Dynamic Waveform Charts for Real-Time Medical Monitoring

Jakob Strauss

Jakob Strauss

Dynamic Waveform Charts for Real-Time Medical Monitoring

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 15 May 2017

Jakob Strauss

Dynamic Waveform Charts for Real-Time Medical Monitoring

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Informatik

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 15. Mai 2017

Diese Arbeit ist in englischer Sprache verfasst.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Date/Datum	Signature/Unterschrift

Abstract

An application was developed to compare different technologies to display dynamic line (waveform) diagrams with a large set of data points. Initially, several different GUI libraries were compared and Qt5 was picked to implement the application. Then, five different renderers for the waveforms were created with Qt native, Qt with OpenGL, Qt with QML and a HTML5 2D Canvas, Qt WebView with Pixi.js, and Qt WebView with D3. The goal was to achieve a rendering frame rate of at least 60 frames per second (fps) for flicker-free display on a 60 Hz monitor. A performance comparison showed that only the Qt native and Qt with OpenGL renderers achieved 60 fps. This work was conducted in cooperation with ImPress, a local startup company.

Kurzfassung

Es wurde eine Software entwickelt, die um verschiedene Technologien zu testen,Liniendiagramme mit vielen Datenpunkten darstellt. Dazu wurden verschiedene GUI Bibliotheken verglichen. Qt in Verbindung mit C++ wurde dann verwendet um die Waveform Software zu schreiben. Fünf verschiedene Renderer wurden implementiert. Qt native, Qt mit OpenGL, Qt mit QML und einen HTML5 Canvas, Qt WebView mit Pixi.js und Qt WebView mit D3. Das Ziel war eine Framerate von mindestens 60 fps zu erreichen damit kein Flackern auf einen 60 Hz Bildschirm zu sehe ist. Nach der Implementierung wurde eine Leistungsüberprüfung durchgeführt die zeigte dass nur die Qt native und die Qt mit OpenGL Implementierungen dieses Ziel erfüllen konnten. Die Arbeit wurde in Zusammenarbeit mit ImPress, einem lokalen Startup, durchgeführt.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I especially wish to thank my advisor, Keith Andrews, for his attention to my questions and endless hours of toil in correcting draft versions of this thesis.

Special mention goes to Arnulf Heller for providing me the opportunity to work at his company ImPress.

Last but not least, without the support and understanding of my family, this thesis would not have been possible.

Jakob Strauss
Graz, Austria, May 2017

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews, 2012].
- PhysioNet for there collection and maintaining of the WFDB files [Goldberger et al., 2000].
- Figure 2.3 was taken from Sven [2017] and is used under the terms of the Creative Commons Attribution-Share Alike 3.0 Unported [CC, 2017] licence.

All other diagrams and images were created by the author of this thesis.

Chapter 1

Introduction

A C++ application called Waveform was developed to compare different technologies to display dynamic line diagrams (waveforms) with a large number of data points, upwards of 1000 per line. The goal was to find a rendering technology capable of 60 frames per second for real-time medical monitoring.

In Chapter 2, several Graphical User Interface (GUI) libraries for different operation systems , like Qt, GTK+, wxWidgets, Java Swing, Java SWT, JavaFX, The WinAPi, MFC, the Android GUI system and Unity.

Chapter 3 discussed and compared underlying graphics libraries, like OpenGL,OpenGL ES, WebGL, ANGLE, Mesa, DirectX and Direct 3D. An Overview of available graphics shaders was created.

Chapter 4 introduces the Waveform program itself. It is written mostly in C++ and Qt. Five different rendering technologies for waveforms were implemented:

1. Qt/native renderer using Qt widgets to draw the waveform.
2. Qt/OpenGL renderer using QGLWidget to provide an OpenGL context. The actual drawing of the waveforms is done with shader-based OpenGL calls.
3. Qt/QML/HTML5 canvas renderer using QML to arrange interface components and a HTML5 Canvas 2D to draw the waveforms.
4. Qt/WebView/Pixi renderer using Qt to provide a WebKit window. This window loads a HTML page which uses the Pixi library to draw the waveform. Pixi itself uses WebGL to accelerate the drawing.
5. Qt/WebView/D3 renderer using Qt to provide a WebKit window. This window loads a HTML page which uses the D3 library to draw the waveform by dynamically creating SVG nodes in the browser DOM.

The performance of the five different renderers was measured. The goal was to achieve a refresh rate of 60 frames per second (fps) to obtain flicker-free display on a 60 Hz screen. Only the Qt/native and the Qt/OpenGL renderers actually reached the goal. The slow speed of the other implementations might be improved by a better multi-threaded design of the software.

Chapter 5 describes the SmartNIBP software suite, which was designed with the results from the Waveform test in mind. SmartNIBP is a device for real-time monitoring of blood pressue in a hospital setting. This work was conducted in cooperation with ImPress, a local startup company.

Chapter 2

User Interface Technologies for Embedded Devices

The main goal of a Graphical User Interface (GUI) library is to provide components, which can be used to construct an intuitive user interface. In addition, most GUI libraries also provide support for audio and video output and network connections. This chapter surveys some of the most important GUI libraries.

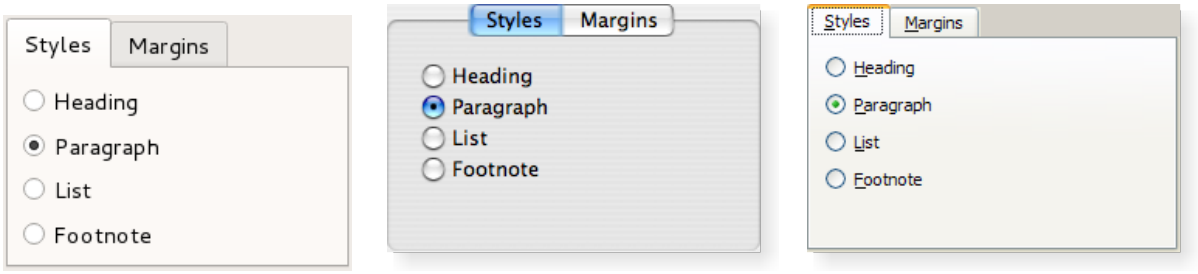
2.1 Qt

Qt [QtC, 2017c] provides an open source framework for cross-platform graphical interface development. Supported platforms are Linux with X11 or Wayland, Windows and its derivatives, Symbian OS, Android, Mac OS X, and iOS. Qt is licensed under three licenses: the GNU General Public License (GPL)[GNU, 2017b], the GNU Lesser General Public License(LGPL) [GNU, 2017a], and a proprietary license. The latter is only needed for commercial applications. Bindings are available for many languages including Python, Java, and Ruby [QtC, 2017b].

One of the largest projects written with the help of Qt is the K Desktop Environment (KDE) for Linux. Qt can use either the native host operating system to draw, or can use its own paint engine. As shown in Figure 2.1, it can also adapt its look-and-feel for different host platforms, or can use its own look-and-feel, called Fusion. Since Qt4, the framework is split into modules which are grouped into essentials and add-ons. The essential modules define the base of Qt and are supported on all platforms. Add-ons bring additional functionality and are only supported on specific platforms. There are also enterprise add-ons, like Qt Data Visualization, Qt for Device Creation, and Qt Quick Compiler which needs the enterprise version of Qt [Bocklage-Ryannel and Thelin, 2015].

There are several different ways to build a GUI in Qt5. The first method uses QTWidgets, where every component is provided by QT and is defined in a C++ class. The second method uses QTQuick2 which is based on OpenGL ES and is managed with a scene graph. To design the GUI with QTQuick2, the interpreted declarative language Qt Modeling Language (QML) is used. QML elements are defined and populated with properties, and elements can be nested. Child elements can use properties from parents via the parent operator. A typical application has a back-end written in C++, the front end in QML, and the movement of QML elements is scripted with JavaScript. QML can draw simple shapes directly and a canvas element was introduced in QTQuick2 based on the HTML 5 canvas element, where arbitrary paths can be rendered. A good description of QTQuick and QML is provided in the *QMLBook* by Bocklage-Ryannel and Thelin [2016, Section 1.4].

A small example QML file is shown in Listing 2.1. The output can be seen in Figure 2.2. The window contains a grey rectangle whose size is defined such that its width is always twice its height. Inside the rectangle is a text, displaying “Hello World!”. The rectangle contains a MouseArea which registers mouse



(a) Qt Fusion. **(b)** Mac OS X. **(c)** Windows 10.

Figure 2.1: Qt can adapt its look-and-feel, depending on the host operating system, or can use its own look-and-feel [Screenshots taken by the author.].

```
1  Window {
2    visible: true
3    width: 640
4    height: 320
5    title: qsTr("Hello World")
6
7    Rectangle {
8      color: "grey"
9      x: 50; y: 50
10     height: 250
11     width: 2*height
12
13     Text {
14       id: label
15       color: "black"
16       text: "Hello World!"
17       anchors.centerIn: parent
18     }
19
20     MouseArea {
21       anchors.fill: parent
22       onClicked: {
23         label.text = "Hello QML!";
24       }
25     }
26   }
27 }
```

Listing 2.1: QML code for a window containing a grey rectangle.

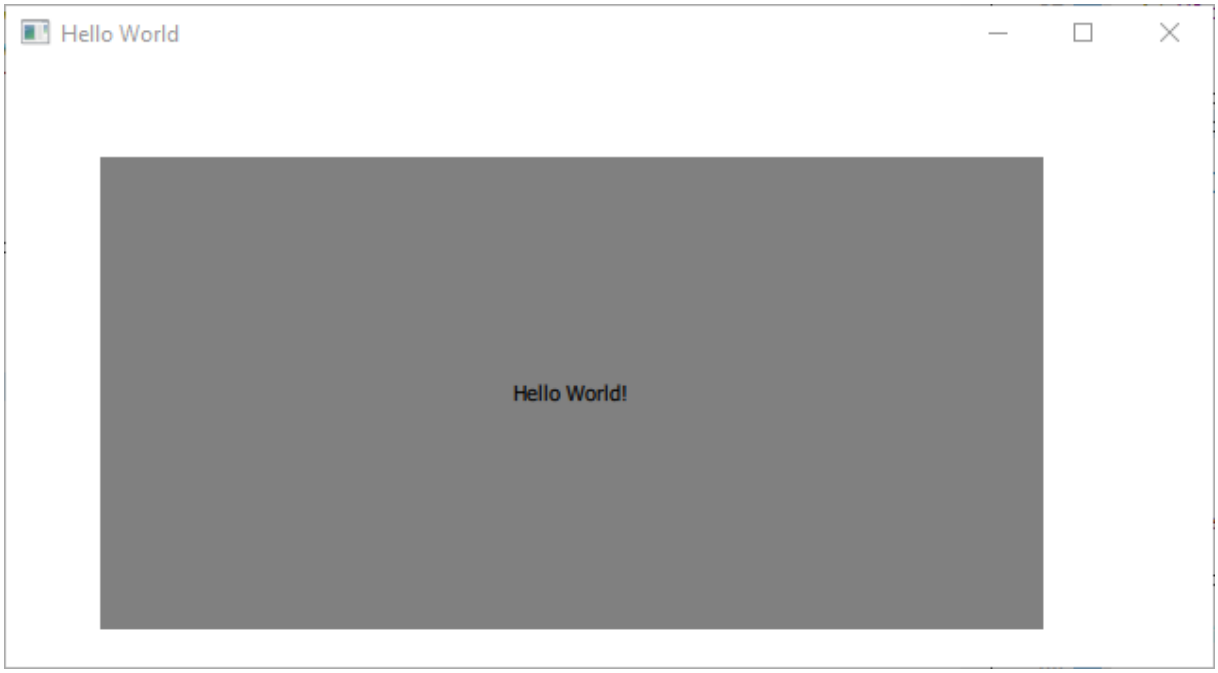


Figure 2.2: The output of Listing 2.1. Initially, a grey rectangle is drawn containing the text "Hello World!". Once the rectangle is clicked, the text is changed to "Hello QML!" [Screenshots taken by the author.].

clicks and changes the text to "Hello QML!" when clicked.

Qt comes with its own development environment, QtCreator, or can be used in Visual Studio with an add-on. The add-on supports Qt-specific concepts including:

- A preprocessor called Meta Object Compiler (MOC) is used to add non-standard C++ functionality to the application. The MOC functions like a normal preprocessor and expands Qt-specific keywords to C++ code, which than can be compiled as usual [QtC, 2017d].
- Qt uses a signal slot concept to transmit messages between objects. The signal slot system is slightly slower than an implementation with callbacks, but has the advantage that it is type safe.
- Qmake generates build files similar to Unix makefiles, which are used for easy integration of the meta object compiler and the resource compiler from Qt. It is possible to generate build files for other IDEs such as makefiles, Windows solution files, and Apple Xcode project files.

2.2 GTK+

The GIMP Toolkit (GTK+) [GTK+, 2017c] is a widget toolkit for creating graphical user interfaces. It is licensed under the GNU Lesser General Public License which allows everyone to use the library, even for commercial software [GTK+, 2017c]. GTK+ was first created to provide a GUI toolkit for the image editor GIMP [GTK+, 2017a]. Nowadays, it is not only used for GIMP, but also for the desktop environments GNOME, Xfce, and for many other programs. Supported platforms are Windows, X11, Wayland, Mac OS X, and HTML5.

The first version was written in C and was named GIMP Toolkit. After a rewrite in object-oriented C, it was renamed GTK+. GIMP used GTK+ from version 0.99 onwards. The library consists of several components, including GLib in which C functions are stored, and GDK (GTK+ Drawing Kit). The GDK provides low-level drawing methods for the various supported operating systems. Following a switch to

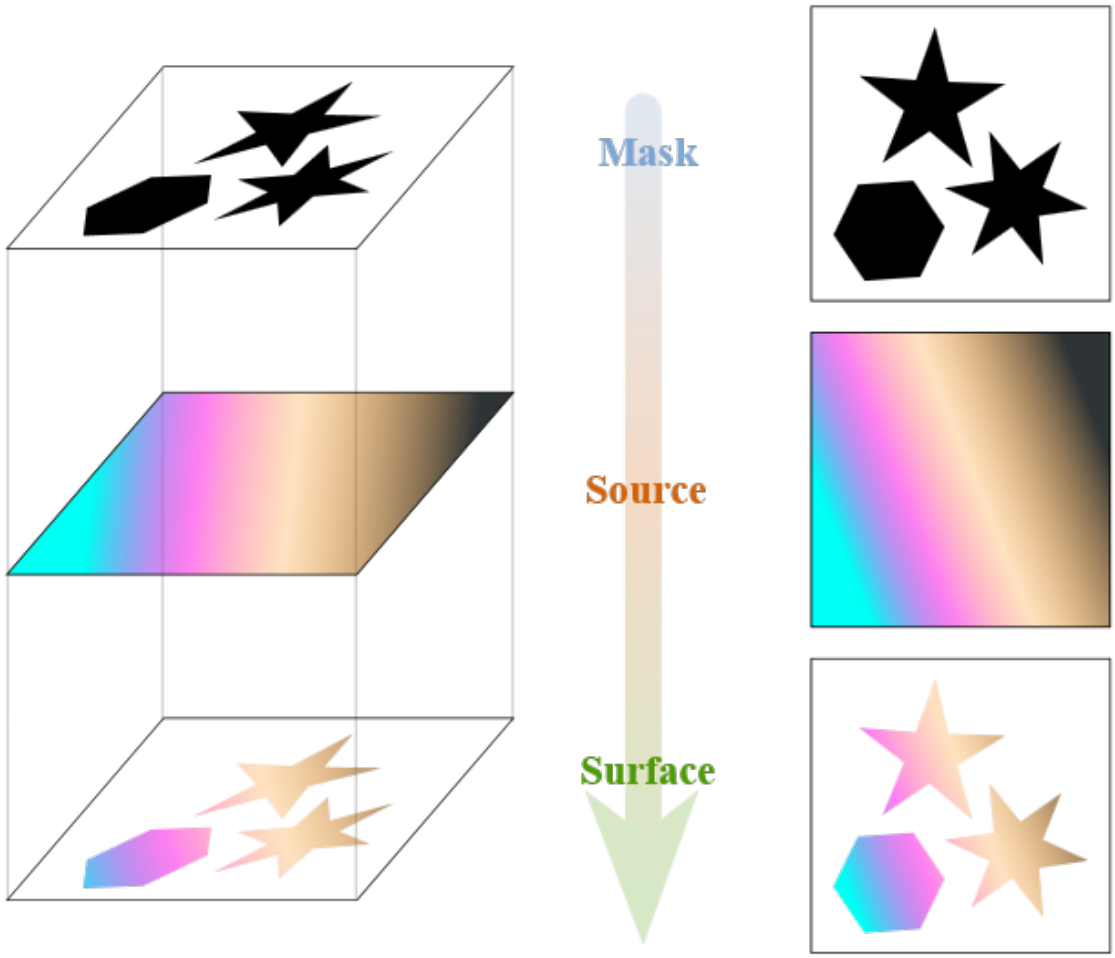
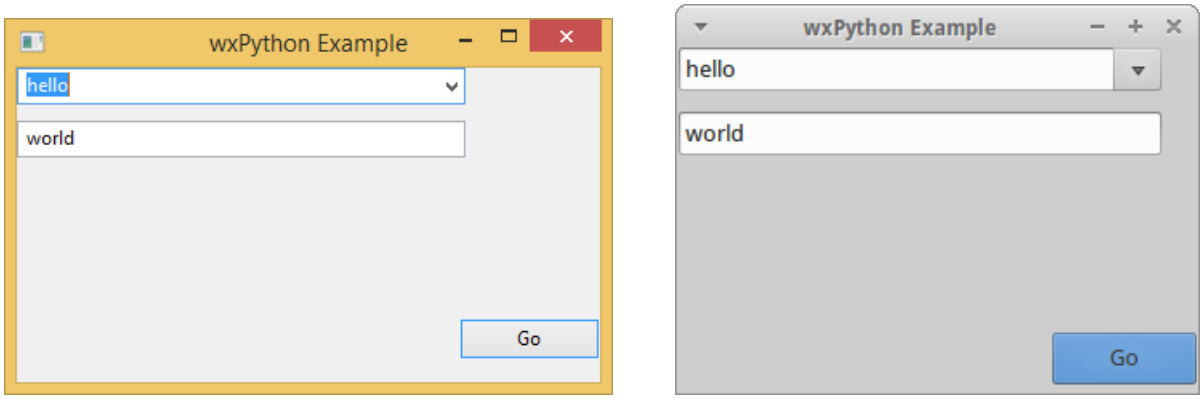


Figure 2.3: The drawing process used by Cairo is based on the concepts of a mask, a source, and a surface [Image from Sven [2017] and used under the terms of CC BY-SA 3.0 [CC, 2017]].

the text rendering engine Pango, the addition of a new theme engine, and the switch to UTF-8, the new version was called GTK+2. The large set of changes made it necessary to break compatibility with the 1.x series.

GTK+3 changed many interfaces, old deprecated functions were removed, and Cairo was introduced as a hardware accelerated 2D vector graphics library. Cairo is a platform-independent vector graphics library which can render to multiple back-ends like the X Window System, the Win32 APi, OpenGL surfaces, and several image formats. Cairo creates a mask of the outlines of a model to be drawn. Then the source of the colour is defined and combined with the mask to create the surface. Finally, the surface is drawn. The process is shown in Figure 2.3. The Cairo library itself is free software and available under the GNU Lesser General Public License (LGPL) version 2.1 or the Mozilla Public License version 1.1 [Cairo, 2017].

A GTK+ interface can either be written by hand, or a WYSISWG GUI designer can be used. GTK+ is multi-platform and multi-language. Several language bindings exists [GTK+, 2017b] and the look-and-feel can be adapted according to the platform [GNOME, 2017].



(a) Windows 10. **(b)** Ubuntu 16.04.

Figure 2.4: An example dialogue from wxWidgets. wxWidgets can display different look-and-feel, depending upon the host operating system [Screenshots made by the author.].

2.3 wxWidgets

WxWidgets [wxWidgets, 2017a] is an open source, cross-platform, C++ GUI library. It was first developed in 1992 at the University of Edinburgh, originally under the name *WxWindows*, until Microsoft requested in 2003 that the framework be renamed [wxWidgets, 2017c]. Windows, Mac OS X, and Linux are supported, and there are language bindings for Python, Perl, and C#. WxWidgets can use the native platform's look-and-feel or use its own style, as shown in Figure 2.4. The library provides more than pure GUI functionality, including multi-threading, network sockets, and image handling [wxWidgets, 2017d]. WxWidgets is distributed under the wxWindows License [wxWidgets, 2017b] which is based on the L-GPL license with an exception to allow the linking of a program to wxWidgets without the requirement to distribute the source code of the application. Programs known to use wxWidgets include Audacity [Audacity, 2017] and Filezilla [FileZilla, 2017].

2.4 Java GUI Libraries

Java [Oracle, 2017g] is an object-oriented programming language, designed with as few platform dependencies as possible. The code is compiled to bytecode, which then runs on a Java Virtual Machine (JVM). All platform dependencies are in the JVM. Powerful GUI libraries provide a large selection of possible look-and-feels to match the various platforms. Java has three major GUI libraries: Swing, SWT, and JavaFX. Java was introduced by Sun in 1995, then bought by Oracle in 2010, the current version is Java SE 8, released in March 2015.

2.4.1 Swing

Swing [Oracle, 2017a] is a Java GUI library contained inside the Java Foundation Classes. It is based on the Abstract Window Toolkit (AWT), an older Java GUI framework. The first version of Swing was released with Java 1.1.5 at the end of 1997. Swing components are lightweight, which means the components are rendered by Java itself and not by the underlying window system. In the beginning, Swing had problems with slow performance and the design of the elements was poorly received. Since then, both the performance and design were greatly improved and are on par with native GUI systems.

Java has its own look-and-feel called Nimbus [Oracle, 2017e], which does not resemble the style of any existing operating system. For almost every operating system supported by Java, there is a compat-

ible look-and-feel following the guidelines of that system, as shown in Figure 2.5. It is also possible to generate a look-and-feel based on an XML configuration file. The code for a Swing interface can be created by hand or using a WYSIWYG tool like Netbeans [Oracle, 2017d].

2.4.2 Standard Widget Toolkit (SWT)

SWT [Eclipse Foundation, 2017b] was developed by IBM and is now maintained by the Eclipse Foundation. It is a GUI toolkit which uses native GUI elements. If no supported native elements are available, SWT has a fall-back solution and can use its own GUI implementation. The toolkit is licensed under the Eclipse Public License [Eclipse Foundation, 2017a].

2.4.3 JavaFX

JavaFX [Oracle, 2017c], first released in 2008 by Oracle, was intended to become the standard GUI library. It is a standard part of the Java installation since version 7 of the runtime library [Oracle, 2017b]. JavaFX uses a scene graph to organise the components of a GUI. Each element in the graph is called node and can have children, effects, and handlers. JavaFX uses a declarative design approach to designing a GUI, using the XML-based FXML markup language. The JavaFX Scene Builder is a visual design tool which generates FXML code. The JavaFX application be styled with Cascading Style Sheets (CSS) to customise the application. JavaFX is available under the Oracle Binary Code License Agreement [Oracle, 2017f].

2.5 Microsoft GUI Libraries

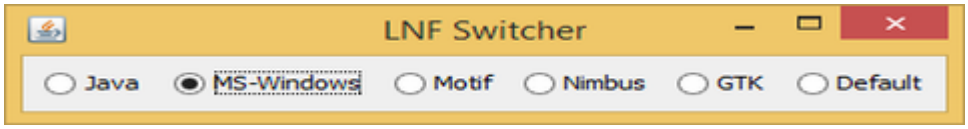
Microsoft provides the WinAPI [Microsoft, 2017d] to create a GUI for Microsoft Windows applications. Depending on the version of Windows, the API chooses the correct look-and-feel. WinAPI is written in C, and language bindings exist for many languages. Every major Windows version also updated the WinAPI. Wrappers such as the Microsoft Foundation Classes (MFC) make using the WinAPI easier.

2.5.1 Microsoft Foundation Classes (MFC)

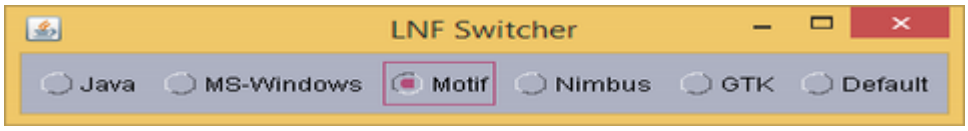
The Microsoft Foundation Classes (MFC) [Microsoft, 2017b] were called Application Framework Extensions at first, and are an object-oriented library wrapped around the non-object-oriented WinAPI calls. The framework was first introduced in 1992. The MFC are bundled with the Microsoft C++ compiler and the runtime is bundled in the Visual Studio C++ runtime. The library is backwards compatible and supports recent versions of the Windows design, like Metro and the ribbon-oriented Aero. Recent versions of MFC make it easy to port programs to the newer and Microsoft preferred .NET framework [Microsoft, 2017c].

2.5.2 Windows Presentation Foundation (WPF)

The Windows Presentation Foundation(WPF) [Microsoft, 2017e] is a subsystem introduced to the .NET Framework in version 3.0. The WPF is responsible for rendering the graphical user interface of a .NET application with the help of DirectX. WPF is resolution-independent and uses the system settings to render elements at the correct size. This makes it possible to support vector graphics, and to prevent the generation of artefacts when graphics are scaled. The framework supports animation for its elements, and also manages the playback of audio and video files. An interface in WPF is separated from the logic of the code and is designed with an XML-based meta-language called XAML [MacDonald, 2009, page 5].

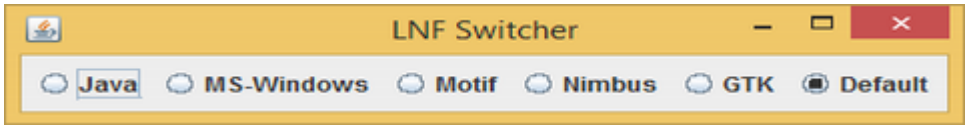


(a) Windows style on Windows 8.



(b) Motif style on Windows 8.

(c) Nimbus style on Windows 8.



(d) Default style on Windows 8.

(e) GTK style on Xfce.

(f) Java style on Xfce.

(g) Motif style on Xfce.

(h) Nimbus style on Xfce.

Figure 2.5: Java Swing can adapt its look-and-feel depending upon the host operating system [Screenshots made by the author.].

2.5.3 Extensible Application Markup Language (XAML)

The Extensible Application Markup Language (XAML) is used in the .NET Framework from version 3.0 onwards and is based on XML. It is primarily used to design a WPF user interface and any animations. Typically, XAML code is not written by hand, but is generated by a graphical design tool such as Visual Studio or Microsoft Expression Blend [MacDonald, 2009, page 21]. It is possible for a designer to create the user interface of an application, without any knowledge of XAML syntax or general knowledge of programming languages [MacDonald, 2009, page 22]. XAML is used in Silverlight and Windows applications from Windows 8 upwards.

2.6 Android GUI

Android [Google, 2017a] uses OpenGL ES in combination with EGL [Khronos, 2017b]. EGL is an interface between OpenGL ES and the underlying native platform windowing system. Android provides an API to create the user interface for apps. To achieve a homogeneous user experience across apps, the Android design principles [Google, 2017b] explain the intended experience of every element. Every major upgrade to Android brought a change to these guidelines, like the focus on *Material Design* in version 5 [Google, 2017c]. The user interface is either designed with a graphical tool or written by hand in XML. One of the special problems of app design is the different form factors of tablets and smartphones and the change in orientation from landscape to portrait mode. Android uses components called fragments as basis elements, where each fragment is designed to perform one task. Fragments can be re-arranged according to the orientation and available screen space. Buttons and other user elements of an application are placed on fragments.

2.7 Unity

Unity [Unity, 2017] is a 3D game engine created with the goal of making it easier to design games, but it is also possible to generate other interactive programs. The IDE runs on Windows and Mac and it is possible to deploy the generated programs to Linux, PS3, Xbox, Wii, iOs, Android, Mac, and Windows. Unity provides a scene graph, where the scene is saved, and it can display the scene in a window to manipulate it by drag-and-drop. Unity can use OpenGL or Direct3D to render the scene depending on the target system. Most popular shaders are already implemented in the engine and it is possible to write custom shaders too.

Chapter 3

Graphics Libraries

A number of 3D graphics libraries are available to access the 3D hardware capabilities of modern graphics chips. These provide functions to draw rapidly on the screen. In the beginning of 3D computer graphics, only fixed function pipelines were available. These pipelines had a pre-defined set of steps and methods, which only provided limited freedom. Today, only shader-based pipelines remain. Shaders are programmable hardware or software modules, which can perform small tasks in parallel for every piece of geometry in a scene.

The two most common graphics libraries are OpenGL and DirectX. OpenGL is available on almost every platform. DirectX runs only on products from Microsoft such as Xbox, Windows, or Windows Phone.

3.1 Open Graphics Library (OpenGL)

The Open Graphics Library (OpenGL) [Khronos, 2017c] is a low-level library to create platform-independent and language-independent real-time 2D and 3D computer graphics [Khronos, 2017f]. The library is designed as a state machine for hardware implementation. Only functions which are not supported by the graphics card are emulated in software. OpenGL can only draw in a so-called *context*, which holds the state of the instance of OpenGL, and represents a frame buffer. It is possible to use more than one context in one application. A platform-dependent library provides the context, so as to maintain the independence of OpenGL. This is also the reason OpenGL provides only graphics rendering and no other inputs or outputs.

The first version of OpenGL was developed by Silicon Graphics in 1991. It is now managed by the Khronos Group. OpenGL is free to use without a license, but hardware manufacturers need a license to create a hardware OpenGL implementation [SGI, 2017]. The library provides the possibility to create extensions to expand the functions of the current release. If an extension becomes commonly used, the OpenGL architecture review board might integrate it into the standard for the next release. In the beginning, only a fixed function pipeline approach was supported. With the introduction of OpenGL version 2.0, the C-like shader language *OpenGL Shading Language* (GLSL) was added. Version 3.0 deprecated the fixed function part of the library and in version 3.1 the deprecated functions where removed. The library changed to a completely shader-based one. In the 3.1 update, profiles were added to the specification. Since every OpenGL version and every profile supports different parts of OpenGL, it is possible to obtain a context for a particular version of OpenGL [Khronos, 2017a]. The standard reference for OpenGL is the *OpenGL Programming Guide: The Official Guide To Learning OpenGL* [Shreiner et al., 2013].

3.1.1 Open Graphics Library for Embedded Systems (OpenGL ES)

The Open Graphics Library For Embedded Systems (OpenGL ES) [Khronos, 2017e] is a subset of OpenGL designed to satisfy the needs of an embedded device. The API is lightweight compared to the standard API. To reduce the size of the API, all redundant functions were removed. If there was more than one way to achieve a goal, only the most powerful technique was retained. The first version of the library, OpenGL ES 2.0, eliminated the fixed function pipeline and supported only float and no double variables. Versions 3.0 and 3.1(the latest release) are specified against OpenGL 3.3 with almost the same functional scope. For the shader, the GLSL ES shading language is used, which is very similar to the GLSL language. OpenGL ES runs natively on the major smartphone platforms like Android, iOS, and Windows Phone.

3.1.2 OpenGL 4

OpenGL version 4 is the latest major release of OpenGL, with added functions to stay compatible with Direct3D 11. The update brought compatibility to the API between the the normal and the ES version. The most recent release is OpenGL 4.5 from August 2014 [Khronos, 2017d].

3.1.3 WebGL

WebGL [Khronos, 2017g] is a standard for low-level 3D graphics. The HTML5 canvas element provides the OpenGL context. The features are the same as in OpenGL ES 2.0, but with some minor changes to match the memory managed behaviour of Javascript. WebGL is royalty-free and is maintained by the Khronos Group.

3.1.4 ANGLE

ANGLE [ANGLE, 2017] is an OpenGL ES 2.0 implementation which translates OpenGL calls to DirectX 9 or DirectX 11 calls. ANGLE is used as the WebGL backend for Windows in Google Chrome and Mozilla Firefox. Chrome also uses ANGLE also for all other graphical rendering in Windows.

3.1.5 Mesa

Mesa [Mesa, 2017b] is an open source implementation of the OpenGL standard used primarly on Linux, but also available for Windows. Depending on the supported hardware, all or none of the functions may be emulated by software. The project was started in 1993 and was first released in version 1.0 in 1995 [Mesa, 2017b]. Soon, many people contributed to the project with patches. SGI who had the rights to OpenGL recognised that Mesa brought OpenGL to platforms where it was not officially supported. Vmware, Tungsten Graphics, Intel. and RedHat are the main contributors to the project. The core parts of Mesa are licensed under an MIT license [Mesa, 2017a]. The latest release is version 10.x and supports OpenGL version 3.3. The X.org windowing system uses Mesa as its OpenGL core.

3.2 DirectX and Direct3D

DirectX [Microsoft, 2017a] is a collection of APIs used on Microsoft Windows for all media input and output. It provides high-level functions for 2D and 3D graphics, sound, network communication, input, and output. DirectX is also used on the Microsoft console, Xbox. In the beginning of Microsoft Windows, only WinAPI was available to generate Windows-compatible programs, but this API did not support optimised access to the graphics chip. Programs like games or CAD programs, which needed to use these features, were often created for DOS, because there the control of the hardware lies in the

hands of the programmer. To encourage industry to develop for Windows 95, Microsoft created an API allowing developers better control of the underlying hardware. This lead to the creation of DirectX.

DirectX has several parts: Direct2D for 2D graphics, Direct3D for 3D graphics, DirectSound to play sound effects, and DirectMusic to play MIDI music. After the release of the Xbox, XAudio2 replaced DirectSound. DirectInput is responsible for input devices like mouse, gamepad, joystick, and supports force-feedback. XInput is used exclusively for the connection of the Xbox-360 controller on Windows. DirectSetup checks if the existing DirectX version is new enough for the installed program. DirectX provides a hardware abstraction layer for every aspect of its library to provide the necessary speed for complex games. Microsoft maintains the documentation of DirectX [Microsoft, 2014].

Direct3D is the display driver for 3D graphics. In version 8, the first programmable pipeline was added with pixel and vertex shaders. Also, the DirectDraw framework was replaced by Direct3D which is now also responsible for drawing 2D graphics. In DirectX version 9.0, the High Level Shading Language (HLSL) was added with many other enhancements. This version remained the standard for many years, because the next version released needed Windows Vista to run, which was not as popular as Windows XP. At the time, many people skipped Vista and waited until Windows 7 was released.

The update to DirectX 10 brought a substantial update of the shader model in Direct3D, with the introduction of the geometry shader and the unification of access to resources in different shaders. Also, compatibility bits, which communicated the hardware features present on the system, were removed and a minimum standard was defined that is needed to run DirectX 10. This effort was made to reduce the necessary code to check if the needed functions are available and provide alternatives if they are not present. In version 10, the fixed function pipeline was removed in favour of programmable pipelines. With Windows 7, DirectX was upgraded to version 11. Shader model 5 was introduced to Direct3D, opening the shader pipeline for non-graphical calculations like scientific calculations, or physics modeling. The update brought the concepts of feature levels to Direct3D, addressing features in DirectX 9 to 11 compatible graphics cards. The programmer develops one rendering pipeline and DirectX manages the different available features of the hardware.

3.3 Shaders

Shaders describe how data in a programmable graphics pipeline is processed. The graphics chip has shader units, which can be programmed by shader programs. Each shader unit operates in parallel. In the beginning, there were separate hardware units for every shader type, but nowadays there are unified shader units. Unified shader units can calculate every shader type and make it easy to add new types to the rendering pipeline or to use the rendering pipeline for general purpose computation. A modern GPU has up to a few thousand shader cores. Shaders are programmed using special programming languages such as HLSL, GLSL, Cg, or even assembler.

3.3.1 Vertex Shaders

A vertex shader transforms the 3D model coordinates of the given vertex to 2D screen coordinates. Only existing vertices can be transformed, no new vertices can be created.

3.3.2 Fragment Shaders (Pixel Shaders)

A fragment or pixel shader calculates the colour of each given fragment. The fragment shader does not know the geometry of the scene, only screen coordinates. Fragment shaders can be used to apply techniques like filtering, edge detection, blur, or other 2D post-processing effects. It is also possible to obtain additional input channels for every pixel and generate effects like bump mapping.

3.3.3 Tesselation Shaders

Tesselation shaders are relatively new, introduced in OpengGL 4.0 and DirectX11. This class of shader comprises two shaders: tesselation control shaders and tesselation evaluation shaders. These two in combination can take high-level surfaces and generate a tesselation dependent on various variables. A typical example is dynamic level of detail, where elements far from the camera have lower detail and are less tesselated than elements close to the camera.

3.3.4 Geometry Shaders

A geometry shader is an optional stage in the render process, which takes a single geometric shape as input and can make changes to the shape by adding, merging, or deleting fragments. There is an upper limit to how many new fragments can be generated. Geometry shaders are used for layered rendering, where one primitive is rendered to multiple outputs. The second common task is the transformation of geometry and feeding it back to be stored in an object. These objects can be sent through the rendering pipeline to repeatedly compute complex tasks.

Chapter 4

Waveform Testbed for Real-Time Signal Display

Waveform is a C++ application to compare different methods of drawing a dynamic line chart for real-time monitoring. The application consists of a Render Window and a Settings Panel. The Render Window is composed of the abstract components shown in Figure 4.1. As an example, the Qt Native Render Window is shown in Figure 4.2. The Settings Panel shown in Figure 4.3.

Qt version 5.5 was used to build the GUI and to provide the basis for OpenGL or WebView. Qmake was used to manage the build. Each of the rendering technologies is compiled as its own library. The goal of the software is to achieve a refresh rate of at least 60 frames per second (fps) for a steady flicker-free picture on a 60Hz display.

4.1 Program Structure

The application is structured according to the model-view-controller paradigm. Each of the different rendering technologies is implemented in its own view, but uses the same model and controller. As an example, the class structure of the OpenGL renderer is shown in Figure 4.4. A Render Window contains two Grids. The upper grid is called Grid 1 and contains two signal groups. The lower grid is called Grid 2 and contains one signal group. A signal group defines the range of for the Y axis for all signals in that grid. Signal groups can display signals or bands. A signal is a single waveform. A band consists of two waveforms, an upper and a lower bound and the space enclosed in between. The signals displayed are hard-coded in software and can only be changed there. Each of the signals has approximately 1,400 data points. The model makes the different data sources available for the views. The controller manages all signals and updates the views according to changes in the Settings Panel or when a new frame should be drawn. In order to achieve a frame rate of 60 fps, each frame must be drawn within 16 ms.

4.2 Settings Panel

The Settings Panel shown in Figure 4.3 can be used to specify the renderer, the data source, the drawing style, and the time frame of the data. Two sources of data are supported, shown in the top right of the Settings Panel. The simulator is for testing the application. It generate a sine wave, a rectangular, and a sawtooth signal on Grid 1, signal group 1. Signal group 2 displays another rectangular signal. The second option uses binary WFDB files from PhysioNet [PhysioNet, 2017b]. These files are anonymous real world samples of patient data including ECG and blood pressure. Example files from the Massachusetts General Hospital/Marquette Foundation (MGH/MF) Waveform Database [PhysioNet, 2017c] are used. This database contains recordings of hemodynamic and electrocardiographic waveforms of 250 patients

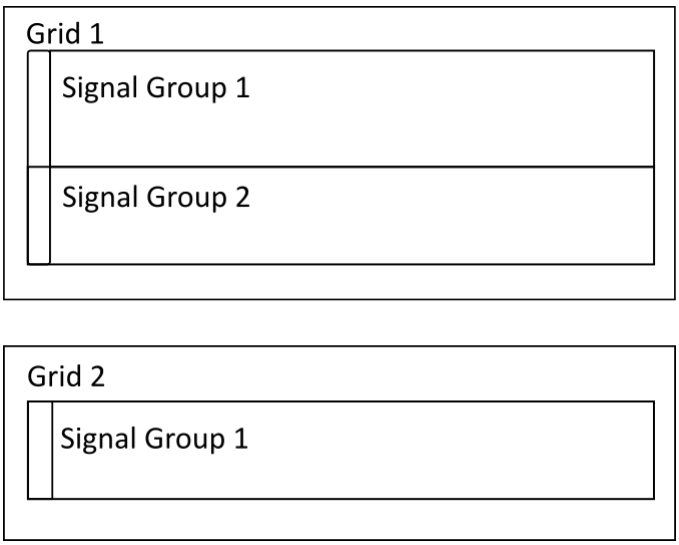


Figure 4.1: The components of the Render Window [Diagram created by the author with Inkscape [Inkscape Community, 2017].].

from critical care units, operating rooms, and cardiac catheterisation laboratories, each around one hour long. For each recording, a patient guide is available containing additional information such as age, gender, diagnosis, and an ECG interpretation [PhysioNet, 2017a]. The WFDB library [PhysioNet, 2017d] is used to read the WFDB files. Grid 1 displays "ECG Lead I", "ECG Lead II", and "ECG Lead V" on signal group 1 and the ART signal(arterial blood pressure) on signal group 2. Grid 2 displays a band for the signals "BP:Sys" and "BP:Diasy", the systolic blood pressure and diastolic blood pressure.

Within a view, three different modes to display the data are available. In Scroll mode, new data enters the view from the right and leaves the view to the left, scrolling through. In Overwrite mode, new data is drawn from left to right overwriting any previously drawn data. In Fixed mode, the view displays a signal starting at a specific time code given by RefTime. It is possible to pan the signal using the mouse, the single step button, or by manually inputting a time code.

4.3 Render Window

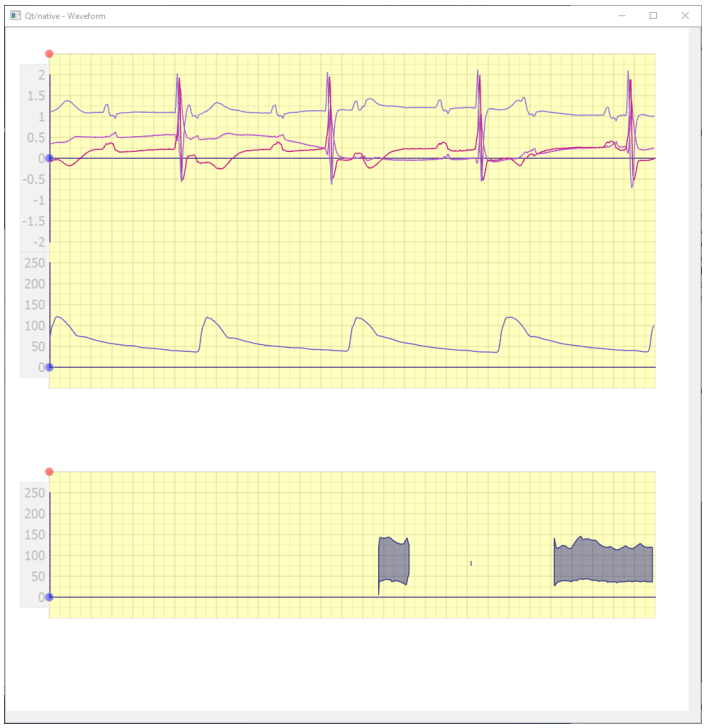
The Render Window shows the selected waveform data using one of the five available renderers.

4.3.1 Qt Native

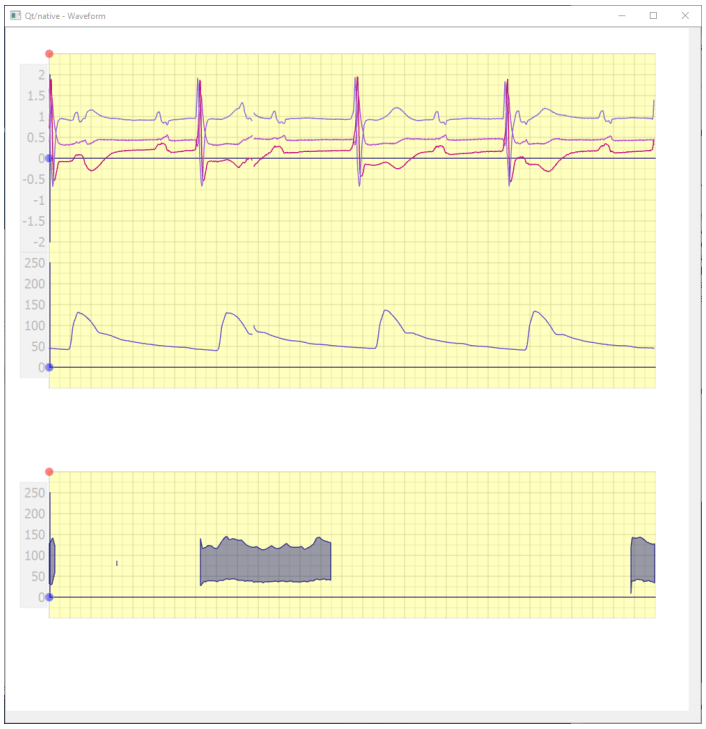
The Qt native implementation of the Waveform renderer, shown in Figure 4.2, served as a reference for the design of the output and the software. The Qt classes QGraphicsPathItem and QPaintDevice were used to construct the QGraphicsScene, which is displayed with a QGraphicsView, as shown in Listing 4.1. All grids and signals are added to the m_Scene variable. The QGraphicsView uses OpenGL to accelerate the rendering of its content. The function to draw a signal is shown in Listing 4.2. These classes are part of the Graphics View Framework [QtC, 2017a] of Qt5.

4.3.2 OpenGL

The OpenGL implementation shown in Figure 4.5 uses the Qt widget QGLWidget to provide the necessary OpenGL context. Two simple OpenGL shaders are used to draw the necessary 2D graphics, the



(a) Scroll.



(b) Overwrite.

Figure 4.2: The Qt native Render Window [Screenshots taken by the author.].

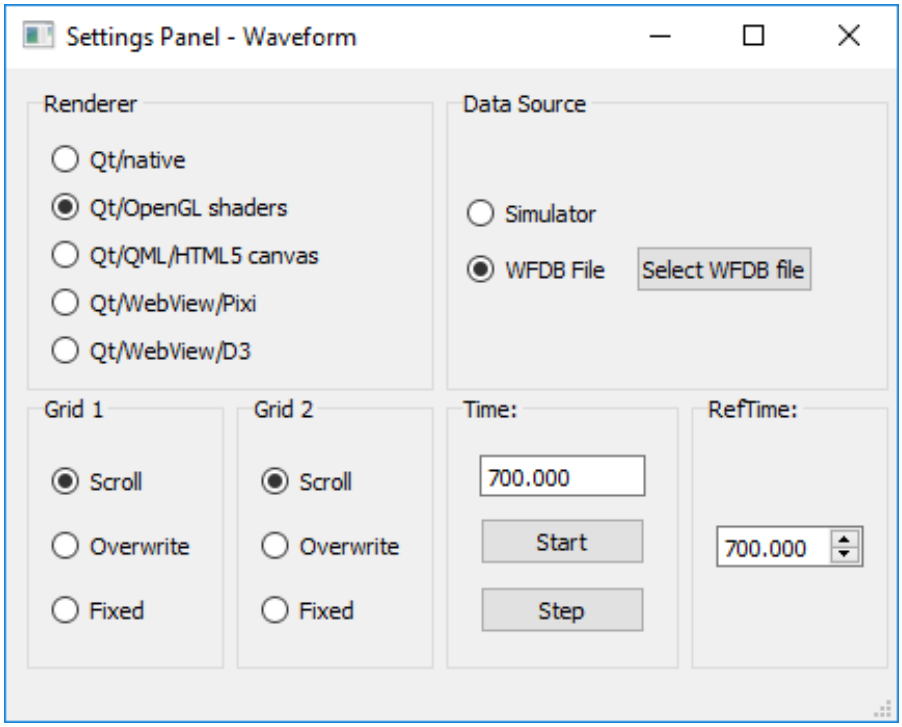


Figure 4.3: The Settings Panel of Waveform shows the various control settings [Screenshot taken by the author.].

```
1  QGraphicsView view;
2  // To enable OpenGL support
3  view.setViewport(new QGLWidget(QGLFormat(QGL::SampleBuffers)));
4  view.setRenderHint(QPainter::HighQualityAntialiasing);
5  view.setViewportUpdateMode(QGraphicsView::BoundingRectViewportUpdate
6  );
7
8  // VGraphicsScene is a QGraphicsScene derived class providing grid
9  coordinates
10 VGraphicsScene* m_Scene = new VGraphicsScene(m_Controller);
11 view.setScene(m_Scene);
```

Listing 4.1: Qt/Native setup of the scene and view.

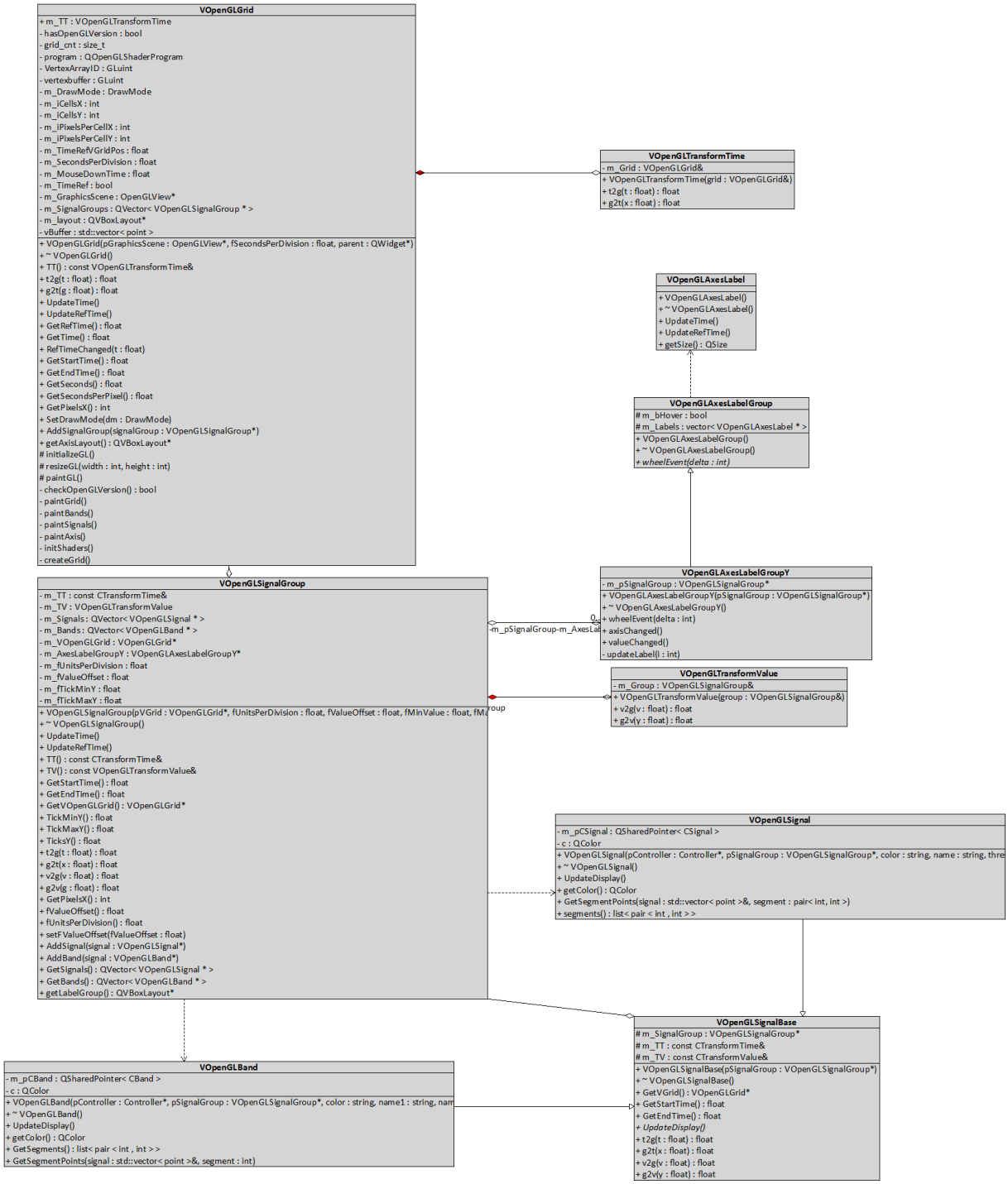


Figure 4.4: Class diagram of the OpenGL implementation [Diagram created by the author with Umbrello.].

```
1  // get signal data from controller
2  vector<float> x = m_pCSignal->x();
3  vector<float> y = m_pCSignal->y();
4
5  //get segments of data from controller
6  list<pair<int,int> > segments = m_pCSignal->segments();
7
8  size_t pixels = x.size();
9  if ( pixels > 0 )
10 {
11   // new line
12   QPainterPath path;
13
14   pair<int,int> p;
15   foreach (p, segments)
16   {
17     int idx0 = p.first, idx1 = p.second;
18
19     // move line to starpoint
20     path.moveTo(x[idx0],y[idx0]);
21
22     // add datapoints to line
23     for ( unsigned int i = idx0; i < idx1; i++ )
24     {
25       path.lineTo(x[i],y[i]);
26     }
27   }
28
29   // set the path to the class,it is a QGraphicsPathItem
30   setPath(path);
31 }
32
33 // finally let the class paint itself
34 QGraphicsPathItem::paint(painter, option, widget);
```

Listing 4.2: Qt/Native drawing of one signal with a QGraphicsPathItem.

vertex shader `vshader.vsh` shown in Listing 4.3, and the fragment shader `fshader.fsh` shown in Listing 4.4. To compile, link and bind the shaders, Qt provides the QOpenGLShaderProgram class, its usage is shown in Listing 4.5. Labels are drawn using Qt rather then OpenGL. Using two different drawing methods for signals and labels creates problems positioning the signals with regard to the labels. The viewport is scaled to match the size of the reference implementation, so no further scaling of the data coming from the model is necessary. The signals are drawn as line strips, the data points of the bands are sorted in such a way that triangle strips can be used to draw the interior of the bands. The function to draw the signals is shown in Listing 4.6. The signals are anti-aliased, the grid is not.

4.3.3 QML/HTML5 Canvas

The QML/HTML5 Canvas implementation shown in Figure 4.6 is split into C++ and QML files. The C++ code prepares and organises the data so that the QML code simply has to draw it. To show a QML view in the render window it must be wrapped in a QWidget, as shown at Line 8 of Listing 4.7. To call public C++ methods from QML, the corresponding class must be registered with QML. Then, methods

```
attribute vec2 coord2d;
uniform mat4 matrix;

void main(void) {
  gl_Position = matrix x vec4(coord2d.xy, 0, 1);
}
```

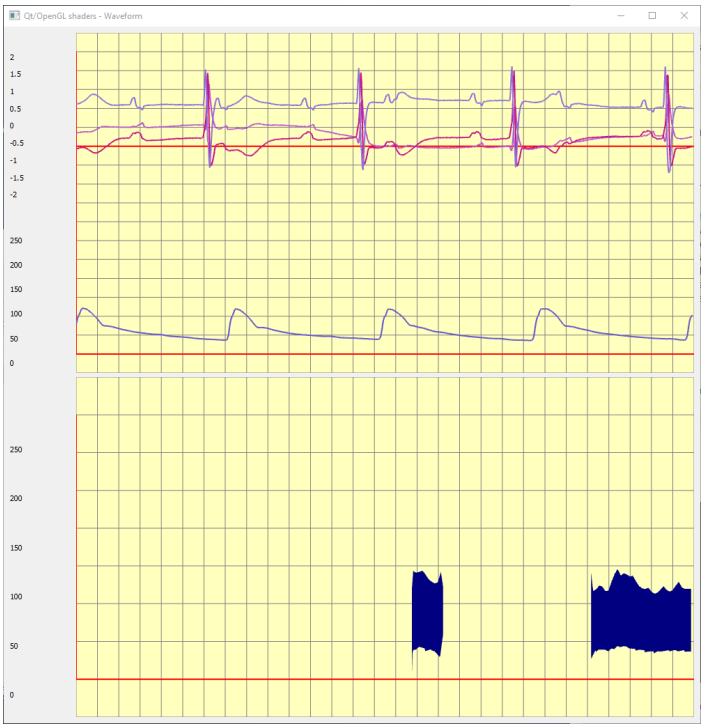
Listing 4.3: The Qt/OpenGL vertex shader `vshader.vsh` used for signal drawing.

```
uniform vec4 f_color;

void main(void)
{
  gl_FragColor = f_color;
}
```

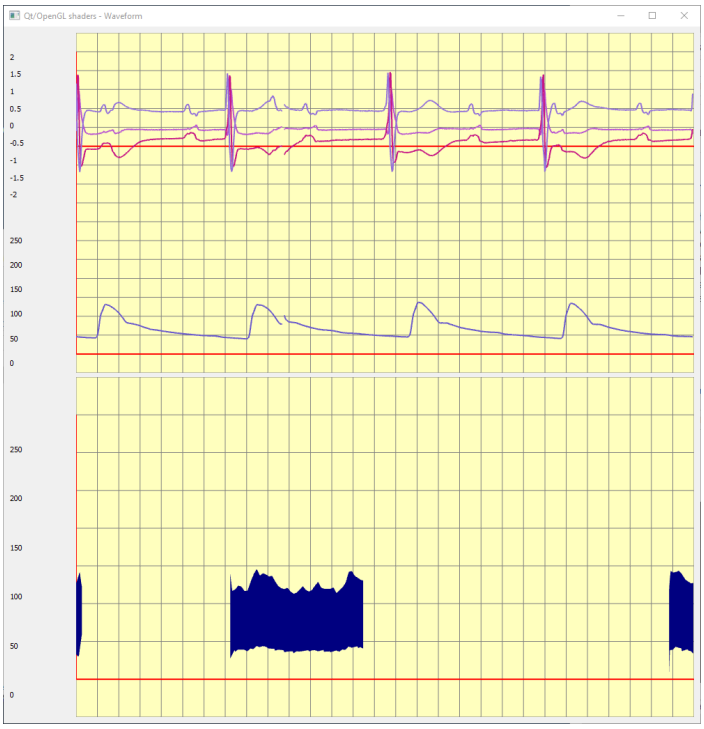
Listing 4.4: The Qt/OpenGL fragment shader `fshader.vsh` used for signal drawing.

```
QOpenGLShaderProgram program;
// Compile vertex shader
if (!program.addShaderFromSourceFile(
      QOpenGLShader::Vertex, ":/shader/vshader.vsh"))
close();

// Compile fragment shader
if (!program.addShaderFromSourceFile(
      QOpenGLShader::Fragment, ":/shader/fshader.fsh"))
close();

// Link shader pipeline
if (!program.link())
close();

// Bind shader pipeline for use
if (!program.bind())
close();
```

Listing 4.5: Qt/OpenGL compiling, linking, and binding the shaders.



(a) Scroll.



(b) Overwrite.

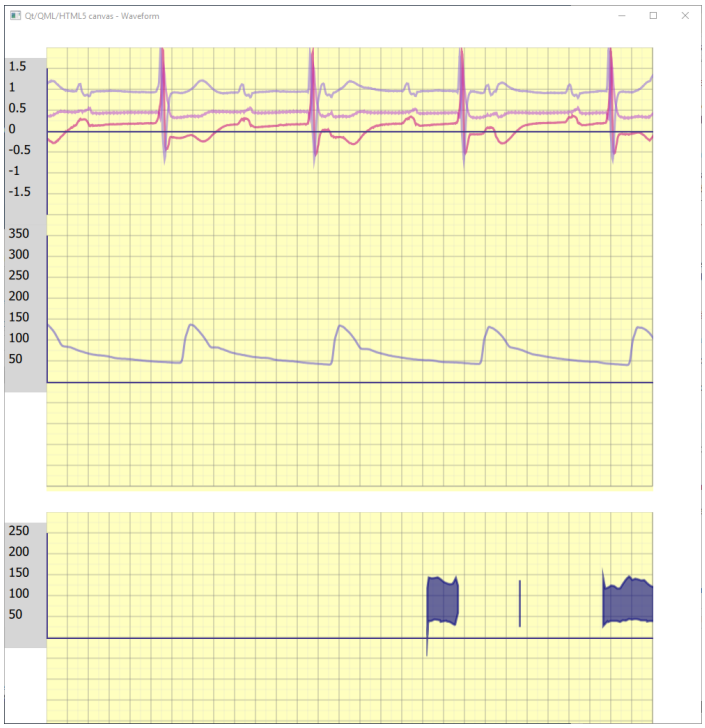
Figure 4.5: The OpenGL Render Window [Screenshots taken by the author].

```
1  void VOpenGLGrid::paintSignals()
2  {
3  int colorLocation = program.uniformLocation("f_color");
4  foreach (VOpenGLSignalGroup* sigGroup, m_SignalGroups)
5  {
6    QVector<VOpenGLSignal*> sigs = sigGroup->GetSignals();
7    foreach (VOpenGLSignal* sig, sigs)
8    {
9      //set color in the shader
10     QColor c = sig->getColor();
11     program.setUniformValue(colorLocation, c);
12
13     list<pair<int,int> > segments = sig->segments();
14     for (int i = 0; i < segments.size(); ++i)
15     {
16       //copy values of segment to OpenGL datatypes
17       std::vector<point> signal;
18       sig->GetSegmentPoints(signal,segments.front());
19       segments.pop_front();
20
21       //configure Buffer and draw the signal
22       glBufferData(GL_ARRAY_BUFFER, signal.size()*sizeof(point),
23         &(signal[0]), GL_DYNAMIC_DRAW);
24       glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
25       glDrawArrays(GL_LINE_STRIP, 0, signal.size());
26     }
27   }
28 }
29 }
```

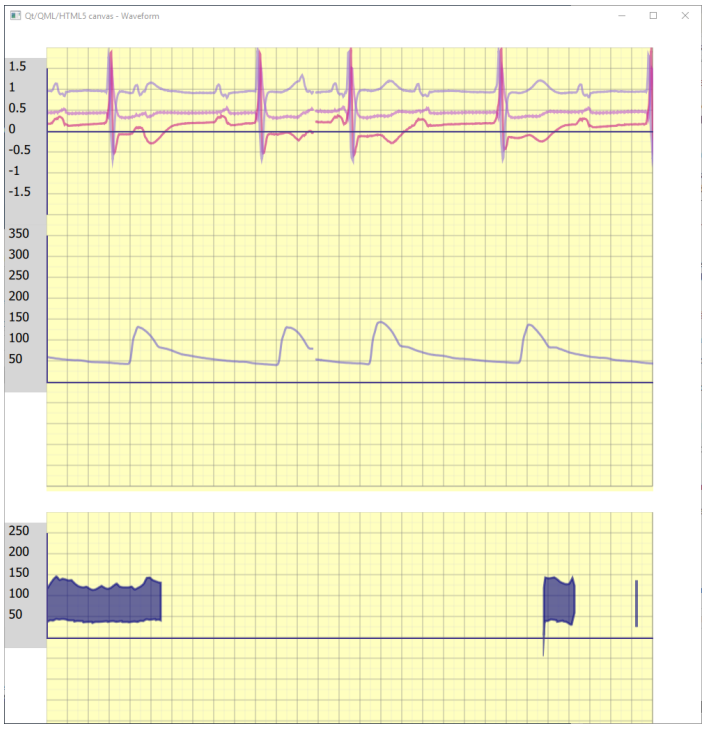
Listing 4.6: Qt/OpenGL drawing signals using OpenGL shaders.

from that class marked with the Q_INVOKABLE macro can be called. Line 11 of Listing 4.7 shows the registering of a C++ class to all QML elements under the name `_ViewQml`. To connect signals to QML slots named children can be searched at the C++ side as shown in Line 15 of Listing 4.7. These slots are then connected to suitable signals of the C++ side.

Drawing is done in a dedicated QML HTML5 2D Canvas element [W3Schools, 2017] which supports drawing simple elements like lines and curves. The definition of the canvas element and its paint slot can be seen in Line 14 of Listing 4.8. The canvas uses the Canvas.Image QImage render target, which can perform background rendering so as not to block the UI during longer paint operations. The second render target is Canvas.FramebufferObject which uses OpenGL hardware acceleration for faster rendering. The drawing of a single waveform signal is shown in Listing 4.8. The canvas is refreshed at 60 fps, other QML elements like labels are only redrawn as necessary, such as after a resize. The QML elements are anchored relative to each other to guarantee a resizable view.



(a) Scroll.



(b) Overwrite.

Figure 4.6: The QML and HTML5 Canvas Render Window [Screenshots taken by the author].

```
1  // path to QML main file
2  QUrl url("qrc:///qml/VQmlGraphicScene.qml");
3  // create view for QML set qml mail file
4  view = new QQuickView();
5  view->setSource(url);
6
7  // put view in Container widget to show in renderer window
8  m_QMLView = QWidget::createWindowContainer(view);
9
10 // set this class as context
11 view->rootContext()->setContextProperty("_ViewQml",this);
12
13 // find QML elements and save in list to connect signals to them
14 QQuickItem* root = view->rootObject();
15 QList<QObject*> grids = root->findChildren<QObject*>(QString("grid"));
16 QList<QObject*> axes = root->findChildren<QObject*>(QString("axis"));
17
18 //connect elments from QML to C++ signals
19 foreach (QObject* axis, axes) {
20   QObject::connect(this, SIGNAL(init()),axis, SLOT(axisChanged()));
21 }
22 foreach (QObject* grid, grids) {
23   QObject::connect(this, SIGNAL(timer()),grid, SLOT(requestPaint()));
24   QObject::connect(this, SIGNAL(rePaint()),grid, SLOT(requestPaint()));
25 }
26 //initialize the QML elements
27 emit init();
```

Listing 4.7: Qt/QML/HTML5 setting up the renderer.

```
import "DrawFunctions.js" as MyDraw
// Per grid one rectangle filled with a HTML5 Canvas
Rectangle {
  id:grid
  objectName: "grid"
  anchors.fill: parent
  anchors.leftMargin: m_leftMargin
  anchors.topMargin: m_topMargin
  anchors.rightMargin: m_rightMargin

  color: "#40FFFF00"

  // Canvas with the size of the parent
  Canvas{
    id: myCanvas
    anchors.fill: parent

    onPaint: {
      var ctx = getContext("2d");
      MyDraw.draw(ctx,_grid1,width,height,m_iPixelsPerCellX,m_iPixelsPerCellY);
    }
  }
}
```

Listing 4.8: Qt/QML/HTML5 creating the HTML5 canvas element and its 2D context.

```
function drawSignal(ctx,_grid,signalGroup,signal)
{
  ctx.strokeStyle = _grid.getSignalColor(signalGroup,signal);
  var segments = _grid.getSegments(signalGroup,signal);

  for(var seg = 0; seg < segments.length; seg+=2)
  {
    // Array data for X Y coords in the form of xyxyxyxy...
    var XYcoords = _grid.getData(signalGroup,signal,segments[seg],segments[seg+1]);
    ctx.beginPath();
    ctx.moveTo(XYcoords[0],0 - XYcoords[1]);
    for (var i = 0; i < XYcoords.length-1; i+=2)
    {
      ctx.lineTo(XYcoords[i],0 - XYcoords[i+1]);
    }
    ctx.stroke();
  }
}
```

Listing 4.9: Qt/QML/HTML5 drawing the signals.

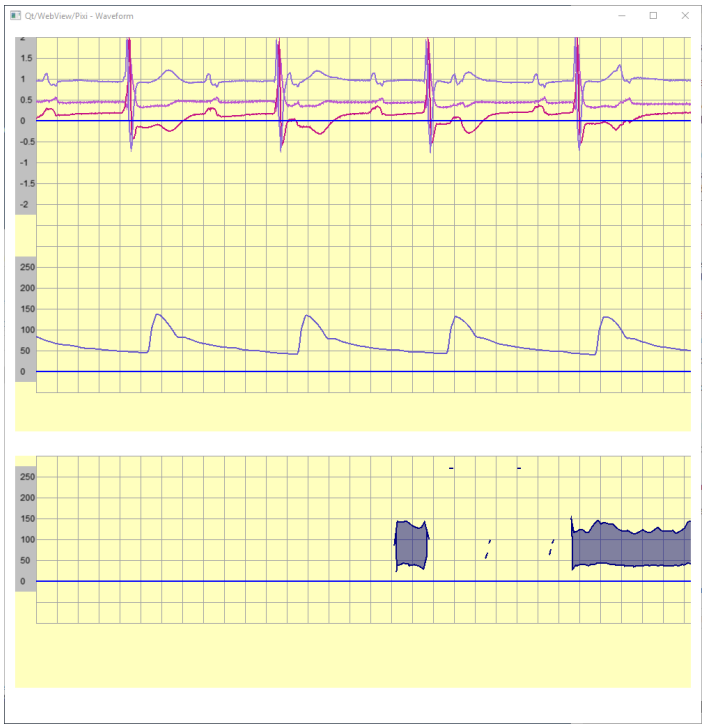
```
1  void ViewWebView::loadFinished(bool)
2  {
3    // register the class with the name QTWebView
4    page()->mainFrame()->addToJavaScriptWindowObject("QTWebView", this);
5    // run javascript functions
6    page()->mainFrame()->evaluateJavaScript("getWidth();");
7    page()->mainFrame()->evaluateJavaScript("setupPixi();");
8
9    foreach (VWebViewGrid* grid, m_grid)
10   {
11     grid->updateData();
12   }
13 }
```

Listing 4.10: Qt/WebView/Pixi registering the C++ class to JavaScript and calling a JavaScript function from C++.

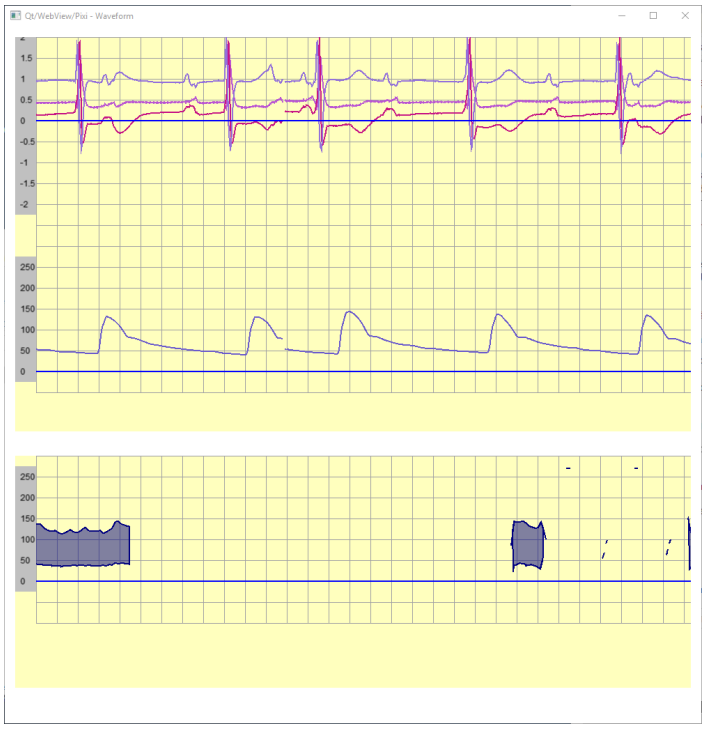
4.3.4 WebView/Pixi

A web page can be displayed in a QT widget powered by WebKit. The page is shown in a QWebPage element which provides the link between the web page and the C++ side of the application. The data is edited on the C++ side and only a vector of the processed data is sent and displayed on a web page. From the C++ side the method evaluateJavaScript(String script) can be used to run JavaScript code on the web page. To communicate from the web page to C++, an object has to be registered with the QWebpage with a name. This name can be used to call methods from this object from JavaScript. The code to register the C++ class and call two JavaScript functions is shown in Listing 4.10. To access values from C++ lists a type conversion to compatible QVariant typs is necessary. Qt Web Inspector is available to debug the JavaScript code.

PixiJS [Groves, 2017] is an open source JavaScript graphics library for 2D drawing. Version 4.5.1 of Pixi was used, loaded locally from the hard disk. The WebView Pixi renderer is shown in Figure 4.7. The Pixi rendering engine uses WebGL, if available to accelerate drawing by using the GPU. If no WebGL support is available, Pixi has a seamless automatic fallback mechanism to a slower SVG mode. To use Pixi, the rendering canvas is first added to an HTML page, which is then displayed by a WebView instance, as shown in Listing 4.11. The function to draw one signal is shown in Listing 4.12. The waveform is a Pixi.Graphics element. This element is added to the Pixi stage, its colour is set and it is moved to the starting point of the waveform. The lineTo command is used to draw the remaining points of the line. For each frame, the whole scene is completely redrawn.



(a) Scroll.



(b) Overwrite.

Figure 4.7: The Webview/Pixi Canvas Render Window [Screenshots taken by the author.].

```
function setupPixi()
{
  document.body.remove
  stage,lenght = 0;

  m_iPixelsPerCellX = 1;
  m_iPixelsPerCellY = 1;
  m_iPixelsPerCellX = QTWebView.GetiPixelsPerCellX(0);
  m_iPixelsPerCellY = QTWebView.GetiPixelsPerCellY(0);

  // create an new instance of a pixi stage
  stage[0] = new PIXI.Container();
  stage[1] = new PIXI.Container();

  // create a renderer instance.
  var size = getWindowSize();
  renderer = PIXI.autoDetectRenderer(size.x-m_iPixelsPerCellX,
    size.y/3.0*1.7,{backgroundColor : 0xFFFFBE});
  renderer2 = PIXI.autoDetectRenderer(size.x-m_iPixelsPerCellX,
    size.y/3.0*1.0,{backgroundColor : 0xFFFFBE});

  // add the renderer view element to the DOM

  document.body.replaceChild(renderer.view,document.body.childNodes[0]);

  document.body.replaceChild(renderer2.view,document.body.childNodes[1]);
  renderer.view.style.padding = m_iPixelsPerCellX/2.0 + 'px';
  renderer2.view.style.padding = m_iPixelsPerCellX/2.0 + 'px';

  animate();
}
```

Listing 4.11: Qt/WebView/Pixi Setup of the Pixi Renderer.

```
function drawSignal(signalGroup,grid,iCellsX,iCellsY)
{
  // QTWebView is the registered C++ Class
  for(var signal = 0; signal <
    QTWebView.GetSignalCnt(grid,signalGroup); signal++)
  {
    for(var segment = 0; segment <
      QTWebView.GetSegmentsCnt(grid,signalGroup,signal,false); segment++)
    {
      var line = new PIXI.Graphics();
      // add it the stage so we see it on our screens
      stage[grid].addChild(line)

      var color = QTWebView.GetColorFromSignal(grid,signalGroup,signal);
      // Qt used # and Pixi needs 0x for Hex
      color = color.replace('#','0x');
      line.lineStyle(2, color, 1);

      var dataList = QTWebView.GetSegmentSignalData(grid,
        signalGroup,signal,segment);

      // *m_iPixelsPerCellX to scale to the viewport
      line.moveTo(dataList[0]*m_iPixelsPerCellX, -dataList[1]*m_iPixelsPerCellY);
      for(var cnt = 2; cnt < dataList.length; cnt+=2)
      {
        line.lineTo(dataList[cnt]*m_iPixelsPerCellX,
          -dataList[cnt+1]*m_iPixelsPerCellY);
      }
    }
  }
}
```

Listing 4.12: Qt/WebView/Pixi drawing a signal.

```
function setupSVG(id)
{
  // get Browser window size and calculate usable width/height
  var size = getWindowSize();
  var width =  size.x - margin.left - margin.right;
  var height = size.y/2 - margin.top*2 - margin.bottom*2;

  // remove all grids from previous calls to setupSVG
  d3.select("body").select("#grid"+ id.toString()).remove();

  // add the new Grid paragraph
  var p = d3.select("body").append("p")
                           .attr("id","grid" + id.toString());
  // add the SVG element to the paragraph
  // stores all other SVG elements
  var svg = p.append("svg")
             .attr("width", width + margin.right)
             .attr("height", height);

  // translate whole graph
  var g = svg.append("g")
             .attr("transform", "translate(" + margin.left + "," + 0 + ")");

  // yellow background color of the grid
  g.append("rect")
   .attr("width", "100%")
   .attr("height", "100%")
   .attr("fill", "rgb(255,255,190)");

  // group element for signals, used to store the signals
  g.append("g").attr("id","signals");
}
```

Listing 4.13: Qt/WebView/D3 creating the grids and setting up the SVG elements.

4.3.5 WebView/D3

D3 [Bostock, 2017] is a well known JavaScript library used to draw graphical primitives by dynamical inserting SVG nodes into the browser DOM. D3 v3.5.17 is used and is loaded locally from the hard disk. The WebView D3 renderer is shown in Figure 4.8. Initially, SVG and background elements are added to the DOM, as shown in Listing 4.13. For every frame, the SVG elements representing the signals are deleted, and new elements representing the updated state of the signals are inserted. Listing 4.14 shows how elements are selected and deleted in D3. Incoming data is prepared and managed on the C++ side. The data points are then scaled to the viewport, and added to a array as x and y coordinates, as shown in Listing 4.15. A line generator transforms the data points of the array to a line. The whole array is put in the line generator and the result is then appended to the SVG group element, as shown in Listing 4.16. For each frame, only the signals are deleted and redrawn, the remaining elements stay the same and need not be touched or redrawn.

```
// Select html body
var body = d3.select("body");

// Select grids
var p = body.select("#grid" + grid.toString());

// Select all svg elements in grid
var svg = p.select("svg");

// Select group element of all signals
var g = svg.select("#signals");

// select all children of the group element and delete them.
g.selectAll("*").remove();
```

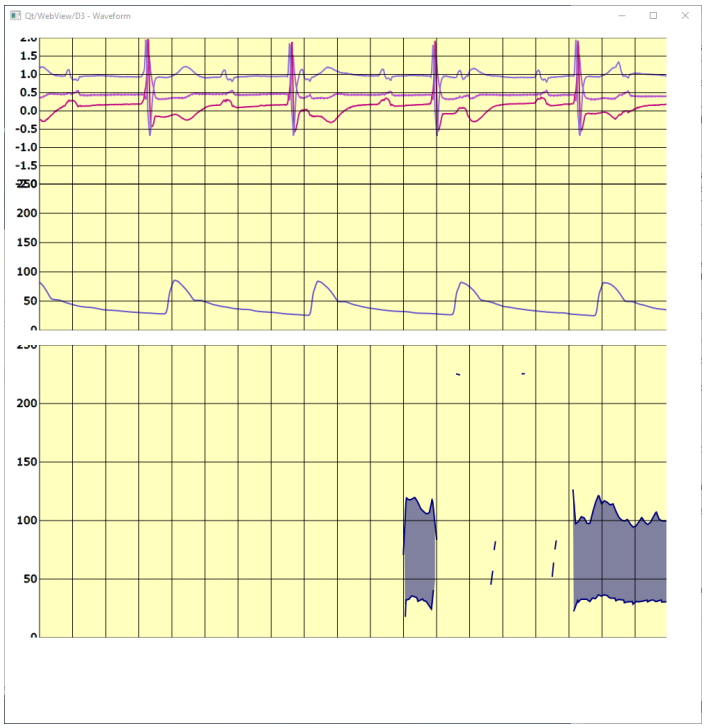
Listing 4.14: Qt/WebView/D3 selecting elements and deleting signals in D3.

```
// the scale functions
var y = d3.scale.linear().domain([yMax, yMin]).range([0, height]);
var x = d3.scale.linear().domain([0, cellsX]).range([0, width]);

// add incoming datapoints to the list and scale them
for(var cnt = 0; cnt<dataList.length; cnt+=2)
{
  data.push(
            { x:dataList[cnt],
              y:dataList[cnt+1] }
  );
}
```

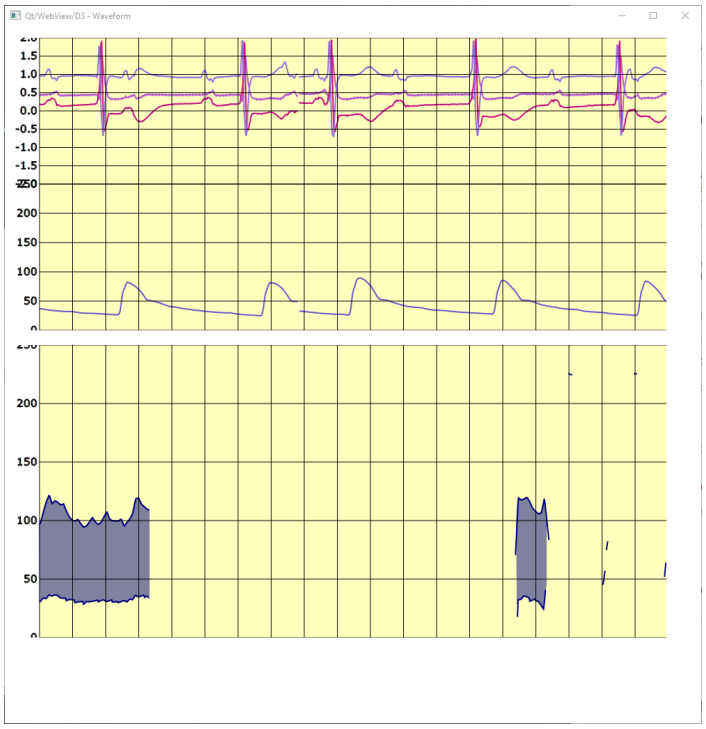
Listing 4.15: Qt/WebView/D3 scaling and adding data points.

```
// line generator
var line = d3.svg.line()
            .x(function(d) { return d.x; })
            .y(function(d) { return d.y; });

// finally add the line to the group svg element
g.append("svg:path").attr("d", line(data));
```

Listing 4.16: Qt/WebView/D3 convert the data array to a line and adding the line to the SVG.



(a) Scroll.



(b) Overwrite.

Figure 4.8: The WebView/D3 Render Window [Screenshots taken by the author.].

Renderer	fps
Qt/native	62.30
Qt/OpenGL	62.30
Qt/QML/HTML5 Canvas	29.17
Qt/WebView/Pixi	35.63
Qt/WebView/D3	11.43

Table 4.1: The performance of the five rendering engines in frames per second(fps).

4.4 Performance Comparison

The goal is to draw waveforms with a refresh rate of at least 60 frames per second (fps). The application was run five times for 20 seconds with each of the five renderers. The exact time in milliseconds that the program ran and the number of frames drawn were used to calculate the fps. QElapsedTimer was used to gather the running time of the application. QElapsedTimer was specially designed to measure how much time a function needs. On Microsoft Windows, it uses the high-resolution performance counter provided by the operating system. On Unix derivatives, an equivalent counter is available. This clock is monotonic and does not overflow. After every run, the program was restarted afresh to ensure the same data was used to get comparable results. The test was run on a Windows 10, 64 bit machine with an i7-4770 Cpu with 3,4Ghz and 8 cores, 8GB ram and the integrated Intel HD Graphics 4600 graphics card.

The drawing rates for each of the five waveform renderers can be seen in Table 4.1. Only the native Qt implementation and the OpenGL renderer achieved the goal of 60 fps. The WebView/D3 renderer was the slowest at 11.43 fps, which is perhaps not surprising since D3 uses SVG without GPU acceleration. The three slowest renderers (QML/HTML5 Canvas, WebView/Pixi, and WebView/D3) all registered CPU usage of 12.4% on the test machine, suggesting that one of the eight cores is completely saturated and better multi-threading might speed up the implementation. The two WebView implementations have to transport and convert all the data from the C++ side to the JavaScript side as well as maintaining the overhead of a WebKit browser.

Chapter 5

SmartNIBP Blood Pressure Monitor

The motivation to test different methods to draw waveform line graphs arose during the development of the SmartNIBP real-time blood pressure monitoring device. The areas of application of the device are medical surgeries and patients of intensive care units, where staff can monitor the status of patients from a single place. The SmartNIBP device can be seen in Figure 5.1. The front has eight inputs for ECG leads. On the back, there are connectors for network and power. The schematic structure of the device is shown in Figure 5.2.

The device measures the blood pressure indirectly by means of the ECG. ECG electrodes are placed on the chest and on the back of a patient in specific locations. The heart generates locally different electrical fields so spatially different signals are captured. The different signals are called leads and are numbered with roman numbers. The expansion and contraction of the artery changes the voltage drop between different ECG sensors. After calibration with a conventional blood pressure upper arm cuff, it is possible to obtain a continuous measurement. Recalibration is required at set intervals to counter drift.

The SmartNIBP device has no attached display. Data is sent using TCP/IP over a fibreoptic cable to another computer to be displayed. The SmartNIBP Display application shown in Figure 5.3 can be used to data sent over the network. This application can also be used to save the data to disk. Another application, SmartNIBP Monitor, is under development for remote viewing of the live data using a Raspberry Pi3. A screenshot of the unfinished application can be seen in Figure 5.4. The SmartNIBP Analyser application shown in Figure 5.5 can be used to watch and analyse the saved data later.

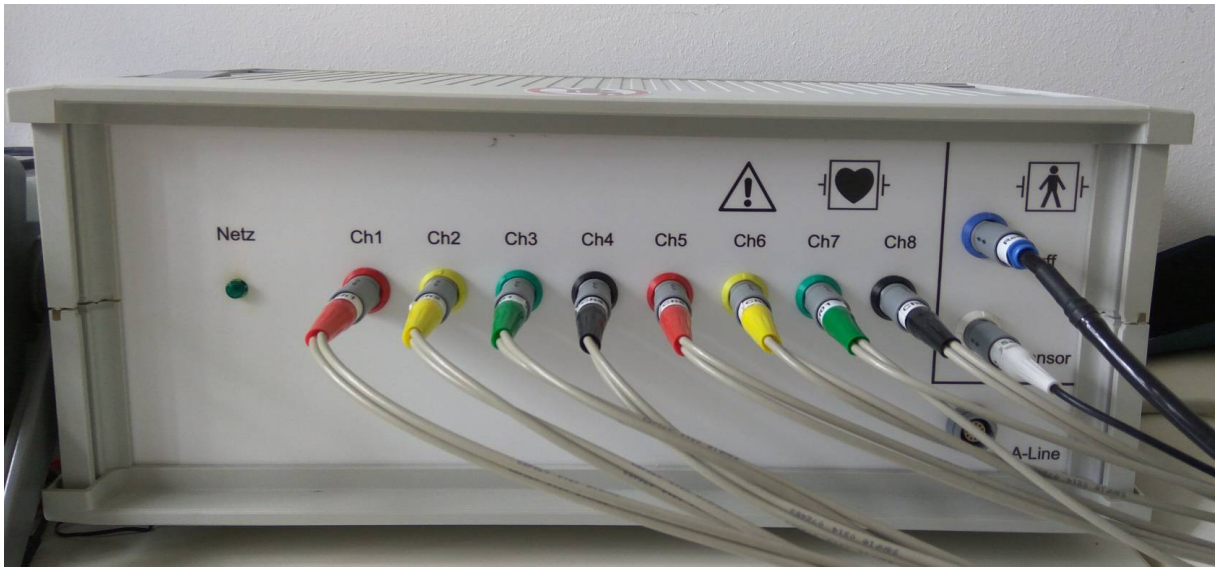


Figure 5.1: SmartNIBP front with eight inputs for ECG leads. [Photograph taken by the author.].

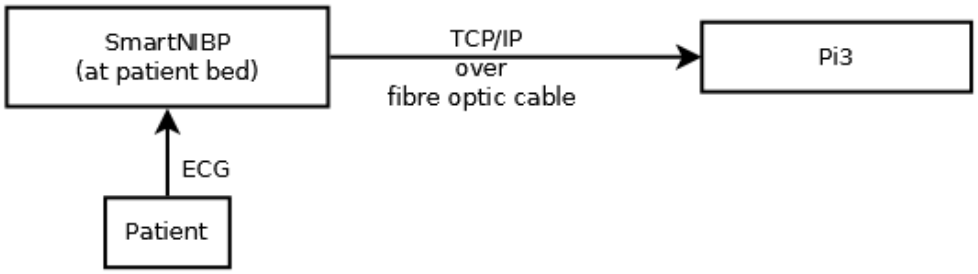


Figure 5.2: The architecture of the SmartNIBP blood pressure monitoring device [Picture created by the author with Dia [McCann, 2017].].

Figure 5.3: SmartNIBP display. On the right side are the controls to switch views and to save and pause the data. [Screenshot taken by the author.].

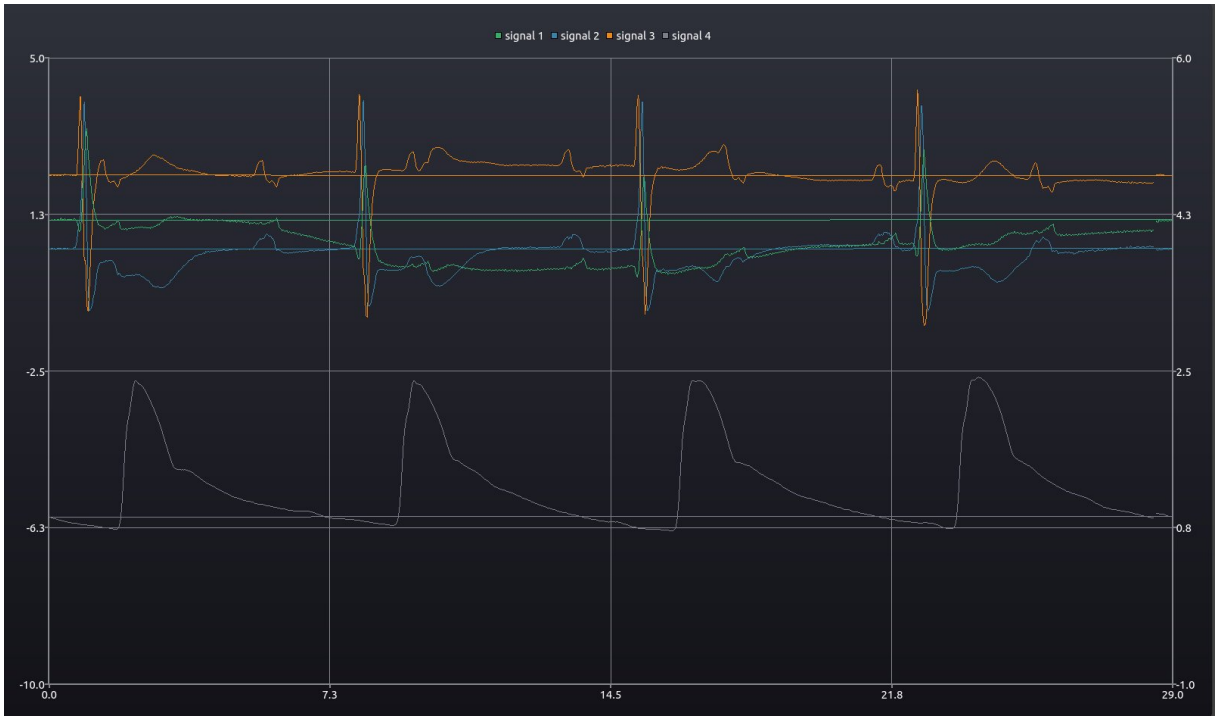


Figure 5.4: The SmartNIBP Monitor prototype. [Screenshot taken by the author.].

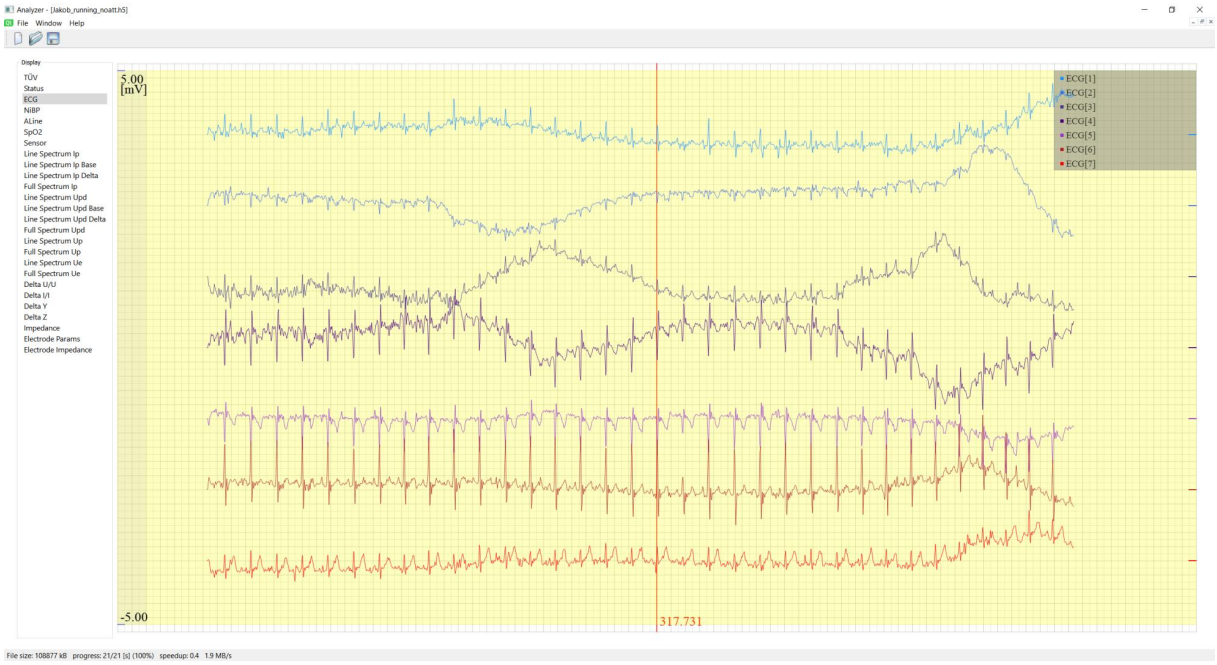


Figure 5.5: The SmartNIBP Analyzer application is used to view recorded data [Screenshot taken by the author.].

Chapter 6

Future Work

In the future, SmartNIBP will use a combination of QML and OpenGL. QML is great for specifying the position and behaviour of interface components, making it easy to create layouts for different kinds of display. A custom QML element containing a frame buffer with OpenGL content will be used to draw the actual waveforms. The frame buffer is managed in C++ but draws with hardware-accelerated OpenGL shaders.

An experimental implementation using QtCharts [Qtc, 2017a] will also be built. QtCharts uses QtGraphicsView and provides typical chart components for Qt. A prototype running on the Pi3 can be seen in Figure 6.1.

In the near future, it will be necessary to switch from QtWebKit to the new QtWebEngine, because the QtWebKit module has been removed from Qt from version 5.6 [Qtc, 2017b] on. However, the latest version of QtWebEngine is still somewhat buggy and is lacking some essential functionality.

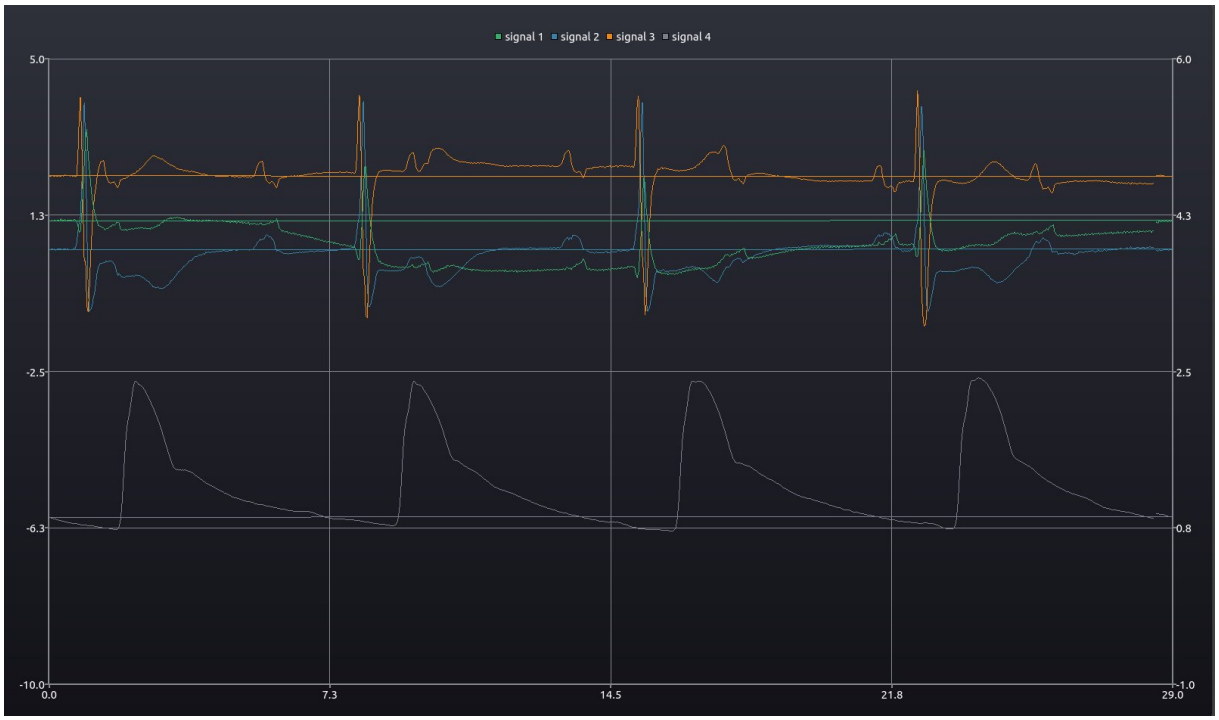


Figure 6.1: Qt/QtCharts prototype of the display running on the Pi3 [Screenshot created by the author.].

Chapter 7

Concluding Remarks

The Waveform software was created to test different graphics technologies to display dyanamic waveforms with a large number of data points. The intended application is non-invasive, continuous blood pressure measurement in a hospital setting.

In Chapter 2, several different Graphical User Interface (GUI) libraries were researched and compared. Qt was chosen for this project. t is an open source C++ framework for cross-platform GUI development. Qt is useable with different programming languages and can either adopt the look-and-feel of the underlying host operating system or can use its own style called Fusion.

A number of underlying graphics libraries were compared in Chapter 3, including OpenGL and its variants and DirectX. The shaders used in modern GPUs were also described.

Chapter 4 described the Waveform software itself. The application was built with Qt5 and consists of a Settings Panel and a Render Window based on the model-view-controller paradigm. Five different renderers were developed as separate views, all using the same model and the controller:

1. Qt/native renderer uses Qt widgets to draw the waveform. This served as reference for the other implementations.

2. Qt/OpenGL renderer uses a QGLWidget to provide an OpenGL context. Waveforms are drawn with shader-based OpenGL.

3. Qt/QML/HTML5 Canvas renderer uses QML to arrange the interface components and a HTML5 2D Canvas to draw the waveforms.

4. Qt/WebView/Pixi uses Qt to provide a WebKit window. This window loads a HTML page which uses the Pixi JavaScript library to draw the waveforms using WebGL.

5. Qt/WebView/D3 uses Qt to provide a WebKit window. This window loads a HTML page which uses the D3 JavaScript library to draw the waveforms by dynamically inserting SVG nodes into the browser DOM.

Every renderer contains two Grids and each Grid consists of one or two signal groups. Each signal group displays one or more signals with the same value range on common axes. Two different data sources are provided. The simulator creates a sine wave, a rectangular wave, and a sawtooth signal. WFDB files containimg real patient data can be read and displayed.

At the end of Chapter 4, the results of a performance comparison were presented. The performance was measured for every renderer five times for 20 seconds. Both the Qt/native and the Qt/OpenGL implementations achieved the desired refresh rate of 60 frames per second (fps).

Chapter 5 describes the prototype SmartNIBP blood pressure monitor. Based on the results of this work, the SmartNIBP graphics display is being built with a combination of QML and OpenGL.

Appendix A

User Guide

The Waveform application comprises a Render Window and a Settings Panel. The Render Window is composed of the abstract components shown in Figure A.1. For example, the Qt Native Render Window is shown in Figure A.2. The Settings Panel shown in Figure A.3.

A.1 Renderers

In the Renderer section of the Settings Panel, one of the five available renderers can be selected:

1. Qt/native renderer uses native Qt widgets to draw the waveform.
2. Qt/OpenGL renderer uses a QGLWidget to provide an OpenGL context. Waveforms are drawn with shader-based OpenGL.
3. Qt/QML/HTML5 Canvas renderer uses QML to arrange the interface components and a HTML5 2D Canvas to draw the waveforms.
4. Qt/WebView/Pixi uses Qt to provide a WebKit window. This window loads a HTML page which uses the Pixi JavaScript library to draw the waveforms using WebGL.
5. Qt/WebView/D3 uses Qt to provide a WebKit window. This window loads a HTML page which uses the D3 JavaScript library to draw the waveforms by dynamically inserting SVG nodes into the browser DOM.

A Render Window contains two Grids. The upper grid is called Grid 1 and contains two signal groups. The lower grid is called Grid 2 and contains one signal group. A signal group defines the range of for the Y axis for all signals in that grid. Signal groups can display signals or bands. A signal is a single waveform. A band consists of two waveforms, an upper and a lower bound and the space enclosed in between. The signals displayed are hard-coded in software and can only be changed there.

A.2 Data Sources

Two different data sources are supported. The simulator generates sine waves, rectangular waves, and sawtooth signals. Alternatively, WFDB files containing signals can be read and displayed. Clicking the "Select WFDB file" button opens the file chooser dialogue shown in Figure A.4. The first input field specifies the path to a WFDB database folder. The dropdown menu in the middle specifies which database from the folder is chosen. The lower dropdown menu specifies the exact dataset.

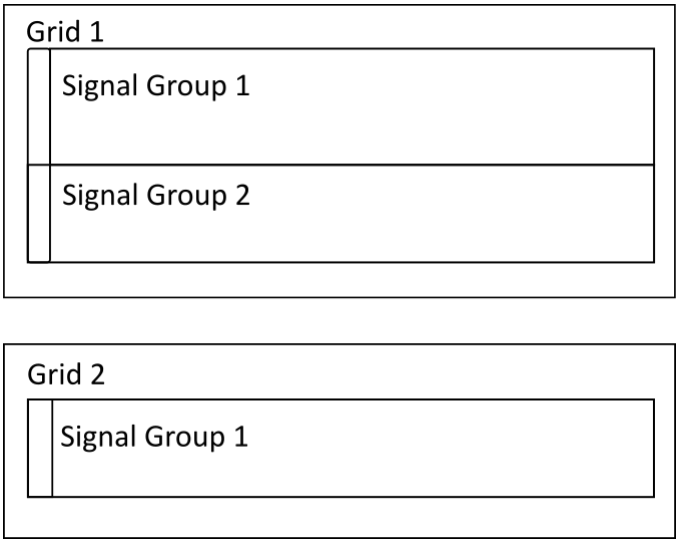
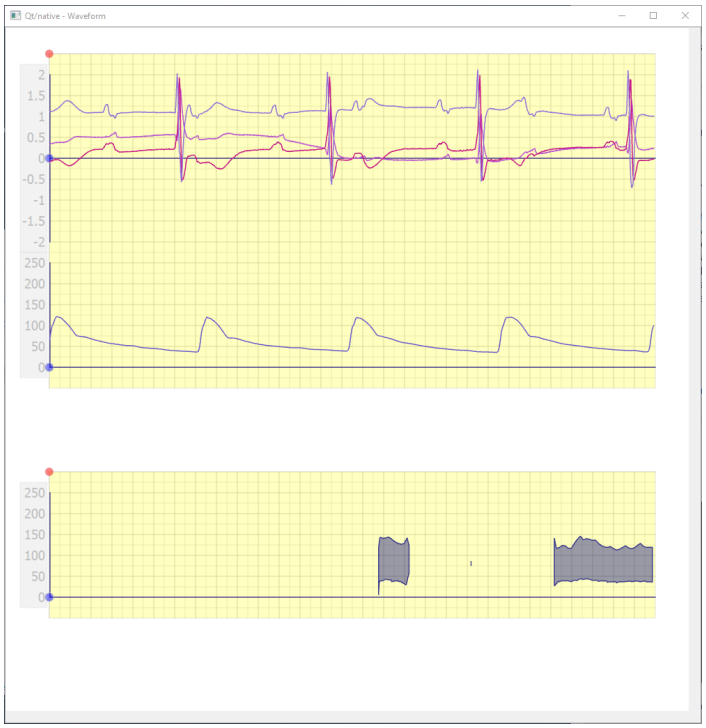


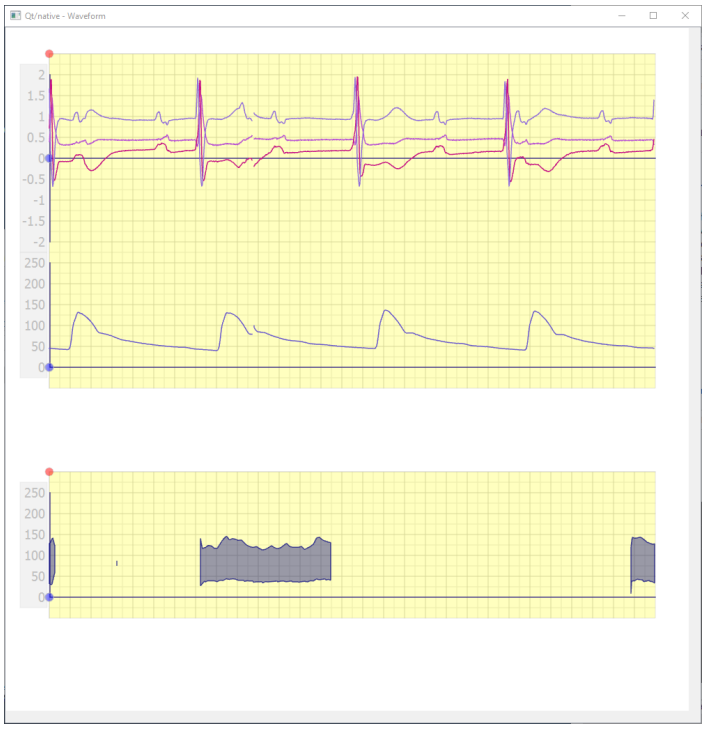
Figure A.1: The components of the Render Window [Diagram created by the author with Inkscape[Inkscape Community, 2017].].

A.3 Grid 1 and Grid 2

Within a view, three different modes to display the data are available. In Scroll mode, new data enters the view from the right and leaves the view to the left, scrolling through. In Overwrite mode, new data is drawn from left to right overwriting any previously drawn data. In Fixed mode, the view displays a signal starting at a specific time code given by RefTime. It is possible to pan the signal using the mouse, the single step button, or by manually inputting a time code.



(a) Scroll.



(b) Overwrite.

Figure A.2: The Qt native Render Window [Screenshots taken by the author.].

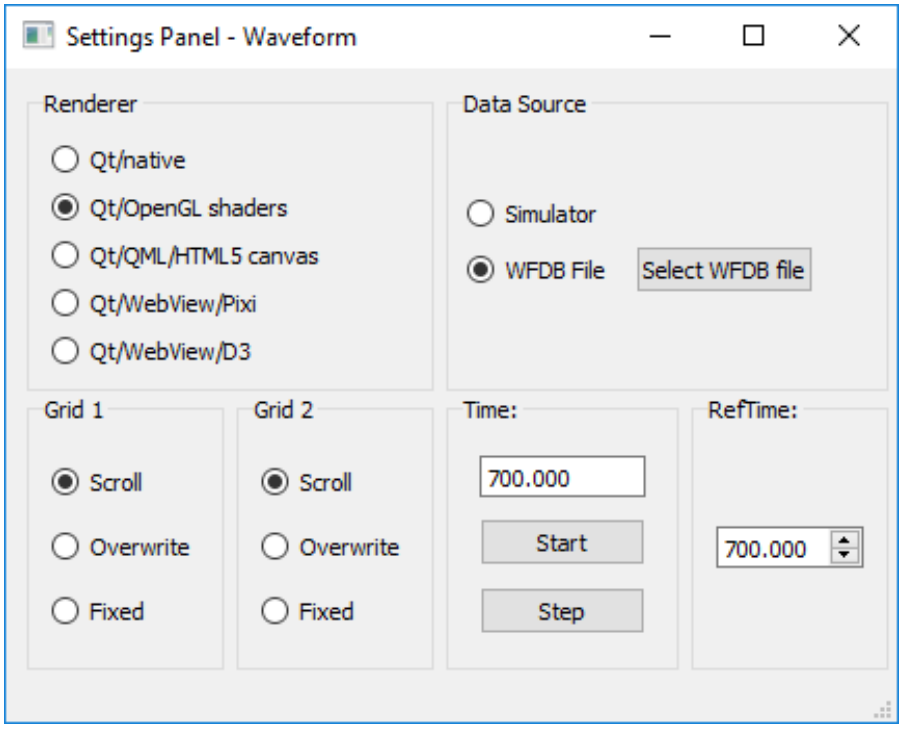


Figure A.3: The Settings Panel of Waveform shows the control options [Screenshot taken by the author.].

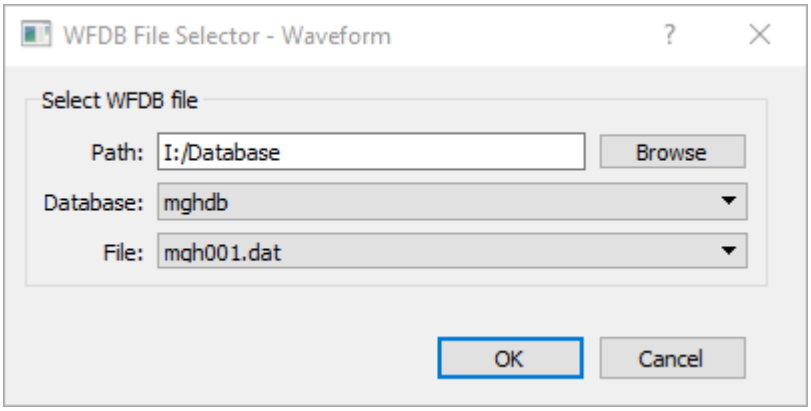


Figure A.4: The file chooser dialogue to select a WFDB file [Screenshot taken by the author.].

Bibliography

Andrews, Keith [2012]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 22 Oct 2012. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page xi).

ANGLE [2017]. *ANGLE*. 25 Apr 2017. https://github.com/google/angle/ (cited on page 12).

Audacity [2017]. *Audacity*. 25 Apr 2017. https://audacityteam.org/ (cited on page 7).

Bocklage-Ryannel, Juergen and Johan Thelin [2015]. *Qt Modules*. 15 Mar 2015. http://qmlbook.org/ch01/index.html#qt-modules (cited on page 3).

Bocklage-Ryannel, Juergen and Johan Thelin [2016]. *QMLBook*. 21 Mar 2016. http://qmlbook.org/ch04 (cited on page 3).

Bostock, Mike [2017]. *Data-Driven Documents*. 25 Apr 2017. https://d3js.org/ (cited on page 31).

Cairo [2017]. *Cairo*. 25 Apr 2017. http://cairographics.org/ (cited on page 6).

CC [2017]. *Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)*. Creative Commons. 25 Apr 2017. http://creativecommons.org/licenses/by-sa/3.0 (cited on pages xi, 6).

Eclipse Foundation [2017a]. *Eclipse Public License - v 1.0*. 25 Apr 2017. https://eclipse.org/legal/epl-v10.html (cited on page 8).

Eclipse Foundation [2017b]. *SWT: The Standard Widget Toolkit*. 25 Apr 2017. https://eclipse.org/swt (cited on page 8).

FileZilla [2017]. *Compiling FileZilla 3 under Windows*. 25 Apr 2017. https://wiki.filezilla-project.org/Compiling_FileZilla_3_under_Windows (cited on page 7).

GNOME [2017]. *GTK+ Look And Feel*. 25 Apr 2017. https://gtk.org/features.php (cited on page 6).

GNU [2017a]. *GNU General Public License v3*. 25 Apr 2017. https://gnu.org/copyleft/gpl.html (cited on page 3).

GNU [2017b]. *GNU Lesser General Public License, version 2.1*. 25 Apr 2017. https://gnu.org/licenses/lgpl-2.1.html (cited on page 3).

Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, eorge B. Moody amd Chung-Kang Peng, and H. Eugene Stanley [2000]. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals". *Circulation* 101.23 [13 Jun 2000], e215–e220. doi:10.1161/01.CIR.101.23.e215. http://circ.ahajournals.org/content/101/23/e215.full (cited on page xi).

Google [2017a]. *Android*. 25 Apr 2017. https://android.com/ (cited on page 10).

Google [2017b]. *Design Principles*. 25 Apr 2017. https://developer.android.com/design/get-started/principles.html (cited on page 10).

Google [2017c]. *Material Design*. 25 Apr 2017. https://developer.android.com/design/material (cited on page 10).

Groves, Mat [2017]. *PixiJS*. Goodboy Digital. 25 Apr 2017. http://pixijs.com/ (cited on page 27).

GTK+ [2017a]. *GTK+ History*. 25 Apr 2017. https://people.redhat.com/mclasen/Usenix04/notes/x29.html (cited on page 5).

GTK+ [2017b]. *Gtk+ Language Bindings*. 25 Apr 2017. https://gtk.org/language-bindings.php (cited on page 6).

GTK+ [2017c]. *THe GTK+ Project*. 25 Apr 2017. https://gtk.org/ (cited on page 5).

Inkscape Community [2017]. *Inkscape*. 25 Apr 2017. https://inkscape.org/ (cited on pages 16, 44).

Khronos [2017a]. *Core And Compatibility in Contexts*. Khronos Group. 25 Apr 2017. https://opengl.org/wiki/Core_And_Compatibility_in_Contexts (cited on page 11).

Khronos [2017b]. *EGL - Native Platform Interface*. Khronos Group. 25 Apr 2017. https://khronos.org/egl (cited on page 10).

Khronos [2017c]. *OpenGL*. Khronos Group. 25 Apr 2017. https://opengl.org/ (cited on page 11).

Khronos [2017d]. *OpenGL 4.5*. Khronos Group. 25 Apr 2017. https://khronos.org/opengl/ (cited on page 12).

Khronos [2017e]. *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*. Khronos Group. 25 Apr 2017. https://khronos.org/opengles/ (cited on page 12).

Khronos [2017f]. *OpenGL Overview*. Khronos Group. 25 Apr 2017. http://opengl.org/about/ (cited on page 11).

Khronos [2017g]. *WebGL - OpenGL ES 2.0 for the Web*. Khronos Group. 25 Apr 2017. https://khronos.org/webgl/ (cited on page 12).

MacDonald, Matthew [2009]. *Pro WPF In C# 2008 Windows Presentation Foundation With NET 4.5*. 2nd Edition. 15 Jun 2009. ISBN 1430243651 (cited on pages 8, 10).

McCann, William Jon [2017]. *Dia*. 25 Apr 2017. https://wiki.gnome.org/Apps/Dia/ (cited on page 36).

Mesa [2017a]. *License / Copyright Information*. 25 Apr 2017. https://mesa3d.org/license.html (cited on page 12).

Mesa [2017b]. *The Mesa 3D Graphics Library*. 25 Apr 2017. https://mesa3d.org/intro.html (cited on page 12).

Microsoft [2014]. *In-Depth Guidance for Windows Store Apps*. 20 Jun 2014. http://msdn.microsoft.com/en-us/library/windows/apps/br211375.aspx (cited on page 13).

Microsoft [2017a]. *DirectX*. 25 Apr 2017. https://msdn.microsoft.com/library/windows/apps/hh452744 (cited on page 12).

Microsoft [2017b]. *MFC Desktop Applications*. 25 Apr 2017. https://msdn.microsoft.com/en-us/en-en/library/d06h2x6e.aspx (cited on page 8).

Microsoft [2017c]. *.NET*. 25 Apr 2017. https://msdn.microsoft.com/en-us/en-en/vstudio/aa496123 (cited on page 8).

Microsoft [2017d]. *Windows API Index*. 25 Apr 2017. https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx (cited on page 8).

Microsoft [2017e]. *Windows Presentation Foundation*. 25 Apr 2017. https://msdn.microsoft.com/en-us/library/ms754130(v=vs.100).aspx (cited on page 8).

Oracle [2017a]. *About the JFC and Swing*. 25 Apr 2017. https://docs.oracle.com/javase/tutorial/uiswing/start/about.html (cited on page 7).

Oracle [2017b]. *Is JavaFX included in Java SE?* 25 Apr 2017. http://oracle.com/technetwork/java/javafx/overview/faq-1446554.html#5 (cited on page 8).

Oracle [2017c]. *JavaFX - The Rich Client Platform*. 25 Apr 2017. http://oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html (cited on page 8).

Oracle [2017d]. *NetBeans IDE - The Smarter and Faster Way to Code*. 25 Apr 2017. https://netbeans.org/features/index.html (cited on page 8).

Oracle [2017e]. *Nimbus Look and Feel*. 25 Apr 2017. https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/nimbus.html (cited on page 7).

Oracle [2017f]. *Oracle Binary Code License Agreement*. 25 Apr 2017. http://oracle.com/technetwork/java/javase/terms/license/index.html (cited on page 8).

Oracle [2017g]. *What is Java?* 25 Apr 2017. https://java.com/en/about/whatis_java.jsp (cited on page 7).

PhysioNet [2017a]. *MGH/MF Waveform Database Patient Guide*. 25 Apr 2017. https://physionet.org/physiobank/database/mghdb/patient-guide.shtml (cited on page 16).

PhysioNet [2017b]. *PhysioNet*. PhysioNet Research Resource for Complex Physiologic Signals. 25 Apr 2017. https://physionet.org/ (cited on page 15).

PhysioNet [2017c]. *The MGH/MF Waveform Database*. 25 Apr 2017. https://physionet.org/physiobank/database/mghdb (cited on page 15).

PhysioNet [2017d]. *WFDB library*. 25 Apr 2017. https://physionet.org/physiotools/wfdb.shtml#library (cited on page 16).

QtC [2017a]. *Graphics View Framework*. 25 Apr 2017. https://doc.qt.io/qt-5/graphicsview.html (cited on page 16).

QtC [2017b]. *Programming Language Support and Language Bindings*. 25 Apr 2017. https://wiki.qt.io/Category:LanguageBindings/ (cited on page 3).

QtC [2017c]. *QT*. 25 Apr 2017. https://www.qt.io/ (cited on page 3).

QtC [2017d]. *Using The Meta-Object Compiler (moc)*. 25 Apr 2017. http://doc.qt.io/qt-4.8/moc.html (cited on page 5).

Qtc [2017a]. *Qt Charts*. 25 Apr 2017. https://doc.qt.io/qt-5/qtcharts-index.html (cited on page 39).

Qtc [2017b]. *Removed Modules*. 25 Apr 2017. https://wiki.qt.io/New_Features_in_Qt_5.6#Removed_Modules (cited on page 39).

SGI [2017]. *OpenGL license*. 25 Apr 2017. https://sgi.com/tech/opengl/?/license.html (cited on page 11).

Shreiner, Dave, Graham Sellers, John M. Kessenich, and Bill Licea-Kane [2013]. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. 8th Edition. Addison Wesley, 30 Mar 2013. ISBN 0321773039 (cited on page 11).

Sven [2017]. *Cairo's Drawing Model*. Wikimedia Commons User Sven. 4 Nov 2017. http://commons.wikimedia.org/wiki/File:Cairo%27s_drawing_model.svg (cited on pages xi, 6).

Unity [2017]. *Unity Game Engine*. Unity Technologies. 25 Apr 2017. https://unity3d.com/ (cited on page 10).

W3Schools [2017]. *HTML5 Canvas*. 25 Apr 2017. https://w3schools.com/html/html5_canvas.asp (cited on page 23).

wxWidgets [2017a]. *wxWidgets*. 25 Apr 2017. https://wxwidgets.org/ (cited on page 7).

wxWidgets [2017b]. *WxWidgets License*. 25 Apr 2017. https://wxwidgets.org/about/licence/ (cited on page 7).

wxWidgets [2017c]. *WxWidgets Name Change*. 25 Apr 2017. https://wxwidgets.org/about/name-change/ (cited on page 7).

wxWidgets [2017d]. *WxWidgets Overview*. 25 Apr 2017. https://wxwidgets.org/about/ (cited on page 7).