Multi-Threaded Image Processing using Qt Camera

This article shows how to apply live image processing effects to the camera viewfinder using multi-threading and capture full resolution snapshots.

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

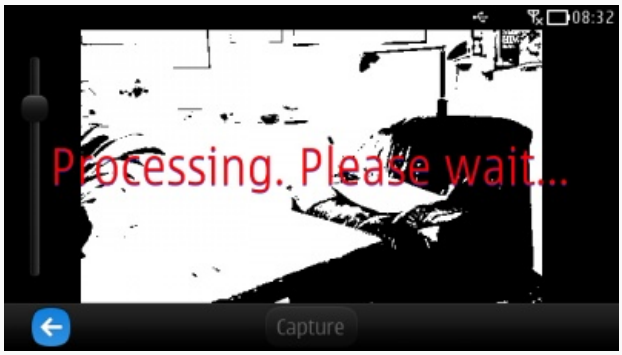
Introduction

Image processing is a resource intensive task. This article describes means how to apply image processing effects to the camera viewfinder using multi-threading to keep the user interface responsive. The following topics are covered in detail:

- Using the Qt Mobility camera class in QML.
- Spawning a worker thread for background image processing.
- Adding a simple black and white effect.
- Discussion about hardware acceleration.
- Conclusion.

The code provided here shows only the most important functional parts. The full code can be downloaded File:Fipcamera.zip.

FIP Camera Mainview



FIP Camera processing full resolution snapshot

Using the Qt Mobility Camera in QML

The QML camera component provides basic means to view and capture camera images directly from the QML scripting language. For our purpose the QML camera is not suitable because we need (i) live viewfinder image data stream and (ii) the final image as a data stream. In this article a stripped-down version of the custom QML camera component from the Qt Camera Demo is used which uses the Qt Mobility Camera classes.

Project Preparation

First, the Qt Mobility dependency and Symbian capabilities have to be added to the project (*.pro) file:

```
symbian: {
    TARGET.CAPABILITY += LocalServices \  # camera
         ReadUserData \                   #
         WriteUserData \                  # writing image file
         UserEnvironment                  # camera
}
```

On Symbian, depending on the expected memory usage the heap- and stack sizes should be increased as well:

```
symbian: {
    TARGET.EPOCSTACKSIZE = 0x14000
```


(C) Copyright Nokia 2014. All rights reserved.

```
    TARGET.EPOCHEAPSIZE = 0x20000 0x8000000
}
```

Receiving viewfinder frames from the camera

To receive video frames from the camera the QAbstractVideoSurface has to be implemented. The video surface has basically two functions: First, it tells the camera which image formats (for instance ARGB, UYVY, etc.) are supported by our application. Our sample application supports ARGB format only (caution: the Nokia N9 supports only UYVY format, thus either the effect processing has to be changed, or the UYVY data has to be converted to ARGB format before processing as for instance described here):

```
QList<QVideoFrame::PixelFormat> VideoSurface::supportedPixelFormats(
        QAbstractVideoBuffer::HandleType handleType) const
{
    Q_UNUSED(handleType);

    return QList<QVideoFrame::PixelFormat>() << QVideoFrame::Format_ARGB32; //N9:
Format_UYVY
}
```

Second it notifies our application over the FrameObserver interface when new image data is available:

```
class FrameObserver {
public:
    virtual bool updateFrame(const QVideoFrame &frame) = 0;
};
```

Defining a custom QML camera view

Next we define the `CustomCamera` class in C++ which communicates with the camera hardware using Qt Mobility camera and shows the live viewfinder image stream in QML. This class extends QDeclarativeItem which is required for including the camera class as a QML view and implements the interface `FrameObserver` to get notifications about new frames arriving from the camera viewfinder. We also define some properties which can be later accessed from QML:

- `cameraState`: information about the camera state, for instance if the camera is loaded properly.
- `availableDevices`: a list of available cameras. These are usually the front- and back facing cameras.
- `effectThreshold`: a parameter for our live image processing effect.

```
class CustomCamera : public QDeclarativeItem, public FrameObserver
{
    Q_OBJECT
    Q_ENUMS(State)

    // State properties
    Q_PROPERTY(State cameraState READ cameraState NOTIFY cameraStateChanged)

    // Devices properties
    Q_PROPERTY(QStringList availableDevices READ availableDevices)

    // Effect properties
    Q_PROPERTY(int effectThreshold READ effectThreshold WRITE effectThreshold)
```

The `updateFrame` method which receives viewfinder images is implemented from the `FrameObserver` interface. If the worker thread

(C) Copyright Nokia 2014. All rights reserved.

is not busy then the frame is copied for later processing, else it is dropped.

```
bool CustomCamera::updateFrame(const QVideoFrame &frame)
{
    if (!frame.isValid()) {
        return false;
    }

    if (m_fipThread->isProcessing()) {
        // Discard frame if worker thread is busy.
        return true;
    }

    QVideoFrame f = frame;
    if (f.map(QAbstractVideoBuffer::ReadOnly)) {
        m_fipThread->setNewFrame(&f); // send frame to worker thread
        f.unmap(); // ready for next frame from camera
    }

     return true;
}
```

Next we define the start method to initialize and start the camera:

```
void CustomCamera::start(const QString &device)
{
    destroyResources();

    m_camera = new QCamera(device.toLatin1(), this);

    // Make sure the camera is in loaded state.
    m_camera->load();

    m_videoSurface = new VideoSurface(this, m_camera);
    m_camera->setViewfinder(m_videoSurface);

    // Set the image capturing objects.
    m_cameraImageCapture = new QCameraImageCapture(m_camera);
    m_cameraImageCapture->setCaptureDestination(
                QCameraImageCapture::CaptureToBuffer);

    // Camera API
    connect(m_camera, SIGNAL(locked()), this, SIGNAL(locked()));
    connect(m_camera, SIGNAL(lockFailed()), this, SIGNAL(lockFailed()));

    connect(m_camera, SIGNAL(stateChanged(QCamera::State)),
            this, SLOT(cameraStateChanged(QCamera::State)));
    connect(m_camera, SIGNAL(stateChanged(QCamera::State)),
            this, SIGNAL(cameraStateChanged()));

    // Image capture API
    connect(m_cameraImageCapture, SIGNAL(imageCaptured(int, const QImage&)),
            this, SIGNAL(imageCaptured(int, const QImage&)));

    connect(m_cameraImageCapture, SIGNAL(imageAvailable(int, const QVideoFrame&)),
```


(C) Copyright Nokia 2014. All rights reserved.

```
            this, SLOT(imageAvailable(int, const QVideoFrame&)));

    // Set the initial capture mode to image capturing.
    m_camera->setCaptureMode(QCamera::CaptureStillImage);

    // Begin the receiving of view finder frames.
    m_camera->start();
}
```

The capture destination is set to QCameraImageCapture::CaptureToBuffer resulting in an image buffer of the captured image (instead of automatically writing it to a file). This method is available since Qt Mobility 1.2. The captured image buffer is sent through the slot `imageAvailable()`. When a full-resolution picture arrives it is copied to the worker thread (see next section).

```
void CustomCamera::imageAvailable(int id, const QVideoFrame &frame)
{
    if (frame.map(QAbstractVideoBuffer::ReadOnly))
    {
        m_fipThread->setFullResolutionFrame(&frame);
        frame.unmap();
    }
}
```

The worker thread notifies the `CustomCamera` class when a viewfinder image is processed and tells the QML view to repaint (update):

```
void CustomCamera::processedFrameAvailable()
{
    update();
}
```

The `paint` method pulls the latest processed image from the worker thread and draws it on the center of the `CustomCamera` QML view:

```
void CustomCamera::paint(QPainter *painter,
                         const QStyleOptionGraphicsItem *option,
                         QWidget *widget)
{
    // Get processed image from worker thread and draw it.
    QImage *ptrImage = m_fipThread->getLatestProcessedImage();

    if (ptrImage)
    {
        QPointF upperLeft = boundingRect().center() -
                QPointF(ptrImage->width() / 2,
                        ptrImage->height() / 2);


        // Draw the black borders.
        painter->fillRect(0, 0, upperLeft.x(), boundingRect().height(),
                          Qt::black);
        painter->fillRect(upperLeft.x() + ptrImage->width(), 0,
                          boundingRect().right(), boundingRect().bottom(),
                          Qt::black);
```


(C) Copyright Nokia 2014. All rights reserved.

```
        painter->drawImage(QRect(upperLeft.x(), upperLeft.y(),
                                 ptrImage->width(),
                                 ptrImage->height()), *ptrImage);

        // unlock
        m_fipThread->getLatestProcessedImageReady();
    }
}
```

The paint method presented above works only in Symbian. In Meego we have to draw the image using an OpenGL texture and OpenGL ES 2 shaders because the output of `painter->drawImage(...)` is overwritten by the OpenGL drawing routines. QML 2.0 which will be included with Qt 5.0 offers a scene graph API which hides the OpenGL complexity from the user and allows painting of QImages on the Nokia N9 as well. The procedure for drawing in OpenGL is as follows (not included in the example source code):

```
painter->beginNativePainting();
// 1.) Upload texture
glBindTexture(...);
glTexSubImage2d(...); // update texture data on GPU

// 2.) Bind shader
glBindProgram(program_id);

// 3.) Draw geometry with texture
glDrawElements(...);
painter->endNativePainting();
```

Before we can use our `CustomCamera` class in QML, it has to be registered somewhere before loading the QML source code (e.g. in the application's main method):

```
void FIPMain::show()
{
    qmlRegisterType<CustomCamera>("CustomElements", 1, 0, "CustomCamera");
    m_qmlView.setSource(QUrl("qrc:/qml/MainView.qml"));
    m_qmlView.showFullScreen();
}
```

The `CustomCamera` can now be easily used in QML:

```
import CustomElements 1.0

Page {
    Component.onCompleted: {
        camera.start();
    }

    CustomCamera {
        id: camera
        anchors.fill: parent
    }
}
```

Spawning a worker thread for background image processing

(C) Copyright Nokia 2014. All rights reserved.

To keep the user interface responsive a worker thread is created which handles all image effect processing. More information about threading in Qt can be found here. First, we define our class FIPThread which is responsible for image processing work:

```
class FIPThread : public QThread
{
    Q_OBJECT
public:
    // Worker loop
    void run();

    // Is an image currently processed?
    inline bool isProcessing() const {
        return m_stateProcessing;
    }

Q_SIGNALS:
    void newFrameReady();
    void fullImageSaved(QString fn);
private:
    enum TMode {
        EMode_Live,
        EMode_Captured
    };

    TMode m_currentMode;

    int m_frameIdx; // current buffer marked as ready
    QImage m_frames[2]; // double buffer
    QImage m_fullResFrame;

    bool m_stateProcessing;

    QMutex m_mutex;
    QWaitCondition m_condition;

    bool m_abort;
    bool m_restart;

    int m_effectThreshold;
};
```

`FIPThread` emits two different signals:

- `newFrameReady` is emitted when a viewfinder frame is ready.
- `fullImageSaved` is emitted when the captured image has been processed and saved.

The following member variables are defined:

- `m_currentMode`: `EMode_Live` if working on a viewfinder image or `EMode_Captured` if working on a full resolution image.
- `m_frameIdx`, `m_frames[2]`: two QImage objects are used for double buffering. One buffer at position `m_frameIdx` holds the latest processed image, while the other buffer is used during processing. If `m_frameIdx` == -1 then no processed image is available.
- `m_fullResFrame`: holds the full resolution captured image (not processed). The image is automatically freed after processing.
- `m_stateProcessing`: indicates whether the thread is currently processing an image.
- `m_effectThreshold`: the effect's parameter value.

(C) Copyright Nokia 2014. All rights reserved.

New viewfinder frames are added to the worker thread with the following method:

```
void FIPThread::setNewFrame(QVideoFrame *ptrFrame)
{
    // Drop frame if last frame is still being processed or not in live mode
    if (m_stateProcessing || m_currentMode != EMode_Live)
        return;

    QMutexLocker locker(&m_mutex);

    // Select buffer which is not in use at the moment
    if (m_frameIdx < 0) m_frameIdx = 0;
    int bufferIdx = 1 - m_frameIdx;

    if (m_frames[bufferIdx].isNull() || m_frames[bufferIdx].width() != ptrFrame->width()
||
        m_frames[bufferIdx].height() != ptrFrame->height()) {
        m_frames[bufferIdx] = QImage(ptrFrame->width(), ptrFrame->height(),
QImage::Format_ARGB32);
    }

    // Copy data to local buffer
    memcpy(m_frames[bufferIdx].bits(), ptrFrame->bits(), ptrFrame->mappedBytes());

    // Start processing
    m_abort = false;
    if (!isRunning()) {
        start(LowPriority);
    } else {
        m_restart = true;
        m_condition.wakeOne();
    }
}
```

The `setNewFrame` method copies the frame data to the locked double buffer, and starts or restarts the thread. The QMutexLocker is used to automatically release the mutex lock when the method is left. For full-resolution captured images the following method is used which incorporates decoding of the frame data (from usually EXIF Jpeg) to QImage:

```
void FIPThread::setFullResolutionFrame(QVideoFrame *ptrFrame)
{
    QMutexLocker locker(&m_mutex);

    // Decode and copy frame data to local buffer.
    // "loadFromData()" consumes a lot of time. To improve performance, the raw data
could be copied here
    // and "loadFromData()" be called in "run()" method.
   //  We want to avoid too much data copying here and thus decode in the main thread.
    if (m_fullResFrame.loadFromData(ptrFrame->bits(), ptrFrame->mappedBytes()))
    {
        m_currentMode = EMode_Captured;

        // Start processing
        m_abort = false;
        if (!isRunning()) {
```


(C) Copyright Nokia 2014. All rights reserved.

```
            start(LowPriority);
        } else {
            m_restart = true;
            m_condition.wakeOne();
        }
    }
}
```

The image processing is performed in the thread's `run` method:

```
void FIPThread::run()
{
    forever
    {
        int effectThreshold;
        TMode currentMode;
        BlackAndWhiteEffect effect;
        int curIdx;
        QImage *ptrImage;

        // We "freeze" the state by copying class variables to local variables.
        m_mutex.lock();
        m_stateProcessing = true;
        effectThreshold = m_effectThreshold;
        currentMode = m_currentMode;
        m_mutex.unlock();

        // In live mode we use double buffering
        if (currentMode == EMode_Live)
        {
            curIdx = 1 - m_frameIdx;
            ptrImage = &m_frames[curIdx];
        }
        else
        {
            curIdx = m_frameIdx;
            ptrImage = &m_fullResFrame;
        }

        // Apply effect directly to the source image (overriding the source image).
        effect.applyEffect(*ptrImage, *ptrImage, effectThreshold);

        if (currentMode == EMode_Captured)
        {
            // Save image
            QString fn =
QDesktopServices::storageLocation(QDesktopServices::PicturesLocation) +
                    QDateTime::currentDateTime().toString("yyyy-MM-dd-hh-mm-ss.jpg");
            if (ptrImage->save(fn))
                emit fullImageSaved(fn);

            // Free memory of full-resolution buffer
            m_fullResFrame = QImage();
        }
        else
        {
```


(C) Copyright Nokia 2014. All rights reserved.

```
            // Signal that a new processed frame is available.
            // There is no guarantee that *this* frame is available with
"getLatestProcessedImage()".
            // For this scenario the latest frame is sufficient.
            emit newFrameReady();
        }

        // Now we are ready for the next frame.
        m_mutex.lock();
        m_frameIdx = curIdx;
        m_stateProcessing = false;

        if (m_abort)
        {
            m_mutex.unlock();
            return;
        }
        if (!m_restart)
        {
            // Block the loop and wait for new data
            m_condition.wait(&m_mutex);
        }
        m_restart = false;
        m_mutex.unlock();
    }
}
```

First we copy member variables to local variables which might change outside the run loop during processing. A mutex locks to prevent concurrent access to memory during copying. In live mode the buffers are swapped after processing while in capture mode the full resolution image is processed, saved to a file, and memory is freed. Finally we check if the thread is about to exit (`m_abort`==true) or more work has to be done (`m_restart`==true). If both, `m_abort` and `m_restart`, evaluate to false then we wait for more work.

To get the latest processed viewfinder image the following method is used:

```
QImage * FIPThread::getLatestProcessedImage()
{
    m_mutex.lock();
    if (m_frameIdx == -1 || m_frames[m_frameIdx].isNull())
    {
        m_mutex.unlock();
        return NULL;
    }
    return &m_frames[m_frameIdx];
}
```

The mutex is locked to prevent writing to the image buffer during reading. Thus, after reading it has to be released:

```
void FIPThread::getLatestProcessedImageReady()
{
    m_mutex.unlock();
}
```

Before the worker thread can be released it has to stop processing of eventual remaining work:

(C) Copyright Nokia 2014. All rights reserved.

```
FIPThread::~FIPThread()
{
    // Wait for the worker thread to finish.
    m_mutex.lock();
    m_abort = true;
    m_condition.wakeOne();
    m_mutex.unlock();
    wait();
}
```

Adding a simple black and white effect

For this sample application a simple threshold-based black and white effect is applied:

```
bool BlackAndWhiteEffect::applyEffect(const QImage &srcImg, QImage &dstImg, const int
&thresh)
{
    // Check if in/out images match
    if (srcImg.size() != dstImg.size() || srcImg.format() != dstImg.format())
    {
        return false;
    }

    // Parameters
    int w1 = 76; // (0.299f);
    int w2 = 149; // (0.587f);
    int w3 = 29; // (0.114f);

    int intensity;
    int threshold = thresh;

    // Process image
    uint r,g,b;
    uint *ptrSrc = (uint*)srcImg.bits();
    uint *ptrDst = (uint*)dstImg.bits();
    uint *end = ptrSrc + srcImg.width() * srcImg.height();
    while (ptrSrc != end) {
        // Extract RGB components from the source image pixel
        r = (*ptrSrc&0xff);
        g = (((*ptrSrc)>>8)&0xff);
        b = (((*ptrSrc)>>16)&0xff);

        // Gray (intensity) from RGB
        intensity = ((w1 * r) + (w2 * g) + (w3 * b)) >> 8;

        // Decide between black and white based on threshold
        if (intensity < threshold)
        {
            r = g = b = 0;
        }
        else
        {
            r = g = b = 255;
        }

        // "Mix" rgb values and save to destination image
```


(C) Copyright Nokia 2014. All rights reserved.

```
        *ptrDst = r | (g<<8) | (b<<16) | 0xFF000000;

        // Jump to next pixel
        ptrSrc++;
        ptrDst++;
    }

    return true;
}
```

First, we check if source and destination images' metrics match. Then for each pixel the intensity is calculated. If the intensity is below a given threshold then the pixel color is set to black, else to white. Intensity is calculated by weighting the red, green, and blue color components (assuming red, green, blue are in the range between 0 and 255):

```
Intensity = red*0.299 + green*0.587 + blue*0.114;
```

When image data is processed on the CPU (not GPU) then integer operations are often much faster than floating point operations (note: some compilers convert/optimize floating point operations to integer operations automatically). For image processing this can account in huge processing speed gains. Our intensity value can be calculated using only integers (at the cost of loss of accuracy):

```
Intensity = (red*76 + green*149 + blue*29) / 256;
```

The float values have been converted to integers by multiplication of 256. The accuracy can be increased by using higher factors than 256 but it must be paid attention to buffer overruns. Another mean to gain performance is the use of shift operations (often automatically applied by the compiler). Here the trick is for instance to get rid of multiplication and division of integers by using shifts, where "<<" shifts left (multiplication) and ">>" shifts right (division):

```
Intensity = (red*76 + green*149 + blue*29) >> 8;
```

Adding a control for live user interaction

Our simple black and white effect has one parameter, a threshold, which decides which intensities are marked as black, and which ones are white. In the QML file we add a slider to control this threshold:

```
Slider {
    id: sldThreshold
    minimumValue: 0
    maximumValue: 255
    stepSize: 1
    orientation: Qt.Vertical
    onValueChanged: camera.effectThreshold = value
}
```

Each time the slider's value is changed, the `CustomCamera` component is notified which forwards the parameter to the worker thread:

```
void CustomCamera::effectThreshold(int thresh)
{
    m_fipThread->setEffectThreshold(thresh);
}
```


(C) Copyright Nokia 2014. All rights reserved.

```
void FIPThread::setEffectThreshold(const int &thresh)
{
    QMutexLocker locker(&m_mutex);
    m_effectThreshold = thresh;
}
```

Discussion about hardware acceleration

The presented effect is calculated on the CPU without specific hardware acceleration. Generally, there are two means for hardware acceleration on mobile devices:

- ARM specific instructions (assembler)
- OpenGL ES shaders

ARM assembler code using vectorization has huge performance potential but is hard to develop and can be incompatible between different device models. A use-case for ARM assembler is for instance UYVY to RGB24 conversion on the Nokia N9 (as an alternative to the method presented in this article).

OpenGL ES shaders are compatible between different models (with small tweaks) starting with Symbian^3 but performance is heavily affected by the time required to upload image data to the GPU. Besides, many mobile GPUs are limited memory and texture size wise allowing only small images (e.g. viewfinder size) to be easily processed. A use case for OpenGL is the live effect preview using QML 1.2 on the Nokia N9 (see here). Other use cases for OpenGL are effects which heavily incorporate floating point calculations which cannot be converted to integer arithmetic.

The next table gives a comparison of processing and drawing times in milliseconds (ms) for ARM, OpenGL, and CPU-based conversion of UYVY to RGBA data on a Nokia N9 PR 1.2 (mean over 300 runs; α=0.05 for t-test):

ARM	CPU	ARM/CPU drawing	OpenGL + drawing	OpenGL upload
5	9.09	5.69	7.82	18.03

It can be seen that ARM-based conversion is the fastest and sums up to 10.69ms for conversion and drawing. The CPU-based conversion requires almost double the time and sums up to 14.78ms. OpenGL ES 2 shader based drawing and conversion requires 7.82ms but the time for upload is very high and requires 18.03 which sums up to a total of 25.85ms.

From a performance perspective the best means for data conversion and drawing is ARM-based conversion with "normal" drawing. But with QML 1.2 on the Nokia N9 only OpenGL-based drawing is supported. Thus, if QML is used, the best method is to do conversion and drawing with OpenGL ES 2 shaders. Finally, it should be noted that all methods are fine for live video preview because with a frame rate of 30fps the available frame time is 33.33ms.

When should multi-threading be used?

Multi-threading incorporates additional processing and memory costs for thread management, inter-thread communication, locking, and data copying. As a rule-of-thumb multi-threading should be used if a task (e.g. image processing) is expected to take longer than 1 second (see Gnome guidelines for desktop here; users of mobile devices accept only very short response times!).

This article focuses on multi-threading for user interface responsiveness by processing live data in a worker thread. But multi-threading is also useful when multiple tasks can be run in parallel, as for instance in the QHdrCamera project where image processing starts during capturing of remaining pictures which decreases processing time significantly.

Summary

This article presents a brief overview on how to apply near-real time image processing effects to a camera viewfinder using QML and how to capture full-resolution snapshots. It outlines how image processing can be moved to a worker thread and how to handle concurrent access using double buffering and mutex. Finally, optimizations are discussed.

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.