Access camera viewfinder data

This article shows how access the raw image data on the camera viewfinder using Qt.

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

Introduction

QML has a basic and easy to use camera component, as shown in the QML Camera example. Unfortunately in Qt 4.x this component does not provide any way to process the view finder frames while the camera is running.

This article explains you how to access raw viewfinder data from the camera on your Symbian-device using Qt's C++ Camera API. We then show how to integrate with the QZXing barcode scanner library in order to process QR codes straight from the viewfinder. The techniques demonstrated can be used for any number of other purposes, including manipulating captured frames prior to display.

The principles described also apply to MeeGo Harmattan, albeit with some changes required to support the device's native pixel formats, as explained in the article MeeGo Camera VideoSurface manipulation.

Accessing camera viewfinder data from Qt

To get access to viewfinder frames you need to create a QAbstractVideoSurface subclass that you set as the viewfinder for your QCamera instance. Then in your videosurface class you get access to the QVideoFrame's coming from the camera.

Subclassing QAbstractVideoSurface

Create a class that inherits `QAbstractVideoSurface` and implement the abstract methods:

```
virtual QList<QVideoFrame::PixelFormat> supportedPixelFormats(
    QAbstractVideoBuffer::HandleType handleType = QAbstractVideoBuffer::NoHandle) const
= 0;
virtual bool present(const QVideoFrame &frame) = 0;
```

In the `supportedPixelFormat()`-method you return the pixel format (usually QVideoFrame::Format_ARGB32) that you support. The `present()`-method is called when you have set the surface to your `QCamera`a as a viewfinder and you start getting frames. Here you map the image data from a `QVideoFrame` to your own QImage for processing.

See videosurface.h in CamTest for an example.

Creating a QML-element for the viewfinder

To get the viewfinder to also show on the screen we wrap it inside a small custom QML element. This is done by creating a QDeclarativeItem subclass and adding a `QCamera` instance and your newly created `QAbstractVideoSurface` subclass instance as it's members. Then you override the `paint()` method to draw the frames received from the surface to the QML element.

See viewfinderwrapper.h in CamTest for an example.

Setting the surface to QCamera

Inside your QML viewfinder element you need to initialize a `QCamera` instance normally and then set your own surface implementation as it's viewfinder.

```
    m_camera = new QCamera(this);
    if (m_camera) {
        connect(m_camera, SIGNAL(stateChanged(QCamera::State)), this,
SLOT(onStateChanged(QCamera::State)));
        connect(m_camera, SIGNAL(statusChanged(QCamera::Status)), this,
SLOT(onStatusChanged(QCamera::Status)));
        connect(m_camera, SIGNAL(error(QCamera::Error)), this,
SLOT(onCameraError(QCamera::Error)));
```


(C) Copyright Nokia 2014. All rights reserved.

```
        m_camera->start();
        m_cameraActive = true;

        m_camera->setViewfinder( static_cast<QAbstractVideoSurface*>( &m_surface ) );
        m_viewFinderActive = true;
        emit runningChanged();

    }
```

See viewfinderwrapper.cpp in CamTest for an example.

Using the QML-viewfinder

In your application's **main.cpp** register your newly created QML element so that it can be used in your QML files:

```
#include "viewfinderwrapper.h"

qmlRegisterType<ViewFinderWrapper>("Codemancers", 1, 0, "ViewFinder");
```

See main.cpp in CamTest for an example.

Then you can use the viewfinder in your QML files:

```
ViewFinder
{
    id: viewFinder
    width: parent.width
    height: width*3/4
    Component.onCompleted: {
        console.debug("ViewFinder QML-component loaded");
    }
}
```

Adding a thread for processing

It is good practice not to block the UI-thread with heavy operations so the final step is to add a thread to do the actual processing of the frames. This is done by subclassing QThread and implementing the `run()` method:

```
#ifndef PROCESSINGTHREAD_H
#define PROCESSINGTHREAD_H

#include <QThread>
#include <QImage>
#include <QQueue>

class ProcessingThread : public QThread
{
    Q_OBJECT
public:
    explicit ProcessingThread(QObject *parent = 0);
    virtual ~ProcessingThread();
signals:
    void frameProcessed();
    void queueFull();
public:
    void stop();
```


(C) Copyright Nokia 2014. All rights reserved.

```
    void addFrameToProcessingQueue(QImage frame);
private:
    virtual void run();
private:
    QQueue<QImage> m_queue;
    int m_queueMaxLength;
    bool m_stopped;
};

#endif // PROCESSINGTHREAD_H
```

We added a method to add frames to the queue for processing so we have a small buffer if some frames take longer to process than others. We also set the maximum length for the queue and when it is reached we start skipping frames to catch up.

```
#include "processingthread.h"
#include <QDebug>

static const int QUEUE_MAX_LENGTH = 3;
static const int THREAD_SLEEP_MS = 25;

ProcessingThread::ProcessingThread(QObject *parent) :
    QThread(parent), m_stopped(false), m_queueMaxLength(QUEUE_MAX_LENGTH)
{
}

ProcessingThread::~ProcessingThread()
{
    stop();
}

void ProcessingThread::stop()
{
    m_stopped = true;
}

void ProcessingThread::addFrameToProcessingQueue(QImage frame)
{
    if (m_queue.length() < m_queueMaxLength) {
        QImage threadCopy = frame.copy();
        m_queue.enqueue(threadCopy);
    } else {
        emit queueFull();
    }
}

void ProcessingThread::run()
{
    // Process until stop() called
    while (!m_stopped)
    {
        if (!m_queue.isEmpty())
        {
            QImage currentFrame = m_queue.dequeue();

            // Here you can do whatever processing you need on the frame, like detect
barcodes, etc.
```


(C) Copyright Nokia 2014. All rights reserved.

```
            emit frameProcessed();
        }
        else
        {
            // No frames in queue, sleep for a short while
            msleep(THREAD_SLEEP_MS);
        }
    }
    qDebug() << "Processing thread ending";
    exit(0);
}
```

In the `ViewFinderWrapper` we create the thread when we create the camera:

```
void ViewFinderWrapper::startCamera()
{
    Q_ASSERT(!m_camera);
    Q_ASSERT(!m_processor);

    m_processor = new ProcessingThread(this);
    connect(m_processor, SIGNAL(frameProcessed()), this, SLOT(onFrameProcessed()));
    connect(m_processor, SIGNAL(queueFull()), this, SLOT(onThreadCongested()));

    m_processor->start();

    m_camera = new QCamera(this);
    if (m_camera) {
        connect(m_camera, SIGNAL(stateChanged(QCamera::State)), this,
SLOT(onStateChanged(QCamera::State)));
        connect(m_camera, SIGNAL(statusChanged(QCamera::Status)), this,
SLOT(onStatusChanged(QCamera::Status)));
        connect(m_camera, SIGNAL(error(QCamera::Error)), this,
SLOT(onCameraError(QCamera::Error)));
        m_camera->start();
        m_cameraActive = true;
    }
}
```

When we received the frame from the videosurface, we just pass it along for processing to the thread:

```
void ViewFinderWrapper::frameReady()
{
    m_receivedFrameCounter++;
    emit frameCountChanged(m_receivedFrameCounter);

    // Get the current frame from the video surface
    QImage frame = m_surface.frame();
    // Add received frame to processing thread for processing
    if (m_processor) {
        m_processor->addFrameToProcessingQueue(frame);
    }

    // And take a copy for ourselves for drawing it on the screen
    m_currentFrame = QPixmap::fromImage(frame);
```


(C) Copyright Nokia 2014. All rights reserved.

```
    // Update the UI
    update();
}
```

Integrating QZXing

QZXing is a Qt-port of the popular ZXing barcode reading library and can be found here

Integrating it to our example is easy. Just follow the instructions in the project wiki (copy three folders to your Symbian Qt SDK). Then add a few lines to your .pro-file:

```
    # Add QZXing
    LIBS += -lqzxing
    customrules.pkg_prerules  = \
        ";QZXing" \
        "@\"$$(EPOCROOT)Epoc32/InstallToDevice/QZXing_selfsigned.sis\",(0xE618743C)"\
        " "
    DEPLOYMENT += customrules
```

In your processing thread code we instantiate a QZXing-instance and in the run-loop give QImages for it to process:

```
ProcessingThread::ProcessingThread(QObject *parent) :
    QThread(parent), m_stopped(false), m_queueMaxLength(QUEUE_MAX_LENGTH)
{
    m_zxing = new QZXing(this);
    // By default all formats are enabled but if you want to select only some:
    m_zxing->setDecoder(QZXing::DecoderFormat_QR_CODE);
}

void ProcessingThread::run()
{
    // Process until stop() called
    while (!m_stopped)
    {
        if (!m_queue.isEmpty())
        {
            QImage currentFrame = m_queue.dequeue();

            // Here you can do whatever processing you need on the frame, like detect
barcodes, etc.
            QString text = m_zxing->decodeImage(currentFrame);
            if (!text.isEmpty() && text.length() > 0) {
                emit stringDecoded(text);
            }

            emit frameProcessed();
        }
        else
        {
            // No frames in queue, sleep for a short while
            msleep(THREAD_SLEEP_MS);
        }
    }
    qDebug() << "Processing thread ending";
    exit(0);
```


(C) Copyright Nokia 2014. All rights reserved.

```
}
```

Then it is all just a matter of handling the `stringDecoded()` signal (the QZXing-version can be found in CamTest2.zip).

Note: Decoding QR-codes from viewfinder works well only with devices that have auto-focus camera like N8 or 808 PureView

Example

This article only shows you the basic principles involved. For a more concrete example see the full demo application attached in this article: CamTest.zip

- Downloadable SIS for Symbian Anna/Belle: CamTest1.sis
- Downloadable SIS for Symbian Anna/Belle with QZXing integrated: CamTest2.sis
- Source code as a ZIP-package: CamTest.zip
- Source code as a ZIP-package with QZXing integrated: CamTest2.zip

Note: you need to install the QZXing package for Symbian from: here

(C) Copyright Nokia 2014. All rights reserved.