Custom QML Component: Website Thumbnails

Introduction

12 Dec 2010

This tutorial shows how to embrace the Qt Quick style of application development by extending the Qt Quick framework to expose your C++ business logic to the QML. You are then able to use these components in different ways, with a proper separation between business logic and user interface. In this tutorial we demonstrate some of the possibilities, by implementing from scratch a C++ based Website thumbnailer which can then be used from QML.

The rest is hopefully up to your imagination!

The media player is loading...

Prerequisites

You should be familiar with Qt Quick, and have some basic QML experience, as well as some experience with Qt C++, to be able to fully understand and possibly extend the examples provides in this tutorial. A good introduction is given in the official Qt Quick documentation. This tutorial is based on Qt version 4.7.1, and has been developed on a desktop. Once Qt 4.7 becomes available for Symbian as part of the Nokia Qt SDK we will update this tutorial to include running it on a Symbian device.

Goal

There are various applications that benefit from the ability to show a webpage as a small image. Examples are for example the "homescreens" of web browsers such as Opera and Google Chrome, but one can also think of bookmark managers, favourite link previews, and so on.

We'd like to be able to use such a type, the *WebThumbnail*, in our QML user interface, like this:

```
...
WebThumbnail {
  id: thumbnail
  url: "http://www.developer.nokia.comhttp://www.developer.nokia.com/Community/Wiki/Wiki_Home"
  width: 256
  height: 256
}
...
```

Since a *WebThumbnail* this is not a part of Qt Quick itself, it will need to be imported as a separate module at the top of any QML file which uses it, like so:

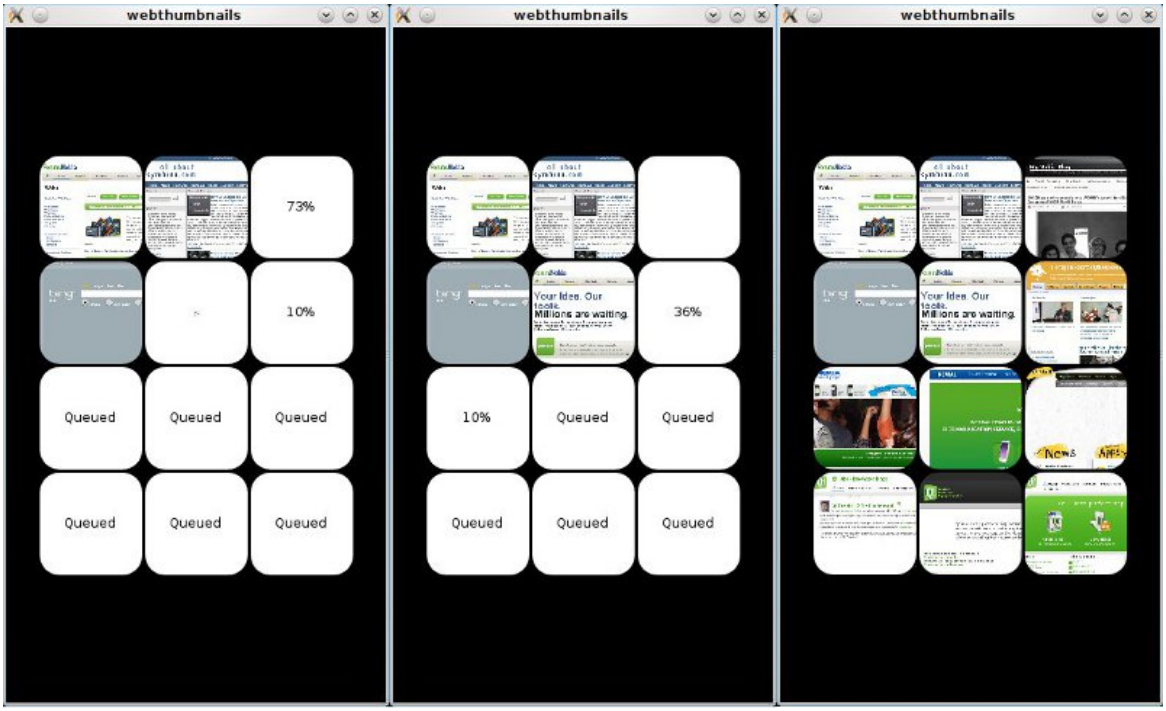
```
import examples.webthumbnail 1.0
```

You are free to name both the module and QML type how you wish; in this tutorial we'll use the "examples.webthumbnail" package name.

The result should look something like this, embedded in the QML interface.

which allows us to do create something like this:

(C) Copyright Nokia 2013. All rights reserved.



of perhaps a coverflow?

The right approach ?

Before we continue developing this new QML component, we should ask ourselves if this is the correct way of approaching this. For example, what about a QDeclarativeImageProvider ? We could create a new image type; then, to use it, our QML would look something like this. Note that the embedded URL may need to be escaped for this example to work at all:

```
// Hypothetical, non working example, where some QDeclarativeImageProvider
// exists which understands the "webthumbnail" type

Image {
  source: "image://webthumbnail/http://www.developer.nokia.comhttp://www.developer.nokia.com/Community/Wiki/Wiki_Home"
}
```

This approach is a bit harder to use, when the URL of the website we want to show becomes slightly more complicated, for example with parameters. Furthermore, we have little control. Perhaps we want to enable or disable javascript when rendering a given website to a thumbnail; perhaps we want to know how much data was transferred to load the thumbnail; perhaps we want a detailed status property with progress indication. We will see that all of these are easily implemented in a custom component, but would be hard or nearly impossible to add if we would be using a QDeclarativeImageProvider.

Another approach would be to simply load a small QML WebView, however this is too resource intensive. We want only a thumbnail of the website, not a fully loaded miniature website with live elements that can be interacted with. We can easily show a dozen thumbnails in a mobile UI, but a dozen live websites is asking a bit much. Of course a website needs to be loaded in order to render its thumbnail, but it can be discarded immediately after. If we carefully control the concurrency, i.e. the number of simultaneous website loads, we can achieve good performance even on mobile devices.

(C) Copyright Nokia 2013. All rights reserved.

Specification and planning

Here we approach the implementation roughly in the order in which you would develop and evolve it, trying to demonstrate the thinking process that may help you implement your own QML extensions.

Thumbnailing

One of the first questions to ask is how we actually create a thumbnail of a website. A nice Qt Labs blog entry, with a follow-up article, outlines how to use a QWebView to load a website and then render it to a (thumbnail) image. The approach we take in this tutorial is based on this technique.

It boils down to using a QWebView to load a website, creating a QPainter with a QImage backend, rendering the website to this painter and thus this image, and then scaling the image to the appropriate format, using both scaling and cropping to achieve the effect you like.

One very expensive operation (in terms of performance) is rendering a loaded website to an image. We therefore avoid rendering to a large image, as done in the Qt blog articles, but immediately clip to our required size. The loading, parsing and rendering of the websites unfortunately still happens in the main UI thread; this is not a great solution since it will create noticeable jerkiness at certain points. Trying to offload these to a background thread is an interesting topic for another article, but out of the scope of this tutorial. Once the thumbnails are loaded, however, the UI will be incredibly smooth (with a QGLWidget backend).

Centralized settings and control

As mentioned in the QDeclarativeImage example, there are a number of possible settings we may want to control from within QML. It would also be nice if we can enable some kind of caching mechanism, so that thumbnails can be given an expiry interval, for example 2 days, and get reloaded if they are older than that. This provides a nice way to improve performance.

In order to control these settings, we could have something like the following approach:

```
import example.webthumbnail 1.0

Item {
  // Our main UI object
  ...

  WebThumbnailContext {
    id: thumbnail_context
    javascript: true
    plugins: false
    java: false
    concurrency: 2
    thumbnail_width: 256
    thumbnail_height: 256
  }
  ...
}
```

which can then be referenced from all of the *WebThumbnail* instances:

```
  WebThumbnail {
    context: thumbnail_context
    url: "http://www.developer.nokia.comhttp://www.developer.nokia.com/Community/Wiki/Wiki_Home"
    width: 256
    height: 256
    radius: 20    // Create rounded corners
    border: false
  }
}
```

As can be seen above, the *WebThumbnail* now has a *context* property, which assigns the thumbnail to a given request manager (a context) which takes care of queueing and managing requests. One important role of the manager is to collapse identical requests into one; that is, if two seperate thumbnails refer to the same URL, the manager re-uses a previous request, finished or not, for the second thumbnail.

This is more likely to happen than you may expect; if the *WebThumbnail* is used in a delegate, for example for some kind of bookmark model, it will be created and destroyed often, based on its visiblity in the view. It is possible the delegate will be destroyed before the thumbnail has finished loading because the user scrolls down through the bookmarks, and is then recreated when the user scrolls back up. In this case the original request will have simply continued loading, and the thumbnail may in fact already be ready to show immediately.

(If you are unsure of how a delegate is used, or about the use of model/view in QML in general, there is a very nice article on QML ListViews you may want to read)

Final WebThumbnail QML component thoughts

So far we have a context, and a url property. It is possible to think of many different uses and useful properties, but to keep this tutorial simple enough we will not try to implement the kitchen sink; instead we will, at the end of this tutorial, give a list of possible addition you may want to try.

A *WebThumbnail* can be in several states:

- Idle: the thumbnail is not configured or not yet completely initialized. This state should be very short-lived and is used mostly during construction, because as soon as the URL and context properties have been set the state should change to one of the following:
- Queued: its context manager has queued this thumbnail for rendering, but is still busy doing something else, or is perhaps waiting for a network connection.

(C) Copyright Nokia 2013. All rights reserved.

- Loading: the context manager is now busy loading and rendering this thumbnail.
- Finished: a ready made thumbnail exists for this particular thumbnail's URL.

These states will be represented in a *status* property (to avoid confusion with QML's built-in state property); this way we can for example show a loading animation while a thumbnail is loading. This is a read-only property.

The QWebView class has a progress indicator signal, which gives the percentage of completion of the loading process. While the thumbnail is in the Loading state, it might be nice to show the progress, either as text or via a progress bar. The *WebThumbnail* will have a "progress" property, an integer with a range from 0 to 100, which is 0 when in the Queued state, 100 when in the Finished state, and anywhere inbetween during the Loading state.

Some additional properties you could implement are the amount of data transferred, the time it took to load the website and render the thumbnail, or the age of the thumbnail. If the manager supports saving and loading cached thumbnails, which will not be implemented in this tutorial, there could be a property to indicate whether this is a fresh or a cached copy.

Now that we know roughly what the *WebThumbnail* will look like, we will immediately create a new component completely in QML, called WT.

This component adds support for the state and progress indicators, and will be used in the QML user interface of the demo applications. This is an important step, since the appearance of the loading and queued states is likely to be very UI specific, and not at all relevant to the C++ side. Therefore we build the extra functionality on top of our C++ component directly in QML.

If you're writing a bookmark browser, this new component would for example be part of the delegate. It is not required to do this, but it will make further examples shorter is more like how QML would be used in a real application.

The following is saved in the file WT.qml, and allows you to instantiate *WT* objects in your QML, as will be shown in the later examples. This example also shows how to set properties on the root of your new component, such as "url", "border" and "radius", which can then be referenced from the inner components such as the actual *WebThumbnail*. This encapsulates the implementation of the WT component, which means it is easier to replace it with some other implementation in the future, perhaps for a different UI, without affecting the rest of your application, as long as the API stays the same.

```
// WT.qml
// This implements a widget based on the WebThumbnail, adding support for loading and progress indication
import QtQuick 1.0
import examples.webthumbnail 1.0

Item {
    property url url;
    property variant context;
    property bool border: false
    property int radius: 20

    WebThumbnail {
        id: wt
        context: parent.context
        url: parent.url
        width: parent.width
        height: parent.height
        radius: parent.radius
        border: parent.border
        smooth: true
        opacity: wt.status == WebThumbnail.Ready ? 1.0 : 0.0
        scale: wt.status == WebThumbnail.Ready ? 1.0 : 0.0

        Behavior on scale {
            NumberAnimation { duration: 450; easing.type: Easing.OutQuad }
        }

        Behavior on opacity {
            NumberAnimation { duration: 350; easing.type: Easing.OutQuad }
        }
    }

    /* Visible during queueing and loading */
    Rectangle {
        anchors.fill: parent
        radius: parent.radius
        color: "#999999"
        smooth: true
        border.width: 2
        border.color: "black"
        opacity: wt.status == WebThumbnail.Ready ? 0.0 : 1.0

        Behavior on opacity {
            NumberAnimation { duration: 350; easing.type: Easing.OutQuad }
        }

        Text {
            anchors.centerIn: parent
            color: "black"
            text: "Queued";
            smooth: true
            opacity: wt.status == WebThumbnail.Queued || wt.status == WebThumbnail.Idle ? 1.0 : 0.0
        }

        Text {
            anchors.centerIn: parent
            color: "black"
            text: wt.progress + "%";
            smooth: true
            opacity: wt.status == WebThumbnail.Loading ? 1.0 : 0.0
        }
    }
}
```

Note: There are different ways to implement the above. You may also use QML States and Transitions to achieve a similar effect. The above is for illustration, you are encouraged to experiment with your own ideas!

Final WebThumbnailContext QML component thoughts

In the above examples we've seen the "plugins", "javascript", "java", "thumbnail_width", "thumbnail_height", and "concurrency" properties. In addition, a *WebThumbnailContext* could provide some more information in the form of read-only properties, which could be used elsewhere in the QML of the

(C) Copyright Nokia 2013. All rights reserved.

application. For example, it could have a "active_requests" property, which can enable a global network activity indicator somewhere else in the application. It could have a "bytes_transferred" property, to show global network traffic. It might have a "thumbnail_count" property to show how many *WebThumbnail*s are currently bound to this context. Thinking even further, you could create an offline/online property, a refresh trigger that reloads all thumbnails, settings for caching purposes, etc. However, in this tutorial we will keep things simple.

Implementation

A QML component, written in C++, is derived from the QDeclarativeItem class, which is in turn derived from theQGraphicsObject class (which a QGraphicsItem that inherits from QObject, and as such is part of the meta-object system and can have signals, slots and properties), and a the QDeclarativeParserStatus.

The latter is interesting to point out. Imagine the following QML:

```
WebThumbnail {
  id: thumbnail
  context: some_context
  url: "http://www.developer.nokia.comhttp://www.developer.nokia.com/Community/Wiki/Wiki_Home"
}
```

in which order will these properties be set? If the URL is set before the context is set, there is no context yet to request the URL to be rendered, but if the context is set before the URL, we don't yet know which URL we want to render.

Of course it is possible to work around this with some careful code, but it is easier to take advantage of the QDeclarativeParserStatus::componentComplete() method by reimplementing it, and not activating the *WebThumbnail* until this has been called.

As for the QGraphicsObject inheritence, it simply means a QDeclarativeItem is essentially a QGraphicsItem, and in fact the whole QML scene is based on the QGraphicsView and QGraphcisScene. There is currently research ongoing into an OpenGL scenegraph backend, which can enable more optimizations than the current QPainter based backend, in which case there may or may not be backwards compatibility to new QML objects, i.e. QML objects may no longer be QGraphicsItems. This does not change anything on the QML side of things, but custom C++ based declarative items may need to be modified to reflect the new back end. In most cases, including this particular example, this will be not be too difficult.

Back to the QGraphicsObject, it means that the *WebThumbnail* component will do its drawing in the *paint()* method which it re-implements. One important detail which can be very frustrating if you miss it, is that by default a QDeclarativeItem sets the "QGraphicsItem::ItemHasNoContents" flag, causing it not to be drawn. For the WebThumbnailContext class this is fine; for the WebThumbnail obviously not. Remember to unset this flag when needed.

WebThumbnail (C++)

The first half of the WebThumbnail class looks like this:

```
class WebThumbnail: public QDeclarativeItem
{
    Q_OBJECT
    Q_ENUMS(Status)

    Q_PROPERTY(QUrl url READ getUrl WRITE setUrl NOTIFY urlChanged);
    Q_PROPERTY(WebThumbnailContext * context READ getContext WRITE setContext NOTIFY contextChanged);
    Q_PROPERTY(Status status READ getStatus NOTIFY statusChanged);
    Q_PROPERTY(int progress READ getProgress NOTIFY progressChanged);
    Q_PROPERTY(int radius READ getRadius WRITE setRadius NOTIFY radiusChanged);
    Q_PROPERTY(bool border READ getBorder WRITE setBorder NOTIFY borderChanged );

    public:
        enum Status { Idle, Queued, Loading, Ready, Error };
```

All of the properties defined via Q_PROPERTY (for more information on Qt properties see the documentation) will be exported automatically to QML. In fact, these properties are bound such that when we change any property, for example the progress indication, the QML will automatically update itself to reflect this new value. Behind the scenes QML uses the NOTIFY signals we have specified.

Drawing is done in the QGraphicsItem::paint(..) method which we override, and we will reimplement the QDeclarativeParserStatus::componentComplete() method.

Variables:

```
protected:
        WebThumbnailContext  * m_context;
        QPixmap m_result;
        int m_progress;
        enum Status m_status;
        QUrl m_url;
        int m_radius;
        bool m_border;
```

The QPixmap is an implictly shared class, so if we have multiple thumbnails with the same URL they will each own a cheap copy which refers to the real pixmap data.

We have getters, setters and signals for the properties:

```
    public:
        WebThumbnail(QDeclarativeItem *parent=0);
        ~WebThumbnail();
```


(C) Copyright Nokia 2013. All rights reserved.

```
        const QUrl & getUrl() const;
        WebThumbnailContext * getContext() const;
        enum Status getStatus() const;
        int getProgress() const;
        int getRadius() const;
        bool getBorder() const;
        void setRadius(int);
        void setBorder(bool);

        void setUrl(const QUrl &);
        void setContext(WebThumbnailContext *);

    public slots:
        void started();
        void progress(int);
        void finished(const QPixmap&);
        void error();

    signals:
        void urlChanged();
        void contextChanged();
        void statusChanged();
        void progressChanged();
        void radiusChanged();
        void borderChanged();
};
```

Note that the finished callback includes the resulting QPixmap object which will be stored in the m_result member, and drawn in the paint() method. The error slot is called when a thumbnail request can not be fulfilled, for example because the URL is invalid or because there is no network connection. In a real application you will want to handle these cases gracefully, perhaps showing a different icon.

We will not show all of the implementation code here, as the full sources are included at the end of this tutorial, but we will highlight some of it.

Since this is a visible item, we need to unset the ItemHasNoContents flag:

```
WebThumbnail::WebThumbnail(QDeclarativeItem *parent)
    : QDeclarativeItem(parent),
      m_context(0),
      m_result(),
      m_progress(0),
      m_status(Idle),
      m_url(),
      m_radius(0),
      m_border(false)
{
    setFlag( QGraphicsItem::ItemHasNoContents, false );
}
```

The paint() method checks that state is Ready, that the result pixmap contains data, and then simply draws the pixmap, scaling to the current with and height of the QML component. The QML itself will define the right width and height, for example during transitions or animations it may happen that the thumbnail becomes stretched or otherwise distorted, but these are of no concern to the C++ code.

```
void WebThumbnail::paint(QPainter * p, const QStyleOptionGraphicsItem *, QWidget *)
{
    if (m_status == Ready && !m_result.isNull()) {
        p->save();
        if (m_radius > 0) {
            QPainterPath path;
            path.addRoundedRect( QRect(0, 0, width(), height()), m_radius, m_radius );
            p->setClipPath( path );
        }
        p->setRenderHint( QPainter::Antialiasing, true );
        p->setRenderHint( QPainter::HighQualityAntialiasing, true );
        p->setRenderHint( QPainter::SmoothPixmapTransform, true );
        p->drawPixmap( 0, 0, width(), height(),  m_result );
        if (m_border) {
            QPen pen;
            pen.setWidth(1);
            pen.setColor(Qt::black);
            p->setPen(pen);
            p->drawRoundedRect( QRectF(2, 2, width()-4, height()-4 ), m_radius, m_radius );
        }
        p->restore();
    }
}
```

Here is an example of a typical "setter" method; first we check if we're actually changing the value, and if so, we emit the corresponding "changed" signal.

```
void WebThumbnail::setUrl(const QUrl &url)
{
    if (url == m_url)
        return;
    m_url = url;
    emit urlChanged();
    if (m_context != 0 && m_status == Idle) {
        // Late initialization of the url property (maybe we are a delegate)
        m_context->startRequest(this);
    }
}
```

Our "componentComplete()" method looks like this, note that it is possible that the URL is set later, such as when the thumbnail is used in a delegate.

```
void WebThumbnail::componentComplete()
{
    if (!m_context) {
        qWarning() << "No context configured, can not continue";
        return;
    }
    if (!m_url.isEmpty())
        m_context->startRequest(this);
}
```


(C) Copyright Nokia 2013. All rights reserved.

The "finished()" slot looks like this:

```
void WebThumbnail::finished(const QPixmap &pm)
{
    m_status = Ready;
    m_result = pm;
    emit statusChanged();
    update(); // We want a repaint since now we have something interesting
}
```

Note also the following setter for the "context" property:

```
void WebThumbnail::setContext(WebThumbContext *context)
{
    if (m_context != 0) {
        qWarning() << "Context manager can only be set once";
        return;
    }
    m_context = context;
    emit contextChanged();
}
```

In theory it is possible to re-assign a thumbnail to a different context manager, however in our example here this does not make much sense, and would required more code in order to detach a request from one context manager and pass it to another, etc. In your own examples you may well need such functionality, but here we simply allow this property to be set exactly once, which is just fine for a *WebThumbnail*.

WebThumbnailManager (C++)

```
typedef QList<WebThumbJob *> JobList;

class WebThumbnailContext: public QDeclarativeItem
{
    Q_OBJECT

    Q_PROPERTY(int queued READ getQueued NOTIFY queuedChanged);
    Q_PROPERTY(int active READ getActive NOTIFY activeChanged);
    Q_PROPERTY(int thumbnailCount READ getThumbnailCount NOTIFY thumbnailCountChanged);
    Q_PROPERTY(int concurrency READ getConcurrency WRITE setConcurrency NOTIFY concurrencyChanged);

    protected:
        JobList m_thumbnails;
        JobList m_queue;
        int m_activecount;
        int m_totalcount;
        int m_concurrency;
```

Note the introduction of the WebThumbnailJob class; this class represents a loading job for a given URL. Its definition is explained in the next section.

We see again the familiar use of the Q_PROPERTY macro, with four read-only properties and one read-write property.

The variables are used as follows:

- m_thumbnails: contains rendering jobs which are either in progress, or have finished.
- m_queue: contains rendering jobs which are scheduled, but not yet started, because of the concurrency limit.
- m_activecount: represents the active number of rendering jobs (real QWebViews being loaded)
- m_totalcount: represents the total number of bound thumbnails

Note that the context manager does not store any *WebThumbnail* pointers itself. WebThumbnail instances can be very shortlived (for example when used in a delegate), instead, the manager stores the rendering jobs (and results), which are identified through their URL. The interaction between jobs and thumbnails is handled via signals and slots; the thumbnails and jobs don't really know about each other. When a thumbnail is deleted (for example because the QML scene changes, or a delegate is scrolled out of view) Qt automatically cleans up the signals between it and all the objects it is connected to.

If you would implement a method of caching thumbnails, the structure of this class would look differently, of course. As long as the interface remains the same on the QML side (i.e. the availability of the given exported properties) you can plug in a different rendering manager without having to change any of the QML itself.

Some interesting methods:

startRequest(), where we first look if there is already a (finished) job for the given URL, and if not, create a new one.

```
void WebThumbnailContext::startRequest(WebThumbnail *wt)
{
    // Do we already have an active/completed job for this URL ?
    foreach(WebThumbnailJob * job, m_thumbnails) {
        if (job->getUrl() == wt->getUrl()) {
            bindThumbnailJob(wt, job);
            return;
        }
    }
    // Do we already have a queued job for this URL ?
    foreach(WebThumbnailJob * job, m_queue) {
        if (job->getUrl() == wt->getUrl()) {
            bindThumbnailJob(wt, job);
            return;
        }
    }
    // Create new job for this URL
    WebThumbnailJob * job = new WebThumbnailJob(wt->getUrl());
    bindThumbnailJob(wt, job);
    m_queue.append(job);
    process();
}
```


(C) Copyright Nokia 2013. All rights reserved.

The process() method, which keeps emptying the queue within the concurrency constraints. Note that if the concurrency is set to zero, this means unlimited, to mean that there is no restriction. Use with care!

```
void WebThumbnailContext::process()
{
    while (m_queue.count() > 0 && (m_activecount < m_concurrency || m_concurrency == 0)) {
        WebThumbnailJob * job = m_queue.takeFirst();
        connect(job, SIGNAL(jobDone(WebThumbnailJob *)), this, SLOT(jobDone(WebThumbnailJob *)));
        m_thumbnails.append(job);
        m_activecount++;
        emit activeChanged();
        // No plugins, no javascript
        job->start(false, false);
    }
}
```

And finally the internal "bindThumbnailJob" method, which associates a given thumbnail to a given job. Note that if the job has already finished earlier, which means a thumbnail is already available for the given URL, we immediately trigger the *webthumbnail*'s finished() slot. If the job is still in progress, or in the queue, we connect the various signals, for example the progress indication, to the webthumbnail instance.

```
void WebThumbnailContext::bindThumbnailJob(WebThumbnail *wt, WebThumbnailJob *job)
{
    wt->setJob(job);
    switch(job->getState()) {
        case WebThumbnailJob::Loaded: {
            wt->finished(job->getPixmap());
            break;
        }
        case WebThumbnailJob::Error: {
            wt->error();
            break;
        }
        default: {
            connect(job, SIGNAL(started()), wt, SLOT(started()));
            connect(job, SIGNAL(progress(int)), wt, SLOT(progress(int)));
            connect(job, SIGNAL(finished()), wt, SLOT(finished()));
            connect(job, SIGNAL(error()), wt, SLOT(error()));
        }
    }
}
```

WebThumbnailJob (C++)

The final piece missing is the rendering job class. This is not a QDeclarativeItem, and is in fact a private implementation detail of the entire tutorial. You could re-design the entire C++ part and still be able to use the original QML, as long as the original API is still implemented. More clearly, from the QML point of view, this *WebThumbnailJob* class does not exist.

The *WebThumbnailJob* is derived from QObject, and looks like this:

```
class WebThumbnailJob: public QObject
{
    Q_OBJECT

    public:
        enum State { Queued, Loading, Loaded, Error };

    protected:
        QWebPage  * m_webpage;
        QUrl m_url;
        QPixmap m_pixmap;
        enum State m_state;

    public:
        WebThumbnailJob(const QUrl &url, QObject *parent=0);
        ~WebThumbnailJob();

        void start(bool js, bool plugins);

        const QUrl & getUrl() const;
        const QPixmap & getPixmap() const;
        enum State getState() const;

    protected slots:
        /* To connect to m_webPage */
        void loadStarted();
        void loadFinished(bool);
        void loadProgress(int);

    signals:
        void started();
        void progress(int);
        void finished();
        void error();

        void jobDone(WebThumbnailJob *);
};
```

The QWebView member is a pointer; this is a potentially resource-intensive class which we want to create only when needed, and destroy immediately after in order to consume as few resources as possible.

Interesting parts are the loading and creating the thumbnail:

```
void WebThumbnailJob::start( bool js, bool plugins, int width, int height )
{
    m_webpage = new QWebPage;
    QWebSettings * settings = m_webpage->settings();
    settings->setAttribute( QWebSettings::AutoLoadImages, true);
    settings->setAttribute( QWebSettings::JavascriptEnabled, js);
    settings->setAttribute( QWebSettings::PluginsEnabled, plugins);
    settings->setAttribute( QWebSettings::JavaEnabled, false);
    settings->setAttribute( QWebSettings::DeveloperExtrasEnabled, false);
    settings->setAttribute( QWebSettings::OfflineStorageDatabaseEnabled, false);
```


(C) Copyright Nokia 2013. All rights reserved.

```
    settings->setAttribute( QWebSettings::OfflineWebApplicationCacheEnabled, false);
    settings->setAttribute( QWebSettings::LocalStorageDatabaseEnabled, false);
    m_webpage->mainFrame()->setScrollBarPolicy(Qt::Horizontal, Qt::ScrollBarAlwaysOff);
    m_webpage->mainFrame()->setScrollBarPolicy(Qt::Vertical, Qt::ScrollBarAlwaysOff);

    // This is an interesting property to experiment with, range 0-1
    m_webpage->mainFrame()->setZoomFactor( 0.5 );
    m_webpage->setViewportSize( QSize(width, height) );

    connect(m_webpage, SIGNAL(loadStarted()), this, SLOT(loadStarted()));
    connect(m_webpage, SIGNAL(loadProgress(int)), this, SLOT(loadProgress(int)));
    connect(m_webpage, SIGNAL(loadFinished(bool)), this, SLOT(loadFinished(bool)));
    m_webpage->mainFrame()->load( m_url );
}

// We are hardcoding 256x256 as the internal size of the thumbnails, although you can use
// any size you like in the QML (but it should be square to look normal)
void WebThumbnailJob::loadFinished(bool b)
{
    if (b) {
        QImage image( QSize(256, 256), QImage::Format_ARGB32_Premultiplied);
        QPainter painter(&image);
        painter.setRenderHint(QPainter::Antialiasing, true);
        painter.setRenderHint(QPainter::TextAntialiasing, true);
        painter.setRenderHint(QPainter::SmoothPixmapTransform, true);
        m_webpage->mainFrame()->render(&painter, QRegion(0, 0, 256, 256) );
        painter.end();

        m_pixmap = QPixmap::fromImage(image);
        delete m_webpage;
        m_webpage = 0;
        m_state = Loaded;
        emit finished(m_pixmap);
    } else {
        m_state = Error;
        emit error();

        delete m_webpage;
        m_webpage = 0;
    }
    emit jobDone(this);
}
```

The fun part

Now that the hard work in C++ is done, we can start to use the WebThumbnail in QML environments. For example, if we would like some bounce-in elastic effect when a thumbnail is loaded, like in this screenshot, this is easily done using QML, although perhaps a bit over the top. Forgive the quality of the scaling, as this was taken during an animation, during which lower quality scaling is applied for performance, the idea being that you will not notice this while an actual animation is running. The clipping of the rounded corners of the thumbnails is currently not anti-aliased and looks jagged when rotated.

And finally an example of use as a model delegate. We have used a list of the most popular websites in the world (according to this site) but this could be for example a list of bookmarks, a list of links in an RSS feed, and so on. Look into the QML XmlListModel for more inspiration and examples.

(C) Copyright Nokia 2013. All rights reserved.

The first screenshot shows the list as soon as it is loaded, after a tiny bit of scrolling downwards. The statistics at the bottom are the *WebThumbnailer* properties we discussed earlier, and they are updated in real time. What it tells us, is that we have 8 delegates currently instantiated, 7 requests in the queue, and 2 requests being rendered, which follows our concurrency setting of 2. Note that it is *not* necessarily the case that total equals active + queued! When a delegate is destroyed (since it is not needed for the current viewport) it is taken out of the total count, but its URL remains in the queue or finished list.

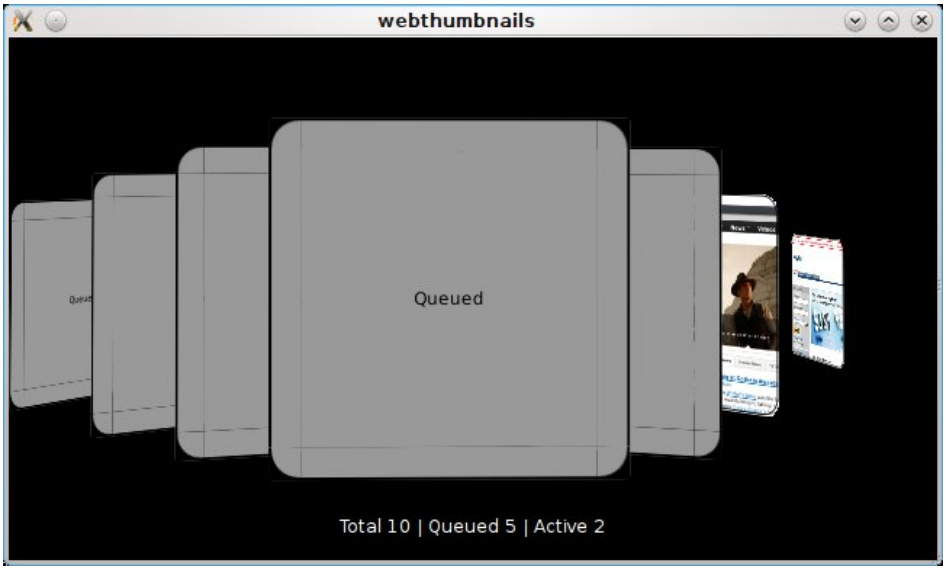
In this screenshot we've scrolled all the way to the bottom. This means that we've very briefly instantiated a delegate for every URL in the model, which is why we now have 33 requests in the queue. There are now only 7 delegates needed to render this view (this depends on how many fit in the viewport, and in this case it is 7 and not 8 because at the bottom we are exactly ending at the boundary of a thumbnail, instead of showing a partial one), and we still only render 2 requests at a time.

Tip: One improvement you can make here is to prioritize requests that are visible, when you take new requests from the queue, in order to always try and have a thumbnail for the user to look at.

And finally a screenshot where we've rendered a number of thumbnails already, which are visible and shown here. Because our queue is a simple FIFO, the rendered thumbnails are all at the top of the list. 17 requests remain in the queue.

(C) Copyright Nokia 2013. All rights reserved.

Similarly for the PathView element. The grid lines around the gray squares (edges and corners) are actually rendering artifacts that should not be there.



(C) Copyright Nokia 2013. All rights reserved.

Conclusion

Hopefully this tutorial has demonstrated how you can expose your custom objects, both visual and non-visual, to QML. Properly used, Qt Quick provides a hard separation between functionality and visualization, or business rules and presentation rules. Comments and suggestions are welcome.

Files

The source code is included here for you to play with. It has been tested only on Linux, although it should work on Windows and OS X as well. It would be very nice to see how this performs on a mobile device; this will be tested once Qt 4.7 is standard in the Nokia Qt SDK (for mobile devices). The screenshots are in nHD resolution, 360x640, i.e. the same resolution as most Nokia mobile devices, including the (former) Symbian^3 devices.

The code is free to use however you like.

File:WebThumbnail SourceCode.zip

Where from here, or future ideas

Some ideas on how this can be improved to be integrated in a real application.

- The thumbnails are currently clipped to create the rounded corners; this created jagged edges, seen especially well during rotations. Nicer would be a soft feathered clipping. Any suggestions on the best way to approach this are welcome.
- The QWebView can not easily be placed in another thread, as it is a GUI component. However, if a smooth consistent UI is needed, some of the more expensive HTML parsing operations or javascript execution need to be made into smaller parts so that the eventloop gets a chance to run and process input and drawing operations. Another approach is to delegate to a background thread, since all we care about here is the resulting thumbnail. This is an interesting subject for further study, so watch this space.
- Similarly, the final QWebFrame::render(QPainter *) operation can take from a few tens to several hunderd milliseconds - on a fast desktop. This completely blocks the UI of the application, the effect is not so bad on a desktop but will be very noticeable on a mobile device. The few 100 milliseconds are not the problem, but blocking the eventloop for that long is. Also, when the purpose is to render a thumbnail (or otherwise such a small version of the original) a lot of rendering shortcuts could be made that probably aren't, and which would reduce this rendering time further.
- It will speed up things to set a new QNetworkAccessManager for the QML operations, one which uses a disk cache. Even better is caching of the thumbnails themselves.
- We have cheated a little bit with the thumbnail sizes, and always internally render to 256x256. This is fine for QML, as long as you remember to set a square size and don't expect to look at any thumbnails much larger than that. A smarter implementation might render a website at a width of say 256, and a height which matches that width (think of a typical website being one screen wide but several screens high). The *WebThumbnail* can then always crop the correct part of the image so that it can handle different aspect ratios properly.
- For some websites you may want them to "run" for a few seconds before taking the thumbnail "snapshot". For example a video could have started playing, or some advertisement scrolled out of the way, or an annoying introduction finished.
- A very nice effect would be to render the snapshot at say 512x512, and then let the *WebThumbnail* gently slide a 256x256 "window" over this image. This can be done directly in C++ in the *WebThumbnail* component, or it can be done via QML if the *WebThumbnail* gets some extra properties such as an X and Y scrolling offset and window size. This could be particularly nice for a bookmark overview or web browser "start" page. Nokia are you listening? :)

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.