How to add multithread support to your Qt application

This article explains how to use multi-threading on Qt, using the context of the QHdrCamera for the explanation.

Introduction

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

QHdrCamera is a QML component for creating High Dynamic Range images. The component captures three photos at different exposure ranges and combines the best-exposed regions in each using appropriate algorithms (see QHdrCamera component for High Dynamic Range Imaging for more information).

The original single-threaded implementation first captured and saved all the images and then loaded them and started calculation of the HDR image. This was very slow, primarily because of the time taken to save and load images on some hardware.

Part of the solution to this problem has been to use multi-threading: as soon as the first image is saved we now start loading and processing it. As much of the loading and processing is now concurrent with image capture, the total execution time is much reduced.

The solution provided uses QRunnable interface and QThreadPool class.

Multithread Image Loading

`QRunnable` is an interface class for new threads - all you need to do is implement its `run()` function with the code that can be run concurrently. The runnable objects are added to the `QThreadPool` which allows the `QRunnable` objects to be re-used (this is more efficient than creating multiple threads).

The first step is to implement the `QRunnable` interface:

```
class LoadingThread : public QRunnable
{
public:
    LoadingThread(QString*, QImage*);

private:
    QString* path;
    QImage* image;

protected:
    void run();
};

LoadingThread::LoadingThread(QString* path, QImage* image){
    this->path = path;
    this->image = image;
}

void LoadingThread::run()  {
    if( image->load(*path) ){
        qDebug() << "Image loaded";
    } else {
        qDebug() << "Error during image loading " << *path;
    }
}
```

QThreadPool

The QML Camera component provides the `onImageSaved ( path )` signal, called after the image has been written to the filesystem, in this context connected to `setShot` slots. Here we create the `LoadingThread` to start memory loading process in the background.

```
void QHdrCamera::setShot1(QString path) {
    if( !QFile::rename(path, QDir::toNativeSeparators(s_shot1)) ) {
        qDebug() << "Impossible to rename shot1:" << path;
    } else {
        LoadingThread* tLoad = new LoadingThread(&s_shot1, &shot1);
        QThreadPool::globalInstance()->start(tLoad);
        emit savedShot1();
    }
}

void QHdrCamera::setShot2(QString path) {
    if( !QFile::rename(path, s_shot2) ) {
        qDebug() << "Impossible to rename shot2";
    } else {
        LoadingThread* tLoad = new LoadingThread(&s_shot2, &shot2);
        QThreadPool::globalInstance()->start(tLoad);
        emit savedShot2();
    }
}

void QHdrCamera::setShot3(QString path) {
    if( !QFile::rename(path, s_shot3) ){
        qDebug() << "Impossible to rename shot3";
```


(C) Copyright Nokia 2013. All rights reserved.

```
    } else {
        LoadingThread* tLoad = new LoadingThread(&s_shot3, &shot3);
        QThreadPool::globalInstance()->start(tLoad);
        emit savedShot3();
    }
}
```

The application has to wait until all threads finish their jobs. `QThreadPool` provides a method called `waitForDone()` that does that.

```
QThreadPool::globalInstance()->waitForDone();
```

Summary

Multi-threading has enormously improved the performance of the QHdrCamera QML component by allowing the long running image save/load and initial processing steps to be carried out concurrently with image capture using the camera. Not only is the total execution time reduced, but the perceived behaviour is also better because the UI remains responsive to user input during calculation.

While multi-threading might not always provide a "real" improvement in execution time it is very useful in cases like this, where the different tasks don't need to use the same resources at the same time. Even where the total execution time remains the same, threading should be considered for its benefit of keeping the UI responsive.

Using a `QThreadPool` and `QRunnable` is an efficient way to implement multi-threading on Qt as threads are created when needed and can be re-used.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.