
Qt5 Cadaques

Release 2015-03

JRyannel,JTheelin

March 14, 2018 at 02:55 CET

1	Meet Qt 5	3
1.1	Preface	3
1.2	Qt 5 Introduction	4
1.3	Qt Building Blocks	8
1.4	Qt Project	10
2	Get Started	11
2.1	Installing Qt 5 SDK	11
2.2	Hello World	11
2.3	Application Types	13
2.4	Summary	21
3	Qt Creator IDE	23
3.1	The User Interface	23
3.2	Registering your Qt Kit	24
3.3	Managing Projects	24
3.4	Using the Editor	25
3.5	Locator	25
3.6	Debugging	26
3.7	Shortcuts	26
4	Quick Starter	27
4.1	QML Syntax	27
4.2	Basic Elements	32
4.3	Components	36
4.4	Simple Transformations	39
4.5	Positioning Elements	42
4.6	Layout Items	46
4.7	Input Elements	49
4.8	Advanced Techniques	53
5	Fluid Elements	55
5.1	Animations	55
5.2	States and Transitions	70
5.3	Advanced Techniques	74
6	Model-View-Delegate	75
6.1	Concept	75
6.2	Basic Models	76
6.3	Dynamic Views	80
6.4	Delegate	89
6.5	Advanced Techniques	96
6.6	Summary	105

7	Canvas Element	107
7.1	Convenient API	109
7.2	Gradients	110
7.3	Shadows	111
7.4	Images	112
7.5	Transformation	113
7.6	Composition Modes	114
7.7	Pixel Buffers	115
7.8	Canvas Paint	116
7.9	Porting from HTML5 Canvas	118
8	Particle Simulations	125
8.1	Concept	125
8.2	Simple Simulation	125
8.3	Particle Parameters	128
8.4	Directed Particles	129
8.5	Particle Painters	132
8.6	Affecting Particles	134
8.7	Particle Groups	138
8.8	Summary	145
9	Shader Effects	147
9.1	OpenGL Shaders	147
9.2	Shader Elements	148
9.3	Fragment Shaders	150
9.4	Wave Effect	154
9.5	Vertex Shader	156
9.6	Curtain Effect	164
9.7	Qt GraphicsEffect Library	167
10	Multimedia	171
10.1	Playing Media	171
10.2	Sound Effects	173
10.3	Video Streams	174
10.4	Capturing Images	175
10.5	Advanced Techniques	177
10.6	Summary	178
11	Networking	179
11.1	Serving UI via HTTP	179
11.2	Templating	182
11.3	HTTP Requests	182
11.4	Local files	185
11.5	REST API	186
11.6	Authentication using OAuth	191
11.7	Engin IO	192
11.8	Web Sockets	192
11.9	Summary	197
12	Storage	199
12.1	Settings	199
12.2	Local Storage - SQL	200
12.3	Other Storage APIs	204
13	Dynamic QML	205
13.1	Loading Components Dynamically	205
13.2	Creating and Destroying Objects	209
13.3	Tracking Dynamic Objects	212
13.4	Summary	214

14 JavaScript	215
14.1 Browser/HTML vs QtQuick/QML	216
14.2 The Language	216
14.3 JS Objects	218
14.4 Creating a JS Console	219
15 Qt and C++	223
15.1 A Boilerplate Application	224
15.2 The QObject	227
15.3 Build Systems	229
15.4 Common Qt Classes	232
15.5 Models in C++	237
16 Extending QML with C++	247
16.1 Understanding the QML Run-time	247
16.2 Plugin Content	249
16.3 Creating the plugin	250
16.4 FileIO Implementation	251
16.5 Using FileIO	252
16.6 Summary	259
17 Assets	261
17.1 Offline Books	261
17.2 Source Code Examples	261

Last Build: March 11, 2018 at 15:30 CET

Welcome to the online book of Qt5 Cadaques! Why Qt5? Because Qt5 is awesome! Why cadaques? Because one of the authors had a great holiday in this rocky coast line in the north-east of Spain.

The entire collection of chapters covering Qt5 programming, written by Juergen Bocklage-Ryannel and Johan Thelin, is available here. All book content is licensed under the [Creative Commons Attribution Non Commercial Share Alike 4.0](#) license and examples are licensed under the [BSD](#) license.

We are heavily working on this book and that means several things:

1. **It's not done.** We will be releasing new chapters from time to time and updating existing chapters on the go.
2. **We love your support.** If you find any errors or have suggestions, please use our feedback system (the [Issues: Create | View](#) links). It will create a new ticket-entry in our ticket system and help us to keep track.
3. **Be patient.** We are working in our spare time on the book and we depend on the support of our companies and family.

Enjoy!

Content

Meet Qt 5

Section author: jryannel

Note: The source code of this chapter can be found in the assets folder.

This book shall provide you a walk through the different aspect of application development using Qt version 5.x. It focuses on the new Qt Quick technology but also provides necessary information of writing C++ back-ends and extension for Qt Quick.

This chapter provides a high level overview of Qt 5. It shows the different application models available for developers and a Qt 5 showcase application to get a sneak preview of things to come. Additionally the chapter aims to provide a wide overview of the Qt 5 content and how to get in touch with the makers of Qt 5.

Preface

History

Qt 4 has evolved since 2005 and provided a solid ground for thousands of applications and even full desktop and mobile systems. The usage patterns of computer users changed in the recent years. From stationary PCs towards portable notebook and nowadays mobile computers. The classical desktop is more and more replaced with mobile touch-based always connected screens. With it the desktop UX paradigms also changes. Where as in the past Windows UI has dominated the world we spend more time nowadays on other screens with another UI language.

Qt 4 was designed to satisfy the desktop world to have a coherent set of UI widgets available on all major platforms. The challenge for Qt users has changed today and it lies more to provide a touch-based user interface for a customer driven user interface and to enable modern user interface on all major desktop and mobile systems. Qt 4.7 started to introduce the Qt Quick technology which allows users to create a set of user interface components from simple elements to achieve a complete new UI, driven by customer demands.

Qt 5 Focus

Qt 5 is a complete refreshing of the very successful Qt 4 release. With Qt 4.8, the Qt 4 release is almost 7 years old. It's time to make an amazing toolkit even more amazing. Qt 5 is focused on the following:

- **Outstanding Graphics:** Qt Quick 2 is based on OpenGL (ES) using a scene graph implementation. The recomposed graphics stack allows a new level of graphic effects combined with an ease of use never seen before in this field.
- **Developer Productivity:** QML and JavaScript are the primary means for UI creation. The back-end will be driven by C++. The split between JavaScript and C++ allows a fast iteration for front-end developers concentrating on creating beautiful user interfaces and back-end C++ developers concentrating on stability, performance and extending the runtime.

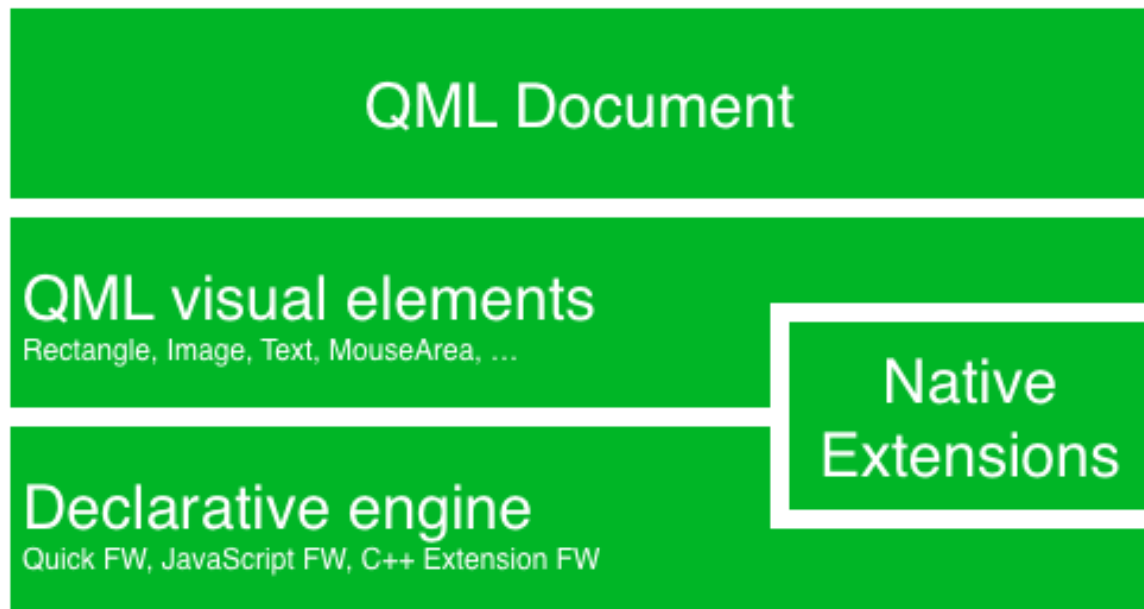
- **Cross-platform portability:** With the consolidated Qt Platform Abstraction, it is now possible to port Qt to a wider range of platforms easier and faster. Qt 5 is structured around the concept of Qt Essentials and Add-ons, which allows OS developer to focus on the essentials modules and leads to a smaller runtime altogether.
- **Open Development:** Qt is now a truly open-governance project hosted at qt.io. The development is open and community driven.

Qt 5 Introduction

Qt Quick

Qt Quick is the umbrella term for the user interface technology used in Qt 5. Qt Quick itself is a collection of several technologies:

- QML - Markup language for user interfaces
- JavaScript - The dynamic scripting language
- Qt C++ - The highly portable enhanced c++ library



Similar to HTML, QML is a markup language. It is composed of tags called elements in Qt Quick enclosed in curly brackets `Item {}`. It was designed from the ground up for the creation of user interfaces, speed and easier reading for developers. The user interface can be enhanced using JavaScript code. Qt Quick is easily extendable with your own native functionality using Qt C++. In short the declarative UI is called the front-end and the native parts are called the back-end. This allows you to separate the computing intensive and native operation of your application from the user interface part.

In a typical project the front-end is developed in QML/JavaScript and the back-end code, which interfaces with the system and does the heavy lifting, is developed using Qt C++. This allows a natural split between the more design oriented developers and the functional developers. Typically the back-end is tested using Qt own unit testing framework and exported for the front-end developers to be used.

Digesting an User Interface

Let's create a simple user interface using Qt Quick, which showcases some aspects of the QML language. At the end we will have a paper windmill with rotating blades.



We start with an empty document called `main.qml`. All QML files will have the ending `.qml`. As a markup language (like HTML) a QML document needs to have one and only one root element, which in our case is the `Image` element with a width and height based on the background image geometry:

```
import QtQuick 2.5

Image {
    id: root
    source: "images/background.png"
}
```

As QML does not make any restriction which element type is the root element we use an `Image` element with the `source` property set to our background image as the root element.

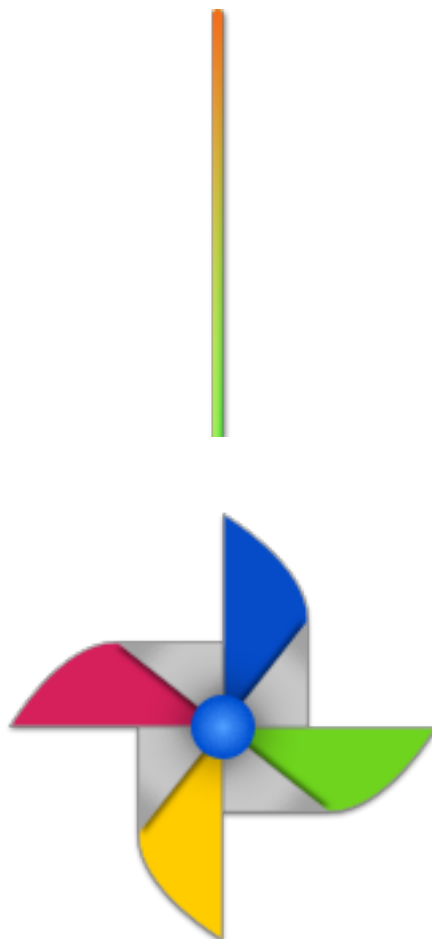


Note: Each element has properties, e.g. an image has a `width`, `height` but also other properties like a `source` property. The size of the image element is automatically deduced from the image size. Otherwise we would need to set the `width` and `height` property to some useful pixel values.

The most standard elements are located in the `QtQuick` module which we include in the first line with the `import` statement.

The `id` special property is optional and contains an identifier to reference this element later in other places in the document. Important: An `id` property cannot be changed after it has been set and it cannot be set during runtime. Using `root` as the `id` for the root-element is just a habit by the author and makes referencing the top-most element predictable in larger QML documents.

The foreground elements pole and pin wheel of our user interface are placed as separate images.



The pole needs to be placed in the horizontal center of the background towards the bottom. And the pinwheel can be placed in the center of the background.

Normally your user interface will be composed of many different element types and not only image elements like in this example.

```
Image {  
    id: root  
    ...  
    Image {  
        id: pole  
        anchors.horizontalCenter: parent.horizontalCenter  
        anchors.bottom: parent.bottom  
        source: "images/pole.png"  
    }  
  
    Image {  
        id: wheel  
        anchors.centerIn: parent  
        source: "images/pinwheel.png"  
    }  
}
```



```
...
}
```

To place the pin wheel at the central location we use a complex property called `anchor`. Anchoring allows you to specify geometric relations between parent and sibling objects. E.g. Place me in the center of another element (`anchors.centerIn: parent`). There are left, right, top, bottom, centerIn, fill, verticalCenter and horizontalCenter relations on both ends. Sure, they need to match. It does not make sense to anchor my left side to the top side of an element.

So we set the pinwheel to be centered in the parent our background.

Note: Sometime you will need to make small adjustments on the exact centering. This would be possible with `anchors.horizontalCenterOffset` or with `anchors.verticalCenterOffset`. Similar adjustments properties are also available to all the other anchors. Please consult the documentation for a full list of anchors properties.

Note: Placing an image as a child element of our root element (the `Image` element) shows an important concept of a declarative language. You describe the user interface in the order of layers and grouping, where the topmost layer (our rectangle) is drawn first and the child layers are drawn on top of it in the local coordinate system of the containing element.

To make the showcase a little bit more interesting, we would like to make the scene interactive. The idea is to rotate the wheel when the user pressed the mouse somewhere in the scene.

We use the `MouseArea` element and make it as big as our root element.

```
Image {
    id: root
    ...
    MouseArea {
        anchors.fill: parent
        onClicked: wheel.rotation += 90
    }
    ...
}
```

The mouse area emit signals when a user clicks inside it covered area. You can hook onto this signal overriding the `onClicked` function. In this case the reference the wheel image and change its rotation by +90 degree.

Note: This works for every signal, the naming is `on + SignalName` in title cases. Also all properties emit a signal when their value changed. The naming is:

`on + PropertyName + Changed`

If a width property is changing you can observe it with `onWidthChanged: print(width)` for example.

Now the wheel will rotate, but it is still not fluid yet. The rotation property changes immediately. What we would like that the property changes by 90 degree over time. Now animations come into play. An animation defines how a property change is distributed over a duration. To enable this we use an animation type called property behavior. The `Behaviour` does specify an animation for a defined property for every change applied to that property. In short every time the property changes, the animation is run. This is only one of several ways of declaring an animation in QML.

```
Image {
    id: root
    Image {
        id: wheel
        Behavior on rotation {
```

```
        NumberAnimation {  
            duration: 250  
        }  
    }  
}
```

Now whenever the property rotation of the wheel changes it will be animated using a `NumberAnimation` with a duration of 250 ms. So each 90 degree turn will take 250 ms.



Note: You will not actually see the wheel blurred. This is just to indicate the rotation. But a blurred wheel is in the assets folder. Maybe you want to try to use that.

Now the wheel looks already much better. I hope this has given you a short idea of how Qt Quick programming works.

Qt Building Blocks

Qt 5 consists of a large amount of modules. A module in general is a library for the developer to use. Some modules are mandatory for a Qt enabled platform. They form a set called *Qt Essentials Modules*. Many modules are optional and form the *Qt Add-On Modules*. It's expected that the majority of developers will not have the need to use them, but it's good to know them as they provide invaluable solutions to common challenges.

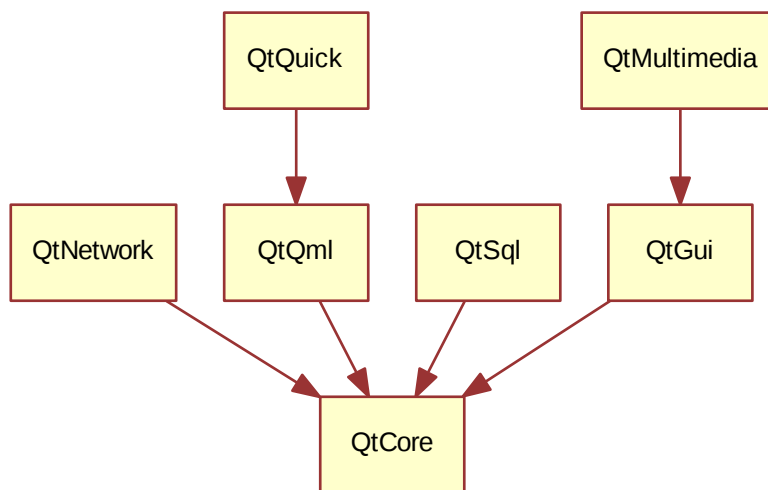
Qt Modules

The Qt Essentials modules are mandatory for a Qt enabled platform. They offer the foundation to develop a modern Qt 5 Application using Qt Quick 2.

Core-Essential Modules

The minimal set of Qt 5 modules to start QML programming.

Module	Description
Qt Core	Core non-graphical classes used by other modules
Qt GUI	Base classes for graphical user interface (GUI) components. Includes OpenGL.
Qt Multimedia	Classes for audio, video, radio and camera functionality.
Qt Network	Classes to make network programming easier and more portable.
Qt QML	Classes for QML and JavaScript languages.
Qt Quick	declarative framework for building highly dynamic applications with custom user interfaces.
Qt SQL	Classes for database integration using SQL.
Qt Test	Classes for unit testing Qt applications and libraries.
Qt WebKit	Classes for a WebKit2 based implementation and a new QML API. See also Qt WebKit Widgets in the add-on modules.
Qt WebKit Widgets	WebKit1 and QWidget-based classes from Qt 4.
Qt Widgets	Classes to extend Qt GUI with C++ widgets.



Qt Addon Modules

Besides the essential modules, Qt offers additional modules for software developers, which are not part of the release. Here is a short list of add-on modules available.

- Qt 3D - A set of APIs to make 3D graphics programming easy and declarative.
- Qt Bluetooth - C++ and QML APIs for platforms using Bluetooth wireless technology.
- Qt Contacts - C++ and QML APIs for accessing addressbooks / contact databases
- Qt Location - Provides location positioning, mapping, navigation and place search via QML and C++ interfaces. NMEA backend for positioning
- Qt Organizer - C++ and QML APIs for accessing organizer events (todos, events, etc.)
- Qt Publish and Subscribe
- Qt Sensors - Access to sensors via QML and C++ interfaces.
- Qt Service Framework - Enables applications to read, navigate and subscribe to change notifications.
- Qt System Info - Discover system related information and capabilities.

- Qt Versit - Support for vCard and iCalendar formats
- Qt Wayland - Linux only. Includes Qt Compositor API (server), and Wayland platform plugin (clients)
- Qt Feedback - Tactile and audio feedback to user actions.
- Qt JSON DB - A no-SQL object store for Qt.

Note: As these modules are not part of the release the state differ between modules, depending how many contributors are active and how well it's get tested.

Supported Platforms

Qt supports a variety of platforms. All major desktop and embedded platforms are supported. Through the Qt Application Abstraction, nowadays it's easier to port Qt over to your own platform if required.

Testing Qt 5 on a platform is time consuming. A sub-set of platforms was selected by the Qt Project to build the reference platforms set. These platforms are thoroughly tested through the system testing to ensure the best quality. Mind you though: no code is error free.

Qt Project

From the [Qt Project wiki](#):

“The Qt Project is a meritocratic consensus-based community interested in Qt. Anyone who shares that interest can join the community, participate in its decision making processes, and contribute to Qt’s development.”

The Qt Project is an organisation which develops the open-source part of the Qt further. It forms the base for other users to contribute. The biggest contributor is DIGIA, which holds also the comercial rights to Qt.

Qt has an open-source aspect and a comercial aspect for companies. The comercial aspect is for companies which can not or will not comply with the open-source licenses. Without the comercial aspect these companies would not be able to use Qt and it would not allow DIGIA to contribute so much code to the Qt Project.

There are many companies world-wide, which make their living out of consultancy and product development using Qt on the various platforms. There are many open-source projects and open-source developers, which rely on Qt as their major development library. It feels good to be part of this vibrant community and to work with this awesome tools and libraries. Does it make you a better person? Maybe:-)

Contribute here: <http://wiki.qt.io/>

Get Started

Section author: jryannel

This chapter will introduce you to developing with Qt 5. We will show you how to install the Qt SDK and how you can create as well as run a simple *hello world* application using the Qt Creator IDE.

Note: The source code of this chapter can be found in the assets folder.

Installing Qt 5 SDK

The Qt SDK include the tools needed to build desktop or embedded applications. The latest version can be grabbed from the [Qt-Company](#) homepage. There are offline and online installer. The author personally prefers the online installer package as it allows you to install and update several Qt releases. This is would be the recommended way to start. The SDK itself has a maintenance tool which will allow you to update the SDK to the latest version.

The Qt SDK is easy to install and comes with its own IDE for rapid development called *Qt Creator*. The IDE is a highly productive environment for Qt coding and recommended to all readers. Many developers use Qt from the command line and you are free to use a code editor of your choice.

When installing the SDK, you should select the default option and ensure that Qt 5.x is enabled. Then you are ready to go.

Hello World

To test your installation, we will create a small *hello world* application. Please open Qt Creator and create a Qt Quick UI Project (*File* → *New File or Project* → *Qt Quick Project* → *Qt Quick UI*) and name the project HelloWorld.

Note: The Qt Creator IDE allows you to create various types of applications. If not otherwise stated, we always use a Qt Quick UI project.

Hint: A typical Qt Quick application is made out of a runtime called the QmlEngine which loads the initial QML code. The developer can register C++ types with the runtime to interface with the native code. These C++ types can also be bundled into a plugin and then dynamically loaded using an import statement. The `qmlscene` and `qml` tool are pre-made runtimes, which can be used directly. For the beginning we will not cover the native side of development and focus only on the QML aspects of Qt 5.

Qt Creator will create several files for you. The `HelloWorld.qmlproject` file is the project file where the relevant project configuration is stored. This file is managed by Qt Creator so don't edit.

Another file, `HelloWorld.qml`, is our application code. Open it and try to guess what the application does and then continue to read on.

```
// HelloWorld.qml

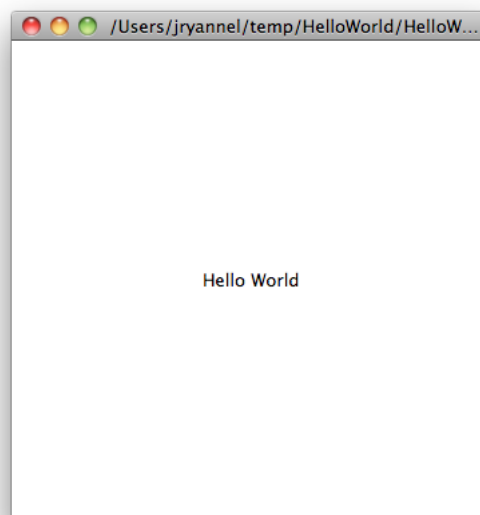
import QtQuick 2.5

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Hello World"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

The `HelloWorld.qml` is written in the QML language. We will discuss the QML language in more depth in the next chapter. QML describes the user interface as a tree of hierarchical elements. In this case, a rectangle of 360 x 360 pixels with a centered text reading “Hello World”. To capture user clicks a mouse area spans the whole rectangle and when the user clicks it, the application quits.

To run the application on your own, please press the  *Run* tool on the left side or select *Build* → *Run* from the menu.

Qt Creator will start the `qmlscene` and passes the QML document as the first argument. The `qmlscene` will parse the document and launch the user interface. Now you should see something like this:



Qt 5 seems to be working and we are ready to continue.

Tip: If you are a system integrator, you’ll want to have Qt SDK installed to get the latest stable Qt release as well as a Qt version compiled from source code for your specific device target.

Build from Scratch

If you'd like to build Qt 5 from the command line, you'll first need to grab a copy of the code repository and build it.

```
git clone git://gitorious.org/qt/qt5.git
cd qt5
./init-repository
./configure -prefix $PWD/qtbase -opensource
make -j4
```

After a successful compilation and 2 cups of coffee, Qt 5 will be available in the `qtbase` folder. Any beverage will suffice, however, we suggest coffee for best results.

If you want to test your compilation, simply start `qtbase/bin/qmlscene` and select a Qt Quick example to run it ...or follow just us into the next chapter.

To test your installation, we will create a small hello world application. Please create a simple `example.qml` file using your favorite text editor and paste the following content inside:

```
// HelloWorld.qml

import QtQuick 2.5

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Greetings from Qt 5"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

You can run now the example by using the default runtime which comes with Qt 5:

```
$ qtbase/bin/qmlscene
```

Application Types

This section is a run through of the different possible application types someone could write with Qt 5. It's not limited to the presented selection but it should give the reader a better idea about what can be done with Qt 5 in general.

Console Application

A console application does not provide any graphical user interface and will normally be called as part of a system service or from the command line. Qt 5 comes with a series of ready-made components which help you to create console cross platform applications very efficiently. For example the networking file APIs. Also string handling and, since Qt 5.1, efficient command line parser. As Qt is a high-level API on top of C++, you get programming speed paired with execution speed. Don't think of Qt as being *just* a UI toolkit – it has so much more to offer.

String Handling

In the first example we demonstrate how someone could very simply add 2 constant strings. This is not a very useful application but it gives you an idea of what a native C++ application, without an event loop, could look like.

```
// module or class includes
#include <QtCore>

// text stream is text-encoding aware
QTextStream cout(stdout, QIODevice::WriteOnly);

int main(int argc, char** argv)
{
    // avoid compiler warnings
    Q_UNUSED(argc)
    Q_UNUSED(argv)
    QString s1("Paris");
    QString s2("London");
    // string concatenation
    QString s = s1 + " " + s2 + "!";
    cout << s << endl;
}
```

Container Classes

This example adds a list and list iteration to the application. Qt comes with a large collections of container classes which are easy to use and use the same API paradigms as the rest of Qt classes.

```
QString s1("Hello");
QString s2("Qt");
QList<QString> list;
// stream into containers
list << s1 << s2;
// Java and STL like iterators
QListIterator<QString> iter(list);
while(iter.hasNext()) {
    cout << iter.next();
    if(iter.hasNext()) {
        cout << " ";
    }
}
cout << "!" << endl;
```

Here we show some advanced list function, which allow you to join a list of strings into one string. This is very handy when you need to proceed line based text input. The inverse (string to string-list) is also possible using `QString::split()` function.

```
QString s1("Hello");
QString s2("Qt");
// convenient container classes
QStringList list;
list << s1 << s2;
// join strings
QString s = list.join(" ") + "!";
cout << s << endl;
```


File IO

In the next snippet we read a CSV file from the local directory and loop over the rows to extract the cells from each row. Doing this we get the table data from the CSV file in ca. 20 lines of code. File reading gives us just a byte stream, to be able to convert it into a valid Unicode text we need to use the text stream and pass in the file as a lower-level stream. For writing CSV files you would just need to open the file in the write mode and pipe the lines into the text stream.

```
QList<QStringList> data;
// file operations
QFile file("sample.csv");
if(file.open(QIODevice::ReadOnly)) {
    QTextStream stream(&file);
    // loop forever macro
    forever {
        QString line = stream.readLine();
        // test for null string 'String()'
        if(line.isNull()) {
            break;
        }
        // test for empty string 'QString("")'
        if(line.isEmpty()) {
            continue;
        }
        QStringList row;
        // for each loop to iterate over containers
        foreach(const QString& cell, line.split(",")) {
            row.append(cell.trimmed());
        }
        data.append(row);
    }
}
// No cleanup necessary.
```

This concludes our section about console based application with Qt.

Widget Application

Console based applications are very handy but sometimes you need to have a UI to show. In addition, UI-based applications will likely need a back-end to read/write files, communicate over the network, or keep data in a container.

In this first snippet for widget-based applications we do as little as needed to create a window and show it. A widget without a parent in the Qt world is a window. We use the scoped pointer to ensure the widget is deleted when the scoped pointer goes out of scope. The application object encapsulates the Qt runtime and with the `exec()` call we start the event loop. From there on the application reacts only on events triggered by mouse or keyboard or other event providers like networking or file IO. The application will only exit when the event loop is exited. This is done by calling `quit()` on the application or by closing the window.

When you run the code you will see a window with the size of 240 x 120 pixel. That's all.

```
#include <QtGui>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QScopedPointer<QWidget> widget(new CustomWidget());
    widget->resize(240, 120);
    widget->show();
    return app.exec();
}
```

Custom Widgets

When you work on user interfaces, you will need to create custom made widgets. Typically a widget is a window area filled with painting calls. Additional the widget has internal knowledge of how to handle keyboard or mouse input and how to react to external triggers. To do this in Qt we need to derive from *QWidget* and overwrite several functions for painting and event handling.

```
#ifndef CUSTOMWIDGET_H
#define CUSTOMWIDGET_H

#include <QtWidgets>

class CustomWidget : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
private:
    QPoint m_lastPos;
};

#endif // CUSTOMWIDGET_H
```

In the implementation, we draw a small border on our widget and a small rectangle on the last mouse position. This is very typical for a low-level custom widget. Mouse or keyboard events change the internal state of the widget and trigger a painting update. We don't want to go into too much detail into this code, but it is good to know that you have the ability. Qt comes with a large set of ready-made desktop widgets, so that the probability is high that you don't have to do this.

```
#include "customwidget.h"

CustomWidget::CustomWidget(QWidget *parent) :
    QWidget(parent)
{
}

void CustomWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QRect r1 = rect().adjusted(10,10,-10,-10);
    painter.setPen(QColor("#33B5E5"));
    painter.drawRect(r1);

    QRect r2(QPoint(0,0), QSize(40,40));
    if(m_lastPos.isNull()) {
        r2.moveCenter(r1.center());
    } else {
        r2.moveCenter(m_lastPos);
    }
    painter.fillRect(r2, QColor("#FFBB33"));
}

void CustomWidget::mousePressEvent(QMouseEvent *event)
{
    m_lastPos = event->pos();
    update();
}

void CustomWidget::mouseMoveEvent(QMouseEvent *event)
```

```
{
    m_lastPos = event->pos();
    update();
}
```

Desktop Widgets

The Qt developers have done all of this for you already and provide a set of desktop widgets, which will look native on different operating systems. Your job is then to arrange these different widgets in a widget container into larger panels. A widget in Qt can also be a container for other widgets. This is accomplished by the parent-child relationship. This means we need to make our ready-made widgets like buttons, check boxes, radio button but also lists and grids a child of another widget. One way to accomplish this is displayed below.

Here is the header file for a so called widget container.

```
class CustomWidget : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidget(QWidget *parent = 0);
private slots:
    void itemClicked(QListWidgetItem* item);
    void updateItem();
private:
    QListWidget *m_widget;
    QLineEdit *m_edit;
    QPushButton *m_button;
};
```

In the implementation, we use layouts to better arrange our widgets. Layout managers re-layout the widgets according to some size policies when the container widget is re-sized. In this example we have a list, a line edit, and a button arranged vertically to allow to edit a list of cities. We use Qt's signal and slots to connect sender and receiver objects.

```
CustomWidget::CustomWidget(QWidget *parent) :
    QWidget(parent)
{
    QVBoxLayout *layout = new QVBoxLayout(this);
    m_widget = new QListWidget(this);
    layout->addWidget(m_widget);

    m_edit = new QLineEdit(this);
    layout->addWidget(m_edit);

    m_button = new QPushButton("Quit", this);
    layout->addWidget(m_button);
    setLayout(layout);

    QStringList cities;
    cities << "Paris" << "London" << "Munich";
    foreach(const QString& city, cities) {
        m_widget->addItem(city);
    }

    connect(m_widget, SIGNAL(itemClicked(QListWidgetItem*)), this,
    ↪SLOT(itemClicked(QListWidgetItem*)));
    connect(m_edit, SIGNAL(editingFinished()), this, SLOT(updateItem()));
    connect(m_button, SIGNAL(clicked()), qApp, SLOT(quit()));
}
```

```
void CustomWidget::itemClicked(QListWidgetItem *item)
{
    Q_ASSERT(item);
    m_edit->setText(item->text());
}

void CustomWidget::updateItem()
{
    QListWidgetItem* item = m_widget->currentItem();
    if(item) {
        item->setText(m_edit->text());
    }
}
```

Drawing Shapes

Some problems are better visualized. If the problem at hand looks faintly like geometrical objects, qt graphics view is a good candidate. A graphics view arranges simple geometrical shapes on a scene. The user can interact with these shapes or they are positioned using an algorithm. To populate a graphics view you need a graphics view and a graphics scene. The scene is attached to the view and populates with graphics items. Here is a short example. First the header file with the declaration of the view and scene.

```
class CustomWidgetV2 : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidgetV2(QWidget *parent = 0);
private:
    QGraphicsView *m_view;
    QGraphicsScene *m_scene;
};
```

In the implementation the scene gets attached to the view first. The view is a widget and get arranged in our container widget. At the end we add a small rectangle to the scene, which then is rendered on the view.

```
#include "customwidgetv2.h"

CustomWidget::CustomWidget(QWidget *parent) :
    QWidget(parent)
{
    m_view = new QGraphicsView(this);
    m_scene = new QGraphicsScene(this);
    m_view->setScene(m_scene);

    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->setMargin(0);
    layout->addWidget(m_view);
    setLayout(layout);

    QGraphicsItem* rect1 = m_scene->addRect(0,0, 40, 40, Qt::NoPen, QColor("#FFBB33
↪"));
    rect1->setFlags(QGraphicsItem::ItemIsFocusable|QGraphicsItem::ItemIsMovable);
}
```

Adapting Data

Up to now we have mostly covered basic data types and how to use widgets and graphic views. Often in your application you will need larger amount of structured data, which also has to be persistently stored. The data also

needs to be displayed. For this Qt uses models. A simple model is the string list model, which gets filled with strings and then attached to a list view.

```
m_view = new QListView(this);
m_model = new QStringListModel(this);
view->setModel(m_model);

QList<QString> cities;
cities << "Munich" << "Paris" << "London";
model->setStringList(cities);
```

Another popular way to store or retrieve data is SQL. Qt comes with SQLite embedded and also has support for other database engines (MySQL, PostgreSQL, ...). First you need to create your database using a schema, like this:

```
CREATE TABLE city (name TEXT, country TEXT);
INSERT INTO city value ("Munich", "Germany");
INSERT INTO city value ("Paris", "France");
INSERT INTO city value ("London", "United Kingdom");
```

To use sql we need to add the sql module to our .pro file

```
QT += sql
```

And then we can open our database using C++. First we need to retrieve a new database object for the specified database engine. With this database object we open the database. For SQLite it's enough to specify the path to the database file. Qt provides some high-level database model, one of them is the table model, which uses a table identifier and an option where clause to select the data. The resulting model can be attached to a list view as the other model before.

```
QSqlDatabase db = QSqlDatabase::addDatabase("SQLITE");
db.setDatabaseName('cities.db');
if(!db.open()) {
    qFatal("unable to open database");
}

m_model = QSqlTableModel(this);
m_model->setTable("city");
m_model->setHeaderData(0, Qt::Horizontal, "City");
m_model->setHeaderData(1, Qt::Horizontal, "Country");

view->setModel(m_model);
m_model->select();
```

For higher level of model operations Qt provides a sort filter proxy model, which allows you in the basic form to sort and filter another model.

```
QSortFilterProxyModel* proxy = new QSortFilterProxyModel(this);
proxy->setSourceModel(m_model);
view->setModel(proxy);
view->setSortingEnabled(true);
```

Filtering is done based on the column to be filters and a string as filter argument.

```
proxy->setFilterKeyColumn(0);
proxy->setFilterCaseSensitive(Qt::CaseInsensitive);
proxy->setFilterFixedString(QString)
```

The filter proxy model is much more powerful than demonstrated here. For now it is enough to remember its exists.

Note: This was an overview of the different kind of classical application you could develop with Qt 5. The

desktop is moving and soon the mobile devices will be our desktop of tomorrow. Mobile devices have a different user interface design. They are much more simplistic than desktop applications. They do one thing and they do simply and focused. Animations are an important part of the experience. A user interface needs to feel alive and fluent. The traditional Qt technologies are not well suited for this market.

Coming next: Qt Quick for the rescue.

Qt Quick Application

There is an inherent conflict in modern software development. The user interface is moving much faster than our back-end services. In a traditional technology you develop the so called front-end at the same pace as the back-end. This results in conflicts when customers want to change the user interface during a project, or develop the idea of an user interface during the project. Agile projects, require agile methods.

Qt Quick provides a declarative environment where your user interface (the front-end) is declared like HTML and your back-end is in native C++ code. This allows you to get the best of both worlds.

This is a simple Qt Quick UI below

```
import QtQuick 2.5

Rectangle {
    width: 240; height: 1230
    Rectangle {
        width: 40; height: 40
        anchors.centerIn: parent
        color: '#FFBB33'
    }
}
```

The declaration language is called QML and it needs a runtime to execute in. Qt provides a standard runtime called `qmlscene` but it's also not so difficult to write a custom runtime. For this we need a quick view and set the main QML document as source. The only thing left is to show the user interface.

```
QQuickView* view = new QQuickView();
QUrl source = QUrl::fromLocalFile("main.qml");
view->setSource(source);
view.show();
```

Coming back to our earlier examples. In one example we used a C++ city model. It would be great if we could use this model inside our declarative QML code.

To enable this, we first code our front-end to see how we would want to use a city model. In this case the front-end expects a object named `cityModel` which we can use inside a list view.

```
import QtQuick 2.5

Rectangle {
    width: 240; height: 120
    ListView {
        width: 180; height: 120
        anchors.centerIn: parent
        model: cityModel
        delegate: Text { text: model.city }
    }
}
```

To enable the `cityModel` we can mostly re-use our previous model and add a context property to our root context (the root context is the other root-element in the main document)

```
m_model = QSqlTableModel(this);  
... // some magic code  
QHash<int, QByteArray> roles;  
roles[Qt::UserRole+1] = "city";  
roles[Qt::UserRole+2] = "country";  
m_model->setRoleNames(roles);  
view->rootContext()->setContextProperty("cityModel", m_model);
```

Hint: This is not completely correct, as the SQL table model contains the data in columns and a QML model expects the data as roles. So there needs to be a mapping between columns and roles. Please see [QML](#) and [QSqlTableModel](#) wiki page.

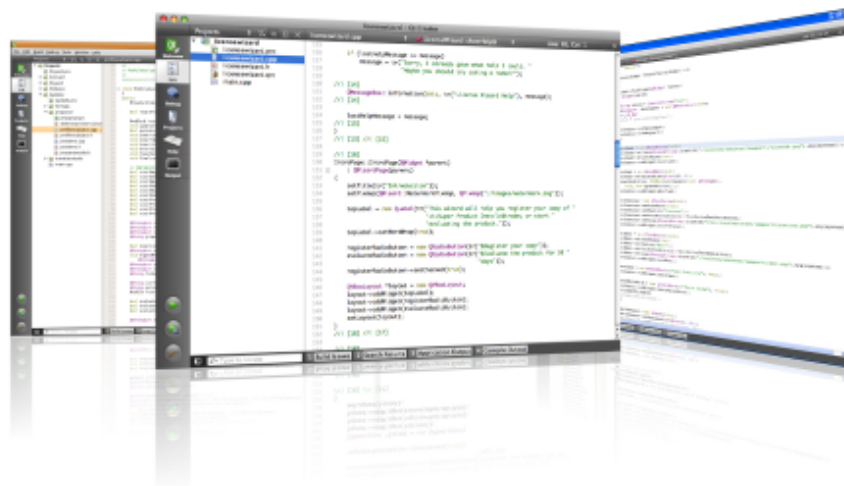
Summary

We have seen how to install the Qt SDK and how to create our first application. Then we walked you through the different application types to give you an overview of Qt, showing off some features Qt offers for application development. I hope you got a good impression that Qt is a very rich user interface toolkit and offers everything an application developer can hope for and more. Still, Qt does not lock you into specific libraries, as you always can use other libraries or extend Qt yourself. It is also rich when it comes to supporting different application models: console, classical desktop user interface and touch user interface.

Qt Creator IDE

Section author: jryannel

Qt Creator is the default integrated development environment for Qt. It's written from Qt developers for Qt developers. The IDE is available on all major desktop platforms, e.g. Windows/Mac/Linux. We have already seen customers using Qt Creator on an embedded device. Qt Creator has a lean efficient user interface and it really shines in making the developer productive. Qt Creator can be used to run your Qt Quick user interface but also to compile c++ code and this for your host system or for another device using a cross-compiler.



Note: The source code of this chapter can be found in the assets folder.

The User Interface

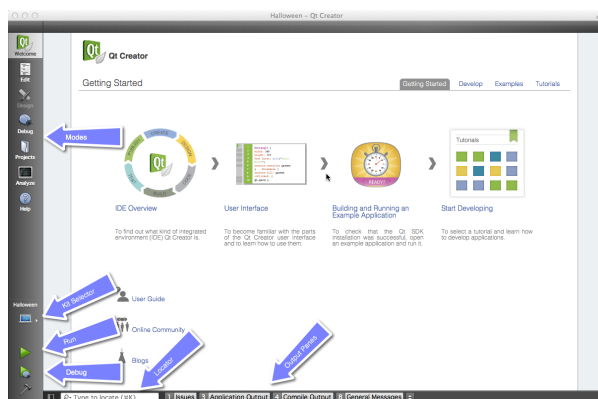
When starting Qt Creator you are greeted by the *Welcome* screen. There you will find the most important hints on how to continue inside Qt Creator and your recently used projects. You will also see the sessions list, which might be empty for you. A session is a collection of projects stored for your reference. This comes really handy when you have several customers with larger projects.

On the left side you will see the mode-selector. The mode selectors contain typical steps from your work flow.

- Welcome mode: For your orientation.
- Edit mode: Focus on the code
- Design mode: Focus on the UI design

- Debug mode: Retrieve information about a running application
- Projects mode: Modify your projects run and build configuration
- Analyze mode: For detecting memory leaks and profiling
- Help mode: Easy access to the Qt documentation

Below the mode-selectors you will find the actual project-configuration selector and the run/debug



Most of the time you will be in the edit mode with the code-editor in the central panel. From time to time, you will visit the Projects mode when you need to configure your project. And then you press Run. Qt Creator is smart enough to ensure your project is fully built before running it.

In the bottom are the output panes for issues, application messages, compile messages, and other messages.

Registering your Qt Kit

The Qt Kit is probably the most difficult aspect when it comes for working with Qt Creator initially. A Qt Kit is a set of a Qt version, compiler and device and some other settings. It is used to uniquely identify the combination of tools for your project build. A typical kit for the desktop would contain a GCC compiler and a Qt version (e.g. Qt 5.1.1) and a device (“Desktop”). After you have created a project you need to assign a kit to a project before qt creator can build the project. Before you are able to create a kit first you need to have a compiler installed and have a Qt version registered. A Qt version is registered by specifying the path to the `qmake` executable. Qt Creator then queries `qmake` for information required to identify the Qt version.

Adding a kit and registering a Qt version is done in the *Settings* → *Build & Run* entry. There you can also see which compilers are registered.

Note: Please first check if your Qt Creator has already the correct Qt version registered and then ensure a Kit for your combination of compiler and Qt and device is specified. You can not build a project without a kit.

Managing Projects

Qt Creator manages your source code in projects. You can create a new project by using *File* → *New File or Project*. When you create a project you have many choices of application templates. Qt Creator is capable of creating desktop, mobile applications. Application which use Widgets or Qt Quick or Qt Quick and controls or even bare-bone projects. Also project for HTML5 and python are supported. For a beginner it is difficult to choose, so we pick three project types for you.

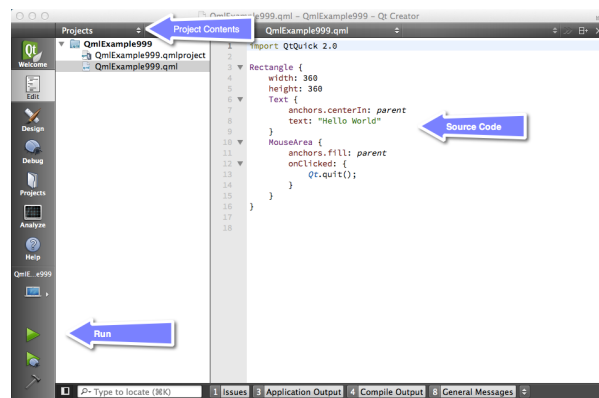
- **Applications / Qt Quick 2.0 UI:** This will create a QML/JS only project for you, without any C++ code. Take this if you want to sketch a new user interface or plan to create a modern UI application where the native parts are delivered by plug-ins.

- **Libraries / Qt Quick 2.0 Extension Plug-in:** Use this wizard to create a stub for a plug-in for your Qt Quick UI. A plug-in is used to extend Qt Quick with native elements.
- **Other Project / Empty Qt Project:** A bare-bone empty project. Take this if you want to code your application with c++ from scratch. Be aware you need to know what you are doing here.

Note: During the first parts of the book we will mainly use the Qt Quick 2.0 UI project type. Later to describe some c++ aspects we will use the Empty-Qt-Project type or something similar. For extending Qt Quick with our own native plug-ins we will use the *Qt Quick 2.0 Extension Plug-in* wizard type.

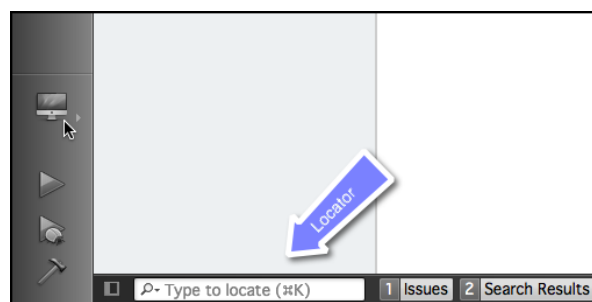
Using the Editor

When you open a project or you just created a new project Qt Creator will switch to the edit mode. You should see on the left your project files and in the center area the code editor. Selecting files on the left will open them in the editor. The editor provides syntax highlighting, code-completion and quick-fixes. Also it supports several commands for code refactoring. When working with the editor you will have the feeling that everything reacts immediately. This is thanks to the developers of Qt Creator which made the tool feel really snappy.



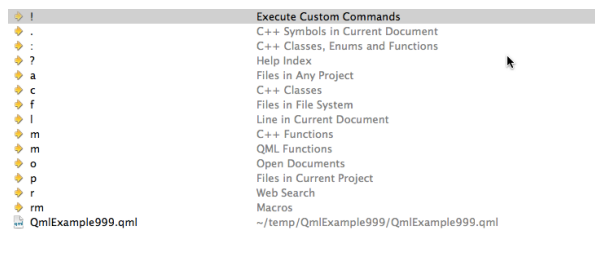
Locator

The locator is a central component inside Qt Creator. It allows developers to navigate fast to specific locations inside the source code or inside the help. To open the locator press `Ctrl+K`.



A pop-up is coming from the bottom left and shows a list of options. If you just search a file inside your project just hit the first letter from the file name. The locator also accepts wild-cards, so `*main.qml` will also work. Otherwise you can also prefix your search to search for specific content type.

Please try it out. For example to open the help for the QML element Rectangle open the locator and type `? rectangle`. While you type the locator will update the suggestions until you found the reference you are looking for.



Debugging

Qt Creator comes with C++ and QML debugging support.

Note: Hmm, I just realized I have not used debugging a lot. I hope this is a good sign. Need to ask someone to help me out here. In the meantime have a look at the [Qt Creator documentation](#).

Shortcuts

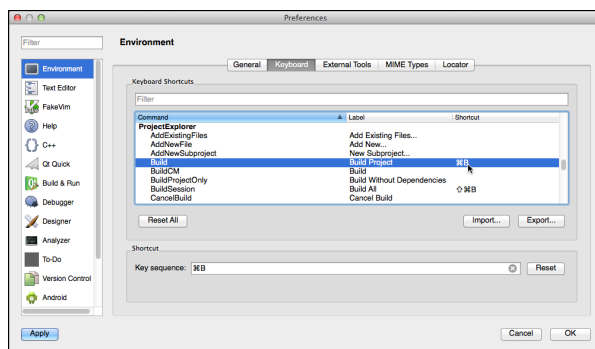
Shortcuts are the difference between a nice to use system and a professional system. As a professional you spend hundred of hours in front of your application. Each shortcut which makes your work-flow faster counts. Luckily the developers of Qt Creator think the same and have added literally hundreds of shortcuts to the application.

To get started we have collection some basic shortcuts (in Windows notation):

- `Ctrl+B` - Build project
- `Ctrl+R` - Run Project
- `Ctrl+Tab` - Switch between open documents
- `Ctrl+K` - Open Locator
- `Esc` - Go back (hit several times and you are back in the editor)
- `F2` - Follow Symbol under cursor
- `F4` - Switch between header and source (only useful for c++ code)

List of [Qt Creator shortcuts](#) from the documentation.

Note: You can edit the shortcuts from inside creator using the settings dialog.



Quick Starter

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

This chapter provides an overview of QML, the declarative user interface language used in Qt 5. We will discuss the QML syntax, which is a tree of elements, followed by an overview of the most important basic elements. Later we will briefly look at how to create our own elements, called components and how to transform elements using property manipulations. Towards the end we will look how to arrange elements together in a layout and finally have a look at elements where the user can provide input.

QML Syntax

QML is a declarative language used to describe the user interface of your application. It breaks down the user interface into smaller elements, which can be combined to components. QML describes the look and the behavior of these user interface elements. This user interface description can be enriched with JavaScript code to provide simple but also more complex logic. In this perspective it follows the HTML-JavaScript pattern but QML is designed from the ground up to describe user interfaces not text-documents.

In its simplest way QML is a hierarchy of elements. Child elements inherit the coordinate system from the parent. An x, y coordinate is always relative to the parent.

Let's start with a simple example of a QML file to explain the different syntax.

```
// RectangleExample.qml

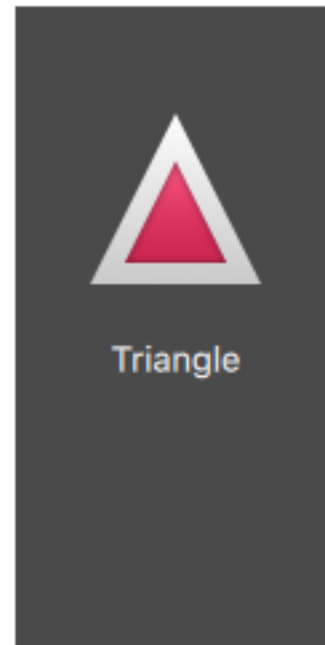
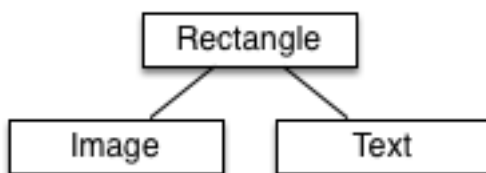
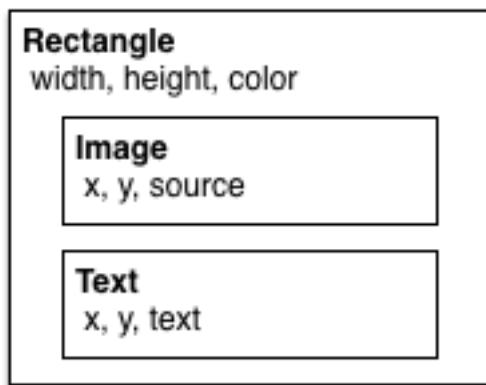
import QtQuick 2.5

// The root element is the Rectangle
Rectangle {
    // name this element root
    id: root

    // properties: <name>: <value>
    width: 120; height: 240

    // color property
    color: "#4A4A4A"

    // Declare a nested element (child of root)
    Image {
        id: triangle
```



```

// reference the parent
x: (parent.width - width)/2; y: 40

source: 'assets/triangle_red.png'
}

// Another child of root
Text {
    // un-named element

    // reference element by id
    y: triangle.y + triangle.height + 20

    // reference root element
    width: root.width

    color: 'white'
    horizontalAlignment: Text.AlignHCenter
    text: 'Triangle'
}
}

```

- The `import` statement imports a module in a specific version. In general you always want to import *QtQuick 2.0* as your initial set of elements
- Comments can be made using `//` for single line comments or `/* */` for multi-line comments. Just like in C/C++ and JavaScript
- Every QML file needs to have exactly one root element, like HTML
- An element is declared by its type followed by `{ }`
- Elements can have properties, they are in the form `name : value`
- Arbitrary elements inside a QML document can be accessed by using their `id` (an unquoted identifier)
- Elements can be nested, meaning a parent element can have child elements. The parent element can be accessed using the `parent` keyword

Tip: Often you want to access a particular element by id or a parent element using the `parent` keyword. So it's good practice to name your root element "root" using `id: root`. Then you don't have to think about how the root element is named in your QML document.

Hint: You can run the example using the Qt Quick runtime from the command line from your OS like this:

```
$ $QTDIR/bin/qmlscene RectangleExample.qml
```

Where you need to replace the *\$QTDIR* to the path to your Qt installation. The *qmlscene* executable initializes the Qt Quick runtime and interprets the provided QML file.

In Qt Creator you can open the corresponding project file and run the document `RectangleExample.qml`.

Properties

Elements are declared by using their element name but are defined by using their properties or by creating custom properties. A property is a simple key-value pair, e.g. `width : 100`, `text: 'Greetings'`, `color: '#FF0000'`. A property has a well-defined type and can have an initial value.

```
Text {
    // (1) identifier
    id: thisLabel

    // (2) set x- and y-position
    x: 24; y: 16

    // (3) bind height to 2 * width
    height: 2 * width

    // (4) custom property
    property int times: 24

    // (5) property alias
    property alias anotherTimes: thisLabel.times

    // (6) set text appended by value
    text: "Greetings " + times

    // (7) font is a grouped property
    font.family: "Ubuntu"
    font.pixelSize: 24

    // (8) KeyNavigation is an attached property
    KeyNavigation.tab: otherLabel

    // (9) signal handler for property changes
    onHeightChanged: console.log('height:', height)

    // focus is need to receive key events
    focus: true

    // change color based on focus value
    color: focus?"red":"black"
}
```

Let's go through the different features of properties:

1. `id` is a very special property-like value, it is used to reference elements inside a QML file (called “document” in QML). The `id` is not a string type but rather an identifier and part of the QML syntax. An `id` needs to be unique inside a document and it can’t be re-set to a different value, nor may it be queried. (It behaves more like a pointer in the C++ world.)
2. A property can be set to a value, depending on its type. If no value is given for a property, an initial value will be chosen. You need to consult the documentation of the particular element for more information about the initial value of a property.
3. A property can depend on one or many other properties. This is called *binding*. A bound property is updated, when its dependent properties change. It works like a contract, in this case the `height` should always be two times the `width`.
4. Adding own properties to an element is done using the `property` qualifier followed by the type, the name and the optional initial value (`property <type> <name> : <value>`). If no initial value is given a system initial value is chosen.

Note: You can also declare one property to be the default property if no property name is given by prepending the property declaration with the `default` keyword. This is used for example when you add child elements, the child elements are added automatically to the default property `children` of type `list` if they are visible elements.

5. Another important way of declaring properties is using the `alias` keyword (`property alias <name> : <reference>`). The `alias` keyword allows us to forward a property of an object or an object itself from within the type to an outer scope. We will use this technique later when defining components to export the inner properties or element ids to the root level. A property alias does not need a type, it uses the type of the referenced property or object.
6. The `text` property depends on the custom property `times` of type `int`. The `int` based value is automatically converted to a `string` type. The expression itself is another example of binding and results into the `text` being updated every time the `times` property changes.
7. Some properties are grouped properties. This feature is used when a property is more structured and related properties should be grouped together. Another way of writing grouped properties is `font { family: "Ubuntu"; pixelSize: 24 }`.
8. Some properties are attached to the element itself. This is done for global relevant elements which appear only once in the application (e.g. keyboard input). The writing is `<Element>.<property>: <value>`.
9. For every property you can provide a signal handler. This handler is called after the property changes. For example here we want to be notified whenever the `height` changes and use the built-in console to log a message to the system.

Warning: An element `id` should only be used to reference elements inside your document (e.g. the current file). QML provides a mechanism called dynamic-scoping where later loaded documents overwrite the element `id`’s from earlier loaded documents. This makes it possible to reference element `id`’s from earlier loaded documents, if they are not yet overwritten. It’s like creating global variables. Unfortunately this frequently leads to really bad code in practice, where the program depends on the order of execution. Unfortunately this can’t be turned off. Please only use this with care or even better don’t use this mechanism at all. It’s better to export the element you want to provide to the outside world using properties on the root element of your document.

Scripting

QML and JavaScript (also known as ECMAScript) are best friends. In the *JavaScript* chapter we will go into more detail on this symbiosis. Currently we just want to make you aware about this relationship.


```

Text {
    id: label

    x: 24; y: 24

    // custom counter property for space presses
    property int spacePresses: 0

    text: "Space pressed: " + spacePresses + " times"

    // (1) handler for text changes
    onTextChanged: console.log("text changed to:", text)

    // need focus to receive key events
    focus: true

    // (2) handler with some JS
    Keys.onSpacePressed: {
        increment()
    }

    // clear the text on escape
    Keys.onEscapePressed: {
        label.text = ''
    }

    // (3) a JS function
    function increment() {
        spacePresses = spacePresses + 1
    }
}

```

1. The text changed handler `onTextChanged` prints the current text every-time the text changed due to a space-bar key pressed
2. When the text element receives the space-bar key (because the user pressed the space-bar on the keyboard) we call a JavaScript function `increment()`.
3. Definition of a JavaScript function in the form of `function <name>(<parameters>) { ... }`, which increments our counter `spacePressed`. Every time `spacePressed` is incremented, bound properties will also be updated.

Note: The difference between the QML : (binding) and the JavaScript = (assignment) is, that the binding is a contract and keeps true over the lifetime of the binding, whereas the JavaScript assignment (=) is a one time value assignment. The lifetime of a binding ends, when a new binding is set to the property or even when a JavaScript value is assigned to the property. For example a key handler setting the text property to an empty string would destroy our increment display:

```

Keys.onEscapePressed: {
    label.text = ''
}

```

After pressing escape, pressing the space-bar will not update the display anymore as the previous binding of the text property (`text: "Space pressed: " + spacePresses + " times"`) was destroyed.

When you have conflicting strategies to change a property as in this case (text updated by a change to a property increment via a binding and text cleared by a JavaScript assignment) then you can't use bindings! You need to use assignment on both property change paths as the binding will be destroyed by the assignment (broken contract!).

Basic Elements

Elements can be grouped into visual and non-visual elements. A visual element (like the `Rectangle`) has a geometry and normally presents an area on the screen. A non-visual element (like a `Timer`) provides general functionality, normally used to manipulate the visual elements.

Currently, we will focus on the fundamental visual elements, such as `Item`, `Rectangle`, `Text`, `Image` and `MouseArea`.

Item Element

`Item` is the base element for all visual elements as such all other visual elements inherit from `Item`. It doesn't paint anything by itself but defines all properties which are common across all visual elements:

Group	Properties
Geometry	<code>x</code> and <code>y</code> to define the top-left position, <code>width</code> and <code>height</code> for the expand of the element and also the <code>z</code> stacking order to lift elements up or down from their natural ordering
Layout handling	<code>anchors</code> (<code>left</code> , <code>right</code> , <code>top</code> , <code>bottom</code> , <code>vertical</code> and <code>horizontal</code> center) to position elements relative to other elements with their margins
Key handling	attached <code>Key</code> and <code>KeyNavigation</code> properties to control key handling and the <code>input focus</code> property to enable key handling in the first place
Transformation	<code>scale</code> and <code>rotate</code> transformation and the generic <code>transform</code> property list for <code>x,y,z</code> transformation and their <code>transformOrigin</code> point
Visual	<code>opacity</code> to control transparency, <code>visible</code> to show/hide elements, <code>clip</code> to restrain paint operations to the element boundary and <code>smooth</code> to enhance the rendering quality
State definition	<code>states</code> list property with the supported list of states and the <code>current state</code> property as also the <code>transitions</code> list property to animate state changes.

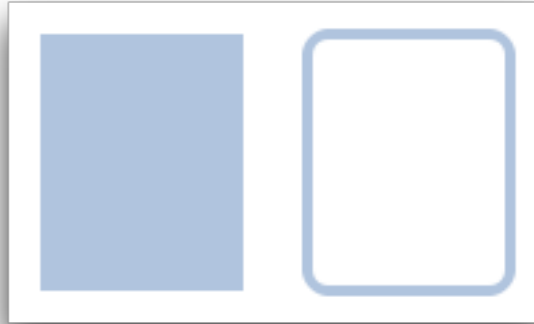
To better understand the different properties we will try to introduce them throughout this chapter in context of the element presented. Please remember these fundamental properties are available on every visual element and work the same across these elements.

Note: The `Item` element is often used as a container for other elements, similar to the `div` element in HTML.

Rectangle Element

The `Rectangle` extends `Item` and adds a fill color to it. Additionally it supports borders defined by `border.color` and `border.width`. To create rounded rectangles you can use the `radius` property.

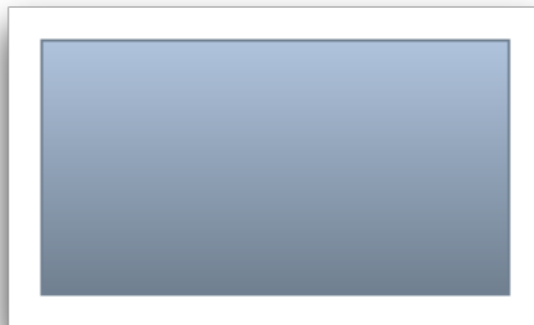
```
Rectangle {
    id: rect1
    x: 12; y: 12
    width: 76; height: 96
    color: "lightsteelblue"
}
Rectangle {
    id: rect2
    x: 112; y: 12
    width: 76; height: 96
    border.color: "lightsteelblue"
    border.width: 4
    radius: 8
}
```



Note: Valid colors values are colors from the SVG color names (see <http://www.w3.org/TR/css3-color/#svg-color>). You can provide colors in QML in different ways, but the most common way is an RGB string ('#FF4444') or as a color name (e.g. 'white').

Besides a fill color and a border the rectangle also supports custom gradients.

```
Rectangle {
    id: rect1
    x: 12; y: 12
    width: 176; height: 96
    gradient: Gradient {
        GradientStop { position: 0.0; color: "lightsteelblue" }
        GradientStop { position: 1.0; color: "slategray" }
    }
    border.color: "slategray"
}
```



A gradient is defined by a series of gradient stops. Each stop has a position and a color. The position marks the position on the y-axis (0 = top, 1 = bottom). The color of the `GradientStop` marks the color at that position.

Note: A rectangle with no *width/height* set will not be visible. This happens often when you have several rectangles width (height) depending on each other and something went wrong in your composition logic. So watch out!

Note: It is not possible to create an angled gradient. For this it's better to use predefined images. One possibility would be to just rotate the rectangle with the gradient, but be aware the geometry of an rotated rectangle will not change and thus will lead to confusion as the geometry of the element is not the same as the visible area. From

the authors perspective it's really better to use designed gradient images in that case.

Text Element

To display text, you can use the `Text` element. Its most notable property is the `text` property of type `string`. The element calculates its initial width and height based on the given text and the font used. The font can be influenced using the font property group (e.g. `font.family`, `font.pixelSize`, ...). To change the color of the text just use the `color` property.

```
Text {
    text: "The quick brown fox"
    color: "#303030"
    font.family: "Ubuntu"
    font.pixelSize: 28
}
```



Text can be aligned to each side and the center using the `horizontalAlignment` and `verticalAlignment` properties. To further enhance the text rendering you can use the `style` and `styleColor` property, which allows you render the text in outline, raised and sunken mode. For longer text you often want to define a *break* position like *A very ... long text*, this can be achieved using the `elide` property. The `elide` property allows you to set the elide position to the left, right or middle of your text. In case you don't want the '...' of the elide mode to appear but still want to see the full text you can also wrap the text using the `wrapMode` property (works only when width is explicitly set):

```
Text {
    width: 40; height: 120
    text: 'A very long text'
    // '...' shall appear in the middle
    elide: Text.ElideMiddle
    // red sunken text styling
    style: Text.Sunken
    styleColor: '#FF4444'
    // align text to the top
    verticalAlignment: Text.AlignTop
    // only sensible when no elide mode
    // wrapMode: Text.WordWrap
}
```

A `Text` element only displays the given text. It does not render any background decoration. Besides the rendered text the `Text` element is transparent. It's part of your overall design to provide a sensible background to the text element.

Note: Be aware a `Text` initial width (height) is depending on the text string and on the font set. A `Text` element with no width set and no text will not be visible, as the initial width will be 0.

Note: Often when you want to layout `Text` elements you need to differentiate between aligning the text inside the `Text` element boundary box or to align the element boundary box itself. In the former you want to use the `horizontalAlignment` and `verticalAlignment` properties and in the later case you want to manipulate the element geometry or use anchors.

Image Element

An `Image` element is able to display images in various formats (e.g. PNG, JPG, GIF, BMP, WEBP). *For the full list of supported image formats, please consult the Qt documentation.* Besides the obvious `source` property to provide the image URL it contains a `fillMode` which controls the resizing behavior.

```
Image {
    x: 12; y: 12
    // width: 72
    // height: 72
    source: "assets/triangle_red.png"
}
Image {
    x: 12+64+12; y: 12
    // width: 72
    height: 72/2
    source: "assets/triangle_red.png"
    fillMode: Image.PreserveAspectCrop
    clip: true
}
```



Note: A URL can be a local path with forward slashes (`"/images/home.png"`) or a web-link (e.g. `"http://example.org/home.png"`).

Note: Image elements using `PreserveAspectCrop` should also enable the clipping to avoid image data being rendered outside the `Image` boundaries. By default clipping is disabled (`clip : false`). You need to enable clipping (`clip : true`) to constrain the painting to the elements bounding rectangle. This can be used on any visual element.

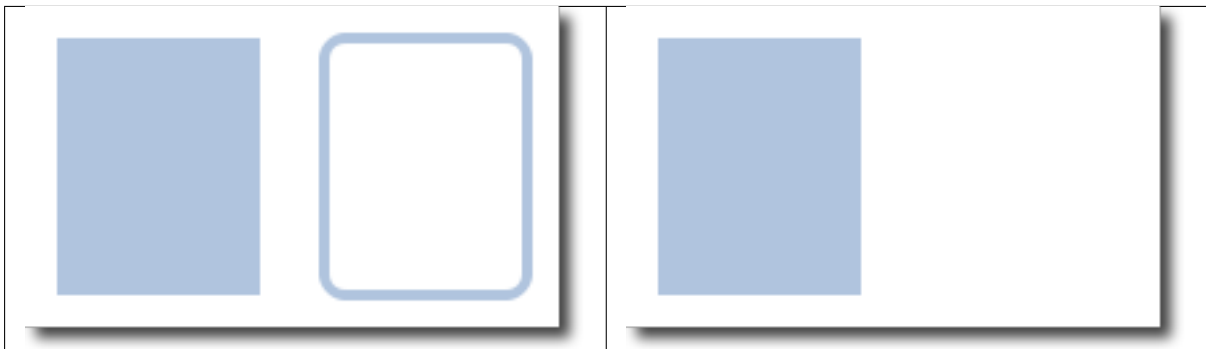
Tip: Using C++ you are able to create your own image provider using `QQmlImageProvider`. This allows you to create images on the fly and threaded image loading.

MouseArea Element

To interact with these elements you often will use a `MouseArea`. It's a rectangular invisible item in where you can capture mouse events. The mouse area is often used together with a visible item to execute commands when the user interacts with the visual part.

```
Rectangle {
    id: rect1
    x: 12; y: 12
    width: 76; height: 96
    color: "lightsteelblue"
    MouseArea {
        id: area
        width: parent.width
        height: parent.height
        onClicked: rect2.visible = !rect2.visible
    }
}

Rectangle {
    id: rect2
    x: 112; y: 12
    width: 76; height: 96
    border.color: "lightsteelblue"
    border.width: 4
    radius: 8
}
```



Note: This is an important aspect of Qt Quick, the input handling is separated from the visual presentation. By this it allows you to show the user an interface element, but the interaction area can be larger.

Components

A component is a reusable element and QML provides different ways to create components. Currently we will look only at the simplest form - a file based component. A file based component is created by placing a QML element in a file and give the file an element name (e.g. `Button.qml`). You can use the component like every other element from the QtQuick module, in our case you would use this in your code as `Button { ... }`.

For example, let's create a rectangle containing a text component and a mouse area. This resembles a simple button and doesn't need to be more complicated for our purposes.

```
Rectangle { // our inlined button ui
    id: button
    x: 12; y: 12
    width: 116; height: 26
    color: "lightsteelblue"
    border.color: "slategrey"
```

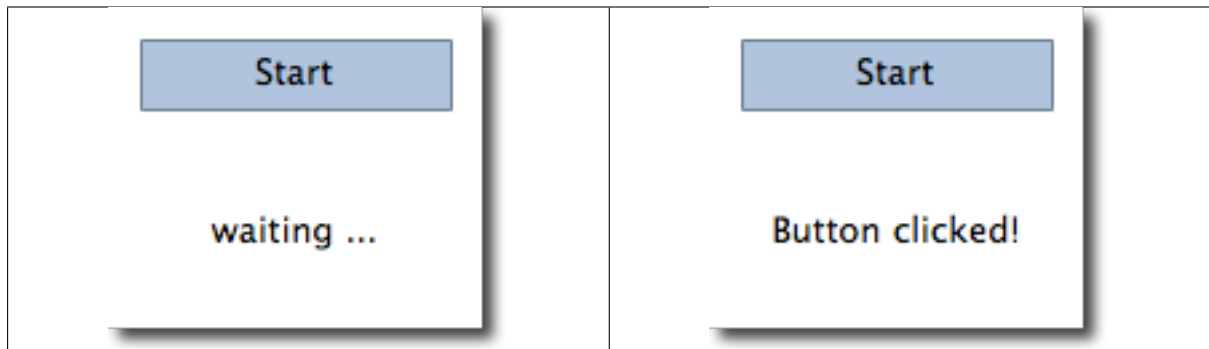
```

    Text {
        anchors.centerIn: parent
        text: "Start"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            status.text = "Button clicked!"
        }
    }
}

Text { // text changes when button was clicked
    id: status
    x: 12; y: 76
    width: 116; height: 26
    text: "waiting ..."
    horizontalAlignment: Text.AlignHCenter
}

```

The UI will look similar to this. On the left the UI in the initial state, on the right after the button has been clicked.



Our task is now to extract the button UI in a reusable component. For this we shortly think about a possible API for our button. You can do this by imagining how someone else should use your button. Here's what I came up with:

```

// minimal API for a button
Button {
    text: "Click Me"
    onClicked: { // do something }
}

```

I would like to set the text using a `text` property and to implement my own click handler. Also I would expect the button to have a sensible initial size, which I can overwrite (e.g. with `width: 240` for example).

To achieve this we create a `Button.qml` file and copy our button UI inside. Additionally we need to export the properties a user might want to change on the root level.

```

// Button.qml
import QtQuick 2.5

Rectangle {
    id: root
    // export button properties
    property alias text: label.text
    signal clicked

    width: 116; height: 26
    color: "lightsteelblue"
}

```

```
border.color: "slategrey"

Text {
    id: label
    anchors.centerIn: parent
    text: "Start"
}
MouseArea {
    anchors.fill: parent
    onClicked: {
        root.clicked()
    }
}
```

We have exported the text and clicked signal on the root level. Typically we name our root element root to make the referencing easier. We use the `alias` feature of QML, which is a way to export properties inside nested QML elements to the root level and make this available for the outside world. It is important to know, that only the root level properties can be accessed from outside this file by other components.

To use our new `Button` element we can simply declare it in our file. So the earlier example will become a little bit simplified.

```
Button { // our Button component
    id: button
    x: 12; y: 12
    text: "Start"
    onClicked: {
        status.text = "Button clicked!"
    }
}

Text { // text changes when button was clicked
    id: status
    x: 12; y: 76
    width: 116; height: 26
    text: "waiting ..."
    horizontalAlignment: Text.AlignHCenter
}
```

Now you can use as many buttons as you like in your UI by just using `Button { ... }`. A real button could be more complex, e.g providing feedback when clicked or showing a nicer decoration.

Note: Personally you could even go a step further and use an item as a root element. This prevents users to change the color of our designed button, and provides us more control about the exported API. The target should be to export a minimal API. Practically this means we would need to replace the root `Rectangle` with an `Item` and make the rectangle a nested element in the root item.

```
Item {
    id: root
    width: 116; height: 26

    property alias text: label.text
    signal clicked

    Rectangle {
```



```

        anchors.fill parent
        color: "lightsteelblue"
        border.color: "slategrey"
    }
    ...
}

```

With this technique, it is easy to create a whole series of reusable components.

Simple Transformations

A transformation manipulates the geometry of an object. QML Items can in general be translated, rotated and scaled. There is a simple form of these operations and a more advanced way.

Let's start with the simple transformations. Here is our scene as our starting point.

A simple translation is done via changing the `x, y` position. A rotation is done using the `rotation` property. The value is provided in degrees (0 .. 360). A scaling is done using the `scale` property and a value `<1` means the element is scaled down and `>1` means the element is scaled up. The rotation and scaling does not change your geometry. The items `x, y` and `width/height` haven't changed. Just the painting instructions are transformed.

Before we show off the example I would like to introduce a little helper: The `ClickableImage` element. The `ClickableImage` is just an image with a mouse area. This brings up a useful rule of thumb - if you have copied a chunk of code three times, extract it into a component.

```

// ClickableImage.qml

// Simple image which can be clicked

import QtQuick 2.5

Image {
    id: root
    signal clicked

    MouseArea {
        anchors.fill: parent
        onClicked: root.clicked()
    }
}

```



We use our clickable image to present three objects (box, circle, triangle). Each object performs a simple transformation when clicked. Clicking the background will reset the scene.

```
// transformation.qml

import QtQuick 2.5

Item {
    // set width based on given background
    width: bg.width
    height: bg.height

    Image { // nice background image
        id: bg
        source: "assets/background.png"
    }

    MouseArea {
        id: backgroundClicker
        // needs to be before the images as order matters
        // otherwise this mousearea would be before the other elements
        // and consume the mouse events
        anchors.fill: parent
        onClicked: {
            // reset our little scene
            circle.x = 84
            box.rotation = 0
            triangle.rotation = 0
            triangle.scale = 1.0
        }
    }

    ClickableImage {
        id: circle
        x: 84; y: 68
        source: "assets/circle_blue.png"
        antialiasing: true
        onClicked: {
            // increase the x-position on click
            x += 20
        }
    }

    ClickableImage {
        id: box
        x: 164; y: 68
        source: "assets/box_green.png"
        antialiasing: true
        onClicked: {
            // increase the rotation on click
            rotation += 15
        }
    }

    ClickableImage {
        id: triangle
        x: 248; y: 68
        source: "assets/triangle_red.png"
        antialiasing: true
        onClicked: {
            // several transformations
            rotation += 15
        }
    }
}
```

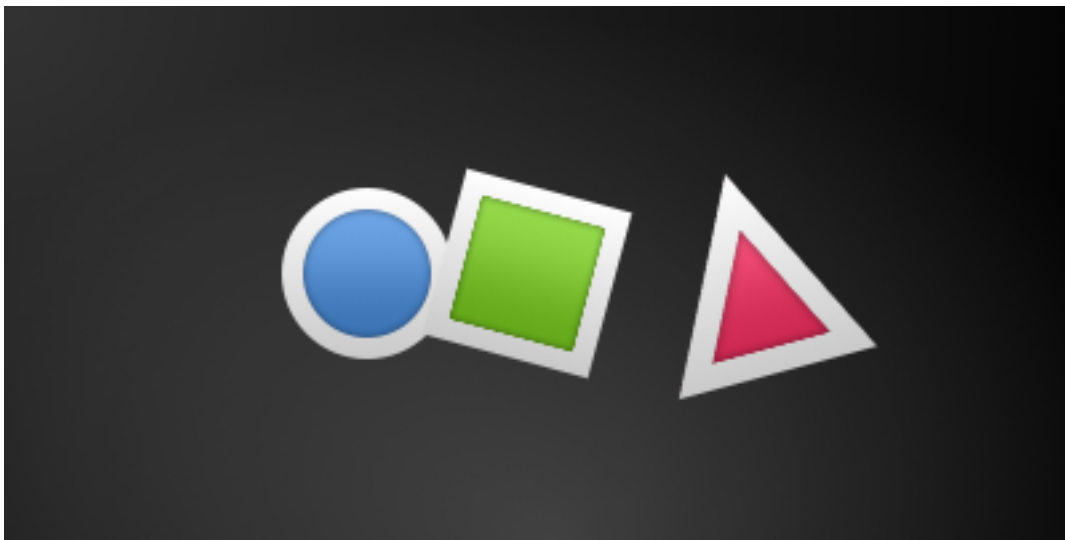
```

        scale += 0.05
    }

    function _test_transformed() {
        circle.x += 20
        box.rotation = 15
        triangle.scale = 1.2
        triangle.rotation = -15
    }

    function _test_overlap() {
        circle.x += 40
        box.rotation = 15
        triangle.scale = 2.0
        triangle.rotation = 45
    }
}

```



The circle increments the x-position on each click and the box will rotate on each click. The triangle will rotate and scale the image down on each click, to demonstrate a combined transformation. For the scaling and rotation operation we set `antialiasing: true` to enable anti-aliasing, which is switched off (same as the clipping property `clip`) for performance reasons. In your own work, when you see some rasterized edges in your graphics, then you should probably switch smooth on.

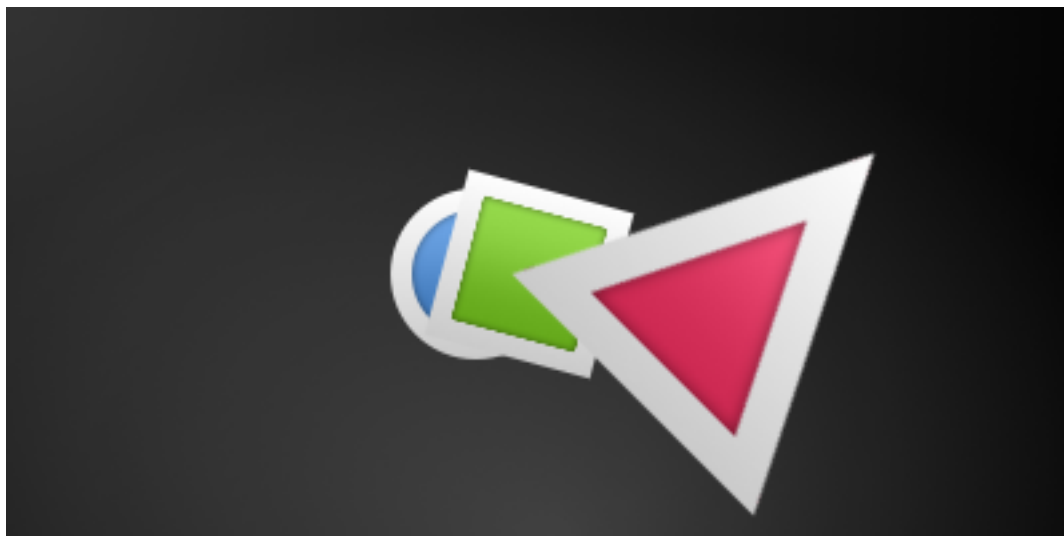
Note: To achieve better visual quality when scaling images it is recommended to scale images down instead of up. Scaling an image up with a larger scaling factor will result into scaling artifacts (blurred image). When scaling an image you should consider using `antialiasing : true` to enable the usage of a higher quality filter.

The background `MouseArea` covers the whole background and resets the object values.

Note: Elements which appear earlier in the code have a lower stacking order (called z-order). If you click long enough on `circle` you will see it moves below `box`. The z-order can also be manipulated by the `z`-property of an `Item`.

This is because `box` appears later in the code. The same applies also to mouse areas. A mouse area later in the code will overlap (and thus grab the mouse events) of a mouse area earlier in the code.

Please remember: *The order of elements in the document matters.*



Positioning Elements

There are a number of QML elements used to position items. These are called positioners and the following are provided in the QtQuick module `Row`, `Column`, `Grid` and `Flow`. They can be seen showing the same contents in the illustration below.

Note: Before we go into details, let me introduce some helper elements. The red, blue, green, lighter and darker squares. Each of these components contains a 48x48 pixels colored rectangle. As reference here is the source code for the RedSquare:

```
// RedSquare.qml

import QtQuick 2.5

Rectangle {
    width: 48
    height: 48
    color: "#ea7025"
    border.color: Qt.lighter(color)
}
```

Please note the use of `Qt.lighter(color)` to produce a lighter border color based on the fill color. We will use these helpers in the next examples to make the source code more compact and hopefully readable. Please remember, each rectangle is initial 48x48 pixels.

The `Column` element arranges child items into a column by stacking them on top of each other. The `spacing` property can be used to distance each of the child elements from each other.

```
// column.qml

import QtQuick 2.5

DarkSquare {
    id: root
    width: 120
    height: 240

    Column {
        id: row
        anchors.centerIn: parent
    }
}
```



```

    spacing: 8
    RedSquare { }
    GreenSquare { width: 96 }
    BlueSquare { }
}

```

The `Row` element places its child items next to each other, either from the left to the right, or from the right to the left, depending on the `layoutDirection` property. Again, `spacing` is used to separate child items.



```

// row.qml

import QtQuick 2.5

BrightSquare {
    id: root
    width: 400; height: 120

    Row {
        id: row
        anchors.centerIn: parent
        spacing: 20
        BlueSquare { }
        GreenSquare { }
    }
}

```

```
    RedSquare { }  
  }  
}
```

The `Grid` element arranges its children in a grid, by setting the `rows` and `columns` properties, the number or rows or columns can be constrained. By not setting either of them, the other is calculated from the number of child items. For instance, setting rows to 3 and adding 6 child items will result in 2 columns. The properties `flow` and `layoutDirection` are used to control the order in which the items are added to the grid, while `spacing` controls the amount of space separating the child items.



```
// grid.qml  
  
import QtQuick 2.5  
  
BrightSquare {  
    id: root  
    width: 160  
    height: 160  
  
    Grid {  
        id: grid  
        rows: 2  
        columns: 2  
        anchors.centerIn: parent  
        spacing: 8  
        RedSquare { }  
        RedSquare { }  
        RedSquare { }  
        RedSquare { }  
    }  
}
```

The final positioner is `Flow`. It adds its child items in a flow. The direction of the flow is controlled using `flow` and `layoutDirection`. It can run sideways or from the top to the bottom. It can also run from left to right or in the opposite direction. As the items are added in the flow, they are wrapped to form new rows or columns as needed. In order for a flow to work, it must have a width or a height. This can be set either directly, or through anchor layouts.

```
// flow.qml  
  
import QtQuick 2.5  
  
BrightSquare {
```



```

id: root
width: 160
height: 160

Flow {
    anchors.fill: parent
    anchors.margins: 20
    spacing: 20
    RedSquare { }
    BlueSquare { }
    GreenSquare { }
}

```

An element often used with positioners is the `Repeater`. It works like a for-loop and iterates over a model. In the simplest case a model is just a value providing the amount of loops.



```
// repeater.qml
```

```
import QtQuick 2.5

DarkSquare {
    id: root
    width: 252
    height: 252
    property variant colorArray: ["#00bde3", "#67c111", "#ea7025"]

    Grid{
        anchors.fill: parent
        anchors.margins: 8
        spacing: 4
        Repeater {
            model: 16
            Rectangle {
                width: 56; height: 56
                property int colorIndex: Math.floor(Math.random()*3)
                color: root.colorArray[colorIndex]
                border.color: Qt.lighter(color)
                Text {
                    anchors.centerIn: parent
                    color: "#f0f0f0"
                    text: "Cell " + index
                }
            }
        }
    }
}
```

In this repeater example, we use some new magic. We define our own color property, which we use as an array of colors. The repeater creates a series of rectangles (16, as defined by the model). For each loop he creates the rectangle as defined by the child of the repeater. In the rectangle we chose the color by using JS math functions `Math.floor(Math.random()*3)`. This gives us a random number in the range from 0..2, which we use to select the color from our color array. As noted earlier, JavaScript is a core part of Qt Quick, as such the standard libraries are available for us.

A repeater injects the `index` property into the repeater. It contains the current loop-index. (0,1,..15). We can use this to make our own decisions based on the index, or in our case to visualize the current index with the `Text` element.

Note: More advanced handling of larger models and kinetic views with dynamic delegates is covered in an own model-view chapter. Repeaters are best used when having a small amount of static data to be presented.

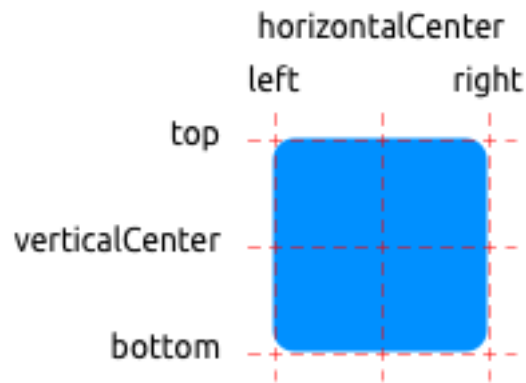
Layout Items

Todo

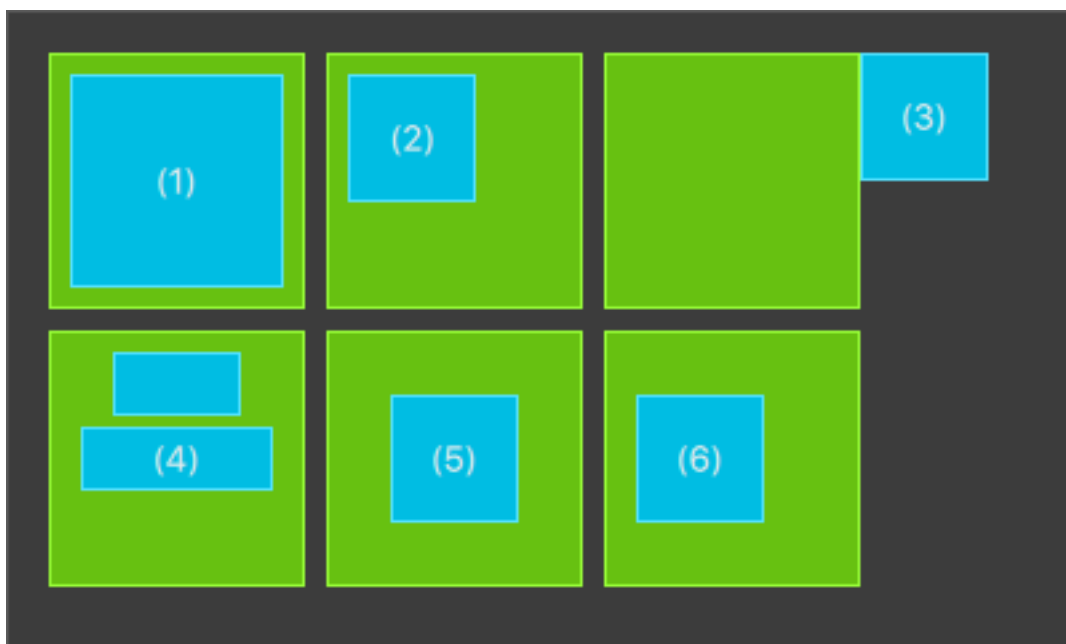
do we need to remove all uses of anchors earlier?

QML provides a flexible way to layout items using anchors. The concept of anchoring is part of the `Item` fundamental properties and available to all visual QML elements. An anchors acts like a contract and is stronger than competing geometry changes. Anchors are expressions of relativeness, you always need a related element to anchor with.

An element has 6 major anchor lines (top, bottom, left, right, horizontalCenter, verticalCenter). Additional there is the baseline anchor for text in `Text` elements. Each anchor line comes with an offset. In the case of top, bottom,



left and right they are called margins. For horizontalCenter, verticalCenter and baseline they are called offsets.



1. An element fills a parent element

```
GreenSquare {
    BlueSquare {
        width: 12
        anchors.fill: parent
        anchors.margins: 8
        text: '(1)'
    }
}
```

2. An element is left aligned to the parent

```
GreenSquare {
    BlueSquare {
        width: 48
        y: 8
        anchors.left: parent.left
        anchors.leftMargin: 8
        text: '(2)'
    }
}
```

```
}
```

3. An element left side is aligned to the parents right side

```
GreenSquare {
    BlueSquare {
        width: 48
        anchors.left: parent.right
        text: '(3)'
    }
}
```

4. Center aligned elements. Blue1 is horizontal centered on the parent. Blue2 is also horizontal centered but on Blue1 and it's top is aligned to the Blue1 bottom line.

```
GreenSquare {
    BlueSquare {
        id: blue1
        width: 48; height: 24
        y: 8
        anchors.horizontalCenter: parent.horizontalCenter
    }
    BlueSquare {
        id: blue2
        width: 72; height: 24
        anchors.top: blue1.bottom
        anchors.topMargin: 4
        anchors.horizontalCenter: blue1.horizontalCenter
        text: '(4)'
    }
}
```

5. An element is centered on a parent element

```
GreenSquare {
    BlueSquare {
        width: 48
        anchors.centerIn: parent
        text: '(5)'
    }
}
```

6. An element is centered with an left-offset on a parent element using horizontal and vertical center lines

```
GreenSquare {
    BlueSquare {
        width: 48
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.horizontalCenterOffset: -12
        anchors.verticalCenter: parent.verticalCenter
        text: '(6)'
    }
}
```

Note: Our squares have been enhanced to enable dragging. Try the example and drag around some squares. You will see that (1) can't be dragged as it's anchored on all sides, sure you can drag the parent of (1) as it's not anchored at all. (2) can be vertically dragged as only the left side is anchored. Similar applies to (3). (4) can only be dragged vertically as both squares are horizontal centered. (5) is centered on the parent and as such can't be dragged, similar applies to (7). Dragging an element means changing their *x*, *y* position. As anchoring is stronger than geometry changes such as *x*, *y*, dragging is restricted by the anchored lines. We will see this effect later

when we discuss animations.

Input Elements

We have already used the `MouseArea` as a mouse input element. Next, we'll focus on keyboard input. We start off with the text editing elements: `TextInput` and `TextEdit`.

TextInput

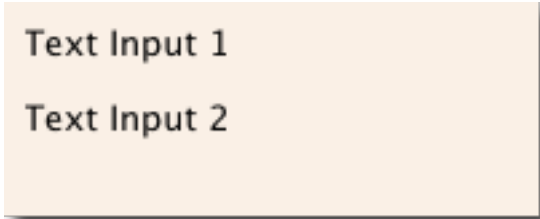
The `TextInput` allows the user to enter a line of text. The element supports input constraints such as `validator`, `inputMask`, and `echoMode`.

```
// textinput.qml
import QtQuick 2.5

Rectangle {
    width: 200
    height: 80
    color: "linen"

    TextInput {
        id: input1
        x: 8; y: 8
        width: 96; height: 20
        focus: true
        text: "Text Input 1"
    }

    TextInput {
        id: input2
        x: 8; y: 36
        width: 96; height: 20
        text: "Text Input 2"
    }
}
```



The user can click inside a `TextInput` to change the focus. To support switching the focus by keyboard, we can use the `KeyNavigation` attached property.

```
// textinput2.qml
import QtQuick 2.5

Rectangle {
    width: 200
    height: 80
    color: "linen"
```

```
TextInput {
    id: input1
    x: 8; y: 8
    width: 96; height: 20
    focus: true
    text: "Text Input 1"
    KeyNavigation.tab: input2
}

TextInput {
    id: input2
    x: 8; y: 36
    width: 96; height: 20
    text: "Text Input 2"
    KeyNavigation.tab: input1
}
}
```

The `KeyNavigation` attached property supports a preset of navigation keys where an element id is bound to switch focus on the given key press.

A text input element comes with no visual presentation besides a blinking cursor and the entered text. For the user to be able to recognize the element as an input element it needs some visual decoration, for example a simple rectangle. When placing the `TextInput` inside an element you need make sure you export the major properties you want others be able to access.

We move this piece of code into our own component called `TLineEditV1` for reuse.

```
// TLineEditV1.qml

import QtQuick 2.5

Rectangle {
    width: 96; height: input.height + 8
    color: "lightsteelblue"
    border.color: "gray"

    property alias text: input.text
    property alias input: input

    TextInput {
        id: input
        anchors.fill: parent
        anchors.margins: 4
        focus: true
    }
}
```

Note: If you want to export the `TextInput` completely, you can export the element by using `property alias input: input`. The first `input` is the property name, where the 2nd `input` is the element id.

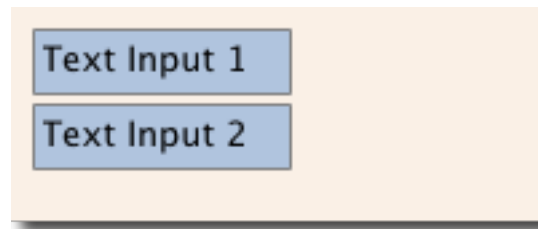
We rewrite our `KeyNavigation` example with the new `TLineEditV1` component.

```
Rectangle {
    ...
    TLineEditV1 {
        id: input1
        ...
    }
    TLineEditV1 {
        id: input2
    }
}
```

```

    ...
}
}

```



And try the tab key for navigation. You will experience the focus does not change to `input2`. The simple use of `focus:true` is not sufficient. The problem arises, that the focus was transferred to the `input2` element the top-level item inside the `TLineEditV1` (our `Rectangle`) received focus and did not forward the focus to the `TextInput`. To prevent this QML offers the `FocusScope`.

FocusScope

A focus scope declares that the last child element with `focus:true` receives the focus if the focus scope receives the focus. So it's forward the focus to the last focus requesting child element. We will create a 2nd version of our `TLineEdit` component called `TLineEditV2` using the focus scope as root element.

```

// TLineEditV2.qml

import QtQuick 2.5

FocusScope {
    width: 96; height: input.height + 8
    Rectangle {
        anchors.fill: parent
        color: "lightsteelblue"
        border.color: "gray"
    }

    property alias text: input.text
    property alias input: input

    TextInput {
        id: input
        anchors.fill: parent
        anchors.margins: 4
        focus: true
    }
}

```

Our example will now look like this:

```

Rectangle {
    ...
    TLineEditV2 {
        id: input1
        ...
    }
    TLineEditV2 {
        id: input2
        ...
    }
}

```

```
}  
}
```

Pressing the tab key now successfully switches the focus between the 2 components and the correct child element inside the component is focused.

TextEdit

The `TextEdit` is very similar to `TextInput` and support a multi-line text edit field. It doesn't have the text constraint properties as this depends on querying the painted size of the text (`paintedException`, `paintedException`). We also create our own component called `TTextEdit` to provide a edit background and use the focus scope for better focus forwarding.

```
// TTextEdit.qml  
  
import QtQuick 2.5  
  
FocusScope {  
    width: 96; height: 96  
    Rectangle {  
        anchors.fill: parent  
        color: "lightsteelblue"  
        border.color: "gray"  
    }  
  
    property alias text: input.text  
    property alias input: input  
  
    TextEdit {  
        id: input  
        anchors.fill: parent  
        anchors.margins: 4  
        focus: true  
    }  
}
```

You can use it like the `TLineEdit` component

```
// textedit.qml  
  
import QtQuick 2.5  
  
Rectangle {  
    width: 136  
    height: 120  
    color: "linen"  
  
    TTextEdit {  
        id: input  
        x: 8; y: 8  
        width: 120; height: 104  
        focus: true  
        text: "Text Edit"  
    }  
}
```



Keys Element

The attached property `Keys` allows executing code based on certain key presses. For example to move a square around and scale we can hook into the up, down, left and right keys to translate the element and the plus, minus key to scale the element.

```
// keys.qml

import QtQuick 2.5

DarkSquare {
    width: 400; height: 200

    GreenSquare {
        id: square
        x: 8; y: 8
    }
    focus: true
    Keys.onLeftPressed: square.x -= 8
    Keys.onRightPressed: square.x += 8
    Keys.onUpPressed: square.y -= 8
    Keys.onDownPressed: square.y += 8
    Keys.onPressed: {
        switch(event.key) {
            case Qt.Key_Plus:
                square.scale += 0.2
                break;
            case Qt.Key_Minus:
                square.scale -= 0.2
                break;
        }
    }
}
```

Advanced Techniques

Todo

To be written



Fluid Elements

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

Till now, we have mostly looked at simple graphical elements and how to arrange and manipulate them. This chapter is about how to control these changes in a way that a value of a property not just changes instantly, it's more how the value changes over time: an animation. This technology is one of the key foundations for modern slick user interfaces and can be extended with a system to describe your user interface using states and transitions. Each state defines a set of property changes and can be combined with animations on state changes, called transitions.

Animations

Animations are applied to property changes. An animation defines the interpolation curve when for property value changes to create smooth transitions from one value to another. An animation is defined by a series of target properties to be animated, an easing curve for the interpolation curve and in the most cases a duration, which defines the time for the property change. All animations in Qt Quick are controlled by the same timer, and are therefore synchronized. This improves the performance and visual quality of animations.

Note: Animations control how property changes, i.e. value interpolation. This is a fundamental concept. QML is based on elements, properties and scripting. Every element provides dozens of properties, each property is waiting to get animated by you. During the book you will see this is a spectacular playing field. You will caught yourself at looking at some animations and just admire their beauty and for sure also your creative genius. Please remember then: *Animations control property changes and every element has dozens of properties at your disposal.*

Unlock the power!



```
// animation.qml
```

```
import QtQuick 2.5

Image {
    id: root
    source: "assets/background.png"

    property int padding: 40
    property int duration: 400
    property bool running: false

    Image {
        id: box
        x: root.padding;
        y: (root.height-height)/2
        source: "assets/box_green.png"

        NumberAnimation on x {
            to: root.width - box.width - root.padding
            duration: root.duration
            running: root.running
        }
        RotationAnimation on rotation {
            to: 360
            duration: root.duration
            running: root.running
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: root.running = true
    }
}
```

The example above shows a simple animation applied on the `x` and `rotation` property. Each animation has a duration of 4000 milliseconds (msecs) and loops forever. The animation on `x` moves the `x` coordinate from the object gradually over to 240px. The animation on `rotation` runs from the current angle to 360 degree. Both animations run in parallel and are started as soon as the UI is loaded.

Now you can play around with the animation by changing the `to` and `duration` property or you could add another animation for example on the `opacity` or even the `scale`. Combining these it could look like the object is disappearing in the deep space. Try it out!

Animation Elements

There are several types of animation elements, each optimized for a specific use case. Here is a list of the most prominent animations:

- `PropertyAnimation` - Animates changes in property values
- `NumberAnimation` - Animates changes in qreal-type values
- `ColorAnimation` - Animates changes in color values
- `RotationAnimation` - Animates changes in rotation values

Besides these basic and widely used animation elements, Qt Quick provides also more specialized animations for specific use cases:

- `PauseAnimation` - Provides a pause for an animation
- `SequentialAnimation` - Allows animations to be run sequentially

- `ParallelAnimation` - Allows animations to be run in parallel
- `AnchorAnimation` - Animates changes in anchor values
- `ParentAnimation` - Animates changes in parent values
- `SmoothedAnimation` - Allows a property to smoothly track a value
- `SpringAnimation` - Allows a property to track a value in a spring-like motion
- `PathAnimation` - Animates an item along a path
- `Vector3dAnimation` - Animates changes in `QVector3d` values

We will learn later how to create a sequence of animations. While working on more complex animations there comes up the need to change a property or to run a script during an ongoing animation. For this Qt Quick offers the action elements, which can be used everywhere where the other animation elements can be used:

- `PropertyAction` - Specifies immediate property changes during animation
- `ScriptAction` - Defines scripts to be run during an animation

The major animation types will be discussed during this chapter using small focused examples.

Applying Animations

Animation can be applied in several ways:

- *Animation on property* - runs automatically after element is fully loaded
- *Behavior on property* - runs automatically when the property value changes
- *Standalone Animation* - runs when animation is explicitly started using `start()` or `running` is set to `true` (e.g. by a property binding)

Later we see also how animations can be used inside state transitions.

Extended ClickableImage Version 2

To demonstrate the usage of animations we reuse our `ClickableImage` component from an earlier chapter and extended it with a text element.

```
// ClickableImageV2.qml
// Simple image which can be clicked

import QtQuick 2.5

Item {
    id: root
    width: container.childrenRect.width
    height: container.childrenRect.height
    property alias text: label.text
    property alias source: image.source
    signal clicked

    Column {
        id: container
        Image {
            id: image
        }
        Text {
            id: label
            width: image.width
            horizontalAlignment: Text.AlignHCenter
            wrapMode: Text.WordWrap
            color: "#ececec"
        }
    }
}
```

```
anchors.fill: parent
onClicked: root.clicked()
}
```

To organize the element below the image we used a `Column` positioner and calculated the width and height based on the column's `childrenRect` property. We exposed two properties: `text` and the image `source` as also the `clicked` signal. We also wanted that the text is as wide as the image and it should wrap. We achieve the latter by using the `Text` elements `wrapMode` property.

Note: Due to the inversion of the geometry-dependency (parent geometry depends on child geometry) we can't set a width/height on the `ClickableImageV2`, as this will break our width/height binding. This is a limitation on our internal design and as a designer of components you should be aware of this. Normally you should prefer the child's geometry to depend on the parent's geometry.

The objects ascending.



The three objects are all at the same y-position ($y=200$). They need to travel all to $y=40$. Each of them using a different method with different side-effects and features.

```
ClickableImageV2 {
    id: greenBox
    x: 40; y: root.height-height
    source: "assets/box_green.png"
    text: "animation on property"
    NumberAnimation on y {
        to: 40; duration: 4000
    }
}
```

1st object

The 1st object travels using the `Animation` on `<property>` strategy. The animation starts immediately. When an object is clicked their y-position is reset to the start position, this applies to all objects. On the 1st object the reset does not have any effect as long as the animation is running. It's even disturbing as the y-position is set for a fraction of a second to a new value before the animation starts. *Such competing property changes should be avoided.*

```
ClickableImageV2 {
    id: blueBox
    x: (root.width-width)/2; y: root.height-height
    source: "assets/box_blue.png"
    text: "behavior on property"
    Behavior on y {
        NumberAnimation { duration: 4000 }
    }

    onClicked: y = 40
    // random y on each click
    //     onClicked: y = 40+Math.random()*(205-40)
}
```

2nd object

The 2nd object travels using a `behavior` on `animation`. This behavior tells the property, every time the property value changes, it changes through this animation. The behavior can be disabled by `enabled : false` on the `Behavior` element. The object will start traveling when you click it (y-position is then set to 40). Another click has no influence as the position is already set. You could try to use a random value (e.g. `40+(Math.random()*(205-40))`) for the y-position. You will see that the object will always animate to the new position and adapt its speed to match the 4 seconds to the destination defined by the animations duration.

```
ClickableImageV2 {
    id: redBox
    x: root.width-width-40; y: root.height-height
    source: "assets/box_red.png"
    onClicked: anim.start()
    //     onClicked: anim.restart()

    text: "standalone animation"

    NumberAnimation {
        id: anim
        target: redBox
        properties: "y"
        to: 40
        duration: 4000
    }
}
```

3rd object

The 3rd object uses a `standalone animation`. The animation is defined as its own element and could be everywhere in the document. The click will start the animation using the animations function `start()`. Each animation has a `start()`, `stop()`, `resume()`, `restart()` function. The animation itself contains much more information than the other animation types earlier. We need to define the `target` and `properties` to declare the target element to be animated and which properties we want to animate. We need to define a `to` value and in this case we define also a `from` value to allow a re-start of the animation.



A click on the background will reset all objects to their initial position. The 1st object can't be restarted except by re-starting the program which triggers the re-loading of the element.

Note: Another way to start/stop an animation is to bind a property to the `running` property of an animation. This is especially useful when the user-input is in control of properties:

```
NumberAnimation {  
    ...  
    // animation runs when mouse is pressed  
    running: area.pressed  
}  
MouseArea {  
    id: area  
}
```

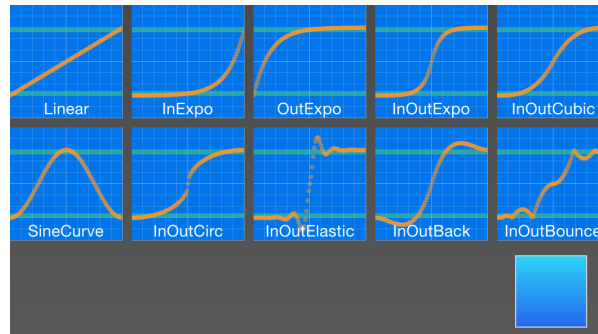
Easing Curves

The value change of a property can be controlled by an animation. Easing attributes allows influencing the interpolation curve of a property change. All animations we have defined by now use a linear interpolation because the initial easing type of an animation is `Easing.Linear`. It's best visualized with a small plot, where the y-axis is the property to be animated and the x-axis is the time (*duration*). A linear interpolation would draw a straight line from the `from` value at the start of the animation to the `to` value at the end of the animation. So the easing type defines the curve of change. Easing types are carefully chosen to support a natural fit for a moving object, for example when a page slides out. Initially the page should slide out slowly and then gaining the speed to finally slide out on high speed, similar to turning the page of a book.

Note: Animations should not be overused. As other aspects of UI design also animations should be designed

carefully and support the UI flow and not dominate it. The eye is very sensitive to moving objects and animations can easily distract the user.

In the next example we will try some easing curves. Each easing curve is displayed by a click-able image and, when clicked, will set a new easing type on the `square` animation and then trigger a `restart()` to run the animation with the new curve.



The code for this example was made a little bit more complicated. We first create a grid of `EasingTypes` and a `Box` which is controlled by the easing types. A easing type just displays the curve which the box shall use for its animation. When the user clicks on an easing curve the box moves in a direction according to the easing curve. The animation itself is a standalone-animation with the target set to the box and configured for x-property animation with a duration of 2 secs.

Note: The internals of the `EasingType` renders the curve in real time and the interested reader can look it up in the `EasingCurves` example.

```
// EasingCurves.qml

import QtQuick 2.5
import QtQuick.Layouts 1.2

Rectangle {
    id: root
    width: childrenRect.width
    height: childrenRect.height

    color: '#4a4a4a'
    gradient: Gradient {
        GradientStop { position: 0.0; color: root.color }
        GradientStop { position: 1.0; color: Qt.lighter(root.color, 1.2) }
    }

    ColumnLayout {

        Grid {
            spacing: 8
            columns: 5
            EasingType {
                easingType: Easing.Linear
                title: 'Linear'
                onClicked: {
                    animation.easing.type = easingType
                    box.toggle = !box.toggle
                }
            }
            EasingType {
                easingType: Easing.InExpo
```

```
        title: "InExpo"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.OutExpo
        title: "OutExpo"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.InOutExpo
        title: "InOutExpo"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.InOutCubic
        title: "InOutCubic"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.SineCurve
        title: "SineCurve"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.InOutCirc
        title: "InOutCirc"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.InOutElastic
        title: "InOutElastic"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
        easingType: Easing.InOutBack
        title: "InOutBack"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
    EasingType {
```



```

        easingType: Easing.InOutBounce
        title: "InOutBounce"
        onClicked: {
            animation.easing.type = easingType
            box.toggle = !box.toggle
        }
    }
}
Item {
    height: 80
    Layout.fillWidth: true
    Box {
        id: box
        property bool toggle
        x: toggle?20:root.width-width-20
        anchors.verticalCenter: parent.verticalCenter
        gradient: Gradient {
            GradientStop { position: 0.0; color: "#2ed5fa" }
            GradientStop { position: 1.0; color: "#2467ec" }
        }
        Behavior on x {
            NumberAnimation {
                id: animation
                duration: 500
            }
        }
    }
}
}
}
}

```

As you play with it, please observe the change of speed during an animation. Some animations feel more natural for the object and some feel irritating.

Besides the duration and `easing.type` you are able to fine tune animations. For example the general `PropertyAnimation` where most animation inherit from additionally supports an `easing.amplitude`, `easing.overshoot` and `easing.period` property which allows you to fine-tune the behavior of particular easing curves. Not all easing curves support these parameters. Please consult the [easing table](#) from the `PropertyAnimation` documentation to check if an easing parameter has influence on an easing curve.

Note: Choosing the right animation for the element in the user interface context is crucial for the outcome. Remember the animation shall support the UI flow; not irritate the user.

Grouped Animations

Often animations will be more complex then just animating one property. You might want to run several animations at the same time or one after another or even execute a script between two animations. For this, the grouped animation offer you a possibility. As the named suggests it's possible to group animations. Grouping can be done in two ways: parallel or sequential. You can use the `SequentialAnimation` or the `ParallelAnimation` element, which act as animation containers for other animation elements. These grouped animations are animations themselves and can be used exactly as such.

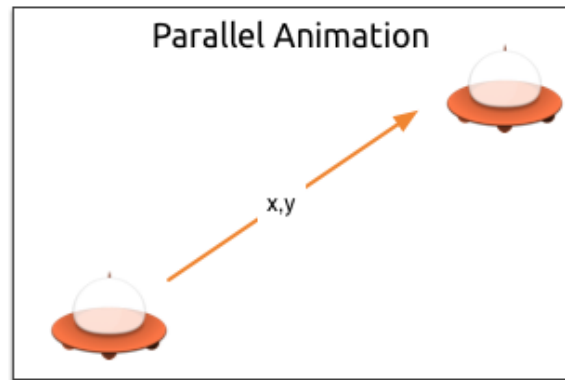
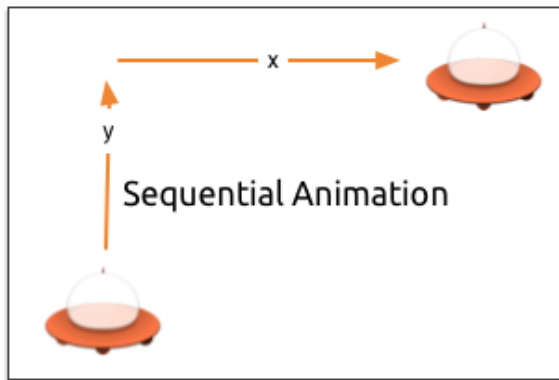
All direct child animations of a parallel animation will run in parallel, when started. This allows you to animate different properties at the same time.

```

// parallelanimation.qml
import QtQuick 2.5

BrightSquare {

```



```

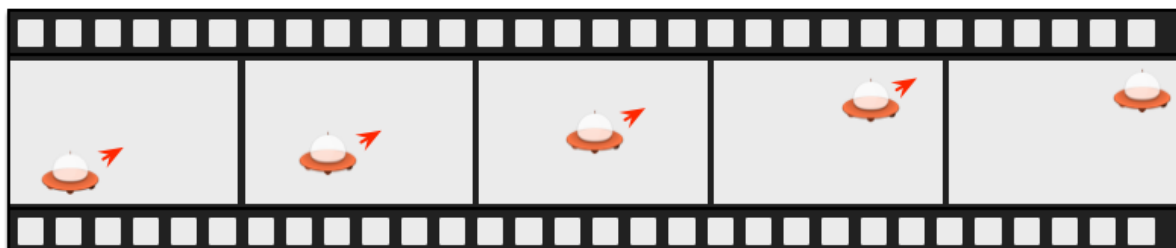
id: root
width: 600
height: 400
property int duration: 3000
property Item ufo: ufo

Image {
    anchors.fill: parent
    source: "assets/ufo_background.png"
}

ClickableImageV3 {
    id: ufo
    x: 20; y: root.height-height
    text: 'ufo'
    source: "assets/ufo.png"
    onClicked: anim.restart()
}

ParallelAnimation {
    id: anim
    NumberAnimation {
        target: ufo
        properties: "y"
        to: 20
        duration: root.duration
    }
    NumberAnimation {
        target: ufo
        properties: "x"
        to: 160
        duration: root.duration
    }
}

```



A sequential animation will first run the first child animation and then continue from there.

```
// sequentialanimation.qml
import QtQuick 2.5

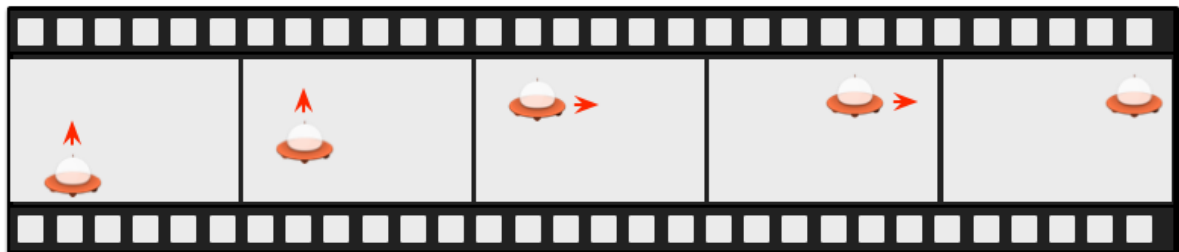
BrightSquare {
    id: root
    width: 600
    height: 400
    property int duration: 3000

    property Item ufo: ufo

    Image {
        anchors.fill: parent
        source: "assets/ufo_background.png"
    }

    ClickableImageV3 {
        id: ufo
        x: 20; y: root.height-height
        text: 'rocket'
        source: "assets/ufo.png"
        onClicked: anim.restart()
    }

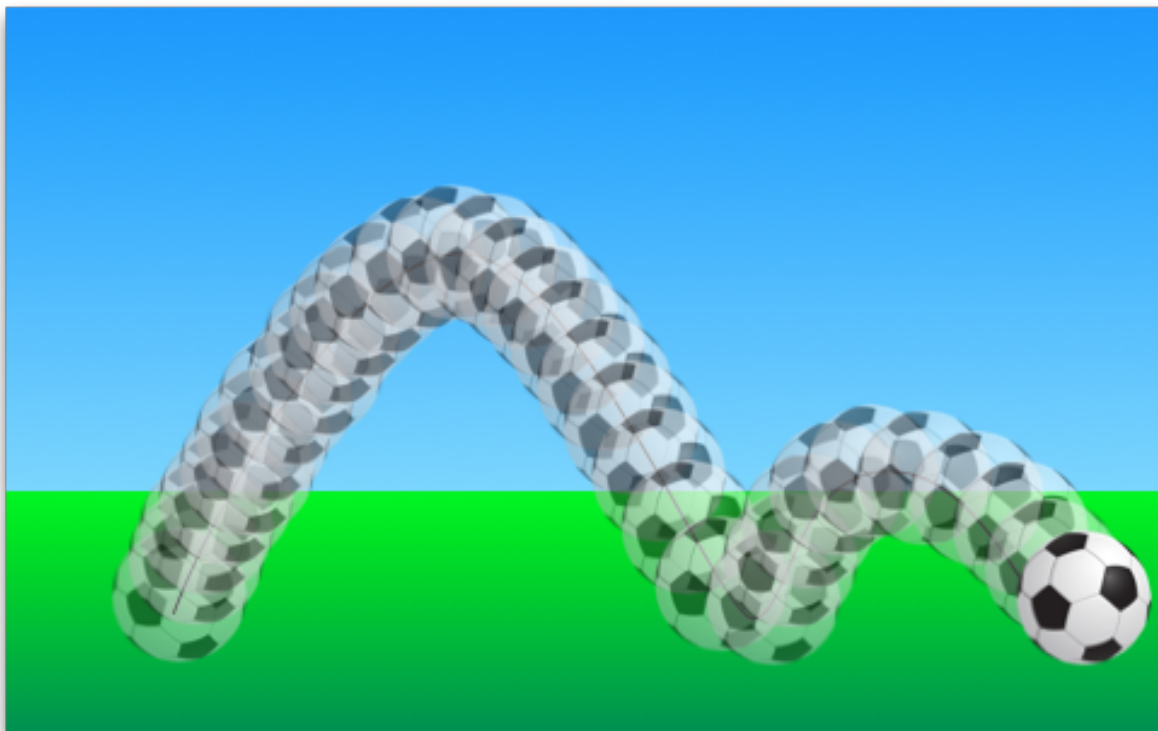
    SequentialAnimation {
        id: anim
        NumberAnimation {
            target: ufo
            properties: "y"
            to: 20
            // 60% of time to travel up
            duration: root.duration*0.6
        }
        NumberAnimation {
            target: ufo
            properties: "x"
            to: 400
            // 40% of time to travel sideways
            duration: root.duration*0.4
        }
    }
}
```



Grouped animation can also be nested, for example a sequential animation can have two parallel animations as child animations, and so on. We can visualize this with a soccer ball example. The idea is to throw a ball from left to right and animate its behavior.

To understand the animation we need to dissect it into the integral transformations of the object. We need to remember animation does animate property changes. Here are the different transformations:

- An x-translation from left-to-right (X1)



- An y-translation from down to up (Y1) followed by a translation from up to down (Y2) with some bouncing
- A rotation over 360 over the whole animation duration (ROT1)

The whole duration of the animation should take three seconds.

We start with an empty item as root element of the width of 480 and height of 300.

```
import QtQuick 2.5

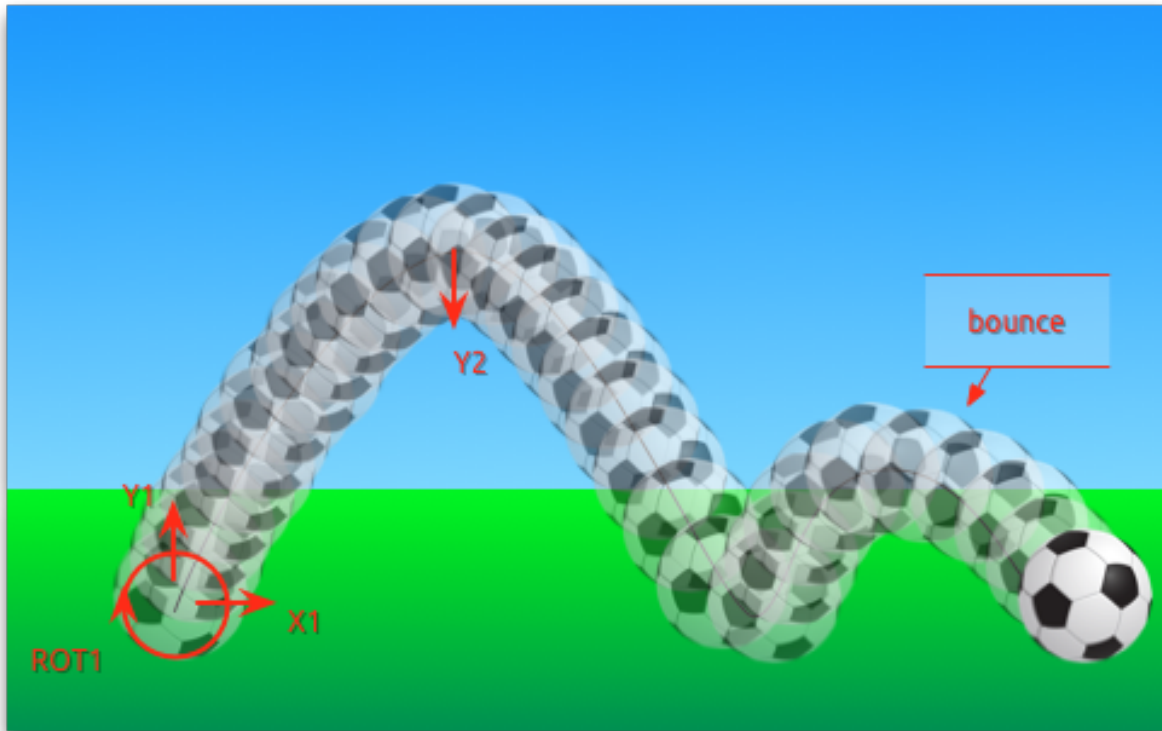
Item {
    id: root
    width: 480
    height: 300
    property int duration: 3000

    ...
}
```

We have defined our total animation duration as reference to better synchronize the animation parts.

The next step would be to add the background, which in our case are 2 rectangles with a green and blue gradients.

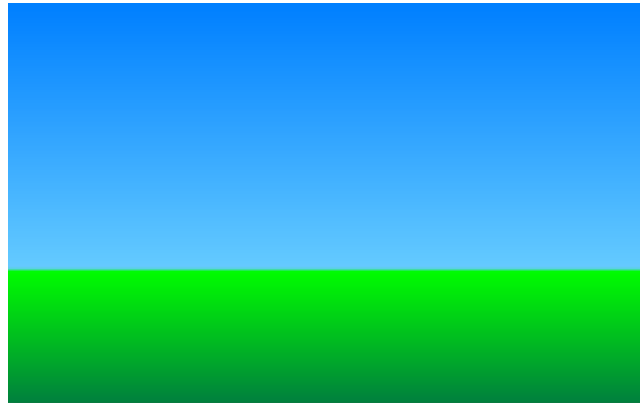
```
Rectangle {
    id: sky
    width: parent.width
    height: 200
    gradient: Gradient {
        GradientStop { position: 0.0; color: "#0080FF" }
        GradientStop { position: 1.0; color: "#66CCFF" }
    }
}
Rectangle {
    id: ground
    anchors.top: sky.bottom
    anchors.bottom: root.bottom
}
```



```

width: parent.width
gradient: Gradient {
    GradientStop { position: 0.0; color: "#00FF00" }
    GradientStop { position: 1.0; color: "#00803F" }
}

```



The upper blue rectangle takes 200 pixel of the height and the lower one is anchored to the top on the sky and to the bottom on the root element.

Let's bring the soccer ball onto the green. The ball is an image, stored under "assets/soccer_ball.png". For the beginning we would like to position it in the lower left corner, near the edge.

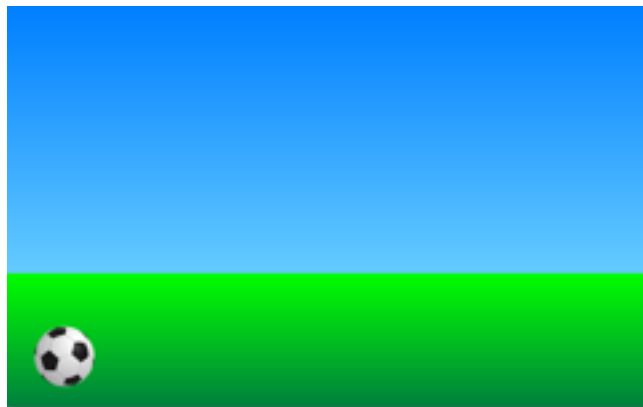
```

Image {
    id: ball
    x: 0; y: root.height-height
    source: "assets/soccer_ball.png"

    MouseArea {

```

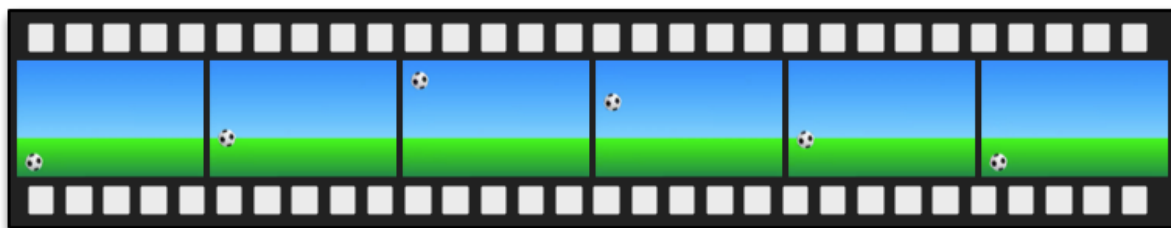
```
anchors.fill: parent
onClicked: {
    ball.x = 0;
    ball.y = root.height-ball.height;
    ball.rotation = 0;
    anim.restart()
}
}
```



The image has a mouse area attached to it. If the ball is clicked the position of the ball will reset and the animation restarted.

Let's start with an sequential animation for the two y translations first.

```
SequentialAnimation {
    id: anim
    NumberAnimation {
        target: ball
        properties: "y"
        to: 20
        duration: root.duration * 0.4
    }
    NumberAnimation {
        target: ball
        properties: "y"
        to: 240
        duration: root.duration * 0.6
    }
}
```



This specifies that 40% of the total animation duration is the up animation and 60% the down animation. One animation after another as a sequence. The transformations are animated on a linear path but there is no curve currently. Curves will be added later using the easing curves, at the moment we're concentrating on getting the transformations animated.

Next, we need to add the x-translation. The x-translation shall run in parallel with the y-translation so we need to encapsulate the sequence of y-translations into a parallel animation together with the x-translation.

```
ParallelAnimation {
    id: anim
    SequentialAnimation {
        // ... our Y1, Y2 animation
    }
    NumberAnimation { // X1 animation
        target: ball
        properties: "x"
        to: 400
        duration: root.duration
    }
}
```



At the end we would like the ball to be rotating. For this we need to add another animation to the parallel animation. We choose the `RotationAnimation` as it's specialized for rotation.

```
ParallelAnimation {
    id: anim
    SequentialAnimation {
        // ... our Y1, Y2 animation
    }
    NumberAnimation { // X1 animation
        // X1 animation
    }
    RotationAnimation {
        target: ball
        properties: "rotation"
        to: 720
        duration: root.duration
    }
}
```

That's the whole animation sequence. The one thing left is to provide the correct easing curves for the movements of the ball. For the *Y1* animation I use a `Easing.OutCirc` curve as this should look more like a circular movement. *Y2* is enhanced using an `Easing.OutBounce` as the ball should bounce and the bouncing should happen at the end (try an `Easing.InBounce` and you see the bouncing will start right away). The *X1* and *ROT1* animation are left as is with a linear curve.

Here is the final animation code for your reference:

```
ParallelAnimation {
    id: anim
    SequentialAnimation {
        NumberAnimation {
            target: ball
            properties: "y"
            to: 20
            duration: root.duration * 0.4
            easing.type: Easing.OutCirc
        }
        NumberAnimation {
            target: ball
            // ... other properties
        }
    }
}
```

```
        properties: "y"
        to: root.height-ball.height
        duration: root.duration * 0.6
        easing.type: Easing.OutBounce
    }
}
NumberAnimation {
    target: ball
    properties: "x"
    to: root.width-ball.width
    duration: root.duration
}
RotationAnimation {
    target: ball
    properties: "rotation"
    to: 720
    duration: root.duration
}
}
```

States and Transitions

Often parts of a user interface can be described in states. A state defines a set of property changes and can be triggered by a certain condition. Additionally these state switches can have a transition attached which defines how these changes should be animated or any additional actions shall be applied. Actions can also be applied when a state is entered.

States

You define states in QML with the `State` element, which needs to be bound to the `states` array of any item element. A state is identified through a state name and consists, in its simplest form, of a series of property changes on elements. The default state is defined by the initial properties of the element and is named `""` (the empty string).

```
Item {
    id: root
    states: [
        State {
            name: "go"
            PropertyChanges { ... }
        },
        State {
            name: "stop"
            PropertyChanges { ... }
        }
    ]
}
```

A state is changed by assigning a new state name to the `state` property of the element with the states defined.

Note: Another way to switch states is using the `when` property of the `State` element. The `when` property can be set to an expression that evaluates to true when the state should be applied.

```
Item {
    id: root
    states: [
```

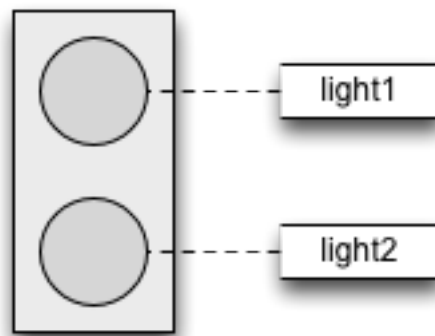


```

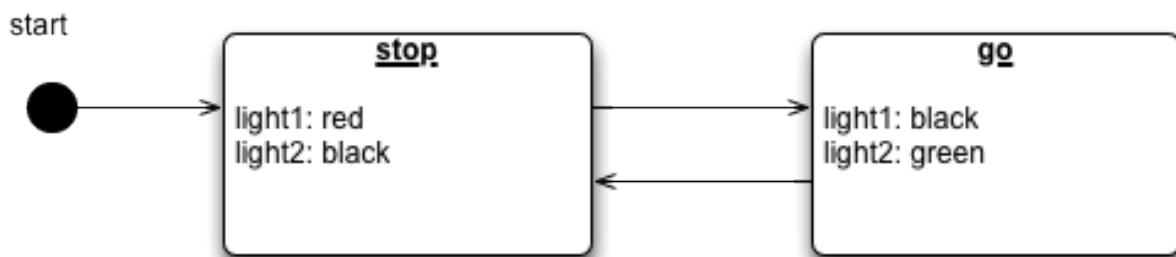
    ...
}

Button {
    id: goButton
    ...
    onClicked: root.state = "go"
}
}

```



For example, a traffic light might have two signaling lights. The upper one signaling stop with a red color and the lower one signaling go with a green color. In this example both lights should not shine at the same time. Let's have a look at the state chart diagram.



When the system is switched on it goes automatically into the stop mode as default state. The stop state changes the `light1` to red and `light2` to black (off). An external event can now trigger a state switch to the "go" state. In the go state we change the color properties from `light1` to black (off) and `light2` to green to indicate the passers may walk now.

To realize this scenario we start sketching our user interface for the 2 lights. For simplicity we use 2 rectangles with the radius set to the half of the width (and the width is the same as the height, which means it's a square).

```

Rectangle {
    id: light1
    x: 25; y: 15
    width: 100; height: width
    radius: width/2
    color: root.black
    border.color: Qt.lighter(color, 1.1)
}

Rectangle {
    id: light2
    x: 25; y: 135
    width: 100; height: width
    radius: width/2
    color: root.black
}

```

```
border.color: Qt.lighter(color, 1.1)
}
```

As defined in the state chart we want to have two states one the "go" state and the other the "stop" state, where each of them changes the traffic lights respective to red or green. We set the state property to stop to ensure the initial state of our traffic light is the stop state.

Note: We could have achieved the same effect with only a "go" state and no explicit "stop" state by setting the color of light1 to red and the color of light2 to black. The initial state "" defined by the initial property values would then act as the "stop" state.

```
state: "stop"

states: [
  State {
    name: "stop"
    PropertyChanges { target: light1; color: root.red }
    PropertyChanges { target: light2; color: root.black }
  },
  State {
    name: "go"
    PropertyChanges { target: light1; color: root.black }
    PropertyChanges { target: light2; color: root.green }
  }
]
```

Using `PropertyChanges { target: light2; color: "black" }` is not really required in this examples as the initial color of light2 is already black. In a state it's only necessary to describe how the properties shall change from their default state (and not from the previous state).

A state change is triggered using a mouse area which covers the whole traffic light and toggles between the go and stop state when clicked.

```
MouseArea {
    anchors.fill: parent
    onClicked: parent.state = (parent.state == "stop"? "go" : "stop")
}
```

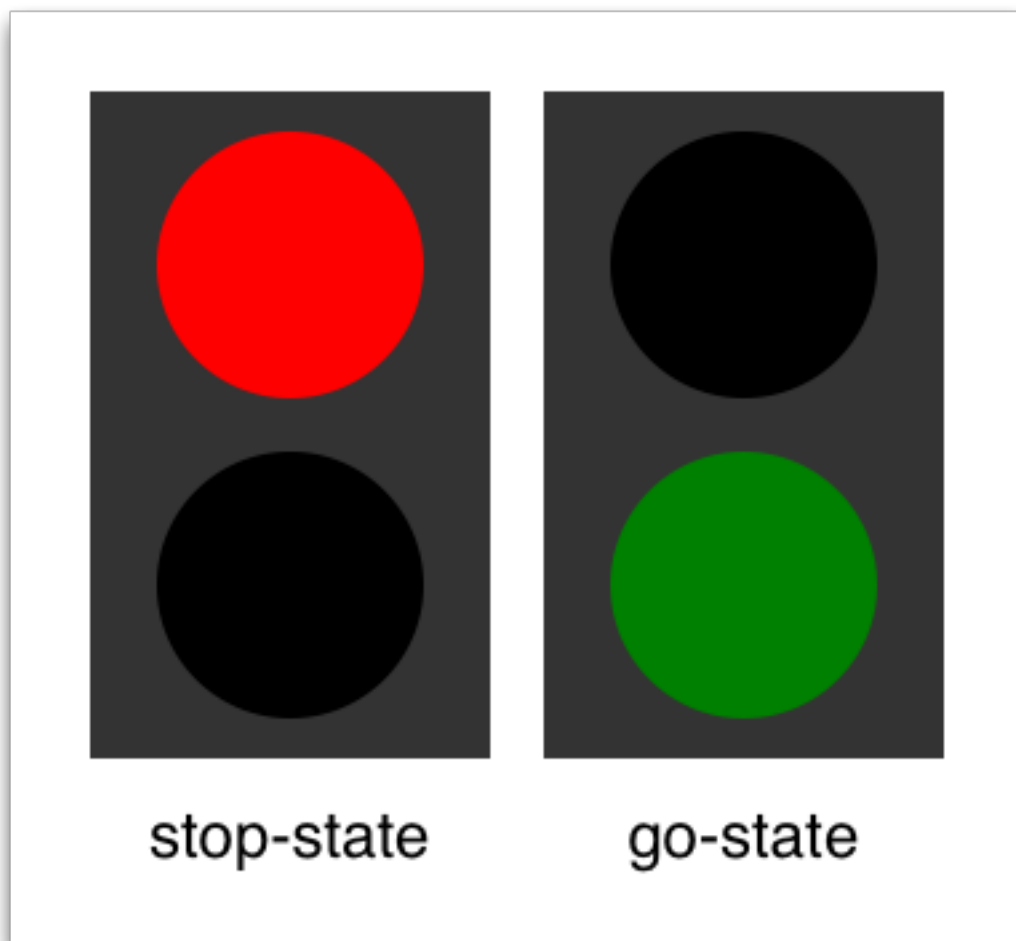
We are now able to successfully change the state of the traffic lamp. To make the UI more appealing and look natural we should add some transitions with animation effects. A transition can be triggered by a state change.

Note: It's possible to create a similar logic using scripting instead of QML states. Developers can easily fall into the trap of writing more a JavaScript program than a QML program.

Transitions

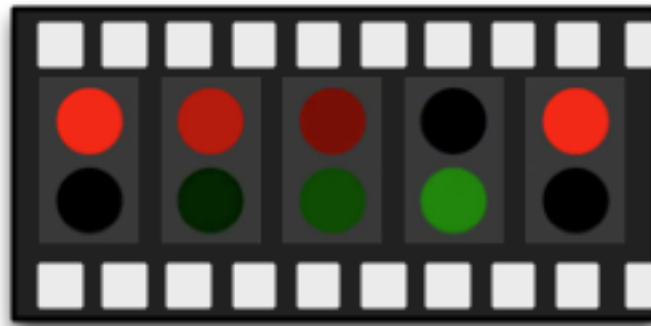
A series of transitions can be added to every item. A transition is executed by a state change. You can define on which state change a particular transition can be applied using the `from:` and `to:` properties. These two properties act like a filter, when the filter is true the transition will be applied. You can also use the wild-card "*" which means "any state". For example `from: "*"; to: "*"` means from any state to any other state and is the default value for `from` and `to`, which means the transition is applied to every state switch.

For this example we would like to animate the color changes when switching state from "go" to "stop". For the other reversed state change ("stop" to "go") we want to keep an immediate color change and don't apply a transition. We restrict the transition with the `from` and `to` properties to filter only the state change from "go" to "stop". Inside the transition we add two color animations for each light, which shall animate the property changes defined in the state description.



```
transitions: [
    Transition {
        from: "stop"; to: "go"
//        from: "*" to: "*"
        ColorAnimation { target: light1; properties: "color"; duration: 2000 }
        ColorAnimation { target: light2; properties: "color"; duration: 2000 }
    }
]
```

You can change the state though clicking the UI. The state is applied immediately and will also change the state while a transition is running. So try to click the UI while the state is in transition from “stop” to “go”. You will see the change will happen immediately.



You could play around with this UI by, for example, scaling the inactive light down to highlight the active light. For this you would need to add another property change for scaling to the states and also handle the animation for the scaling property in the transition. Another option would be to add an “attention” state where the lights are blinking yellow. For this, you would need to add a sequential animation to the transition for one second going to yellow (“to” property of the animation and one sec going to “black”). Maybe you would also want to change the easing curve to make it more visually appealing.

Advanced Techniques

Todo

To be written

Model-View-Delegate

Section author: e8johan

Note: Last Build: March 11, 2018 at 15:30 CET

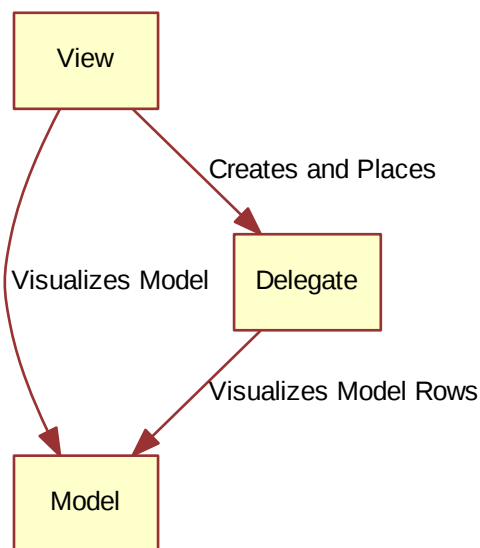
The source code for this chapter can be found in the assets folder.

In Qt Quick, data is separated from the presentation through a model-view separation. For each view, the visualization of each data element is separated into a delegate. Qt Quick comes with a set of predefined models and views. To utilize the system, one must understand these classes and know how to create appropriate delegates to get the right look and feel.

Concept

One of the most important aspects when developing user interfaces, is to keep the representation of the data separate from the visualization. For instance, a phonebook could be arranged as a vertical list of text entries or a grid of pictures of the contacts. In both cases, the data is identical: the phonebook, but the visualization differs. This division is commonly referred to as the model-view pattern. In this pattern the data is referred to as the model, while the visualization is handled by the view.

In QML, the model and view are joined by the delegate. The responsibility are divided as follows. The model provides the data. For each data item, there might be multiple values. In the example above, each phonebook entry has a name, a picture and a number. The data is arranged in a view, in which each item is visualized using a delegate. The task of the view is to arrange the delegates, while each delegate shows the values of each model item to the user.



Basic Models

The most basic way to separate the data from the presentation is to use the `Repeater` element. It is used to instantiate an array of items, and is easy to combine with a positioner to populate a part of the user interface. A repeater uses a model, which can be anything from the number of items to instantiate, to a fully blown model gathering data from the Internet.

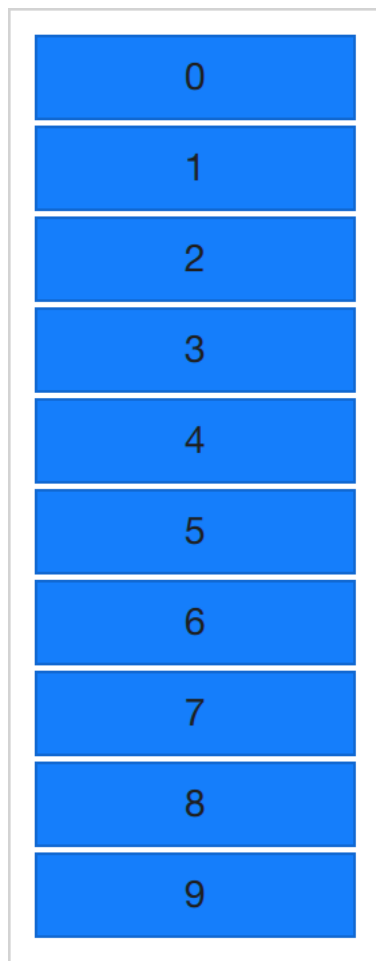
In its simplest form, the repeater can be used to instantiate a specified number of items. Each item will have access to an attached property, the variable `index`, that can be used to tell the items apart. In the example below, a repeater is used to create 10 instances of an item. The number of items are controlled using the `model` property. For each item, the `Rectangle` containing a `Text` element found inside the `Repeater` item, is instantiated. As you can tell, the `text` property is set to the `index` value, thus the items are numbered from zero to nine.

```
import QtQuick 2.5
import "../common"

Column {
    spacing: 2

    Repeater {
        model: 10
        BlueBox {
            width: 120
            height: 32
            text: index
        }
    }
}
```

As nice as lists of numbered items are, it is sometimes interesting to display a more complex data set. By replacing the integer `model` value with a JavaScript array, we can achieve that. The contents of the array can be of any type, be it strings, integers or objects. In the example below, a list of strings is used. We can still access and use the `index` variable, but we also have access to `modelData` containing the data for each element in the array.



```
import QtQuick 2.5
import "../common"

Column {
    spacing: 2

    Repeater {
        model: ["Enterprise", "Columbia", "Challenger", "Discovery", "Endeavour",
        ↪ "Atlantis"]

        BlueBox {
            width: 100
            height: 32
            radius: 3

            text: modelData + ' (' + index + ')'
        }
    }
}
```



Being able to expose the data of an array, you soon find yourself in a position where you need multiple pieces of data per item in the array. This is where models enter the picture. One of the most trivial models, and one of the most commonly used, is the `ListModel`. A list model is simply a collection of `ListElement` items. Inside each list element, a number of properties can be bound to values. For instance, in the example below, a name and a color is provided for each element.

The properties bound inside each element are attached to each instantiated item by the repeater. This means that the variables `name` and `surfaceColor` are available from within the scope of each `Rectangle` and `Text` item created by the repeater. This not only makes it easy to access the data, it also makes it easy to read the source code. The `surfaceColor` is the color of the circle to the left of the name, not something obscure as data from column `i` of row `j`.

```
import QtQuick 2.5
import "../common"

Column {
    spacing: 2

    Repeater {
        model: ListModel {
```



```

        ListElement { name: "Mercury"; surfaceColor: "gray" }
        ListElement { name: "Venus"; surfaceColor: "yellow" }
        ListElement { name: "Earth"; surfaceColor: "blue" }
        ListElement { name: "Mars"; surfaceColor: "orange" }
        ListElement { name: "Jupiter"; surfaceColor: "orange" }
        ListElement { name: "Saturn"; surfaceColor: "yellow" }
        ListElement { name: "Uranus"; surfaceColor: "lightBlue" }
        ListElement { name: "Neptune"; surfaceColor: "lightBlue" }
    }

    BlueBox {
        width: 120
        height: 32

        radius: 3
        text: name

        Box {
            anchors.left: parent.left
            anchors.verticalCenter: parent.verticalCenter
            anchors.leftMargin: 4

            width: 16
            height: 16

            radius: 8

            color: surfaceColor
        }
    }
}

```



The contents of the repeater that is being instantiated for each item is actually what is bound to the default property, `delegate`. This means that the code of example *Example 01* is synonymous to the code shown below. Notice

that the only difference is that the `delegate` property name is spelled out explicitly in the latter.

```
import QtQuick 2.5
import "../common"

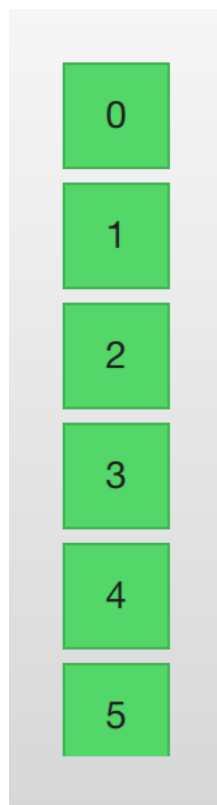
Column {
    spacing: 2

    Repeater {
        model: 10

        delegate: BlueBox {
            width: 100
            height: 32
            text: index
        }
    }
}
```

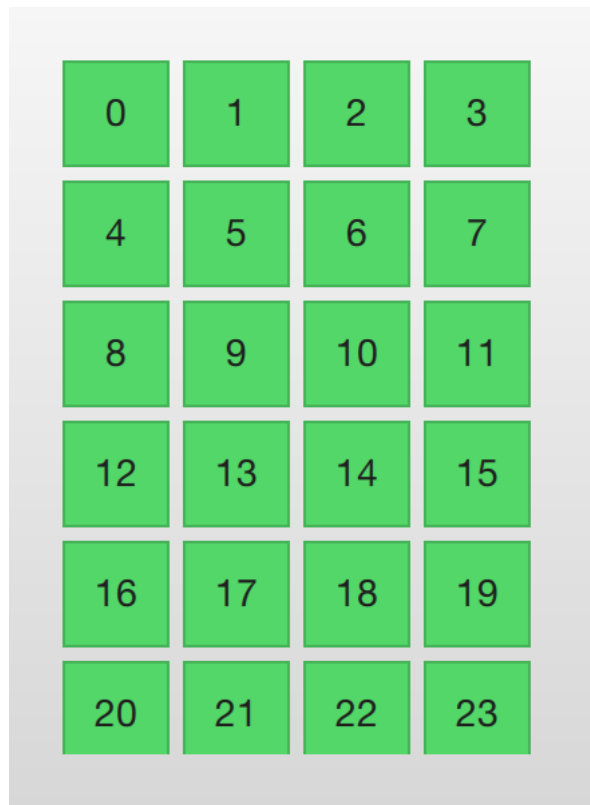
Dynamic Views

Repeaters work well for limited and static sets of data, but in the real world, models are commonly more complex – and larger. Here, a smarter solution is needed. For this, Qt Quick provides the `ListView` and `GridView` elements. These are both based on a `Flickable` area, so the user can move around in a larger data set. At the same time, they limit the number of concurrently instantiated delegates. For a large model, that means fewer elements in the scene at once.



The two elements are similar in their usage. Thus, we will begin with the `ListView` and then describe the `GridView` with the former as the starting point of the comparison.

The `ListView` is similar to the `Repeater` element. It uses a model, instantiates a delegate and between the delegates, there can be spacing. The listing below shows how a simple setup can look.



```

import QtQuick 2.5
import "../common"

Background {
    width: 80
    height: 300

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

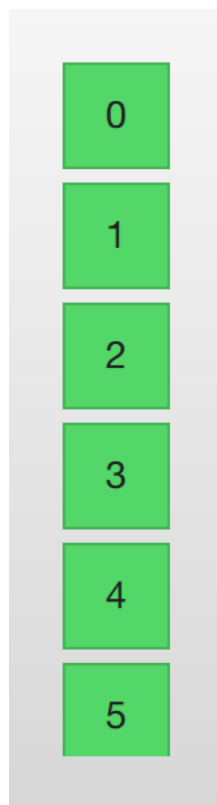
        delegate: numberDelegate
        spacing: 5
    }

    Component {
        id: numberDelegate

        GreenBox {
            width: 40
            height: 40
            text: index
        }
    }
}

```

If the model contains more data than can fit onto the screen, the `ListView` only shows part of the list. However, as a consequence of the default behavior of Qt Quick, the list view does not limit the screen area within which the delegates are shown. This means that delegates may be visible outside the list view, and that the dynamic creation



and destruction of delegates outside the list view is visible to the user. To prevent this, clipping must be activated on the `ListView` element by setting the `clip` property to `true`. The illustration below shows the result of this, compared to when the `clip` property is left as `false`.

To the user, the `ListView` is a scrollable area. It supports kinetic scrolling, which means that it can be flicked to quickly move through the contents. By default, it also can be stretched beyond the end of contents, and then bounces back, to signal to the user that the end has been reached.

The behavior at the end of the view is controlled using the `boundsBehavior` property. This is an enumerated value and can be configured from the default behavior, `Flickable.DragAndOvershootBounds`, where the view can be both dragged and flicked outside its boundaries, to `Flickable.StopAtBounds`, where the view never will move outside its boundaries. The middle ground, `Flickable.DragOverBounds` lets the user drag the view outside its boundaries, but flicks will stop at the boundary.

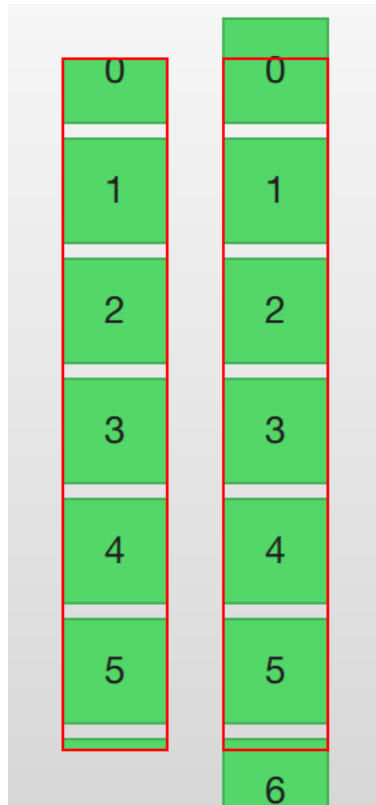
It is possible to limit the positions where a view is allowed to stop. This is controlled using the `snapMode` property. The default behavior, `ListView.NoSnap`, lets the view stop at any position. By setting the `snapMode` property to `ListView.SnapToItem`, the view will always align the top of an item with its top. Finally, the `ListView.SnapOneItem`, the view will stop no more than one item from the first visible item when the mouse button or touch was released. The last mode is very handy when flipping through pages.

Orientation

The list view provides a vertically scrolling list by default, but horizontal scrolling can be just as useful. The direction of the list view is controlled through the `orientation` property. It can be set to either the default value, `ListView.Vertical`, or to `ListView.Horizontal`. A horizontal list view is shown below.

```
import QtQuick 2.5
import "../common"

Background {
    width: 480
    height: 80
}
```



```

ListView {
    anchors.fill: parent
    anchors.margins: 20
    spacing: 4
    clip: true
    model: 100
    orientation: ListView.Horizontal
    delegate: numberDelegate
}

Component {
    id: numberDelegate

    GreenBox {
        width: 40
        height: 40
        text: index
    }
}

```



As you can tell, the direction of the horizontal flows from the left to the right by default. This can be controlled through the `layoutDirection` property, which can be set to either `Qt.LeftToRight` or `Qt.RightToLeft`, depending on the flow direction.

Keyboard Navigation and Highlighting

When using a `ListView` in a touch based setting, the view itself is enough. In a scenario with a keyboard, or even just arrow keys to select an item, a mechanism to indicate the current item is needed. In QML, this is called highlighting.

Views support a highlight delegate which is shown in the view together with the delegates. It can be considered an additional delegate, only that it is only instantiated once, and is moved into the same position as the current item.

In the example below this is demonstrated. There are two properties involved for this to work. First, the `focus` property is set to `true`. This gives the `ListView` the keyboard focus. Second, the `highlight` property is set to point out the highlighting delegate to use. The highlight delegate is given the `x`, `y` and `height` of the current item. If the `width` is not specified, the width of the current item is also used.

In the example, the `ListView.view.width` attached property is used for width. The attached properties available to delegates are discussed further in the delegate section of this chapter, but it is good to know that the same properties are available to highlight delegates as well.

```
import QtQuick 2.5
import "../common"

Background {
    width: 240
    height: 300

    ListView {
        id: view
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

        delegate: numberDelegate
        spacing: 5

        highlight: highlightComponent
        focus: true
    }

    Component {
        id: highlightComponent

        GreenBox {
            width: ListView.view.width
        }
    }

    Component {
        id: numberDelegate

        Item {
            width: ListView.view.width
            height: 40

            Text {
                anchors.centerIn: parent

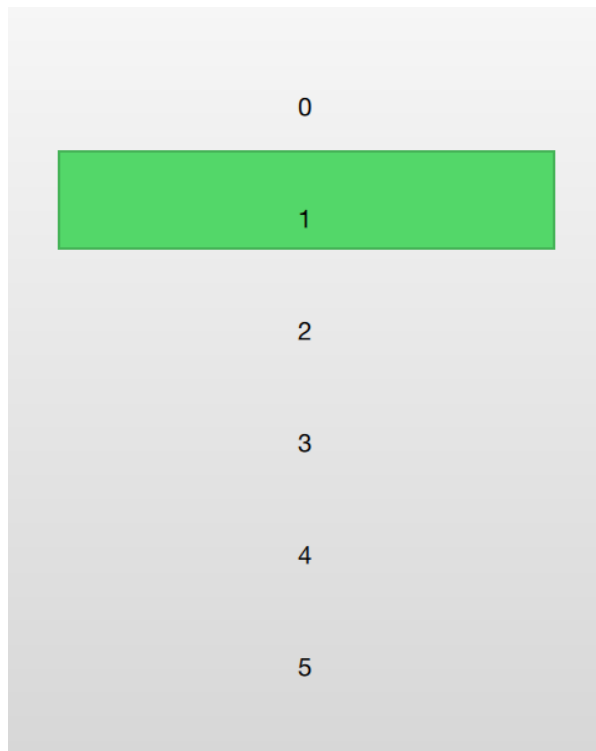
                font.pixelSize: 10

                text: index
            }
        }
    }
}
```

```

    }
}

```



When using a highlight in conjunction with a `ListView`, a number of properties can be used to control its behavior. The `highlightRangeMode` controls how the highlight is affected by what is shown in the view. The default setting, `ListView.NoHighlightRange` means that the highlight and the visible range of items in the view not are related at all.

The value `ListView.StrictlyEnforceRange` ensures that the highlight is always visible. If an action attempts to move the highlight outside the visible part of the view, the current item will change accordingly, so that the highlight remains visible.

The middle ground is the `ListView.ApplyRange` value. It attempts to keep the highlight visible, but does not alter the current item to enforce this. Instead, the highlight is allowed to move out of view if necessary.

In the default configuration, the view is responsible for moving the highlight into position. The speed of the movement and resizing can be controlled, either as a speed or as a duration. The properties involved are `highlightMoveSpeed`, `highlightMoveDuration`, `highlightResizeSpeed` and `highlightResizeDuration`. By default, the speed is set to 400 pixels per second, and the duration is set to -1, indicating that the speed and distance control the duration. If both a speed and a duration is set, the one that results in the quickest animation is chosen.

To control the movement of the highlight more in detail, the `highlightFollowCurrentItem` property can be set to `false`. This means that the view is no longer responsible for the movement of the highlight delegate. Instead, the movement can be controlled through a `Behavior` or an animation.

In the example below, the `y` property of the highlight delegate is bound to the `ListView.view.currentItem.y` attached property. This ensures that the highlight follows the current item. However, as we do not let the view move the highlight, we can control how the element is moved. This is done through the `Behavior` on `y`. In the example below, the movement is divided into three steps: fading out, moving, before fading in. Notice how `SequentialAnimation` and `PropertyAnimation` elements can be used in combination with the `NumberAnimation` to create a more complex movement.

```
Component {
    id: highlightComponent

    Item {
        width: ListView.view.width
        height: ListView.view.currentItem.height

        y: ListView.view.currentItem.y

        Behavior on y {
            SequentialAnimation {
                PropertyAnimation { target: highlightRectangle; property:
↪"opacity"; to: 0; duration: 200 }
                NumberAnimation { duration: 1 }
                PropertyAnimation { target: highlightRectangle; property:
↪"opacity"; to: 1; duration: 200 }
            }
        }

        GreenBox {
            id: highlightRectangle
            anchors.fill: parent
        }
    }
}
```

Header and Footer

At the end of the `ListView` contents, a `header` and a `footer` element can be inserted. These can be considered special delegates places at the beginning or end of the list. For a horizontal list, these will not appear at the head or foot, but rather at then beginning or end, depending on the `layoutDirection` used.

The example below illustrates how an header and footer can be used to enhance the perception of the beginning and end of a list. There are other uses for these special list elements. For instance, they can be used to keep buttons to load more contents.

```
import QtQuick 2.5
import "../common"

Background {
    width: 240
    height: 300

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 4

        delegate: numberDelegate
        spacing: 2

        header: headerComponent
        footer: footerComponent
    }

    Component {
        id: headerComponent
    }
}
```



```

    YellowBox {
        width: ListView.view.width
        height: 20
        text: 'Header'
    }
}

Component {
    id: footerComponent

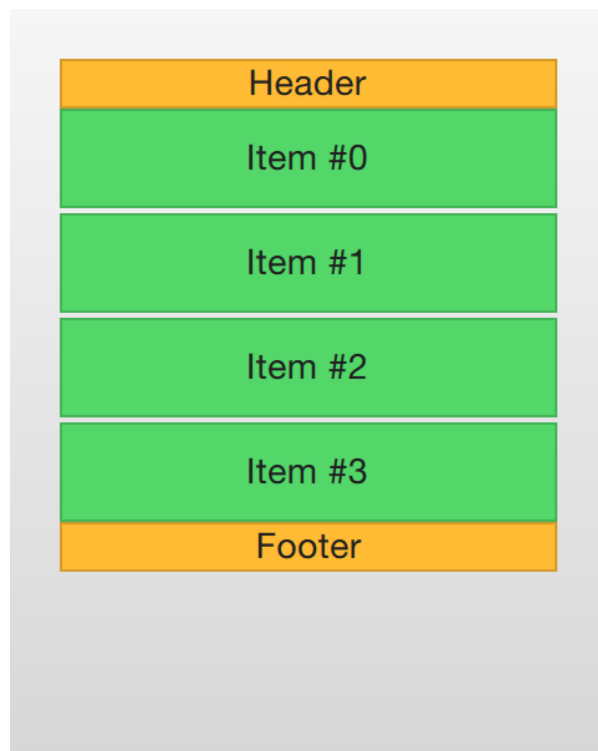
    YellowBox {
        width: ListView.view.width
        height: 20
        text: 'Footer'
    }
}

Component {
    id: numberDelegate

    GreenBox {
        width: ListView.view.width
        height: 40
        text: 'Item #' + index
    }
}
}

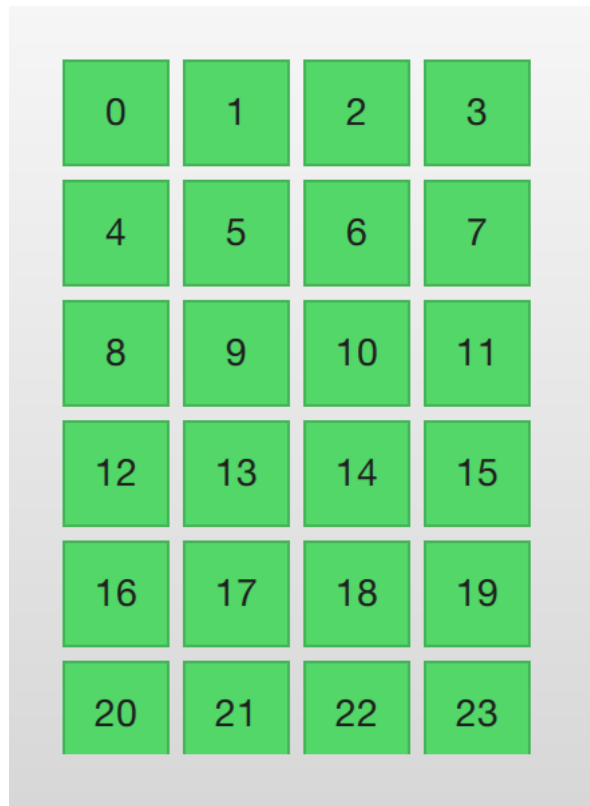
```

Note: Header and footer delegates do not respect the `spacing` property of a `ListView`, instead they are placed directly adjacent to the next item delegate in the list. This means that any spacing must be a part of the header and footer items.



The GridView

Using a `GridView` is very similar to using a `ListView`. The only real difference is that the grid view places the delegates in a two dimensional grid instead of in a linear list.



Compared to a list view, the grid view does not rely on spacing and the size of its delegates. Instead, it uses the `cellWidth` and `cellHeight` properties to control the dimensions of the contents delegates. Each delegate item is then places in the top left corner of each such cell.

```
import QtQuick 2.5
import "../common"

Background {
    width: 220
    height: 300

    GridView {
        id: view
        anchors.fill: parent
        anchors.margins: 20

        clip: true

        model: 100

        cellWidth: 45
        cellHeight: 45

        delegate: numberDelegate
    }

    Component {
        id: numberDelegate
```

```

        GreenBox {
            width: 40
            height: 40
            text: index
        }
    }
}

```

A `GridView` contains headers and footers, can use a highlight delegate and supports snap modes as well as various bounds behaviors. It can also be orientated in different directions and orientations.

The orientation is controlled using the `flow` property. It can be set to either `GridView.LeftToRight` or `GridView.TopToBottom`. The former value fills a grid from the left to the right, adding rows from the top to the bottom. The view is scrollable in the vertical direction. The latter value adds items from the top to the bottom, filling the view from left to right. The scrolling direction is horizontal in this case.

In addition to the `flow` property, the `layoutDirection` property can adapt the direction of the grid to left-to-right or right-to-left languages, depending on the value used.

Delegate

When it comes to using models and views in a custom user interface, the delegate plays a huge role in creating a look. As each item in a the model are visualized through a delegate, what is actually visible to the user are the delegates.

Each delegate gets access to a number of attached properties, some from the data model, others from the view. From the model, the properties convey the data for each item to the delegate. From the view, the properties convey state information related to the delegate within the view.

The most commonly used properties attached from the view are `ListView.isCurrentItem` and `ListView.view`. The first is a boolean indicating if the item is the current item, while the latter is a read-only reference to the actual view. Through access to the view, it is possible to create general, reusable delegates that adapt to the size and nature of the view in which they are contained. In the example below, the `width` of each delegate is bound to the `width` of the view, while the background `color` of each delegate depends on the attached `ListView.isCurrentItem` property.

```

import QtQuick 2.5

Rectangle {
    width: 120
    height: 300

    gradient: Gradient {
        GradientStop { position: 0.0; color: "#f6f6f6" }
        GradientStop { position: 1.0; color: "#d7d7d7" }
    }

    ListView {
        anchors.fill: parent
        anchors.margins: 20

        clip: true

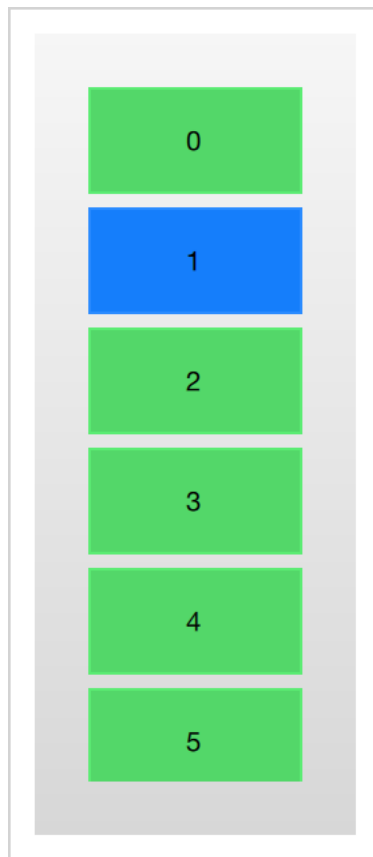
        model: 100

        delegate: numberDelegate
        spacing: 5

        focus: true
    }
}

```

```
Component {  
    id: numberDelegate  
  
    Rectangle {  
        width: ListView.view.width  
        height: 40  
  
        color: ListView.isCurrentItem? "#157efb": "#53d769"  
        border.color: Qt.lighter(color, 1.1)  
  
        Text {  
            anchors.centerIn: parent  
  
            font.pixelSize: 10  
  
            text: index  
        }  
    }  
}
```



If each item in the model is associated with an action, for instance, clicking an item acts upon it, that functionality is a part of each delegate. This divides the event management between the view, which handles the navigation between items in the view, and the delegate which handles actions on a specific item.

The most basic way to do this is to create a `MouseArea` within each delegate and act on the `onClicked` signal. This is demonstrated in the example in the next section of this chapter.

Animating Added and Removed Items

In some cases, the contents shown in a view changes over time. Items are added and removed as the underlying data model is altered. In these cases, it is often a good idea to employ visual cues to give the user a sense of direction and to help the user understand what data is added or removed.

Conveniently enough, QML views attaches two signals, `onAdd` and `onRemove`, to each item delegate. By connecting animations to these, it is easy to create the movement necessary to aid the user in identifying what is taking place.

The example below demonstrates this through the use of a dynamically populated `ListModel`. At the bottom of the screen, a button for adding new items is shown. When it is clicked, a new item is added to the model using the `append` method. This triggers the creation of a new delegate in the view, and the emission of the `GridView.onAdd` signal. The `SequentialAnimation` attached to the signal causes the item to zoom into view by animating the `scale` property of the delegate.

When a delegate in the view is clicked, the item is removed from the model through a call to the `remove` method. This causes the `GridView.onRemove` signal to be emitted, triggering another `SequentialAnimation`. This time, however, the destruction of the delegate must be delayed until the animation has completed. To do this, `PropertyAction` element are used to set the `GridView.delayRemove` property to `true` before the animation, and `false` after. This ensures that the animation is allowed to complete before the delegate item is removed.

```
import QtQuick 2.5

Rectangle {
    width: 480
    height: 300

    gradient: Gradient {
        GradientStop { position: 0.0; color: "#dbddde" }
        GradientStop { position: 1.0; color: "#5fc9f8" }
    }

    ListModel {
        id: theModel

        ListElement { number: 0 }
        ListElement { number: 1 }
        ListElement { number: 2 }
        ListElement { number: 3 }
        ListElement { number: 4 }
        ListElement { number: 5 }
        ListElement { number: 6 }
        ListElement { number: 7 }
        ListElement { number: 8 }
        ListElement { number: 9 }
    }

    Rectangle {
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        anchors.margins: 20

        height: 40

        color: "#53d769"
        border.color: Qt.lighter(color, 1.1)

        Text {
            anchors.centerIn: parent
```

```
        text: "Add item!"
    }

    MouseArea {
        anchors.fill: parent

        onClicked: {
            theModel.append({"number": ++parent.count});
        }
    }

    property int count: 9
}

GridView {
    anchors.fill: parent
    anchors.margins: 20
    anchors.bottomMargin: 80

    clip: true

    model: theModel

    cellWidth: 45
    cellHeight: 45

    delegate: numberDelegate
}

Component {
    id: numberDelegate

    Rectangle {
        id: wrapper

        width: 40
        height: 40

        gradient: Gradient {
            GradientStop { position: 0.0; color: "#f8306a" }
            GradientStop { position: 1.0; color: "#fb5b40" }
        }

        Text {
            anchors.centerIn: parent

            font.pixelSize: 10

            text: number
        }

        MouseArea {
            anchors.fill: parent

            onClicked: {
                theModel.remove(index);
            }
        }

        GridView.onRemove: SequentialAnimation {
            PropertyAction { target: wrapper; property: "GridView.delayRemove";
↪ value: true }
            NumberAnimation { target: wrapper; property: "scale"; to: 0; ↵
↪ duration: 250; easing.type: Easing.InOutQuad }
```

```

        PropertyAction { target: wrapper; property: "GridView.delayRemove";
→ value: false }
    }

    GridView.onAdd: SequentialAnimation {
        NumberAnimation { target: wrapper; property: "scale"; from: 0; to: 1;
→ duration: 250; easing.type: Easing.InOutQuad }
    }
}
}
}

```

Shape-Shifting Delegates

A commonly used mechanism in lists is that the current item is expanded when activated. This can be used to dynamically let the item expand to fill the screen to enter a new part of the user interface, or it can be used to provide slightly more information for the current item in a given list.

In the example below, each item is expanded to the full extent of the `ListView` containing it when clicked. The extra space is then used to add more information. The mechanism used to control this is a state, `expanded` that each item delegate can enter, where the item is expanded. In that state, a number of properties are altered.

First of all, the height of the wrapper is set to the height of the `ListView`. The thumbnail image is then enlarged and moved down to make it move from its small position into its larger position. In addition to this, the two hidden items, the `factsView` and `closeButton` are shown by altering the `opacity` of the elements. Finally, the `ListView` is setup.

Setting up the `ListView` involves setting the `contentsY`, that is the top of the visible part of the view, to the `y` value of the delegate. The other change is to set `interactive` of the view to `false`. This prevents the view from moving. The user can no longer scroll through the list or change the current item.

As the item first is clicked, it enters the `expanded` state, causing the item delegate to fill the `ListView` and the contents to rearrange. When the close button is clicked, the state is cleared, causing the delegate to return to its previous state and re-enabling the `ListView`.

```

import QtQuick 2.5

Item {
    width: 300
    height: 480

    Rectangle {
        anchors.fill: parent
        gradient: Gradient {
            GradientStop { position: 0.0; color: "#4a4a4a" }
            GradientStop { position: 1.0; color: "#2b2b2b" }
        }
    }

    ListView {
        id: listView

        anchors.fill: parent

        delegate: detailsDelegate
        model: planets
    }

    ListModel {
        id: planets
    }
}

```

```

        ListElement { name: "Mercury"; imageSource: "images/mercury.jpeg"; facts:
        ↪ "Mercury is the smallest planet in the Solar System. It is the closest planet to
        ↪ the sun. It makes one trip around the Sun once every 87.969 days." }
        ListElement { name: "Venus"; imageSource: "images/venus.jpeg"; facts:
        ↪ "Venus is the second planet from the Sun. It is a terrestrial planet because it
        ↪ has a solid, rocky surface. The other terrestrial planets are Mercury, Earth and
        ↪ Mars. Astronomers have known Venus for thousands of years." }
        ListElement { name: "Earth"; imageSource: "images/earth.jpeg"; facts: "The
        ↪ Earth is the third planet from the Sun. It is one of the four terrestrial
        ↪ planets in our Solar System. This means most of its mass is solid. The other
        ↪ three are Mercury, Venus and Mars. The Earth is also called the Blue Planet,
        ↪ 'Planet Earth', and 'Terra'." }
        ListElement { name: "Mars"; imageSource: "images/mars.jpeg"; facts: "Mars
        ↪ is the fourth planet from the Sun in the Solar System. Mars is dry, rocky and
        ↪ cold. It is home to the largest volcano in the Solar System. Mars is named after
        ↪ the mythological Roman god of war because it is a red planet, which signifies
        ↪ the colour of blood." }
    }

    Component {
        id: detailsDelegate

        Item {
            id: wrapper

            width: listView.width
            height: 30

            Rectangle {
                anchors.left: parent.left
                anchors.right: parent.right
                anchors.top: parent.top

                height: 30

                color: "#333"
                border.color: Qt.lighter(color, 1.2)
                Text {
                    anchors.left: parent.left
                    anchors.verticalCenter: parent.verticalCenter
                    anchors.leftMargin: 4

                    font.pixelSize: parent.height-4
                    color: '#fff'

                    text: name
                }
            }

            Rectangle {
                id: image

                width: 26
                height: 26

                anchors.right: parent.right
                anchors.top: parent.top
                anchors.rightMargin: 2
                anchors.topMargin: 2

                color: "black"
            }
        }
    }

```



```

        Image {
            anchors.fill: parent

            fillMode: Image.PreserveAspectFit

            source: imageSource
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: parent.state = "expanded"
    }

    Item {
        id: factsView

        anchors.top: image.bottom
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom

        opacity: 0

        Rectangle {
            anchors.fill: parent

            gradient: Gradient {
                GradientStop { position: 0.0; color: "#fed958" }
                GradientStop { position: 1.0; color: "#fecc2f" }
            }
            border.color: "#000000"
            border.width: 2

            Text {
                anchors.fill: parent
                anchors.margins: 5

                clip: true
                wrapMode: Text.WordWrap
                color: "#1f1f21"

                font.pixelSize: 12

                text: facts
            }
        }
    }

    Rectangle {
        id: closeButton

        anchors.right: parent.right
        anchors.top: parent.top
        anchors.rightMargin: 2
        anchors.topMargin: 2

        width: 26
        height: 26

        color: "#157efb"
        border.color: Qt.lighter(color, 1.1)
    }

```

```
        opacity: 0

        MouseArea {
            anchors.fill: parent
            onClicked: wrapper.state = ""
        }
    }

    states: [
        State {
            name: "expanded"

            PropertyChanges { target: wrapper; height: listView.height }
            PropertyChanges { target: image; width: listView.width; height:
→ listView.width; anchors.rightMargin: 0; anchors.topMargin: 30 }
            PropertyChanges { target: factsView; opacity: 1 }
            PropertyChanges { target: closeButton; opacity: 1 }
            PropertyChanges { target: wrapper.ListView.view; contentY:
→ wrapper.y; interactive: false }
        }
    ]

    transitions: [
        Transition {
            NumberAnimation {
                duration: 200;
                properties: "height,width,anchors.rightMargin,anchors.
→ topMargin,opacity,contentY"
            }
        }
    ]
}
}
```

The techniques demonstrated here to expand the delegate to fill the entire view can be employed to make an item delegate shift shape in a much smaller way. For instance, when browsing through a list of songs, the current item could be made slightly larger, accommodating more information about that particular item.

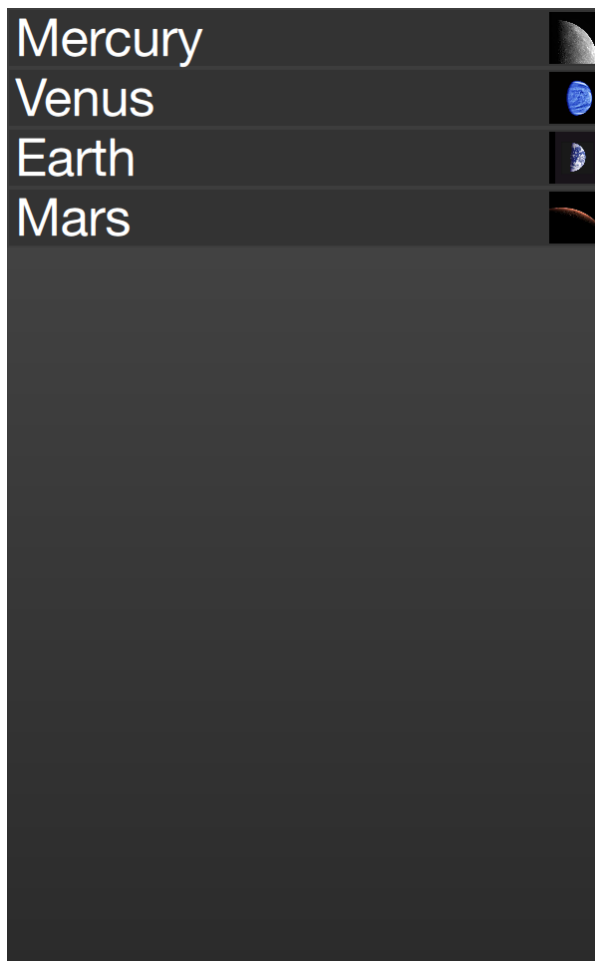
Advanced Techniques

The PathView

The `PathView` element is the most powerful, but also the most complex, view provided in Qt Quick. It makes it possible to create a view where the items are laid out along an arbitrary path. Along the same path, attributes such as scale, opacity and more can be controlled in detail.

When using the `PathView`, you have to define a delegate and a path. In addition to this, the `PathView` itself can be customized through a range of properties. The most common being `pathItemCount`, controlling the number of visible items at once, and the highlight range control properties `preferredHighlightBegin`, `preferredHighlightEnd` and `highlightRangeMode`, controlling where along the path the current item is to be shown.

Before looking at the highlight range control properties in depth, we must look at the path property. The path property expects a `Path` element defining the path that the delegates follows as the `PathView` is being scrolled. The path is defined using the `startX` and `startY` properties in combinations with path elements such as `PathLine`, `PathQuad` and `PathCubic`. These elements are joined together to form a two-dimensional path.





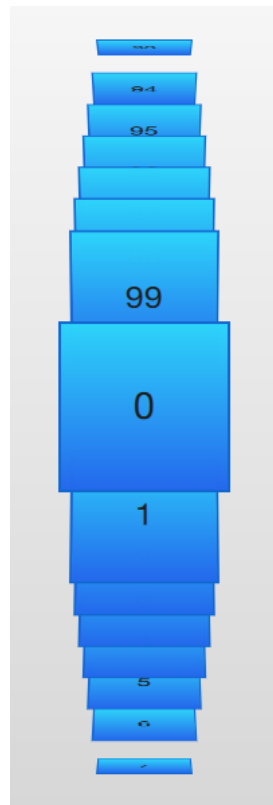
Todo

do we cover the line, quad and cubic through an illustration, or do we need a paragraph on them?

When the path has been defined, it is possible to further tune it using `PathPercent` and `PathAttribute` elements. These are placed in between path elements and provides a more fine grained control over the path and the delegates on it. The `PathPercent` controls how large a portion of the path that has been covered between each element. This, in turn, controls the distribution of delegates along the path, as they are distributed proportionally to the percentage progressed.

This is where the `preferredHighlightBegin` and `preferredHighlightEnd` properties of the `PathView` enters the picture. They both expect real values in the range between zero and one. The end is also expected to be more or equal to the beginning. Setting both these properties to, for instance, 0.5, the current item will be displayed at the location fifty percent along the path.

In the `Path`, the `PathAttribute` elements are placed between elements, just as `PathPercent` elements. They let you specify property values that are interpolated along the path. These properties are attached to the delegates and can be used to control any conceivable property.



The example below demonstrates how the `PathView` element is used to create view of cards that the user can flip through. It employs a number of tricks to do this. The path consists of three `PathLine` elements. Using `PathPercent` elements, the central element is properly centered and provided enough space not to be cluttered by other elements. Using `PathAttribute` elements, the rotation, size and z-value are controlled.

In addition to the path, the `pathItemCount` property of the `PathView` has been set. This controls how densely populated the path will be. The `preferredHighlightBegin` and `preferredHighlightEnd` the `PathView.onPath` is used to control the visibility of the delegates.

```
PathView {
    anchors.fill: parent

    delegate: flipCardDelegate
}
```

```

model: 100

path: Path {
    startX: root.width/2
    startY: 0

    PathAttribute { name: "itemZ"; value: 0 }
    PathAttribute { name: "itemAngle"; value: -90.0; }
    PathAttribute { name: "itemScale"; value: 0.5; }
    PathLine { x: root.width/2; y: root.height*0.4; }
    PathPercent { value: 0.48; }
    PathLine { x: root.width/2; y: root.height*0.5; }
    PathAttribute { name: "itemAngle"; value: 0.0; }
    PathAttribute { name: "itemScale"; value: 1.0; }
    PathAttribute { name: "itemZ"; value: 100 }
    PathLine { x: root.width/2; y: root.height*0.6; }
    PathPercent { value: 0.52; }
    PathLine { x: root.width/2; y: root.height; }
    PathAttribute { name: "itemAngle"; value: 90.0; }
    PathAttribute { name: "itemScale"; value: 0.5; }
    PathAttribute { name: "itemZ"; value: 0 }
}

pathItemCount: 16

preferredHighlightBegin: 0.5
preferredHighlightEnd: 0.5
}

```

The delegate, shown below, utilizes the attached properties `itemZ`, `itemAngle` and `itemScale` from the `PathAttribute` elements. It is worth noticing that the attached properties of the delegate only are available from the wrapper. Thus, the `rotX` property is defined to be able to access the value from within the `Rotation` element.

Another detail specific to `PathView` worth noticing is the usage of the attached `PathView.onPath` property. It is common practice to bind the visibility to this, as this allows the `PathView` to keep invisible elements for caching purposes. This can usually not be handled through clipping, as the item delegates of a `PathView` are placed more freely than the item delegates of `ListView` or `GridView` views.

```

Component {
    id: flipCardDelegate

    BlueBox {
        id: wrapper

        width: 64
        height: 64
        antialiasing: true

        gradient: Gradient {
            GradientStop { position: 0.0; color: "#2ed5fa" }
            GradientStop { position: 1.0; color: "#2467ec" }
        }

        visible: PathView.onPath

        scale: PathView.itemScale
        z: PathView.itemZ

        property variant rotX: PathView.itemAngle
        transform: Rotation {

```

```

        axis { x: 1; y: 0; z: 0 }
        angle: wrapper.rotX;
        origin { x: 32; y: 32; }
    }
    text: index
}
}

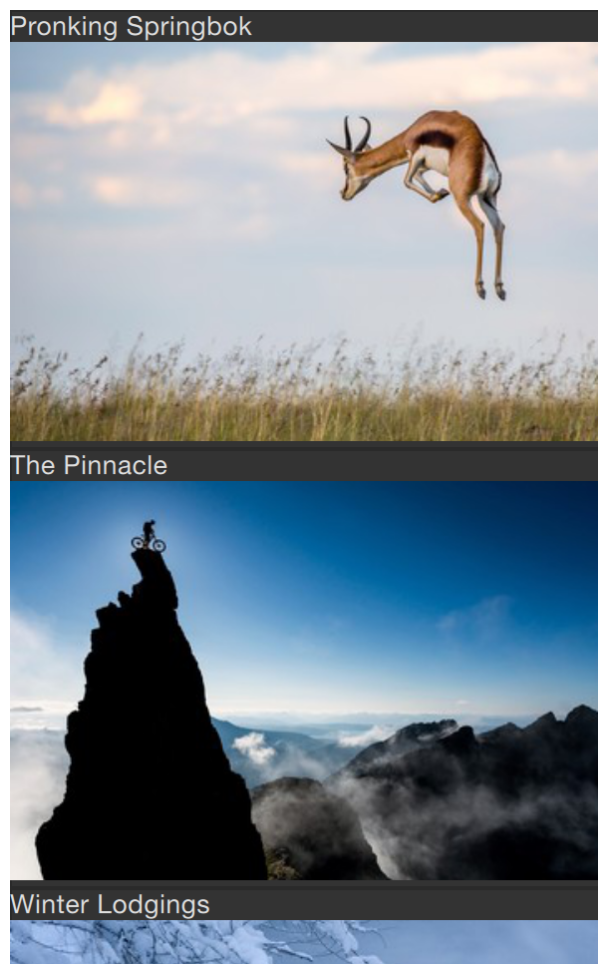
```

When transforming images or other complex elements on in `PathView`, a performance optimization trick that is common to use is to bind the `smooth` property of the `Image` element to the attached property `PathView.view.moving`. This means that the images are less pretty while moving, but smoothly transformed when stationary. There is no point spending processing power on smooth scaling when the view is in motion, as the user will not be able to see this anyway.

A Model from XML

As XML is an ubiquitous data format, QML provides the `XmlListModel` element that exposes XML data as a model. The element can fetch XML data locally or remotely and then processes the data using XPath expressions.

The example below demonstrates fetching images from an RSS flow. The `source` property refers to a remote location over HTTP, and the data is automatically downloaded.



When the data has been downloaded, it is processed into model items and roles. The `query` property is an XPath representing the base query for creating model items. In this example, the path is `/rss/channel/item`, so for every item tag, inside a channel tag, inside an RSS tag, a model item is created.

For every model item, a number of roles are extracted. These are represented by `XmlRole` elements. Each role is given a name, which the delegate can access through an attached property. The actual value of each such property is determined through the XPath query for each role. For instance, the `title` property corresponds to the `title/string()` query, returning the contents between the `<title>` and `</title>` tags.

The `imageSource` property is more interesting as it not only extracts a string from the XML, but also processes it. In the stream provided, every item contains an image, represented by an `<img src=` tag. Using the `substring-after` and `substring-before` XPath functions, the location of the image is extracted and returned. Thus the `imageSource` property can be used directly as the source for an `Image` element.

```
import QtQuick 2.5
import QtQuick.XmlListModel 2.0
import "../common"

Background {
    width: 300
    height: 480

    Component {
        id: imageDelegate

        Box {
            width: listView.width
            height: 220
            color: '#333'

            Column {
                Text {
                    text: title
                    color: '#e0e0e0'
                }
                Image {
                    width: listView.width
                    height: 200
                    fillMode: Image.PreserveAspectCrop
                    source: imageSource
                }
            }
        }
    }

    XmlListModel {
        id: imageModel

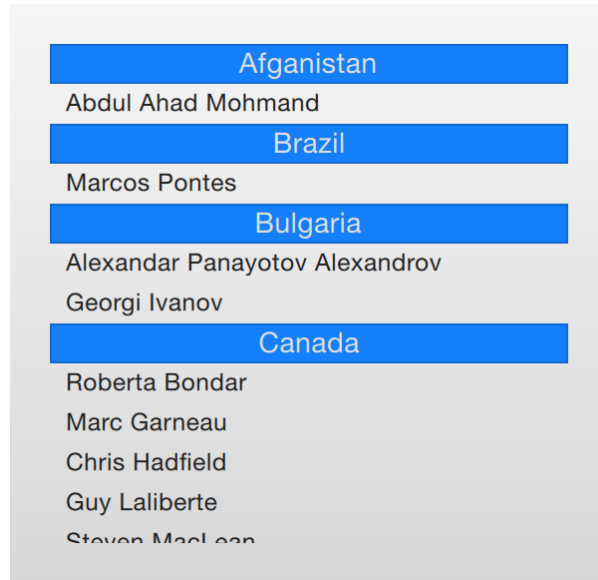
        source: "http://feeds.nationalgeographic.com/ng/photography/photo-of-the-
        ↪day/"
        query: "/rss/channel/item"

        XmlRole { name: "title"; query: "title/string()" }
        XmlRole { name: "imageSource"; query: "substring-before(substring-
        ↪after(description/string(), 'img src=\'\'), '\')" }
    }

    ListView {
        id: listView
        anchors.fill: parent
        model: imageModel
        delegate: imageDelegate
    }
}
```


Lists with Sections

Sometimes, the data in a list can be divided into sections. It can be as simple as dividing a list of contacts into sections under each letter of the alphabet or music tracks under albums. Using a `ListView` it is possible to divide a flat list into categories, providing more depth to the experience.



In order to use sections, the `section.property` and `section.criteria` must be setup. The `section.property` defines which property to use to divide the contents into sections. Here, it is important to know that the model must be sorted so that each section consists of continuous elements, otherwise, the same property name might appear in multiple locations.

The `section.criteria` can be set to either `ViewSection.FullString` or `ViewSection.FirstCharacter`. The first is the default value and can be used for models that have clear sections, for example tracks of music albums. The latter takes the first character of a property and means that any property can be used for this. The most common example being the last name of contacts in a phone book.

When the sections have been defined, they can be accessed from each item using the attached properties `ListView.section`, `ListView.previousSection` and `ListView.nextSection`. Using these properties, it is possible to detect the first and last item of a section and act accordingly.

It is also possible to assign a section delegate component to the `section.delegate` property of a `ListView`. This creates a section header delegate which is inserted before any items of a section. The delegate component can access the name of the current section using the attached property `section`.

The example below demonstrates the section concept by showing a list of space men sectioned after their nationality. The `nation` is used as the `section.property`. The `section.delegate` component, `sectionDelegate`, shows a heading for each nation, displaying the name of the nation. In each section, the names of the space men are shown using the `spaceManDelegate` component.

```
import QtQuick 2.5
import "../common"

Background {
    width: 300
    height: 290

    ListView {
        anchors.fill: parent
        anchors.margins: 20
```

```
        clip: true

        model: spaceMen

        delegate: spaceManDelegate

        section.property: "nation"
        section.delegate: sectionDelegate
    }

    Component {
        id: spaceManDelegate

        Item {
            width: ListView.view.width
            height: 20
            Text {
                anchors.left: parent.left
                anchors.verticalCenter: parent.verticalCenter
                anchors.leftMargin: 8
                font.pixelSize: 12
                text: name
                color: '#1f1f1f'
            }
        }
    }

    Component {
        id: sectionDelegate

        BlueBox {
            width: ListView.view.width
            height: 20
            text: section
            fontColor: '#e0e0e0'
        }
    }

    ListModel {
        id: spaceMen

        ListElement { name: "Abdul Ahad Mohmand"; nation: "Afganistan"; }
        ListElement { name: "Marcos Pontes"; nation: "Brazil"; }
        ListElement { name: "Alexandar Panayotov Alexandrov"; nation: "Bulgaria"; }
        ListElement { name: "Georgi Ivanov"; nation: "Bulgaria"; }
        ListElement { name: "Roberta Bondar"; nation: "Canada"; }
        ListElement { name: "Marc Garneau"; nation: "Canada"; }
        ListElement { name: "Chris Hadfield"; nation: "Canada"; }
        ListElement { name: "Guy Laliberte"; nation: "Canada"; }
        ListElement { name: "Steven MacLean"; nation: "Canada"; }
        ListElement { name: "Julie Payette"; nation: "Canada"; }
        ListElement { name: "Robert Thirsk"; nation: "Canada"; }
        ListElement { name: "Bjarni Tryggvason"; nation: "Canada"; }
        ListElement { name: "Dafydd Williams"; nation: "Canada"; }
    }
}
```

Tuning Performance

The perceived performance of a view of a model depends very much on the time needed to prepare new delegates. For instance, when scrolling downwards through a `ListView`, delegates are added just outside the view on the bottom and are removed just as they leave sight over the top of the view. This becomes apparent if the `clip` property is set to `false`. If the delegates takes too much time to initialize, it will become apparent to the user as soon as the view is scrolled too quickly.

To work around this issue you can tune the margins, in pixels, on the sides of a scrolling view. This is done using the `cacheBuffer` property. In the case described above, vertical scrolling, it will control how many pixels above and below the `ListView` that will contain prepared delegates. Combining this with asynchronously loading `Image` elements can, for instance, give the images time to load before they are brought into view.

Having more delegates sacrifices memory for a smoother experience and slightly more time to initialize each delegate. This does not solve the problem of complex delegates. Each time a delegate is instantiated, its contents is evaluated and compiled. This takes time, and if it takes too much time, it will lead to a poor scrolling experience. Having many elements in a delegate will also degrade the scrolling performance. It simply costs cycles to move many elements.

To remedy the two later issues, it is recommended to use `Loader` elements. These can be used to instantiate additional elements when they are needed. For instance, an expanding delegate may use a `Loader` to postpone the instantiation of its detailed view until it is needed. For the same reason, it is good to keep the amount of JavaScript to a minimum in each delegate. It is better to let them call complex pieced of JavaScript that reside outside each delegate. This reduces the time spent compiling JavaScript each time a delegate is created.

Summary

In this chapter, we have looked at models, views and delegates. For each data entry in a model, a view instantiates a delegate visualizing the data. This separates the data from the presentation.

A model can be a single integer, where the `index` variable is provided to the delegate. If a JavaScript array is used as model, the `modelData` variable represents the data of the current index of the array, while `index` holds the index. For more complex cases, where multiple values needs to be provided by each data item, a `ListModel` populated with `ListElement` items is a better solution.

For static models, a `Repeater` can be used as the view. It is easy to combine it with a positioner such as `Row`, `Column`, `Grid` or `Flow` to build user interface parts. For dynamic or large data models, a view such as `ListView` or `GridView` are more appropriate. These create delegate instances on the fly as they are needed, reducing the number of elements live in the scene at once.

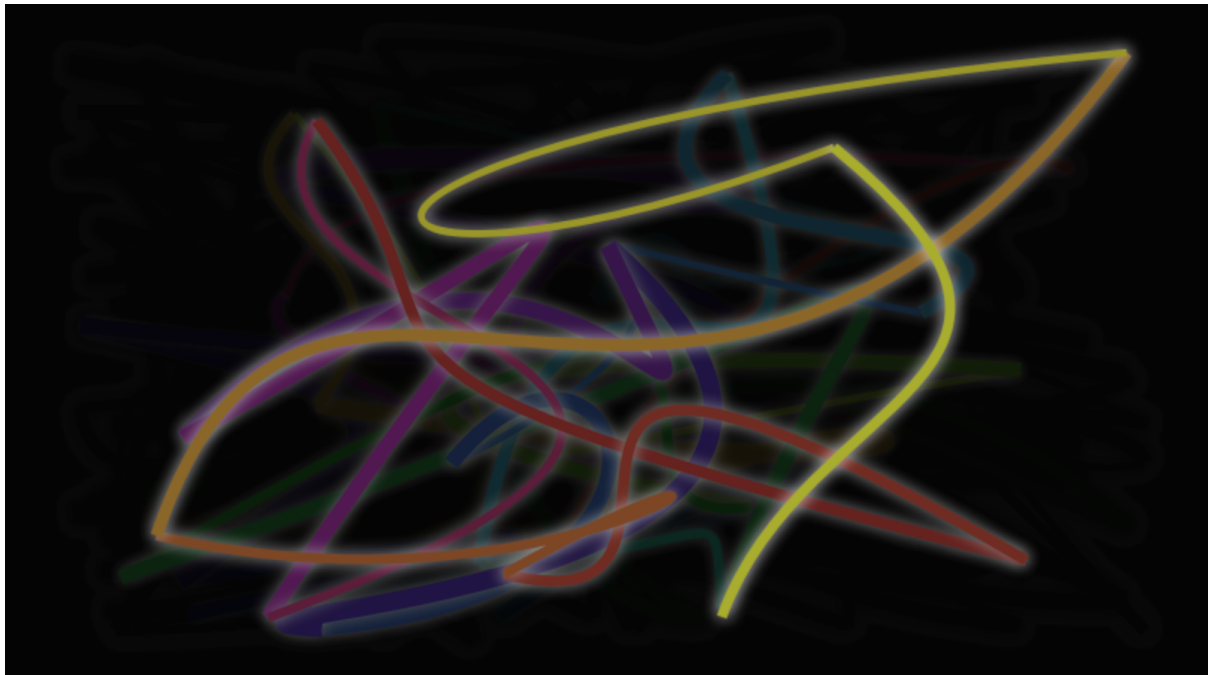
The delegates used in the views can be static items with properties bound to data from the model, or they can be dynamic, with states depending on if they are in focus or not. Using the `onAdd` and `onRemove` signals of the view, they can even be animated as they appear and disappear.

Canvas Element

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.



Early on when QML was introduced in Qt4 there were some discussions about if Qt Quick needs an ellipse. The problem with the ellipse is that others can argue other shapes need also be supported. So there is no ellipse in Qt Quick only rectangular shapes. If you needed one in Qt4 you would need to use an image or write your own C++ ellipse element.

To allow scripted drawings Qt 5 introduces the canvas element. The canvas element provides a resolution-dependent bitmap canvas, which can be used for graphics, games or to paint other visual images on the fly using JavaScript. The canvas element is based on the HTML5 canvas element.

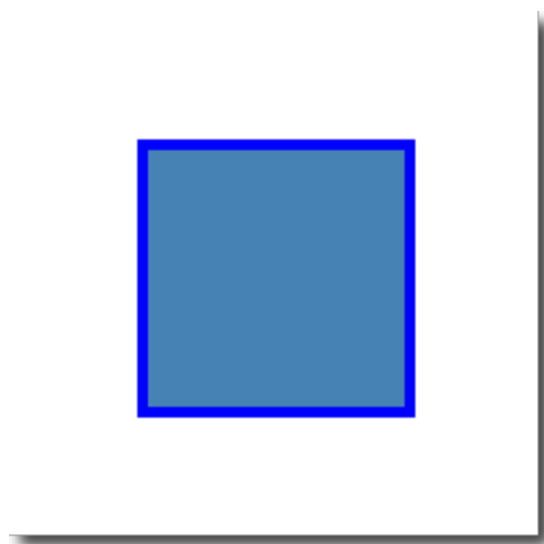
The fundamental idea of the canvas element is to render paths using a context 2D object. The context 2D object, contains the necessary graphics functions, whereas the canvas acts as the drawing canvas. The 2D context supports strokes, fills, gradients, text and a different sets of path creation commands.

Let's see an example of a simple path drawing:

```
import QtQuick 2.5

Canvas {
    id: root
    // canvas size
    width: 200; height: 200
    // handler to override for drawing
    onPaint: {
        // get context to draw with
        var ctx = getContext("2d")
        // setup the stroke
        ctx.lineWidth = 4
        ctx.strokeStyle = "blue"
        // setup the fill
        ctx.fillStyle = "steelblue"
        // begin a new path to draw
        ctx.beginPath()
        // top-left start point
        ctx.moveTo(50,50)
        // upper line
        ctx.lineTo(150,50)
        // right line
        ctx.lineTo(150,150)
        // bottom line
        ctx.lineTo(50,150)
        // left line through path closing
        ctx.closePath()
        // fill using fill style
        ctx.fill()
        // stroke using line width and stroke style
        ctx.stroke()
    }
}
```

This produces a filled rectangle with a starting point at 50,50 and a size of 100 and a stroke used as a border decoration.



The stroke width is set to 4 and uses a blue color define by `strokeStyle`. The final shape is setup to be filled through the `fillStyle` to a “steelblue” color. Only by calling `stroke` or `fill` the actual path will be drawn and they can be used independently from each other. A call to `stroke` or `fill` will draw the current path. It’s not possible to store a path for later reuse only a drawing state can be stored and restored.

In QML the `Canvas` element acts as a container for the drawing. The 2D context object provides the actual drawing operation. The actual drawing needs to be done inside the `onPaint` event handler.

```
Canvas {
    width: 200; height: 200
    onPaint: {
        var ctx = getContext("2d")
        // setup your path
        // fill or/and stroke
    }
}
```

The canvas itself provides a typical two dimensional Cartesian coordinate system, where the top-left is the (0,0) point. A higher y-value goes down and a high x-value goes to the right.

A typical order of commands for this path based API is the following:

1. Setup stroke and/or fill
2. Create path
3. Stroke and/or fill

```
onPaint: {
    var ctx = getContext("2d")

    // setup the stroke
    ctx.strokeStyle = "red"

    // create a path
    ctx.beginPath()
    ctx.moveTo(50,50)
    ctx.lineTo(150,50)

    // stroke path
    ctx.stroke()
}
```

This produces a horizontal stroked line from point P1 (50, 50) to point P2 (150, 50).



Note: Typically you always want to set a start point when you reset your path, so the first operation after `beginPath` is often `moveTo`.

Convenient API

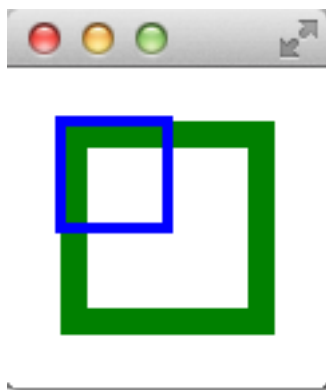
For operations on rectangles a convenience API is provided which draws directly and does need a stroke or fill call.

```
// convenient.qml
```

```
import QtQuick 2.5

Canvas {
    id: root
    width: 120; height: 120
    onPaint: {
        var ctx = getContext("2d")
        ctx.fillStyle = 'green'
        ctx.strokeStyle = "blue"
        ctx.lineWidth = 4

        // draw a filled rectangle
        ctx.fillRect(20, 20, 80, 80)
        // cut out an inner rectangle
        ctx.clearRect(30, 30, 60, 60)
        // stroke a border from top-left to
        // inner center of the larger rectangle
        ctx.strokeRect(20, 20, 40, 40)
    }
}
```



Note: The stroke area extends half of the line width on both sides of the path. A 4 px `lineWidth` will draw 2 px outside the path and 2 px inside.

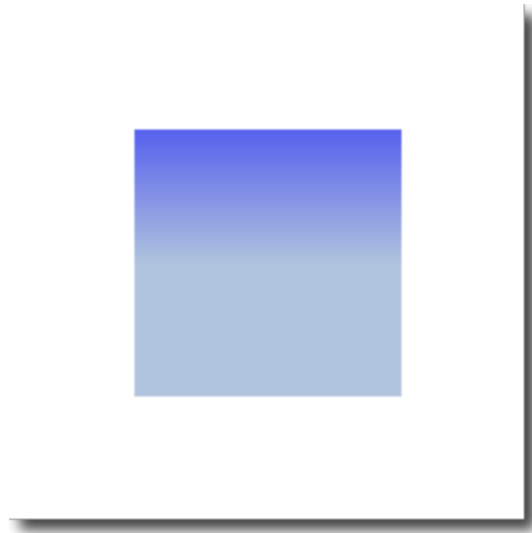
Gradients

Canvas can fill shapes with color but also with gradients or images.

```
onPaint: {
    var ctx = getContext("2d")

    var gradient = ctx.createLinearGradient(100, 0, 100, 200)
    gradient.addColorStop(0, "blue")
    gradient.addColorStop(0.5, "lightsteelblue")
    ctx.fillStyle = gradient
    ctx.fillRect(50, 50, 100, 100)
}
```

The gradient in this example is defined along the starting point (100,0) to the end point (100,200), which gives a vertical line in the middle of our canvas. The gradient stops can be defined as a color from 0.0 (gradient start point) to 1.0 (gradient end point). Here we use a “blue” color at 0.0 (100,0) and a “lightsteelblue” color at the 0.5 (100,200) position. The gradient is defined much larger than the rectangle we want to draw, so the rectangle clips the gradient to its defined geometry.



Note: The gradient is defined in canvas coordinates not in coordinates relative to the path to be painted. A canvas does not have the concept of relative coordinates, as we are used to by now from QML.

Shadows

A path can be visually enhanced using shadows with the 2D context object. A shadow is an area around the path with an offset, color and specified blurring. For this you need can specify a `shadowColor`, `shadowOffsetX`, `shadowOffsetY` and a `shadowBlur`. All of this needs to be defined using the 2D context. The 2D context is your only API to the drawing operations.

A shadow can also be used to create a glow effect around a path. In the next example we create a text “Canvas” with a white glow around. All this on a dark background for better visibility.

First we draw the dark background:

```
// setup a dark background
ctx.strokeStyle = "#333"
ctx.fillRect(0,0,canvas.width,canvas.height);
```

then we define our shadow configuration, which will be used for the next path:

```
// setup a blue shadow
ctx.shadowColor = "#2ed5fa";
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 10;
```

Finally we draw our “Canvas” text using a large bold 80px font from the *Ubuntu* font family.

```
// render green text
ctx.font = 'bold 80px Ubuntu';
ctx.fillStyle = "#24d12e";
ctx.fillText("Canvas!", 30, 180);
```



Images

The QML canvas supports image drawing from several sources. To use an image inside the canvas the image needs to be loaded first. We will use the `Component.onCompleted` handler to load the image in our example.

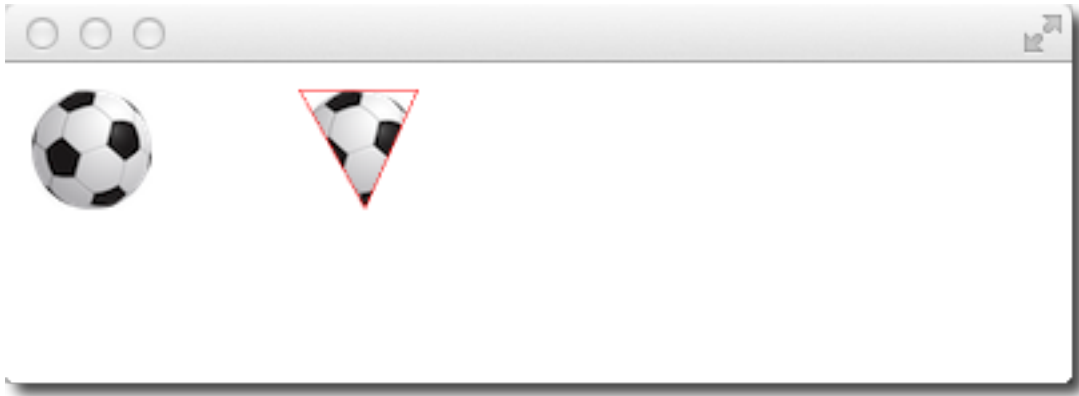
```
onPaint: {
    var ctx = getContext("2d")

    // draw an image
    ctx.drawImage('assets/ball.png', 10, 10)

    // store current context setup
    ctx.save()
    ctx.strokeStyle = '#ff2a68'
    // create a triangle as clip region
    ctx.beginPath()
    ctx.moveTo(110,10)
    ctx.lineTo(155,10)
    ctx.lineTo(135,55)
    ctx.closePath()
    // translate coordinate system
    ctx.clip() // create clip from the path
    // draw image with clip applied
    ctx.drawImage('assets/ball.png', 100, 10)
    // draw stroke around path
    ctx.stroke()
    // restore previous context
    ctx.restore()
}

Component.onCompleted: {
    loadImage("assets/ball.png")
}
```

The left shows our ball image painted at the top-left position of 10x10. The right image shows the ball with a clip path applied. Images and any other path can be clipped using another path. The clipping is applied by defining a path and calling the `clip()` function. All following drawing operations will now be clipped by this path. The clipping is disabled again by restoring the previous state or by setting the clip region to the whole canvas.



Transformation

The canvas allows you to transform the coordinate system in several ways. This is very similar to the transformation offered by QML items. You have the possibility to scale, rotate, translate the coordinate system. In difference to QML the transform origin is always the canvas origin. For example to scale a path around it's center you would need to translate the canvas origin to the center of the path. It is also possible to apply a more complex transformation using the transform method.

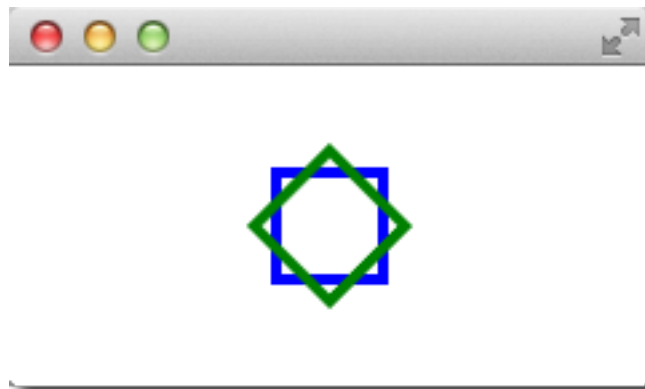
```
// transform.qml

import QtQuick 2.5

Canvas {
    id: root
    width: 240; height: 120
    onPaint: {
        var ctx = getContext("2d")
        ctx.strokeStyle = "blue"
        ctx.lineWidth = 4

        ctx.beginPath()
        ctx.rect(-20, -20, 40, 40)
        ctx.translate(120, 60)
        ctx.stroke()

        // draw path now rotated
        ctx.strokeStyle = "green"
        ctx.rotate(Math.PI/4)
        ctx.stroke()
    }
}
```



Besides translate the canvas allows also to scale using `scale(x,y)` around x and y axis, to rotate using `rotate(angle)`, where the angle is given in radius (*360 degree = $2 * \text{Math.PI}$*) and to use a matrix transformation using the `setTransform(m11,m12,m21,m22,dx,dy)`.

Note: To reset any transformation you can call the `resetTransform()` function to set the transformation matrix back to the identity matrix:

```
ctx.resetTransform()
```

Composition Modes

Composition allows you to draw a shape and blend it with the existing pixels. The canvas supports several composition modes using the `globalCompositeOperation(mode)` operation.

- source-over
- source-in
- source-out
- source-atop

```
onPaint: {
  var ctx = getContext("2d")
  ctx.globalCompositeOperation = "xor"
  ctx.fillStyle = "#33a9ff"

  for(var i=0; i<40; i++) {
    ctx.beginPath()
    ctx.arc(Math.random()*400, Math.random()*200, 20, 0, 2*Math.PI)
    ctx.closePath()
    ctx.fill()
  }
}
```

This little examples iterates over a list of composite modes and generates a rectangle with a circle.

```
property var operation : [
  'source-over', 'source-in', 'source-over',
  'source-atop', 'destination-over', 'destination-in',
  'destination-out', 'destination-atop', 'lighter',
  'copy', 'xor', 'qt-clear', 'qt-destination',
  'qt-multiply', 'qt-screen', 'qt-overlay', 'qt-darken',
  'qt-lighten', 'qt-color-dodge', 'qt-color-burn',
  'qt-hard-light', 'qt-soft-light', 'qt-difference',
  'qt-exclusion'
]

onPaint: {
  var ctx = getContext('2d')

  for(var i=0; i<operation.length; i++) {
    var dx = Math.floor(i%6)*100
    var dy = Math.floor(i/6)*100
    ctx.save()
    ctx.fillStyle = '#33a9ff'
    ctx.fillRect(10+dx,10+dy,60,60)
    // TODO: does not work yet
    ctx.globalCompositeOperation = root.operation[i]
    ctx.fillStyle = '#ff33a9'
```

```

        ctx.globalAlpha = 0.75
        ctx.beginPath()
        ctx.arc(60+dx, 60+dy, 30, 0, 2*Math.PI)
        ctx.closePath()
        ctx.fill()
        ctx.restore()
    }
}

```

Pixel Buffers

When working with the canvas you are able to retrieve pixel data from the canvas to read or manipulate the pixels of your canvas. To read the image data use `createImageData(sw,sh)` or `getImageData(sx,sy,sw,sh)`. Both functions return an `ImageData` object with a width, height and a data variable. The data variable contains a one-dimensional array of the pixel data retrieved in the *RGBA* format, where each value varies in the range of 0 to 255. To set pixels on the canvas you can use the `putImageData(imagedata,,dx,dy)` function.

Another way to retrieve the content of the canvas is to store the data into an image. This can be achieved with the Canvas functions `save(path)` or `toDataURL(mimeType)`, where the later function returns an image url, which can be used to be loaded by an Image element.

```

import QtQuick 2.5

Rectangle {
    width: 240; height: 120
    Canvas {
        id: canvas
        x: 10; y: 10
        width: 100; height: 100
        property real hue: 0.0
        onPaint: {
            var ctx = getContext("2d")
            var x = 10 + Math.random(80)*80
            var y = 10 + Math.random(80)*80
            hue += Math.random()*0.1
            if(hue > 1.0) { hue -= 1 }
            ctx.globalAlpha = 0.7
            ctx.fillStyle = Qt.hsla(hue, 0.5, 0.5, 1.0)
            ctx.beginPath()
            ctx.moveTo(x+5,y)
            ctx.arc(x,y, x/10, 0, 360)
            ctx.closePath()
            ctx.fill()
        }
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            var url = canvas.toDataURL('image/png')
            print('image url=', url)
            image.source = url
        }
    }
}

Image {
    id: image
    x: 130; y: 10
    width: 100; height: 100
}

```

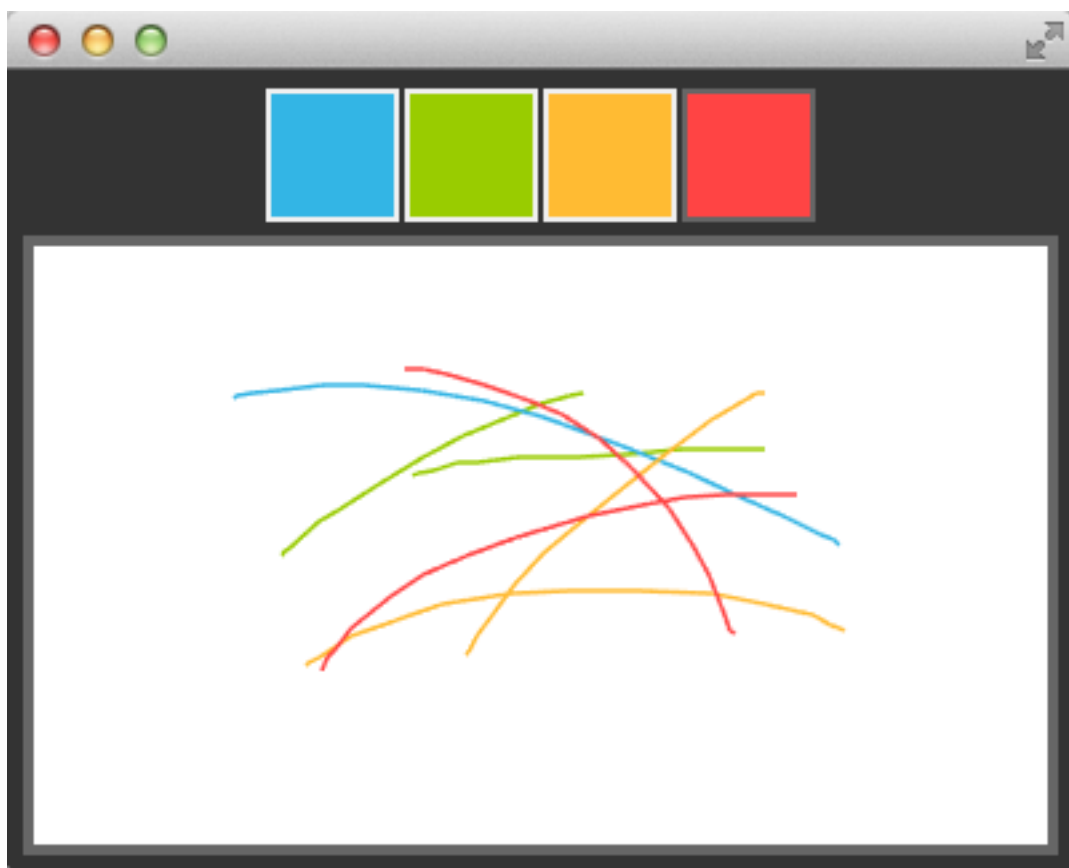
```
Timer {  
    interval: 1000  
    running: true  
    triggeredOnStart: true  
    repeat: true  
    onTriggered: canvas.requestPaint()  
}  
}
```

In our little example we paint every second a small circle in the left canvas. When the user clicks on the mouse area the canvas content is stored and a image url is retrieved. On the right side of our example the image is then displayed.

Note: Retrieving image data seems not to work currently in the Qt 5 Alpha SDK.

Canvas Paint

In this example we would like to create a small paint application using the `Canvas` element.



For this we arrange four color squares on the top of our scene using a row positioner. A color square is a simple rectangle filled with a mouse area to detect clicks.

```
Row {  
    id: colorTools  
    anchors {  
        horizontalCenter: parent.horizontalCenter  
    }  
}
```

```

        top: parent.top
        topMargin: 8
    }
    property variant activeSquare: red
    property color paintColor: "#33B5E5"
    spacing: 4
    Repeater {
        model: ["#33B5E5", "#99CC00", "#FFBB33", "#FF4444"]
        ColorSquare {
            id: red
            color: modelData
            active: parent.paintColor == color
            onClicked: {
                parent.paintColor = color
            }
        }
    }
}

```

The colors are stored in an array and the paint color. When one the user clicks in one of the squares the color of the square is assigned to the `paintColor` property of the row named `colorTools`.

To enable tracking of the mouse events on the canvas we have a `MouseArea` covering the canvas element and hooked up the pressed and position changed handlers.

```

Canvas {
    id: canvas
    anchors {
        left: parent.left
        right: parent.right
        top: colorTools.bottom
        bottom: parent.bottom
        margins: 8
    }
    property real lastX
    property real lastY
    property color color: colorTools.paintColor

    onPaint: {
        var ctx = getContext('2d')
        ctx.lineWidth = 1.5
        ctx.strokeStyle = canvas.color
        ctx.beginPath()
        ctx.moveTo(lastX, lastY)
        lastX = area.mouseX
        lastY = area.mouseY
        ctx.lineTo(lastX, lastY)
        ctx.stroke()
    }
    MouseArea {
        id: area
        anchors.fill: parent
        onPressed: {
            canvas.lastX = mouseX
            canvas.lastY = mouseY
        }
        onPositionChanged: {
            canvas.requestPaint()
        }
    }
}

```

A mouse press stores the initial mouse position into the `lastX` and `lastY` properties. Every change on the mouse

position triggers a paint request on the canvas, which will result into calling the *onPaint* handler.

To finally draw the users stroke, in the *onPaint* handler we begin a new path and move to the last position. Then we gather the new position from the mouse area and draw a line with the selected color to the new position. The mouse position is stored as the new *last* position.

Porting from HTML5 Canvas

- https://developer.mozilla.org/en/Canvas_tutorial/Transformations
- <http://en.wikipedia.org/wiki/Spirograph>

It is fairly easy to port a HTML5 canvas graphics over to use the QML canvas. From the thousands of examples, we picked one and tried it ourself.

Spiro Graph

We use a [spiro graph](#) example from the Mozilla project as our foundation. The original HTML5 was posted as part of the [canvas tutorial](#).

There where a few lines we needed to change:

- Qt Quick requires you to declare variable, so we needed to add some *var* declarations

```
for (var i=0;i<3;i++) {  
    ...  
}
```

- Adapted the draw method to receive the Context2D object

```
function draw(ctx) {  
    ...  
}
```

- we needed to adapt the translation for each spiro due to different sizes

```
ctx.translate(20+j*50,20+i*50);
```

Finally we inmpleted our *onPaint* handler. Inside we acquire a context and call our draw function.

```
onPaint: {  
    var ctx = getContext("2d");  
    draw(ctx);  
}
```

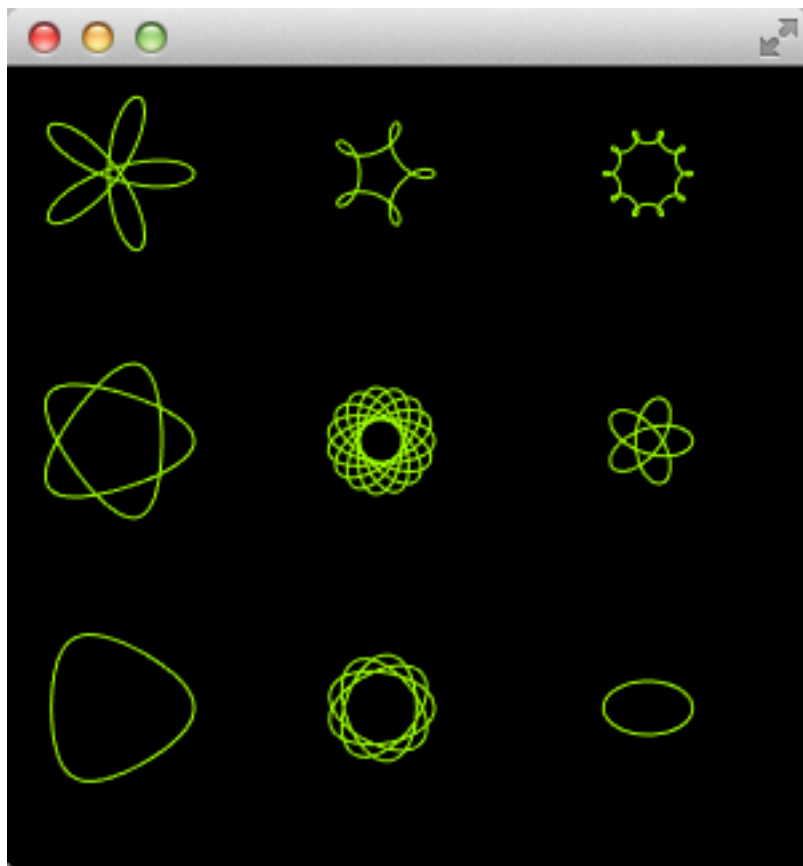
The result is a ported spiro graph graphics running using the QML canvas

That's all.

Glowing Lines

Here is another more complicated port from the W3C organization. The original [pretty glowing lines](#) has some pretty nice aspects, which makes the porting more challenging.

```
<!DOCTYPE HTML>  
<html lang="en">  
<head>  
    <title>Pretty Glowing Lines</title>  
</head>  
<body>  
  
<canvas width="800" height="450"></canvas>  
<script>  
var context = document.getElementsByTagName('canvas')[0].getContext('2d');
```

```
// initial start position
var lastX = context.canvas.width * Math.random();
var lastY = context.canvas.height * Math.random();
var hue = 0;

// closure function to draw
// a random bezier curve with random color with a glow effect
function line() {

    context.save();

    // scale with factor 0.9 around the center of canvas
    context.translate(context.canvas.width/2, context.canvas.height/2);
    context.scale(0.9, 0.9);
    context.translate(-context.canvas.width/2, -context.canvas.height/2);

    context.beginPath();
    context.lineWidth = 5 + Math.random() * 10;

    // our start position
    context.moveTo(lastX, lastY);

    // our new end position
    lastX = context.canvas.width * Math.random();
    lastY = context.canvas.height * Math.random();

    // random bezier curve, which ends on lastX, lastY
    context.bezierCurveTo(context.canvas.width * Math.random(),
        context.canvas.height * Math.random(),
        context.canvas.width * Math.random(),
        context.canvas.height * Math.random(),
        lastX, lastY);

    // glow effect
    hue = hue + 10 * Math.random();
    context.strokeStyle = 'hsl(' + hue + ', 50%, 50%)';
    context.shadowColor = 'white';
    context.shadowBlur = 10;
    // stroke the curve
    context.stroke();
    context.restore();
}

// call line function every 50msecs
setInterval(line, 50);

function blank() {
    // makes the background 10% darker on each call
    context.fillStyle = 'rgba(0,0,0,0.1)';
    context.fillRect(0, 0, context.canvas.width, context.canvas.height);
}

// call blank function every 50msecs
setInterval(blank, 40);

</script>
</body>
</html>
```

In HTML5 the Context2D object can paint at any time on the canvas. In QML it can only point inside the `onPaint` handler. The timer in usage with `setInterval` triggers in HTML5 the stroke of the line or to blank

the screen. Due to the different handling in QML it's not possible to just call these functions, because we need to go through the `onPaint` handler. Also the color presentations needs to be adapted. Let's go through the changes on by one.

Everything starts with the canvas element. For simplicity we just use the `Canvas` element as the root element of our QML file.

```
import QtQuick 2.5

Canvas {
    id: canvas
    width: 800; height: 450

    ...
}
```

To untangle the direct call of the functions through the `setInterval`, we replace the `setInterval` calls with two timers which will request a repaint. A `Timer` is triggered after a short interval and allows us to execute some code. As we can't tell the paint function which operation we would like trigger we define for each operation a bool flag request an operation and trigger then a repaint request.

Here is the code for the line operation. The blank operation is similar.

```
...
property bool requestLine: false

Timer {
    id: lineTimer
    interval: 40
    repeat: true
    triggeredOnStart: true
    onTriggered: {
        canvas.requestLine = true
        canvas.requestPaint()
    }
}

Component.onCompleted: {
    lineTimer.start()
}
...
```

Now we have a an indication which (line or blank or even both) operation we need to perform during the `onPaint` operation. As we enter the `onPaint` handler for each paint request we need to extract the initialization of the variable into the canvas element.

```
Canvas {
    ...
    property real hue: 0
    property real lastX: width * Math.random();
    property real lastY: height * Math.random();
    ...
}
```

Now our paint function should look like this:

```
onPaint: {
    var context = getContext('2d')
    if(requestLine) {
        line(context)
        requestLine = false
    }
    if(requestBlank) {
```

```
    blank(context)
    requestBlank = false
}
}
```

The *line* function was extracted for a canvas as argument.

```
function line(context) {
    context.save();
    context.translate(canvas.width/2, canvas.height/2);
    context.scale(0.9, 0.9);
    context.translate(-canvas.width/2, -canvas.height/2);
    context.beginPath();
    context.lineWidth = 5 + Math.random() * 10;
    context.moveTo(lastX, lastY);
    lastX = canvas.width * Math.random();
    lastY = canvas.height * Math.random();
    context.bezierCurveTo(canvas.width * Math.random(),
        canvas.height * Math.random(),
        canvas.width * Math.random(),
        canvas.height * Math.random(),
        lastX, lastY);

    hue += Math.random()*0.1
    if(hue > 1.0) {
        hue -= 1
    }
    context.strokeStyle = Qt.hsla(hue, 0.5, 0.5, 1.0);
    // context.shadowColor = 'white';
    // context.shadowBlur = 10;
    context.stroke();
    context.restore();
}
```

The biggest change was the use of the QML `Qt.rgb()` and `Qt.hsla()` functions, which required to adapt the values to the used 0.0 ... 1.0 range in QML.

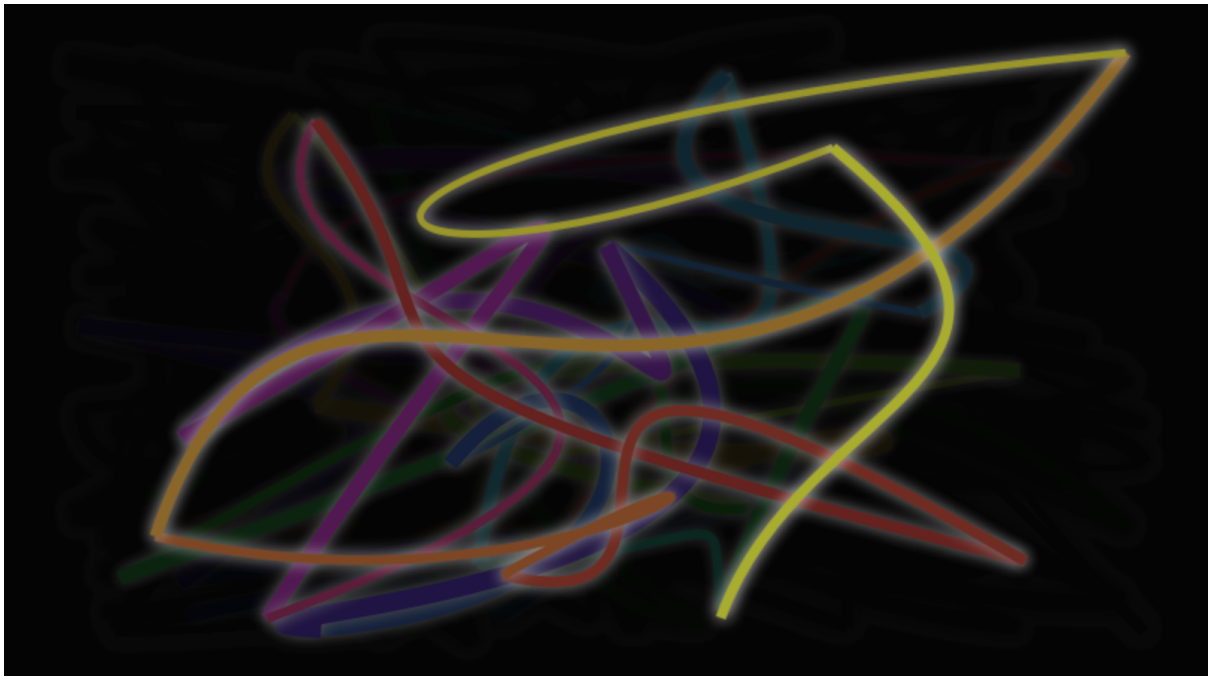
Same applies to the *blank* function.

```
function blank(context) {
    context.fillStyle = Qt.rgb(0,0,0,0.1)
    context.fillRect(0, 0, canvas.width, canvas.height);
}
```

The final result will look similar to this.

See also:

- [W3C HTML Canvas 2D Context Specification](#)
- [Mozilla Canvas Documentation](#)
- [HTML5 Canvas Tutorial](#)



Particle Simulations

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

Particles are a computer graphics techniques to visualize certain graphics effects. Typical effects could be: falling leaves, fire, explosions, meteors, clouds, etc.

It differs from other graphics rendering as particles rendering is based on fuzzy aspects. The outcome is not exactly predictable on pixel-base. Parameters to the particle system describe the boundaries for the stochastic simulation. The phenomena rendered with particles is often difficult to visualize with traditional rendering techniques. The good thing is you can let QML elements interact with the particles systems. Also as parameters are expressed as properties they can be animated using the traditional animation techniques.

Concept

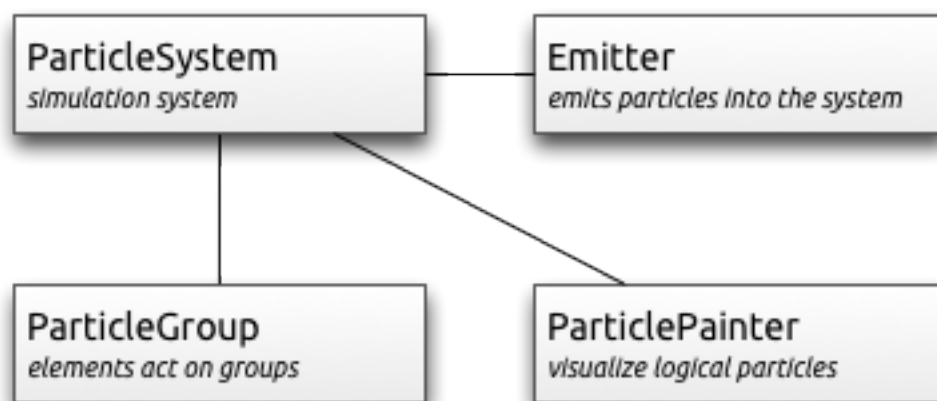
In the heart of the particle simulation is the `ParticleSystem` which controls the shared time-line. A scene can have several particles systems, each of them with an independent time-line. A particle is emitted using an `Emitter` element and visualized with a `ParticlePainter`, which can be an image, QML item or a shader item. An emitter provides also the direction for particle using a vector space. A particle ones emitted can't be manipulated by the emitter anymore. The particle module provides the `Affector`, which allows to manipulate parameters of the particle after is has been emitted.

Particles in a system can share timed transitions using the `ParticleGroup` element. By default every particle is on the empty (") group.

- `ParticleSystem` - manages shared time-line between emitters
- `Emitter` - emits logical particles into the system
- `ParticlePainter` - particles are visualized by a particle painter
- `Direction` - vector space for emitted particles
- `ParticleGroup` - every particle is a member of a group
- `Affector` - manipulates particles after they have been emitted

Simple Simulation

Let us have a look at a very simple simulation to get started. Qt Quick makes it actually very simple to get started with particle rendering. For this we need:



- A `ParticleSystem` which binds all elements to an simulation
- An `Emitter` which emits particles into the system
- A `ParticlePainter` derived element, which visualize the particles

```

import QtQuick 2.5
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 160
    color: "#1f1f1f"

    ParticleSystem {
        id: particleSystem
    }

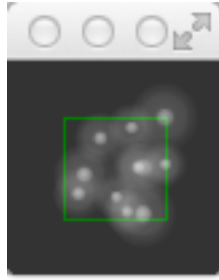
    Emitter {
        id: emitter
        anchors.centerIn: parent
        width: 160; height: 80
        system: particleSystem
        emitRate: 10
        lifeSpan: 1000
        lifeSpanVariation: 500
        size: 16
        endSize: 32
        Tracer { color: 'green' }
    }

    ImageParticle {
        source: "assets/particle.png"
        system: particleSystem
    }
}

```

The outcome of the example will look like this:

We start with a a 80x80 pixel dark rectangle as our root element and background. Therein we declare a `ParticleSystem`. This is always the first step as the system binds all other elements together. Typically the next element is the `Emitter`, which defines the emitting area based on it's bounding box and basic parameters for the to be emitted particles. The emitter is bound to the system using the `system` property.



The emitter in this example emits 10 particles per second (`emitRate: 10`) over the area of the emitter with each a life span of 1000 msec (`lifeSpan: 1000`) and a life span variation between emitted particles of 500 msec (`lifeSpanVariation: 500`). A particle shall start with a size of 16px (`size: 16`) and at the end of it's life shall be 32px (`endSize: 32`).

The green bordered rectangle is a tracer element to show the geometry of the emitter. This visualizes that also while the particles are emitted inside the emitters bounding box the rendering is not limited to the emitters bounding box. The rendering position depends upon life-span and direction of the particle. This will get more clear when we look into howto change the direction particles.

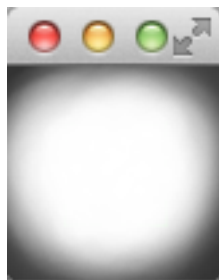
The emitter emits logical particles. A logical particle is visualized using a `ParticlePainter` in this example we use an `ImageParticle`, which takes an image URL as the source property. The image particle has also several other properties, which control the appearance of the average particle.

- `emitRate`: particles emitted per second (defaults to 10 per second)
- `lifeSpan`: milliseconds the particle should last for (defaults to 1000 msec)
- `size, endSize`: size of the particles at the beginning and end of their life (defaults to 16 px)

Changing these properties can influence the result in a drastically way

```
Emitter {
    id: emitter
    anchors.centerIn: parent
    width: 20; height: 20
    system: particleSystem
    emitRate: 40
    lifeSpan: 2000
    lifeSpanVariation: 500
    size: 64
    sizeVariation: 32
    Tracer { color: 'green' }
}
```

Besides increasing the emit rate to 40 and the life span to 2 seconds the size now starts at 64 pixel and decreases 32 pixel at the end of a particle life span.



Increasing the `endSize` even more would lead to a more or less white background. Please note also when the particles are only emitted in the area defined by the emitter the rendering is not constrained to it.

Particle Parameters

We saw already how to change the behavior of the emitter to change our simulation. The particle painter used allows us how the particle image is visualized for each particle.

Coming back to our example we update our `ImageParticle`. First we change our particle image to a small sparking star image:

```
ImageParticle {  
    ...  
    source: 'assets/star.png'  
}
```

The particle shall be colored in an gold color which varies from particle to particle by +/- 20%:

```
color: '#FFD700'  
colorVariation: 0.2
```

To make the scene more alive we would like to rotate the particles. Each particle should start by 15 degrees clockwise and varies between particles by +/-5 degrees. Additional the particle should continuously rotate with the velocity of 45 degrees per second. The velocity shall also vary from particle to particle by +/- 15 degrees per second:

```
rotation: 15  
rotationVariation: 5  
rotationVelocity: 45  
rotationVelocityVariation: 15
```

Last but not least, we change the entry effect for the particle. This is the effect used when a particle comes to life. In this case we want to use the scale effect:

```
entryEffect: ImageParticle.Scale
```

So now we have rotating golden stars appearing all over the place.



Here is the code we changed for the image particle in one block.

```
ImageParticle {  
    source: "assets/star.png"  
    system: particleSystem  
    color: '#FFD700'  
    colorVariation: 0.2  
    rotation: 0  
    rotationVariation: 45
```

```

rotationVelocity: 15
rotationVelocityVariation: 15
entryEffect: ImageParticle.Scale
}

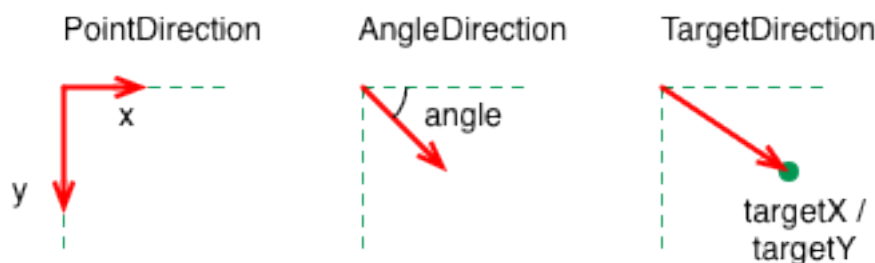
```

Directed Particles

We have seen particles can rotate. But particles can also have a trajectory. The trajectory is specified as the velocity or acceleration of particles defined by a stochastic direction also named a vector space.

There are different vector spaces available to define the velocity or acceleration of a particle:

- `AngleDirection` - a direction that varies in angle
- `PointDirection` - a direction that varies in x and y components
- `TargetDirection` - a direction towards the target point



Let's try to move the particles over from the left to the right side of our scene by using the velocity directions.

We first try the `AngleDirection`. For this we need to specify the `AngleDirection` as an element of the velocity property of our emitter:

```
velocity: AngleDirection { }
```

The angle where the particles are emitted is specified using the angle property. The angle is provided as value between 0..360 degree and 0 points to the right. For our example we would like the particles to move to the right so 0 is already the right direction. The particles shall spread by +/- 5 degree:

```

velocity: AngleDirection {
    angle: 0
    angleVariation: 15
}

```

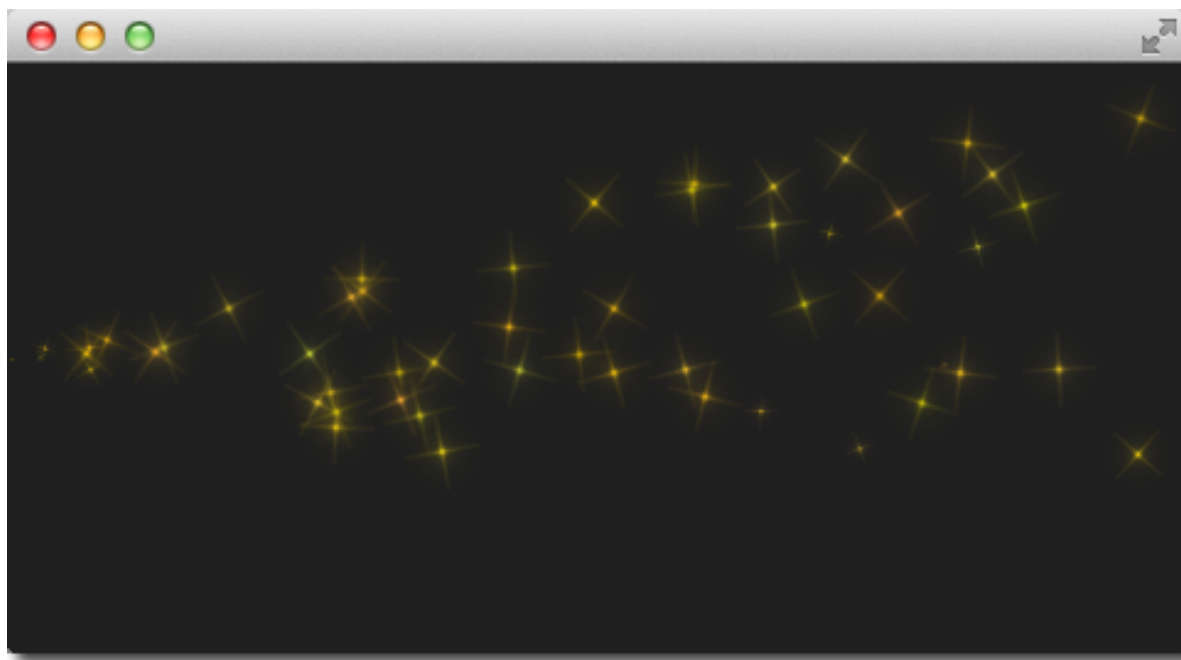
Now we have set our direction, the next thing is to specify the velocity of the particle. This is defined by a magnitude. The magnitude is defined in pixels per seconds. As we have ca. 640px to travel 100 seems to be a good number. This would mean by an average life time of 6.4 secs a particle would cross the open space. To make the traveling of the particles more interesting we vary the magnitude using the `magnitudeVariation` and set this to the half of the magnitude:

```

velocity: AngleDirection {
    ...
    magnitude: 100
    magnitudeVariation: 50
}

```

Here is the full source code, with an average life time set to 6.4 seconds. We set the emitter width and height to 1px. This means all particles are emitted at the same location and from thereon travel based on our given trajectory.



```

Emitter {
    id: emitter
    anchors.left: parent.left
    anchors.verticalCenter: parent.verticalCenter
    width: 1; height: 1
    system: particleSystem
    lifeSpan: 6400
    lifeSpanVariation: 400
    size: 32
    velocity: AngleDirection {
        angle: 0
        angleVariation: 15
        magnitude: 100
        magnitudeVariation: 50
    }
}

```

So what is then the acceleration doing? The acceleration add a acceleration vector to each particle, which changes the velocity vector over time. For example let's make a trajectory like an arc of stars. For this we change our velocity direction to -45 degree and remove the variations, to better visualize a coherent arc:

```

velocity: AngleDirection {
    angle: -45
    magnitude: 100
}

```

The acceleration direction shall be 90 degree (down direction) and we choose one fourth of the velocity magnitude for this:

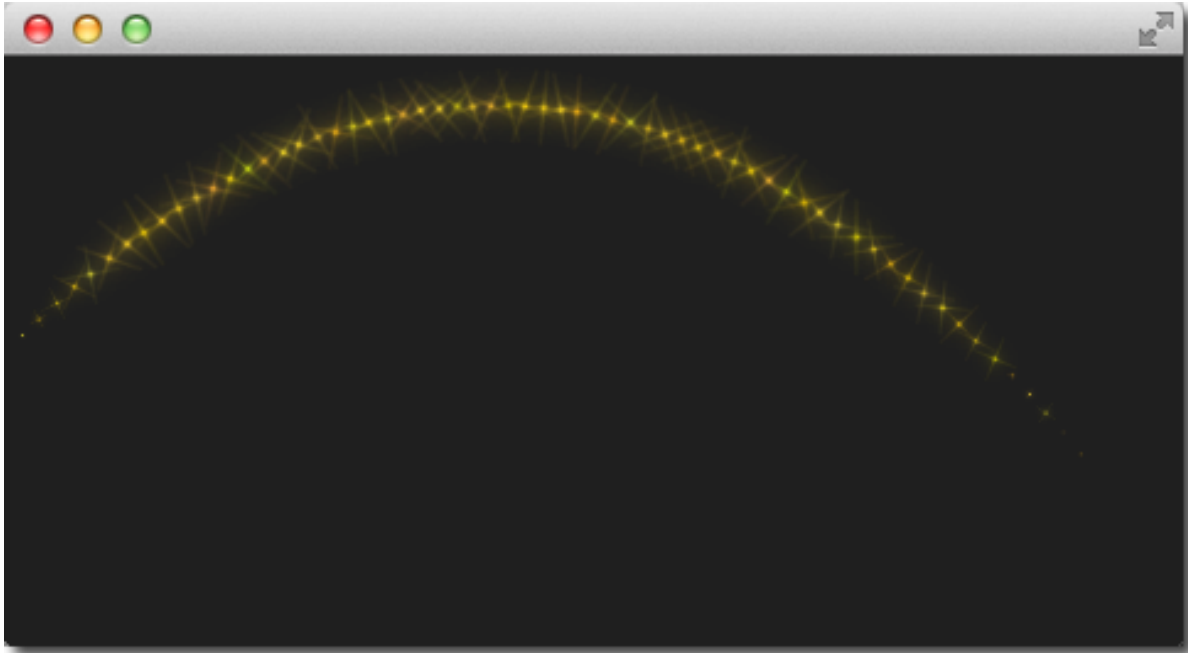
```

acceleration: AngleDirection {
    angle: 90
    magnitude: 25
}

```

The result is an arc going from the center left to the bottom right.

The values are discovered by try-and-error.



Here is the full code of our emitter.

```
Emitter {
    id: emitter
    anchors.left: parent.left
    anchors.verticalCenter: parent.verticalCenter
    width: 1; height: 1
    system: particleSystem
    emitRate: 10
    lifeSpan: 6400
    lifeSpanVariation: 400
    size: 32
    velocity: AngleDirection {
        angle: -45
        angleVariation: 0
        magnitude: 100
    }
    acceleration: AngleDirection {
        angle: 90
        magnitude: 25
    }
}
```

In the next example we would like that the particles again travel from left to right but this time we use the `PointDirection` vector space.

A `PointDirection` derived it's vector space from a x and y component. For example if you want the particles travel in a 45 degree vector, you need to specify the same value for x and y.

In our case we want the particles travel from left-to-right building a 15 degree cone. For this we specify a `PointDirection` as our velocity vector space:

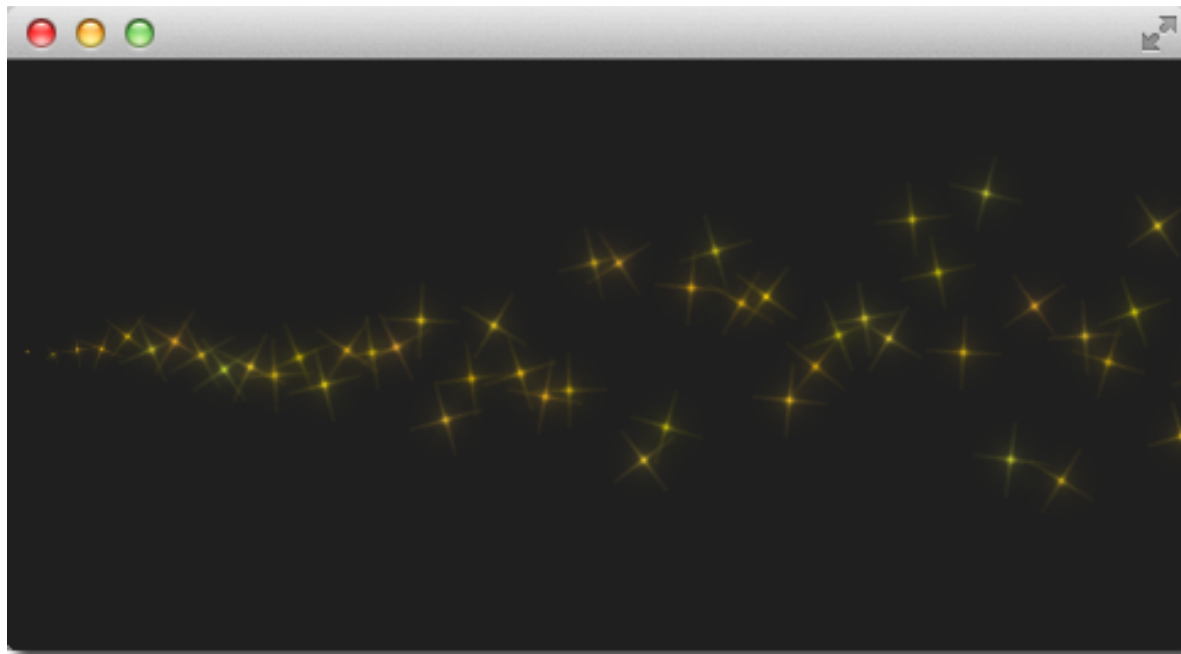
```
velocity: PointDirection { }
```

To achieve a traveling velocity of 100 px per seconds we set our x component to 100. For the 15 degree (which is 1/6 th of 90 degree) we specify an y variation of 100/6:

```
velocity: PointDirection {
    x: 100
```

```
y: 0
xVariation: 0
yVariation: 100/6
}
```

The result should be particles traveling in a 15 degree cone from right to left.



Now coming to our last contender, the `TargetDirection`. The target direction allows us to specify a target point as an x and y coordinate relative to the emitter or an item. When an item is specified the center of the item will become the target point. You can achieve the 15 degree cone by specifying a target variation of 1/6 th of the x target:

```
velocity: TargetDirection {
    targetX: 100
    targetY: 0
    targetVariation: 100/6
    magnitude: 100
}
```

Note: Target direction are great to use when you have a specific x/y coordinate you want the stream of particles emitted towards.

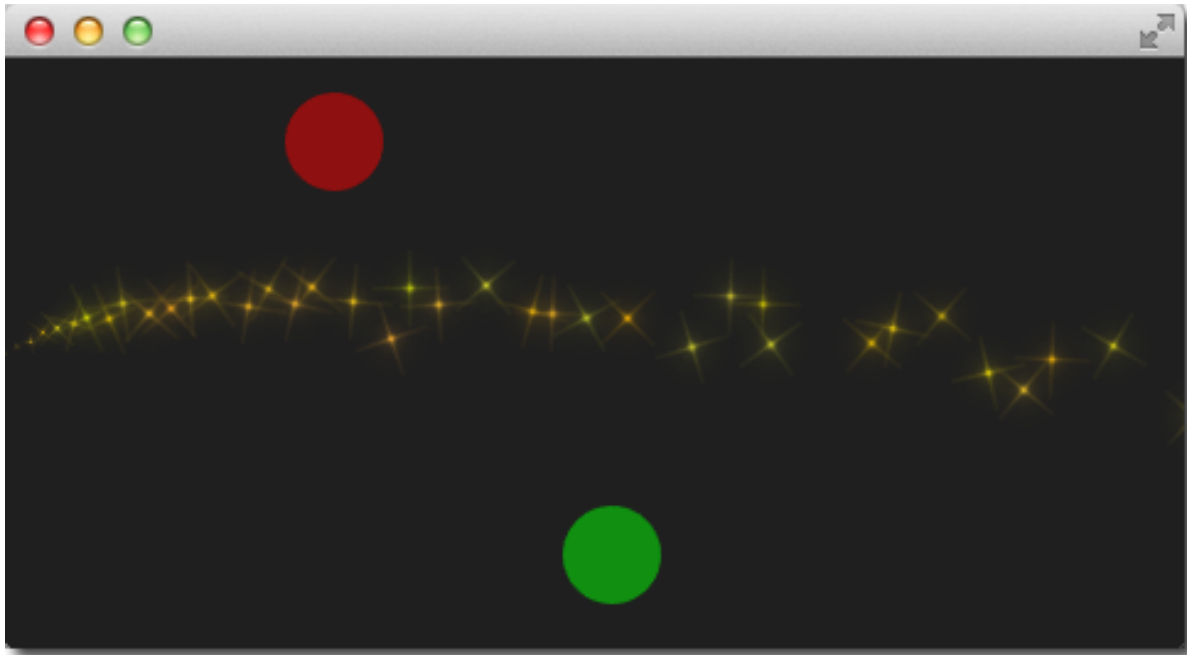
I spare you the image as it looks the same as the previous one, instead I have a quest for you.

In the following image the red and the green circle specify each a target item for the target direction of the velocity respective the acceleration property. Each target direction has the same parameters. Here the question: Who is responsible for velocity and who is for acceleration?

Particle Painters

Till now we have only used the image based particle painter to visualize particles. Qt comes also with other particle painters:

- `ItemParticle`: delegate based particle painter



- CustomParticle: shader based particle painter

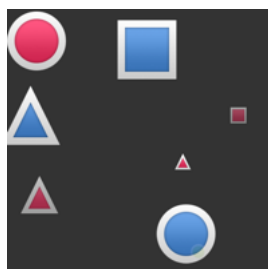
The ItemParticle can be used to emit QML items as particles. For this you need to specify your own delegate to the particle.

```
ItemParticle {
    id: particle
    system: particleSystem
    delegate: itemDelegate
}
```

Our delegate in this case is a random image (using *Math.random()*), visualized with a white border and a random size.

```
Component {
    id: itemDelegate
    Item {
        id: container
        width: 32*Math.ceil(Math.random()*3); height: width
        Image {
            anchors.fill: parent
            anchors.margins: 4
            source: 'assets/'+images[Math.floor(Math.random()*9)]
        }
    }
}
```

We emit 4 images per second with a life span of 4 seconds each. The particles fade automatically in and out.



For more dynamic cases it is also possible to create an item on your own and let the particle take control of it with `take(item, priority)`. By this the particle simulation takes control of your particle and handles the item like an ordinary particle. You can get back control of the item by using `give(item)`. You can influence item particles even more by halt their life progression using `freeze(item)` and resume their life using `unfreeze(item)`.

Affecting Particles

Particles are emitted by the emitter. After a particle was emitted it can't be changed anymore by the emitter. The `affector` allows you to influence particles after they have been emitted.

Each type of affector affects particles in a different way:

- `Age` - alter where the particle is in its life-cycle
- `Attractor` - attract particles towards a specific point
- `Friction` - slows down movement proportional to the particle's current velocity
- `Gravity` - set's an acceleration in an angle
- `Turbulence` - fluid like forces based on a noise image
- `Wander` - randomly vary the trajectory
- `GroupGoal` - change the state of a group of a particle
- `SpriteGoal` - change the state of a sprite particle

Age

Allows particle to age faster. the *lifeLeft* property specified how much life a particle should have left.

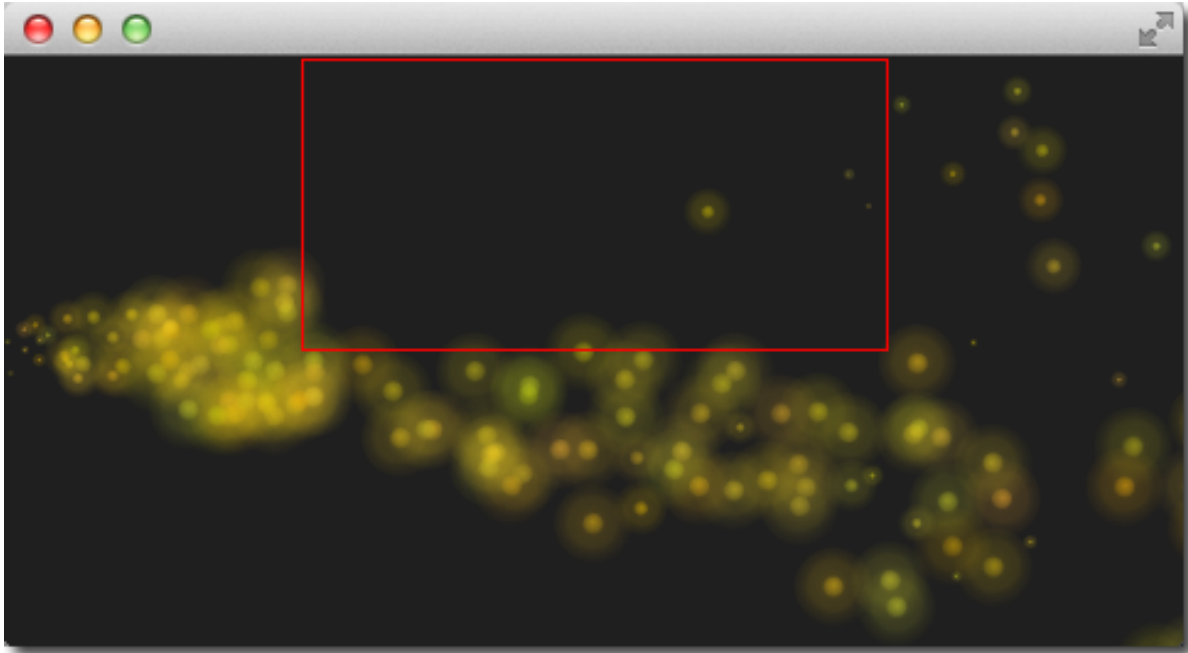
```
Age {  
    anchors.horizontalCenter: parent.horizontalCenter  
    width: 240; height: 120  
    system: particleSystem  
    advancePosition: true  
    lifeLeft: 1200  
    once: true  
    Tracer {}  
}
```

In the example we shorten the life of the upper particles once, when they reach the age affector to 1200 msecs. As we have set the *advancePosition* to true, we see the particle appearing again on a position when the particle has 1200 msecs left to life.

Attractor

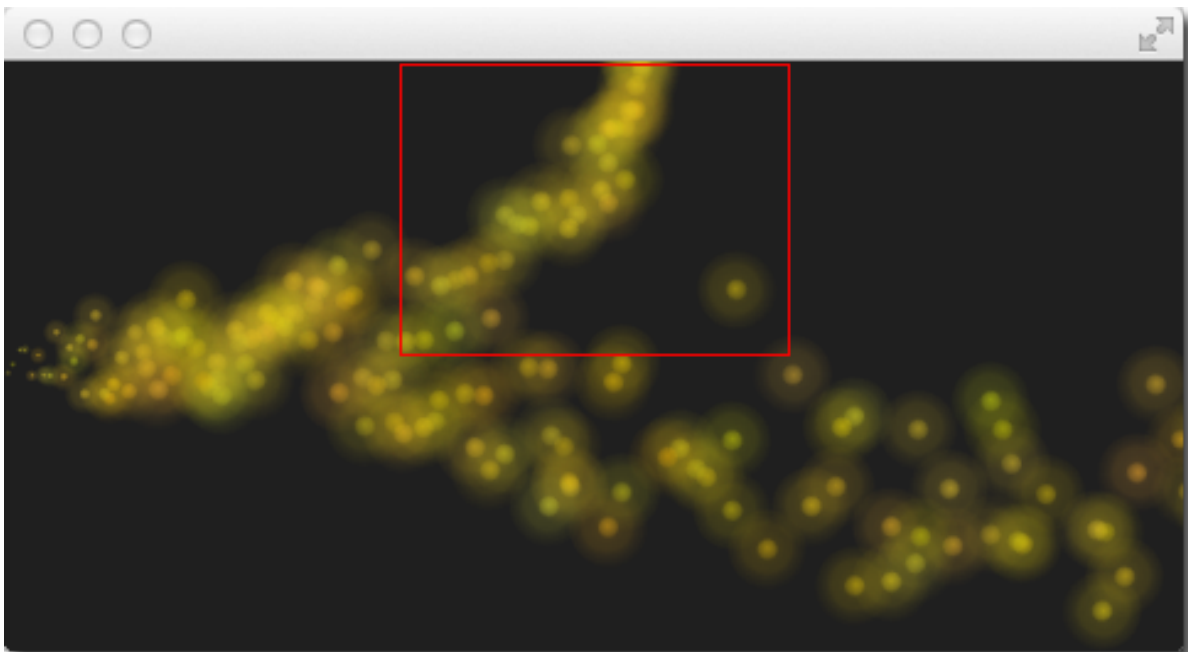
The attractor attracts particles towards a specific point. The point is specified using `pointX` and `pointY`, which is relative to the attractor geometry. The strength specifies the force of attraction. In our example we let particles travel from left to right. The attractor is placed on the top and half of the particles travel through the attractor. Affector only affect particles while they are in their bounding box. This split allows us to see the normal stream and the affected stream simultaneous.

```
Attractor {  
    anchors.horizontalCenter: parent.horizontalCenter  
    width: 160; height: 120  
    system: particleSystem  
    pointX: 0  
    pointY: 0  
}
```

```
strength: 1.0  
Tracer {}  
}
```

It's easy to see that the upper half of the particles are affected by the attractor to the top. The attraction point is set to top-left (0/0 point) of the attractor with a force of 1.0.

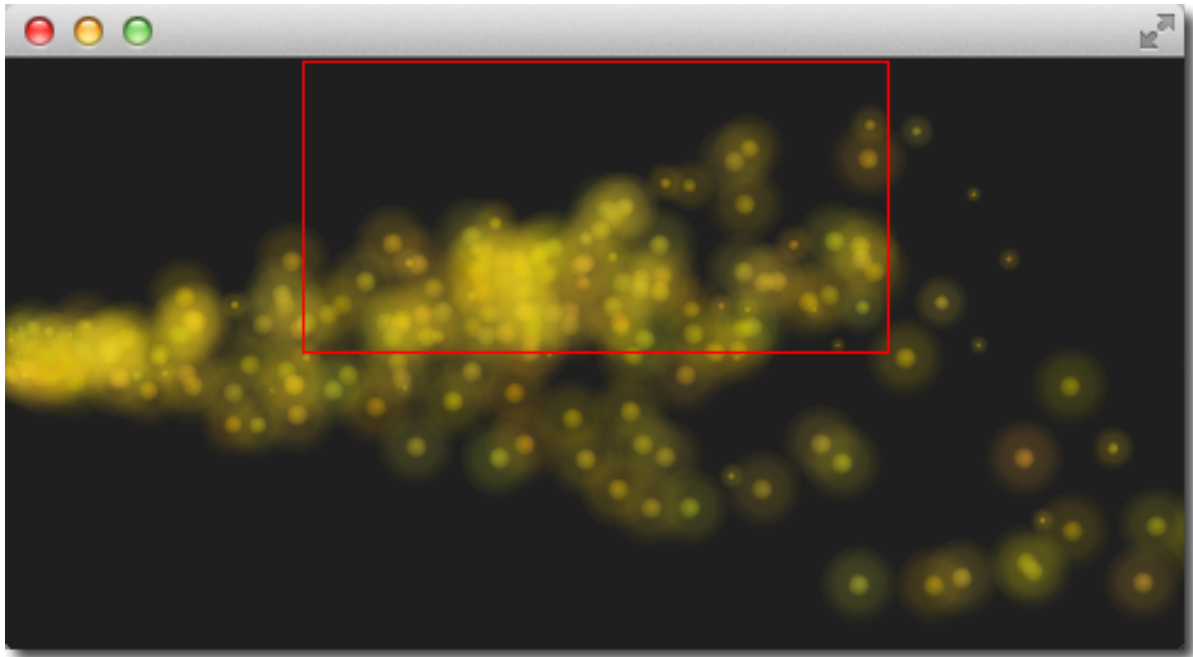


Friction

The friction affector slows down particles by a factor until a certain threshold is reached.

```
Friction {  
    anchors.horizontalCenter: parent.horizontalCenter  
    width: 240; height: 120  
    system: particleSystem  
    factor : 0.8  
    threshold: 25  
    Tracer {}  
}
```

In the upper friction area, the particles are slowed down by a factor of 0.8 until the particle reach 25 pixels per seconds velocity. The threshold act's like a filter. Particles traveling above the threshold velocity are slowed down by the given factor.



Gravity

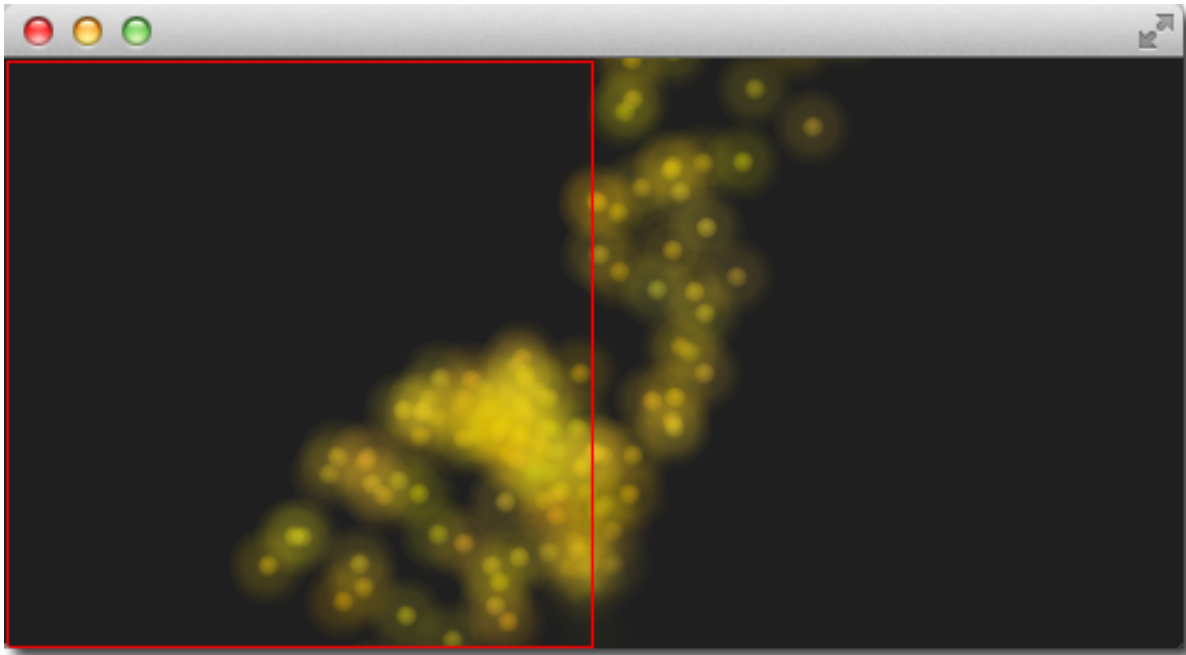
The gravity affector applies an acceleration In the example we stream the particles from the bottom to the top using an angle direction. The right side is unaffected, where on the left a gravity affect is applied. The gravity is angled to 90 degree (bottom-direction) with a magnitude of 50.

```
Gravity {  
    width: 240; height: 240  
    system: particleSystem  
    magnitude: 50  
    angle: 90  
    Tracer {}  
}
```

Particles on the left side try to climb up, but the steady applied acceleration towards the bottom drags them into the direction of the gravity.

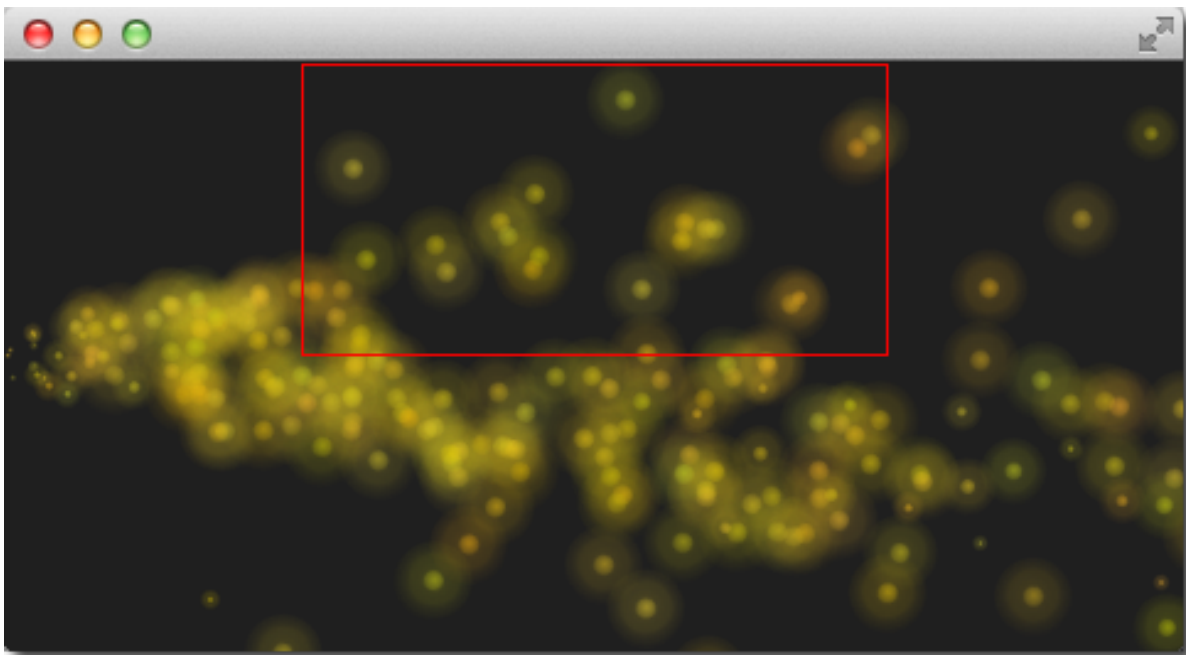
Turbulence

The turbulence affector, applies a *chaos* map of force vectors to the particles. The chaos map is defined by a noise image, which can be define with the *noiseSource* property. The strength defines how strong the vector will be applied on the particle movements.



```
Turbulence {  
    anchors.horizontalCenter: parent.horizontalCenter  
    width: 240; height: 120  
    system: particleSystem  
    strength: 100  
    Tracer {}  
}
```

In the upper area of the example, particles are influenced by the turbulence. Their movement is more erratic. The amount of erratic deviation from the original path is defined by the strength.

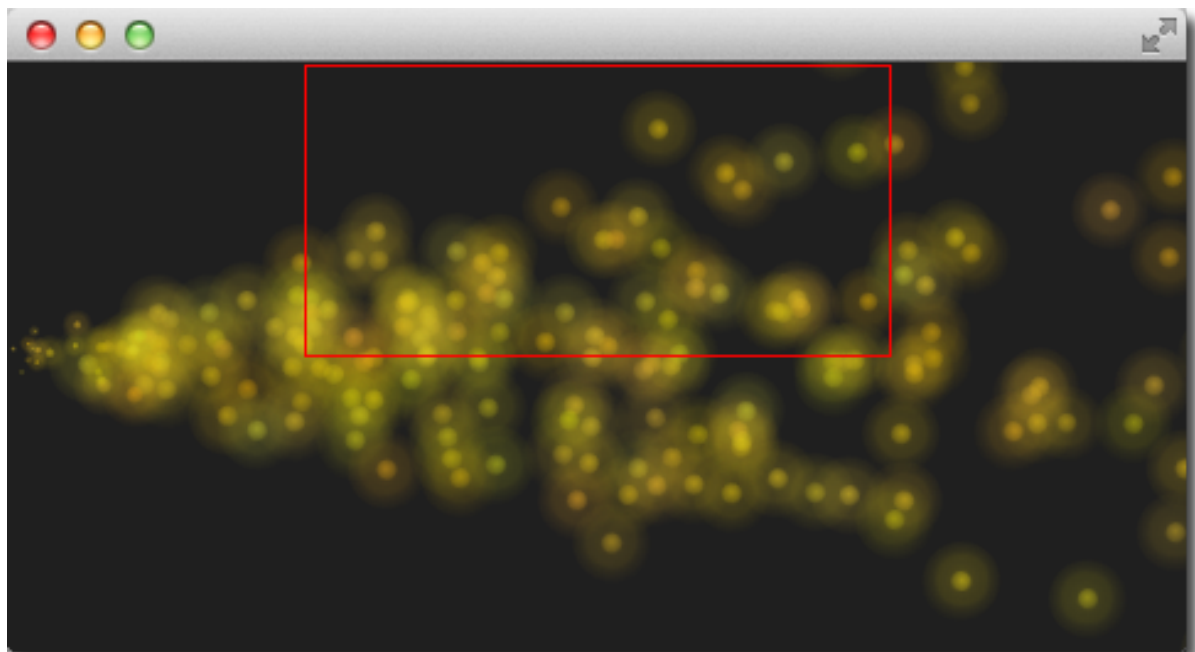


Wander

The wander manipulates the trajectory. With the property *affectedParameter* can be specified which parameter (velocity, position or acceleration) is affected by the wander. The *pace* property specifies the maximum of attribute changes per second. The *yVariance* and *xVariance* specifies the influence on x and y component of the particle trajectory.

```
Wander {  
    anchors.horizontalCenter: parent.horizontalCenter  
    width: 240; height: 120  
    system: particleSystem  
    affectedParameter: Wander.Position  
    pace: 200  
    yVariance: 240  
    Tracer {}  
}
```

In the top wander affector particles are shuffled around by random trajectory changes. In this case the position is changed 200 times per second in the y-direction.



Particle Groups

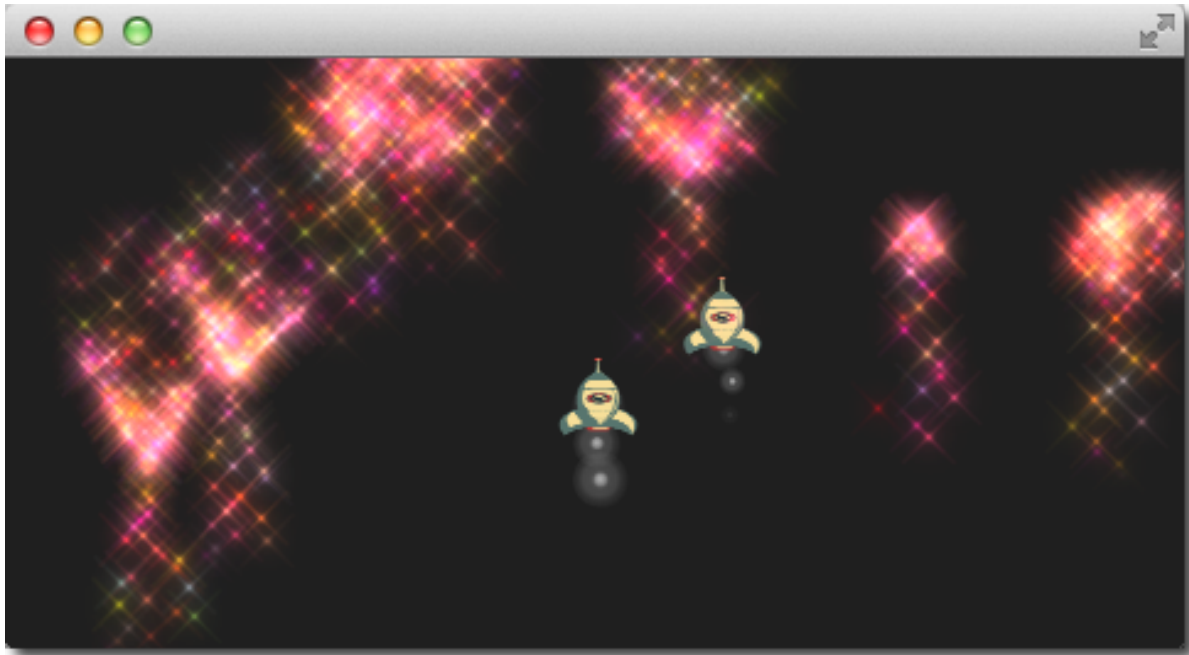
At the beginning of this chapter we stated particles are in groups, which is by default the empty group (''). Using the *GroupGoal* affector is it possible to let the particle change groups. To visualize this we would like to create a small firework, where rockets start into space and explode in the air into a spectacular firework.

The example is divided into 2 parts. The 1st part called "Launch Time" is concerned to setup the scene and introduce particle groups and the 2nd part called "Let there be fireworks" focuses on the group changes.

Let's get start!

Launch Time

To get it going we create a typical dark scene:



```
import QtQuick 2.5
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false
}
```

The tracer property will be used to switch the tracer scene wide on and off. The next thing is to declare our particle system:

```
ParticleSystem {
    id: particleSystem
}
```

And our two image particles (one for the rocket and one for the exhaust smoke):

```
ImageParticle {
    id: smokePainter
    system: particleSystem
    groups: ['smoke']
    source: "assets/particle.png"
    alpha: 0.3
    entryEffect: ImageParticle.None
}

ImageParticle {
    id: rocketPainter
    system: particleSystem
    groups: ['rocket']
    source: "assets/rocket.png"
    entryEffect: ImageParticle.None
}
```

You can see in on the images, they use the *groups* property to declare to which group the particle belong. It is enough to just declare a name and an implicit group will be created by Qt Quick.

Now it's time to emit some rockets into the air. For this we create an emitter on the bottom of our scene and set the velocity into an upward direction. To simulate some gravity we set an acceleration downwards:

```
Emitter {
    id: rocketEmitter
    anchors.bottom: parent.bottom
    width: parent.width; height: 40
    system: particleSystem
    group: 'rocket'
    emitRate: 2
    maximumEmitted: 4
    lifeSpan: 4800
    lifeSpanVariation: 400
    size: 32
    velocity: AngleDirection { angle: 270; magnitude: 150; magnitudeVariation: 10 }
    acceleration: AngleDirection { angle: 90; magnitude: 50 }
    Tracer { color: 'red'; visible: root.tracer }
}
```

The emitter is in the group 'rocket', the same as our rocket particle painter. Through the group name they are bound together. The emitter emits particles into the group 'rocket' and the rocket particle painter will paint them.

For the exhaust we use a trail emitter, which follows our rocket. It declares an own group called 'smoke' and follows the particles from the 'rocket' group:

```
TrailEmitter {
    id: smokeEmitter
    system: particleSystem
    emitHeight: 1
    emitWidth: 4
    group: 'smoke'
    follow: 'rocket'
    emitRatePerParticle: 96
    velocity: AngleDirection { angle: 90; magnitude: 100; angleVariation: 5 }
    lifeSpan: 200
    size: 16
    sizeVariation: 4
    endSize: 0
}
```

The smoke is directed downwards to simulate the force the smoke comes out of the rocket. The *emitHeight* and *emitWidth* specify the area around the particle followed from where the smoke particles shall be emitted. If this is not specified then the area of the particle followed is taken but for this example we want to increase the effect that the particles stem from a central point near the end of the rocket.

If you start the example now you will see the rockets fly up and some are even flying out of the scene. As this is not really wanted we need to slow them down before they leave the screen. A friction affector can be used here to slow the particles down to a minimum threshold:

```
Friction {
    groups: ['rocket']
    anchors.top: parent.top
    width: parent.width; height: 80
    system: particleSystem
    threshold: 5
    factor: 0.9
}
```

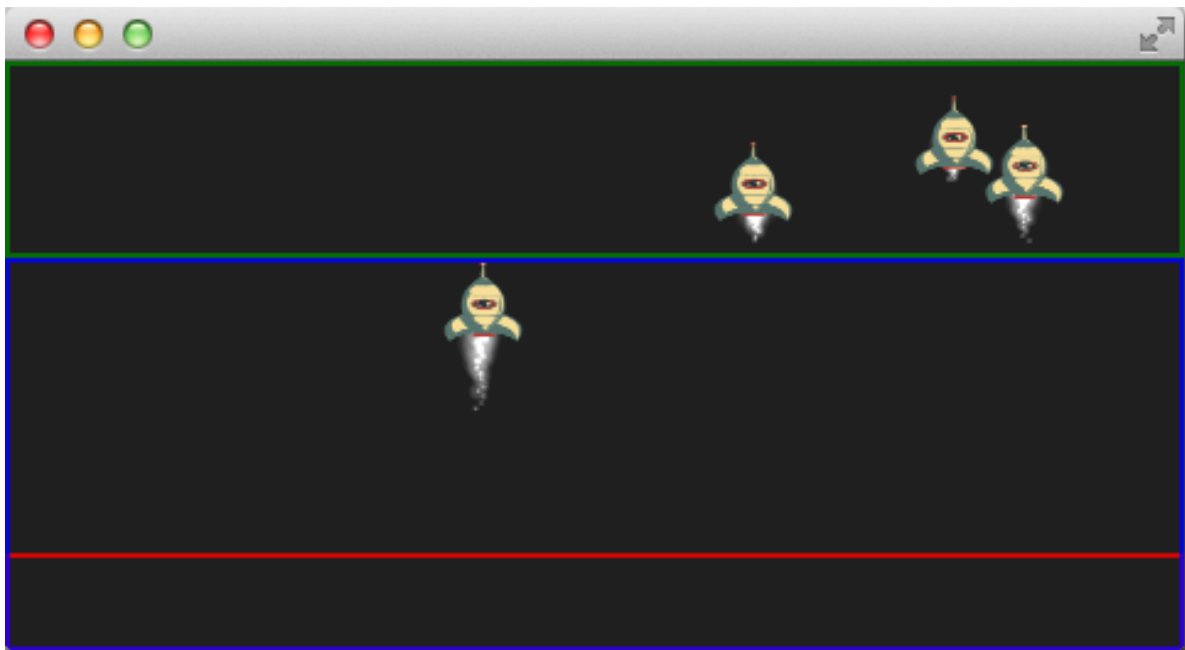
In the friction affector you also need to declare which groups of particles it shall affect. The friction will slow all rockets, which are 80 pixel downwards from the top of the screen down by a factor of 0.9 (try 100 and you will see they almost stop immediately) until they reach a velocity of 5 pixel per second. As the particles have still an acceleration downwards applied the rockets will start sinking toward the ground after they reach the end of their life-span.

As climbing up in the air is hard work and a very unstable situation we want to simulate some turbulences while the ship is climbing:

```
Turbulence {
  groups: ['rocket']
  anchors.bottom: parent.bottom
  width: parent.width; height: 160
  system: particleSystem
  strength: 25
  Tracer { color: 'green'; visible: root.tracer }
}
```

Also the turbulence need to declare which groups it shall affect. The turbulence it self reaches from the bottom 160 pixel upwards (until it reaches the border of the friction). They also could overlap.

When you start the example now you will see the rockets are climbing up and then will be slowed down by the friction and fall back to ground by the still applied downwards acceleration. The next thing would be to start the firework.



Note: The image shows the scene with the tracers enabled to show the different areas. Rocket particles are emitted in the red area and then affected by the turbulence in the blue area. Finally they are slowed down by the friction affector in the green area and start falling again, because of the steady applied downwards acceleration.

Let there be fireworks

To be able to change the rocket into a beautiful firework we need add a `ParticleGroup` to encapsulate the changes:

```
ParticleGroup {
  name: 'explosion'
  system: particleSystem
}
```

We change to the particle group using a `GroupGoal` affector. The group goal affector is placed near the vertical center of the screen and it will affect the group 'rocket'. With the `groupGoal` property we set the target group for

the change to ‘explosion’, our earlier defined particle group:

```
GroupGoal {
    id: rocketChanger
    anchors.top: parent.top
    width: parent.width; height: 80
    system: particleSystem
    groups: ['rocket']
    goalState: 'explosion'
    jump: true
    Tracer { color: 'blue'; visible: root.tracer }
}
```

The *jump* property states the change in groups shall be immediately and not after a certain duration.

Note: In the Qt 5 alpha release we could the *duration* for the group change not get working. Any ideas?

As the group of the rocket now changes to our ‘explosion’ particle group when the rocket particle enters the group goal area we need to add the firework inside the particle group:

```
// inside particle group
TrailEmitter {
    id: explosionEmitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 750
    emitRatePerParticle: 200
    size: 32
    velocity: AngleDirection { angle: -90; angleVariation: 180; magnitude: 50 }
}
```

The explosion emits particles into the ‘sparkle’ group. We will define soon a particle painter for this group. The trail emitter used follows the rocket particle and emits per rocket 200 particles. The particles are directed upwards and vary by 180 degree.

As the particles are emitted into the ‘sparkle’ group, we also need to define a particle painter for the particles:

```
ImageParticle {
    id: sparklePainter
    system: particleSystem
    groups: ['sparkle']
    color: 'red'
    colorVariation: 0.6
    source: "assets/star.png"
    alpha: 0.3
}
```

The sparkles of our firework shall be little red stars with a almost transparent color to allow some shine effects.

To make the firework more spectacular we also add a second trail emitter to our particle group, which will emit particles in a narrow cone downwards:

```
// inside particle group
TrailEmitter {
    id: explosion2Emitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 250
    emitRatePerParticle: 100
    size: 32
}
```



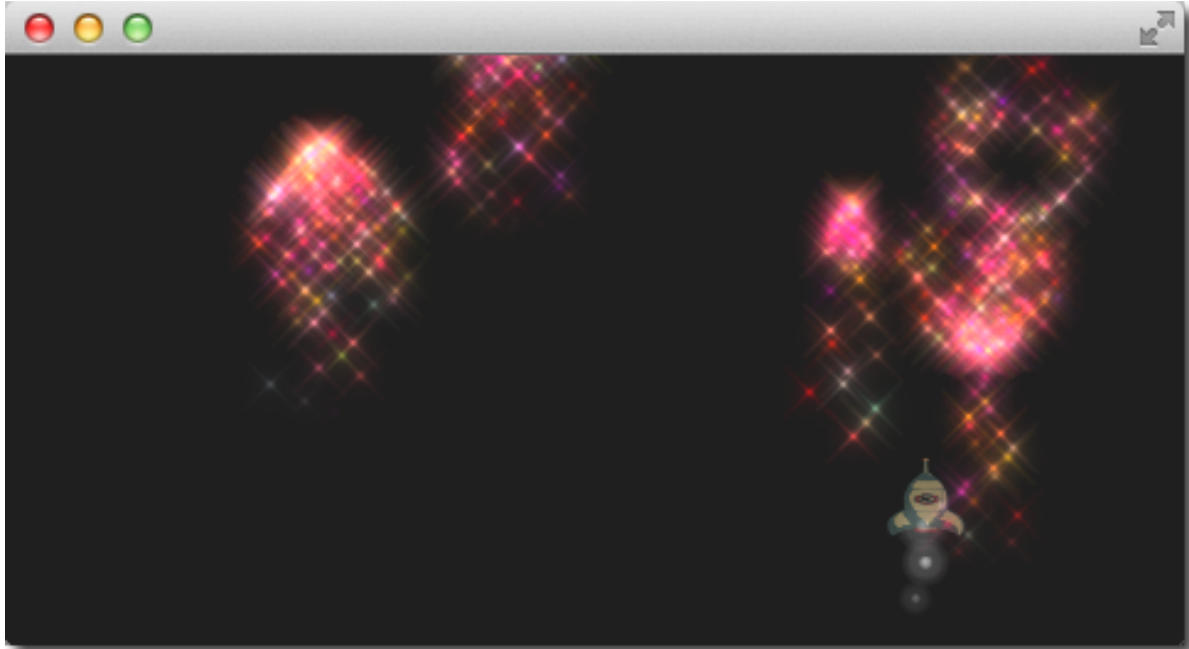
```

    velocity: AngleDirection { angle: 90; angleVariation: 15; magnitude: 400 }
}

```

Otherwise the setup is similar to the other explosion trail emitter. That's it.

Here is the final result.



Here is the full source code of the rocket firework.

```

import QtQuick 2.5
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false

    ParticleSystem {
        id: particleSystem
    }

    ImageParticle {
        id: smokePainter
        system: particleSystem
        groups: ['smoke']
        source: "assets/particle.png"
        alpha: 0.3
    }

    ImageParticle {
        id: rocketPainter
        system: particleSystem
        groups: ['rocket']
        source: "assets/rocket.png"
        entryEffect: ImageParticle.Fade
    }
}

```

```

Emitter {
    id: rocketEmitter
    anchors.bottom: parent.bottom
    width: parent.width; height: 40
    system: particleSystem
    group: 'rocket'
    emitRate: 2
    maximumEmitted: 8
    lifeSpan: 4800
    lifeSpanVariation: 400
    size: 128
    velocity: AngleDirection { angle: 270; magnitude: 150; magnitudeVariation: 10 }
    acceleration: AngleDirection { angle: 90; magnitude: 50 }
    Tracer { color: 'red'; visible: root.tracer }
}

TrailEmitter {
    id: smokeEmitter
    system: particleSystem
    group: 'smoke'
    follow: 'rocket'
    size: 16
    sizeVariation: 8
    emitRatePerParticle: 16
    velocity: AngleDirection { angle: 90; magnitude: 100; angleVariation: 15 }
    lifeSpan: 200
    Tracer { color: 'blue'; visible: root.tracer }
}

Friction {
    groups: ['rocket']
    anchors.top: parent.top
    width: parent.width; height: 80
    system: particleSystem
    threshold: 5
    factor: 0.9
}

Turbulence {
    groups: ['rocket']
    anchors.bottom: parent.bottom
    width: parent.width; height: 160
    system: particleSystem
    strength: 25
    Tracer { color: 'green'; visible: root.tracer }
}

ImageParticle {
    id: sparklePainter
    system: particleSystem
    groups: ['sparkle']
    color: 'red'
    colorVariation: 0.6
    source: "assets/star.png"
    alpha: 0.3
}

GroupGoal {
    id: rocketChanger
    anchors.top: parent.top

```

```

        width: parent.width; height: 80
        system: particleSystem
        groups: ['rocket']
        goalState: 'explosion'
        jump: true
        Tracer { color: 'blue'; visible: root.tracer }
    }

    ParticleGroup {
        name: 'explosion'
        system: particleSystem

        TrailEmitter {
            id: explosionEmitter
            anchors.fill: parent
            group: 'sparkle'
            follow: 'rocket'
            lifeSpan: 750
            emitRatePerParticle: 200
            size: 32
            velocity: AngleDirection { angle: -90; angleVariation: 180; magnitude:
↪50 }
        }

        TrailEmitter {
            id: explosion2Emitter
            anchors.fill: parent
            group: 'sparkle'
            follow: 'rocket'
            lifeSpan: 250
            emitRatePerParticle: 100
            size: 32
            velocity: AngleDirection { angle: 90; angleVariation: 15; magnitude:
↪400 }
        }
    }
}

```

Summary

Particles are a very powerful and fun way to express graphical phenomena like smoke, firework, random visual elements. The extended API in Qt 5 is very powerful and we have just scratched on the surface. There are several elements we haven't yet played with like sprites, size tables or color tables. Also when the particles look very playful they have a great potential when used wisely to create some eye catcher in any user interface. Using too many particle effects inside an user interface will definitely lead to the impression towards a game. Creating games is also the real strength of the particles.

Shader Effects

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

Objective

- <http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html>
- <http://www.opengl.org/registry/doc/GLSLangSpec.4.20.6.clean.pdf>
- http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf
- <http://www.lighthouse3d.com/opengl/glsl/>
- http://wiki.delphigl.com/index.php/Tutorial_glsl
- Qt5Doc qtquick-shaders

Give a short introduction to shader effects and then present the shader effects and their use.

Shaders allows us to create awesome rendering effects on top to the SceneGraph API leveraging directly the power of OpenGL running on the GPU. Shaders are implemented using the ShaderEffect and ShaderEffectSource elements. The shader algorithm itself is implemented using the OpenGL Shading Language.

Practically it means you mix QML code with shader code. On execution will the shader code be sent over to the GPU and compiled and executed on the GPU. The shader QML elements allow you to interact through properties with the OpenGL shader implementation.

Let's first have a look what OpenGL shaders are.

OpenGL Shaders

OpenGL uses a rendering pipeline split into stages. A simplified OpenGL pipeline would contain a vertex and fragment shader.



The vertex shader receives vertex data and must assign it to the *gl_Position* at the end of the routine. In the next stage the vertexes are clipped, transformed and rasterized for pixel output. From there the fragments (pixels) arrive in the fragment shader and can further be manipulated and the resulting color needs to be assigned to *gl_FragColor*. The vertex shader is called for each corner point of your polygon (vertex = point in 3D) and is responsible of any 3D manipulation of these points. The fragment (fragment = pixel) shader is called for each pixel and determines the color of that pixel.

Shader Elements

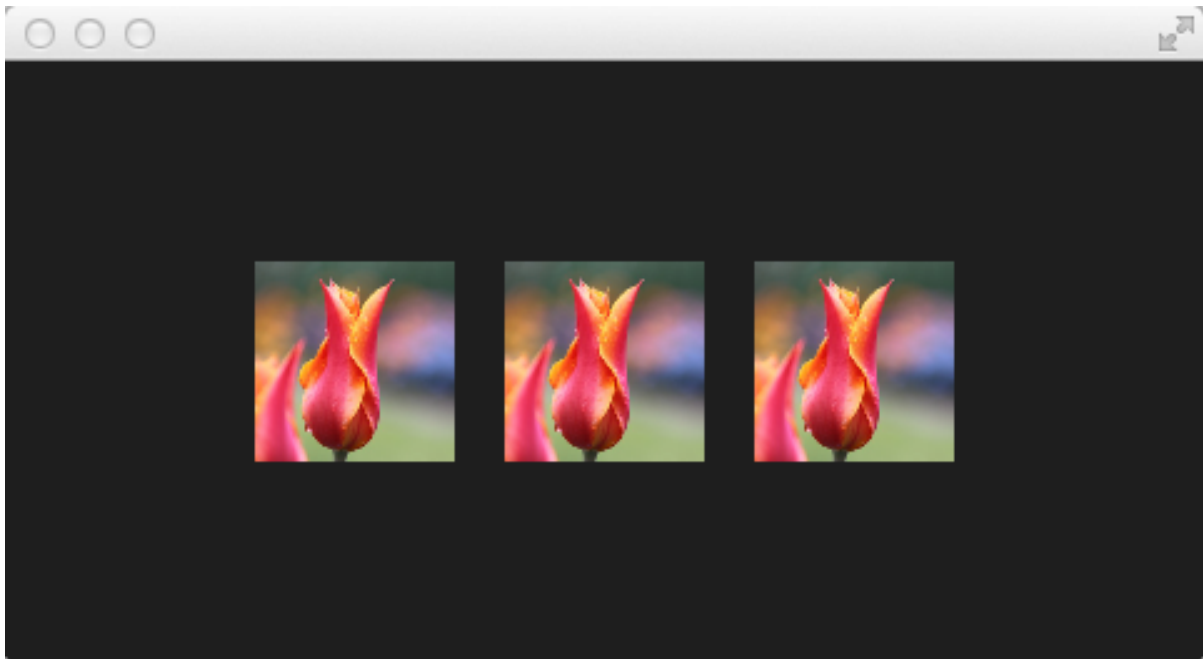
For programing shaders Qt Quick provides two elements. The *ShaderEffectSource* and the *ShaderEffect*. The shader effect applies custom shaders and the shader effect source renders a QML item into a texture and renders it. As shader effect can apply a custom shaders to it's rectangular shape and can use sources for the shader operation. A source can be an image, which is used as a texture or a shader effect source.

The default shader uses the source and renders it unmodified.

```
import QtQuick 2.5

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Row {
        anchors.centerIn: parent
        spacing: 20
        Image {
            id: sourceImage
            width: 80; height: width
            source: 'assets/tulips.jpg'
        }
        ShaderEffect {
            id: effect
            width: 80; height: width
            property variant source: sourceImage
        }
        ShaderEffect {
            id: effect2
            width: 80; height: width
            // the source where the effect shall be applied to
            property variant source: sourceImage
            // default vertex shader code
            vertexShader: "
                uniform highp mat4 qt_Matrix;
                attribute highp vec4 qt_Vertex;
                attribute highp vec2 qt_MultiTexCoord0;
                varying highp vec2 qt_TexCoord0;
                void main() {
                    qt_TexCoord0 = qt_MultiTexCoord0;
                    gl_Position = qt_Matrix * qt_Vertex;
                }"
            // default fragment shader code
            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * qt_Opacity;
                }"
        }
    }
}
```



In the above example we have a row of 3 images. The first is the real image. The second is rendered using the default shader and the third is rendered using the default shader code for the fragment and vertex extracted from the Qt 5 source code.

Note: If you don't want to see the source image and only the effected image you can set the *Image* to invisible (`visible : false`). The shader effects will still use the image data just the *Image* element will not be rendered.

Let's have a closer look at the shader code.

```
vertexShader: "
    uniform highp mat4 qt_Matrix;
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    varying highp vec2 qt_TexCoord0;
    void main() {
        qt_TexCoord0 = qt_MultiTexCoord0;
        gl_Position = qt_Matrix * qt_Vertex;
    }"
```

Both shaders are from the Qt side a string bound to the *vertexShader* and *fragmentShader* property. Every shader code has to have a *main()* { ... } function, which is executed by the GPU. Variable starting with *qt_* are provided by default by Qt already.

Here a short rundown on the variables:

uniform	value does not change during processing
attribute	linkage to external data
varying	shared value between shaders
highp	high precision value
lowp	low precision value
mat4	4x4 float matrix
vec2	2=dim float vector
sampler2D	2D texture
float	floating scalar

A better reference is the [OpenGL ES 2.0 API Quick Reference Card](#)

Now we might be better able to understand what the variable are:

- `qt_Matrix`: model-view-projection matrix
- `qt_Vertex`: current vertex position
- `qt_MultiTexCoord0`: texture coordinate
- `qt_TexCoord0`: shared texture coordinate

So we have available the projection matrix, the current vertex and the texture coordinate. The texture coordinate relates to the texture given as source. In the `main()` function we store the texture coordinate for later use in the fragment shader. Every vertex shader need to assign the `gl_Position` this is done using here by multiplying the project matrix with the vertex, our point in 3D.

The fragment shader receives our texture coordinate from the vertex shader and also the texture from our QML source property. It shall be noted how easy it is to pass variable between the shader code and QML. Beautiful. Additional we have the opacity of the shader effect available as `qt_Opacity`. Every fragment shader needs to assign the `gl_FragColor` variable, this is done in the default shader code by picking the pixel from the source texture and multiplying it with the opacity.

```
fragmentShader: "  
    varying highp vec2 qt_TexCoord0;  
    uniform sampler2D source;  
    uniform lowp float qt_Opacity;  
    void main() {  
        gl_FragColor = texture2D(source, qt_TexCoord0) * qt_Opacity;  
    }"
```

During the next examples we will playing around with some simple shader mechanics. First we concentrate on the fragment shader and then we will come back to the vertex shader.

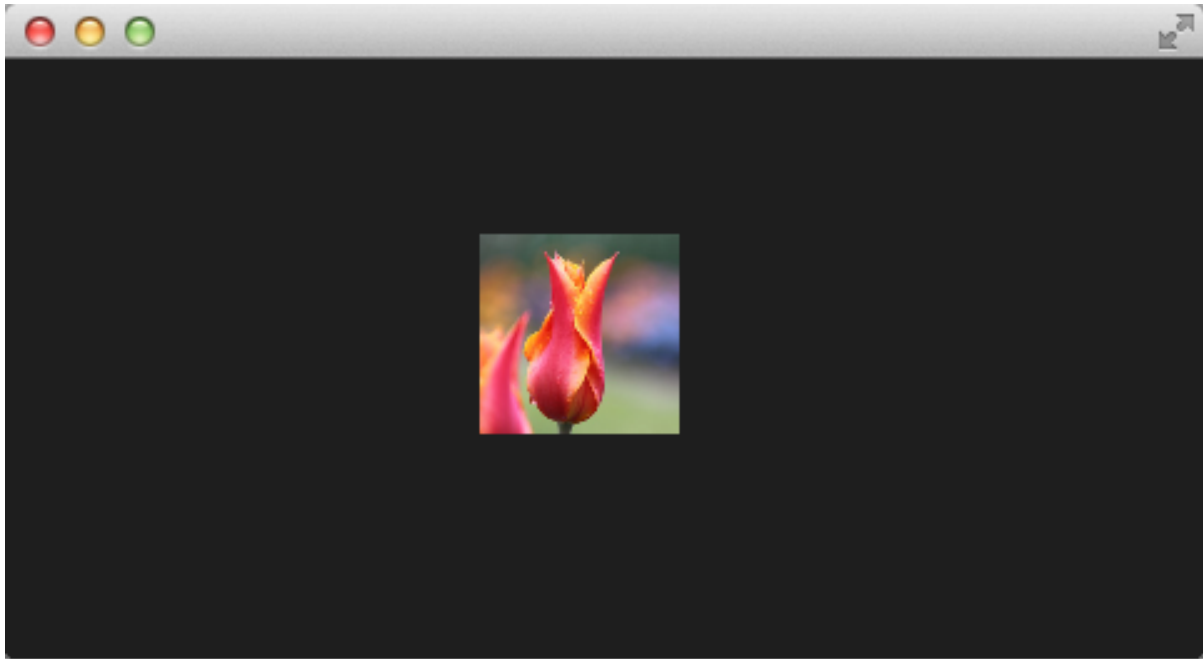
Fragment Shaders

The fragment shader is called for every pixel to be rendered. We will develop a small red lens, which will increase the red color channel value of the image.

Setting up the scene

First we setup our scene, with a grid centered in the field and our source image be displayed.

```
import QtQuick 2.5  
  
Rectangle {  
    width: 480; height: 240  
    color: '#1e1e1e'  
  
    Grid {  
        anchors.centerIn: parent  
        spacing: 20  
        rows: 2; columns: 4  
        Image {  
            id: sourceImage  
            width: 80; height: width  
            source: 'assets/tulips.jpg'  
        }  
    }  
}
```

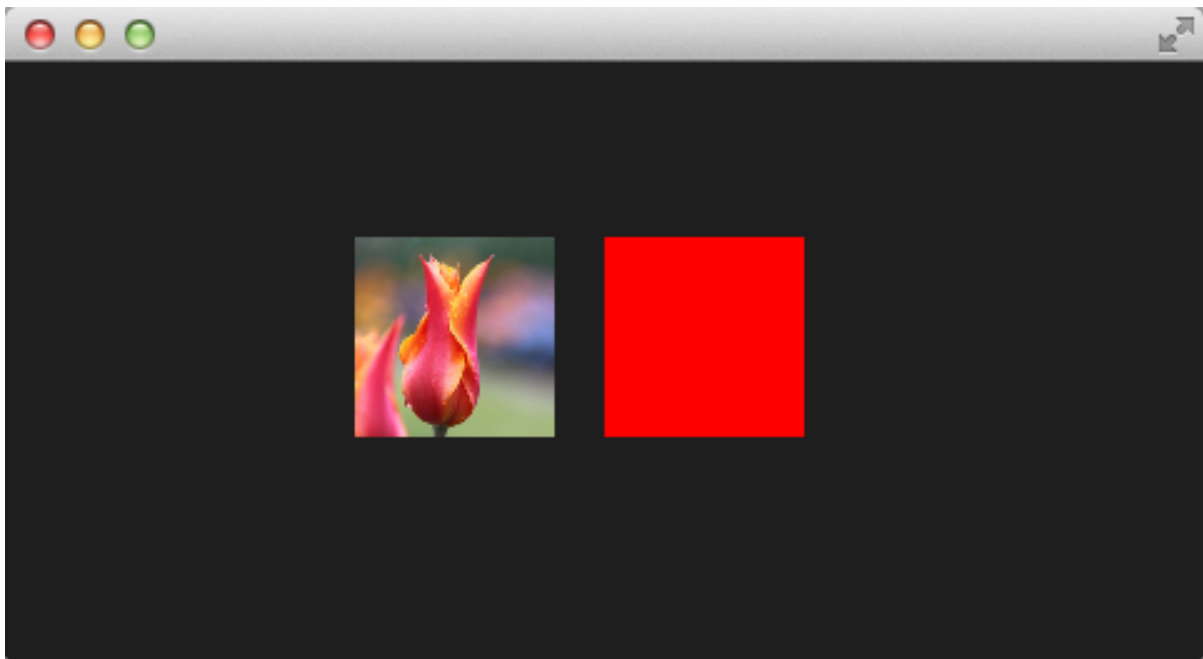



A red shader

Next we will add a shader, which displays a red rectangle by providing for each fragment a red color value.

```
fragmentShader: "  
    uniform lowp float qt_Opacity;  
    void main() {  
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0) * qt_Opacity;  
    }"
```

In the fragment shader we simply assign a *vec4(1.0, 0.0, 0.0, 1.0)* which represents a red color with full opacity (alpha=1.0) to the *gl_FragColor* for each fragment.



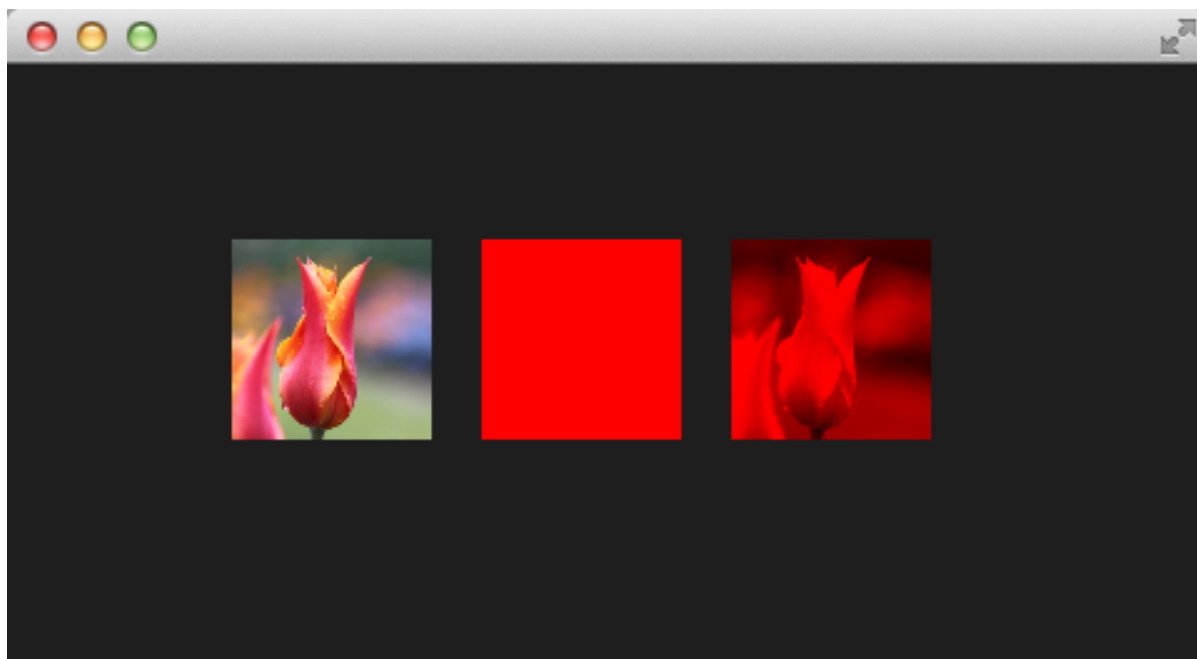
A red shader with texture

Now we want to apply the red color to each texture pixel. For this we need the texture back in the vertex shader. As we don't do anything else in the vertex shader the default vertex shader is enough for us.

```
ShaderEffect {
    id: effect2
    width: 80; height: width
    property variant source: sourceImage
    visible: root.step>1
    fragmentShader: "
        varying highp vec2 qt_TexCoord0;
        uniform sampler2D source;
        uniform lowp float qt_Opacity;
        void main() {
            gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(1.0, 0.0,
→ 0.0, 1.0) * qt_Opacity;
        }"
}
```

The full shader contains now back our image source as variant property and we have left out the vertex shader, which if not specified is the default vertex shader.

In the fragment shader we pick the texture fragment *texture2D(source, qt_TexCoord0)* and apply the red color to it.



The red channel property

It's not really nice to hard code the red channel value, so we would like to control the value from the QML side. For this we add a *redChannel* property to our shader effect and also declare a *uniform lowp float redChannel* inside our fragment shader. That's all to make a value from the shader code available to the QML side. Very simple.

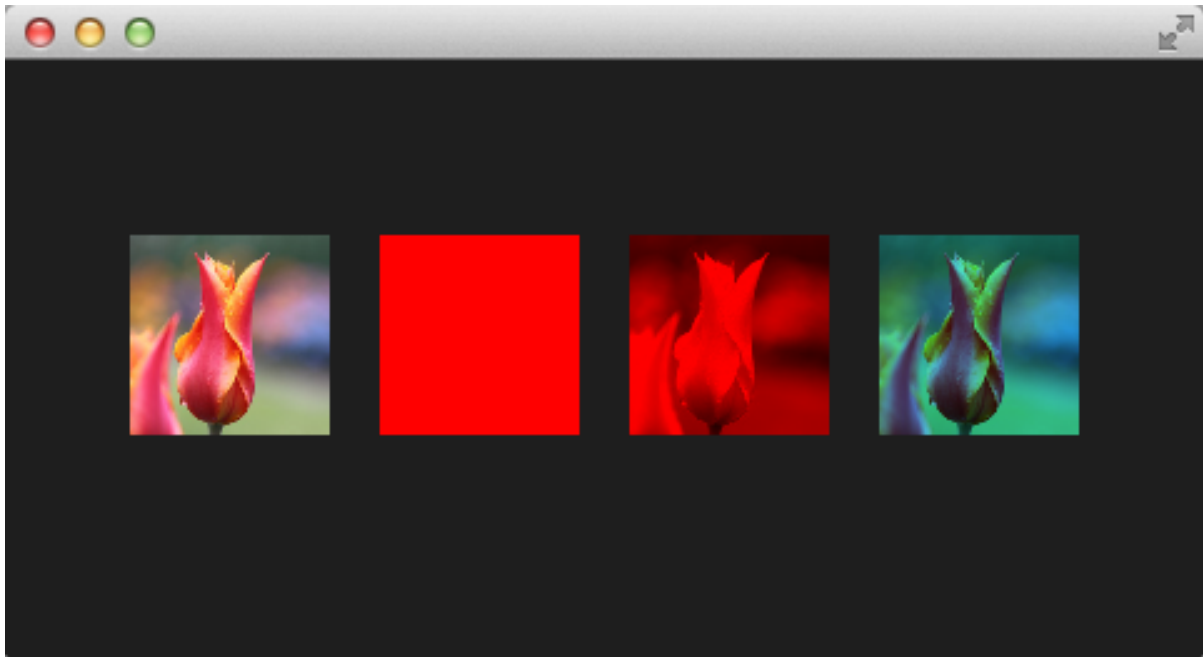
```
ShaderEffect {
    id: effect3
    width: 80; height: width
    property variant source: sourceImage
```

```

property real redChannel: 0.3
visible: root.step>2
fragmentShader: "
    varying highp vec2 qt_TexCoord0;
    uniform sampler2D source;
    uniform lowp float qt_Opacity;
    uniform lowp float redChannel;
    void main() {
        gl_FragColor = texture2D(source, qt_TexCoord0) *
→vec4(redChannel, 1.0, 1.0, 1.0) * qt_Opacity;
    }"
}

```

To make the lens really a lens, we change the *vec4* color to be *vec4(redChannel, 1.0, 1.0, 1.0)* so that the other colors are multiplied by 1.0 and only the red portion is multiplied by our *redChannel* variable.



The red channel animated

As the *redChannel* property is just a normal property it can also be animated as all properties in QML. So we can use QML properties to animate values on the GPU to influence our shaders. How cool is that!

```

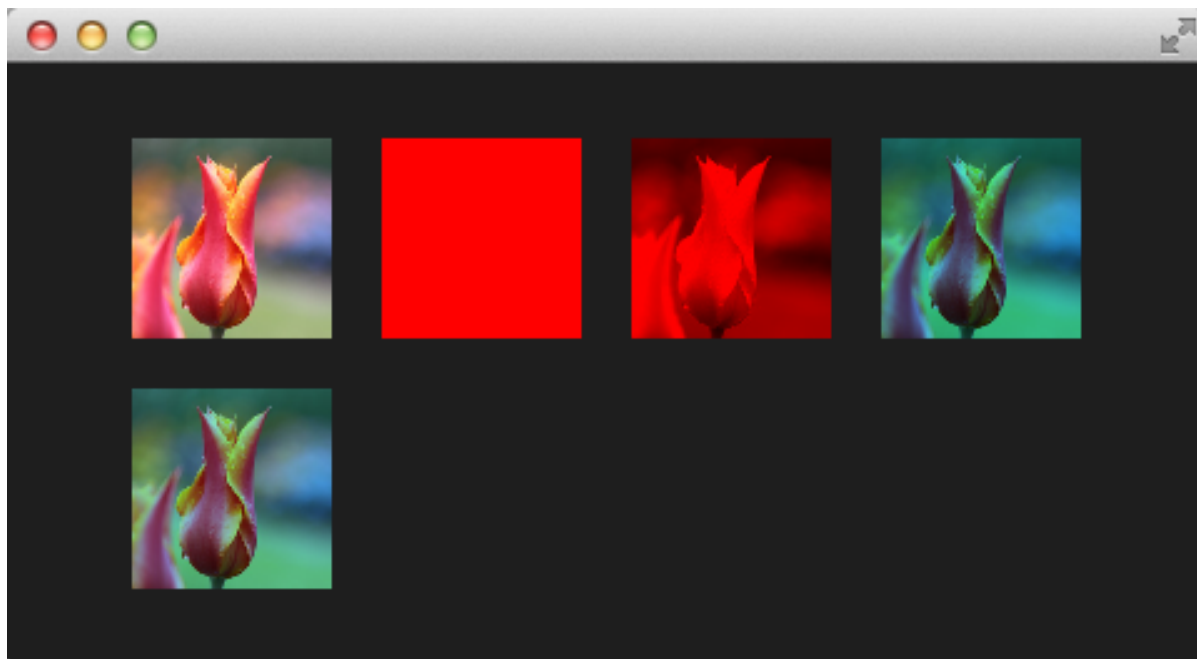
ShaderEffect {
    id: effect4
    width: 80; height: width
    property variant source: sourceImage
    property real redChannel: 0.3
    visible: root.step>3
    NumberAnimation on redChannel {
        from: 0.0; to: 1.0; loops: Animation.Infinite; duration: 4000
    }

    fragmentShader: "
        varying highp vec2 qt_TexCoord0;
        uniform sampler2D source;
        uniform lowp float qt_Opacity;
        uniform lowp float redChannel;
    "
}

```

```
        void main() {  
            gl_FragColor = texture2D(source, qt_TexCoord0) *_  
→vec4(redChannel, 1.0, 1.0, 1.0) * qt_Opacity;  
        }"  
    }
```

Here the final result.



The shader effect on the 2nd row is animated from 0.0 to 1.0 with a duration of 4 seconds. So the image goes from no red information (0.0 red) over to a normal image (1.0 red).

Wave Effect

In this more complex example we will create a wave effect with the fragment shader. The wave form is based on the sinus curve and it influences the texture coordinates used for the color.

```
import QtQuick 2.5  
  
Rectangle {  
    width: 480; height: 240  
    color: '#1e1e1e'  
  
    Row {  
        anchors.centerIn: parent  
        spacing: 20  
        Image {  
            id: sourceImage  
            width: 160; height: width  
            source: "assets/coastline.jpg"  
        }  
        ShaderEffect {  
            width: 160; height: width  
            property variant source: sourceImage  
            property real frequency: 8  
            property real amplitude: 0.1  
            property real time: 0.0  
        }  
    }  
}
```

```

NumberAnimation on time {
    from: 0; to: Math.PI*2; duration: 1000; loops: Animation.Infinite
}

fragmentShader: "
    varying highp vec2 qt_TexCoord0;
    uniform sampler2D source;
    uniform lowp float qt_Opacity;
    uniform highp float frequency;
    uniform highp float amplitude;
    uniform highp float time;
    void main() {
        highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
        highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -
→pulse.x);
        gl_FragColor = texture2D(source, coord) * qt_Opacity;
    }
}"
    }
}

```

The wave calculation is based on a pulse and the texture coordinate manipulation. The pulse equation gives us a sine wave depending on the current time and the used texture coordinate:

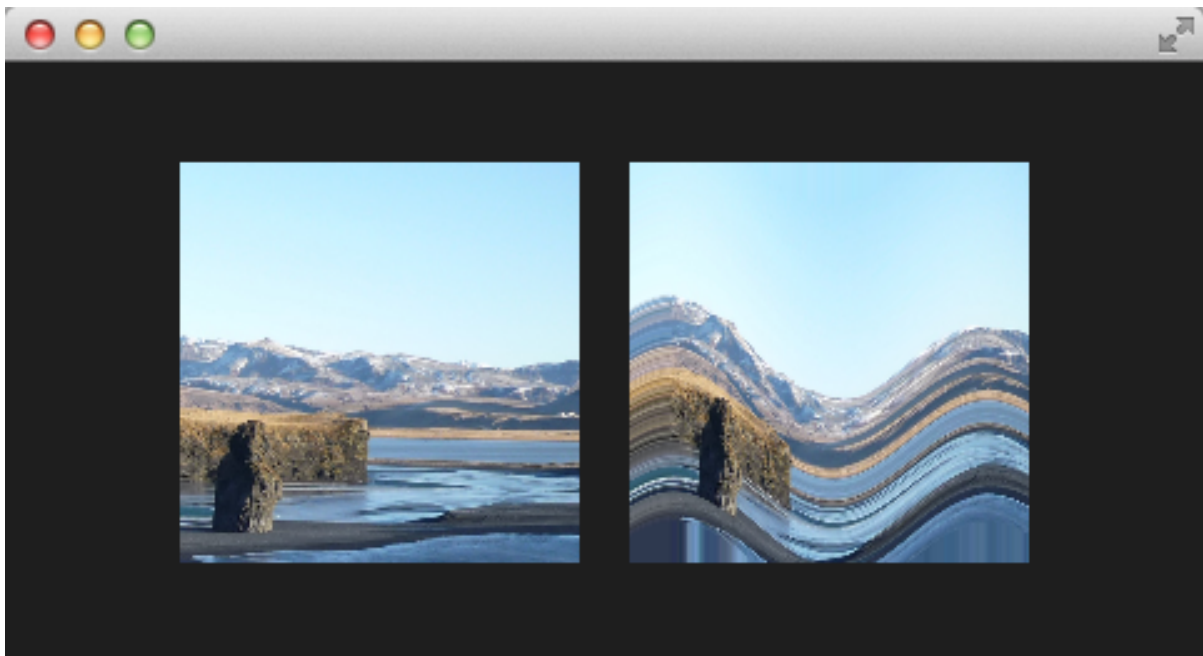
```
highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
```

Without the time factor we would just have a distortion but not a traveling distortion, like waves are.

For the color we use the color at a different texture coordinate:

```
highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -pulse.x);
```

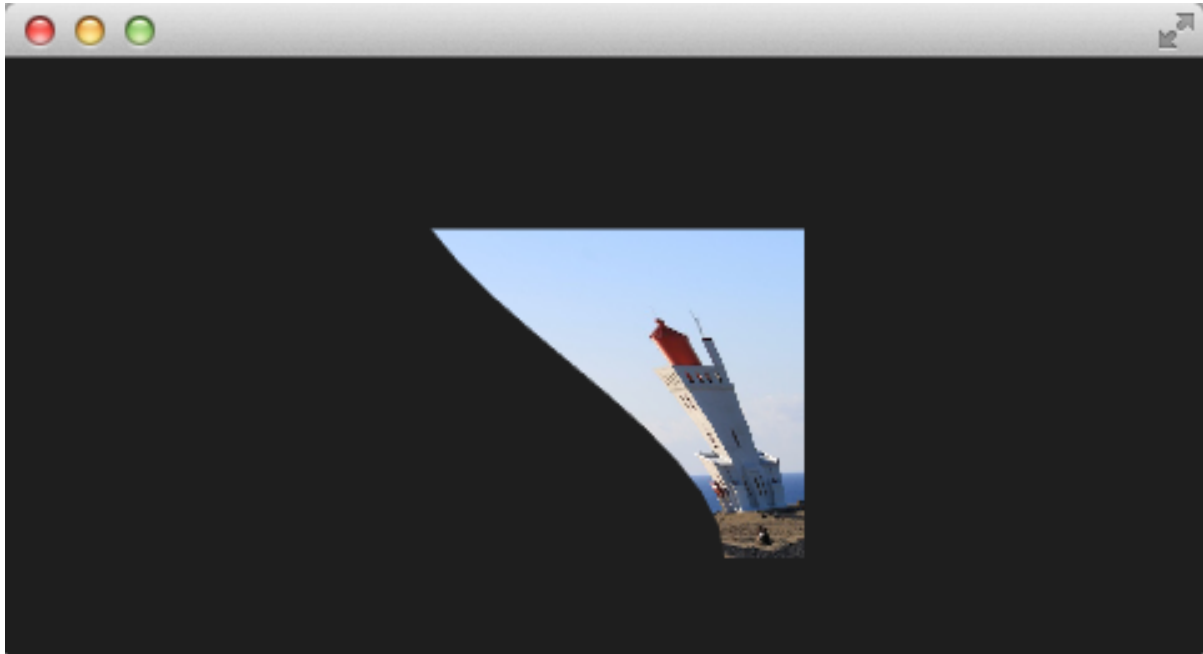
The texture coordinate is influenced by our pulse x-value. The result of this is a moving wave.



Also if we haven't moved pixels in this fragment shader the effect would look at first like a job for a vertex shader.

Vertex Shader

The vertex shader can be used to manipulate the vertices provided by the shader effect. In normal cases the shader effect has 4 vertices (top-left, top-right, bottom-left and bottom-right). Each vertex reported is from type `vec4`. To visualize the vertex shader we will program a genie effect. This effect is often used to let a rectangular window area vanish into one point.



Setting up the scene

First we will setup our scene again.

```
import QtQuick 2.5

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

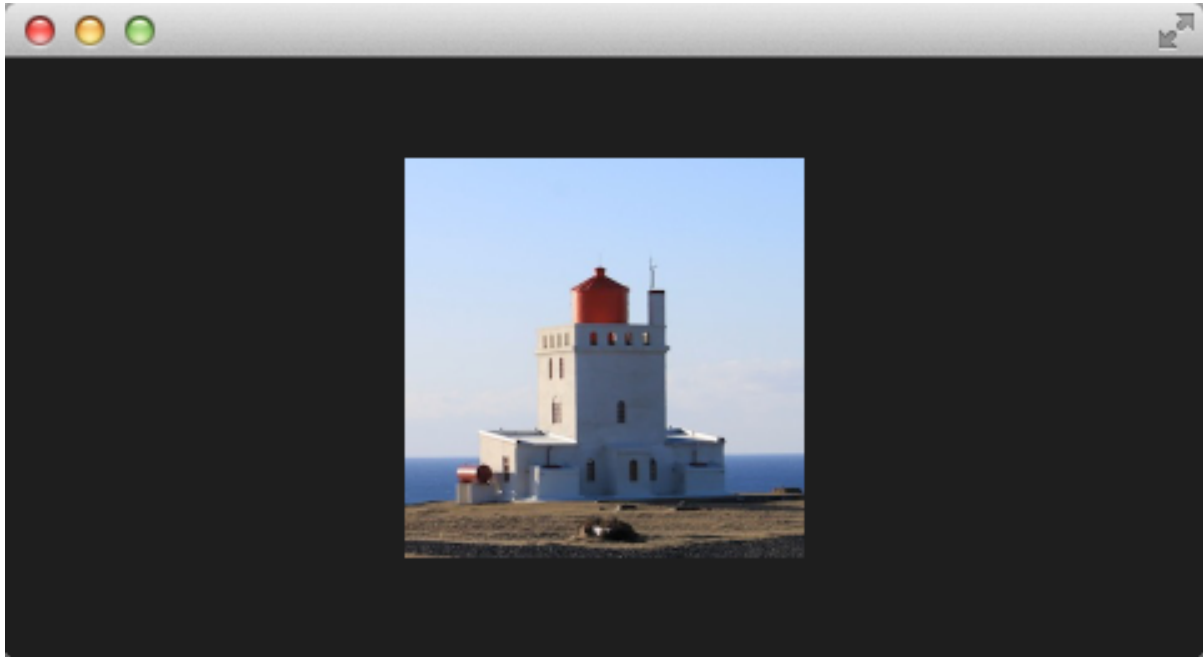
    Image {
        id: sourceImage
        width: 160; height: width
        source: "assets/lighthouse.jpg"
        visible: false
    }
    Rectangle {
        width: 160; height: width
        anchors.centerIn: parent
        color: '#333333'
    }
    ShaderEffect {
        id: genieEffect
        width: 160; height: width
        anchors.centerIn: parent
        property variant source: sourceImage
        property bool minimized: false
        MouseArea {
            anchors.fill: parent
        }
    }
}
```

```

        onClicked: genieEffect.minimized = !genieEffect.minimized
    }
}

```

This provides as a scene with a dark background and a shader effect using an image as the source texture. The original image is not visible on the image produced by our genie effect. Additionally we added a dark rectangle on the same geometry as the shader effect so we can better detect where we need to click to revert the effect.



The effect is triggered by clicking on the image, this is defined by the mouse area covering the effect. In the *onClicked* handler we toggle the custom boolean property *minimized*. We will use this property later to toggle the effect.

Minimize and normalize

After we have setup the scene, we define a property of type real called *minimize*, the property will contain the current value of our minimization. The value will vary from 0.0 to 1.0 and is controlled by a sequential animation.

```

property real minimize: 0.0

SequentialAnimation on minimize {
    id: animMinimize
    running: genieEffect.minimized
    PauseAnimation { duration: 300 }
    NumberAnimation { to: 1; duration: 700; easing.type: Easing.InOutSine }
    PauseAnimation { duration: 1000 }
}

SequentialAnimation on minimize {
    id: animNormalize
    running: !genieEffect.minimized
    NumberAnimation { to: 0; duration: 700; easing.type: Easing.InOutSine }
    PauseAnimation { duration: 1300 }
}

```

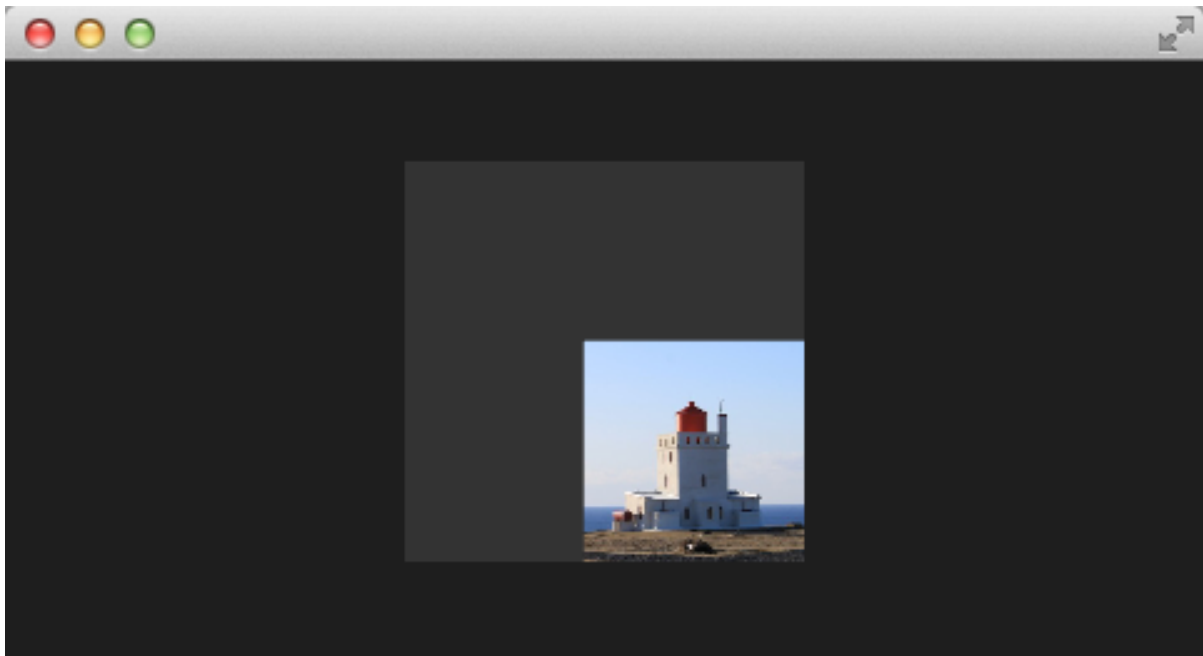
The animation is triggered by the toggling of the *minimized* property. Now that we have setup all our surroundings we finally can look at our vertex shader.

```
vertexShader: "  
    uniform highp mat4 qt_Matrix;  
    attribute highp vec4 qt_Vertex;  
    attribute highp vec2 qt_MultiTexCoord0;  
    varying highp vec2 qt_TexCoord0;  
    uniform highp float minimize;  
    uniform highp float width;  
    uniform highp float height;  
    void main() {  
        qt_TexCoord0 = qt_MultiTexCoord0;  
        highp vec4 pos = qt_Vertex;  
        pos.y = mix(qt_Vertex.y, height, minimize);  
        pos.x = mix(qt_Vertex.x, width, minimize);  
        gl_Position = qt_Matrix * pos;  
    }"
```

The vertex shader is called for each vertex so four times, in our case. The default qt defined parameters are provided, like *qt_Matrix*, *qt_Vertex*, *qt_MultiTexCoord0*, *qt_TexCoord0*. We have discussed the variable already earlier. Additionally we link the *minimize*, *width* and *height* variables from our shader effect into our vertex shader code. In the main function we store the current texture coordinate in our *qt_TexCoord0* to make it available to the fragment shader. Now we copy the current position and modify the x and y position of the vertex:

```
highp vec4 pos = qt_Vertex;  
pos.y = mix(qt_Vertex.y, height, minimize);  
pos.x = mix(qt_Vertex.x, width, minimize);
```

The *mix(...)* function provides a linear interpolation between the first 2 parameters on the point (0.0-1.0) provided by the 3rd parameter. So in our case we interpolate for y between the current y position and the height based on the current minimize value, similar for x. Bear in mind the minimize value is animated by our sequential animation and travels from 0.0 to 1.0 (or vice versa).



The resulting effect is not really the genie effect but is already a great step towards it.

Todo

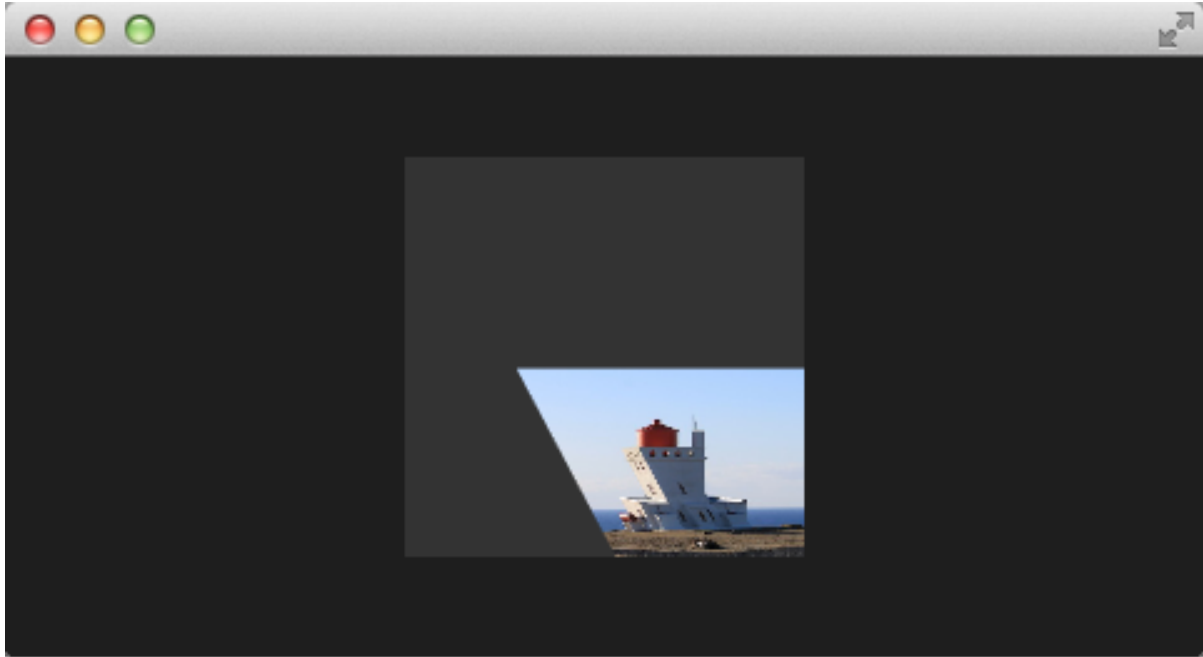
better explanation, maybe draw the 4 vertexes and their interpolation

Primitive Bending

So minimized the x and y components of our vertexes. Now we would like to slightly modify the x manipulation and make it depending of the current y value. The needed changes are pretty small. The y-position is calculated as before. The interpolation of the x-position depends now on the vertexes y-position:

```
highp float t = pos.y / height;
pos.x = mix(qt_Vertex.x, width, t * minimize);
```

This results into an x-position tending towards the width when the y-position is larger. In other words the upper 2 vertexes are not affected at all as they have an y-position of 0 and the lower two vertexes x-positions both bend towards the width, so they bend towards the same x-position.



```
import QtQuick 2.5

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Image {
        id: sourceImage
        width: 160; height: width
        source: "assets/lighthouse.jpg"
        visible: false
    }
    Rectangle {
        width: 160; height: width
        anchors.centerIn: parent
        color: '#333333'
    }
    ShaderEffect {
        id: genieEffect
        width: 160; height: width
        anchors.centerIn: parent
        property variant source: sourceImage
        property real minimize: 0.0
        property bool minimized: false
    }
}
```

```
SequentialAnimation on minimize {
    id: animMinimize
    running: genieEffect.minimized
    PauseAnimation { duration: 300 }
    NumberAnimation { to: 1; duration: 700; easing.type: Easing.InOutSine }
    PauseAnimation { duration: 1000 }
}

SequentialAnimation on minimize {
    id: animNormalize
    running: !genieEffect.minimized
    NumberAnimation { to: 0; duration: 700; easing.type: Easing.InOutSine }
    PauseAnimation { duration: 1300 }
}

vertexShader: "
    uniform highp mat4 qt_Matrix;
    uniform highp float minimize;
    uniform highp float height;
    uniform highp float width;
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    varying highp vec2 qt_TexCoord0;
    void main() {
        qt_TexCoord0 = qt_MultiTexCoord0;
        // M1>>
        highp vec4 pos = qt_Vertex;
        pos.y = mix(qt_Vertex.y, height, minimize);
        highp float t = pos.y / height;
        pos.x = mix(qt_Vertex.x, width, t * minimize);
        gl_Position = qt_Matrix * pos;
    }
}
```

Better Bending

As the bending is not really satisfying currently we will add several parts to improve the situation. First we enhance our animation to support an own bending property. This is necessary as the bending should happen immediately and the y-minimization should be delayed shortly. Both animation have in the sum the same duration (300+700+1000 and 700+1300).

```
property real bend: 0.0
property bool minimized: false

// change to parallel animation
ParallelAnimation {
    id: animMinimize
    running: genieEffect.minimized
    SequentialAnimation {
        PauseAnimation { duration: 300 }
        NumberAnimation {
            target: genieEffect; property: 'minimize';
            to: 1; duration: 700;
            easing.type: Easing.InOutSine
        }
        PauseAnimation { duration: 1000 }
    }
}
// adding bend animation
SequentialAnimation {
```

```

    NumberAnimation {
        target: genieEffect; property: 'bend'
        to: 1; duration: 700;
        easing.type: Easing.InOutSine }
    PauseAnimation { duration: 1300 }
}

```

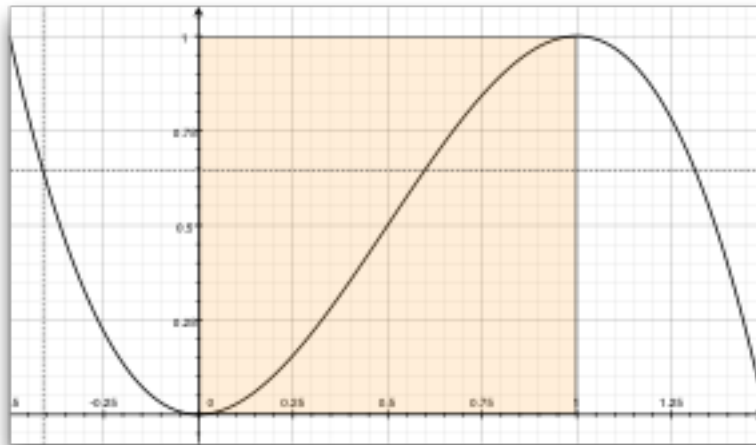
Additional to make the bending a smooth curve the y-effect on the x-position is not modified by a curved function from 0..1 and the `pos.x` depends now on the new bend property animation:

```

highp float t = pos.y / height;
t = (3.0 - 2.0 * t) * t * t;
pos.x = mix(qt_Vertex.x, width, t * bend);

```

The curve starts smooth at the 0.0 value, grows then and stops smoothly towards the 1.0 value. Here is a plot of the function in the specified range. For us only the range from 0..1 is from interest.



The most visual change is by increasing our amount of vertex points. The vertex points used can be increased by using a mesh:

```

mesh: GridMesh { resolution: Qt.size(16, 16) }

```

The shader effect now has an equality distributed grid of 16x16 vertexes instead of the 2x2 vertexes used before. This makes the interpolation between the vertexes look much smoother.

You can see also the influence of the curve being used, as the bending smoothes at the end nicely. This is where the bending has the strongest effect.

Choosing Sides

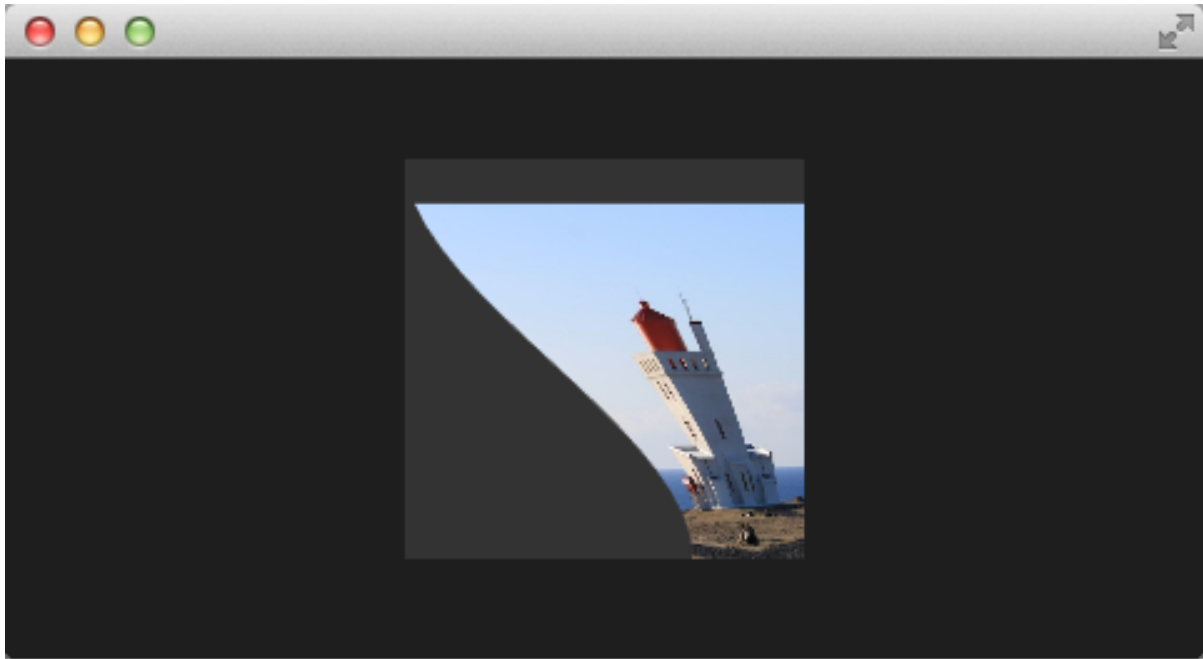
As a final enhancement we want to be able to switch sides. The side is towards which point the genie effect vanishes. Till now it vanishes always towards the width. By adding a *side* property we are able to modify the point between 0 and width.

```

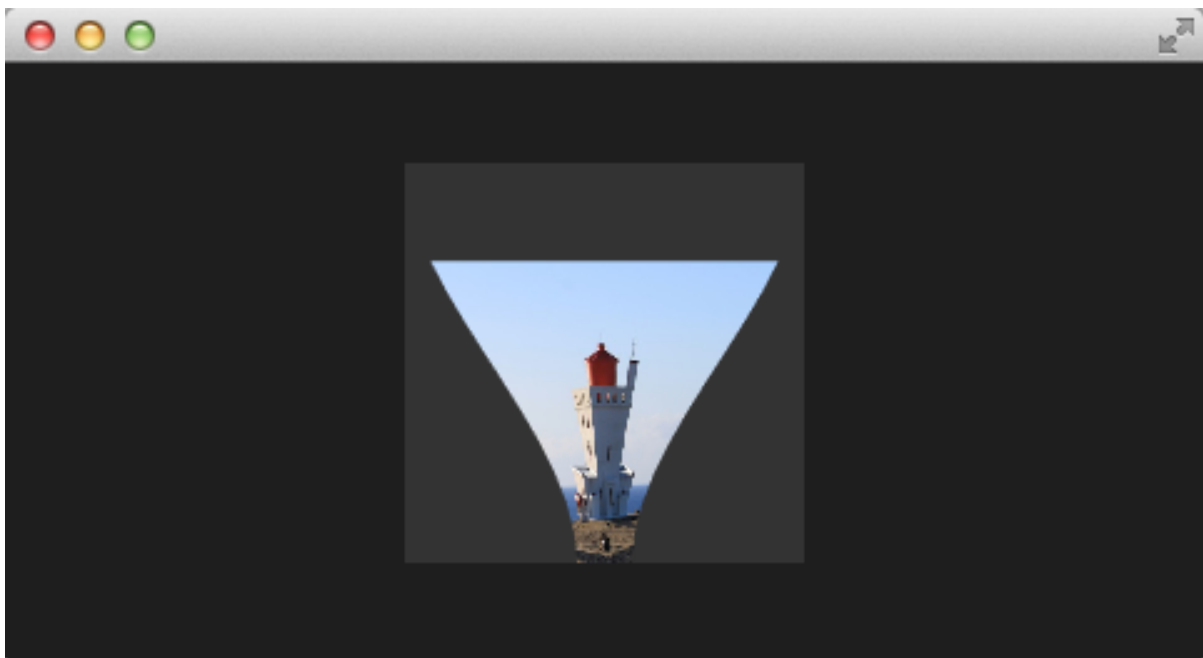
ShaderEffect {
    ...
    property real side: 0.5

    vertexShader: "
        ...
        uniform highp float side;
    "
}

```



```
...  
    pos.x = mix(qt_Vertex.x, side * width, t * bend);  
    "  
}
```



Packaging

The last thing to-do is package our effect nicely. For this we extract our genie effect code into an own component called *GenieEffect*. It has the shader effect as the root element. We removed the mouse area as this should not be inside the component as the triggering of the effect can be toggled by the *minimized* property.

```

import QtQuick 2.5

ShaderEffect {
    id: genieEffect
    width: 160; height: width
    anchors.centerIn: parent
    property variant source
    mesh: GridMesh { resolution: Qt.size(10, 10) }
    property real minimize: 0.0
    property real bend: 0.0
    property bool minimized: false
    property real side: 1.0

    ParallelAnimation {
        id: animMinimize
        running: genieEffect.minimized
        SequentialAnimation {
            PauseAnimation { duration: 300 }
            NumberAnimation {
                target: genieEffect; property: 'minimize';
                to: 1; duration: 700;
                easing.type: Easing.InOutSine
            }
            PauseAnimation { duration: 1000 }
        }
        SequentialAnimation {
            NumberAnimation {
                target: genieEffect; property: 'bend'
                to: 1; duration: 700;
                easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }
    }

    ParallelAnimation {
        id: animNormalize
        running: !genieEffect.minimized
        SequentialAnimation {
            NumberAnimation {
                target: genieEffect; property: 'minimize';
                to: 0; duration: 700;
                easing.type: Easing.InOutSine
            }
            PauseAnimation { duration: 1300 }
        }
        SequentialAnimation {
            PauseAnimation { duration: 300 }
            NumberAnimation {
                target: genieEffect; property: 'bend'
                to: 0; duration: 700;
                easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }
    }

    vertexShader: "
        uniform highp mat4 qt_Matrix;
        attribute highp vec4 qt_Vertex;
        attribute highp vec2 qt_MultiTexCoord0;
        uniform highp float height;
        uniform highp float width;
        uniform highp float minimize;
    "

```

```
uniform highp float bend;
uniform highp float side;
varying highp vec2 qt_TexCoord0;
void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;
    highp vec4 pos = qt_Vertex;
    pos.y = mix(qt_Vertex.y, height, minimize);
    highp float t = pos.y / height;
    t = (3.0 - 2.0 * t) * t * t;
    pos.x = mix(qt_Vertex.x, side * width, t * bend);
    gl_Position = qt_Matrix * pos;
}"
}
```

You can use now the effect simply like this:

```
import QtQuick 2.5

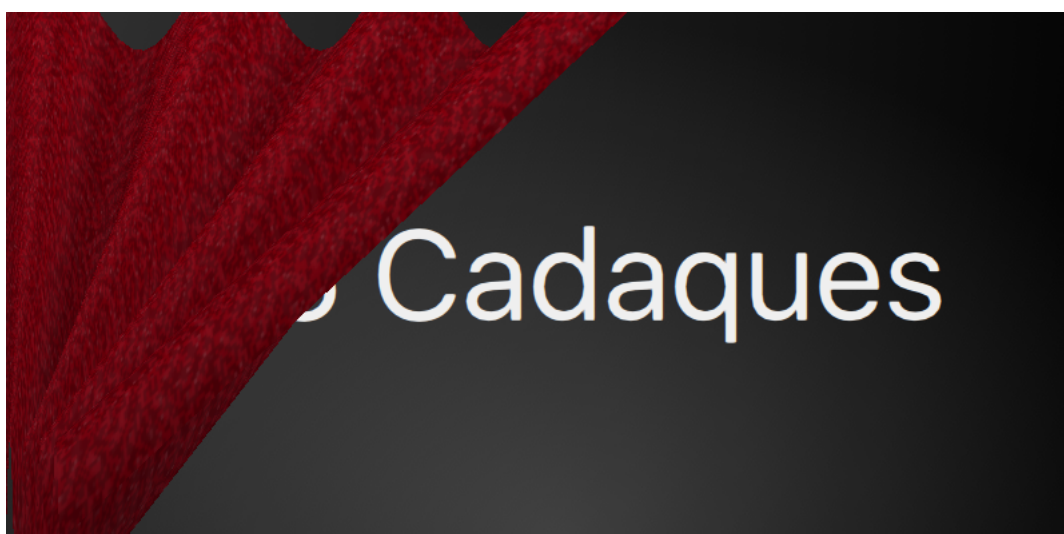
Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    GenieEffect {
        source: Image { source: 'assets/lighthouse.jpg' }
        MouseArea {
            anchors.fill: parent
            onClicked: parent.minimized = !parent.minimized
        }
    }
}
```

We have simplified the code by removing our background rectangle and we assigned the image directly to the effect, instead of loading it inside a standalone image element.

Curtain Effect

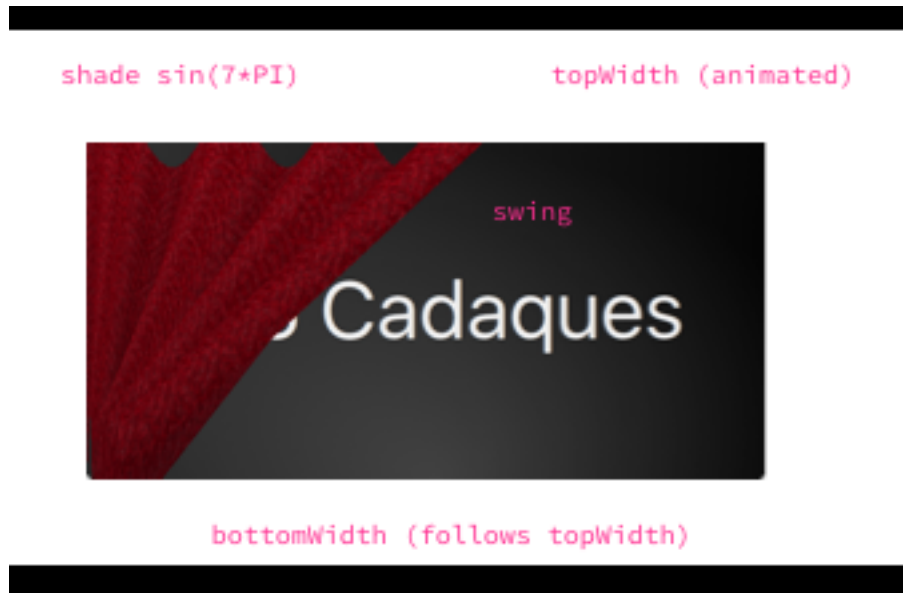
In the last example for custom shader effects I would like to bring you the curtain effect. This effect was published first in May 2011 as part of [Qt labs for shader effects](#).



At that time I really loved these effects and the curtain effect was my favorite out of them. I just love how the curtain opens and hide the background object.

I took the code and adapted it towards Qt 5, which was straightforward. Also O did some simplifications to be able to use it better for a showcase. So if you are interested in the full example, please visit the labs blog.

Just a little bot for the background, the curtain is actually an image called *fabric.jpg* and it is the source for a shader effect. The effect uses the vertex shader to swing the curtain and uses the fragment shader to provide some shades. Here is a simple diagram to make you hopefully better understand the code.



The waved shades of the curtain are computed through a sin curve with 7 up/downs ($7 \cdot \text{PI} = 21.99\dots$) on the width of the curtain. The other important part is the swing. The *topWidth* of the curtain is animated when the curtain is opened or closed. The *bottomWidth* follows the *topWidth* with a *SpringAnimation*. By this we create the effect of the swinging bottom part of the curtain. The calculated *swing* provides the strength of this swing interpolated over the y-component of the vertexes.

The curtain effect is located in the `CurtainEffect.qml` component where the fabric image act as the texture source. There is nothing new on the use of shaders here, only a different way to manipulate the *gl_Position* in the vertex shader and the *gl_FragColor* in the fragment shader.

```
import QtQuick 2.5

ShaderEffect {
    anchors.fill: parent

    mesh: GridMesh {
        resolution: Qt.size(50, 50)
    }

    property real topWidth: open?width:20
    property real bottomWidth: topWidth
    property real amplitude: 0.1
    property bool open: false
    property variant source: effectSource

    Behavior on bottomWidth {
        SpringAnimation {
            easing.type: Easing.OutElastic;
            velocity: 250; mass: 1.5;
            spring: 0.5; damping: 0.05
        }
    }

    Behavior on topWidth {
```

```

    NumberAnimation { duration: 1000 }
}

ShaderEffectSource {
    id: effectSource
    sourceItem: effectImage;
    hideSource: true
}

Image {
    id: effectImage
    anchors.fill: parent
    source: "assets/fabric.png"
    fillMode: Image.Tile
}

vertexShader: "
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    uniform highp mat4 qt_Matrix;
    varying highp vec2 qt_TexCoord0;
    varying lowp float shade;

    uniform highp float topWidth;
    uniform highp float bottomWidth;
    uniform highp float width;
    uniform highp float height;
    uniform highp float amplitude;

    void main() {
        qt_TexCoord0 = qt_MultiTexCoord0;

        highp vec4 shift = vec4(0.0, 0.0, 0.0, 0.0);
        highp float swing = (topWidth - bottomWidth) * (qt_Vertex.y / height);
        shift.x = qt_Vertex.x * (width - topWidth + swing) / width;

        shade = sin(21.9911486 * qt_Vertex.x / width);
        shift.y = amplitude * (width - topWidth + swing) * shade;

        gl_Position = qt_Matrix * (qt_Vertex - shift);

        shade = 0.2 * (2.0 - shade) * ((width - topWidth + swing) / width);
    }"

fragmentShader: "
    uniform sampler2D source;
    varying highp vec2 qt_TexCoord0;
    varying lowp float shade;
    void main() {
        highp vec4 color = texture2D(source, qt_TexCoord0);
        color.rgb *= 1.0 - shade;
        gl_FragColor = color;
    }"
}

```

The effect is used in the `curtaindemo.qml` file.

```

import QtQuick 2.5

Item {
    id: root
    width: background.width; height: background.height

```



```

Image {
    id: background
    anchors.centerIn: parent
    source: 'assets/background.png'
}

Text {
    anchors.centerIn: parent
    font.pixelSize: 48
    color: '#efefef'
    text: 'Qt5 Cadaques'
}

CurtainEffect {
    id: curtain
    anchors.fill: parent
}

MouseArea {
    anchors.fill: parent
    onClicked: curtain.open = !curtain.open
}

```

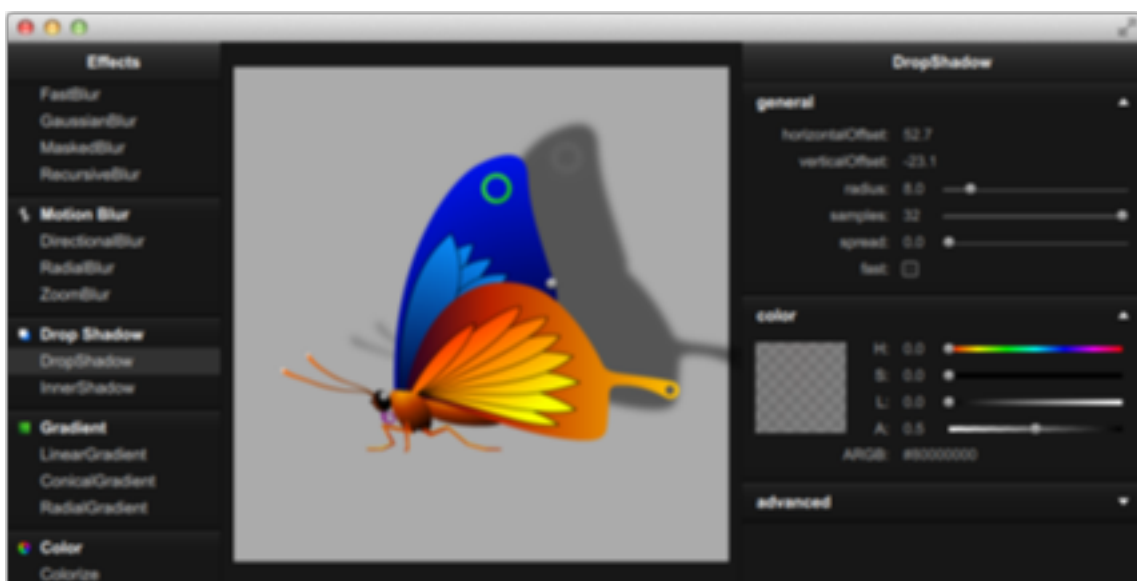
The curtain is opened through a custom *open* property on the curtain effect. We use a *MouseArea* to trigger the opening and closing of the curtain.

Qt GraphicsEffect Library

The graphics effect library is a collection of shader effects. Ready made by the Qt developers. It's a great tool-set to be used in your application but also a great source to learn how to build shaders.

The graphics effects library comes with a so called manual testbed which is a great tool to interactively discover the different effects.

The testbed is located under `$QTDIR/qtgraphicaleffects/tests/manual/testbed`.



The effects library contains ca 20 effects. A list of the effect and a short description can be found below.

Graphics Effects List

Table 9.1: Graphics Effects List

Category	Effect	Description
Blend	<i>Blend</i>	merges two source items by using a blend mode
Color	<i>BrightnessContrast</i>	adjusts brightness and contrast
	<i>Colorize</i>	sets color in the HSL color space
	<i>ColorOverlay</i>	applies a color layer
	<i>Desaturate</i>	reduces color saturation
	<i>GammaAdjust</i>	adjusts luminance
	<i>HueSaturation</i>	adjusts colors in the HSL color space
	<i>LevelAdjust</i>	adjusts colors in the RGB color space
Gradient	<i>ConicalGradient</i>	draws a conical gradient
	<i>LinearGradient</i>	draws a linear gradient
	<i>RadialGradient</i>	draws a radial gradient
Distortion	<i>Displace</i>	moves the pixels of the source item according to the specified displacement source
Drop Shadow	<i>DropShadow</i>	draws a drop shadow
	<i>InnerShadow</i>	draws an inner shadow
Blur	<i>FastBlur</i>	applies a fast blur effect
	<i>GaussianBlur</i>	applies a higher quality blur effect
	<i>MaskedBlur</i>	applies a varying intensity blur effect
	<i>RecursiveBlur</i>	blurs repeatedly, providing a strong blur effect
Motion Blur	<i>DirectionalBlur</i>	applies a directional motion blur effect
	<i>RadialBlur</i>	applies a radial motion blur effect
	<i>ZoomBlur</i>	applies a zoom motion blur effect
Glow	<i>Glow</i>	draws an outer glow effect
	<i>RectangularGlow</i>	draws a rectangular outer glow effect
Mask	<i>OpacityMask</i>	masks the source item with another item
	<i>ThresholdMask</i>	masks the source item with another item and applies a threshold value

Here is a example using the *FastBlur* effect from the *Blur* category:

```
import QtQuick 2.5
import QtGraphicalEffects 1.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    Row {
        anchors.centerIn: parent
        spacing: 16

        Image {
            id: sourceImage
            source: "assets/tulips.jpg"
            width: 200; height: width
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
        }

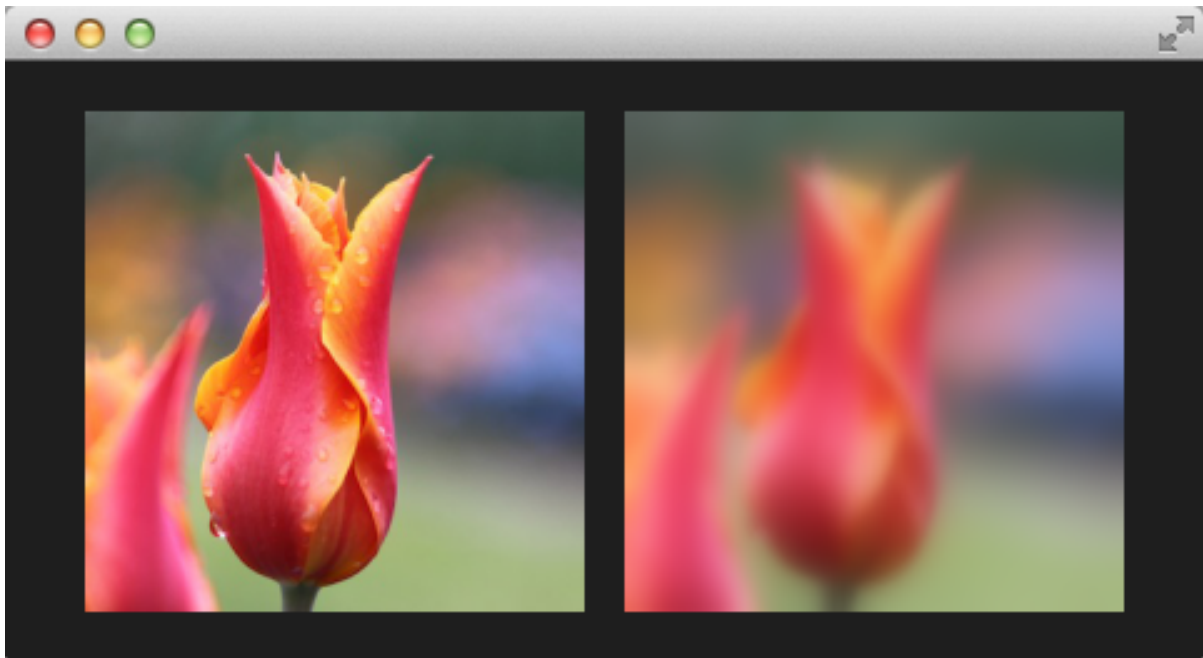
        FastBlur {
            width: 200; height: width
            source: sourceImage
        }
    }
}
```

```
radius: blurred?32:0
property bool blurred: false

Behavior on radius {
    NumberAnimation { duration: 1000 }
}

MouseArea {
    id: area
    anchors.fill: parent
    onClicked: parent.blurred = !parent.blurred
}
}
```

The image to the left is the original image. Clicking the image on the right will toggle blurred property and animated the blur radius from 0 to 32 during 1 second. The image on the left show the blurred image.



Multimedia

Section author: e8johan

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

The multimedia elements in the QtMultimedia makes it possible to playback and record media such as sound, video or pictures. Decoding and encoding is handled through platform specific backends. For instance, the popular gstreamer framework is used on Linux, while DirectShow is used on Windows and QuickTime on OS X.

The multimedia elements are not a part of the Qt Quick core API. Instead, they are provided through a separate API made available by importing QtMultimedia 5.6 as shown below:

```
import QtMultimedia 5.6
```

Playing Media

The most basic case of multimedia integration in a QML application is for it to playback media. This is done using the `MediaPlayer` element, optionally in combination with a `VideoOutput` element if the source is an image or video. The `MediaPlayer` element has a `source` property pointing at the media to play. When a media source has been bound, it is simply a matter of calling the `play` function to start playing.

If you want to play visual media, i.e. pictures or video, you must also setup a `VideoOutput` element. The `MediaPlayer` running the playback is bound to the video output through the `source` property.

In the example shown below, the `MediaPlayer` is given a file with video contents as `source`. A `VideoOutput` is created and bound to the media player. As soon as the main component has been fully initialized, i.e. at `Component.onCompleted`, the player's `play` function is called.

```
import QtQuick 2.5
import QtMultimedia 5.6

Item {
    width: 1024
    height: 600

    MediaPlayer {
        id: player
        source: "trailer_400p.ogv"
    }

    VideoOutput {
        anchors.fill: parent
        source: player
    }
}
```

```
    }

    Component.onCompleted: {
        player.play();
    }
}
```

Basic operations such as altering the volume when playing media is controlled through the `volume` property of the `MediaPlayer` element. There are other useful properties as well. For instance, the `duration` and `position` properties can be used to build a progress bar. If the `seekable` property is `true`, it is even possible to update the `position` when the progress bar is tapped. The example below shows how this is added to the basic playback example above.

```
Rectangle {
    id: progressBar

    anchors.left: parent.left
    anchors.right: parent.right
    anchors.bottom: parent.bottom
    anchors.margins: 100

    height: 30

    color: "lightGray"

    Rectangle {
        anchors.left: parent.left
        anchors.top: parent.top
        anchors.bottom: parent.bottom

        width: player.duration>0?parent.width*player.position/player.duration:0

        color: "darkGray"
    }

    MouseArea {
        anchors.fill: parent

        onClicked: {
            if (player.seekable) {
                player.position = player.duration * mouse.x/width;
            }
        }
    }
}
```

The `position` property is only updated once per second in the default case. This means that the progress bar will update in large steps unless the duration of the media is long enough, compared to the number pixels that the progress bar is wide. This can, however, be changed through accessing the `mediaObject` property and its `notifyInterval` property. It can be set to the number of milliseconds between each position update, increasing the smoothness of the user interface.

```
Connections {
    target: player
    onMediaObjectChanged: {
        if (player.mediaObject) {
            player.mediaObject.notifyInterval = 50;
        }
    }
}
```

Todo

The code above does not have any effect on the update interval! There seems to be no media object...

When using `MediaPlayer` to build a media player, it is good to monitor the `status` property of the player. It is an enumeration of the possible statuses, ranging from `MediaPlayer.Buffered` to `MediaPlayer.InvalidMedia`. The possible values are summarized in the bullets below:

- `MediaPlayer.UnknownStatus`. The status is unknown.
- `MediaPlayer.NoMedia`. The player has no media source assigned. Playback is stopped.
- `MediaPlayer.Loading`. The player is loading the media.
- `MediaPlayer.Loaded`. The media has been loaded. Playback is stopped.
- `MediaPlayer.Stalled`. The loading of media has stalled.
- `MediaPlayer.Buffering`. The media is being buffered.
- `MediaPlayer.Buffered`. The media has been buffered, this means that the player can start playing the media.
- `MediaPlayer.EndOfMedia`. The end of the media has been reached. Playback is stopped.
- `MediaPlayer.InvalidMedia`. The media cannot be played. Playback is stopped.

As mentioned in the bullets above, the playback state can vary over time. Calling `play`, `pause` or `stop` alters the state, but the media in question can also have effect. For example, the end can be reached, or it can be invalid, causing playback to stop. The current playback state can be tracked through the `playbackState` property. The values can be `MediaPlayer.PlayingState`, `MediaPlayer.PausedState` or `MediaPlayer.StoppedState`.

Using the `autoPlay` property, the `MediaPlayer` can be made to attempt go to the playing state as soon as a the `source` property is changed. A similar property is the `autoLoad` causing the player to try to load the media as soon as the `source` property is changed. The latter property is enabled by default.

It is also possible to let the `MediaPlayer` to loop a media item. The `loops` property controls how many times the `source` is to be played. Setting the property to `MediaPlayer.Infinite` causes endless looping. Great for continious animations or a looping background song.

Sound Effects

When playing sound effects, the response time from requesting playback until actually playing becomes important. In this situation, the `SoundEffect` element comes in handy. By setting up the `source` property, a simple call to the `play` function immediately starts playback.

This can be utilized for audio feedback when tapping the screen, as shown below.

```

SoundEffect {
    id: beep
    source: "beep.wav"
}

Rectangle {
    id: button

    anchors.centerIn: parent

    width: 200
    height: 100

    color: "red"

```

```
MouseArea {
    anchors.fill: parent
    onClicked: beep.play()
}
```

The element can also be utilized to accompany a transition with audio. To trigger playback from a transition, the `ScriptAction` element is used.

```
SoundEffect {
    id: swosh
    source: "swosh.wav"
}

transitions: [
    Transition {
        ParallelAnimation {
            ScriptAction { script: swosh.play(); }
            PropertyAnimation { properties: "rotation"; duration: 200; }
        }
    }
]
```

In addition to the `play` function, a number of properties similar to the ones offered by `MediaPlayer` are available. Examples are `volume` and `loops`. The latter can be set to `SoundEffect.Infinite` for infinite playback. To stop playback, call the `stop` function.

Note: When the `PulseAudio` backend is used, `stop` will not stop instantaneously, but only prevent further loops. This is due to limitations in the underlying API.

Video Streams

The `VideoOutput` element is not limited to usage in combination with `MediaPlayer` elements. It can also be used directly with video sources to show a live video stream. Using a `Camera` element as `source` and the application is complete. The video stream from a `Camera` can be used to provide a live stream to the user. This stream works as the search view when capturing photos.

```
import QtQuick 2.5
import QtMultimedia 5.6

Item {
    width: 1024
    height: 600

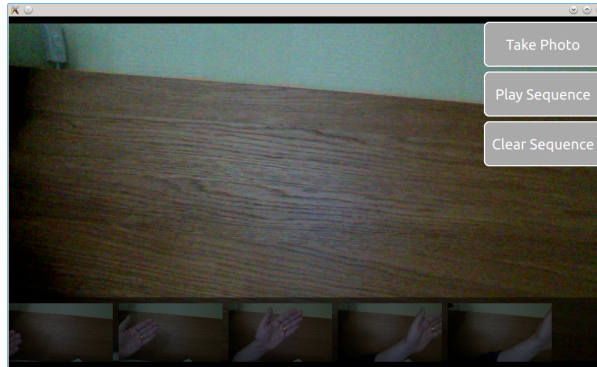
    VideoOutput {
        anchors.fill: parent
        source: camera
    }

    Camera {
        id: camera
    }
}
```


Capturing Images

One of the key features of the `Camera` element is that it can be used to take pictures. We will use this in a simple stop-motion application. In it, you will learn how to show a viewfinder, snap photos and to keep track of the pictures taken.

The user interface is shown below. It consists of three major parts. In the background, you will find the viewfinder, to the right, a column of buttons and at the bottom, a list of images taken. The idea is to take a series of photos, then click the Play Sequence button. This will play the images back, creating a simple stop-motion film.



The viewfinder part of the camera is simply a `Camera` element used as source in a `VideoOutput`. This will show the user a live videostream from the camera.

```
VideoOutput {
    anchors.fill: parent
    source: camera
}

Camera {
    id: camera
}
```

The list of photos is a `ListView` oriented horizontally shows images from a `ListModel` called `imagePaths`. In the background, a semi-transparent black `Rectangle` is used.

```
ListModel {
    id: imagePaths
}

ListView {
    id: listView

    anchors.left: parent.left
    anchors.right: parent.right
    anchors.bottom: parent.bottom
    anchors.bottomMargin: 10

    height: 100

    orientation: ListView.Horizontal
    spacing: 10

    model: imagePaths

    delegate: Image {
        height: 100
        source: path
        fillMode: Image.PreserveAspectFit
    }
}
```

```
    }

    Rectangle {
        anchors.fill: parent
        anchors.topMargin: -10

        color: "black"
        opacity: 0.5
    }
}
```

For the shooting of images, you need to know that the `Camera` element contains a set of sub-elements for various tasks. To capture still pictures, the `Camera.imageCapture` element is used. When you call the `capture` method, a picture is taken. This results in the `Camera.imageCapture` emitting first the `imageCaptured` signal followed by the `imageSaved` signal.

```
Button {
    id: shotButton

    text: "Take Photo"
    onClicked: {
        camera.imageCapture.capture();
    }
}
```

To intercept the signals of a sub-element, a `Connections` element is needed. In this case, we don't need to show a preview image, but simply add the resulting image to the `ListView` at the bottom of the screen. Shown in the example below, the path to the saved image is provided as the `path` argument with the signal.

```
Connections {
    target: camera.imageCapture

    onImageSaved: {
        imagePath.append({"path": path})
        listView.positionViewAtEnd();
    }
}
```

For showing a preview, connect to the `imageCaptured` signal and use the `preview` signal argument as source of an `Image` element. A `requestId` signal argument is sent along both the `imageCaptured` and `imageSaved`. This value is returned from the `capture` method. Using this, the capture of an image can be traced through the complete cycle. This way, the preview can be used first and then be replaced by the properly saved image. This, however, is nothing that we do in the example.

The last part of the application is the actual playback. This is driven using a `Timer` element and some JavaScript. The `_imageIndex` variable is used to keep track of the currently shown image. When the last image has been shown, the playback is stopped. In the example, the `root.state` is used to hide parts of the user interface when playing the sequence.

```
property int _imageIndex: -1

function startPlayback()
{
    root.state = "playing";
    setImageIndex(0);
    playTimer.start();
}

function setImageIndex(i)
{
    _imageIndex = i;
}
```

```

    if (_imageIndex >= 0 && _imageIndex < imagePaths.count)
        image.source = imagePaths.get(_imageIndex).path;
    else
        image.source = "";
}

Timer {
    id: playTimer

    interval: 200
    repeat: false

    onTriggered: {
        if (_imageIndex + 1 < imagePaths.count)
        {
            setImageIndex(_imageIndex + 1);
            playTimer.start();
        }
        else
        {
            setImageIndex(-1);
            root.state = "";
        }
    }
}

```

Advanced Techniques

Todo

The Camera API of Qt 5 is really lacking in documentation right now. I would love to cover more advanced camera controls such as exposure and focusing, but there are no ranges or values, nor clear guides to how to use the APIs in the reference docs right now.

Implementing a Playlist

The Qt 5 multimedia API does not provide support for playlists. Luckily, it is easy to build one. The idea is to be able to set it up with a model of items and a `MediaPlayer` element, as shown below. The `Playlist` element is responsible for setting the `source` of the `MediaPlayer`, while the playstate is controlled via the player.

```

MediaPlayer {
    id: player
    playlist: Playlist {
        PlaylistItem { source: "trailer_400p.ogg" }
        PlaylistItem { source: "trailer_400p.ogg" }
        PlaylistItem { source: "trailer_400p.ogg" }
    }
}

```

The first half of the `Playlist` element, shown below, takes care of setting the `source` element given an index in the `setIndex` function. It also implements the `next` and `previous` functions to navigate the list.

```

Item {
    id: root

```

```
property int index: 0
property MediaPlayer mediaPlayer
property ListModel items: ListModel {}

function setIndex(i) {
    console.log("setting index to: " + i);

    index = i;

    if (index < 0 || index >= items.count) {
        index = -1;
        mediaPlayer.source = "";
    } else {
        mediaPlayer.source = items.get(index).source;
    }
}

function next() {
    setIndex(index + 1);
}

function previous() {
    setIndex(index + 1);
}
```

The trick to make the playlist continue to the next element at the end of each element is to monitor the `status` property of the `MediaPlayer`. As soon as the `MediaPlayer.EndOfMedia` state is reached, the index is increased and playback resumed, or, if the end of the list is reached, the playback is stopped.

```
Connections {
    target: root.mediaPlayer

    onStopped: {
        if (root.mediaPlayer.status == MediaPlayer.EndOfMedia) {
            root.next();
            if (root.index == -1) {
                root.mediaPlayer.stop();
            } else {
                root.mediaPlayer.play();
            }
        }
    }
}
```

Summary

The media API provided by Qt provides mechanisms for playing and capturing video and audio. Through the `VideoOutput` element and video source can be displayed in the user interface. Through the `MediaPlayer` element, most playback can be handled, even though the `SoundEffect` can be used for low-latency sounds. For capturing, or only showing a live video stream, the `Camera` element is used.

Networking

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

Qt 5 comes with a rich set of networking classes on the C++ side. There are for example high level classes on the http protocol layer in a request-reply fashion such as `QNetworkRequest`, `QNetworkReply` and `QNetworkAccessManager`. But also lower levels classes on the TCP/IP or UDP protocol layer such as `QTcpSocket`, `QTcpServer` and `QUdpSocket`. Additional classes exists to manage proxies, network cache and also the systems network configuration.

This chapter will not be about C++ networking, this chapter is about Qt Quick and networking. So how can I connect my QML/JS user interface directly with a network service or how can I serve my user interface via a network service. There are good books and references out there to cover network programming with Qt/C++. Then it is just a manner to read the chapter about C++ integration to come up with an integration layer to feed your data into the Qt Quick world.

Serving UI via HTTP

To load a simple user interface via HTTP we need to have a web-server, which serves the UI documents. We start of with our own simple web-server using a python one-liner. But first we need to have our demo user interface. For this we create a small `main.qml` file in our project folder and create a red rectangle inside.

```
// main.qml
import QtQuick 2.5

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'
}
```

To serve this file we launch a small python script:

```
$ cd <PROJECT>
# python -m SimpleHTTPServer 8080
```

Now our file should be reachable via `http://localhost:8080/main.qml`. You can test it with:

```
$ curl http://localhost:8080/main.qml
```

Or just point your browser to the location. Your browser does not understand QML and will not be able to render the document through. We need to create now such a browser for QML documents. To render the document we need to point our `qmlscene` to the location. Unfortunately the `qmlscene` is limited to local files only. We could overcome this limitation by writing our own `qmlscene` replacement or simply dynamically load it using QML. We choose the dynamic loading as it works just fine. For this we use a loader element to retrieve for us the remote document.

```
// remote.qml
import QtQuick 2.5

Loader {
    id: root
    source: 'http://localhost:8080/main2.qml'
    onLoad: {
        root.width = item.width
        root.height = item.height
    }
}
```

Now we can ask the `qmlscene` to load the local `remote.qml` loader document. There is one glitch still. The loader will resize to the size of the loaded item. And our `qmlscene` needs also to adapt to that size. This can be accomplished using the `--resize-to-root` option to the `qmlscene`:

```
$ qmlscene --resize-to-root remote.qml
```

Resize to root tells the `qml scene` to resize its window to the size of the root element. The remote is now loading the `main.qml` from our local server and resizes itself to the loaded user interface. Sweet and simple.

Note: If you do not want to run a local server you can also use the gist service from GitHub. Gist is a clipboard like online service like PasteBin and others. It is available under <https://gist.github.com>. I created for this example a small gist under the url <https://gist.github.com/jryannel/7983492>. This will reveal a green rectangle. As the gist url will provide the web-site as HTML code we need to attach a `/raw` to the url to retrieve the raw file and not the HTML code.

```
// remote.qml
import QtQuick 2.5

Loader {
    id: root
    source: 'https://gist.github.com/jryannel/7983492/raw'
    onLoad: {
        root.width = item.width
        root.height = item.height
    }
}
```

To load another file over the network you just need to reference the component name. For example a `Button.qml` can be accessed as normal, as long it is in the same remote folder.

Todo

Is this true? What are the rules?

Networked Components

Let us create a small experiment. We add to our remote side a small button as a reusable component.

```
- src/main.qml
- src/Button.qml
```

We modify our `main.qml` to use the button and save it as `main2.qml`:

```
import QtQuick 2.5

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Button {
        anchors.centerIn: parent
        text: 'Click Me'
        onClicked: Qt.quit()
    }
}
```

And launch our web-server again:

```
$ cd src
# python -m SimpleHTTPServer 8080
```

And our remote loader loads the main QML via http again:

```
$ qmlscene --resize-to-root remote.qml
```

What we see is an error:

```
http://localhost:8080/main2.qml:11:5: Button is not a type
```

So QML can not resolve the button component when it is loaded remotely. If the code would be locally `qmlscene src/main.qml` this would be no issue. Locally Qt can parse the directory and detect which components are available but remotely there is no “list-dir” function for http. We can force QML to load the element using the `import` statement inside `main.qml`:

```
import "http://localhost:8080" as Remote

...

Remote.Button { ... }
```

This will work then when the `qmlscene` is run again:

```
$ qmlscene --resize-to-root remote.qml
```

Here the full code:

```
// main2.qml
import QtQuick 2.5
import "http://localhost:8080" 1.0 as Remote

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Remote.Button {
        anchors.centerIn: parent
        text: 'Click Me'
        onClicked: Qt.quit()
    }
}
```

```
}  
}
```

A better option is to use the `qmlDir` file on the server side to control the export.

```
// qmlDir  
Button 1.0 Button.qml
```

And then updating the `main.qml`:

```
import "http://localhost:8080" 1.0 as Remote  
  
...  
  
Remote.Button { ... }
```

Note: Loading

When using components from a local file system, they are created immediately without a latency. When components are loaded via the network they are created asynchronously. This has the effect that the time of creation is unknown and an element may not yet be fully loaded when others are already completed. Take this into account when working with components loaded over the network.

Templating

When working with HTML projects they often use template driven development. A small HTML stub is expanded on the server side with code generated by the server using a template mechanism. For example for a photo list the list header would be coded in HTML and the dynamic image list would be dynamically generated using a template mechanism. In general this can also be done using QML but there are some issues with it.

First it is not necessary. The reason HTML developers are doing this is to overcome limitations on the HTML backend. There is no component model yet in HTML so dynamic aspects have to be covered using these mechanism or using programmatically javascript on the client side. Many JS frameworks are out there (jQuery, dojo, backbone, angular, ...) to solve this issue and put more logic into the client-side browser to connect with a network service. The client would then just use a web-service API (e.g. serving JSON or XML data) to communicate with the server. This seems also the better approach for QML.

The second issue is the component cache from QML. When QML accesses a component it caches the render-tree and just loads the cached version for rendering. A modified version on disk or remote would not be detected without restarting the client. To overcome this issue we could use a trick. We could use URL fragments to load the url (e.g. <http://localhost:8080/main.qml#1234>), where ‘#1234’ is the fragment. The HTTP server serves always the same document but QML would store this document using the full URL, including the fragment. Every time we would access this URL the fragment would need to change and the QML cache would not get a positive hit. A fragment could be for example the current time in milli seconds or a random number.

```
Loader {  
    source: 'http://localhost:8080/main.qml#' + new Date().getTime()  
}
```

In summary templating is possible but not really recommended and does not play to the strength of QML. A better approach is to use web-services which serve JSON or XML data.

HTTP Requests

A http request is in Qt typically done using `QNetworkRequest` and `QNetworkReply` from the c++ site and then the response would be pushed using the Qt/C++ integration into the QML space. So we try to push the

envelope here a little bit to use the current tools Qt Quick gives us to communicate with a network endpoint. For this we use a helper object to make http request, response cycle. It comes in the form of the java script XMLHttpRequest object.

The XMLHttpRequest object allows the user to register a response handle function and a url. A request can be sent using one of the http verbs (get, post, put, delete, ...) to make the request. When the response arrive the handle function is called. The handle function is called several times. Every-time the request state has changed (for example headers have arrived or request is done).

Here a short example:

```
function request() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
            print('HEADERS_RECEIVED');
        } else if (xhr.readyState === XMLHttpRequest.DONE) {
            print('DONE');
        }
    }
    xhr.open("GET", "http://example.com");
    xhr.send();
}
```

For a response you can get the XML format or just the raw text. It is possible to iterate over the resulting XML but more commonly used is the raw text nowadays for a JSON formatted response. The JSON document will be used to convert text to a JS object using `JSON.parse(text)`.

```
...
} else if (xhr.readyState === XMLHttpRequest.DONE) {
    var object = JSON.parse(xhr.responseText.toString());
    print(JSON.stringify(object, null, 2));
}
```

In the response handler, we access the raw response text and convert it into a javascript object. This JSON object is now a valid JS object (in javascript an object can be an object or an array).

Note: It seems the `toString()` conversion first makes the code more stable. Without the explicit conversion I had several times parser errors. Not sure what the cause it.

Flickr Calls

Let us have a look on a more real world example. A typical example is to use the Flickr service to retrieve a public feed of the new uploaded pictures. For this we can use the `http://api.flickr.com/services/feeds/photos_public.gne` url. Unfortunately it returns by default an XML stream, which could be easily parsed by the `XmlListModel` in qml. For the sake of the example we would like to concentrate on JSON data. To become a clean JSON response we need to attach some parameters to the request: `http://api.flickr.com/services/feeds/photos_public.gne?format=json&nojsoncallback=1`. This will return a JSON response without the JSON callback.

Note: A JSON callback wraps the JSON response into a function call. This is a shortcut used on HTML programming where a script tag is used to make a JSON request. The response will trigger a local function defined by the callback. There is no mechanism which works with JSON callbacks in QML.

Let us first examine the response by using curl:

```
curl "http://api.flickr.com/services/feeds/photos_public.gne?format=json&
↳nojsoncallback=1&tags=munich"
```

The response will be something like this:

```
{
  "title": "Recent Uploads tagged munich",
  ...
  "items": [
    {
      "title": "Candle lit dinner in Munich",
      "media": {"m": "http://farm8.staticflickr.com/7313/11444882743_2f5f87169f_m.
↳jpg"},
      ...
    }, {
      "title": "Munich after sunset: a train full of \"must haves\" =",
      "media": {"m": "http://farm8.staticflickr.com/7394/11443414206_a462c80e83_m.
↳jpg"},
      ...
    }
  ]
  ...
}
```

The returned JSON document has a defined structure. An object which has a title and an items property. Where the title is a string and items is an array of objects. When converting this text into a JSON document you can access the individual entries, as it is a valid JS object/array structure.

```
// JS code
obj = JSON.parse(response);
print(obj.title) // => "Recent Uploads tagged munich"
for(var i=0; i<obj.items.length; i++) {
  // iterate of the items array entries
  print(obj.items[i].title) // title of picture
  print(obj.items[i].media.m) // url of thumbnail
}
```

As a valid JS array we can use the `obj.items` array also as a model for a list view. We will try to accomplish this now. First we need to retrieve the response and convert it into a valid JS object. And then we can just set the `response.items` property as a model to a list view.

```
function request() {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if(...) {
      ...
    } else if(xhr.readyState === XMLHttpRequest.DONE) {
      var response = JSON.parse(xhr.responseText.toString());
      // set JS object as model for listview
      view.model = response.items;
    }
  }
  xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?
↳format=json&nojsoncallback=1&tags=munich");
  xhr.send();
}
```

Here is the full source code, where we create the request, when the component is loaded. The request response is then used as model for our simple list view.

```
import QtQuick 2.5
```

```

Rectangle {
    width: 320
    height: 480
    ListView {
        id: view
        anchors.fill: parent
        delegate: Thumbnail {
            width: view.width
            text: modelData.title
            iconSource: modelData.media.m
        }
    }

    function request() {
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
                print('HEADERS_RECEIVED')
            } else if (xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE')
                var json = JSON.parse(xhr.responseText.toString())
                view.model = json.items
            }
        }
        xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?
→format=json&nojsoncallback=1&tags=munich");
        xhr.send();
    }

    Component.onCompleted: {
        request()
    }
}

```

When the document is fully loaded (`Component.onCompleted`) we request the latest feed content from Flickr. On arrival we parse the JSON response and set the `items` array as the model for our view. The list view has a delegate, which displays the thumbnail icon and the title text in a row.

An other option would be to have a placeholder `ListModel` and append each item onto the list model. To support larger models it is required to support pagination (e.g page 1 of 10) and lazy content retrieval.

Local files

Is it also possible to load local (XML/JSON) files using the `XMLHttpRequest`. For example a local file named “colors.json” can be loaded using:

```
xhr.open("GET", "colors.json");
```

We use this to read a color table and display it as a grid. It is not possible to modify the file from the Qt Quick side. To store data back to the source we would need a small REST based HTTP server or a native Qt Quick extension for file access.

```

import QtQuick 2.5

Rectangle {
    width: 360
    height: 360
    color: '#000'

    GridView {

```

```
        id: view
        anchors.fill: parent
        cellWidth: width/4
        cellHeight: cellWidth
        delegate: Rectangle {
            width: view.cellWidth
            height: view.cellHeight
            color: modelData.value
        }
    }

    function request() {
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
                print('HEADERS_RECEIVED')
            } else if (xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE');
                var obj = JSON.parse(xhr.responseText.toString());
                view.model = obj.colors
            }
        }
        xhr.open("GET", "colors.json");
        xhr.send();
    }

    Component.onCompleted: {
        request()
    }
}
```

Instead of using the XMLHttpRequest is also possible to use the XmlListModel to access local files.

```
import QtQuick.XmlListModel 2.0

XmlListModel {
    source: "http://localhost:8080/colors.xml"
    query: "/colors"
    XmlRole { name: 'color'; query: 'name/string()' }
    XmlRole { name: 'value'; query: 'value/string()' }
}
```

With the XmlListModel it is only possible to read XML files and not JSON files.

REST API

To use a web-service, we first need to create one. We will use Flask (<http://flask.pocoo.org>) a simple HTTP app server based on python to create a simple color web-service. You could also use every other web server which accepts and returns JSON data. The idea is to have a list of named colors, which can be managed via the web-service. Managed in this case means CRUD (create-read-update-delete).

A simple web-service in Flask can be written in one file. We start with an empty `server.py` file. Inside this file, we create some boiler-code and load our initial colors from an external JSON file. See also the Flask [quickstart](#) documentation.

```
from flask import Flask, jsonify, request
import json

colors = json.load(file('colors.json', 'r'))
```

```
app = Flask(__name__)

# ... service calls go here

if __name__ == '__main__':
    app.run(debug = True)
```

When you run this script, it will provide a web-server at <http://localhost:5000>, which does not serve anything useful yet.

We will now start adding our CRUD (Create,Read,Update,Delete) endpoints to our little web-service.

Read Request

To read data from our web-server, we will provide a GET method for all colors.

```
@app.route('/colors', methods = ['GET'])
def get_colors():
    return jsonify( { "colors" : colors })
```

This will return the colors under the '/colors' endpoint. To test this we can use curl to create a http request.

```
curl -i -GET http://localhost:5000/colors
```

Which will return us the list of colors as JSON data.

Read Entry

To read an individual color by name we provide the details endpoint, which is located under '/colors/<name>'. The name is a parameter to the endpoint, which identifies an individual color.

```
@app.route('/colors/<name>', methods = ['GET'])
def get_color(name):
    for color in colors:
        if color["name"] == name:
            return jsonify( color )
```

And we can test it with using curl again. For example to get the red color entry.

```
curl -i -GET http://localhost:5000/colors/red
```

It will return one color entry as JSON data.

Create Entry

Till now we have just used HTTP GET methods. To create an entry on the server side, we will use a POST method and pass the new color information with the POST data. The endpoint location is the same as to get all colors. But this time we expect a POST request.

```
@app.route('/colors', methods= ['POST'])
def create_color():
    color = {
        'name': request.json['name'],
        'value': request.json['value']
    }
    colors.append(color)
    return jsonify( color ), 201
```

Curl is flexible enough to allow us to provide JSON data as the new entry inside the POST request.

```
curl -i -H "Content-Type: application/json" -X POST -d '{"name":"gray1","value":
↪#333}' http://localhost:5000/colors
```

Update Entry

To update an individual entry we use the PUT HTTP method. The endpoint is the same as to retrieve an individual color entry. When the color was updated successfully we return the updated color as JSON data.

```
@app.route('/colors/<name>', methods= ['PUT'])
def update_color(name):
    for color in colors:
        if color["name"] == name:
            color['value'] = request.json.get('value', color['value'])
            return jsonify( color )
```

In the curl request we only provide the values to be updated as JSON data and the a named endpoint to identify the color to be updated.

```
curl -i -H "Content-Type: application/json" -X PUT -d '{"value":"#666"}' http://
↪localhost:5000/colors/red
```

Delete Entry

Deleting an entry is done using the DELETE HTTP verb. It also uses the same endpoint for an individual color, but this time the DELETE HTTP verb.

```
@app.route('/colors/<name>', methods=['DELETE'])
def delete_color(name):
    success = False
    for color in colors:
        if color["name"] == name:
            colors.remove(color)
            success = True
    return jsonify( { 'result' : success } )
```

This request looks similar like the GET request for an individual color.

```
curl -i -X DELETE http://localhost:5000/colors/red
```

Now we can read all colors, read a specific color, create a new color, update a color and delete a color. Also we know the HTTP endpoints to our API.

Action	HTTP	Endpoint
Read All	GET	http://localhost:5000/colors
Create Entry	POST	http://localhost:5000/colors
Read Entry	GET	<a href="http://localhost:5000/colors/<name>">http://localhost:5000/colors/<name>
Update Entry	PUT	<a href="http://localhost:5000/colors/<name>">http://localhost:5000/colors/<name>
Delete Entry	DELETE	<a href="http://localhost:5000/colors/<name>">http://localhost:5000/colors/<name>

Our little REST server is complete now and we can focus on QML and the client side. To create an easy to use API we need to map each action to an individual HTTP request and provide a simple API to our users.

Client REST

To demonstrate a REST client we write a small color grid. The color grid displays the colors retrieved from the web-service via HTTP requests. Our user interface provides the following commands:

- Get color list
- Create color
- Read last color
- Update last color
- Delete last color

We bundle our API into an own JS file called `colorservice.js` and import it into our UI as `Service`. Inside the service module we create a helper function to make the HTTP requests for us:

```
// colorservice.js
function request(verb, endpoint, obj, cb) {
    print('request: ' + verb + ' ' + BASE + (endpoint? '/' + endpoint: ''))
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        print('xhr: on ready state change: ' + xhr.readyState)
        if(xhr.readyState === XMLHttpRequest.DONE) {
            if(cb) {
                var res = JSON.parse(xhr.responseText.toString())
                cb(res);
            }
        }
    }
    xhr.open(verb, BASE + (endpoint? '/' + endpoint: ''));
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.setRequestHeader('Accept', 'application/json');
    var data = obj?JSON.stringify(obj):''
    xhr.send(data)
}
```

It takes four arguments. The `verb`, which defines the HTTP verb to be used (GET, POST, PUT, DELETE). The second parameter is the endpoint to be used as postfix to the BASE address (e.g. `'http://localhost:5000/colors'`). The third parameter is the optional `obj`, to be send as JSON data to the service. The last parameter defines a callback to be called, when the response returns. The callback receives a response object with the response data. Before we send the request, we indicate that we send and accept JSON data by modifying the request header.

Using this request helper function we can implement the simple commands we defined earlier (create, read, update, delete):

```
// colorservice.js
function get_colors(cb) {
    // GET http://localhost:5000/colors
    request('GET', null, null, cb)
}

function create_color(entry, cb) {
    // POST http://localhost:5000/colors
    request('POST', null, entry, cb)
}

function get_color(name, cb) {
    // GET http://localhost:5000/colors/<name>
    request('GET', name, null, cb)
}

function update_color(name, entry, cb) {
    // PUT http://localhost:5000/colors/<name>
    request('PUT', name, entry, cb)
}

function delete_color(name, cb) {
    // DELETE http://localhost:5000/colors/<name>
}
```

```
request('DELETE', name, null, cb)
}
```

This code resides in the service implementation. In the UI we use the service to implement our commands. We have a ListModel with the id gridModel as data provider for the GridView. The commands are indicated using a Button ui element.

Reading the color list from the server.

```
// rest.qml
import "colorservice.js" as Service
...
// read colors command
Button {
    text: 'Read Colors';
    onClicked: {
        Service.get_colors( function(resp) {
            print('handle get colors resp: ' + JSON.stringify(resp));
            gridModel.clear();
            var entries = resp.data;
            for(var i=0; i<entries.length; i++) {
                gridModel.append(entries[i]);
            }
        });
    }
}
```

Create a new color entry on the server.

```
// rest.qml
import "colorservice.js" as Service
...
// create new color command
Button {
    text: 'Create New';
    onClicked: {
        var index = gridModel.count-1
        var entry = {
            name: 'color-' + index,
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toString()
        }
        Service.create_color(entry, function(resp) {
            print('handle create color resp: ' + JSON.stringify(resp))
            gridModel.append(resp)
        });
    }
}
```

Reading a color based on its name.

```
// rest.qml
import "colorservice.js" as Service
...
// read last color command
Button {
    text: 'Read Last Color';
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        Service.get_color(name, function(resp) {
            print('handle get color resp: ' + JSON.stringify(resp))
            message.text = resp.value
        });
    }
}
```



```

    }
}

```

Update a color entry on the server based on the color name.

```

// rest.qml
import "colorservice.js" as Service
...
// update color command
Button {
    text: 'Update Last Color'
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        var entry = {
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toString()
        }
        Service.update_color(name, entry, function(resp) {
            print('handle update color resp: ' + JSON.stringify(resp))
            var index = gridModel.count-1
            gridModel.setProperty(index, 'value', resp.value)
        });
    }
}

```

Delete a color by the color name.

```

// rest.qml
import "colorservice.js" as Service
...
// delete color command
Button {
    text: 'Delete Last Color'
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        Service.delete_color(name)
        gridModel.remove(index, 1)
    }
}

```

This concludes the CRUD (create, read, update, delete) operations using a REST API. There are also other possibilities to generate a Web-Service API. One could be module based and each module would have an one endpoint. And the API could be defined using JSON RPC (<http://www.jsonrpc.org/>). Sure also XML based API are possible and but the JSON approach has great advantages as the parsing is build into the QML/JS as part of JavaScript.

Authentication using OAuth

OAuth is an open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. OAuth is used to authenticate a client against common web-services such as Google, Facebook and Twitter.

Note: For a custom web-service you could also use the standard HTTP authentication for example by using the XMLHttpRequest username and password in the get method (e.g. `xhr.open(verb,url,true,username,password)`)

OAuth is currently not part of a QML/JS API. So you would need to write some C++ code and export the authentication to QML/JS. Another issue would be the secure storage of the access token.

Here are some links which we find useful:

- <http://oauth.net/>
- <http://hueniverse.com/oauth/>
- <https://github.com/pipacs/o2>
- <http://www.johanpaul.com/blog/2011/05/oauth2-explained-with-qt-quick/>

Engin IO

Engin.IO is a web-service run by DIGIA. It enables to access from inside Qt/QML application to the NoSQL storage from Engin.IO. It is a cloud based storage object store with an easy access Qt/QML API and an administration console. If you want to store some data in the cloud from a QML application, this would be an easy entry path with an excellent QML/JS support.

Please refer to the [EnginIO](#) documentation for further help.

Web Sockets

The WebSockets module provides an implementation of the WebSockets protocol for WebSockets clients and servers. It mirrors the Qt CPP module. It allows to send string and binary messages using a full duplex communication channel. A websocket is normally established by making a HTTP connection to the server and the server then “upgrades” the connection to a WebSocket connection.

In Qt/QML you can also simply use the *WebSocket* and *WebSocketServer* objects to create a direct websocket connection. The websocket protocol uses the “ws” url schema or “wss” for a secure connection.

You can use the web socket qml module by importing it first.

```
import Qt.WebSockets 1.0

WebSocket {
    id: socket
}
```

To test your web socket we will use the echo server from <http://websocket.org>.

```
import QtQuick 2.5
import Qt.WebSockets 1.0

Text {
    width: 480
    height: 48

    horizontalAlignment: Text.AlignHCenter
    verticalAlignment: Text.AlignVCenter

    WebSocket {
        id: socket
        url: "ws://echo.websocket.org"
        active: true
        onTextMessageReceived: {
            text = message
        }
        onStatusChanged: {
            if (socket.status == WebSocket.Error) {
                console.log("Error: " + socket.errorString)
            } else if (socket.status == WebSocket.Open) {
                socket.sendMessage("ping")
            }
        }
    }
}
```

```

    } else if (socket.status == WebSocket.Closed) {
        text += "\nSocket closed"
    }
}
}
}

```

You should see the ping message we send `socket.sendMessage("ping")` as response in the text field.



WS Server

You can easily create your own WS server using the C++ part of the Qt WebSocket or use a different WS implementation, which I find very interesting. It is interesting because it allows to connect the amazing rendering quality of QML with the great expanding web application servers. In this example we will use a Node JS based web socket server using the `ws` module. For this you first need to install `node.js`. Afterwards create a `ws_server` folder and install the `ws` package using the node package manager (npm).

The code shall create a simple echo server in NodeJS to echo our messages back to our QML client.

```

$ cd ws_server
$ npm install ws

```

The npm tool downloads and installs the `ws` package and dependencies into your local folder.

A `server.js` file will be our server implementation. The server code will create a web socket server on port 3000 and listens to an incoming connection. On an incoming connection it will send out a greeting and waits for client messages. Each message a client sends on a socket will be send back to the client.

```

var WebSocketServer = require('ws').Server;

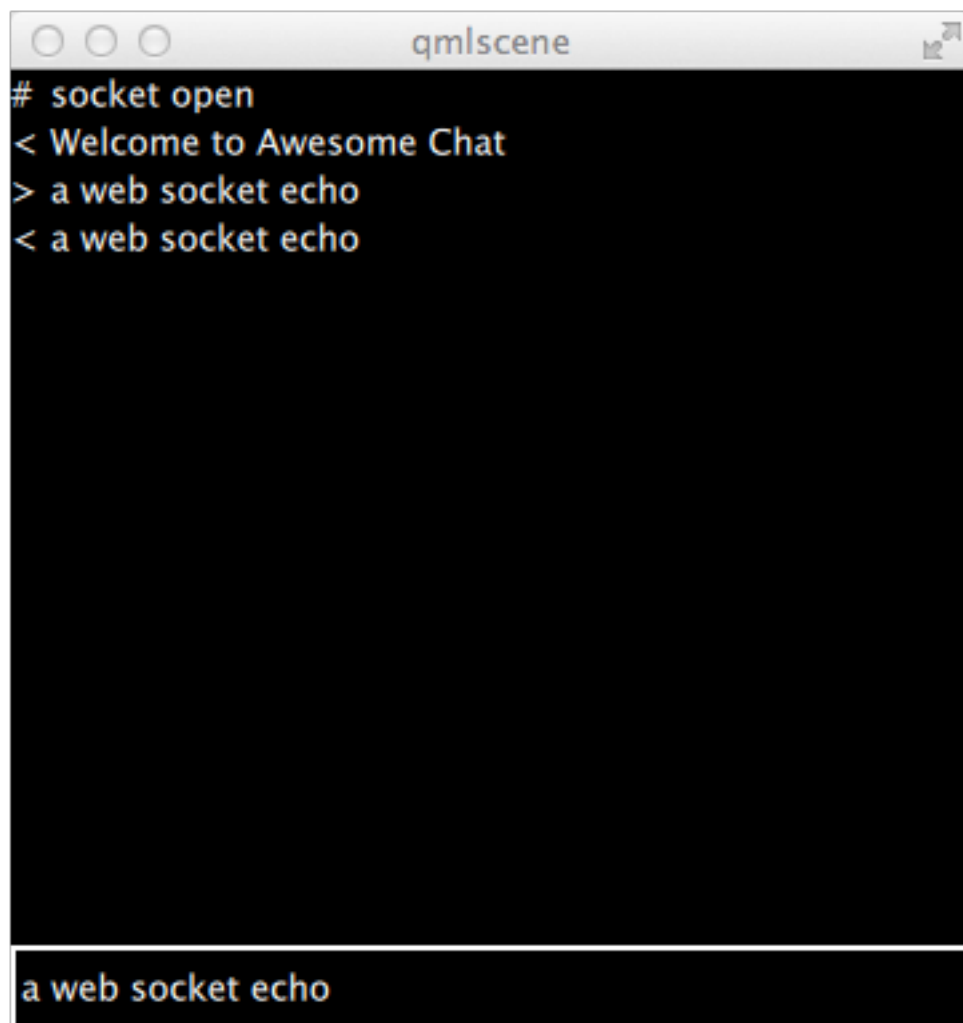
var server = new WebSocketServer({ port : 3000 });

server.on('connection', function(socket) {
    console.log('client connected');
    socket.on('message', function(msg) {
        console.log('Message: %s', msg);
        socket.send(msg);
    });
    socket.send('Welcome to Awesome Chat');
});

console.log('listening on port ' + server.options.port);

```

You need to get used to the notation of JavaScript and the function callbacks.



WS Client

On the client side we need a list view to display the messages and a `TextInput` for the user to enter a new chat message.

We will use a label with white color in the example.

```
// Label.qml
import QtQuick 2.5

Text {
    color: '#fff'
    horizontalAlignment: Text.AlignLeft
    verticalAlignment: Text.AlignVCenter
}
```

Our chat view is a list view, where the text is appended to a list model. Each entry is displayed using a row of prefix and message label. We use a cell width `cw` factor to split the width into 24 columns.

```
// ChatView.qml
import QtQuick 2.5

ListView {
    id: root
    width: 100
    height: 62

    model: ListModel {}

    function append(prefix, message) {
        model.append({prefix: prefix, message: message})
    }

    delegate: Row {
        width: root.width
        height: 18
        property real cw: width/24
        Label {
            width: cw*1
            height: parent.height
            text: model.prefix
        }
        Label {
            width: cw*23
            height: parent.height
            text: model.message
        }
    }
}
```

The chat input is just a simple text input wrapped with a colored border.

```
// ChatInput.qml
import QtQuick 2.5

FocusScope {
    id: root
    width: 240
    height: 32
    Rectangle {
        anchors.fill: parent
        color: '#000'
        border.color: '#fff'
    }
}
```

```
        border.width: 2
    }

    property alias text: input.text

    signal accepted(string text)

    TextInput {
        id: input
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.verticalCenter: parent.verticalCenter
        anchors.leftMargin: 4
        anchors.rightMargin: 4
        onAccepted: root.accepted(text)
        color: '#fff'
        focus: true
    }
}
```

When the web socket receives a message it appends the message to the chat view. Same applies for a status change. Also when the user enters a chat message a copy is appended to the chat view on the client side and the message is send to the server.

```
// ws_client.qml
import QtQuick 2.5
import Qt.WebSockets 1.0

Rectangle {
    width: 360
    height: 360
    color: '#000'

    ChatView {
        id: box
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: input.top
    }
    ChatInput {
        id: input
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        focus: true
        onAccepted: {
            print('send message: ' + text)
            socket.sendMessage(text)
            box.append('>', text)
            text = ''
        }
    }
}

WebSocket {
    id: socket

    url: "ws://localhost:3000"
    active: true
    onTextMessageReceived: {
        box.append('<', message)
    }
    onStatusChanged: {

```

```
        if (socket.status == WebSocket.Error) {
            box.append('#', 'socket error ' + socket.errorString)
        } else if (socket.status == WebSocket.Open) {
            box.append('#', 'socket open')
        } else if (socket.status == WebSocket.Closed) {
            box.append('#', 'socket closed')
        }
    }
}
```

You need first run the server and then the client. There is no retry connection mechanism in our simple client.

Running the server

```
$ cd ws_server
$ node server.js
```

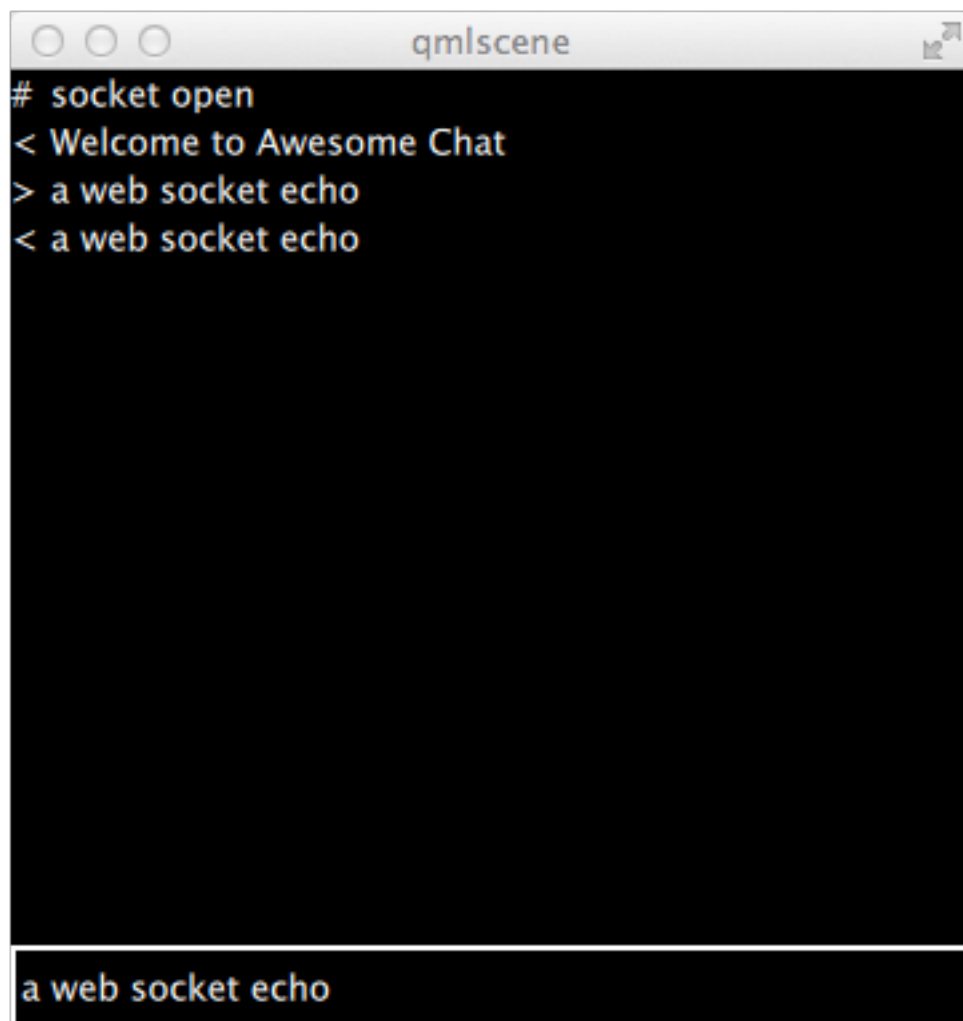
Running the client

```
$ cd ws_client
$ qmlscene ws_client.qml
```

When entering text and pressing enter you should see something like this.

Summary

This concludes our chapter about QML networking. Please bear in mind Qt has on the native side a much richer networking API as on the QML side currently. But the idea of the chapter is to push the boundaries of QML networking and how to integrate with cloud based services.



```
# socket open
< Welcome to Awesome Chat
> a web socket echo
< a web socket echo

a web socket echo
```

Storage

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

This chapter will cover storing data using Qt Quick in Qt 5. Qt Quick offers only limited ways of storing local data directly. In this sense it acts more like a browser. In many projects storing data is handled by the C++ backend and the required functionality is exported to the Qt Quick frontend side. Qt Quick does not provide you with access to the host file system to read and write files as you are used from the Qt C++ side. So it would be the task of the backend engineer to write such a plugin or maybe use a network channel to communicate with a local server, which provides these capabilities.

Every application need to store smaller and larger information persistently. This can be done locally on the file system or remote on a server. Some information will be structured and simple (e.g. settings), some will be large and complicated for example documentation files and some will be large and structured and will require some sort of database connection. Here we will mainly cover the built in capabilities of Qt Quick to store data as also the networked ways.

Settings

Qt comes on its native side with the C++ `QSettings` class, which allows you to store the application settings (aka options, preferences) in a system dependent way. It uses the infrastructure available from your OS. Additional it supports a common INI file format for handling cross platform settings files.

In Qt 5.2 `Settings` have entered the QML world. The API is still in the labs module, which means the API may break in the future. So be aware.

Here is a small example, which applies a color value to a base rectangle. Every time the user clicks on the window a new random color is generated. When the application is closed and relaunched again you should see your last color. The default color should be the color initially set on the root rectangle.

```
import QtQuick 2.5
import Qt.labs.settings 1.0

Rectangle {
    id: root
    width: 320; height: 240
    color: '#000000'
    Settings {
        id: settings
        property alias color: root.color
    }
    MouseArea {
```

```
anchors.fill: parent
onClicked: root.color = Qt.hsla(Math.random(), 0.5, 0.5, 1.0);
}
}
```

The settings value are stored every time the value changes. This might be not always what you want. To store the settings only when required you can use standard properties.

```
Rectangle {
    id: root
    color: settings.color
    Settings {
        id: settings
        property color color: '#000000'
    }
    function storeSettings() { // executed maybe on destruction
        settings.color = root.color
    }
}
```

It is also possible to store settings into different categories using the `category` property.

```
Settings {
    category: 'window'
    property alias x: window.x
    property alias y: window.y
    property alias width: window.width
    property alias height: window.height
}
```

The settings are stored according your application name, organization and domain. This information is normally set in the main function of your c++ code.

```
int main(int argc, char** argv) {
    ...
    QApplication::setApplicationName("Awesome Application");
    QApplication::setOrganizationName("Awesome Company");
    QApplication::setOrganizationDomain("org.awesome");
    ...
}
```

Local Storage - SQL

Qt Quick supports an local storage API known from the web browsers the local storage API. the API is available under “import QtQuick.LocalStorage 2.0”.

In general it stores the content into a SQLITE database in system specific location in an unique ID based file based on the given database name and version. It is not possible to list or delete existing databases. You can find the storage location from `QSqlEngine::offlineStoragePath()`.

You use the API by first creating a database object and then creating transactions on the database. Each transaction can contain one or more SQL queries. The transaction will roll-back when a SQL query will fail inside the transaction.

For example to read from a simple notes table with a text column you could use the local storage like this:

```
import QtQuick 2.5
import QtQuick.LocalStorage 2.0

Item {
```

```
Component.onCompleted: {  
    var db = LocalStorage.openDatabaseSync("MyExample", "1.0", "Example_  
↳database", 10000);  
    db.transaction( function(tx) {  
        var result = tx.executeSql('select * from notes');  
        for(var i = 0; i < result.rows.length; i++) {  
            print(result.rows[i].text);  
        }  
    });  
}
```

Crazy Rectangle

As an example assume we would like to store the position of a rectangle on our scene.



Here our base example.

```
import QtQuick 2.5

Item {
    width: 400
    height: 400

    Rectangle {
        id: crazy
        objectName: 'crazy'
        width: 100
        height: 100
        x: 50
        y: 50
        color: "#53d769"
        border.color: Qt.lighter(color, 1.1)
        Text {
            anchors.centerIn: parent
            text: Math.round(parent.x) + '/' + Math.round(parent.y)
        }
        MouseArea {
            anchors.fill: parent
            drag.target: parent
        }
    }
}
```

You can drag the rectangle freely around. When you close the application and launch it again the rectangle is at the same position.

Now we would like to add that the x/y position of the rectangle is stored inside the SQL DB. For this we need to add an `init`, `read` and `store` database function. These function are called when on component completed and on component destruction.

```
import QtQuick 2.5
import QtQuick.LocalStorage 2.0

Item {
    // reference to the database object
    property var db;

    function initDatabase() {
        // initialize the database object
    }

    function storeData() {
        // stores data to DB
    }

    function readData() {
        // reads and applies data from DB
    }

    Component.onCompleted: {
        initDatabase();
        readData();
    }

    Component.onDestruction: {
        storeData();
    }
}
```

You could also extract the DB code in an own JS library, which does all the logic. This would be the preferred way if the logic gets more complicated.

In the database initialization function we create the DB object and ensure the SQL table is created.

```
function initDatabase() {
    print('initDatabase()')
    db = LocalStorage.openDatabaseSync("CrazyBox", "1.0", "A box who remembers its_
    ↪position", 100000);
    db.transaction( function(tx) {
        print('... create table')
        tx.executeSql('CREATE TABLE IF NOT EXISTS data(name TEXT, value TEXT)');
    });
}
```

The application next calls the read function to read existing data back from the database. Here we need to differentiate if there is already data in the table. To check we look into how many rows the select clause has returned.

```
function readData() {
    print('readData()')
    if(!db) { return; }
    db.transaction( function(tx) {
        print('... read crazy object')
        var result = tx.executeSql('select * from data where name="crazy"');
        if(result.rows.length === 1) {
            print('... update crazy geometry')
            // get the value column
            var value = result.rows[0].value;
            // convert to JS object
            var obj = JSON.parse(value)
            // apply to object
            crazy.x = obj.x;
            crazy.y = obj.y;
        }
    });
}
```

We expect the data is stored a JSON string inside the value column. This is not typical SQL like, but works nicely with JS code. So instead of storing the x,y as properties in the table we store them as a complete JS object using the JSON stringify/parse methods. At the end we get a valid JS object with x and y properties, which we can apply on our crazy rectangle.

To store the data, we need to differentiate the update and insert cases. We use update when a record already exists and insert if no record under the name “crazy” exists.

```
function storeData() {
    print('storeData()')
    if(!db) { return; }
    db.transaction( function(tx) {
        print('... check if a crazy object exists')
        var result = tx.executeSql('SELECT * from data where name = "crazy"');
        // prepare object to be stored as JSON
        var obj = { x: crazy.x, y: crazy.y };
        if(result.rows.length === 1) { // use update
            print('... crazy exists, update it')
            result = tx.executeSql('UPDATE data set value=? where name="crazy"',
            ↪[JSON.stringify(obj)]);
        } else { // use insert
            print('... crazy does not exists, create it')
            result = tx.executeSql('INSERT INTO data VALUES (?,?)', ['crazy', JSON.
            ↪stringify(obj)]);
        }
    });
}
```

```
}
```

Instead of selecting the whole record set we could also use the SQLITE count function like this: `SELECT COUNT(*) from data where name = "crazy"` which would return use one row with the amount of rows affected by the select query. Otherwise this is common SQL code. As an additional feature, we use the SQL value binding using the `?` in the query.

Now you can drag the rectangle and when you quit the application the database stores the x/y position and applies it on the next application run.

Other Storage APIs

To store directly from within QML these are the major storage types. The real strength of Qt Quick comes from the fact to extend it with C++ to interface with your native storage systems or use the network API to interface with a remote storage system, like the Qt cloud.

Dynamic QML

Section author: e8johan

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

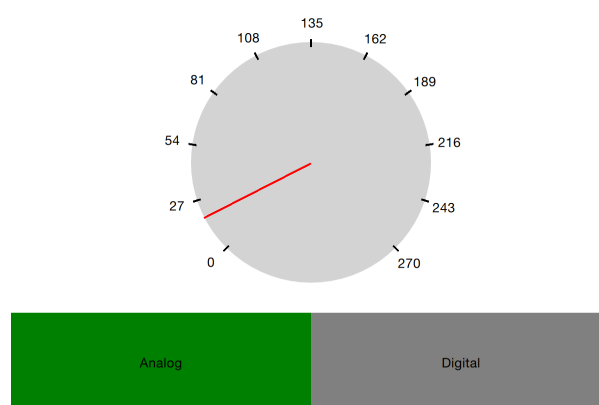
Until now, we have treated QML as a tool for constructing a static set of scenes and navigating between them. Depending on various states and logic rules, a live and dynamic user interface is constructed. By working with QML and JavaScript in a more dynamic manner, the flexibility and possibilities expand even further. Components can be loaded and instantiated at run-time, elements can be destroyed. Dynamically created user interfaces can be saved to disk and later restored.

Loading Components Dynamically

The easiest way to dynamically load different parts of QML is to use the `Loader` element. It serves as a placeholder to the item that is being loaded. The item to load is controlled through either the `source` property or the `sourceComponent` property. The former loads the item from a given URL, while the latter instantiates a component.

As the loader serves as a placeholder for the item being loaded, its size depends on the size of the item, and vice versa. If the `Loader` element has a size, either by having set `width` and `height` or through anchoring, the loaded item will be given the loader's size. If the `Loader` has no size, it is resized in accordance to the size of the item being loaded.

The example described below demonstrates how two separate user interface parts can be loaded into the same space using a `Loader` element. The idea is to have a speed dial that can be either digital or analog, as shown in the illustration below. The code surrounding the dial is unaffected by which item that is loaded for the moment.



41 kph



The first step in the application is to declare a `Loader` element. Notice that the `source` property is left out. This is because the `source` depends on which state the user interface is in.

```
Loader {
    id: dialLoader

    anchors.fill: parent
}
```

In the `states` property of the parent of `dialLoader` a set of `PropertyChanges` elements drives the loading of different QML files depending on the state. The `source` property happens to be a relative file path in this example, but it can just as well be a full URL, fetching the item over the web.

```
states: [
    State {
        name: "analog"
        PropertyChanges { target: analogButton; color: "green"; }
        PropertyChanges { target: dialLoader; source: "Analog.qml"; }
    },
    State {
        name: "digital"
        PropertyChanges { target: digitalButton; color: "green"; }
        PropertyChanges { target: dialLoader; source: "Digital.qml"; }
    }
]
```

In order to make the loaded item come alive, its `speed` property must be bound to the root `speed` property. This cannot be done as a direct binding as the item not always is loaded and changes over time. Instead a `Binding` element must be used. The `target` property of the binding is changed every time the `Loader` triggers the `onLoaded` signal.

```
Loader {
    id: dialLoader

    anchors.left: parent.left
    anchors.right: parent.right
    anchors.top: parent.top
    anchors.bottom: analogButton.top

    onLoaded: {
        binder.target = dialLoader.item;
    }
}
Binding {
    id: binder

    property: "speed"
```



```

    value: speed
}

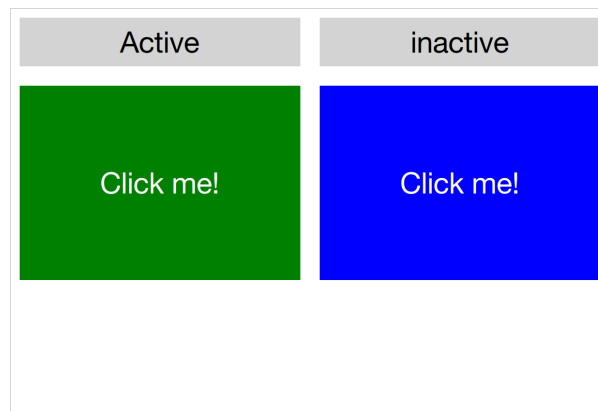
```

The `onLoaded` signal lets the loading QML act when the item has been loaded. In a similar fashion, the QML being loaded can rely on the `Component.onCompleted` signal. This is signal actually available for all components, regardless how they are loaded. For instance, the root component of an entire application can use it to kick-start itself when the entire user interface has been loaded.

Connecting Indirectly

When creating QML elements dynamically, you cannot connect to signals using the `onSignalName` approach used for static setup. Instead, the `Connections` element must be used. It connects to any number of signals of a target element.

Having set the `target` property of a `Connections` element, the signals can be connected as usual, that is, using the `onSignalName` approach. However, by altering the `target` property, different elements can be monitored at different times.



In the example shown above, a user interface consisting of two clickable areas is presented to the user. When either area is clicked, it is flashed using an animation. The left area is shown in the code snippet below. In the `MouseArea`, the `leftClickedAnimation` is triggered, causing the area to flash.

```

Rectangle {
    id: leftRectangle

    width: 290
    height: 200

    color: "green"

    MouseArea {
        id: leftMouseArea
        anchors.fill: parent
        onClicked: leftClickedAnimation.start();
    }

    Text {
        anchors.centerIn: parent
        font.pixelSize: 30
        color: "white"
        text: "Click me!"
    }
}

```

In addition to the two clickable areas, a `Connections` element is used. This triggers a third animation when the active, i.e. the target of the element, is clicked.

```
Connections {
    id: connections
    onClicked: activeClickedAnimation.start();
}
```

To determine which `MouseArea` to target, two states are defined. Notice that we cannot set the `target` property using a `PropertyChanges` element, as it already contains a `target` property. Instead a `StateChangeScript` is utilized.

```
states: [
    State {
        name: "left"
        StateChangeScript {
            script: connections.target = leftMouseArea
        }
    },
    State {
        name: "right"
        StateChangeScript {
            script: connections.target = rightMouseArea
        }
    }
]
```

When trying out the example, it is worth noticing that when multiple signal handlers are used, all are invoked. The execution order of these is, however, undefined.

When creating a `Connections` element without setting the `target` property, the property defaults to `parent`. This means that it explicitly has to be set to `null` to avoid catching signals from the `parent` until the `target` is set. This behavior does make it possible to create custom signal handler components based on a `Connections` element. This way, the code reacting to the signals can be encapsulated and re-used.

In the example below, the `Flasher` component can be put inside any `MouseArea`. When clicked, it triggers an animation, causing the parent to flash. In the same `MouseArea` the actual task being triggered can also be carried out. This separates the standardized user feedback, i.e. the flashing, from the actual action.

```
import QtQuick 2.5

Connections {
    onClicked: {
        // Automatically targets the parent
    }
}
```

To use the `Flasher`, simply instantiate a `Flasher` within each `MouseArea`, and it all works.

```
import QtQuick 2.5

Item {
    // A background flasher that flashes the background of any parent MouseArea
}
```

When using a `Connections` element to monitor the signals of multiple types of target elements, you sometimes find yourself in a situation where the available signals vary between the targets. This results in the `Connections` element outputting run-time errors as signals are missed. To avoid this, the `ignoreUnknownSignal` property can be set to `true`. This ignores all such errors.

Note: It is usually a bad idea to suppress error messages.

Binding Indirectly

Just as it is not possible to connect to signals of dynamically created elements directly, nor it is possible to bind properties of a dynamically created element without working with a bridge element. To bind a property of any element, including dynamically created elements, the `Binding` element is used.

The `Binding` element lets you specify a target element, a property to bind and a value to bind it to. Through using a *Binding* element, it is, for instance, possible to bind properties of a dynamically loaded element. This was demonstrated in the introductory example in this chapter, as shown below.

```
Loader {
    id: dialLoader

    anchors.left: parent.left
    anchors.right: parent.right
    anchors.top: parent.top
    anchors.bottom: analogButton.top

    onLoaded: {
        binder.target = dialLoader.item;
    }
}
Binding {
    id: binder

    property: "speed"
    value: speed
}
```

As the target element of a `Binding` not always is set, and perhaps not always has a given property, the `when` property of the `Binding` element can be used to limit the time when the binding is active. For instance, it can be limited to specific modes in the user interface.

Creating and Destroying Objects

The `Loader` element makes it possible to populate part of a user interface dynamically. However, the overall structure of the interface is still static. Through JavaScript it is possible to take one more step and to instantiate QML elements completely dynamically.

Before we dive into the details of creating elements dynamically, we need to understand the workflow. When loading a piece of QML from a file or even over the Internet, a component is created. The component encapsulates the interpreted QML code and can be used to create items. This means that loading a piece of QML code and instantiating items from it is a two stage process. First the QML code is parsed into a component. Then the component is used to instantiate actual item objects.

In addition to creating elements from QML code stored in files or on servers, it is also possible to create QML objects directly from text strings containing QML code. The dynamically created items are then treated in a similar fashion once instantiated.

Dynamically Loading and Instantiating Items

When loading a piece of QML, it is first interpreted into a component. This includes loading dependencies and validating the code. The location of the QML being loaded can be either a local file, a Qt resource, or even a distance network location specified by a URL. This means that the loading time can be everything from instant, for instance a Qt resource located in RAM without any non-loaded dependencies, to very long, meaning a piece of code located on a slow server with multiple dependencies that needs to be loaded.

The status of a component being created can be tracked by its `status` property. The available values are `Component.Null`, `Component.Loading`, `Component.Ready` and `Component.Error`. The `Null`

to Loading to Ready is the usual flow. At any stage the status can change to Error. In that case, the component cannot be used to create new object instances. The `Component.errorString()` function can be used to retrieve a user readable error description.

When loading components over slow connections, the `progress` property can be of use. It ranges from 0.0, meaning nothing has been loaded, to 1.0 indicating that all has been loaded. When the component's status changes to Ready, the component can be used to instantiate objects. The code below demonstrates how that can be achieved, taking into account the event of the component becoming ready or failing to be created directly, as well as the case where the component is ready slightly later.

```
var component;

function createImageObject() {
    component = Qt.createComponent("dynamic-image.qml");
    if (component.status === Component.Ready || component.status === Component.
    ↪Error) {
        finishCreation();
    } else {
        component.statusChanged.connect(finishCreation);
    }
}

function finishCreation() {
    if (component.status === Component.Ready) {
        var image = component.createObject(root, {"x": 100, "y": 100});
        if (image === null) {
            console.log("Error creating image");
        }
    } else if (component.status === Component.Error) {
        console.log("Error loading component:", component.errorString());
    }
}
```

The code above is kept in a separate JavaScript source file, referenced from the main QML file.

```
import QtQuick 2.5
import "create-component.js" as ImageCreator

Item {
    id: root

    width: 1024
    height: 600

    Component.onCompleted: ImageCreator.createImageObject();
}
```

The `createObject` function of a component is used to create object instances, as shown above. This not only applies to dynamically loaded components, but also `Component` elements inlined in the QML code. The resulting object can be used in the QML scene like any other object. The only difference is that it does not have an `id`.

The `createObject` function takes two arguments. The first is a parent object of the type `Item`. The second is a list of properties and values on the format `{"name": value, "name": value}`. This is demonstrated in the example below. Notice that the properties argument is optional.

```
var image = component.createObject(root, {"x": 100, "y": 100});
```

Note: A dynamically created component instance is not different to an in-line `Component` element. The in-line `Component` element also provides functions to instantiate objects dynamically.

Dynamically Instantiating Items from Text

Sometimes, it is convenient to be able to instantiate an object from a text string of QML. If nothing else, it is quicker than putting the code in a separate source file. For this, the `Qt.createQmlObject` function is used.

The function takes three arguments: `qml`, `parent` and `filepath`. The `qml` argument contains the string of QML code to instantiate. The `parent` argument provides a parent object to the newly created object. The `filepath` argument is used when reporting any errors from the creation of the object. The result returned from the function is either a new object, or `null`.

Warning: The `createQmlObject` function always returns immediately. For the function to succeed, all the dependencies of the call must be loaded. This means that if the code passed to the function refers to a non-loaded component, the call will fail and return `null`. To better handle this, the `createComponent / createObject` approach must be used.

The objects created using the `Qt.createQmlObject` function resembles any other dynamically created object. That means that it is identical to every other QML object, apart from not having an `id`. In the example below, a new `Rectangle` element is instantiated from in-line QML code when the `root` element has been created.

```
import QtQuick 2.5

Item {
    id: root

    width: 1024
    height: 600

    function createItem() {
        Qt.createQmlObject("import QtQuick 2.5; Rectangle { x: 100; y: 100; width: 100; height: 100; color: \"blue\" }", root, "dynamicItem");
    }

    Component.onCompleted: root.createItem();
}
```

Managing Dynamically Created Elements

Dynamically created objects can be treated as any other object in a QML scene. However, there are some pitfalls that need to be handled. The most important is the concept of creation contexts.

The creation context of a dynamically created object is the context within it is being created. This is not necessarily the same context as the parent exists in. When the creation context is destroyed, so are the bindings concerning the object. This means that it is important to implement the creation of dynamic objects in a place in the code which will be instantiated during the entire life-time of the objects.

Dynamically created objects can also be dynamically destroyed. When doing this, there is a rule of thumb: never attempt to destroy an object that you have not created. This also includes elements that you have created, but not using a dynamic mechanism such as `Component.createObject` or `createQmlObject`.

An object is destroyed by calling its `destroy` function. The function takes an optional argument which is an integer specifying how many milliseconds the objects shall exist before being destroyed. This is useful to, for instance, let the object complete a final transition.

```
item = Qt.createQmlObject(...);
...
item.destroy();
```

Note: It is possible to destroy an object from within, making it possible to create self-destroying popup windows for instance.

Tracking Dynamic Objects

Working with dynamic objects, it is often necessary to track the created objects. Another common feature is to be able to store and restore the state of the dynamic objects. Both these tasks are easily handled using a `ListModel` that we populate dynamically.

In the example shown below two types of elements, rockets and ufos, can be created and moved around by the user. In order to be able to manipulate the entire scene of dynamically created elements, we use a model to track the items.

Todo

illustration

The model, a `ListModel`, is populated as the items are created. The object reference is tracked along side the source URL used when instantiating it. The latter is not strictly needed for tracking the objects, but will come in handy later.

```
import QtQuick 2.5
import "create-object.js" as CreateObject

Item {
    id: root

    ListModel {
        id: objectsModel
    }

    function addUfo() {
        CreateObject.create("ufo.qml", root, itemAdded);
    }

    function addRocket() {
        CreateObject.create("rocket.qml", root, itemAdded);
    }

    function itemAdded(obj, source) {
        objectsModel.append({"obj": obj, "source": source})
    }
}
```

As you can tell from the example above, the `create-object.js` is a more generalized form of the JavaScript introduced earlier. The `create` method uses three arguments: a source URL, a root element and a callback to invoke when finished. The callback gets called with two arguments: a reference to the newly created object and the source URL used.

This means that each time `addUfo` or `addRocket` functions are called, the `itemAdded` function will be called when the new object has been created. The latter will append the object reference and source URL to the `objectsModel` model.

The `objectsModel` can be used in many ways. In the example in question, the `clearItems` function relies on it. This function demonstrates two things. First, how to iterate over the model and perform a task, i.e. calling the `destroy` function for each item to remove it. Secondly, it highlights the fact that the model is not updated as objects are destroyed. Instead of removing the model item connected to the object in question, the `obj` property

of that model item is set to null. To remedy this, the code explicitly has to clear the model item as the objects are removed.

```
function clearItems() {
    while(objectsModel.count > 0) {
        objectsModel.get(0).obj.destroy();
        objectsModel.remove(0);
    }
}
```

Having a model representing all dynamically created items, it is easy to create a function that serializes the items. In the example code, the serialized information consists of the source URL of each object along its *x* and *y* properties. These are the properties that can be altered by the user. The information is used to build an XML document string.

```
function serialize() {
    var res = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n<scene>\n";

    for(var ii=0; ii < objectsModel.count; ++ii) {
        var i = objectsModel.get(ii);
        res += "  <item>\n    <source>" + i.source + "</source>\n    <x>" + i.
    obj.x + "</x>\n    <y>" + i.obj.y + "</y>\n  </item>\n";

        res += "</scene>";

    return res;
}
```

The XML document string can be used with a *XmlListModel* by setting the *xml* property of the model. In the code below, the model is shown along the *deserialize* function. The *deserialize* function kickstarts the deserialization by setting the *dsIndex* to refer to the first item of the model and then invoking the creation of that item. The callback, *dsItemAdded* then sets that *x* and *y* properties of the newly created object. It then updates the index and creates the nexts object, if any.

```
XmlListModel {
    id: xmlModel
    query: "/scene/item"
    XmlRole { name: "source"; query: "source/string()" }
    XmlRole { name: "x"; query: "x/string()" }
    XmlRole { name: "y"; query: "y/string()" }
}

function deserialize() {
    dsIndex = 0;
    CreateObject.create(xmlModel.get(dsIndex).source, root, dsItemAdded);
}

function dsItemAdded(obj, source) {
    itemAdded(obj, source);
    obj.x = xmlModel.get(dsIndex).x;
    obj.y = xmlModel.get(dsIndex).y;

    dsIndex ++;

    if (dsIndex < xmlModel.count)
        CreateObject.create(xmlModel.get(dsIndex).source, root, dsItemAdded);
}

property int dsIndex
```

The example demonstrates how a model can be used to track created items, and how easy it is to serialize and deserialize such information. This can be used to store a dynamically populated scene such as a set of widgets. In

the example, a model was used to track each item.

An alterante solution would be to use the `children` property of the root of a scene to track items. This, however, requires the items themselves to know the source URL to use to re-create them. It also requires the scene to consist only of dynamically created items, to avoid attempting to serialize and later deserialize any statically allocated objects.

Summary

In this chapter we have looked at creating QML elements dynamically. This lets us create QML scenes freely, opening the door for user configurability and plug-in based architectures.

The easiest way to dynamically load a QML element is to use a `Loader` element. This acts as a placeholder for the contents being loaded.

For a more dynamic approach, the `Qt.createQmlObject` function can be used to instantiate a string of QML. This approach does, however, have limitations. The full blown solution is to dynamically create a `Component` using the `Qt.createComponent` function. Objects are then created by calling the `createObject` function of a `Component`.

As bindings and signal connections rely on the existence of an object `id`, or access to the object instantiation, an alternate approach must be used for dynamically created objects. To create a binding, the `Binding` element is used. The `Connections` element makes it possible to connect to signals of a dynamically created object.

One of the challenges of working with dynamically created items is to keep track of them. This can be done using a `ListModel`. By having a model tracking the dynamically created items, it is possible to implement functions for serialization and deserialization, making it possible to store and restore dynamically created scenes.

JavaScript

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

JavaScript is the lingua-franca on web client development. It also starts to get traction on web server development mainly by node.js. As such it is a well suited addition as an imperative language onto the side of declarative QML language. QML itself as a declarative language is used to express the user interface hierarchy but is limited to express operational code. Sometimes you need a way to express operations, here JavaScript comes into play.

Note: There is an open question in the Qt community about the right mixture about QML/JS/QtC++ in a modern Qt application. The commonly agreed recommended mixture is to limit the JS part of your application to a minimum and do your business logic inside QtC++ and the UI logic inside QML/JS.

This book pushes the boundaries, which is not always the right mix for a product development and not for everyone. It is important to follow your team skills and your personal taste. In doubt follow the recommendation.

Here a short example how JS looks like, mixed in QML:

```
Button {
    width: 200
    height: 300
    property bool checked: false
    text: "Click to toggle"

    // JS function
    function doToggle() {
        checked = !checked
    }

    onTriggered: {
        // this is also JavaScript
        doToggle();
        console.log('checked: ' + checked)
    }
}
```

So JavaScript can come in many places inside QML as a standalone JS function, as a JS module and it can be on every right side of a property binding.

```
import "util.js" as Util // import a pure JS module

Button {
```

```
width: 200
height: width*2 // JS on the right side of property binding

// standalone function (not really useful)
function log(msg) {
    console.log("Button> " + msg);
}

onTriggered: {
    // this is JavaScript
    log();
    Qt.quit();
}
}
```

Within QML you declare the user interface, with JavaScript you make it functional. So how much JavaScript should you write? It depends on your style and how familiar you are with JS development. JS is a loosely typed language, which makes it difficult to spot type defects. Also functions expect all argument variations, which can be a very nasty bug to spot. The way to spot defects is rigorous unit testing or acceptance testing. So if you develop real logic (not some glue lines of code) in JS you should really start using the test-first approach. In general mixed teams (Qt/C++ and QML/JS) are very successful when they minimize the amount of JS in the frontend as the domain logic and do the heavy lifting in Qt C++ in the backend. The backend should then be rigorous unit tested so that the frontend developers can trust the code and focus on all these little user interface requirements.

Note: In general: backend developers are functional driven and frontend developers are user story driven.

Browser/HTML vs QtQuick/QML

The browser is the runtime to render HTML and execute the Javascript associated with the HTML. Nowadays modern web applications contain much more JavaScript than HTML. The Javascript inside the browser is a standard ECMAScript environment with some browser additions. A typical JS environment inside the browser knows the `window` object to access the browser window. There are also the basic DOM selectors which are used by jQuery to provide the CSS selectors. Additionally there is a `setTimeout` function to call a function after a certain time. Besides these the environment is a standard JavaScript environment similar to QML/JS.

What is also different is where JS can appear inside HTML and QML. In HTML you can only add JS on event handlers (e.g. page loaded, mouse pressed). For example your JS initializes normally on page load, which is comparable to `Component.onCompleted` in QML. For example you can not use JS for property bindings (at least not directly, AngularJS enhances the DOM tree to allow these, but this is far away from standard HTML).

So in QML JS is much more a first-class citizen and much deeper integrated into the QML render tree. Which makes the syntax much more readable. Besides this people which have developed HTML/JS applications will feel at home inside QML/JS.

The Language

This chapter will not give you a general introduction to JavaScript. There are other books out there for a general introduction to JavaScript, please visit this great site on [Mozilla Developer Network](#).

On the surface JavaScript is a very common language and does not differ a lot from other languages:

```
function countdown() {
    for(var i=0; i<10; i++) {
        console.log('index: ' + i)
    }
}
```

```
function countdown2() {
    var i=10;
    while( i>0 ) {
        i--;
    }
}
```

But be warned JS has function scope and not block scope as in C++ (see [Functions and function scope](#)).

The statements `if ... else`, `break`, `continue` also work as expected. The `switch` case can also compare other types and not just integer values:

```
function getAge(name) {
    // switch over a string
    switch(name) {
        case "father":
            return 58;
        case "mother":
            return 56;
    }
    return unknown;
}
```

JS knows several values which can be false, e.g. `false`, `0`, `"`, `undefined`, `null`). For example a function returns by default `undefined`. To test for false use the `===` identity operator. The `==` equality operator will do type conversion to test for equality. If possible use the faster and better `===` strict equality operator which will test for identity (see [Comparison operators](#)).

Under the hood javascript has its own ways of doing things. For example arrays:

```
function doIt() {
    var a = [] // empty arrays
    a.push(10) // addend number on arrays
    a.push("Monkey") // append string on arrays
    console.log(a.length) // prints 2
    a[0] // returns 10
    a[1] // returns Monkey
    a[2] // returns undefined
    a[99] = "String" // a valid assignment
    console.log(a.length) // prints 100
    a[98] // contains the value undefined
}
```

Also for people coming from C++ or Java which are used to a OO language JS just works different. JS is not purely an OO language it is a so called prototype based language. Each object has a prototype object. An object is created based on his prototype object. Please read more about this in the book [Javascript the Good Parts](#) by Douglas Crockford or watch the video below.

To test some small JS snippets you can use the online [JS Console](#) or just build a little piece of QML code:

```
import QtQuick 2.5

Item {
    function runJS() {
        console.log("Your JS code goes here");
    }
    Component.onCompleted: {
        runJS();
    }
}
```

JS Objects

While working with JS there are some objects and methods which are more frequently used. This is a small collection of them.

- `Math.floor(v)`, `Math.ceil(v)`, `Math.round(v)` - largest, smallest, rounded integer from float
- `Math.random()` - create a random number between 0 and 1
- `Object.keys(o)` - get keys from object (including QObject)
- `JSON.parse(s)`, `JSON.stringify(o)` - conversion between JS object and JSON string
- `Number.toFixed(p)` - fixed precision float
- `Date` - Date manipulation

You can find them also at: [JavaScript reference](#)

Here some small and limited examples how to use JS with QML. They should give you an idea how you can use JS inside QML

Print all keys from QML Item

```
Item {
    id: root
    Component.onCompleted: {
        var keys = Object.keys(root);
        for(var i=0; i<keys.length; i++) {
            var key = keys[i];
            // prints all properties, signals, functions from object
            console.log(key + ' : ' + root[key]);
        }
    }
}
```

Parse an object to a JSON string and back

```
Item {
    property var obj: {
        key: 'value'
    }

    Component.onCompleted: {
        var data = JSON.stringify(obj);
        console.log(data);
        var obj = JSON.parse(data);
        console.log(obj.key); // > 'value'
    }
}
```

Current Date

```
Item {
    Timer {
        id: timeUpdater
        interval: 100
        running: true
        repeat: true
    }
}
```

```

    onTriggered: {
        var d = new Date();
        console.log(d.getSeconds());
    }
}
}

```

Call a function by name

```

Item {
    id: root

    function doIt() {
        console.log("doIt()")
    }

    Component.onCompleted: {
        // Call using function execution
        root["doIt"]();
        var fn = root["doIt"];
        // Call using JS call method (could pass in a custom this object and arguments)
        fn.call()
    }
}

```

Creating a JS Console

As a little example we will create a JS console. We need an input field where the user can enter his JS expressions and ideally there should be a list of output results. As this should more look like a desktop application we use the QtQuick Controls module.

Note: A JS console inside your next project can be really beneficial for testing. Enhanced with a Quake-Terminal effect it is also good to impress customers. To use it wisely you need to control the scope the JS console evaluates in, e.g. the current visible screen, the main data model, a singleton core object or all together.

We use Qt Creator to create a Qt Quick UI project using QtQuick controls. We call the project *JSConsole*. After the wizard has finished we have already a basic structure for the application with an application window and a menu to exit the application.

For the input we use a TextField and a Button to send the input for evaluation. The result of the expression evaluation is displayed using a ListView with a ListModel as the model and two labels to display the expression and the evaluated result.

```

// part of JSConsole.qml
ApplicationWindow {
    id: root

    ...

    ColumnLayout {
        anchors.fill: parent
        anchors.margins: 9
        RowLayout {
            Layout.fillWidth: true
            TextField {
                id: input
            }
        }
    }
}

```



```

        Layout.fillWidth: true
        focus: true
        onAccepted: {
            // call our evaluation function on root
            root.jsCall(input.text)
        }
    }
    Button {
        text: qsTr("Send")
        onClicked: {
            // call our evaluation function on root
            root.jsCall(input.text)
        }
    }
}
Item {
    Layout.fillWidth: true
    Layout.fillHeight: true
    Rectangle {
        anchors.fill: parent
        color: '#333'
        border.color: Qt.darker(color)
        opacity: 0.2
        radius: 2
    }

    ScrollView {
        id: scrollView
        anchors.fill: parent
        anchors.margins: 9
        ListView {
            id: resultView
            model: ListModel {
                id: outputModel
            }
            delegate: ColumnLayout {
                width: ListView.view.width
                Label {
                    Layout.fillWidth: true
                    color: 'green'
                    text: "> " + model.expression
                }
                Label {
                    Layout.fillWidth: true
                    color: 'blue'
                    text: "" + model.result
                }
                Rectangle {
                    height: 1
                    Layout.fillWidth: true
                    color: '#333'
                    opacity: 0.2
                }
            }
        }
    }
}
}
}
}

```

The evaluation function `jsCall` does the evaluation not by itself this has been moved to a JS module (`jsconsole.js`) for clearer separation.

```
// part of JSConsole.qml

import "jsconsole.js" as Util

...

ApplicationWindow {
    id: root

    ...

    function jsCall(exp) {
        var data = Util.call(exp);
        // insert the result at the beginning of the list
        outputModel.insert(0, data)
    }
}
```

For safety we do not use the `eval` function from JS as this would allow the user to modify the local scope. We use the `Function` constructor to create a JS function on runtime and pass in our scope as this variable. As the function is created every time it does not act as a closure and stores its own scope, we need to use `this.a = 10` to store the value inside the `this` scope of the function. The `this` scope is set by the script to the scope variable.

```
// jsconsole.js
.pragma library

var scope = {
    // our custom scope injected into our function evaluation
}

function call(msg) {
    var exp = msg.toString();
    console.log(exp)
    var data = {
        expression : msg
    }
    try {
        var fun = new Function('return (' + exp + ');');
        data.result = JSON.stringify(fun.call(scope), null, 2)
        console.log('scope: ' + JSON.stringify(scope, null, 2) + 'result: ' +
→result)
    } catch(e) {
        console.log(e.toString())
        data.error = e.toString();
    }
    return data;
}
```

The data return from the `call` function is a JS object with a `result`, `expression` and `error` property: `data: { expression: {}, result: {}, error: {} }`. We can use this JS object directly inside the `List-Model` and access it then from the delegate, e.g. `model.expression` gives us the input expression. For the simplicity of the example we ignore the error result.

Qt and C++

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

Qt is a C++ toolkit with an extension for QML and Javascript. There exists many language bindings for Qt, but as Qt is developed in C++, the spirit of C++ can be found throughout the classes. In this section, we will look at Qt from a C++ perspective to build a better understanding how to extend QML with native plugins developed using C++. Through C++, it is possible to extend and control the execution environment provided to QML.

This chapter will, just as Qt, require the reader to have some basic knowledge of C++. Qt does not rely on advanced C++ features, and I generally consider the Qt style of C++ to be very readable, so do not worry if you feel that your C++ knowledge is shaky.

Approaching Qt from a C++ direction, you will find that Qt enriches C++ with a number of modern language features enabled through making introspection data available. This is made possible through the use of the `QObject` base class. Introspection data, or meta data, maintains information of the classes at run-time, something that ordinary C++ does not do. This makes it possible to dynamically probe objects for information about such details as their properties and available methods.

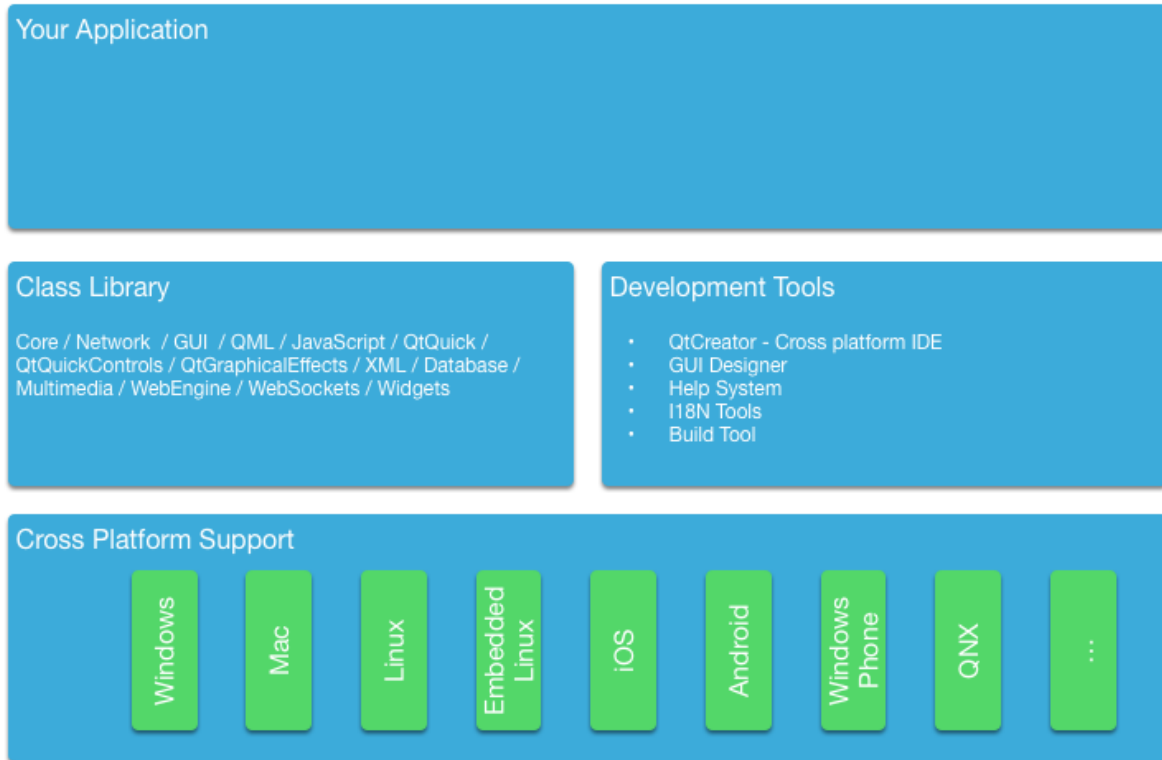
Qt uses this meta information to enable a very loosely bound callback concept using signals and slots. Each signal can be connected to any number of slots or even other signals. When a signal is emitted from an object instance, the connected slots are invoked. As the signal emitting object does not need to know anything about the object owning the slot and vice versa, this mechanism is used to create very reusable components with very few inter-component dependencies.

The introspection features are also used to create dynamic language bindings, making it possible to expose a C++ object instance to QML and making C++ functions callable from Javascript. Other bindings for Qt C++ exist and besides the standard Javascript binding a popular one is the Python binding called `PyQt`.

In addition to this central concept, Qt makes it possible to develop cross platform applications using C++. Qt C++ provides a platform abstraction on the different operating systems, which allows the developer to concentrate on the task at hand and not the details of how you open a file on different operating systems. This means you can re-compile the same source code for Windows, OS X and Linux and Qt takes care of the different OS ways of handling certain things. The end result are natively built applications with the look and feel of the target platform. As the mobile is the new desktop, newer Qt versions can also target a number of mobile platforms using the same source code, e.g. iOS, Android, Jolla, BlackBerry, Ubuntu Phone, Tizen.

When it comes to re-use, not only can source code be re-used but developer skills are also reusable. A team knowing Qt can reach out to far more platforms than a team just focusing on a single platform specific technology and as Qt is so flexible the team can create different system components using the same technology.

For all platform, Qt offers a set of basic types, e.g. strings with full unicode support, lists, vectors, buffers. It also provides a common abstraction to the target platform's main loop, and cross platform threading and networking support. The general philosophy is that for an application developer Qt comes with all required functionality



included. For domain specific tasks such as to interface to your native libraries Qt comes with several helper classes to make this easier.

A Boilerplate Application

The best way to understand Qt is to start from a small demonstration application. This application creates a simple "Hello World!" string and writes it into a file using unicode characters.

```
#include <QCoreApplication>
#include <QString>
#include <QFile>
#include <QDir>
#include <QTextStream>
#include <QDebug>

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

    // prepare the message
    QString message("Hello World!");

    // prepare a file in the users home directory named out.txt
    QFile file(QDir::home().absoluteFilePath("out.txt"));
    // try to open the file in write mode
    if(!file.open(QIODevice::WriteOnly)) {
        qWarning() << "Can not open file with write access";
        return -1;
    }
    // as we handle text we need to use proper text codecs
    QTextStream stream(&file);
```

```

// write message to file via the text stream
stream << message;

// do not start the eventloop as this would wait for external IO
// app.exec();

// no need to close file, closes automatically when scope ends
return 0;
}

```

The simple example demonstrates the use of file access and the correct way of writing text into a file using text codecs via the text stream. For binary data there is a cross platform binary stream called `QDataStream`. The different classes we use are included using their class name. Another possibility would be to use a module and class name e.g. `#include <QtCore/QFile>`. For the lazy there is also the possibility to include a whole module using `#include <QtCore>`. E.g. in `QtCore` you have the most common classes used for an application, which are not UI dependent. Have a look at the [QtCore class list](#) or the [QtCore overview](#).

You build the application using `qmake` and `make`. `QMake` reads a project file and generates a `Makefile` which then can be called using `make`. The project file is platform independent and `qmake` has some rules to apply the platform specific settings to the generated make file. The project can also contain platform scopes for platform specific rules, which are required in some specific cases. Here is an example of a simple project file.

```

# build an application
TEMPLATE = app

# use the core module and do not use the gui module
QT      += core
QT      -= gui

# name of the executable
TARGET = CoreApp

# allow console output
CONFIG += console

# for mac remove the application bundling
macx {
    CONFIG -= app_bundle
}

# sources to be build
SOURCES += main.cpp

```

We will not go into depth into this topic. Just remember Qt uses project files for projects and `qmake` generates the platform specific make files from these project files.

The simple code example above just writes the text and exits the application. For a command line tool this is good enough. For a user interface you would need an event loop which waits for user input and somehow schedules re-draw operations. So here follows the same example now uses a desktop button to trigger the writing.

Our `main.cpp` surprisingly got smaller. We moved code into an own class to be able to use signal/slots for the user input, e.g. the button click. The signal/slot mechanism normally needs an object instance as you will see shortly.

```

#include <QtCore>
#include <QtGui>
#include <QtWidgets>
#include "mainwindow.h"

int main(int argc, char** argv)
{

```

```
QApplication app(argc, argv);

MainWindow win;
win.resize(320, 240);
win.setVisible(true);

return app.exec();
}
```

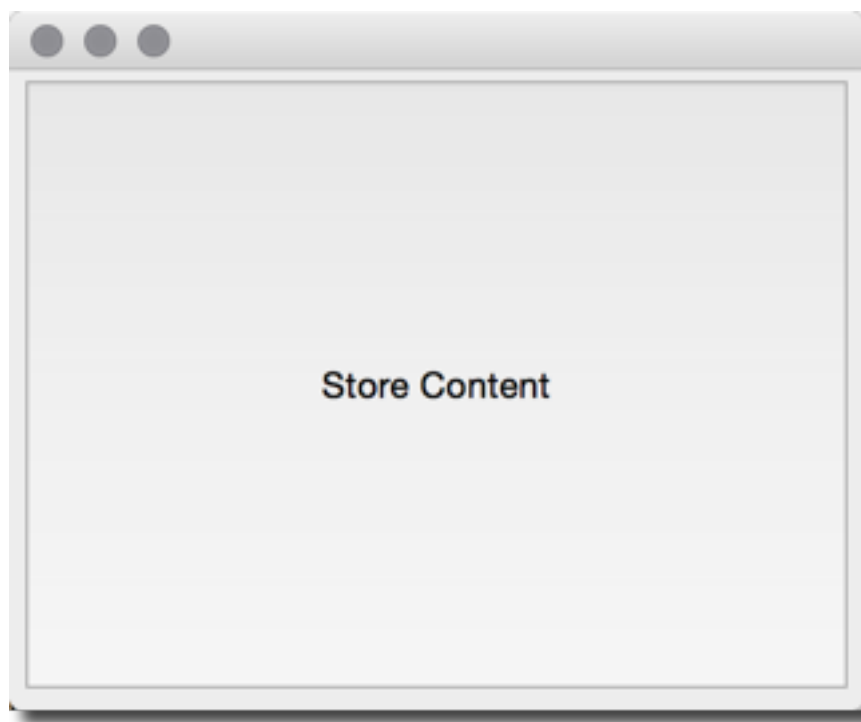
In the main function we simply create the application object and start the event loop using `exec()`. For now the application sits in the event loop and waits for user input.

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv); // init application

    // create the ui

    return app.exec(); // execute event loop
}
```

Qt offers several UI technologies. For this example we use the Desktop Widgets user interface library using pure Qt C++. We create a main window which will host a push button to trigger the functionality and also the main window will host our core functionality which we know from the previous example.



The main window itself is a widget. It becomes a top level window as it does not have any parent. This comes from how Qt sees a user interface as a tree of ui elements. In this case the main window is the root element, thus becomes a window, while the push button a child of the main window and becomes a widget inside the window.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtWidgets>

class MainWindow : public QMainWindow
{

```

```

public:
    MainWindow(QWidget* parent=0);
    ~MainWindow();
public slots:
    void storeContent();
private:
    QPushButton *m_button;
};

#endif // MAINWINDOW_H

```

Additionally we define a public slot called `storeContent()` which shall be called when the button is clicked. A slot is a C++ method which is registered with the Qt meta object system and can be dynamically called.

```

#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    m_button = new QPushButton("Store Content", this);

    setCentralWidget(m_button);
    connect(m_button, &QPushButton::clicked, this, &MainWindow::storeContent);
}

MainWindow::~MainWindow()
{
}

void MainWindow::storeContent()
{
    qDebug() << "... store content";
    QString message("Hello World!");
    QFile file(QDir::home().absoluteFilePath("out.txt"));
    if(!file.open(QIODevice::WriteOnly)) {
        qWarning() << "Can not open file with write access";
        return;
    }
    QTextStream stream(&file);
    stream << message;
}

```

In the main window we first create the push button and then register the signal `clicked()` with the slot `storeContent()` using the `connect` method. Every time the signal `clicked` is emitted the slot `storeContent()` is called. As simple as this, objects communicate via signal and slots with loose coupling.

The QObject

As described in the introduction, the `QObject` is what enables Qt's introspection. It is the base class of almost all classes in Qt. Exceptions are value types such as `QColor`, `QString` and `QList`.

A Qt object is a standard C++ object, but with more abilities. These can be divided into two groups: introspection and memory management. The first means that a Qt object is aware of its class name, its relationship to other classes, as well as its methods and properties. The memory management concept means that each Qt object can be the parent of child objects. The parent *owns* the children, and when the parent is destroyed, it is responsible for destroying its children.

The best way of understanding how the `QObject` abilities affect a class is to take a standard C++ class and Qt enable it. The class shown below represents an ordinary such class.

The person class is a data class with a name and gender properties. The person class uses Qt's object system to add meta information to the c++ class. It allows users of a person object to connect to the slots and get notified when the properties get changed.

```
class Person : public QObject
{
    Q_OBJECT // enabled meta object abilities

    // property declarations required for QML
    Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
    Q_PROPERTY(Gender gender READ gender WRITE setGender NOTIFY genderChanged)

    // enables enum introspections
    Q_ENUMS (Gender)

public:
    // standard Qt constructor with parent for memory management
    Person(QObject *parent = 0);

    enum Gender { Unknown, Male, Female, Other };

    QString name() const;
    Gender gender() const;

public slots: // slots can be connected to signals
    void setName(const QString &);
    void setGender(Gender);

signals: // signals can be emitted
    void nameChanged(const QString &name);
    void genderChanged(Gender gender);

private:
    // data members
    QString m_name;
    Gender m_gender;
};
```

The constructor passes the parent to the super class and initialize the members. Qt's value classes are automatically initialized. In this case `QString` will initialize to a null string (`QString::isNull()`) and the gender member will explicitly initialize to the unknown gender.

```
Person::Person(QObject *parent)
    : QObject(parent)
    , m_gender(Person::Unknown)
{
}
```

The getter function is named after the property and is normally a simple `const` function. The setter emits the changed signal when the property really has changed. For this we insert a guard to compare the current value with the new value. And only when the value differs we assign it to the member variable and emit the changed signal.

```
QString Person::name() const
{
    return m_name;
}

void Person::setName(const QString &name)
{
    if (m_name != name) // guard
    {
        m_name = name;
        emit nameChanged(m_name);
    }
}
```

```
}
}
```

Having a class derived from `QObject`, we have gained more meta object abilities we can explore using the `metaObject()` method. For example retrieving the class name from the object.

```
Person* person = new Person();
person->metaObject()->className(); // "Person"
Person::staticMetaObject.className(); // "Person"
```

There are many more features which can be accessed by the `QObject` base class and the meta object. Please check out the `QMetaObject` documentation.

Build Systems

Building software reliably on different platforms can be a complex task. You will encounter different environments with different compilers, paths, and library variations. The purpose of Qt is to shield the application developer from these cross platform issues. For this Qt introduced the `qmake` build file generator. `qmake` operates on a project file with the ending `.pro`. This project file contains instructions about the application and the sources to be used. Running `qmake` on this project file will generate a `Makefile` for you on unix and mac and even under windows if the mingw compiler toolchain is being used. Otherwise it may create a visual studio project or an xcode project.

A typical build flow in Qt under unix would be:

```
$ edit myproject.pro
$ qmake // generates Makefile
$ make
```

Qt allows you also to use shadow builds. A shadow build is a build outside of your source code location. Assume we have a `myproject` folder with a `myproject.pro` file. The flow would be like this:

```
$ mkdir build
$ cd build
$ qmake ../myproject/myproject.pro
```

We create a build folder and then call `qmake` from inside the build folder with the location of our project folder. This will setup the make file in a way that all build artifacts are stored under the build folder instead of inside our source code folder. This allows us to create builds for different qt versions and build configurations at the same time and also it does not clutter our source code folder which is always a good thing.

When you are using Qt Creator it does these things behind the scenes for you and you do not have to worry about these steps usually. For larger projects and for a deeper understanding of the flow, it is recommended that you learn to build your qt project from the command line.

QMake

QMake is the tool which reads your project file and generates the build file. A project file is a simplified write down of your project configuration, external dependencies, and your source files. The simplest project file is probably this:

```
// myproject.pro

SOURCES += main.cpp
```

Here we build an executable application which will have the name `myproject` based on the project file name. The build will only contain the `main.cpp` source file. And by default we will use the `QtCore` and `QtGui` module

for this project. If our project were a QML application we would need to add the QtQuick and QtQml module to the list:

```
// myproject.pro

QT += qml quick

SOURCES += main.cpp
```

Now the build file knows to link against the QtQml and QtQuick Qt modules. QMake use the concept of =, += and -= to assign, add, remove elements from a list of options, respectively. For a pure console build without UI dependencies you would remove the QtGui module:

```
// myproject.pro

QT -= gui

SOURCES += main.cpp
```

When you want to build a library instead of an application, you need to change the build template:

```
// myproject.pro

TEMPLATE = lib

QT -= gui

HEADERS += utils.h
SOURCES += utils.cpp
```

Now the project will build as a library without UI dependencies and used the `utils.h` header and the `utils.cpp` source file. The format of the library will depend on the OS you are building the project.

Often you will have more complicated setups and need to build a set of projects. For this, qmake offers the `subdirs` template. Assume we would have a `mylib` and a `myapp` project. Then our setup could be like this:

```
my.pro
mylib/mylib.pro
mylib/utils.h
mylib/utils.cpp
myapp/myapp.pro
myapp/main.cpp
```

We know already how the `mylib.pro` and `myapp.pro` would look like. The `my.pro` as the overarching project file would look like this:

```
// my.pro

TEMPLATE = subdirs

subdirs = mylib \
    myapp

myapp.depends = mylib
```

This declares a project with two subprojects: `mylib` and `myapp`, where `myapp` depends on `mylib`. When you run qmake on this project file it will generate file a build file for each project in a corresponding folder. When you run the make file for `my.pro`, all subprojects are also built.

Sometimes you need to do one thing on one platform and another thing on other platforms based on your configuration. For this qmake introduces the concept of scopes. A scope is applied when a configuration option is set to true.

For example to use a unix specific utils implementation you could use:


```
unix {
    SOURCES += utils_unix.cpp
} else {
    SOURCES += utils.cpp
}
```

What it says is if the CONFIG variable contains a unix option then apply this scope otherwise use the else path. A typical one is to remove the application bundling under mac:

```
macx {
    CONFIG -= app_bundle
}
```

This will create your application as a plain executable under mac and not as a .app folder which is used for application installation.

QMake based projects are normally the number one choice when you start programming Qt applications. There are also other options out there. All have their benefits and drawbacks. We will shortly discuss these other options next.

References

- [QMake Manual](#) - Table of contents of the qmake manual
- [QMake Language](#) - Value assignment, scopes and so like
- [QMake Variables](#) - Variables like TEMPLATE, CONFIG, QT are explained here

CMake

CMake is a tool create by Kitware. Kitware is very well known for their 3D visualitation software VTK and also CMake, the cross platform makefile generator. It uses a series of CMakeLists.txt files to generate platform specific make files. CMake is used by the KDE project and as such has a special relationship with the Qt community.

The CMakeLists.txt is the file used to store the project configuration. For a simple hello world using QtCore the project file would look like this:

```
// ensure cmake version is at least 3.0
cmake_minimum_required(VERSION 3.0)
// adds the source and build location to the include path
set(CMAKE_INCLUDE_CURRENT_DIR ON)
// Qt's MOC tool shall be automatically invoked
set(CMAKE_AUTOMOC ON)
// using the Qt5Core module
find_package(Qt5Core)
// create excutable helloworld using main.cpp
add_executable(helloworld main.cpp)
// helloworld links against Qt5Core
target_link_libraries(helloworld Qt5::Core)
```

This will build a helloworld executable using main.cpp and linked agains the external Qt5Core library. The build file can be modified to be more generic:

```
// sets the PROJECT_NAME variable
project(helloworld)
cmake_minimum_required(VERSION 3.0)
set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_AUTOMOC ON)
find_package(Qt5Core)
```

```
// creates a SRC_LIST variable with main.cpp as single entry
set(SRC_LIST main.cpp)
// add an executable based on the project name and source list
add_executable(${PROJECT_NAME} ${SRC_LIST})
// links Qt5Core to the project executable
target_link_libraries(${PROJECT_NAME} Qt5::Core)
```

You can see that CMake is quite powerful. It takes some time to get used to the syntax. In general, it is said that CMake is better suited for large and complex projects.

References

- [CMake Help](#) - available online but also as QtHelp format
- [Running CMake](#)
- [KDE CMake Tutorial](#)
- [CMake Book](#)
- [CMake and Qt](#)

Common Qt Classes

The `QObject` class forms the foundations of Qt, but there are many more classes in the framework. Before we continue looking at QML and how to extend it, we will look at some basic Qt classes that are useful to know about.

The code examples shown in this section are written using the Qt Test library. It offers a great way to explore the Qt API and store it for later reference. `QVERIFY`, `QCOMPARE` are functions provided by the test library to assert a certain condition. We will use `{ }` scopes to avoid name collisions. So do not get confused.

QString

In general, text handling in Qt is unicode based. For this you use the `QString` class. It comes with a variety of great functions which you would expect from a modern framework. For 8-bit data you would use normally the `QByteArray` class and for ASCII identifiers the `QLatin1String` to preserve memory. For a list of strings you can use a `QList<QString>` or simply the `QStringList` class (which is derived from `QList<QString>`).

Here are some examples of how to use the `QString` class. `QString` can be created on the stack but it stores its data on the heap. Also when assigning one string to another, the data will not be copied - only a reference to the data. So this is really cheap and lets the developer concentrate on the code and not on the memory handling. `QString` uses reference counters to know when the data can be safely deleted. This feature is called [Implicit Sharing](#) and it is used in many Qt classes.

```
QString data("A,B,C,D"); // create a simple string
// split it into parts
QStringList list = data.split(",");
// create a new string out of the parts
QString out = list.join(",");
// verify both are the same
QVERIFY(data == out);
// change the first character to upper case
QVERIFY(QString("A") == out[0].toUpper());
```

Here we will show how to convert a number to a string and back. There are also conversion functions for float or double and other types. Just look for the function in the Qt documentation used here and you will find the others.

```
// create some variables
int v = 10;
int base = 10;
// convert an int to a string
QString a = QString::number(v, base);
// and back using and sets ok to true on success
bool ok(false);
int v2 = a.toInt(&ok, base);
// verify our results
QVERIFY(ok == true);
QVERIFY(v == v2);
```

Often in text you need to have parameterized text. One option could be to use `QString("Hello" + name)` but a more flexible method is the `arg` marker approach. It preserves the order also during translation when the order might change.

```
// create a name
QString name("Joe");
// get the day of the week as string
QString weekday = QDate::currentDate().toString("dddd");
// format a text using paramters (%1, %2)
QString hello = QString("Hello %1. Today is %2.").arg(name).arg(weekday);
// This worked on Monday. Promise!
if(QDate::currentDate().dayOfWeek() == Qt::Monday) {
    QCOMPARE(QString("Hello Joe. Today is Monday."), hello);
} else {
    QVERIFY(QString("Hello Joe. Today is Monday.") != hello);
}
```

Sometimes you want to use unicode characters directly in you code. For this you need to remember how to mark them for the `QChar` and `QString` classes.

```
// Create a unicode character using the unicode for smile :-)
QChar smile(0x263A);
// you should see a :-) on you console
QDebug() << smile;
// Use a unicode in a string
QChar smile2 = QString("\u263A").at(0);
QVERIFY(smile == smile2);
// Create 12 smiles in a vector
QVector<QChar> smilies(12);
smilies.fill(smile);
// Can you see the smiles
QDebug() << smilies;
```

This gives you some examples of how to easily treat unicode aware text in Qt. For non-unicode the `QByteArray` class also has many helper functions for conversion. Please read the Qt documentation for `QString` as it contains tons of good examples.

Sequential Containers

A list, queue, vector or linked-list is a sequential container. The mostly used sequential container is the `QList` class. It is a template based class and needs to be initialized with a type. It is also implicit shared and stores the data internally on the heap. All container classes should be created on the stack. Normally you never want to use `new QList<T>()`, which means never use `new` with a container.

The `QList` is as versatile as the `QString` class and offers a great API to explore your data. Below is a small example how to use and iterate over a list using some new C++ 11 features.

```
// Create a simple list of ints using the new C++11 initialization
// for this you need to add "CONFIG += c++11" to your pro file.
```

```

QList<int> list{1,2};

// append another int
list << 3;

// We are using scopes to avoid variable name clashes

{ // iterate through list using Qt for each
    int sum(0);
    foreach (int v, list) {
        sum += v;
    }
    QVERIFY(sum == 6);
}

{ // iterate through list using C++ 11 range based loop
    int sum = 0;
    for(int v : list) {
        sum += v;
    }
    QVERIFY(sum == 6);
}

{ // iterate through list using JAVA style iterators
    int sum = 0;
    QListIterator<int> i(list);

    while (i.hasNext()) {
        sum += i.next();
    }
    QVERIFY(sum == 6);
}

{ // iterate through list using STL style iterator
    int sum = 0;
    QList<int>::iterator i;
    for (i = list.begin(); i != list.end(); ++i) {
        sum += *i;
    }
    QVERIFY(sum == 6);
}

// using std::sort with mutable iterator using C++11
// list will be sorted in descending order
std::sort(list.begin(), list.end(), [](int a, int b) { return a > b; });
QVERIFY(list == QList<int>({3,2,1}));

int value = 3;
{ // using std::find with const iterator
    QList<int>::const_iterator result = std::find(list.constBegin(), list.
→constEnd(), value);
    QVERIFY(*result == value);
}

{ // using std::find using C++ lambda and C++ 11 auto variable
    auto result = std::find_if(list.constBegin(), list.constEnd(),
→[value](int v) { return v == value; });
    QVERIFY(*result == value);
}

```

Associative Containers

A map, a dictionary, or a set are examples of associative containers. They store a value using a key. They are known for their fast lookup. We demonstrate the use of the most used associative container the `QHash` also demonstrating some new C++ 11 features.

```
QHash<QString, int> hash({{"b", 2}, {"c", 3}, {"a", 1}});
QDebug() << hash.keys(); // a,b,c - unordered
QDebug() << hash.values(); // 1,2,3 - unordered but same as order as keys

QVERIFY(hash["a"] == 1);
QVERIFY(hash.value("a") == 1);
QVERIFY(hash.contains("c") == true);

{ // JAVA iterator
    int sum = 0;
    QHashIterator<QString, int> i(hash);
    while (i.hasNext()) {
        i.next();
        sum += i.value();
        qDebug() << i.key() << " = " << i.value();
    }
    VERIFY(sum == 6);
}

{ // STL iterator
    int sum = 0;
    QHash<QString, int>::const_iterator i = hash.constBegin();
    while (i != hash.constEnd()) {
        sum += i.value();
        qDebug() << i.key() << " = " << i.value();
        i++;
    }
    VERIFY(sum == 6);
}

hash.insert("d", 4);
QVERIFY(hash.contains("d") == true);
hash.remove("d");
QVERIFY(hash.contains("d") == false);

{ // hash find not successfull
    QHash<QString, int>::const_iterator i = hash.find("e");
    VERIFY(i == hash.end());
}

{ // hash find successfull
    QHash<QString, int>::const_iterator i = hash.find("c");
    while (i != hash.end()) {
        qDebug() << i.value() << " = " << i.key();
        i++;
    }
}

// QMap
QMap<QString, int> map({{"b", 2}, {"c", 2}, {"a", 1}});
QDebug() << map.keys(); // a,b,c - ordered ascending

QVERIFY(map["a"] == 1);
QVERIFY(map.value("a") == 1);
QVERIFY(map.contains("c") == true);

// JAVA and STL iterator work same as QHash
```

File IO

It is often required to read and write from files. `QFile` is actually a `QObject` but in most cases it is created on the stack. `QFile` contains signals to inform the user when data can be read. This allows reading chunks of data asynchronously until the whole file is read. For convenience it also allows reading data in blocking mode. This should only be used for smaller amounts of data and not large files. Luckily we only use small amounts of data in these examples.

Besides reading raw data from a file into a `QByteArray` you can also read data types using the `QDataStream` and unicode string using the `QTextStream`. We will show you how.

```
QStringList data({"a", "b", "c"});
{ // write binary files
    QFile file("out.bin");
    if(file.open(QIODevice::WriteOnly)) {
        QDataStream stream(&file);
        stream << data;
    }
}
{ // read binary file
    QFile file("out.bin");
    if(file.open(QIODevice::ReadOnly)) {
        QDataStream stream(&file);
        QStringList data2;
        stream >> data2;
        QCOMPARE(data, data2);
    }
}
{ // write text file
    QFile file("out.txt");
    if(file.open(QIODevice::WriteOnly)) {
        QTextStream stream(&file);
        QString sdata = data.join(",");
        stream << sdata;
    }
}
{ // read text file
    QFile file("out.txt");
    if(file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        QStringList data2;
        QString sdata;
        stream >> sdata;
        data2 = sdata.split(",");
        QCOMPARE(data, data2);
    }
}
```

More Classes

Qt is a rich application framework. As such it has thousands of classes. It takes some time to get used to all of these classes and how to use them. Luckily Qt has a very good documentation with many useful examples includes. Most of the time you search for a class and the most common use cases are already provided as snippets. Which means you just copy and adapt these snippets. Also Qt's examples in the Qt source code are a great help. Make sure you have them available and searchable to make your life more productive. Do not waste time. The Qt community is always helpful. When you ask, it is very helpful to ask exact questions and provide a simple example which displays your needs. This will drastically improve the response time of others. So invest a little bit of time to make the life of others who want to help you easier :-).

Here some classes whose documentation the author thinks are a must read: `QObject`, `QString`, `QByteArray`, `QFile`, `QDir`, `QFileInfo`, `QIODevice`, `QTextStream`, `QDataStream`, `QDebug`, `QLoggingCategory`, `QTcpServer`, `QTcp-`

Socket, QNetworkRequest, QNetworkReply, QAbstractItemModel, QRegExp, QList, QHash, QThread, QProcess, QJsonDocument, QJsonValue.

That should be enough for the beginning.

Models in C++

Models in QML serve the purpose of providing data to `ListViews`, `PathViews` and other views which take a model and create an instance of a delegate for each entry in the model. The view is smart enough to only create these instances which are visible or in the cache range. This makes it possible to have large models with tens of thousands of entries but still have a very slick user interface. The delegate acts like a template to be rendered with the model entries data. So in summary: a view renders entries from the model using a delegate as a template. The model is a data provider to views.

When you do not want to use C++ you can also define models in pure QML. You have several ways to provide a model to the view. For handling of data coming from C++ or large amount of data the C++ model is more suitable than these pure QML approaches. But often you only need a few entries then these QML models are well suited.

```
ListView {
    // using a integer as model
    model: 5
    delegate: Text { text: 'index: ' + index }
}

ListView {
    // using a JS array as model
    model: ['A', 'B', 'C', 'D', 'E']
    delegate: Text { 'Char[' + index + ']: ' + modelData }
}

ListView {
    // using a dynamic QML ListModel as model
    model: ListModel {
        ListElement { char: 'A' }
        ListElement { char: 'B' }
        ListElement { char: 'C' }
        ListElement { char: 'D' }
        ListElement { char: 'E' }
    }
    delegate: Text { 'Char[' + index + ']: ' + model.char }
}
```

The QML views knows how to handle these different models. For models coming from the C++ world the view expects a specific protocol to be followed. This protocol is defined in an API (`QAbstractItemModel`) together with documentation for the dynamic behavior. The API was developed for the desktop widget world and is flexible enough to act as a base for trees, or multi column tables as well as lists. In QML, we almost only use the list version of the API (`QAbstractListModel`). The API contains some mandatory functions to be implemented and some are optional. The optional parts mostly handle the dynamic use case of adding or removing of data.

A simple model

A typical QML C++ model derives from `QAbstractListModel` and implements at least the `data` and `rowCount` function. In this example we will use a series of SVG color names provided by the `QColor` class and display them using our model. The data is stored into a `QList<QString>` data container.

Our `DataEntryModel` is derived from `QAbstractListModel` and implements the mandatory functions. We can ignore the parent in `rowCount` as this is only used in a tree model. The `QModelIndex` class provides the row and column information for the cell, for which the view wants to retrieve data. The view is pulling

information from the model on a row/column and role base. The `QAbstractListModel` is defined in `QtCore` but `QColor` in `QtGui`. That is why we have the additional `QtGui` dependency. For QML applications it is okay to depend on `QtGui` but it should normally not depend on `QtWidgets`.

```
#ifndef DATAENTRYMODEL_H
#define DATAENTRYMODEL_H

#include <QtCore>
#include <QtGui>

class DataEntryModel : public QAbstractListModel
{
    Q_OBJECT
public:
    explicit DataEntryModel(QObject *parent = 0);
    ~DataEntryModel();

public: // QAbstractItemModel interface
    virtual int rowCount(const QModelIndex &parent) const;
    virtual QVariant data(const QModelIndex &index, int role) const;
private:
    QList<QString> m_data;
};

#endif // DATAENTRYMODEL_H
```

On the implementation side the most complex part is the data function. We first need to make a range check. And then we check for the display role. The `Qt::DisplayRole` is the default text role a view will ask for. There is a small set of default roles defined in Qt which can be used, but normally a model will define its own roles for clarity. All calls which do not contain the display role are ignored at the moment and the default value `QVariant()` is returned.

```
#include "dataentrymodel.h"

DataEntryModel::DataEntryModel(QObject *parent)
    : QAbstractListModel(parent)
{
    // initialize our data (QList<QString>) with a list of color names
    m_data = QColor::colorNames();
}

DataEntryModel::~DataEntryModel()
{
}

int DataEntryModel::rowCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    // return our data count
    return m_data.count();
}

QVariant DataEntryModel::data(const QModelIndex &index, int role) const
{
    // the index returns the requested row and column information.
    // we ignore the column and only use the row information
    int row = index.row();

    // boundary check for the row
    if(row < 0 || row >= m_data.count()) {
        return QVariant();
    }
}
```



```

// A model can return data for different roles.
// The default role is the display role.
// it can be accesses in QML with "model.display"
switch(role) {
    case Qt::DisplayRole:
        // Return the color name for the particular row
        // Qt automatically converts it to the QVariant type
        return m_data.value(row);
}

// The view asked for other data, just return an empty QVariant
return QVariant();
}

```

The next step would be to register the model with QML using the `qmlRegisterType` call. This is done inside the `main.cpp` before the QML file was loaded.

```

#include <QtGui>
#include <QtQml>

#include "dataentrymodel.h"

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

    // register the type DataEntryModel
    // under the url "org.example" in version 1.0
    // under the name "DataEntryModel"
    qmlRegisterType<DataEntryModel>("org.example", 1, 0, "DataEntryModel");

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}

```

Now you can access the `DataEntryModel` using the QML import statement `import org.example 1.0` and use it just like other QML item `DataEntryModel {}`.

We use this in this example to display a simple list of color entries.

```

import org.example 1.0

ListView {
    id: view
    anchors.fill: parent
    model: DataEntryModel {}
    delegate: ListDelegate {
        // use the defined model role "display"
        text: model.display
    }
    highlight: ListHighlight { }
}

```

The `ListDelegate` is a custom type to display some text. The `ListHighlight` is just a rectangle. The code was extracted to keep the example compact.

The view can now display a list of strings using the C++ model and the `display` property of the model. It is still very simple, but already usable in QML. Normally the data is provided from outside the model and the model would act as an interface to the view.

More Complex Data

In reality the model data is often much more complex. So there is a need to define custom roles so that the view can query other data via properties. For example the model could provide not only the color as hex string, but maybe also the hue, saturation and brightness from the HSV color model as “model.hue”, “model.saturation” and “model.brightness” in QML.

```
#ifndef ROLEENTRYMODEL_H
#define ROLEENTRYMODEL_H

#include <QtCore>
#include <QtGui>

class RoleEntryModel : public QAbstractListModel
{
    Q_OBJECT
public:
    // Define the role names to be used
    enum RoleNames {
        NameRole = Qt::UserRole,
        HueRole = Qt::UserRole+2,
        SaturationRole = Qt::UserRole+3,
        BrightnessRole = Qt::UserRole+4
    };

    explicit RoleEntryModel(QObject *parent = 0);
    ~RoleEntryModel();

    // QAbstractItemModel interface
public:
    virtual int rowCount(const QModelIndex &parent) const override;
    virtual QVariant data(const QModelIndex &index, int role) const override;
protected:
    // return the roles mapping to be used by QML
    virtual QHash<int, QByteArray> roleNames() const override;
private:
    QList<QColor> m_data;
    QHash<int, QByteArray> m_roleNames;
};

#endif // ROLEENTRYMODEL_H
```

In the header we added the role mapping to be used for QML. When QML tries now to access a property from the model (e.g. “model.name”) the listview will lookup the mapping for “name” and ask the model for data using the NameRole. User defined roles should start with `Qt::UserRole` and need to be unique for each model.

```
#include "roleentrymodel.h"

RoleEntryModel::RoleEntryModel(QObject *parent)
    : QAbstractListModel(parent)
{
    // Set names to the role name hash container (QHash<int, QByteArray>)
    // model.name, model.hue, model.saturation, model.brightness
    m_roleNames[NameRole] = "name";
    m_roleNames[HueRole] = "hue";
    m_roleNames[SaturationRole] = "saturation";
    m_roleNames[BrightnessRole] = "brightness";

    // Append the color names as QColor to the data list (QList<QColor>)
    for(const QString& name : QColor::colorNames()) {
        m_data.append(QColor(name));
    }
}
```

```

}

RoleEntryModel::~RoleEntryModel()
{
}

int RoleEntryModel::rowCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return m_data.count();
}

QVariant RoleEntryModel::data(const QModelIndex &index, int role) const
{
    int row = index.row();
    if(row < 0 || row >= m_data.count()) {
        return QVariant();
    }
    const QColor& color = m_data.at(row);
    qDebug() << row << role << color;
    switch(role) {
    case NameRole:
        // return the color name as hex string (model.name)
        return color.name();
    case HueRole:
        // return the hue of the color (model.hue)
        return color.hueF();
    case SaturationRole:
        // return the saturation of the color (model.saturation)
        return color.saturationF();
    case BrightnessRole:
        // return the brightness of the color (model.brightness)
        return color.lightnessF();
    }
    return QVariant();
}

QHash<int, QByteArray> RoleEntryModel::roleNames() const
{
    return m_roleNames;
}

```

The implementation now has changed only in two places. First in the initialization. We now initialize the data list with QColor data types. Additionally we define our role name map to be accessible for QML. This map is returned later in the `::roleNames` function.

The second change is in the `::data` function. We extend the switch to cover the other roles (e.g hue, saturation, brightness). There is no way to return a SVG name from a color, as a color can take any color and SVG names are limited. So we skip this. Storing the names would require to create a structure `struct { QColor, QString }` to be able to identify the named color.

After registering the type we can use the model and its entries in our user interface.

```

ListView {
    id: view
    anchors.fill: parent
    model: RoleEntryModel {}
    focus: true
    delegate: ListDelegate {
        text: 'hsv(' +
            Number(model.hue).toFixed(2) + ',' +
            Number(model.saturation).toFixed() + ',' +
            Number(model.brightness).toFixed() + ')'
    }
}

```

```
        color: model.name
    }
    highlight: ListHighlight { }
}
```

We convert the returned type to a JS number type to be able to format the number using fixed-point notation. The code would also work without the Number call (e.g. `plain model.saturation.toFixed(2)`). Which format to choose, depends how much you trust the incoming data.

Dynamic Data

Dynamic data covers the aspects of inserting, removing and clearing the data from the model. The `QAbstractListModel` expect a certain behavior when entries are removed or inserted. The behavior is expressed in signals which needs to be called before and after the manipulation. For example to insert a row into a model you need first to emit the signal `beginInsertRows`, then manipulate the data and then finally emit `endInsertRows`.

We will add the following functions to our headers. These functions are declared using `Q_INVOKABLE` to be able to call them from QML. Another way would be to declare them a public slots.

```
// inserts a color at the index (0 at beginning, count-1 at end)
Q_INVOKABLE void insert(int index, const QString& colorValue);
// uses insert to insert a color at the end
Q_INVOKABLE void append(const QString& colorValue);
// removes a color from the index
Q_INVOKABLE void remove(int index);
// clear the whole model (e.g. reset)
Q_INVOKABLE void clear();
```

Additionally we define a `count` property to get the size of the model and a `get` method to get a color at the given index. This is useful when you would like to iterate over the model content from QML.

```
// gives the size of the model
Q_PROPERTY(int count READ count NOTIFY countChanged)
// gets a color at the index
Q_INVOKABLE QColor get(int index);
```

The implementation for `insert` checks first the boundaries and if the given value is valid. Only then do we begin inserting the data.

```
void DynamicEntryModel::insert(int index, const QString &colorValue)
{
    if(index < 0 || index > m_data.count()) {
        return;
    }
    QColor color(colorValue);
    if(!color.isValid()) {
        return;
    }
    // view protocol (begin => manipulate => end)
    emit beginInsertRows(QModelIndex(), index, index);
    m_data.insert(index, color);
    emit endInsertRows();
    // update our count property
    emit countChanged(m_data.count());
}
```

`Append` is very simple. We reuse the `insert` function with the size of the model.

```
void DynamicEntryModel::append(const QString &colorValue)
{
}
```

```
insert(count(), colorValue);
}
```

Remove is similar to insert but it calls according to the remove operation protocol.

```
void DynamicEntryModel::remove(int index)
{
    if(index < 0 || index >= m_data.count()) {
        return;
    }
    emit beginRemoveRows(QModelIndex(), index, index);
    m_data.removeAt(index);
    emit endRemoveRows();
    // do not forget to update our count property
    emit countChanged(m_data.count());
}
```

The helper function count is trivial. It just returns the data count. The get function is also quite simple.

```
QColor DynamicEntryModel::get(int index)
{
    if(index < 0 || index >= m_data.count()) {
        return QColor();
    }
    return m_data.at(index);
}
```

You need to be careful that you only return a value which QML understands. If it is not one of the basic QML types or types known to QML you need to register the type first with `qmlRegisterType` or `qmlRegisterUncreatableType`. You use `qmlRegisterUncreatableType` if the user shall not be able to instantiate its own object in QML.

Now you can use the model in QML and insert, append, remove entries from the model. Here is a small example which allows the user to enter a color name or color hex value and the color is then appended onto the model and shown in the list view. The red circle on the delegate allows the user to remove this entry from the model. After the entry is removed the list view is notified by the model and updates its content.



And here is the QML code. You find the full source code also in the assets for this chapter. The example uses the `QtQuick.Controls` and `QtQuick.Layouts` module to make the code more compact. These controls module provides a set of desktop related ui elements in `QtQuick` and the layouts module provides some very useful layout managers.

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.5
import QtQuick.Layouts 1.2

// our module
import org.example 1.0

Window {
    visible: true
    width: 480
    height: 480

    Background { // a dark background
        id: background
    }

    // our dyanmic model
    DynamicEntryModel {
        id: dynamic
        onCountChanged: {
            // we print out count and the last entry when count is changing
            print('new count: ' + count);
            print('last entry: ' + get(count-1));
        }
    }

    ColumnLayout {
        anchors.fill: parent
        anchors.margins: 8
        ScrollView {
            Layout.fillHeight: true
            Layout.fillWidth: true
            ListView {
                id: view
                // set our dynamic model to the views model property
                model: dynamic
                delegate: ListDelegate {
                    width: ListView.view.width
                    // construct a string based on the models proeprties
                    text: 'hsv(' +
                        Number(model.hue).toFixed(2) + ', ' +
                        Number(model.saturation).toFixed() + ', ' +
                        Number(model.brightness).toFixed() + ')'
                    // sets the font color of our custom delegates
                    color: model.name

                    onClicked: {
                        // make this delegate the current item
                        view.currentIndex = index
                        view.focus = true
                    }
                    onRemove: {
                        // remove the current entry from the model
                        dynamic.remove(index)
                    }
                }
            }
            highlight: ListHighlight { }
        }
    }
}
```

```

        // some fun with transitions :-)
        add: Transition {
            // applied when entry is added
            NumberAnimation {
                properties: "x"; from: -view.width;
                duration: 250; easing.type: Easing.InCirc
            }
            NumberAnimation { properties: "y"; from: view.height;
                duration: 250; easing.type: Easing.InCirc
            }
        }
        remove: Transition {
            // applied when entry is removed
            NumberAnimation {
                properties: "x"; to: view.width;
                duration: 250; easing.type: Easing.InBounce
            }
        }
        displaced: Transition {
            // applied when entry is moved
            // (e.g because another element was removed)
            SequentialAnimation {
                // wait until remove has finished
                PauseAnimation { duration: 250 }
                NumberAnimation { properties: "y"; duration: 75
                }
            }
        }
    }
}
TextEntry {
    id: textEntry
    onAppend: {
        // called when the user presses return on the text field
        // or clicks the add button
        dynamic.append(color)
    }

    onUp: {
        // called when the user presses up while the text field is focused
        view.decrementCurrentIndex()
    }
    onDown: {
        // same for down
        view.incrementCurrentIndex()
    }
}
}
}

```

Model view programming is one of the hardest tasks in Qt. It is one of the very few classes where you have to implement an interface as a normal application developer. All other classes you just use normally. The sketching of models should always start on the QML side. You should envision how your users would use your model inside QML. For this it is often a good idea to create a prototype first using the `ListModel` to see how this best works in QML. This is also true when it comes to defining QML APIs. Making data available from C++ to QML is not only a technology boundary it is also a programming paradigm change from imperative to declarative style programming. So be prepared for some set backs and aha moments:-).

Advanced Techniques

Extending QML with C++

Section author: jryannel

Note: Last Build: March 11, 2018 at 15:30 CET

The source code for this chapter can be found in the assets folder.

Executing QML within the confined space that QML as a language offers can sometimes be limiting. By extending the QML run-time with native functionality written in C++, the application can utilize the full performance and freedom of the base platform.

Understanding the QML Run-time

When running QML, it is being executed in a run-time environment. The run-time is implemented in C++ in the `QtQml` module. It consists of an engine, responsible for the execution of QML, contexts, holding the properties accessible for each component, and components, the instantiated QML elements.

```
#include <QtGui>
#include <QtQml>

int main(int argc, char **argv)
{
    QGuiApplication app(argc, argv);
    QUrl source(QStringLiteral("qrc:/main.qml"));
    QQmlApplicationEngine engine;
    engine.load(source);
    return app.exec();
}
```

In the example the `QGuiApplication` encapsulates all that is related to the application instance (e.g. application name, command line arguments and managing the event loop). The `QQmlApplicationEngine` manages the hierarchical order of contexts and components. It requires typically a qml file to be loaded as the starting point of your application. In this case it is a `main.qml` containing a window and a text type.

Note: Loading a `main.qml` with a simple `Item` as the root type through the `QmlApplicationEngine` will not show anything on your display, as it requires a window to manage a surface for rendering. The engine is capable of loading qml code which does not contain any user interface (e.g. plain objects). Because of this it does not create a window for you by default. The `qmlscene` or the new qml runtime will internally first check if the main qml file contains a window as a root item and if not create one for you and set the root item as a child to the newly created window.

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 512
    height: 300

    Text {
        anchors.centerIn: parent
        text: "Hello World!"
    }
}
```

In the qml file we declare our dependencies here it is `QtQuick` and `QtQuick.Window`. These declaration will trigger a lookup for these modules in the import paths and on success will load the required plugins by the engine. The newly loaded types will then be made available to the qml file controlled by a `qmlDir`.

Is it also possible to shortcut the plugin creation by adding our types directly to the engine. Here we assume we have a `CurrentTime` `QObject` based class.

```
QQmlApplicationEngine engine;

qmlRegisterType<CurrentTime>("org.example", 1, 0, "CurrentTime");

engine.load(source);
```

Now we can also use the `CurrentTime` type in our qml file.

```
import org.example 1.0

CurrentTime {
    // access properties, functions, signals
}
```

For the really lazy there is also the very direct way through context properties.

```
QScopedPointer<CurrentTime> current(new CurrentTime());

QQmlApplicationEngine engine;

engine.rootContext().setContextProperty("current", current.value())

engine.load(source);
```

Note: Do not mix up `setContextProperty()` and `setProperty()`. The first one sets a context property on a qml context, and `setProperty()` sets a dynamic property value on a `QObject` and will not help you.

Now you can use the `current` property everywhere in your application. Thanks to context inheritance.

```
import QtQuick 2.5
import QtQuick.Window 2.0

Window {
    visible: true
    width: 512
    height: 300

    Component.onCompleted: {
        console.log('current: ' + current)
    }
}
```

```
}
}
```

Here are the different ways you can extend QML in general:

- Context properties - `setContextProperty()`
- Register type with engine - calling `qmlRegisterType` in your `main.cpp`
- QML extension plugins - To be discussed next

Context properties are easy to use for small applications. They do not require many effort you just expose your system API with kind of global objects. It is helpful to ensure there will be no naming conflicts (e.g by using a special character for this (\$) for example `$.currentTime`). \$ is a valid character for JS variables.

Registering QML types allows the user to control the lifecycle of an c++ object from QML. This is not possible with the context properties. Also it does not pollute the global namespace. Still all types need to be registered first and by this all libraries need to be linked on application start, which in most cases is not really a problem.

The most flexible system is provided by the **QML extension plugins**. They allow you to register types in a plugin which is loaded when the first QML file calls the import identifier. Also by using a QML singleton there is no need to pollute the global namespace anymore. Plugins allow you to reuse modules across projects, which comes quite handy when you do more than one project with Qt.

For the remainder of this chapter will focus on the qml extension plugins. As they provide the greatest flexibility and reuse.

Plugin Content

A plugin is a library with a defined interface, which is loaded on demand. This differs from a library as a library is linked and loaded on startup of the application. In the QML case the interface is called `QQmlExtensionPlugin`. There are two methods interesting for us `initializeEngine()` and `registerTypes()`. When the plugin is loaded first the `initializeEngine()` is called, which allows us to access the engine to expose plugin objects to the root context. In the majority you will only use the `registerTypes()` method. This allows you to register your custom QML types with the engine on the provided url.

Let us step back a little bit and think about a potential file IO type which would allow us to read/write small text files from QML. A first iteration could look like this in a mocked QML implementation.

```
// FileIO.qml (good)
QObject {
    function write(path, text) {};
    function read(path) { return "TEXT" }
}
```

This is a pure qml implementation of a possible C++ based QML API for exploring an API. We see we should have a read and write function. Where the write function takes a path and a text and the read function takes a path and returns a text. As it looks path and text are common parameters and maybe we can extract them as properties.

```
// FileIO.qml (better)
QObject {
    property url source
    property string text
    function write() { // open file and write text };
    function read() { // read file and assign to text };
}
```

Yes this looks more like a QML API. We use properties to allow our environment to bind to our properties and react on changes.

To create this API in C++ we would need to create an interface something like this.

```
class FileIO : public QObject {
    ...
    Q_PROPERTY(QUrl source READ source WRITE setSource NOTIFY sourceChanged)
    Q_PROPERTY(QString text READ text WRITE setText NOTIFY textChanged)
    ...
public:
    Q_INVOKABLE void read();
    Q_INVOKABLE void write();
    ...
}
```

This FileIO type need to be registered with the QML engine. We want to use it under the “org.example.io” module

```
import org.example.io 1.0

FileIO {
}
```

A plugin could expose several types with the same module. But it can not expose several modules from one plugin. So there is a one to one relationship between modules and plugins. This relationship is expressed by the module identifier.

Creating the plugin

Qt Creator contains a wizard to create a **QtQuick 2 QML Extension Plugin** we use it to create a plugin called fileio with a FileIO object to start with in the module “org.example.io”.

The plugin class is derived from QQmlExtensionPlugin and implements the registerTypes() function. The Q_PLUGIN_METADATA line is mandatory to identify the plugin as an qml extension plugin. Besides this there is nothing spectacular going on.

```
#ifndef FILEIO_PLUGIN_H
#define FILEIO_PLUGIN_H

#include <QQmlExtensionPlugin>

class FileioPlugin : public QQmlExtensionPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.QQmlExtensionInterface")

public:
    void registerTypes(const char *uri);
};

#endif // FILEIO_PLUGIN_H
```

In the implementation of the registerTypes we simply register our FileIO class using the qmlRegisterType function.

```
#include "fileio_plugin.h"
#include "fileio.h"

#include <qqml.h>

void FileioPlugin::registerTypes(const char *uri)
{
    // @uri org.example.io
```

```
qmlRegisterType<FileIO>(uri, 1, 0, "FileIO");
}
```

Interestingly we can not see here the module URI (e.g. **org.example.io**). This seems to be set from the outside.

When you look into your project directory you will find a `qmlDir` file. This file specifies the content of your `qml` plugin or better the QML side of your plugin. It should look like this for you.

```
module org.example.io
plugin fileio
```

The module is the URI under which your plugin is reachable by others and the plugin line must be identical with your plugin file name (under mac this would be `libfileio_debug.dylib` on the file system and `fileio` in the `qmlDir`). These files were created by Qt Creator based on the given information. The module uri is also available in the `.pro` file. There is is used to build up the install directory.

When you call `make install` in your build folder the library will be copied into the Qt `qml` folder (for Qt 5.4 on mac this would be `~/Qt/5.4/clang_64/qml`). The exact path depends on you Qt installation location and the used compiler on your system). There you will find a the library inside the `org/example/io` folder. The content are these two files currently

```
libfileio_debug.dylib
qmlDir
```

When importing a module called `org.example.io`, the `qml` engine will look in one of the import paths and tries to locate the `org/example/io` path with a `qmlDir`. The `qmlDir` then will tell the engine which library to load as a `qml` extension plugin using which module URI. Two modules with the same URI will override each other.

FileIO Implementation

The `FileIO` implementation is straightforward. Remember the API we want to create should look like this.

```
class FileIO : public QObject {
    ...
    Q_PROPERTY(QUrl source READ source WRITE setSource NOTIFY sourceChanged)
    Q_PROPERTY(QString text READ text WRITE setText NOTIFY textChanged)
    ...
public:
    Q_INVOKABLE void read();
    Q_INVOKABLE void write();
    ...
}
```

We will leave out the properties, as they are simple setters and getters.

The `read` method opens a file in read mode and reads the data using a text stream.

```
void FileIO::read()
{
    if(m_source.isEmpty()) {
        return;
    }
    QFile file(m_source.toLocalFile());
    if(!file.exists()) {
        qWarning() << "Does not exists: " << m_source.toLocalFile();
        return;
    }
    if(file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        m_text = stream.readAll();
    }
}
```

```

        emit textChanged(m_text);
    }
}

```

When the text is changed it is necessary to inform others about the change using `emit textChanged(m_text)`. Otherwise property binding will not work.

The write method does the same but opens the file in write mode and uses the stream to write the contents.

```

void FileIO::write()
{
    if(m_source.isEmpty()) {
        return;
    }
    QFile file(m_source.toLocalFile());
    if(file.open(QIODevice::WriteOnly)) {
        QTextStream stream(&file);
        stream << m_text;
    }
}

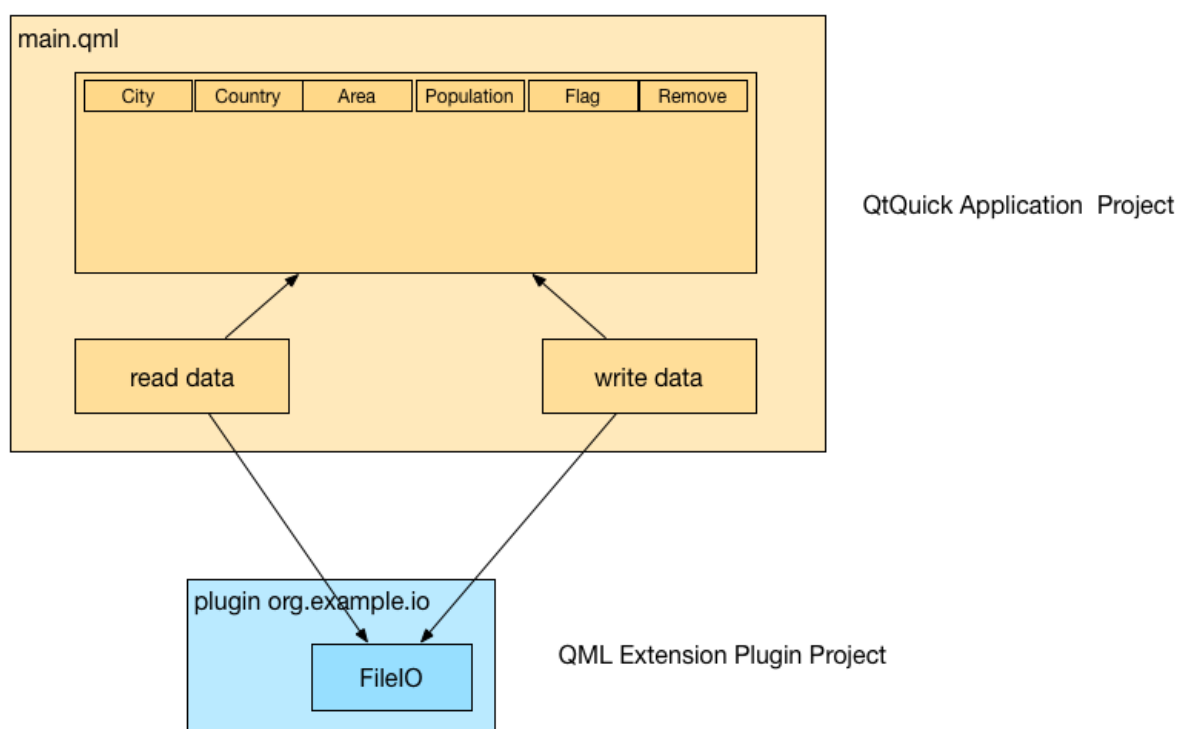
```

Do not forget to call `make install` at the end. Otherwise your plugin files will not be copied over to the `qml` folder and the `qml` engine will not be able to locate the module.

As the reading and writing is blocking you should only use this `FileIO` for small texts, otherwise you will block the UI thread of Qt. Be warned!

Using FileIO

Now we can use our newly created file to access some nice data. For this example we want to read some city data in a JSON format and display it in a table. We will use two projects, one the extension plugin (called `fileio`) which provides us a way to read and write text from a file and the other one, which displays the data in a table (`CityUI`) by using the `file io` for reading and writing of files. The data used in this example is in the `cities.json` file.



JSON is just text, which is formatted in such a way that it can be converted into a valid JS object/array and back to text. We use our `FileIO` to read the JSON formatted data and convert it into a JS object using `JSON.parse()`. The data is later used as a model to the table view. This is roughly the content of our read document function. For saving we convert the data back into a text format and use the write function for saving.

The city JSON data is a formatted text file, with a set of city data entries, where each entry contains interesting data about the city.

```
[
  {
    "area": "1928",
    "city": "Shanghai",
    "country": "China",
    "flag": "22px-Flag_of_the_People's_Republic_of_China.svg.png",
    "population": "13831900"
  },
  ...
]
```

The Application Window

We use the Qt Creator QtQuick Application wizard to create a Qt Quick controls based application. We will not use the new QML forms as this is difficult to explain in a book, although the new forms approach with a *ui.qml* file is much more usable than previous. So you can remove/delete the forms file for now.

The basic setup is an `ApplicationWindow` which can contain a toolbar, menubar and statusbar. We will only use the menubar to create some standard menu entries for opening and saving the document. The basic setup will just display an empty window.

```
import QtQuick 2.5
import QtQuick.Controls 1.3
import QtQuick.Window 2.2
import QtQuick.Dialogs 1.2

ApplicationWindow {
    id: root
    title: qsTr("City UI")
    width: 640
    height: 480
    visible: true
}
```

Using Actions

To better use/reuse our commands we use the QML `Action` type. This will allow us later to use the same action also for a potential tool bar. The open and save and exit actions are quite standard. The open and save action do not contain any logic yet, this we will come later. The menubar is created with a file menu and these three action entries. Additional we prepare already a file dialog, which will allow us to pick our city document later. A dialog is not visible when declared, you need to use the `open()` method to show it.

```
...
Action {
    id: save
    text: qsTr("&Save")
    shortcut: StandardKey.Save
    onTriggered: { }
}

Action {
```

```
    id: open
    text: qsTr("&Open")
    shortcut: StandardKey.Open
    onTriggered: {}
}

Action {
    id: exit
    text: qsTr("E&xit")
    onTriggered: Qt.quit();
}

menuBar: MenuBar {
    Menu {
        title: qsTr("&File")
        MenuItem { action: open }
        MenuItem { action: save }
        MenuSeparator { }
        MenuItem { action: exit }
    }
}

...

FileDialog {
    id: openDialog
    onAccepted: { }
}
```

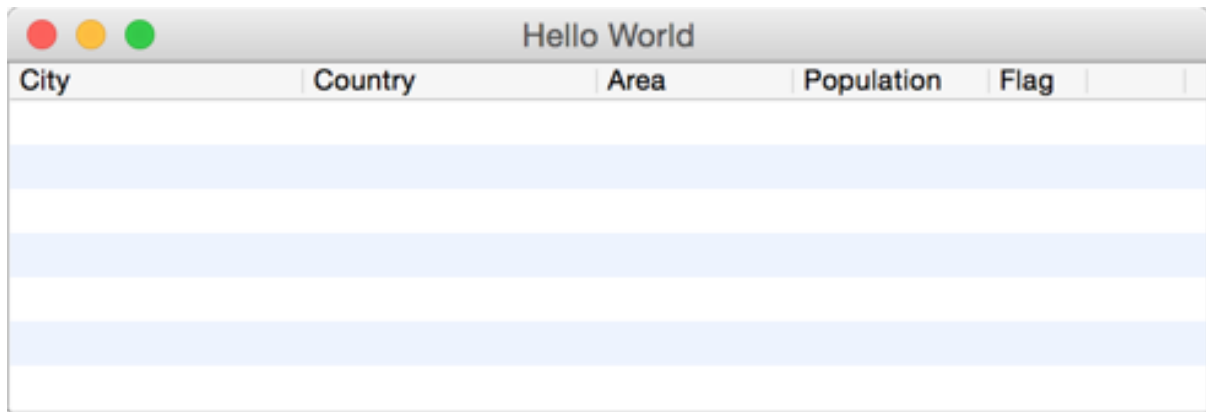
Formatting the Table

The content of the city data shall be displayed in a table. For this we use the `TableView` control and declare 4 columns: city, country, area, population. Each column is a standard `TableViewColumn`. Later we will add columns for the flag and remove operation which will require a custom column delegate.

```
TableView {
    id: view
    anchors.fill: parent
    TableViewColumn {
        role: 'city'
        title: "City"
        width: 120
    }
    TableViewColumn {
        role: 'country'
        title: "Country"
        width: 120
    }
    TableViewColumn {
        role: 'area'
        title: "Area"
        width: 80
    }
    TableViewColumn {
        role: 'population'
        title: "Population"
        width: 80
    }
}
```

Now the application should show you a menubar with a file menu and an empty table with 4 table headers. The

next step will be to populate the table with useful data using our *FileIO* extension.



The `cities.json` document is an array of city entries. Here is an example.

```
[
  {
    "area": "1928",
    "city": "Shanghai",
    "country": "China",
    "flag": "22px-Flag_of_the_People's_Republic_of_China.svg.png",
    "population": "13831900"
  },
  ...
]
```

Our job is it to allow the user to select the file, read it, convert it and set it onto the table view.

Reading Data

For this we let the open action open the file dialog. When the user has selected a file the `onAccepted` method is called on the file dialog. There we call the `readDocument()` function. The `readDocument()` function sets the url from the file dialog to our `FileIO` object and calls the `read()` method. The loaded text from `FileIO` is then parsed using the `JSON.parse()` method and the resulting object is directly set onto the table view as a model. How convenient is that.

```
Action {
    id: open
    ...
    onTriggered: {
        openFileDialog.open()
    }
}

...

FileDialog {
    id: openFileDialog
    onAccepted: {
        root.readDocument()
    }
}

function readDocument() {
    io.source = openFileDialog.fileUrl
    io.read()
    view.model = JSON.parse(io.text)
```

```
}

FileIO {
    id: io
}
```

Writing Data

For saving the document, we hook up the save action to the `saveDocument()` function. The save document function takes the model from the view, which is a JS object and converts it into a string using the `JSON.stringify()` function. The resulting string is set to the `text` property of our `FileIO` object and we call `write()` to save the data to disk. The “null” and “4” parameters on the `stringify` function will format the resulting JSON data using indentation with 4 spaces. This is just for better reading of the saved document.

```
Action {
    id: save
    ...
    onTriggered: {
        saveDocument()
    }
}

function saveDocument() {
    var data = view.model
    io.text = JSON.stringify(data, null, 4)
    io.write()
}

FileIO {
    id: io
}
```

This is basically the application with reading, writing and displaying a JSON document. Think about all the time spend by writing XML readers and writers. With JSON all you need is a way to read and write a text file or send receive a text buffer.

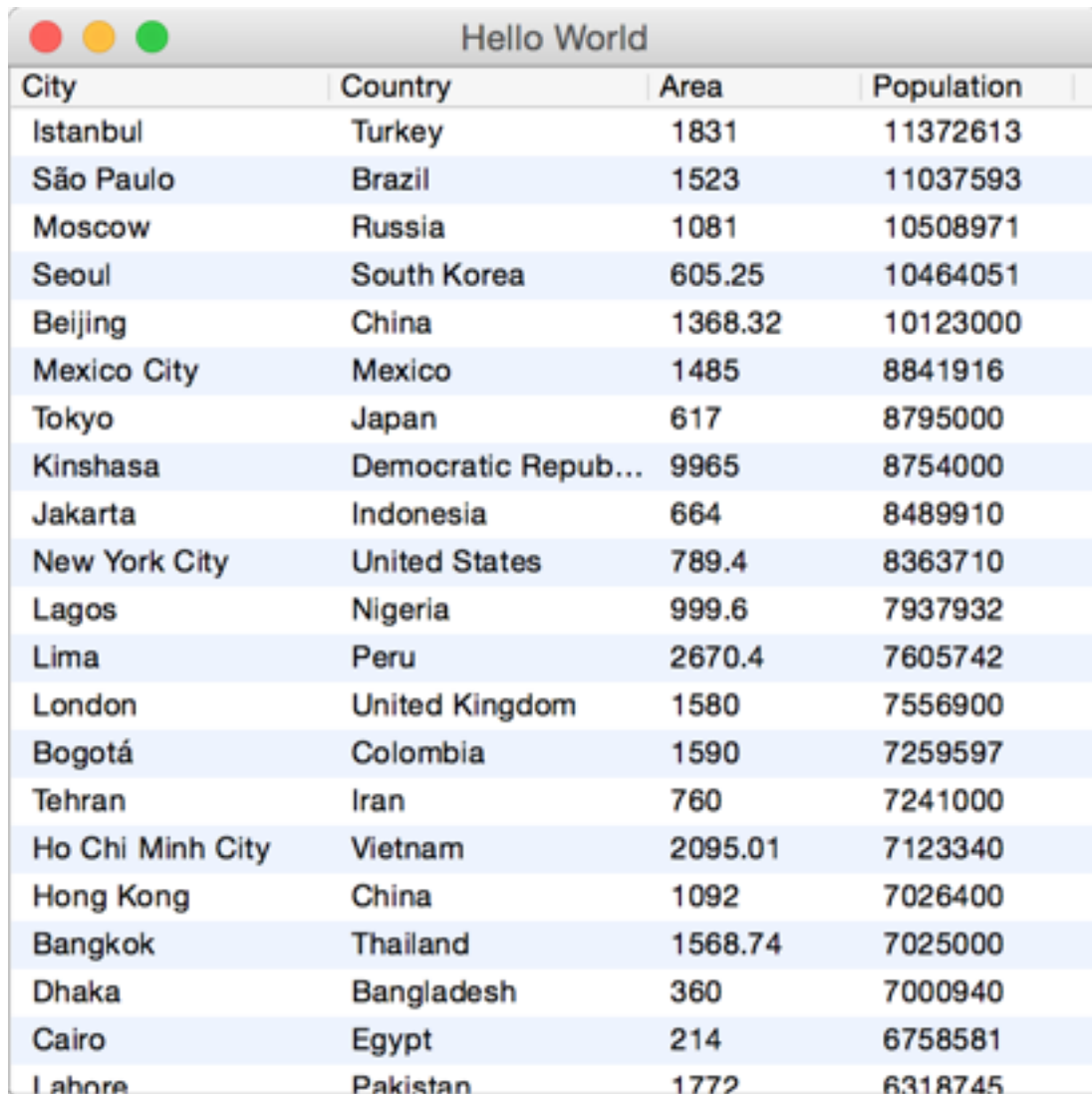
Finishing Touch

The application is not fully ready yet. We still want to show the flags and allow the user to modify the document by removing cities from the model.

The flags are stored for this example relative to the `main.qml` document in a *flags* folder. To be able to show them the table column needs to define a custom delegate for rendering the flag image.

```
TableViewColumn {
    delegate: Item {
        Image {
            anchors.centerIn: parent
            source: 'flags/' + styleData.value
        }
    }
    role: 'flag'
    title: "Flag"
    width: 40
}
```

That is all. It exposes the `flag` property from the JS model as `styleData.value` to the delegate. The delegate then adjust the image path to pre-pend `'flags/'` and displays it.



















































City	Country	Area	Population
Istanbul	Turkey	1831	11372613
São Paulo	Brazil	1523	11037593
Moscow	Russia	1081	10508971
Seoul	South Korea	605.25	10464051
Beijing	China	1368.32	10123000
Mexico City	Mexico	1485	8841916
Tokyo	Japan	617	8795000
Kinshasa	Democratic Repub...	9965	8754000
Jakarta	Indonesia	664	8489910
New York City	United States	789.4	8363710
Lagos	Nigeria	999.6	7937932
Lima	Peru	2670.4	7605742
London	United Kingdom	1580	7556900
Bogotá	Colombia	1590	7259597
Tehran	Iran	760	7241000
Ho Chi Minh City	Vietnam	2095.01	7123340
Hong Kong	China	1092	7026400
Bangkok	Thailand	1568.74	7025000
Dhaka	Bangladesh	360	7000940
Cairo	Egypt	214	6758581
Lahore	Pakistan	1772	6318745

For removing we use a similar technique to display a remove button.

```
TableViewColumn {
    delegate: Button {
        iconSource: "remove.png"
        onClicked: {
            var data = view.model
            data.splice(styleData.row, 1)
            view.model = data
        }
    }
    width: 40
}
```

For the data removal operation we get hold on the view model and then remove one entry using the JS `splice` function. This method is available to us as the model is from the type JS array. The `splice` method changes the content of an array by removing existing elements and/or adding new elements.

A JS array is unfortunately not so smart as a Qt model like the `QAbstractItemModel`, which will notify the view about row changes or data changes. The view will not show any updated data by now as it is never notified about any changes. Only when we set the data back to the view, the view recognizes there is new data and refreshes the view content. Setting the model again using `view.model = data` is a way to let the view know there was a data change.

Hello World						
City	Country	Area	Population	Flag		
Istanbul	Turkey	1831	11372613			
São Paulo	Brazil	1523	11037593			
Moscow	Russia	1081	10508971			
Seoul	South Korea	605.25	10464051			
Beijing	China	1368.32	10123000			
Mexico City	Mexico	1485	8841916			
Tokyo	Japan	617	8795000			
Kinshasa	Democratic Repub...	9965	8754000			
Jakarta	Indonesia	664	8489910			
New York City	United States	789.4	8363710			
Lagos	Nigeria	999.6	7937932			
Lima	Peru	2670.4	7605742			
London	United Kingdom	1580	7556900			
Bogotá	Colombia	1590	7259597			
Tehran	Iran	760	7241000			
Ho Chi Minh City	Vietnam	2095.01	7123340			
Hong Kong	China	1092	7026400			
Bangkok	Thailand	1568.74	7025000			
Dhaka	Bangladesh	360	7000940			
Cairo	Egypt	214	6758581			
Lahore	Pakistan	1772	6318745			
Rio de Janeiro	Brazil	1182	6186710			
Chongqing	China	5467	5954800			
Bangalore	India	709.5	5840155			
Tianjin	China	2057	5800000			
Baghdad	Iraq	1134	5402486			

Summary

The plugin created is a very simple plugin but it can be re-used now and extended by other types for different applications. Using plugins creates a very flexible solution. For example you can now start the UI by just using the `qmlscene`. Open the folder where your `CityUI` project is and start the UI with `qmlscene main.qml`. I really encourage you to write your applications in a way so that they work with a `qmlscene`. This has a tremendous increase in turnaround time for the UI developer and it is also a good habit to keep a clear separation.

Using plugins has one drawback the deployment gets more difficult for simple applications. You need now to deploy your plugin with your application. If this is a problem for you you can still use the same `FileIO` object to register it directly in your `main.cpp` using `qmlRegisterType`. The QML code would stay the same.

Often in larger projects you do not use an application as such. You have a simple qml runtime similar to `qmlscene` and require all native functionality to come as plugins. And your projects are simple pure qml projects using these qml extension plugins. This provides a great flexibility and removes the compilation step for UI changes. After editing a QML file you just need to run the UI. This allows the user interface writers to stay flexible and agile to make all these little changes to push pixels.

Plugins provide a nice and clean separation between C++ backend development and QML frontend development. When developing QML plugins always have the QML side in mind and do not hesitate to start with a QML only mockup first to validate your API before you implement it in C++. If an API is written in C++ people often hesitate to change it or not to speak of to rewrite it. Mocking an API in QML provides much more flexibility and less initial investment. When using plugins the switch between a mocked API and the real API is just changing the import path for the qml runtime.

Assets

The assets contain all files for reading the book offline and also the chapter examples as downloadable format.

Offline Books

- Book as eBook
- Book as PDF
- Book as QtHelp

Source Code Examples

- Chapter 01 examples (ch01-assets.tgz)
- Chapter 04 examples (ch04-assets.tgz)
- Chapter 05 examples (ch05-assets.tgz)
- Chapter 06 examples (ch06-assets.tgz)
- Chapter 07 examples (ch07-assets.tgz)
- Chapter 08 examples (ch08-assets.tgz)
- Chapter 09 examples (ch09-assets.tgz)
- Chapter 10 examples (ch10-assets.tgz)
- Chapter 11 examples (ch11-assets.tgz)
- Chapter 12 examples (ch12-assets.tgz)
- Chapter 13 examples (ch13-assets.tgz)
- Chapter 14 examples (ch14-assets.tgz)
- Chapter 15 examples (ch15-assets.tgz)
- Chapter 16 examples (ch16-assets.tgz)

Chapter Queue

The chapter queue are the chapters we are working on. They might be almost ready or in an infante state. Give it a try if you like.

A

anchors, 46
Animations, 55

B

binding, 27
Bouncing Ball, 55

C

ClickableImage, 55
ClickableImage Helper, 39
ColorAnimation, 55
Column, 42
components, 36

E

Easing Curves, 55

F

Flow, 42
focus, 49
FocusScope, 49

G

gradients, 32
Grid, 42
GridView, 75
Grouped Animations, 55

I

Image, 32
Item, 32

K

KeyNavigation, 49
Keys, 49

L

ListElement, 75
ListModel, 75
ListView, 75

M

MouseArea, 32

N

NumberAnimation, 55

P

ParallelAnimation, 55
PathView, 75
properties, 27

Q

qmlscene, 27

R

Rectangle, 32
Repeater, 42, 75
Rotation, 39
Row, 42

S

Scaling, 39
scripting, 27
SequentialAnimation, 55
Square Helper, 42
Stacking order, 39
States, 55
syntax, 27

T

Text, 32
TextEdit, 49
TextInput, 49
Transformation, 39
Transition, 55
Transitions, 55
Translation, 39

X

XmlListModel, 75