Multifunctional Image Tool: A base for image interaction

This article explains how to create a QML image interaction component with pinch and rotation gestures along with auxiliary functions for the conversion of QML Image elements to QImage instances and vice versa.

Note: This is an entry in the PureView Imaging Competition 2012Q2

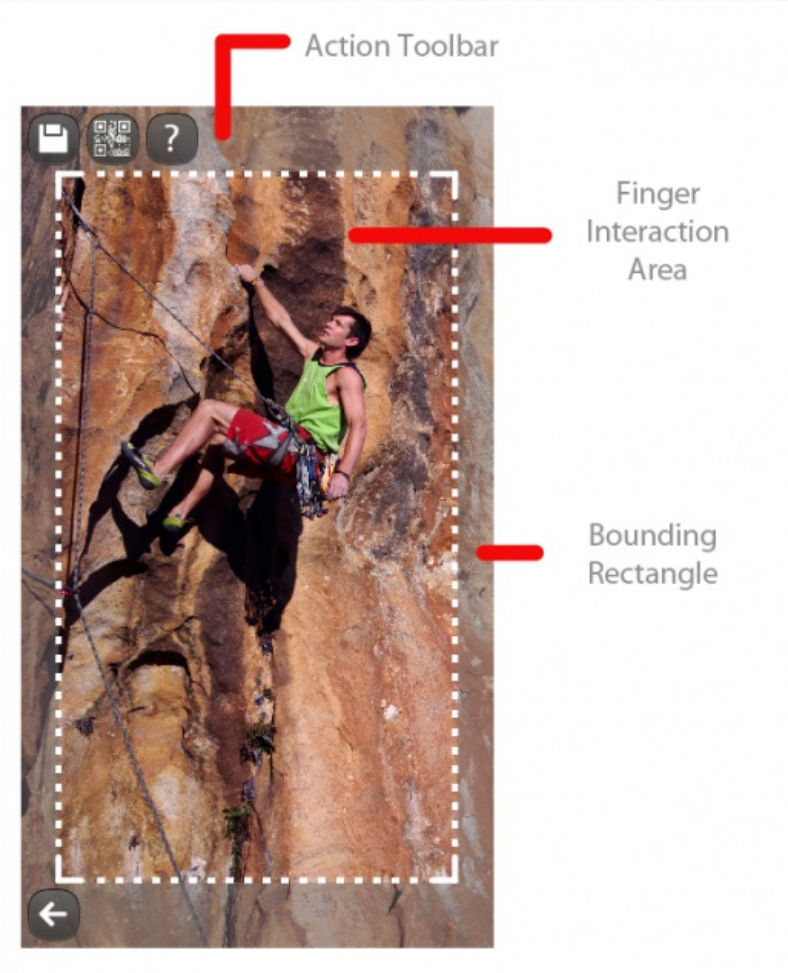
Introduction

As it is well-known by now, mobile phone world is tightly coupled with the imaging world. A great example of this saying is the creation Nokia PureView 808. A mobile phone with super-extended photographic capabilities (compared to any other mobile phone). Its huge image sensor can be compared in size with the ones placed in a number of DSLR photographic cameras...of course leaving out the changeable lenses. 41MegaPixles are able to provide us with images reaching the size of 7728 x 4354 (16:9 aspect ration). All those pixels have to be fitted into screens way smaller than them. This, creates the necessity of having a way to manipulate those images effectively and be able to interact with the image but most importantly with portions of the image.

Summary

The goal of this article is to create a QML extendable base component that can be used in image tools. It will start by explaining the finger interaction with the image which will provide scaling, rotation and the selection of image portions. Then we define the bounding rectangle that will be the user reference point to the selection of the image portion followed by the creation of an extendable button area over it to lay the Action Tools. For closing, a C++ tool named ImageHandler will be explained which is responsible for various image manipulations between C++ and QML worlds. Follows 2 screenshots (portrait & landmark) depicting the visible aforementioned components.

(C) Copyright Nokia 2014. All rights reserved.



Interaction Area

By saying Interaction Area we define the component where the image is loaded and shown which provides the ability to zoom/unzoom into the image and navigate around it. It is purely written in QML using a Flickable element, a PichArea element and an Image element. Lets see the code and right after it will be explained step by step:

```
import QtQuick 1.1

Item {
    id: mainPage

    property alias source: image.source
    property alias offsetX: flick.contentX
    property alias offsetY: flick.contentY
    property alias contentWidth: flick.contentWidth
    property alias contentHeight: flick.contentHeight
    property alias sourceSize: image.sourceSize
    property alias imageElement: originalImage

    Flickable {
        id: flick
        anchors.fill: parent
        contentWidth: minimumWidth
        contentHeight: minimumHeight
```


(C) Copyright Nokia 2014. All rights reserved.

```
        property real minimumWidth: inPortrait ? mainPage.height * aspectRatio :
mainPage.width
        property real minimumHeight: inPortrait ? mainPage.height : mainPage.width
/aspectRatio

        property real aspectRatio: (image.sourceSize.width / image.sourceSize.height)

        Image {
            id:image
            width: flick.contentWidth
            height: flick.contentHeight

            MouseArea {
                anchors.fill: parent
                onDoubleClicked: {
                    flick.contentWidth = flick.minimumWidth
                    flick.contentHeight = flick.minimumHeight
                }
            }
        }

        PinchArea {
            id: pinchArea
            width: Math.max(flick.contentWidth, flick.width)
            height: Math.max(flick.contentHeight, flick.height)

            function distance(p1, p2) {
                var dx = p2.x-p1.x;
                var dy = p2.y-p1.y;
                return Math.sqrt(dx*dx + dy*dy);
            }

            property real initialDistance
            property real initialContentWidth
            property real initialContentHeight

            onPinchStarted: {
                initialDistance = distance(pinch.point1, pinch.point2);
                initialContentWidth = flick.contentWidth;
                initialContentHeight = flick.contentHeight;
            }

            onPinchUpdated: {
  flick.contentX += pinch.previousCenter.x - pinch.center.x
  flick.contentY += pinch.previousCenter.y - pinch.center.y

  var currentDistance = distance(pinch.point1, pinch.point2);
  if(currentDistance < 5)
   return;
                var scale = currentDistance/initialDistance;

                var newHeight = initialContentHeight*scale
                var newWidth = initialContentWidth*scale

                flick.resizeContent(newWidth, newHeight, pinch.center)
            }
```


(C) Copyright Nokia 2014. All rights reserved.

```
            onPinchFinished: {
                var finalWidth = Math.max(flick.contentWidth, flick.minimumWidth)
                var finalHeight = Math.max(flick.contentHeight, flick.minimumHeight)

                //Reasure the maximum Scale
                finalWidth = Math.min(finalWidth, image.sourceSize.width)
                finalHeight = Math.min(finalHeight, image.sourceSize.height)

                flick.resizeContent(finalWidth, finalHeight, pinch.center)

                flick.returnToBounds()
            }
        }
    }

    Image {
        id:originalImage
        visible: false
        source: mainPage.source
    }
}
```

First step is to load the image and show it to the screen. For that an Image element will be used. To load an image to the Image element you just have to set the *source* property of the element and it will be loaded. Hold that the source can be either the absolute path of an image, the relative path or a URL used to request the image from a custom Image Provider (this will be also explaned along with ImageHandler further on). Keep in mind that if using the Camera element, the captured images are also provided by a default Image Provider so in method onImageCaptured the *preview* argument will work just fine as the source. So the image is loaded but with its default size which is for example 7700x4400 (just a number, nothing specific). So since the screen is normally 360x640 we will be able to see only a small part of the image. For that reason next step was to add the image in a Flickable element for us to be able to scroll/navigate around the image. The Flickable element will be anchored to fill the screen thus making our screen the "window" to the image. By setting the contentWidth and contentHeight of the Flickable element we define the actual size of the image which will define the scrolling bounds when flicking. For convenience in next steps, the size of the Image element is chosen to be bound to the size of the content of the Flickable element and not the other way around (which would also be acceptable). By this binding we ensure that the size of the content of the Flickable element always equals the size of the image making the whole image visible.

Having a so big image only at its full scale makes it useless. For that reason next step is to support zooming. To begin with, it is chosen that the initial content size of the Flickable element will be the such that the whole image will fit to screen. To make that possible we first calculate the aspect ratio of the image to be sure that we will never stretch or disform the image. These values are keep in properties *minimumWidth* & *minimumHeight*. Having the image in its mimimum scale we proceed in adding a PinchArea element above the image. A PinchArea element is an invisible element which can be applied above any other element and is used to handle 2-finger gestures. By default it supports to set a target element, the upper and lower limits of the scaling and rotation that will be applied and we should be ready. But we are not. Unfortunately PinchArea transformations are applied having as center the center of the element that it is to be controlled. Naturally a user when making a gesture, expects the gesture to be applied according to the center of the distance of his 2 fingers. Imagine the case where the user wants to zoom in the upper left corner, piches and zooms in at the center of the image. That is unacceptable. So the target element support can´t be used. In its place we will be handling the raw touch events through the functions onPichStarted, onPinchUpdated, onPinchFinished to calculate the scaling factor that will be applied centered to the center of the gesture. At every call of *onPinchUpdated* the scaling factor is calculated and is applied by resizing the content of the Flickable area using as center the center of the gesture. This takes place by the use of Flickable::resizeContent function.**Note:** this function is still preliminary. The other way would be to use a Scale element as a transformation to the Image element but that would require to manually update the x,y offset of the viewed content in the Flickable. Finally *onPinchFinished* we check the scaling to be between the selected range (**minimum**: image fits to screen - **maximum**: image is in full scale == 1).

The code of Interaction Area is placed in InteractionArea.qml making simply accessible to the rest of the program. To make its interconnection with the rest of the program there are placed a few aliases that exposes various data out of the InteractionArea.

(C) Copyright Nokia 2014. All rights reserved.

- *source*: Sets the source of the image to be interacted upon
- *offsetX*, *offsetY*: the current X,Y offset of the content in the Flickable area
- *contentWidth*, *contentHeight*: the current width of the content (or else said, the current width of the image). By dividing the original size of the image to the current size we get the total scaling factor.
- *sourceSize*: the original size of the image (*sourceSize.width*, *sourceSize.height*)
- *imageElement*: an image element holding the original image loaded without any tranformation applied. It is a separate Image element at the bottom lines of the code and is used to retrieve the unaffected image later on.

Solved issues

An interesting issue that came up when testing on real devices and desktops was that at the end of the pinch gesture the image always got disappeared. After a lot of gazing at the screen i realized that a few times there was a frame showing the image to get really small before it get disappeared. That turned out to be an error when the user removed his fingers from the screen. When the user removes the two fingers from the PinchArea, it is humanly impossible to remove them at the exact millisecond. So what was happening, there was a millisecond gap between the raising of the fingers which led the *pinch.point1* to exactly match with *pinch.point2* since the points of the 2 fingers where now represented by only on finger. This fact caused in the *onPinchUpdated* to make the scale go zero and the image to disappear. That was fixed by making a check the ensure that the distance of the two fingers is always bigger than 5 pixels or else it gets ignored. The funny thing is that at the Simulator where the first tests were held everything was ok. That makes sense because the pinch event was simulated by clicking somewhere to define the center and then any mouse dragging is replicated by creating the second touch opposite the other using as origin the center of the gesture. Any way, when leaving the mouse button both touch points get vanished thus not making a problem.

Notes about the Interaction Area

At the begging i started trying to support rotation along with the scaling. But there were bugs that at this point i couldn't overcome so the code controlling the rotation of the image is left out of the File:Multifunctional Image Tool source.zip...for now. I believe it is crucial the rotation to get embedded in the above code and i hope it will time any time soon.

Bounding Rectangle

This is a visual aid to show the user that indeed this tool has to do with cropping and generally manipulating portions of images. A frame is created with a semi-transparent grey and an inner Rectangle with the classic dotted border to indicate that something is going to be "cut". This frame consists of 4 Rectangles placed such that regardless the orientation they keep the aspect ratio of the screen. It is also used as the panel where the Action Buttons will be placed. (To see the code look up the source code at BoundingRectangle.qml)

Action Bar

The element where the action buttons will be placed. For any new operation that will be supported, a new button is appended. Each operation emits a signal to notify the other elements about the request of an action. In this implementation the buttons are defined twice. Once inside a Row which is visible only when orientation is landmark and once inside a Column when been in portrait orientation. (To see the code look up the source code at ActionBar.qml). Current action buttons used:

- Crop button: The button with the "save" image is used to initiate the saving of the selected portion of the image
- Barcode Decoder button: Initiates a barcode decoding at the specified portion of the image. Used as an example on how this tool can be used
- Question mark: ....well this is just an other button that does nothing. It is left there to indicate that this operation bar can be extended :P

Main Page

All the above elements form our application screen. Putting them all together is done in the MainPage.qml. Follows a small piece of code from that element:

```
    function saveCroppedImage()
    {
        var geometry = extractPortionGeometry();

        imageHandler.save(interactionArea.imageElement, "E:/croppedImage.png",
                          geometry.x, geometry.y, geometry.width, geometry.height)
```


(C) Copyright Nokia 2014. All rights reserved.

```
    }

    function extractPortionGeometry()
    {
        var geometry = new Object();
        var scaleRatio = 1;

        if(inPortrait)
            scaleRatio = interactionArea.contentHeight /
interactionArea.sourceSize.height
        else
            scaleRatio = interactionArea.contentWidth / interactionArea.sourceSize.width

        geometry.x = (interactionArea.offsetX + boundingRectangle.topLeft.x) /
scaleRatio
        geometry.y = (interactionArea.offsetY + boundingRectangle.topLeft.y) /
scaleRatio

        geometry.width = boundingRectangle.cropWidth / scaleRatio
        geometry.height = boundingRectangle.cropHeight / scaleRatio

        return geometry;
    }
```

The *extractPortionGeometry* function is an auxiliary function that takes as input the Bounding Rectangle and the data by Interaction Area and calculates the size of the image portion translated in coordinates that apply to the original image. Calling this function we have the exact offset in the original image from where the cropping will take place and the width & height define till where.

Image Handler

The only C++ class of the project is created to give advanced operations that QML by itself can't provide. Follows the header file of the class and later on method-by-method the source code will be explained. Initially the class was created to be registered as a context property to be used as the medium to extract the data from an Image QML element. Having the data of the image in a QImage instance the whole world of Qt opens being able to do any operation needed, in our case save the image to the file system. Later on i thought, why supporting only the passing from QML to C++ and not both ways. That is when the class got a second role, the role of an Image Provider by inheriting from QDeclerativeImageProvider. Now it is able to also provide to any QML element of any image requested as long as the image is made accessible through the function *makeImageAvailable*. An interesting point here is that by having a QMap storing images represented by a unique string the ImageHandler can work as a runtime image storage. Create any image you want at runtime, store it there and it will be accessible whenever needed. The whole packet in my opinion is a very good tool that unties our hands from issues of interconnection between C++ and QML.

```
#ifndef IMAGEHANDLER_H
#define IMAGEHANDLER_H

#include <QDeclarativeImageProvider>
#include <QImage>
#include <QMap>

class ImageHandler : public QObject, public QDeclarativeImageProvider
{
    Q_OBJECT
public:
    explicit ImageHandler(QString providerName, QObject *parent = 0);

    QString providerName();
```


(C) Copyright Nokia 2014. All rights reserved.

```
    /**
      * The method through which the images are provided
      */
    QImage requestImage(const QString &id, QSize *size, const QSize &requestedSize);

public slots:
    /**
      * Called from QML. Argument imageObj is the Image QML element.
      * The rest define the geometry of the portion that is to be
      * acquired.
      */
    QImage extractQImage(QObject *imageObj,
                         const double offsetX, const double offsetY,
                         const double width, const double height);

    /**
      * Called from QML. Argument imageObj is the Image QML element.
      * path is the actual path that the image provided will be saved
      * The rest define the geometry of the portion of the image that
      * is to be acquired.
      */
    void save(QObject *item, const QString &path,
              const double offsetX = 0, const double offsetY = 0,
              const double width = 0, const double height = 0);

    /**
      * Returns whether the file pointed by the path already exists
      */
    bool imageFileExists(const QString &path);

    /**
      * Add a QImage to the image repository to make it accesible from QML
      */
    void makeImageAvailable(const QImage image, const QString& uniqueName);

    /**
      * Remove a previously added image from repository. This will make image
      * inaccessible from QML.
      */
    void removeImage(const QString& uniqueName);

private:
    QMap<QString, QImage> imageRepository;
    QString _providerName;
};

#endif // IMAGEHANDLER_H
```

- extractQImage (QObject *imageObj, const double offsetX, const double offsetY, const double width, const double height)

The main function used to convert an Image QML element to a QImage instance. First argument is the Image QML element. In QML when calling the function use the id of the Image Element as first argument and it will be fine. The rest arguments, if not used will indicate that the QImage returned will contain exactly the same image as the original image. If any of those is set the function will return a QImage containing the portion of the image define by those variables.

- requestImage(const QString &id, QSize *size, const QSize &requestedSize)

(C) Copyright Nokia 2014. All rights reserved.

The virtual function of QDeclarativeImageProvider that had to be implemented. The implementation is pretty clear, a simple lookup to see if the image requested exists. If it does, it returns the image. If not, it returns white image matching the size of the requestedSize.

```
#include "imagehandler.h"
#include <QGraphicsObject>
#include <QImage>
#include <QPainter>
#include <QStyleOptionGraphicsItem>
#include <QDebug>
#include <QFileInfo>

ImageHandler::ImageHandler(QString providerName, QObject *parent) :
    QObject(parent),
    QDeclarativeImageProvider(QDeclarativeImageProvider::Image),
    _providerName(providerName)
{
}

QString ImageHandler::providerName()
{
    return _providerName;
}

QImage ImageHandler::extractQImage(QObject *imageObj,
                                   const double offsetX, const double offsetY,
                                   const double width, const double height)
{
    QGraphicsObject *item = qobject_cast<QGraphicsObject*>(imageObj);

    if (!item) {
        qDebug() << "Item is NULL";
        return QImage();
    }

    QImage img(item->boundingRect().size().toSize(), QImage::Format_RGB32);
    img.fill(QColor(255, 255, 255).rgb());
    QPainter painter(&img);
    QStyleOptionGraphicsItem styleOption;
    item->paint(&painter, &styleOption);

    if(offsetX == 0 && offsetY == 0 && width == 0 && height == 0)
        return img;
    else
    {
        return img.copy(offsetX, offsetY, width, height);
    }
}

void ImageHandler::save(QObject *imageObj, const QString &path,
                        const double offsetX, const double offsetY,
                        const double width, const double height)
{
    QImage img = extractQImage(imageObj, offsetX, offsetY, width, height);
    img.save(path);
}
```


(C) Copyright Nokia 2014. All rights reserved.

```
bool ImageHandler::imageFileExists(const QString &path)
{
    return QFileInfo(path).exists();
}

void ImageHandler::makeImageAvailable(const QImage image, const QString& uniqueName)
{
    imageRepository[uniqueName] = image;
}

void ImageHandler::removeImage(const QString& uniqueName)
{
    if(imageRepository.contains(uniqueName))
        imageRepository.remove(uniqueName);
}

QImage ImageHandler::requestImage(const QString &id, QSize *size, const QSize
&requestedSize)
{
    QImage* selectedImage = NULL;
    if(imageRepository.contains(id))
        selectedImage = &imageRepository[id];
    else
        selectedImage = new QImage(requestedSize, QImage::Format_ARGB32);

    if(size)
        *size = QSize(selectedImage->width(), selectedImage->height());

    return *selectedImage;
}
```

How to make the class visible to QML

After the creation of an instance of ImageProvider 2 function calls have to be done to make it fully available to QML. First add it as a context property. Thus it will be accessible (and all its methods) through QML by the name given. The second call is to register the class as an Image Provider to the declarative engine.

From the main.cpp

```
...
    ImageHandler imageHandler("imageRepository");

    QmlApplicationViewer viewer;
    viewer.rootContext()->setContextProperty("imageHandler", &imageHandler);
    viewer.engine()->addImageProvider(imageHandler.providerName(), &imageHandler);
...
```

Of course this could be implemented as a QML plugin so the above step would be unnecessary...but that is keep for the future.

Source Code project

The source code can be found File:Multifunctional Image Tool source.zip as a complete project.

EDIT: the source code has been updated to hold some small changes. The changes include the turning into platform independent code. All the Symbian specific code is commented out as well as QZXing code to make it easier for you to try it out. Also the

(C) Copyright Nokia 2014. All rights reserved.

installer for Symbian is updated which still holds the barcode decoding capabilities.

favoritas37 17:10, 29 May 2012 (EEST)

(C) Copyright Nokia 2014. All rights reserved.