Bachelor's thesis

Information and Communications Technology

2019

Atte Laakso

QT FOR WEBASSEMBLY

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2019 | 28 pages, 2 pages in appendices

Atte Laakso

QT FOR WEBASSEMBLY

WebAssembly is a new technology that brings programming languages such as C and C++ to the web. It aims to provide an alternative to JavaScript especially for tasks that require performance, for example, encryption.

Qt is a C++ framework for creating graphical user interface applications. Qt for WebAssembly allows applications written with the Qt framework to be compiled to WebAssembly.

The aim of this thesis was to provide an overview of Qt for WebAssembly. The thesis discusses what is required to start compiling Qt applications into WebAssembly and what its limitations are. Another target was to create a WebAssembly application, that simulates an In-Vehicle-Infotainment system written with the Qt framework. This application could be used to demonstrate the In-Vehicle-Infotainment system without the actual physical system.

The performance of WebAssembly was examined through existing research on the topic. Its performance was found to be excellent especially when compared to JavaScript. Crucial limitations in Qt for WebAssembly were found out during the development of the simulation application and because of these limitations the development was halted. The main reason was the lack of support for several Qt modules that were used in the In-Vehicle-Infotainment system.

KEYWORDS:

C++, WebAssembly, Qt

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2019 | 28 sivua, 2 liitesivua

Atte Laakso

QT FOR WEBASSEMBLY

WebAssembly on uusi teknologia joka mahdollistaa eri ohjelmointikielien kuten C:n ja C++:n ajamisen verkossa. WebAssembly pyrkii tuomaan vaihtoehdon JavaScriptille etenkin suorituskykyä vaativiin tehtäviin, kuten salausalgoritmien ajoon.

Qt on C++ -ohjelmointikehys graafisten käyttöliittymien tekemiseen. Qt for WebAssembly mahdollistaa Qt-sovellusten kääntämisen WebAssemblyksi.

Työn tavoitteena oli luoda yleiskatsaus Qt for WebAssemblyyn. Tutkimuskysymyksinä olivat, mitä vaaditaan Qt sovelluksen WebAssemblyksi kääntämiseen sekä millaisia rajoitteita ja ongelmia tällä on. Lisäksi tavoitteena oli tehdä WebAssembly-sovellus, joka simuloi Qt:lla tehdyn auton mediakeskusjärjestelmän käyttöliittymää. Tätä sovellusta voisi käyttää mediakeskuksen demonstroimiseen ilman varsinaisen järjestelmän fyysistä laitteistoa.

WebAssemblyn suorituskykyä tutkittiin jo tehtyjen tutkimusten perusteella. WebAssemblyn suorituskyky todettiin hyväksi etenkin JavaScriptiin verrattuna. Mediakeskuksen käyttojärjestelmän simulaatio -sovelluksen kehitys keskeytettiin kehityksen aikana ilmenneiden Qt for WebAssemblyn rajoitteiden vuoksi. Suurin rajoitus oli Qt for WebAssemblyn tukemat Qt-moduulit. Useita mediakeskuksessa tarvittavia moduuleita ei ollut mahdollista vielä käyttää.

ASIASANAT:

C++, WebAssembly, Qt

CONTENTS

APPENDICES

FIGURES

1 INTRODUCTION

WebAssembly is a binary format that runs in the browser. It is a compilation target for software written in other programming languages such as C, C++ or Rust. A major future goal for WebAssembly is to also provide runtimes outside of the browser.

Qt is a C++ based framework used for cross platform development across desktop, mobile and embedded platforms. Qt for WebAssembly makes it possible to run Qt applications on another major platform — the Web.

The goal of this thesis is to provide an overview of WebAssembly and Qt for WebAssembly, what these technologies provide, what they can be used for and what limitations they have. Comparing the performance of WebAssembly versus JavaScript and native code is a major part of this thesis.

The structure of this thesis is as follows: First, an introduction to the Qt framework is given. Then follows an overview of WebAssembly and Qt for WebAssembly. Finally, the thesis describes the practical work of the thesis where an existing Qt based application is compiled into WebAssembly.

2 QT

Qt is a C++ framework for cross-platform GUI application programming [1]. Applications written with Qt will run on all supported platforms and require only small or no changes to the source code. Qt-supported platforms include Linux, Windows, macOS, Android, iOS and several embedded platforms, for example, Embedded Linux and QNX [2].

Qt offers two options for building graphical user interfaces. The first option is QML, which is a declarative language with good support for touch screens and animations. The second option is Qt Widgets which is more suited for traditional desktop user interfaces. A Qt application may use both QML and Qt Widgets in its user interface. [3]

2.1 QML

Qt Modeling Language (QML) is a declarative language with support for imperative JavaScript expressions. Its syntax resembles JSON. A QML user interface may be connected to any C++ back-end. [4]

QML is built around the behaviour of UI components and how the components are connected. Visual effects are an important part of QML UIs and may be supplemented with particle and shader effects. [5]

2.2 Qt Widgets

Qt Widgets are C++-based UI elements that integrate well with Windows, Linux and macOS providing a native look and experience on these platforms. Qt Widgets do not support touch screens or highly animated user interfaces as well as QML-based user interfaces do and they are more suited for traditional desktop style user interfaces. [3]

2.3 Signals, Slots, qmake and the Meta-Object System

Communication between objects in a GUI is usually needed. For example, clicking a button needs to notify another object that the button was clicked and that object shall respond somehow. Other frameworks often handle this with callback functions. Qt uses a mechanism called **signals and slots** to achieve this instead. [6]

A signal may be emitted from any Qt Object and it may be connected to any slot of any Qt Object. Slots are functions that are called in response to the signals they are connected to. Signals and slots are strongly encapsulated. A signal does not know which slots it is connected to nor do slots know which signals they are connected to. Figure 1 shows an abstract example of how signals and slots might be connected between objects. [6]

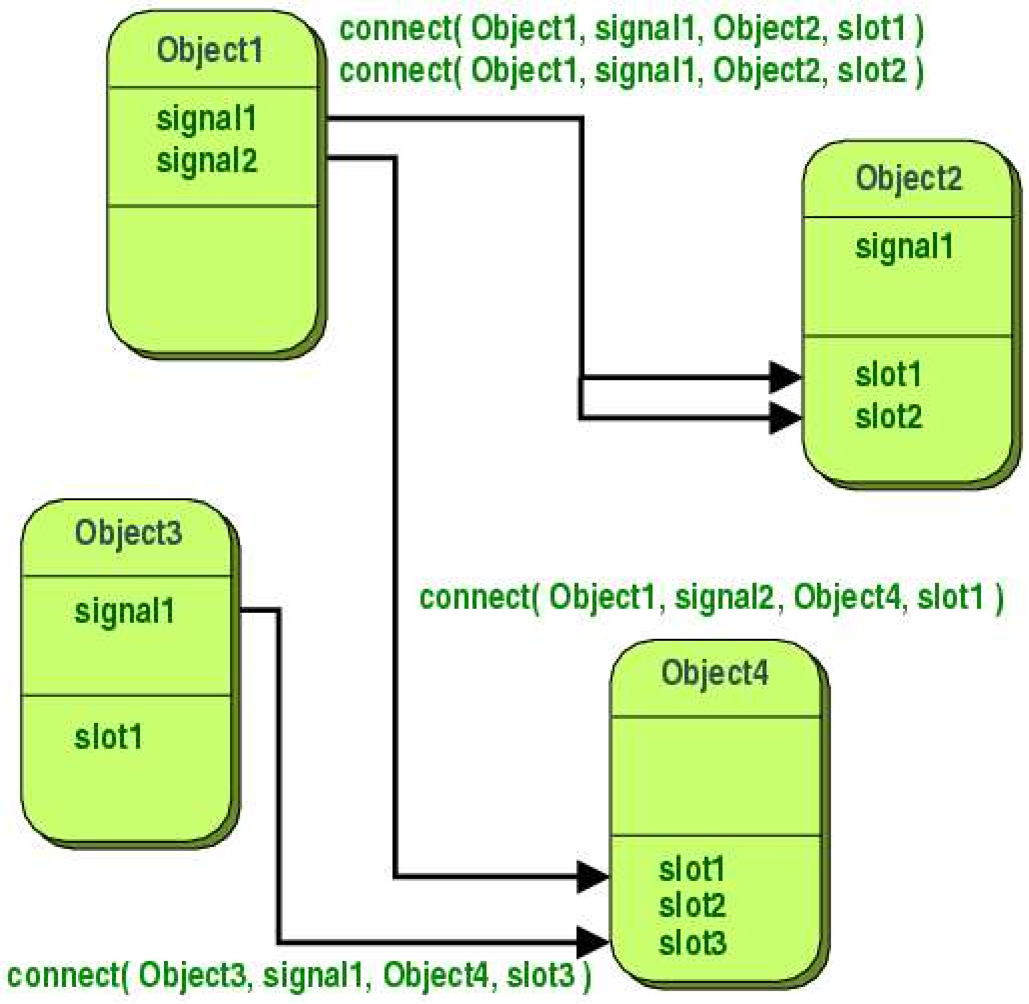

Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Figure 1. Abstract Connections [6].

The signal and slot mechanism is provided by Qt’s meta-object system. It is composed of the QObject class, the Q_OBJECT macro and the meta-object compiler (moc). QObject is a base class for all Qt objects. The moc produces C++ source files necessary to use the signal and slot mechanism for any QObject classes that contain the

Q_OBJECT macro. Qt provides a tool called qmake for creating makefiles. Qmake and moc are well integrated and when using qmake for creating the makefiles it will automatically create rules for calling the moc when necessary. [6, 7, 8]

3 WEBASSEMBLY

For a long time JavaScript has been the only natively supported programming language on the web platform. JavaScript is a flexible, dynamically typed, and well readable language but it is not well suited for tasks that require high performance such as video editing, CAD, or encryption. WebAssembly aims to load, parse, and execute more quickly than JavaScript. It does not mean to replace JavaScript but rather work side by side with it, handling the performance-intensive tasks. [9]

WebAssembly is a low-level language designed to run on the web, but without web-specific assumptions or features, thus making it possible to be employed to other environments as well. WebAssembly is not meant to be manually written but compiled from other languages such as C or C++. [10, 11]

WebAssembly differs from actual assembly languages in that it is not a machine language for a physical machine, but for a conceptual machine [12]. WebAssembly is not tied to the browser or any single operating system, the same code can run on multiple systems [13].

3.1 Beginning of WebAssembly

Emscripten is a tool for transpiling C/C++ or any code that can be translated to LLVM bytecode into JavaScript [14]. The JavaScript generated by Emscripten is slow but Mozilla engineers found a way to make it run faster than plain JavaScript and created a subset of JavaScript called asm.js that runs faster than plain JavaScript. Eventhough asm.js is more performant than plain JavaScript, it is still JavaScript with its limitations – the need for a new language was there which inspired the creation of WebAssembly [15].

The initial design goals for WebAssembly were set .The goals were to create a language-agnostic (not just for C and C++), fast running, compact compile target with the ability to support pointers or pointerlike features. The first release of WebAssembly was launched in 2017. [15]

3.2 Future Goals for WebAssembly

Making use of modern hardware capabilities. Support parallelism with multithreading and Single Instruction Multiple Data (SIMD), where a single instruction is performed on multiple data points. Memory is currently limited to 4 gigabytes because of 32-bit addressing and 64-bit addresses. These are currently under development. [15]

Loadtime improvements with streaming compilation and caching. With streaming compilation the WebAssembly compilation starts during the download. This is supported on Firefox and Chrome. Caching the modules to prevent compiling the module every time the page is loaded is under development. [15]

Interoperability with JavaScript. When WebAssembly is only used for some small tasks the calls between JavaScript and WebAssembly should be fast. On Firefox some JavaScript to WebAssembly calls are faster than JavaScript to JavaScript calls. Returning the data fast from WebAssembly to JavaScript is a goal but no results are available yet. [15, 16]

Integrating WebAssembly modules into existing tools such as the node package manager is a goal and proposals have been made but no further information is available. [15]

Backwards compability. Something needs to be supplied for users with older browsers that have no support for WebAssembly. A tool to convert WebAssembly into JavaScript exists. [15] However it is not capable of converting all WebAssembly code [17].

Handling exceptions is possible but the implementation is slow. Currently only at research and development level. [15]

Debugging. It is possible to step through the WebAssembly code with browser developer tools but stepping through the actual source code is not possible. [15]

WebAssembly aims to support functional programming languages that allow tail calls, but tail calls are not yet supported [15].

As WebAssembly is a language for a conceptual machine and not a physical machine, running the same WebAssembly code on different hardware architectures is possible. The WebAssembly System Interface aims to bring a standardized system interface to

WebAssembly allowing access to system resources and run WebAssembly outside of the browser. [13]

3.3 Performance of WebAssembly

Surma [18] tests the effect of replacing a single JavaScript function with WebAssembly. The tested function rotates an image by 90, 180 or 270 degrees. The rotation is achieved by copying each pixel of the image to a new position. Figure 2 shows the difference between the original JavaScript code and WebAssembly compiled from C, AssemblyScript and Rust. On browsers 1 and 2 the difference was not significant, but on browser 3 WebAssembly decreased the runtime on average by 85 % and on browser 4 the average decrease was 94 %.

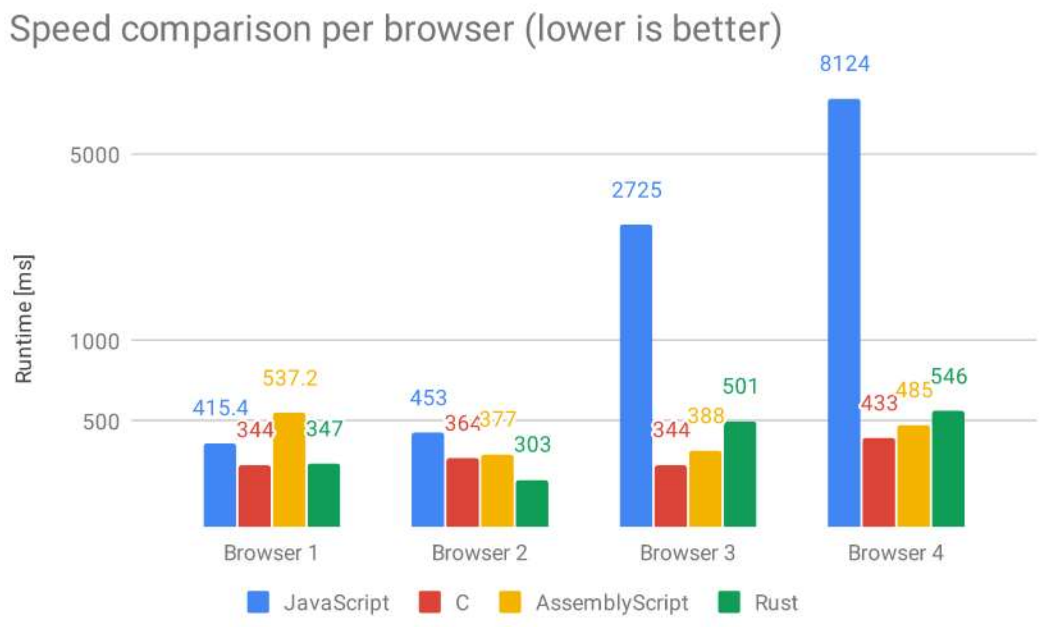


Figure 2. Performance comparison [18].

The PSPDFKit WebAssembly benchmark [19] tests the time taken to perform various actions on a PDF document and the time taken to load the module. The results were compared to a JavaScript version of the PSPDFKit. Figure 3 shows the results. On Firefox the WebAssembly version performed considerably better than the JavaScript version, but on Chrome there was no substantial difference and on Edge the

WebAssembly versions performs considerably worse than the JavaScript version.

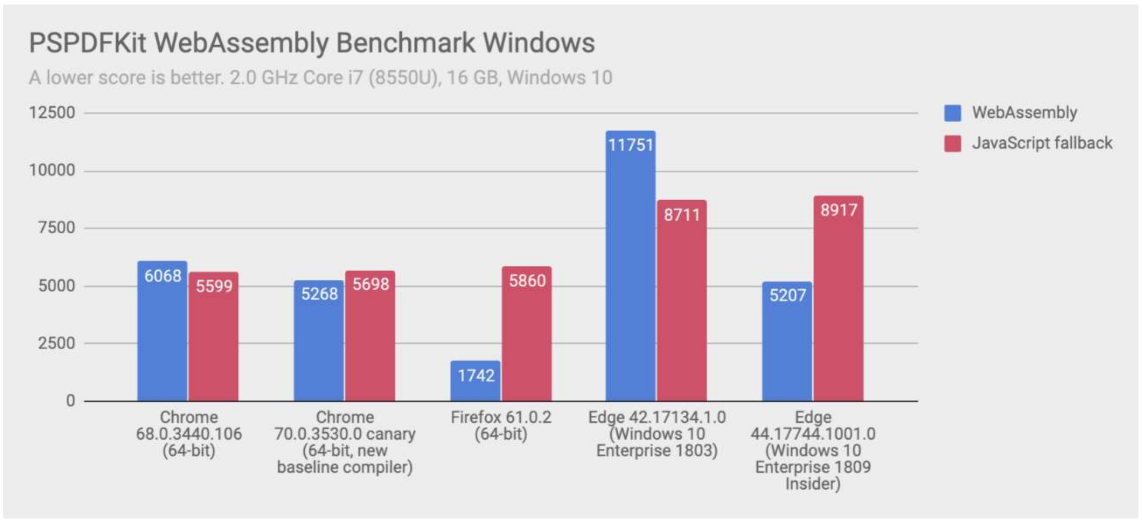


Figure 3. The PSPDFKit Benchmark on Windows [19].

The tests were conducted in 2018 and the benchmark suite is publicly available. Figure 4 shows the benchmark scores ran by the author at the time of writing this thesis. The WebAssembly version now performs considerably better than the JavaScript version on Firefox and Chrome, but on Edge the JavaScript version is still better performing.

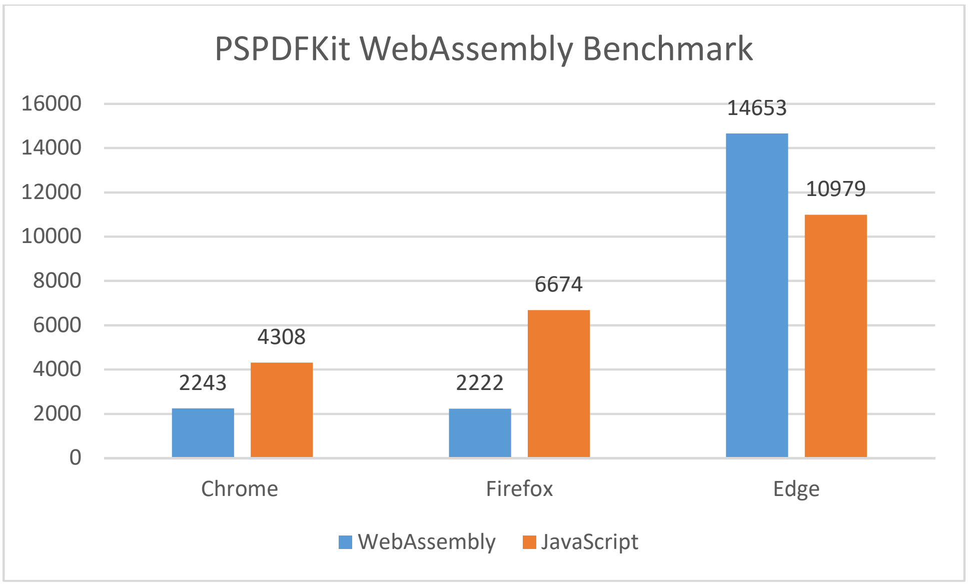
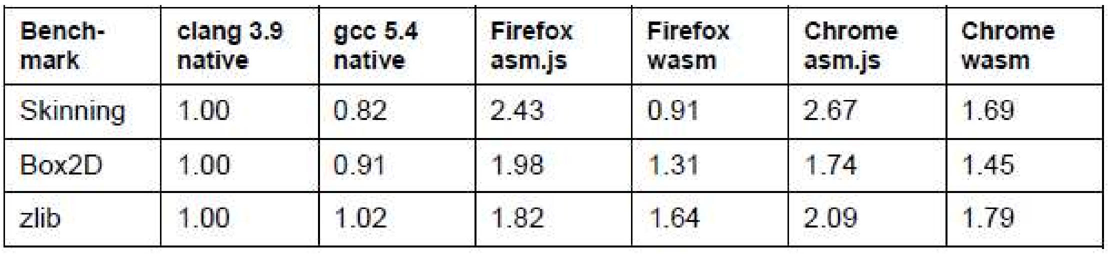


Figure 4. Results from the PSPDFKit Benchmark ran by the author at the time of writing this thesis.

In Figure 5 Zakai [20] shows 3 benchmarks where WebAssembly and asm.js code were compared to native code. Skinning is a vertex skinning algorithm, Box2D is a physics engine and zlib is a data compression software. A Clang native compiler is used as the point of reference. On average, WebAssembly ran 1.46 times slower than native code.



Bench-mark	clang 3.9 native	gcc 5.4 native	Firefox asm.js	Firefox wasm	Chrome asm.js	Chrome wasm
Skinning	1.00	0.82	2.43	0.91	2.67	1.69
Box2D	1.00	0.91	1.98	1.31	1.74	1.45
zlib	1.00	1.02	1.82	1.64	2.09	1.79

Figure 5. Performance measurements of asm.js and WebAssembly compared to native code [20: p.12].

Haas et al. [21] ran the PolyBenchC benchmark suite on Chrome and Firefox. The benchmark suite tests the execution speed of various numerical computations such as matrix multiplication, Cholesky decomposition and covariance computation [22]. The benchmark results are shown in Figure 6. Four of the 24 benchmarks ran quicker than native code and 7 of the benchmarks ran within 110 % of native code execution time and only 1 benchmark took more than twice the time of the native code execution speed.

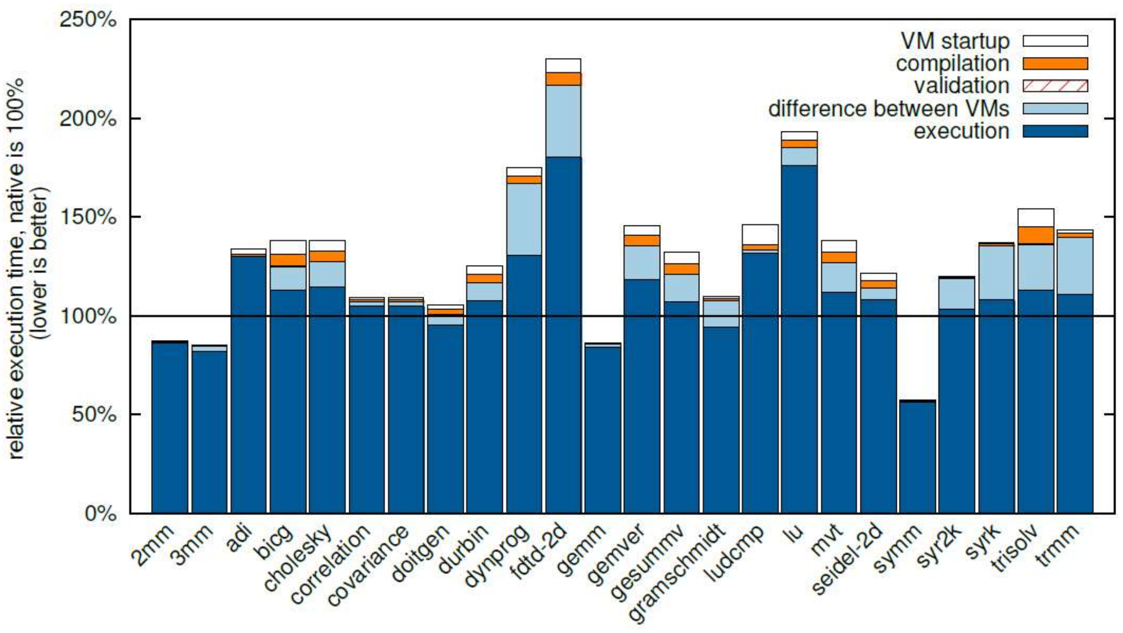


Figure 6. Relative execution time of the PolyBenchC benchmarks on WebAssembly normalized to native code [21: p.197].

Jangda et al. [23] criticized Haas et al. [21] for the use of the PolyBenchC benchmark suite to test WebAssembly saying that the small scientific computing functions are ’not necessarily representative of applications that target the browser.’ They used a modified version of the SPEC CPU Benchmark Suite which consists of larger tests than the PolyBenchC benchmark suite. The SPEC CPU Benchmark Suite contains for example video compression and XML to HTML conversion benchmarks. Figure 7 shows the results of this benchmark. The results show a larger performance difference than the PolyBenchC benchmark (Figure 6). On average the execution time does not exceed 200 % of the native code execution speed.

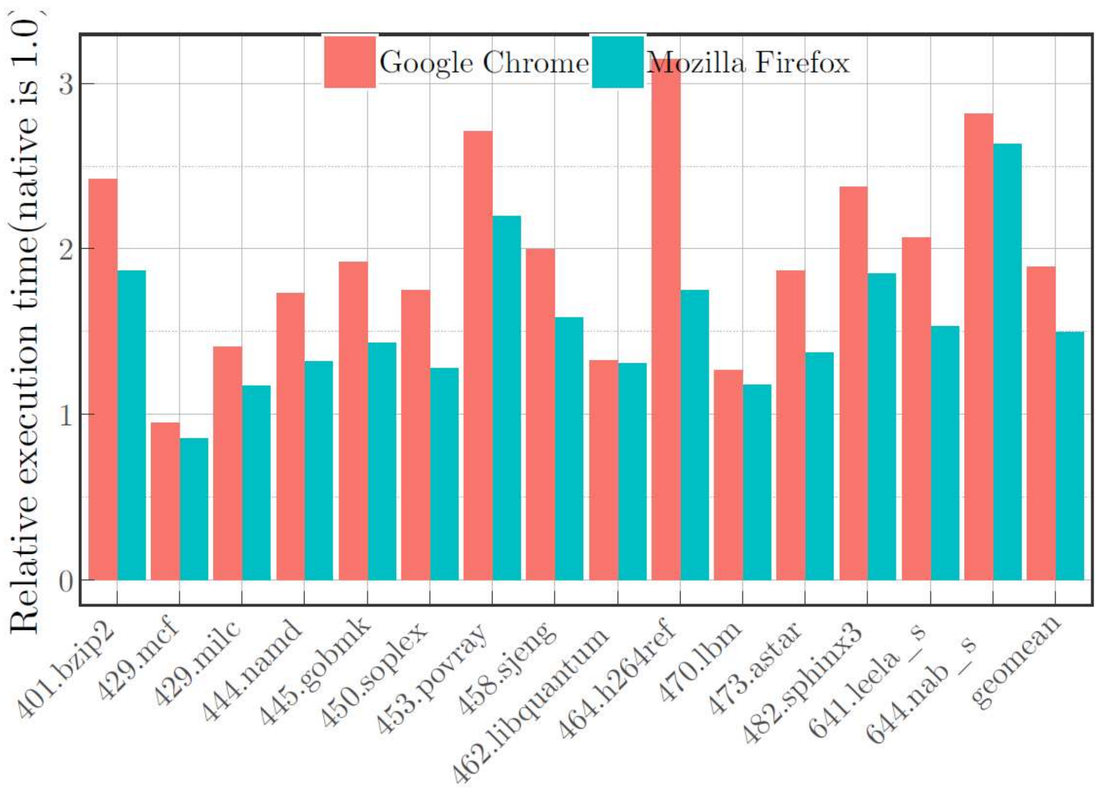


Figure 7. SPEC CPU Benchmark results [23 p.3].

4 QT FOR WEBASSEMBLY

The Web is is another platform target for Qt to be truly cross platform and compiling to WebAssembly makes it possible to run Qt applications on the Web. [24]

Current limitations of Qt for WebAssembly include no threads, no multimedia, no clipboard support, no virtual keyboard on mobile devices and no accessibility support [24].

The WebAssembly sandbox causes restrictions on local file access and networking. The HTML API provides file download and upload functions that can be used for local file access. QFile API works, but supplying the file needs to be done via the HTML API. [24]

HTTP access to resources is possible to same origin or a Cross-Origin Resource Sharing (CORS) enabled host. WebSocket connections are possible to any host and QAbstractSocket API can be used via Websockify. [24]

The Emscripten toolchain is used to compile Qt applications into WebAssembly. Emscripten creates a basic HTML page, a WebAssembly module, a JavaScript runtime and a JavaScript loader to load the application into the HTML page. The application lives in a single HTML canvas element. Emscripten also provides an API for interoperating between C++ and JavaScript. [24]

Qt for WebAssembly is in technology preview in Qt version 5.12. Release and pre-built binary packages are planned for the Qt 5.13 release. Multithreading support is planned for Qt 5.13 [24]. At the time of writing Qt 5.13 was in beta and not yet released.

4.1 Performance

Figure 8 shows the execution time of an image mirroring function running in a Qt QML based application. The function is similar to Surmas [18], it iterates over each pixel in the image and places them into a new image.

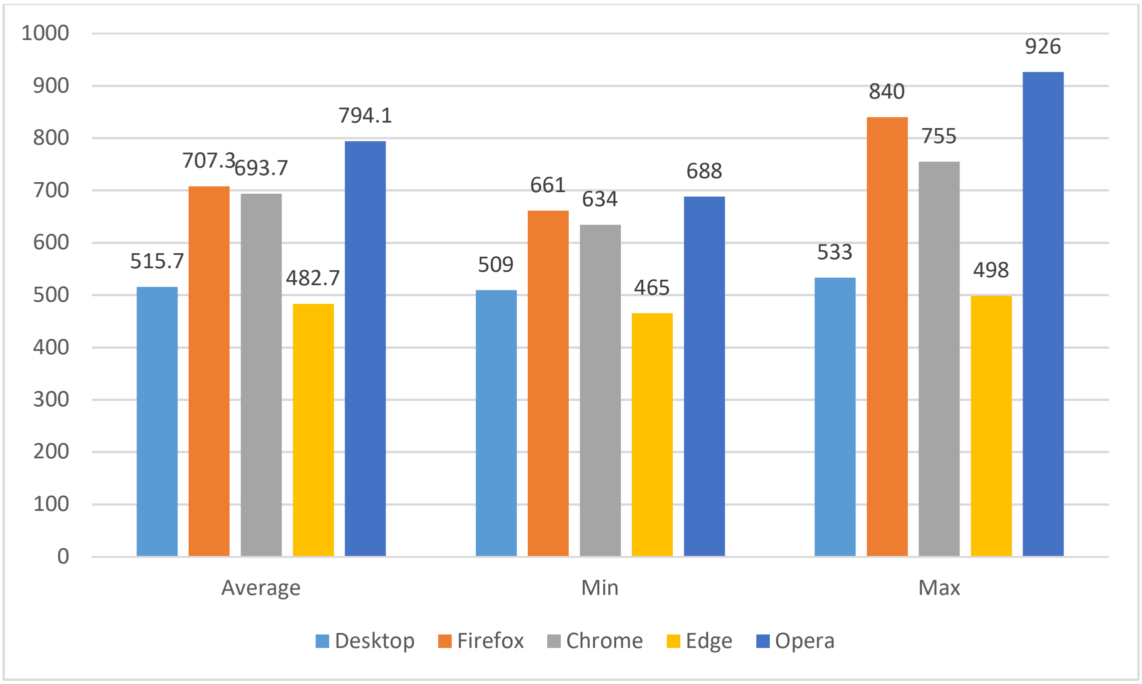
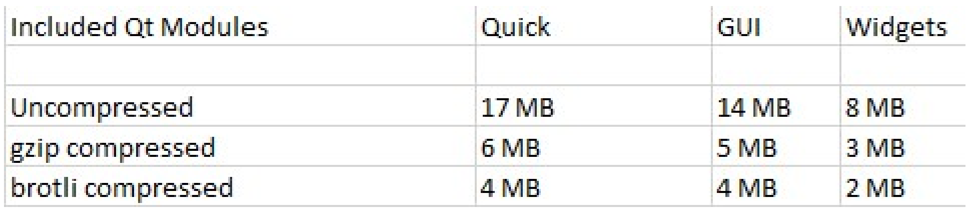


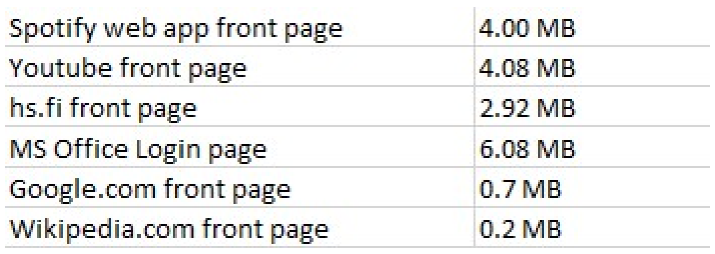
Figure 8. Execution time of mirroring a 3872 x 2592 image pixel by pixel in milliseconds.

The WebAssembly modules produced by Qt are large. They do however compress well. At the time of writing this thesis receiving gzip compressed content is supported by effectively all used browsers and brotli compressed content is supported by 90 % of used browsers according to caniuse.com [25, 26]. Figure 9 shows sizes of various “Hello World” -Qt applications when using different Qt modules. Figure 10 shows the amount of data downloaded when loading the initial front page of various websites for comparison.



Included Qt Modules	Quick	GUI	Widgets
Uncompressed	17 MB	14 MB	8 MB
gzip compressed	6 MB	5 MB	3 MB
brotli compressed	4 MB	4 MB	2 MB

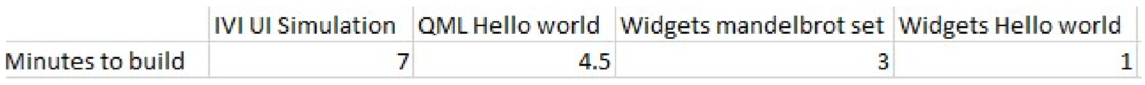
Figure 9. WebAssembly module sizes of various Qt applications.



Spotify web app front page	4.00 MB
Youtube front page	4.08 MB
hs.fi front page	2.92 MB
MS Office Login page	6.08 MB
Google.com front page	0.7 MB
Wikipedia.com front page	0.2 MB

Figure 10. Amount of data downloaded when visiting the front page of various websites.

The time to build a Qt for WebAssembly application can get very long depending on the size of the application. Figure 11 shows some build times in minutes. IVI UI Simulation refers to the application developed in Chapter 5. Widgets mandelbrot set refers to an example application provided by Qt that calculates and draws the Mandelbrot set.



	IVI UI Simulation	QML Hello world	Widgets mandelbrot set	Widgets Hello world
Minutes to build	7	4.5	3	1

Figure 11. Time taken to build various Qt applications into WebAssembly.

5 DEVELOPMENT

5.1 Overview of the Application

An automotive In-Vehicle-Infotainment system that consists of 2 applications, the main application and a mobile application. The main application contains the center console and an instrument cluster. The center console contains views for climate controls (air conditioning and heating), media, diagnostics, navigation and phone (calling). The instrument cluster contains various error messages (e.g. low battery), an odometer, a speedometer and a map. The mobile application contains the same views as the main application except the phone view. The mobile views are also a bit simplified to better fit a mobile devices screen.

The mobile application is connected to the main application through a Bluetooth connection. Bluetooth is also used to provide media and calls from the phone to the main application.

Producing a WebAssembly application that simulates the user interface of the IVI system is one goal of this thesis. This application could be used for demonstrating and marketing the IVI system without access to the physical equipment necessary for running the system. The user will only need a browser and a link to a website running the application. As the application is just a simulation of the real system its complexity may be reduced, for example a Bluetooth connection is not necessary for simulating the UI.

5.2 Building Qt for WebAssembly

Qt version 5.12 is used to compile the application into WebAssembly and this version does not come with pre-built Qt for WebAssembly. Development will be done on a Windows 10 machine and Qt 5.12 for WebAssembly does not support Windows 10, but Windows Subsystem for Linux is supported.

At the time of writing, the existing documentation for building Qt for WebAssembly was not very detailed. The build process was carried out in a trial and error style. Sometimes the build process finished unsuccessfully due to missing required packages, sometimes the build process finished seemingly successfully but did not work and searching through build logs indicated that at least not all modules were built. During the build process modules not supported by Qt for WebAssembly were identified. These modules include the following modules used by the IVI system: Bluetooth, Multimedia, 3d, and Location.

How to successfully build Qt 5.12 for WebAssembly on a machine running Windows 10 is documented in Appendix 1.

5.3 Identifying possible problems

The following problems were identified before starting the development of the application:

1. The IVI system consists of 2 separate applications. The application will need to combine these 2 applications into 1 application.
2. Applications have in total 3 top-level Window components. The main view, the instrument cluster and the mobile view are all separate Window components and application will use only 1 screen.
3. The main application runs on a Linux-based system and uses Linux system calls that are not available in WebAssembly.
4. How to handle Bluetooth connection between the applications as Bluetooth module is not supported by Qt 5.12 for WebAssembly.
5. A 3d model of the vehicle is shown in the application. 3d module is not supported by Qt 5.12 for WebAssembly.
6. How to handle playing music in the application as the Multimedia module is not supported by Qt 5.12 for WebAssembly.
7. How to handle navigation feature in the application as the Positioning and Location modules are not supported by Qt 5.12 for WebAssembly.
8. How to mock a phone connected to the application.
    a. Mocking the phone plugin. No real phone will be connected to the application.
    b. Mocking a media player. No real phone will be connected to the application.
9. Responsiveness of the IVI UI. It is made to run on a specific device on a specific screen and will probably not scale at all. Mobile application is responsive and should not require major changes.

5.4 Implementing solutions

As the application is just a simulation of the user interface of the In-Vehicle-Infotainment system some functionality may just be left out. The main and mobile applications will be bundled into one application making the Bluetooth connection unnecessary. Instead of using Bluetooth to share information the backend is directly accessible wherever it is needed and all Bluetooth calls are removed. This solves problems 1 and 4.

Development was started by combining the main and mobile applications into one application. Having 3 top level windows (Problem 2) was easily solved by putting the Window objects under one QtObject component and having just 1 Window visible at once. After this the views were added one by one, building to WebAssembly after adding each view. At this point all code that caused the build to fail was commented out.

None of the Linux system calls used by the main application are necessary for simulating the user interface and problem 3 was solved by just removing the system calls.

For problems 5 - 7 an attempt to use the unsupported modules was made. Using the beta version of Qt 5.13 which includes multithreading support, as the 3d and Location modules require multithreading. The unsupported modules are configured by default to not build at all but this can be overridden by modifying the configuration files of the Qt sources. Through trial and error the modules did successfully compile. Building a Qt application that uses the 3d module or the Location module was also successful. However an application that uses the 3d module does not launch at all. An application that uses the Location module could be launched successfully but immediately after launching it crashes. At this point a decision to skip all parts of the application that use unsupported modules was made.

As the multimedia module is also unsupported no functioning media player could be used. This problem was solved by creating an object that mocks the functions the media player requires and serves dummy data to the player so that the user interface may be simulated. A similar object mocking the functions that the phone view of the application requires was created to make simulating phone calls possible.

Problem 9 was the most time consuming problem of the development process excluding the time it took to build Qt and time spent building the application. The mobile views were responsive as they need to scale on different screen sizes, but the main applications views did not scale at all. Creating responsive user interfaces with QML is simple due to it’s anchoring system and declarative nature. A lot of the size properties of the components were declared as pixels and all these needed to be changed to be relative to the screen size or relative to another component. This problem was not hard to solve but it was very time consuming.

5.5 Performance

The WebAssembly modules produced by Qt for WebAssembly are large as previously seen in Figure 9. The WebAssembly module of this application is 30 megabytes. Compressing the module with gzip reduces the size to 13 megabytes.

Figure 12 shows the start up time for the application on Firefox, Chrome, Edge and Opera. Start up times are fairly quick and consistent except on Opera the average time is fairly long and the start up time varies a lot.

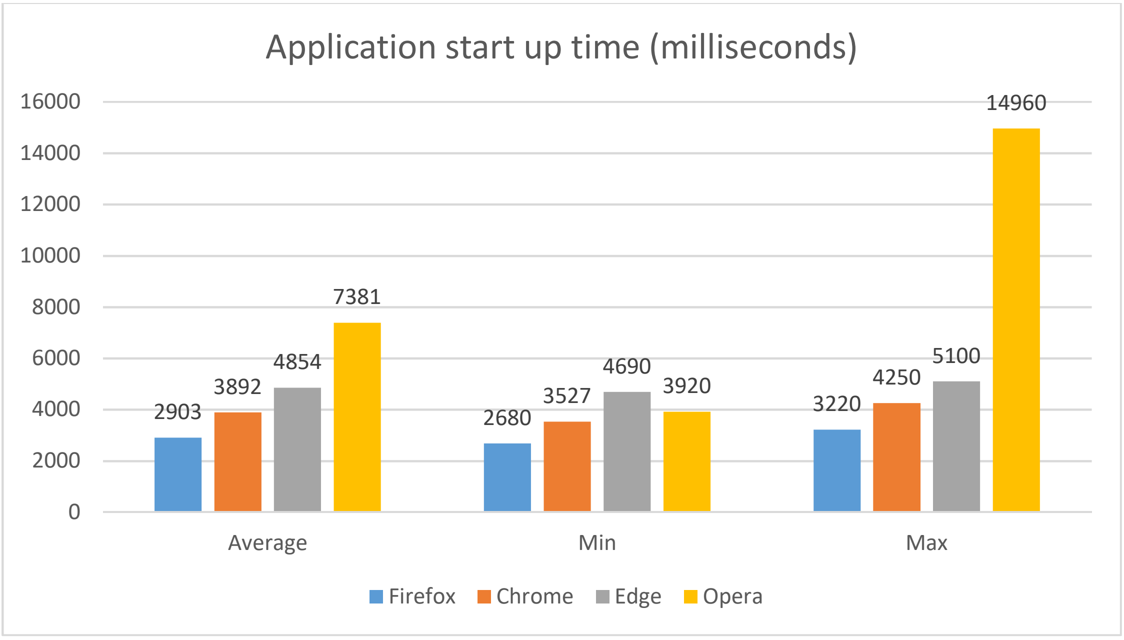


Figure 12 Application start up time

The application runs at a stable frame rate on tested browsers except for one view. In the mobile versions media view the frame rate is unstable. When the currently playing songs name does not fit onto the screen the text is animated to move to the left and loop around. When this animation runs the frame rate varies heavily. Figure 13, Figure 14 and

Figure 15 show a frame rate measurement on Chrome, Opera and Edge. The measurements are performed on the browsers native profiler tools.

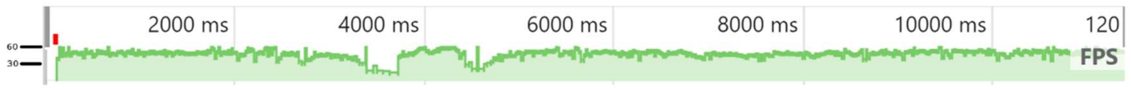


Figure 13 Inconsistent frame rate on Chrome.

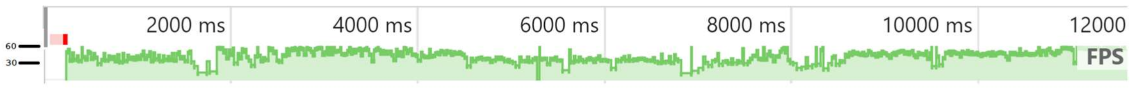


Figure 14 Inconsistent frame rate on Opera.

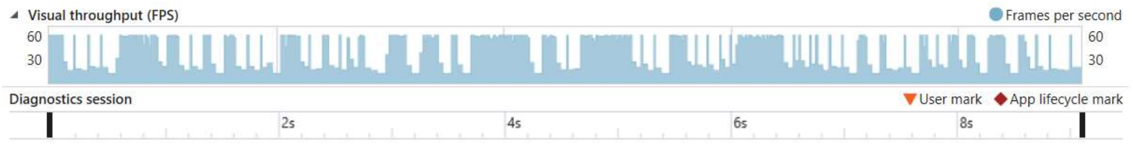


Figure 15 Inconsistent frame rate on Edge.

The animation is quite straining on the desktop version as well. However the WebAssembly version causes a much higher CPU load than the desktop version. Figure 16 shows the total CPU load of the test system when the animation is running on the desktop version and Figure 17 shows the CPU load for the WebAssembly build.

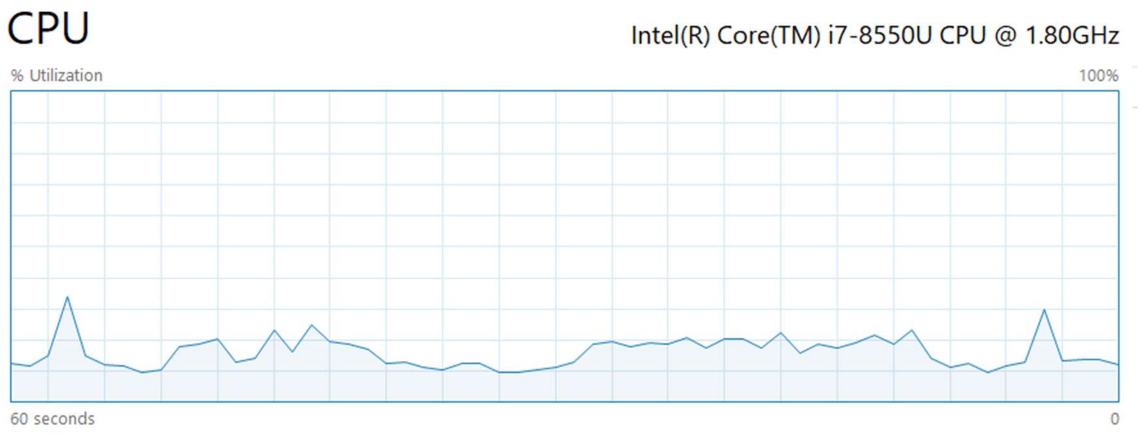


Figure 16 CPU load when running the desktop build.

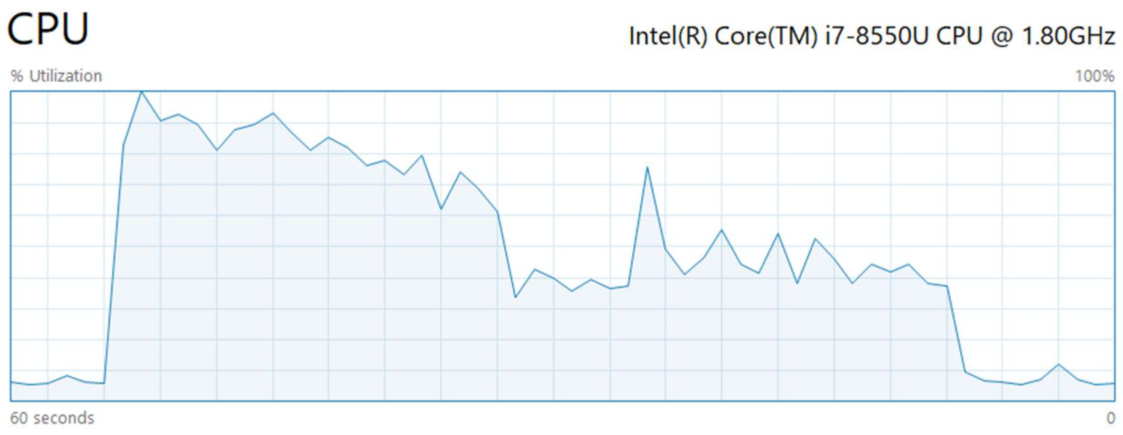


Figure 17 CPU when running the WebAssembly build on Firefox.

6 CLOSING CHAPTER

The goal of this thesis was to research WebAssembly and Qt for WebAssembly and produce a Qt for WebAssembly application that simulates the user interface of a Qt based In-Vehicle-Infotainment system.

WebAssembly is a new technology and its performance and future look promising. Compared to JavaScript, WebAssembly is fast and compared to native code it is not that fast, but considering that WebAssembly runs in the browser makes comparisons to JavaScript more reasonable than comparisons to native code.

Qt for WebAssembly allows Qt experts to write applications that run in browsers. At the time of writing Qt for WebAssembly was not yet officially released and this is reflected on its state. Many modules were not supported and QML user interfaces did not perform nearly as well as they did on other platforms.

Due to unsupported modules the user interface simulation application developed as a part of this thesis did not meet all its requirements. The development of the application will be put on hold until the unsupported modules are supported.

REFERENCES

[1] Blanchette J, Summerfield M. *C++ GUI Programming with Qt 4*, Second Edition. Westford, Massachusetts, USA: Prentice Hall, 2008. 718 p. ISBN 978-0-13-235416-5

[2] The Qt Company. *Supported Platforms*. Available from https://doc.qt.io/qt-5/supported-platforms.html [Accessed 19th April 2019]

[3] The Qt Company. *User Interfaces*. Available from https://doc.qt.io/qt-5/topics-ui.html [Accessed 19th April 2019]

[4] The Qt Company. *QML Applications*. Available from https://doc.qt.io/qt-5/qmlapplications.html [Accessed 19th April 2019]

[5] The Qt Company. *Qt Quick*. Available from https://doc.qt.io/qt-5/qtquick-index.html [Accessed 19th April 2019]

[6] The Qt Company. *Signals & Slots*. Available from https://doc.qt.io/qt-5/signalsandslots.html [Accessed 19th April 2019]

[7] The Qt Company. *The Meta-Object System*. Available from https://doc.qt.io/qt-5/metaobjects.html [Accessed 19th April 2019]

[8] The Qt Company. *Using the Meta-Object Compiler (moc)*. Available from https://doc.qt.io/qt-5/moc.html [Accessed 19th April 2019]

[9] Yegulalp S. What is WebAssembly? The next-generation web platform explained. *InfoWorld.com; San Mateo*. San Mateo, San Mateo, USA: Infoworld Media Group, 2018. Available from https://www.proquest.com/products-services/ProQuest-Research-Library.html [Accessed 22nd April 2019]

[10] WebAssembly. *Introduction*. Available from https://webassembly.github.io/spec/core/intro/introduction.html [Accessed 25th April 2019]

[11] Mozilla. *What is WebAssembly*. Available from https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts [Accessed 22nd April 2019]

[12] Clark L. *Creating and Working with WebAssembly modules*. 2017. Available from https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/ [Accessed 13th May 2019]

[13] Clark L. *Standardizing WASI: A system interface to run WebAssembly outside the web*. Available from https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ [Accessed 26th April 2019]

[14] Emscripten. *About Emscripten.* Available from https://emscripten.org/docs/introducing_emscripten/about_emscripten.html [Accessed 25th April 2019]

[15] Clark L, Schneiderkeit T, Wagner L. *WebAssembly's post-MVP future: A cartoon skill tree* Available from https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/ [Accessed 25th April 2019]

[16] Clark L. *Calls between JavaScript and WebAssembly are finally fast*. Available from https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-%f0%9f%8e%89/ [Accessed 13th May 2019]

[17] WebAssembly. *Binaryen*. GitHub repository, 2019. Available from https://github.com/WebAssembly/binaryen [Accessed 25th April 2019]

[18] Surma. *Replacing a hot path in your app's JavaScript with WebAssembly*. Available from https://developers.google.com/web/updates/2019/02/hotpath-with-wasm [Accessed 8th May 2019]

[19] Gurgone G, Spiess P. *A Real-World WebAssembly Benchmark*. 2018. Available from https://pspdfkit.com/blog/2018/a-real-world-webassembly-benchmark/ [Accessed 12th May 2019]

[20] Zakai A. Fast Physics on the Web Using C++, JavaScript and Emscripten. *Computing in Science & Engineering*. 2018;20(1): 11 – 19. Available from doi:10.1109/MCSE.2018.110150345

[21] Haas A, Rossberg A, Schuff DL, Titzer BL, Holman M, Gohman D, Wagner L, Zakai A, Bastien JF. Bringing the web up to speed with WebAssembly. In: *PLDI 2017 Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 18 – 23 June 2017, Barcelona, Spain*. New York: ACM; 2017. p.185-200. Available from doi:10.1145/3062341.3062363

[22] MatthiasJReisinger. *PolyBenchC-4.2.1*. GitHub repository, 2019. Available from https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 [Accessed 14th May 2019]

[23] Jangda A, Guha A, Powers B, Berger E. *Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code.* University of Massachusetts Amherst 2019. Available from arXiv:1901.09056v2 [Accessed 20th May 2019]

[24] Sørvig MJ. *Overview: Qt for WebAssembly*. Video webinar. 2018. Available from https://resources.qt.io/qt-on-demand-webinars/overview-qt-for-webassembly-on-demand-webinar [Accessed 6th May 2019]

[25] ”Can I use”. *gzip*. Available from https://caniuse.com/#search=gzip [Accessed 15th May 2019]

[26] ”Can I use”. *brotli.* Available from https://caniuse.com/#feat=brotli [Accessed 15th May 2019]

How to build Qt 5.12 for WebAssembly on Windows 10

This guide is written for Windows 10 users. Qt 5.12 for WebAssembly does not support Windows 10 directly so the Windows Subsystem for Linux will be used.

If you cannot access the Windows Store see https://docs.microsoft.com/en-us/windows/wsl/install-manual for instructions on installing WSL.

The following modules are supported: module-qtbase module-qtdeclarative module-qtquickcontrols2 module-qtwebsockets module-qtmqtt module-qtsvg module-qtcharts. Other modules may but are not guaranteed to work.

Enable Windows Subsystem for Linux by running the following command in PowerShell as Administrator.

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-
Windows-Subsystem-Linux
```

From the Windows Store install Ubuntu 18.04.

Open Ubuntu 18.04 and follow the instructions to finish the installation.

Update the system with the following commands

```
sudo apt-get update
sudo apt-get upgrade
```

Install necessary packages

```
sudo apt-get install python nodejs build-essential '^libxcb.*-
dev' libx11-xcb-dev libglu1-mesa-dev libxrender-dev libxi-dev
```

Download, install and activate emscripten and set the PATH variables. In this guide the following commands are run and emscripten is downloaded in “~”.

```
git clone https://github.com/emscripten-core/emsdk.git

cd emsdk

./emsdk install sdk-1.38.16-64bit

./emsdk activate sdk-1.38.16-64bit

source ./emsdk_env.sh
```

Download and unpack the Qt 5.12 sources. . In this guide the following commands are run and the Qt sources are downloaded in “~”

```
wget https://download.qt.io/official_releases/qt/5.12/5.12.0
/single/qt-everywhere-src-5.12.0.tar.xz

tar -xvf qt-everywhere-src-5.12.0.tar.xz
```

Create a directory where to install Qt. In this guide the following commands are run in "~" and Qt will be installed to "~/QtWasm"

```
mkdir QtWasm

cd QtWasm
```

From this directory call configure. When the script is "checking for valid makespec..." it may seem like it's hanging but it just takes a long time. Up to 10 minutes is not uncommon.

```
~/qt-everywhere-src-5.12.0/configure -xplatform wasm-emscripten
-nomake examples -prefix $PWD/qtbase
```

Make the modules you require. Building may take a long time. Building modules qtbase, qtdeclarative, qtquickcontrols2, qtwebsockets and qtcharts took about 3 hours.

```
make module-qtbase module-qtdeclarative [other modules]
```

Find your project. access the windows files with the following command. Replace "c" with the correct drive.

```
cd /mnt/c
```

Build the project. It will take some time. Building a hello world application took about 5 minutes.

In the project directory run

```
[path to qmake] && make
```

For example:

```
~/QtWasm/qtbase/bin/qmake && make
```

Serve the files to any http server. For example with python (replace 8080 with another port if necessary):

```
python -m SimpleHTTPServer 8080
```

Open `localhost:8080/[your_projects_name].html` in a browser.