Performance tips and tricks for Qt/QML applications

General

Do not use JavaScript, but rather implement the UI with QML and the engine with C++

JavaScript should be avoided as much as possible. Do not use it at all.

- JavaScript code should be kept to a minimum. C++ should be used to implement application's the business logic.
- JavaScript is especially bad for performance when it is run during animation, e.g. in lists the JavaScript monster gets evaluated on every frame of the scrolling animation as many times as there are instantiated delegates.
- If you must use JavaScript anyway, simplify your JavaScript expressions as much as possible and write them inline if they fit on one line. QML is compiled to an optimized bytecode-like stream and the JavaScript expressions pass through an optimized evaluator for simple expressions. More info: http://doc.qt.nokia.com/latest/qdeclarativejavascript.html#inline-javascript

- It is also recommended to set values for properties using QML property binding than JavaScript value assignment
- Wrong:

```
itemTitle: {
        if(targetFeedItem == null){
            return "";
        }
        return targetFeedItem.title;
    }
```

- Right:

```
itemTitle: targetFeedItem == undefined ? "" : targetFeedItem.title;
```

Use threads or multiprocessing to allow responsive UI

If you don't use threads for long-lasting operations, the UI is blocked. Use threads for the long lasting operations such as data retrieval to keep UI responsive.

- In QML the WorkerScript element can be used to run JavaScript operations in a new thread (you shouldn't use JavaScript in a serious application)
  - **DO NOT use multiple WorkerScripts or you are in trouble.**
  - If you initialize a large list synchronously in a JavaScript block (this is something that a serious application would never do...) it is very likely that UI jamming will occure with high memory consumption. The memory consumption is likely due to the fact that garbage collector does not get a chance to collect any items while a synchronous loop is processing.
  - More info on WorkerScript: http://doc.qt.nokia.com/latest/qml-workerscript.html
- **C++ threads should be used for all serious applications**
  - This can be done by creating a `QDeclarativeExtensionPlugin` and exposing the needed QObjects to QML
  - In the QObject one exposes the properties, invokable methods, signals and slots needed in the UI
  - For any long (>2ms) operations one should use QRunnable with `QThreadPool` or QFuture see. More info: http://doc.trolltech.com/latest/threads.html
  - The long lasting operations (> N ms) must be implemented as asyncronous server calls or by using threads so that the UI can update itself while the application is doing long lasting backend operations

Use anchors rather than bindings to position items relative to each other

Wrong way to bind rect2 to rect1:

```
Rectangle {
     id: rect1
     x: 20
     width: 200; height: 200
 }
 Rectangle {
     id: rect2
```

```
    x: rect1.x
    y: rect1.y + rect1.height
    width: rect1.width - 20
    height: 200
}
```

Right way to do this, anchor rect2 to rect1:

```
Rectangle {
    id: rect1
    x: 20
    width: 200; height: 200
}
Rectangle {
    id: rect2
    height: 200
    anchors.left: rect1.left
    anchors.top: rect1.bottom
    anchors.right: rect1.right
    anchors.rightMargin: 20
}
```

More info: http://doc.qt.nokia.com/latest/qdeclarativeperformance.html#anchors-vs-binding

App start

Use Booster

On Symbian and MeeGo it is possible to speed up application start by picking up pre-instantiated QApplication and QDeclarativeView from cache. Please contact Nokia for details how to do this.

Use loaders to dynamically load/unload QML

QML application starts quite slow if the whole application is implemented in one huge QML file. Partition our application wisely into logical entities, load minimum QML at start and load more as you need it using Loaders .

- The Loader item can be used to dynamically load and unload visual QML components defined in a QML file or items/components defined within a QML file
- This dynamical behavior allows the developer to control the memory usage and startup speed of an application. More info: http://doc.qt.nokia.com/latest/qml-loader.html#details
- Partition your application into several QML files so that each file contains a logical UI entity. This way loading and unloading are easier to control. **DO NOT have one huge QML file per application.**
- Load absolutely minimum amount of QML at application start to make your application start as quickly as possible. You can connect to network and show spinner etc. after the application UI is visible. If your first view is very complex and requires lots of QML to be loaded, show a splash screen or something.
- You should load pieces of UI only by demand e.g. when the user navigates to another view, but on the other hand it may require more time to navigate between views
- Creating a dialog can take more than 100ms of time. **Make sure you use Loader to load the dialog.**

```
import QtQuick 1.0


Item {
    width: 200; height: 200


    Loader { id: pageLoader }
```

```
    MouseArea {
    anchors.fill: parent
    onClicked: pageLoader.source = "Page1.qml"
    }
}
```

Images

Bitmap formats vs. vector graphics formats

Images can be supplied in any of the standard image formats supported by Qt, including bitmap formats such as PNG and JPEG, and vector graphics formats such as SVG.

However, rendering an SVG is slow compared to a bitmap image.

Load large images asynchronously

If you load images synchronously the UI is blocked. In many cases images are not needed visible immediately, but they can be lazy-loaded.

- You should load images asynchronously in a separate thread if an image doesn't need to be displayed immediately. This can be done by settings QML Image asynchronous to true.
- This way the user interface stays responsive user interface.
- Note that this property is only valid for images read from the local filesystem. Images that are loaded via a network resource (e.g. HTTP) are always loaded asynchonously.
- More info: http://doc.qt.nokia.com/latest/qml-image.html#asynchronous-prop

Avoid resizing/scaling

Resizing/scaling is a very heavy operation in QML. Use images that are in their native size rather than resize/scale large images to correct size.

Use sourceSize with large images

Images are often the greatest user of memory in QML user interfaces.

- sourceSize should be used with large images because property sets the actual number of pixels stored for the loaded image
- If you have a HUGE image 3264 x 2448, but you set sourceSize 204x153, then it scaled down and will be stored as 204x153 in memory.
- If the image's actual size is larger than the sourceSize, the image is scaled down. This way large images do not use more memory than necessary.
- This is especially important for content that is loaded from external sources or provided by the user
- Beware that changing this property dynamically causes the image source to be reloaded, potentially even from the network, if it is not in the disk cache.
- More info on Image::sourceSize property: http://doc.qt.nokia.com/latest/qml-image.html#sourceSize-prop

- Images are cached and shared internally, so if several Image elements have the same source, only one copy of the image will be loaded.
- More info: http://doc.qt.nokia.com/latest/qml-image.html#source-prop

Enable Image.smooth only when necessary

Enabling Image.smooth is bad for performance. Use images in their natural size or disable smooth filtering in animation.

- Image.smooth property causes the image to be smoothly filtered when scaled or transformed
- Smooth filtering gives better visual quality, but is slower
- If the image is displayed at its natural size, Image.smooth has no visual or performance effect
- If you really need to enable Image.smooth, disable smooth filtering at the beginning of the animation and re-enable it at the end of the animation (scaling artifacts are only visible if the image is stationary on the screen)
- More info on Image.smooth property: http://doc.qt.nokia.com/latest/qml-image.html#smooth-prop

Avoid composing an image from a number of elements

It is more efficient to compose an image from a single image than from a number of elements.

- For example, a frame with a shadow could be created using a Rectangle placed over an Image providing the shadow
- It is more efficient to provide an image that includes the frame and the shadow
- More info: http://doc.qt.nokia.com/latest/painting-imagecomposition.html
- More info: http://doc.qt.nokia.com/latest/qdeclarativeperformance.html#image-resources-over-composition

Lists

Ensure that your data model is as fast as possible

In many cases the slow model is actually the bottleneck in list scrolling performance. Ensure that the data model is as fast as possible.

- Delegates must be created quickly as the view is flicked. In order to keep the UI responsive, your data() function must handle over 1000 calls in a second.
- Any additional functionality that is only needed when the delegate is clicked, for example, should be created by a Loader as it is needed.

- Keep the amount of QML to a minimum in your delegate. The fewer elements in a delegate, the faster a view can be scrolled.
- In real-world applicationit is advised to write as much code as possible in C++ and use QML only for the user interface. This is especially important for list delegates. More info: http://cdumez.blogspot.com/2010/11/how-to-use-c-list-model-in-qml.html

CacheBuffer usage in ListView/GridView

In somes cases, cacheBuffer can be useful in improving your ListView/GridView performance. The default cacheBuffer is zero.

- cacheBuffer property determines whether delegates are instantiated outside the visible area of the view
- Note that cacheBuffer is defined in pixels, e.g. if delegate is 20 pixels high, cacheBuffer is set to 40 -> up to 2 delegates above and 2 delegates below the visible area may be retained in memory
- Setting this value can improve the smoothness of scrolling behavior at the expense of additional memory usage. Data is not cached, but the instantiated delegates that requested the data are.
- Define cacheBuffer if you have a short list where every item can be cached
- For longer lists cacheBuffer doesn't bring benefit because items are created at the same rate as if there were no cache while scrolling fast. The cacheBuffer just postpones the problem, i.e. only pushes the position where delegates are created further above/below the visible part of list/grid.
- More info: http://doc.qt.nokia.com/latest/qml-listview.html#cacheBuffer-prop

Avoid useless painting

You should always prevent painting the same area several times. For example, prevent QDeclarativeView from painting its window background if you provide the background of your application:

```
QDeclarativeView window;

window.setAttribute(Qt::WA_OpaquePaintEvent);

window.setAttribute(Qt::WA_NoSystemBackground);

window.viewport()->setAttribute(Qt::WA_OpaquePaintEvent);

window.viewport()->setAttribute(Qt::WA_NoSystemBackground);
```

Also, consider using Item as root element rather than Rectangle to avoid painting the background several times:

- If your root element is a Rectangle, you're painting every pixel, maybe even several times
- The system (QDeclarativeView) first paints the background and then paints all the QML elements
- You might have a Rectangle as the root element and inside that you have a lot of elements without opacity covering most of the Rectangle. In this kind of case the system is doing useless painting.
- You could instead use Item as root element because it has no visual appearance
- If you need to paint background, but have static UI elements covering part of the screen, you can still use Item as the root element and anchor a Rectangle between those static items. This way you don't do useless painting.

More info: http://doc.qt.nokia.com/latest/qdeclarativeperformance.html#rendering
http://doc.qt.nokia.com/latest/qml-item.html
http://juhaturunen.com/blog/2011/04/a-qml-rendering-performance-trick-so-simple-that-it-hurts/

Other

Fixed length lists with Flickable+Column+Repeater

If you have a simple list with fixed length, you could try to optimize performance using Flickable+Column+Repeater instead of QML ListView. This way list creation would be slower, but list scrolling smoother.

Beware of string operations

- Multiple uses of the '+' operator usually means multiple memory allocations
- Use StringBuilder for more efficient strings. QStringBuilder uses expression templates and reimplements the '%' operator so that when you use '%' for string concatenation instead of '+', multiple substring concatenations will be postponed until the final result is about to be assigned to a QString. At this point, the amount of memory required for the final result is known. The memory allocator is then called once to get the required space, and the substrings are copied into it one by one.
- Define: QT_USE_FAST_CONCATENATION, QT_USE_FAST_OPERATOR_PLUS
- See more: http://doc.trolltech.com/latest/qstring.html#more-efficient-string-construction

Animate as small area of the screen as possible in transition animations

If you need to move 3 elements in a second try moving each at a time in 300ms. The system works so that it calculates the bounds of items that need repainting and paints everything inside those bounds. This can be really bad two little things are animating in opposite corners.

Avoid complex clipping

You should enable clipping only when you really need it. The default clip value is false.

- If clipping is enabled, an item will clip its own painting, as well as the painting of its children, to its bounding rectangle
- More info on Item.clip: http://doc.qt.nokia.com/latest/qml-item.html#clip-prop

Does it help performance if you strip off comments or white space from QML files?

Not really. The files are reprocessed into a binary in-memory representation at startup, so by runtime there should be no performance difference. You might be lucky and get 0.5% improvement, and then only on start-up, nowhere else.

Avoid unnecessary conversion

QML performs a type conversion if the given value for a property does not match with the type specified for the property. This conversion consumes additional memory.

- For example, `Image` and `BorderImage` have a source, which is of type `url`. Conversion is needed if image source's property is defined as `string`, it should rather be `property url`.
- Wrong way:

```
property string messageAvatar: ""
```

- Right way:

```
property url messageAvatar: ""
```

Anchors and memory consumption

Anchoring can be done in two ways:

- The normal way:

```
Item {
    width: 800
    height: 400
    Button  {
        id: button;
        objectName: "Button";
        anchors.top: parent.top
        anchors.left: parent.left
        anchors.right: parent.right
        text: "Press me";
    }
}
```

- This way consumes slightly less memory:

```
    Item {
        width: 800
        height: 400
        Button  {
            id: button;
            objectName: "Button";
            anchors { top: parent.top; left: parent.left; right: parent.right
}
            text: "Press me";
        }
    }
```

Qt C++

Controlling QObject Ownership

Qt Script uses garbage collection to reclaim memory used by script objects when they are no longer needed (only if QObject has QScriptEngine::ScriptOwnership); an object's memory can be automatically reclaimed when it is no longer referenced anywhere in the scripting environment. Qt Script lets you control what happens to the underlying C++ QObject when the wrapper object is reclaimed, i.e. whether the QObject is deleted or not. You can only do this when you create an object by passing an ownership mode as the second argument to QScriptEngine::newQObject(). By default QObjects are "Qt Ownership" == objects are never deleted, even if you try do so in scriptengine side.http://doc.qt.nokia.com/latest/scripting.html#controlling-qobject-ownership

Turn off sample buffers

Turn off sample buffers, check if the graphical results are acceptable to you.

```
    QGLFormat format = QGLFormat::defaultFormat();

    format.setSampleBuffers(false);

    QGLWidget *glWidget = new QGLWidget(format);

    //### potentially faster, but causes junk to appear if top-level is Item,
not Rectangle

    //glWidget->setAutoFillBackground(false);

    view->setViewport(glWidget);
```

Integrating with a QWidget-based UI

Creating large numbers of QDeclarativeView objects may lead to performance degradation.

- QDeclarativeView is slower to initialize and uses more memory than a QWidget, and creating large numbers of QDeclarativeView objects may lead to performance degradation
- Iit may be better to rewrite your widgets in QML, and load the widgets from a main QML widget instead of using QDeclarativeView
- QWidgets were designed for a different type of user interface than QML, so it is not always a good idea to port a QWidget-based application to QML
- QWidgets are a better choice if your UI is comprised of a small number of complex and static elements, and QML is a better choice if your UI is comprised of a large number of simple and dynamic elements

- More info: http://doc.qt.nokia.com/latest/qml-integration.html#integrating-with-a-qwidget-based-ui

Integrating QML with a QGraphicsView-based UI

If you have an existing UI based on the Graphics View Framework, you can integrate QML widgets directly into your QGraphicsScene . The following QGraphicsView options are recommended for optimal performance of QML UIs:

- QGraphicsView ::setOptimizationFlags(QGraphicsView ::DontSavePainterState)
- QGraphicsView ::setViewportUpdateMode(QGraphicsView ::BoundingRectViewportUpdate)
- QGraphicsScene ::setItemIndexMethod(QGraphicsScene ::NoIndex)
- More info: http://doc.qt.nokia.com/latest/qml-integration.html#integrating-with-a-qgraphicsview-based-ui

Use showFullScreen()

It is advised to use QDeclarativeView::showFullScreen() instead of QDeclarativeView::show() to avoid compositing overhead. This should improve the performance a little.

Avoid unnecessary QRegExp construction

QRegExp is faster at matching rather than construction. So if you plan to do a lot of regular expression matching, construct your QRegExp once, eg as a const static, and use it many times.

Creating QServiceManager instance is costly during application startup

This causes 40-50ms delay for application launch. Create QServiceManager during idle time instead.

Reduce expense of Qt::SmoothTranformation without reducing image quality

A combination of Qt::FastTransformation and halfscaling helps to reduce effort for scaling images without loosing the quality of the resulting image. The trick is to do the scaling in two steps:

1. Scale down to 2 times of the destination size using Qt::FastTransformation
2. Scale to the destination size by averaging two pixels in both dimensions using:

```
quint32 avg = (((c1 ^ c2) & 0xfefefefeUL) >> 1) + (c1 & c2);
```

NOTE: Previously we had suggested to scale down to a factor of 1.5 and apply smoothscaling then however this version should result in much better quality and be much faster. It should be faster as the new algorithm just requires bit arithmetics and bit shifting. So no floating point operations are

needed anymore and there are no branches anymore that would require branch predictions (as would be the case for all kinds of bilinear filtering).

Drawback: This optimization does not make any sense for smaller images. Therefore we need a heuristic to decide whether it should be used.

Simple demo implementation:

```
    // Smooth scaling is very expensive (size^2). Therefore we reduce the
size

    // to 1.5 of the destination size and using fast transformation
afterwards.

    // Therefore we speed up but don't loose quality..

    if ( canvasPixmap.size().width() > ( 4 * size.toSize().width() ) ) //
Primitive heuristc to decide whether optimization should be used.

        {

            // Improve scaling speed by add an intermediate fast
transformation..

            QSize intermediate_size = QSize( size.toSize().width() * 2,
size.toSize().height() * 2 );

            canvasPixmap = canvasPixmap.scaled( intermediate_size,

                                                aspectRatioMode,

                                                Qt::FastTransformation ); //
Cheap operation!

        }

        canvasPixmap = canvasPixmap.scaled( size.toSize(),

                                            aspectRatioMode,

                                            Qt::SmoothTransformation ); //
Expensive operation!
```

Avoid duplicating containers by accident

Qt's container classes like `QList`, `QStringList`, `QVector`, `QMap`, ... are all *implicitly shared*. That means that a simple copy from one to the other (e.g., as a function return value) doesn't cost much:

Only a *shallow copy* is made, i.e. just the container's management structure is really duplicated; the items are not copied, just a reference counter is incremented.

As long as you don't try to modify the copy, that is. And even then a *deep copy* (i.e., all items are copied, too) is only made if the reference counter is greater than 1.

This is why the coding conventions insist that *const_iterators* should be used whenever possible: A non-const iterator is a write access to the container, causing a deep copy. `begin()` and `end()` on a non-const (!) container are what get the deep copy started.

Sometimes you get lucky. This code, for example, is harmless:

```
    QDir dir(myPath);

    QStringList files=dir.entryList();

    for ( QStringList::iterator it=files.begin(); it != files.end(); ++files
) {

        myWidget->addItem(*it);

    }
```

But this is **just by dumb luck**. The call to `files.begin()` would normally initiate copying the `files` QStringList that was returned by `dir.entryList()`. But since its reference count is back to 1 after returning from that function, there is no reason for a deep copy; `files` now holds the only reference to the items.

Better than relying on dumb luck would be to be explicit:

```
    QDir dir(myPath);

    QStringList files=dir.entryList();

    for ( QStringList::const_iterator it=files.constBegin(); it !=
files.constEnd(); ++files ) {

        myWidget->addItem(*it);

    }
```

Avoid implicit copies caused by operator[]

Not so harmless, however, is this piece of code:

```
    QStringList nameList=myClass.nameList();

    if ( nameList[0]==searchName ) { // deep copy of nameList
```

```
    ...

}
```

This seemingly harmless `nameList[0]` will cause a deep copy of `nameList`, duplicating each string it contains in the process. From this copy, element #0 is taken and compared with `searchName`. After the `if()` statement this temporary copy of `nameList` is immediately destroyed again.

A lot of operations. And so useless.

Why is a temporary object created? Because `nameList[]` is an *LValue* - a value that can be used on the *left* side of an assignment - like this:

```
nameList[0]=someValue;
```

So this `operator[]` is a *write access* on its argument, and Qt's container classes initiate a deep copy on a write access.

You can get lucky here again: QList etc. have overloaded versions of `operator[]()` for *const* containers. In that case, no deep copy will occur. So this is safe:

```
void myFunc( const QStringList &nameList )

{

    if ( nameList[0]==searchedName) { // no deep copy: nameList is const

        ...

    }

}
```

But again, this is just dumb luck. Better not rely on it. Better be explicit.

Use at() instead of operator[] if you don't want to assign a value to that element.

Avoid a deep copy with code like that:

```
QStringList nameList=myClass.nameList();

if ( nameList.at(0)==searchName ) { // no deep copy

    ...

}
```

In the special case of `at(0)` it is even a little more efficient and and considerably better readable to use `first()` :

```
QStringList nameList=myClass.nameList();

if ( nameList.first()==searchName ) {

    ...

}
```

Prefer `first()` to `at(0)` .

Most Qt containers have `first()`. Regrettably, `QString` or `QByteArray` don't.

Avoid connecting to QGraphicsScene ::changed() signal

To avoid rather expensive change area calculations, do not connect to QGraphicsScene ::changed() signal.
More info: http://doc.qt.nokia.com/latest/qgraphicsscene.html#changed