

Contents

Pimp My Widgets Developer Contest	1
Writing ODF Files with Qt	1
Keeping the GUI Responsive	3
The Panel Stack Pattern	6
Poppler: Displaying PDF Files with Qt	9
Qt News	12

Pimp My Widgets Developer Contest

Qt Developer Days, Munich – 14 October 2008 - Do you break out into a rash when you see an ugly user interface? Does adding style, animation, functionality and bling to applications large and small make you grin? Then the “Pimp My Widgets” Developer Contest is for you!



Qt Developer Contest 2008

Qt Software is kickin' out a new developer contest, throwin' down a challenge to the Qt community of developers to create innovative, attractive and functional and blingin' widgets using Qt across all supported platforms.

Qt developers are able to “pimp” widgets and simple applications from several categories for a chance to win the grand prize of a Segway® i2 Personal Transporter, or one of three Nokia N810 Internet Tablets. A call for entries was issued this morning via a YouTube Video, <http://www.youtube.com/watch?v=p2pXXAVkhW8>, and developers are encouraged to submit videos of their projects in response.

Submission deadline: 31 December 2008

More information, including guidelines, categories and judging criteria can be found at <http://trolltech.com/pimpmywidgets>

Technology Preview release of Qt 4.5

Qt 4.5 brings a ton of new features aimed at achieving our three main design goals for the release:

1. To improve the runtime performance of Qt-based applications.
2. To upgrade our Qt Webkit Integration to allow you to realize the full potential of the best browser engine on the market in your Qt applications.
3. To future proof your investment on the Mac platform by adding 64-bit support.

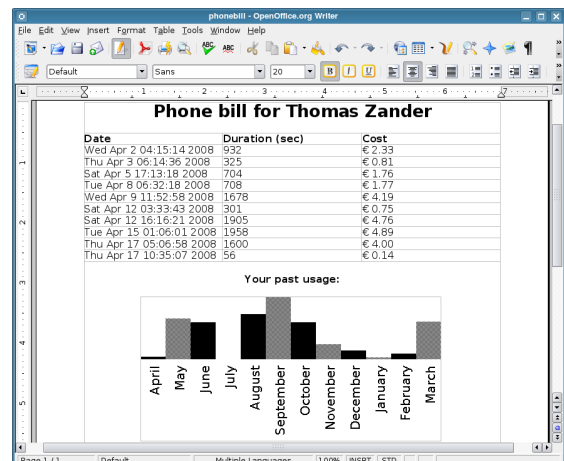
Preview packages are available now on our Website for download—try out the new features for yourself, and tell us what you think! For more information, please visit the Qt Developer Zone:

<http://www.trolltech.com/developer/preview-qt-4.5>

Writing ODF Files with Qt

The upcoming release of Qt 4.5 marks the appearance of the QTextDocumentWriter class, making it possible to create OpenDocument Format (ODF) files from any Qt text document. This opens the door to automated document creation and distribution in a standards-compliant format that users can open in a wide variety of word processors.

A prime use case for automated document creation is report generation. An example of this is a store that performs periodic inventory checks and, thus, needs to know if what the database thinks is on the shelves actually reflects real life. Consider some software that takes the database records and creates a readable document for one day's worth of work. That document can then be exported to ODF and sent to the person who does the work.



In this article we will write a simple report generator in the form of a phone bill creator. We will create one class, the PhoneBillWriter, to represent the phone bill for one client, and you can modify it or use it as a starting point for your own reporting needs.

Getting Started

The PhoneBillWriter class has the following definition:

```
class PhoneBillWriter
{
public:
    PhoneBillWriter(const QString &client);
    ~PhoneBillWriter();
    struct PhoneCall {
        QDateTime date;
        int duration; // in seconds
        int cost; // in euro-cents
    };
    void addPhoneCall(const PhoneCall &call);
    void addGraph(QList<int> values,
                  const QString &subtext);
    void write(const QString &fileName);

private:
    QTextDocument * const m_document;
    QTextCursor m_cursor;
};
```

This allows us to create one `PhoneBillWriter` for each bill and fill it with data, using calls to the `addPhoneCall()` method. After we have added all our data we call the `write()` method to finish the current bill. This is a classic case of using the builder pattern.

Internally, the class uses a `QTextDocument` as can be seen in the list of private members. The `QTextDocument` class is used in a lot of places in Qt and its widgets. `QTextEdit` uses one and you can access it using the `QTextEdit::document()` method. The `QTextDocument` is a text document with the ability to contain structured text as well as markup (bold and italic) and much more. See Qt's *Text Edit* demo for an overview.

The class `QTextCursor` is provided to allow the content of a text document to be manipulated. This is very much in the spirit of having a blinking cursor on your word processor and having the ability to move the cursor around and insert text. The `QTextCursor` class gives us the ability to do all this programatically.

Writing the Bill

The approach used in this report writer is to create a `QTextDocument` in the constructor and add the header and nice formatting information that will be the same for each phone bill. Here's a simple implementation:

```
PhoneBillWriter::PhoneBillWriter(const QString &client)
    : m_document(new QTextDocument()),
      m_cursor(m_document)
{
    m_cursor.insertText(QObject::tr(
        "Phone bill for %1\n").arg(client));

    QTextTableFormat tableFormat;
    tableFormat.setCellPadding(5);
    tableFormat.setHeaderRowCount(1);
    tableFormat.setBorderStyle(
        QTextFrameFormat::BorderStyle_Solid);
    tableFormat.setWidth(QTextLength(
        QTextLength::PercentageLength, 100));
    m_cursor.insertTable(1, 3, tableFormat);
    m_cursor.insertText(QObject::tr("Date"));
    m_cursor.movePosition(QTextCursor::NextCell);
    m_cursor.insertText(QObject::tr("Duration (sec)"));
    m_cursor.movePosition(QTextCursor::NextCell);
    m_cursor.insertText(QObject::tr("Cost"));
}

PhoneBillWriter::~PhoneBillWriter()
{
    delete m_document;
}
```

We start by adding a personal note and follow this with a table and the table header, which we fill them with the header labels. We don't need to specify in advance how many rows there are in the table.

The interesting bits are the `m_cursor` usage, which includes call to methods like `insertTable()` and `insertText()` to actually modify the document.

The `QTextCursor` also understands the concepts of selecting and removing text. A very powerful method on the cursor is `movePosition()`, to which you can pass one of the many values from its `MoveOperation` enum. This makes the class really behave like a cursor since all the usual navigation operations are there. In this case we use a navigation operation that's new in Qt 4.5: the `NextCell` operation moves the cursor to the start of the next table cell. As a result, the text passed to the following `insertText()` call will end up at the start of the table cell.

After setting up the header of the document we are ready to add actual user information to it. The caller can add individual phone calls using the `addPhoneCall()` method, passing in the individual fields that we show in the table.

```
void PhoneBillWriter::addPhoneCall(
    const PhoneBillWriter::PhoneCall &call)
{
    QTextTable *table = m_cursor.currentTable();
    table->appendRows(1);
    m_cursor.movePosition(QTextCursor::PreviousRow);
    m_cursor.movePosition(QTextCursor::NextCell);
    m_cursor.insertText(call.date.toString());
    m_cursor.movePosition(QTextCursor::NextCell);
    m_cursor.insertText(QString::number(call.duration));
    m_cursor.movePosition(QTextCursor::NextCell);

    QChar euro(0x20ac);
    m_cursor.insertText(QString("%1 %2").arg(euro)
        .arg(call.cost / (double) 100, 0, 'f', 2));
}
```

To show the phone call later, we add a row to our table and then continue to add the text to the document for each of the table cells. We call the appropriate `QString` methods to convert the integer data into text so we can fine-tune the way our data is shown. After all, the goal in our example is to make a pretty looking version of the raw data, and properly formatted text is the way to get there.

Having just text, even with tables, makes for boring reading. Adding graphs or other pictures is always a good way to make a document much more readable. This document talks about creating an ODF document, and the OpenDocument Format allows us to embed images into the final file, unlike HTML, for example.

In order to get the document into the final ODF file all we need to do is insert the image into the `QTextDocument`. It will not be a big surprise to learn that there is a `QTextCursor::insertImage()` method to do exactly this.

For our example document, we wanted to add a graph to show the amount of calls the client made in the last few months. For this the following code was used:

```
QList<int> callsPerMonth;
callsPerMonth << 6 << 84 << 76 << 0 << 93 << 128 << 76
               << 31 << 19 << 4 << 12 << 78;
phoneBill.addGraph(callsPerMonth, "Your past usage:");
```

To actually create the graph, our solution is to create a `QImage`, use a `QPainter` to draw the values onto it, and then insert the image into the document at the right location.

```
void PhoneBillWriter::addGraph(QList<int> values, const
    QString &subtext)
{
    const int columnSize = 10;
    int width = values.count() * columnSize;
    int max = 0;
    foreach (int x, values)
        max = qMax(max, x);
    QImage image(width, 100, QImage::Format_Mono);
    QPainter painter(&image);
    painter.fillRect(0, 0, image.width(), image.height(),
        Qt::white); // background
    for (int index = 0; index < values.count(); ++index) {
        // Adjust scale to our 100 pixel tall image:
        int height = values[index] * 100 / max;
        painter.fillRect(index * columnSize,
            image.height() - height, columnSize, height,
            Qt::black);
    }
    painter.end();

    QTextCursor cursor(m_document);
    cursor.movePosition(QTextCursor::End);
    cursor.insertText(subtext);
    cursor.insertBlock();
    cursor.insertImage(image);
}
```

First, we calculate the required width of our graph based on the amount of values passed in. We choose to always have a height

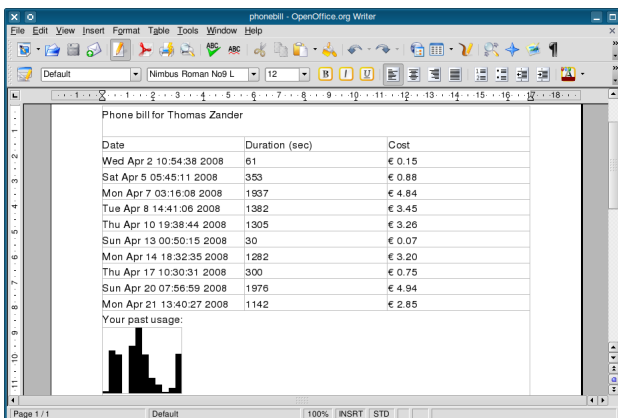
of 100 pixels and scale the values to fit in that range since most graphs are designed to show the relative sizes of the values to each other.

We use a monochrome format for the image, which means only two colors. You can use any image format you want for ODF, though. Two colors just seems enough for this use case.

After creation of the actual image we create a new `QTextCursor` and move it to the end of the document, where we insert the graph. An important note is that this cursor has a position that may be different from the `m_cursor` object. The reason for creating a new cursor here is to make sure the user can still call `addPhoneCall()` after the `addGraph()` call without affecting the positioning of the textual content.

The last part of our `PhoneBillWriter` class is to actually create the OpenDocument Format file from all the information we put into the writer object. The hard work is all done by the `QTextDocumentWriter` class, which is capable of writing to various formats, like plain text, HTML and ODF. The default is ODF, making the implementation trivial for us:

```
void PhoneBillWriter::write(const QString &fileName)
{
    QTextDocumentWriter writer(fileName);
    writer.write(m_document);
}
```



The above image shows the final document in OpenOffice.org. As it stands, it's intentionally simple for demonstration purposes, but there's a slightly more decorative example included alongside it in the archive available from the *Qt Quarterly* Web site.

Drawing Conclusions

The example we have presented is fairly simple and is focused on showing the basic ODF writing capabilities of `QTextDocumentWriter` rather than making the output look as pretty as possible.

Qt has plenty of features that we can use to improve the appearance of the phone bill and usefulness of the writer. For example, we could use the Qt SQL module to extract real information from an existing database and use `QPainter` to draw more visually appealing graphs and charts.

We could also create a nice frontend application to help the user create reports, or use Qt's ODF capabilities to do something completely unrelated to phone bill, reports or accounting. What you create with this new feature is up to you!

Thomas Zander is a Software Engineer at Nokia, Qt Software in Oslo, Norway where he works on everything related to text. In his spare time, when not working on KOffice, he enjoys spending time in the Norwegian countryside.



Keeping the GUI Responsive

At QtCentre people come to us with the recurring problem of their GUIs freezing during long operations. The issue is not difficult to overcome but you can do it in many different ways, so I'd like to present a range of possible options that can be used depending on the situation you are dealing with.

Performing Long Operations

The first thing to do is to state the domain of the problem and outline paths that can be taken to solve it. The aforementioned problem can take one of two forms. The first variation is when a program has to execute a calculation-intensive task that is described as a series of operations to be performed in sequence in order to obtain the final result. An example of such a task is calculating a Fast Fourier Transform.

The second variation is when a program has to trigger some activity (for instance a network download) and wait for it to be completed before continuing to the next step of the algorithm. This variation of the problem is, in itself, easy to avoid when using Qt because most of the asynchronous tasks performed by the framework emit a signal when they have finished doing their job, and you can connect it to a slot that will continue the algorithm.

During the calculations (regardless of any usage of signals and slots) all event processing gets halted. As a result, the GUI is not refreshed, user input is not processed, network activity stops and timers don't fire—the application looks like it's frozen and, in fact, the part of it not related to the time-intensive task is frozen. How long are "long operations"? Everything that will distract the end user from interacting with the application is long. One second is long, everything longer than two seconds is definitely *too long*.

Our aim in this article is to keep the functionality while preventing the end user from getting irritated by a frozen GUI (and the network and timers). To do that let's take a look at possible classes of solutions and domains of the problem.

We can reach our final goal of performing calculations in one of two ways—either by doing the computation in the main thread (the *single-threaded approach*) or in separate threads (the *multithreaded approach*). The latter is widely known and used in the Java world, but it is sometimes abused where one thread would do the job just fine. Contrary to popular opinion, threads can often slow down your application instead of speeding it up, so unless you are sure your program can benefit from being multithreaded (either in terms of speed or simplicity), try to avoid spawning new threads simply because you can.

The domain of the problem can be treated as one of two cases. Either we can or cannot divide the problem into smaller parts like steps, iterations or sub-problems (usually it shouldn't be monolithic). If the task can be split into chunks, each of them can either depend on others or not. If they are independent, we can process them at any time and in arbitrary order. Otherwise, we have to synchronize our work. In the worst case, we can only do one chunk at once and we can't start the next one until the previous one is finished. Taking all these factors into consideration, we can choose from different solutions.

Manual event processing

The most basic solution is to explicitly ask Qt to process pending events at some point in the computation. To do this, you have to call `QCoreApplication::processEvents()` periodically.¹ The follow-

¹Actually you should call `QCoreApplication::sendPostedEvents()`. See the documentation for `processEvents()`.

ing example shows how to do this:

```
for (int i = 3; i <= sqrt(x) && isPrime; i += 2) {
    label->setText(tr("Checking %1...").arg(i));
    if (x % i == 0)
        isPrime = false;
    QApplication::processEvents();
    if (!pushButton->isChecked()) {
        label->setText(tr("Aborted"));
        return;
    }
}
```

This approach has significant drawbacks. For example, imagine you wanted to perform two such loops in parallel—calling one of them would effectively halt the other until the first one is finished (so you can't distribute computing power among different tasks). It also makes the application react with delays to events. Furthermore the code is difficult to read and analyze, therefore this solution is only suited for short and simple problems that are to be processed in a single thread, such as splash screens and the monitoring of short operations.

Using a Worker Thread

A different solution is to avoid blocking the main event loop by performing long operations in a separate thread. This is especially useful if the task is performed by a third party library in a blocking fashion. In such a situation it might not be possible to interrupt it to let the GUI process pending events.

One way to perform an operation in a separate thread that gives you most control over the process is to use `QThread`. You can either subclass it and reimplement its `run()` method, or call `QThread::exec()` to start the thread's event loop, or both: subclass and, somewhere in the `run()` method, call `exec()`. You can then use signals and slots to communicate with the main thread—just remember that you have to be sure that `QueuedConnection` will be used or else threads may lose stability and cause your application to crash.

There are many examples of using threads in both the Qt reference documentation and online materials, so we won't make our own implementation, but instead concentrate on other interesting aspects.

Waiting in a Local Event Loop

The next solution I would like to describe deals with waiting until an asynchronous task is completed. Here, I will show you how to block the flow until a network operation is completed without blocking event processing. What we could do is essentially something like this:

```
task.start();
while (!task.isFinished())
    QApplication::processEvents();
```

This is called *busy waiting* or *spinning*—constantly checking a condition until it is met. In most cases this is a bad idea, it tends to eat all your CPU power and has all the disadvantages of manual event processing.

Fortunately, Qt has a class to help us with the task: `QEventLoop` is the same class that the application and modal dialogs use inside their `exec()` calls. Each instance of this class is connected to the main event dispatching mechanism and, when its `exec()` method is invoked, it starts processing events until you tell it to stop using `quit()`.

We can use the mechanism to turn asynchronous operations into synchronous ones using signals and slots—we can start a local event loop and tell it to exit when it sees a particular signal from a particular object:

```
QNetworkAccessManager manager;
QEventLoop q;
QTimer tT;

tT.setSingleShot(true);
connect(&tT, SIGNAL(timeout()), &q, SLOT(quit()));
connect(&manager, SIGNAL(finished(QNetworkReply*)),
        &q, SLOT(quit()));
QNetworkReply *reply = manager.get(QNetworkRequest(
    QUrl("http://www.qtcentre.org")));

tT.start(5000); // 5s timeout
q.exec();

if(tT.isActive()){
    // download complete
    tT.stop();
} else {
    // timeout
}
```

We use one of the new additions to Qt—a network access manager—to fetch a remote URL. As it works in an asynchronous fashion, we create a local event loop to wait for a `finished()` signal from the downloader. Furthermore, we instantiate a timer that will terminate the event loop after five seconds in case something goes wrong. After connecting appropriate signals, submitting the request and starting the timer we enter the newly created event loop. The call to `exec()` will return either when the download is complete or five seconds have elapsed (whichever comes first). We find out which is the case by checking if the timer is still running. Then we can process the results or tell the user that the download has failed.

We should note two more things here. Firstly, that a similar approach is implemented in a `QxtSignalWaiter` class that is part of the `libqxt` project (<http://www.libqxt.org>). Another thing is that, for some operations, Qt provides a family of “wait for” methods (for example `QIODevice::waitForBytesWritten()`) that will do more or less the same as the snippet above but without running an event loop. However, the “wait for” solutions will freeze the GUI because they do not run their own event loops.

Solving a Problem Step by Step

If you can divide your problem into subproblems then there is a nice path you can take to perform the computation without blocking the GUI. You can perform the task in short steps that will not obstruct event processing for longer periods of time. Start processing and, when you notice you have spent some defined time on the task, save its state and return to the event loop. There needs to be a way to ask Qt to continue your task after it is done with events.

Fortunately there is such a way, or even two. One of them is to use a timer with an interval set to zero. This special value will cause Qt to emit the timeout signal on behalf of the timer once its event loop becomes idle. If you connect to this signal with a slot, you will get a mechanism of calling functions when the application is not busy doing other things (similar to how screen savers work). Here is an example of finding prime numbers in the background:

```
class FindPrimes : public QObject
{
    Q_OBJECT
public:
    FindPrimes(QObject *parent = 0) : QObject() {}
public slots:
    void start(qlonglong _max);
private slots:
    void calculate();
signals:
    void prime(qlonglong);
    void finished();
```



```
private:
    qulonglong cand, max, curr;
    double sqrt;
    void next(){ cand+=2; curr = 3; sqrt = ::sqrt(cand); }
};

void FindPrimes::start(qulonglong _max)
{
    emit prime(1); emit prime(2); emit prime(3);
    max = _max; cand = 3; curr = 3;
    next();
    QTimer::singleShot(0, this, SLOT(calculate()));
}

void FindPrimes::calculate()
{
    QTime t;
    t.start();
    while (t.elapsed() < 150) {
        if (cand > max) {
            emit finished(); // end
            return;
        }
        if (curr > sqrt) {
            emit prime(cand); // prime
            next();
        } else if (cand % curr == 0)
            next(); // not prime
        else
            curr += 2; // check next divisor
    }
    QTimer::singleShot(0, this, SLOT(calculate()));
}
```

The `FindPrimes` class makes use of two features—it holds its current calculation state (`cand` and `curr` variables) so that it can continue calculations where it left off, and it monitors (by the use of `QTime::elapsed()`) how long it has been performing the current step of the task. If the time exceeds a predefined amount, it returns to the event loop but, before doing so, it starts a single-shot timer that will call the method again (you might call this approach “deferred recurrency”).

I mentioned there were two possibilities of doing a task in steps. The second one is to use `QMetaObject::invokeMethod()` instead of timers. This method allows you to call any slot from any object. One thing that needs to be said is that, for this to work in our case, we need to make sure the call is made using the `Qt::QueuedConnection` connection type, so that the slot is called in an asynchronous way (by default, slot calls within a single thread are synchronous). Therefore we might substitute the timer call with the following:

```
QMetaObject::invokeMethod(this, "calculate",
                           Qt::QueuedConnection);
```

The advantage of this over using timers is that you can pass arguments to the slot (for example, passing it the current state of a calculation). Apart from that the two methods are equivalent.

Parallel Programming

Finally, there is the situation where you have to perform a similar operation on a set of data—for instance, creating thumbnails of pictures from a directory. A trivial implementation would look like this:

```
QList<QImage> images = loadImages(directory);
QList<QImage> thumbnails;
foreach (const QImage &image, images) {
    thumbnails << image.scaled(QSize(300,300),
                               Qt::KeepAspectRatio, Qt::SmoothTransformation);
    QCoreApplication::sendPostedEvents();
}
```

A disadvantage of such an approach is that creating a thumbnail of a single image might take quite long, and for that time the GUI

would still be frozen. A better approach would be to perform the operation in a separate thread:

```
QList<QImage> images = loadImages(directory);
ThumbThread *thread = new ThumbThread;
connect(thread, SIGNAL(finished(QList<QImage>)),
        this, SLOT(showThumbnails(QList<QImage>)));
thread->start(images);
```

This solution is perfectly fine, but it doesn’t take into consideration that computer systems are evolving in a different direction than five or ten years ago—instead of having faster and faster processing units they are being equipped with multiple slower units (multicore or multiprocessor systems) that together offer more computing cycles with lower power consumption and heat emission. Unfortunately, the above algorithm uses only one thread and is thus executed on a single processing unit, which results in slower execution on multicore systems than on single core ones (because a single core in multicore systems is slower than in single core ones).

To overcome this weakness, we must enter the world of parallel programming—we divide the work to be done into as many threads as we have processing units available. Starting with Qt 4.4, there are extensions available to let us do parallel programming: these are provided by `QThreadPool` and *Qt Concurrent*.

The first possible course of action is to use so called *runnables*—simple classes whose instances can be executed by a thread. Qt implements runnables through its `QRunnable` class. You can implement your own runnable based on the interface offered by `QRunnable` and execute it using another entity offered by Qt. I mean a thread pool—an object that can spawn a number of threads that execute arbitrary jobs. If the number of jobs exceeds the number of available threads, jobs will be queued and executed when a thread becomes available.

Let’s go back to our example and implement a runnable that would create an image thumbnail using a thread pool.

```
class ThumbRunnable : public QRunnable {
public:
    ThumbRunnable(...) : QRunnable(), ... {}
    void run(){ m_result = m_image.scaled(...); }
    const QImage &result() const{ return m_result; }
};

QList<ThumbRunnable *> runnables;
foreach(const QImage &image, images){
    ThumbRunnable *r = new ThumbRunnable(image, ...);
    r->setAutoDelete(false);
    QThreadPool::globalInstance()->start(r);
    runnables << r;
}
```

Basically, everything that needs to be done is to implement the `run()` method from `QRunnable`. It is done the same way as subclassing `QThread`, the only difference is that the job is not tied to a thread it creates and thus can be invoked by any existing thread. After creating an instance of `ThumbRunnable` we make sure it won’t be deleted by the thread pool after job execution is completed. We need to do that because we want to fetch the result from the object. Finally, we ask the thread pool to queue the job using the global thread pool available for each application and add the runnable to a list for future reference.

We then have to periodically check each runnable to see if its result is available, which is boring and troublesome, but fortunately there is a better approach when you need to fetch a result. Qt Concurrent introduces a number of paradigms that can be invoked to perform SIMD (*Single Instruction Multiple Data*) operations. Here we will take a look only at one of them, the simplest one that lets us process each element of a container and have the results ready in another container.

```
typedef QFutureWatcher<QImage> ImageWatcher;
QImage makeThumb(const QString &img)
{
    return QImage(img).scaled(QSize(300,300), ...);
}

QStringList images = imageEntries(directory);
ImageWatcher *watcher = new ImageWatcher(this);
connect(watcher, SIGNAL(progressValueChanged(int)),
        progressBar, SLOT(setValue(int)));
QFuture<QImage> result
    = QtConcurrent::mapped(images, makeThumb);
watcher->setFuture(result);
```

Easy, isn't it? Just a few lines of code and the watcher will inform us of the state of the SIMD program held in the `QFuture` object. It will even let us cancel, pause and resume the program. Here, we used the simplest possible variant of the call—using a standalone function. In a real situation you would use something more sophisticated than a simple function—*Qt Concurrent* lets you use functions, class methods and function objects. Third party solutions allow you to further extend the possibilities by using bound function arguments.

Conclusion

I have shown you a whole spread of solutions based on type and complexity of problem related to performing time consuming operations in Qt-based programs. Remember that these are only the basics—you can build upon them, for example by creating your own “modal” objects using local event loops, swift data processors using parallel programming, or generic job runners using thread pools. For simple cases, there are ways to manually request the application to process pending events, and for more complex ones dividing the task into smaller subtasks could be the right direction.

You can download complete examples that use techniques described in this article from the *Qt Quarterly* Web site. If you would like to discuss your solutions, work together with others to solve a problem that's bothering you, or simply spend some time with other Qt (cute?) developers, you can always reach me and others willing to share their knowledge at <http://www.qtcentre.org>

Never again let your GUI get frozen!

Witold Wysota is one of the administrators of *Qt-Centre* and a PhD student at Warsaw University of Technology where he gives lectures on Qt. When not dealing with Qt he can be found practicing traditional kung-fu or exploring the Universe with his brand new 150/1200 Newton telescope.



Qt Quarterly is published by Nokia Corporation, Oslo, Norway. Copyright © 2008 by Nokia Corporation or original authors.

Authors: Johan Thelin, Witold Wysota, Thomas Zander.

Contributors: Morten Sørvig.

Production: Katherine Barrios.

Editor: David Boddie.

A selection of *Qt Quarterly*'s articles and source code is available from <http://doc.trolltech.com/qq/>. Potential contributors should contact the editors at qq@trolltech.com

No part of this newsletter may be reproduced, in any form or by any means, without permission in writing from the copyright holder.

Nokia, Qt and their respective logos are trademarks of Nokia Corporation in Finland and/or other countries worldwide. All other products named are trademarks of their respective owners. Produced in Norway.

The Panel Stack Pattern

When reading about Qt for Embedded Linux, many examples seen in the online press are related to media gizmos, mobile phones and other graphics-intensive applications. One common, but less discussed application is in control panels, used for home automation, industrial machinery and alarm systems.

The typical control panel does not feature a windowed interface. It does not run multiple user applications at once. They generally do not feature a full sized keyboard. Instead a touch screen or a very basic set of buttons is used.

In this article, I will present a basic pattern that I've been using when implementing this type of applications. I call it a pattern because each of these systems that I've come across has its own peculiarities and it is hard to make them all in one mould.

The Stack

All these systems are based around full screen dialogs—I choose to call these panels. Most of these panels are fairly static and can thus be created at start-up. To avoid having to create them all at once, the singleton pattern can be used. This postpones the creation of panels until they are first used, but prevents them from being created more than once.

We will wait just a few moments with this code as there is one important building block that forms the basis of the pattern: the panel stack. All panels are kept in a `QStackedWidget`. This stack of panels is kept in a widget that acts as the application's “main window”. As there can be only one main window, this is a singleton class and it is called the `PanelStack`. The panel stack holds all the panels and lets you show each one of them in turn by referring to the index that you get when you add it to the stack.

```
class PanelStack : public QWidget
{
    Q_OBJECT

public:
    static PanelStack *instance();
    int addPanel(AbstractPanel *);
    void showPanel(int);

private:
    ...
};
```

So, each panel is actually an implementation of `AbstractPanel`, and almost all panels are singletons. The idea is that, the first time an instance of a panel is called, it adds itself to the panel stack. Then, each panel provides a method called `showPanel()`. It initializes the panel and tells the panel stack to show it. This means that each panel keeps its own index, and all that you need to know when implementing your application is that you call `showPanel()` in the panel that you want to show.

```
class AbstractPanel : public QWidget
{
protected:
    AbstractPanel(QWidget *parent = 0);

    void addPanelToStack();
    int panelIndex() const;

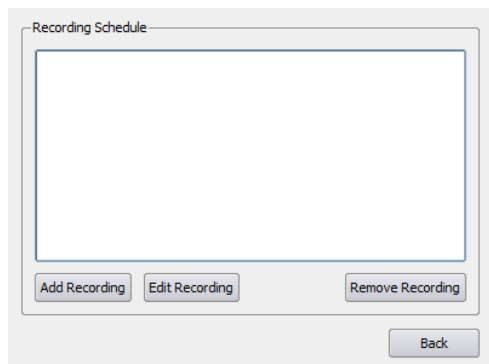
private:
    ...
};
```

The actual singleton implementation is made in each subclass of `AbstractPanel`. That is also where the `showPanel()` method is added. I have left out a virtual `showPanel()` method, as different panels need different arguments when being initialized.

An Example

Let's continue by looking at an example. In this case, a VCR control panel. It consists of four panels: the main menu, the live panel (holding the play, pause and stop buttons), the recordings panel (allowing the user to manage schedules for recording), and finally, the recording details panel (letting the user configure a given schedule for recording).

Each of these panels will be designed using *Qt Designer* and their interface will look very much the same. We will focus on the `RecordingsPanel` widget in this article. The entire example is available for download from the *Qt Quarterly* Web site.



Before we continue, a short word about keeping data away from the user interface code. We use a recordings class in parallel to the panel classes shown here and the panel stack class. This allows all panels to access the data independently of each other.

In this trivial example, we use a `QStringList` to keep all the recordings, but in a real world application this would most probably be a singleton, possibly coupled with a `QAbstractModel` interface. We start by declaring the class as a singleton and providing the `showPanel()` method. In this case, the panel always shows all recordings, so there is no need for arguments.

```
class RecordingsPanel : public AbstractPanel
{
public:
    static RecordingsPanel *instance();
    void showPanel();
    ...
private:
    RecordingsPanel();
    static RecordingsPanel *m_instance;
    ...
};
```

The implementation follows. Notice that the implementation method adds the panel to the stack. This is to allow the `addPanel()` method to use methods, signals and slots of the actual class being implemented and not the base class. If the `addPanelToStack()` call had been made from the `AbstractPanel`'s constructor, the virtual methods, signals and slots would have been those of the `AbstractPanel` and not of the `RecordingsPanel`.

```
RecordingsPanel *RecordingsPanel::m_instance = 0;
RecordingsPanel *RecordingsPanel::instance()
{
    if (!m_instance)
    {
        m_instance = new RecordingsPanel();
        m_instance->addPanelToStack();
    }
    return m_instance;
}

void RecordingsPanel::showPanel()
{
    PanelStack::instance()->showPanel(panelIndex());
}
```

The singleton pattern, as shown above, is used for all panels. Now looking at the panel itself (in the image opposite) you can see that this panel consists of a list of recordings and four buttons. Each of these buttons has a corresponding slot.

```
class RecordingsPanel : public AbstractPanel
{
public:
    static RecordingsPanel *instance();
    void showPanel();

private slots:
    void addClicked();
    void editClicked();
    void removeClicked();
    void backClicked();
};
```

The slots leading to navigation between panels are `add()`, `edit()` and `back()`. The **Remove Recording** button simply removes a recording from the list, but keeps showing the list.

The **Back** button asks the main menu to show itself using the `showPanel()` method, while the **Add Recording** and **Edit Recording** buttons pass on the recording's index to the recording details panel. For new recordings, an index of -1 is passed to indicate that a new recording is to be created.

```
void RecordingsPanel::addClicked()
{
    RecordingDetailsPanel::instance()->showPanel(-1);
}

void RecordingsPanel::editClicked()
{
    RecordingDetailsPanel::instance()->showPanel(
        ui.recordingsList->currentRow());
}

void RecordingsPanel::backClicked()
{
    MainMenu::instance()->showPanel();
}
```

In the `RecordingDetailsPanel` class, the `showPanel()` method looks at then initializes the user interface accordingly.

```
void RecordingDetailsPanel::showPanel(int recordIndex)
{
    m_currentIndex = recordIndex;
    if (m_currentIndex == -1)
        ui.lineEdit->setText("");
    else
        ui.lineEdit->setText(recordings[m_currentIndex]);
    PanelStack::instance()->showPanel(panelIndex());
}
```

The implementation of the panel stack to handle this case is fairly trivial. The widget more or less wraps a `QStackedWidget`. When adding a panel to the `PanelStack`, it adds the panel to its `QStackedWidget` and returns the relevant index.

```
int PanelStack::addPanel(AbstractPanel *panel)
{
    return m_panelStack->addWidget(panel);
}
```

In a similar fashion, the `showPanel()` method simply changes the currently shown widget in the stacked widget.

```
void PanelStack::showPanel(int index)
{
    m_panelStack->setCurrentIndex(index);
}
```

Just wrapping a widget in another widget is not a good way to move forward. However, if we add some functionality to the panel stack we can start to make some gains. We'll examine one way to do this in the next section.

Keeping History in the Stack

In the previous example, we built a set of panels that are navigated according to a static pattern. Clicking **Back** when in the recordings panel always leads to the main menu, and so on. This is all very well in most small systems, but it means that we cannot reuse panels in different places in the navigation tree.

To avoid this problem, we can keep the navigation history in the panel stack. Each time a panel is shown, its index is added to a history stack. Instead of moving to the parent when leaving, a `back()` method is added to the panel stack. This moves one step back in the history stack until the first panel is encountered. In the true spirit of the Web browser metaphor, a `home()` method is also added. It takes us all the way to the first panel in one simple call.

Panel reuse is one of the benefits of using this pattern. Another benefit is that the source code dependencies are reduced: each panel only needs to know of the panel stack and the panels located deeper in the navigation trees. The parent panels are no longer important. Finally, we make the `back()` and `home()` methods slots, thus removing the need for those slots in the panels.

```
class PanelStack : public QWidget
{
    ...
    int addPanel(AbstractPanel *);
    void showPanel(int);

public slots:
    void back();
    void home();

private:
    ...
};
```

The changes that we need to make involve creating the `m_history` stack and pushing indexes onto it when `showPanel()` is called. When `back()` is called, we pop one index off, and when `home()` is called we pop all but one index off it.

```
void PanelStack::showPanel(int index)
{
    m_history.push(index);
    showTopPanel();
}

void PanelStack::back()
{
    if (m_history.count() <= 1)
        return; // Cannot go back past the first panel.

    m_history.pop();
    showTopPanel();
}

void PanelStack::home()
{
    if (m_history.count() <= 1) // Either already home
        return; // or there are no panels to return to.

    while (m_history.count() > 1)
        m_history.pop();

    showTopPanel();
}
```

This design forces a rule on the user—the first panel shown is the menu and you cannot pop the history beyond that. As you can see, the `home()`, `back()` and `showPanel()` methods do not update the widget stack themselves. Instead they call the `showTopPanel()` method. This is a private method that takes care of the details.

```
void PanelStack::showTopPanel()
{
    m_panelStack->setCurrentIndex(m_history.top());
    dynamic_cast<AbstractPanel*>(m_panelStack->widget(
        m_history.top()))->enterPanel();
}
```

It not only shows the right panel, but it calls the `enterPanel()` method of that panel. The `enterPanel()` method is a virtual method added to the `AbstractPanel` class. The default implementation is a dummy one that does nothing.

The idea here is that a panel can appear both as a result of its `showPanel()` method being called and from a panel stack event, resulting from a call to `home()` or `back()`. In the latter case the panel needs to know that it is about to be shown so that it can update its contents. For example, the recordings panel updates its list of recordings here.

Pulling Apart the Stack

We now have a method that we know is called every time a new panel is about to be shown. So, why not use it to update some common parts of the user interface. For instance, if a title bar is a part of every panel, why not place it in the panel stack and add a virtual method called `titleText()` to each abstract panel?

```
class AbstractPanel : public QWidget
{
public:
    virtual void enterPanel() {}
    virtual QString titleText() const = 0;
    ...
};
```

A small change to the panel stack updates the `m_titleLabel` label each time a new panel is changed.

```
void PanelStack::showTopPanel()
{
    m_panelStack->setCurrentIndex( m_history.top() );
    AbstractPanel *panel = dynamic_cast<AbstractPanel*>(
        m_panelStack->widget(m_history.top()));
    panel->enterPanel();
    m_titleLabel->setText(panel->titleText());
}
```

There are numerous other elements that can be added to the panel stack to form a common infrastructure. If nothing else, **Back** and **Home** buttons, and perhaps a couple of shortcut buttons leading to specific forms. The buttons can then be hidden and shown by adding virtual methods such as `hasBack()` and `hasHome()` to the `AbstractPanel` class.

More through the Stack

The stacked approach does not only allow for the transfer of information from the panels to the stack. It can work the other way around, too. This is very helpful when dealing with systems with rudimentary control methods.

For example, implementing a keyboard plugin for a keypad consisting of four buttons placed next to the screen can feel like an overkill solution. Instead, one can intercept the events in the panel stack and, from that stack, pass the events on to the current panel through a set of virtual methods. A custom solution, yes, but as you are probably dealing with custom hardware that is only to be expected.

I have found the panel stack approach useful in numerous customer cases and it always performs well while being flexible and adaptable to application-specific requirements. I hope you find it to be a useful way to think about embedded user interfaces.

Johan Thelin is the author of the Foundations of Qt Development book available from Apress and has a soft spot for embedded systems. He is also involved in the QtCentre and works as a consultant for BitSim AB.

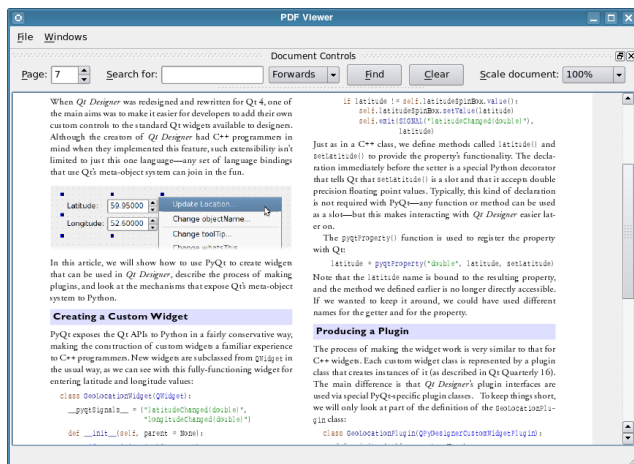


Poppler: Displaying PDF Files with Qt

As we saw earlier in this issue, Qt can be used to generate documents in an ever expanding range of formats that can be viewed and edited with external applications. Qt also comes with facilities to display HTML “out of the box”, and can generate its own print previews, but what about files that originate outside Qt applications?

Fortunately, there are third party libraries available for some of the things that Qt doesn’t provide. One of these is Poppler, a Portable Document Format (PDF) rendering library that forms the basis of a number of widely-used PDF viewing applications. Poppler is a fork of the Xpdf PDF viewer that is licensed under the GNU General Public License. Xpdf can also be obtained under other licensing terms.

Poppler is designed in a way that allows it to be used with any toolkit or framework as long as a suitable rendering backend is available. Qt application developers are fortunate in that there is also a Qt frontend available—a set of Qt-style classes that use Qt classes to describe parts of PDF documents.



In this article, we’ll take a brief look at some of the features provided by Poppler in the context of creating a simple PDF viewing application.

Setting Things Up

Developers using Linux should find that Poppler and the Qt 4 frontend are available as a package for most recent distributions. Developers on Windows, Mac OSX, and other Unix platforms can download source code from the poppler.freedesktop.org Web site.

By default, Poppler is built with all kinds of frontends and backends. If you compile Poppler from source, you can exclude some of these to save compile time. When configuring the build, it may be easier to set the installation prefix to that used for the Qt installation—this prefix is the directory under which subdirectories containing executables, libraries and data files are stored.

It is important to know where the Poppler library and header files will be installed because our example will need them.

Rendering Documents

In our example, we provide a simple user interface to display PDF files, displaying a single page at a time and providing controls to let the user move between pages. Each page is displayed in a custom widget, `DocumentWidget`, held in the main window’s central widget, a scroll area.

The user opens a new file via a file dialog, which we open in response to an action being triggered. The path to the file is

passed to the `DocumentWidget` so that the document it contains can be fed to the Poppler library.

Unlike with many Qt classes, we load a document using a static function in the following way:

```
Poppler::Document *doc = Poppler::Document::load(path);
```

If the document returned is not null, we have a document that we can explore. Note that our example takes ownership of the document, so we must remember to dispose of it when we have finished with it.

Each document contains a series of pages that can be obtained one by one using the `Document::page()` function. Although the `Document` class has a collection of functions to control the appearance of the document, actual rendering is performed by each `Page` object. In our example, we render pages into `QImage` objects that we display using the `DocumentWidget`, itself just a simple `QLabel` subclass.

The key part of our `DocumentWidget::showPage()` function looks like this:

```
void DocumentWidget::showPage(int page)
{
    QImage image = doc->page(currentPage)->renderToImage(
        scaleFactor * physicalDpiX(),
        scaleFactor * physicalDpiY());
    ...
    setPixmap(QPixmap::fromImage(image));
}
```

In the above code we pass the resolution of the image to be created, multiplied by a scale factor that the user controls via the example’s user interface. We have to be careful with the range of scale factors available because it is easy to request extremely large images. In practice, we restrict the user’s choice to a set of predefined scale factors.

Searching for Text

One of the many useful features that Poppler provides is the ability to locate specific text strings in PDF documents. Since PDF is designed to store printable rather than editable documents, it is not always easy to easily access and reconstruct the author’s original text. However, Poppler does a good job of locating text in many documents, and we can expose this feature in our example.

The API for locating text provides conventional features such as case-insensitive and directional searching, but also returns information about the position of any located text on the page—since PDF is a display format, this is really the only useful information about the text we can obtain. This information can be used to indicate where any subsequent searches should begin.

Basically, the code to perform a forward search in a given page looks like this:

```
bool found = page->search(text, searchLocation,
    Poppler::Page::NextResult,
    Poppler::Page::CaseInsensitive);
```

Here, `searchLocation` is a `QRectF` object that indicates where the search should start from on the given page. Initially, when we perform a search, we just pass a default constructed `QRectF` object to start from the page origin.

The rectangle we obtain from the `Page::search()` function can be used when we render the page to highlight the located text and scroll the view to make sure it is visible. However, the position and dimensions of the rectangle are given in points (1 inch = 72 points), so we need to transform the rectangle to cover the correct area on-screen.

Searching through a document for a piece of text is slightly more involved than just a single function call. We’ll look at this in more detail later.

Extracting Text

Since the mapping between the author's original text and its location on-screen may be purely visual, it is difficult to automate the extraction of text from PDF files, though there are tools that try very hard to achieve this.

Many document viewers let the user select and export text by making them select a region on-screen, giving the application something to work with, and Poppler supports this approach by providing a function that returns a string for a given rectangle that we call like this:

```
QString text = doc->page(currentPage)->text(selectedRect);
```

The method we use is somewhat different to this. We'll cover it in more detail later.

The Example in More Detail

Having covered the basics of displaying pages, searching, and extracting text from documents, let's take a closer look at how our example uses these features.

We provide two functions to search for text strings supplied by the user via the user interface. For forwards searching, we start by looking for strings on the current page, beginning at the current search location, then try each following page until the end of the document.

```
QRectF DocumentWidget::searchForwards(const QString &text)
{
    int page = currentPage;
    while (page < doc->numPages()) {
        if (doc->page(page)->search(text, searchLocation,
            Poppler::Page::NextResult,
            Poppler::Page::CaseInsensitive)) {
            if (!searchLocation.isNull()) {
                showPage(page + 1);
                return searchLocation;
            }
        }
        page += 1;
        searchLocation = QRectF();
    }
}
```

If we reach the end of the document without finding anything, we search from the beginning until we reach the current page.

```
page = 0;
while (page < currentPage) {
    searchLocation = QRectF();
    if (doc->page(page)->search(text, searchLocation,
        Poppler::Page::NextResult,
        Poppler::Page::CaseInsensitive)) {
        if (!searchLocation.isNull()) {
            showPage(page + 1);
            return searchLocation;
        }
    }
    page += 1;
}
return QRectF();
}
```

As well as rendering pages at different scales, as shown earlier, we would like to highlight the results of searches. To do this, we insert some code to paint on the image obtained from the current page, using a matrix to map the rectangle onto the image.

```
QMatrix DocumentWidget::matrix() const
{
    return QMatrix(scaleFactor * physicalDpiX() / 72.0, 0,
        0, scaleFactor * physicalDpiY() / 72.0,
        0, 0);
}
```

```
void DocumentWidget::showPage(int page)
{
    ...
    QImage image = doc->page(currentPage)->renderToImage(
        scaleFactor * physicalDpiX(),
        scaleFactor * physicalDpiY());
    if (!searchLocation.isEmpty()) {
        QRect highlightRect = matrix().mapRect(
            searchLocation).toRect();
        highlightRect.adjust(-2, -2, 2, 2);
        QImage highlight = image.copy(highlightRect);
        QPainter painter;
        painter.begin(&image);
        painter.fillRect(image.rect(),
            QColor(0, 0, 0, 32));
        painter.drawImage(highlightRect, highlight);
        painter.end();
    }
    setPixmap(QPixmap::fromImage(image));
}
```

The result of this additional effort is shown in the following image—the located text is displayed normally while the rest of the page is slightly darker.

Rendering the OpenGL Scene

Actual OpenGL rendering is done by reimplementing `QGraphicsView`'s `drawBackground()` function. By rendering the OpenGL scene in `drawBackground()`, all widgets and other graphics items are drawn on top. It is possible to reimplement `drawForeground()` to render OpenGL rendering on top of the graphics scene, if that is needed.

Like when subclassing `QGLWidget`, it is not sufficient to set a common state in `initializeGL()` or `resizeGL()`. A painter is created and made active on the GL context when `drawBackground()` is called. The painter changes the GL state when `begin()` is called. For example,

In our example, we allow the user to draw a selection onto the page by reimplementing three of the mouse event handler functions in our `DocumentWidget`. In these we maintain a `QRubberBand` object to keep track of the area selected, following the pattern shown in the `QRubberBand` documentation.

The mouse release event handler is where we start the process of selecting text:

```
void DocumentWidget::mouseReleaseEvent(QMouseEvent *)
{
    ...
    if (!rubberBand->size().isEmpty()) {
        QRectF rect = QRectF(rubberBand->pos(),
            rubberBand->size());
        rect.moveLeft(rect.left() -
            (width() - pixmap()->width()) / 2.0);
        rect.moveTop(rect.top() -
            (height() - pixmap()->height()) / 2.0);
        selectedText(rect);
    }
    rubberBand->hide();
}
```

When the user releases the mouse button, we create a rectangle with coordinates relative to the top-left corner of the image within the label, and we pass this to the `selectedText()` function which is responsible for informing the rest of the application about any text it finds.

As noted earlier, the Poppler `Page` class provides a function to return text within a rectangle in a document. However, in `selectedText()`, we use a more convoluted method to show how much information we can obtain about a document.

We begin by mapping the selection rectangle onto the page, using the inverse of the matrix we used to highlight search results, before obtaining a list of `TextBox` objects, each of which describes a piece of text on the page.

```
void DocumentWidget::selectedText(const QRectF &rect)
{
    QRectF selectedRect = matrix().inverted()
        .mapRect(rect);

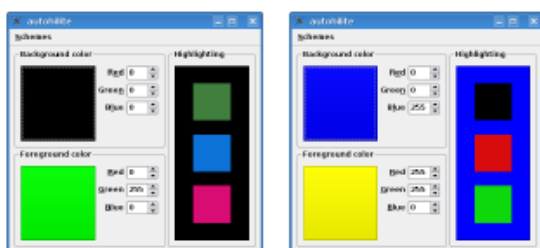
    QString text;
    bool hadSpace = false;
    QPointF center;
    foreach (Poppler::TextBox *box,
        doc->page(currentPage)->textList()) {
        if (selectedRect.intersects(box->boundingBox())) {
            if (hadSpace)
                text += " ";
            if (!text.isEmpty() &&
                box->boundingBox().top() > center.y())
                text += "\n";

            text += box->text();
            hadSpace = box->hasSpaceAfter();
            center = box->boundingBox().center();
        }
    }

    if (!text.isEmpty())
        emit textSelected(text);
}
```

We test whether each piece of text lies within the selection and append it in a `QString` if it does. We also perform some elementary checks to see if we can cleverly insert newline characters in appropriate places.

Note that, while we're satisfied with obtaining whole pieces of text (typically words in a sentence), recent versions of Poppler allow the individual characters in `TextBox` objects to be located.



Clockwise from top-left: White on black, white on dark blue, yellow on blue and green on black.

By playing with the red, green and blue values of each color, you can see how the algorithm generates new combinations of colors. Try to avoid choosing foreground and background colors that are

In the user interface, when the user selects some text, we display it in a text browser so that it can be copied and pasted elsewhere.

Building the Example

The example is provided as a standard Qt project with a simple `pdfviewer.pro` file. Because there is a certain amount of freedom associated with where you can install the Poppler library and header files on your system, you will need to modify this file to use the correct paths.

On Ubuntu 8.04 with the `libpoppler-qt4-dev` package installed, the appropriate paths are as follows:

```
INCLUDEPATH += /usr/include/poppler/qt4
LIBS += -L/usr/lib -lpoppler-qt4
```

Other Linux distributions may install these files in different locations, and developers on other platforms may find it easier to build the library alongside the example instead of installing it.

Other Features and Possible Improvements

Our PDF viewer example only uses the most basic features of the Poppler library. Since many documents use features like encryption, slideshow transitions, tables of contents and annotations, the viewer applications that use Poppler to render documents rely on the library's support for these features.

Poppler includes a number of low level features that are useful for the purpose of analysing PDF files. Access to the list of fonts used in a document and the font data itself can be useful when preparing documents for publication.

Access to the body of text in a document is useful to developers looking to index documents for text mining and subsequent analysis. However, as noted earlier, this might be of limited use for some documents. A good summary of the issues surrounding text extraction can be found on the following page:

<http://www.glyphandcog.com/texttext.html>

Information that is not part of the visible document is also available via the Poppler API. Annotations, scripts (typically written in JavaScript) and the URLs for hyperlinks can all be obtained, though it is up to the application developer to present this information in a meaningful way.

Like Qt's `QPrinter` class, Poppler is also able to write PostScript files, so we could easily add support for file export and conversion. Recent versions also support PDF output, and this opens the door to the use of the library for PDF manipulation. In fact, since the library allows us to examine documents without having to display pages, it is possible to write command line tools to handle documents, and a number of these are supplied with Poppler.

Finding Out More

Poppler is hosted on freedesktop.org, a site dedicated to Free and Open Source desktop projects:

<http://poppler.freedesktop.org/>

Poppler's Qt 4 frontend has its own documentation, which can be obtained via the project's Wiki:

<http://freedesktop.org/wiki/Software/poppler>

Popular PDF viewers which use Poppler include Okular and Evince for the KDE and GNOME desktop environments:

<http://okular.kde.org/>

<http://www.gnome.org/projects/evince/>

The Xpdf application, from which Poppler is derived, can be obtained from the following Web site:

<http://www.foolabs.com/xpdf/>

The source code for the example described in this article can be obtained from the Qt Quarterly Web site.

David Boddie is a Senior Technical Writer at Nokia, Qt Software in Oslo, Norway. He has a long-standing love/hate relationship with document formats, and PDF in particular.



Qt News

Qt Eclipse Integration for Linux Open Sourced

Users of the Eclipse integrated development environment on Linux can now obtain a version of the Qt Eclipse Integration under the GNU General Public License (GPL) versions 2 and 3.

C++ and Java developers can use the integration to create, build, debug and run Qt applications from within the Eclipse IDE. This release now makes it possible for motivated developers to study, modify, improve and redistribute the code that binds Qt, Qt Jambi and Eclipse together.

More details can be found on the Qt Software Web site:

<http://trolltech.com/developer/eclipse-integration>

Qtopia Becomes Qt Extended

At the end of September, Qt Software renamed and relaunched Qtopia with the release of Qt Extended 4.4. This release is more than just a rebranded version of Qtopia, however, as it includes several new features:

- A modular architecture for feature selection.
- An advanced touch-based user interface.
- An IP communications framework based on Telepathy for Instant Messaging and presence.
- A unified inbox for email, SMS, MMS and IM, plus push IMAP e-mail and other messaging enhancements.
- Qt UI Test, a tool for automated system tests of the target device.

More information is available on the Qt Software Web site:

<http://trolltech.com/about/news/qt-extended-4.4-released>

New Web Site for Qt Software

With the transfer of Trolltech to Nokia, the trolltech.com Web site has been overhauled to reflect the updated Qt brand. One of the main aims of the new site is to make it easier for visitors to find resources quickly within a few clicks of the front page.

Most of the resources previously found on the site have been migrated to the new site, and we expect the new blend of old and new material to be the ideal starting point for all kinds of users and developers with an interest in Qt.

Qt Quarterly Goes Digital

Qt Quarterly 27 is the first issue of the newsletter to be distributed purely in a digital format. Commercial licensees will receive *Qt Quarterly* in PDF format, and HTML versions of articles from each issue will be made available to the general public on the *Qt Quarterly* Web site at around the time of publication of the following issue.

Printed copies of back issues continue to be available to customers who missed them the first time round, though there are now limited copies of early issues. Customers should already have access to a range of back issues in PDF format via their support and download areas on trolltech.com.

Qt Centre Foundation Established

Just before the summer the Qt Centre Foundation was established in Warsaw, Poland, to formalize the organization behind the largest online community site for Qt. The founding board consists of Jacek Piotrowski (president) and Witold Wysota (vice-president), and is supported by a council comprising of Axel Jaeger, Daniel Kish and Johan Thelin. The foundation works to promote the use of Qt and open source software through Internet activities, competitions and other initiatives helping the creation of Qt software.

The foundation welcomes donations and other assistance that can allow them to continue building and helping the Qt community. For more information, visit <http://foundation.qtcentre.org> or contact foundation@qtcentre.org

Coin3D 3.0.0 Released

Trolltech partner Kongsberg SIM has released Coin3D 3.0.0! Coin3D is a high-level, retained-mode toolkit for effective 3D graphics development. Coin3D and Qt together present the easiest way to create powerful, cross-platform 3D applications.

Using SoQt or Quarter (a set of GUI bindings included in Coin3D) you can display 3D graphics from Coin (the core 3D rendering library in Coin3D) as a `QWidget`.

Highlighted features and improvements in V3.0.0 are

1. Performance Profiling Kit
2. Z-buffer control
3. Improved support for shaders
4. New shadow node
5. Support for ScXML

The Performance Profiling Kit is intended for use during application development to identify performance bottlenecks in Coin-based applications.

For more information about Coin3D, including a Coin3D and Qt tutorial, visit www.coin3d.org.

KOffice Moves Forward with ODF Support

A project aimed at improving the support for OpenDocument Format (ODF) files in KOffice, the office suite for KDE, has released the source code used to automate the tests that check correct handling of files in this format.

Former Troll, Girish Ramakrishnan, and his colleagues at ForwardBias Technologies have been working towards ODF reading and writing support in the KWord word processor (itself based on Qt's *Scribe* text engine) by analysing existing tests, verifying KWord's behavior and automating the process for each test. The work is described on the ForwardBias Technologies weblog:

<http://blog.forwardbias.in/2008/05/kword-odf-support.html>

<http://blog.forwardbias.in/2008/10/kword-odf-lists.html>

The *ODF-Koffice* project is sponsored by the NLnet Foundation, an organisation with a focus on contribution to an open information society. More information about the project can be found on the following page:

<http://www.nlnet.nl/project/odf-koffice/>