Code less.
Create more.
Deploy everywhere.

Ensuring Maximum Performance with Qt

Modern user interfaces are not only a tool for the user to interact with an application, they must also be pleasing to the eye. This means that developers can attract more users with eye candy and sleek transitions. At the same time the market is moving in a direction where portability becomes more and more important. Netbooks and advanced portable terminals and handsets are expected to deliver attractive user interfaces while providing less number crunching power than the average desktop PC. To facilitate this, one of the key aspects Qt Software focused on with the 4.5 version is to greatly improve performance in Qt.

These improvements have affected most parts of Qt. This paper will show different parts of Qt as well as specific cases. Just to tease your imagination, see Figure 1, showing some of the improvements to the backing store and their impact on performance for opaque widgets. Reading on, you will find more charts like this and tips to make your code perform even better.

To be able to enjoy these performance improvements, it is important for you to understand how Qt applications spend their CPU cycles. This white paper contains tips about what to focus on to ensure performance as well as QtBenchLib, which was introduced in Qt 4.5. QtBenchLib allows you to add performance tests to your test suites, thus getting a better picture of where you need to focus your efforts.

NOKIA

Ensuring Maximum Performance with Qt

Modern user interfaces are not only a tool for the user to interact with an application, they must also be pleasing to the eye. This means that developers can attract more users with eye candy and sleek transitions. At the same time the market is moving in a direction where portability becomes more and more important. Netbooks and advanced portable terminals and handsets are expected to deliver attractive user interfaces while providing less number crunching power than the average desktop PC.

To facilitate this, one of the key aspects Qt Software focused on with the 4.5 version is to greatly improve performance in Qt. While Qt 4.5 also contains a number of new features, this paper will focus purely on performance.

These improvements have affected most parts of Qt. This paper will show different parts of Qt as well as specific cases. Just to tease your imagination, see Figure 1, showing some of the improvements to the backing store and their impact on performance for opaque widgets. Reading on, you will find more charts like this and tips to make your code perform even better.

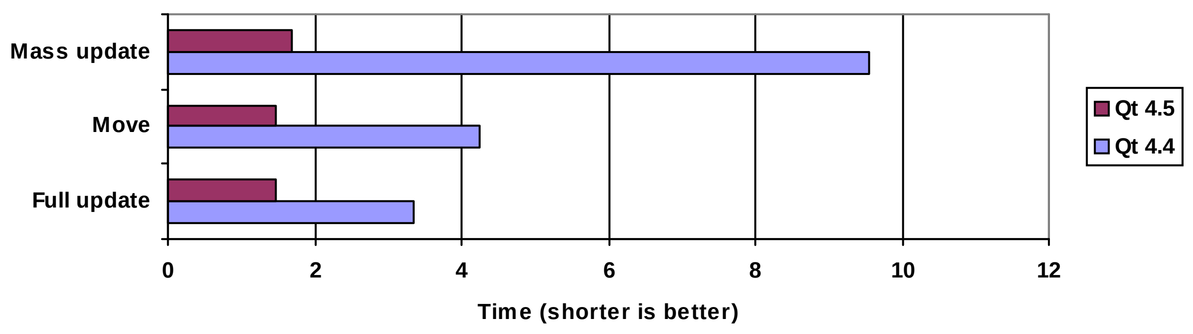


Figure 1 Improvements to the backing store and their impact on performance of opaque widgets.

To be able to enjoy these performance improvements, it is important for you to understand how Qt applications spend their CPU cycles. This white paper contains tips about what to focus on to ensure performance as well as QtBenchLib, which was introduced in Qt 4.5. QtBenchLib allows you to add performance tests to your test suites, thus getting a better picture of where you need to focus your efforts.

Graphics Scene

The graphicsview framework is widely used for creating areas with movable, connectable objects. It is also used to create flexible, animated user interfaces. The framework makes it easy to create dynamic views of pretty much any item. The items can be rotated, scaled, sheared and even rotated in a three dimensional fashion.

Users of the graphicsview framework can expect large performance improvements, sometimes in the order of several magnitudes. This is due to improvements on all levels – from the internal handing of rectangles to the rendering of graphics.

The main improvements have been made in the way that items are transformed from local coordinates, through an item hierarchy, via the scene to actual view and screen coordinates. This is something that affects all graphicsview use cases.

As the improvements affect the entire stack, from the individual items to the view, the improvements will scale, based on the number of items previously used. However, the more items you use, the more you need to be aware of the details concerning the graphicsview framework and how you can tune it for optimum performance.

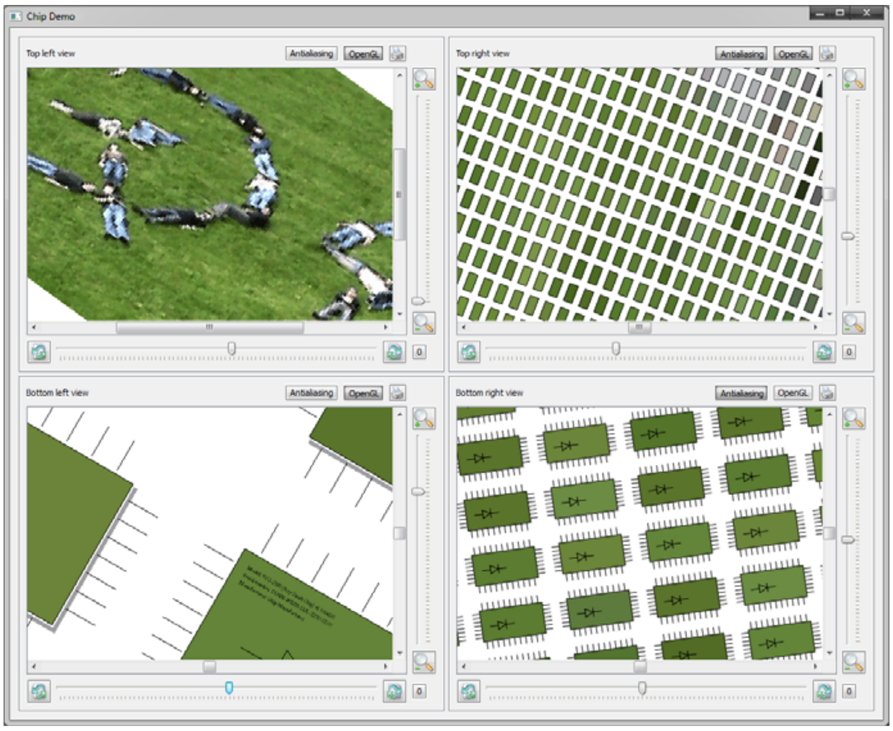


Figure 2 The chip demo – 40000 chips shown through a QGraphicsView.

Looking at the chip demo, shown in Figure 2, the improvements range from around 30% on Windows for most zoomed out operations to 5-100 times better performance on Linux with the raster paint engine. The biggest improvements affects interacting with items. Examples of this is when moving groups of items by moving their parents. When moving a top-level item containing fifty children that, in turn, holds ten children each, in a scene with one hundred top-level items (all in all, moving 551 items in a scene of 55100 items) the speed improvements range from 20-64 times. Details as shown in Figure 3 below.

Regardless of how many items you use in your graphics scene, there are a few things to think about to be able to enjoy the performance. They can all be summarized in one sentence: keep it simple, but let's have a look at the details.

- When painting, try to repaint as little as possible. This means trying to figure out what is inside or outside the `exposedRect` member of the `QStyleOptionGraphicsItem` passed to your `QGraphicsItem::paint` reimplementation.

Figure 3 Moving 551 items in a scene of 55100 items (in a hierarchy of three levels).

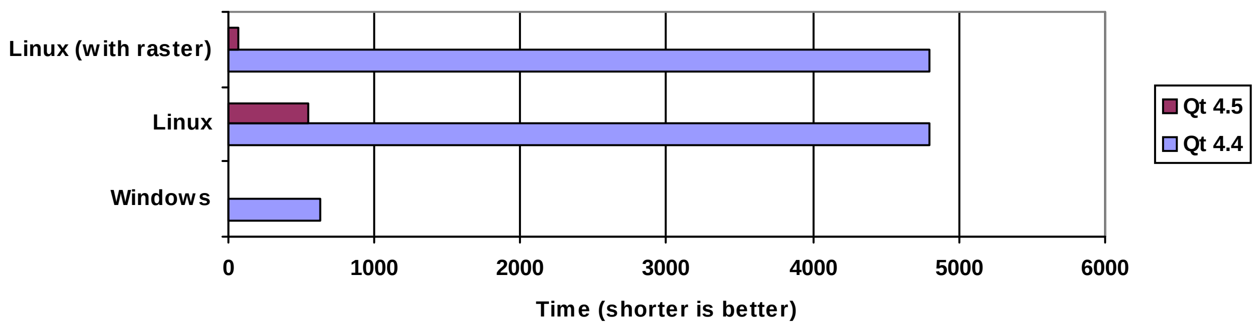
- To determine what needs painting, the graphicsview framework uses the bounding region provided by your graphics items. Ensuring a tight building region, or at least a tight bounding rectangle, means less painting and hence, a quicker user experience of your application.
- Make sure to pick the right viewport update mode for your `QGraphicsView` widget. The right setting of this depends on how you populate your scene and the item's building regions – something that only you as an application developer can tell.
- If your target platform has OpenGL rendering, make sure to utilize it. This is as simple as calling `view.setViewport(new QGLWidget())`, where view is your `QGraphicsView`. If you want antialized graphics, your hardware must support sample buffers. Then, simple pass `QGLFormat(QGL::SampleBuffers)` to your `QGLWidget` constructor.

Painting and Related

The improvements to the painting system of Qt have not been limited to the graphicsview framework. There are a whole range of improvements that have been made to the basic classes involved in all Qt's painting.

The `QRect` class is one of the most basic classes that has been optimized. The `QRect` class is used by almost all code affecting graphics throughout Qt. Since the class affects everything from the layout engine that places and sizes widgets to what to paint where on the screen, the improvements of this class affects all Qt applications.

Even when not painting yourself, it is important to choose the right tool for the task. A common mistake is to use the `QTextEdit` to show and edit plain text, as well as to show read-only rich text.



Using the `QPlainTextEdit` to show and edit plain text greatly improves performance, as does using the `QTextBrowser` widget for showing read-only rich text.

When you do your own painting, you will be happy to learn that the `QTransform` class has been optimized to reduce the number of required multiplications. This affects all platforms, but has an extra big impact on embedded systems where multiplication and division can be considered expensive operations. Transformations are also something that affects all Qt applications that do any sort of painting.

Also affecting all painting, the raster engine that takes painting commands and converts them into actual pixel colors, has been improved. In addition, the backing store, used to cache graphics and merge transparent or non-rectangular widgets with their backgrounds, has also benefited from significant improvements.

Examples of performance improvements are the drawing of smaller images such as icons and decorations. The improvements are even greater when the image is scaled as well. Here we can see improvements of more than five times.

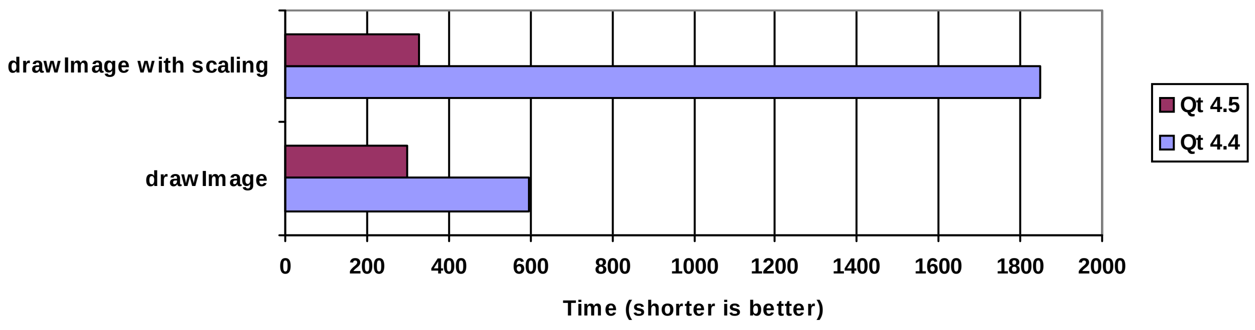


Figure 4 Drawing small images on Windows with 32 bitdepth.

All in all, these optimizations mean you can benefit from quicker graphics for your applications just by upgrading to Qt 4.5. Even more speed can be achieved if your application has been implemented in the prescribed way.

- It is good to know what is fast and what is slow. Fast items are pixmaps, opaque solid color filled rectangles and polygons and cosmetic pens (i.e. a pen with a zero width, indicating one pixel width regardless of resolution). Things that are slow by nature are gradients, thick pens, dashed lines and complex transformations such as perspective transform.
- Something else that is slow, which might pass by unnoticed, is the setting of pens on your `QPainter` object. You will be surprised to learn how much you can save from using one pen at a time instead of swapping back and forth.
- When drawing, the less you draw the quicker the update will be. This means that you can gain from taking the rect member of the `QPaintEvent` passed to your `QWidget::paintEvent` reimplementation.
- There are a number of widget attributes affecting how widgets are repainted. Common examples are `Qt::WA_OpaquePaintEvent`, telling Qt that the widget paints all of its pixels, i.e. that it is not transparent, and `Qt::WA_StaticContents`, telling Qt that the contents is aligned to the top left corner of the widget and is static. This means that resize events only needs to repaint newly expose regions and not the entire widget.

File IO

The performance improvements have not been limited to graphics drawing. Even when you are loading images from disk, you can enjoy a faster experience. The `QImageReader` has been improved, but also the underlying foundation of classes have been improved.

For instance, the `QByteArray` enjoys a speed-up of several magnitudes for certain use cases. This means quicker interaction with files. Also the `QIODevice` class and its ancestor, the `QFile`, has been worked on. As has the `QTemporaryFile` class.

Another class that shows the improvements made in a dramatic way is the `QFileDialog`. Just opening a directory containing ten thousand sub-directories has improved from 21 seconds to 360 milliseconds. This is an improvement of almost a thousand times. The improvement for the same operation for a folder shared over a local network is also very noticeable to the end user: from 77 seconds to 17 seconds.

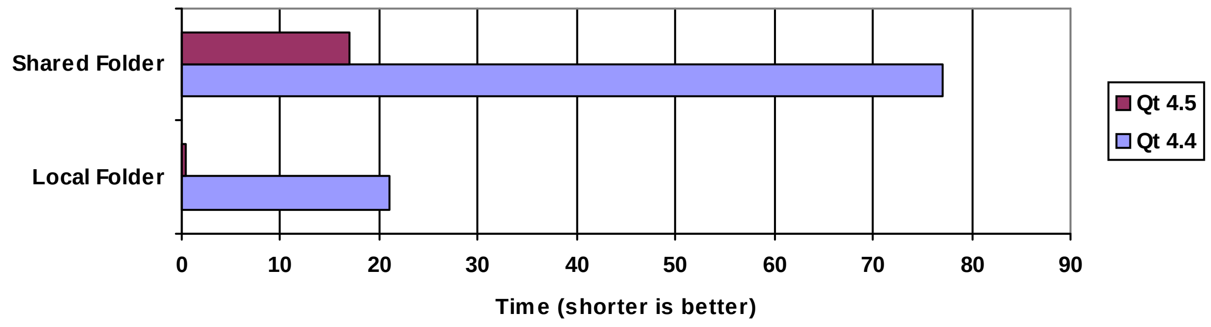


Figure 5 Time for opening a directory with ten thousand sub-directories in a QFileDialog.

When working with IO operations and files, there are a few tricks that you can use to improve performance.

- Read as little as possible. This can be done in two ways – read less data and read your data less frequently. The latter can be implemented by keeping data that you expect to use again in RAM. This option must be evaluated, especially for embedded systems, as RAM can be a limited resource.
- One way to keep down the file sizes is to use binary file formats, as they are more compact by nature. Qt's `QDataStream` provides means to do this in an easy and still platform independent way.
- Design you files' internal structure with size and performance in mind. Make sure to store things in a way so that you avoid large unused areas in your files.

Containers

Containers are one of the most fundamental parts of software applications. Used to keep the data that the application works on, they are the real workhorses of software. Through optimizations, Qt 4.5 has been able to speed up most containers, which means better performance for all your Qt applications.

There are a number of issues to consider when using containers and making the right decisions can make a huge difference. How you structure your data is greatly affected by how you intend to use it. This also affects your choice of algorithms to use and the general structure of your entire application.

The first step is to pick the right container for the task. Qt offers three basic containers with their strengths and weaknesses.

- `QVector` places the data items in adjacent memory locations. This means that accessing data using the [] operator is quick, while inserting and removing data is slow.
- `QLinkedList` implements a classic linked list data pattern. This means that inserting or removing data is quick, but you need to iterate over the list to access the items.

- `QList` tries to please all and is optimized to offer quick access through the [] operator and relatively cheap insertions and removal of items. This is the most commonly container as it isn't slow for any tasks, but it is not the quickest either.

When you have picked the right container for you task you need to consider how you iterate over it. Qt offers both Java-style iterators and STL-style iterators. There is no difference in performance between the two. When using STL-style iterators with a list containing complex items, it is important to use the ++i operator to move to the next item. This is quicker than the i++ operator, as the latter means that the body of the loop works on a copy of the item i.

```
QListIterator<int> i( list );
while( i.hasNext() )
   process( i.next() );
```

Java-style iterator

```
QList<int>::iterator i;
for( i=list.begin(); i!=list.end(); ++i
)
   process( *i );
```

STL-style iterator

Something that does make a difference is if you are using a `const` iterator or not. Using a constant iterator usually always brings a speed-up. The difference between `const` or not is even more vital when using the `foreach` keyword. Here the difference can be measured in magnitudes.

```
foreach( const int &i, list )

   process( i );
```

A const iteration using foreach.

Comparing `foreach` with actually writing a for-loop with an iterator tells us that there is a performance penalty to using `foreach`. But, in the case of a const iterator, the penalty is often neglectable, but not zero.

Qt Internals

When you are using Qt, you leave the main event loop of your application in the hands of the Qt Software engineers. This means that all the behind-the-scenes details such as how is an event passed from the operating system to the receiving object and how are my signals emitted to the slots, are immensely important to the performance of your application.

The performance improvements made for Qt 4.5 run through most of the classes used internally by Qt. One of the most important changes is that the Qt event mechanism has been quite dramatically improved. This means a snapper experience to your users.

The `QObject` class, base class of most Qt classes, has seen lots of improvements as well. This, combined with speed improvements in the `QMetaObject` class, means that the property system will now run faster. This in turn means that the performance of JavaScripts interacting with C++ objects via the `QtWebKit` integration runs faster. These improvements are also enjoyed from `QtScript` as well as from `QtDesigner` and other tools interacting with the Qt object model.

When writing Qt applications there are a few pointers that you can keep in mind to avoid adding unneeded overhead to your code.

- Evaluate if you really need to inherit `QObject`. Sometimes you do it just out of habit.
- Avoid the Q_`OBJECT` macro if you can. You need the macro if you add new signals, slots or

properties, or if you use the `qobject_cast` function with your class.

- If performance counts, use custom events to pass information between threads instead of signals and slots.
- Try to avoid copying data by adding implicit sharing to your complex data carrying classes. See more at http://doc.trolltech.com/4.5/shared.html .

Your Code

Even if the performance of Qt has been improved for version 4.5, much of the end user experience is up to you. By using the right classes in the right places, the right algorithms and writing good code you can provide your users with the best possible experience.

The `QtBenchLib`, introduced with Qt 4.5, lets you evaluate the performance of your code using simple benchmark test cases. If you have used `QTestLib` for unit tests, the `QtBenchLib` will feel familiar to you.

You can add benchmarks to your existing unit test by simply adding a section `QBENCHMARK { /* Code to benchmark */ }` to your existing test cases. This will allow you to measure the time spent between the brackets after the `QBENCHMARK` macro. However, this does not make it possible to compare different benchmark results in an easy way. Instead, the recommended approach is to write a data driven test for benchmarking.

In the source code below, you can see the class declaration of such a test class. The test, merge, is given data from the `merge_data` slot. The purpose of the class is to compare three ways of merging `QStrings` together. The different methods are enumerated in `MergeType`.

```
class StringMergeTest : public QObject
{
      Q_OBJECT

public:
      enum MergeType { Arg, Add, Join };

private slots:
      void merge_data();
      void merge();
};

Q_DECLARE_METATYPE( StringMergeTest::MergeType );
```

`The merge_data slot implementation simply creates three test-case rows, one for each type.`

```
void StringMergeTest::merge_data()
```

```
{
      QTest::addColumn<MergeType>("type");

      QTest::newRow( "Arg" ) << Arg;
      QTest::newRow( "Add" ) << Add;
      QTest::newRow( "Join" ) << Join;
}
```

The merge test case starts by fetching the type for the test and switching to the right test case.

```
void StringMergeTest::merge()
{
      QFETCH(MergeType,type);

      QString res;
      QString argument = QString( "dummy" );

      switch( type )
      {
```

For each case, the scenario is prepared outside the `QBENCHMARK` section to avoid measuring overhead code. Inside the benchmarking section, the actual operation being measured on is performed. The code below shows the `Arg` case, using the `QString::arg` method to join two strings.

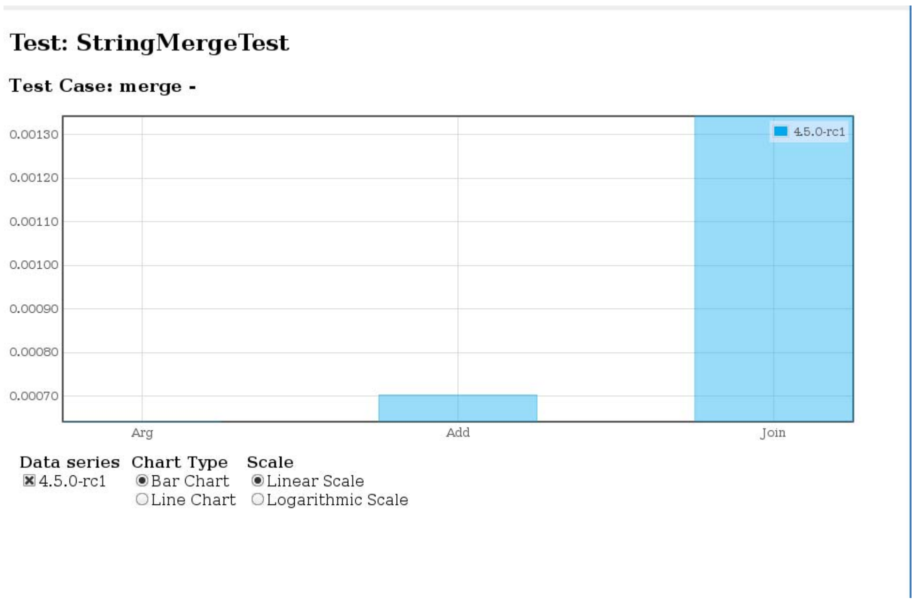
```
      case Arg:
            {
                  const QString start =
                        QString("This is the start, %1,and here it ends.");

                  QBENCHMARK {
                        res = start.arg( argument );
                  }
            }
            break;
```

The implementation is completed by similar cases for Add (adding the strings using the + operator) and Join (placing the strings in a `QStringList` and using `join` to merge them). Building the test and executing it results in a log as the one shown below (emphasis added here).

```
********* Start testing of StringMergeTest *********
Config: Using QTest library 4.5.0-rc1, Qt 4.5.0-rc1
PASS   : StringMergeTest::initTestCase()
RESULT : StringMergeTest::merge():"Arg":
     0.0015 msec per iteration (total: 26, iterations: 16384)
RESULT : StringMergeTest::merge():"Add":
     0.0017 msec per iteration (total: 29, iterations: 16384)
RESULT : StringMergeTest::merge():"Join":
     0.0035 msec per iteration (total: 29, iterations: 8192)
PASS   : StringMergeTest::merge()
PASS   : StringMergeTest::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
********* Finished testing of StringMergeTest *********
```

The report tells us how long each test case took and how many iterations that where used. In this case we used the wall-time method. There are other methods to measure the time spent in each benchmark test case detailed in the reference documentation available at http://doc.trolltech.com/4.5/qtestlib-manual.html .

For those of you who crave a more graphical report, you can download the qtestlib-tools from Qt Labs, http://labs.trolltech.com . This allows you to convert your test reports into a HTML page with bar graphs as shown in figure 1.



Data series: [✓ 4.5.0-rc1]
Chart Type: [✓ Bar Chart] Line Chart
Scale: [✓ Linear Scale] Logarithmic Scale

Figure 6: HTML report from a benchmark test.

As improving performance not always is a simple task, the QtBenchLib will come in handy. It allows you perform measurements in an easy way. Try to keep these things in mind when using QtBenchLib.

- Writing data driven test cases allows you to compare ways of performing the same task. Great for comparing different algorithms.
- Including a QBENCHMARK section wherever applicable in your unit tests allows you to locate weak spots. A class method that takes seconds to complete can seriously hurt your end user experience and must be addressed.
- Use the reports when communicating with Qt Software about your performance problems. Now you can visualize and put figures on your issues.
- Make sure to use it! You can use QtBenchLib as a part of your unit tests. Make sure to run the unit tests and benchmarks as often as possible. This increases both the availability and awareness of performance data and will help you create better applications for your users.

Summary

Qt 4.5 offers greater performance than earlier versions. Improvements have been made throughout Qt and the result is a quicker, snappier experience for the users of your applications.

The goal for Qt Software has been to enable the next generation of user interfaces. Making the user of animations and smooth transitions easy, without requiring abundant computing resources. This benefits both desktop software and embedded applications.

In addition to the performance improvements, Qt Software also gives you the measurement tools to improve your performance. The QtBenchLib makes it easy to add benchmarking to your unit tests. Allowing you to measure and analyze your performance means that you can find your application's weak spots and take action.

About Qt Software:

Qt Software (formerly Trolltech) is a global leader in cross-platform application frameworks. Nokia acquired Trolltech in June 2008, renamed it to Qt Software as a group within Nokia. Qt allows open source and commercial customers to code less, create more and deploy everywhere. Qt enables developers to build innovative services and applications once and then extend the innovation across all major desktop, mobile and other embedded platforms without rewriting the code. Qt is also used by multiple leading consumer electronics vendors to create advanced user interfaces for Linux devices. Qt underpins Nokia strategy to develop a wide range of products and experiences that people can fall in love with.

About Nokia

Nokia is the world leader in mobility, driving the transformation and growth of the converging Internet and communications industries. We make a wide range of mobile devices with services and software that enable people to experience music, navigation, video, television, imaging, games, business mobility and more. Developing and growing our offering of consumer Internet services, as well as our enterprise solutions and software, is a key area of focus. We also provide equipment, solutions and services for communications networks through Nokia Siemens Networks.

Code less.
Create more.
Deploy everywhere.

Oslo, Norway +47 21 60 48 00 ▪ Redwood City, CA, USA +1 650-551-1676 ▪ Beijing, China +86 10 6588 8836 ▪ qt-info@nokia.com www.qtsoftware.com