FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Sami Varanka

PLATFORM-AGNOSTIC DATA VISUALIZATION IN QT FRAMEWORK

Master's Thesis
Degree Programme in Computer Science and Engineering
June 2023

Varanka S. (2023) Platform-Agnostic Data Visualization in Qt Framework. University of Oulu, Degree Programme in Computer Science and Engineering, 54 p.

ABSTRACT

There are a variety of electronic devices making people's lives easier. Many of these devices, such as smart TVs, coffee machines, and electric cars, have high-resolution displays that visualize a rich interface for human-machine interaction. One common cross-platform framework for developing these visual interfaces is Qt.

With its vast majority of modules, Qt offers a rich framework for developing different interfaces. Over the years, various modules have been added and removed from Qt. One such module is a module for visualizing data in 3D. The module was added when Qt relied heavily on the OpenGL rendering backend. Nowadays, in addition to OpenGL, Qt supports rendering 3D graphics on other popular graphics backends. Nevertheless, the data visualization module requires using OpenGL as the rendering backend.

This thesis investigates the optimal way to change Qt's module dedicated to 3D data visualization to work on different rendering backends. Additionally, a design for an extension to public C++ API for Qt's 3D module, used in implementing the new module, is presented. The requirements for the new implementation of the data visualization module are derived from the current module.

The implementation of the new version is evaluated with a tailor-made test application. The results indicate that the new module can improve performance on devices supporting modern graphics backend features.

Keywords: QtQuick3D, graphics abstraction, GUI

Varanka S. (2023) Alustariippumaton datavisualisointi Qt-ohjelmistorungossa. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 54 s.

TIIVISTELMÄ

On olemassa monia erilaisia ihmisten elämää helpottavia elektronisia laitteita. Monissa tällaisissa laitteissa, kuten älytelevisioissa, kahviautomaateissa ja sähköautoissa, on korkearesoluutioinen näyttö, joka visualisoi monipuolisen käyttöliittymän ihmisen ja koneen väliseen vuorovaikutukseen. Yksi yleinen alustariippumaton ohjelmistokehitysrunko näiden visuaalisten rajapintojen kehittämiseen on Qt.

Qt tarjoaa paljon erilaisia moduleita ja runsaan kehyksen erilaisten käyttöliittymien kehitykseen. Vuosien varrella Qt-ympäristöön on lisätty ja siitä on poistettu erilaisia moduleja. Yksi tällainen moduli on tarkoitettu datan visualisointiin 3D-muodossa. Kyseinen moduli lisättiin, kun Qt luotti vielä voimakkaasti OpenGL-renderöintitaustajärjestelmään. Nykyään Qt tukee 3D-grafiikan esittämistä muillakin suosituilla grafiikkataustajärjestelmillä OpenGL:n lisäksi. Datan visualisointimoduli edellyttää kuitenkin OpenGL:n käyttämistä renderöintitaustajärjestelmänä.

Tässä opinnäytetyössä tutkitaan optimaalista tapaa muuttaa Qt:n 3D-tietojen visualisoinnille tarkoitettua modulia toimimaan useammalla eri renderöintitaustajärjestelmällä. Lisäksi esitellään uuden modulin toteutuksessa käytetyn Qt:n 3D-modulin julkisen C++ API:n laajennuksen suunnitelma. Vaatimukset tiedon visualisointimodulin uudelle toteutukselle on johdettu nykyisestä moduulista.

Uuden version toteutusta arvioidaan sitä varten kehitetyllä testisovelluksella. Tulokset osoittavat, että uusi moduuli voi parantaa suorituskykyä laitteissa, jotka tukevat nykyaikaisia renderöintitaustajärjestelmän ominaisuuksia.

Avainsanat: QtQuick3D, grafiikka-abstraktio, GUI

TABLE OF CONTENTS

FOREWORD

This thesis was done as a part of work for a new module at Qt Group, previously Qt Company. Doing this thesis has been interesting. When I started this, I had no experience using the QtQuick3D module. However, I had to learn to use it through the private C++ API. In the process, I received valuable help and advice from my colleagues in the graphics teams in Oulu and Oslo.

I want to thank my manager Tomi Korpipää for providing such an interesting thesis topic and giving me motivating feedback about the technical aspects of this project. Thanks to my university supervisor Matteo Pedone for providing feedback on how to structure the thesis. Also, thanks to Janne Heikkilä, who acted as the second reviewer. Thanks to Esa Törmänen for proofreading the thesis and giving me feedback on the language. Finally, I would like to thank my friends and family for supporting me during the process.

Oulu, June 13th, 2023

Sami Varanka

LIST OF ABBREVIATIONS AND SYMBOLS

D3D12	Direct3D 12
UI	user interface
GPU	graphics processing unit
CPU	central processing unit
API	application programming interface
MCU	microcontroller unit
QML	Qt meta-object language
ANGLE	almost native graphics layer engine
GUI	graphical user interface
QSG	quick scene graph
QSSG	quick spatial scene graph
PBR	physically based rendering
SSAA	supersample antialiasing
MSAA	multisample antialiasing
PAA	progressive antialiasing
TAA	temporal antialiasing
FXAA	fast approximate antialiasing
SSAO	screen space ambient occlusion
QRHI	Qt rendering hardware interface
D3D11	Direct3D 11
FPS	frames per second

1. INTRODUCTION

Three-dimensional computer graphics refers to three-dimensional images generated by a computer. Computer graphics are used in various fields, such as video games, movies, and data visualization [1, 2].

When the scene to be rendered can change arbitrarily in real-time, the graphics are referred to as real-time graphics. One major factor driving real-time graphics development is the video games industry and game developers' constant demand for more astonishing graphics.

As the graphics quality increases, so do the computational requirements for a single rendered image. However, the time to display an image remains the same. The time that the computer is allowed to process a new image depends on the targeted framerate. In video games, for example, the targeted framerate usually is between 30 frames per second (FPS) and 60 FPS, which sets the maximum time to 33.3 ms and the minimum to 16,6 ms [3].

Unlike real-time graphics, offline graphics refers to graphical images generated offline by a computer or a group of computers organized as a render farm [4]. Offline graphics can achieve far greater quality than real-time graphics since more time can be spent processing a single image. Render farms can drastically reduce the time spent rendering frames since the rendering work can be distributed among many computers instead of just one. For example, offline graphics are used in movies since they do not require as much interactivity as real-time graphics applications.

The main topic of this thesis concerns Qt, a framework for building user interfaces (UIs) on various platforms, and its QtDataVisualization, a module for visualizing data in three-dimensional graphs, and its QtQuick3D, a module for displaying three-dimensional content. Currently, the Qt framework is in its sixth major version. The low-level graphics stack has drastically changed between the fifth and the sixth releases. However, the QtDataVisualization module has yet to benefit from those changes fully.

This thesis studies what would be a good way to change the implementation of the QtDataVisualization module so that it would use the QtQuick3D module's rendering engine. In the second chapter, the motivations, like the limitations of the current implementation and the thesis's benefits, are discussed. The Qt framework and its modules involved in this project are presented in the third chapter. In addition, the differences between Qt5 and Qt6 graphic stacks are discussed. The discussion concerning the implementation details is presented in the fourth chapter. It will include topics such as which approaches were considered and why the selected approaches were chosen. In the fifth chapter, the implementation's validation is conducted by running a developed application with the module's old and new versions. From both runs of the application, performance data is collected and compared. After presenting the experiment results, they are discussed in the following chapter. The final chapter summarizes the thesis and discusses future work and possible optimizations.

1.1. GPU and Drivers

A graphics processing unit (GPU) is a piece of computer hardware that handles the generation of images to be displayed. A GPU contains circuitry specialized in doing massive amounts of parallelized floating-point computations involved in image generation. Therefore, it can be referred to as a special-purposed processing unit instead of a central processing unit (CPU) which is a general-purpose processing unit.

A driver is a piece of software that offers means to communicate with a specific type of device [5]. Indeed, in its essence, a driver is an implementation of communication specification. Built-in drivers are installed with the operating system for many hardware components and devices. However, in some cases, the built-in driver is insufficient, for example, if the device provides extra functionalities. In such cases, users must install a specific manufacturer-provided driver for the device.

In the case of computer graphics, a driver handles the communication with the GPU. The application communicates with the driver via the software library, which will be discussed in more detail in the next section.

1.2. Graphics API

Graphics application programming interface (API) is a software library that offers an interface to underlying graphics hardware. Modern graphics APIs such as Vulkan, Direct3D 12 (D3D12), and Metal offer lower abstraction over the GPU than the older OpenGL and OpenGLES APIs [6, 7, 8]. This lower abstraction means that there is less driver overhead. Thus, more performance can be squeezed out from the GPU since the application has more control over how the GPU's resources are used. However, the downside is that having more control also means more care must be taken when implementing graphics applications.

Silicon Graphics developed the 1.0 version of OpenGL, a specification for an open cross-platform graphics library. The first version was released in 1992, and the first extension (1.1) was released in 1997 [9]. In 2006 the maintainership of OpenGL specification was passed on to the Khronos Group [10].

The specification describes the functionalities the library implementations must provide to meet the criteria for the specific version [11]. For example, the latest 4.6 specification defines what the conforming drivers need to provide.

At first, OpenGL's rendering pipeline was implemented as a fixed-function pipeline, whereas now, the programmable pipeline is used. With the fixed-function pipeline, the only way to control the rendering stages was by providing various parameters to the functions. The programmable pipeline was added in the 2.0 version, released in 2004. It enables developers to control the rendering stages using shaders, small programs which are executed on the GPU. The last version supporting the fixed-function pipeline was 3.0, released in 2008 [12].

For low-powered mobile and embedded devices, OpenGL was deemed to be unsuitable. Hence, the Khronos Group began the specification of OpenGL ES, a subset of OpenGL for embedded systems [13, 14]. The first version of it was released in 2003, and it was based on OpenGL 1.3. Like OpenGL, it also featured a fixed-function

pipeline that was replaced with programmable shaders in the 2.0 version released in 2007 [15, 9]. The current 3.2 version was released in 2015.

In 2014, the Khronos Group launched the development of the Vulkan specification, and the 1.0 version was released in 2016. The specification defines a low-level interface to underlying graphics hardware. Unlike D3D12 and Metal, Vulkan is a cross-platform library available on various Windows and Unix platforms. While Vulkan does not have official drivers on Apple platforms, it can be used through the MoltenVK library on those platforms [16, 17, 18].

D3D12 was released in the summer of 2015. It is the latest version of Microsoft's GPU acceleration library for Windows platforms, and it offers more control over GPU resources when compared to the previous Direct3D 11 (D3D11) version [19]. Being closer to metal means it takes more effort to use, which would be enough to repel some developers. Those developers can continue using D3D11.

Metal API is GPU acceleration API for Apple's platforms. The first version was announced for iOS in 2014, and a year later for macOS, watchOS, and tvOS [8]. The latest version of Metal is version 3. Like other modern graphics libraries, it offers low abstraction over graphics hardware [20]. Unlike the developers using other modern libraries, the developers using Metal know precisely which hardware their graphics application uses. Hence, it is easier to optimize.

Graphics APIs and their availability are summarized in Table 1.

Table 1. Graphics API availability on different platforms

OpenGL	Windows, Linux
OpenGLES	Windows, Linux
Vulkan	Windows, Linux
Direct3D	Windows
Metal	MacOS, iOS

It is a tedious task if the application developers want their application to work optimally on various platforms since they would need to dwell deep into the inner workings of different graphics backends. Therefore, different software libraries abstract away the lower details of a graphics backend for application developers. These abstractions offer a uniform interface to underlying graphics APIs, depicted in Figure 1. Thus, the developers do not have to consider various graphics backends and can concentrate on developing their application.

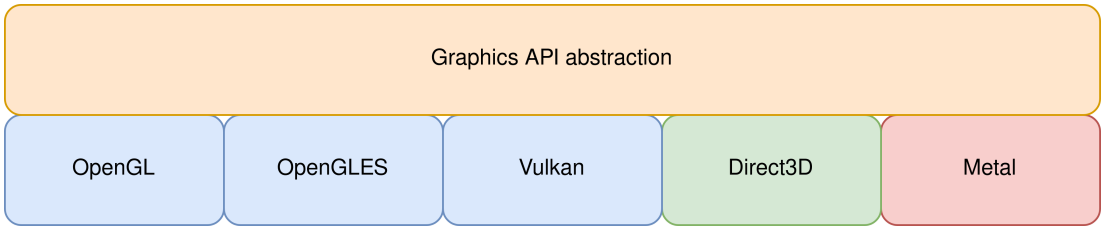

Figure 1. Graphics API abstraction enables access to graphics APIs through a uniform interface.

1.3. Thesis Scope

The primary goal of this thesis is to investigate how to change the rendering engine used in the QtDataVisualization module to use the QtQuick3D rendering engine, which uses a graphics API abstraction layer to work on various graphics backends. Although the module can be used to visualize three different types of graphs, such as bar, scatter, and surface, the thesis aims to implement only one. Otherwise, the scope would grow quite large. The secondary goal is to design a public C++ API for the parts of Quick3D used in the implementation of this project. The software used in this thesis concerns only real-time applications, so the scope is limited to real-time graphics.

2. MOTIVATION

Qt is a framework for building UIs for desktop, mobile, embedded, and microcontroller unit (MCU) platforms. For application developers, Qt offers a robust and easy-to-use UI development framework. It enables the developers to use the same codebase for multiple platforms by abstracting away the lower details of platform-specific functionalities such as window creation or thread management.

Qt enables rapid UI development by employing a specific language called Qt meta-object language (QML). QML is a UI specification and development language describing visual elements of UIs in a hierarchical structure [21]. Besides enabling the description of the visual look of the UIs, it allows the customization of the functionalities with JavaScript. In addition, new logic can be exposed to QML by using C++. Qt has a standard module for UI development with QML. This QtQuick module includes every type needed to develop interactive UIs [22].

The framework's functionality is divided into several modules, which are grouped into two distinct groups: essentials and add-ons. Essential modules are available on all platforms, whereas add-ons provide extra functionality to Qt [23]. The QtDataVisualization and QtQuick3D are both add-on modules.

2.1. Problems with Current QtDataVisualization

Currently, the QtDataVisualization module has a custom-made rendering engine. The engine is made to use only OpenGL as its graphics backend, which is not the most optimal choice on Apple's and Microsoft's platforms. Apple's platforms support OpenGL only to version 4.1 and OpenGL ES up to version 3.0. In addition, Apple deprecated the support for OpenGL in macOS 10.14, and OpenGL ES was deprecated in iOS 12.0 [24, 25, 26].

On Windows, OpenGL has inconsistencies between different driver vendors due to the various extensions allowed by the standard. The standard allows vendors to add new features as extensions to their OpenGL implementation. Those features may then be added to the core standard of OpenGL. However, there are ways to determine which extensions are available in the implementation.

Direct3D specification, on the other hand, is maintained by Microsoft, and it does not have the same kind of extension mechanism as OpenGL, leading to more consistent driver implementations amongst the hardware vendors. In addition, many graphics-intensive applications tend to use Direct3D on Windows, which leads to many Windows systems missing OpenGL drivers altogether [27], or if the drivers exist, they can be unstable. For example, they might claim that they support a newer version of OpenGL API than they do. The driver might also lack some features completely. Even though everything seems correct: the driver informs the correct version, and every feature is implemented. The driver or some of its features might be slow since they are unoptimized. Because many applications use Direct3D instead of OpenGL, it will be difficult to justify the development of good drivers, and thus the drivers continue to be bad. Especially Intel's integrated graphics has had problems with OpenGL drivers.

One solution introduced by Google to tackle the problem of missing drivers is the Almost Native Graphics Layer Engine (ANGLE). Originally ANGLE started as a way

for web browsers to render content using WebGL on Windows systems. Since WebGL is based on OpenGL ES, which is based on OpenGL 2.0, it requires at least OpenGL 2.0 conformant drivers. On Windows-based systems, web browsers cannot rely on having the said drivers available, thus rendering them unable to provide 3D graphics through WebGL. ANGLE works by translating OpenGL ES calls to Direct3D calls. The mapping of OpenGL ES calls to Direct3D calls enables OpenGL ES applications to work on Windows systems where no OpenGL drivers are available [28].

Since the QtDataVisualization rendering engine uses straight-up OpenGL without any abstractions and the module must work on older platforms, the engine can only use the common subset of API functions. The used functions must be available in OpenGL ES 2.0 since it is the oldest supported version. Therefore, it cannot benefit from the newer features of OpenGL, such as the instanced rendering support on platforms where available. Because of the required OpenGL ES 2.0 support, the OpenGL Shading Language is locked to version 1.20. As the OpenGL ES Shading Language 1.0 is based on OpenGL Shading Language 1.20, the shaders used by the engine must use the 1.20 version [29].

The module's rendering engine handles only the displaying of the 3D graphs in the window. The displaying of other UI elements, such as push buttons or textboxes, is enabled by the QtQuick module. The QtQuick's rendering engine uses an abstraction layer that translates the calls to it to underlying graphics API calls. Depending on the platform, the abstraction layer automatically chooses the used graphics API. However, the backend can be selected manually as well. Figure 2 illustrates, how the current QtDataVisualization rendering engine is positioned in the current Qt graphics stack.

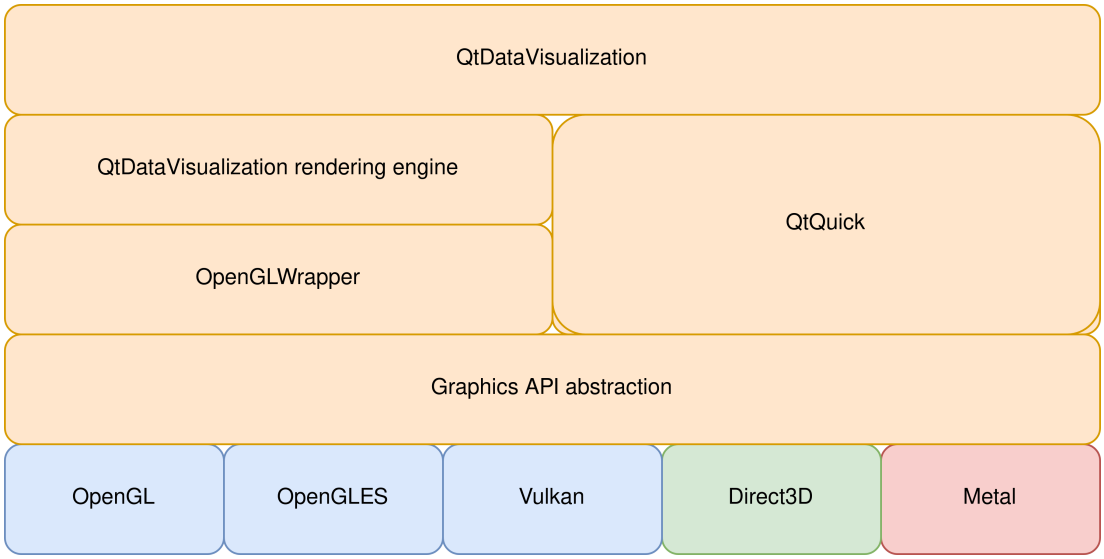

Figure 2. Current QtDataVisualization rendering engine uses OpenGL while QtQuick renders graphics with an abstraction layer.

Since the QtDataVisualization uses OpenGL, it explicitly requires setting OpenGL as the graphics backend for the abstraction layer. Otherwise, the module will crash the first time it tries to create its OpenGL context. That is because it retrieves the OpenGL context created by the abstraction layer and tries to set the module's context surface format to be the same as in the abstraction layer's context. However, the abstraction

layer's OpenGL context is not created if the graphics backend is set, for example, to Direct3D. Since the OpenGL context is not created, the program crashes due to the use of uninitialized memory.

Setting the abstraction layer's backend to OpenGL also means that OpenGL draws every UI element. On platforms where OpenGL is selected automatically as the graphics backend, it will not cause any issues. On the other hand, on platforms like Microsoft's and Apple's, where OpenGL is not the primary graphics backend, it can cause performance issues or malfunction due to the reasons mentioned above.

2.2. Benefits for QtDataVisualization

Qt aims to streamline its development on different graphics backends by using a graphics abstraction layer. The graphics abstraction layer belongs to Qt's private API, so it is only meant to be used internally to develop Qt modules. For developing 3D graphics applications, Qt offers the QtQuick3D module. As QtDataVisualization is a module for visualizing data in 3D, it would be sufficient to use QtQuick3D to implement its visualization. Changing QtDataVisualization to use QtQuick3D is one of the most significant contributions of this thesis since it reduces the maintenance workload of Qt. Hence, it reduces the overall maintenance cost and possible places for program bugs. Moreover, using QtQuick3D enables QtDataVisualization to benefit from different graphics backends than just OpenGL and OpenGL ES, which means it will work on platforms that do not have OpenGL drivers. QtQuick3D is also actively developed and new features are added to it, which improves its capabilities release by release.

2.3. Performance Benefits

There are a wide variety of devices on which the applications developed with the QtDataVisualization module are run. Some of those devices are low-resource embedded devices. For example, one such low-powered device is Toradex Apalis iMX6 Dual. The cheapest version of the board has only 512 Mb of RAM and a CPU clock speed of 1.0 GHz [30]. Therefore, the module must require as few resources as possible while running at a reasonable FPS.

As regards performance, the gaining expectations are not high. Nevertheless, the performance can be expected to be at least as good as in the old implementation. Features in which performance can be expected to be better include QtDataVisualization's static optimization, which causes the drawing of the graph to be done with a single draw call.

In the current QtDataVisualization, static optimization is implemented by loading all graph data to a single array buffer, which enables the whole graph to be drawn with a single draw call. In QtQuick3D, instanced rendering can be used to draw many similar objects with a single draw call.

The problem with the current implementation is that since it loads every data point into a single buffer, it simply acts like one huge mesh. Therefore, the loading process must be repeated if the data is changed later. Hence it is suitable only for large static

data sets. With instanced rendering, however, the static optimization will not be limited to static data sets as it can draw instances of the same mesh with different transforms.

2.4. The Public C++ API for QtQuick3D

The user interface in Qt can be developed with either QML or C++. The module needs to offer a public C++ API to enable the developers to develop applications with C++, which is guaranteed to work in newer releases of Qt. As opposed to public API, the module has a private API. As its name suggests, private API is private, which means it is meant to be used by the developers rather than the users of Qt. A module's division into public and private API is done to enforce binary backward compatibility. The binary backward-compatible module helps distribute updates to the module since the applications using the module do not need to be recompiled to use it. Hence, the users only need to obtain the updated binary.

The private API of a module contains classes that implement the internal functionality of the module. For example, the QtDataVisualization module has a public API class named QScatter3DSeries and a private API class called QScatter3DSeriesPrivate. The users can rely upon the functions and properties offered by the QScatter3DSeries to be the same between newer releases. However, the functionality provided by the QScatter3DSeriesPrivate can change and, therefore, cannot be expected to be the same, as it is meant to be used internally to implement the module.

A module's QML functionality is implemented using the private API, and the parts the users need are exposed to the module's QML API. The functionalities are exposed via various macro definitions offered by the QtQml module [31].

QtQuick3D offers QML API for developing 3D environments and C++ API consisting of classes to implement supporting classes for features, such as instancing [32] and custom geometry [33]. However, the users cannot develop 3D applications with QtQuick3D without writing QML code. It can be a turn-off for application developers accustomed to using C++ as they need to learn a new language to develop their 3D applications. Nevertheless, they could get around that just by using the private API of the module. However, as described above, it is not guaranteed to work between releases. Therefore, QtQuick3D would benefit from extensive public C++ API with which the users could create their 3D environments that would be guaranteed to work between the releases.

3. QT

UIs are used everywhere. There are text-based UIs, where the user interacts by typing various commands. Other UIs include Graphical User Interfaces (GUIs), where the user interaction is not limited only to text. GUIs are used in many places since they offer a richer and more intuitive user experience for regular consumers than text-based UIs.

The Qt Company is the main force maintaining the Qt framework. However, Qt is an open-source project allowing everyone to access and contribute to the source code [34]. Although everyone can contribute to the Qt project, it does not mean every contribution gets integrated into the project. Ultimately, the developers of Qt accept contributions to Qt's source code. Most of the developers work at the Qt Company [35].

One chief maintainer for the Qt project is responsible for leading the group of maintainers [36]. Maintainers are individuals who maintain different modules of Qt [37]. Ideally, every module has a maintainer assigned to it. The module's maintainer takes the main responsibility for the module's success. The maintainer always knows the current state of their module.

3.1. Modules

The functionality of Qt has been divided into modules [23]. The modules are divided into two groups: Qt Essentials and Qt Addons. Qt Essentials are the most important modules since they define the foundation of Qt. They are available on all development platforms, so developers can rely on their availability. The modules provide non-graphical core functionalities of Qt and classes for defining UIs with QML and JavaScript. In addition, the unit testing framework for Qt applications, libraries, and QML applications is included [38, 39].

Qt Addons include modules that provide extra functionality to Qt, such as access to various communication hardware, different sensors, and visualizing data. Both QtDataVisualization and QtQuick3D belong to addon modules [23].

3.2. QtDataVisualization

QtDataVisualization is a Qt module that enables visualizing data in 3D [40]. There are two approaches to developing UIs with QtDataVisualization. The first approach is widget-based, where Qt's widget framework is used. Widgets enable the development of UIs with C++ [41]. The other is QtQuick 2 framework, where the graph rendering is integrated into QtQuick UI [22]. Using QtQuick 2 framework allows the development of UIs with QML. The recommended way is to use QtQuick 2, which has a dedicated scene graph [42].

Adding the third dimension to the graph enables displaying more data in it without sacrificing their interpretation. However, 3D graphs require more ways to interact with them than 2D graphs. The QtDataVisualization module allows viewing graphs from different angles by rotating them.

The first version of QtDataVisualization was released as an external addon for Qt in March 2014 [43]. Back then, the module's release cycle did not follow Qt's release cycle, so its version number was 1.0, while Qt was in 5.2 version [44]. The module already had all three different graph types implemented into it. The ability to use 3D voxel data in graphs was added in version 1.2 [45].

Three essential concepts are needed to visualize data in the QtDataVisualization graph. Those are the *main graph*, a *series*, and a *data proxy* [46, 47, 48]. A series is a type that combines data and properties on how to visualize the data. Every graph type has its distinct series type implemented as a child class of an abstract series type. The abstract series type defines functionality and properties needed by all series types. The data for visualization is given through a data proxy. A data proxy is a type that takes in data in a known format, such as weather data, which describes the monthly amount of rain in a year. The data proxy then transforms that data into a format suitable for presentation. Like every graph type has its corresponding series type, every series type has its corresponding proxy type. Like distinct series types, every proxy type inherits shared functionality from the abstract data proxy type.

The three graph types provided by the module are *bar*, *scatter*, and *surface* [49, 50, 51]. A bar graph enables displaying data in categorized 3D bars. The third dimension allows adding one extra category while maintaining the graph in a compact size. Figure 3 shows how a bar graph is used to visualize income from different months [52]. Here the third dimension displays the income data from different years.

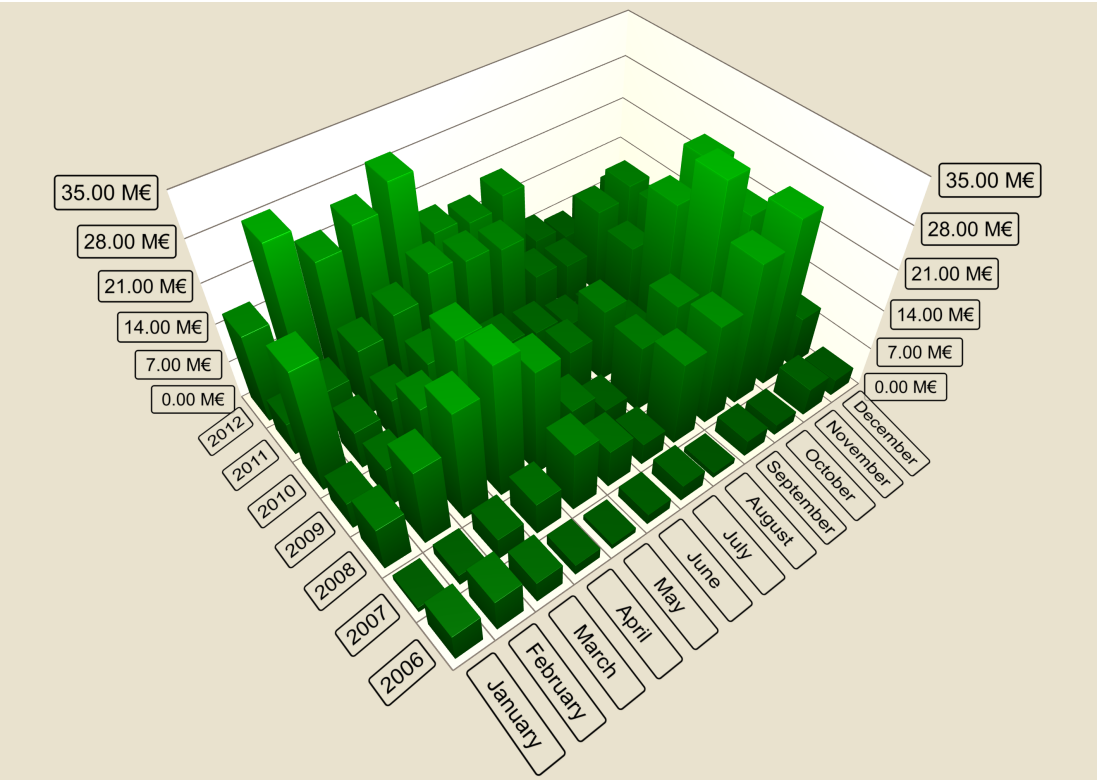

Figure 3. The third dimension can be used to display more categorized data.

A scatter graph visualizes data as points in a three-dimensional space. An example of a scatter graph can be seen in Figure 4 [53].

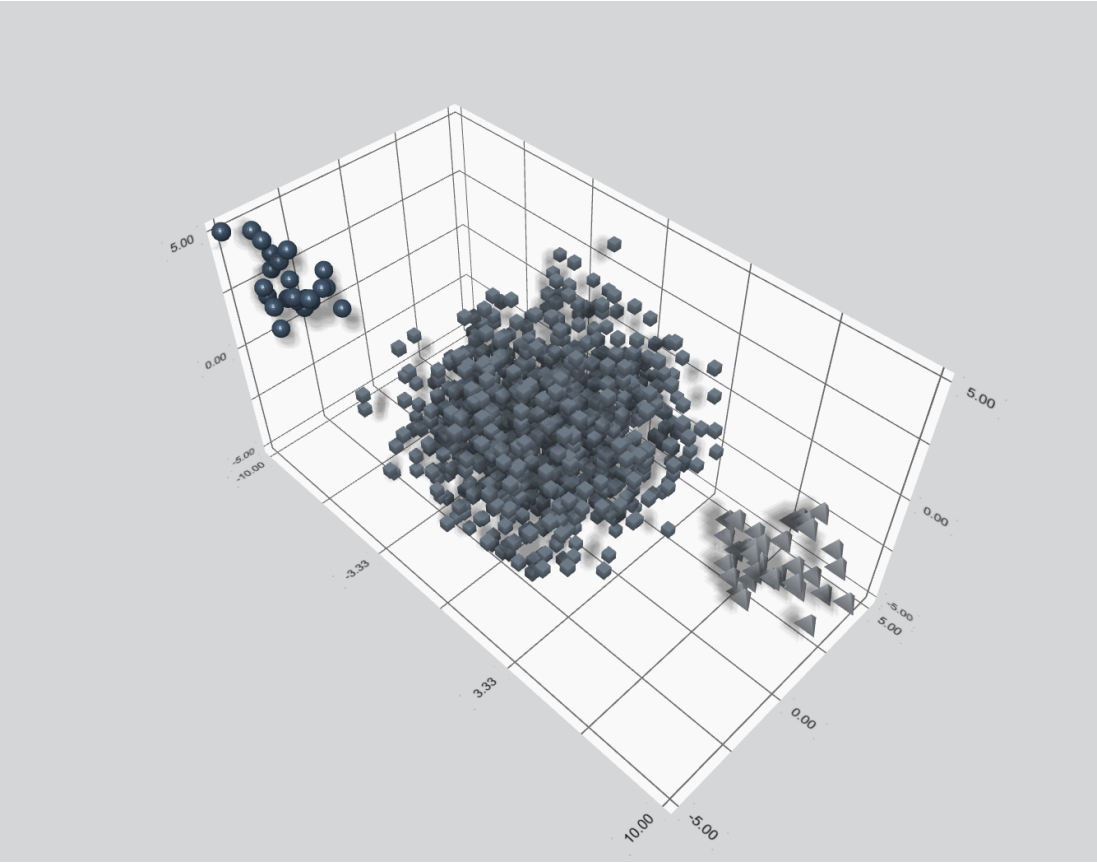

Figure 4. A scatter graph visualizes data as points in a three-dimensional space.

A surface graph displays a 3D surface to visualize data. The data for a surface graph can be provided in a Qt ListModel [54]. In addition, the data can be given as a height map [55]. An example of a surface graph can be seen in Figure 5 [56].

All the graph types support the same ways to interact with graphs: selection, zoom, and rotation [57]. The selection's target for a single graph can be customized, or the selection can be turned off entirely via the selection mode flag [58]. The flag's value defaults to item selection, where the selection applies to only one data point in the graph [59]. Every graph type supports the default selection mode. The scatter graph type supports only the default selection mode, whereas the bar and surface graph types also support the slice selection mode. In the slice selection, the selected row or column is displayed in 2D view. In addition to item and slice selection modes, a bar graph supports selecting a row and column.

The applications made with the QtDataVisualization module can consist of multiple graphs [60]. For example, an application may use scatter and bar graphs. The application might want to turn off zooming for both graphs. In such cases, the developers can disable zooming from the input handler [61]. Indeed, every three ways to interact with the graphs can be turned on or off by the input handler. After all, it is the component responsible for input handling.

As discussed in the motivation chapter, QtDataVisualization uses OpenGL to display graphs on the screen. There is a wide range of supported devices on which QtDataVisualization applications are run. Some of those devices might support only

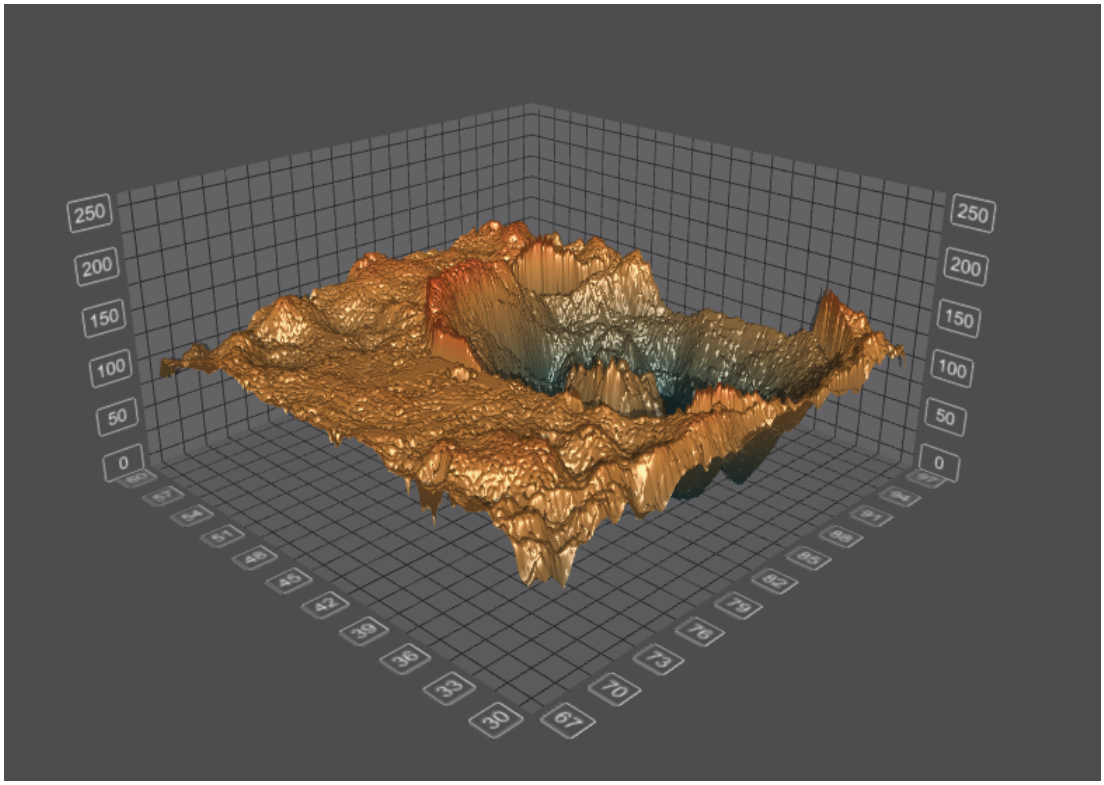

Figure 5. For a surface graph, the data can be given as a height map.

older versions of OpenGL. Therefore, the module's rendering engine uses only a set of features available in OpenGL 2.1.

At the start of a new frame, the module needs to make the window where the graph is drawn current for the used OpenGL context [62, 63]. Here the widget-based and QtQuick 2-based approaches differ. In the former, the graph is the window where the drawing is presented. In the latter, the graph is presented as a QQuickItem. A QQuickItem has a pointer to the window where it is rendered. Apart from the used framework, the rendering processes converge in the Abstract3DController's render function [64]. A 3D controller is a class that controls the graph [65]. In addition, it connects the graph and rendering engine. Every graph type has its corresponding controller type, which inherits shared functionality from the Abstract3DController [66, 67, 68].

With the widget framework, the graph rendering is triggered in the graph's renderNow function [62]. The function calls the graph's render function, which in turn calls the controller's render function. After the rendering, the buffer is presented to the screen with the call to QOpenGLContext's swapBuffers.

The graph's rendering process gets more interesting with QtQuick 2 framework since the graph is integrated into QtQuick UI, and QtQuick 2 uses a graphics abstraction layer under the hood to draw the UI [22]. Moreover, QtQuick 2 has a scene graph where the front end UI items are presented as nodes [69]. Since the QtDatavisualization graph is also a UI item, it has a corresponding node in the scene graph [46]. Since QtDataVisualization has a custom rendering engine, the graph

rendering needs to be done before the final frame rendering where the scene graph is rendered.

Processing on a node before the frame rendering can be done in the node's preprocess function [70]. The QtQuick 2's renderer calls the function when it traverses the scene graph. The class for the node representing the QtDataVisualization graph in the scene graph is named as DeclarativeRenderNode [71]. It is defined as a subclass of QSGGeometryNode, a node representing a QQuickItem that has visual content in the scene graph [72].

The DeclarativeRenderNode class creates the OpenGL frame buffer object where the whole graph scene is rendered [73, 74]. When it is time to render in the node's preprocess stage, the frame buffer is bound, and its OpenGL handle is passed to the controller's render function, which triggers the QtDataVisualization rendering [75]. After the rendering, the frame buffer is unbound. Finally, the window is made current for the Qt's OpenGL context, and the preprocessing stage for the node is completed. Then QtQuick 2 renderer continues the normal processing of the scene graph.

3.3. QtQuick3D

The QtQuick3D module can easily combine 3D with 2D [76]. The approach the module offers works on top of the QtQuick 2 framework. QtQuick3D provides a separate scene graph that keeps track of the objects in the whole 3D scene. That means two different scene graphs are involved when rendering UIs with 3D content: the 2D Quick Scene Graph (QSG) and the 3D Quick Spatial Scene Graph (QSSG) [77, 69]. To UI developers, working on top of the QtQuick 2 framework appears so that there is only one QQuickItem in the module, QQuick3DViewport [78]. When it is time to synchronize the state of front end UI items to the QSG, the QQuickWindow calls the virtual UpdatePaintNode function for every QQuickItem marked dirty [79].

The QQuick3DViewport's UpdatePaintNode is where the QtQuick3D renderer is set up, and the state of the 3D scene is synchronized via the scene manager to the QSSG [79, 80]. In QtQuick3D, the scene manager is responsible for tracking the state of front end 3D objects and ensuring that they are represented in the QSSG. Similar to 2D QQuickItem's updatePaintNode, 3D objects have a virtual updateSpatialNode function. When the scene manager synchronizes the scene's state, it calls that updateSpatialNode function for every 3D object marked dirty [81]. The updateSpatialNode function returns an object representing the state of the front end object in the QSSG.

Depending on the selected render mode for QtQuick3D, the updatePaintNode function returns either the QSG node containing the 3D renderer or, in the case of direct rendering, it returns nullptr [82]. The 3D scene is rendered in the node's preprocess stage in the former case [79]. However, in the latter, rendering happens due to QQuickWindow's beforeRenderPassRecording or afterRenderPassRecording signal [83, 84]. This, again, depends on the selected render mode.

3.3.1. Render Mode

In QtQuick3D, render mode can be set by setting the renderMode variable to one of the four possible values: Offscreen, Underlay, Overlay, and Inline [85]. When the renderMode is not explicitly set, it defaults to Offscreen rendering. In the default mode, the whole QtQuick3D scene is rendered to an offscreen surface. The surface is then used as a texture for a node in the QSG [86].

Both Underlay and Overlay are direct rendering modes where the whole 3D scene is rendered directly to the window containing the QQuick3DViewport [87, 88]. Their only difference is whether the scene be rendered before or after the rendering of the QtQuick 2 UI. When using Underlay, the render function is connected to QQuickWindow's beforeRenderPassRecording signal. Moreover, the 3D scene will be rendered before the QtQuick 2 UI. Hence, the UI is drawn over the 3D content. Overlay, as opposed to Underlay, is connected to the afterRenderPassRecording signal, which means it will be rendered after the QtQuick 2 UI. Hence, the 3D content is drawn over the UI.

The Inline rendering mode is the last of the four modes [89]. Like in the Offscreen render mode, a new node is created in this mode. However, the difference is that in the Inline mode, the new node can insert rendering commands in line with UI rendering. Hence, it does not render to an offscreen surface, which would then be used as a texture. Using Inline rendering mode can cause difficulties since it causes the QtQuick3D renderer to use the same depth buffer as the QtQuick 2 renderer. Moreover, the z-values are treated differently between QtQuick3D and QtQuick.

3.3.2. Front End

On the front end, the QQuick3DViewport is exposed to QML as View3D [90]. The View3D has all the visible 3D objects as children. Every type provided by QtQuick3D is either a direct or indirect subclass of Object3D, a class that adds functionality needed by the QtQuick3D's scene manager [91]. The types can be classified into two distinct categories: spatial and resource objects [81]. Spatial objects can be placed in the 3D scene, whereas resource objects organize configuration data. SceneEnvironment, for example, holds the data used to control how the entire scene should be rendered [92].

For something to be placed in a 3D scene, it must have some basic properties, such as position, rotation, and scale. The QQuick3DNode class provides these essential properties [93]. The QQuick3DNode is a base class for all 3D objects that should be placed in the scene. Even though a 3D object can be placed in a 3D scene, it does not mean the object is visible in the scene.

The model type in QtQuick3D represents a visible node. There are two necessary prerequisites for a model to be renderable. First, it needs to have a mesh. A mesh contains a model's vertex point data that, for example, tells the renderer the local coordinates of the points making up the model. Second, it needs to have a material [94]. A material provides information on how to shade the used mesh.

3.3.3. Materials

Currently, there are four different types of materials offered by QtQuick3D. The four types of materials are DefaultMaterial, PrincipledMaterial, SpecularGlossyMaterial, and CustomMaterial [95, 96, 97, 98]. The DefaulMaterial type is inherited from the older codebase and exists solely for compatibility. It combines physically based rendering (PBR) aspects with bits from the older specular/diffuse shading model. The material's three main properties are specular, roughness, and diffuse color [99]. As the DefaultMaterial type exists solely for legacy reasons, its use is not recommended. Instead of DefaultMaterial, the developers should use PrincipledMaterial or SpecularGlossyMaterial.

The two material types that offer proper PBR-pipeline are PrincipledMaterial and SpecularGlossyMaterial [100, 101]. The same realistic look can be achieved with either of them. Their difference, however, is in the workflow they offer. The workflow offered by PrincipledMaterial is referred to as metal/roughness workflow, whereas the other is called specular/glossiness workflow [102, 103].

The main difference between the workflows is in the three main properties used to control the material's appearance. In metal/roughness, the essential properties are metalness, roughness, and base color [102]. Similarly, in specular/glossiness, the main properties are specular, glossiness, and albedo [103]. In the first of the two workflows, the base color contains data used in the material's diffuse and specular properties. The metalness property affects the interpretation of the base color. When metalness is set to one, most of the base color is interpreted as specular data, and in the case of zero, most are considered diffuse data. The SpecularGlossyMaterial's albedo property differs from the PricipledMaterial's base color property so that it contains only the diffuse information of the material and not any reflectance data.

Among the four material types, the CustomMaterial offers the most freedom, allowing shader-level customization [104]. The CustomMaterial can use customized vertex or fragment shader code [105, 106]. If customized code is not provided for a shader, the shader will work like the PrincipledMaterial's corresponding shader [107].

3.3.4. Antialiasing Methods

QtQuick3D supports a variety of antialiasing methods. The supported methods are:

- Supersample antialiasing (SSAA)
- Multisample antialiasing (MSAA)
- Progressive antialiasing (PAA)
- Temporal antialiasing (TAA)
- Fast approximate antialiasing (FXAA) [108, 109]

All those methods have different tradeoffs between quality and performance. For example, SSAA provides superior quality but is quite expensive for performance [110]. It first renders the whole scene to a texture bigger than the target size, and then the rendered image is downsampled to the target size. Using SSAA to antialias the scene will decrease performance and increase resource usage. Therefore, it should be used only as a last resort. MSAA, on the other hand, also offers excellent quality

but requires less performance than SSAA [111]. This is because it targets only the edges of geometries and thus requires fewer computations. Besides, MSAA is usually implemented on graphics hardware. In PAA, once the scene stops changing, it is rendered multiple times, and each time the rendered frame is blended with the previous one [112]. However, the scene is rendered between the frames from slightly different positions, which is achieved by jiggling the camera. The downside of PAA is that it needs the scene to stop changing. Besides, for the final frame, it needs to render the scene multiple times. For example, if the maximum settings for antialiasing are used, a total of eight frames are blended [113]. TAA works the same as PAA; however, it only re-renders the scene once and can antialias changing scenes [114]. In TAA, as opposed to PAA, the amount of movement adjustment applied to the camera can be controlled via the SceneEnvironment's temporalAAStrength property [115]. The FXAA implementation differs from the other antialiasing techniques as it is not implemented in QQuick3DRenderer. Instead, it is provided as a ready-made post-processing effect [116].

3.3.5. 2D Items in 3D Scene

Sometimes 2D items need to be displayed in a 3D scene, for example, a text label showing a changing number. Therefore, QtQuick3D enables displaying QtQuick 2D items in a 3D scene [117]. There are two different ways to get 2D content displayed in a 3D scene.

The first is to declare a 2D item as a child of a 3D node [118]. In this option, the child 2D item is wrapped into a QQuick3DItem2D object [119]. The QQuick3DItem2D is just a 3D container for the 2D items. When the SceneManager synchronizes 3D objects, QQuick3DItem2D sets up a new sub-renderer in the 3D mode for its branch of the 3D scene [120]. In the 3D mode, the sub-renderer can use the same transform and render target for rendering as the parent 3D node [121].

The second approach is to render the QtQuick 2D scene to a texture [122]. The texture can then be used as an input for a material. The size of the top-level QQuickItem defines the size of the texture.

3.3.6. Scene Rendering

Once the state of the front end is synchronized to the QSSG, the rendering of the scene can start. This process is divided into separate stages [123]. In the first stage, the render target gets set [124]. The actual render target can vary depending on the used render mode. After that, global states, such as viewport size, clear color, and scissor rectangle, are set. After the render target and some of the global states are set, the process moves to the second, the preparation stage [125]. In reality, the stage is separated into two phases.

The first phase is high-level preparation for rendering [126]. In that phase, the QSSG is processed, making it easy to create meaningful render commands. In the beginning, various lighting settings are resolved. This includes, for example, whether the Screen Space Ambient Occlusion (SSAO) is enabled or whether a new environment

map has to be generated. Moreover, how many lights does the graphics driver support? Further, the information on the maximum number of light sources is used when every contributing light source is collected. After that, every node and its children are sorted into different lists depending on what kind of nodes they are. That is followed by finding out which camera is used for rendering. If there are no active explicitly set cameras, the camera found first is used. The selected camera is used to calculate the view-projection matrix. Then it is time to prepare every model, particle, and item2D for rendering. For models, this includes, for example, loading the required meshes and preparing their materials for rendering. For particles, this includes loading the used textures and item2Ds to get their model-view-projection matrices calculated. When these 3D objects are prepared, they are also inserted into various lists. The list where the object is inserted depends on its properties, such as opacity or whether the object requires a screen texture. At the end of the first phase, the reflection probes are prepared if the scene contains them. Also, if PAA or TAA are enabled, specific variables are set to do those antialiasing methods [127, 128].

The second phase of the preparation stage is low-level preparation for rendering [129]. The phase includes preparing every activated render pass for the current frame. Also, every render pass should be rendered before the main pass is rendered. After every render pass preceding the main render pass is rendered, the main render pass is prepared. The preparation of the main render pass includes setting up various uniform buffers, as well as associating samplers with textures. In addition, all transparent, opaque, and objects that need screen texture are sorted. Opaque objects are sorted from nearest to furthest away from the camera, while transparent objects are sorted from furthest to nearest. Objects that require a screen texture are sorted from furthest to nearest. Setting up the main render pass concludes the preparation stage.

Once everything is prepared for rendering, the only remaining mandatory step is to record render commands for the main pass [123]. The Qt's graphics abstraction layer then executes these commands. First, the commands for rendering opaque objects are recorded [130]. That is followed by optional skybox rendering. The third is to render objects that depend on the screen texture. Then every 2D item is recorded. The next step is to record every transparent object. And finally, the infinite grid, if it is enabled. Depending on whether debugging is enabled, one more step can record commands for rendering the debug content.

After rendering the scene comes the post-processing stage. During the post-processing stage, if PAA, TAA, or SSAA is enabled, they are applied after the user-provided post-processing effects [131]. The post-processing effects are applied to the whole rendered image. Therefore, offscreen renderMode needs to be used when using post-processing effects.

3.4. Graphics API Abstraction Layer

Qt supports a variety of graphics backends. Internally this is achieved by using an abstraction layer for accessing the graphics API. This abstraction layer is called the Qt Rendering Hardware Interface (QRHI) [42].

The first preview of QRHI was added in Qt 5.14 version [132]. Back then, it was not the main graphics abstraction but an optional feature. Then with the release of

Qt6, it became the main graphics abstraction used by Qt [133]. The motivating factor behind the QRHI was to move away from direct OpenGL usage since it is not the recommended graphics API on every platform anymore [134].

The QRHI supports four graphics APIs: OpenGL, Vulkan, Metal, and D3D11. The QRHI selects the most suitable API for each platform among these graphics APIs. On Windows, it selects D3D11 [135]. On Apple's platforms, the default is Metal, and on Linux platforms, the default is OpenGL. However, application developers can request using a different graphics API than the platform's default API [136]. One example of when the API needs to be explicitly requested is when the application directly uses a specific graphics API, as with QtDataVisualization [137].

Even though Vulkan can be considered a successor to OpenGL, OpenGL is still a very capable and widely used graphics API; as such, it is still best to default to use it on Linux platforms. However, when Vulkan matures and more devices start to have capable Vulkan drivers, it can be expected to replace OpenGL as the go-to graphics backend on Linux.

As OpenGL is deprecated on Apple's platforms, it does not get any more updates. Instead, Metal should be used on those platforms. Hence, the QRHI defaults to Metal. However, OpenGL can still be used on Apple's platform if requested, although it will lack the newest features.

D3D11 is still quite capable and popular graphics API on Windows, although a newer D3D12 is available. However, the QRHI supports only D3D11 and therefore defaults to D3D11. Currently, there has not been a real need for supporting D3D12 in the QRHI. However, as time goes by and more and more devices start to support D3D12, it may become relevant to support D3D12 as well.

3.4.1. Qt5 Vs Qt6 Graphics

In the Qt5 era, the rendering relied heavily on direct OpenGL usage. This approach worked fine when a device had properly working OpenGL drivers. However, on Windows, the out-of-the-box installation lacks proper OpenGL drivers. Nevertheless, Qt tackled that problem by using ANGLE, which was discussed in the Motivation chapter [28].

On macOS and iOS, OpenGL drivers are included, albeit they have started to be outdated. In the long run, it would have impacted the performance on those platforms. Besides, it could have started to affect other platforms as well since OpenGL would have been locked to the 4.1 version.

One solution to the problem of having outdated OpenGL drivers on macOS and iOS would have been to use a translation layer that translates OpenGL calls into Metal API calls. One such layer is MoltenGL [138]. However, using third-party translation layers solves only half of the problem, as they might be locked to older OpenGL standards. MoltenGL, for example, uses OpenGL ES 2.0. Besides, having Qt rely heavily on third-party solutions would also bring other, for instance, license-related problems [134]. In addition, a translation layer would have to provide access to the underlying native objects, as Qt allows an application to provide its rendering code.

Only the open-source OpenGL Mesa drivers are installed on Linux platforms by default. These Mesa drivers use the software implementation of OpenGL API.

However, hardware acceleration is possible by installing a proprietary driver from the GPU vendor, which is not installed by default.

Having additional third-party libraries as dependencies also brings an extra maintenance burden. This is because Qt is not the one who decides the development direction of those libraries. Libraries, for example, can have updates that break the compatibility with Qt. Even if the libraries were left without an update, it would eventually hinder Qt's development.

3.4.2. QRHI Implementation

The QRHI is implemented as a set of base classes. Those base classes are then implemented for every supported graphics API, such as D3D11. The D3D11 implementation of the QRHI has about 13 classes: one main class for the D3D11 implementation, and the others are resource classes [139, 140]. Through resource classes, the backend's native resource objects can be used.

There is one class named QRhi, which is a top-level class in the sense that it is used by classes that use QRHI, such as QtQuick3D renderer [141]. The top-level QRhi class provides all the functions needed to render graphics, such as the beginFrame function, which starts a new frame [142]. These upper-level functions do nothing alone; instead, they call functions on the QRhi implementation to perform the corresponding graphics operations [143].

The main level QRhi class contains a pointer to the implementation of the QRHI backend, such as D3D11. This QRhi implementation class handles everything related to using the graphics API, such as creating various resource objects and recording commands to the command buffer.

There are 12 different resource classes in the QRHI [144]. Every resource has its corresponding implementation for a QRHI backend implementation. Resources are used through their parent classes. However, their API-specific functionalities are implemented in the subclasses. API-specific functionalities are, for example, creating and destroying a resource. For example, the QRhiBuffer class represents a buffer resource [145]. A buffer resource contains data, such as vertex data, that can be used by rendering hardware.

In the QRHI D3D11 implementation, there is a subclass of QRhiBuffer named QD3D11Buffer [146]. The QD3D11Buffer class implements five functions declared in its parent classes: create, destroy, nativeBuffer, beginFullDynamicBufferUpdateForCurrentFrame, and endFullDynamicBufferUpdateForCurrentFrame [147, 148, 149, 150, 151].

Among these functions, the first two have to be implemented by every subclass of QRhiBuffer as they do not have default implementations. Destroy is first declared in the QRhiResource class, the base class for every resource, and create is declared in QRhiBuffer [152, 153]. The other three functions have default implementations in QRhiBuffer [154, 155, 156]; however, they do nothing. Currently, every supported backend implementation overrides the default implementations.

3.4.3. Initializing QRHI

The top-level QRHI class cannot be instantiated directly, but the users need to call the static create function [157, 158]. The function takes a choice on which graphics API to use as an argument. At the start, it instantiates the QRhi object and then the QRhi backend implementation. A QRhi implementation does not initialize graphics API in its constructor. The initialization of the graphics API happens in the create function of the QRhi implementation. The top-level QRHI create function returns a pointer to the instantiated QRhi object, or in an error case, such as a graphics API initialization failure, it returns a nullptr.

The design of QRHI does not allow QRHI to implement any fallback logic in cases where the requested graphics API does not exist. This means that the initialization of QRHI will fail and cannot be used. Clients like QtQuick will have to implement such fallback logic themselves.

3.4.4. Shader Handling

Shaders are handled by the QtShaderTools module [159]. While the QtShaderTools module does not belong to QRHI, it is used by QRHI for platform-agnostic shader handling.

Every shader needs to be written in a single language [160]. Currently, the language is Vulkan-compatible GLSL. This shader code is then compiled to SPIR-V bytecode using the glslang tool [161]. In addition, the shader's reflection data is generated using the SPIRV-Cross tool [162]. The reflection data contains information on what shader attributes the shader exposes. After this, the SPIR-V bytecode is translated using SPIRV-Cross to shader code accepted by the used graphics API. These shader generation process results are packaged and stored on the disk. This enables graphics API an agnostic way for shader handling.

4. IMPLEMENTATION

This thesis work's primary goal was to lay the foundation for developing a new version of QtDataVisualization. The new version uses QtQuick3D to present the graphs. In this thesis, the scatter graph type was implemented. In addition, the general graph features that apply to the implemented graph type were implemented. One such feature is the range gradient material.

The secondary goal was to design the initial version of the more extensive public C++ API for the QtQuick3D module. The design was done by drawing the UML class diagrams for the used QtQuick3D classes.

4.1. QtDataVisualization Using QtQuick3D

The implementation was mostly done using C++ programming language. In addition, some features needed to be implemented using QML since they require additional setup before they can be used. The range gradient material, for example, uses QQuick3DCustomMaterial, which shaders are modified by the QtQuick3D when the QML component is instantiated.

It was essential to keep QtDataVisualization's QML API intact. This way, the users will have a smooth transition to the new version as they can use it just like the old version. This gave the basic requirements for the project: implement the same features without the current OpenGL rendering engine. The top-level architecture of how different modules fit into the bigger picture is depicted in Figure 6.

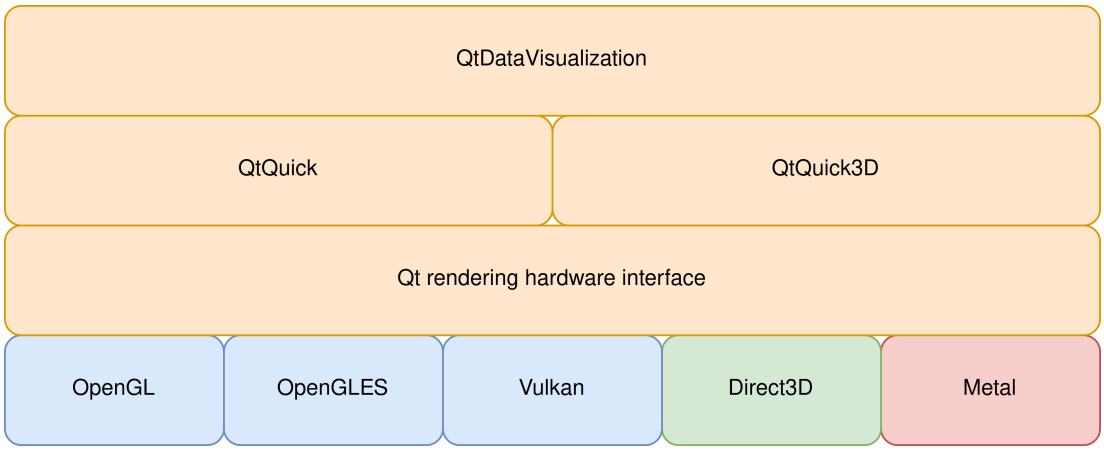

Figure 6. The new version uses QtQuick3D to display 3D content.

The project was concerned with how QtDataVisualization communicates with QtQuick3D. Since the current version of QtDataVisualization uses an internal controller class to communicate with its custom OpenGL rendering engine, it had to be changed to communicate with QtQuick3D. That was done using QtQuick3D's 3D objects; when the front end synchronizes its state to the back end, the QtDataVisualization's controller class updates the state of QtDataVisualization to the QtQuick3D objects.

4.1.1. General Graph

The general graph elements, such as the main viewport for the graph, background, camera, lighting, grids, labels, and input handling, are discussed first. The main viewport for the graph was done by setting the QtDataVisualization graph type to inherit from QQuick3DViewport. Inheriting from QQuick3DViewport enables easy access to the main 3D viewport. After the main viewport is created, the other elements like background, camera, light, and repeater3Ds for grid lines and labels are created.

For the background, one 3D model which represents the background model was created. In addition, three more nodes had to be created. Those nodes were for rotation, scale, and bounding box, one for each. The rotation and scale nodes were created to control the order of the matrix application since the old QtDataVisualization applies them in a different order than QtQuick3D. This way, the same rotation calculations can be used. The bounding box was created for the input handling, which will be discussed later.

When the camera is set up, the graph checks whether the camera should be a perspective or an orthographic camera. The selected camera type is then created. In addition to camera creation, a 3D node is created and set as a parent for the camera. That 3D node acts as a target for the camera. The target node is used to orbit the camera around the target point easily. When the camera needs to rotate around the target, only the node needs to be rotated. Besides, when the camera needs to be set to look at a specific point, the target can be moved to that point, and the camera can be rotated to face that point. Zooming the camera closer and further is done by moving the camera along the z-axis, which will move in its parent's coordinate space.

The initial lighting setup uses directional light, which is set as a child of the camera. The light is set as a child of the camera because the light needs to rotate with the camera. The light's other properties are updated according to the QtDataVisualization settings while the front end and back end synchronize.

There are many same kinds of objects in the grid lines and labels. Since the same object needs to be repeated many times, it is a perfect case for a repeater3D type. The grid line and label types are implemented by creating a QML type. Those types are given to repeater3D type that will instantiate the needed amount and set itself as their parent.

The components from the old QtDataVisualization could be used for input handling. That is because most of the components related to the input handling modify objects from QtDataVisualization, and the properties of QtQuick3D objects are updated from the QtDataVisualization objects during the synchronization. However, features such as zooming to position on the background and selecting a data point in the graph needed extra work. The implementations of these features were added to the controller classes. The zooming works so that when the Qt runtime generates the wheel event, the QtDataVisualization's input handling component stores the event's screen position. That position is then used during the synchronization to do a ray cast to figure out the position on the graph background. For this to work, the background bounding box was added. The selection of an item, does the ray cast right away to see if an item was clicked.

4.1.2. Scatter Graph

In implementing the scatter graph, it had to be decided how to display the graph's data points in the 3D scene. The displaying of the data points was implemented by creating a new 3D node for every data point. Every data point in one series uses the same mesh. However, their other properties, such as position or color, can change. As a solution to this problem, a specific visualizer class was added. The class handles the creation and updating of 3D nodes used to visualize a series in the graph.

When a series is added to the graph, the controller class creates a new series visualizer for the added series. There are differences in the operation of a series visualizer depending on whether default or static optimization is used. When using default optimization, one 3D node is created for every data item. Static optimization, however, is supposed to render every data item with a single draw call and therefore needs to use instanced rendering. Regardless of the used optimization mode, data items' properties are updated while the front end and back end synchronize.

First, the data item position, rotation, and scale are updated. Next comes the updating of the visual properties of items. Updating visuals requires more complex logic than the previous spatial update. Complexity increases because a data item needs to use a different material depending on whether the individual data items use normal color, plain texture, or color from gradient texture.

A PrincipledMaterial type can be used when a data item is colored with a constant color or texture. However, the range gradient material mentioned earlier is used when color from gradient texture is used. The range gradient material uses Quick3D's custom material as it uses a customized shader that samples a color from a gradient texture. The sample location depends on the data point's position on the graph. The sampled color is then used to color every fragment of the data point.

The range gradient material differs depending on whether default or static optimization is used. Therefore, two QML types needed to be implemented: one for default optimization and one for static optimization. That is because the former needs only a customized fragment shader, whereas the latter needs both a customized vertex and fragment shader. When using static optimization, the range gradient material needs to use custom vertex and fragment shader because the static optimization uses QQuick3DInstancing. The QQuick3DInstancing's shaders need to be given instance-specific values. Basic values include position, rotation, scale, and color. In addition to the basic values, custom values can also be given as a four-dimensional vector. This custom data is passed to the vertex shader, which either uses it or passes it to the fragment shader.

4.2. The Public C++ API for QtQuick3D

As the secondary goal of this thesis, an extension for QtQuick3D's public C++ API was designed. The design enables developers to develop QtQuick3D applications through C++ with minimal use of QML. This is an initial design because it only covers part of the QtQuick3D. However, it covers all the QtQuick3D classes related to the implementation of the primary goal of this thesis. Below is a list of the classes related to this project.

- QQuick3DNode
- QQuick3DModel
- QQuick3DDirectionalLight
- QQuick3DMaterial
- QQuick3DPrincipledMaterial
- QQuick3DCustomMaterial
- QQuick3DRepeater
- QQuick3DViewport
- QQuick3DSceneEnvironment
- QQuick3DTexture
- QQuick3DPickResult
- QQuick3DCamera
- QQuick3DPerspectiveCamera

These types must provide at least the same functionality in the public C++ API as in the QML API. Many of these classes have the needed functionality in their public interface. Therefore, it would be sufficient to move them into the public C++ API and have their private interfaces contain a pointer to their internal implementation class. However, more than the described process is needed with some of the classes. That is because they contain a list as in QQuick3DModel or are modified by QtQuick3D when instantiated, as with QQuick3DCustomMaterial. The following lists those classes.

- QQuick3DModel
- QQuick3DCustomMaterial
- QQuick3DSceneEnvironment

For example, the QQuick3DModel would have to be designed as depicted in Figure 7. In addition to the design of the public class, the private class is added to the diagram to show that the actual properties are indeed added to the private class. However, the private class must contain more functionality than just the properties. The additional functionality is omitted here because it belongs to the private API.

Via the public class, users could access every property like in the current QML API. However, those properties would be moved into the private class instead of having them in the actual QQuick3DModel class. Currently, the model class has most of its properties in the private interface, which can be accessed through its public interface. However, as stated earlier, a list needs extra work. That is because simply gaining access to the list through a getter or setter is not enough since it needs to perform extra work on the types, for example, connecting to a signal. In this design, the same four list-handling functions were added as required in the QML API. Those functions can act on the list by adding an element, querying element count, querying an element in a specific position, and clearing all the elements.

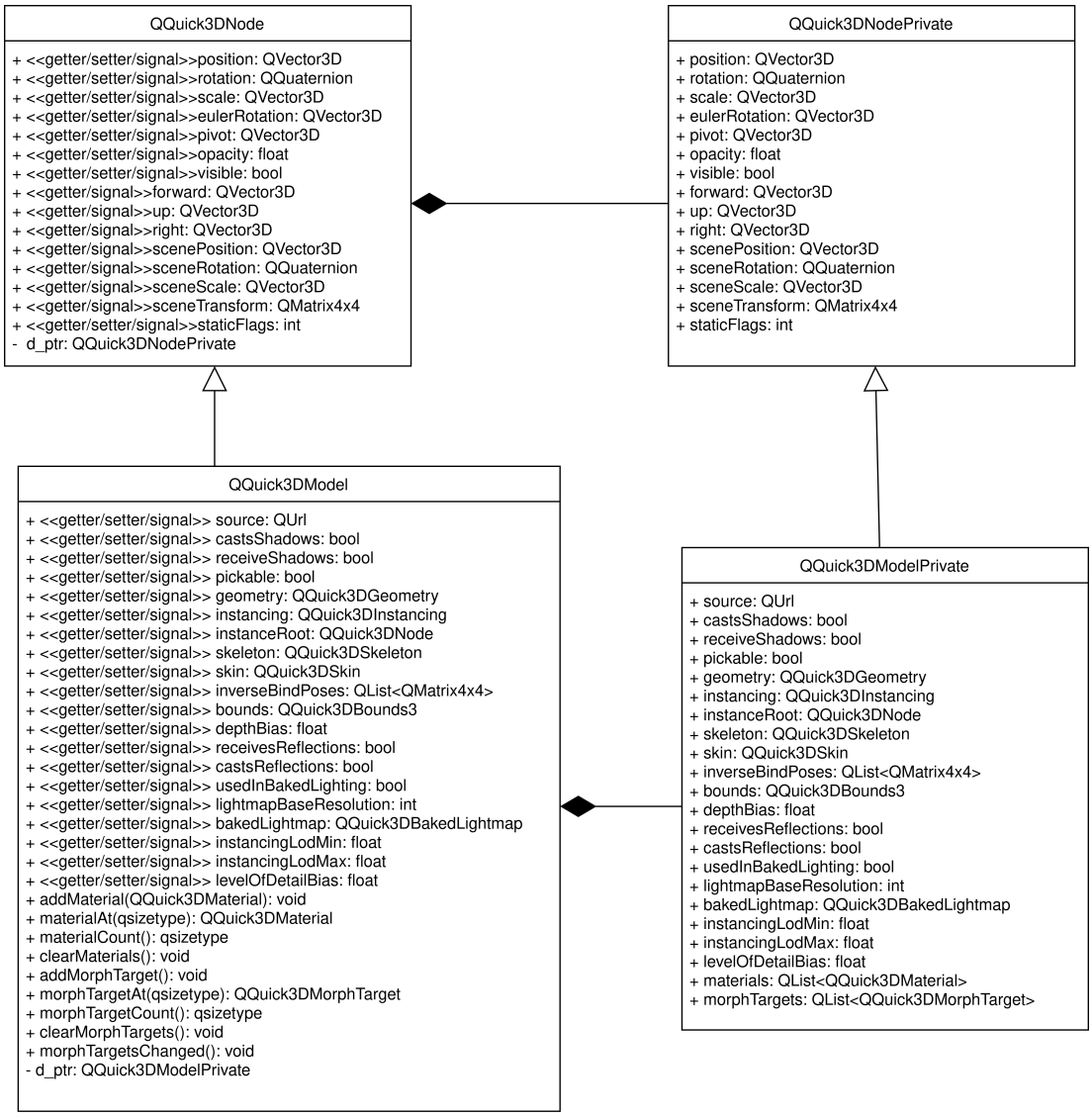

Figure 7. Design of public class for QQuick3DModel.

5. EXPERIMENTS

This thesis had primary and secondary goals. The primary goal was to explore how to change QtDataVisualization to use QtQuick3D's renderer instead of the currently used custom OpenGL renderer. In practice, this meant building and displaying QtDataVisualization graphs using QtQuick3D's objects. The secondary goal was to do the initial design for an extensive public C++ API to QtQuick3D. As the design was the secondary goal, it only needed to cover the classes used to implement the primary goal.

5.1. QtDataVisualization Using QtQuick3D

Three requirements were set for the primary goal: a reasonable performance, the same QML API, and the final graphs should look roughly the same. A new application was developed specifically for measuring the performance, whereas the look of the final graph with the current version and the thesis work was compared visually. The QML API was not tested in this work; nevertheless, it had to be considered.

5.1.1. Performance Measurement

All the designed test cases were run in the application build using the current version of QtDataVisualization and the version developed for this thesis. The datapoint count was tested on a hardware setup using an integrated graphics solution.

Originally it was planned to measure theme time so that when the theme was changed, it was planned to measure how long it took for the changes to become visible. However, that turned out to be quite tricky. One solution that was tried was too unstable. Therefore, the test case was changed so that the change in FPS count was inspected instead of measuring the time. The following is a list of the planned test cases:

- FPS measurement.
- Adding and removing a series.
- Startup time measurement.
- Measuring how theme change affects the FPS.
- Maximum count of displayable datapoints in static optimization.

The FPS was measured by having the test scene rotate at a constant speed and displaying the FPS count. It was mandatory to have the scene constantly changing since QtQuick3D does not render an unchanged scene. Adding a new series was measured by timing how long it took to add a new series to the graph. Removing a series was measured by timing how long it took to remove the last series from the series list. Adding and removing series have minor changes between the old QtDataVisualization and the thesis work. Therefore, that test case was paid less attention than the others.

The startup time was measured by starting a timer when the QML scene object was created. Then, the next time QQuickWindow swapped frames, the timer was

stopped. That time tells how long it took to get the graph on the screen. The startup time was measured multiple times, and then the average time was calculated. Theme changing was measured by changing the theme multiple times and inspecting how the FPS changes. Finally, the maximum datapoint count was measured by adding one series with many datapoints. The number of datapoints to be added to the series was increased so that the application would not start anymore. The application was considered not to start if it crashed or hung up during startup. First, the maximum point count for the old QtDataVisualization was tested. That point count was used as a starting count with the thesis work.

5.1.2. Visual Evaluation

The visual evaluation of the graph was conducted by building the same graph application with the current QtDataVisualization and with the thesis work. A screen capture was then taken from those applications. The resulting screen captures were compared visually to inspect the differences between the modules.

5.1.3. Test Hardware

The performance was measured on three different hardware setups in a Windows environment. The first setup was a desktop computer with a quadcore i7-7700K with a clock frequency of 4.20 GHz. In addition, the computer had NVIDIA GeForce GTX 1080 GPU. The second setup was a laptop computer using an eight-core i7-11850H CPU at a clock frequency of 2.50 GHz. The setup used an integrated Intel UHD graphics solution. The third setup used the same CPU as the second one but it used a dedicated graphics solution instead of an integrated one. The GPU used by the third setup was NVIDIA RTX 3070 laptop GPU.

Only OpenGL could be used when running the test application with the current QtDataVisualization module. However, running the test application with the thesis work enabled using three different graphics backends: OpenGL, D3D11, and Vulkan.

5.1.4. Results

In the result tables below, NG refers to the work developed in this thesis.

Table 2. FPS on RTX 3070 using static optimization with 2000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	40	177	174	60
max	185	186	184	60
avg	100	186	182	60

Table 3. FPS on RTX 3070 using static optimization with 6000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	23	179	181	60
max	184	185	184	60
avg	66	182	181	60

Table 4. FPS on RTX 3070 using static optimization with 10000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	13	183	179	60
max	182	186	183	60
avg	44	183	180	60

Table 5. FPS on RTX 3070 using default optimization with 2000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	73	4	43	40
max	179	169	169	60
avg	147	62	120	49

Table 6. FPS on RTX 3070 using default optimization with 6000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	64	1	1	1
max	167	53	54	45
avg	125	5	4	9

Table 7. FPS on RTX 3070 using default optimization with 10000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	64	1	1	1
max	162	30	31	25
avg	126	2	2	4

Table 8. FPS on integrated UHD graphics using static optimization with 2000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	46	60	58	60
max	60	61	60	60
avg	51	60	59	60

Table 9. FPS on integrated UHD graphics using static optimization with 6000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	16	60	59	60
max	49	60	60	60
avg	25	60	60	60

Table 10. FPS on integrated UHD graphics using static optimization with 10000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	9	40	40	45
max	33	46	42	55
avg	15	43	41	49

Table 11. FPS on integrated UHD graphics using default optimization with 2000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	44	1	1	6
max	60	47	60	60
avg	54	6	17	21

Table 12. FPS on integrated UHD graphics using default optimization with 6000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	32	1	1	1
max	52	47	42	35
avg	44	6	7	4

Table 13. FPS on integrated UHD graphics using default optimization with 10000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	30	1	1	1
max	37	28	25	25
avg	33	2	2	4

Table 14. FPS on GTX 1080 using static optimization with 2000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	41	56	180	60
max	182	61	185	60
avg	97	58	183	60

Table 15. FPS on GTX 1080 using static optimization with 6000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	20	60	183	60
max	184	60	185	60
avg	55	60	184	60

Table 16. FPS on GTX 1080 using static optimization with 10000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	15	60	183	60
max	183	60	184	60
avg	44	60	183	60

Table 17. FPS on GTX 1080 using default optimization with 2000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	71	16	2	8
max	179	60	170	152
avg	147	34	46	48

Table 18. FPS on GTX 1080 using default optimization with 6000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	72	1	1	1
max	171	51	56	43
avg	143	5	3	6

Table 19. FPS on GTX 1080 using default optimization with 10000 datapoints

	Old	NG vulkan	NG OpenGL	NG D3D11
min	53	1	1	1
max	140	27	33	24
avg	99	2	2	4

Table 20. Startup times in milliseconds on RTX 3070 using static optimization

Point count	Old	NG Vulkan	NG OpenGL	NG D3D11
2000	6	3	168	6
6000	4	3	143	6
10000	5	4	151	6

Table 21. Startup times in milliseconds on RTX 3070 using default optimization

Point count	Old	NG Vulkan	NG OpenGL	NG D3D11
2000	6	73	62	82
6000	8	416	371	440
10000	47	1109	1040	1211

Table 22. Startup times in milliseconds on integrated UHD graphics using static optimization

Point count	Old	NG Vulkan	NG OpenGL	NG D3D11
2000	42	5	30	8
6000	259	6	28	8
10000	350	5	30	12

Table 23. Startup times in milliseconds on integrated UHD graphics using default optimization

Point count	Old	NG Vulkan	NG OpenGL	NG D3D11
2000	21	78	81	94
6000	32	419	439	438
10000	39	1115	1076	1134

Table 24. Startup times in milliseconds on GTX 1080 using static optimization

Point count	Old	NG Vulkan	NG OpenGL	NG D3D11
2000	266	N/A	106	N/A
6000	421	N/A	106	N/A
10000	554	N/A	107	N/A

Table 25. Startup times in milliseconds on GTX 1080 using default optimization

Point count	Old	NG Vulkan	NG OpenGL	NG D3D11
2000	159	N/A	328	N/A
6000	161	N/A	1105	N/A
10000	163	N/A	2161	N/A

Table 26. Maximum point count on integrated UHD graphics using static optimization.

Max pointcount	
old	NG D3D11
1200000	1300000

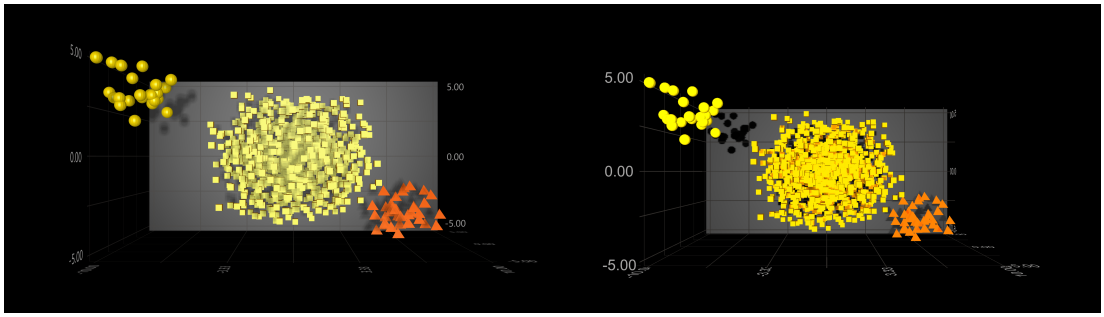

Figure 8. Scatter graph side-by-side comparison. The old one is on the left, and the thesis work is on the right.

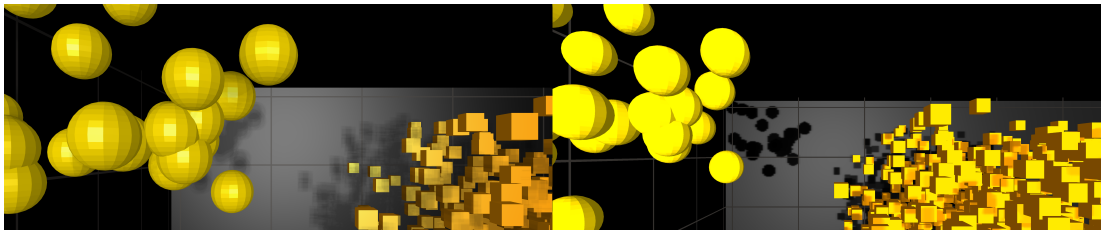

Figure 9. Scatter graph closeup side-by-side comparison. The old one is on the left, and the thesis work is on the right.

6. DISCUSSION

The most significant benefit of this work is that it sets the base for developing the new version of the QtDataVisualization module. Changing the QtDataVisualization to use QtQuick3D helps to reduce the number of different rendering engines in Qt. Moreover, the OpenGL rendering engine used in QtDataVisualization was explicitly made for that module, meaning it had no use outside QtDataVisualization.

Since QtDataVisualization needs to work on older OpenGL ES 2.0 devices, the OpenGL feature set was locked, meaning it could not use newer features even if they were available. However, thanks to QtQuick3D, which uses QRHI, QtDataVisualization can benefit from newer features as QRHI queries their availability.

The benefits gained do not remain merely in newer features for the graphics backend. This is because QtQuick3D works on a variety of graphics backends. Therefore, the QtDataVisualization version developed in this thesis will work with different graphics backends. That is quite an improvement compared to the old QtDataVisualization, which worked only on OpenGL.

6.1. FPS

The difference between the old and the version developed in this work can be seen in the test results. The results show that the old is still clearly better than the version developed in this work when using the default optimization mode. Nevertheless, the results hint at the code path where to direct optimization efforts. Although the results look bleak for the default optimization, they do not differ as much when using the static optimization mode. However, the version developed in this work is more flexible with static optimization than the old QtDataVisualization.

The old static optimization puts every data-point mesh to the same buffer and submits one huge buffer to OpenGL. In comparison, the new static optimization uses instanced rendering. Instanced rendering allows datasets to be more dynamic as every datapoint does not have to be crammed into the same buffer on the client side. However, their location and other custom data are given in a separate buffer.

The performance benefits of the new static optimization can be seen in the results Table 4, Table 10, and Table 16. In the results of the integrated graphics solution, it can be seen that the lowest FPS was 9 with the old QtDataVisualization. In contrast, the lowest with the thesis work was 40, depending on the used graphics backend. In addition, with the old QtDataVisualization, the highest FPS was 33, but it was between 42 and 55 on the thesis work. This again depends on the used graphics backend.

Although instanced rendering improves static optimization, it also drops OpenGL ES 2 from the list of supported graphics backends. That is because OpenGL ES 2 does not have instanced rendering feature. However, since the instanced rendering improves performance on many devices, its use is justified. For the devices that have only OpenGL ES 2 available, some other solution needs to be implemented.

6.2. Startup Time

A similar trend as in the FPS results can be seen in startup time results. The results show that when using default optimization with the thesis work, the startup time increases drastically with the number of datapoints. However, when using the old QtDataVisualization, the startup time increases slower. The results are more stable with static optimization; however, as seen from Table 22, the startup time can take longer on low-end devices with the old version than with the thesis work. Table 20 suggests no significant differences between the old and the thesis work on high-end devices.

Looking at Table 20 and Table 22, it is interesting to notice that the startup times are longer on RTX 3070 with thesis work using OpenGL backend. In comparison, the other backends seem to be slightly faster. However, it must be kept in mind that these results are obtained on the Windows platform, where the default backend is D3D11, and the tables do not show considerable differences in D3D11 results.

The startup time results on three different test devices in Table 20, Table 22, and Table 24 indicate that OpenGL backend performance is worse on the newer NVIDIA GPU; however, that might also be a random occurrence in the test results. The results also show that the integrated graphics solution would be the fastest. That might be because the UHD graphics uses shared memory, and the memory transfers are executed faster since they do not need to travel over the external bus.

6.3. Visual Evaluation

The graph's visual appearance does not precisely match the original. The colors are a bit stronger with the thesis work, which can be noticed by looking at the datapoints in the middle. Besides the colors of the datapoints, the stronger visuals can be seen in the shadows caused by the leftmost sphere-shaped datapoints.

The visual appearance of the graph is affected depending on the shaders used. The old QtDataVisualization module uses simple Phong shading, whereas QtQuick3D's shaders use a PBR pipeline. One solution to this problem would be to use QtQuick3D's custom materials instead of its PBR materials. However, changing all the materials to use custom materials would require writing custom shaders for the materials too. Furthermore, even if the scene lighting would be fixed using custom materials, the shadows could only be fixed by adjusting the properties, as QtQuick3D does not provide a way to modify the shader code for shadow pass. Nevertheless, using custom materials could make the shaders more lightweight, thus improving the performance, which is a reasonable justification.

7. CONCLUSION

This Master's thesis investigated how to develop a new version of QtDataVisualization, which uses the QtQuick3D module to display 3D graphs. The results of this project will work as the foundation for the new version of QtDataVisualization called QtGraphs. In addition, this thesis' secondary goal was to design a more extensive public C++ API for QtQuick3D. That design would work as the base for possible future development.

This work implemented the main parts needed to display the QtDataVisualization scatter graph using QtQuick3D. In addition to the scatter graph, other general graph mechanisms used by all graphs were implemented. Such mechanisms include, for example, camera controls and background rendering. The old QtDataVisualization allows using pixel-sized points as datapoints for the graph. However, this feature was not implemented since QtQuick3D does not offer such a feature.

When the scatter graph implementation was ready, it was tested with a benchmark application. On the benchmark application, tests were conducted, and performance data was collected during execution. The same kind of data was collected using the old QtDataVisualization also. The data were collected on a variety of hardware setups.

The benchmark data showed that the old version is better when using the default optimization mode, but the thesis work is better when static optimization is used. In addition, the results indicated that when using the OpenGL backend, the performance has decreased on newer NVIDIA graphic cards. That, however, would require more investigation in the future.

7.1. Future Work

Since this work was just the beginning of developing a new module, there is still much to implement. First, this study did not investigate how to implement a surface or bar graph type. Those graph types are still essential to implement in this new QtDataVisualization since they have been in QtDataVisualization since 1.0. Therefore, without them, the first test version cannot be included in Qt.

In addition, this work needs to be optimized, especially regarding the default optimization, as the current performance is not suitable. One way to optimize would be to use instanced rendering with default optimization. However, that would cause problems with the OpenGL ES 2 backend since it lacks the instanced rendering feature.

Besides implementing and improving features, the work needs to be tested on Linux and Apple platforms. Testing this on Apple platforms would be especially important since it is the only platform that uses Metal backend, which, being Apple only, was not tested.

8. REFERENCES

[1] Marschner S. & Shirley P. (2016) Fundamentals of Computer Graphics: Edition 4. CRC press.

[2] Wright H. (2007) Introduction to Scientific Visualization. Springer London.

[3] Gregory J. (2014) Game Engine Architecture, Second Edition. CRC Press.

[4] Beane A. (2012) 3D Animation Essentials. Sybex.

[5] Hyde R. (2004) Write great code, Volume 1: Understanding the Machine. No Starch Press.

[6] Luna F D. (2016) Introduction to 3D Game Programming with DirectX 12. Mercury Learning.

[7] Singh P. (2016) Learning Vulkan. Packt Publishing.

[8] Clayton J. (2017) Metal Programming Guide: Tutorial and Reference via Swift. Addison-Wesley Professional.

[9] History of opengl. URL: https://www.khronos.org/opengl/wiki/History_of_OpenGL. Accessed 26.8.2022.

[10] Opengl arb to pass control of opengl specification to khronos group. URL: https://www.khronos.org/news/press/opengl_arb_to_pass_control_of_opengl_specification_to_khronos_group. Accessed 3.8.2022.

[11] The OpenGL® Graphics System: A Specification. URL: https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf.

[12] Fixed function pipeline. URL:https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline. Accessed 26.8.2022.

[13] Opengl es. URL:https://www.khronos.org/opengl/wiki/OpenGL_ES. Accessed 26.8.2022.

[14] Pulli K., Aarnio T., Miettinen V., Roimela K. & Vaarala J. (2008) Mobile 3D Graphics : With OpenGL ES and M3G. Elsevier/Morgan Kaufmann Publishers.

[15] Opengl es 2.0. URL:https://web.archive.org/web/20101228111715/http://www.khronos.org/news/press/releases/finalized_opengl_es_20_specification/. Accessed 3.8.2022.

[16] More on Vulkan and SPIR-V: The future of high-performance graphics . URL: https://www.khronos.org/assets/uploads/developers/library/2015-gdc/Khronos-Vulkan-GDC_Mar15.pdf.

[17] Khronos releases vulkan 1.0 specification. URL:https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification. Accessed 21.7.2022.

[18] The State of Vulkan on Apple Devices. URL: https://www.lunarg.com/wp-content/uploads/2022/05/The-State-of-Vulkan-on-Apple-15APR2022.pdf.

[19] Windows 10 and directx 12 released! URL:https://devblogs.microsoft.com/directx/windows-10-and-directx-12-released/. Accessed 21.7.2022.

[20] Fundamental metal concepts. URL:https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Device/Device.html#//apple_ref/doc/uid/TP40014221-CH2-SW1. Accessed 20.7.2022.

[21] What is QML? URL: https://doc.qt.io/qt-6/qmlapplications.html. Accessed 27.10.2022.

[22] Qt Quick. URL: https://doc.qt.io/qt-6/qtquick-index.html. Accessed 27.10.2022.

[23] All Modules. URL: https://doc.qt.io/qt-6/qtmodules.html. Accessed 3.10.2022.

[24] Opengl programming guide for mac. URL:https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_pg_concepts/opengl_pg_concepts.html. Accessed 5.9.2022.

[25] Opengl es. URL:https://developer.apple.com/documentation/opengles. Accessed 7.9.2022.

[26] Mac computers that use opencl and opengl graphics. URL:https://support.apple.com/en-us/HT202823. Accessed 7.9.2022.

[27] Introducing the ANGLE Project. URL: https://blog.chromium.org/2010/03/introducing-angle-project.html. Accessed 29.9.2022.

[28] ANGLE - Almost Native Graphics Layer Engine. URL: https://github.com/google/angle. Accessed 29.9.2022.

[29] The OpenGL® ES Shading Language. URL: https://registry.khronos.org/OpenGL/specs/es/2.0/GLSL_ES_Specification_1.00.pdf.

[30] Apalis iMX6. URL: https://www.toradex.com/computer-on-modules/apalis-arm-family/nxp-freescale-imx-6#features. Accessed 7.10.2022.

[31] Registering C++ Types with the QML Type System. URL: https://doc.qt.io/qt-6/qtqml-cppintegration-definetypes.html. Accessed 6.10.2022.

[32] QQuick3DInstancing Class. URL: https://doc.qt.io/qt-6/qquick3dinstancing.html. Accessed 6.10.2022.

[33] QQuick3DGeometry Class. URL: https://doc.qt.io/qt-6/qquick3dgeometry.html. Accessed 6.10.2022.

[34] How to Contribute. URL: https://contribute.qt-project.org/. Accessed 8.2.2023.

[35] Qt Maintainers. URL: https://wiki.qt.io/Maintainers. Accessed 8.2.2023.

[36] Chief Maintainer. URL: https://wiki.qt.io/The_Qt_Governance_Model. Accessed 8.2.2023.

[37] Maintainers. URL: https://wiki.qt.io/The_Qt_Governance_Model. Accessed 8.2.2023.

[38] Qt Test Overview. URL: https://doc.qt.io/qt-6/qtest-overview.html. Accessed 8.2.2023.

[39] Introduction. URL: https://doc.qt.io/qt-6/qtquicktest-index.html. Accessed 8.2.2023.

[40] Qt Data Visualization. URL: https://doc.qt.io/qt-6/qtdatavisualization-index.html. Accessed 8.2.2023.

[41] QAbstractGraph3D Class. URL: https://doc.qt.io/qt-6/qabstract3dgraph.html. Accessed 8.2.2023.

[42] Scene Graph Adaptations in Qt Quick. URL: https://doc.qt.io/qt-6/qtquick-visualcanvas-adaptations.html#scene-graph-adaptations-in-qt-quick. Accessed 8.2.2023.

[43] Qt Data Visualization 1.0 released. URL: https://www.qt.io/blog/2014/03/26/qt-data-visualization-1-0-released. Accessed 8.2.2023.

[44] Qt5.2 release plan. URL: https://wiki.qt.io/Qt_5.2_Release. Accessed 8.2.2023.

[45] Qt Data Visualization 1.2. URL: https://github.com/qt/qtdatavis3d/blob/dev/dist/changes-1.2.0. Accessed 8.2.2023.

[46] AbstractGraph3D QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-abstractgraph3d.html. Accessed 8.2.2023.

[47] Abstract3DSeries QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-abstract3dseries.html. Accessed 8.2.2023.

[48] AbstractDataProxy QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-abstractdataproxy.html. Accessed 8.2.2023.

[49] Bars3D QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-bars3d.html. Accessed 8.2.2023.

[50] Surface3D QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-surface3d.html. Accessed 8.2.2023.

[51] Scatter3D QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-scatter3d.html. Accessed 8.2.2023.

[52] Qt Quick 2 Bars Example. URL: https://doc.qt.io/qt-6/qtdatavis3d-qmlbars-example.html. Accessed 8.2.2023.

[53] Qt Quick 2 Scatter Example. URL: https://doc.qt.io/qt-6/qtdatavis3d-qmlscatter-example.html. Accessed 8.2.2023.

[54] ListModel QML Type. URL: https://doc.qt.io/qt-6/qml-qtqml-models-listmodel.html. Accessed 8.2.2023.

[55] HeightMapSurfaceDataProxy QML Type. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-heightmapsurfacedataproxy.html. Accessed 8.2.2023.

[56] Qt Quick 2 Surface Example. URL: https://doc.qt.io/qt-6/qtdatavis3d-qmlsurface-example.html. Accessed 8.2.2023.

[57] Interacting with Data. URL: https://doc.qt.io/qt-6/qtdatavisualization-interacting-with-data.html#interacting-with-data. Accessed 8.2.2023.

[58] AbstractGraph3D::selectionMode. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-abstractgraph3d.html#selectionMode-prop0. Accessed 8.2.2023.

[59] flags QAbstract3DGraph::SelectionFlags. URL: https://doc.qt.io/qt-6/qabstract3dgraph.html#SelectionFlag-enum. Accessed 8.2.2023.

[60] Qt Quick 2 Multiple Graphs Example. URL: https://doc.qt.io/qt-6/qtdatavis3d-qmlmultigraph-example.html. Accessed 8.2.2023.

[61] InputHandler3D::zoomEnabled. URL: https://doc.qt.io/qt-6/qml-qtdatavisualization-inputhandler3d.html#zoomEnabled-prop. Accessed 8.2.2023.

[62] QAbstract3DGraphPrivate::renderNow. URL: https://github.com/qt/qtdatavis3d/blob/0dfb996b629d2815bccf32c481680c249f3133c9/src/datavisualization/engine/qabstract3dgraph.cpp#L1129. Accessed 8.2.2023.

[63] AbstractDeclarative::render. URL: https://github.com/qt/qtdatavis3d/blob/0dfb996b629d2815bccf32c481680c249f3133c9/src/datavisualizationqml/abstractdeclarative.cpp#L550. Accessed 8.2.2023.

[64] Abstract3DController::render. URL: https://github.com/qt/qtdatavis3d/blob/0dfb996b629d2815bccf32c481680c249f3133c9/src/datavisualization/engine/abstract3dcontroller.cpp#L517. Accessed 8.2.2023.

[65] Abstract3DController class. URL: https://github.com/qt/qtdatavis3d/blob/dev/src/datavisualization/engine/abstract3dcontroller_p.h. Accessed 8.2.2023.

[66] Bars3DController class. URL: https://github.com/qt/qtdatavis3d/blob/dev/src/datavisualization/engine/bars3dcontroller_p.h. Accessed 8.2.2023.

[67] Scatter3DController class. URL: https://github.com/qt/qtdatavis3d/blob/dev/src/datavisualization/engine/scatter3dcontroller_p.h. Accessed 8.2.2023.

[68] Surface3DController class. URL: https://github.com/qt/qtdatavis3d/blob/dev/src/datavisualization/engine/surface3dcontroller_p.h. Accessed 8.2.2023.

[69] The Scene Graph in Qt Quick. URL: https://doc.qt.io/qt-6/qtquick-visualcanvas-scenegraph.html. Accessed 7.2.2023.

[70] QSGNode::preprocess. URL: https://doc.qt.io/qt-6/qsgnode.html#preprocess. Accessed 8.2.2023.

[71] DeclarativeRenderNode. URL: https://github.com/qt/qtdatavis3d/blob/dev/src/datavisualizationqml/declarativerendernode_p.h. Accessed 8.2.2023.

[72] QSGGeometryNode Class. URL: https://doc.qt.io/qt-6/qsggeometrynode.html. Accessed 8.2.2023.

[73] DeclarativeRenderNode::updateFBO. URL: https://github.com/qt/qtdatavis3d/blob/0dfb996b629d2815bccf32c481680c249f3133c9/src/

datavisualizationqml/declarativerendernode.cpp#L58. Accessed 8.2.2023.

[74] QOpenGLFramebufferObject Class. URL: https://doc.qt.io/qt-6/qopenglframebufferobject.html. Accessed 8.2.2023.

[75] DeclarativeRenderNode::preprocess. URL: https://github.com/qt/qtdatavis3d/blob/0dfb996b629d2815bccf32c481680c249f3133c9/src/datavisualizationqml/declarativerendernode.cpp#L124. Accessed 8.2.2023.

[76] QtQuick 3D. URL: https://doc.qt.io/qt-6/qtquick3d-index.html. Accessed 7.2.2023.

[77] Qt Quick Spatial Scene Graph. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#qt-quick-spatial-scene-graph. Accessed 7.2.2023.

[78] QQuick3DViewport. URL: https://github.com/qt/qtquick3d/blob/09bb3d697f96e79d698b34ee5a810306e510000c/src/quick3d/qquick3dviewport_p.h#L43. Accessed 7.2.2023.

[79] 3D in 2D Integration. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#3d-in-2d-integration. Accessed 7.2.2023.

[80] Scene Manager. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#scene-manager. Accessed 7.2.2023.

[81] Frontend/Backend Synchronization. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#frontend-backend-synchronization. Accessed 7.2.2023.

[82] QQuick3DViewport::updatePaintNode. URL: https://github.com/qt/qtquick3d/blob/09bb3d697f96e79d698b34ee5a810306e510000c/src/quick3d/qquick3dviewport.cpp#L494. Accessed 7.2.2023.

[83] QQuickWindow::beforeRenderPassRecording. URL: https://doc.qt.io/qt-6/qquickwindow.html#beforeRenderPassRecording. Accessed 7.2.2023.

[84] QQuickWindow::afterRenderPassRecording. URL: https://doc.qt.io/qt-6/qquickwindow.html#afterRenderPassRecording. Accessed 7.2.2023.

[85] View3D::renderMode. URL: https://doc.qt.io/qt-6/qml-qtquick3d-view3d.html#renderMode-prop. Accessed 7.2.2023.

[86] Offscreen. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#offscreen. Accessed 7.2.2023.

[87] Underlay. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#underlay. Accessed 7.2.2023.

[88] Overlay. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#overlay. Accessed 7.2.2023.

[89] Inline. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#inline. Accessed 7.2.2023.

[90] View3D. URL: https://doc.qt.io/qt-6/qml-qtquick3d-view3d.html. Accessed 7.2.2023.

[91] Object3D. URL: https://doc.qt.io/qt-6/qml-qtquick3d-object3d.html. Accessed 7.2.2023.

[92] SceneEnvironment QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-sceneenvironment.html. Accessed 7.2.2023.

[93] Node QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-node.html. Accessed 7.2.2023.

[94] Material QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-material.html. Accessed 7.2.2023.

[95] DefaultMaterial QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-defaultmaterial.html. Accessed 7.2.2023.

[96] PrincipledMaterial QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-principledmaterial.html. Accessed 7.2.2023.

[97] SpecularGlossyMaterial QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-speculargloss ymaterial.html. Accessed 7.2.2023.

[98] CustomMaterial QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-custommaterial.html. Accessed 7.2.2023.

[99] Specular/Glossiness workflow. URL: https://doc.qt.io/qt-6/qml-qtquick3d-defaultmaterial.html#specular-glossiness-workflow. Accessed 7.2.2023.

[100] Detailed Description. URL: https://doc.qt.io/qt-6/qml-qtquick3d-principledmaterial.html#details. Accessed 7.2.2023.

[101] Detailed Description. URL: https://doc.qt.io/qt-6/qml-qtquick3d-speculargloss ymaterial.html#details. Accessed 7.2.2023.

[102] Metal/Roughness workflow. URL: https://doc.qt.io/qt-6/qml-qtquick3d-principledmaterial.html#metal-roughness-workflow. Accessed 7.2.2023.

[103] Specular/Glossiness workflow. URL: https://doc.qt.io/qt-6/qml-qtquick3d-speculargloss ymaterial.html#specular-glossiness-workflow. Accessed 7.2.2023.

[104] Detailed Description. URL: https://doc.qt.io/qt-6/qml-qtquick3d-custommaterial.html#details. Accessed 7.2.2023.

[105] CustomMaterial::vertexShader. URL: https://doc.qt.io/qt-6/qml-qtquick3d-custommaterial.html#vertexShader-prop. Accessed 7.2.2023.

[106] CustomMaterial::fragmentShader. URL: https://doc.qt.io/qt-6/qml-qtquick3d-custommaterial.html#fragmentShader-prop. Accessed 7.2.2023.

[107] Programmability for Materials. URL: https://doc.qt.io/qt-6/qtquick3d-custom.html#programmability-for-materials. Accessed 7.2.2023.

[108] SceneEnvironment::antialiasingMode. URL: https://doc.qt.io/qt-6/qml-qtquick3d-sceneenvironment.html#antialiasingMode-prop. Accessed 7.2.2023.

[109] Fxaa QML Type. URL: https://doc.qt.io/qt-6/qml-qtquick3d-effects-fxaa.html. Accessed 7.2.2023.

[110] Supersample Anti-Aliasing. URL: https://doc.qt.io/qt-6/quick3d-asset-conditioning-anti-aliasing.html. Accessed 7.2.2023.

[111] Multisample Anti-Aliasing. URL: https://doc.qt.io/qt-6/quick3d-asset-conditioning-anti-aliasing.html. Accessed 7.2.2023.

[112] Progressive Anti-Aliasing. URL: https://doc.qt.io/qt-6/quick3d-asset-conditioning-anti-aliasing.html. Accessed 7.2.2023.

[113] SceneEnvironment:antialiasingQuality. URL: https://doc.qt.io/qt-6/qml-qtquick3d-sceneenvironment.html#antialiasingQuality-prop. Accessed 8.2.2023.

[114] Temporal Anti-Aliasing. URL: https://doc.qt.io/qt-6/quick3d-asset-conditioning-anti-aliasing.html. Accessed 7.2.2023.

[115] SceneEnvironment::temporalAAStrength. URL: https://doc.qt.io/qt-6/qml-qtquick3d-sceneenvironment.html#temporalAAStrength-prop. Accessed 7.2.2023.

[116] Post-processing effects. URL: https://doc.qt.io/qt-6/qml-qtquick3d-effect.html#post-processing-effects. Accessed 7.2.2023.

[117] 2D in 3D Integration. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#2d-in-3d-integration. Accessed 7.2.2023.

[118] Direct Path. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#direct-path. Accessed 7.2.2023.

[119] QQuick3DItem2D. URL: https://github.com/qt/qtquick3d/blob/2fbd37f8b718d3a85428193e007ba19e0c41dc19/src/quick3d/qquick3ditem2d_p.h#L31. Accessed 7.2.2023.

[120] QQuick3DItem2D::updateSpatialNode. URL: https://github.com/qt/qtquick3d/blob/2fbd37f8b718d3a85428193e007ba19e0c41dc19/src/quick3d/qquick3ditem2d.cpp#L102. Accessed 7.2.2023.

[121] enum QSGRendererInterface::RenderMode. URL: https://doc.qt.io/qt-6/qsgrendererinterface.html#RenderMode-enum. Accessed 7.2.2023.

[122] Texture Path. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#texture-path. Accessed 7.2.2023.

[123] Scene Rendering. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#scene-rendering. Accessed 7.2.2023.

[124] Set up Render Target. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#set-up-render-target. Accessed 7.2.2023.

[125] Prepare for Render. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#prepare-for-render. Accessed 7.2.2023.

[126] High level render preparation. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#prepare-for-render. Accessed 7.2.2023.

[127] SceneEnvironment::temporalAAEnabled. URL: https://doc.qt.io/qt-6/qml-qtquick3d-sceneenvironment.html#temporalAAEnabled-prop. Accessed 7.2.2023.

[128] SceneEnvironment::antialiasingMode::ProgressiveAA. URL: https://doc.qt.io/qt-6/qml-qtquick3d-sceneenvironment.html#antialiasingMode-prop. Accessed 7.2.2023.

[129] Low level render preparation. URL: https://doc.qt.io/qt-6/qtquick3d-architecture.html#prepare-for-render. Accessed 7.2.2023.

[130] MainPass::renderPass. URL: https://github.com/qt/qtquick3d/blob/2fbd37f8b718d3a85428193e007ba19e0c41dc19/src/runtimerender/rendererimpl/qssgrenderpass.cpp#L735. Accessed 7.2.2023.

[131] QQuick3DSceneRenderer::renderToRhiTexture. URL: https://github.com/qt/qtquick3d/blob/dev/src/quick3d/qquick3dscenerenderer.cpp#L232. Accessed 7.2.2023.

[132] New Features in Qt 5.14. URL: https://wiki.qt.io/New_Features_in_Qt_5.14. Accessed 1.2.2023.

[133] Qt Quick. URL: https://wiki.qt.io/New_Features_in_Qt_6.0. Accessed 8.2.2023.

[134] Qt Quick on Vulkan, Metal, and Direct3D. URL: https://www.qt.io/blog/qt-quick-on-vulkan-metal-direct3d. Accessed 2.2.2023.

[135] QSGRhiSupport::createRhi. URL: https://github.com/qt/qtdeclarative/blob/984825958c59f673af091aefd73cb399f3852eaf/src/quick/scenegraph/qsgrhisupport.cpp#L1071. Accessed 8.2.2023.

[136] Rendering via the Qt Rendering Hardware Interface. URL: https://doc.qt.io/qt-6/qtquick-visualcanvas-scenegraph-renderer.html#rendering-via-the-qt-rendering-hardware-interface. Accessed 8.2.2023.

[137] Qt Data Visualization Known Issues. URL: https://doc.qt.io/qt-6/qtdatavisualization-known-issues.html. Accessed 8.2.2023.

[138] MoltenGL. URL: https://moltengl.com/. Accessed 2.2.2023.

[139] QRhiD3D11 implementation. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L601. Accessed 6.2.2023.

[140] QRhi resource base class. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L657. Accessed 6.2.2023.

[141] QRhi class. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L1624. Accessed 6.2.2023.

[142] QRhi beginFrame-function. URL: https://github.com/qt/qtbase/blob/dev/src/gui/rhi/qrhi_p.h#L1778. Accessed 6.2.2023.

[143] QRhi implementation. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p_p.h#L33. Accessed 6.2.2023.

[144] QRhi resource types. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L660. Accessed 6.2.2023.

[145] QRhiBuffer. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L699. Accessed 6.2.2023.

[146] QD3D11Buffer. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L29. Accessed 6.2.2023.

[147] QD3D11Buffer::create. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L34. Accessed 6.2.2023.

[148] QD3D11Buffer::destroy. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L33. Accessed 6.2.2023.

[149] QD3D11Buffer::nativeBuffer. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L35. Accessed 6.2.2023.

[150] QD3D11Buffer::beginFullDynamicBufferUpdateForCurrentFrame. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L36. Accessed 6.2.2023.

[151] QD3D11Buffer::endFullDynamicBufferUpdateForCurrentFrame. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhid3d11_p_p.h#L37. Accessed 6.2.2023.

[152] QRhiResource::destroy. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L679. Accessed 6.2.2023.

[153] QRhiBuffer::create. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L732. Accessed 6.2.2023.

[154] QRhiBuffer::nativeBuffer. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi.cpp#L2398. Accessed 6.2.2023.

[155] QRhiBuffer::beginFullDynamicBufferUpdateForCurrentFrame. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi.cpp#L2437. Accessed 6.2.2023.

[156] QRhiBuffer::endFullDynamicBufferUpdateForCurrentFrame. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi.cpp#L2447. Accessed 6.2.2023.

[157] QRhi::QRhi. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L1825. Accessed 6.2.2023.

[158] QRhi::create. URL: https://github.com/qt/qtbase/blob/e8322a4cc043e1a150cc4c6b86ee2f9cf858cd24/src/gui/rhi/qrhi_p.h#L1718. Accessed 6.2.2023.

[159] Qt Shader Tools. URL: https://doc.qt.io/qt-6/qtshadertools-index.html. Accessed 6.2.2023.

[160] Qt Shader Tools Overview. URL: https://doc.qt.io/qt-6/qtshadertools-overview.html. Accessed 6.2.2023.

[161] GLSLANG. URL: https://github.com/KhronosGroup/glslang. Accessed 6.2.2023.

[162] SPIRV-Cross. URL: https://github.com/KhronosGroup/SPIRV-Cross. Accessed 6.2.2023.