Developing a Benchmark for Qt on Embedded Platforms

Tyler Beaupre, Khoi Doan, Nata Vacheishvili

Sponsor: The Qt Company

Project Advisor: Mark Claypool

Company Advisor: Gabriel de Dietrich, Senior Software Developer, Qt Company

March, 2018

Abstract

Software benchmarking is measuring the performance of a piece of software via a series of tests. Benchmarking helps compare performance across platforms as well as ensure cross-platform applications function properly. The Qt Company has created a framework that enables users to develop GUI applications that on variety of platforms, but still needs a benchmarking toolkit to analyze its product. We created a benchmark for Qt which allows performance comparison on embedded systems. We researched benchmarking in general, the features that are typically found in a benchmarking suite, and the criteria for measuring the performance of a GUI platform. We created a benchmarking suite with eleven tests, four performance metrics, and additional features such as results graphing, data exporting, and the ability to run multiple tests in sequence using automated test scripts. The final suite allows users to select tests, run the benchmark, analyze and export data on a variety of platforms.

1. Introduction

Qt is a cross-platform application development framework for mobile, desktop and embedded systems [1]. Qt is used to design graphical user interfaces (GUIs) and develop applications which run on a variety of platforms. Qt is capable of porting applications to twelve different platforms including Android, iOS, macOS, Windows, Linux, and other embedded platforms.

While Qt's application functionality does not vary from platform to platform, differences in hardware capabilities may result in deviations in performance. For example, a device without a dedicated graphic card will struggle to support graphics-heavy applications. This problem is exacerbated in embedded systems compared to desktop machines with much more powerful, and larger, graphics processors. For both developers and users, it is important to be able to identify these differences and react accordingly by scaling down taxing graphical effects, number of active elements, and optimize the code.

Identifying performance differences is where software benchmarks can be useful. Benchmarks are sets of tests that attempt to measure performance for applications or hardware platforms. Benchmarks are often used to compare performance between different platforms, or to determine the capability of new hardware. Benchmarks can be used for regression testing as well; by testing different versions of the benchmark on the same platform differences between software versions can be determined.

Currently, the Qt Company has no formal, consistent performance comparison across platforms. To measure performance developers may need to make and test their own metric, which would be duplicate efforts; test on their own applications with different parameters, which may waste time; or even deploy their app and then listen to customer feedback, which may result in unhappy customers.

The goal of our project was to develop a suite of tests for the Qt Company to use as a benchmark for performance on embedded devices. We targeted embedded platforms, more specifically embedded Linux, testing performance during execution, although we also tested desktop Linux and Windows supported by Qt's cross-platform capability.

We started by coding some sample QML tests - Gradient and Particle tests - to study the Qt platform. Then, we designed and developed a framework in which multiple tests can be loaded and run. We added modules collecting data for various metrics, such as frame rate and memory usage, and support for viewing results on a graph as well as exporting results. Once the core framework is up and running we coded more tests and later on add support for exporting data to files and automated test scripts. The benchmark was tested on various platforms, including desktop Windows, Linux and embedded Linux, to verify that it functions properly.

The rest of this report is organized as follows: Chapter 2 details the background information learned in order to implement the application. Chapter 3 shows our process of coding and testing the benchmark suite. Chapter 4 provides detailed overview of the suite and its various components. Finally, chapter 5 summarizes our conclusions and mentions possible future improvements.

2. Background

This section covers our background research, including the Qt framework and how it works, general benchmarking techniques and considerations, benchmarking tools which are currently available, and our pilot tests with using Qt on embedded devices.

2.1 Qt Framework

Qt is a framework written in C++ that is used for developing applications across multiple platforms. While Qt is mainly used to create graphical applications, the Qt API is very extensive and includes data structures, networking, input/output, support for threads and more [7].

One notable feature of Qt is its use of a preprocessor, called the Meta-Object Compiler (MOC), to extend the C++ language with features like signals and slots. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources for them. Thereafter, applications/libraries using it can be compiled by any standard C++ compiler.

Qt is available under various licenses; it is available as free and open source software under several versions of the General Public License. The Qt Company also sells commercial licenses, which allows companies to develop proprietary Qt applications without license restrictions.

Qt comes with its own Integrated Development Environment (IDE) - Qt Creator. Qt Creator runs on Linux, OS X, and Windows. It offers intelligent code completion, syntax highlighting, an integrated help system, debugger and profiler integration. As part of our background research, we built several small applications using Qt Creator in order to gain more information about the framework and prepare for our project.

There are two primary ways to create a Qt application. The first is with QWidgets, which is written in C++ and builds the interface by utilizing widgets commonly found in traditional desktop applications, such as buttons, texts and checkboxes, alongside other important functionalities such as layouts for arranging widgets and a menu bar. The development toolkit of Qt comes with an interactive graphical tool called Qt Designer, which we have examined in detail and used multiple times during in preparation for our project. Qt Designer functions as a code generator for Widget-based GUIs. Qt Designer can be used stand-alone, but is also integrated into Qt Creator [1].

The second way to build Qt apps is Qt Quick. It is a framework for building dynamic and interactive interfaces with graphical effects and transitions, popular on mobile and embedded devices. In lieu of C++, Qt Quick uses QML (Qt Modeling Language), a declarative language whose syntax is similar to and indeed utilizes Javascript, to define the components that will control the interface's behavior and appearance. Backend functionality of Qt Quick applications is still written in C++, and the two parts of the application can communicate easily. The Qt Quick Controls module can provide a set of ready made QML components simulating widgets like those in QWidgets for quickly getting started on interfaces similar to existing QWidgets-based applications.

Traditionally, QWidgets has been used when building Qt applications for desktops. For applications targeting embedded systems though, Qt Quick is preferred and as such was the platform we chose for our project.

2.2 Benchmarks

When creating software it is often important to consider performance along with functionality. Performance can be measured in a number of different ways such as response time and resource efficiency. For software which runs on multiple platforms, it is beneficial to have a system for comparing performance on each platform. Benchmarks are a standard set of tests which can be used to compare software or hardware performance.

Benchmarks have to be made with certain considerations of the underlying hardware in mind in order to be unbiased and informative. Reinhold Weicker discusses how to compare different platforms, more specifically how to take certain elements of a platform in consideration when viewing benchmark results (cache size, compiler, etc.) [5]. He explains that while the user's application may be the most obvious benchmark, the results will be skewed if it is specifically optimised for a certain platform. For Qt, however, these concerns are of little importance since the Qt libraries are already compiled for the available platforms and we would be writing code using these libraries; as such we would be testing the libraries themselves instead of the underlying architecture.

Myers et al. expand on this when they list their goals for benchmarks: they must cover a wide range of styles, should be specific enough so that all implementations are alike in essential details, and must ensure good coverage. In the case of our project with Qt, this means that we needed to consider all possible embedded platforms and cover all functionality [6].

Myers et al. also list some constraints which are specific to benchmarks for user interface toolkits. Benchmarks must test: various widgets, complex dependencies between form elements, constraints on various fields, dependencies among widgets such as default values, reusability of form structures, and user control during time-consuming processes. These aspects of UI toolkits stand at the core a user's experience, so it is imperative that the program using the toolkit retains decent performance while performing the previous mentioned tasks; it is also important however to consider the direct manipulation of on screen objects, text editing, multimedia playback, and graph editing. Whereas Myers et al. created benchmarks for many different toolkits, we would be creating benchmarks for a single toolkit, Qt, on many different embedded platforms.

2.3 Benchmarking Tools

Many benchmarking tools exist on the market for system performance in general (file I/O, processor speed, etc.)., but there are few that attempt to test the performance of a specific GUI toolkit.

Qt Creator comes with a built-in QML profiling tool that can accurately measure the rendering time, CPU and memory usage of specific components within a QML program across its execution [18]. The profiler as shown in Figure 1 is useful in finding causes for typical performance problems in applications such as slowness and unresponsive, stuttering user interfaces.

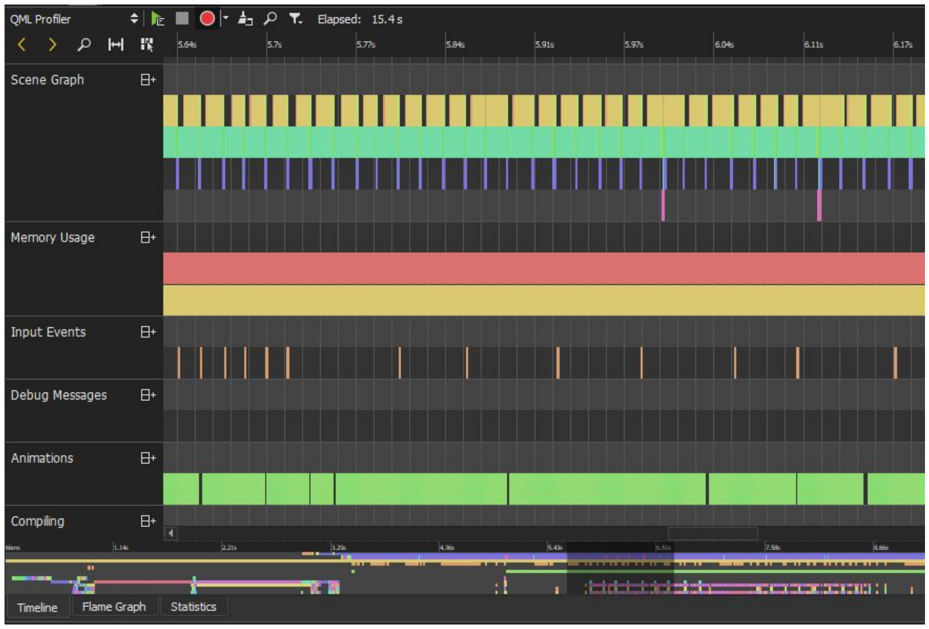


Figure 1 - The interface of Qt Creator's profiling tool.

Hormi explored the benchmarking of QML on embedded devices in his master's thesis [14]. He developed the Qt Benchmark Suite, which is a set of benchmarks that are used to measure the performance of Qt on an embedded device. The program focuses on the Qt side of performance benchmarking; essentially providing an answer to the question "How well does the device handle applications running on top of Qt?". The benchmark itself consists of three parts: the benchmark suite, a frame time measurement plugin and a result reading program.. The Qt Benchmark suite is a program that runs on a target device and goes through all the tests while recording frame time, the time it takes for a single frame to be drawn. Once all the tests have been completed, the program saves the recorded frame times, the number of elements shown/used at the time and, depending on the test, the final element amount that was run to a binary JSON file. The data can be visualized in graph form as shown in Figure 2.

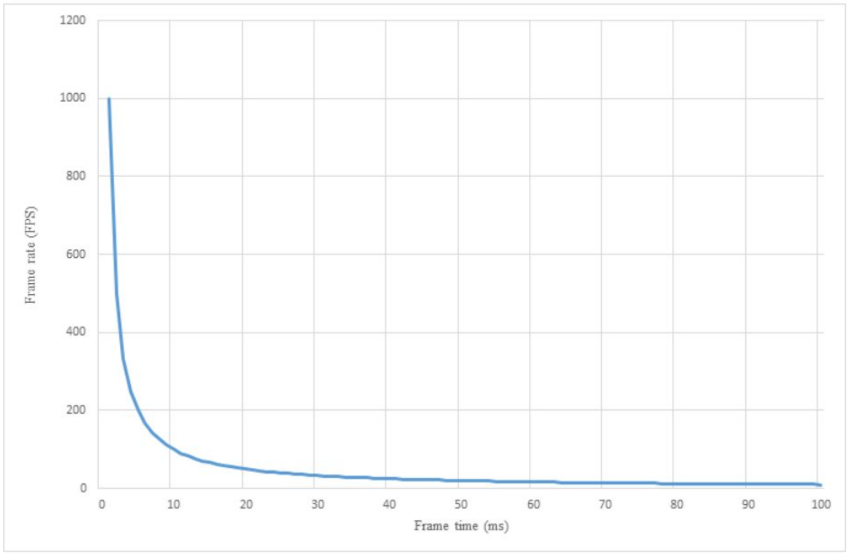
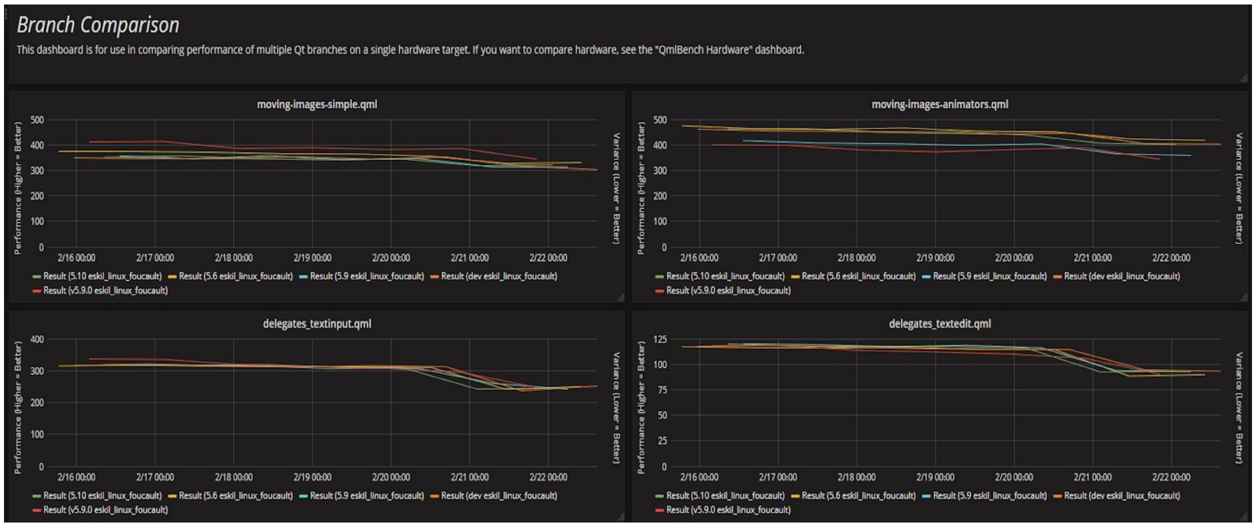


Figure 2 - Sample graph from Hormi's benchmarking tool [14]

DLL injection was used in order to record frame rate and time with high precision, directly measuring the delay between OpenGL calls. While the program and results are relevant to our interests, the code itself is not open and cannot be inspected. Furthermore its specialized nature means that it does not deal with other rendering platforms that Qt can potentially use.

qmlbench is a benchmarking tool that tests a large majority of Qt and QML features. It benchmarks by measuring the frame rate/frame time of the QML widgets currently being run. The program essentially acts as a "shell" on which QML windows can be loaded and measured, and comes with a set of prewritten benchmarks that test a wide range of QML tasks, including component creation and editing, animation, GPU-intensive graphical effects, and even the performance of the Javascript engine underneath. Data can be exported and visualized using an external program as shown in Figure 3. qmlbench is the software that is the closest to the goal of our project, but ours differ in target platforms and usability (qmlbench's command line nature is hard to demonstrate utilize on embedded platforms).

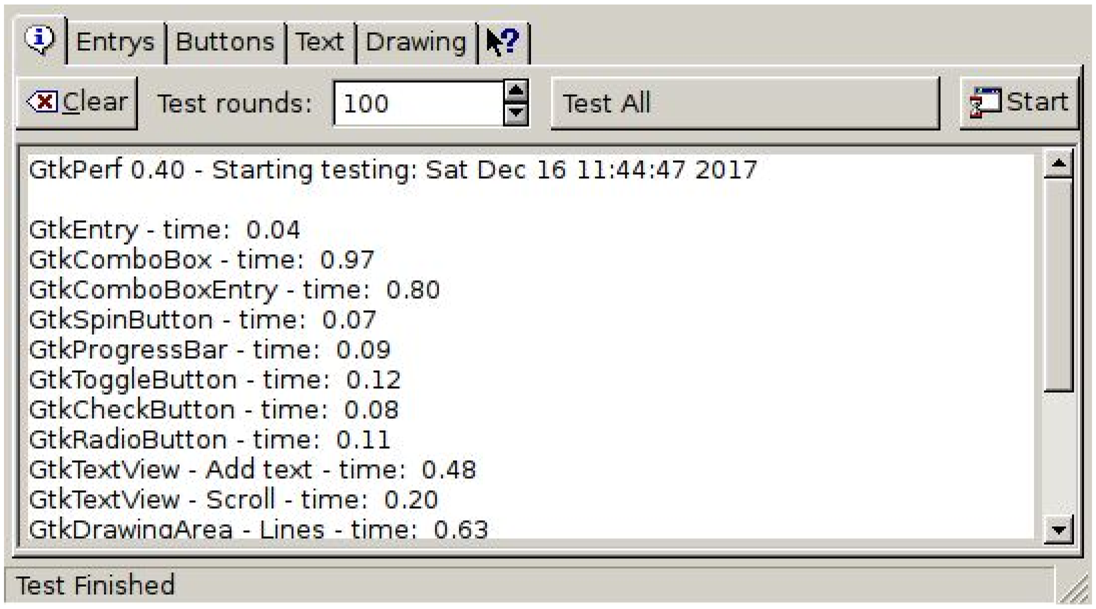

Branch Comparison

This dashboard is for use in comparing performance of multiple Qt branches on a single hardware target. If you want to compare hardware, see the "QmlBench Hardware" dashboard.

Figure 3 - Graphs showing data from four different qml tests [15]

Benchmarks in qmlbench are divided into two types: automated and manual. The automated tests are useful for regression testing, and the manual tests are useful for determining the capabilities of a new piece of hardware. The structure of the program as an extensible benchmarking platform, as well as the tests it utilized, inspired our own application [15][16][17].

Finally, GTKPerf is another benchmarking tool that deals with another widely used GUI toolkit. It tests the performance of various GTK+ 1 widgets by quickly iterating through them and measuring the time it takes for a widget to update after certain actions are performed (ticking checkboxes, scrolling through lists, etc.) as well as testing basic graphics capabilities. Its interface, as shown in Figure 4, allows for configuration of tests, such as changing the length of test or specifying the tests to be run. The tool itself however is old and lacks many features: the only tests available are widget functionality (and only a small subset of available GTK+ widgets) and basic graphics drawing capabilities. The program also only measures time, while frame rate, an extremely important criteria, is not considered [3].



```
GtkPerf 0.40 - Starting testing: Sat Dec 16 11:44:47 2017

GtkEntry - time:  0.04
GtkComboBox - time:  0.97
GtkComboBoxEntry - time:  0.80
GtkSpinButton - time:  0.07
GtkProgressBar - time:  0.09
GtkToggleButton - time:  0.12
GtkCheckButton - time:  0.08
GtkRadioButton - time:  0.11
GtkTextView - Add text - time:  0.48
GtkTextView - Scroll - time:  0.20
GtkDrawingArea - Lines - time:  0.63
```

Figure 4 - GTKPerf showing results after a test on a Ubuntu Linux system

2.4 Embedded Systems

For commercial applications, Qt provides a set of tools called Qt for Device Creation that significantly streamlines the process of deploying to embedded devices. Qt for Device Creation includes Boot2Qt, a custom Linux installation containing the full Qt libraries that can be flashed to certain devices as well as support for cross-compiling to embedded operating systems, including QNX, VxWorks and ARM Linux (used by the aforementioned Boot2Qt stack). For non-supported platforms, the Qt library may have to be compiled manually [4].

Qt applications on embedded platforms can use one of many available platform plugins that control how they are drawn to the screen [8]. For Linux specifically, the most commonly used and recommended plugin is EGLFS, which uses EGL and OpenGL ES 2.0 to draw the program full screen without needing to use any underlying windowing system. If the system requires multiple apps running at a time, the X server or Wayland may be used, which provides a windowing system for drawing windows to the screen.

Unfortunately, we have had no experience with QNX and VxWorks due to their proprietary licenses. Nevertheless, with Qt's cross-platform support via Qt for Device Creation the code used by the benchmarking program should not significantly change if running on another system.

3. Procedure

This section describes our process for the benchmark suite over the course of the project.

3.1 Tests

The first test created was the Particle Test. The Qt documentation on Performance Considerations and Suggestions mentions Qt Quick Particles as a potential cause of poor performance [2]. We discovered a bug which, upon attempting to edit the emission rate on the same frame a particle is destroyed, crashed the application. This issue was later avoided by setting a maximum number of particles.

We created a basic gradient test as an alternate to the Particle Test. The same documentation on Performance Considerations and Suggestions mentions that gradients should be used primarily in static situations, where they are not required to be recalculated. With two tests completed, a single Test Suite Application from which we could run both of our tests was created to measure the first performance metric, Frame Rate. Our process for creating the test suite and metrics is described in greater detail in sections 3.2 and 3.3.

As an alternative to creating Gradients in a Canvas, which may be unnecessarily taxing on a system and might skew the results, we created two new tests to replace the basic gradient test: GradientTest, which consists of one Rectangle filled with a gradient with multiple GradientStops; and GradientRectangleTest, which consists of multiple Rectangles with simple Gradients, rather than one Rectangle with a complex Gradient like GradientTest.

We created a system for automating tests. The system used an automatedTest flag, and some properties which defined the behavior of the test such as start value, maximum value, duration, and the interval between updates to the value. The automated tests worked, but as we added more tests and added automation capabilities to each, we were dealing with a large amount of duplicate code, since every tests require boilerplate code to set up components creation as well as hooks to integrate with the rest of the program, such as functionality as simple as an exit button.

To solve the issues with code duplication, we created the QML equivalent of a base class which the tests inherit from. Since there is no true inheritance in QML, our solution was to use the Base QML object as the root of each test. This approach was also done with the ManualCreationTest, a subclass of Base which is used for tests that create and destroy components as the test variable changes. We were able to move the automated test management, basic UI, and the color changing variable into Base.qml, which nearly eliminated any duplication in the test QML files. This came with the added benefit of allowing us to more easily change default values such as test length and the test variable's range.

3.2 Metrics

Frame rate was the first metric measured. It had two main components, a C++ class and a QML object. The C++ class contained some custom geometry which was drawn every frame, a counter that was incremented for each repaint, and the calculations for frames per second (fps). The QML object held a text object which displayed the current fps in the corner of the screen.

In order to support a larger number of metrics, we created the Metric base class, the class which all metrics must derive from. The Metric class includes three virtual functions which can be overridden in the child classes. These functions are *startTest*, *measure*, and *endTest*. *startTest* and *endTest* are there for metrics which require some extra setup or tear down between test sessions. Overriding those two is optional, but *measure* is abstract and therefore must be overridden. *measure* is called at set time increments throughout the test and returns the data readout of the metric at that time. The value returned is stored in a *QVariantMap* so that it can be easily converted into a JavaScript object for use in QML.

3.3 Test Suite

As mentioned previously, the test suite was originally created after the completion of the Particle and Gradient tests. It allowed us to easily swap between tests and share components between tests. In the original version, we needed to manually populate the ComboBox holding the names of the tests. One of the first improvements to the suite was to automatically populate this list by checking the TestApplications directory for any file named Test.qml. Each test is put in a folder containing the test's name, compartmentalizing any additional files needed for the tests.

The earliest tests each included their own *Exit* button. To eliminate the need for this unnecessary duplication and to remove the test creation logic from the Home screen, the TestEnvironment was created, serving as the staging ground for tests. Originally it managed the different metrics, but this responsibility was later handed off to the MetricManager. The creation of the TestEnvironment was the first step towards modularizing the test suite and removing responsibility from the Home screen.

To view the results of the tests when they are completed, the Results screen was created. This screen was populated with a ChartView showing the data recorded during the test. The original Results screen was somewhat rudimentary, with no legend and no way of toggling data sets. This version of the Results screen was not capable of changing the range of the y-axis, which was not a problem when the only metric was frame rate with its maximum possible value of 60 fps. When new metrics were added the graph became unreadable especially when large values (such as virtual memory usage in bytes) were being compared with small values (such as frame rate).

To make the graph usable, we allowed the user to control the display. A legend area with CheckBoxes next to the data sets' names was introduced. These CheckBoxes toggle the visibility of the data set and resets the scale of the y-axis. Upon toggling a CheckBox, the *regenerateAxis* function looks for the minimum and maximum values of the active data sets, scales to those values, and then uses the *applyNiceNumbers* function to make the graph more informative and readable.

For more advanced analysis using an external program, the test data needed to be able to be exported to a readable format. As such, the CsvWriter class and the associated UI elements (the *Export* button present in the Results screen and the export location selector in the Home screen) was created. The CsvWriter produces an easy to read spreadsheet of the data collected during the test, complete with a header containing the names of the metrics whose data is present. It writes the file to whichever directory the user selects, or the TEMP directory by default. The name of the file created corresponds to the name of the test, with the time of the test’s completion appended to the end, to ensure that tests run in succession are easily identifiable.

3.4 Test platforms

The application was ran on multiple platforms to test its functionality. Our own development laptops running Windows and desktop Linux was tested for functionality on a desktop system. For embedded systems two pieces of hardware was tested - a Raspberry Pi 2 running Raspbian, a Linux distribution, with Qt compiled from scratch to enable EGLFS support, and a Freescale i.MX 6 board running Boot2Qt, which already contains a Qt installation. The use of the embedded devices allows us to utilize the program as it is intended, see if there are any noticeable differences in implementation, as well as find bugs that are otherwise not present on desktop platforms. One such notable bug was a race condition that may cause the metric system to fail to initialize the frame rate component on slower embedded systems, which was fixed by making sure that the Frame Rate metric checks that all pointers are properly initialized before measuring.

4. The Qt Benchmark Suite

This section gives an overview of the benchmark program, its various components, how they work and their functionality.

4.1 Home

The Home screen (Figure 5) acts as a hub which interacts primarily with the TestEnvironment, and Results screen. It handles the flow of events in the *startTest*, *returnToHomeScreen*, and *exportData* functions. The menu is composed of a Switch, a ComboBox containing the names of the tests, a Button for starting the test, an ExportPathSelector for selecting the export location, and another Button for selecting an automated test script to run. The Home screen contains a testQueue, which is filled by either pressing the *Start* button, or selecting an automated script to run. This queue allows multiple tests to be run in sequence.

There are two primary methods for testing. The user can either run individual tests, or write a script which runs one or more tests. If the user wants to run an individual test, they simply choose a test from the drop-down list, and press *Start*. If the user would like to run an automated version of the test, they need only toggle the switch beside the drop-down list. If the user intends to use the raw data from the test, they can specify a location to export the test data to and after the test, press the *Export* button in the top-right corner of the Results screen.

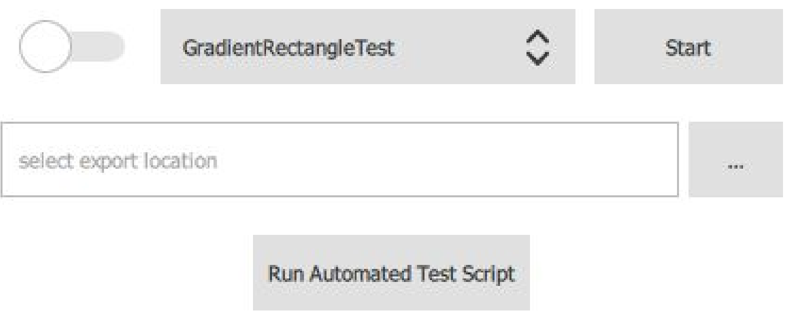


Figure 5 - The Test Suite's home screen

If they would instead like to run a script, they can specify an export location for test data (if none is provided, the TEMP directory is used), press the *Run Automated Script* button, and after selecting a script to run, the tests will automatically begin.

4.2 TestEnvironment

The TestEnvironment handles the creation and execution of tests. The *Exit* button at the top-right corner of the screen is the only UI element in the TestEnvironment. Tests can be ended by either pressing the *Exit* button, or waiting until the end of an automated test. Both of these options result in the *endTest* function being called. This function disables and hides the UI, destroys the test object, and emits the *endTestSession* signal. The Home screen accepts this signal and responds by either opening the Results screen, or continuing onto the next test if an automated test script is running.

4.3 Metrics

The application includes a set of performance metrics that are recorded every second for the duration of the test. We have created four metrics: Frame Rate, CPU Usage, Virtual Memory Usage, and Physical Memory Usage. Additional metrics can be added by creating a C++ class that inherits from the *Metric* base class and appending it to *m_metrics*, a *QList<Metric*>*, within the *MetricManager* class's constructor.

4.3.1 Metric

Metric is the base class from which all metrics must inherit. It contains a number of member variables: *m_name* for identifying the *Metric* and activating it through the automated test scripts, *m_data* for storing the data from the test, and some others for rescaling graphs in the Results screen (*m_maxX*, *m_minY*, and *m_maxY*). All these member variables have Q_PROPERTY macros attached to them, which allows them to be interacted with from QML. The virtual methods *startTest* and *endTest* may be overridden if additional preparation or clean-up is necessary. The abstract method _*measure* must be overridden. This method defines the metric.

4.3.2 The Metrics

- FrameRate (Figure 6): Measures the number of frames rendered per second. In optimal cases the frame rate would equal the refresh rate of the monitor (which is 60Hz on all tested systems, translating to 60fps). It is measured by creating a custom QML component that counts the number of times it was redrawn per second.

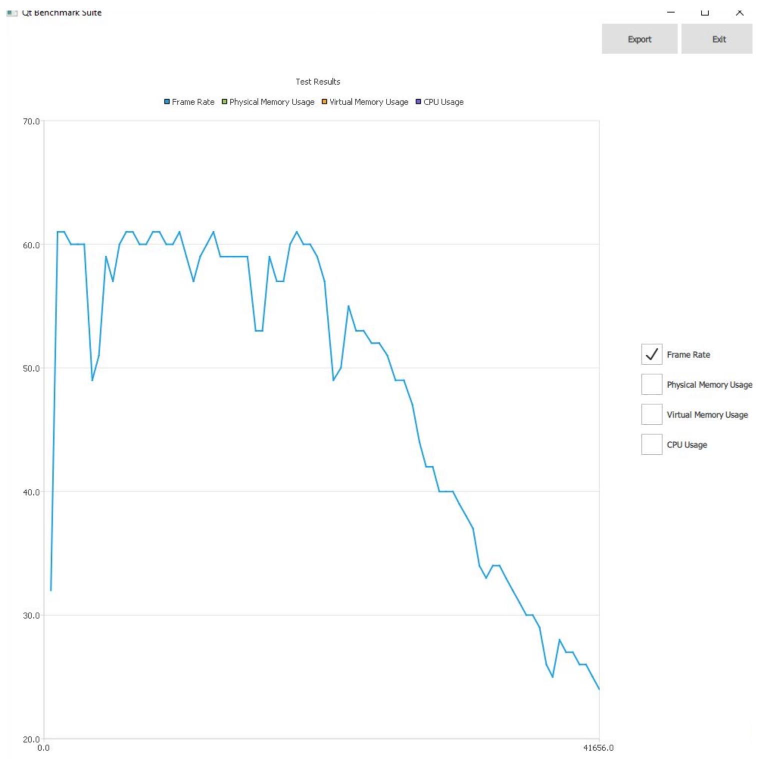


Figure 6 - Result graph of frame rate (in frames per second) against time (in miliseconds)

- PhysicalMemoryUsage (Figure 7): Measures the amount of physical memory used. This is the actual amount of memory on RAM taken up by the process.

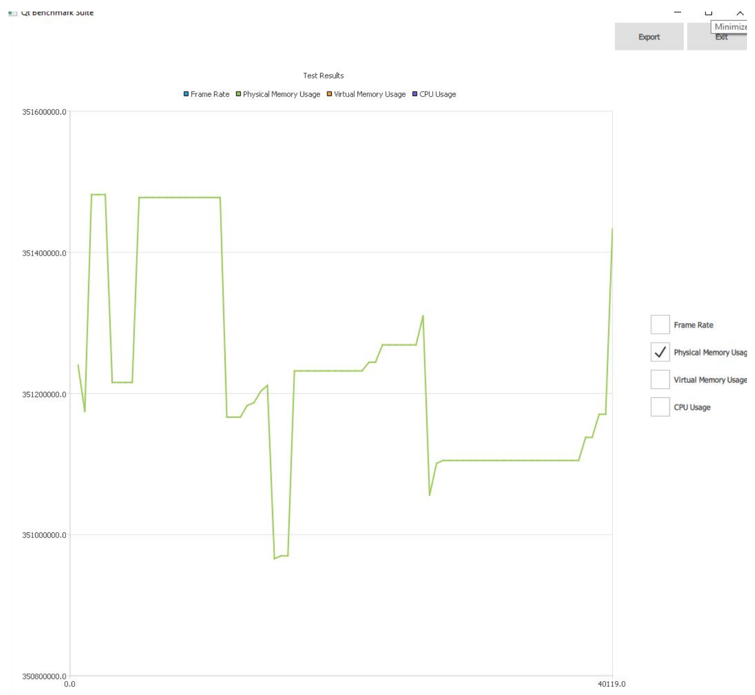


Figure 7 - Result graph of physical memory usage (in kilobytes) against time (in miliseconds)

- VirtualMemoryUsage (Figure 8): Measures the amount of virtual memory used - the size of the address space requested by the process. This is typically much larger than physical memory usage, since not all of virtual memory is in RAM. The rest of the memory space that is not actively being utilized is swapped to storage and can be retrieved as necessary.

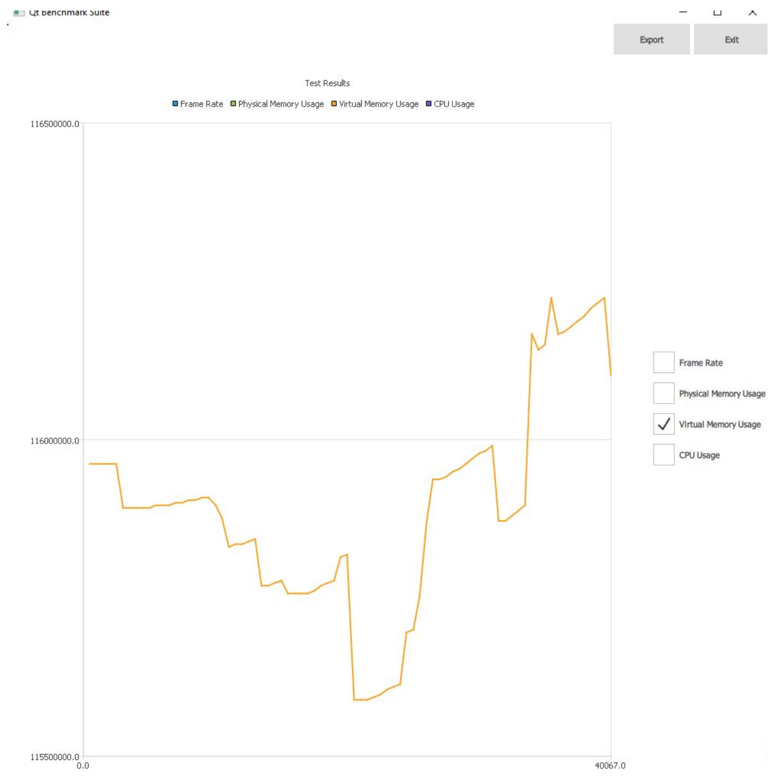


Figure 8 - Result graph of virtual memory usage (in kilobytes) against time (in miliseconds)

- CPUUsage (Figure 9): Measures the percentage of the CPU dedicated to processing the application. It is measured by dividing the CPU time of the application by the system's total CPU time. On a typical quad-core system, at maximum usage a single-threaded process can take up to 25% of total CPU time, equal to one single core running the thread full time.

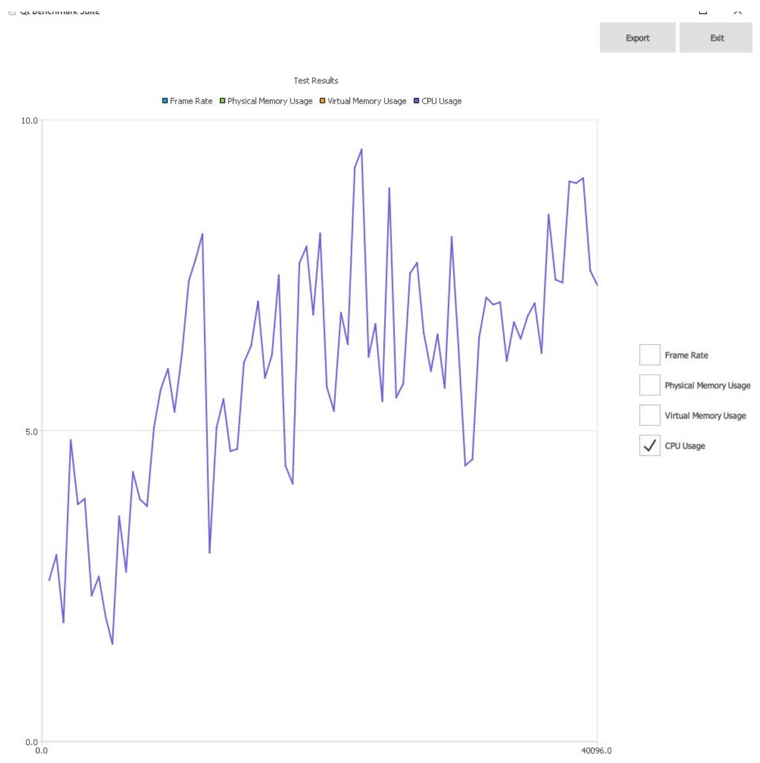


Figure 9 - Result graph of CPU (in percentage) against time (in miliseconds)

FrameRate is measured using built-in QML components and is cross-platform, while the remaining three metrics are dependent on the operating system. Currently Windows and Linux are supported.

4.3.3 MetricManager

The MetricManager is a C++ class which maintains the list of active metrics, starts and stops test sessions, and measures their values throughout test sessions. It has two member variables, two *QList<Metric*>*'s. The first list, *m_metrics*, is the master list of Metrics. The other is *m_activeMetrics*, which contains a subset of m_metrics based on which Metrics the user would like to track. This class has a number of public methods with the *Q_INVOKABLE* macro, allowing them to be called from QML. These methods' names mimic the methods in Metric and run through each *Metric* in the active metrics list, calling the *Metric* method with the same name. For example, calling *MetricManager::startTest()* results in *Metric::startTest()* being called in every *Metric* in *m_activeMetrics*.

4.4 Tests

We have created several tests with which to test Qt's performance. Each test is defined by a QML file named Test.qml which is placed in a directory bearing the name of the test. All of these directories are present in the TestApplications directory. The tests all have Base.qml as their root object (or another child of Base such as ManualCreationTest), which contains almost all the test logic. Some of the simplest tests consist of just a few definitions of properties inherited from ManualCreationTest.

4.4.1 Base

Base is the class that tests are based on. Its properties include the name and default value of the test variable, some properties related to automated tests such as target value, length of test, and updates per second, and some additional properties for color changing. It also includes some UI elements and the timer for running automated tests.

4.4.2 ManualCreationTest

ManualCreationTest is a subclass of Base that provides an easy way to write generic component creation tests with buttons to increase the number of components in manual mode or an increasing number of components over time while running in automated test mode.

4.4.3 TestVar

TestVar is a QML object which can be used to manage test variables and generate the parts of the test's UI which relate to those variables. TestVar is composed of a name to display, the default value, the minimum and maximum values, and the amount that the value should change when the add or remove buttons are pressed. There are two Components inside the TestVar. The first is the textComponent. This Component is used to create the UI of automated tests, where the buttons are not present. The other Component is the uiComponent, which includes add and remove buttons, as well as the text information present in the textComponent. Upon pressing one of the buttons, the current value is compared to the min and max values, and if valid, the current value is reduced or increased by the value of the delta property.

4.4.4 Tests

The Qt Benchmark Suite tests are initially divided into eleven different benchmark groups, each having a different purpose for performance analysis. More tests can be coded as needed.

- GradientRectangleTest (Figure 10) was created to see what effect the creation of multiple rectangles filled with gradients would have on performance for different devices.

  The test uses QML Rectangle Type to create items. It can have a background of solid color or a gradient. For the GradientRectangleTest, we decided to use gradients for each rectangle defined by two colors that will be blended seamlessly [9][10].

  The colors are specified as a set of GradientStop child items. Each of them defines a position on the gradient from 0.0 to 1.0 and a color. The position of each GradientStop is defined by its position property. The color is defined by the GradientStop color property.

  We have placed two GradientStops for each Gradient Type at position 0 and 1. The gradient colors keep changing every 3 seconds as we used Sequential Animation, which allows animations defined to be run one after the other. Sequential Animation is dependent on the current color that can be either of red, orange, yellow, green, blue, indigo or violet. The first GradientStop has the current color set as a color property, while the second one has a darker version of the current color [11].

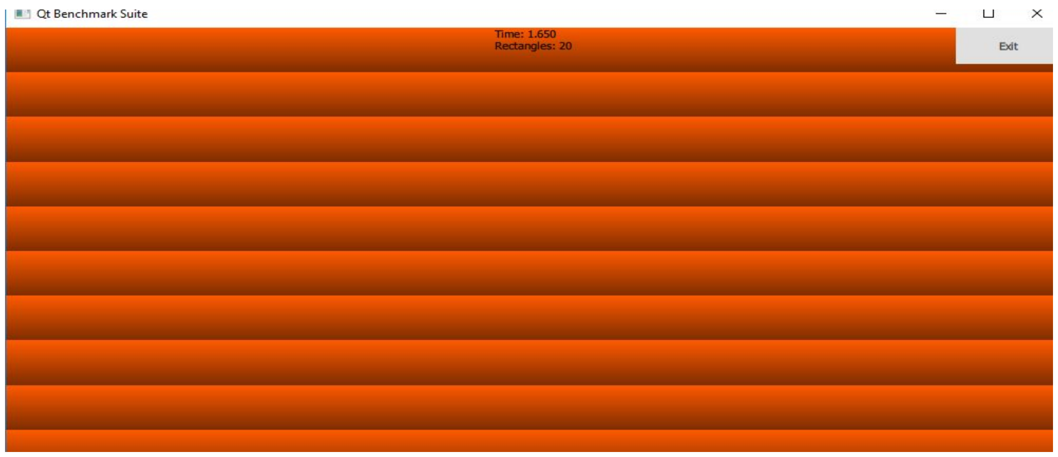


Figure 10 - GradientRectangleTest in automated mode, showing multiple rectangles colored with gradients

- GradientTest (Figure 11) was created to see what effect the creation of multiple GradientStops has on performance for different embedded devices and how it compares to the results of GradientRectangleTest. The test uses QML Rectangle that fills the parent window with a Gradient and further divides it into subsections using GradientStop. The same Sequential Animation principle used in GradientRectangleTest is applied to the GradientTest to change the colors of the gradient.

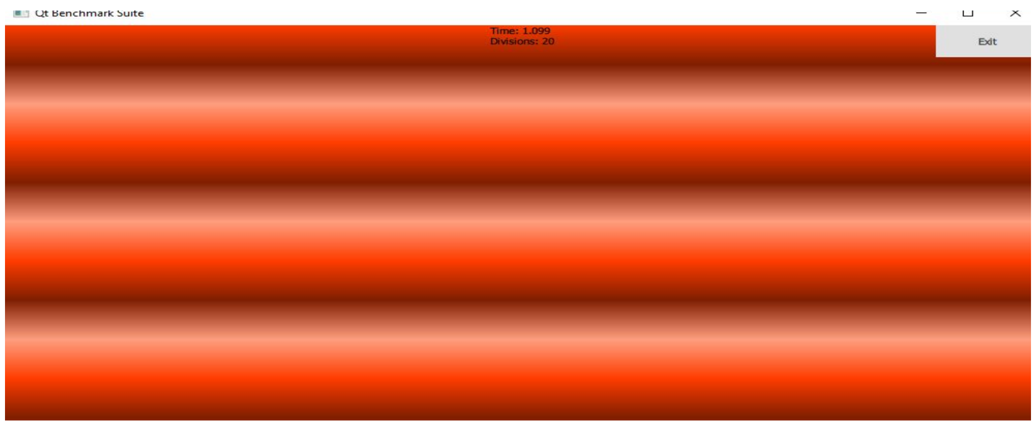


Figure 11 - GradientTest in automated mode, showing a single rectangle colored with multiple gradients

- ImageRotateTest (Figure 12) was made to see how performance would be affected by a large amount of animating images running concurrently. ImageRotateTest creates multiple images that infinitely rotate a full 360 degrees.

At its core is the Image component with a NumberAnimation applied on the rotation property of the image to create the rotation animation.

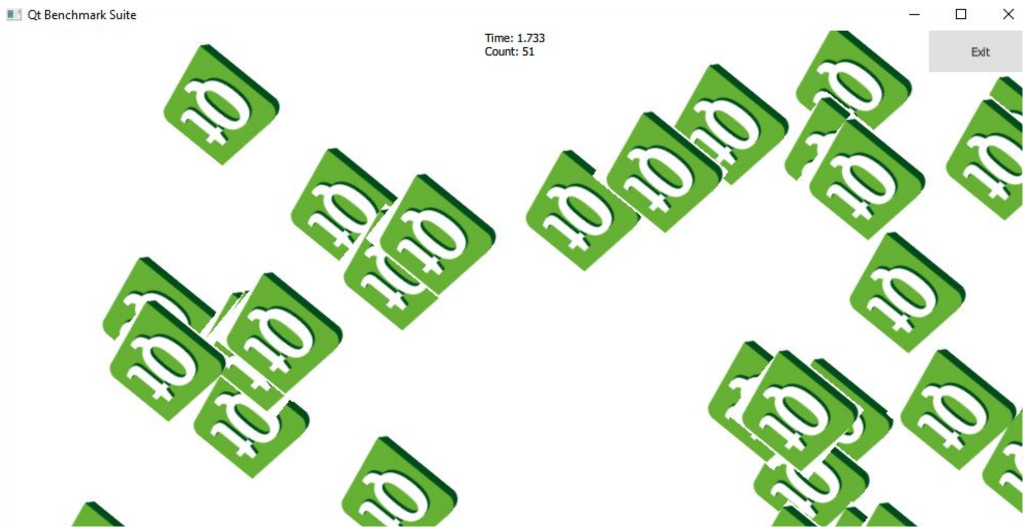


Figure 12 - ImageRotateTest in automated mode, showing several images (the Qt logo) rotating

- ListViewTest (Figure 13) was created to see if the performance slowed down when scrolling through the ListView of rectangles with gradients and texts. ListView displays data from models created from built-in QML types. Our model is a component that is a Rectangle with a gradient and text “Hello Qt! I am new gradient” [12].

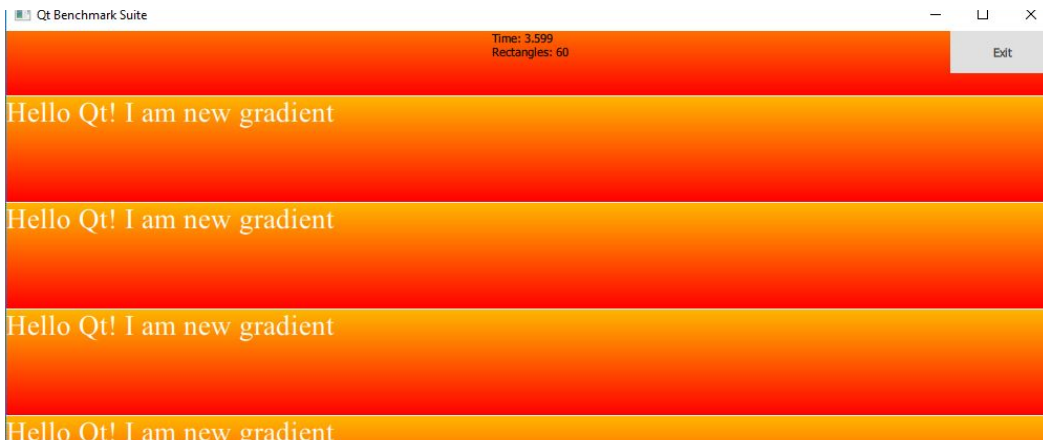
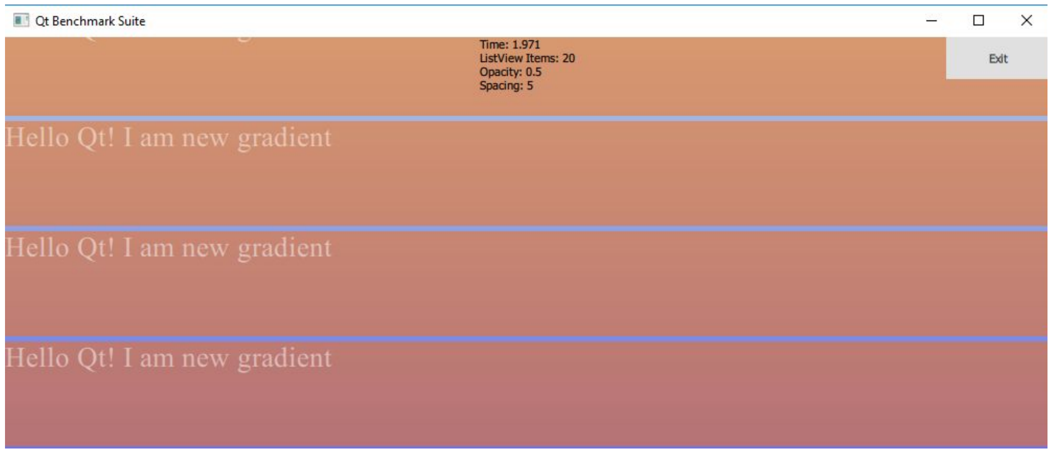

Qt Benchmark Suite
Time: 3.599
Rectangles: 60
Exit
Hello Qt! I am new gradient
Hello Qt! I am new gradient
Hello Qt! I am new gradient
Hello Qt! I am new gradient

Figure 13 - ListViewTest in automated mode, showing a list of gradient-colored rectangles

- RectangleOpacityTextTest (Figure 14) was made to see the difference between the performance of gradient and semitransparent rectangles. The implementation of these two tests differs because RectangleOpacityTextTest's has a complex background and also an ability to add spacing between the rectangles as well as change the opacity of them. The complexity of the background is depicted by filling it with a gradient. The rectangles are not filled with gradients but have a color, which changes every 3 seconds by the use of Sequential Animation. The colors can be either red, orange, yellow, green, blue, indigo or violet.



Time: 1.971
ListView Items: 20
Opacity: 0.5
Spacing: 5

Hello Qt! I am new gradient

Hello Qt! I am new gradient

Hello Qt! I am new gradient

Figure 14 - RectangleOpacityTextTest in automated mode, showing a list of rectangles with 0.5 opacity

- ParticleTest (Figure 15) was created to see how the performance can be affected when the screen is filled with different types of Rectangles such as round or random size with fixed color or filled with a gradient.

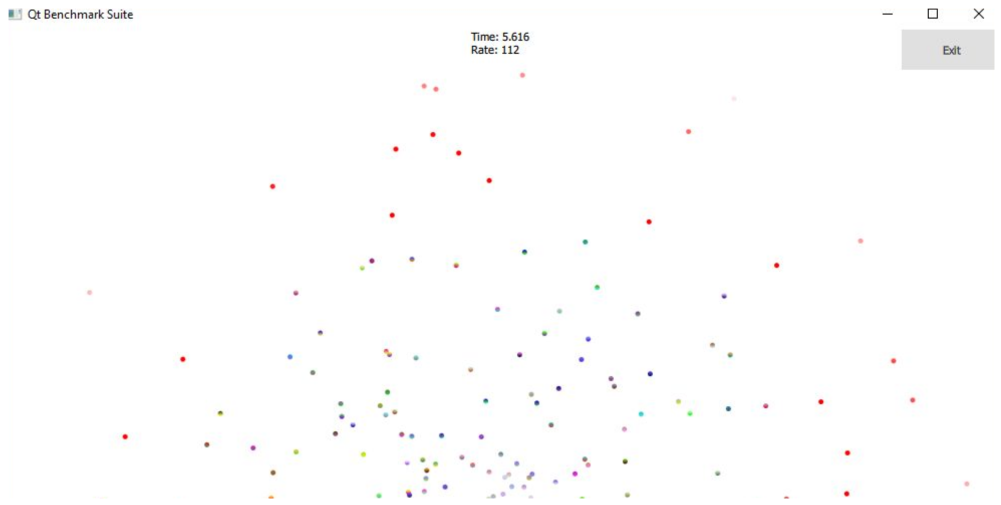


Figure 15 - ParticleTest in automated mode, with particles colored by a gradient

- MultiTest is essentially ParticleTest and GradientTest running concurrently, each on different windows. It is intended to see if performance of a program doing several drawing tasks simultaneously on different windows is different than just one task on one window.
- ParticleWithFileAndNetworkTest is identical to the Particle Test, with the added tasks of file writing and network i/o in the background to see if graphical performance is affected by such tasks running in the background. Aside from simulating particles, the test will also periodically read and write a 20MB file to the temp folder, with a wait time of 1 second between each reading/writing, as well as open a TCP socket on a server and write to it continuously. The server is a separate program that must be run before the test.
- TextCreationTest (Figure 16) was made to test the creation of a large number of text elements. The Text components spawn with a random x and y coordinates and all have identical contents.

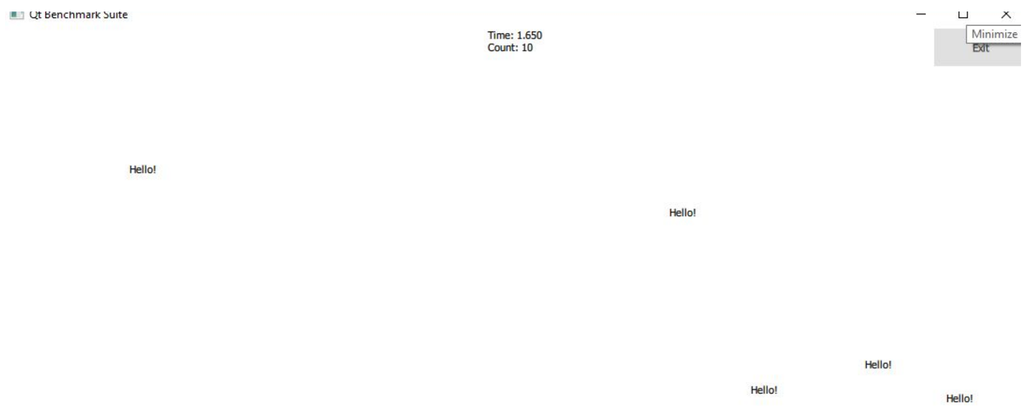


Figure 16 - TextCreationTest in automated mode, showing several text items

- TextChangeTest (Figure 17) is identical to TextCreationTest, but now the text elements can be changed, and was intended to test the changing of text contents. In manual mode there is a text input at the bottom that will change the content of every text element currently being drawn. In automated mode all text elements are periodically changed to a random alphanumeric string.

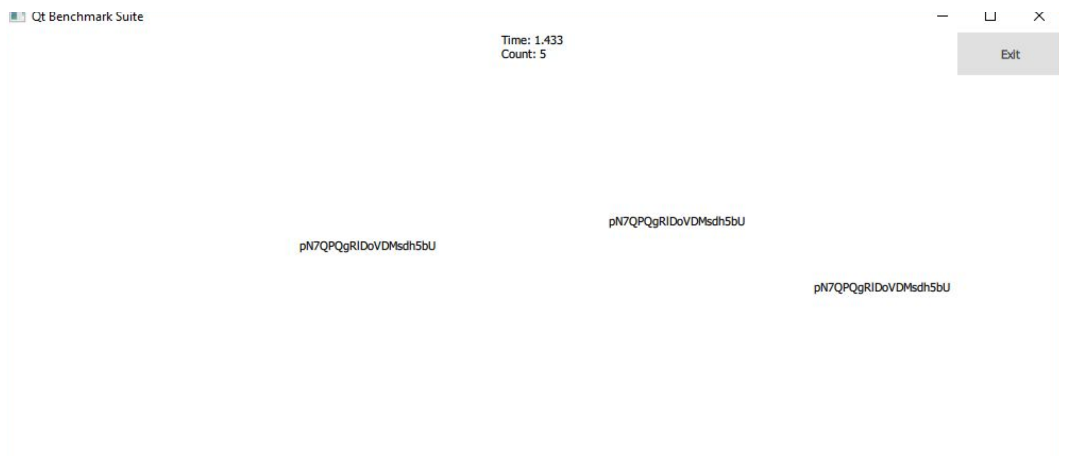


Figure 17 - TextChangeTest in automated mode, showing several text items displaying random string

- WindowTest (Figure 18) was created to see how the creation of multiple child windows affects performance. A Window QML type creates a new top-level window is our Component. The window contains a text depicting the number of the windows created. It is important to note however that this test will not run on embedded systems using the eglfs platforming plugin, due to the plugin drawing only allowing one fullscreen root window. The test will run on desktop operating systems (such as Windows) or embedded Linux using an underlying windowing system, such as X11 or Wayland [13].

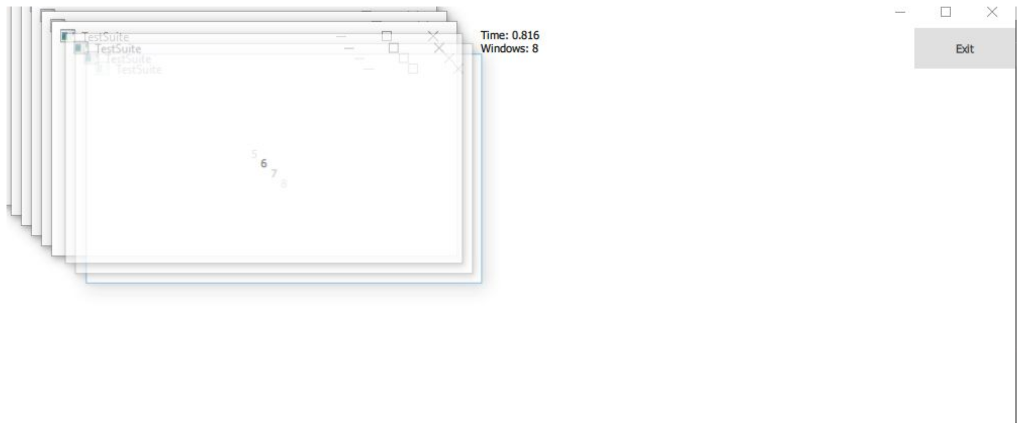


Figure 18 - WindowTest in automated mode, showing a cascade of child windows being created

4.5 Results

The Results screen (Figure 19) uses Qt Quick Charts to display the data as a plot against time. The data lines for each metric can be toggled on and off, causing the axes to scale accordingly. The user can choose to export the data to the specified export location before exiting to the home screen.

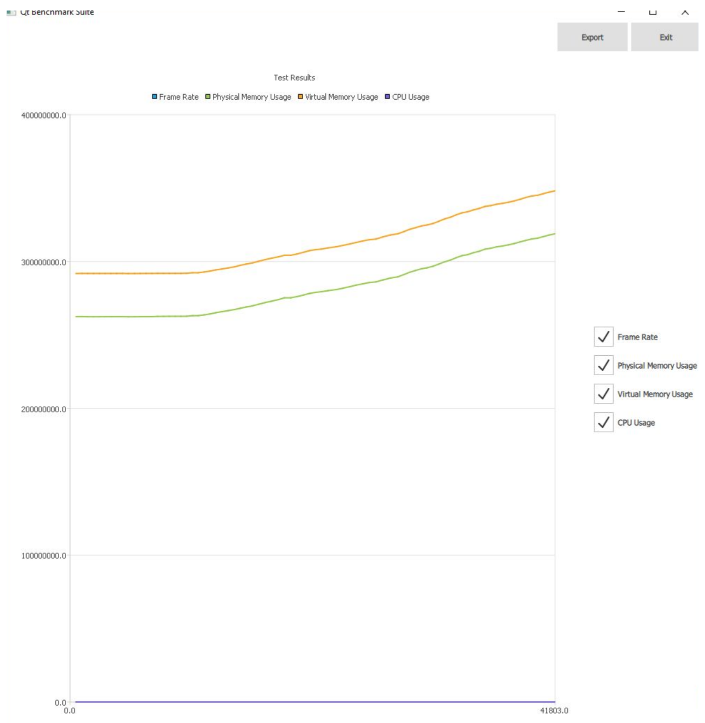
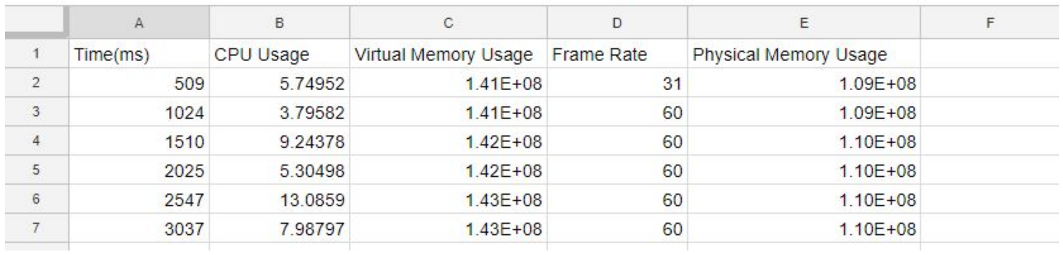


Figure 19 - The Results screen

4.6 DataExporter

4.6.1 CsvWriter

The CsvWriter class writes the test data into a file formatted with comma-separated values. This includes the timestamp and the metrics for that timestamp, which can include frame rate, memory and CPU usage. The user can specify the export directory on the home screen and when a test finishes can export the data at the results screen. The exported data can then be opened with a suitable spreadsheet program as seen in Figure 20.



	A	B	C	D	E	F
1	Time(ms)	CPU Usage	Virtual Memory Usage	Frame Rate	Physical Memory Usage	
2	509	5.74952	1.41E+08	31	1.09E+08	
3	1024	3.79582	1.41E+08	60	1.09E+08	
4	1510	9.24378	1.42E+08	60	1.10E+08	
5	2025	5.30498	1.42E+08	60	1.10E+08	
6	2547	13.0859	1.43E+08	60	1.10E+08	
7	3037	7.98797	1.43E+08	60	1.10E+08	

Figure 20 - Exported data as seen in a spreadsheet program

4.7 AutoTestParser

For automated, sequential executions of tests, a JSON file can be supplied to be automatically parsed at the home screen. The JSON file can have the following objects:

- Metrics: An array that determines which metrics, denoted by strings, to be measured during the text. During run time the application will compare the name of available metrics to this list, and if the metric is listed it will be measured.
- Tests: A array that lists the tests that will be sequentially run one after another. The items of the array are objects with a”Name” member listing the name of the test to be run, and a “Parameters” member showing the options to be applied to the test.

5. Conclusion

Qt is a framework providing a library for developing cross-platform software applications. Currently, there is no way to formally compare the performance of the Qt framework on different platforms, including embedded systems. A Qt benchmark could help assess the performance of the library as it is developed, as well as help others developers analyze the performance of their own programs.

We created a benchmark suite that provides a framework for the performance testing of the library. We identified various Qt components that potentially stressed system performance and created tests for them. We designed and developed a framework to run those tests that measures system load in the form of frame rate, CPU and memory usage and handles the returned data. We also ran the applications on a variety of platforms to test its functionality.

The result is a application that can accurately report performance for graphically taxing effects on any platform that runs the full Qt library. The framework is extendable to new tests and support for other features such as results viewing and exporting.

While the application allows for Qt benchmarking, there is potential future work that can improve it. Right now the set of tests is rather small and more tests could cover more QML features. Gradients, particles, texts and image animation are covered in the initial tests, but other graphical effects such as 3D rendering could be tested as well. Non-graphical and CPU heavy calculations could also be considered.

The set of performance metrics measured, which includes frame rate, CPU usage and memory utilization, could also be expanded upon. As an example, measuring GPU usage could make the tests much more informative and useful, especially considering the heavy focus of the test suite on graphical effects.

Performance optimizations could also be considered. One example is the results graph rendering which is currently implemented in Javascript. Moving such functionalities to C++ could improve performance and prevent the benchmarking program from interfering with performance tests. Additional features such as built in results analysis or the ability to toggle metrics inside the main application could also be created.

References

1. *About Qt* | *Qt*, retrieved Nov-2017. Available: https://wiki.qt.io/About_Qt
2. *Performance Considerations and Suggestions* | *Qt*, retrieved 5-Mar-2017. Available: http://doc.qt.io/qt-5/qtquick-performance.html.
3. *GTKPerf* | *GTKPerf,* retrieved Nov-2017. Available: http://gtkperf.sourceforge.net/
4. *A modern guide for cross-compiling Qt for HW accelerated OpenGL with eglfs on Raspbian and setting up Qt Creator* | *Qt,* retrieved Nov-2017. Available: https://wiki.qt.io/RaspberryPi2EGLFS
5. *An Overview of Common Benchmarks* | *Weicker, Reinhold P.*, 1990, Siemens Nixdorf Information Systems, retrieved Nov-2017.
6. *Using Benchmarks to Teach and Evaluate User Interface Tools* | *Myers, Brad A., et al.*, Carnegie Mellon University, retrieved Nov-2017
7. *Qt Core 5.10* | *Qt,* retrieved Feb-2018. Available: https://doc.qt.io/qt-5.10/qtcore-index.html.
8. *Qt for Embedded Linux.* | *Qt,* retrieved Feb-2018. Available: http://doc.qt.io/qt-5/embedded-linux.html.
9. *Rectangle QML Type* | *Qt Quick 5.10,* retrieved Jan-2018. Available: http://doc.qt.io/qt-5/qml-qtquick-rectangle.html.
10. *Gradient QML Type* | *Qt Quick 5.10,* retrieved Jan-2018. Available: http://doc.qt.io/qt-5/qml-qtquick-gradient.html.
11. *SequentialAnimation QML Type* | *Qt Quick 5.9,* retrieved Feb-2018. Available: https://doc-snapshots.qt.io/qt5-5.9/qml-qtquick-sequentialanimation.html.
12. *ListView QML Type* | *Qt Quick 5.10,* retrieved Jan-2018. Available: http://doc.qt.io/qt-5/qml-qtquick-listview.html.

13. *Window QML Type | Qt Quick 5.10,* retrieved Feb-2018. Available: http://doc.qt.io/qt-5/qml-qtquick-window-window.html.
14. *Qt benchmark suite for embedded devices* | K. Hormu, retrieved Sep-2017. Available: http://jultika.oulu.fi/files/nbnfioulu-201710112978.pdf.
15. *Performance regression testing of Qt Quick* | *Qt Blog*, 27-Apr-2017, retrieved Nov-2017. Available: http://blog.qt.io/blog/2017/04/27/performance-regression-testing-qt-quick/.
16. *Grafana*, retrieved Nov-2017 Available: https://testresults.qt.io/grafana/dashboard/db/qmlbench-branches?orgId=1&var-suite=All&var-benchmark=All&var-branch=All&var-hardwareId=eskil_linux_foucault.
17. *README.md - qt-labs/qmlbench.git - Tool for benchmarking frame rate of Qt Quick* | *M. Curtis and G. Sletta*, retrieved Jan-2018. Available: http://code.qt.io/cgit/qt-labs/qmlbench.git/tree/README.md.
18. *Profiling QML Applications* | *Qt Creator Manual*, retrieved Nov-2017. Available: http://doc.qt.io/qtcreator/creator-qml-performance-monitor.html.