GUI Development with Qt

Multi-Platform Graphical

User-Interface Development

with Qt

Prof. Dr. Stefan Röttger, Stefan.Roettger@th-nuernberg.de

Δ

Q What is Multi-Platform GUI Development?
Program Development on Linux, Mac and Windows

Related Topics
→ Consistent Software Management
→ Transparent Software Development
→ Minimization of Platform-Dependencies
→ Abstraction of Native Platforms and Libraries

What is ?

Qt is a platform-independent graphical user interface

Covered Topics:

Multi-Platform Make (CMake)
Multi-Platform Versioning

Binary Version Search
QMake und moc
Main Window and Menus
Qt Event Loop
Basic Qt UI Elements
Signal-Slot Concept
Key end Mouse Events
Widgets and Layouts
Internationalization (i18n)
Consistent Preferences
Drag and Drop
Threads
Inter-Thread Communication
Graphics with QPainter and QGL

Hands-On Project:

- **Project Info**
- **Project Ideas**

Lessons:

- **Lesson 1: Getting Started**
  - **Qt**
  - **Licensing**
  - **Popularity**
  - **Competitors**
  - **Why Qt**
  - **Practised Platform**
  - **Qt Installation**
  - **Qt Documentation**
- **Lesson 2: A First Qt Example**
  - **Hands-on Qt**
  - **QWidget**
  - **Widget Size**
  - **Compilation with QMake**
  - **Compilation with CMake**
  - **Debugging**
  - **Using SVN**
- **Lesson 3: A Simple Painting Application with QPainter**
  - **QPainter**
  - **Event Loop**
  - **Events**
  - **Specializing QWidget**
  - **Drag and Drop**
  - **Drag and Dropping URLs**
  - **A Painter Application**
- **Lesson 4: An OpenGL Application with QGL**
  - **QGL**
  - **OpenGL**
  - **QGL Example**
- **Lesson 5: A Qt Program with a GUI**
  - **GUI Concept**
  - **Widgets and Layouts**
  - **Layout Policies**
  - **GUI Elements**
    - **QLabel**

1 Project Info

Accompanying the lecture, the learned lessons are excercised in a hands-on programming project. The aim of the project is to get in depth knowledge of the following topics:

- Version control
- Using the CMake build system
- Using the Qt moc pre-compiler
- Using basic Qt objects and classes
- Building a sample application
- Employ **Qt threading** concepts

You define the theme of your project. It can be anything you find interesting, as long as the above topics are covered, in particular that the project employs Qt's threading concept.

Upon successful completion and presention of the project you receive a grade that serves as course assessment.

A few examples and possible project ideas are listed on the following **page...**

2 Project Ideas

The main idea to start a qt project is **not** to start from scratch. The idea is to choose a **well stablished middle-ware** (that is a well known open-source library) and utilize it by writing a threaded **qt user interface** for a particular use case.

Here are some project ideas and the corresponding open-source libraries to be utilized:

Theme	**Use Case**	**Library**
Fractals	*Mandelbrot set* [**1**]	std::complex
Number crunching	Computing Pi	GMP = Gnu Multiple Precision Library
Security	Parallelized password cracking	PCL = Password Cracking Library
Photo-realism	Ray-tracing	POV-Ray
Pattern recognition	Kinect	OpenCV / PCL = Point Cloud Library
Networking	File transfer	Open-SSH
Remote Sensing	NDVI	GDAL / libGrid
...		

3 Lesson 1: Getting Started

3.1 Qt

Qt is an

- object-oriented User-interface written in C++
- it has dual licensing terms (commercial and open-source)
- long tradition (originated at Trolltech in Norway, then Nokia, now Digia)
- it is available for
  - Linux
    - Ubuntu Debian MINT OpenSuSe RedHat
    - ARM (Raspberry Pi → Rasbian)
  - Mac
  - Windows
  - Android (via Qt Mobile / MeeGo / Tizen)
    - News: *Qt 5.1 alpha supports iOS and Android* [**2**]
- programming paradigm a flavor of the well-known Model-View-Controller concept
- user interface is contructed by grouping widgets
  - via dragndrop UI builder (Qt Designer)
  - via creating new C++ objects
    - base class is QWIdget
  - new objects with new functionality is created by deriving from QWidget for example and overriding its members
- widgets are organized as a tree, where each widget can have sub-widgets and so on

3.2 Licensing

Qt is licensed under a commercial and open source license (GNU Lesser General Public License version 2.1).

Remark: back in time there was only a commercial license for windows, but nowadays all platforms share the same licensing terms.

Commercial licensing

- “The commercial License of Qt is the correct license to use for the development of proprietary and/or commercial software with Qt where you need to safeguard your development investment to secure your competitive advantage.”

Open Source Licensing

- “Alternatively Qt is also licensed under under the GNU General Public License (GPL) (version 3) and the GNU Lesser General Public License (LGPL) (version 2.1). You can use this edition of Qt to create and distribute software with licenses that are compatible with these free software licenses. LGPL and GPL are complex licenses that contain many obligations and restrictions you must abide with. Always consult an experienced lawyer before choosing these licenses for your project.”

You can download Qt under an open-source license from **http://qt-project.org/**

Extensive documentation is avalaible at **http://qt-project.org/doc/qt-4.8/**

3.3 Popularity

The dual licensing terms make it a popular UI choice because:

- there is a OSS license to start with
- later on you can opt for a commercial license with support
- documentation is up to date, complete and well written with lots of tutorials to start with
- well-tested ui for broad variety of platforms.
  - a broad subset of functionality (menus, file-selectors, drag and drop etc.) is supported transparently via Qt classes on all platforms
  - abstraction classes for a broad variety of third-party libraries (OpenGL, MySql, etc.)
  - Qt has been known to just work and be very productive with a clean layout of classes and abstractions
  - extensively tested as basis of the KDE Linux Desktop

3.4 Competitors

Multi-Platform Competitors:

- wxWidgets
  - OSS
  - not as clean
  - not as well documented
  - not nearly as bug-free
- GTK+
  - OSS
  - base of the Gnome Linux Desktop
  - no support for windows

Single-Platform Competitors:

- Cocoa (MacOS X)
  - Objective C is odd
  - Clean and powerful functionality (AppleScript XCode AppleGL)
  - Direct access to MacOS X functionality (iTunes streaming etc.)
  - Only on Mac and very special
  - Almost no standards
  - some free and open development tools like XCode

Windows Competitors

- mfc
  - outdated
- wpf
  - deprecated
- .net
  - C#
    - interpreted code
    - attributes via getters/setters
    - garbage collection
    - managed/unmanaged code interface problems
  - vast number of convenience classes
  - direct access to Microsoft products like DirectX Excel Word etc.

- extremely proprietray
- source not available
- no standards
- alternative: Mono

3.5 Why Qt

Qt is the only
- well-designed
- well-tested
- well-documented
- well-portable
- open-source

UI currently out there.

3.6 Practised Platform

In the following we are restricting ourselves to to Unix platform to excercise the Qt lessons. Since Qt is a platform independent UI the qt exmaples will work on other platforms like Windows and Mac as well, although they might look a bit different. But we are not giving advice and support for the development environment those platforms support.

Therefore, it is highly recommended to use Ubuntu (Debian) as development platform. Other Unix flavors like OpenSuse or MINT will work, too.

You can install Ubuntu on your system by downloading the iso-image installer from **ubuntu.com** and burning the iso-image to cd/dvd. Then boot from this install image. Then follow the instructions of the installer. Installation is reliable and straight forward and usually requires only a few clicks if you go with the standard installation options.

It is assumed that you are familiar with object-oriented programming using C++. If not, please read the **C/C++ programming lecture**.

To get used to the Unix development platform it is recommended to start reading the following HowTos:
- **HowTo Use The Shell**
- **HowTo Use The GCC**

3.7 Qt Installation

Prerequisites

Unix software development with Qt requires the installation of the following tools:
- cmake
- gnu/c++ compiler
- svn
- qt/qmake
- [OpenGL]

Installation of GCC and SVN

The GCC (Gnu Compiler Collection including the gcc and g++ compilers) is

usually already installed with Ubuntu (and Mac). To test it, open the unix terminal and type “gcc —version”. On my Mac this gives the following output:

```
i686-apple-darwin9-gcc-4.0.1 (GCC) 4.0.1 (Apple Inc. build 5465)
Copyright (C) 2005 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

To test svn, we tell it to give us some help about the available subversion commands:

```
svn help
```

A list of most common svn subcommands:

```
   add
   checkout (co)
   commit (ci)
   copy (cp)
   delete (del, remove, rm)
   diff (di)
   log
   revert
   status (stat, st)
   update (up)
```

Also see the *SVN HowTo* [**3**] to learn more about Subversion/SVN.

Installation of CMake

Open the Ubuntu software manager, search for the package “cmake” and select it and all its dependencies for installation. Then click the install button.

Installation of OpenGL

The installation of OpenGL (and GLUT) is vendor specific: On MacOS X it is already installed with the XCode development package, on Linux it comes with the “mesa-dev”, “X11-dev” and “free-glut3-dev” development packages whereas on Windows it is usually installed with the MSVC IDE.

Installation of Qt4

On MacOS X and Windows, it is recommended to build and install Qt from source!

On Linux, it is mostly sufficient to **install a recent Qt binary package** using the Ubuntu software manager, for example. The list of available Ubuntu packages is available *online* [**4**].

On the command line installation is even simpler:

```
sudo apt-get install qt4-dev-tools
```

Installation of Qt4 from Source

If you install Qt from source, for example on a Mac or when there is no binary package available, we grab the source tar ball from:

```
ftp://ftp.qt.nokia.com/qt/source/qt-everywhere-opensource-src-4.7.4.tar.gz
```

Create a working directory for your software projects (e.g. ~/qt-projects) and put the Qt source tar ball (.tar.gz. .tgz) there.

Then use the Terminal (unix shell) to navigate (change directory) to the working directory:

```
cd ~/qt-projects
```

Extract the Qt source tarball:

```
tar zxvf qt-everywhere-opensource-src-4.7.4.tar.gz
```

Navigate into the extracted source

```
cd qt-everywhere-opensource-src-4.7.4
```

Then type on the unix console:

```
./configure -opensource && make
```

After the build process has finished (go get yourself a cup of coffee) you need to install the compiled binaries and libraries on the Unix system via the following unix command:

```
sudo make install
```

You will be asked to enter your root password for installation of Qt. The binaries will usually be installed in /usr/local/Trolltech/... or /usr/local/Qt... depending on your system configuration.

Now we are ready to get our hands on the first Qt example.

3.8 Qt Documentation

- *Qt Class Reference* [**5**]
- *Examples and Tutorials* [**6**]
- *Books* [**7**]

4 Lesson 2: A First Qt Example

4.1 Hands-on Qt

Our first Qt example will be to simply create an application that will open a plain window, that is a plain widget (without any functionality yet).

First we run an application by returning execution control to its event loop:

```
#include <QtGui/QApplication>
#include <QtGui/QWidget>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);

  QWidget main;
  main.show();

  return(app.exec());
}
```

4.2 QWidget

Next we create a custom widget with a red background, by deriving from QWidget and telling its constructor to set a color palette with a single red color for the background.

```
class MyQWidget: public QWidget
{
public:

  //! default ctor
  MyQWidget(QWidget *parent = 0)
    : QWidget(parent)
  {
    QColor background="red";
    setPalette(background);
  }

  //! dtor
  ~MyQWidget()
  {}

};
```

Widgets can contain sub-widgets and so on. Therefore, the collection of all widgets is organized as a tree (or acyclic graph) with references to the parent widgets for each child.

4.3 Widget Size

Widget size is not set directly because there is not one size of one widget on all platforms and for all screen resolutions.

Instead, we can provide methods that return the preferred, minimum and maximum sizes. Those methods are questioned by the parent widget when choosing an appropriate layout for the widget for a particular available container size.

```
//! return preferred minimum window size
QSize minimumSizeHint() const
{
  return(QSize(100, 100));
}

//! return preferred window size
QSize sizeHint() const
{
  return(QSize(512, 512));
}
```

4.4 Compilation with QMake

Supposed we have a single cpp module named “main.cpp” that contains our Qt exmaple, we compile it using the QMake platform independent build system with the following QMake project description file:

```
TARGET = main
TEMPLATE = app

QT += core gui

SOURCES += main.cpp
```

QMake takes the project description file (e.g. “main.pro”) and transforms it into a project representation that is suitable for the respective platform:

- On Unix it will produce a “Makefile” to be compiled with “make” on the command line.
- On Mac it will produce a XCode project description (.xcode)to be opened with XCode IDE.
- On Windows it will produce a MSCV project solution (.sln) to be opened with the M$VisualC++IDE$.

On Unix we type the following command to compile:

```
qmake && make
```

On Mac we type the following command to compile:

```
qmake -spec macx-g++ && make
```

4.5 Compilation with CMake

QMake is simple to start with, but offers little control over the build configuration and installation paths. In this respect the multi-platform build tool CMake is superior.

To compile our Qt examples with CMake we create a meta project description named “CMakeLists.txt”. This project description is transformed into Makefiles, XCode projects or Windows solutions similar to QMake.

```
# cmake build file

PROJECT(MyQtApp)

CMAKE_MINIMUM_REQUIRED(VERSION 2.8.3)

# non-standard path to Qt4
SET(CMAKE_PREFIX_PATH ${CMAKE_PREFIX_PATH};
    /usr/local/Trolltech/Qt-4.7.4;
   )

# Qt4 dependency
FIND_PACKAGE(Qt4 COMPONENTS QtCore QtGui REQUIRED)
INCLUDE(${QT_USE_FILE})
ADD_DEFINITIONS(${QT_DEFINITIONS})

# executable
ADD_EXECUTABLE(main main.cpp)
TARGET_LINK_LIBRARIES(main
   ${QT_LIBRARIES}
   )
```

On Unix and Mac we type the following command to compile:

```
cmake . && make
```

To change the configuration (e.g. paths, compiler settings etc.) we type

```
ccmake .
```

to open the interactive cmake configuration tool.

Note: On Windows those tools are integrated into the CMake GUI application.

4.6 Debugging

With a QMake project we can use the QtCreater IDE to manage the projects, edit the source code and UI or add break points.

With CMake we have 2 options to debug Qt code:

- Import the CMake list into the KDevelop IDE and set break points there.
- Configure the CMake project manually to create debug code instead of release code and use the GNU command line debugger to debug it:

```
cmake -DCMAKE_BUILD_TYPE=Debug .
make
gdb myqtapp
```

Alternatively, we can use the ccmake curses configuration tool to set the CMAKE_BUILD_TYPE or other compiler flags manually:

```
ccmake .
make
gdb myqtapp
```

On the gdb command line we can set break points:

```
break main
```

and run the program until the break point is hit:

```
run
```

or list the call stack (e.g. after a crash):

```
where
```

or print variables:

```
p var
```

or simply step over the next line of code:

```
n
```

or simply step into the next command or function:

```
s
```

4.7 Using SVN

The shown Qt examples are available for checkout in a Subversion repository on the following SVN server:

svn://schorsch.efi.fh-nuernberg.de

There is also a *web front end* [**8**] for browsing the repository at:

schorsch.efi.fh-nuernberg.de

To check out the first Qt example we use subversion from the command line:

```
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-01
```

Then compile with CMake and execute:

```
cd example-01
cmake . && make && ./main
```

5 Lesson 3: A Simple Painting Application with QPainter

5.1 QPainter

QPainter is an imperative style drawing backend.

QPainter Class Reference [**9**]

Method excerpt:

- drawEllipse, drawLine, drawPoint, drawPolygon, drawPolyLine, drawRect (fillRect)
- drawPixmap, drawImage, drawPicture
- drawText
  - setFont
- setWorldTransform - see also *Qt Coordinate Systems* [**10**]
  - rotate, scale, translate
- setOpacity(float), setPen(QColor), setBrush(QBrush)
  - setCompositionMode

Geometric vector and attribute specificiations are done via **QPoint**, **QLine**, **QRect** and **QColor** classes. Some examples:

```
QPoint(10,10) QPointF(1.5,9.5)
QRect(0,0,width()-1,height()-1)
QColor(r,g,b,a) QColor("red")
```

The **QPixmap** class is an off-screen image representation that can be used as a paint device.

```
QPixmap pix(100,100);
QPainter p(&pix);
p.drawEllipse(50,50,10,10);
```

The **QImage** class is an image representation with pixel (**QColor**) accessors for and IO access. It can dierctly load JPEG or PNG image files (and a few more picture formats).

```
QImage img;
img.load("image.png");
```

A **QPicture** is a recording of QPainter drawing commands. When drawing a QPicture the recorded QPainter commands are replayed. A QPicture is device independent, serializable and storage-efficient.

```
// record
QPicture pic(100,100);
QPainter p;
```

```
p.begin(&pic);
p.drawEllipse(50,50,10,10);
p.end();

// replay
p.begin(&img);
p.drawPicture(0,0,picture);
p.end();
```

As opposed to QPainter's imperative style, Qt 5.0 supports a retained mode style with the introduction of the **Qt Scene Graph** for descriptive definition of drawings. It is utilizing OpenGL ES on the backend side.

5.2 Event Loop

Qt applications are event driven.

This means that we do not act imperatively, but we react on events being delivered.

For this purpose the first action of each Qt application is to return program control to the so called **event loop**. The event loop gathers events of various kinds and dispatches them to the event handlers of those objects that they belong to.

Each Qt object that is a subclass of QObject has an event handler named QObject::event() that receives events of QEvent type.

The event() method does not handle the events itself. Based on the type of event delivered, it calls a specific event handler for that specific type of event, and notifies the caller whether the event was accepted or ignored.

The normal way for an event to be handled is by calling a virtual function. For example, QPaintEvent is handled by calling QWidget::paintEvent(). This virtual function is responsible for reacting appropriately, normally by repainting the widget. If you do not perform all the necessary work in your implementation of the virtual function, you may need to call the base class's implementation.

For example, the following code handles left mouse button clicks on a custom subclassed QCheckBox widget while passing all other button clicks to the base class:

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
  if (event->button() == Qt::LeftButton)
    // handle left mouse button here
  else
    // pass on other buttons to base class
    QCheckBox::mousePressEvent(event);
}
```

5.3 Events

Some events, such as mouse and key event, come from the window system; some, such as timer event, come from other sources; some come from the application itself.

Basic Event Types

- paint event (QPaintEvent → QObject::paintEvent())
- resize event (QResizeEvent → QObject::resizeEvent())
- mouse event (QMouseEvent → QObject::mouseEvent())
- key event (QKeyEvent → QObject::keyEvent())
- wheel events (QWheelEvent → QObject::wheelEvent())
- timer event (QTimerEvent → QObject::timerEvent())

Each event subclasses QEvent and adds event-specific functions. For example, QResizeEvent adds size() and oldSize() to enable widgets to discover how their dimensions have been changed.

Timer Events

Timer events can be created on any QObject simply by calling its startTimer(ms) method with the number of milliseconds after which the timer event is scheduled to be repeated.

The startTimer() method returns a timer id which can be passed to stopTimer() to stop the timer from firing repeatedly.

Event Filters

Sometimes an object needs to look at, and possibly intercept, the events that are delivered to another object. For example, dialogs commonly want to filter key presses for some widgets.

The QObject::installEventFilter() function enables this by setting up an event filter, causing a nominated QObject filter object to receive the events for a target object in its eventFilter() function. This filter functions gets to see the events before the target object processes them. If the inspected event should not be processd further, because it is handled by the target, the event filter function returns true.

Sending Events

Many applications require to send their own events. Custom events are created by subclassing from QEvent. Sending those events can be done in exactly the same way as the Qt main event loop sends events.

Events can be sent either immediately via QCoreApplication::sendEvent() or stored in a queue for later execution via QCoreApplication::postEvent().

Sending customized events may require a modification of the event handler to check and dispatch the cutom event type.

Events and Threads

Qt methods that modify UI elements need to be called from the same thread as the Qt main event loop. It can’t be done from another thread.

Calling methods on objects in another thread can be achieved by sending custom events to the thread of the target object. Sending events is thread-safe. Events can be sent to all threads that have an event loop running.

QObject is not thread-safe though. If you call a QObject method from multiple threads you have to put a mutex around the code that accesses shared data.

Another solution is to use **Queued Connections** to call methods in other threads.

5.4 Specializing QWidget

In the following we look at a Qt example application that handles paint events to paint a text on the main widget.

First we specialize QWidget by overriding its paintEvent() method.

```
class MyQPainterWidget: public QWidget
{
public:

  //! default ctor
  MyQPainterWidget(QWidget *parent = 0)
    : QWidget(parent)
  {}

  //! dtor
  ~MyQPainterWidget()
  {}

protected:

  //! reimplemented paint event
  void paintEvent(QPaintEvent *)
  {
    // paint events handled here
  }

};
```

Then we reimplement the paintEvent() method by drawing a centered text on the widget's canvas by using the **QPainter** class:

```
#include <QPainter>

//! reimplemented paint event
void paintEvent(QPaintEvent *)
{
  QPainter painter(this);

  painter.setPen(Qt::green);
  painter.setFont(QFont("Arial", 100));
  painter.drawText(rect(), Qt::AlignCenter, "Qt");
}
```

Access the example via WebSVN:
QPainter Example [**11**]

Checkout the Qt example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-02

5.5 Drag and Drop

Qt's Drag and Drop concept is also realized with events.

There are events for a dragged item
- entering a widgets area
- moving in a widgets area
- leaving a widgets area
- dropped within a widgets area

There are according virtual methods that are dispatched when the above events are delivered:

```
protected:
  void dragEnterEvent(QDragEnterEvent *event);
  void dragMoveEvent(QDragMoveEvent *event);
  void dragLeaveEvent(QDragLeaveEvent *event);

public:
  void dropEvent(QDropEvent *event);
```

For that to work, we need to enable drag and drop support in the widget's constructor:

```
setAcceptDrops(true);
```

In case a widget wants to accept a particular item of a specific type being dragged into its area we need to notify this in the respective event handler:

```
void ViewerWindow::dragEnterEvent(QDragEnterEvent *event)
{
  event->acceptProposedAction();
}
```

The same applies for the move and drop events.

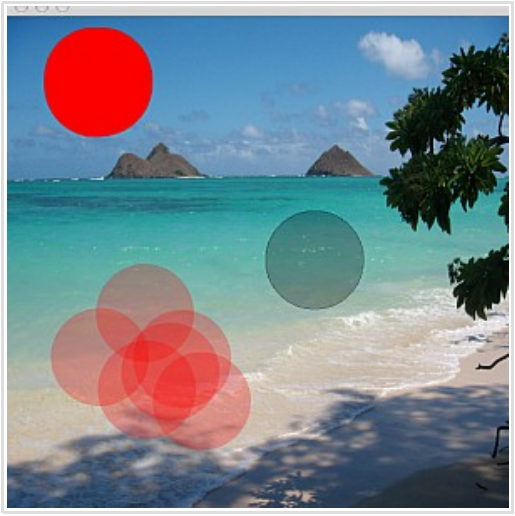
5.6 Drag and Dropping URLs

(:Drag and Dropping URLs:)

If we want to accept only a specific type of items, for example URLs resp. files being dropped into the widget, we check the item type in the drop event and accept the drop if it matches:

```
void ViewerWindow::dropEvent(QDropEvent *event)
{
  const QMimeData *mimeData = event->mimeData();

  if (mimeData->hasUrls())
  {
    event->acceptProposedAction();

    QList<QUrl> urlList = mimeData->urls();

    for (int i=0; i<urlList.size(); i++)
    {
      QUrl qurl = urlList.at(i);

      // run appropriate action for each dropped url here
    }
  }
}
```

5.7 A Painter Application



```
#include <iostream>

#include "painter.h"

const double MyQPainterWidget::fps=30.0; // animated frames per
second

MyQPainterWidget::MyQPainterWidget(QWidget *parent)
  : QWidget(parent)
{
  // init background
  pix=NULL;

  // init brush parameters
  brushSize=50;
  mousePos=QPoint(-brushSize,-brushSize);

  // accept drag and drop
  setAcceptDrops(true);

  // start timer for periodic repainting
  startTimer((int)(1000.0/fps)); // ms=1000/fps
}

MyQPainterWidget::~MyQPainterWidget()
{
  if (pix!=NULL)
    delete pix;
}

QSize MyQPainterWidget::minimumSizeHint() const
{
  return(QSize(100, 100));
}

QSize MyQPainterWidget::sizeHint() const
{
  return(QSize(512, 512));
}
```

```
void MyQPainterWidget::setBackground(QPixmap *p)
  {
  if (pix!=NULL)
    delete pix;

  pix=p;


 *pix=pix->scaled(size(),Qt::IgnoreAspectRatio,Qt::SmoothTransformation);

  repaint();
  }

void MyQPainterWidget::paint(QPoint pos, int size, QColor color,
double opacity)
{
  QPainter painter(pix);

  painter.setRenderHint(QPainter::Antialiasing);

  // paint into background pixmap
  painter.setPen(QColor(0,0,0,0)); // border color
  painter.setBrush(color); // fill color
  painter.setOpacity(opacity); // filling opacity
  painter.drawEllipse(pos, size, size);
}

void MyQPainterWidget::brush()
{
  if (buttonDown)
  {
    paint(mousePos, brushSize, QColor(255,0,0), 0.25);
    repaint();
  }
}

void MyQPainterWidget::paintEvent(QPaintEvent *)
{
  // create paintable pixmap as window background
  if (pix==NULL)
  {
    pix=new QPixmap(size());
    pix->fill(Qt::white);
  }

  // draw painted pixmap as background
  QPainter painter(this);
  painter.drawPixmap(rect(), *pix, pix->rect());

  // draw brush proxy as foreground
  painter.setOpacity(0.25); // drawing opacity
  painter.setPen(QColor(0,0,0)); // border color
  painter.setBrush(QColor(0,0,0)); // fill color
  painter.drawEllipse(mousePos, brushSize, brushSize);
}

void MyQPainterWidget::resizeEvent(QResizeEvent *event)
```

```
{
  if (pix!=NULL)
    setBackground(new QPixmap(*pix));
}

void MyQPainterWidget::mousePressEvent(QMouseEvent *event)
{
  mousePos = event->pos();

  if (event->buttons() & Qt::LeftButton)
    buttonDown = true;
  else
    event->ignore(); // propagate to parent

  brush();
}

void MyQPainterWidget::mouseReleaseEvent(QMouseEvent *event)
{
  buttonDown = false;
}

void MyQPainterWidget::mouseMoveEvent(QMouseEvent *event)
{
  mousePos = event->pos();

  brush();
}

void MyQPainterWidget::mouseDoubleClickEvent(QMouseEvent
*event)
{
  mousePressEvent(event);
  buttonDown = false;
}

void MyQPainterWidget::keyPressEvent(QKeyEvent *event)
{
  if (event->key() == Qt::Key_Greater)
    brushSize+=10;
  else if (event->key() == Qt::Key_Less)
    if (brushSize>10) brushSize-=10;

  brush();
}

void MyQPainterWidget::keyReleaseEvent(QKeyEvent *event)
{
  event->ignore(); // propagate to parent
}

void MyQPainterWidget::wheelEvent(QWheelEvent *event)
{
  event->accept(); // do not propagate to parent
}

void MyQPainterWidget::timerEvent(QTimerEvent *event)
{
```

```
  mousePos = mapFromGlobal(QCursor::pos());

  repaint();
}

void MyQPainterWidget::dragEnterEvent(QDragEnterEvent *event)
{
  event->acceptProposedAction();
}

void MyQPainterWidget::dragMoveEvent(QDragMoveEvent *event)
{
  event->acceptProposedAction();
}

void MyQPainterWidget::dropEvent(QDropEvent *event)
{
  const QMimeData *mimeData = event->mimeData();

  if (mimeData->hasUrls())
  {
    event->acceptProposedAction();

    QList<QUrl> urlList = mimeData->urls();

    QUrl qurl = urlList.at(0);
    QString url = qurl.toString();

    QImage img;

    if (url.startsWith("file://"))
        url=url.remove("file://");

    if (img.load(url))
    {
      QPixmap pixmap;
      pixmap.convertFromImage(img);
      setBackground(new QPixmap(pixmap));
    }
  }
}

void MyQPainterWidget::dragLeaveEvent(QDragLeaveEvent *event)
{
  event->accept();
}
```

Access the example via WebSVN:
Painter Example [**12**]

Checkout the Qt example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-06

6 Lesson 4: An OpenGL Application with QGL

6.1 QGL

QGL is a Qt wrapper for the OpenGL API.

OpenGL is a API for rendering 3D virtual scenes. It utilizes the graphics hardware of a particular platform for high performance rendering. On mobile devices a subset of OpenGL is utilized: OpenGL ES (OpenGL for Embedded Systems).

The main use case for OpenGL in a Qt application is the have a OpenGL rendering context as the 3D rendering area. The QGLWidget class establishes such a context attached to the widget canvas.

So a QGLWidget is a 3D rendering window.

First, we need to check if OpenGL and a proper grafics adaptor is avalable on our platform, so that we can use QGLWidget. Otherwise the application will likely crash.

```
#include <QtOpenGL/qgl.h>

if (!QGLFormat::hasOpenGL())
{
  QMessageBox::warning(0, "NO OPENGL",
                       "Open GL is not available. Program abort.",
                       QMessageBox::Ok);

  exit(1);
}
```

Note: If OpenGL is available, it is also utilized as the rendering backend of QPainter.

6.2 OpenGL

Die Computergrafik-Pipeline (Rendering Pipeline)

- **Die Grafikpipeline**
  - **Aufgaben der Grafikpipeline**
  - **Modellierung von Objekten**
  - **Pipeline Begriffe**
  - **Modellierung der Oberflächen**
  - **Stufen der Pipeline**
  - **Vertex Transformationen**
  - **Beleuchtung**
  - **Rasterisierung**
- **Fixed Function Pipeline**
  - **Programmable Pipeline 1**

    - **Programmable Pipeline 2**
    - **Programmable Pipeline 3**
- **Konkret: Die Game-Engine von AquaNox**

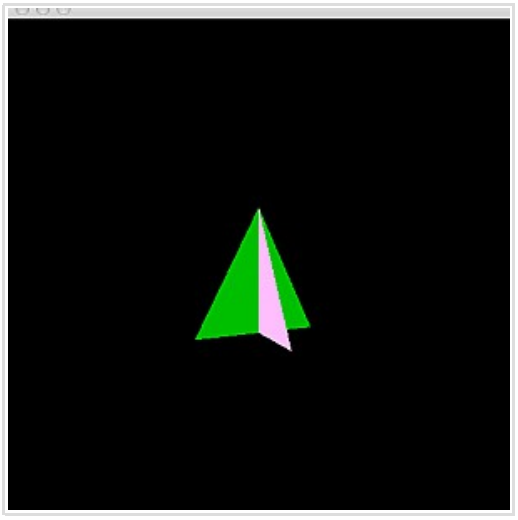
3D Darstellung

- **Matrix Transformationen**
- **Lokale Koordinaten**
- **Viewkoordinaten**
    - **Viewmatrix**
    - **View Dreibein**
- **View Frustum**
    - **View Kamera**
- **Augenkoordinaten**
    - **Projektionsmatrix**
- **MVP Matrix**
- **Viewport Matrix**
- **Z-Puffer**
    - **Normalisierte Projektion**
    - **Nichtlineare Z-Abbildung**
    - **Z-Buffer Fighting**
- **Frustum Clipping**
    - **Line Clipping**
- **Normalen**

Die OpenGL Pipeline

- **GL Vertices**
    - **GL Manpages**
    - **GL Attribute**
- **GL Primitive**
    - **GL Primitiv Beispiel**
- **GL Matrizen**
    - **GL Matrix Mode**
    - **GL Matrix Manipulation**
    - **GL Perspektive**
    - **GL Transformationen**
    - **GL Transformations Beispiel**
- **GL Kamera**
- **GL Matrix Stack**
    - **GL Objekt Hierachie**
    - **GL Matrix Stack Beispiel**
- **GL Triangle Strips**
    - **Batching**
    - **GL Vertex Array**
    - **GL Indexed Face Sets**
- **GL Backface Culling**
    - **GL Backface Test**
    - **GL Backface Beispiel**
- **GL Clip Planes**
- **GL Fogging**
- **GL Tesselierung**
    - **GL Tesselierungsbeispiel**
- **GL PolygonMode**

6.3 QGL Example



Instead of overriding paintEvent(), we derive from QGLWidget and override paintGL() and place OpenGL calls in it.

We also override initializeGL() and resizeGL() as shown in the following example:

```
#include <QtGui/QApplication>
#include <QtGui/QWidget>

#include <QtOpenGL/qgl.h>

#include <GL/gl.h>
#include <GL/glu.h>

static const double fps=30.0; // animated frames per second

class MyQGLWidget: public QGLWidget
{
public:

  //! default ctor
  MyQGLWidget(QWidget *parent = 0)
    : QGLWidget(parent)
  {
    setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer |
QGL::StencilBuffer));

    startTimer((int)(1000.0/fps)); // ms=1000/fps
  }

  //! dtor
  ~MyQGLWidget()
  {}

  //! return preferred minimum window size
  QSize minimumSizeHint() const
  {
    return(QSize(100, 100));
```

```
  }

  //! return preferred window size
  QSize sizeHint() const
  {
    return(QSize(512, 512));
  }

protected:

  void initializeGL()
  {
    qglClearColor(Qt::black);
    glEnable(GL_DEPTH_TEST);
    glDisable(GL_CULL_FACE);
  }

  void resizeGL(int, int)
  {
    glViewport(0, 0, width(), height());
  }

  void paintGL()
  {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // ... opengl rendering commands go here ...

    // setup perspective matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(90.0,1.0,0.1,10.0);

    // setup modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // define local rotated coordinate system
    static double angle=0.0; // rotation angle in degrees
    static const double omega=180.0; // rotation speed in degrees/s
    glTranslated(0.0,0.0,-2.0);
    glRotated(angle,0.0,1.0,0.0);

    // render green triangle
    glBegin(GL_TRIANGLES);
      glColor3f(0.0f,0.75f,0.0f);
      glVertex3d(-0.5,-0.5,0.0);
      glVertex3d(0.5,-0.5,0.0);
      glVertex3d(0.0,0.5,0.0);
      glColor3f(1.0f,0.75f,1.0f);
      glVertex3d(0.0,-0.5,-0.5);
      glVertex3d(0.0,-0.5,0.5);
      glVertex3d(0.0,0.5,0.0);
    glEnd();

    // angle delta equals time delta times omega
    double dt=1.0/fps;
    angle+=dt*omega;
```

```
  }

  void timerEvent(QTimerEvent *)
  {
    repaint();
  }

};

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);

  if (!QGLFormat::hasOpenGL()) return(1);

  MyQGLWidget main;
  main.show();

  return(app.exec());
}
```

Access the example via WebSVN:
QGL Example [**13**]

Checkout the Qt example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-03

7 Lesson 5: A Qt Program with a GUI

7.1 GUI Concept

The GUI consists of a set of building blocks, the so called widgets, which are constructed from sub-widgets ans so on, thus forming a tree.

All widgets are derived from QWidget which is derived from the base class of everything QObject. A widget can either be a readily available basic widget type or a custom type that is derived from basic types via subclassing.

Each widget can contain a collection of sub-widgets by means of a so called layout (QLayout).

7.2 Widgets and Layouts

For a widget to contain children the widget needs to own a so called layout (QLayout).

The layout contains a list of widget children that are owned by the layout. The layout also has settings and policies how how to layout the owned widgets within the available canvas size, such as the expanding direction and spacing.

Simple Example:

```
QWidget
|
QLayout
|      \
QWidget QWidget
```

```
QWidget w;

QLayout *l=new QLayout;
l->setExpandingDirection(Qt:horizontal);
l->setSpacing(100);

l->addWidget(new QWidget);
l->addWidget(new QWidget);

w.setLayout(l);

w.show();
```

Additional helpful layout methods:

- l→itemAt(index) returns the indexed widget from the layout’s widget list
- l→getGeometry() l→setGeometry()

7.3 Layout Policies

The particular graphical appearance of a layout is defined by the size policies of the owned widgets, unless the policy of the layout does not specify a different behavior.

Each widget has a horizontal/vertical layout policy (QSizePolicy): The default policy is Preferred/Preferred, which means that the widget can be freely resized, but prefers to be the size sizeHint() returns. Button-like widgets set the size policy to specify that they may stretch horizontally, but are fixed vertically. Additional policies are:

- expanding direction: vertical or horizontal or both
- item alignment : left, right, centered
- spacing between items
- stretch: how big a item can grow relative to others to accomodate the canvans size
- margins : border size around each item
- etc.

Predefined QHBoxLayout and QVBoxLayout for linear alignment:

- QHBoxLayout organizes the contained items horizontally beneath each other.
- QVBoxLayout organizes the contained items vertically below each other.

Additional control via special items added to a layout:

- l→addSpacerItem()
  - add fixed space between two items
- l→addStretch(int stretch)
  - add growable space between two items to accomodate to the available canvas size

Widgets are normally created without any stretch factor set. When they are laid out in a layout the widgets are given a share of space in accordance with their QWidget::sizePolicy() or their minimum size hint whichever is the greater. Stretch factors are used to change how much space widgets are given in proportion to one another.

7.4 GUI Elements

- QLabel
- QCheckBox
- QRadioButton
- QPushButton
- QSlider
- QLineEdit
- QTabWidget
- QSplitter
- QMessageBox
- QFileDialog

7.4.1 QLabel

```
QLabel *label = new QLabel(this);
label->setText("text");
```

7.4.2 QSlider

```
QSlider *createSlider(int minimum, int maximum, int value)
{
  QSlider *slider = new QSlider(Qt::Horizontal);
  slider->setRange(minimum * 16, maximum * 16);
  slider->setSingleStep(16);
  slider->setPageStep((maximum - minimum) / 10 * 16);
  slider->setTickInterval((maximum - minimum) / 10 * 16);
  slider->setTickPosition(QSlider::TicksBelow);
  slider->setValue(value * 16);
  return(slider);
}

QSlider *slider = createSlider(0, 100, value);
connect(slider, SIGNAL(valueChanged(int)), this, SLOT(slide(int)));
```

7.4.3 QCheckbox

```
QVBoxLayout *layout = new QVBoxLayout;

check1 = new QCheckBox(tr("Check 1"));
check2 = new QCheckBox(tr("Check 2"));

connect(check1, SIGNAL(stateChanged(int)), this,
SLOT(changed1(int)));
connect(check2, SIGNAL(stateChanged(int)), this,
SLOT(changed2(int)));

layout->addWidget(check1);
layout->addWidget(check2);

setLayout(layout);
```

7.4.4 QLineEdit

```
QGroupBox *MyQMainWindow::createEdit(QString name, QString
value,
                                     QLineEdit **lineEdit)
{
  QGroupBox *lineEditGroup = new QGroupBox(name);
  QVBoxLayout *lineEditLayout = new QVBoxLayout;
  lineEditGroup->setLayout(lineEditLayout);
  *lineEdit = new QLineEdit(value);
  lineEditLayout->addWidget(*lineEdit);
  return(lineEditGroup);
}

QGroupBox *lineEditGroup = createEdit(tr("Line Edit"), text,
&lineEdit);
```

```
connect(lineEdit,SIGNAL(textChanged(QString)),this,SLOT(changed1(QString)));
```

7.4.5 QFileDialog

```
QStringList browse(QString title,
                   QString path,
                   bool newfile)
{
  QFileDialog* fd = new QFileDialog(this, title);
  if (fd == NULL) MEMERROR();

  if (!newfile) fd->setFileMode(QFileDialog::ExistingFiles);
  else fd->setFileMode(QFileDialog::AnyFile);
  fd->setViewMode(QFileDialog::List);
  if (newfile) fd->setAcceptMode(QFileDialog::AcceptSave);
  fd->setFilter("All Files (*.*);;Images (*.tif *.tiff *.jpg *.png)");

  if (path!="") fd->setDirectory(path);

  QStringList files;

  if (fd->exec() == QDialog::Accepted)
    for (int i=0; i<fd->selectedFiles().size(); i++)
    {
      QString fileName = fd->selectedFiles().at(i);

      if (!fileName.isNull())
        files += fileName;
    }

  delete fd;

  return(files);
}
```

7.4.6 QSplitter

Same as a QHBoxLayout (or QVBoxLayout).

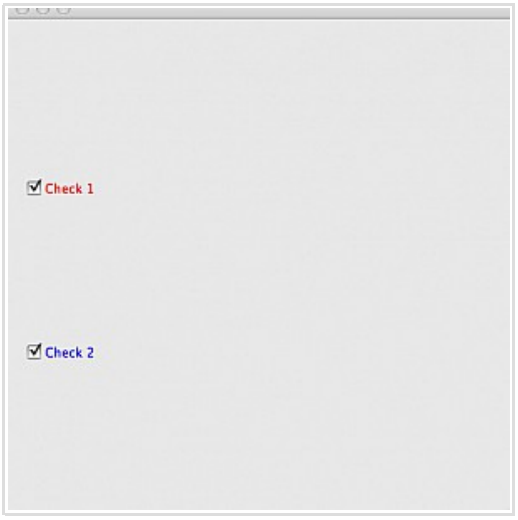
Additionally has handles to resize the area of each widget.

7.4.7 QTabWidget

Same as QHBoxLayout (or QVBoxLayout).

Does not show all contained widgets beneath one another, but shows one selected widget from a tab bar.

7.5 GUI Example



Access the example via WebSVN:
GUI Example [**14**]

Checkout the Qt example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-04

8 Lesson 6: Signals and Slots

8.1 Signals and Slots

Qt follows the concept of “reaction to action”.

More precisely, this means that observed or initiated actions trigger a **signal** that causes a given set of signal receivers, the so called **slots**, to react.

This signal/slot pattern is the usual way of communication between qt objects (usually widgets).

Note: Qt signals are thread-safe.

8.2 Signal and Slot Example

Pattern:

- Method of class A **emits signal**.
- A method of another class B is **registered as a receiver** for the particular signal.
- The latter method is said to be a **slot** that is connected to the signal emitter.
- Then triggering the signal in class A causes the receiver to invoke the corresponding slot in class B.

Setting up a signal/slot connection:

Step 1: Creating a signal emitter

```
class A : public QObject
{
  Q_OBJECT;

public:

  void method()
  {
    emit signal();
  }

signals:

  void signal();

};
```

Step 2: Creating a receiver slot

```
class B : public QObject
{
public slots:

  void slot();
```

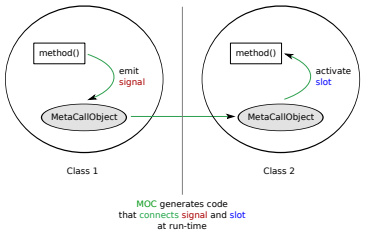
```
};
```

Step 3: Connecting the signal emitter with the signal receiver

```
A *a = new A;
B *b = new B;

connect(a, SIGNAL(signal()), b, SLOT(slot()));
```

8.3 MOC

The MOC (meta object compiler) generates code that places a MetaCallObject in each class through which a signal is routed from the originating class to the receiving class.



- Connection of signals and slots is done at run-time.
- Multiple slots can be connected to a single signal.

C++ Extensions understood and parsed by MOC:

- Connection is implemented via keywords “connect”, SIGNAL, SLOT
- Classes with MetaCallObjects are identified via keyword “Q_OBJECT”
- Slots are identified via “slots” keyword
- Signals are identified via “signals” keyword

8.4 MOC with CMake

Preliminarties:

Any class that emits or receives a signal needs to contain a MetaCallObject by placing the Q_OBJECT keyword as the first token of the class definition.

Then the MOC parser will inject a MetaCallObject into the class definition.

MOC is not a master-piece of software-engineering:

- Q_OBJECT class definitions cannot be header-only.
- Multiple class definitions in one header are not allowed.

CMake has a convenience helper to filter headers thorugh MOC:

```
QT4_WRAP_CPP(OUTFILES *.h)
```

The output files need to be treated as additional modules.

8.5 MOC CMake Example

Example CMakeLists.txt for MOC integration:

```
# cmake build file

PROJECT(MyQtApp)

CMAKE_MINIMUM_REQUIRED(VERSION 2.8.3)

# non-standard path to Qt4
SET(CMAKE_PREFIX_PATH ${CMAKE_PREFIX_PATH};
    /usr/local/Trolltech/Qt-4.7.4;
   )

# Qt4 dependency
FIND_PACKAGE(Qt4 COMPONENTS QtCore QtGui REQUIRED)
INCLUDE(${QT_USE_FILE})
ADD_DEFINITIONS(${QT_DEFINITIONS})

# header list
SET(LIB_HDRS
   module.h
   )

# module list
SET(LIB_SRCS
   module.cpp
   )

# moc
QT4_WRAP_CPP(MOC_OUTFILES ${LIB_HDRS})

# library
SET(LIB_NAME ${PROJECT_NAME})
INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})
ADD_LIBRARY(${LIB_NAME} ${LIB_SRCS} ${LIB_HDRS} ${MOC_OUTFILES})

# executable
ADD_EXECUTABLE(main main.cpp)
TARGET_LINK_LIBRARIES(main
   ${LIB_NAME}
   ${QT_LIBRARIES}
   )
```

8.6 QWidget Slots

Each class derived from QWidget has the following **signals**:

- `void destroyed ( QObject * obj = 0 )`
- `void customContextMenuRequested ( const QPoint & pos )`

E.g. signals of *QCheckBox* [**15**]:

- `void stateChanged ( int state )`
- `4 signals inherited from QAbstractButton (clicked pressed released toggled)`
- `1 signal inherited from QWidget (customContextMenuRequested)`
- `1 signal inherited from QObject (destroyed)`

8.7 QMainWindow

QMainWindow is a widget with a predefined layout, that features a menu bar and other convenience classes.

Since it already has a layout, usage is slightly different from regular windows, menaing that we set the widget's children not by adding a layout (setLayout) but by adding a central widget that consumes the work area of the main window (setCentralWidget).

With a QMainwindow we can easily

- create menu entries and actions
- load and save persistent program settings
- set the main window title

8.7.1 Window Title

```
MyQMainWindow::MyQMainWindow(QWidget *parent)
  : QMainWindow(parent)
{
  initSettings();

  createMenus();
  createWidgets();

  setWindowTitle(tr("Qt Main Window Example"));
}
```

8.7.2 Menus

Adding menu entries to a QMainWindow for the entries "Quit" and "About":

```
QAction *quitAction = new QAction(tr("Q&uit"), this);
quitAction->setShortcuts(QKeySequence::Quit);
quitAction->setStatusTip(tr("Quit the application"));
connect(quitAction, SIGNAL(triggered()), this, SLOT(close()));

QMenu *fileMenu = menuBar()->addMenu(tr("&File"));
fileMenu->addAction(quitAction);

QAction *aboutAction = new QAction(tr("&About"), this);
aboutAction->setShortcut(tr("Ctrl+A"));
aboutAction->setStatusTip(tr("About this program"));
connect(aboutAction, SIGNAL(triggered()), this, SLOT(about()));

QMenu *helpMenu = menuBar()->addMenu(tr("&Help"));
helpMenu->addAction(aboutAction);
```

Selection of the Quit item of the file menu invokes the close() slot.

Selection of the About item of the help menu invokes the about() slot:

```
private slots:

void about()
{
  QMessageBox::about(this, tr("About this program"),
                     tr("just an example"));
}
```

8.7.3 Settings

The persistent settings of an application are accessed with a QSettings object with a unique organization and application name:

```
QSettings settings("www.th-nuernberg.de", "MyApp");
```

Making a user-defined application property persistent, means setting a particular value for a unique property identifier on program exit:

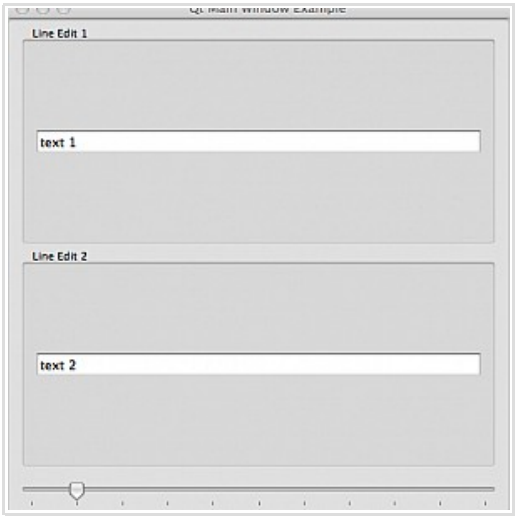
```
QString value="setting";
settings.setValue("id", value);
```

There are overloaded setters for most basic Qt data types like int, double and QString.

The persistent values are retrieved at program start with:

```
if (settings.contains("id"))
  value = settings.value("id").toString();
else
  value = "default";
```

8.8 Layout Example



Access the example via WebSVN:
Layout Example [**16**]

Checkout the Qt example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-05

9 Lesson 7: Threading Concepts

9.1 Blocking and Non-Blocking

When a user clicks at a GUI element and the respective signal triggers an action, the UI thread is blocking while the slot executes. This means that further events in the event queue are not handled until the slot returns.

If the application needs to be running time-consuming computations in a slot, the GUI will not be responsive during that time. To keep the GUI responsive, long running computations need to be non-blocking. This is achieved by executing them concurrently to the UI thread.

9.2 Threads and Processes

A **process** is a program that is being executed in memory with its own memory address range.

Multiple processes can run simultaneously by assigning a time slice to each process (usually a few milliseconds). When a process has run out of time, its state is frozen and execution is passed over to next process waiting to be resumed for another time slice.

Whenever execution is resumed, the virtual address range tables of the MMU and the entire CPU need to be saved, synchronized with the IO state and reloaded. Process switching is therefore costly.

A more lightweight method to execute program code simultaneously is **threading**. A thread shares its virtual address range with other threads of the same process, so that switching from one thread to another just requires the reloading of the CPU registers. Most modern CPUs have multiple copies of its register set to be able to reactivate a set efficiently.

Since threads live in the same process, they share all the resources of that process. If more than a single thread accesses a resource, the **access to the shared resource needs to be synchronized**. Otherwise the **asynchronous access will lead to a so called race condition**! In a race condition the well-defined execution order is violated. As a result the program enters an ill-defined logical state, where all sorts of bad things can happen. Most common effects are a program freeze or a crash.

9.3 Race Condition

Supposed we have a situation where a single resource needs to be shared among multiple users of the resource, but the resource can only be used by one user at a time.

Simple example: Making Tom Kha Gai in a wok:

1. Get wok from its place
2. Put ingredients in wok
3. Cook and stir

4. Clean wok
5. Put wok back at its place

If multiple users want to make Tom Kha Gai they need to synchronize their use of the shared resource wok by waiting until the other person is finished having returned the wok to its place:

```
bool wok_at_place = true;

void tom_kha_gai()
{
  while (!wok_at_place) ; // busy waiting

  wok_at_place = false;

  put_ingredients();
  cook_and_stir();
  clean_wok();

  wok_at_place = true;
}
```

But what is the problem with the race condition then? It happens when claiming the wok!

Supposed, two persons/threads are waiting for the wok. The first one sees the wok at its place and wants to take it. But before it can do so by marking it as being no longer at its place its time slice runs out and the second person/thread is resumed. It also sees the wok at its place because the state has not yet been updated by chance and starts claiming it. Now we have two persons cooking with the same wok at the same time. Not good!

The problem that caused the race condition was that determining if the wok is free and claiming it could be interrupted. This needs to be an atomic non-interruptible operation for the process of cooking Tom Kha Gai to work out well and tasty.

How is that achieved? The operation of requesting and claiming the wok needs to be performed mutually exclusive for all persons/threads. For that purpose we use a so called **mutex**.

Note: I might seem that the chance of being interrupted at the wrong time is very small. Well, let’s have a closer look at the example. It turns out, that each person that has not gained access to the resource is **busy waiting** for it to be returned. Let’s call that the inactive person. If the wok is returned it gets claimed immediately, but the active person/thread will be probably busy preparing the next soup. Now we have the situation that the times slice for the active person is probably not used up, so it will be busy waiting for the wok and claim the wok immediately again. As a consequence, the inactive person will probably never get to see the wok at all if the active person is not cooperative. There is only the slight chance that program execution is interrupted in the very moment the active person is finished. But for that case, the chances are equally high that program execution is interrupted just a few steps later at the very moment of the race condition.

9.4 Threads and Mutexes

In order to cook Tom Kha Gai concurrently, we have to solve two problems:

1) Use a mutually exclusive operation to claim the wok. 2) Establish fair sharing practices.

Both problems are solved with a Mutex. A mutex can be locked and unlocked. If one thread has locked a mutex other threads that want to do the same are put to sleep to be woken up until the lock is released.

```
mutex wok;

void tom_kha_gai()
{
  wok.lock();

  put_ingredients();
  cook_and_stir();
  clean_wok();

  wok.unlock();
}
```

Has that solved the problem with the race condition then?

Yes, because locking and unlocking a mutex is an atomic operation! If there are other persons that want to get the wok, they are put in waiting condition. Persons in waiting condition are invoked when other persons return the lock on the same mutex, so that the latter do not get to claim the wok again.

9.5 QThread and QMutex

With Qt we create a thread by subclassing from QThread. Calling the start() method on a QThread object will execute the run() method in a new concurrent thread. The wait() method waits until the thread is finished by returning from run() (same as join with POSIX threads).

TomKhaGai with QThread:

```
#include <QtCore/QThread>
#include <QtCore/QMutex>

class TomKhaGai: public QThread
{
  public:

  void run()
  {
    wok.lock();

    put_ingredients();
    cook_and_stir();
```

```
    clean_wok();

    wok.unlock();
  }

  ~TomKhaGai()
  {
    wait();
  }

  protected:

  static QMutex wok;
};
```

Now let's have two persons cooking two soups:

```
TomKhaGai soup1, soup2;

soup1.start();
soup2.start();
```

9.6 Scoped Lock

There is no "scoped lock" like boost::mutex::scoped_lock lock(mutex) in plain Qt, but it is easily setup:

```
class QScopedLock
{
  public:

  QScopedLock(QMutex &mutex)
  : mutex_(mutex)
  {
    mutex_.lock();
  }

  ~QScopedLock()
  {
    mutex_.unlock();
  }

  protected:

  QMutex &mutex_;
};
```

With that we rewrite run():

```
void run()
{
```

```
    QScopedLock lock(wok);

    put_ingredients();
    cook_and_stir();
    clean_wok();
}
```

Ok, I cheated: There is a scoped lock in Qt, but it is named QMutexLocker.

9.7 Queued Connections

How to pass the produced soup to a consumer, e.g. to the main thread?

Passing messages between threads is easy with Qt. We simply pass objects through a signal/slot connection via so called *queued connections* [**17**].

Passing a message through a regular slot is done via parametrization:

message sender:

```
emit signal("message");
```

message receiver:

```
public slots:
  void slot(QString &message);
```

Then the receiver is invoked from the same thread as the sender.

To invoke the receiver on a different thread, we connect the signal and the slot with the option of a queued connection:

```
connect(sender, SIGNAL(signal(QString &)),
        receiver, SLOT(slot(QString &)),
        Qt::QueuedConnection);
```

Then a triggered signal in the sender thread has the effect of a copy of the parameters being stored in the event queue. The sender returns immediately after the copy has been posted. The copy is delivered to the receiver when the receiver thread yields to the event loop.

This scheme works as long as

- The type of the passed parameters is a class with a copy constructor.
- Either the sender or the receiver have an event loop running.
- The type of the parameter is known to Qt.

If the data type is unknown we register it before connecting the respective signal:

```
qRegisterMetaType<type>("type");
```

9.8 Producer Consumer Example

```
class Bowl
{
public:

  Bowl() // default constructor
  : empty(true);
  {}

  Bowl(const Bowl &bowl) // copy constructor
  {
    bowl.pour(*this);
  }

  ~Bowl() // destructor
  {}

protected:

  bool empty;

  pour(Bowl &bowl)
  {
    bowl.empty = empty;
    empty = true;
  }

  void put_ingredients()
  {
    empty = false;
  }

  cook_and_stir()
  {
    sleep(5);
  }

public:

  void eat()
  {
    empty = true;
  }

};

class TomKhaGai: public QThread, public Bowl
{
  Q_OBJECT;

public:

  void run()
  {
    wok.lock();
```

```
    put_ingredients();
    cook_and_stir();

    Bowl bowl;
    pour(bowl);

    wok.unlock();

    emit done(bowl);
  }

  ~TomKhaGai()
  {
    wait();
  }

protected:

  static QMutex wok;

signals:

  void done(Bowl &bowl);
};

class Guest
{
  Q_OBJECT;

public:

  Guest()
  {
    soup = new TomKhaGai;

    qRegisterMetaType<Bowl>("Bowl");

    connect(soup, SIGNAL(done(Bowl &)),
            this, SLOT(serve(Bowl &)),
            Qt::QueuedConnection);

    soup->start();
  }

  ~Guest()
  {
    delete soup;
  }

protected:

  TomKhaGai *soup;

private slots:

  void serve(Bowl &bowl)
  {
```

```
    std::cout << "soup is being served" << std::endl;
    bowl.eat(); // mmh, deliciuos!
  }

};
```

Usage example:

```
{
  Guest angela; // soup is being served after 5s
  Guest horst;  // soup is being served after 10s
} // destructor waits 10s
```

9.9 Mandelbrot Example

Another example of queued connections is the *Mandelbrot Example* [**18**].

in this example fractals are computed in a worker thread. The mouse position seeds the fractal iteration. Once a fractal image is finished, the image is passed to the main thread where it is displayed in a window. If another image is requested before the last one is finished, the worker thread is told to cancel it and restart.

10 Lesson 8: A Non-Blocking Qt Application

10.1 Job Queue

In the following we will outline another threading pattern: a job queue.

A job queue executes jobs one after another. There is not one thread for each job but one worker thread that works off all jobs.

First let's define a base class for a job. It is derived from std::string to encode information about the job, e.g. a command line or file to work on. It also has a pure virtual execute() method that does the job:

```
class Job: public std::string
{
  public:

  Job() : std::string() {}
  Job(const std::string &s) : std::string(s) {}
  virtual ~Job() {}

  virtual int execute() = 0;
};
```

Now we define a job queue:

```
typedef std::vector<Job *> Jobs;
```

And now we define a worker class that maintains the job queue:

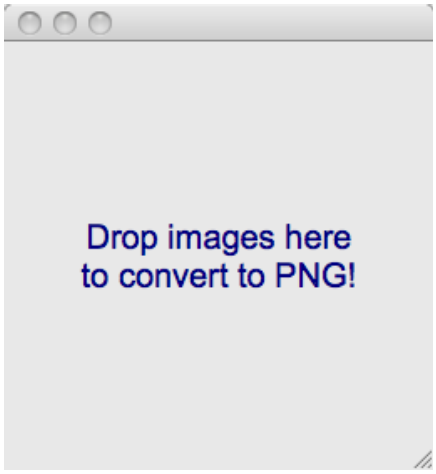
```
class worker : public QThread
{
  Q_OBJECT;

  public:

  //! default constructor
  worker(QObject *parent=0)
    : QThread(parent)
  {
    failure=0;
  }

  //! destructor
  virtual ~worker()
    {abort_jobs();}

  void run_job(Job *job)
  {
```

```
    block_jobs();
    jobs.push_back(job);
    unblock_jobs();
    start_jobs();
  }

  int finish_jobs()
  {
    wait4jobs();
    return(failure);
  }

  protected:

  virtual void block_jobs() {mutex.lock();}
  virtual void unblock_jobs() {mutex.unlock();}

  virtual void start_jobs()
  {
    if (!isRunning())
      start(LowPriority); // calls run() in a new thread
  }

  virtual void wait4jobs() {wait();}

  virtual void run()
  {
    int errorcode;

    block_jobs();

    failure=0;

    while (!jobs.empty())
    {
      Job *job=jobs[0];

      unblock_jobs();

      errorcode=job->execute();

      if (!errorcode)
        emit finishedJob(*job);
      else
      {
        emit failedJob(*job, errorcode);
        failure++;
      }

      block_jobs();

      jobs.erase(jobs.begin());
    }

    unblock_jobs();
  }

  private:
```

```
  Jobs jobs;
  int failure;

  QMutex mutex;

  signals:

  void finishedJob(const std::string &job);
  void failedJob(const std::string &job, int errorcode);
};
```

10.2 Threaded Image Conversion

The application “DropPNG” accepts a drop event and starts an image conversion job for each dropped file:



The image conversion jobs are executed in a background thread:

```
worker *thread = new worker;
```

When a drop event happens, the job queue of the background thread gets another job:

```
void dropEvent(QDropEvent *event)
{
  const QMimeData *mimeData = event->mimeData();

  if (mimeData->hasUrls())
  {
    event->acceptProposedAction();

    QList<QUrl> urlList = mimeData->urls();

    for (int i=0; i<urlList.size(); i++)
      {
```

```
        QUrl qurl = urlList.at(i);
        QString url = qurl.toString();

        if (url.startsWith("file://"))
          {
          url = url.remove("file://");

          std::cout << "start job for " << url.toStdString() << std::endl;
// debug

          MyConverterJob *job = new
MyConverterJob(url.toStdString());

          thread->run_job(job);
          }
        }
  }
}
```

When executing a job in the background thread, the file is loaded and saved as PNG:

```
class MyConverterJob: public Job
{
  public:

  MyConverterJob() : Job() {}
  MyConverterJob(const std::string &s) : Job(s) {}
  virtual ~MyConverterJob() {}

  virtual int execute(worker *worker)
  {
    // get file name to be converted
    QString filename(c_str());

    // try to load file as image
    QImage image;
    if (!image.load(filename)) return(1);

    // remove suffix from file name to yield output name
    QString output = filename;
    if (output.endsWith(".gif", Qt::CaseInsensitive))
output.truncate(output.size()-4);
    if (output.endsWith(".jpg", Qt::CaseInsensitive))
output.truncate(output.size()-4);
    if (output.endsWith(".tif", Qt::CaseInsensitive))
output.truncate(output.size()-4);
    output.append(".png");

    // save image as png
    if (!image.save(output, "PNG")) return(2);

    return(0);
  }
};
```

Access the full example via WebSVN:
Threading Example [**19**]

Checkout the Qt example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/example-07

10.3 KDE4 Integration

With OpenSuse 12.1 the KDE4 desktop is the default desktop manager. For programs to show up in the start menu we need to register them. To do so, we provide basic information in a .desktop file.

To register our image conversion app, we create the file “DropPNG.desktop” in the /usr/share/applications/ folder:

```
[Desktop Entry]
Type=Application
Exec=DropPNG %U
Icon=utilities-terminal
Terminal=false
Name=DropPNG
Categories=Application;Graphics;
```

Putting a .desktop file in the ~/.local/share/applications folder registers the app for a single user. Putting it in the /usr/share/autostart folder will start it automatically when a user logs in.

For an automatic installation, we add the above .desktop file to our source directory. Next, we modify the CMakelists.txt file to contain an install target that copies the .desktop file to the appropriate directory. The Qt executable is also copied to the appropriate binary directory:

```
IF (UNIX AND NOT APPLE)
   INSTALL(FILES DropPNG.desktop DESTINATION /usr/share/applications)
ENDIF (UNIX AND NOT APPLE)

INSTALL(
   TARGETS DropPNG
   RUNTIME DESTINATION bin
)
```

Now after running “make install”, we can right click at the DropPNG entry in the start menu and add it as a shortcut to the desktop.

10.4 KDE4 Drop Icon

Dragging files onto a widget is platform independent.

Dragging a file onto a desktop icon is platform dependent. We exercise it for KDE4.

First we place the DropPNG application into a panel, since only panel apps can be dropped onto.

When dropping a file from the File Manager view onto a panel icon, the dropped

files are passed to the opening program as standard command line arguments. So they end up to be parameters to int main(int argc, char *argv[]).

For that to work the Exec line of the .desktop file needs to contain

- %f as a place holder for an file name
- %F for a list of file names
- %u as a place holder for an url
- %U for a list of urls

To process one argument in the background thread, we add a process method

```
void MyQConverterWidget::process(QString file)
{
  if (file.startsWith("file://"))
    file = file.remove("file://");

  MyConverterJob *job = new MyConverterJob(file.toStdString());

  thread->run_job(job);
}
```

and pass all arguments from the command line to it:

```
QStringList args = QCoreApplication::arguments();
for (int i=1; i<args.size(); i++)
  converter.process(args[i]);
```

10.5 KDE4 Examples

Other Qt example applications for KDE4:

- *LibMini's QT Earth Viewer* [**20**]
- *QT V^3 Volume Renderer* [**21**]

11 Lesson 9: Qt Mobile

11.1 Qt Mobile

Qt is on the verge of being ported to most mobile platforms via an abstraction layer called Lighthouse.

Working (and maybe still experimental ports) are available for
- Android
- Blackberry
- Symbian

It seams that there will be no short-term support for
- iOS
- WinRT

since the licensing conditions do not allow the execution of code from RAM, which is necessary for the V8 Javascript JIT compiler as the backend of the Lighthouse project. Recent news indicate that there is work an a JS interpreter v4vm, which does not have those restrictions, but will have reduced performance compared to V8.

The current state for Android is that Qt 5.0 has support for **Android** for native Qt widget classes and QML, although it is recommended to wait for Qt 5.2 for commercial applications.

The Qt support is brought to Android via a community port of Qt named *Necessitas* [**22**].

11.2 Qt Mobile Features

- QSystemDeviceInfo
  - Battery, Temperature, Manufacturer, Keyboard types
- QGeoPositionInfoSource
  - GPS location
  - delivers QGeoPosition (polar coordinates → latitude/longitude)
- Mobile Device Sensors
  - QAccelerometer, QGyroscope, QMagnetometer
  - Angle and Momentum of Mobile Device
- Media
  - QAudioInput, QAudioOutput
  - QCamera
- Networking
  - QMessage, QContact
- etc...

11.3 Getting Started with Android Qt

Necessitas is the KDE community port of Qt for the Android platform. Here we are going to make a few first steps on the Android platform with Qt.

Let’s grab the Necessitas SDK and install it!

```
http://necessitas.kde.org/necessita/necessitas_sdk_installer.php
```

Here is a youtube video for getting started on Linux:

(:html5video filename=Qt-UI/AndroidQt:)

- **AndroidQt.mp4**

It is strongly suggested to use a Linux box as Necessitas is in beta status at the time of this writing!

Now, that we have installed the Necessitas SDK, what can we do with it? We open QtCreator and manage and develop our projects with it.

To do so, we should fire up QtCreator (on Linux start QtCreator/bin/necessitas from the cmd line) and make sure that QtCreator can see the Android Qt framework. Select the Build & Run section of the program options, then select the Qt Versions tab and check that it reads something like the following:

```
Necessitas 4.8.2n for Android armv5.
```

If that's the case, we are ready to create a new Qt Mobile Android project: Click on the Project tab on the left and create a new project of the type "Mobile Qt Application".

You will be asked to choose the ARM processor architecture. In the Necessitas SDK there is support for ARM v5 and ARM v7a. I am choosing **ARM v5** for backwards compatibility.

Our deployment platform is a Samsung Galaxy Ace 2 with an ARM Cortex-A8 processor core implementing the ARM v7 instruction set architecture. Still, it is better to choose ARM v5, since the emulator does not appear to be working with ARM v7a yet.

Now that we have the toolchain installed and ready to be used, we want to make a test by deploying a Qt Mobile example app. There is a simple test case available from the Qt developer pages that displays the battery status with the Qt Mobile frame work on a Nokia phone (or the Maemo simulator for Symbian phones):

```
http://doc.qt.digia.com/qtcreator-2.1/creator-mobile-example.html
```

We am going to do the same, but deploy a slightly modified Qt Mobile example on our Android smartphone.

That is, we are not going to simply connect the battery status to a widget via

```
#include <QSystemDeviceInfo>

QSystemDeviceInfo *deviceInfo = new QSystemDeviceInfo(this);

connect(deviceInfo, SIGNAL(batteryLevelChanged(int)), widget,
SLOT(setValue(int)));
```

but we are going to listen to geospatial positions updated from the GPS receiver as outlined at

```
http://qt-project.org/wiki/Retrieve_Location_Using_Qt_Mobility
```

11.4 Creating a new Android Qt Project

Now, we are going to create a new project with QtCreator to display the battery status as a starting point and change that later on to spot our GPS location.

In the QtCreator we select File → New File or Project → Applications → Mobile Qt Application → Choose.

Then we provide a name for the mobile application. For our example app that will show the actual GPS postion we choose “GPS-Spot”.

Next we confirm to use the default build kit for the armv5 instruction set. Done.

We have created a project template that contains the following files:

- GPS-Spot.desktop
- GPS-Spot.pro
- GPS-Spot64.png
- GPS-Spot80.png
- GPS-Spot_harmattan.desktop
- deployment.pri
- main.cpp
- mainwindow.cpp
- mainwindow.h
- mainwindow.ui

The files come with the necessary boiler plate code that we have to modify, as described in the following sections.

Declaring the Qt Mobility API

To use the Qt Mobility APIs we have to modify the .pro file to declare the Qt Mobility APIs that you use. Double click at the .pro file in the tree view of the edit pane of QtCreator to open an editor window.

This example uses the Mobility System Info API, so we must declare it, as illustrated by the following code snippet:

```
CONFIG += mobility
MOBILITY = systeminfo
```

Completing the MainWindow Header File

In the Projects view click on the Edit icon on the left and double-click the mainwindow.h file to open it for editing.

We include the System Device Info header file, as illustrated by the following code snippet:

```
#include <QSystemDeviceInfo>
```

Right after the header files (not before!) we declare to use the Qt mobility name space, as illustrated by the following code snippet:

```
QTM_USE_NAMESPACE
```

Also declare a private member variable as a pointer to a device info object to be created later on:

```
private:
   QSystemDeviceInfo *deviceInfo_;
```

Designing the User Interface

Double-click the mainwindow.ui file in the Projects editor view to launch the integrated Qt Designer.

Drag and drop a text label (QLabel) widget to the canvas.

In the Properties pane in the upper left, change the objectName of the QLabel object from “label” to “batteryLabel”.

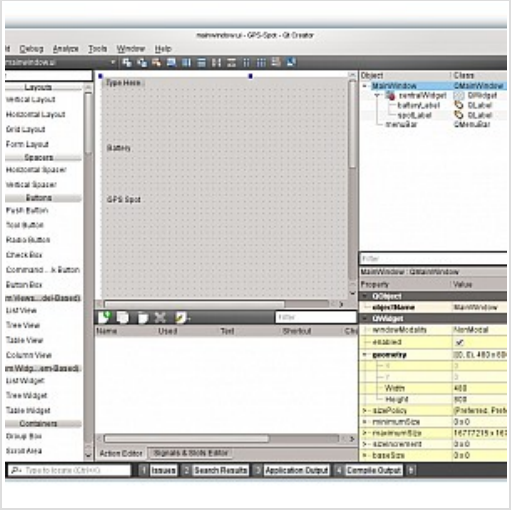
The necessary code to setup all designed UI elements of a widget is integrated into an auto-generated “ui” object. This object is is created in the constructor of the respective class. For example, the constructor of the MainWindow class creates a ui object of the type UI:MainWindow. When calling its setupUI method, the GUI elements are constructed according to the description of the mainwindow.ui file:

```
MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent), ui(new Ui::MainWindow)
{
   ui->setupUi(this);
```

```
}
```

All designed gui elements become a member variable of the ui object, referenced to by ui→objectName. For example, to set a text on a label object with the name objectName we call:

```
ui->objectName->setText("text");
```



Completing the MainWindow Source File

In the Projects view, double-click the mainwindow.cpp file to open it for editing.

Add the following code to the MainWindow constructor to read the battery level from the device info API and pass changing levels on to the battery label:

```
deviceInfo_ = new QSystemDeviceInfo(this);

ui->batteryLabel->setNum(deviceInfo_->batteryLevel());

connect(deviceInfo_, SIGNAL(batteryLevelChanged(int)),
        ui->batteryLabel, SLOT(setNum(double)));
```

And finally, we delete the device info in the MainWindow destructor:

```
~MainWindow()
{
  delete deviceInfo_;

  delete ui;
}
```

11.5 Running an Android Qt App

Now that you have all the necessary code, click on the "Hammer" symbol on the left to compile the project and check for errors and warnings.

To test it, we need to create a simulation device for the emulator.

In QtCreator we select Tools → Options and switch to the Android tab. Now we click at the "Start AVD Manager" button and create a new SD Card with 1024 MiB space for an Android Virtual Device with API Level 10 (Android 2.3.3). We also select WVGA 480×800 screen resolution matching the resolution of our Samsung Galaxy Ace 2 smartphone.

The best way is to clone the settings of the Google Nexus S Device shown in the device tab, which has the same screen resolution as the Galaxy Ace 2.

So we create an Android Virtual Device (AVD) from the Google Nexus S device and name it "Ace2".

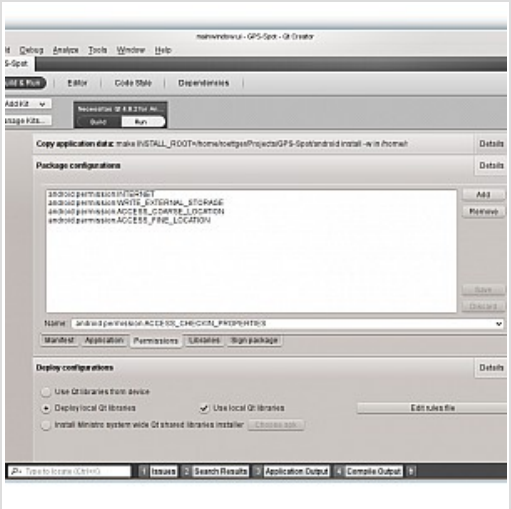
Click on the Ace2 AVD and start the emulator. Keep the emulator window open.

We download the "Ministro II.apk" from *here* [**23**] and run the following cmd line to install the Ministro package on the AVD:

```
adb install <full path file name of the ministro apk>
```

The command "adb" stands for Android Debug Bridge. It is located in the "android-sdk/platform-tools" directory of the SDK.

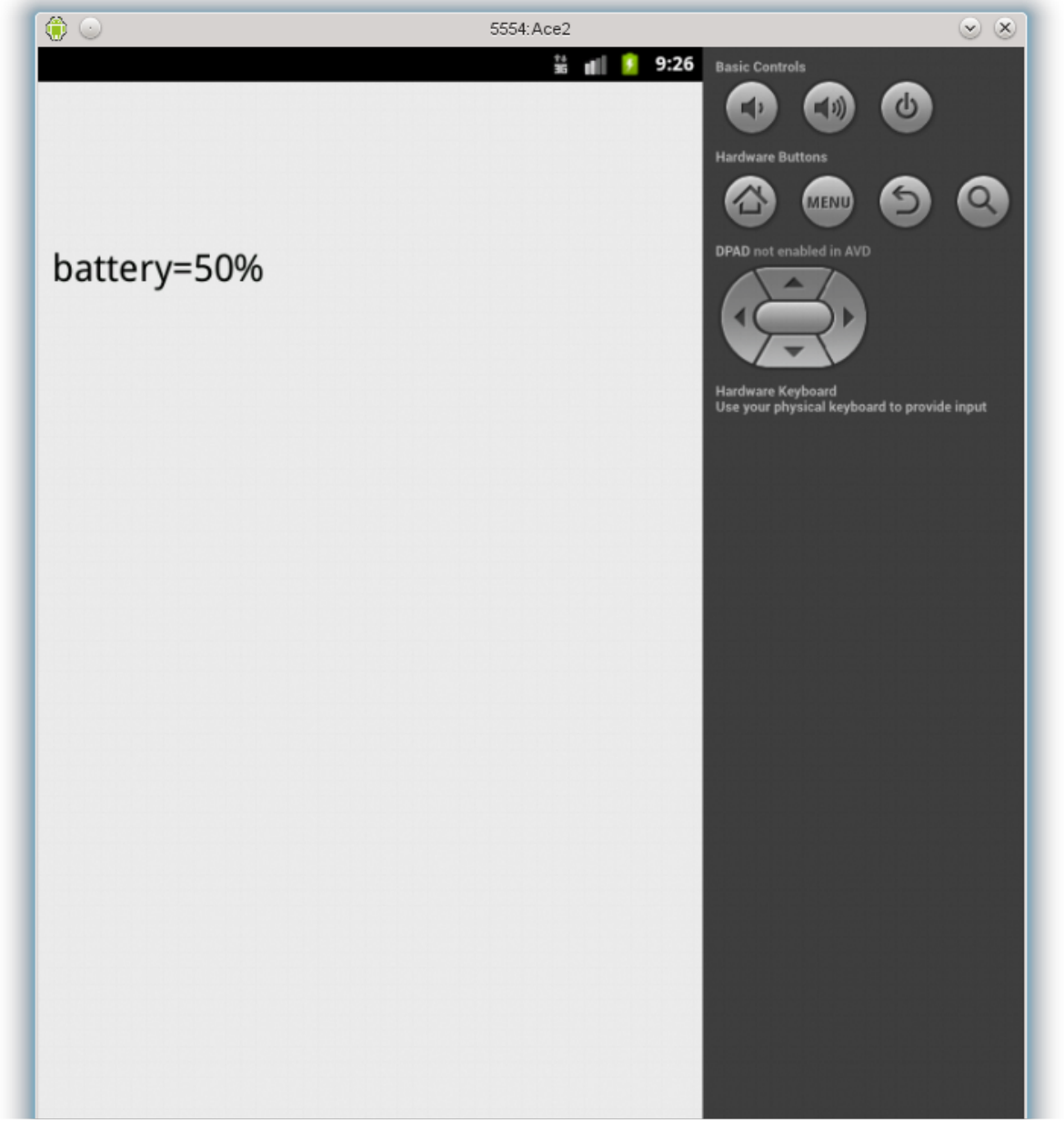
Back in Qt Creator we need to adjust a final setting before we can start the app in the simulator: click at the Project icon on the left side and switch to the tab of the project. Under "Build&Run" we click at the "Run" tab of the "Necessitas SDK for Android" kit and change the deploy settings to "Deploy local Qt libraries".



Now click at the green play button to run the app in the Qt Simulator.

Let's tweak the app a bit: we connect the battery level changed signal to a private slot setBatteryLevel(int) and create a more verbose output:

```
void MainWindow::setBatteryLevel(int level)
{
   QString text=QString("battery=%1%").arg(level);
   ui->batteryLabel->setText(text);
}
```



11.6 Android Qt GPS Example

First, we add another label named “spotLabel” with the integrated Qt Designer.

Then we setup a slot that receives location updates by calling the following setupGPS() method in the MainWindow constructor:

```
void MainWindow::setupGPS()
{
   // obtain the location data source
   locationInfo_ =
QGeoPositionInfoSource::createDefaultSource(this);

   // select positioning method

locationInfo_->setPreferredPositioningMethods(QGeoPositionInfoSource::AllPositioningMethods);

   // when the position has changed the setGPSLocation slot is called
   connect(locationInfo_,
SIGNAL(positionUpdated(QGeoPositionInfo)),
         this, SLOT(setGPSLocation(QGeoPositionInfo)));

   // start listening for position updates
   locationInfo_->startUpdates();
}
```

And output the updated locations, if any, in lat/lon coordinates:

```
void MainWindow::setGPSLocation(QGeoPositionInfo geoPositionInfo)
{
   QString text="Location=unknown";

   if (geoPositionInfo.isValid())
   {
      // get the current location coordinates
      QGeoCoordinate geoCoordinate = geoPositionInfo.coordinate();

      // transform coordinates to lat/lon
      qreal latitude = geoCoordinate.latitude();
      qreal longitude = geoCoordinate.longitude();

      text=QString("Latitude=%1\nLongitude=%2")
            .arg(latitude,0,'g',8)
            .arg(longitude,0,'g',8);
   }

   ui->spotLabel->setText(text);
}
```

Add the following line to the .pro file:

```
MOBILITY += location sql
```

Make sure that “Deploy local Qt libraries” is checked again. Once the libraries have been deployed the first time, this option is unchecked automatically. To

deploy additional libraries it needs to be checked again.

To get access to the location services we need to add appropriate *permissions* [**24**] to the Android manifest.

In the project view, open the “Run” Settings and under “Package Configuration” select the “Permissions” tab. Add the following permissions:

- ACCESS_COARSE_LOCATION
- ACCESS_FINE_LOCATION

Don’t forget to save!

Also select the “Libraries” tab and check that QtLocation and QtSql are selected.

We run the app and provide a test gps location by connecting to the emulator via telnet:

```
telnet localhost 5554
```

Then we can use the geo command to set a geographic position

```
geo fix <longitude value> <latitude value>
```

For example, the geographic position of Kailua, Hawai’i:

```
geo fix -157.739515 21.397370
```

For some reason the GPS location slot is not receiving any updates. Instead the updateTimeout signal is emitted. Also, the value of lastKnownPosition can be bogus when the GPS receiver tries to reestablish a lost satellite connection. After excessive reading and testing, giving me major headaches, it appears that Qt Mobility (Qt 4.8) is to be blamed.

As a workaround we connect the following timeout slot to the updateTimeout signal, so that at least we are getting notified that we do not have a GPS position available:

```
void MainWindow::timeout()
{
   setGPSLocation(locationInfo_->lastKnownPosition());
   locationInfo_->startUpdates();
}
```

It appears we have to wait for Qt 5.0 to fix the GPS support for the emulator. We’ll follow the latest **news** to monitor work on Qt5.

Alternatively, we try the app on a real Android device and not the emulator.

11.7 Android Qt on a Samsung Galaxy Ace 2

First off, we need to see if the Android Debug Bridge recognizes any Android device plugged in via USB. So plug in your phone and run:

```
adb devices
```

If you see it listed, then you are almost done:

- install Ministro on your device
  - adb -d install <path to Ministro apk>
- install your App on your device
  - adb -d install <path to your apk>

If you do not see it listed, make sure that it is actually connected by running "lsusb". If you see a Android device connected on the USB bus, then we adhere to Google's *instructions on setting up a hardware device* [**25**].

Let's see how it works out for the Ace 2 with OpenSuse 12.1:

```
> adb devices
List of devices attached
```

So adb does not see my Ace 2, yet.

```
> lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 004: ID 058f:a014 Alcor Micro Corp.
Bus 001 Device 005: ID 0bda:0139 Realtek Semiconductor Corp.
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 006: ID 0cf3:3005 Atheros Communications, Inc. AR3011 Bluetoo
Bus 003 Device 003: ID 04e8:6860 Samsung Electronics Co., Ltd
```

But it is connected to Bus 3 Device 3, good!

So we follow the Google instructions:

1. Enable USB debugging on your device.
   On most devices running Android 3.2 or older, you can find the option under Settings → Applications → Development. On Android 4.0 and newer, it's in Settings → Developer options.
   Note: On Android 4.2 and newer, Developer options is hidden by default. To make it available, go to Settings → About phone and tap Build number seven times. Return to the previous screen to find Developer options.
2. Also allow Mock locations on your device.

```
> adb devices
List of devices attached
9E3838FEF88FEBEA47D7DCE99F168BE device
```

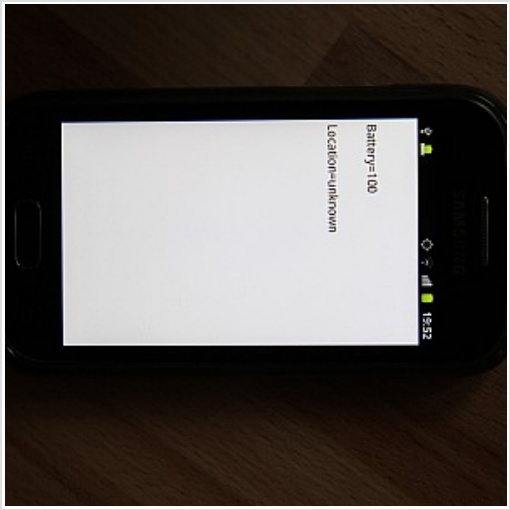
Here we go :)

```
> adb -d install ~/Projects/Ministro\ II.apk
145 KB/s (542653 bytes in 3.631s)
```

```
        pkg: /data/local/tmp/Ministro II.apk
Success
```

Now run your application in QtCreator and it will automatically recognize the attached Android device. Don't forget to check "Deploy local Qt libraries"!

Some indoor screen shots /wo and /w GPS switched on:



Access the full GPS-Spot example via WebSVN:
Qt Mobility Example [**26**]

Checkout the Qt Mobility example via SVN:
svn co svn://schorsch.efi.fh-nuernberg.de/qt-examples/GPS-Spot

12 Lesson 10: Outlook

12.1 Further Reading

- Special Widgets
  - Qt Scene Graph
  - Qt Phonon Media Player
- ModelView Widgets
  - Adaptors and Delegates from Form to Model
  - Rapid Prototyping
- QML and Javascript
- Qt and Python
  - Pyside, PyQt
  - Scripting
- Qt Linuguist
  - Application Translation and Localization
- Qt Mobile
  - Lighthouse
  - *KDE.org hosting Ministro* [**27**]

Links

1. **qt-project.org/doc/qt-4.8/threads-mandelbrot.html**
2. **www.heise.de/newsticker/meldung/Alpha-Version-von-Qt-5-1-unterstuetzt-Android-und-iOS-1837467.html**
3. **schorsch.efi.fh-nuernberg.de/mewiki/index.php/Tutorials/SvnHowTo**
4. **packages.ubuntu.com**
5. **qt-project.org/doc/qt-4.8**
6. **qt-project.org/doc/qt-5.0/qtdoc/qtexamplesandtutorials.html**
7. **qt-project.org/books**
8. **schorsch.efi.fh-nuernberg.de/websvn**
9. **qt-project.org/doc/qt-4.8/qpainter.html**
10. **qt-project.org/doc/qt-4.8/coordsys.html**
11. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/example-02**
12. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/example-06**
13. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/example-03**
14. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/example-04**
15. **qt-project.org/doc/qt-4.8/qcheckbox.html#signals**
16. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/example-05**
17. **qt-project.org/doc/qt-4.8/threads-qobject.html**
18. **qt-project.org/doc/qt-4.8/threads-mandelbrot.html**
19. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/example-07**
20. **code.google.com/p/libmini**
21. **code.google.com/p/vvv**
22. **blog.qt.digia.com/blog/2013/03/13/preview-of-qt-5-for-android**
23. **filesmaster.kde.org/necessitas/installer/release/Ministro%20II.apk**
24. **developer.android.com/reference/android/Manifest.permission.html**
25. **developer.android.com/tools/device.html#setting-up**
26. **schorsch.efi.fh-nuernberg.de/websvn/listing.php?repname=qt-examples&path=/GPS-Spot**
27. **www.omat.nl/2012/08/11/how-necessitas-grew-to-be-a-20tb-of-traffic-a-month-project**