WHITEPAPER

Qt or HTML5?

A Million Dollar Question

Burkhard Stubert
Chief Engineer, EmbeddedUse

October 2, 2017

Burkhard Stubert is Solopreneur and Chief Engineer at Embedded Use with over 20 years of experience in software engineering. He has developed numerous embedded and desktop applications with Qt and QML. He was the first to give QML trainings back in early 2010, when QML was still far away from an alpha release. His latest major products include driver terminals for forage and sugar root harvesters, infotainment systems for US and European car OEMs, an in-flight entertainment system and a display computer for e-bikes.

He offers independent professional services for developing embedded systems – preferably with a QML GUI and Qt/C++ middleware. Burkhard worked and lived in India, England and Norway and moved back to his native country, Germany, a couple of years ago. In his spare time, he enjoys hiking, biking and skiing through the Bavarian Alpes.

Executive Summary

This white paper explains how one of the world's largest home appliance manufacturers could save millions by using Qt over Web technologies.

Both Facebook and Netflix implemented their eponymous apps with Web. Despite spending millions of dollars, neither of them could achieve an iPhone-like user experience (60 frames per second and less than 100ms response to user inputs) on anything less powerful than a system-on-chip (SoC) with four ARM Cortex-A9 cores.

In contrast, numerous products like infotainment systems, in-flight entertainment systems, harvester terminals and home appliances prove that you can achieve an iPhone-like user experience (UX) on single-core Cortex-A8 SoCs. Our above-mentioned manufacturer HAM Inc. (renamed for the sake of confidentiality) verified these results by building both a Web and Qt prototype.

At a volume of one million units, the ***SoC for Qt is 11 euros cheaper per unit than the SoC for Web*** *– to achieve the same user experience.*

At a volume of one million units, an industrial-grade NXP i.MX53 SoC with a single Cortex-A8 core costs roughly 10 euros. This is enough for Qt. At the same volume, an NXP i.MX6 SoC with four Cortex-A9 cores required for Web costs roughly 21 euros.

This means that Qt can reduce hardware costs by over 53 percent!

Even if HAM offset the SoC costs against the costs of the commercial Qt license, HAM would have to pay millions of euros more for a Web than for a Qt solution. And, HAM would have no way to scale down the Web solution to mid-range and low-end appliances.

Table of Contents

1 Introduction

Let me introduce you to HAM Inc. Its real name was changed for confidentiality, but it is one of the world's biggest home appliance manufacturers. HAM produces millions of appliances like ovens, cooktops, washing machines, dish washers and refrigerators per year. More than 90% of the appliances are "powered" by a microcontroller with no operating system at all or a very simple real-time operating system.

The HMI consists of physical knobs, buttons and displays without touch (many with 7-segment displays, some with TFT displays). Only premium appliances have a touch display, an operating system (e.g., QNX, Linux) and a system-on-chip (SoC) with a microprocessor (CPU) and a graphical processor (GPU). The SoC sports a single-core ARM Cortex-A8, which is in the lower middle class with respect to performance. The GPU supports OpenGL. Premium appliances make up for 5% of all appliances produced by HAM and cost 2000 euros and more.

Within the next 10 years, HAM expects to move its premium appliances up to SoCs with multiple Cortex-A9 cores. The use of 64-bit SoCs with ARMv8-A architecture is unlikely, because they are far too expensive. It also expects to use SoCs with a single Cortex-A5/A7/A8 or ARM11 core in most of its mid-range appliances. Low-end appliances will stay on microcontrollers.

HAM also wants to retain its reputation as a manufacturer of high quality products, which is why HAM aims to deliver an iPhone-like user experience across its product lines.

In early 2016, HAM had to start looking for a new HMI and application framework, because the framework used by HAM was discontinued. HAM quickly narrowed down the possible contenders to Web (AngularJS HMI on Blink, Chromium's rendering engine) and Qt (QML HMI on Qt/C++).

AngularJS is released under MIT license and Blink mostly under BSD license. It seemed to HAM that Web technologies would cost nothing. The same is true for Qt under LGPLv3. However, HAM's lawyers prohibited the use of LGPLv3 mainly because of the anti-tivoisation clause. If HAM wanted to use Qt, it had to use Qt under a commercial license. Using Qt Commercial entails paying per-developer and per-device license fees.

The situation looked dire for Qt, when HAM asked me in early 2017 to make a case for Qt – despite its presumably much higher costs than Web. It was clear to me that Qt would only stand a chance if I argued in the only currency that purchasing departments all over the world understand: dollars, millions of dollars! I had to find some substantial hidden costs of the Web solution that HAM had overlooked. I set out to prove this hypothesis:

A Web solution requires a considerably more powerful and more expensive system-on-chip (SoC) than a Qt solution to achieve an iPhone-like user experience (UX).

2 Recap of ARM SoCs

A brief recap of ARM SoCs may help to understand my argument. The first table shows example devices for some ARM core designs. ARM9 and ARM11 cores implement the 32-bit architectures ARMv5 and ARMv6, respectively. They always have a single core. ARM9 cores do not have a GPU, whereas ARM11 cores may have one.

The Cortex-A8, A9 and A15 cores are based on the 32-bit ARMv7-A architecture. They all have a GPU with OpenGL acceleration. The Cortex-A8 is always single core, whereas the A9 and A15 are multi-core.

The high-end Cortex-A57/53 implements the 64-bit ARMv8-A architecture. These SoCs have a performance like current low-end desktop PCs.

If I note a class of products in generic terms like “IVI in middle-class cars (2017)”, I know one or more products with this SoC. I am under a non-disclosure agreement though and are not allowed to give you the company’s name.

Core	Example Devices
Cortex-A57/53	IVI in premium cars (2015), Samsung GS6, Raspberry Pi 3
Cortex-A15	IVI in middle-class cars (2017), Samsung GS4
Cortex-A9	iPhone 4S, IVI in middle-class cars (2013), agricultural terminals (2017)
Cortex-A8	Premium ovens (2013), Nest thermostat, iPhone 4, In-flight entertainment (2014), agricultural terminals (2013), Nokia N9
ARM11	Raspberry Pi 1, iPhone 3G, Nokia N8
ARM9	Nintendo DSi, Lego Mindstorm EV3, VoIP phones (2007)

The next table shows the prices of well-known industrial-grade SoCs for volumes of 1, 100, 10,000 (10K) and 1,000,000 (1M) units. All prices are in Euros. I sampled the prices for 1 and 100 units from the websites of electronics distributors. Prices for higher volumes are not publicly available. I extrapolated the prices for 10,000 and 1,000,000 by applying a 33% discount per 100 times multiple. The extrapolated prices for one million units were in the ballpark I knew from several projects I had worked on.

Consumer SoCs, which do not operate in extreme temperatures from -30 °C to +70 °C and which do not have to resist a high-pressure cleaner or extreme dust, are considerably cheaper. The same goes for SoCs with less cores. For example, an industrial-grade i.MX6 with two instead of four cores goes for nearly half the price.

Core	Architecture	Cores	1	100	1K	1M
R-CAR M3	Cortex-A57/53	4/4	202.50	135.00	90.00	60.00
TI AM5728	Cortex-A15	2	124.00	82.65	55.10	36.75
NXP i.MX6	Cortex-A9	4	71.35	47.55	31.70	21.15
NXP i.MX53	Cortex-A8	1	33.05	22.05	14.70	9.80
NXP i.MX35	ARM11	1	16.00	10.65	7.10	4.75
NXP i.MX25	ARM9	1	11.30	7.55	5.05	3.35

3 Web Scales Down Badly

Facebook could not achieve a satisfactory user experience with Web for its app on an iPhone 4S. The iPhone 4S was powered by a dual-core Cortex-A9 SoC. Hence, Facebook switched from Web to native for its smartphone apps in 2012 and has never looked back since. HAM had the same experience with its Web prototype in 2017.

Netflix never gave up on Web. It developed its own rendering engine and a highly optimised version of ReactJS. Both are proprietary. They came close to an iPhone-like UX (30 frames per second, 110 ms response times to user inputs) on higher end devices at least powered by a Cortex-A8, if not a Cortex-A9. Netflix spent millions of dollars to get a decent UX on TVs, STBs and BD players. There are few companies in the world who have the developer talent and the money to pull off this feat.

3.1 Facebook – Multi-Million-Dollar Mistake with Web

In an interview at the Disrupt SF 2012 conference (Olanoff, 2012), Mark Zuckerberg (CEO and founder of Facebook) conceded a multi-million-dollar mistake.

“The biggest mistake that we made as a company is betting too much on HTML5 as opposed to native [...] We burned two years.”

He was talking about the Facebook app on smartphones. This insight led Facebook to change from Web to native apps on iOS and Android. Zuckerberg also pointed out the reason for this move to native.

“The mobile [user] experience is so good that good enough is not good enough. We need to have something that is at the highest quality level. The only way we are going to get there is by going native.”

In other words, the world’s top Web developers at Facebook were not able to achieve a good user experience (UX) on a 2012 smartphone. Facebook could not achieve a good-enough user experience on an iPhone 4s. The iPhone 4s (iPhone 4S, 2011) sports a dual-core Cortex-A9 SoC clocked at 800 MHz with a multi-core GPU for OpenGL graphics acceleration and with 512 MB RAM.

Facebook’s representative on the W3C Advisory Committee, Tobie Langel, gives more details about the technical issues (Langel, 2012).

- Not enough RAM and a lack of tools to figure out what is going wrong: “The biggest issues we've been facing here are memory related. Given the size of our content, it's not uncommon for our application to exhaust the hardware capabilities of the device, causing crashes. Unfortunately, it's difficult for us to understand exactly what's causing these issues.”
- Scrolling performance “[...] is one of our most important issues. It’s typically a problem on the newsfeed and on Timeline which use infinite scrolling [...] and end up containing large amounts of content.”
- “Inconsistent framerates, UI thread lag (stuttering).”
- Not only true for different operating systems but also for different rendering engines: “Native momentum scrolling has a different feel across operating systems. JS implementation end up being tailored for one OS and feels wrong on other ones (uncanny valley).”

You may object that Facebook’s blunder happened more than five years ago and that Web has improved tremendously in that time. The findings of HAM Inc. refute these objections.

In 2017, HAM implemented a crucial part of an oven HMI once with Web and once with Qt. The Web variant yielded a bad user experience on anything less powerful than a quad-core Cortex-A9 (NXP i.MX6). The user experience on the quad-core Cortex-A9 was acceptable, but not good. The main problems were long start-up times, high RAM consumption and stuttering during animations and scrolling.

Using 64 MB of RAM for Qt instead of 512 MB for Web makes quite a big cost difference – especially for high volumes. Lacking good tools for finding performance problems (memory, speed), increases the non-recurring engineering costs significantly, and debugging and profiling costs many times more than coding.

3.2 Netflix – Muddling Through with Web

Netflix faced an enormous fragmentation problem, when moving from DVDs to video streaming in 2007. The Netflix application had to run on TVs, set-top boxes (STBs), DVD/BD players, gaming consoles, smartphones, tablets, laptops and desktop PCs. The CPUs of these devices ranged from the low end (e.g., ARM9) to the high end (e.g., Intel Core i7). Some devices had a GPU, some not. The screen resolutions and formats varied widely. This is not unlike HAM's situation, although Netflix's target devices are more powerful.

It was impossible for Netflix to develop its application for each device. They focused on a few devices at first. However, they had to reach more devices and more customers to grow their business. Netflix turned to Web in 2009/10 to achieve this goal. They chose a hybrid approach. The HMI was written in standard HTML5, CSS and JavaScript and rendered with the QtWebkit library. In contrast to browsers, QtWebkit allows the application to access hardware capabilities directly. Browsers run in a sandbox and allow only very limited access to hardware capabilities.

A presentation of two Netflix engineers (McCarthy & Trott, 2011) in 2011 gives a good idea about the problems with this approach. If you know the Netflix app from those times, you also know that the user experience was simply awful. In comparison to the Facebook HMI, the Netflix HMI is simple: Two to three horizontal cover flows with up to six images are visible at the same time.

Netflix got away with such a bad user experience, because they were the only game in town. This changed around 2014, when other players like Apple, Amazon and HBO entered the game. Netflix's user experience was not competitive any more. They had to change a lot.

Netflix built their own custom-tailored rendering engine, Gibbon. Gibbon is written entirely in JavaScript and runs on a non-JIT version of JavaScriptCore. The Netflix engineers rewrote the entire HTML5 HMI with ReactJS (React, 2013), a JavaScript library for building user interfaces. Driven by thorough profiling, they morphed ReactJS into a proprietary variant React-Gibbon, which is highly optimised for their Gibbon rendering engine. If ReactNative had been around in 2014, they would have started with that.

The blog post "Building the New Netflix Experience for TV" (Nel, 2013) and the video "Performance without Compromise" (McGuire, 2016) show how difficult it was to achieve response times to user inputs of 110ms and not quite fluid animations at 30 frames per second (fps) on most devices. Moreover, the Netflix engineers were helped by the fact that TVs, set-top boxes (STBs) and DVD/BD players gained more powerful CPUs supported by GPUs to cope with full HD. For example, the 5th-generation Roku player is powered by a 64-bit, quad-core ARM Cortex-53 like the Raspberry Pi 3.

This effort must have cost Netflix dozens of person years and millions of dollars. Nevertheless, it still falls short of the golden standard of less than 100ms and 60fps set by the iPhone.

It is also important to understand that only very few Web developers in the world can optimise the performance of a web application on an embedded device in the way Netflix did. Typical web developers build web applications on computers that are more than 400 times more powerful than the average TVs on which the Netflix application runs, and most developers struggle to get 100ms response times and 30 fps on these computers.

Compare that with the experience of one of HAM's competitors.

Two developers, one beginner and one experienced developer, could develop a Qt-based HMI for an oven on a single-core Cortex-A8 with a good user experience in less than 1.5 years.

The developers did not have to write their own rendering engine and did not have to create their own variant of QML.

The moral of the Netflix story is that Web incurs huge extra development costs (read: millions of dollars) to achieve a decent user experience on SoCs like a Cortex-A9, let alone on less powerful SoCs.

3.3 Hardly Any Embedded Web Applications

I could only find very few embedded devices using Web technologies for their core HMI. The most prominent user of Web is, obviously, Netflix. Another example from the same industry is Livebox Play (SoftAtHome, 2013), the STB by the French telecom Orange. Many apps on TVs and STBs are written with Web technology (e.g., HbbTV, proprietary subsets of HTML5).

For example, the catch-up TV apps of Germany's public TV channels are written with HbbTV, a subset of HTML5. The problem with these apps is a lacking user experience.

I am not aware of a single home-appliance maker who uses Web technology. Electrolux explicitly decided against using Web technology in 2011.

The automotive industry also makes little use of Web. I could only find one infotainment system that has a core HMI built with HTML5: the Porsche 918 Spyder's (S1nn, 2014).

JavaScript frameworks like Angular and ReactNative advertise themselves for mobile and desktop, not for embedded. Angular's tagline is "One framework. Mobile & desktop.", ReactNative' is "Learn once, write anywhere: Build mobile apps with React". It should be noted that smartphone SoCs are much more powerful than the typical embedded SoCs.

4 Qt Scales Down Well

Numerous embedded Qt applications used daily by millions of people speak for themselves of how well Qt scales down. The Electrolux hit a nerve with HAM, as Electrolux went through a very similar decision process as HAM – but already 5 years earlier.

4.1 Numerous Embedded Qt Applications

The following diagram shows five industries – automotive, agriculture, avionics, fitness and home appliances – where Qt is widely used. Qt is also making big inroads into industrial automation and med tech. You can find more success stories from many other industries on the Built with Qt (qt.io, 2017) web page.

🚗	💪	🚜	✈️	🍳
• 7 Top-15 OEM's • 2 EV OEM's • 12+ tier1 Suppliers • More...	• eGym • e-Bikes • Precor	• 1 of Top-3 • ROPA • CCI • Krone • More...	• 2 of Top-3 OEM's including Panasonic	• Electrolux • HAM

The SoCs in these Qt products range from ARM11s over Cortex-A8s and Cortex-A9 all the way up to Cortex-57/53s.

4.2 Qt Success Story: Electrolux

Electrolux went through a very similar decision process as HAM in 2011/12. Electrolux talked about its decision (Penacchio, 2014) publicly at the Qt Day in Italy, 2014. Electrolux had compared several HMI and application frameworks including Qt and Web. The winner was Qt:

"[In 2011] Electrolux decided to invest globally in Qt and specifically in Qt Quick for the development of high-end user interfaces for appliances [...] Qt has the potential to be a strategic user interface development platform for all mid/high-end appliances for Electrolux."

Electrolux said that it has released one oven with Qt (Electrolux, 2015) and that more appliances would follow. Given the time frame from 2012 to 2014 for the oven and the razor-thin profit margins for home appliances, the SoC is likely an ARM11 with GPU or a Cortex-A8. The cover flow with the recipes looks very smooth.

Electrolux gave a meaningful list of pros and cons for using Qt.

+ Pros	Cons –
Faster implementation and learning curve	License selection (Commercial vs. LGPL) difficult
Much cheaper than other HMI frameworks	Missing integration with UI design tools like Photoshop, Illustrator (feature implemented in Qt 5.10)
More robust and powerful	
Clear separation of GUI and backend: designers can change GUI	
Easy integration with C/C++	
Smooth animations	
And more ...	

Electrolux's findings were quite like HAM's – although five years earlier.

5 Nothing Will Change Anytime Soon

Even after having seen all the evidence against Web, one Web supporter at HAM said: “Browser-based HMIs for home appliances will come, but it will still take some time. Probably on a different SoC.”
For this to come true, Web would have to overcome some inherent problems on embedded systems.

5.1 Flash, RAM and Power

The two shared libraries Qt5Quick and Qt5Qml, which include the QML rendering engine and the JavaScript engine, have a combined size of 8 MB (Qt 5.9.1, stripped release build for Ubuntu 16.04 LTS).

The shared library Qt5WebEngineCore, which includes the Web rendering engine and the JavaScript engine, comes in at 103 MB. The same library had a size of 82 MB in Qt 5.7.1. Five years ago, the predecessor library, Qt5Webkit, had a size of 15 MB. You could bring down the size to 11 MB with some compile switches.

The standard installation of the Chromium browser uses 42 MB on Ubuntu 16.04 LTS. This is still five times more space than needed by QML. The huge Web library sizes have critical implications.

A Web application uses considerably more RAM than a Qt application. If you force a Web application into the same RAM size as a Qt application, it will incur cache misses and must load more pages from flash. The Web application is slower.

A Qt application for a home appliance can comfortably run on a system with 64 MB of flash and main memory. A Web application certainly needs more than 64 MB, probably 256 MB to 512 MB. More flash and main memory means higher costs for the Web solution.

Most Web solutions including HAM’s would use Blink as the web engine. John Gruber from Daring Fireball (Gruber, 2017) ran the same script, simulating reading of web pages on the same MacBook Pro once with Chrome and once with Safari. The battery lasted 5:30 h for Safari and 3:40 h for Chrome. This does not make Web appealing, because home appliances must be extremely power conscious to achieve AAA ratings.

5.2 Start-up Times

The huge size of the Chromium browser also implies slower start-up times for Web applications. Starting the Chromium web browser on a high-end laptop with a quad-core Intel Core i7 at 2.5 GHz, 16 GB RAM and a fast SSD takes more than 2 seconds. Starting Chromium on a Raspberry Pi 3 takes 6 seconds. Starting the web browser in less than 10 seconds on a Cortex-A8 SoC would require a significant time-consuming optimisation effort with uncertain results. These start-up times do not include the time for starting the operating system and loading the Web application.

Compare this with the Qt case. Starting both Linux and a QML instrument cluster on a quad-core ARM Cortex-A9 (1 GHz, 1GB RAM, 8GB flash) takes 1.5 seconds (Avila, 2016). The QML application takes 0.25 seconds and Linux 1.25 seconds. This is less than the Chromium web browser needs on a high-end laptop. Users expect devices like ovens, cooktops, printers, STBs and TVs to be instantly on. Start-up times of more than three seconds are an immediate knock-out criterion for potential buyers.

The fast start-up of the QML application depends critically on static linking of the Qt libraries and on the QML compiler. Both features are only available with the commercial Qt license.

Static linking reduces the size of executable further, as it removes all library parts that are not needed by the application. When the operating system starts an application, it must load the executable and all dependent libraries from flash memory. Reading from flash memory is extremely slow compared to reading from RAM. Hence, the bigger the executable and libraries the longer the start-up takes.

Even with static linking, the Chromium browser had a size of 42 MB compared to 8 MB of the shared QML libraries. Even with static linking, Web executables will be five times bigger than Qt executables. Hence, Web applications will take five times longer to load than Qt applications.

The QML compiler translates QML code into C++ code, which is translated into machine code by the C++ compiler as usual. The QML compiler also compiles JavaScript functions and expressions. Hence, the just-in-time (JIT) compilation of JavaScript engines is done at compile time and not at run time.

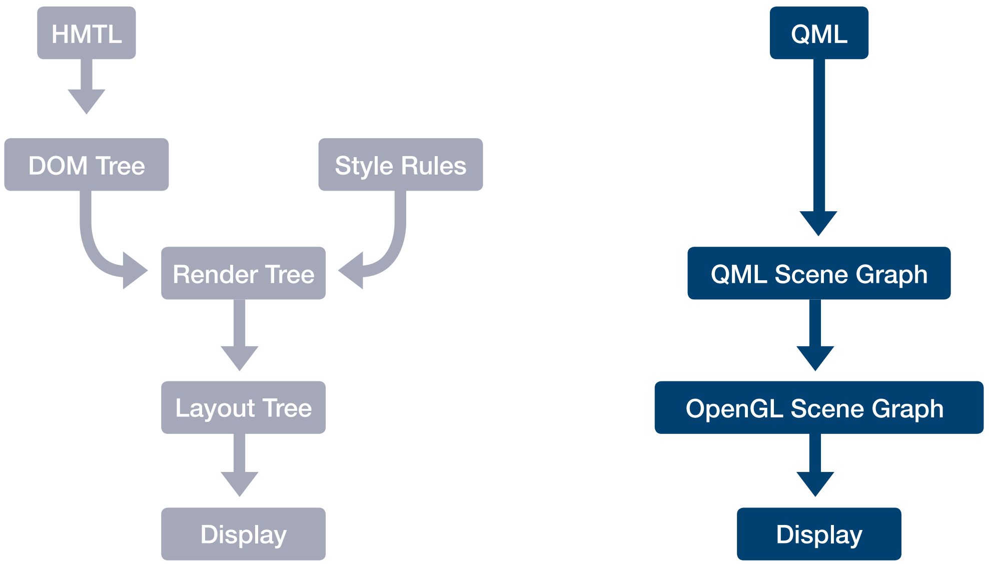
I achieved a 30% faster start-up for the QML application of a harvester terminal (Stubert, 2016) – just by using the QML compiler.

Web has nothing like an "HTML/CSS/JavaScript" compiler. Writing such a compiler would be very difficult, because it had to cover the complete, unwieldy HTML, CSS and JavaScript standards. If feasible at all, Web's catch-up would take years, in which QML development would not rest.

Even without the QML compiler, Web would not gain an edge over QML because of the huge performance improvements of the JavaScript engines over the last years. The Qt developers can use the same techniques to speed up their JavaScript engine. QML is easier to optimise, as it is a much simpler language than HTML and CSS and as the Qt developers have full control over QML.

5.3 Rendering Flows

The next diagram shows the Web rendering flow (left) and the QML rendering flow (right).



Here is a brief description of the Web rendering flow (see here (Garsiel & Irish, 2011) for a detailed explanation).

- **Step 1:** The HTML parser creates the DOM tree from the HTML documents. The HTML grammar is not context-free and hence hard to parse.
- **Step 2:** The CSS parser creates the style rules from the style sheets.
- **Step 3:** The style rules are applied to the nodes in the DOM tree. It is hard to figure out, which rule applies to which DOM node because of the cascading nature of the style rules. The result of this step is the render tree, which contains the visual nodes in the right rendering order.
- **Step 4:** The layout step calculates the position and the size of each node.
- **Step 5:** The painting step traverses the render tree node by node and paints each node on the display.

Steps 1, 2 and 3 are the most expensive steps. Applying the cascading style rules in step 3 is especially expensive. Every CSS rule can lead to a costly transformation of the render tree. Steps 4 and 5 are similar for Web and QML.

Netflix optimises the first three steps heavily. They use as little HTML5 and CSS as possible to reduce the number of CSS rules and the size of the Render Tree. They apply algorithms to reduce the size even further. ReactNative uses a similar approach.

Hence, the QML flow needs only one step instead of Web's first three steps

QML does not separate content (HTML) and style (CSS). Hence, the QML flow needs only one step instead of Web's first three steps. The QML flow avoids the costly step 3 of applying CSS style rules to the DOM tree. It is highly optimised towards a very direct and simple mapping onto the OpenGL scene graph.

If you choose Web, your developers will have to write the application code in a very special way with a constant focus on optimisation. Moreover, they will have to change the Web rendering engine to optimise the Render Tree.

While the Web developers are still trying to optimise their code and tools, the QML developers will have finished the application code. Sequality's experiment corroborates this (Larndorfer, 2017): Developers achieve less in the same time with Web than with Qt. And, the Web solution is less fluid and less responsive than the Qt solution.

6 Conclusion

The Facebook and Netflix stories show that you cannot achieve an iPhone-like user experience for Web applications with anything less than a quad-core Cortex-A9 SoC. The Netflix story highlights that this would cost millions of dollars.

HAM tested this by building both a Web and a Qt prototype for critical parts of their current HMI. Anything less than a quad-core Cortex-A9 SoC lead to poor user experience. Even on such a powerful SoC the Web solution suffered from long start-up times, huge memory consumption and sometimes stuttering scrolling and animations. HAM would have had to spend a considerable amount of time and money on optimising the Web solution – in addition to the higher SoC costs.

Many Qt products like the Electrolux oven, infotainment systems, harvester terminals and in-flight entertainment systems achieve an iPhone-like user experience on a single-core Cortex-A8 SoC. Qt scales down well even further. ARM11 SoCs with GPU like the Raspberry Pi 1 enable an iPhone-like UX as well. Even ARM11 SoCs without GPU or ARM9 SoCs offer a good tradeoff between a cheap SoC and a good-enough UX.

These findings allowed me to substantiate my hypothesis.

A Web solution requires at least a quad-core Cortex-A9 SoC to achieve an iPhone-like UX, whereas a Qt solution requires at most a single-core Cortex-A8 SoC.

With the prices for SoCs in chapter two, we can calculate the total cost for 10 000 and 1 000 000 units for Web and Qt respectively.

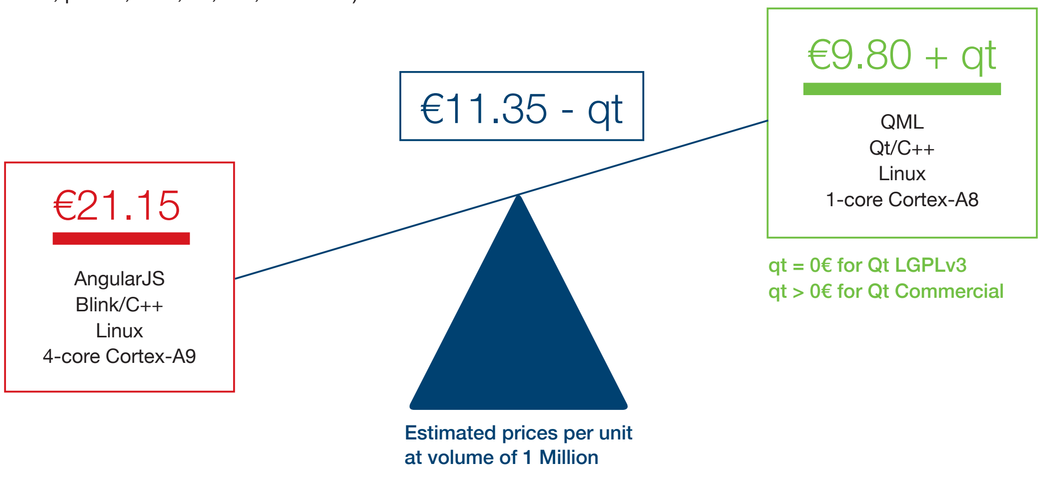
	€ / 10k units	**€ / 1M units**
Web: Cortex-A9 NXP i.MXP6 quad	3 170 000	21 150 000
Qt: Cortex-A8 NXP i.MX53	1 470 000	9 800 000
Cost Difference (€)	**1 700 000**	**11 350 000**

The cost for the SoCs are €1 700 000 and €11 350 000 lower for Qt at the different production levels. In other words, **Qt can save around 53% in hardware costs**!

The following diagram shows the per-unit cost difference assuming a volume of one million units. The variable qt denotes the per-unit costs of the commercial license. For Qt LGPLv3, qt is 0. For Qt Commercial, qt is greater than 0.

Same Application

(e.g., appliance, printer, STB, TV, IFE, terminals)



As qt was a fraction of the 11-euro cost difference, Qt would save HAM millions of euros. HAM's decision to go with Qt was easy.

7 Sources

Avila, R. (2016, 04 20). Fast-Booting Qt Devices, Part 1: Automotive Instrument Cluster. Retrieved from qt.io:
http://blog.qt.io/blog/2016/04/20/fast-booting-qt-devices-part-1-automotive-instrument-cluster/

Electrolux. (2015). Ovens Electrolux Color Touch Screen. Retrieved from Youtube:
https://www.youtube.com/watch?v=BpI7dku7Yr8

Garsiel, T., & Irish, P. (2011, 08 05). How Browsers Work: Behind the scenes of modern web browsers.
Retrieved from Html5 Rocks: https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#Rendering_engines

Gruber, J. (2017, 05 24). Safari vs. Chrome on the Mac. Retrieved from Daring Fireball:
https://daringfireball.net/2017/05/safari_vs_chrome_on_the_mac

iPhone 4S. (2011). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/IPhone_4S

KDAB. (2017, 09 28). CCI – putting intelligence into agriculture, using Qt. Retrieved from KDAB:
https://www.kdab.com/cci-putting-intelligence-agriculture-using-qt/?utm_source=Master+List+06-16&utm_campaign=d-ca3188043-EMAIL_CAMPAIGN_2017_09_25&utm_medium=email&utm_term=0_bdde4cdc11-dca3188043-101570617

Langel, T. (2012, 09 15). Perf Feedback - What's slowing down Mobile Facebook. Retrieved from W3C:
http://lists.w3.org/Archives/Public/public-coremob/2012Sep/0021.html

McCarthy, M., & Trott, K. (2011, 07 29). Netflix Webkit-Based UI for TV Devices. Retrieved from Slideshare:
https://www.slideshare.net/mattmccarthy_nflx/netflix-webkitbased-ui-for-tv-devices-9168822

McGuire, S. (2016, 03 23). Performance without Compromise. Retrieved from Netflix Technology Blog:
https://medium.com/netflix-techblog/performance-without-compromise-40d6003c6037

Nel, J. (2013, 11 18). Building the New Netflix Experience for TV. Retrieved from Netflix Technology Blog:
https://medium.com/netflix-techblog/building-the-new-netflix-experience-for-tv-920d71d875de

Olanoff, D. (2012, 09 11). Mark Zuckerberg: Our Biggest Mistake Was Betting Too Much On HTML5. Retrieved from Techcrunch:
https://techcrunch.com/2012/09/11/mark-zuckerberg-our-biggest-mistake-with-mobile-was-betting-too-much-on-html5/

Penacchio, N. (2014). Next-gen appliance e Qt Quick in Electrolux. Retrieved from Youtube:
https://www.youtube.com/watch?v=bnRYKNPf2x0

qt.io. (2017). Built with Qt. Retrieved from qt.io: https://www1.qt.io/built-with-qt/

React. (2013). Retrieved from ReactJS: https://reactjs.org

S1nn. (2014, 05 06). Porsche 918 Spyder Infotainment System Is Based on HTML5 Technology. Retrieved from Pddnet:
https://www.pddnet.com/news/2014/06/porsche-918-spyder-infotainment-system-based-html5-technology

SoftAtHome. (2013). Orange Livebox Play. Retrieved from SoftAtHome:
https://www.softathome.com/pages/customer-success-stories

Stubert, B. (2016, 11 20). 30% Faster Startup Thanks to QtQuick Compiler. Retrieved from Embedded Use:
http://www.embeddeduse.com/2016/11/20/30-faster-startup-thanks-to-qtquick-compiler/