

Case Study: Porting Qt to Windows Runtime

Andrew Knight

University of Tampere

School of Information Sciences

Interactive Technology:

User Interface Software Development

M.Sc. thesis

Supervisor: Roope Raisamo

June 2014

University of Tampere
School of Information Sciences
Interactive Technology: User Interface Software Development
Andrew Knight: Case Study: Porting Qt to Windows Runtime
M.Sc. thesis, 49 pages, 18 index and appendix pages
June 2014

With the abundance of operating system choices available to end-users, particularly for mobile devices, application developers look for ways to cut development time while increasing the portability and maintainability of their source code. One solution to this challenge can be found through use of cross-platform frameworks. Cross-platform frameworks function by abstracting the system-specific details of incompatible platforms into a common programming interface which developers can use to target many different devices and operating systems.

This thesis studies the abstraction architecture of Qt, a leading cross-platform C++ graphical user interface framework, with the goal of bringing a new platform, Windows Runtime, to the framework's set of supported targets. Windows Runtime is a collective programming interface for the Microsoft Windows 8 family of operating systems, including Windows 8, Windows Phone 8, and Windows RT. While Qt already supports a range of desktop and mobile operating systems – including Windows, Mac OSX, Linux/X11, Android, iOS, BlackBerry, and Sailfish – support for Windows Runtime is a new feature of the framework brought forth by this case study.

Current trends in cross-platform frameworks, particularly declarative user interface frameworks with a mobile emphasis, are assessed and compared to Qt's offering, and the implementation of Qt for Windows Runtime is prepared with these trends in mind. The implementation contributes to the open-source Qt Project, with the contributions included in the official Qt 5.3 release. Using the released version of Qt 5.3, a canonical Qt application is ported to the new platform and is certified and published in the Windows Store. Through this porting and publication process, an evaluation of the project's success is constructed within a cross-platform context.

The outlook for Windows Runtime as a growing platform is positive, as is the outlook for the uptake of Qt (and cross-platform frameworks in general) within modern device ecosystems. Moving forward, the quality and feature parity of Qt for Windows Runtime (as compared to competing frameworks) is expected to improve as users and open-source contributors make this new offering part of their respective development workflows and software projects.

Keywords: user interface software development, cross-platform, mobile, abstraction, Windows Runtime, C++, Qt, Qt Quick, QML, declarative, OpenGL, Direct3D

Contents

1 Introduction	2
2 Background	5
2.1 Cross-platform as an approach	5
2.1.1 C++ as a cross-platform language	5
2.1.2 GUI Portability	6
2.1.3 Paradigm shift in cross-platform GUIs	7
2.1.4 Feature parity and open-source	9
2.1.5 Weighing the options	10
2.2 Modern UI: an overview	12
2.2.1 Enter the grid	12
2.2.2 Managing the desktop	13
2.2.3 Auxiliary controls	14
2.3 The constraints of integration	16
2.3.1 A new set of interfaces	16
2.3.2 Working with the runtime	17
2.3.3 Interfacing with native UI	18
2.4 Requirements for a complete port	20
2.4.1 Tweaking the build system	20
2.4.2 Playing nicely in the sandbox	21
2.4.3 Addressing the issue of OpenGL	22
2.4.4 Improving tooling	23
3 Method	24
3.1 Addressing core issues	24
3.1.1 Build system changes	24
3.1.2 Core library functionality	26
3.1.3 Bootstrapping applications	27
3.2 Platform abstraction	29
3.2.1 Display management	29
3.2.2 Pointing device handling	30
3.2.3 Key handling	31
3.2.4 Desktop services	32
3.3 Enabling Qt Quick	34
3.3.1 JavaScript in the sandbox	34
3.3.2 OpenGL and ANGLE	35
3.3.3 Handling shader compilation at runtime	36
3.4 Sharpening the tools	38
3.4.1 Two paths: Qt Creator and Visual Studio	38
3.4.2 Creating the runner	39
3.4.3 Integrating with the IDE	40
4 Evaluation	42
4.1 Porting a complex application	42
4.2 Windows Store certification	44
5 Closing remarks	46
References	50
Glossary	63

1 Introduction

Qt, WinRT, and the importance of cross-platform UI

Since the introduction of the *desktop metaphor*, an influx of window management systems has pervaded the personal computing space. These windowing systems (typically coupled with a particular operating system and programming interface, forming a *platform*) have created a need for *cross-platform* application frameworks to ease the challenges of developing software for multiple, incompatible computing environments. As newer (often mobile-oriented) platforms carve out their own slices of the personal computing arena, they fragment developer mindshare across digital marketplaces, furthering the importance of cross-platform user interfaces (*UI*) as a tool to target multiple ecosystems while reducing development effort and software maintenance burden.

Windows Runtime (*WinRT*) provides an application programming interface (*API*) for writing applications which operate within the Windows 8 *Modern UI* environment, a touchscreen-friendly user experience available on PCs, tablets, and smartphones. For PCs, it blurs the boundaries between the traditional notions of desktop (driven by a keyboard and mouse) and mobile (touch-oriented) interfaces, while asserting a level of system trust by running applications within a platform security "*sandbox*". Such applications may be eligible for distribution in the *Windows Store*, a software marketplace for Windows devices. WinRT can be considered a platform "target" encompassing the operating systems utilizing the Windows Runtime API, including Windows 8, *Windows RT*, and *Windows Phone 8*.

Among the many cross-platform frameworks in active development is Qt [205], a veteran *open-source* solution written in C++. Qt provides a consistent programming model for developers to utilize the same application source code across a variety of target platforms, and it does so by abstracting platform-specific interfaces into a common API designed to work everywhere. Beyond platform abstraction, it provides an array of supplementary functionality through its many add-on modules, from multimedia playback to network services to text rendering (in fact, the document you are now reading was rendered using Qt's text layout engine). Despite its large feature set across a wide range of desktop and mobile operating systems, it has yet to add WinRT to its list of platform targets.

An assortment of incomplete solutions

Cross-platform UI frameworks have existed for decades, with wxWidgets (first appearing in 1992 [27]), Qt (its public debut in 1995 [28]), and GTK+ (1998 [29]) being well-known, *native* (that is, built with a compiled language such as C++) examples still in use today. As Letner et al. [1] have explored, the cross-platform playing field is becoming increasingly populated; beyond native frameworks, virtual machine solutions (using languages such as Java [30] or Ruby [31]), containers for web-based (or *hybrid*) applications, and even game engines can prove doubly useful as multi-platform UI toolkits. Research from Palmieri et al. [2] as well as Ohrt and Turau [3] has shown that mobile application frameworks, in particular, have recently grown out of a need to combat the constant rise and fall (disruption) of operating systems in the smartphone and tablet markets.

As an example of this disruption, WinRT introduces not only a new API for writing applications across devices, but a new window manager as well. In Meyers' terminology [4], the Modern UI uses a tiled approach to window positioning, as opposed to the "traditional" desktop interface or the fullscreen approach of modern tablet and smartphone operating systems. For Qt (and other cross-platform frameworks), WinRT brings new challenges by combining a touch-oriented system of viewports with a selection of features only found within the traditional desktop. Frameworks can stay competitive by integrating with a platform's native services; the combination of Qt and WinRT is no different in this regard.

Within the constraints of *abstraction* across many platforms (while being *adapted* to fit well to individual platforms), the framework must define its own ways of maximizing developer workflow efficiency and code reuse. There are many areas where this activity can be seen: Wojtczyk and Knoll [5] have examined such efforts in the build system, for example, while Bishop and Horspool [6] posit that *declarative* programming language technologies can improve code reusability (a paradigm described by Abrams et al. [7] already at the turn of the century). A trend in modern frameworks has been to adopt declarative techniques (such as web technologies like HTML and CSS) to aid in both rapid prototyping and more semantic programming of user interfaces, possibly as high-level languages to hardware-accelerated graphics APIs. Just as in the case of the build system, a high-level UI language requires supporting tools to help facilitate a high development pace and positive cross-platform development experience. In other words, effective cross-platform frameworks bring together tooling and programming patterns in a way that the framework might be described as "easy" or "enjoyable" to use (in addition to being "powerful" or "efficient").

In addition to tailored tooling and high-level GUI technology, the availability of a framework's source code brings its own advantages. As a study by Gary et al. [8] suggests, open source projects give the opportunity for a community of users and contributors to form around the framework, reducing development costs and improving its market position. Qt holds its own as a cross-platform framework, given its advanced tooling (via its integrated development environment, Qt Creator), its high-level GUI language and scene graph (via its Qt Qml [206] and Qt Quick [207] modules), and its strong open-source community (the Qt Project [32]). However, its lack of support for Microsoft's latest platforms (i.e., those utilizing the WinRT API) is something which may drive developers to alternative solutions.

Bringing Qt to Windows Runtime

Given this shortcoming in Qt's offering, examining and implementing the changes necessary to bring Qt to WinRT devices makes for a compelling case study. Over the course of this study, the evolution of Qt's cross-platform architecture and its integration with WinRT is documented, the necessary changes are implemented, and the value of this effort is assessed within a cross-platform context. In effect, this study serves two purposes: to discuss the theoretical and practical implications of *porting* a new target to a well-established cross-platform framework, and to bring forth a marriage of two technologies (Qt and WinRT) in the hope that it will prove useful to developers worldwide.

The catalyst for this study occurred in late 2011 as a job interview challenge, when I was tasked with showing a Qt application running within the Modern UI environment on a pre-release version

of Windows 8. The initial investigation yielded Qt on Metro [33], a technique for running an interactive Qt 4 application within "Metro", the branded name for the Modern UI environment at the time. The resulting demo garnered attention both from the press [34] and the community [36], and was especially pertinent in light of Nokia's recent announcement of its partnership with Microsoft [35]. Qt on Metro was only a limited proof-of-concept, though, so the effort was refocused in late 2012 with Qt 5 as a basis, and an early prototype was demonstrated at Qt Developer Days in November 2012 [37]. With efforts by myself, my colleagues, and members of the community, an announcement was made detailing the kick-starting and continued development of the project [38]. A technology preview [39] of Qt for Windows WinRT was released alongside Qt 5.2 in November 2013, and a highly functional supported Beta was released in May 2014 as part of the official Qt 5.3 release [40].

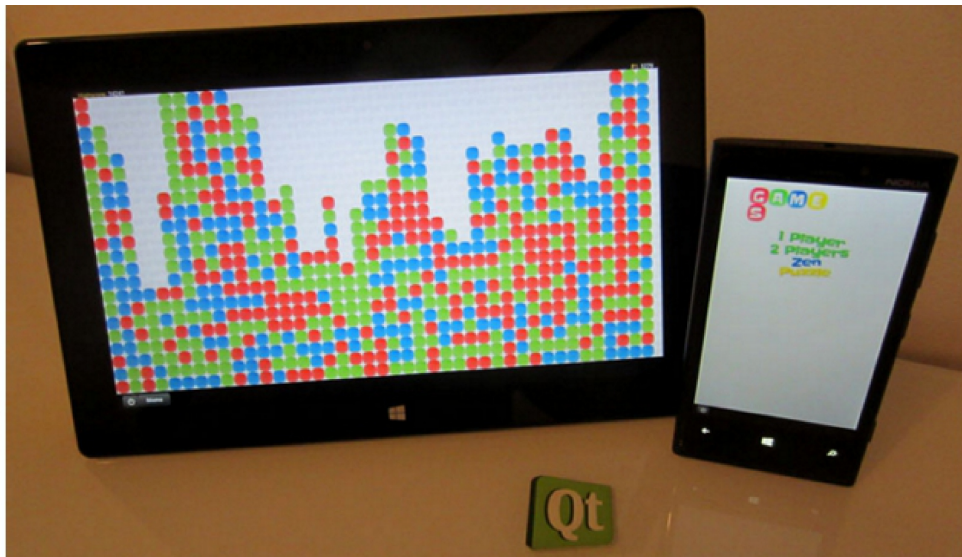


Figure 1: Qt and WinRT, a natural fit: the Qt Quick Same Game demo [208] running on the Microsoft Surface RT (left) and Nokia Lumia 920 (right) [41].

While Qt 5 is engineered to be the easiest version yet to port to new platforms [42], any porting effort is not without its challenges. As these challenges are investigated and implemented, the goal can be clarified to that of obtaining full usability of Qt's GUI APIs on WinRT devices. In order to go from a completely unsupported platform to a functional Qt port, there is a mix of low-level (*toolchain* and build system) requirements, middleware details (such integration with the graphics stack), and high-level integration points such as platform-specific UI controls. Additionally, a number of supporting tools also receive attention, with the goal of bringing Qt for WinRT to a similar level of support within Qt's *IDE*, Qt Creator, as it has for other platform targets.

Through this combination of platform integration and improved tooling, a "standard" Qt experience is packaged and distributed via the Qt Project for download and use by developers, with a gathering of feedback collected and noted via the channels provided by the Qt Project. The completeness and usability of the port is evaluated by building a canonical cross-platform demo (naturally including support for the target platforms, Windows 8 and Windows Phone 8) and documenting the challenges faced when publishing this application in the Windows Store.

Background

2.1 Cross-platform as an approach

"Nothing is more disagreeable to the hacker than duplication of effort. The first and most important mental habit that people develop when they learn how to write computer programs is to generalize, generalize, generalize. To make their code as modular and flexible as possible, breaking large problems down into small subroutines that can be used over and over again in different contexts."

–Neal Stephenson

As Stephenson points out in his famous essay on the history of operating systems [43], duplicating effort goes directly against a fundamental principle in software engineering: *don't repeat yourself* (DRY). Even so, DRY is not always an easy mantra to follow in the divisive world of GUI programming. A denizen of the technology society interacts, whether actively or passively, with dozens of computing platforms each day: apart from the typical personal computer, workstation, or mobile phone, the user may rely on computing systems in automobiles, public transportation systems, digital signage, as well as services operating remotely within the "cloud". As computing becomes ever more ubiquitous, the variety of hardware and software configurations for different tasks becomes vast and complex. With so many platforms in use – and many with similar goals – it becomes increasingly important for software to operate cross-platform: if not only for economic reasons, but also for the sanity of the programmer who is tasked with maintaining an application targeted at multiple devices and operating systems.

2.1.1 C++ as a cross-platform language

While GUI construction tends to be the topic of cross-platform development in contemporary frameworks, evolving hardware architectures and core libraries have faced incompatibilities long before the desktop arrived. To make native code programming easier - rather than programming in a low-level, platform-specific machine code - the advent of compiled languages such as C grew out of a need for a more natural, human-readable syntax that could be rebuilt (compiled) into machine instructions for any hardware architecture or operating system. The C language itself might be considered one of the original cross-platform "frameworks", as compilers translate platform-agnostic standard C procedures into hardware-specific machine code. Having its roots in the 1960s [9], C has endured and continues to be considered one of the most portable languages in existence [10]. From the Linux kernel [44] to the Mars Rover [45], C is ubiquitous... and while it may seem prosaic when compared to interpreted languages like Ruby or *JavaScript*, it continues to evolve and influence popular higher-level languages as well.

Given its foundation, C++ inherits much of the portability of C. The language itself is no silver bullet, though - it also takes cross-platform tools and *libraries* to promote real source portability, allowing for an abstraction of the platform-specific details met by the programmer. For strong adoption, the cross-platform library should use a royalty-free API to allow use anywhere, preferably with free and open source implementations to back it. This is one of the reasons the standard template library (*STL*) was developed - not only to provide a solid foundation of core functionality

for the C++ language, but also to allow compiler vendors to supply their own implementations (and optimizations) for the high-level primitives STL provides. Therefore, the C++ STL continues to provide one of the most portable code models for applications today, being used on Windows, Unix-style operating systems (Linux/BSD/Mac OS X), embedded real-time operating systems, as well as mobile offerings like *Android*, *iOS*, and Windows Phone 8. Assuming a compliant implementation exists for the platform (ideally available by default), STL source code should be compilable and runnable there.

2.1.2 GUI Portability

Despite STL's success, it is not comprehensive – certainly, it makes no attempt to provide GUI functionality. In the words of Bjarne Stroustrup (the creator of C++), "C++ is a language, not a complete system" [11]; for this reason, GUI applications tend to get "locked in" to a given library (typically the platform's) which is likely not as portable as the STL code it might use underneath. As Kassinen et al. [12] describe the problem, "many programming languages provide good cross-platform support in the sense that they can be compiled for, or interpreted on, several platforms", but they continue to state that this is "not sufficient" in "the real world", as different APIs and restrictions of the operating environment must be taken into account. While a developer might be able to use C (or C++) as a universal language, using a given platform's native UI library outside of that operating system is usually not an option.

Apart from switching to a language (and corresponding UI toolkit) that runs in a virtual machine (such as *JavaFX* or Apache Flex [46]), the developer may opt for a native library that helps them write their UI in a cross-platform way. A high-level solution like Apache Cordova [47], which provides web application solutions for a variety of platforms, allows applications to use a set of "universal" standards like those defined by the W3C (i.e. *HTML5* - a platform in itself, as Mikkonen and Taivalsaari [13] passionately assert). On the lower-level side of things exist graphics APIs such as *OpenGL*, the de-facto standard API for programmable *graphics pipelines*. The middle ground includes native toolkits such as Qt and wxWidgets, which aim to make platform toolkits equally accessible through a common API and a compiled language such as C++.

Native cross-platform toolkits like Qt are typically built with the platform compiler, against the libraries included in the platform's software development kit (*SDK*). The toolkit provides an abstract API (so that platform-specific code can be minimized) in a compiled native library (possibly, with bindings for a higher-level interpreted language as well). This has the advantage that users of the toolkit need not learn the platform's API in order to write applications for that platform. Assuming that the abstraction layer is lightweight (as is typical when using native code), the cost of an extra layer of indirection should have a negligible effect on performance. In consequence, combining the efficiency of C/C++ with the portability of a one-size-fits-all abstraction layer will help promote portability in UI code (after all, the STL itself is an abstraction layer across vendor implementations).

Bishop and Horspool [6] claim that cross-platform development is "a software engineering problem, but not a well-known one" – a statement made, ironically, when frameworks like GTK+, Motif [48], Qt, and even Wine [49] had been beating the cross-platform drum for over a decade, as Babcock

points out [14]. The point of contention lies not within the availability of such solutions, but within the solutions' fundamental approach to the problem: these cross-platform frameworks, seemingly contradictorily, subscribe to a platform-specific approach to their programming models. Frameworks like these have been based on the idea of "wrapping" native API calls with a lowest-common-denominator approach, rather than building up the framework around the idea that the interface description should be made to endure interpretation by potentially multiple frameworks.

In more recently examples, both *Sailfish OS* and *Blackberry 10* – operating systems with user interfaces which happen to be built with Qt – have opted to construct their own UI enhancements outside of the Qt community, making them API-incompatible with each other and other Qt-based UI components like Qt Quick Controls [210]. This, perhaps, is no different than the wide range of Qt-based interfaces offered by *KDE*, giving source and license incompatibilities to the point that they are of little use outside the KDE ecosystem. It would seem that, even in a larger community which thrives off a common core, fragmentation continues to propagate in the higher levels of the framework. Bishop and Harspool suggest that a solution is available through a top-down approach, whereby the interfaces (particularly the graphical variety) are defined first (in their example, via an XML-based schema) and interpreted by any compatible framework (in their case, by using *reflection* to produce the resulting objects), without a need for changing the schema's source. While most frameworks are moving toward using high-level user interface languages, none of these traditional C++ frameworks attempt to support a universal, *cross-toolkit*, UI description language as suggested by the researchers. For Qt, all it can do is provide a consistent API for such controls (as it has it done with its Qt Widgets [209] module, and is continuing with Qt Quick Controls), and hoping that Qt-based platforms will align with this API as they extend it.

2.1.3 Paradigm shift in cross-platform GUIs

By the mid 1990s, as Cusumano and Yoffie [15] note, two approaches became increasingly clear in cross-platform work: either use a virtualized environment, where code is built for an ideal virtual machine (which, of course, makes the system-specific system calls "under the hood"), or use a native toolkit which compiles against the system libraries and performs its native calls directly. By the late 1990s, marked by the success of HTML as a declarative medium, some saw a need for an even higher level of abstraction - in effect, a universal user interface description language. Standardization attempts, not limited to the now ubiquitous HTML and XML specifications, marked the beginning of a declarative trend in UI framework development. Abrams' UIML proposal [7] (first introduced in 1997, with standardization attempts in 2001), for instance, demonstrates a growing need for the "distillation" of user interface descriptions to a human-readable document which is easily tooled and interpreted across platforms. While this design pattern finds itself in modern frameworks – even if few have reached the status of being "standard" (apart from, perhaps, web technologies) – the language itself still does not dictate how these interfaces should be rendered cross-platform: whether through *wrapping* native calls or emulating native controls through direct *painting*. From a language perspective, declarative UI does not care how it is drawn (or it is drawn at all). Nonetheless, there is clearly a shift into a declaratively, directly-painted "canvas" approach to UI – as Qt Quick, Apache Flex, JavaFX, Microsoft's *XAML*, and HTML5 can all attest.

The story of combining a native programming with supported tooling is echoed by many frameworks. One such example is provided by Cusumano's and Yoffie's depiction of Netscape's early browser work: the frustration of the Java virtual machine led Netscape engineers to "abandon Java in favor of C and C++" in 1998. To this day, the Mozilla browser continues to build its user interfaces on a foundation of C/C++. Not one to shirk the value of fashion, Mozilla has further adopted a declarative interface language, XUL [50], on top of this native base. Ohrt and Turau [3] show that others echo this sentiment, noting that Microsoft's XAML and Adobe's *MXML* were developed around the same time. This paradigm – a native compiled base combined with a high-level interpreted UI language – thrives because UI needs to operate in user time. As user time is variable and generally not performance critical (as long as it delivers a fluid user experience), it can make concessions in efficiency in exchange for increases in flexibility and portability. Perhaps more importantly, declarative UI allows for the separation of presentation logic and business logic, allowing developers with different expertise areas to work collaboratively while working in languages that are efficient for their goals. This decoupling lends itself to fast prototyping (via less compilation and higher-level building blocks), even to the point of UI code being generated and edited via a graphical design tools.

Examining this shift, Corral et al. [16] go as far as to state that "mobile web development tools will be preferred by designers and programmers thanks to their versatility, economy and usefulness, less dependent on specific platforms and SDKs". Certainly, the challenges of tracking a native SDK is a burden that framework developers have, and which developers using the framework may struggle with, as indicated by Humayoun et al. [17] in a case study of three cross-platform mobile frameworks. Even so, these trends shifts can be seen beyond hybrid web frameworks, as Hui et al. [18] notes, with "cross-platform" (using the native SDK with an abstract API, such as Qt) and "interpreted" (using a virtual machine or language runtime, such as the Java-based XMLVM [55]) solutions exhibiting similar approaches. For example, a declarative user interface toolkit by Hanus and Kluß [19], based on Curry [56] (a language based on Haskell [57]), was constructed with a syntax that semantically divides the user interface description's structure, layout, and function into separate language elements: these elements having direct analogs to HTML, CSS, and JavaScript events, respectively. Similarly, the Qt Modeling Language (QML [211]) retains a highly-readable hierarchy of elements to describe the UI's structure, with *property bindings* to map relationships between the elements and with imperative code blocks (in JavaScript, or calls to C++) to provide application logic. In effect, the fundamental principles of UI development found in of web frameworks are also in use by non-web frameworks, showing that this paradigm shift is not limited to HTML5, and that alternative offerings have no reason not to remain competitive with web technologies going forward.

In the terminology used by Babcock, frameworks tend to work by one of two approaches: either by "wrapping" a native UI library's components, or by "emulating" it by drawing the components directly with a graphics API. To illustrate further, Ohrt and Turau [3] refer to wrapped components as "native" elements, and emulated components (regardless of whether the native style is emulated) as "custom" elements. The forerunners in the field (wxWidgets and Qt) used the term *widgets* to describe their wrapped native controls, and the concept of widget has accordingly been associated with such a drawing paradigm. When looking at more recent examples, the widget approach has

fallen by the wayside in favor of innovating upon a "blank canvas". In a survey of nine mobile development frameworks examined by Letner et al. [1], four tools chose an exclusively canvas-based approach to UI element rendering, three took a hybrid approach (using both native and custom components), and only two frameworks used an entirely native approach. As the number of platforms increases (and hence the number of different native component sets), one might expect that it becomes more effective to implement a framework's painting architecture on top of a generic drawing API rather than providing an abstraction of every native component available on each platform. In summary, three driving factors can be observed within this movement:

- As the number of platforms increases, an abstract drawing API is easier to maintain than an abstract widget library. This may even be true when the platform attempts to emulate the native style itself using pure drawing commands, as at least all the styles use the same drawing API.
- Cross-platform drawing libraries like OpenGL already exist, and are widely adopted. An embedded platform may not have a natural "native" widget set to begin with, so standardizing on the drawing library becomes a more appropriate solution.
- Web technologies have affected how we think about UI, as they demand flexibility across screen sizes and while having less concern for platform *look-and-feel*.

2.1.4 Feature parity and open-source

Painting, of course, is only one piece of the multi-platform puzzle. Cross-platform frameworks must abstract a wide array of other issues, such as window management, input devices, networking, file I/O, sensors (such as GPS, accelerometer, and compass), and native databases (e.g. a user's contacts). In a survey of five cross-platform products intended for mobile development by Palmieri et al. [2], it was found that no single framework abstracted every native API that the researchers examined. From a completeness point of view, there may always be trade-offs in what is offered by the framework, forcing the developer to dive into platform-specific native code when needed. Having the freedom to do so, while having the option to drop into platform-specific code when needed, can also be seen as a valuable attribute when choosing the multi-platform toolkit.

Missing features (e.g. the abstraction hides some control that the native API provides), or bugs in the toolkit's implementation (especially differences in cross-platform behavior) may end up being more pertinent issues in practice. These concerns are often mitigated by the availability of the toolkit's source code. Frameworks which are free and open source software *FOSS* allow developers to modify the toolkit itself if necessary, as well as allowing these changes to be fed back to the framework's community. Having the source code available can help to instill confidence in the framework (by allowing auditing and verification), allow community members to fix bugs themselves, and even *require*, under certain circumstances, that code modifications remain open-source when the work is licensed under *copyleft* terms like those in the GNU Public License (*GPL*). In effect, open-source methods can be a powerful tool in keeping a lively active community around the framework.

The constraints of open-source are well acknowledged in a case study by Gary et al. [8], where researchers found open-source "requires participation in a community, and that decisions are made as part of the community". They further explain that "credibility is 'earned' through participation for individuals, institutions, and companies alike", a notion not unlike Qt Project's openly governed

'meritocracy'. While these constraints may become burdensome for small companies trying to control their own open-source project (as was the conclusion in the study), having a larger community of individuals and institutions (as seen in Qt) can result in a lively exchange of ideas and contributions. Digia claims that Qt has a "thriving community of 500,000 developers [59]", an estimate based on the frequency of SDK downloads from the Qt Project website. Qt 5.2, for instance, was downloaded over one million times [60] within the first four months of its release. As a more practical statistic, the Qt Project developer network (which consists of an on-line forum and wiki) has nearly 35,000 registered users, with over 5,000 having been active within the past three months [61]. A key advantage to open-source communities – in addition to getting "free" bug fixes and code development – is the possibility for increased visibility and market position. A large and active community fosters continued investment into the framework, perpetuating the product's development.

2.1.5 Weighing the options

Framework	Licensing model	Approach	Programming Language	UI/Scripting Language	IDE	Desktop platforms	Mobile platforms	Embedded platforms
Adobe Integrated Runtime (Air)	Proprietary runtime, FOSS Flex components (MPL)	Virtual machine	ActionScript	MXML	FlashBuilder	Mac OS X, Windows	Android, iOS	--
Appcelerator Titanium	FOSS (Apache v2)	Virtual machine, wrapped native widgets	JavaScript		Titanium Studio	--	Android, Blackberry, iOS, Tizen	--
Cordova	FOSS (Apache v2)	Web Runtime	JavaScript	HTML5	Aptana	Mac OS X, Windows, Ubuntu, and any other platform which supports Qt Webkit	Android, Bada, Blackberry, Firefox OS, iOS, Tizen, WebOS, Windows Phone (7 & 8)	Linux (via Qt)
LiveCode	FOSS (GPL), commercial option	Virtual machine, wrapped native widgets	LiveCode		LiveCode IDE	Linux, Mac OS X, Windows	Android, iOS	--
Marmalade	Tiered commercial pricing, proprietary	Native, OpenGL	C++	Marmalade Quick (Lua)	Marmalade Studio	Mac OS X, Windows	Android, Blackberry, iOS, Tizen, Windows Phone	LG Smart TV (WebOS)
MoSync	FOSS (Apache v2)	Native, Web Runtime	C++, JavaScript	HTML5	MoSync IDE	--	Android, iOS, Windows Phone	--
Qt 5	FOSS (GPL/LGPL), commercial options	Native OpenGL, wrapped widgets, web view on some platforms	C++	C++, QML, JavaScript	Qt Creator	Mac OS X, Unixes (including BSD, Linux), Windows	Android, Blackberry, iOS, Sailfish, [Windows Runtime, Windows Phone 8]	Linux, QNX, VxWorks, Windows CE
Rhodes	FOSS (MIT)	Virtual machine, web view	Ruby	HTML5	RhoMobile Suite	Windows	Android, iOS, Windows Mobile, Windows Phone 8	Windows CE
Xamarin	Tiered commercial pricing, proprietary	CLI	C#	None (platform-specific)	Xamarin IDE, Visual Studio	--	Android, iOS, Windows Runtime, Windows Phone 8	--

Table 1: a comparison of mobile-oriented cross-platform frameworks, as gathered from two studies: Orht and Turau [3] and Heitkötter et al [20].

As consumer attention shifts toward mobile operating systems, fragmentation across software marketplaces is perpetuated. In most of these cases, the classically native programming option is not even the primary toolkit choice (Android, for instance, uses Java as its primary SDK, with native toolkit support as a secondary option). Because of this, developers are (by default) faced with creating platform-specific applications if they use a platform's primary UI toolkit, a phenomenon which has lead to an assortment of incomplete solutions, just as in desktop operating systems. Based on the topics already discussed – painting style, programming language, licensing options, and platform compatibility – nine frameworks have been collected to paint a landscape of offerings (Table 1). What can be seen in this table is that Windows Runtime is not only lacking an

implementation in many frameworks, it is also a good fit for Qt's cross-platform portfolio, particularly the requirements of a native paint engine with C++ language support.

Cross-platform libraries are big business - especially in the mobile segment - as framework ventures can gain revenue from licensing, cloud services, consulting, or tooling. While a software project's platform choice will certainly not be limited to only these factors, these do help to give a picture of the developer experience trends which can be seen from the frameworks. Some notable trends can be observed, such as:

- Every framework supports both iOS and Android, suggesting that these are the essential mobile platforms. Interestingly, over half of these support some form of desktop development in addition. Windows Phone, despite being one of the newest mobile OSes, has support from over half of these frameworks as well.
- There is a fairly even split between UI paradigms: two frameworks rely on wrapped native widgets (Titanium, LiveCode), three use primarily HTML5 (Rhodes, Cordova, MoSync), two frameworks use a custom canvas-based approach (Air, Marmalade), and one framework has boldly decided not to perform any UI abstraction at all (Xamarin). While Qt supports natively wrapped widgets and HTML5 via Qt Webkit, its primary UI platform is Qt Quick, which falls into the canvas-based approach.
- Virtual machine languages are trendy: two-thirds of frameworks use such a language, while the remainder all have an alternative scripting language to augment the native C++ option. Again, Qt has followed the trend by supporting JavaScript inside QML.
- Free and open source licensing appears advantageous to most frameworks, with only two frameworks being completely proprietary. Perhaps surprisingly, there is only one framework (LiveCode) with a strictly copyleft license (GPL), while others have more permissive options. Like Qt, LiveCode has a commercial licensing option available for applications which cannot adhere to the more restrictive legal requirements of the GPL.
- Every framework has an IDE optimized for use with the framework, suggesting that proper tooling is an important factor for developers.

What can be taken from this comparison is that there are many competing factors in toolkit choice, and that Qt is certainly not alone in the cross-platform UI effort. While Qt has good platform diversity (especially in embedded), it still lags behind in modern mobile adoption (given that over half of the competing frameworks have already adopted Windows Phone support). Compared to the alternatives, it stands its ground by integrating directly with the platform's native SDK and toolchain, providing a high-level declarative UI technology backed by OpenGL, and boasting a supporting IDE (which itself is written with Qt). The foundation for WinRT is already there - only a few bricks and a bit of mortar are required to build up Qt (and Qt Quick) into a usable toolkit for use on these devices. The next sections identify these building blocks, allowing for a blueprint of action points to bring the new platform up to speed.

2.2 Modern UI: an overview

"It's not about adornments. It's about typography, color, motion. That's the pixel."

—Sam Moreau, Microsoft Design

Through Microsoft Design's creation and application of the "Metro" design language, a new set of user experiences for Windows 8 devices were forged. As Moreau describes [51], Metro is about modern design (clean, minimalist), international typographic style (clear, focused, direct), and motion design (cinematographic fluidity). Through the exporation of these principles comes the largest series of changes to the Windows user experience since the move from Windows 3.1 to Windows 95. Along with a new visual face, Windows 8 brings a new interface for programmers to tap into the Metro (Modern UI) experience.

2.2.1 Enter the grid

The Modern UI environment is aims to be a clean, grid-driven paradigm for a new set of devices and use cases. From the user perspective, Windows 8 might be perceived as a hybrid of two quite different environments: the traditional, familiar desktop sits under a cover of the new Start menu and Modern UI experience. These two worlds are split in such a way that the desktop itself feels like its own application within the Modern UI: a stranger in a world driven by full-screen, touch-friendly views. The new Start menu - a total redesign of the nested menu which debuted in Windows 95 and continued through Windows 7 - is modeled after the mobile "home screen" archetype, filling the screen as opposed to being overlaid on the desktop.

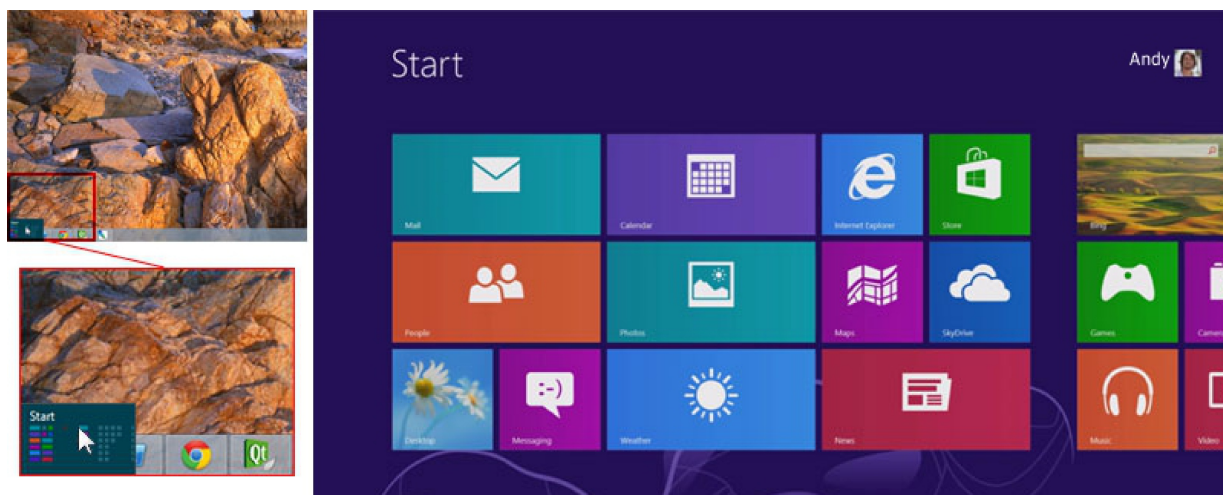


Figure 2: Going from the desktop to the Start menu via a "hot corner" (left, detail), and the fullscreen Start menu (right) with live tiles in Windows 8.

In being the main landing view for the user, the Start menu is composed of a grid of "live tiles" (a concept introduced in Windows Phone 7), which are rectangular containers for icons, information, and other application-specific content. While desktop users can effectively ignore the Modern UI and use the desktop as they would on previous versions of Windows, the new Start menu itself cannot be ignored, as it prescribes the Modern UI experience to the user as the more natural environment. Consequently, only Modern UI-ready applications are available in the Windows Store,

a centralized software marketplace for trusted applications. Windows Phone 8 - being a purely handheld OS - has no traditional desktop, but shares a similar grid of live tiles for its homescreen, as well as the same simplicity of design, clear typography, and view transition fluidity - as well as access to the Windows Store for its software.

2.2.2 Managing the desktop

A desktop window manager (*DWM*) is responsible for positioning, sizing, and graphical composition of windows within the desktop environment; key examples being Microsoft Windows, *X11* (used on Linux/Unix variants), and Mac OS X. The hallmark of the Modern UI window manager is the how little windows are actually managed. On PCs and tablets, applications are by default full-screen (on phone, all applications are full-screen, although the user may switch between running applications by holding the back button). The user may then resize the application's width to take up part of the screen, allowing multiple apps to share horizontal space on the display. Gestures from the left allow other applications to be brought into focus or "docked" in view, and splitters between the applications allow for redistribution of available space. A gesture from the right side of the screen brings up the "Charms" bar overlay for access to system and application settings, and a gesture from the top or left can be used to move a window's docked position (or, when the window is dragged downward, to close the application).

This "sliding door" approach to window management is not necessarily a new idea, considering that space distribution in window managers was a prevailing research topic in PC user experience of the 1980s (as, for example, the 1986 constraint-based approach by Cohen et al. [21] shows). By 1988, Meyers [4] had constructed a taxonomy of window management which identified that a paradigm split between "tiled" and "overlapping" window managers had emerged. While the desktop metaphor (with windows acting like overlapping papers or photos on a work desk) is much more widely used by contemporary systems, the handful of actively-developed tiled window managers for *X11* (such as awesome [52], Matchbox [53], and *xmonad* [54]) show that demand for tiled windowing systems, with their screen-use maximizing qualities, still exists.

The Modern UI undeniably falls into the tiled category, as its vertically-filled and horizontally adjacent views do not overlap unless they are being moved into place from a side gesture. While Windows 7 already enabled different snapping of windows to half of the screen for multitasking purposes (reducing the visual clutter of an overlapped window scheme), Windows 8 distills this snapping system to its essentials: the possibilities of window resizing and moving are limited to a splitter between top-level windows, with the added ability to drop a window to the left or right of the screen. The shift away from traditional, overlapped and composited windows can be seen as a nod to the classical, tiled approach of DWMs of nearly three decades prior, while at the same time a move toward future-proofing the operating system for a world of enigmatic devices which teeter between the classical desktop workhorse and the humanist handheld.

The Modern UI form of docking window management, though simple, can provide various types of multi-tasking not currently possible on purely fullscreen window managers like those found on Android and iOS. For example, drag-and-drop between applications is now possible, because multiple applications can be seen at a time. Similarly, docking an application (e.g. instant

messaging, video chat, or email) to the side can keep it visible while work inside another application continues. According to Shibata and Omura [22], docked window management can increase productivity in multitasking operations by keeping important application components on the screen at all times, while visually separating them to maintain the user's mental model of the application as a "toolbox" of many compartments. While the traditional multi-tasking desktop easily accomplishes that (contingent on the user actually using the docking features of the DWM), the Modern UI distills this into a limited set of multi-tasking scenarios. The goal (and advantage) is reduced visual clutter (with no extraneous *window chrome*), less wasted space (no desktop showing "through the cracks"), and more precise size expectations. Applications can be made to maintain flexible layouts, but with the assumption on a minimum height (matching the screen's height) and width (320 pixels, but configurable to larger size if needed).

Given enough constraints of the window manager, a certain window management *style* might even be suggested (if not enforced). A dichotomy of management styles for desktop users, as described by Stegman et al. [23] (based on earlier research by Kang and Stasko [24]), categorizes multi-tasking users as follows: "toggler", who prefer their apps to remain fullscreen and quickly switch between them (e.g. by using the Alt+Tab key combination), and "resizers", who prefer to use switch between overlapping, non-maximized windows (possibly resizing them when needed). The Modern UI caters to "toggler" in that it hides distracting content (such as the Charms bar) by default, as well as providing only the minimal application chrome necessary. With the flick from the left, it gives fast application switching to a touch screen (as well as mouse) gesture. On the other hand, windows can still be docked and resized, while saving "resizer" users extra presses by eliminating overlap and vertical sizing. Additionally, apps can be moved from one side of the screen to the other, displacing other apps on the screen and again saving the user from additional presses required to reposition the other windows around manually.

Certainly, this is a simplified approach when compared to traditional DWMs, but there are advantages to this simplicity. Applications, even when running on PCs, can effectively be treated like mobile applications when designing the interaction and layout. This is because application space always takes the height of the screen, and is comprised of a single, top-level window. Size change handlers are needed for the sliding window manager, but this is little more effort than handling both portrait and landscape on a smartphone. Touch-sized interactive areas tend to have a larger physical size than those designed for the mouse cursor, but this allows for a higher mouse speed (and less movement of the hand when pointing). In other words, the Modern UI tailors itself to tablet users, while leaving itself compatible with control via mouse and keyboard.

2.2.3 Auxiliary controls

Given that window decorations (or "*chrome*") are scarce within the Modern UI, a few common controls seem to be left out of the picture - the top-level, context-sensitive utility windows like dialogs, popup menus, and tooltips. These well-known paradigms are not absent from the simplified Modern UI, but simply more structured and generalized: a replacement for each of them exists, and is done in such a way to ensure a common user experience across applications:

- Message dialogs are modal overlays upon the application, limited to text and command

buttons. Compare this to the traditional message dialog which is typically a movable, modal window. The major departure is that the same dialog style used by message dialogs is often used by other floating windows in a desktop app - such is not the case in the Modern UI, where more interactive dialogs should be built directly into the application. An example of this is the Settings Charm, which is recommended for use as a top-level entry point to an application's settings, providing a consistent location across applications.

- Live tile updates aren't part of the application per se, but an extension of the application which provides additional feedback to the user. One could draw some parallels to the system tray icon, notification area icons which are typically used when the application is running in the background. Such a control allows the application to relay state information visually, while also providing the ability to display unobtrusive messages and quick access to the application's full UI. "Toasts", temporary notifications which can be pushed to the screen by a background application (e.g. an incoming email or phone call) and be compared with the alert bubble of such a tray icon, while live tile updates can be compared to the changed icon of an application in the system tray.
- The context-sensitive popup menu has not changed greatly between the Modern UI and what is expected from desktop. However, the user experience enforcement is in place here as well – Modern popup menus are limited to six items and support no submenus. The principle of simplification continues here, with generalization toward supporting mobile use cases.

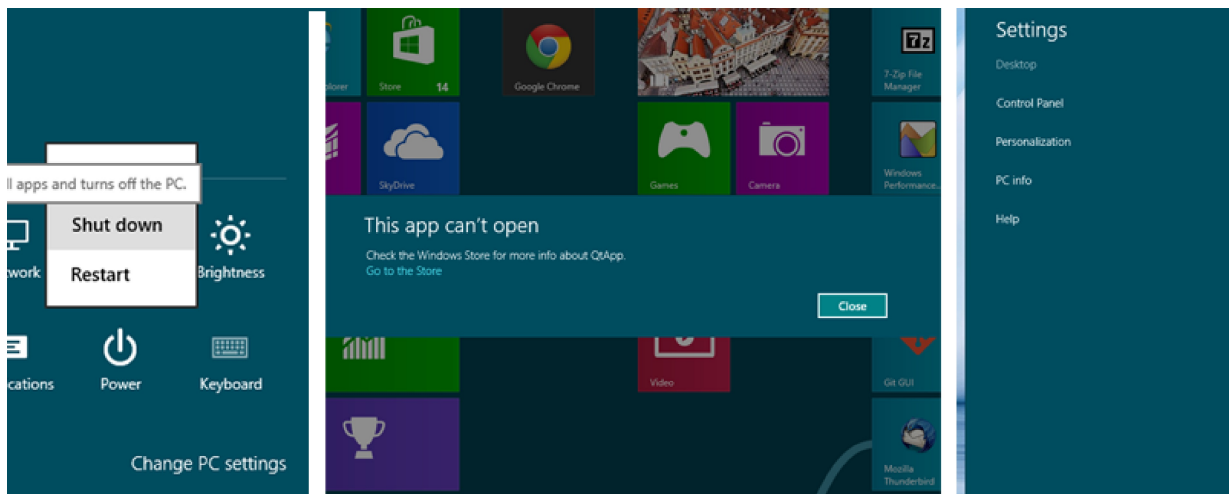


Figure 3: New integration points. Left: Popup (context) menu. Middle: Modal dialog. Right: Settings pane.

The Modern UI consolidates many application-level user interface controls into centralized, universal controls overlaid on the application itself. While this is a trend seen on mobile operating systems, it is a rather new set of integration points for the conventional desktop. Combined with a sliding window manager and a grid-based homescreen, the Modern UI is a hybrid of desktop and mobile paradigms, allowing it to cater to both categories of devices. Going forward, it will be important for Qt to interface with these integration points in ways which are useful and meaningful to the programmer.

2.3 The constraints of integration

Given the constraints of the window manager and other native UI discussed in the previous section, this section looks at the practical implications of the Windows Runtime API and how it fits in with Qt Platform Abstraction (QPA [212]). QPA, as the name implies, is a system for abstracting platform differences so that platform specifics within Qt can be concentrated into modular plugins.

2.3.1 A new set of interfaces

For application developers, the three user environments (Desktop, Modern, Phone) have various levels of access to the WinRT API. While desktop development continues to use the existing Win32 API [162], much of the WinRT API can also be used in desktop applications. Modern UI apps are given access primarily to the WinRT API in addition to a selection of "safe" interfaces within the existing Win32 API set. Windows Phone applications have access to most of the same interfaces as Modern UI apps, as well as a few additional APIs which only make sense in the context of a smartphone operating system (the *Windows Phone Runtime*). Given the overlap in API availability, some parts of an application may be written for all three environments, using the same source code. The bulk of user interface APIs is the same on Modern UI and Windows Phone, making most UI code compatible between those platforms.

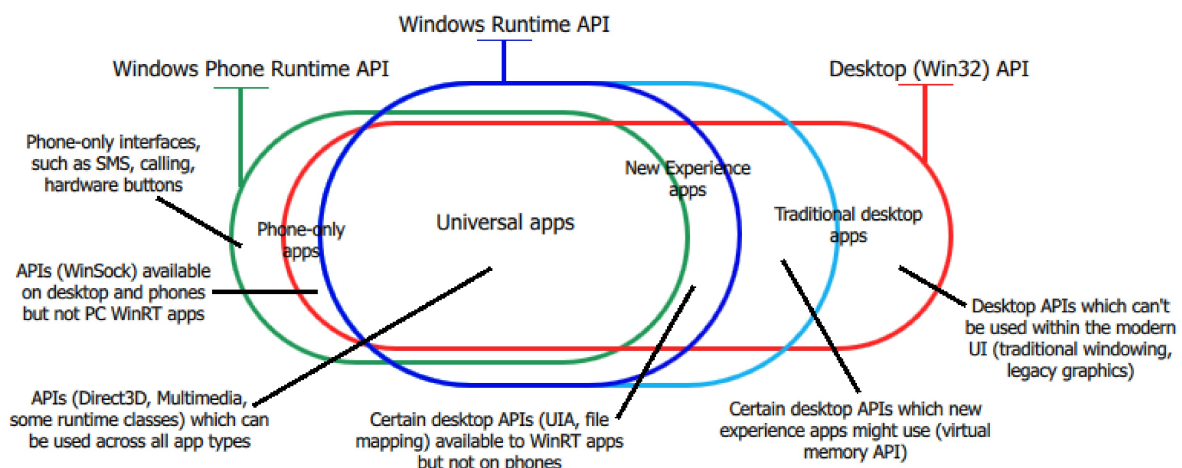


Figure 4: Overlap between Desktop (Win32), Windows Runtime, and Windows Phone Runtime API sets. The "new experience" concept is included for completeness.

Given that desktop Windows is already a well-covered platform for Qt, the WinRT port of Qt is quite sensibly limited to the Modern UI environment, and not intended to be used from the Windows Desktop. It is not inconceivable, though, that future backends for e.g. multimedia or networking (where the WinRT API may also be used in desktop applications), might be written cross-environment. It is also worth noting that a third, hybrid option (called the "new experience" [62]) is a possibility offered as an olive branch to web browser vendors; allowing a browser to run in the Modern UI without dropping all the affordances of Win32. While this might also be considered a target environment as well, adoption of this type of application is expected to be marginal: although both Google Chrome [63] and Mozilla Firefox [64] were earlier adopters of the approach, Mozilla discontinued development on the Modern UI version of their browser two years later [65]. New experience applications are not eligible for the Windows Store, either, as they use APIs which are not sandbox-safe.

2.3.2 Working with the runtime

As might be expected, given Microsoft's history with .NET (backing languages like C# and Visual Basic), the Windows Runtime API is designed to be language agnostic: the three chief offerings being C++, C#, and JavaScript. Because of this, WinRT is not a pure C API like its predecessor Win32. Rather, the C++ interface – following Microsoft's traditional Component Object Model (COM) – is generated from the interface description language (IDL) of the corresponding Windows Metadata (*WinMD*), an abstract interface description shared between language bindings. To aid in this task, the Windows Runtime Template Library (WRL [163]) can be used to help manage memory and cast between types in lieu of higher-level language bindings with features like garbage collection.

The WRL and COM WinRT bindings should feel comfortable for those familiar with Win32 and STL, but it is worth noting that MSDN only documents the C++/CX variant of C++ (enabled by the compiler's `/ZW` flag [164]). By utilizing WinMD, the C++/CX language bindings can offer the same functionality as the COM classes via a more elegant, polymorphic API (COM does not use the standard C++ inheritance model, and casts must typically be done via `QueryInterface` calls). Syntax-wise, C++/CX does depart from C++ in significant ways; it appears almost identical to C++/CLI, Microsoft's existing language for interfacing managed (.NET) types in C++. Through these extensions, the goal is to provide a "flavor" of C++ in which the developer can be less concerned about mundane details like memory management and COM typecasting, provided by automatic reference counting of WinRT types through the use of (non-standard) smart pointers and an exception-driven programming model with WinRT return types (as opposed to COM's HRESULT-based [165], out-parameter API).

While CX may ease C++ development when compared to the traditional COM, it has the challenge of being syntactically incompatible with existing compilers, syntax highlighters, and code editors. And while this could be worked around in Qt by tucking CX code into private implementations (the compiler allows CX to be freely mixed with standard C++), Qt as a library is designed to be exception-free, so trying to wrap C++ exceptions in all library code could quickly counteract the code savings provided by the CX extensions over the HRESULT checking of COM. The Qt Project tends to avoid these types of vendor-lockin scenarios when possible, and it was decided between the Qt for WinRT developers that these CX extensions would not be used in Qt source [66], and that the WRL would be used extensively. This choice was considered by some members of the Qt community as not going far enough to be Microsoft-independent: developers from the VLC project [97] have expressed their disappointment, as free toolchains like MinGW have yet to adopt the WRL or a provide an alternative for it. While it might be possible to avoid the WRL altogether (perhaps, by adding some internal smart pointers to take its place), using it within Qt appears to be the most sustainable solution. In any case, developers can still use these CX extensions or third-party toolchains in conjunction with pre-compiled Qt libraries if they prefer, as Qt's use of the WRL does not affect binary compatibility (and disabling the `/ZW` flag ensures binary compatibility with standard C++).

2.3.3 Interfacing with native UI

When examining the range of window managers which Qt has been ported, a dichotomy emerges between the complex DWM plugins and the simple embedded and mobile plugins. While DWMs provide APIs for sizing and positioning windows – and generally do all composition internally – mobile and embedded platforms may provide only a simple fullscreen surface and no geometry manager at all. This constraint served as a major development driver within Qt for Embedded Linux: to provide a windowing system (the Qt Windowing System, QWS [213]) where none was provided. After Nokia's acquisition of Trolltech in 2008, the maintenance burden of having two more of their own integrations to maintain – Symbian OS and Embedded Linux (*Maemo/Meego*) – led to increased focus on platform abstraction. It was this challenge which eventually led to a revamp in the entire porting strategy of Qt, and the development of QPA (and still well-known by its codename, "Lighthouse" [67]). Unlike QWS, QPA itself is not responsible for graphical composition of windows; it is only an access layer to an abstract windowing system. In general, it provides a path for Qt applications to draw to the device's screen (or even an offscreen surface). This gives the platform implementor the freedom to provide only the needed entry points (e.g. framebuffer drawing and input handling) while leaving other portions (e.g. window decorations or platform theming) unimplemented if desired. Due to these minimalist requirements, the task of porting Qt to a new platform tends to be much easier as compared to its predecessor, QWS.

As discussed in section 2.2.2, the Windows 8 Modern UI does not use a DWM with traditional window geometry; rather, it uses a tiled approach, whereby application windows cannot be layered or composed atop one another, and they always have the same height as the screen they are running on. The single, top-level window for WinRT applications suggests a simpler implementation for the platform integrator: sizing Qt windows becomes trivial (they are always the size of the native window) and no compositing is done. This simplicity works to Qt's advantage, as the QPA plugin can be expected to deal with fewer window geometry and compositioning concerns. Beyond window management, there are still matters to consider, such as input event mapping, hardware-accelerated graphics support, and native desktop "services" such as clipboard and URL support.

Input handling

Once a platform integration plugin can create native windows (and hopefully paint upon them), interaction support can be added. QPA handles this by providing a platform-dependent layer for which to translate and queue events into the Qt event loop. The problem of abstracting input events is not a new one. Consider Linux, which has several competing APIs for functional user-mode event access. This is because on the lowest level, the events may be accessible via kernel interfaces – event devices, essentially local sockets – which can be read from using a specific protocol. The tedium associated with this low-level approach has caused middleware projects such as *mtdev* [68] and *libinput* [69] to be developed; providing higher-level abstraction for various Linux input event types. We can see parallels to this in the WinRT API, as it drops much of the cruft of earlier designs and take on something higher-level and more object-oriented than found in Win32. All pointer events, for example, originate from the same event type (whether they come from a mouse, pen, or touchscreen), and are based on asynchronous event listeners with full-fledged C++ objects containing the event arguments (as opposed to raw C structures or control codes found in low-level event systems). These "ready-made" events promise as a thinner layer of "glue" between the native event system and the translated Qt events.

OpenGL adaptation layer

Given the importance of OpenGL in Qt, another core objective of QPA is to provide an access layer to it. While the drawing library itself is standardized and widely implemented, the process of creating a context within which to use it has historically been platform-specific. In the past, the use of libraries such as GLEW [70] has been popular to smooth out differences between platforms; Qt solves this problem by abstracting placing these access points in QPA, so that the developer generally does not need to deal with them directly. The simplest implementations tend to be done through *EGL*, the Khronos standard OpenGL access layer. For EGL usage, the developer generally only needs an object representing the native window, and possibly the display. While the setup to obtain these native handles can be complex, the passing of these handles to EGL is standard and trivial. By hiding all this initialization code into the QPA plugin, Qt can operate on the principle that the plugin is capable of initializing a drawing surface for which Qt can perform its OpenGL duties.

Other native UI

Any integration point which lives outside of the application's client area is eligible for integration with Qt as well. Native controls which live inside the client area are more difficult to commit to, though, as they require more intricate weaving between Qt's own rendering technologies and those of the platform's.

By integration the additional controls discussed in section 2.2.3, a more native look and feel can be provided by Qt. Context and system menus, for example, are typically defined by the operating system (and not necessarily painted by Qt). As previously stated, frameworks might "wrap" these native controls (Titanium), "emulate" the control by painting directly (Marmalade), while others embrace the idea of writing native platform UI directly instead (Xamarin). Native look-and-feel on WinRT is provided by its XAML component set, with Pivot [169] controls, GridView [170] layouts, and the lower CommandBar [171] being notable examples. Indeed, many of the additional integration points mentioned in the previous section have C++ APIs and can be integrated with the QPA plugin, but how they are integrated visually really depends on the control. If the control can be faithfully emulated within Qt's paint routines and painted within Qt's canvas, this is generally a good approach. On the other hand, controls that can live outside the client area and overlaid upon the application (such as context menus, dialogs, and Charms), should use the native API. Where a Qt API exist, an abstraction tends to already be made, while for features which don't translate well to other platforms - such as live tiles and Charms - can be placed in a platform-specific support library like Qt Windows Extras [214].

A roadmap for integration

Given this background on how the WinRT API works, and as well as how Qt's abstraction layer serves Qt applications, it should be possible to connect these native integration points to the existing QPA architecture in an elegant and predictable way.

2.4 Requirements for a complete port

From the understanding of cross-platform abstraction, the experience gained with Qt on Metro, and armed with an understanding of how Qt integrates with the native WinRT interfaces, a strategy for bringing Qt to WinRT emerges. In order to get a high degree of functionality – including support for the core modules and Qt Quick 2 – there five key areas are addressed:

- Modify the build system for compiling Qt itself, as well as Qt applications. This includes any platform-specific manifest files and packaging.
- Identify disallowed Win32 APIs used within the desktop Windows port and replace them with comparable WinRT APIs.
- Create a Qt platform abstraction (QPA) plugin to drive the Qt event loop, integrate with the graphics subsystem, and deliver user input to Qt applications.
- Provide solutions for missing middleware such as OpenGL.
- Adjust the tooling, such as the Qt Creator IDE, to help provide a "standard" Qt developer experience on the new platform.

By addressing these five requirements, a basic blueprint for completing the Qt for WinRT case study can be seen. The roadblocks for running Qt Quick applications on the new platform can be lifted, and application developers can begin to use Qt as a cross-platform solution on WinRT devices.

2.4.1 Tweaking the build system

Outside the Qt library codebase itself, there is considerable build system code which must be adjusted when a new toolchain is introduced. Even though C++ compilers and linkers may be considered standards-compliant, there is no universal front-end for invoking them. To borrow a statement from Wojtczyk and Knoll [5] (who prepared an API abstraction of camera capture libraries across the three major desktop environments), a platform-independent project "often already fails at the beginning of the toolchain – the build system or the source code project management". The argument stands that, while much source code is expected to build across a variety of toolchains and easily linked with associated standard libraries, the build systems themselves may not be inherently cross-platform, leading to fundamental structural issues from the beginning of the project. While the authors were discussing CMake [71] – a cross-build system makefile generator notably used in Qt's influential partner project KDE – Qt certainly has been tasked to provide good build system support itself, and strives to do so via qmake [215].

Before building the core Qt modules, there is a bootstrap process to provide a minimal configuration of the QtCore library; enough to build qmake and corresponding host tools to complete the rest of the build. The tool that starts this "bootstrapping" on Windows, `configure.exe`, is an essential element which requires modification with each new toolchain which is added to Qt. Certainly, qmake itself shares the same problem when it comes to adding support for new build targets, and can be expected to require changes as well. Perhaps the biggest difference to desktop Windows compilation is that Qt for WinRT must *always* be cross-compiled (that is, the resulting binaries are built for different platform than the one they were built in). While not very common, the practice of cross-compiling binaries on Windows has been in use for years (e.g. by the Windows CE port), so some precedent to the issue can be expected.

2.4.2 Playing nicely in the sandbox

While QPA does cover most aspects of window management and user input, it does not deal with other cross-platform challenges such as file I/O or networking. Much of Qt's codebase lives within private implementations (*PIMPLs*) which fall outside the administration of QPA. One of the reasons for this is that QPA is only used for GUI applications, while Qt supports non-GUI applications as well; hence, non-visual operations such as file I/O are not abstracted on the same level as the GUI portions of the port.

Naturally, the "base platform" for WinRT is Windows - much like Linux is the base platform for Android and Mac OS X is the base platform for iOS. In other words, the base platform already provides most of the platform-dependent codepaths; the extended platform is essentially an adjustment to this. Using Windows as a base, the Win32 PIMPLs provide a solid foundation for these implementations, but it is to be expected that some of this implementation must be rewritten for WinRT. As a result, the basic procedure of working through the core portions of Qt involves the following:

- Define a global platform macro (i.e. `Q_OS_WINRT`) for use in conditional compilation. `Q_OS_WIN` acts as the parent define, being defined as it is for all Windows platforms. Additional conditions for Windows Phone can be handled with `Q_OS_WINPHONE`.
- Find references to Win32 APIs that are not supported using WinRT. This can be done simply by attempting to compile Qt using the Windows 8 SDK. The SDK provides a macro, `WINAPI_FAMILY`, which defines which APIs are allowed for which particular Windows platforms. WinRT applications may set this to `WINAPI_FAMILY_APP`, which hides all unsupported APIs from the headers and results in compilation errors when they are used.
- When possible, find a reasonable equivalent for the Win32 API. When not possible, mark the Qt API as unimplemented.
- Test the functionality once everything can be compiled. Eventually, run and pass Qt unit tests on these new implementations.

Another well-known challenge to framework developers has been WinRT's removal of access to the Windows virtual memory APIs [166]. These APIs allow an application to allocate memory which can be marked for execution. Executable memory can then be populated with generated machine code – such as code emitted by a just-in-time (*JIT*) compiler – and executed. Access to this system feature is crucial for providing good performance in interpreted languages like JavaScript. This is relevant, because Qt has had support for evaluating JavaScript statements since the introduction of Qt Script in Qt 4.3 [72], and JavaScript forms the auxiliary scripting language of QML. Having an embedded JavaScript engine allows programmers to extend their applications with runtime dynamic expressions: Qt properties, signals, and slots could now be bound together in ways which are not restricted (or evaluated) at compile time. With the release of Qt 5.0, Google's V8 JavaScript engine [73] (used by projects like Chromium [74] and node.js [75]) shipped as the JavaScript engine in use within Qt. Applying workarounds for use of these JIT compilation techniques is crucial for any sandboxed platform, including WinRT.

2.4.3 Addressing the issue of OpenGL

As pointed out earlier, OpenGL is the de-facto standard graphics API for programmable graphics hardware, and a hard requirement for Qt Quick. While controversial, the lack of OpenGL support should probably come as no surprise, as Microsoft supports its own hardware graphics API, Direct3D. On Windows, Direct3D has historically held better support from graphics chip vendors compared to OpenGL, largely due to the importance of Windows in the PC gaming industry (Microsoft itself being one of the biggest publishers). Despite embedded and mobile developers being generally more adept to using OpenGL (largely in its ES 2 flavor, due to the wide availability of mobile GPUs supporting this technology), it has been suggested by a study of open-source tools for game programming [25] that "currently developers are more familiar with Direct3D, but the ability to use OpenGL across such a wide variety of devices and not just Microsoft platforms helps mitigate this limiting factor". While Windows has its own OpenGL layer for vendors to implement drivers, OpenGL becomes a "legacy graphics [167]" API in Windows 8. This no doubt has a polarizing effect, with application developers moving exclusively to the better-supported Direct3D API (ignoring OpenGL) or in the other direction, away from Windows altogether. With the lack of OpenGL on WinRT, Direct3D is the only choice for hardware-accelerated 3D graphics there - whether through direct use or via a wrapper library.

While it has no direct support for WinRT out of the box, a gateway to running Qt Quick successfully within the Modern UI environment exists: the Almost Native Graphics Layer Engine (ANGLE [58]). ANGLE is an open-source project, authored by Google and several collaborating companies, which provides an *OpenGL ES 2* implementation running on top of Direct3D 9 (and more recently, Direct3D 11). As the principle authors (Koch and Daniels) explain [26], ANGLE was originally developed to improve graphics acceleration support in Google's Chromium web engine, particularly for Windows machines with adequate Direct3D support in contrast with poor or buggy OpenGL drivers. This translates to better graphics performance in the web browser (i.e. for WebGL applications), while also providing an OpenGL ES implementation for applications which integrate ANGLE into their products. Qt, for example, adopted ANGLE as an option for its Windows port to improve the out-of-the-box experience with Qt Quick 2 [76]. Similarly to Google Chrome and WebGL, the use of ANGLE enables use of Qt Quick on Windows machines which lack proper OpenGL support.

Besides mapping OpenGL calls to Direct3D, the difference in window management can also affect the implementation of EGL. EGL is an abstraction of the platform windowing system and can be used with OpenGL-based technologies to create drawing contexts for native display surfaces. Just as an OpenGL ES chipset vendor would do, ANGLE provides its own implementation of EGL as an access layer to its version of the OpenGL ES 2 library. Within EGL, native types (e.g. window handles) are mapped to platform-independent EGL types. With this mapping in place, calls made through the EGL API are directed to platform-dependent private implementations, providing a cross-platform API for initializing the windowing system for use with OpenGL.

2.4.4 Improving tooling

Qt is more than a library - it is a collection of libraries and supporting tools. Because of this, simply porting Qt to a new platform is not enough to keep developers happy; the platform toolchain should be integrated with the common Qt Creator workflow. That is, developers should be able to write, launch, and debug their WinRT applications from the Qt IDE. Qt Creator aids in the development and debugging of Qt applications while also supporting general-purpose code editing and project management. Project files can be visualized and edited, Qt tools (e.g. Qt Linguist) can be invoked, the toolchain can be used to configure and compile projects, and the resulting application binaries can be interactively launched and debugged. Qt Creator also includes excellent code highlighting, navigation, and auto-completion facilities; particularly in the case of Qt-based technologies like QML. It also hosts a plugin architecture, allowing it to be extended with varied functionality: from version control integration to remote device deployment to code beautification tools.

In addition to the crucial aspects of code editing and debugging, the IDE is also responsible for interacting with the build system for proper packaging and deployment. With sandboxed applications, packaging schemes become critical when compared to traditional desktop environments where files can be shipped to nearly any directory (assuming proper permissions), the environment can be modified, and third-party libraries such as Qt can be deployed system-wide. As these capabilities are limited or unavailable on platforms like WinRT, the IDE should also support some form of packaging tooling as well as an automated process for deploying/installing these generated packages.

One ongoing goal for Qt on Windows is to provide a convenient method for developers to use the native IDE (Visual Studio) when working with Qt projects on Windows. As qmake can generate Visual Studio project files which can then be opened and used within Visual Studio, development can take place within that IDE as well. As a short-term goal, WinRT projects should be able to open in Visual Studio so that they can be properly deployed and debugged. When it comes to Qt technologies such as QML, the feature parity of Visual Studio is never likely to stay in step with Qt Creator. In order to deliver a "standard" Qt experience, the long-term developer experience goals should be that of full Qt Creator integration.

Method

3.1 Addressing core issues

One of the first steps to evaluating the state of any C++ source code is to compile (and link) it. For large projects, this usually requires installing dependencies and utilizing a configuration system to prepare the build process. Qt is no different; it uses its own build system based on a configuration bootstrap utility and the Qt *makefile* generator, *qmake*. Given its size and diversity (over 6.7 million lines of source code from 866 contributors [77]), building Qt on a new platform is bound to reveal missing (or changed) APIs, unavailable libraries, and even compiler incompatibilities. Conveniently for the case of WinRT, the native build system is the same as is already in use for the traditional Windows targets, so makefiles generated by *qmake* can be used still be used with *nmake* and the Microsoft Visual Studio Compiler (MSVC [174]).

Few external dependencies are required for Qt, and they are typically shipped along with Qt inside the `3rdparty` directory (allowing them to be patched as necessary). Most dependencies can be disabled at configure time if they pose compatibility issues, possibly resulting in the loss of certain features within Qt. Also useful - as compared to Qt 4 - is Qt 5's improved modularity, making it simpler to compile parts of Qt independently of others. Through the flexibility of the build system to disable unneeded features and the ability to use existing codepaths inherited from desktop Windows, the number of WinRT-specific adjustments can be minimized. This section describes the most crucial changes required within the build system and core libraries to get Qt bootstrapped and compiling for WinRT.

3.1.1 Build system changes

WinRT is not simply one platform, but a collection of five operating systems and architectures: Windows 8 x86 32-bit, Windows 8 x86 64-bit, Windows RT ARM, Windows Phone ARM, and Windows Phone x86 (Emulator). This number doubles when both compiler versions (MSVC 2012 and MSVC 2013) are considered. Each of these environments comes with its own variant of the compiler and link libraries. To deal practically with all variations, a make specification – the *mkspec* – was introduced for each compilation target [111]. The *mkspec* informs the Qt build system about the necessary paths and executables for not only building Qt applications, but also Qt itself. These *mkspecs* were written with a common inherited base, allowing settings to be shared wherever possible.

Tuning the host tools

While the *mkspec* controls most build system variables, a number of build tool changes were also required. Many of these changes were contributed by community members, notably Kamil Trzciński [78], during the project's early stages. Modifications to the configuration tool, *configure.exe*, allowed for certain parts of Qt to be left out of the compilation process, resulting in fewer dependencies in exchange for potentially missing features. For example, Qt's SQLite [79] plugin needed to be disabled on Windows Phone [129] due to missing memory mapping APIs [124]. These of types of workarounds can be common in the porting process, as they allow coming back to less important features after more core functionality has been established.

While the host tools such as `qmake` and `moc` [216] use the desktop (host) compilation environment, all target binaries require a clean "sub-environment" for which to build within. To solve this issue, I modified `qmake` to follow a similar path as it does for building Windows CE: the cross-compilation configuration is defined by `qmake` and written directly to the makefile, so that all the headers, libraries, and compilers for that specific target are defined by the makefile instead of the host environment [130]. This allows the developer to build Qt for any target from a standard MSVC x86 desktop command prompt, rather than launching a target-specific command prompt. It should be noted that this technique is aligned with the future direction of the Qt build system, as `qmake` will eventually be replaced by `qbs` [217], a tool which directly invokes the platform's toolchain without use of a platform-specific makefile. The changes done for `qmake` will likely find their way into `qbs` as well.

Application Manifest Generation

The packaging system for Windows Store apps is guided by an application manifest, an XML file containing various metadata about the application package. This includes both visual details, such as icons and colors, as well as required features, such as access to the network, camera, or sensor data. This packaging system, *Appx*, is based on Microsoft's public schema and the open packaging conventions (OPC [80]) and is well-suited to be edited by both the user and external tools. While Windows Phone has switched to *Appx* packaging starting with version 8.1, Windows Phone 8.0 applications use the *Xap* packaging system inherited from Windows Phone 7. *Xap* uses with a different schema and file name (`WMAppManifest.xml`) for the manifest, but is also user-editable and follows a similar structure to the *Appx* manifest. Both systems will need to be supported for a long enough period for *Xap* to be phased out of use (as Windows Phone devices get updated to 8.1, *Xap* packaging becomes obsolete).

Just as `qmake` transparently creates platform-specific makefiles, it is also responsible for creating platform-specific manifest files and packages. This helps to ensure a smooth developer experience by reducing the burden of creating packaging files manually for each platform, and to automatically select the best options based on values set in the `qmake` project file. To this end, I introduced a `qmake` feature [142] which encapsulates all the commonly used variables of both manifest schemas, and generates an output manifest file based on the values of these variables. This way, most applications work "out of the box" with the default manifest, because all of their content (executable, icons) and metadata (title, publisher, genre) is pre-populated. The developer can then modify these variables (all of which are documented [218]), or supply their own manifest template for which to have the variables replaced.

When packaging the application for installation, the manifest, executable, and all required Qt libraries are compressed into a *.ZIP* file and the *extension* is renamed to *.appx* or *.xap* (depending on the manifest type). A user can then install this package manually to their device (or local PC) using the `Add-AppxPackage` [172] PowerShell *cmdlet* or the `XapDeploy` [173] utility for Windows Phone.

3.1.2 Core library functionality

With the aforementioned changes, the build system was now functional and the compilation process could begin. Reaching a fully compilable Qt required working through the core functionality and removing, hiding, or replacing APIs which were incompatible with WinRT.

Environment Variables

Due to sandboxing and API restrictions, WinRT applications cannot access or set environment variables [168]. Therefore, it was decided that the WinRT port would simply write to/from a global in-memory map to emulate the functionality of environment variables. This is useful, as many parts of Qt change behavior based on variables within the environment; the developer is still able to change this behavior by setting the variable in code (even though it does not affect the exterior environment). While this works well for changing internal Qt behaviors, it naturally does not allow for the traditional use case of such variables, changing an application's behavior based on its environment. Developers will need to use different methods, such as a settings UI, to control such behavior in their applications.

File System

Not unlike environment variable access, apps also have limited file system access. They may only open files within their package, or, with permission, user document directories. The application may not write to any part of the system apart from its own local storage area (or, with permission from the user, to document directories). Apart from the app being aware of its location due to the path being passed at startup [116], the application tends to access all files from a relative path, and this sandbox must be kept in mind when functionalizing various parts of Qt which access the file system. This is particularly important in Qt for plugin loading, as the modularized nature of Qt requires that plugins be dynamically loaded at runtime. Several changes [110] [119] were made to make file handling act more "relative" in cases where Qt accessed files directly. This work was done with Maurice Kalinowski, as part of his investigation into the related tasks of packaging and deployment of Qt as a WinRT framework [81].

Threading

Threading is very important in Qt UIs, as the concept of a main "GUI" thread and any number of "worker" threads for non-GUI tasks prevails within the framework. The Qt Quick Scene Graph even utilizes a threaded renderer for offloading OpenGL tasks into a non-GUI thread. A full discussion of threading is outside the scope of this study, however, it is important to note that classic Win32 threads are not available in the WinRT API. Rather, WinRT introduces a number of higher-level threading primitives for use within parallel programming, as well as C++11's `std::thread`.

While it can't cover every `QThread` use case, `std::thread` was used to replace the existing Win32 threading calls [113]. Additionally, the Thread Local Storage (TLS) API was replaced with usage of the thread compiler attribute. The remaining Win32 unsupported APIs, such as the synchronization primitives `WaitForMultipleObjects` [175] and `CreateMutex` [176], could be replaced with their supported equivalents, `WaitForMultipleObjectsEx` [177] and `CreateMutexEx` [178]. In some cases, thread waiting had to be done in a more "brute force" manner using timers, though, as shown by one fix shortly before the 5.3.0 release [152]. While `std::thread` offers a mostly-functional

threading experience, it lacks features such as thread priorities, and the `Windows.System.Threading` namespace [179] could allow more complete threading support in the future. Threading on Qt for WinRT, in its current state, can be considered to be workable yet incomplete.

Networking

Like threading, a complete discussion of network on WinRT is beyond the scope of this study. Nonetheless, it is worth mentioning a few aspects of the Qt Network [219] port for WinRT, especially since Qt Quick depends on Qt Network for its network transparency support. Networking presents a rather large challenge on WinRT due to the Windows Sockets (WinSock2) API being dropped for WinRT and replaced by the `Windows.Networking` namespace [180]. Curiously, Windows Phone supports the existing WinSock2 API, although this support was also dropped for universal Windows Phone 8.1 applications. From the git history, initial commits [113] fixed up the WinSock2 codepaths for Windows Phone, while splitting up the functionality so that networking could be temporarily disabled for the rest of WinRT [114]. For a long period of time during the port's evolution, there was simply no working network support.

Given that a code split is not very practical over the long term (and that networking needs to work on PC as well), it was decided that the entire network stack be ported to WinRT's new networking APIs, allowing the WinSock2 calls to be removed altogether. As last-minute changes to TCP sockets [153] before Qt 5.3.0's release can attest, a complete QtNetwork port is a (partially complete) ongoing effort. One important missing feature is SSL support, although that might be remedied by the native `StreamSocket::UpgradeToSslAsync` [181] method in a future release. Other mobile platforms, such as iOS, face similar challenges with secure sockets, a fact which prompted one of the Qt Network maintainers to propose a minimal subset of the secure networking API for such use cases [82]. Work in that area will set the internal API for the platform-specific implementation of SSL on WinRT (hitherto, such details have been delegated to the OpenSSL library [83]).

3.1.3 Bootstrapping applications

Following the build modifications and workarounds for threading and networking, the "mundane" details of compilation were resolved. Getting to the "Hello World" moment – that is, actually *running* a Qt application – would soon be reached via a wrapping of the WinRT application container system.

Getting those famous words on the screen was not as straightforward as writing a simple `main()` function, given the COM multi-threading environment of WinRT. Qt requires that applications perform most GUI operations within the GUI thread, typically the main thread implied by the C runtime (*CRT*) entry point. This allows the application container to support multiple application states within the application lifecycle (such as a "suspended" mode), as well as enabling the application to be *activated* by several different means (such as file associations, URL schemes, or the Start menu) all from a single invocation of the CRT entry point. This also contributes to the application security model; such an application may only start within the Modern UI environment, and will terminate if it is invoked directly from the desktop. When this CRT "stub" is run, it must

create an application view *factory class* which is passed back to the system launcher for instantiation. This class factory then calls the `Run()` method, which becomes the *logical* application entry point (providing a new GUI thread). For use of Qt within this environment, the `QGuiApplication` [220] instance must be created from within this method - preferably with all of these application container details tucked neatly out of the developer's view.

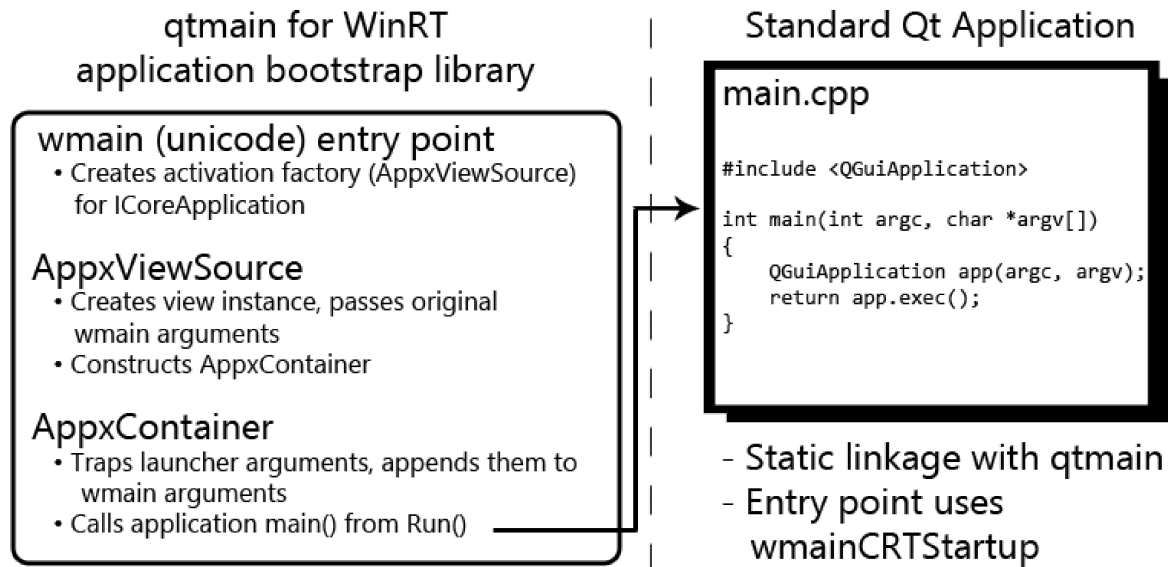


Figure 5: Design of the WinRT bootstrap to hide implementation detail from the programmer.

The process of "bootstrapping" a Qt application with a private main entry point is actually fairly common across Qt platforms - in addition to Windows, Android and Mac OS X also rely on this functionality. In Qt for Windows, this is accomplished by a static library called "qtmain", which does some initial setup before calling the `main()` entry point defined by the programmer. For WinRT, qtmain was modified to have an alternative WinRT codepath [112] which instantiates the needed WinRT classes and then passes control to the programmer's own `main()` function. Arguments from the Windows-dictated `WinMain()` function are combined with any activation parameters (e.g. from the Start Screen or a file type association) and the application's `main()` is invoked with the combined argument list. Elegantly, this allows Qt for WinRT applications to require no additional configuration or manipulation of the source code in order to operate within the application container, while programmers using Qt are able to write their main entry point as they would on any other platform.

Setup complete

The collective build system changes, compilation fixes, and basic application bootstrapping allowed Qt to compile using MSVC and the Windows 8 SDK, and even provided non-GUI applications with an entry point to run and log output to the debug console. This formed the important first milestone in bringing Qt to the WinRT platform, allowing development to move forward into the visual interaction space.

3.2 Platform abstraction

With core libraries compiling and applications bootstrapped, Qt on WinRT began to emerge as a reality. The next step – getting real applications functional within the Modern UI environment – required the most significant additions to the Qt codebase. Having the lower levels in place allowed most of the remaining details to be addressed inside Qt's platform abstraction layer, QPA. Through QPA, support for display management, input handling, and basic desktop services was brought forward.

3.2.1 Display management

From a display management perspective, QPA plugins can be split into two basic categories: those which interface with a DWM (such as Windows or X11) and those which don't (such as the *Linux framebuffer* or *EGLFS* platform plugins). A DWM is generally responsible for managing the geometry, layering, and decorations of application windows, so a non-DWM QPA plugin typically avoids this altogether. Non-DWM QPA plugins typically host applications consisting of a single, fullscreen, top-level window without window chrome. This is particularly valuable in embedded and mobile contexts, where there is not likely to be a DWM (of course, there may be some form of compositor, but applications tend to limit themselves to a single, non-resizable window). Although lightweight, the Modern UI is indeed a window manager, so the WinRT plugin should respond to changes in the application window accordingly. Even so, WinRT shares much in common with the non-DWM category of QPA plugins (perhaps even more so in the case of Windows Phone), as the operating system provides little more than a single undecorated surface for the application to draw within. Following the lead of embedded and mobile QPA plugins, it was decided that fullscreen window behavior [125] would be enforced; top-level Qt windows are always sized to match the application drawing surface size.

For Qt to be usable as a GUI toolkit, it needs a reliable and efficient way to present applications on the screen. For pure C++ applications on WinRT, developers must use one of two graphics APIs: Direct3D [183] or Direct2D [182], both which utilize the DirectX Graphics Interface (DXGI [184]). For these interfaces, a DXGI *swap chain* is created which allows the presentation of video frames to the screen. For applications doing rasterization operations on the CPU (such as Qt applications using the QPainter [223] API), the swap chain can be configured for bit-block transfers (*blitting*), allowing the application to render its contents to the screen. This technique was prototyped with Direct2D in the Qt on Metro project, and was rewritten using Direct3D in Qt 5 [134].

Raster applications (such as those using the Qt Widgets module) "flush" their changes to the framebuffer using QBackingStore API. When a portion of the screen becomes changed - or "damaged" - the backing store is updated and the window is notified of the damage. In the WinRT/Direct3D implementation, a common double-buffered "page flip" is used, whereby two hardware buffers are used to control the currently-displayed (front) image and the next-to-be-displayed (back) image. It is this back buffer which is updated when a damage event occurs, and is flipped to the screen at the next available opportunity. To further optimize flipping, non-phone devices are configured to take advantage of a Direct3D 11.1 API which flips only the damaged portions to the screen (as opposed to the entire buffer). Phone devices, on the other hand, have different graphics hardware which restricts the configuration to single hardware buffers with wholesale page flips

[131], so the hardware architecture is actually queuing frames behind the scene. With swapping support in place, raster applications could now be shown on the screen.

3.2.2 Pointing device handling

Pointing devices - i.e., mice, touchscreens, and pens - under WinRT are interesting because their events are now delivered via a unified structure in the `Windows.Devices.Input` namespace [202]. QPA requires these be sent to the application as separate event types, so the first step in the pointer handling logic is to forward the event to their respective handler. Once there, the position and status information of the pointer event is extracted, converted to Qt events and injected into the Qt event system. Mouse-like devices provide the button state (up to five mouse buttons are represented in the WinRT API) and pen-like devices give additional information such as tilt, pressure, and rotation. In the case of touchscreens (and some trackpads), touch points are tracked over time and marked as "pressed", "stationary", "moved", or "released".

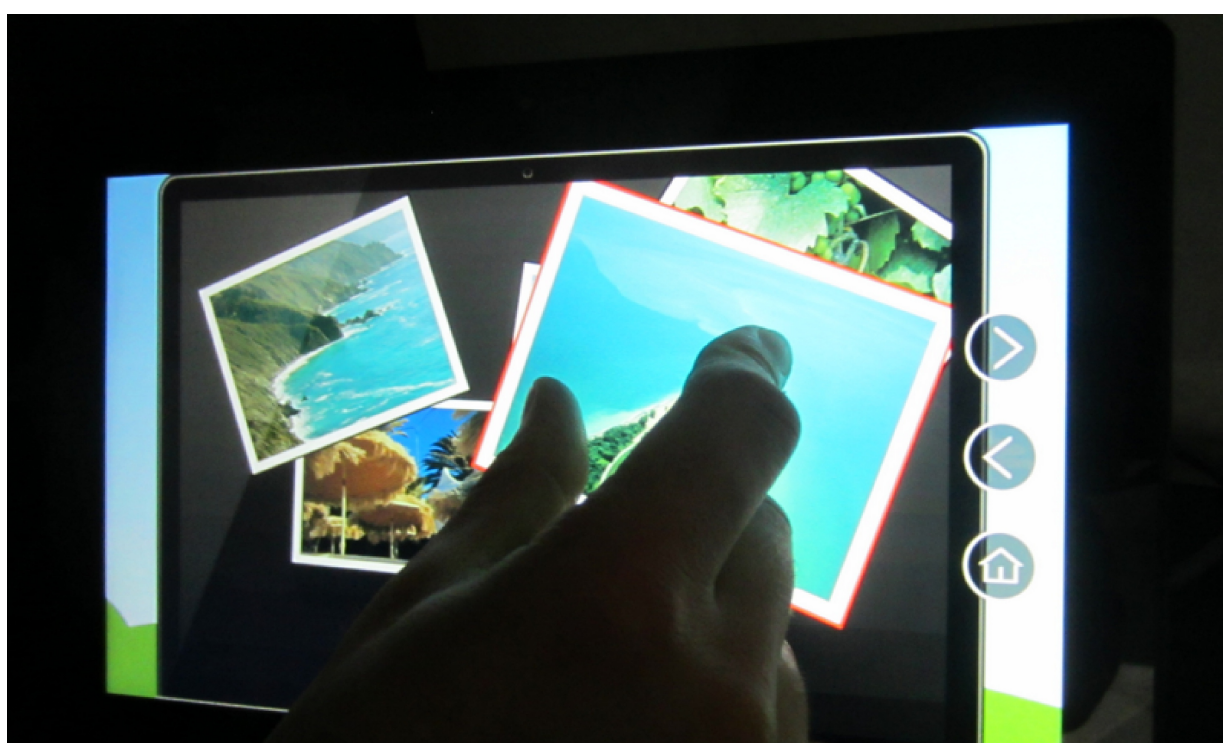


Figure 6: The "Photo Surface" Qt demo [224], shown here running on the Microsoft Surface Pro 2 tablet, utilizes multi-touch interaction enabled by the WinRT QPA plugin.

Pointer event handling proved long-winded, yet straightforward to implement [128]. One interesting caveat was discovered: on phone devices, the `isInContact` property always returns true, even if the touch point is released - that is, the software did not properly report the contact state of the touch point. Interestingly, it was discovered that the `isLeftButtonPressed` property could be used instead to provide the actual state of the touch point [138]. With pointer handling mostly in place - particularly touch handling - the ability to manipulate objects using pinch and pan gestures became possible. Accordingly, developers can use components such as `PinchArea` [221] as a high-level interface to the otherwise low-level multi-touch events provided by the system (see Figure 6).

Another requirement of pointer handling, at least for mouse events, is setting of the cursor image. Since Qt applications do their own painting, the operating system must be informed when the mouse cursor needs to change its visual identity, such as when hovering over a link or resizing a control. Implementation of this requirement [127] was straightforward, but a few limitations were discovered; while most of the cursor shapes requested by Qt are available in the WinRT API, a few notable icons are absent: split resizers (↔/↕) and panning hands (☞/☜). This isn't a critical issue in itself, as any missing cursors can be embedded into the application binary (although they may not match the user's active cursor theme). This is, however, complicated by another caveat of the WinRT cursor API: a custom cursor can only be loaded from a resource ID, not e.g., an array of bytes (as was possible in Win32). This also means that dynamic cursors are not possible within this API (that is, cursors painted into memory). A workaround to this issue would be to hide the native cursor and perform custom cursor composition within the graphics pipeline, an approach used by other platform plugins such as EGLFS. Alternatively, a new constructor could be added to the QCursor [222] API to allow for loading a cursor by resource ID. Either solution is worth considering for a future Qt release.

3.2.3 Key handling

Key handling can be a surprisingly complex task, as applications must consider variables such as different keyboard layouts, system locales, special key combinations (such as Alt-key character escapes), and input from software input methods like on-screen keyboards and handwriting recognition systems. Some Qt platform plugins, such as X11 and desktop Windows, have complex utility classes to handle the many corner cases of mapping operating system key events to Qt. A key handling scheme for WinRT was derived from the window's character event handler [120], based on native messages not only for key presses and releases, but also for characters (in cases where a keystroke generates a character). These character messages are already translated into UTF16 strings (required by QString) based on the user's key layout and locale, so no application-level translation is needed (apart from meta keys, as the corresponding control codes would otherwise send non-printable characters to the UI).

As Qt only expects to receive key events (not fully composed character events), it is the job of the platform plugin to map the incoming character events to the interleaved key events - or, in the case of "spontaneous" characters (e.g. those those coming from another application) to generate simulated key events for the character. In the end, the implementation proved simpler than other platforms, as WinRT handles keystroke translation and even filters out and translates special key combinations (such as Alt-numberpad characters) before they reach the application; perhaps, Microsoft's engineers acknowledged the complexity of the Win32 API when designing the character event system for Windows Runtime, simplifying it for the better.

To round out the key handling implementation, it was important to address the issue of software input (Figure 7), a natural requirement for a platform geared toward devices with touchscreens (many of which may not even have a hardware keyboard). QPA provides an API which allows applications to request that the touch keyboard be opened or closed, as well as querying its screen geometry, allowing applications to adjust the view of the application accordingly. For Windows Phone, a special phone-only keyboard API, `InputPane::TryShow()` [185], allows for a direct

mapping to the Qt abstraction, allowing touch keyboard visibility to map nicely to QPA [133].

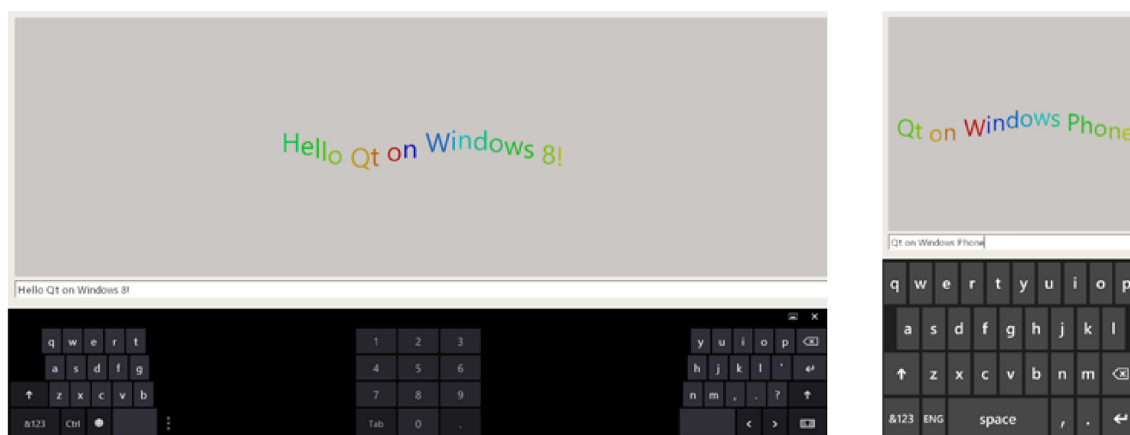



Figure 7: The "Wiggly" Qt demo [225] running under Windows 8 (left) and Windows Phone 8 (right) with touch keyboards open. The system controls the visibility of the touch keyboard on non-phone devices; a touch screen is required to be interacted with before the software input method can be displayed.

For non-phone devices, however, the situation is less optimal. If a device (such as a tablet) has a touch screen, the touch keyboard may appear only when a text input control has focus and no hardware keyboard is attached. This is done automatically by XAML and HTML controls, but cannot be done by Direct3D applications (as Qt applications are, under WinRT) without informing the operating system about text accessibility through the UI Automation API (UIA [186]). To this end, the PC/tablet implementation utilizes the UIA framework to enable opening of the virtual keyboard when a text input control is focused [120]. Though this solves the software input panel visibility problem, there is still much which could be done to support the selection and autocompletion capabilities within the accessibility framework.

A final "key handling" issue to address was the use of hardware keys on Windows Phone devices. Like Android, Windows Phone utilizes an always-visible (typically, etched into the device) hardware back button. This allows the user (after drilling down several views into an application) to back out of the application toward the home screen. To implement this, a callback was registered with the back button [148], and a *synchronous* event was queued into the event system (by default, all QPA input events are asynchronously queued). This way, the event could be flushed to the application (as a key event with the moniker `Qt::Key_Back`), allowing the application to accept or reject the event. Applications can listen for the key back event and react accordingly (such as moving backward in the application's view stack). When ignoring the event (the default behavior), the application closes and the OS brings up the home screen or the previous open application.

3.2.4 Desktop services

Currently, QPA provides integration of two "desktop" services: URL handling and file opening. URLs typically point to webpages (but may point to any resource), and files are typically documents known to (and specified by) the user. Both open with the default application associated with the URL or file format. Supporting these services was simple to implement [126], as WinRT readily supports these use cases. Even so, a curious challenge was discovered: both WinRT APIs are

asynchronous, while the QPA support for them is expected to be synchronous. To make a non-blocking API blocking, an approach borrowed from Qt's dialog system was borrowed: use a local event loop to process all GUI events until the asynchronous operation completes. Doing so allows the programmer to be informed whether the open operation succeeds, while keeping the GUI responsive during the procedure. In proper style-enforcing fashion, Windows even switches to the "working cursor" () automatically, notifying the user of a possibly long-running event.

Another "service" provided by the WinRT API (and handled by QPA) is the display orientation event. When the user changes the device's screen orientation - whether it be through the a system setting or by physically rotating the device - an event is raised which informs the application of the change. In some environments, the operating system may change the window size (swapping with & height) automatically. In other environments, such as on Window Phone, the application resolution remains the same (making it the responsibility of the application developer to take advantage of the event if desired). Quite conveniently, a one-to-one mapping was found between the WinRT API and the QPA API, resulting in a clean implementation [132].

A promising start

With the basic elements of QPA in place, the platform port started to come alive. Qt raster applications can draw to the screen, receive input from the user, and even integrate with some finer points of the operating system. With these capabilities in place, the Qt for WinRT port provided convincing evidence that a successful port was possible on the platform. Around the completion of these aspects, a technology preview of Qt for WinRT was published [39] along with the release of Qt 5.2. Not long after, Albert Timashev ported *Dream Calendar* – a mobile application built with Qt Widgets available for Android, Blackberry, and iOS – to Qt for WinRT and published it to the Windows Phone Store [84]. This independently showed that, even without Qt Quick, Qt for WinRT had already matured enough to prove useful to developers building cross-platform GUI applications.

3.3 Enabling Qt Quick

Integrating the QPA plugin meant that raster (non-OpenGL, such as Qt Widgets) applications could now be hosted within the Modern UI environment. While useful for many types of applications, Qt Widgets are not optimized for touch use, and the C++ API is not as convenient and productive as the declarative QML language. Qt Quick is certainly the lauded at the future of Qt, with a focus on a high-level declarative language (QML) which describes UI controls rendered by an efficient OpenGL scene graph. Echoing the sentiment I conveyed in an article for the *Qt Blog* [41], bringing the "magic" of Qt to the WinRT is really about bringing Qt Quick's UI technology to the platform.

The unfortunate reality for Qt Quick is that OpenGL is simply unavailable on some platforms and/or hardware – a fact which, at least superficially, also holds true for WinRT. Besides OpenGL, there is also the challenge of the QML JavaScript engine: in sandboxed environments like WinRT, scripting engines typically cannot make use of runtime code generation (JIT compilation) due to security restrictions, making JavaScript engines like V8 unavailable there.

3.3.1 JavaScript in the sandbox

As detailed in chapter 2.3, Qt 5 shipped with a JavaScript engine based on Google's V8. Unlike its spiritual predecessor JavaScriptCore [85], V8 only has one code compilation mode: JIT machine code generation. While great for performance, it utilizes APIs which can't be used within a sandbox (as allowing runtime-generated machine code tends to be a security nightmare). When initially investigating the JavaScript solution for QML, I even looked into using Microsoft's own WinJS namespace as a possible solution; after all, HTML and JavaScript are a supported toolkit within the WinRT environment. However, this soon proved to be a dead-end: the WinJS API doesn't support evaluating arbitrary expressions, or populating the JavaScript context with C++ proxy objects. Also, it would be a tremendous effort to replace all of QML's private backend with such a different beast than V8.

Fortunately, another sandboxed platform (iOS) had a similar memory API restriction, and a solution was developed by Qt's lead engineers for Qt 5.2: the V4 virtual machine [86]. Unlike V8, V4 doesn't have to fire on all cylinders: it can disable the JIT compiled codepath for a slower, yet sandbox-compatible, interpreted code path. For normal QML use cases, particularly the evaluation of property bindings, V4 promises to have comparable performance [87] even when JIT compilation is inactive. This helps to reinforce JavaScript as a syntax for use within QML, rather than a complete runtime environment for building up entire applications. C++, after all, is the primary language for use within Qt. In fact, Digia even released a compiler for QML as a commercial add-on, enabling build-time translation of QML and JavaScript directly into C++. This feature can further alleviate any concerns with runtime code generation restrictions, as well as providing comparable performance to the JIT solution.

Simply put, enabling QML and JavaScript under WinRT was a matter of switching off the JIT and switching on the interpreted codepath [139]. In addition (and like other portions of Qt), this patch also replaced some calls to Win32 functions which are unsupported under WinRT.

3.3.2 OpenGL and ANGLE

As stated in the section 2.3.3, ANGLE offers a compelling solution to provide OpenGL, given that Direct3D is the only option for hardware graphics on WinRT. ANGLE is also conveniently already used by Qt, with established configuration options and maintenance conventions as a third-party library.

Upgrade and integration

Direct3D 11 is the latest incarnation of the DirectX 3D graphics API and the required version for use on WinRT. To use ANGLE's Direct3D 11 backend, it was first necessary to upgrade the version of ANGLE used in Qt, which had grown stagnant since its initial import before the release of Qt 5.0. This upgrade landed in Qt 5.1 [121], based on the "dx11-proto" development branch of ANGLE. Since then, this "prototype" version of ANGLE became the master branch, making it easier for the Qt Project to track ANGLE upstream, and Qt's version was again updated for Qt 5.3.0 [147]. A few additional patches were submitted to resolve failing test cases [122] and crashes [123], and to make Direct3D 9 and Direct3D 11 codepaths mutually exclusive. By enabling compilation of a Direct3D 11-only version of ANGLE (via the `-angle-d3d11` configuration option), the groundwork for using ANGLE under WinRT was laid.

After the updated ANGLE integration, further WinRT-specific patches were contributed. Much like what was done to Qt Core at the beginning of the porting process, several unsupported Win32 APIs were replaced [117] with supported versions. For example, thread local storage (TLS) API usage was replaced with the threading attribute, `__declspec(thread)`. Similarly, `LocalAlloc/LocalFree` [187] dynamic memory methods were replaced with the heap API methods (`HeapAlloc/HeapFree` [188]). Finally, dynamically-loaded DLLs were changed to use direct-linking where possible (in the WinRT sandbox, system DLLs cannot be resolved dynamically).

Beyond these basic compilation fixes, a more invasive change was added to support Direct3D "feature level 9" codepaths within the Direct3D 11 renderer [135]. This change was needed because older devices and mobile hardware (tablets and phones) only support a subset of Direct3D 11 features, and the ANGLE authors had decided not to support these devices, due to challenges in "achieving good WebGL conformance" with them [88].

An example of such a challenge can be seen in Direct3D 11's dropped support for the point sprite drawing mode: rather, implementors should use other means, such as a geometry shader, to paint point sprites (this is what ANGLE uses, although Nevraev points out that there are several other solutions to the problem [89]). Feature level 9 cards do not have geometry shader support, resulting in a failure to render when using `GL_POINTS` mode. While in this case, Qt already has workarounds for its use of point sprites in Qt Quick, it may be worthwhile to explore an alternative code path to solve this issue. While these types of compatibility problems will need to be fixed if they interfere with normal functionality, their impact can be expected to lessen as older GPUs are phased out of use.

An EGL interface for WinRT

To support the WinRT windowing system, changes to ANGLE's EGL adaptation were also required. In ANGLE, the `EGLNativeWindowType` is defined as `HWND` (Win32 window handle), and the

EGLNativeDisplayType is defined as HDC (Win32 device context handle). These types are meaningless under WinRT, and instead are defined as ICoreWindow * [189] (pointer to a CoreWindow [190] class instance) and int (the default for a general identifier). In the private implementation, methods using EGLNativeDisplayType required minimal changes: essentially all display-dependent code was simply skipped using conditional compilation. This works because of the Modern UI's simplicity; it requires far less management of window geometry.

Because the native display type was defined as an int, the only meaningful value is EGL_DEFAULT_DISPLAY: this instructs the EGL implementation to access to the default platform display internally, instead of relying on a user-supplied handle. To provide a useful implementation of EGLNativeWindowType, WinRT codepaths were inserted [118] so that ANGLE EGL could deal with initialization and state tracking where appropriate. This important for initializing the Direct3D swap chain and delegating viewport geometry changes within the Modern UI. Naturally, some adjustment's to WinRT's QPA plugin (which manages the creation of platform OpenGL contexts), were made to switch from using the Direct3D-based buffer swap (described in the previous section) to a fully EGL-based mechanism following the introduction of WinRT's EGL interface [136].

3.3.3 Handling shader compilation at runtime

Even with ANGLE demonstratively working, WinRT presented another in that it required all shader binaries to be compiled prior to application packaging (this has changed in Windows 8.1 and Windows Phone 8.1, which now allow runtime compilation). While requiring shader precompilation does have some advantages – such as conserving CPU usage by not compiling the shader source at runtime – it greatly reduces developer flexibility. OpenGL applications almost exclusively employ runtime shader compilation, as the bytecode generated by the graphics driver in a compiled shader is always GPU-specific (meaning that you can't generally precompile the shader and distribute it with your application, as the target GPU is likely to be unknown). Accordingly, any developer using ANGLE will face the same problem, as will anyone using Qt Quick 2 as an application framework.

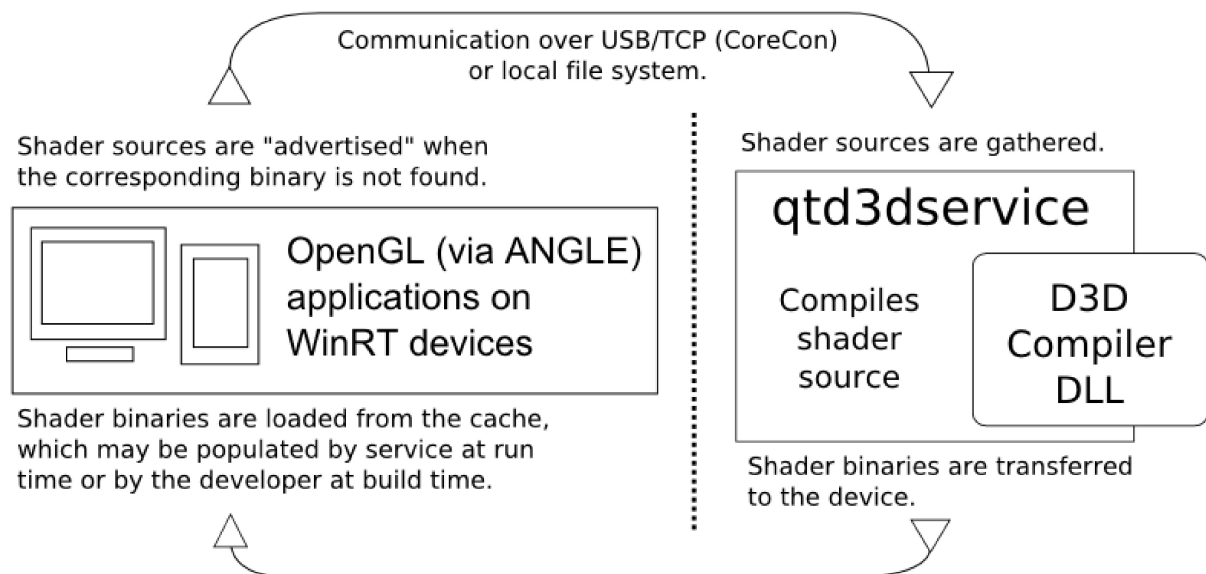


Figure 8: The mechanics of the D3D compiler service.

A solution was devised to provide runtime-compiled shader blobs to WinRT applications by using an inbox/outbox approach when a shader is compiled. When an application uses ANGLE, it loads a DLL - the DLL compiler library - which contains the routines necessary for turning High Level Shading Language source code into bytecode which can be run on the GPU. Because this library is not available on Windows Phone 8.0, or allowed by Windows 8 Store Apps, an API-compatible proxy was created [137]. This proxy, called `d3dcompiler_qt`, intercepts the shader source and "posts" it (by saving it in a designated directory) for a monitoring service to pick up. Naturally, a reference monitoring service, named `qtd3dservice`, was also introduced [141]. It is the job of `qtd3dservice` to observe the shader source, compile it using the workstation's shader compiler, and ship it back to the program which is expecting a shader binary. This binary is then cached for the program to use later, and the developer can query `qtd3dservice` for a list of compiled shaders so that they can be shipped with the application in a production release [226].

The `qtd3dservice` is mostly automated: the user only needs to start the service, which then enumerates all connected devices and monitors all running applications, compiling shaders and shipping the binaries back to the applications as needed. While still more convenient than a manual solution, it does burden the developer to run through their entire application to obtain and keep track of the shader sources generated by ANGLE. While this task is unnecessary for those targeting Windows 8.1 or Windows Phone 8.1, developers may continue to use `qtd3dservice` to take advantage of ahead-of-time compilation of shader sources.

A feeling of progress



Figure 9: Two Qt OpenGL (via ANGLE) [227] [228] demos running simultaneously (left, center) within the Modern UI, alongside a system Windows Store App (right).

With the adjustments to ANGLE, integration with WinRT's QPA plugin, and the D3D shader compiler proxy service, OpenGL ES 2 under Qt became a possibility on WinRT devices. With the addition of the V4 JavaScript engine, authoring QML applications was also made possible. With these changes in place, the principle goal of making Qt (and Qt Quick) compatible with WinRT was largely accomplished.

3.4 Sharpening the tools

"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

—Abraham H. Maslow [90]

Good cross-platform support is as much about tooling as it is libraries: beyond allowing the developer to write cross-platform code, they can also develop in a cross-platform manner by using a consistent environment regardless of their host or target platform. While Qt ships with a number of tools, the most advanced of these is Qt Creator, the Qt integrated development environment (IDE). Additionally, other IDEs (such as Visual Studio) are supported as well, with scripts and plugins allowing for integration with the lower-level build utilities Qt provides. Whether using Qt Creator or the platform IDE, it should be possible to package, launch, and debug Qt for WinRT applications in a convenient and transparent way.

3.4.1 Two paths: Qt Creator and Visual Studio

Naturally, the more the IDE supports the developer's workflow, the more likely it is to be used by the developer. At the same time, the tooling hide some of the complexity of deploying and initializing an application, so there should be a balance between automation and transparency of operations. As it can be reasoned that better tooling can lead to an increased uptake of the associated framework (leading to a more active open-source community and more commercial customers providing revenue to the project), it is important for Qt Creator to provide tooling for WinRT as well as it does for other platforms. To this end, a plugin for Qt Creator was prototyped to provide initial support for launching WinRT applications [91].

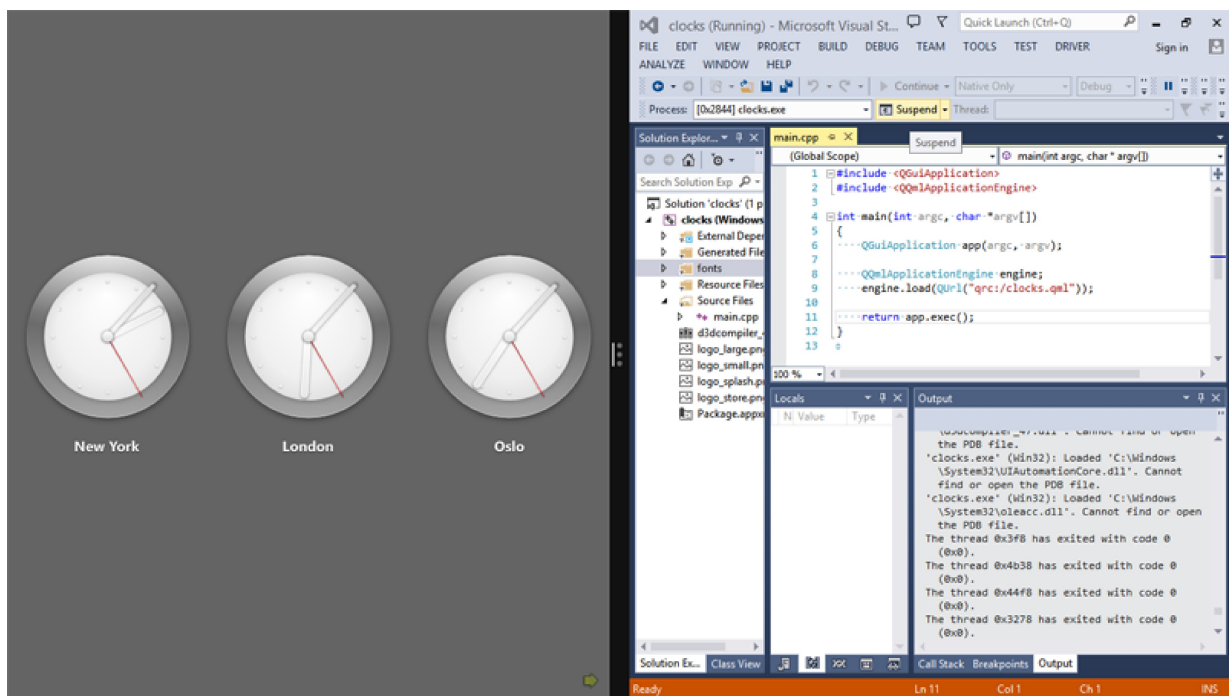


Figure 10: Two-pane debugging session: QtQuick Windows Store App (left) being debugged in Visual Studio from the desktop (right).

As this effort wained (due to focus on the port itself), patches from community members for improved Visual Studio support followed [115]. The qmake tool can generate Visual Studio project files (a capability which has been around for over a decade [92]), but this must be tweaked with each release of the Visual Studio IDE and compiler. Along with the tweaks made earlier on in the project, as well as the changes mentioned in the earlier sections regarding build system changes, a "best effort" Visual Studio experience can be obtained by simply generating a project file from qmake. When opening this project file and launching the application from Visual Studio, a full debugging session can take place. Since WinRT applications run in fullscreen, it is wise to employ a multi-monitor setup in order to see both the application and the debugger at the same time (Figure 10).

While Visual Studio will remain supported, getting the full source code autocompletion, QML highlighting and debugging, and qmake integrated tooling still requires Qt Creator. The vision of a seamless Qt developer experience is still within the Creator, not Visual Studio.

3.4.2 Creating the runner

As explained in section 2.2.1, WinRT applications cannot be executed directly, but must be started via the Windows Store launcher. There are several related developer APIs (IPackageDebugSettings [191], PackageManager [192], and the Appx Packaging API [193]) for managing packages and launching them, so that these features can be integrated into IDEs and other development tools. These were used to build the prototype plugin, allowing for a simple package management UI and ability to launch local Modern UI applications straight from the IDE. However, concerns that the plugin could only be built on Windows 8 meant that it would not be part of a Qt Creator release for the foreseeable future (as release builds must be done on Windows 7 for compatibility reasons). This meant that another approach would be necessary.

The solution to this problem came as a side effect of another, related project goal: as Qt has a comprehensive automatic unit test suite, running the tests (and fixing failures) is one path to verifying behavior, functionality, and stability. In order to get a measure of the port's maturity in this area, Maurice Kalinowski hacked together a command line utility [140], based on the existing Qt Creator WinRT plugin code. This utility allowed Qt auto-tests to be packaged, registered, started within the Modern UI. This led to a discussion of whether we could essentially move all of the WinRT plugin code into this new tool, now named winrtrunner [229], and use that as a backend for a more universal plugin. With these goals in mind, I refactored the runner into a bootstrapped Qt tool (much like moc or rcc [230]), and added support for installation, removal, launching, and monitoring of the application [143]. It now exists as a project in the qttools repository, and is shipped with release copies of Qt for WinRT.

One caveat to the package management APIs is that they only deal with local application packages, not remote package management like that needed for a Windows Phone device. In order to bring Windows Phone support to winrtrunner, a deeper investigation (including some reverse engineering) was made. By monitoring system calls made by Visual Studio, it was observed that for the emulator and phone, Core Connectivity (CoreCon [194]), a partially-documented legacy API from the Windows CE SDK, is used to facilitate communication between Windows desktop

applications and Windows CE "smart devices". CoreCon is accessible via COM, and all interfaces are registered as in-process server DLLs, so it can still be used within a custom C++ application. After re-creating these interfaces by hand (via the MSDN documentation and some IDL introspection), the facilities for installation, removal, and querying of application contents became available for use within winrtrunner [144]. By utilizing the existing command-line interface with this new backend, winrtrunner gained the ability to create XAP packages which could be remotely installed, launched, monitored, and stopped on a Windows Phone device connected over USB.

3.4.3 Integrating with the IDE

With the runner tool complete, a simplified version of the WinRT Qt Creator plugin was prepared by Jörg Bornemann [146], designed to compile on all platforms and invoke the runner tool for package installation and launch.

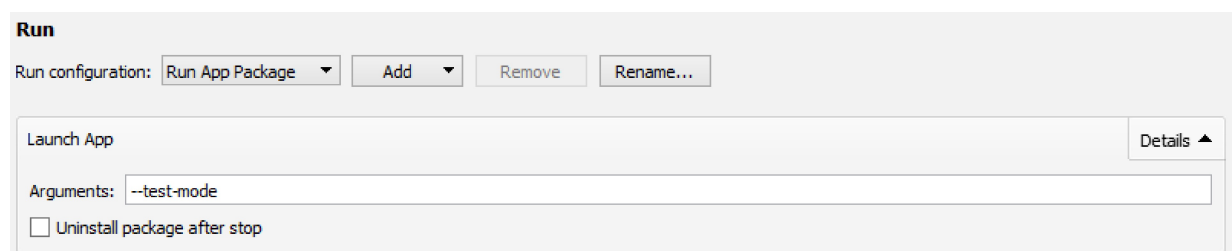


Figure 11: Qt Creator WinRT Application Launcher: a simple interface to winrtrunner providing additional arguments to be passed to the application.

Once the application is launched, the process identifier (*PID*) of the app can be passed back to the launcher. This *PID* can be used to monitor the application's lifetime and also forcefully terminate it if necessary. Most importantly, though, it can be passed to a debugger. As Qt Creator already supports the Microsoft Console Debugger (cdb [195]), this can also be used in debugging local WinRT applications. Local debugging functionality was added in the original plugin prototype, but did not make it into the refreshed plugin for Qt Creator's packaged release (3.1.1) with Qt 5.3. However, David Schulz brought back debugging support for local applications [151], to be part of Qt Creator 3.2. Remote debugging (including Windows Phone devices) is still absent from Qt Creator, and will certainly be a goal for future plugin versions. For these reasons, it was stated in an introductory article on the *Qt Blog* that developers should use Visual Studio when debugging Qt for WinRT applications [93], while Qt Creator can still be used for most other tasks.

Remote debugging insights

While debugging support is far from complete, it is still an important goal. Additional investigation into the topic of remote debugging has taken place [150], possibly leading to a complete implementation via CoreCon. Microsoft has unified their remote debugging solution for WinRT through an application called *msvsmon*, the Visual Studio Remote Debugging Monitor [196]. It is a service which runs on the client device (the debuggee) and allows a remote machine to launch interactive debugging sessions. This service runs on both Windows Phone devices, as well as Windows PCs which have the package "Remote Debugging Tools for Windows" installed. Rather than the existing remote debugging protocol used by cdb (supported by Qt Creator), *msvsmon* uses Windows Web Services (WSSAPI) to communicate over-the-wire. This may not be a complete loss,

as WSSAPI applications rely on a schema which can be implemented in a client-agnostic manner. Unfortunately, Microsoft has not made this schema public, making it difficult to interface with msvsmon if this schema is not made available. As an alternative, it may be possible to tap into the Visual Studio debugging API directly (available to Visual Studio plugin developers) to create a client which can host the debugging runtime, yet runs outside of the Visual Studio environment.

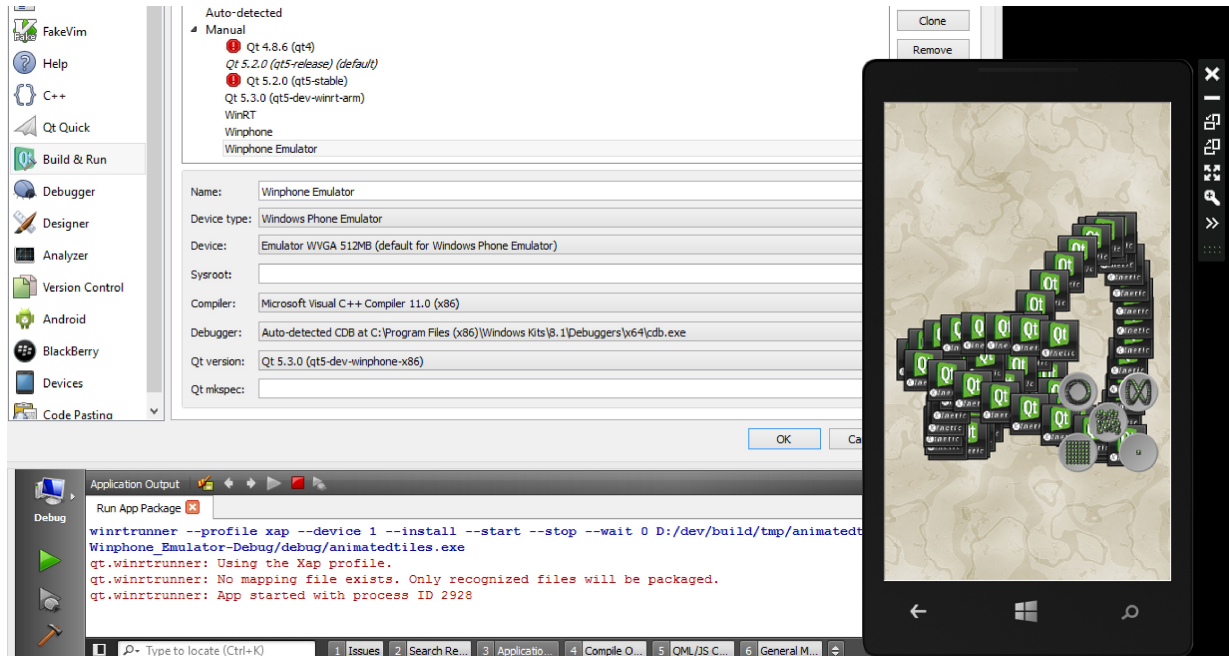


Figure 12: Qt Creator launching an application on a "remote" device, the Windows Phone Emulator [93]. In Qt Creator 3.1, a debugger cannot be attached to the application, but possibilities exist to improve the situation.

Another option would be to use cdb for remote debugging (just as it is used for local debugging). This has already been tested to work with the existing remote debugging support in Creator, but lacks proper tooling for deploying and launching apps between devices. For PCs, this feature could be remedied by using a remote control utility (such as remote.exe [197] or winrm [198]) to start and stop a remotely-installed winrtrunner, while files could be transferred using network shares. One problem to this approach is that it would not work on Windows RT targets, unless winrtrunner could be made to run as a Windows Store application, or a special policy (such as the ARM Kits policy [199]) could be installed on the device to allow self-signed code. A similar problem exists for Windows Phone, as cdb would need to be started as a remote agent [200] using the CoreCon API and winrtrunner would also need to be signed. In fact, msvsmon is deployed via CoreCon as well, so there is a precedent to this technique.

In summary, problems of remote debugging are fairly well understood, but the hurdles to overcome them may take significant effort. For the timeframe of this study, Visual Studio remains the only fully-featured debugging solution for use with Qt for WinRT.

4 Evaluation

By addressing the five key points described in section 2.4, Qt for WinRT went from zero platform support to a state where most Qt examples can be run "out of the box" from Qt Creator and Visual Studio. This was all in preparation for the port's inaugural release as a supported, Beta-level platform with Qt 5.3.

In order to take this series of changes to a full evaluation, a canonical application was chosen, *Quick Forecast* [98], to not only get running on WinRT devices, but also to be certified and published in the WinRT online marketplaces. Quick Forecast is already available from Google Play [99] and the iTunes App Store [100] as free downloads. By bringing Quick Forecast to the Windows and Windows Phone Stores, an objective evaluation of the port can be made by exploring the feasibility of taking a complex Qt application, building it for WinRT, and having it certified and published for users to download.

4.1 Porting a complex application

Quick Forecast has been used as a showcase for building cross-platform mobile UIs written with Qt Quick [103], as well as an example of Qt's localization support [104]. The application itself can be used to view weather information for a given city, display the ten-day forecast for that place, and even drill down further to view finer-grained forecast conditions throughout the day. Given that the application is known to work on popular desktops, Android, iOS, and various embedded operating systems, it was expected that porting the application to WinRT would be straightforward as well.

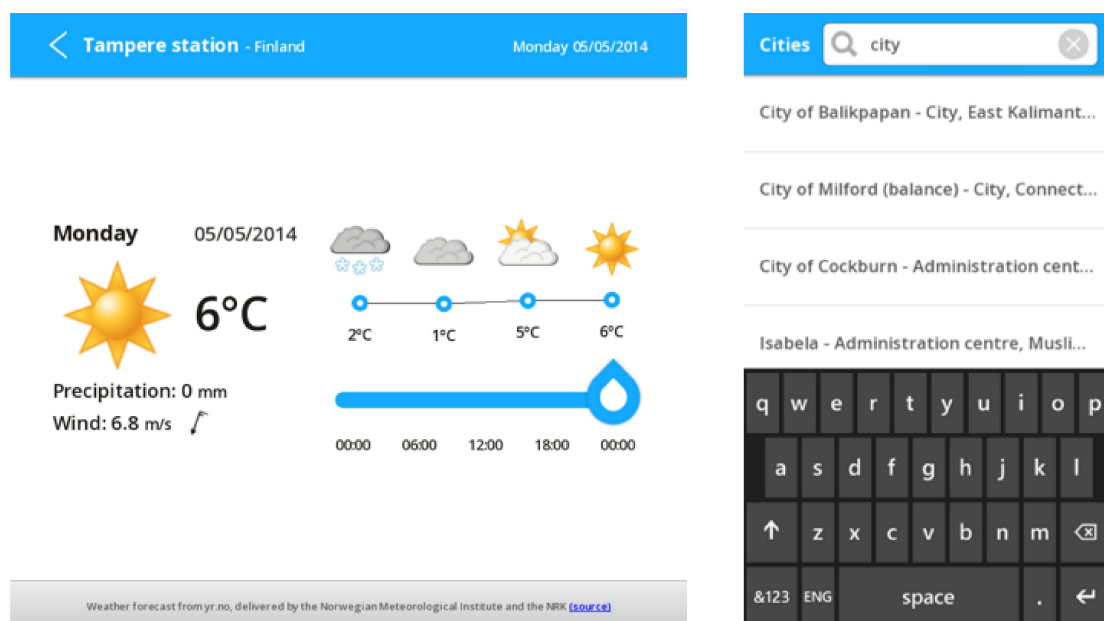


Figure 13: Views from Quick Forecast: a complete demo application demonstrating the use of QML and Qt Quick Controls across a variety of devices.

Compiling the application with Qt 5.3.0 for WinRT posed no challenges - it simply built and ran on both Windows 8 64-bit and Windows Phone 8 ARM. Running the project directly from Qt Creator 3.1 and Visual Studio 2013 were both possible as well. Overall, the application appeared to be responsive and functional. A number of minor issues were discovered (and remedied):

- As the application accessed the network, the first change needed was to add `WINRT_MANIFEST.capabilities += internetClient` to the qmake project file (without this, the application is not allowed access to the network). As this is likely to be a common problem (provided the majority of applications are indeed networked), this issue is a good indication that automatically setting this capability by would improve the default developer experience. One solution would be to simply enable this capability whenever an application declares `QT += network` in its project file.
- When a network request is made, a warning about `QAuthenticator` [231] not being available is repeatedly printed to the debug output. This is because proxy and SSL support, for the time being, have been disabled for the WinRT port (as discussed in section 3.1.1). While this warning does not affect application functionality or user experience, a fix was submitted [156] to resolve the issue.
- The application has been designed to scale text and images on small screens, leading to some artifacts (pixelation) when scaling these elements. To avoid this problem, UI scaling was disabled and some font and UI elements were resized so that all controls fit comfortably on the lowest Windows Phone resolution, 480x800 pixels. The problem was discovered to be linked to the special patches done to enable Direct3D level 9 hardware in ANGLE (as discussed in section 3.3.2), and a subsequent patch was made [160].
- After fixing the minimum resolution issue, it was discovered that higher-resolution devices where rendering in similar resolutions as the low-end devices. This is because the platform specifies the platform window in device-independent pixels (*DIPs*), which is a logical pixel designed to be consistent in physical size across devices, in contrast with a physical pixel, which may vary in size depending on screen size and resolution (pixel density). In this case, running on a Lumia 920, the scale was 160%, meaning that the physical resolution of the device would be 1.6 times the reported resolution. Patches to ANGLE [158] and the platform plugin [159] were submitted to better handle high pixel densities, effectively making Qt use the physical resolution of the device while the UI is authored in DIPs. This keeps Qt consistent with WinRT's XAML and HTML component sets, while allowing Qt Quick applications to render in the full resolution of the device.
- The application supports backstepping when pressing the hardware back button (this is a requirement for Android). Despite having implemented support for the back button (section 3.2.3), the expectations of the application were not entirely met. To fix the issue (which checked for acceptance of the *release* event instead of the *press* event), a change was introduced [154], making the backstepping work properly in the application and consistent with the Android implementation.
- As Quick Forecast supports multiple languages, language tags needed to be added to the application manifest after it was generated so that they would be visible in the store. Language support in the manifest had been an oversight when developing the package manifest feature, so a patch to support application languages was committed [157].

- After adding multi-language support to the app, each language was tested. It was found that the devanagari (Indic) and hanzi (Chinese) scripts were not part of the packaged font, (Open Sans [94]), so those glyphs did not appear when the app was run in that locale. Given that no supported font was readily available for free distribution - and that the Windows Phone 8.0 version does not allow loading of system fonts - these languages were dropped from the package before store submission. As with some of the other constraints of Windows Phone 8.0, this problem becomes a non-issue on Windows Phone 8.1, as it supports the same DirectWrite font loading strategy added for Windows 8 [149]. With that, a more appropriate font such as Nirmala [95] (for devanagari) or DengXian [201] (for hanzi) can be loaded instead.

4.2 Windows Store certification

With the initial porting issues out of the way, the certification process could begin. This process consists of a series of automated tests to check for responsiveness, API compliance, and application behavior under special conditions such as a software crash or system load. While these automated tests can be done by the developer before submission, some additional checks for security and content compliance (possibly performed by human testers) are completed upon submission to the store.

To test the application locally, the *Windows App Certification Kit* (ACK [96]) was run on the local PC. As the ACK does not support Windows Phone 8.0, the Windows Phone version of the software was evaluated with the less-intensive "Windows Store Kit" testing feature of Visual Studio 2012. The ACK takes about five minutes to run, launching and closing the application several times as the tests are performed. This is in contrast to the Windows Phone Store Kit, which only runs API and packaging tests, and only takes a few seconds.

For the phone version of the software, all automated tests in the Windows Store Kit passed. Common problems for not passing would be missing files referenced in the application package, use of debug libraries, or linking to unsupported APIs (any "out-of-the-box" Qt example should not have these problems, as the Qt libraries themselves are tested in this process). Using this as a green light for the Windows Phone Store, the package was submitted and a series of automated tests were run on Microsoft's servers. All tests passed, allowing the application to be published in the Windows Phone Store.

For the PC version of the software, the application failed one ACK test: *Direct3D Trim after Suspend*. This test fails if an application does not "trim" (deallocate unneeded graphics resources) when it is suspended (placed in the background). To solve this, a patch for ANGLE was added to listen for the application to enter the suspending state, with a call to `DXGIDevice3::Trim()` [203] inside the callback [155]. Following the fix, all certification tests passed and the PC version could also be submitted to the store.

Within the online certification process, the application passed all technical, but failed on one content compliance requirement: "*Your app must have a privacy statement if it is network-capable*" [204]. The exact requirement is that the Settings Charm should contain a link to the publisher's privacy policy. Unfortunately, no such Qt wrapper was implemented for this functionality during the porting process (such function might fit into `QMenuBar` [232], as it already has a QPA abstraction).

The problem was solved by using the WinRT C++ API directly in the application [161], and the application was resubmitted, passing certification. While developers can simply copy the provided code added to Quick Forecast (as it is provided under a permissive license), this is not the clean, long-term solution which Qt should provide, and a more dedicated solution should be done in a future release of the port.

A successful publishing

With both versions (PC and Phone) published, apps were installed to separate PCs and phones to verify that they could be properly used. In both cases, the app could be downloaded, installed, and used without issue. With versions in the Windows Phone Store [101] and the Windows Store [102], the goal of publishing Qt Quick applications in the official WinRT marketplaces was achieved.

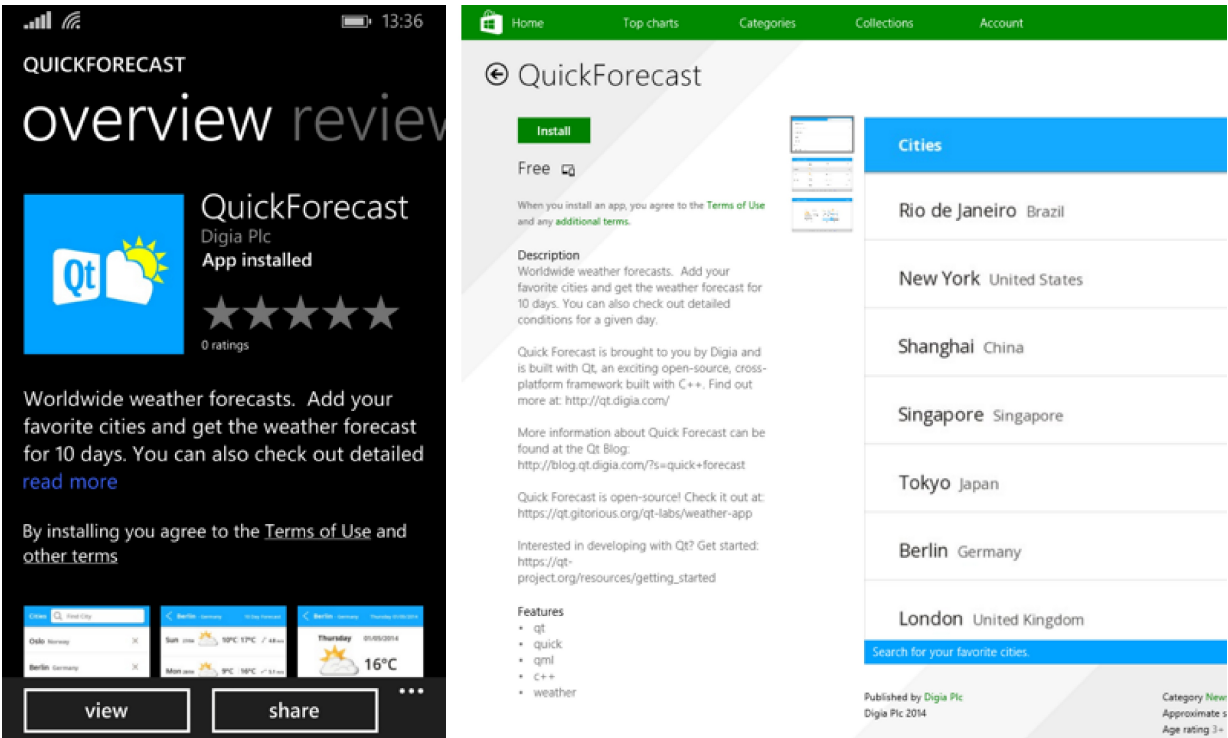


Figure 14: Screenshots of Quick Forecast from the Windows Phone Store (left) and Windows Store (right).

While not without its challenges, the successful publication of Quick Forecast proves that moderately complex applications utilizing the Qt Quick Scene Graph can be implemented and submitted to the Windows Store. From this criteria alone, the project can be deemed a success, allowing developers to consider Qt 5.3 (or a later release) for their next cross-platform project.

5 Closing remarks

Over the course of this case study, the Windows Runtime API and its window manager, the Modern UI, was analyzed and utilized in order to provide a supported implementation for it within Qt. Qt, a natively-compiled cross-platform framework with a declarative UI language (QML) and OpenGL scene graph (Qt Quick), strives to support every feasible platform target in use by modern devices. From these features, Qt can be considered a modern, canvas-driven cross-platform framework which stands competitively alongside other contemporary solutions such as Marmalade, Apache Flex, and HTML5. By bringing support for WinRT to Qt, a major contribution has been made to the open-source Qt Project, allowing developers utilizing this native, modern framework to target their applications to the Windows 8 family of operating systems.

Through porting a canonical Qt Quick demo to Windows Phone 8 and the Windows 8 Modern UI, resolving store certification issues within the Qt framework, and getting the application certified and published within the store, I believe the Qt for WinRT port can be deemed a success. While it does not yet cover all Qt modules (such as Qt Multimedia [233]), it covers enough to make fairly complex applications using Qt Widgets or Qt Quick. While the biggest challenges of bringing the Qt for WinRT port have been tackled, there are a number of other points to consider further. This section includes a set of goals for critical discussion: debugging, native look-and-feel, and additional module support. All of things need to improve in order to strengthen the port and make it competitive, laying the foundation for a truly complete offering once implemented fully.

Improved developer experience and debugging

One of the most important aspects of Qt as a cross-platform toolkit is to allow its users to "break away" from reliance on platform-specific tools and APIs. Qt Creator has done this with many other platforms: one can debug directly from Qt Creator when building iOS apps (instead of using XCode [105]); one can even deploy directly to embedded Linux devices from a Windows machine. Breaking the dependencies of certain platform environments for certain targets is key to the developer experience, and this includes both host operating systems and their native IDEs.

For WinRT, getting complete debugging and packaging support into Qt Creator should be a critical requirement going forward, as this would relieve the programmer from switching between Qt Creator and Visual Studio at development time. Although support for debugging local WinRT applications is possible in the upcoming Qt Creator release, this support has yet to reach remote deployments, such as tablet devices like the Microsoft Surface. Perhaps more importantly, Creator should have support for debugging Windows Phone devices over USB, much like it already supports deploying to them. Once this is in place, it is easier to justify having release packaging built right into Creator, because Visual Studio is no longer needed on a daily basis. Compared to the development packaging done now, release packaging would contain all the required content for Windows Store publication, and also allow the user to launch the package directly into the Windows App Certification Kit.

Apart from breaking free from Visual Studio, there is even a chance the developer could break free from Windows as a development platform. Consider that toolkits like Adobe Air and Xamarin allow developers to deploy to iOS devices from their Windows PCs - having such a universal deployment

scheme would allow a developer to target WinRT devices from a Linux environment, for example (as mentioned in section 2.3.2, developers of the VLC project have also voiced their desire for such an option). This may actually be within the realm of possibility, as LLVM's Clang for Windows project (which can produce code compatible with the MSVC ABI and libraries) has been marked as stable since version 3.4 [106]. Having a Clang cross-compiler which is configured to link to pre-built Qt binaries and the Windows SDK could allow a developer in any environment to build compatible binaries for Windows devices, and only Qt Creator would need the backend to deploy and launch the applications (which a remote debugging server running) to make this a completely free-wielding tool. All of that takes time, some work outside of Qt (to get Clang's debugging server working on Windows Phone), and likely some reverse-engineering (e.g. to get a CoreCon equivalent working within Linux). Even so, this *truly cross-platform* toolchain might be an achievable dream.

Native look and feel

Apart from a selection of integration points (several of them low-hanging fruit), the Qt for WinRT port makes no attempt to provide a native look-and-feel to the UIs it allows developers to create. This is certainly not specific to WinRT, as none of the other mobile ports have added this capability either. Consider, though, that the visual style found in the Modern UI is quite simple; it consists of clean lines and high-contrast visual elements, and most of the icons are supplied via the system typefaces. Taking steps to provide a fully native look to Qt Quick applications would round out the offering in ways not yet achieved on other mobile Qt platforms.

Qt Quick Controls has support for custom styling of all its components. Imagine controls based on the with Modern UI style: buttons would be simple rectangles with bold typography done up in the system theme colors, and they would tilt in the direction of the user's press. List views items would have iconic side blocks and high-contrast text rendered in Segoe UI [108], the quintessential Windows 8 typeface. The Qt Quick Controls TabView [235] (a tab-navigated page stack) would be transformed into the Modern UI PivotView when the WinRT style is applied. The Modern UI bottom menu and Windows Phone application menu bar would be integrated into Qt Quick Control's MenuBar control, and living happily within the scene graph with no platform API involved. To add to the emulation factor, all animation timings could come directly from WinRT's native API. All of this possible, it "simply" needs to be implemented.

Along these same lines, text input selection and editing built into Qt Quick are also a far cry from the capabilities found in WinRT's XAML controls. Integrating with the native overlays and styles for text selection handles and copy/paste iconography could go a long way in giving the native experience as well. This can be done by implementing a more complete accessibility backend, something which should be done anyway to support visually-impaired users. WinRT offers a screen reading API already, so including that into the plans could really boost Qt Quick on WinRT as an accessible user interface technology, while improving the text editing for common users as well.

Tighter integration

Some native integration points are also missing. Consider the Settings charm as discussed in the previous chapter: having a Qt API even for platform-specific features can greatly improve readability and code efficiency while reducing platform domain knowledge requirements. More charms could be added, such as the Search charm, to allow a more platform-consistent search

experience (this might be useful in Quick Forecast, even). The Devices charm is crucial in supporting multi-screen applications (via the "Project" action), and would be useful in delivering a platform experience when printing (the "Print" action). Similarly, the Share charm could provide a path into backing a Qt-level abstraction for social networking services. Tapping into these controls would certainly increase the level of immersion within Qt for WinRT applications.

Another aspect of immersion revolves around the future of the desktop Windows platform: as it moves forward, it will likely continue to embrace the WinRT API until the Modern UI and the traditional desktop are harmonized into one. As discussed in section 2.3.1, the "new experience" model allows applications to mix WinRT and Win32 to form hybrid user experiences; one practical application of this would be to create a Qt configuration in which the desktop Windows QPA plugin would be built alongside the WinRT QPA plugin, but the rest of Qt would use only WinRT-compatible APIs. In this way, a universal Qt build for both desktop and Modern UI would be possible, all while having no concern about Windows Store certification issues for apps using the WinRT QPA plugin. It would come with the advantage that developers could easily switch between the desktop plugin without recompiling the application (let alone Qt itself), saving compilation time, disk space consumption, and testing maintenance.

More module support

As already pointed out, Qt module support is incomplete. Of the "essentials" (the modules intended to be supported on every platform), both Qt Multimedia and Qt WebKit are missing. Qt Multimedia provides video and audio playback, key features for modern applications, especially on mobile entertainment devices like tablets and smartphones. Qt WebKit, however, is never likely to be supported on WinRT; beyond the fact that it isn't supported on Android or iOS either, it is being replaced by Qt WebEngine [107], which means that WebKit to WinRT would be a dubious use of effort.

Some multimedia support has already been evaluated, via a proof-of-concept QAudioOutput [236] backend I wrote for an entry in the 2014 48-hour Global Game Jam [109]. Moving from audio to video (and integration with the scene graph) will likely require more than a weekend worth of work, but the payoff will be greater as well. As the Windows Runtime API for multimedia is also available to Windows 8 desktop applications, this support could apply to a wider audience. Some sideline features, such as radio tuning and playback support, are also valuable for mobile applications and would set Qt apart from its competitors – particularly game frameworks – which only focus on the bare essentials of multimedia.

Beyond the essential modules, there is a series of additional modules – the Qt "Add-ons" – which could be beneficial to implement on WinRT as well. As mentioned above, the Devices charm supports printing integration, so Qt Print Support would make a fine addition to have. For mobile devices, the Qt NFC, Qt Bluetooth, and Qt Location modules would be valuable additions as well. A backend for Qt Sensors support – which handles onboard sensors for accelerations, magnetic field, and ambient light – has already been implemented [145], though not all sensors are yet supported. Generally, implementing support for any of these modules is expected to be mostly about wrapping the native APIs, as most Qt Addon-on backends only need platform accessors to raw data.

A positive outlook for the future

In many ways, it is astonishing to think that the former proprietor of the Qt framework (Nokia) is now part of Microsoft. Had Qt stayed at Nokia, it is unlikely that this port would have ever seen the light of day. Ironically, it is because Qt was able to remain free to continue its successful open-source position that users and developers – both from Qt and Microsoft communities – can utilize the much-invested Nokia technology on a new platform. Qt and Windows are very much alive, and it is their separation which makes them strong together.

Seeing Qt working on this new set of platforms is incredibly encouraging. Having worked alongside Nokia colleagues during the evolution of Qt on Symbian and Qt on MeeGo, I feel a sense of accomplishment seeing that torch carried to WinRT along other underdogs like Sailfish and Blackberry 10. With the leaders in mobile (Android and iOS), the big three desktop environments, and the array of embedded operating systems Qt supports, adding another platform may seem like a drop in the bucket. However, WinRT is not just a platform: it is an API and a user experience blueprint for a whole range of existing and future Windows-based devices. Qt is no newbie either, and even though they've been around along time, Windows and Qt share an enduring history that also looks to the future, and there's no reason not to look in the same direction. Adding support for the WinRT API to Qt has helped secure this position, and it is my hope that the Qt development community (and the larger community of Qt users) picks up on this exciting range of new opportunities.

Perhaps more importantly, the positive outcomes of exploring the emergence of a new platform within Qt is a testament to its architectural flexibility. Even with Qt's existing support for Windows, technical challenges appeared on many different levels with porting to WinRT. Generally though, there was already a system in place to allow for accessible platform-specific integration of each requirement. One might expect that few players in the cross-platform game, particularly newer frameworks focused primarily on iOS and Android, might not offer the degree of backend elasticity which Qt has evolved over its two-decade history. Fewer would have had the infrastructure to evaluate and integrate the large code contributions which were required for this project. In effect, even if Windows Runtime does not see great success as a platform (or as a supported Qt port), the work done in this case study can be expected to help strengthen the design and iteration of Qt's cross-platform infrastructure for the future betterment of the framework.

References

Academic

- [1] M. Letner et al, Mobile Platform Architecture Review: Android, iPhone, Qt. In: *Proc. of European Computer Aided Systems Theory* **2** (2009), 544-551.
- [2] Manuel Palmieri et al, Comparison of Cross-Platform Mobile Development Tools. In: *Proc. of 16th International Conference on Intelligence in Next Generation Networks* (2012), 179-186.
- [3] Julian Ohrt and Volker Turau, Cross-platform Development tools for smartphone applications. *Computer* **45**, 2012, 72-79.
- [4] B. Meyers. A taxonomy of window manager user interfaces, *IEEE Computer Graphics and Applications*, **8**, 1988, 65-84.
- [5] Martin Wojtczyk and Alois Knoll, A Cross Platform Development Workflow for C/C++ Applications. In: *Proc. of Third International Conference on Software Engineering Activities* (2008), 224-229.
- [6] Judith Bishop and Nigel Horspool, Cross-Platform Development: Software that Lasts. *Computer* **39**, 2006, 26-35.
- [7] M. Abrams et al, UIML: An Appliance-Independent XML User Interface Language. In: *Proc. of the Eighth International World Wide Web Conference* (1999), 617-630.
- [8] Kevin Gary et al, A Case Study: Open Source Community and the Commercial Enterprise. In: *Proc. of Sixth International Conference on Information Technology: New Generations* (2009), 940-945.
- [9] Dennis M. Christie, The Development of the C Language. In: *Proc. of Second History of Programming Languages Conference* (1993), 201-208. Available: <http://cm.bell-labs.com/who/dmr/chist.html>
- [10] Clay Dowling, Using C for CGI Programming. *Linux Journal* **132**, 2005. Available: <http://www.linuxjournal.com/article/6863>
- [11] Bjarne Stroustrup, Evolving a language in and for the real world: C++ 1991-2006. In: *Proc. of the third ACM SIGPLAN conference on History of programming languages* **4** (2007), 1-59.
- [12] Otso Kassinen et al, Guidelines for the implementation of cross-platform mobile middleware. In: *Proc. of international journal of software engineering and its applications* **4** (2010), 43-58.
- [13] Tomi Mikkonen and Antero Taivalsaari, Reports of the web's death are greatly exaggerated. *Computer* **44**, 2011, 30-36.
- [14] Michael Babcock, The importance of the GUI in cross platform development. *Linux Journal* **49**, 1998.
- [15] Michael Cusumano and David B. Yoffie, What Netscape learned from cross-platform software development. *Communications of the ACM* **42**, 1999, 72-78.
- [16] Luis Corral et al, Evolution of mobile software development from platform-specific to web-based multiplatform paradigm. In: *Onward! Proc. of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (2011), 181-183.
- [17] Shah Rukh Humayoun et al, Developing mobile apps using cross-platform frameworks: a case study. In: *Proc. of the 15th International Conference on Human-Computer Interaction* **1** (2013), 370-380.
- [18] Hui et al, Cross-platform mobile applications for Android and iOS. Presented: *The Sixth Joint IFIP Wireless and Mobile Networking Conference* (2013).
- [19] Michael Hanus and Christof Kluß, Declarative programming of user interfaces. In: *Proc. of the Eleventh International Symposium of Practical Aspects of Declarative Languages* (2009), 16-30.

- [20] Henning Heitkötter et al, Evaluating cross-platform development approaches for mobile applications. In: *8th International Conference of Web Information Systems and Technologies* **140** (2012), 120-138.
- [21] E.S. Cohen et al, Constraint-based tiled windows. *IEEE Computer Graphics and Applications* **6**, 1986, 35-45.
- [22] H. Shibata and K. Omura, Docking window framework: supporting multitasking by docking windows. In: *Proc. of Asia Pacific Conference on Computer Human Interaction* (2012), 227-236.
- [23] Alex Stegman et al, A comparison between single and dual monitor productivity and the effects of window management styles on performance. In: *Proc. of Human Computer Interaction International* **2** (2011), 84-93.
- [24] Y. Kang and J. Stasko, Lightweight task/application performance using single versus multiple monitors: a comparative study. In: *Proc. of the Graphics Interface* (2008), 17-24.
- [25] Ruairi Fahy and Liam Krewer, Using open source libraries in cross platform games development. In: *Proc. of 2012 IEEE International Games Innovation Conference* (2012), 1-5.
- [26] Daniel Koch and Nicolas Capens, The ANGLE Project: Implementing OpenGL ES 2.0 on Direct3D. In: Patrick Cozzi and Christophe Riccio (eds.), *OpenGL Insights*, CRC Press, 2012, 543-570. Available: <http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-ANGLE.pdf>

Popular

This section contains citations from news articles, blog posts, white papers, software publications, and other "popular" literary sources.

- [27] wxWidgets History. Internet: <http://www.wxwidgets.org/about/history/>, March 10, 2014 [May 8, 2014]
- [28] Jasmin Blanchette and Mark Summerfield, A Brief History of Qt. In: *C++ GUI Programming with Qt 4*. Prentice-Hall, 2008, xv-xvii.
- [29] Pater Mattis et al, GTK+ 3.12. Internet: <http://ftp.gnome.org/pub/gnome/sources/gtk+/3.12/>, March 25, 2014 [May 8, 2014]
- [30] Oracle et al., Learn about Java Technologies. Internet: <https://www.java.com/en/about/>, May 21, 2014 [May 30, 2014]
- [31] Yukihiro Matsumoto et al., Ruby. Internet: <https://www.ruby-lang.org/en/about/license.txt>, May 24, 2014 [May 30, 2014]
- [32] Qt Project, The Qt Governance Model. Internet: https://qt-project.org/wiki/The_Qt_Governance_Model, November 1, 2013 [May 8, 2014]
- [33] Andrew Knight, Qt on Metro. Internet: http://projects.developer.nokia.com/qt_metro, September 16, 2011 [July 12 2012]. Archive: https://web.archive.org/web/20121020014103/http://projects.developer.nokia.com/qt_metro [May 8, 2014]
- [34] Ilari Sani, Löytääkö Qt uuden kodin Windows 8:sta? (Will Qt find a new home on Windows 8?). *Tietoviikko*. Internet: <http://www.tietoviikko.fi/kehittaja/article741411.ece>, December 16, 2011 [May 8, 2014]
- [35] Nokia Conversations, Open Letter from CEO Stephen Elop, Nokia and CEO Steve Ballmer, Microsoft. Internet: <http://conversations.nokia.com/2011/02/11/open-letter-from-ceo-stephen-elop-nokia-and-ceo-steve-ballmer-microsoft/>, February 11, 2011 [May 8, 2014]
- [36] Jeff Tranter (KDAB), Qt 5 on Windows 8 and Metro UI. Internet: <http://qt-project.org/wiki/Qt-5-on-Windows-8-and-Metro-UI>, June 6, 2012 [September 30, 2013]
- [37] Friedemann Kleint (Digia), Digia at Qt Developer Days 2012 Berlin (Video), 3:03-4:10. Internet: <https://www.youtube.com/watch?v=jRQTzrsNeVk>, November 14, 2012 [May 8, 2014]

- [38] Friedemann Kleint (Digia), Port to Windows Runtime Kick-started. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2013/02/15/port-to-windows-runtime-kick-started/>, February 15, 2013 [May 8, 2014]
- [39] Maurice Kalinowski (Digia), Qt for Windows Runtime Technology Preview Released. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2013/12/12/qt-for-windows-runtime-technology-preview-released/>, December 12, 2013 [May 8, 2014]
- [40] Lars Knoll (Digia), Qt 5.3 Released. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2014/05/20/qt-5-3-released/>, May 20, 2014 [May 30, 2014]
- [41] Andrew Knight (Digia), Bringing the magic of Qt to Windows Runtime. *Qt Blog*. Internet: <https://blog.qt.digia.com/blog/2014/03/25/bringing-the-magic-of-qt-to-windows-runtime/>, March 25, 2014 [May 8, 2014]
- [42] Lars Knoll (Digia), Introducing Qt 5.0. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2012/12/19/qt-5-0/>, December 19, 2012 [May 8, 2014]
- [43] Neal Stephenson. (1999, November 9). *In the Beginning... was the Command Line*. William Morrow Paperbacks. Available: <http://www.cryptonomicon.com/beginning.html> [May 8, 2014]
- [44] Jonathan Corbet et al. (Linux Foundation), Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It, 2012. Available: <http://go.linuxfoundation.org/who-writes-linux-2012>
- [45] NASA Jet Propulsion Laboratory, Validated Toolchain on Mars Rover. Internet: <http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/37779/1/05-0825.pdf> [May 8, 2014]
- [46] Apache Software Foundation, About Apache Flex. Internet: <https://flex.apache.org/about-what-is.html>, May 3, 2014 [May 31, 2014]
- [47] Apache Software Foundation, About Apache Cordova. Internet: <https://cordova.apache.org/#about>, May 25, 2014 [May 30, 2014]
- [48] Integrated Computer Solutions, Motif 2.3.4. Internet: <http://sourceforge.net/projects/motif/files/Motif%202.3.4%20Source%20Code/>, 22 October 2012 [May 8, 2014]
- [49] Alexandre Julliard et al, Wine 1.7.15. Internet: <http://sourceforge.net/projects/motif/files/Motif%202.3.4%20Source%20Code/>, April 5, 2014 [May 8, 2014]
- [50] Mozilla Developer Network, XUL. Internet: <https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL>, April 14, 2014 [May 30, 2014]
- [51] Moreau, Samuel (Microsoft), Designing Metro style: principles and personality. Presented at: *BUILD 2011*. Available: <http://channel9.msdn.com/Events/BUILD/BUILD2011/APP-395T>, September 14, 2011 [May 8, 2014]
- [52] Julien Danjou et al, Awesome v3.5.5. Internet: <http://awesome.naquadah.org/changelogs/v3.5.5>, April 14, 2014 [May 8, 2014]
- [53] Matthew Allum et al, Matchbox 1.11. *Yocto Project*. Internet: <http://downloads.yoctoproject.org/releases/matchbox/libmatchbox/1.11/>, August 14, 2013, [May 8, 2014]
- [54] Spencer Janssen, xmonad 0.11. *Hackage 2*. Internet: <http://hackage.haskell.org/package/xmonad>, January 1, 2013 [May 8, 2014]
- [55] Arno Puder et al, XMLVM. Internet: <http://xmlvm.org/overview/> [May 18, 2014]
- [56] Michael Hanus et al, Curry: A Truly Integrated Functional Logic Language. Internet: <http://www-ps.informatik.uni-kiel.de/currywiki/>, September 5, 2013 [May 18, 2014]
- [57] Simon Peyton Jones et al, The Haskell Programming Language. Internet: <http://www.haskell.org/haskellwiki/Haskell>, September 9, 2013 [May 18, 2014]

- [58] Google Inc. et al, Almost Native Graphics Layer Engine. Internet: <https://code.google.com/p/angleproject/> [May 9, 2014]
- [59] Digia, About Qt: Fast Facts. Internet: <http://qt.digia.com/About-Us/> [May 5, 2014]
- [60] Katherine Barrios (Digia), Qt 5.2 Over 1 Million Downloads. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2014/04/16/qt-5-2-over-1-million-downloads/>, April 16, 2014 [May 8, 2014]
- [61] Tero Kojo (Qt Online Community Manager), Qt-Project.org users. Personal e-mail, May 12, 2014.
- [62] Microsoft, Developing a New Experience Enabled Desktop Browser. Internet: <http://go.microsoft.com/fwlink/p/?linkid=243079>, March 14, 2014 [May 8, 2014]
- [63] Carlos Pizano (Google), Try Chrome in Metro mode. *Chromium Blog*. Internet: <http://blog.chromium.org/2012/06/try-chrome-in-metro-mode.html>, June 7, 2012 [May 8, 2014]
- [64] Brian Bondy (Mozilla), Firefox Metro development begins, status update. Internet: <http://www.brianbondy.com/blog/id/129/firefox-metro-development-begins-status-update/>, March 17, 2012 [May 8, 2014]
- [65] Johnathan Nightingale (Mozilla), Update on Metro. *Mozilla Blog*. Internet: <https://blog.mozilla.org/futurereleases/2014/03/14/metro/>, March 14, 2014 [May 8, 2014]
- [66] Oliver Wolff (Digia), Qt's WinRT Port and its C++/CX Usage. *Qt Blog*. <http://blog.qt.digia.com/blog/2013/04/19/qts-winrt-port-and-its-ccx-usage/>, April 19, 2013 [May 8, 2014]
- [67] Paul Olav Tvete (Nokia), Qt Lighthouse has Grown Up Now. *Qt Blog*. <http://blog.qt.digia.com/blog/2011/05/31/lighthouse-has-grown-up-now/>, May 31, 2011 [May 5, 2014]
- [68] Henrik Rydberg and Canonical Ltd, mtdev v1.1.5. Internet: <http://bitmath.org/code/mtdev/>, February 28 2014 [May 8, 2014]
- [69] Kristian Høgsberg et al, libinput 0.1.0. Internet: <http://freedesktop.org/wiki/Software/libinput/>, February 26, 2014 [May 8, 2014]
- [70] Lev Povalahev et al, GLEW 1.10.0. Internet: <http://glew.sourceforge.net/>, July 22, 2013 [May 8, 2014]
- [71] Kitware, Inc. and Insight Software Consortium, CMake. Internet: <http://www.cmake.org/cmake/project/license.html> [May 31, 2014]
- [72] Kent Hansen (Trolltech), Say hello to QtScript! *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2007/01/05/say-hello-to-qtscript/>, January 5, 2007 [May 8, 2014]
- [73] Google Inc. et al, V8 JavaScript Engine. Internet: <http://src.chromium.org/viewvc/chrome/trunk/src/LICENSE>, March 4, 2014 [May 8, 2014]
- [74] The Chromium Authors, Chromium. Internet: <http://src.chromium.org/viewvc/chrome/trunk/src/LICENSE>, 2014 [May 5, 2014]
- [75] Joyent, Inc. et al, node.js. Internet: <http://nodejs.org/>, May 2, 2014 [May 5, 2014]
- [76] Jason Barron (Nokia), Graphics on Windows from a different angle. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2012/10/24/graphics-on-windows-from-a-different-angle/>, October 24, 2012 [May 8, 2014]
- [77] Robin Burchell et al. (managers), Qt 5. *Ohloh Black Duck Open Hub*. Internet: <https://www.ohloh.net/p/qt5>, May 4, 2014 [May 8, 2014]
- [78] Kamil Trzcinski, Qt 5 and WinRT. Internet: <http://ayufan.eu/projects/qt5-windows-phone-8/>, February 2013 [March 28, 2013]
- [79] SQLite. Internet: <https://sqlite.org/about.html>, May 27, 2014 [May 30, 2014]

- [80] Jack Davis, OPC: A New Standard For Packaging Your Data. *MSDN Magazine*. Internet: <http://msdn.microsoft.com/en-us/magazine/cc163372.aspx>, August 2007 [October 21, 2013]
- [81] Maurice Kalinowski (Digia), Introduction to Windows RT Frameworks. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2013/06/14/introduction-to-windows-rt-frameworks/>, June 14, 2013 [May 5, 2013]
- [82] Richard Moore, RFC: Managing the Addition of New SSL Backends. *Qt Project Development mailing list*. Available: <https://www.mail-archive.com/development@qt-project.org/msg15859.html>, May 3, 2014 [May 9, 2013]
- [83] Eric Young et al, OpenSSL 1.0.1g. *The OpenSSL Project*. <https://www.openssl.org/source/>, April 7, 2014 [May 9, 2014]
- [84] Albert Timashev (ArtUrania), Dream Calendar for Windows Phone. Internet: <http://www.windowsphone.com/en-us/store/app/dream-calendar/b2d3b04f-94aa-45fa-ad00-92fa3e9866f9>, January 26 2014 [May 8, 2014]
- [85] The WebKit Open Source Project, JavaScriptCore. <https://www.webkit.org/projects/javascript/> [May 9, 2014]
- [86] Lars Knoll (Digia), Evolution of the QML engine, part 1. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2013/04/15/evolution-of-the-qml-engine-part-1/>, April 15, 2013 [October 21, 2013]
- [87] Thomas McGuire (KDAB), QML Engine Internals, Part 2: Bindings. Internet: <http://www.kdab.com/qml-engine-internals-part-2-bindings/>, August 10, 2012 [April 14, 2013]
- [88] Daniel Koch and Shannon Woods, ANGLE Project: Windows 8 RT App Store Support. Internet: <https://code.google.com/p/angleproject/issues/detail?id=363>, July 1, 2013 [May 3, 2013]
- [89] Ivan Nevraev (Microsoft), Multiple Ways to Render Point Sprites in DX11. *MSDN Blogs*. Internet: <http://blogs.msdn.com/b/ivanne/archive/2012/01/04/multiple-ways-to-render-point-sprites-in-dx11.aspx>, January 4, 2012 [May 4, 2014]
- [90] Abraham Maslow, *The Psychology of Science: A Reconnaissance*. Ann Kaplan, 1966/2002, 18.
- [91] Andrew Knight, Initial commit of WinRT launcher plugin. Internet: <https://qt.gitorious.org/qt-creator/aknights-qt-creator/commit/6eae1aa540798153ccec6c003167bf1901028183>, November 5, 2012 [May 5, 2014]
- [92] Marius Storm-Olsen (Trolltech), First glance of the .NET generator. Qt 4 internal source tree, commit 5de6706071ca8f637f5ffd6d6d75127810fec744, August 12, 2002.
- [93] Oliver Wolff, Experimental Version of Qt Creator's WinRT Plugin. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2014/03/05/experimental-version-of-qt-creators-winrt-plugin/>, March 5, 2014 [May 3, 2014]
- [94] Steve Matteson (Google), Open Sans. Internet: <http://www.opensans.com/>, 2014 [May 8, 2014]
- [95] David Brezina et al. (Tiro Typeworks). *Microsoft Typography*. Internet: <http://www.microsoft.com/typography/fonts/font.aspx?FMID=1989>, February 4, 2011 [May 5, 2014]
- [96] Microsoft, Using the Windows App Certification Kit (whitepaper). Internet: <http://www.microsoft.com/en-us/download/details.aspx?id=27414>, June 21, 2013 [May 5, 2014]
- [97] Jean-Baptiste Kempf, Technical update on the WinRT port. Internet: <http://www.jbkempf.com/blog/post/2013/Technical-update-on-the-WinRT-port>, February 12, 2013 [May 8, 2014]
- [98] Digia, Quick Forecast Source. Internet: <https://qt.gitorious.org/qt-labs/weather-app/>, May 9, 2014 [May 9, 2014]
- [99] Digia, Quick Forecast. *Google Play*. Internet: <https://play.google.com/store/apps/details?id=org.qtproject.quickforecast>, March 19, 2014 [May 8, 2014]

- [100] Digia, Quick Forecast. *App Store on iTunes*. Internet: <https://itunes.apple.com/no/app/quick-forecast/id736658981>, March 20, 2014 [May 8, 2014]
- [101] Digia, Quick Forecast. *Windows Phone Store*. Internet: <http://www.windowsphone.com/en-us/store/app/quickforecast/35572287-c6d6-4d5c-9799-46555f7fc459>, April 24, 2014 [May 8, 2014]
- [102] Digia, Quick Forecast. *Windows Store*. Internet: <http://apps.microsoft.com/windows/en-us/app/ad41d87d-9cb0-4b76-9a4a-5e2c739161e6>, May 19, 2014 [May 30, 2014]
- [103] Caroline Chao (Digia), Cross-Platform Applications in iOS and Android Stores with Qt. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2013/12/10/cross-platform-applications-in-ios-and-android-stores-with-qt/>, December 10, 2013 [May 8, 2014]
- [104] Leena Miettinen (Digia), Qt Weekly #2: Localizing Qt Quick Apps. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2014/03/19/qt-weekly-2-localizing-qt-quick-apps/>, March 19, 2014 [May 8, 2014]
- [105] Eike Ziller (Digia), Qt Creator 3.1.0 released. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2014/04/15/qt-creator-3-1-0-released/> [May 9, 2014]
- [106] LLVM Team, University of Illinois at Urbana-Champaign, LLVM 3.4. Internet: <http://www.llvm.org/releases/3.4/docs/ReleaseNotes.html>, January 2, 2014 [May 30, 2014]
- [107] Zeno Albisser (Digia), Experimenting with Chromium and Qt. *Qt Blog*. Internet: <http://blog.qt.digia.com/blog/2013/06/25/experimenting-with-chromium-and-qt/>, June 25, 2013 [May 30, 2014]
- [108] Microsoft and Agfa Monotype Corporation, Segoe UI. Internet: <http://www.microsoft.com/typography/fonts/font.aspx?FMID=1941>, May 30, 2012 [May 30, 2014]
- [109] Global Game Jam, FGJ Tampere Games. Internet: <http://www.globalgamejam.org/2014/jam-sites/fgj-tampere/games>, January 26, 2014 [May 30, 2014]

Code contributions

This section includes references to relevant code changes through the Qt Project's openly governed code review system at <https://codereview.qt-project.org/>.

- [110] Maurice Kalinowski, WinRT cannot handle library loading outside application bundle. Internet: <https://codereview.qt-project.org/39166>, November 12, 2012.
- [111] Andrew Knight, Initial mkspec and platform detection of WinRT. Internet: <https://codereview.qt-project.org/39875>, December 4, 2012.
- [112] Andrew Knight, Updated winmain for use in WinRT. Internet: <https://codereview.qt-project.org/39876>, December 4, 2012.
- [113] Kamil Trzcinski, Windows RT and Windows Phone preliminary support. Internet: <https://codereview.qt-project.org/46916>, February 12, 2013.
- [114] Oliver Wolff, Removed network for winrt non phone builds. Internet: <https://codereview.qt-project.org/47039>, February 14, 2013.
- [115] Trzcinski Kamil, qmake: added WinRT and WinPhone configuration switch. Internet: <https://codereview.qt-project.org/47559>, February 12, 2013.
- [116] Andrew Knight, WinRT: Enable command line passing from main. Internet: <https://codereview.qt-project.org/51187>, March 21, 2013.
- [117] Andrew Knight, Patch ANGLE to support WinRT. Internet: <https://codereview.qt-project.org/51857>, April 25, 2013.

- [118] Andrew Knight, Implement OpenGL ES 2 support for WinRT QPA. Internet: <https://codereview.qt-project.org/51858>, April 27, 2013.
- [119] Andrew Knight, WinRT: Use relative paths when calling LoadPackagedLibrary. Internet: <https://codereview.qt-project.org/52251>, March 27, 2013.
- [120] Andrew Knight, WinRT: Basic Input Context Support. Internet: <https://codereview.qt-project.org/52603>, April 22, 2013.
- [121] Andrew Knight, Upgrade ANGLE to DX11 Proto. Internet: <https://codereview.qt-project.org/52810>, April 8, 2013.
- [122] Andrew Knight, ANGLE DX11: Prevent assert when view is minimized or size goes to 0x0. Internet: <https://codereview.qt-project.org/52811>, April 8, 2013.
- [123] Andrew Knight, ANGLE: Avoid memory copies on buffers when data is null. Internet: <https://codereview.qt-project.org/53037>, April 8, 2013.
- [124] Andrew Knight, Unimplement shared memory for WinPhone. Internet: <https://codereview.qt-project.org/54225>, April 19, 2013.
- [125] Andrew Knight, WinRT: Top-level windows should always be fullscreen. Internet: <https://codereview.qt-project.org/54284>, April 19, 2013.
- [126] Andrew Knight, WinRT: Introduce Platform Services. Internet: <https://codereview.qt-project.org/54374>, April 25, 2013.
- [127] Andrew Knight, WinRT: Implement Platform Cursor. Internet: <https://codereview.qt-project.org/54375>, April 22, 2013.
- [128] Andrew Knight, WinRT: Refactor pointer handling. Internet: <https://codereview.qt-project.org/54383>, April 24, 2013.
- [129] Oliver Wolff, Disable sqlite for Windows Phone 8 builds. Internet: <https://codereview.qt-project.org/54438>, April 25, 2013.
- [130] Andrew Knight, Fix winphone makefile generator. Internet: <https://codereview.qt-project.org/54495>, April 24, 2013.
- [131] Andrew Knight, Don't use dirty flip on phone. Internet: <https://codereview.qt-project.org/54547>, April 23, 2013.
- [132] Andrew Knight, WinRT: Support screen orientation changes. Internet: <https://codereview.qt-project.org/54575>, April 24, 2013.
- [133] Andrew Knight, WinRT: Improve key handling. Internet: <https://codereview.qt-project.org/56450>, May 19, 2013.
- [134] Andrew Knight et al, WinRT QPA plugin. Internet: <https://codereview.qt-project.org/64459>, September 3, 2013. Original: <https://qt.gitorious.org/~aknight/qt/aknights-qtbase/commit/605f4f91ebc1d48dc5b96b2ec71edca999689a34>, November 5, 2012.
- [135] Andrew Knight, ANGLE: Enable D3D11 for feature level 9 cards. Internet: <https://codereview.qt-project.org/64933>, September 9, 2013.
- [136] Andrew Knight, WinRT: ANGLE-based backing store. Internet: <https://codereview.qt-project.org/64934>, September 9, 2013.
- [137] Andrew Knight, Introducing d3dcompiler_qt. Internet: <https://codereview.qt-project.org/67386>, October 6, 2013.

- [138] Maurice Kalinowski and Andrew Knight. WinRT QPA: Fix touch release on phone. Internet: <https://codereview.qt-project.org/67389>, October 6, 2013.
- [139] Andrew Knight, Fix build on WinRT. Internet: <https://codereview.qt-project.org/67391>, October 6, 2013.
- [140] Maurice Kalinowski, add testrunner for WinRT. Internet: <https://codereview.qt-project.org/7245572>, November 26, 2013.
- [141] Andrew Knight, Introducing Qt D3D Compiler service. Internet: <https://codereview.qt-project.org/73047>, December 3, 2013.
- [142] Andrew Knight, WinRT: Provide qmake feature for generating a package manifest. Internet: <https://codereview.qt-project.org/74410>, December 26, 2013.
- [143] Andrew Knight, Introducing winrtrunner. Internet: <https://codereview.qt-project.org/74534>, January 2, 2014.
- [144] Andrew Knight, Windows Phone backend for winrtrunner. Internet: <https://codereview.qt-project.org/75071>, January 10, 2014.
- [145] Andrew Knight, Initial sensors backend for WinRT/Windows Phone. Internet: <https://codereview.qt-project.org/78135>, February 13, 2014.
- [146] Jörg Bornemann, Andrew Knight, and Friedemann Kleint. Long live the Windows RT plugin!. Internet: <https://codereview.qt-project.org/78719>, February 20, 2014.
- [147] Andrew Knight, Upgrade ANGLE to 1.3.5bb7ec572d0a. Internet: <https://codereview.qt-project.org/78775>, February 20, 2014.
- [148] Andrew Knight, Windows Phone: Handle back-button press. Internet: <https://codereview.qt-project.org/80314>, March 7, 2014.
- [149] Andrew Knight, WinRT: Load system fonts using DirectWrite. Internet: <https://codereview.qt-project.org/80425>, March 10, 2014.
- [150] Andrew Knight, winrtrunner: Provide a remote agent for Windows Phone. Internet: <https://codereview.qt-project.org/80804>, March 13, 2014.
- [151] David Schulz, WinRT: Enable debugging for local packages. Internet: <https://codereview.qt-project.org/82615>, April 3, 2014.
- [152] Andrew Knight, WinRT: Don't use the native handle for waiting. Internet: <https://codereview.qt-project.org/83217>, April 10, 2014.
- [153] Andrew Knight, WinRT: Fix TCP socket reads. Internet: <https://codereview.qt-project.org/83216>, April 10, 2014.
- [154] Andrew Knight, WinRT: Handle back button as press or release. Internet: <https://codereview.qt-project.org/83803>, April 22, 2014.
- [155] Andrew Knight, ANGLE WinRT: Call Trim() when application suspends. Internet: <https://codereview.qt-project.org/83804>, April 22, 2014.
- [156] Andrew Knight, Add missing QT_NO_NETWORKPROXY guards around HTTP connect statements. Internet: <https://codereview.qt-project.org/84172>, April 28, 2014.
- [157] Andrew Knight, Windows Phone: add language control to the package manifest. Internet: <https://codereview.qt-project.org/84187>, April 28, 2014.
- [158] Andrew Knight, ANGLE WinRT: Create swap chain using physical resolution. Internet: <https://codereview.qt-project.org/84699>, May 5, 2014.

[159] Andrew Knight, WinRT: Create windows in physical resolution. Internet: <https://codereview.qt-project.org/84700>, May 5, 2014.

[160] Andrew Knight, ANGLE D3D11: Don't use mipmaps in level 9 textures. Internet: <https://codereview.qt-project.org/84743>, May 6, 2014.

[161] Andrew Knight, Add privacy policy link for WinRT. Internet: <https://codereview.qt-project.org/85497>, May 15, 2014.

MSDN documentation

This section contains references to relevant Microsoft documentation, including a descriptive summary sentence from the referenced text.

[162] MSDN, Windows API. *"The Microsoft Windows application programming interface (API) provides services used by all Windows-based applications."* Internet: <http://msdn.microsoft.com/en-us/library/cc433218.aspx>

[163] MSDN, Windows Runtime C++ Template Library (WRL). *"The Windows Runtime C++ Template Library (WRL) is a template library that provides a low-level way to author and use Windows Runtime components."* Internet: <http://msdn.microsoft.com/en-us/library/hh438466.aspx>

[164] MSDN, /ZW (Windows Runtime Compilation). *"Compiles source code to support Visual C++ component extensions (C++/CX) for the creation of Windows Store apps."* Internet: <http://msdn.microsoft.com/en-us/library/hh561383.aspx>

[165] MSDN, Error Handling in COM. *"Almost all COM functions and interface methods return a value of the type HRESULT. The HRESULT (for result handle) is a way of returning success, warning, and error values."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms679692.aspx>

[166] MSDN, Virtual Memory Functions. *"The virtual memory functions enable a process to manipulate or determine the status of pages in its virtual address space."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/aa366916.aspx>

[167] MSDN, Legacy Graphics: OpenGL. *"As a software interface for graphics hardware, OpenGL renders multidimensional objects into a framebuffer."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/dd374278.aspx>

[168] MSDN, CRT functions not supported with /ZW. *"Many C runtime (CRT) functions are not available when you build Windows Store apps."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/jj606124.aspx>

[169] MSDN, Pivot class. *"The Pivot control provides a quick way to manage the navigation of views within an application."* Internet: <http://msdn.microsoft.com/en-us/library/windowsphone/develop/windows.ui.xaml.controls.pivot.aspx>

[170] MSDN, GridView class. *"Represents a control that displays a horizontal grid of data items."* Internet: <http://msdn.microsoft.com/en-us/library/windowsphone/develop/windows.ui.xaml.controls.gridview.aspx>

[171] MSDN, CommandBar class. *"Represents a specialized app bar that provides layout for AppBarButton and related command elements."* Internet: <http://msdn.microsoft.com/en-us/library/windowsphone/develop/windows.ui.xaml.controls.commandbar.aspx>

[172] MSDN, Add-AppxPackage. *"Adds a signed app package (.appx) to a user account."* Internet: <http://technet.microsoft.com/en-us/library/hh856048.aspx>

[173] MSDN, Deploying an app with the Application Deployment tool. *"You can also use the Windows Phone Application Deployment tool (XapDeploy.exe) to deploy your app."* Internet: http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402565.aspx#BKMK_tool

[174] MSDN, Visual C++ in Visual Studio 2013. *"The Visual C++ language and development tools help you develop native Windows Store apps, native desktop apps, and managed apps that run on the .NET Framework."* Internet: <http://msdn.microsoft.com/en-us/library/vstudio/60k1461a.aspx>

[175] MSDN, WaitForMultipleObjects function. *"Waits until one or all of the specified objects are in the signaled state or the time-out interval elapses."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms687025.aspx>

[176] MSDN, CreateMutex function. *"Creates or opens a named or unnamed mutex object."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms682411.aspx>

[177] MSDN, WaitForMultipleObjectsEx function. *"Waits until one or all of the specified objects are in the signaled state, an I/O completion routine or asynchronous procedure call (APC) is queued to the thread, or the time-out interval elapses."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms687028.aspx>

[178] MSDN, CreateMutexEx. *"Creates or opens a named or unnamed mutex object and returns a handle to the object."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms682418.aspx>

[179] MSDN, Windows.System.Threading namespace. *"Enables an application to use the thread pool to run work items."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.system.threading.aspx>

[180] MSDN, Windows.Networking namespace. *"Provides access to hostnames and endpoints used by network apps."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.networking.aspx>

[181] MSDN, StreamSocket.UpgradeToSslAsync. *"Starts an asynchronous operation to upgrade a connected socket to use SSL on a StreamSocket object."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.networking.sockets.streamsocket.upgradetosslasync.aspx>

[182] MSDN, Direct2D. *"Direct2D is a hardware-accelerated, immediate-mode, 2-D graphics API that provides high performance and high-quality rendering for 2-D geometry, bitmaps, and text."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/dd370990.aspx>

[183] MSDN, Direct3D 11 Graphics. *"You can use Microsoft Direct3D 11 graphics to create 3-D graphics for games and scientific and desktop applications."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/ff476080.aspx>

[184] MSDN, DXGI. *"Microsoft DirectX Graphics Infrastructure (DXGI) handles enumerating graphics adapters, enumerating display modes, selecting buffer formats, sharing resources between processes (such as, between applications and the Desktop Window Manager (DWM)), and presenting rendered frames to a window or monitor for display."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/hh404534.aspx>

[185] MSDN, InputPane.TryShow method. *"Shows the InputPane if it is hidden."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.viewmanagement.inputpane.tryshow.aspx>

[186] MSDN, Windows Automation API: UI Automation. *"Microsoft UI Automation is an accessibility framework that enables Microsoft Windows applications to provide and consume programmatic information about user interfaces (UIs)."* Internet: <http://msdn.microsoft.com/en-us/library/ms726294.aspx>

[187] MSDN, Global and Local Functions. *"The global and local functions are supported for porting from 16-bit code, or for maintaining source code compatibility with 16-bit Windows."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/aa366596.aspx>

[188] MSDN, Heap Functions. *"Each process has a default heap provided by the system. Applications that make frequent allocations from the heap can improve performance by using private heaps."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/aa366711.aspx>

[189] MSDN, ICoreWindow interface. *"Specifies an interface for a window object and its input events as well as basic user interface behaviors."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.core.icorewindow.aspx>

- [190] MSDN, CoreWindow class. *"Represents the Windows Store app with input events and basic user interface behaviors."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.core.corewindow.aspx>
- [191] MSDN, IPackageDebugSettings interface. *"Enables debugger developers control over the lifecycle of a Windows Store app, such as when it is suspended or resumed."* Internet: <http://msdn.microsoft.com/en-us/library/hh438393.aspx>
- [192] MSDN, Package Manager class. *"Manages the software available to a user."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.management.deployment.packagemanager.aspx>
- [193] MSDN, Packaging API. *"Learn about the packaging API, which you can use to create, read, and write app packages. Each app package contains the files that constitute a Windows Store app, and a manifest file that describes the software to Windows."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/hh446766.aspx>
- [194] MSDN, Core Connectivity Reference (Compact 2013). *"The Core Connectivity infrastructure provides the basic components required for connectivity between a desktop and a Windows Embedded Compact powered device."* Internet: <http://msdn.microsoft.com/en-us/library/ee481381.aspx>
- [195] MSDN, Debugging Using CDB and NTSD. *"This section describes how to perform basic debugging tasks using the Microsoft Console Debugger (CDB) and Microsoft NT Symbolic Debugger (NTSD)."* Internet: <http://msdn.microsoft.com/en-us/library/windows/hardware/hh406277.aspx>
- [196] MSDN, Start the Remote Debugging Monitor. *"The Remote Debugging Monitor (msvsmon.exe) is a small application that Visual Studio connects to for remote debugging."* Internet: <http://msdn.microsoft.com/en-us/library/xf8k2h6a.aspx>
- [197] MSDN, The Remote.exe Utility. *"The remote.exe utility is a versatile server/client tool that allows you to run command-line programs on remote computers."* Internet: <http://msdn.microsoft.com/en-us/library/windows/hardware/ff558875.aspx>
- [198] MSDN, Windows Remote Management. *"Windows Remote Management (WinRM) is the Microsoft implementation of WS-Management Protocol, a standard Simple Object Access Protocol (SOAP)-based, firewall-friendly protocol that allows hardware and operating systems, from different vendors, to interoperate."* Internet: <http://msdn.microsoft.com/en-us/library/aa384426.aspx>
- [199] MSDN, ARM Kits policy information. *"A new ARM Kits policy (Microsoft-Windows-Kits-Secure-Boot-Policy.p7b) comes with the Windows SDK for Windows 8.1. This policy enables developers to use various Microsoft tools and kits on ARM devices, and at the same time, preserves the integrity of ARM devices that ship with a production policy."* Internet: <http://msdn.microsoft.com/en-US/windows/desktop/dn469188>
- [200] MSDN, Unmanaged Device-Side Smart Device Connectivity API. *"Visual C++ device projects that use this API are called device agent applications. Desktop applications that use the Smart Device Connectivity API can deploy device agents to the device and communicate with them by exchanging packet data."* Internet: <http://msdn.microsoft.com/en-us/library/bb907096.aspx>
- [201] MSDN, UI Fonts supported on Windows Phone. *"The following table lists all UI fonts that are supported on a Windows Phone device."* Internet: http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202920.aspx#BKMK_SupportedUIFontsinWindowsPhone
- [202] MSDN, Windows.Devices.Input namespace. *"Provides support for identifying the input devices available (pointer, touch, mouse, and keyboard) and retrieving information about those devices."* Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.input.aspx>
- [203] MSDN, IDXGIDevice3::Trim method. *"Trims the graphics memory allocated by the IDXGIDevice3 DXGI device on the app's behalf."* Internet: <http://msdn.microsoft.com/en-us/library/windows/desktop/dn280346.aspx>

[204] MSDN, App certification requirements for the Windows Store. "4.1.1 Your app must have a privacy statement if it is network-capable." Internet: <http://msdn.microsoft.com/en-us/library/windows/apps/hh694083.aspx>

Qt documentation

This section contains references to relevant Qt documentation, including a descriptive summary sentence from the referenced text.

[205] Qt Documentation, Qt 5.3. "Qt is a full development framework with tools designed to streamline the creation of applications and user interfaces for desktop, embedded, and mobile platforms." Internet: </doc/qt-5/index.html>

[206] Qt Documentation, Qt QML. "The Qt QML module provides a framework for developing applications and libraries with the QML language." Internet: </doc/qt-5/qtqml-index.html>

[207] Qt Documentation, Qt Quick. "The Qt Quick module is the standard library for writing QML applications." Internet: </doc/qt-5/qtquick-index.html>

[208] Qt Documentation, Qt Quick Demo - Same Game. "A QML implementation of the popular puzzle game by Kuniaki Moribe." Internet: <https://qt-project.org/doc/qt-5/qtquick-demos-samegame-example.html>

[209] Qt Documentation, Qt Widgets. "The Qt Widgets Module provides a set of UI elements to create classic desktop-style user interfaces." Internet: </doc/qt-5/qtwidgets-index.html>

[210] Qt Documentation, Qt Quick Controls. "The Qt Quick Controls module provides a set of controls that can be used to build complete interfaces in Qt Quick." Internet: </doc/qt-5/qtquickcontrols-index.html>

[211] Qt Documentation, QML Applications. "QML is a declarative language that allows user interfaces to be described in terms of their visual components and how they interact and relate with one another." Internet: </doc/qt-5/qmlapplications.html>

[212] Qt Documentation, Qt Platform Abstraction. "The Qt Platform Abstraction (QPA) is the platform abstraction layer for Qt 5 and replaces Qt for Embedded Linux and the platform ports from Qt 4." Internet: <https://qt-project.org/doc/qt-5/qpa.html>

[213] Qt Documentation, Compact and Efficient Windowing System (QWS). "Qt builds on the standard API for embedded Linux devices with its own compact window system [which writes] directly to the Linux framebuffer, eliminating the need for the X11 windowing system." Internet: <https://qt-project.org/doc/qt-4.8/embeddedlinux-support.html#compact-and-efficient-windowing-system-qws>

[214] Qt Documentation, Qt Windows Extras. "Qt Windows Extras provide classes and functions that enable you to use miscellaneous Windows-specific functions." Internet: </doc/qt-5/qtwinextras-index.html>

[215] Qt Documentation, qmake Manual. "The qmake tool helps simplify the build process for development projects across different platforms." Internet: </doc/qt-5/qmake-manual.html>

[216] Qt Documentation, Using the Meta-Object Compiler (moc). "The Meta-Object Compiler, moc, is the program that handles Qt's C++ extensions." Internet: <https://qt-project.org/doc/qt-5/moc.html>

[217] Qt Documentation, Qbs Manual. "Qt Build Suite (Qbs) is a tool that helps simplify the build process for developing projects across multiple platforms." Internet: </doc/qbs-1.2/index.html>

[218] Qt Documentation, QMake Variable Reference: WINRT_MANIFEST. "Specifies parameters to be passed to the application manifest on Windows Runtime." Internet: <http://doc-snapshot.qt-project.org/qt5-stable/qmake-variable-reference.html#wirt-manifest>

[219] Qt Documentation, Qt Network. "Qt Network provides a set of APIs for programming applications that use TCP/IP." Internet: <https://qt-project.org/doc/qt-5/qtnetwork-index.html>

- [220] Qt Documentation, QGuiApplication. *"The QGuiApplication class manages the GUI application's control flow and main settings."* Internet: <https://qt-project.org/doc/qt-5/qguiapplication.html>
- [221] Qt Documentation, PinchArea. *"Enables simple pinch gesture handling."* Internet: <https://qt-project.org/doc/qt-5/qml-qtquick-pincharea.html>
- [222] Qt Documentation, QCursor. *"The QCursor class provides a mouse cursor with an arbitrary shape."* Internet: [/doc/qt-5/qcursor.html](https://qt-project.org/doc/qt-5/qcursor.html)
- [223] Qt Documentation, QPainter. *"The QPainter class performs low-level painting on widgets and other paint devices."* Internet: [/doc/qt-5/qpainter.html](https://qt-project.org/doc/qt-5/qpainter.html)
- [224] Qt Documentation, Qt Quick Demo - Photo Surface. *"A touch-based app for shuffling photos around a virtual surface."* Internet: [/doc/qt-5/qtquick-demos-photosurface-example.html](https://qt-project.org/doc/qt-5/qtquick-demos-photosurface-example.html)
- [225] Qt Documentation, Wiggly Example. *"The Wiggly example shows how to animate a widget using QBasicTimer and timerEvent()."* Internet: [/doc/qt-5/qtwidgets-widgets-wiggly-example.html](https://qt-project.org/doc/qt-5/qtwidgets-widgets-wiggly-example.html)
- [226] Qt Documentation, Packaging Shaders for Deployment. *"While qtd3dservice will pick up shader sources and generate shader binaries during runtime, it obviously cannot be used in published applications. It is the responsibility of the developer to package these shader "blobs" with the application before publishing."* Internet: [/doc/qt-5/winrt-support.html#packaging-shaders-for-deployment](https://qt-project.org/doc/qt-5/winrt-support.html#packaging-shaders-for-deployment)
- [227] Qt Documentation, OpenGL Window Example. *"This example shows how to create a minimal QWindow based application for the purpose of using OpenGL."* Internet: <https://qt-project.org/doc/qt-5/qtgui-openglwindow-example.html>
- [228] Qt Documentation, Cube OpenGL ES 2.0 example. *"The Cube OpenGL ES 2.0 example shows how to write mouse rotateable textured 3D cube using OpenGL ES 2.0 with Qt."* Internet: <https://qt-project.org/doc/qt-5/qtopengl-cube-example.html>
- [229] Qt Documentation, WinRT Runner Tool. *"The WinRT Runner Tool [...] is intended to aid in the deployment, launching, and debugging of Qt for WinRT applications."* Internet: <https://qt-project.org/doc/qt-5/winrt-support.html#wintr-runner-tool>
- [230] Qt Documentation, Resource Compiler (rcc). *"The rcc tool is used to embed resources into a Qt application during the build process."* Internet: <https://qt-project.org/doc/qt-5/rcc.html>
- [231] Qt Documentation, QAuthenticator. *"The QAuthenticator class provides an authentication object."* Internet: <https://qt-project.org/doc/qt-5/qauthenticator.html>
- [232] Qt Documentation, QMenuBar. *"The QMenuBar class provides a horizontal menu bar."* Internet: <https://qt-project.org/doc/qt-5/qmenubar.html>
- [233] Qt Documentation, Qt Multimedia. *"Qt Multimedia is an essential module that provides a rich set of QML types and C++ classes to handle multimedia content."* Internet: <https://qt-project.org/doc/qt-5/qtmultimedia-index.html>
- [234] Qt Documentation, Qt Positioning. *"The Qt Positioning API provides positioning information via QML and C++ interfaces."* Internet: <https://qt-project.org/doc/qt-5/qtpositioning-index.html>
- [235] Qt Documentation, TabView. *"A control that allows the user to select one of multiple stacked items."* Internet: <https://qt-project.org/doc/qt-5/qml-qtquick-controls-tabview.html>
- [236] Qt Documentation, QAudioOutput. *"The QAudioOutput class provides an interface for sending audio data to an audio output device."* Internet: <https://qt-project.org/doc/qt-5/qaudiooutput.html>

Glossary of terms

This section contains brief definitions of domain terminology used throughout the text.

ABI

Abstract Binary Interface, the low-level interface between libraries typically defined by the exported methods' signatures and calling conventions.

Android

A smartphone and tablet operating system by Google and open source contributors.

API

Application Programming Interface, a defined set of procedures to allow a software component to interact with other software.

Appx

The packaging scheme for Windows Store applications.

ARM

A provider of microprocessor specifications commonly used in mobile and embedded CPUs.

blitting

Bit-block transfer, the process of copying a section of a backing graphics resource to another surface, typically only the section or sections which changed.

BlackBerry 10

A smartphone and tablet operating system by BlackBerry.

C

A strongly-typed, structured imperative programming language created in the early 1970s by Dennis Ritchie at Bell Labs and still in wide use today.

C++

An strongly-typed, object-oriented programming based on C begun by Bjarne Stroustrup at Bell Labs in 1979. It is one of the most popular programming languages in use, and is commonly found (similarly to C) as a native binding choice for operating system APIs.

C++11

A recent revision of the C++ language and *STL* standard.

chrome

Non-content parts of the application window or desktop environment which typically deal with window management, such as title bars, window borders, maximize/minimize/close buttons, or scrollbars.

cmdlet

A PowerShell script.

cross-platform

Software which is capable of operating on multiple types of devices and/or operating systems.

CRT

C Runtime, the system library enabling C/C++ programs to run.

COM

Component Object Model, commonly used in (but not limited to) Microsoft APIs, whereby objects are instantiated opaquely to the application (in the same process, a different process, or even a remote machine), and expose one or more interfaces to communicate with that object via proxy.

copyleft

The concept of a license which requires the source code of an application (in whole or part) to remain freely accessible if a program uses portions of a software product and is then modified or distributed.

declarative

A programming paradigm which focuses on modeling the structure and relationships of objects in a high-level data structure, typically marked by a high degree of human readability and the ability to be manipulated by visual design tools.

desktop metaphor

In window management, a workspace analogy in which windows are arranged overlapping, like sheets of paper on a desk.

DIP

Device-independent pixel, a logical pixel for use in user interface frameworks (e.g. CSS), typically when physical pixel size or amount would be too diverse to provide consistent representation across devices.

DWM

Desktop Window Manager: the software which is responsible for compositing application windows within the screen space available to the user, as well as managing their geometry, focus, and top-level interaction.

EGL

A cross-platform adaptation layer specified by Khronos for creating OpenGL contexts given platform-specific window and screen handles.

EGLFS

A basic QPA platform plugin utilizing OpenGL ES2 as the primary means of displaying content on the screen. An example of a full-screen, hardware-accelerated, typically embedded Qt solution.

factory class

A design pattern in which a special class (a "factory") which is used to construct object instances as opposed to, e.g., using the new operator to instantiate objects directly.

file extension

A convention in which the section of the file name following the last period (.), denotes a standard association to a file format. For example, a JPEG image file might be named "image.jpg" or "photo.jpeg".

FOSS

Free and open source software, or software which has freely available source code and may be distributed under terms which afford protections to the receiver rights to use the software, as well as possible extended rights and obligations (*copyleft* for the source code to remain open and available).

Git

The distributed version control system in use by the Qt Project.

graphics pipeline

The subsystem which describes how drawing commands (such as OpenGL) are queued to (typically dedicated) hardware for rendering a graphical scene.

GUI

Graphical User Interface, a visual (image-based) *user interface*.

GPL

GNU Public License, a widely used *copyleft* _FOSS_ license.

HTML5

A collection of modern web-standard programming languages (HTML, CSS, and JavaScript, among others) which together to provide rich user experiences within a web browser.

hybrid

An application or application framework combining web and native technologies.

IDE

Integrated Development Environment, an application which combines an editor and supporting project management tools to create, package, and deploy applications for a particular platform or framework.

IDL

Interface Description Language, a language-independent description of a class or classes, from which concrete interfaces (e.g. C++ headers) can be generated.

iOS

A smartphone and tablet operating system by Apple.

JavaFX

A declarative syntax for creating GUIs within the Java graphics engine. JavaFX 1 influenced the syntax chosen for QML.

JavaScript

A widely implemented (e.g. in web browsers and QML) interpreted programming language resembling C (and to some extent, C++ and Java) and formalized as ECMAScript/ECMA-262.

JIT

Just-in-time, typically referring to compilers which generate machine code at runtime (e.g. from interpreted code like JavaScript or Regular Expressions) in order to deliver improved performance.

KDE

K Desktop Environment, a *DWM* for *X11* and suite of associated software applications based on Qt for Linux.

LinuxFB

A basic QPA platform plugin utilizing the Linux fbdev for screen flushing. An example of a software-rasterized and composited Qt solution.

look and feel

The styling and behavior of a *user interface* element or environment, typically defined by the platform style guide.

makefile

A simple scripting file format for describing the environment and tools to be invoked in order to build a software project. The exact format is typically dictated by the make tool, e.g. nmake or GNU Make.

MeeGo

A discontinued venture between Nokia and Intel based on their respective smartphone Linux distributions, Maemo and Moblin. MeeGo forms the basis of Mer, the core of *Sailfish OS*.

mkspec

Make specification, a set of definitions used by qmake when building for a given platform.

moc

Meta object compiler, a tool used by Qt to generate C++ *meta object* code.

Modern UI

Formerly known as "Metro", Microsoft's touch-oriented UI environment found on Windows 8 products and designed for running Windows Store Apps.

MSDN

Microsoft Developer Network, a website documenting Microsoft's various APIs and tools.

MXML

An XML language for describing Adobe Flash/Apache Flex-based user interfaces.

nmake

The Microsoft Program Maintenance Utility, a makefile interpreter created for use with *MSVC*.

Open source

Referring to a work in which the source material is freely accessible. Open-source software has its source code available so that users can view the internal workings of the software and, in many cases, modify the software under a free license.

OpenGL

A royalty-free drawing API widely implemented on platforms with hardware-accelerated graphics drawing support.

OpenGL ES 2

An limited OpenGL profile geared toward embedded and mobile hardware.

painting

In computer graphics, the routines performed to translate drawing commands into pixels on the screen (or other drawing surface, such as an image in memory).

platform

In the context of GUI frameworks, the combination of an operating system and window manager. The platform helps define what APIs, toolchains, and system libraries are available for use by software, thereby requiring the developer to write platform-specific code paths as a result.

PID

Process identifier, a handle to a running program on a system.

PIMPL

An acronym for private implementation, a programming design pattern which hides implementation details from an external API in order to reduce complexity, improve security, and improve binary and source compatibility through use of an internal API that is opaque to the public interface.

porting

The act of migrating software designed for one system to be operational on another system. It may include changes to the build system, the use of underlying libraries, or even moving to an entirely different programming language. The resulting product is called the "port".

PowerShell

A command line environment (shell) on Windows.

reflection

The ability of a programming language to examine the functional components of an object, to provide meta information about an object's type, such as its properties, methods, and inheritance.

Sailfish OS

A smartphone operating system by Jolla, based on the embedded Linux distribution Mer (an OS based on MeeGo).

sandbox

A container within which an application is run in which the system capabilities of the application (such as accessing memory or loading libraries) is strictly controlled in order to provide a security "sandbox" in which the application can run without the risk of adversely affecting the rest of the system.

SDK

Software Development Kit, a collection of libraries, tools, and documentation to aid a developer in writing applications for a particular platform or service.

library

A software component with an API designed to expose functionality which is reusable across applications.

STL

C++ Standard Template Library, which provides container classes, algorithms, and memory management functionality to C++.

Swap chain

The mechanism which flips front and back frame buffers in order to present new content to the screen.

toolchain

The compiler, linker, and supporting "chain of tools"; i.e., the set of programs used to create a new executable binary (computer program) from source code.

UI

User interface, the space where interaction between a human and machine occurs.

W3C

World Wide Web Consortium, a think tank for analyzing and standardizing file format specifications and protocols for the web.

widget

A basic user interface component such as a button, scrollbar, or text entry field, and typically referring to a component which is painted by the operating system or window manager.

Window chrome

The area around an application window typically consisting of a frame and a title bar.

Windows RT

An operating system based on Windows 8 designed for *ARM*-powered tablet devices such as the Microsoft Surface. Applications utilize the *Windows Runtime* API.

Windows 8

Microsoft's latest operating system for PCs and tablets. Related to Windows RT and Windows Phone 8.

Windows Phone 8

Microsoft's latest smartphone operating system.

Windows Store

An online marketplace operated by Microsoft for the distribution of applications and other media.

Windows Store App

An application (*app*) packaged with *Appx* or *XAP* with the possibility of being distributed via the *Windows Store*.

WinMD

Windows Metadata, a file format describing the interfaces of a Windows Runtime component.

WinRT

Windows Runtime, the *API* and runtime library defined by Microsoft for its Windows 8 family of operating systems.

X11

A windowing system protocol widely employed on UNIX-like operating systems.

XAP

The packaging scheme used for Windows Phone and Silverlight applications compressed using the *.ZIP* format.

XAML

eXtensible Application Markup Language, an XML-based declarative UI language used primarily by Microsoft's .NET family of programming languages.

.ZIP file

A file format which compresses one or more files as an archive.