Photomosaic App with Qt

This article explains how to create an app that will take a photo, upload it to a server, wait for a response image, and display it on the phone. This application uses Nokia N9 (MeeGo Harmattan), is written in Qt and QML, and uses MeeGo Touch Framework. The image created on the server is a photomosaic, and there are many other features to the application: uploading photos from galleries, viewing the photomosaic with pinch zooming, uploading the result to Facebook, and more. The server setup is explained as well.

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

Introduction

The app described in this article uses a database of 100000 open source photos from the Internet to convert any picture to a photomosaic. The images below show a photo taken using a mobile device and the resulting photomosaic.

The following sections describe the code for the application and the server.

Server setup

Since the photomosaic requires a large number of images and a lot of processing power, it cannot be run on the phone directly. In this section we describe how to set-up a Linux server with Apache and PHP that will accept requests from the phone, create the photomosaic using metapixel, and send back the image.

Image database with Metapixel

The image tiles on our test server are created from public images downloaded from Flickr® via the Flickr® API. For example, use the following script with the Ruby flickraw API:

```
require 'rubygems'
require 'flickraw'
require 'active_support'
require 'open-uri'

FlickRaw.api_key="your_api_key"
FlickRaw.shared_secret="your_shared_secret"

for j in 1..200 #number of pages of images to get
        photos = flickr.photos.getRecent :per_page => 500, :page => j #get another 500 images (500 is the maximum allowed number of
        photos.each { |x| f.puts "id:#{x.id} \"title:#{x.title}\" (#{x.latitude},#{x.longitude},#{x.accuracy}) - #{x.tags}"
                File.open("images/" + x.id + ".jpg",'w') do |i| #save each image in the images folder
                        open("http://farm#{x.farm}.staticflickr.com/#{x.server}/#{x.id}_#{x.secret}_z.jpg") { |wimage| i.puts wimage
                end
        }
end
```

Once you have a large collection of images, use the metapixel bash script to create a database of tiles for photomosaics. First, install metapixel and imagemagick

```
sudo yum install metapixel imagemagick
```

and then run metapixel-prepare to prepare the tiles (This will take a while)

```
metapixel-prepare -r /home/bob/Documents/images/ /home/bob/Documents/metapixel_tiles/ --width=50 --height=50
```

LAMP setup

This is explained in various tutorials over the net. Depending on your server architecture, use one of the following:
LAMP on Ubuntu
LAMP of Fedora
LAMP on Arch

Make sure that uploading is enabled!

PHP script

Once you have the metapixel image tiles and the server running, use a script as follows to receive a POST request, use exec to run the bash command, and keep the file available in the mosaics folder. This ensures that if the images are sent or linked from Facebook (for example), they will stay available.

```
<?php
if (($_FILES["file"]["type"] == "image/gif")
|| ($_FILES["file"]["type"] == "image/jpeg")
|| ($_FILES["file"]["type"] == "image/pjpeg")
|| ($_FILES["file"]["type"] == "image/png"))
&& ($_FILES["file"]["size"] < 2000000)) { //file is an image, and has at most 2 Mb
        if ($_FILES["file"]["error"] > 0) { //the file could not be uploaded
                echo "Return Code: " . $_FILES["file"]["error"] . "<br />";
        } else {
                if (file_exists("upload/" . $_FILES["file"]["name"])) {
                        echo $_FILES["file"]["name"] . " already exists. ";
                } else {
                        move_uploaded_file($_FILES["file"]["tmp_name"], "/var/www/upload/" . $_FILES["file"]["name"]); //move the up
                        exec ("./make_photomosaic.sh ". escapeshellarg($_FILES["file"]["name"]), &$output); //run the script that cr
                        echo "http://yourserver.com/mosaics/".$output[0].".jpg"; //the mosaic is now in mosaics for later use, with
                }
        }
} else {
        echo "Invalid file";
}
?>
```

This script requires the bash script make_photomosaic.sh:

```
#!/bin/bash
rand=$RANDOM$RANDOM$RANDOM$RANDOM #generate a random and unique name
convert -resize 640x640 upload/$1 image_${rand}.jpg #make the image to a standard format
#generate the photomosaic
metapixel --metapixel /var/www/upload/image_${rand}.jpg /var/www/upload/tmp_${rand}.jpg --library /home/bob/Documents/metapixel_tile
convert -resize 80% tmp_${rand}.jpg mosaics/${rand}.jpg #make the photomosaic smaller and move it to the mosaics folder
mv upload/$1 mosaics/original_${rand}.jpg #move the original image to mosaics (not necessary)
rm image_${rand}.jpg tmp_${rand}.jpg #delete temporary files
```

Smartphone setup

This section describes how to implement the key features of the Qt application on the Smartphone. Basic knowledge of QML and Qt programming is expected. For an introduction to these, look here for QML, and here for Qt.

QML Camera

This example uses the QML Camera Element to obtain the image from the camera. At time of writing this only supports still photos using the rear camera - however that is all that is needed for this app. Its `captureImage()` method is used to capture the image - implement the callback `onImageCaptured()` and set the Image displayed to take the value of the `preview` variable. We also implement the callback to `onImageSaved()` to save the image to a file whose location is set by `capturedImagePath`. The preview is available before the image is saved, so it is useful to display it to the user instead of waiting for the saved image. Naturally, the saved image is in higher quality, so the preview image is not ideal for all applications. For more information, look at the documentation for these: [1]

```
import QtMultimediaKit 1.1
Page {
    orientationLock: PageOrientation.LockLandscape ;
    Camera {
        id:rearcam;
        anchors.fill: parent;
        captureResolution: "1152x648" /* 16:9 resolution suitable for n9 full screen */
        onImageCaptured: { /* "preview" of captured image is available */
            previewImage.source = preview
        }
        onImageSaved: { /* capturedImage is available */
            uploader.attachment = capturedImagePath;
            uploader.send();
        }
    }
```

```
    Image {
        id: previewImage
        anchors.fill: parent;
        visible: !rearcam.visible
    }
    Button {
        width : 100; height : 100;
        anchors.right : parent.right;
        anchors.verticalCenter : parent.verticalCenter
        iconSource : "image://theme/icon-l-camera-standby"
        onClicked: {
            rearcam.captureImage();
        }
    }
}
```

Uploader

In order to upload the image via HTTP to the server, the image data are sent in binary format using HTTP POST. The response from the server is sent back to the mobile device. The image is rescaled to a smaller resolution, so that the uploaded size is ~25Kb.

The uploader code is written using Qt C++ and exposed to QML as the "Uploader" type. The connection between QML and Qt is implemented in C++ using:

```
qmlRegisterType<Uploader>("cz.vutbr.fit.pcmlich", 1, 0, "Uploader");
```

where <Uploader> is the class name, "cz.vutbr.fit.pcmlich" is the namespace, followed by the version, and "Uploader" is the class name.

This is then used in QML as shown:

```
import cz.vutbr.fit.pcmlich 1.0
Uploader {
}
```

Saving the image and sharing on Facebook is done similarly. (see source: Media:PhotoMosaicSource.tar.gz)

```
#ifndef UPLOADER_H
#define UPLOADER_H
#include <QtNetwork>
#include <QObject>
#include <QImage>
class Uploader : public QObject
{
    Q_OBJECT
private:
    QString m_attachment;
    QImage m_image;
    QString m_response;
    QNetworkAccessManager *manager;
    QNetworkReply* mainReply;
public:
    explicit Uploader(QObject *parent = 0);
    Q_INVOKABLE void send();
    Q_PROPERTY(QString attachment READ getAttachment WRITE setAttachment) // local name of uploaded file e.g. "/home/user/MyDocs/DCI
    Q_PROPERTY(QString response READ getResponse WRITE setResponse NOTIFY uploadFinished) // the http response

    // methods required by Q_PROPERTYs
    QString getResponse() { return m_response; }
    void setResponse(QString _response) { m_response = _response; }
    QString getAttachment() { return m_attachment; }
    void setAttachment(QString _attachment) { m_attachment = _attachment; }

public slots:
    void finished(QNetworkReply *reply);
signals:
    void uploadFinished(); // signal for busyIndicator
    void uploadStarted(); // signal for busyIndicator
};

#endif // UPLOADER_H
```

```
#include <QDebug>
#include <QtNetwork>
#include <QtGui>
#include <QtDeclarative>
#include "uploader.h"

Uploader::Uploader(QObject *parent) :
    QObject(parent)
{
    manager = new QNetworkAccessManager;
    connect(manager, SIGNAL(finished(QNetworkReply*)), this, SLOT(finished(QNetworkReply*)));
}

/**
 * The method which handles server http responses
 */

void Uploader::finished(QNetworkReply *reply) {
    if (reply->error() != QNetworkReply::NoError) {
        qDebug() << reply->errorString();
    }
    setResponse(reply->readAll());
    emit uploadFinished();
//    qDebug() << reply->readAll();

}

/**
 * method which sends the data to server
 * it is necessary to set ".attachment" property before.
 */
```

```
void Uploader::send() {
    if (mainReply)
        return;

    char boundary[] = "AyV04a234DsHeKHcvNds"; //random unique divider

    // loads image from file
    m_image.load(m_attachment);

    if (m_image.isNull()) {
        return;
    }
    emit uploadStarted();

    QImage tmpImage = m_image.scaled ( QSize(640, 480),Qt::KeepAspectRatio, Qt::FastTransformation );

    // save image into the buffer
    QByteArray body;
    QBuffer buffer(&body);
    buffer.open(QIODevice::WriteOnly);
    tmpImage.save(&buffer, "JPG");

    qDebug() << "image size" << tmpImage.size();
    buffer.close();

    QByteArray b;
    b.append("--").append(boundary).append("\r\n");
    b.append("Content-Disposition: form-data; name=\"file\"; filename=\"camera.jpeg\"\r\n");
    b.append("Content-Type: image/jpeg\r\n");
    b.append("\r\n");
    b.append(body);
    b.append("\r\n");

    QNetworkRequest req = QNetworkRequest(QUrl("http://yourserver.com/upload_file.php"));
    req.setHeader(QNetworkRequest::ContentTypeHeader, QVariant(QString("multipart/form-data; boundary=")+boundary));
    req.setHeader(QNetworkRequest::ContentLengthHeader, QString::number(b.size()));
    req.setRawHeader("Connection", "Close");
    req.setRawHeader("Cache-Control", "no-cache");
    req.setRawHeader("Keep-Alive", "1");
    manager->post(req, b); //POST

}
```

QML ZoomableImage

The returned image has ~250KB and is much larger than the smartphone display. In order to display the it effectively, use the ZoomableImage component from the project QuickFlickr. This component displays the image and allows zooming and scrolling using Pinch and Pan gestures.

ImageSaver

Using the QML element `Image` you can view the image from its source, but there is no direct interface to the image itself. Saving the image locally, or accessing it pixel-by-pixel is therefore difficult. The following code show typical usage of the `Image` object.

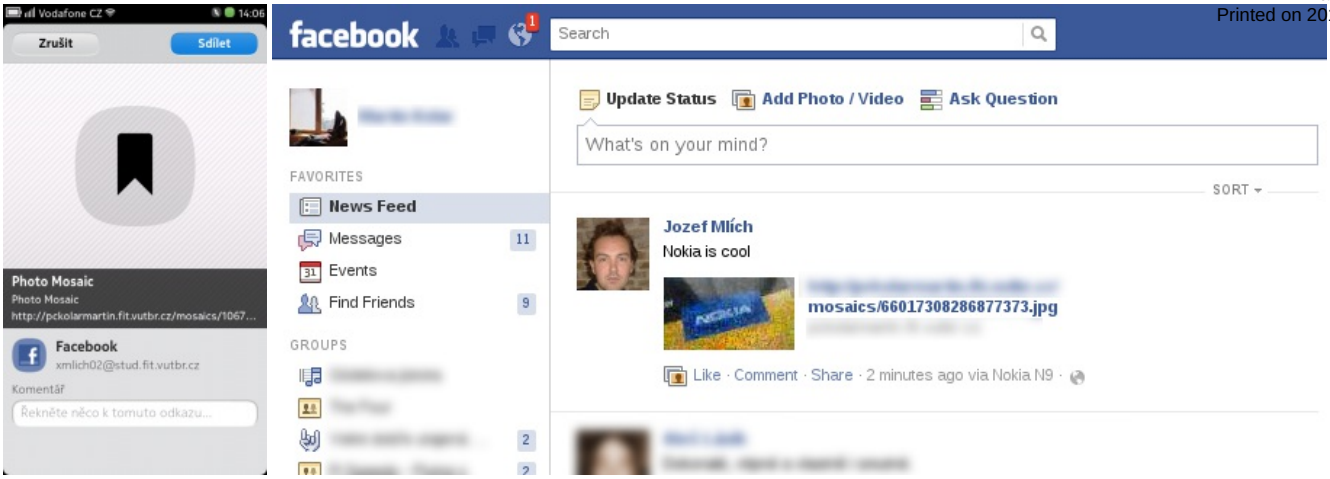
```
Image {
    source: "http://yourdomain.com/mosaics/image.jpg"
}
```

In order to access the image directly, the Image can be extended by the ImageSaver [2] wrapper, which copies the image to the QImage structure.

```
ImageSaver{
    id: saver
}
Image {
    id: photo
    source: "http://yourdomain.com/mosaics/image.jpg"
    onStatusChanged: {
        if (status == Image.Ready) {
            saver.save(photo,"/home/user/MyDocs/Pictures/mosaic.jpeg")
        }
    }
}
```

ShareHelper

To share the photomosaic in a social network (including Facebook, Twitter, DropBox, and all others compatible with the Sharing API), use the ShareHelper wrapper around the `MDataUri` class. ShareHelper provides a QML API for this.




```
CONFIG += meegotouch
CONFIG += shareuiinterface-maemo-meegotouch mdatauri
```

The code below is based on Butaca.zip. In order to see the full c++ code in context, the rest of the interface can be found in the source: Media:PhotoMosaicSource.tar.gz

```
#ifndef SHAREHELPER_H
#define SHAREHELPER_H
#include <QObject>
class ShareHelper : public QObject {
    Q_OBJECT
public:
    explicit ShareHelper(QObject *parent =
public slots:
    void share(QString title, QString url);
};
#endif // SHAREHELPER_H
```

```
#include <QDeclarativeContext>
#ifndef QT_SIMULATOR
    #include <maemo-meegotouch-interfaces/shareuiinterface.h>
    #include <MDataUri>
#endif

ShareHelper::ShareHelper(QObject *parent) :
    QObject(parent) {
}

void ShareHelper::share(QString title, QString url) {
#ifndef QT_SIMULATOR
  /* First, the data as a title, description, url i.e. "content", and its mime type are setted */
    MDataUri dataUri;
    dataUri.setMimeType("text/x-url");
    dataUri.setTextData(url);
    dataUri.setAttribute("title", title);
    dataUri.setAttribute("description", tr("Photo Mosaic"));

    QStringList items;
    items << dataUri.toString();
    /* ShareUIInterface is using N9 accounts to share data to internet */
    ShareUiInterface shareIf("com.nokia.ShareUi");
    if (shareIf.isValid()) {
      shareIf.share(items);
    } else {
      qCritical() << "Invalid interface";
  }
#else
    Q_UNUSED(title)
    Q_UNUSED(url)
#endif
}
```

The following code uses the same method as the similar code it the Upload section.

```
qmlRegisterType<ShareHelper>("cz.vutbr.fit.pcmlich", 1, 0, "ShareHelper");
```

```
import cz.vutbr.fit.pcmlich 1.0
ShareHelper {
  id: shareHelper
}
onClicked: {
  shareHelper.share("Photo Mosaic", result)
}
```

Summary

This tutorial demonstrates how simple it is to make an application that uploads an image to a server and displays the response. Many interesting features can be implemented with this, such as image search by similarity, object recognition, face recognition, an many others. It demonstrates some of the many features of the QML and Qt Mobility framework, such as social sharing, multi-touch picture browsing, and communication with a server.

We created a short video to demonstrate this application, available here: The media player is loading...