file contains the PNG for all the icons in a specific map style. For OpenStreetMap style, requesting the sprite PNG returns the image shown in Figure L.1.
- **Extract icons from the image file using the index file data**: The style sheet will specify the name of the icon to be used in a certain layer using the key "icon-image", the value for the key would be used to extract the icon from the image file. The value of the "icon-image" should correspond to a key in the sprite index file. The value corresponding to the key in the index file has an object value containing the necessary

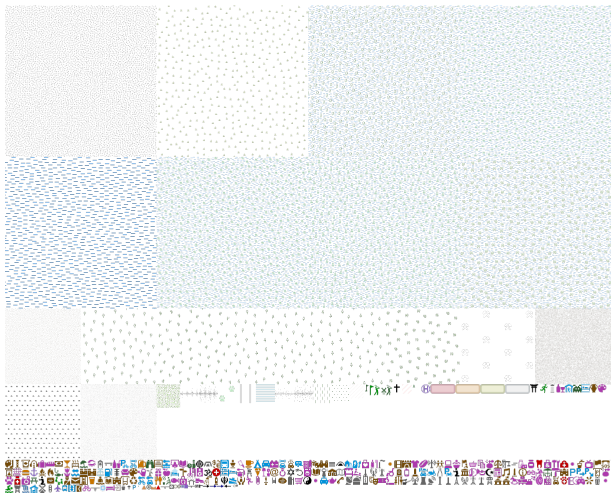


Figure L.1: openstreetmap map style sprite image

information to extract the icon. For example, a layer in the style sheet could include the following field:

```
    "icon-image": "camping"
```

Listing L.2: image-icon JSON field example

This will then be used as a key to look up the icon details in the index file. The index file should include something similar to this:
We then use this information to extract the icon. We go to the exact pixel with coordinated "x" and "y" specified in the index file, which is at the top-left of the camping icon as shown in Figure L.2 And finally, we cut a rectangle with a width equal to the "width" value and a height equal to the "height" value from the "camping" object in the index file, we then obtain the exact icon as shown in Figure L.3.

Suggested QPainter Improvements

The QPainter class has a solid API and a wide range of functionality for software-accelerated rendering. However, during the development of our application, some functionality that would be expected from the QPainter class is missing, especially for text rendering. this functionality would be expected by developers who use the framework. These functionalities are namely:

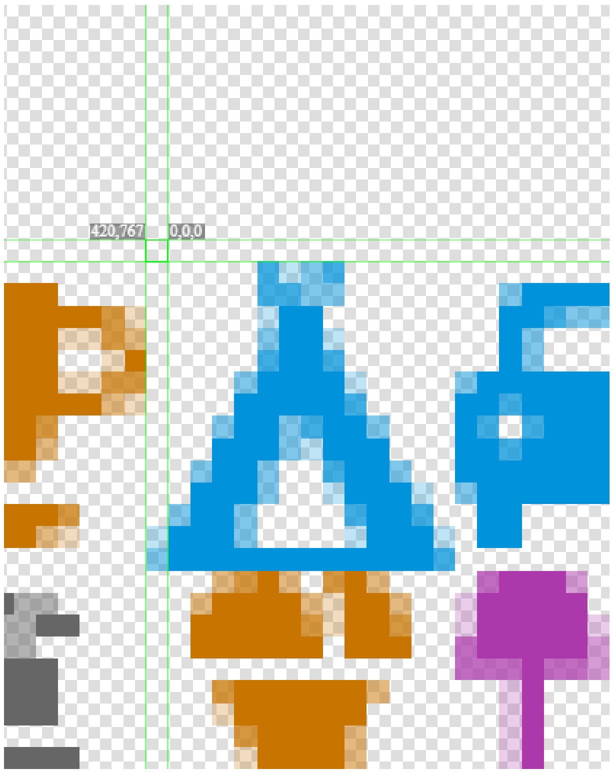


Figure L.2: camping icon top-left coordinate location

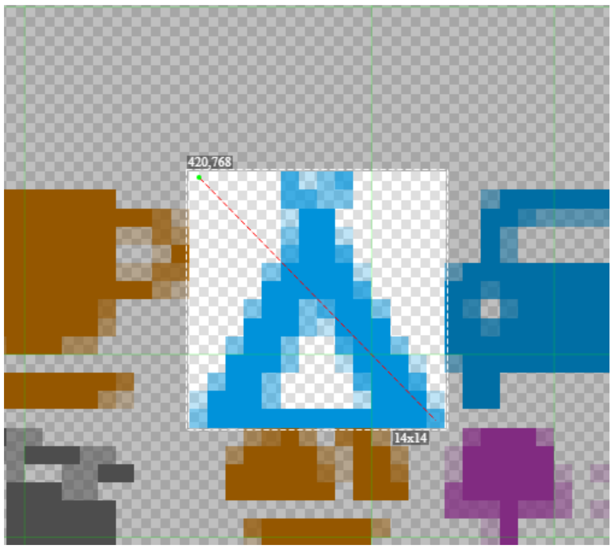


Figure L.3: camping icon

```
    "camping": {
        "width": 14,
        "height": 14,
        "x": 420,
        "y": 768,
        "pixelRatio": 1.0
    }
```

Listing L.3: icon details from sprite index file

- **Support curved text-rendering**: A crucial feature that might be beneficial to many developers using the Qt framework would be rendering text along a path. In our application, we opted for a custom solution to render text, specifically road names, on a curved path. However, this feature would be expected from the QPainter's text rendering capabilities. The suggested improvement is to have a function to which you can pass a QPainterPath and a String as parameters that would render the text along the path. Optionally an overload for this function could take an enum and a float to configure the rendering so that the function can make the text either appear once on the path or render it so that the text repeats as long as there is space for it using the the passed float a spacing between text.
- **Support text outlining and wrapping with QPainter**: Rendering text with outlines and having text-wrapping at the same time was another challenge that we had to make a custom solution for. The QPainter class does not support text outlines however it does include automatic text-wrapping. The QTextLayout class does support text outlines but has no text-wrapping capabilities. The solution we opted for was using QTextLayout with a custom function for text wrapping. The suggested improvement is to include the option for text outlines in the QPainter text rendering functionality. This feature should be simple to include since the QPainter uses QTextLayout internally.

NTNU
Norwegian University of
Science and Technology