

---

# **Qt5 Cadaques**

***Release 2014-01***

**JRyannel,JThelin**

January 20, 2014 at 18:00 CET



## CONTENTS

|          |                                |            |
|----------|--------------------------------|------------|
| <b>1</b> | <b>Content</b>                 | <b>3</b>   |
| 1.1      | Meet Qt 5 . . . . .            | 3          |
| 1.2      | Get Started . . . . .          | 9          |
| 1.3      | Qt Creator IDE . . . . .       | 19         |
| 1.4      | Quick Starter . . . . .        | 22         |
| 1.5      | Fluid Elements . . . . .       | 49         |
| 1.6      | Model-View-Delegate . . . . .  | 67         |
| 1.7      | Canvas Element . . . . .       | 99         |
| 1.8      | Particle Simulations . . . . . | 115        |
| 1.9      | Shader Effects . . . . .       | 137        |
| 1.10     | Multimedia . . . . .           | 160        |
| 1.11     | Networking . . . . .           | 168        |
| <b>2</b> | <b>Chapter Queue</b>           | <b>187</b> |
|          | <b>Index</b>                   | <b>189</b> |



Last Build: January 20, 2014 at 18:00 CET

## Welcome

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.

## About this book.

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.

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.

Download this book in PDF, or ePUB form for free.

Enjoy!



## CONTENT

### 1.1 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 an 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.

#### 1.1.1 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 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 QtQuick 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.

##### Qt5 Focus

Qt 5 is a complete refreshment 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 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.

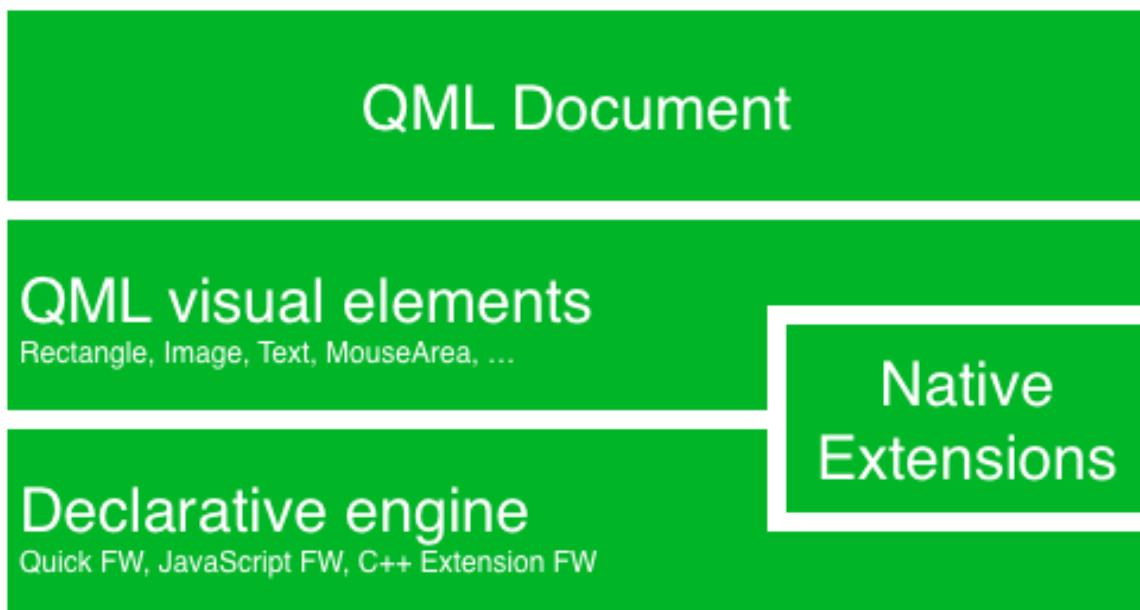
- **Cross-platform portability:** With the consolidated Qt Platform Abstraction, it is now possible to port Qt to a wider range of platforms easier and faster. Qt 5 is structured around the concept of Qt Essentials and Add-ons, which allows OS developer to focus on the essentials modules and leads to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

### 1.1.2 Qt5 Introduction

#### Qt Quick

Qt Quick is the umbrella term for the user interface technology used in Qt5. 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 QtQuick 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's own unit testing framework and exported for the front-end developers to be used.

#### Digesting an User Interface

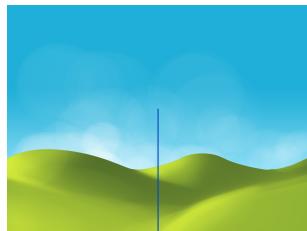
Let's create a simple user interface using QtQuick, which showcases some aspects of the QML language. At the end we will have a paper windmill with rotating blades.

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 2.0

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. a `Image` has a `width`, `height` but also other properties like a `source` property. The size of the `Image` element is automatically deducted 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 2.0` 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 can not be changed after it has been set and it can not 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 pin wheel our foreground element of our user interface is placed as another `Image`.



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: 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 rectangle 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 fluent 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.

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

### 1.1.3 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.

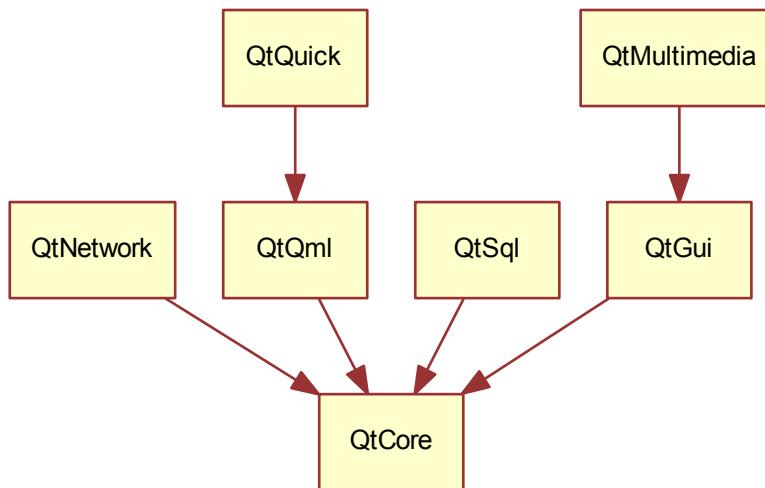
#### Qt Modules

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     | WebKit1 and QWidget-based classes from Qt 4.  |
| Widgets       |   |
| Qt Widgets    | Widgets 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.

### 1.1.4 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 commercial rights to Qt.

Qt has an open-source aspect and a commercial aspect for companies. The commercial aspect is for companies which can not or want not to comply with the open-source licenses. Without the commercial 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-project.org>

## 1.2 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.

---

### 1.2.1 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-Project homepage](#). This is 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. For any circumstances Qt can be used from the command line and you are free to use a code editor of your choice.

---

#### **Todo**

“the Qt SDK” or simply “Qt SDK”

---

#### **Todo**

“For any circumstances”

---

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

### 1.2.2 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 as well as either native plug-ins or native code which launches the runtime code. For the beginning we will not cover the native side of development and focus only of the runtime aspects of Qt5.

---

#### **Todo**

can native plug-ins launch the runtime code?

---

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.

---

#### **Todo**

“read-on”

---

```
// HelloWorld.qml

import QtQuick 2.0

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, just so much it's describes the user interface in a series of hierarchical elements. This code in particular show a rectangle of 300 x 300 pixel geometry with a centered text “Hello World”. A mouse area spans the whole rectangle and when the user clicks it, the application quits.

---

### Todo

“just so much it's describes the user interface in a series of hierarchical elements”

---

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

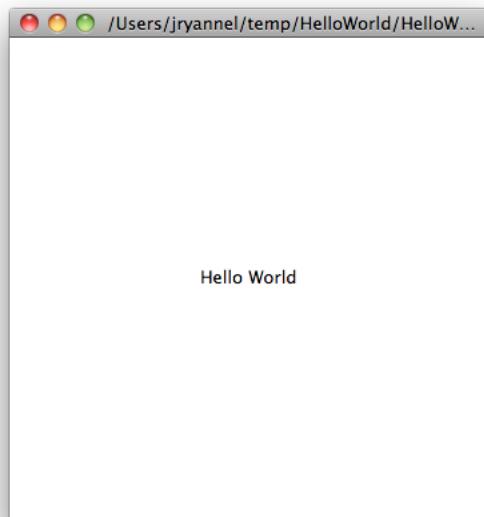
If everything goes well you should see now something like this:

---

### Todo

“If everything goes well”

---



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 Qt5 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 QtQuick 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 2.0

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Greetings from Qt5"
    }
    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
```

## 1.2.3 Application Types

This section is a run through the different types of possible application types someone could write with Qt5. It's not limited to the presented selection but it should give the reader a better idea about what could be done with Qt5 in general.

### Console Application

A console application does not provide any graphical HMI and will normally be called as part of a system service or from the command line. Qt5 comes with a series of ready made components which help you to create console applications cross platform and very efficient. For example networking API or file API. Also string handling or 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 as Qt being *just* as UI toolkit it has so much more to offer.

## String Handling

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

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

// text stream is text-codec 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 lists 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 then 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. By 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 empty string 'QString("")'
        if(line.isEmpty()) {
            continue;
        }
        // test for null string 'String()'
        if(line.isNull()) {
            break;
        }
        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 closes our section about console based application with Qt.

## Widget Application

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

In the first snippet for widget based applications we do as little as creating a window and show it. A widget without a parent is in the Qt world a window. We use the scope pointer to ensure the widget is deleted when the scoped pointer goes out of scope. The application object encapsulate the qt runtime and with the exec call we start the event loop. From there on the application re-acts only on events triggered by mouse or keyboard or other event providers like networking or file IO. The application will only exits when the event loop is exited, this is done by call ‘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 of you need to create custom made widgets. Typically a widget is a window area filled with painting calls. Additional the widget has internal knowledge how to handle keyboard or mouse

input of how to react on 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 to much detail into this code, good to know that you have the possibilities. 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 the different systems. Your job is then to arrange these different widgets in a widget container to 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. They re-layout the widgets according to some size policies when the container widget is re-sized. In this example we have a list, line edit and 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 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

Till 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 the 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 has also support for other data base engines (e.g. 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 data base 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("QSQLITE");
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 file 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 then 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 Qt5. 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 then desktop applications. They do one thing and this simple and focused. Animations is an important part. 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 then our back-end services. In a traditional technology you develop the so called front-end with the same pace as the back-

end. Which results into 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 both from both worlds.

This is a simple Qt Quick UI below

```
import QtQuick 2.0

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

The declaration language is called QML and it needs to runtime to run it. 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.

```
QQQuickView* view = new QQuickView();
QUrl source = QUrl::fromLocalUrl("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 want to use a possible city model. In this case the front-end expects a object named `cityModel` which we can use inside a list view.

```
import QtQuick 2.0

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 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);
```

**Warning:** 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.

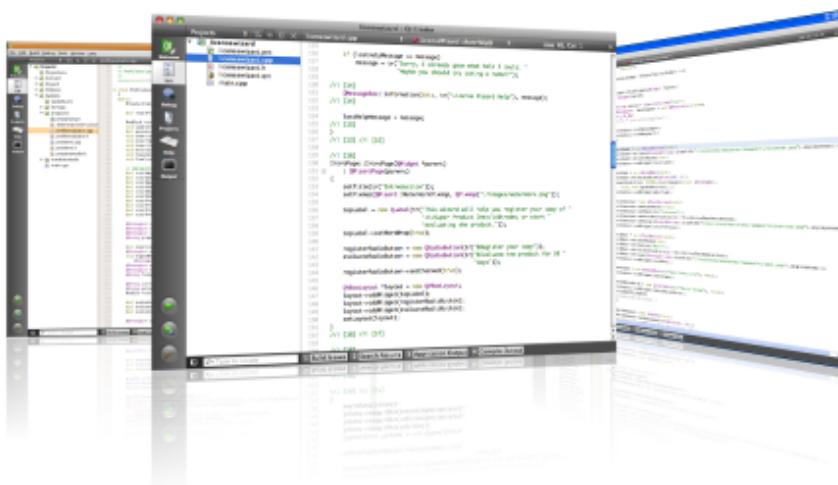
## 1.2.4 Summary

We have seen how to install the Qt SDK and how to create our first application. The we walked you through the different application types to give you an overview of Qt. Showed off some features Qt offers you 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 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.

## 1.3 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 can be used 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 shines really in making the developer productive. Qt Creator can be used to run your QtQuick user interface but also to compile c++ code and this from your host system or using a cross-compiler for your device system.




---

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

---

### 1.3.1 The User Interface

When starting Qt Creator you it comes up in a *Welcome* screen. There you will find the most important hints how to continue inside Qt Creator but also 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: For people who need from time time read 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. When you need to configure your project you will visit from time to time the Projects mode. And then you press Run. Qt Creator is smart enough to ensure your project is fully build before running it.

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

### 1.3.2 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.

---

### 1.3.3 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 QtQuick or QtQuick 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 / QtQuick 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 QtQuick UI. A plug-in is used to extend QtQuick 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.

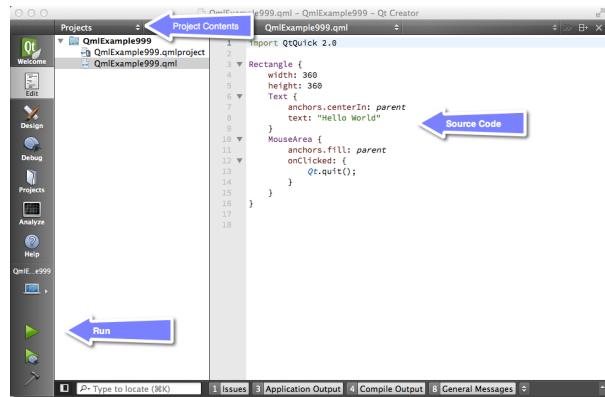
---

**Note:** During the first parts of the book we will mainly use the QtQuick 2.0 UI project type. Later to describe

some c++ aspects we will use the Empty-Qt-Project type or something similar. For extending QtQuick with our own native plug-ins we will use the *Qt Quick 2.0 Extension Plug-in* wizard type.

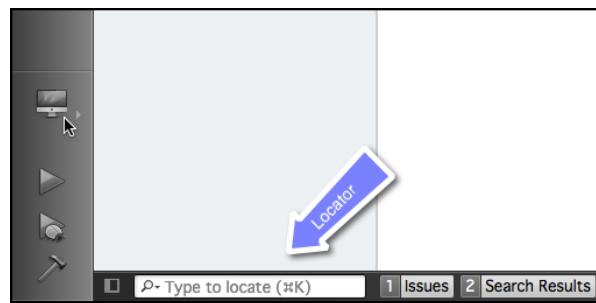
### 1.3.4 Using the Editor

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.

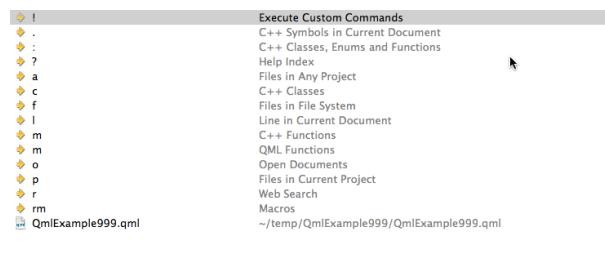


### 1.3.5 Locator

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.

## 1.3.6 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](#).

---

## 1.3.7 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.

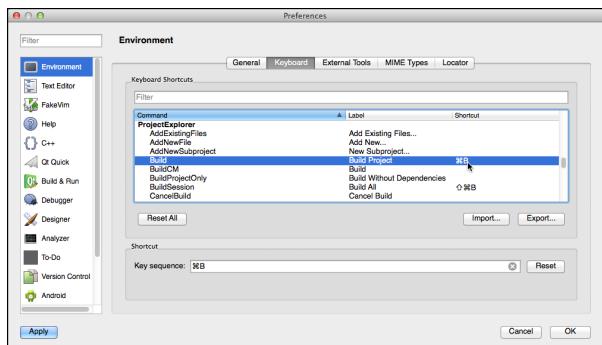
To get started we have collection some basic shortcuts (in Windows notation):

- 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.



---

## 1.4 Quick Starter

*Section author: jryannel*

---

**Note:** Last Build: January 20, 2014 at 18:00 CET

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

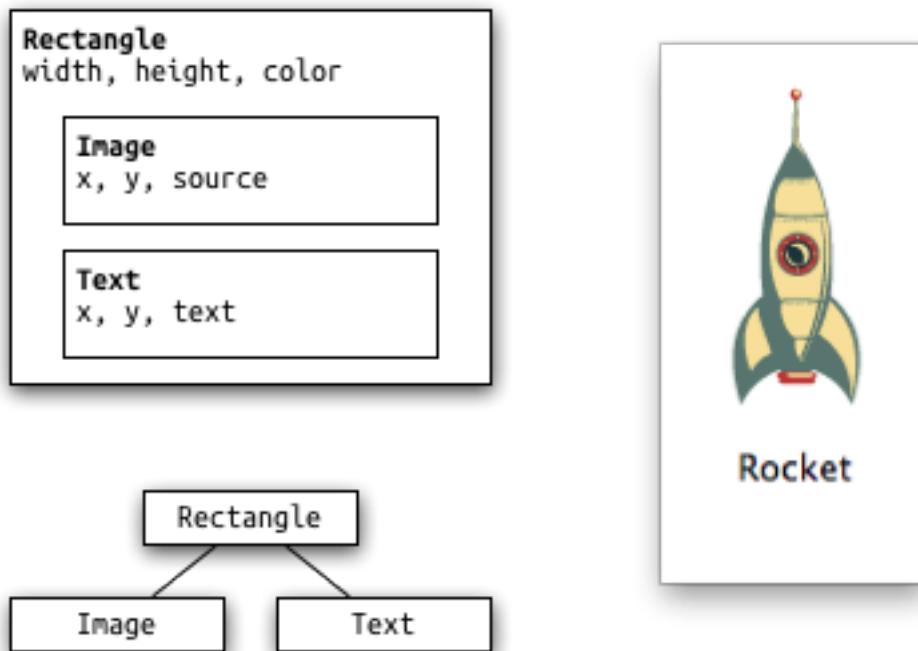
---

This chapter provides an overview of QML, the declarative user interface language used in Qt5. 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.

### 1.4.1 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. A `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.

```
// rectangle.qml

import QtQuick 2.0

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

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

    // color property
    color: "#D8D8D8"

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

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

        source: 'assets/rocket.png'
    }

    // Another child of root
}
```

```
Text {
    // un-named element

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

    // reference root element
    width: root.width

    horizontalAlignment: Text.AlignHCenter
    text: 'Rocket'
}
}
```

- 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, means 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 QtQuick runtime from the command line from your OS like this:

```
$ $QTDIR/bin/qmlscene rectangle.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 `rectangle.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 needed 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, neither 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 an 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 leads normally 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 `spacePresses`. Every time `spacePresses` 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 binding! You need to use assignment on both property change paths as the binding will be destroyed by the assignment (broken contract!).

## 1.4.2 Basic Elements

Elements can be grouped into visual and non-visual elements. A visual element (like the `Rectangle`) has a geometry and normally present 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> (left, right, top, bottom, vertical and horizontal center) to position elements relative to other elements with their <code>margins</code>  |
| 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 current <code>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.

---

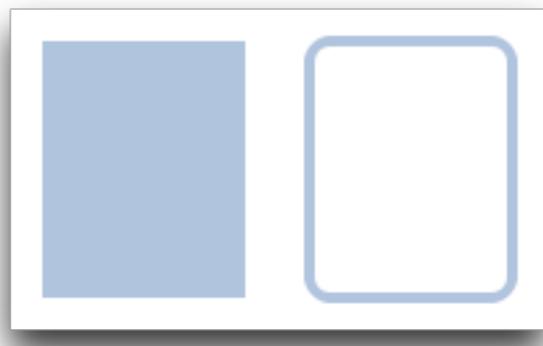
**Note:** The `Item` element is often used as a container for other elements, similar to the `div` element in HTML.

---

### Rectangle Element

The `Rectangle` extends `Item` and adds a fill color to it. Additional 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  
}
```



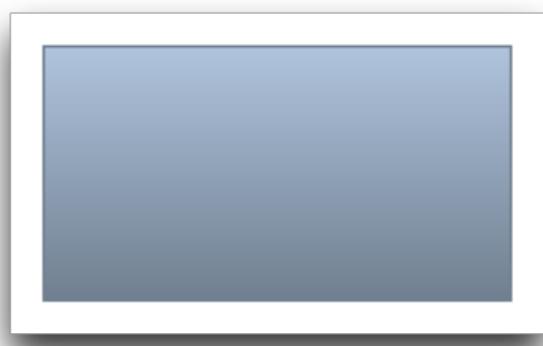
---

**Note:** The named colors used 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 the most common ways are as 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 is 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
}
```



The quick brown fox

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.

---

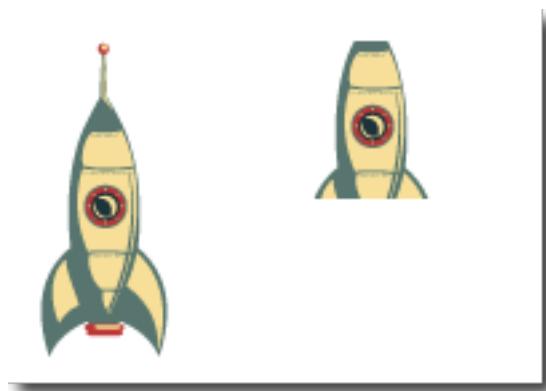
**Note:** Often when you want to layout `Text` elements you need to differentiate between aligning the text inside the `Text` element boundary box or to align the element boundary box itself. In the former you want to use the `horizontalAlignment` and `verticalAlignment` properties and in the later case 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). *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: 48
    // height: 118
    source: "assets/rocket.png"
}
Image {
    x: 112; y: 12
    width: 48
    height: 118/2
    source: "assets/rocket.png"
    fillMode: Image.PreserveAspectCrop
    clip: true
}
```



---

**Note:** An 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 to be rendered outside the `Image` boundaries. By default clipping is disabled (`clip : false`). You need to enable clipping (`clip : true`) to restrain 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.

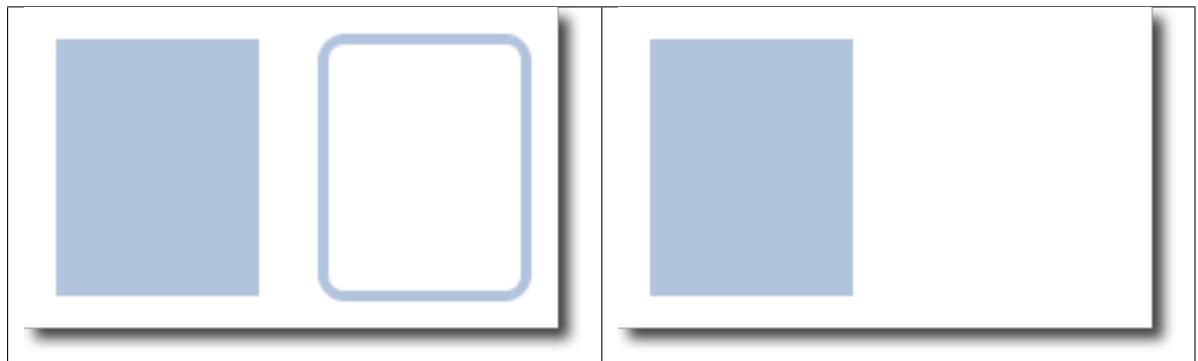
---

## MouseArea Element

To interact with these elements you often will use a MouseArea. Its 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 QtQuick, 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.

---

### 1.4.3 Components

A component is a reusable element and QML provides different ways to create components. But currently we are only interested in only way: 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 { ... }`.

Let's go for this example. We create a rectangle containing a text and a mouse area. This resembles a simple button and doesn't need to be more complicated for our purpose.

```

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 imagine yourself how someone else should use your button. Here is my imagination:

```

// my ideal 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 2.0

Rectangle {
    id: root
    // export button properties

```

```

property alias text: label.text
signal clicked

width: 116; height: 26
color: "lightsteelblue"
border.color: "slategray"

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 simple 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
    Rectangle {
        anchors.fill parent
        color: "lightsteelblue"
    }
}

```

```
    border.color: "slategrey"
}
...
}
```

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

#### 1.4.4 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 operation 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` are still the same. 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. By this we follow a simple rule, after three times using the same code it is better to extract a component.

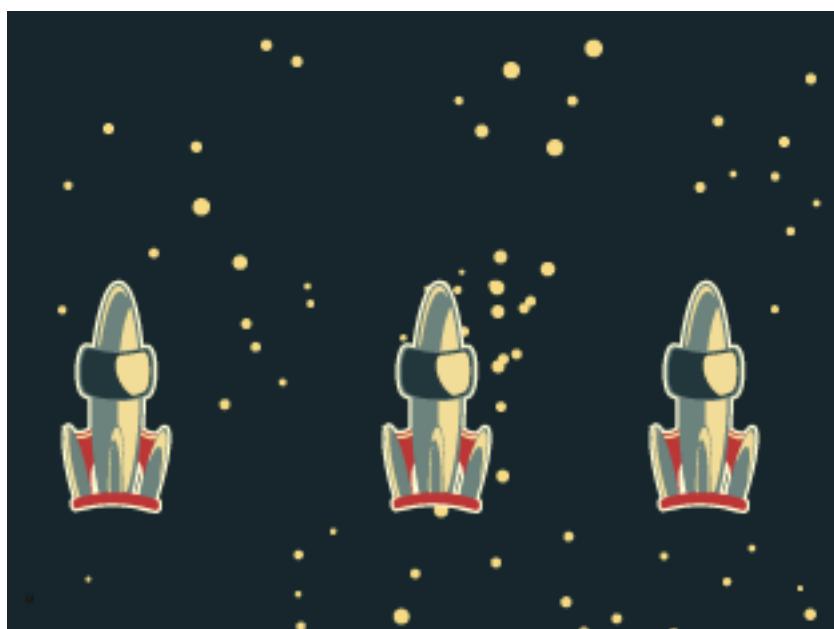
```
// ClickableImage.qml

// Simple image which can be clicked

import QtQuick 2.0

Image {
    id: root
    signal clicked

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



We use our clickable image to present three rockets. Each rocket performs a simple transformation when clicked. Clicking the background will reset the scene.

```
// transformation.qml

import QtQuick 2.0

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

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

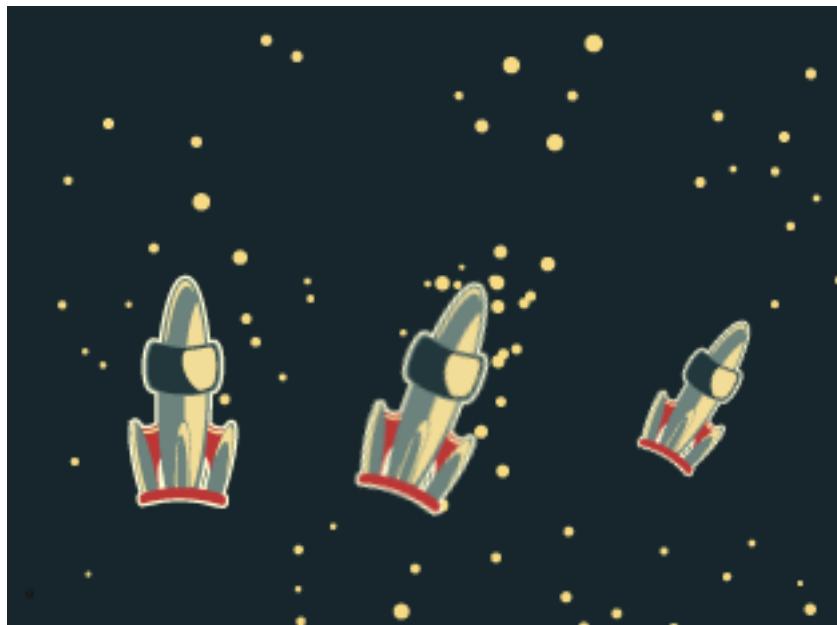
    MouseArea {
        id: backgroundClicker
        // needs to be before the images as order matters
        // otherwise this mousearea would be before the other elements
        // and consume the mouse events
        anchors.fill: parent
        onClicked: {
            // reset our little scene
            rocket1.x = 20
            rocket2.rotation = 0
            rocket3.rotation = 0
            rocket3.scale = 1.0
        }
    }

    ClickableImage {
        id: rocket1
        x: 20; y: 100
        source: "assets/rocket.png"
        onClicked: {
            // increase the x-position on click
            x += 5
        }
    }

    ClickableImage {
        id: rocket2
        x: 140; y: 100
        source: "assets/rocket.png"
        smooth: true // need antialising
        onClicked: {
            // increase the rotation on click
            rotation += 5
        }
    }

    ClickableImage {
        id: rocket3
        x: 240; y: 100
        source: "assets/rocket.png"
        smooth: true // need antialising
        onClicked: {
            // several transformations
            rotation += 5
            scale -= 0.05
        }
    }
}
```

```
    }  
}  
}
```



Rocket-1 increments the x-position by 5 px on each click and rocket-2 will continue to rotate on each click. Rocket-3 will rotate and scale the image down on each click. For the scaling and rotation operation we set `smooth : true` to enable anti-aliasing, which is switched off (same as the clipping property `clip`) for performance reasons. When you see in your own work some rasterized edges in your graphics, then probably you would like to switch `smooth` on.

---

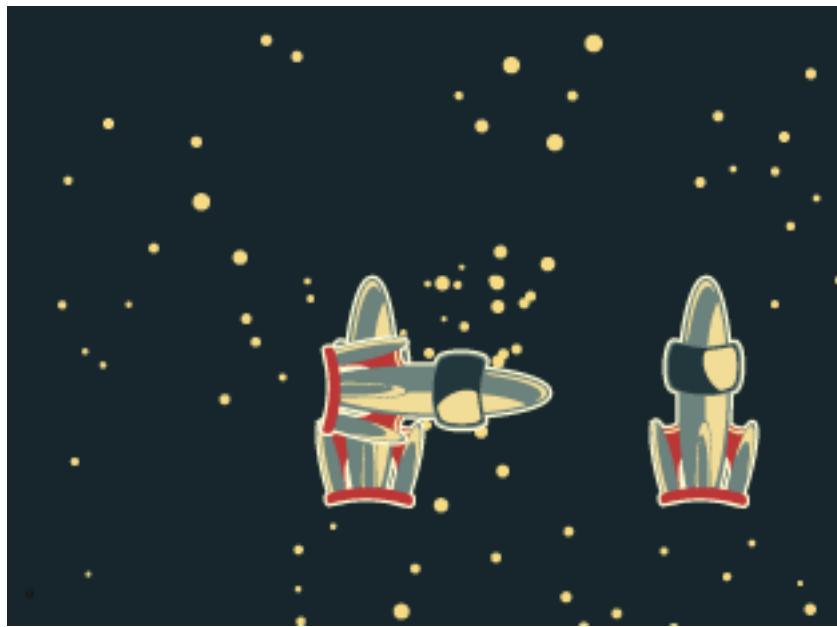
**Note:** To achieve better visual quality when scaling images it is recommended to scale images down instead of up. Scaling an image up with a larger scaling factor will result into scaling artifacts (blurred image). When scaling an image you should consider using `smooth : true` to enable the usage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

---

The background clicker `MouseArea` covers the whole background and resets the rocket values.

---

**Note:** Elements which appear earlier in the code have a lower stacking order (called z-order). If you click long enough on `rocket1` you will see it moves below `rocket2`. The z-order can also be manipulated by the `z`-property of an Item.



This is because `rocket2` 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.*

---

#### 1.4.5 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 square. Each of these components contains a 48x48 pixel colorized rectangle. As reference here is the source code for the RedSquare:

```
// RedSquare.qml

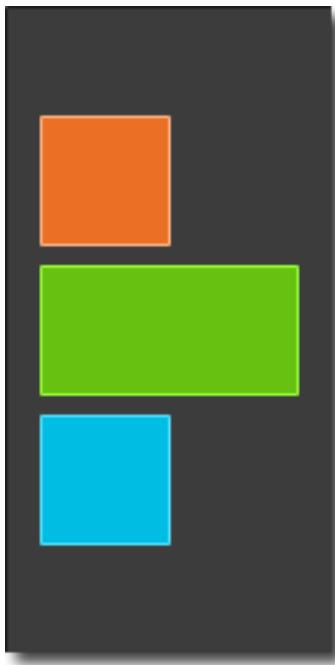
import QtQuick 2.0

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 pixel.

---

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 2.0

DarkSquare {
    id: root
    width: 120
    height: 240

    Column {
        id: row
        anchors.centerIn: parent
        spacing: 8
        RedSquare { }
        GreenSquare { width: 96 }
        BlueSquare { }
    }
}
```

```
// M1<<
```

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 2.0
```

```
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 of 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 2.0

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.



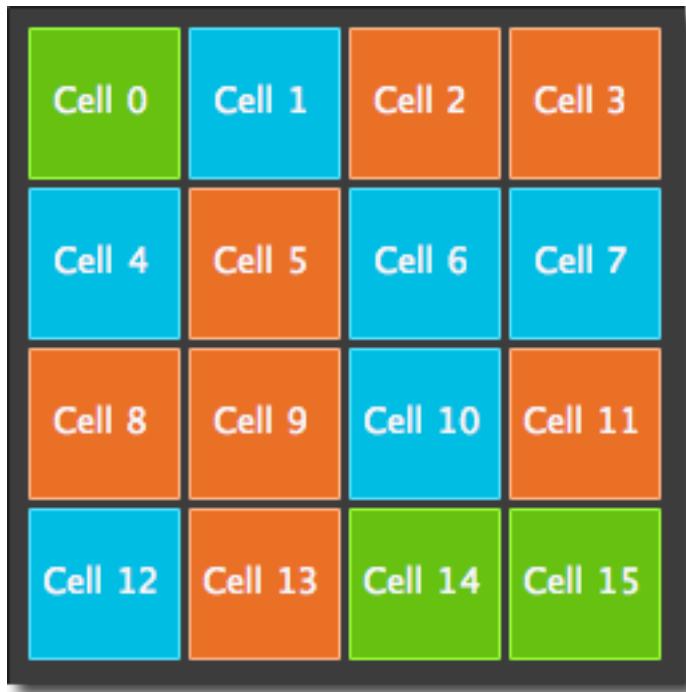
```
// flow.qml

import QtQuick 2.0

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 2.0

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 as 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 QtQuick, 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.

---

## 1.4.6 Layout Items

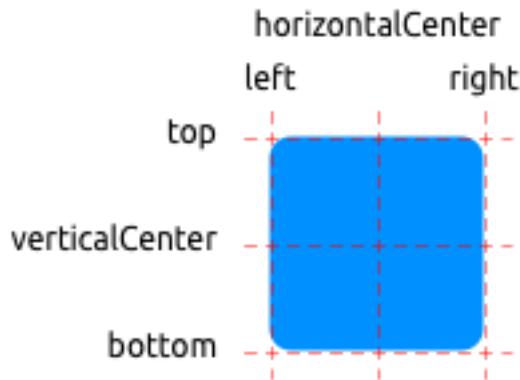
---

### Todo

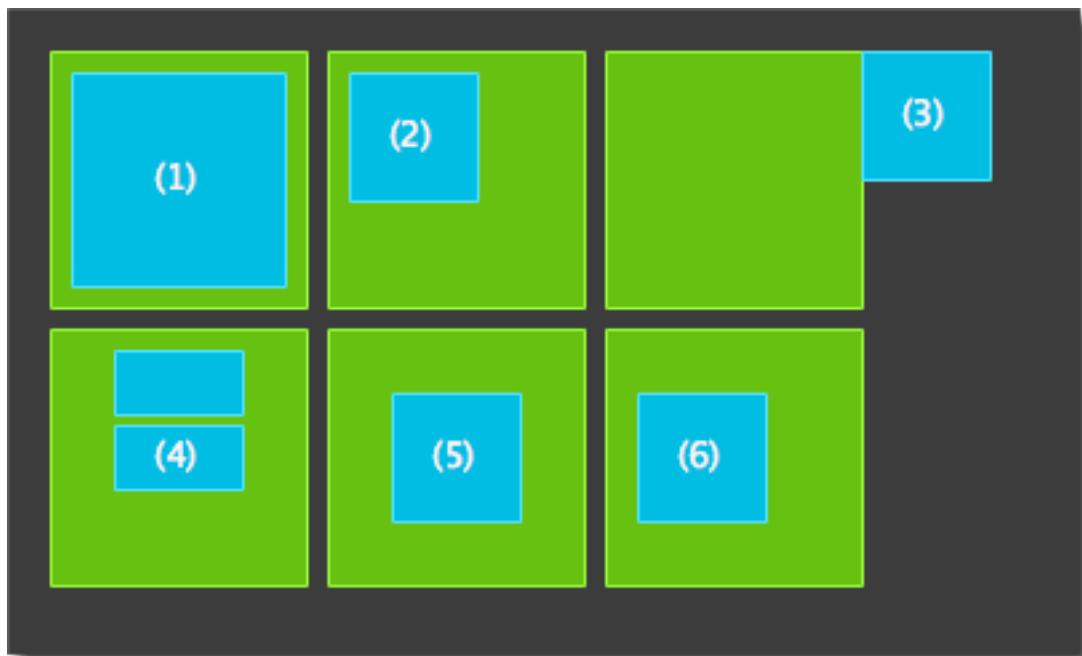
do we need to remove all uses of anchors earlier?

---

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 relativity, 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. A element is centered on a parent element

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

6. A 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` changes dragging is restricted by the anchored lines. We will see this effect later when we discuss animations.

---

## 1.4.7 Input Elements

We have used already the `MouseArea` as mouse input element. Here we would like to focus more on the keyboard input possibilities. 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` and `inputMask` as also an `echoMode`.

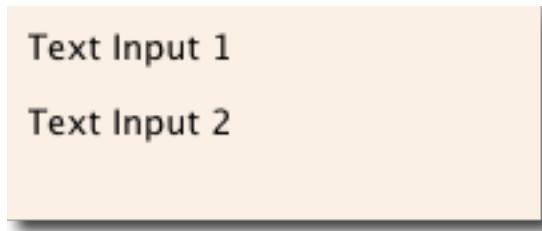
```
// textinput.qml

import QtQuick 2.0

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 2.0

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 to ensure to 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 2.0

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
}
}
```

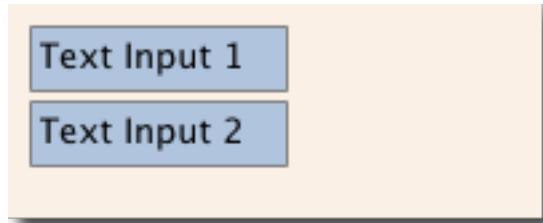
---

**Note:** If you like 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 2.0

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
}
}

```

And 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 misses the text constraint properties for this it provides querying the painted size of the text (`paintedHeight`, `paintedWidth`). 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 2.0

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 2.0

Rectangle {
    width: 136
    height: 120

```

```
color: "linen"

TTTextEdit {
    id: input
    x: 8; y: 8
    width: 120; height: 104
    focus: true
    text: "Text Edit"
}
}
```



## Keys Element

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

```
// keys.qml

import QtQuick 2.0

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;
        }
    }
}
```



## 1.4.8 Advanced Techniques

---

### **Todo**

To be written

---

## 1.5 Fluid Elements

*Section author: jryannel*

---

**Note:** Last Build: January 20, 2014 at 18:00 CET

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

---

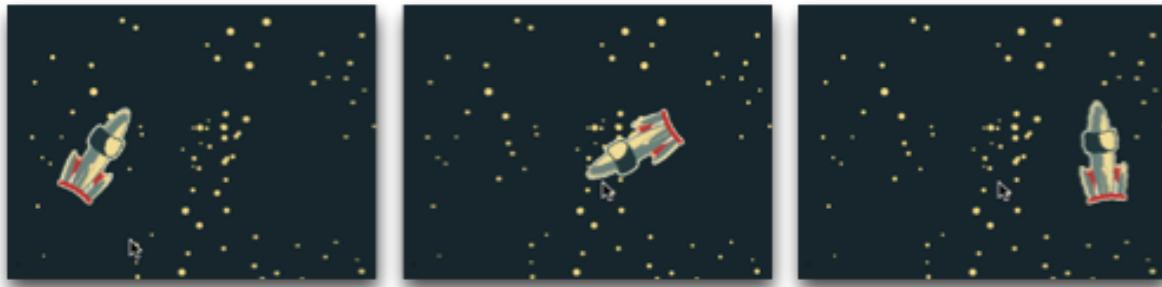
Till now, we have mostly looked at simple graphical elements and how to arrange and manipulate them. This chapter is about how to control these changes in a way that a value of a property not just changes instantly, it's more how the value changes over time: an animation. This technology is one of the 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.

### 1.5.1 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 QtQuick 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 2.0

Image {
    source: "assets/background.png"

    Image {
        x: 40; y: 80
        source: "assets/rocket.png"

        NumberAnimation on x {
            to: 240
            duration: 4000
            loops: Animation.Infinite
        }
        RotationAnimation on rotation {
            to: 360
            duration: 4000
            loops: Animation.Infinite
        }
    }
}
```

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 rocket 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 rocket is disappearing in the endless 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, QtQuick 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 QtQuick 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 2.0

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: "#111111"
        }
    }

    MouseArea {
        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 rocket race can start.



The three rockets 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.

```
ClickableImageV3 {
    id: rocket1
    x: 40; y: 200
    source: "assets/rocket2.png"
    text: "animation on property"
    NumberAnimation on y {
        to: 40; duration: 4000
    }
}
```

### 1st rocket

The 1st rocket travels using the Animation on <property> strategy. The animation starts immediately. When a rocket is clicked their y-position is reset to the start position, this applies to all rockets. On the 1st rocket 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.*

```
ClickableImageV3 {
    id: rocket2
    x: 152; y: 200
    source: "assets/rocket2.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 rocket

The 2nd rocket 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 rocket 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 rocket will always animate to the new position and adapt its speed to match the 4 seconds to the destination defined by the animations duration.

```
ClickableImageV3 {
    id: rocket3
    x: 264; y: 200
    source: "assets/rocket2.png"
    onClicked: anim.start()
//      onClicked: anim.restart()

    text: "standalone animation"

    NumberAnimation {
        id: anim
        target: rocket3
        properties: "y"
        from: 205
        to: 40
        duration: 4000
    }
}
```

## 3rd rocket

The 3rd rocket uses a standalone animation. The animation is defined as an 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 then 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 rockets to their initial position. The 1st rocket can't be restarted only be 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
}
```

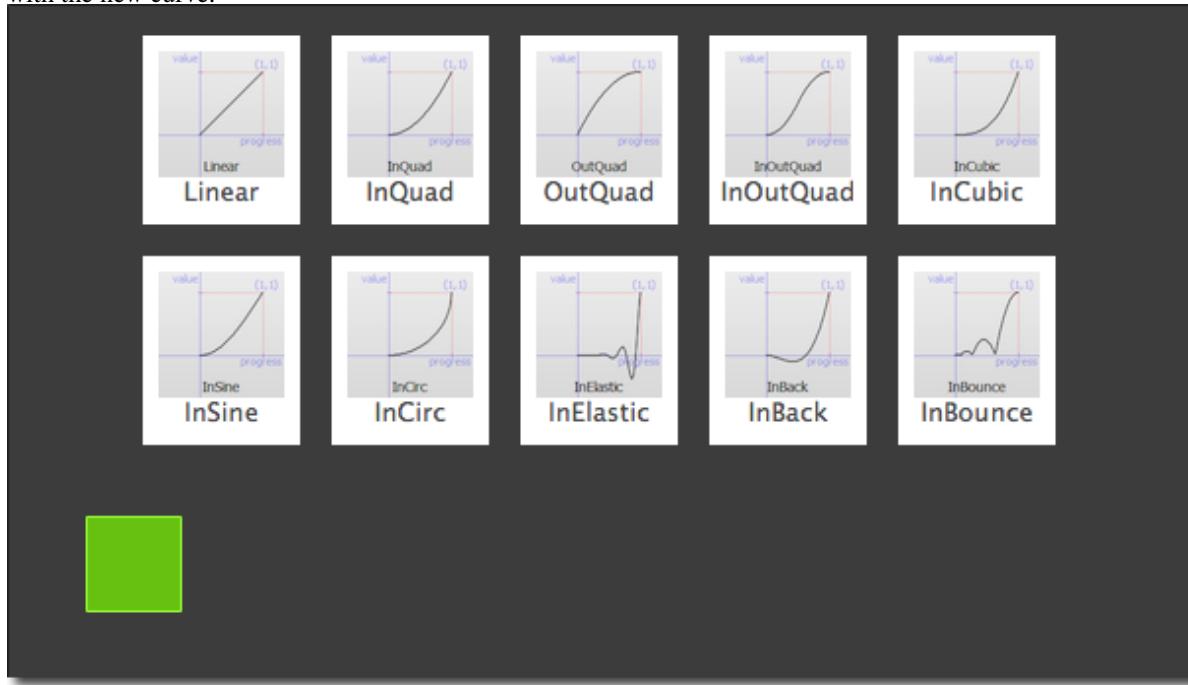
## Easing Curves

The value change of a property can be controlled by an animation. Easing attributes allows to influence 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 a book side.

**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 reacts very sensible on moving objects and 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 as also trigger a restart () to run the animation with the new curve.



### Extended ClickableImage V3

We enhanced the ClickableImage for another time be able to add a small frame around the image and text. For that we added a property bool framed: false property to our API and based on framed value we set the visibility of the frame to not break existing element users. Here are the changes we did.

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

import QtQuick 2.0

Item {
    id: root
    width: container.childrenRect.width + 16
    height: container.childrenRect.height + 16
    property alias text: label.text
    property alias source: image.source
    signal clicked

    // M1>>
    // ... add a framed rectangle as container
    property bool framed : false

    Rectangle {
        anchors.fill: parent
        color: "white"
        visible: root.framed
    }
}
```

The code for this example was made compact. We use an array of easing curve names (property variant easings) and assign them inside a Repeater element to a ClickableImage. The image source is defined through a naming scheme, so an easing curve named “InQuad” will have a respective diagram under the location “curves/InQuad.png”. If you click one curve diagram the click handler will assign the easing type to the animation and then restart the animation. The animation itself is a standalone-animation with the target set to the square and configured for x-property animation with a duration of 2 secs.

```
// easingtypes.qml

import QtQuick 2.0

DarkSquare {
    id: root
    width: 600
    height: 340

    // A list of easing types
    property variant easings : [
        "Linear", "InQuad", "OutQuad", "InOutQuad",
        "InCubic", "InSine", "InCirc", "InElastic",
        "InBack", "InBounce" ]

    Grid {
        id: container
        anchors.top: parent.top
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.margins: 16
        height: 200
        columns: 5
        spacing: 16
        // iterates over the 'easings' list
        Repeater {
            model: easings
            ClickableImageV3 {
                framed: true
                // the current data entry from 'easings' list
                text: modelData
                source: "curves/" + modelData + ".png"
                onClicked: {
                    // set the easing type on the animation
                    anim.easing.type = modelData
                    // restart the animation
                    anim.restart()
                }
            }
        }
    }
}

// The square to be animated
GreenSquare {
    id: square
    x: 40; y: 260
}

// The animation to test the easing types
NumberAnimation {
    id: anim
    target: square
    from: 40; to: root.width - 40 - square.width
    properties: "x"
    duration: 2000
}
}
```

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

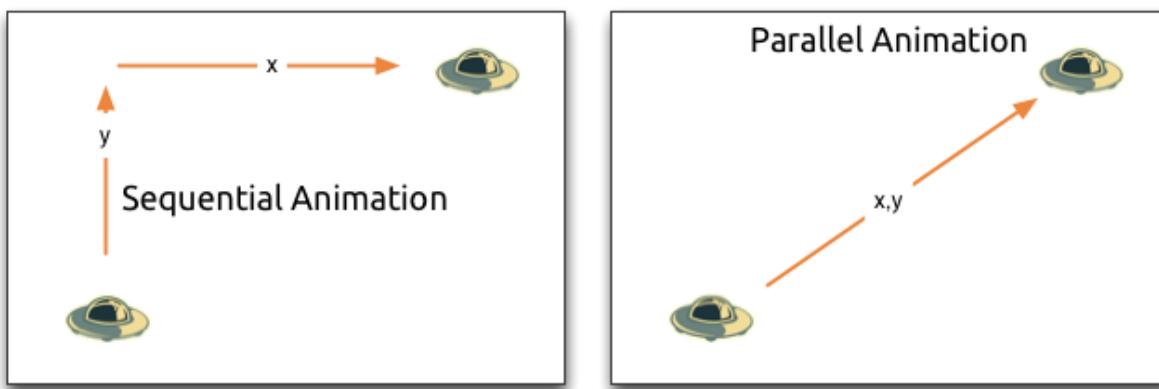
Besides the duration and easing.type you are able to fine tune animations. For example the general PropertyAnimation where most animation inherit from supports additionally an easing.amplitude,

`easing.overshoot` and `easing.period` property which allows you to fine-tune the behavior of particular easing curve. 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 than just animating one property. It might be you want to run several animation combined 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 name suggests it's possible to group animation. Grouping can be in two ways: parallel or sequential. For this you can use the `SequentialAnimation` or the `ParallelAnimation` element, which act as an 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.

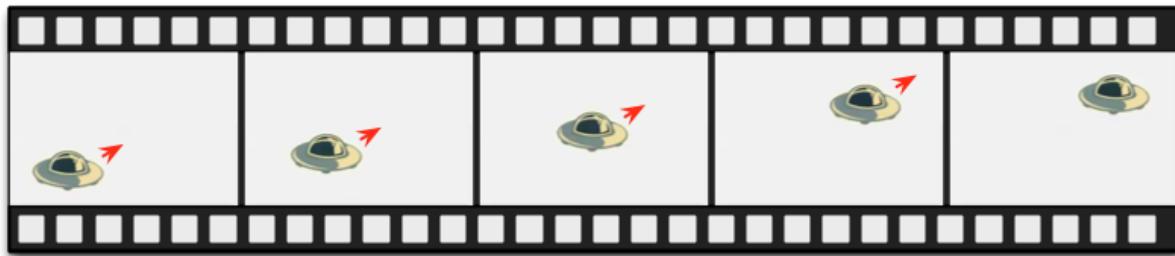
```
// parallelanimation.qml
import QtQuick 2.0

BrightSquare {
    id: root
    width: 300
    height: 200
    property int duration: 3000

    ClickableImageV3 {
        id: rocket
        x: 20; y: 120
        source: "assets/rocket2.png"
        onClicked: anim.restart()
    }

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

```
        duration: root.duration  
    }  
}
```



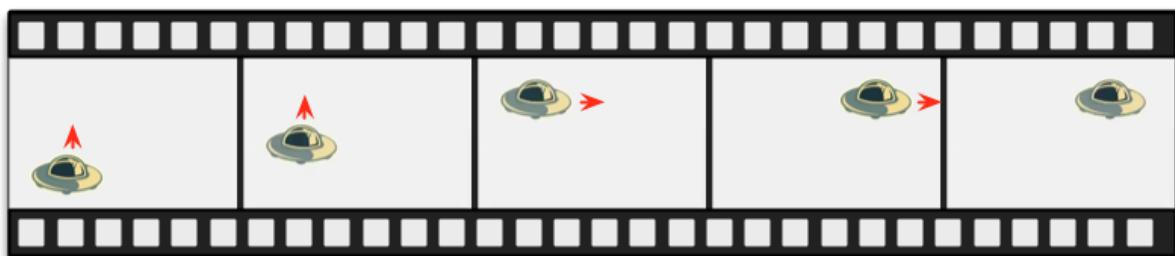
A sequential animation will first run the first child animation and then continue from there.

```
// sequentialanimation.qml
import QtQuick 2.0

BrightSquare {
    id: root
    width: 300
    height: 200
    property int duration: 3000

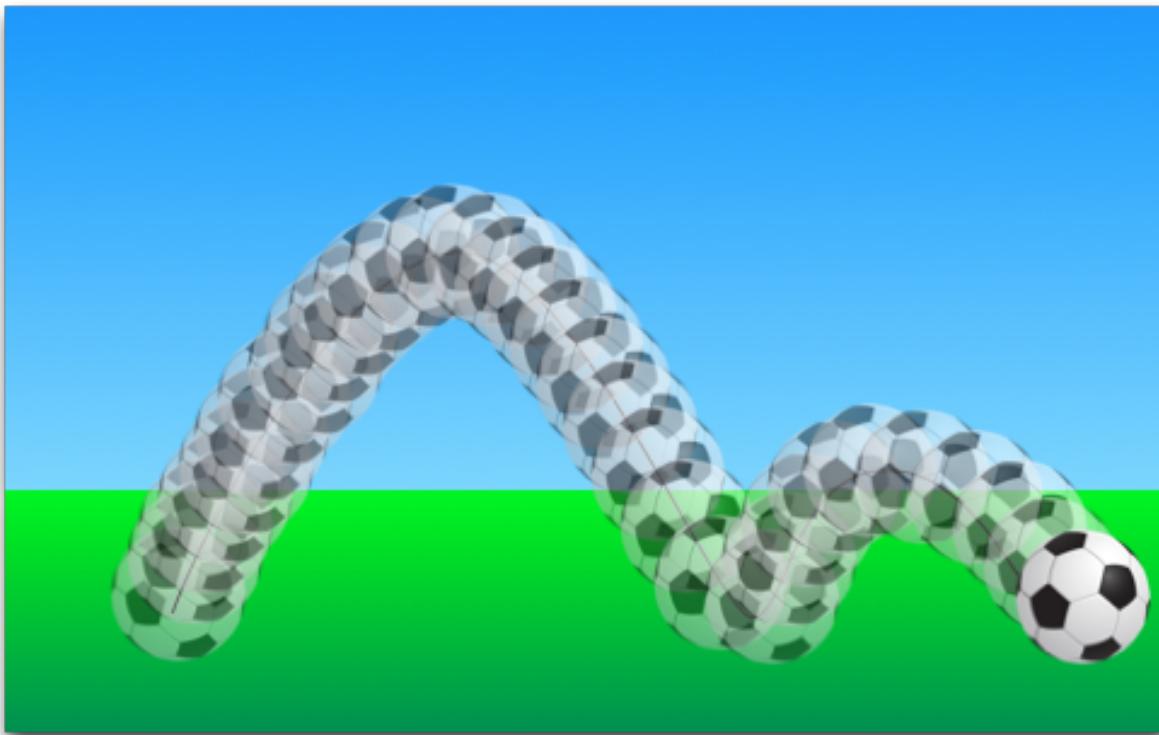
    ClickableImageV3 {
        id: rocket
        x: 20; y: 120
        source: "assets/rocket2."
        onClicked: anim.restart()
    }

    SequentialAnimation {
        id: anim
        NumberAnimation {
            target: rocket
            properties: "y"
            to: 20
            // 60% of time to trigger next animation
            duration: root.duration * 0.6
        }
        NumberAnimation {
            target: rocket
            properties: "x"
            to: 160
            // 40% of time to trigger next animation
            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

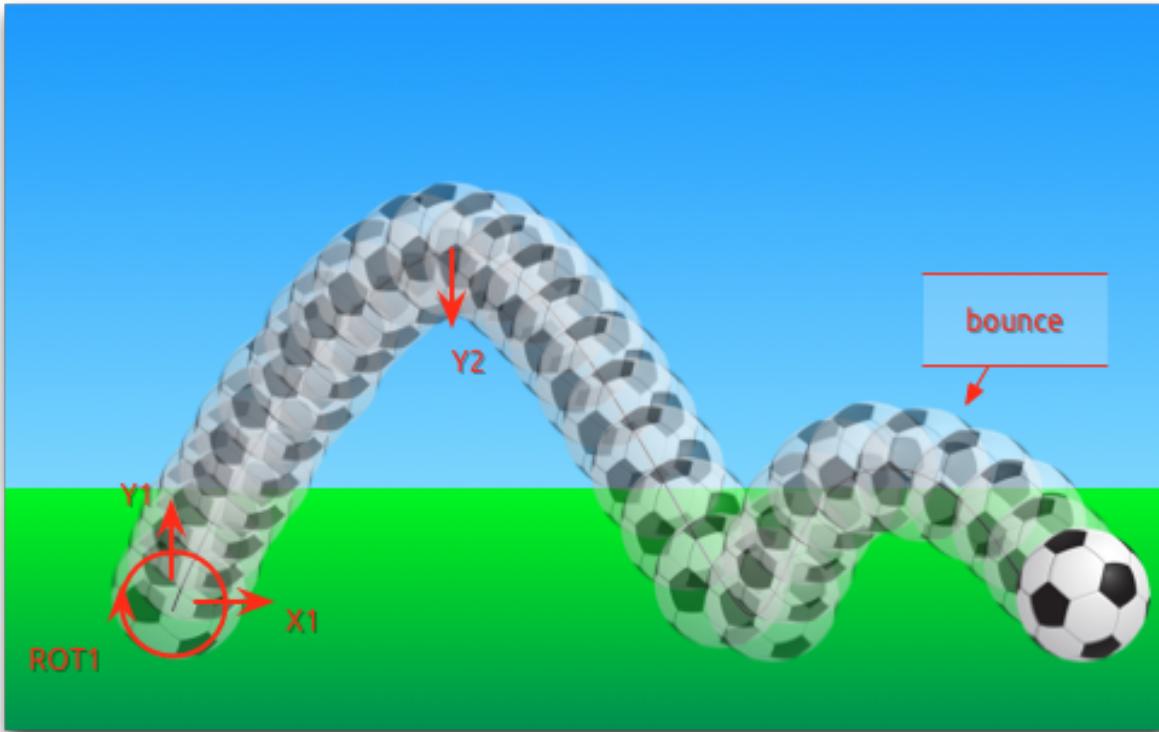
to right an animate its behavior.



To understand the animation we need to dissect it into the integral transformations of the object. We need to remember animation do 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 1.1

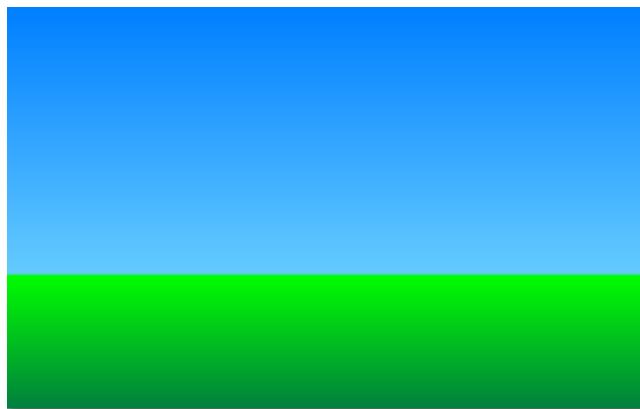
Item {
    id: root
    width: 480
    height: 300
    property int duration: 3000

    ...
}
```

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.

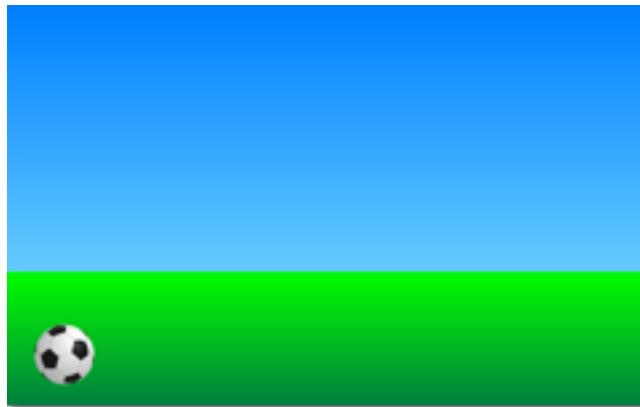
```
Image {
    id: ball
    x: 20; y: 240
    source: "assets/soccer_ball.png"

    MouseArea {
        anchors.fill: parent
```

```

    onClicked: {
        ball.x = 20; ball.y = 240
        anim.restart()
    }
}
}

```



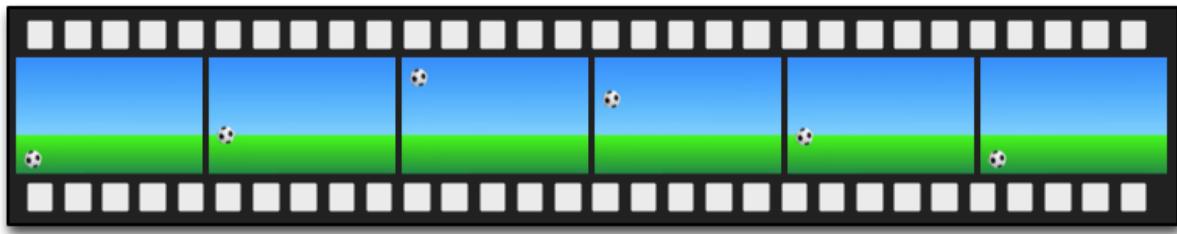
The image has a mouse area attached to it. If the ball is clicked the position of the ball will reset and the animation restarted.

Let's start with an sequential animation for the two y translation 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 gives during 40% of the total animation duration the up animation and for 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 concentrate on getting the transformations animated.

Now 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.

```

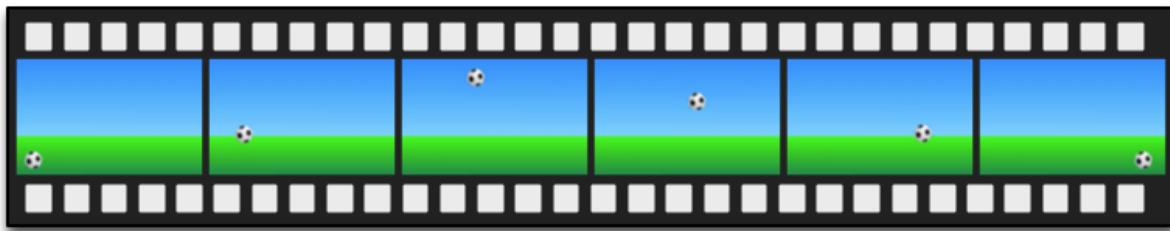
ParallelAnimation {
    id: anim
    SequentialAnimation {
        // ... our Y1, Y2 animation
    }
}

```

```

    }
    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 shall look more like a circular movement. *Y2* is enhanced using a `Easing.OutBounce` as the ball shall bounce and the bouncing shall 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
            properties: "y"
            to: 240
            duration: root.duration * 0.6
            easing.type: Easing.OutBounce
        }
    }
    NumberAnimation {

```

```
        target: ball
        properties: "x"
        to: 400
        duration: root.duration
    }
    RotationAnimation {
        target: ball
        properties: "rotation"
        to: 720
        duration: root.duration * 1.1
    }
}
```

## 1.5.2 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. Additional 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 consist in its simplest form of a series of property changes 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.

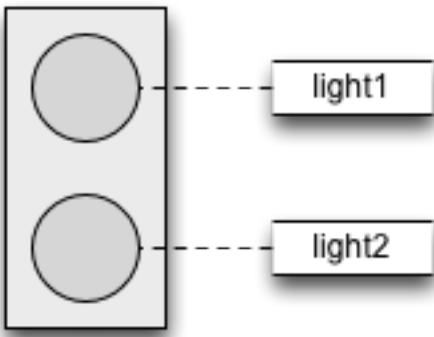
---

**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.

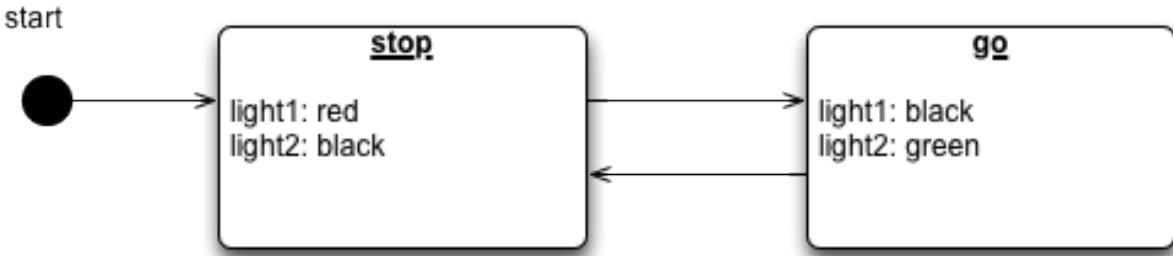
---

```
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 trigger now 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: "black"
}

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

```

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: [

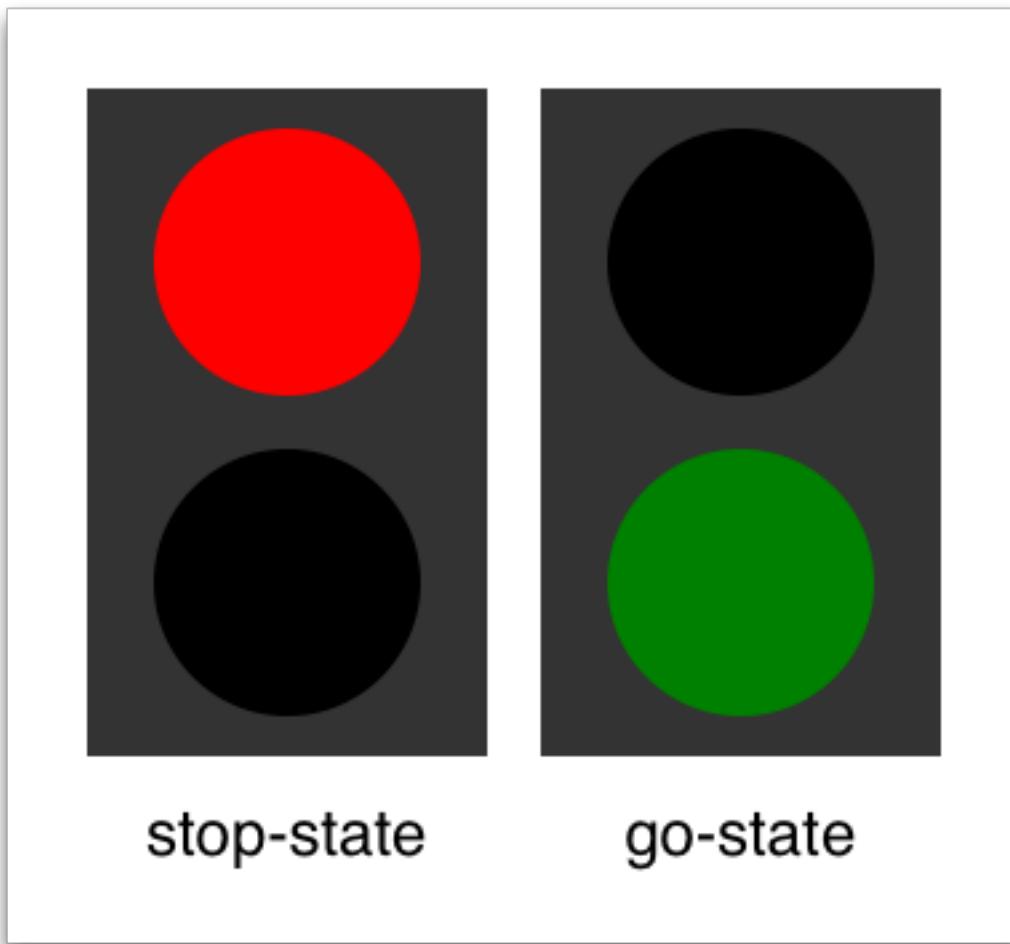
```

```
State {
    name: "stop"
    PropertyChanges { target: light1; color: "red" }
    PropertyChanges { target: light2; color: "black" }
},
State {
    name: "go"
    PropertyChanges { target: light1; color: "black" }
    PropertyChanges { target: light2; color: "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.

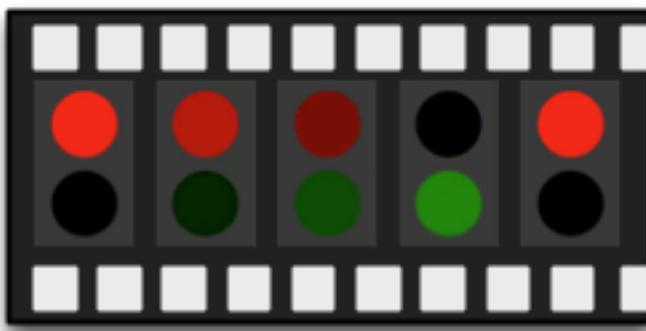
## Transitions

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-cast "\*" 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"
        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 for example by 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 a "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 also want to change the easing curve to make it more appealing.

### 1.5.3 Advanced Techniques

---

#### Todo

To be written

---

## 1.6 Model-View-Delegate

*Section author: e8johan*

---

**Note:** Last Build: January 20, 2014 at 18:00 CET

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

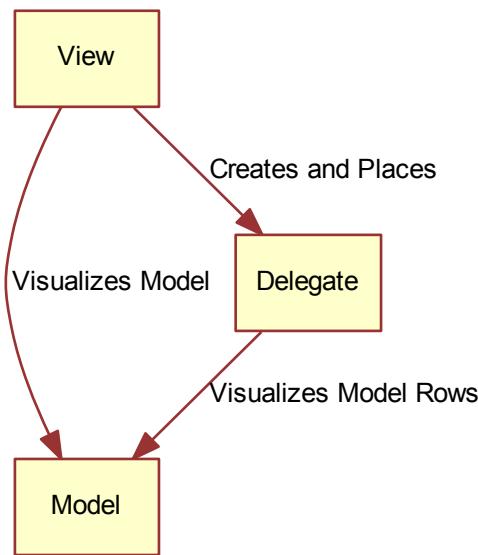
---

In QtQuick, 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. QtQuick 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.

### 1.6.1 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 responsibilities 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.



### 1.6.2 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 the most basic 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 2.0

Column {
    spacing: 2

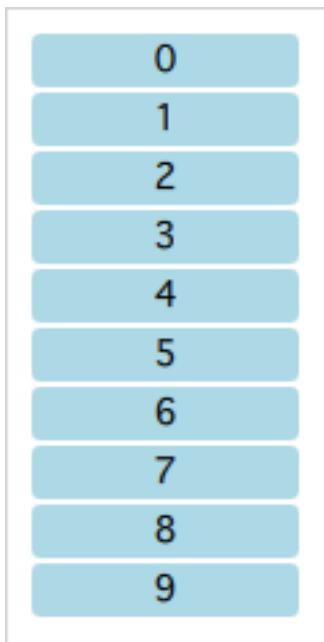
    Repeater {
        model: 10

        Rectangle {
            width: 100
            height: 20

            radius: 3

            color: "lightBlue"

            Text {
                anchors.centerIn: parent
                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 2.0

Column {
    spacing: 2

    Repeater {
        model: ["Enterprise", "Colombia", "Challenger", "Discovery", "Endeavour", "Atlantis"]
```

```
    Rectangle {
        width: 100
        height: 20

        radius: 3

        color: "lightBlue"

        Text {
            anchors.centerIn: parent
            text: index +": "+modelData
        }
    }
}
```



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 an image URL 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 2.0

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" }
        }

        Rectangle {
            width: 100
            height: 20
        }
    }
}
```

```
    radius: 3

    color: "lightBlue"

    Text {
        anchors.centerIn: parent
        text: name
    }

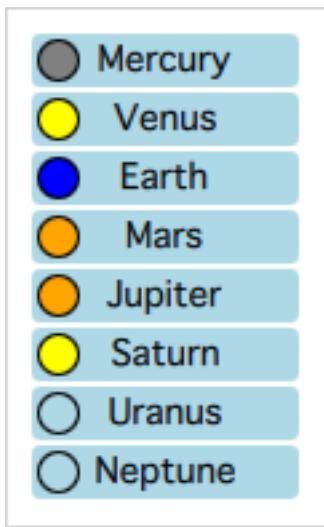
    Rectangle {
        anchors.left: parent.left
        anchors.verticalCenter: parent.verticalCenter
        anchors.leftMargin: 2

        width: 16
        height: 16

        radius: 8

        border.color: "black"
        border.width: 1

        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 2.0

Column {
    spacing: 2

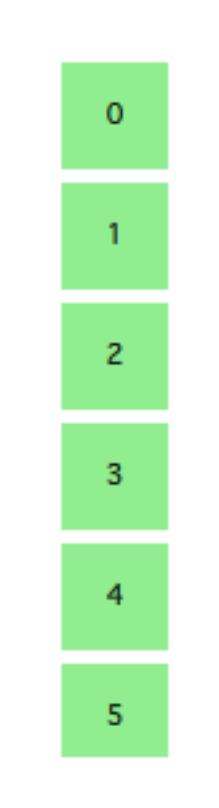
    Repeater {
        model: 10

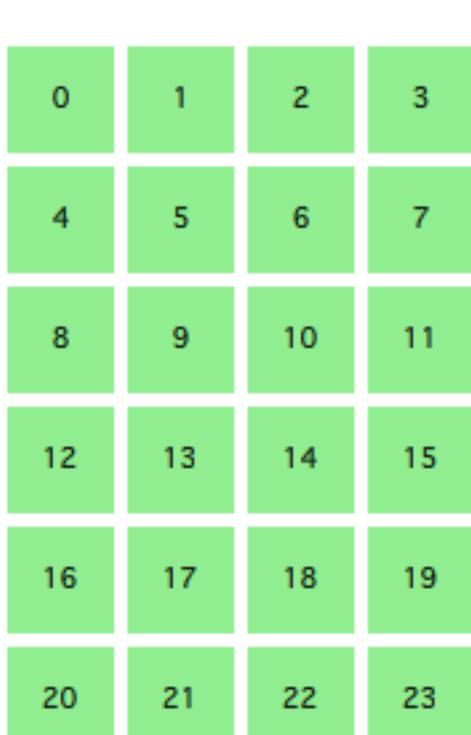
        delegate: Rectangle {
            width: 100
            height: 20
        }
    }
}
```

```
    radius: 3
    color: "lightBlue"
    Text {
        anchors.centerIn: parent
        text: index
    }
}
}
```

### 1.6.3 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, QtQuick 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.

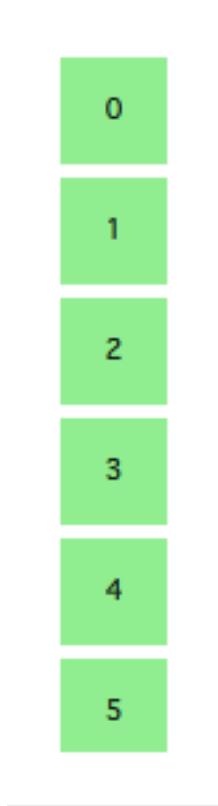
```
import QtQuick 2.0

Rectangle {
    width: 80
    height: 300
    color: "white"

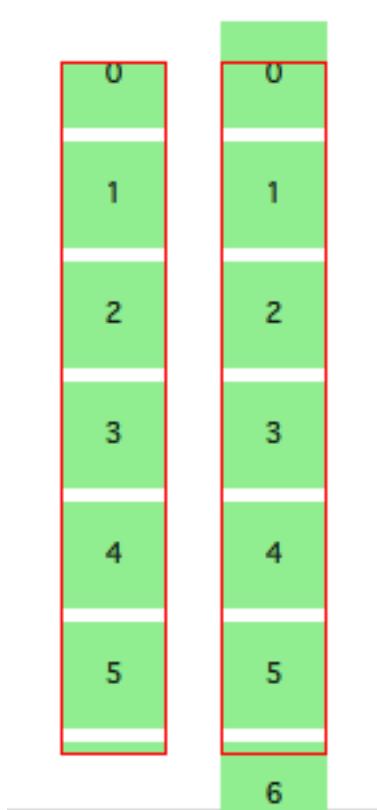
    ListView {
        anchors.fill: parent
        anchors.margins: 20
        clip: true
        model: 100
        delegate: numberDelegate
        spacing: 5
    }

    Component {
        id: numberDelegate
        Rectangle {
            width: 40
            height: 40
            color: "lightGreen"
        }
    }
}
```

```
Text {
    anchors.centerIn: parent
    font.pixelSize: 10
    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 QtQuick, 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 2.0

Rectangle {
    width: 480
    height: 80
    color: "white"

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

```
clip: true

model: 100

orientation: ListView.Horizontal

delegate: numberDelegate
spacing: 5
}

Component {
    id: numberDelegate

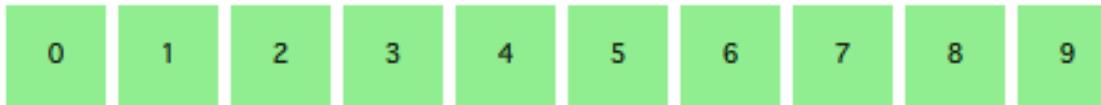
    Rectangle {
        width: 40
        height: 40

        color: "lightGreen"

        Text {
            anchors.centerIn: parent

            font.pixelSize: 10

            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 2.0
```

```
Rectangle {
```

```
width: 240
height: 300

color: "white"

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

    clip: true

    model: 100

    delegate: numberDelegate
    spacing: 5

    highlight: highlightComponent
    focus: true
}

Component {
    id: highlightComponent

    Rectangle {
        width: ListView.view.width
        color: "lightGreen"
    }
}

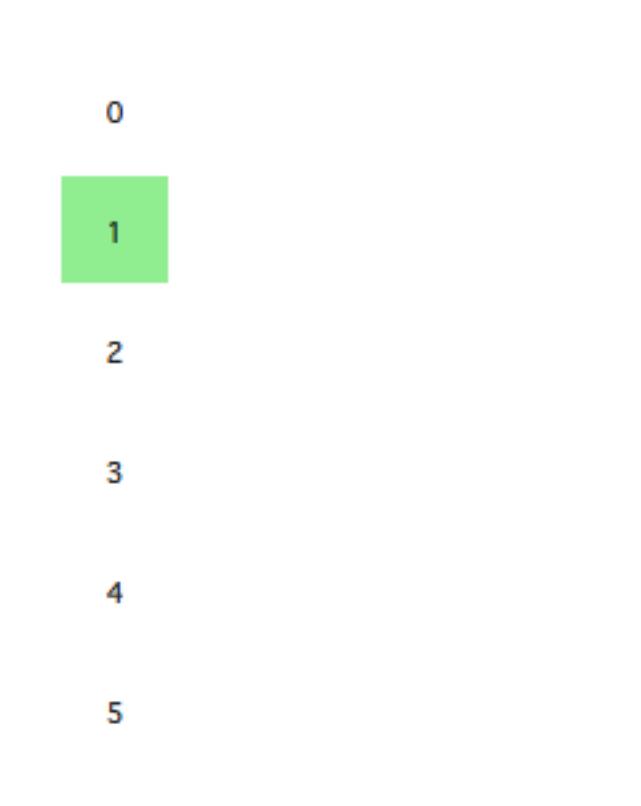
Component {
    id: numberDelegate

    Item {
        width: 40
        height: 40

        Text {
            anchors.centerIn: parent

            font.pixelSize: 10

            text: index
        }
    }
}
// M1>>
```



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 are not related at all.

The value `ListView.StrictlyEnforceRange` ensures that the highlight always is 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: 100 }
        NumberAnimation { duration: 1 }
        PropertyAnimation { target: highlightRectangle; property: "opacity"; to: 1; duration: 100 }
    }
}

Rectangle {
    id: highlightRectangle
    anchors.fill: parent
    color: "lightGreen"
}
}
}
}

```

## 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 2.0

Rectangle {
    width: 80
    height: 300

    color: "white"

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

        clip: true

        model: 4

        delegate: numberDelegate
        spacing: 5

        header: headerComponent
        footer: footerComponent
    }

    Component {
        id: headerComponent

        Rectangle {
            width: 40
            height: 20

            color: "yellow"
        }
    }
}

```

```
}

Component {
    id: footerComponent

    Rectangle {
        width: 40
        height: 20

        color: "red"
    }
}

Component {
    id: numberDelegate

    Rectangle {
        width: 40
        height: 40

        color: "lightGreen"

        Text {
            anchors.centerIn: parent

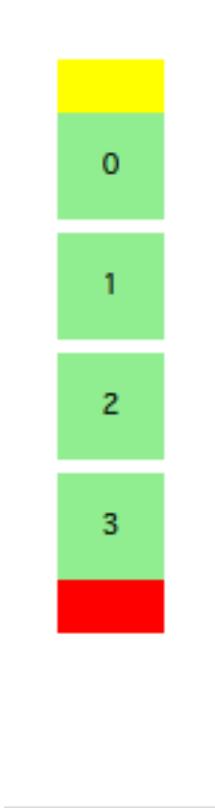
            font.pixelSize: 10

            text: index
        }
    }
}
```

---

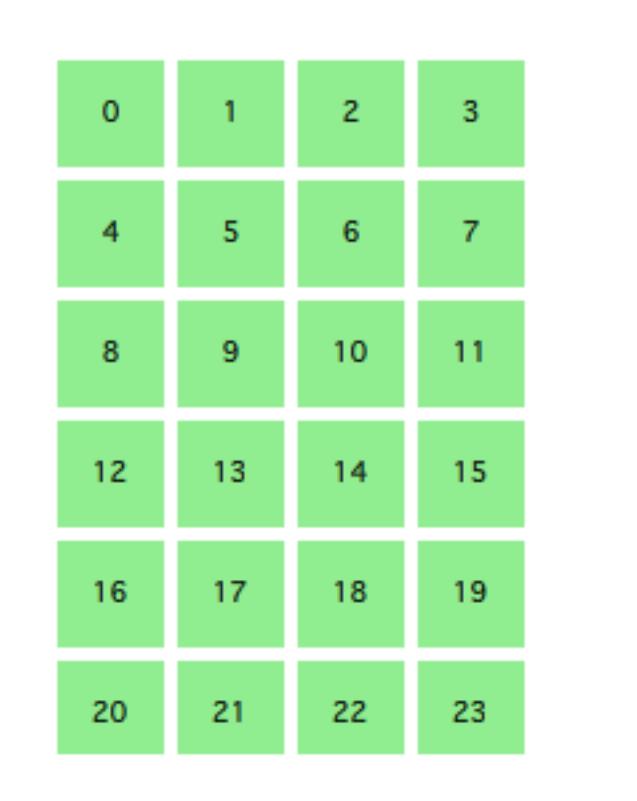
**Note:** Header and footer delegates do not respect the `spacing` property of a `ListView`, instead they are placed directly adjacent to the next item delegate in the list. This means that any spacing must be a part of the header and footer items.

---



### 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 2.0

Rectangle {
    width: 240
    height: 300

    color: "white"

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

        clip: true

        model: 100

        cellWidth: 45
        cellHeight: 45

        delegate: numberDelegate
    }

    Component {
        id: numberDelegate

        Rectangle {
            width: 40
            height: 40

            color: "lightGreen"

            Text {
                anchors.centerIn: parent

                font.pixelSize: 10

                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.

## 1.6.4 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 2.0

Rectangle {
    width: 120
    height: 300
    color: "white"

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

        clip: true

        model: 100

        delegate: numberDelegate
        spacing: 5

        focus: true
    }

    Component {
        id: numberDelegate

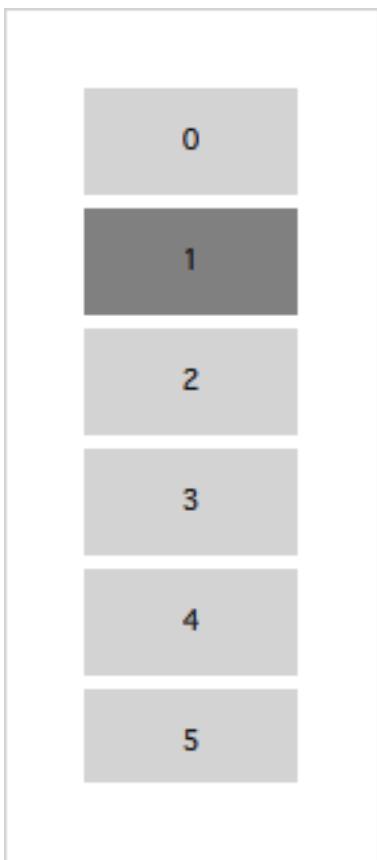
        Rectangle {
            width: ListView.view.width
            height: 40

            color: ListView.isCurrentItem?"gray":"lightGray"

            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. 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 2.0
```

```

Rectangle {
    width: 480
    height: 300

    color: "white"

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: "darkGreen"

    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
        color: "lightGreen"

        Text {
            anchors.centerIn: parent
            font.pixelSize: 10
            text: number
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                if (!wrapper.GridView.delayRemove)
                    theModel.remove(index);
            }
        }
    }

    GridView.onRemove: SequentialAnimation {
        PropertyAction { target: wrapper; property: "GridView.delayRemove"; value: true }
        NumberAnimation { target: wrapper; property: "scale"; to: 0; duration: 250; easing: OutQuart }
        PropertyAction { target: wrapper; property: "GridView.delayRemove"; value: false }
    }

    GridView.onAdd: SequentialAnimation {
        NumberAnimation { target: wrapper; property: "scale"; from: 0; to: 1; duration: 250 }
    }
}
}
```

## 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 2.0

Item {
    width: 300
    height: 480

    ListView {
        id: listView

        anchors.fill: parent

        delegate: detailsDelegate
        model: planets
    }

    ListModel {
        id: planets

        ListElement { name: "Mercury"; imageSource: "images/mercury.jpeg"; facts: "Mercury is the t..."}
        ListElement { name: "Venus"; imageSource: "images/venus.jpeg"; facts: "Venus is the second..."}
        ListElement { name: "Earth"; imageSource: "images/earth.jpeg"; facts: "The Earth is the th..."}
        ListElement { name: "Mars"; imageSource: "images/mars.jpeg"; facts: "Mars is the fourth p..."}
    }

    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: "#ffaa00"

                Text {
                    anchors.left: parent.left
                    anchors.verticalCenter: parent.verticalCenter

                    font.pixelSize: parent.height-4

                    text: name
                }
            }

            Rectangle {
                id: image

                color: "black"

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

```

```
    width: 26
    height: 26

    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
        color: "#cccccc"

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

            clip: true
            wrapMode: Text.WordWrap

            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: "red"

    opacity: 0

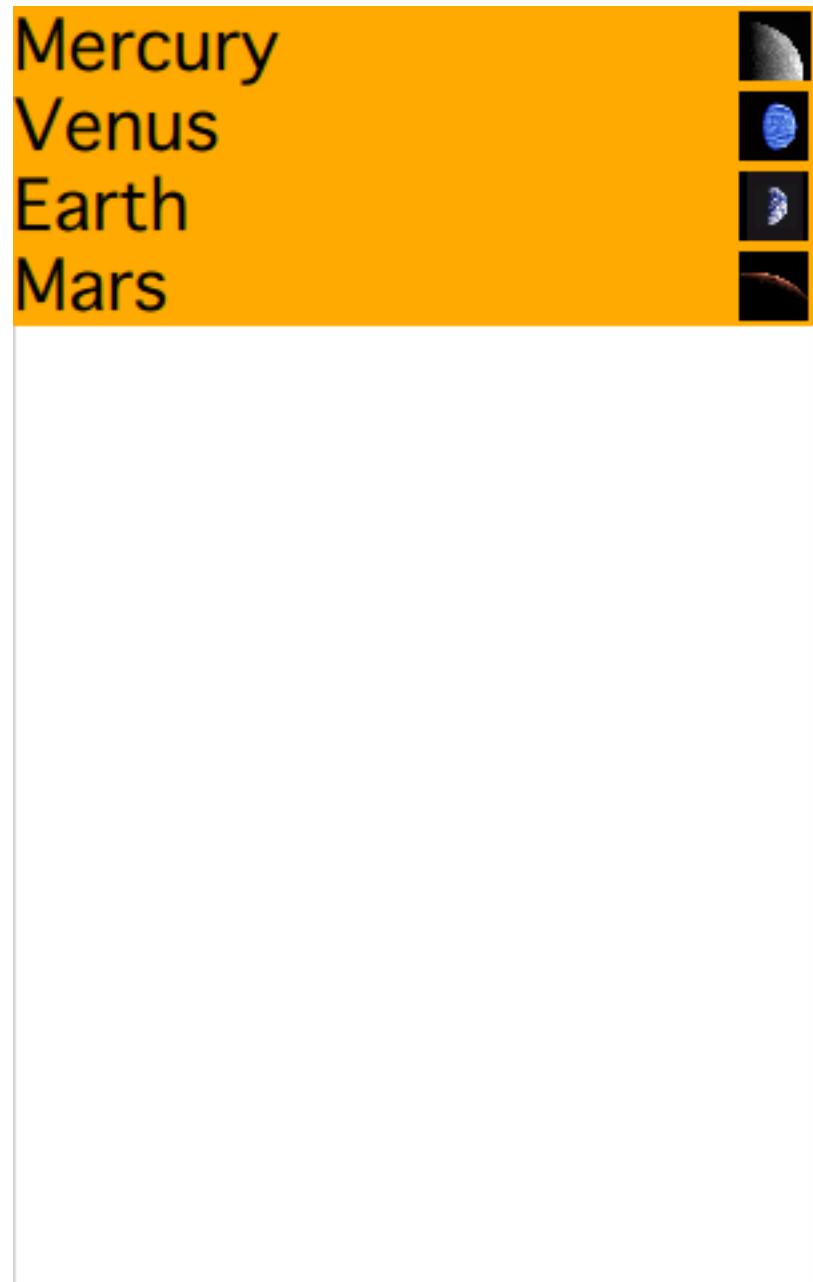
    MouseArea {
```

```
        anchors.fill: parent
        onClicked: wrapper.state = ""
    }
}

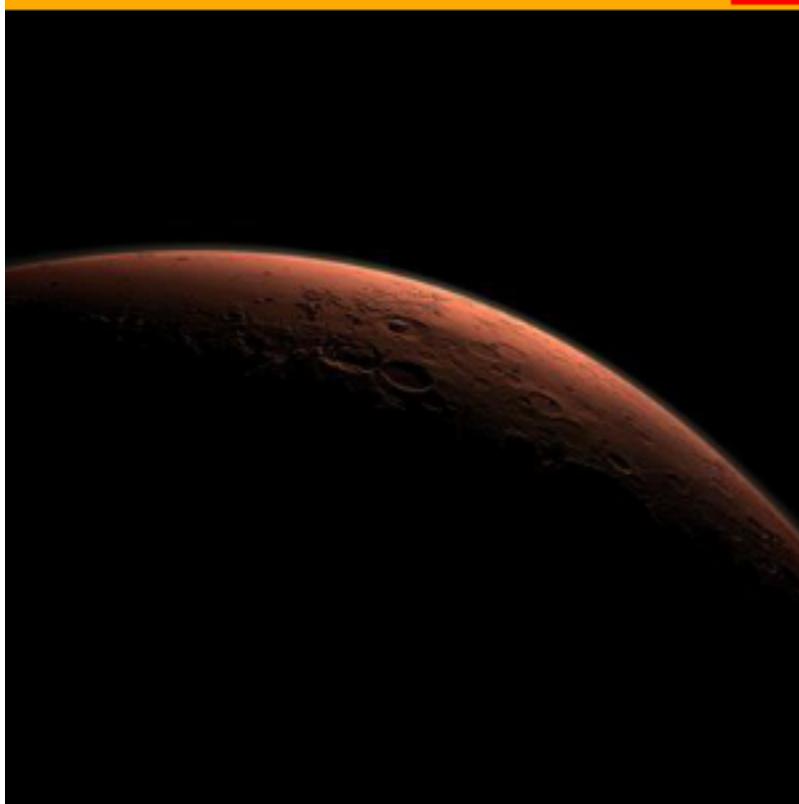
states: [
    State {
        name: "expanded"

        PropertyChanges { target: wrapper; height: listView.height }
        PropertyChanges { target: image; width: listView.width; height: listView.width }
        PropertyChanges { target: factsView; opacity: 1 }
        PropertyChanges { target: closeButton; opacity: 1 }
        PropertyChanges { target: wrapper.ListView.view; contentY: wrapper.y; interact: true }
    }
]

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



# Mars



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.

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.

## 1.6.5 Advanced Techniques

### The PathView

The PathView element is the most powerful, but also the most complex, view provided in QtQuick. 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

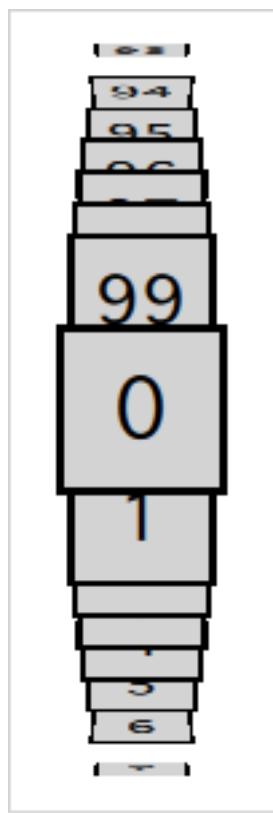
do we cover the line, quad and cubic through an illustration, or do we need a paragraph on them?

---

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

    Item {
        id: wrapper

        width: 64
        height: 64

        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; z: 0 } }

        Rectangle {
            anchors.fill: parent
            color: "lightGray"
        }
    }
}

```

```
        border.color: "black"
        border.width: 3
    }

Text {
    anchors.centerIn: parent
    text: index
    font.pixelSize: 30
}
}
```

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.

**Landscape, Latvia****Festival Performers, Japan**

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 2.0
import QtQuick.XmlListModel 2.0

Item {
    width: 300
```

```
height: 480

Component {
    id: imageDelegate

    Item {
        width: listView.width
        height: 400

        Column {
            Text {
                text: title
            }

            Image {
                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(), "
    }

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.

**Afghanistan**

Abdul Ahad Mohmand

**Brazil**

Marcos Pontes

**Bulgaria**

Alexandar Panayotov Alexandrov

Georgi Ivanov

**Canada**

Roberta Bondar

Marc Garneau

Chris Hadfield

Guy Laliberte

Stéphanie Dion

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 2.0

Rectangle {
    width: 300
    height: 290

    color: "white"

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

```
clip: true

model: spaceMen

delegate: spaceManDelegate

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

Component {
    id: spaceManDelegate

    Item {
        width: 260
        height: 20

        Text {
            anchors.left: parent.left
            anchors.verticalCenter: parent.verticalCenter
            anchors.leftMargin: 10

            font.pixelSize: 12

            text: name
        }
    }
}

Component {
    id: sectionDelegate

    Rectangle {
        width: 260
        height: 20

        color: "lightGray"

        Text {
            anchors.left: parent.left
            anchors.verticalCenter: parent.verticalCenter
            anchors.leftMargin: 10

            font.pixelSize: 12
            font.bold: true

            text: section
        }
    }
}

ListModel {
    id: spaceMen

    ListElement { name: "Abdul Ahad Mohmand"; nation: "Afghanistan"; }
    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 for the user as soon as the view is scrolled too quickly.

To work around this issue you can tune the, 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.

### 1.6.6 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.

## 1.7 Canvas Element

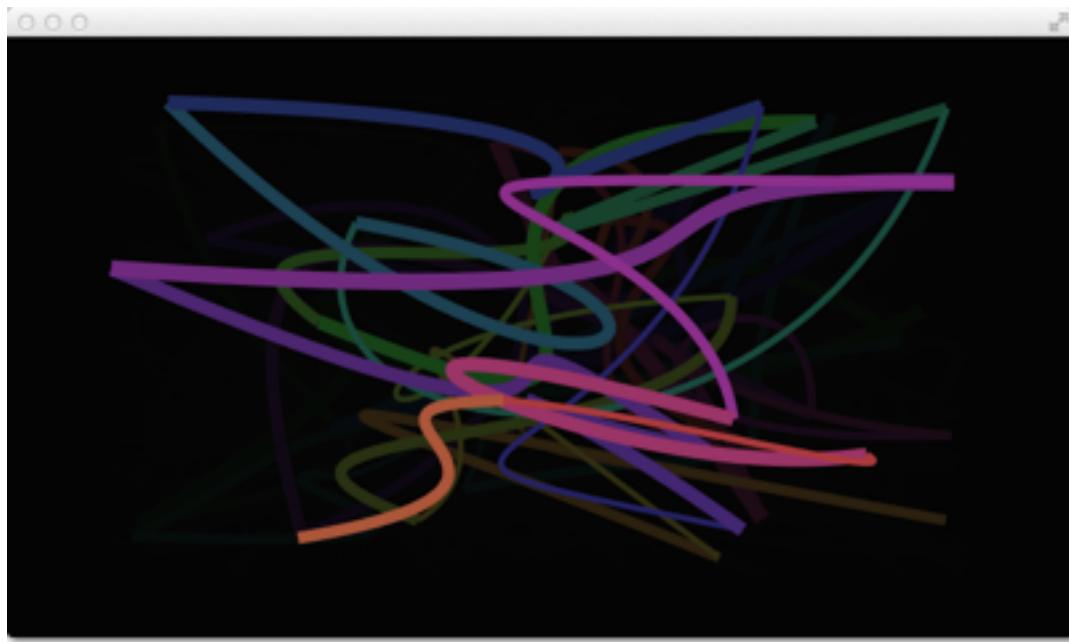
*Section author: jryannel*

---

**Note:** Last Build: January 20, 2014 at 18:00 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 QtQuick 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 Qt5 introduces the canvas element. The canvas elements 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 2.0

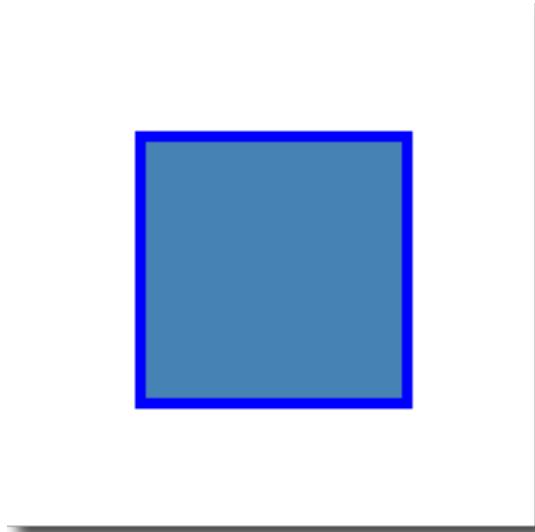
Canvas {
    id: root
    // canvas size
    width: 200; height: 200
    // handler to override for drawing
    onPaint: {
        // get context to draw with
        var ctx = getContext("2d")
        // setup the stroke
        ctx.lineWidth = 4
        ctx.strokeStyle = "blue"
        // setup the 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)
        // close the path
        ctx.closePath()
        // fill the path
        ctx.fill()
    }
}
```

```

    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 hight 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.

---

### 1.7.1 Convenient API

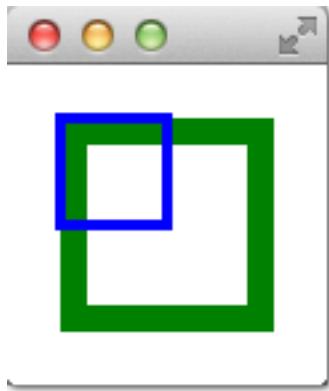
For operations on rectangles a convenience API is provided which draws directly and does not need a stroke or fill call.

```
// convenient.qml

import QtQuick 2.0

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.

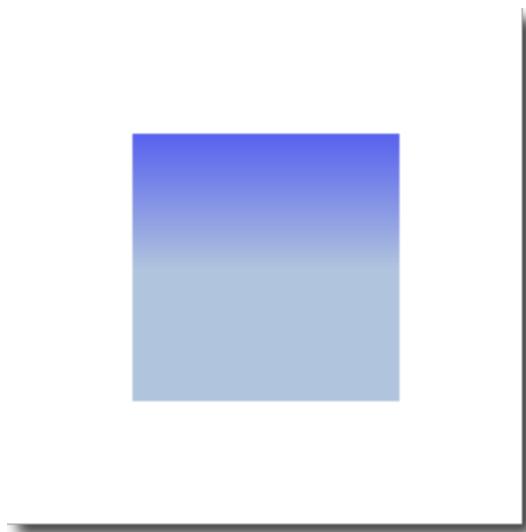
## 1.7.2 Gradients

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 define 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 then the rectangle we want to draw, so the rectangle clips gradient to it’s 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.

### 1.7.3 Shadows

---

**Note:** We had difficulties with shadows in the Qt5 alpha release.

---

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 to 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 “Earth” 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:

```
ctx.shadowColor = "blue";
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
// next line crashes
// ctx.shadowBlur = 10;
```

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

```
ctx.font = 'Bold 80px Ubuntu';
ctx.fillStyle = "#33a9ff";
ctx.fillText("Earth",30,180);
```

---

#### Todo

show screen-shot from example, when shadows will work on Qt5

---

### 1.7.4 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 = 'red'
    // create a triangle as clip region
    ctx.beginPath()
    ctx.moveTo(10,10)
    ctx.lineTo(55,10)
    ctx.lineTo(35,55)
    ctx.closePath()
    // translate coordinate system
    ctx.translate(100,0)
    ctx.clip() // create clip from triangle path
    // draw image with clip applied
```

```

    ctx.drawImage('assets/ball.png', 10, 10)
    // draw stroke around path
    ctx.stroke()
    // restore previous setup
    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.



### 1.7.5 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 its 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 2.0

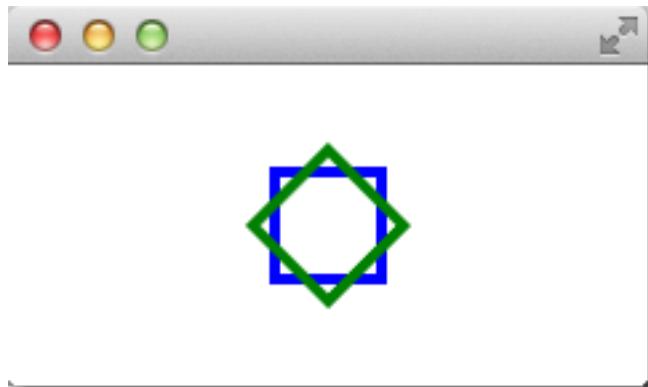
Canvas {
    id: root
    width: 240; height: 120
    onPaint: {
        var ctx = getContext("2d")
        ctx.strokeStyle = "blue"
        ctx.lineWidth = 4

        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)`.

**Warning:** The transformation seems to behave different than the the HTML5 canvas. Not sure if this is a bug.

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

```
ctx.resetTransform()
```

## 1.7.6 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()
    }
}

```

### 1.7.7 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 2.0

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.

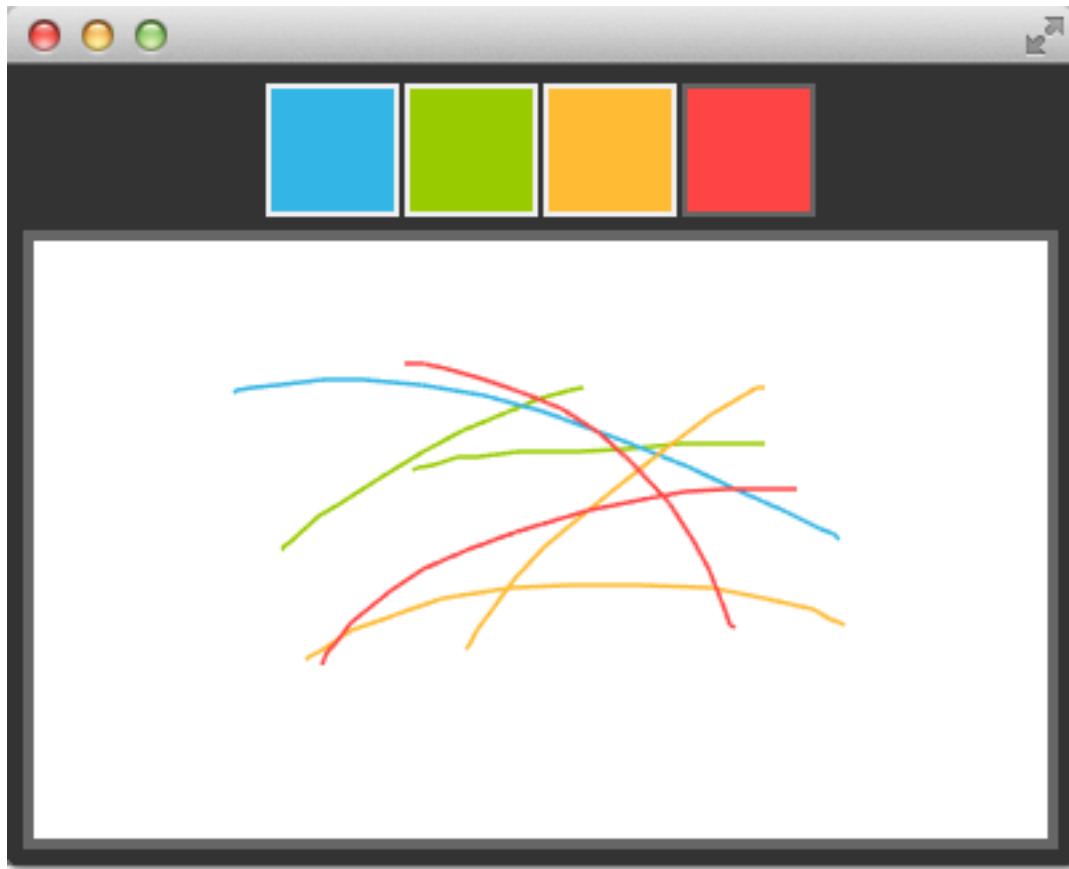
---

**Note:** Retrieving image data seems not to work currently in the Qt5 Alpha SDK.

---

## 1.7.8 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 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.

### 1.7.9 Porting from HTML5 Canvas

- [https://developer.mozilla.org/en/Canvas\\_tutorial/Transformations](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) {
    ...
}
```

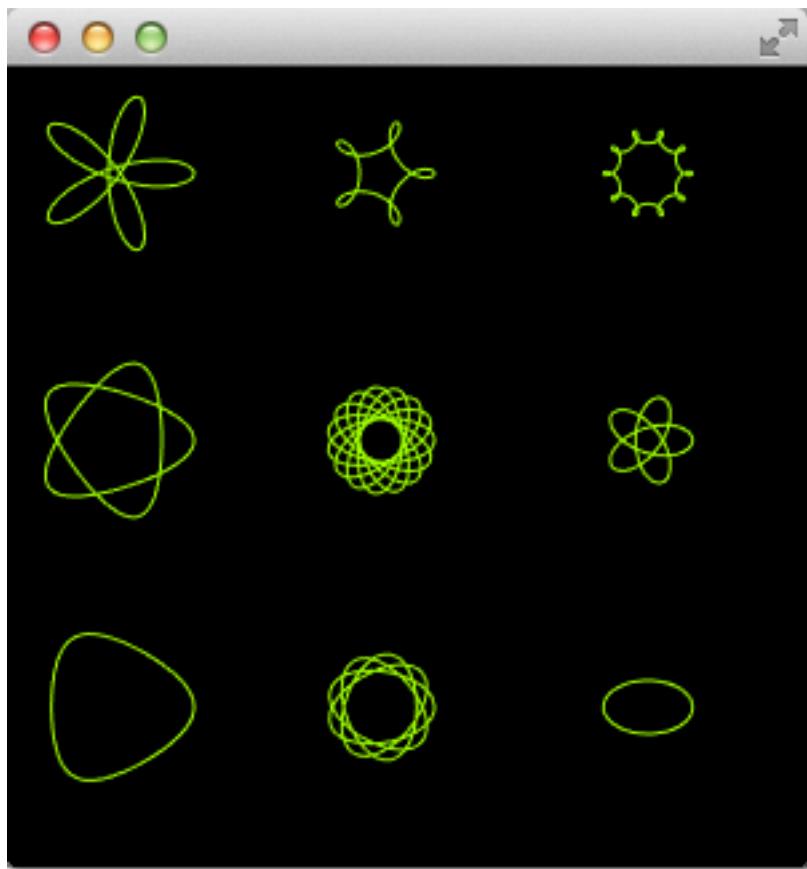
- we needed to adapt the translation for each spiro due to different sizes

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

Finally we implemented 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();

    context.bezierCurveTo(
        lastX + 100 * Math.random(),
        lastY + 100 * Math.random(),
        lastX + 200 * Math.random(),
        lastY + 200 * Math.random(),
        lastX + 300 * Math.random(),
        lastY + 300 * 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 2.0

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 `requestLine` 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 gor 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 (without the shadows currently) will look similar to this.




---

## Todo

There are currently heavy issues with the shadow implementation. This currently does not work

---

### Note: Adobe Illustrator

Adobe Illustrator has a plug-in to export the artwork as HTML5 canvas code.

- [Adobe Illustrator Plug-in to HTML5 Canvas](#)
- 

### See also:

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

## 1.8 Particle Simulations

*Section author: jryannel*

---

**Note:** Last Build: January 20, 2014 at 18:00 CET

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

---

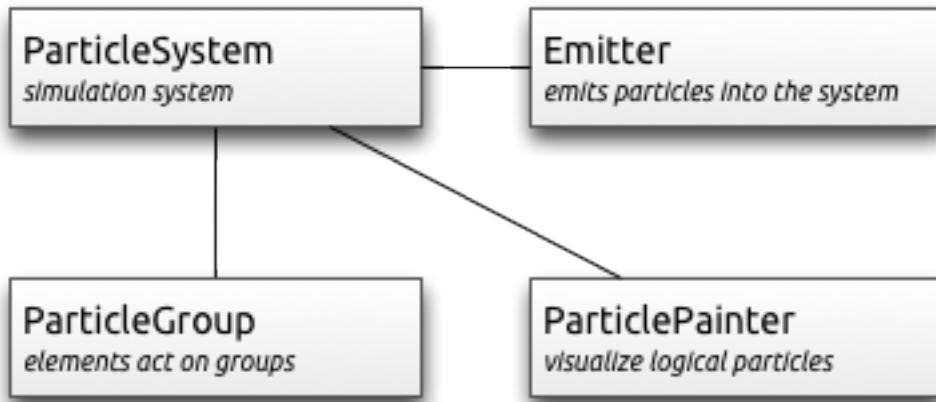
Particles are a computer graphics techniques to visualize certain graphics effects. Typical effects could be: falling leaves, fire, explosions, meteors, clouds, etc.

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.

### 1.8.1 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 it 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

### 1.8.2 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 2.0
import QtQuick.Particles 2.0

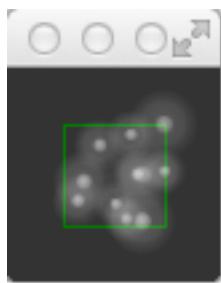
Rectangle {
    id: root
    width: 480; height: 160
    color: "#1f1f1f"

    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 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.

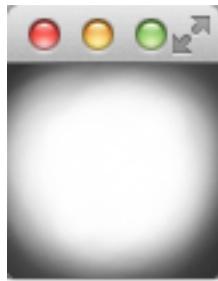
The emitter emits logical particles. A logical particle is visualized using a `ParticlePainter` in this example we use an `ImageParticle`, which takes an image URL as the source property. The image particle has also several other properties, which control the appearance of the average particle.

- `emitRate`: particles emitted per second (defaults to 10 per second)
- `lifeSpan`: milliseconds the particle should last for (defaults to 1000 msecs)
- `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.

### 1.8.3 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 sparkling star image:

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

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

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

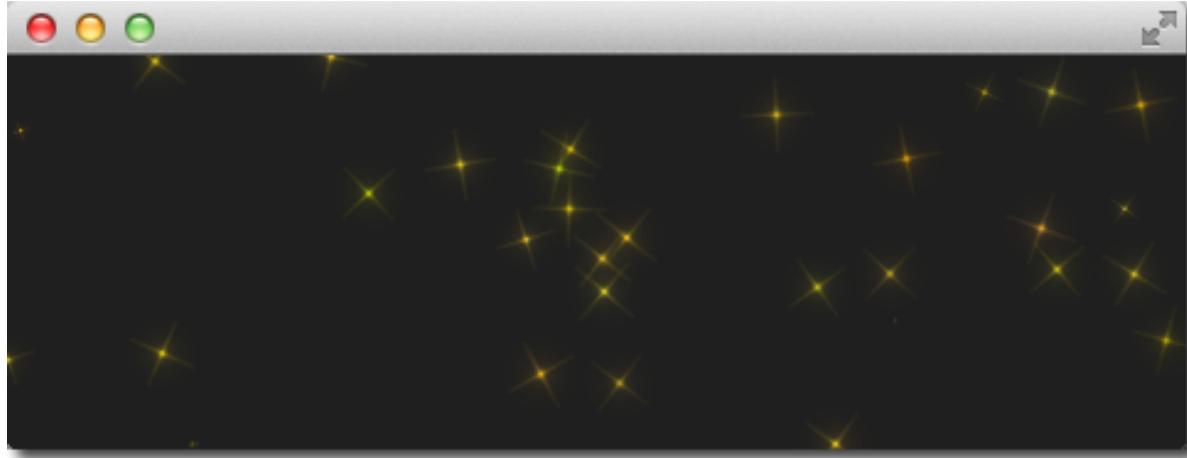
To make the scene more alive we would like to rotate the particles. Each particle should start by 15 degrees clockwise and varies between particles by +/- 5 degrees. Additional the particle should continuously rotate with the velocity of 45 degrees per second. The velocity shall also vary from particle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

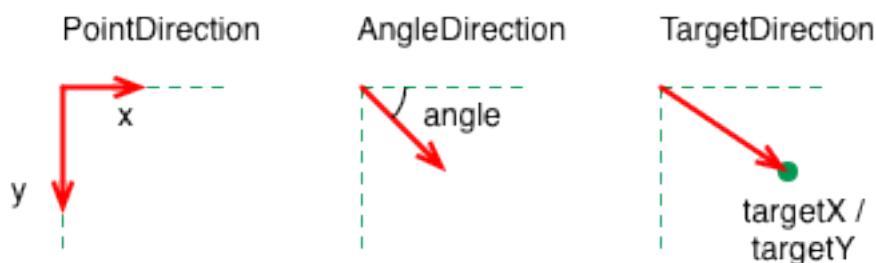
```
ImageParticle {
    source: "assets/star.png"
    system: particleSystem
    color: '#FFD700'
    colorVariation: 0.2
    rotation: 0
    rotationVariation: 45
    rotationVelocity: 15
    rotationVelocityVariation: 15
    entryEffect: ImageParticle.Scale
}
```

#### 1.8.4 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:

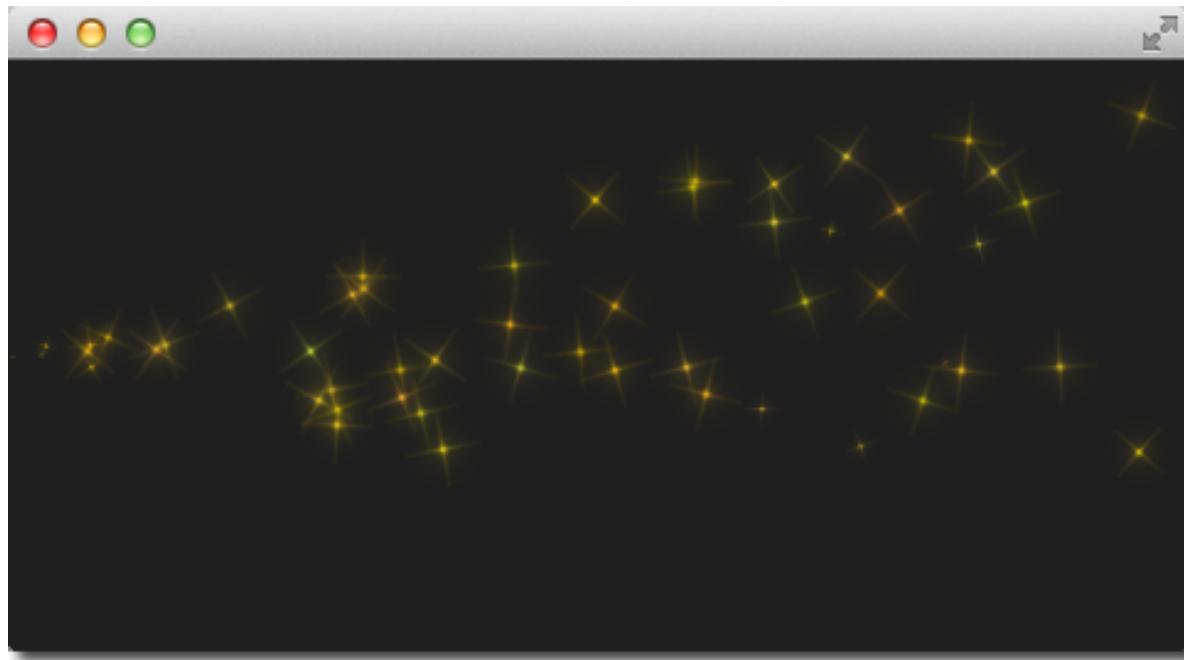
```
velocity: AngleDirection { }
```

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
}

```

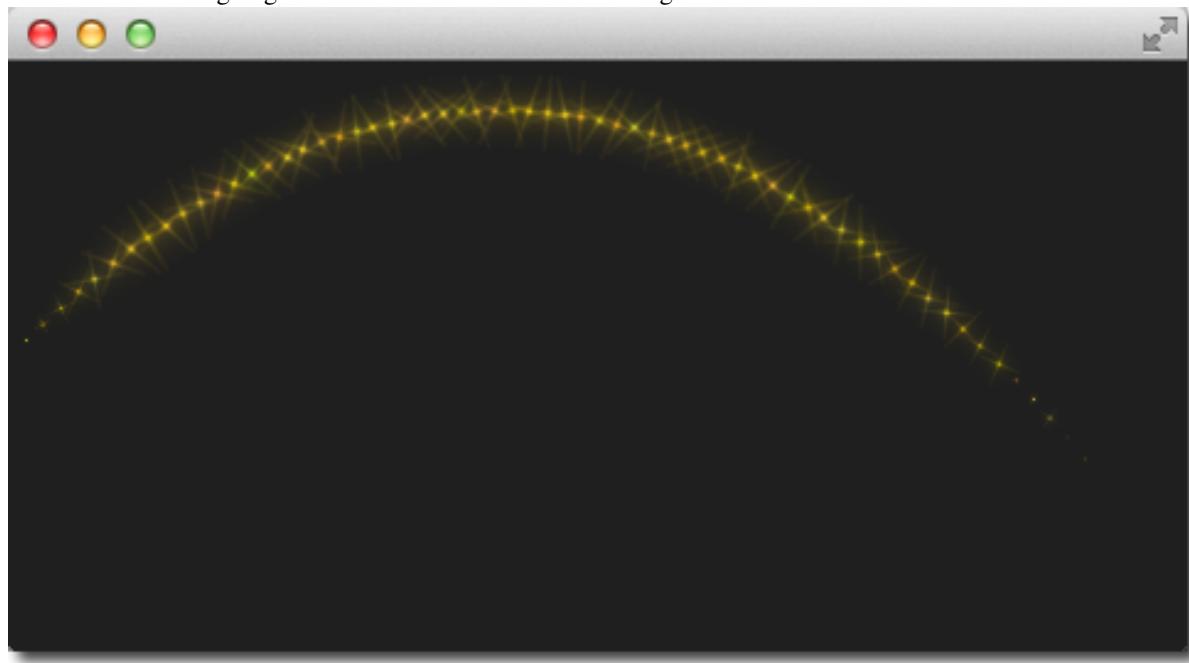
The acceleration direction shall be 90 degree (down direction) and we choose one fourth of the velocity magnitude for this:

```

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.

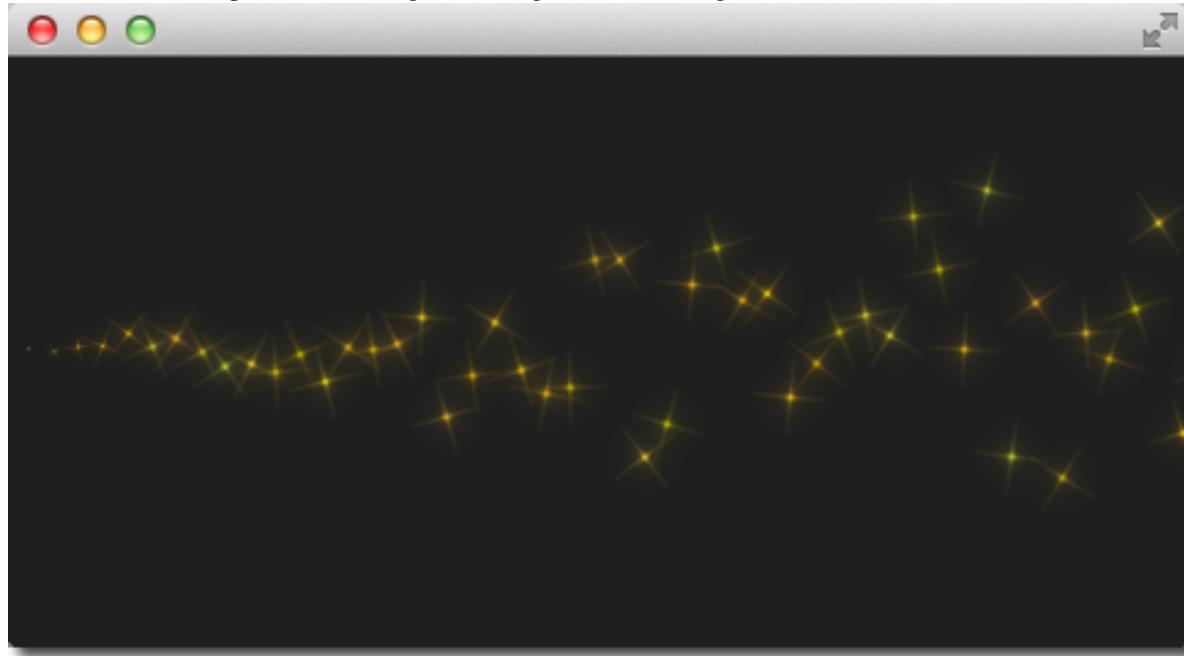
In our case we want the particles travel from left-to-right building a 15 degree cone. For this we specify a PointDirection as our velocity vector space:

```
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:

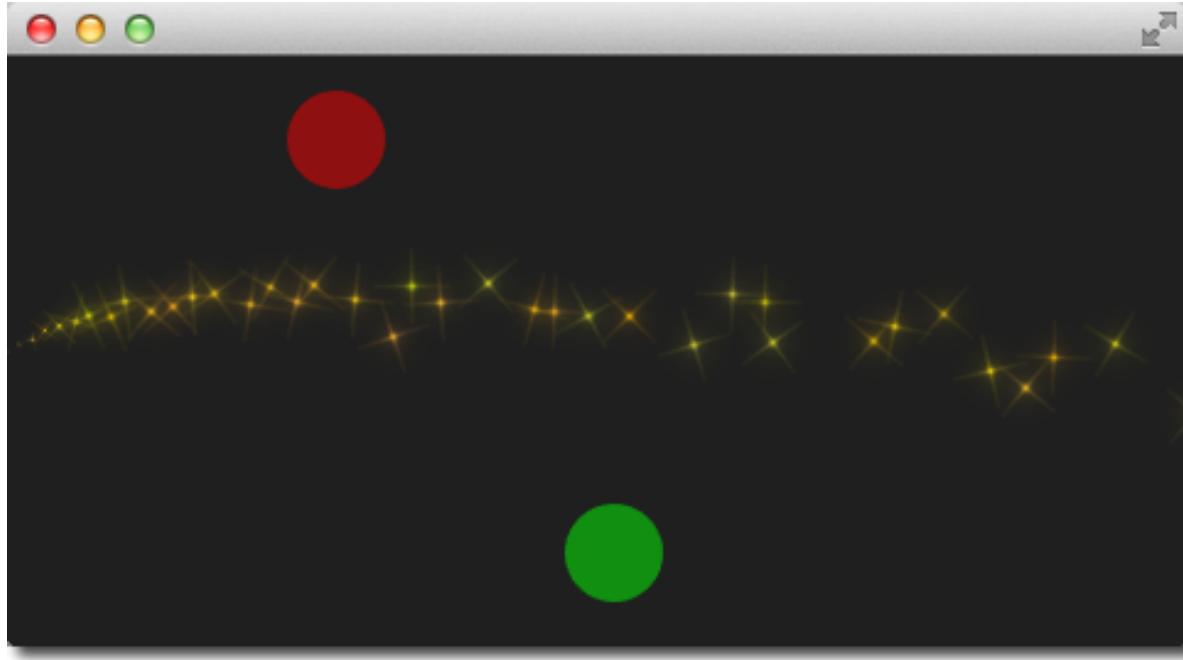
```
velocity: TargetDirection {
    targetX: 100
    targetY: 0
    targetVariation: 100/6
```

```
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?



## 1.8.5 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.

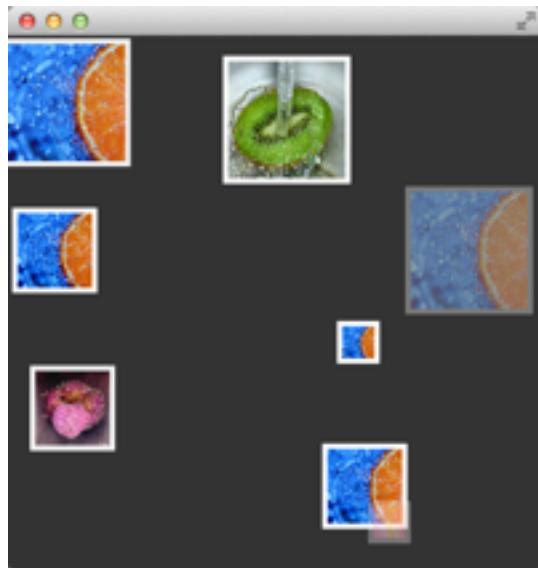
```
ItemParticle {
    id: particle
    system: particleSystem
    delegate: itemDelegate
}
```

Our delegate in this case is a random image (using `Math.random()`), visualized with a white border and a random size.

```
Component {
    id: itemDelegate
    Rectangle {
        id: container
        width: 32*Math.ceil(Math.random()*3); height: width
        color: 'white'
```

```
Image {
    anchors.fill: parent
    anchors.margins: 4
    source: 'assets/fruits'+Math.ceil(Math.random()*10)+'.jpg'
}
}
```

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)`.

## 1.8.6 Affecting Particles

Particles are emitted by the emitter. After a particle was emitted it can't be changed anymore by the emitter. The affectors 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

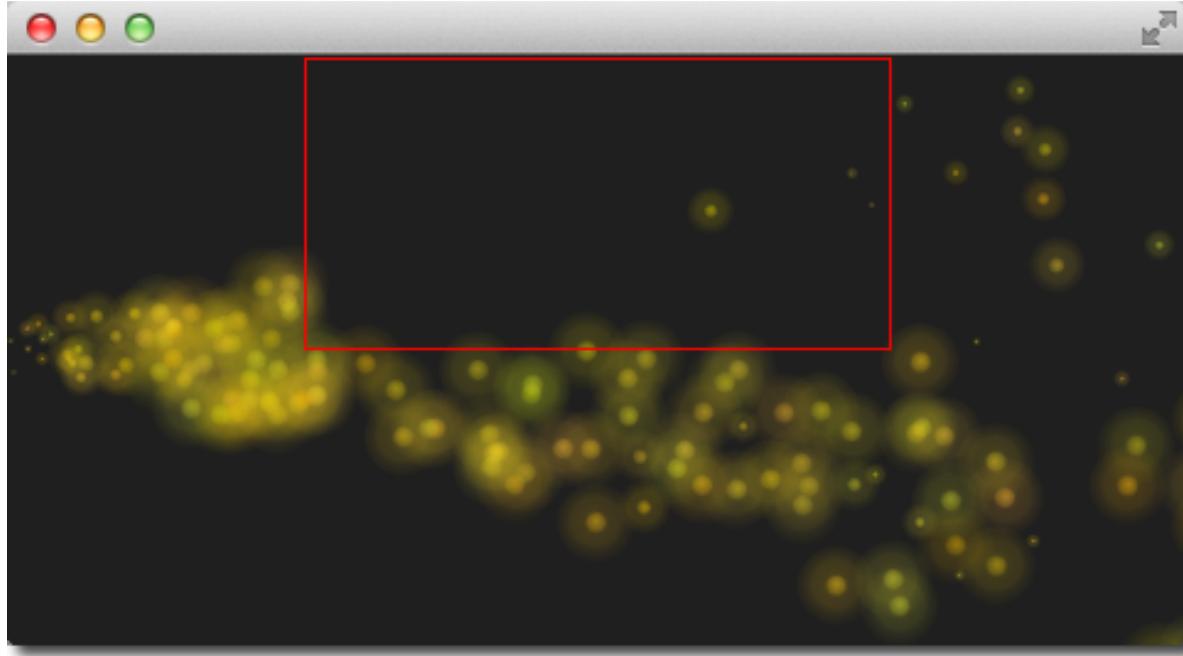
Allows particle to age faster. the `lifeLeft` property specified how much life a particle should have left.

```

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 affecter 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

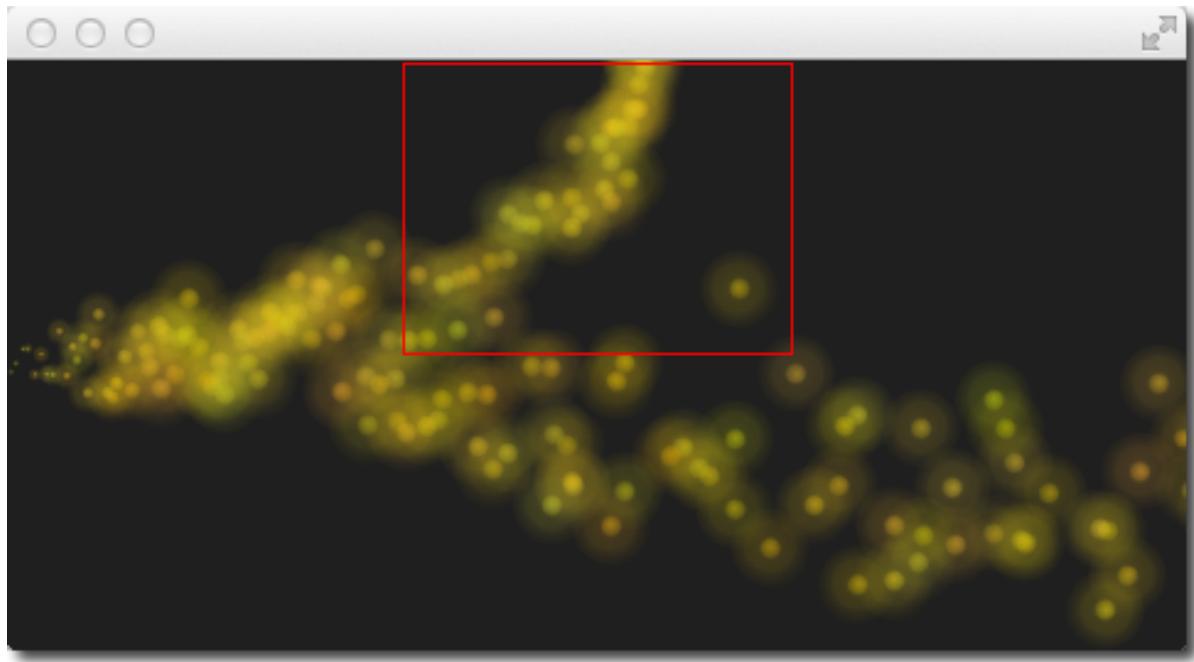
The attractor attracts particles towards a specific point. The point is specified using *pointX* and *pointY*, which is relative to the attractor geometry. The strength specifies the force of attraction. In our example we let particles travel from left to right. The attractor is placed on the top and half of the particles travel through the attractor. Affecter 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.

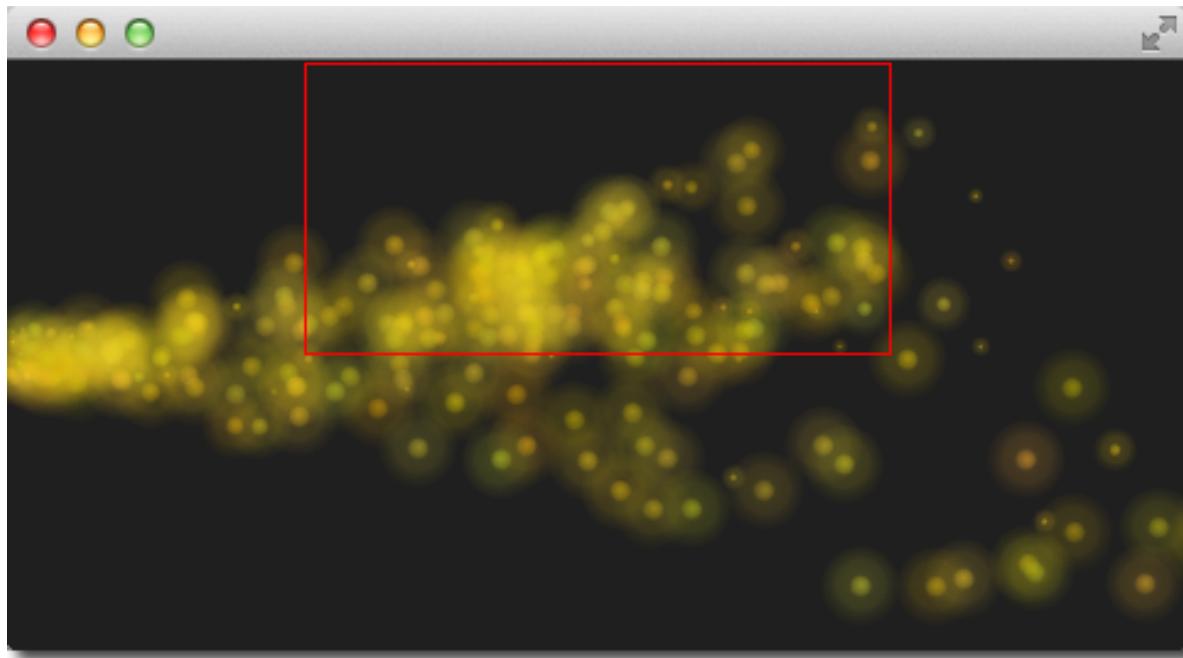


## Friction

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.

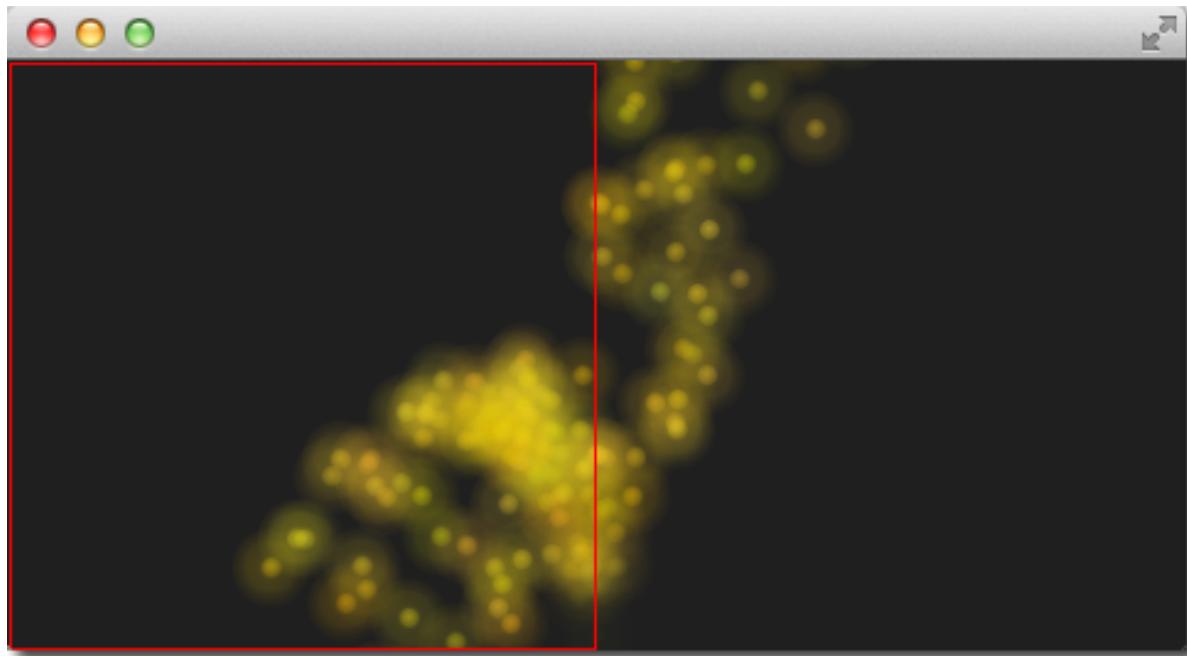


## Gravity

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.

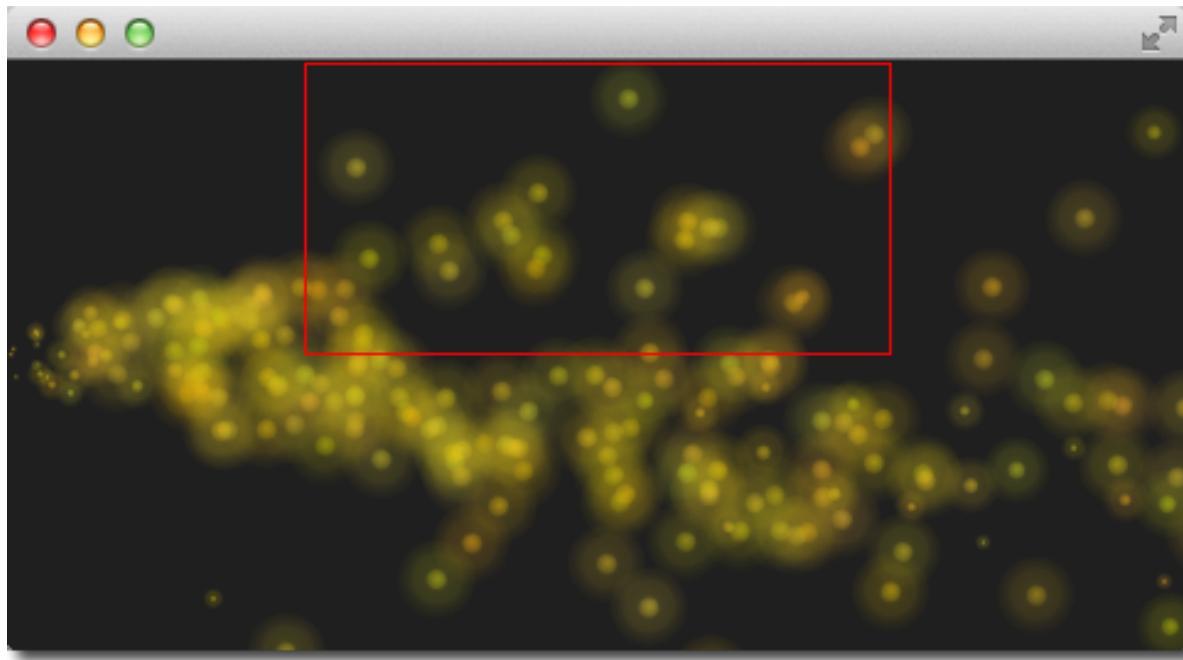


## Turbulence

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.

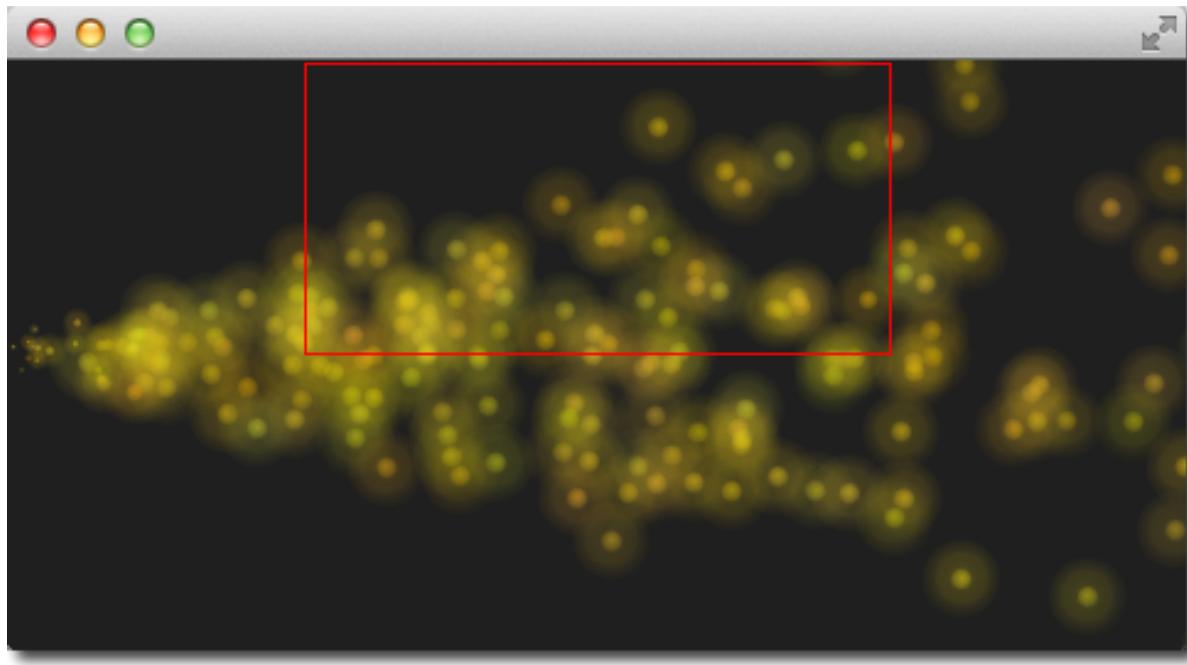


## Wander

The wander manipulates the trajectory. With the property `affectedParameter` can be specified which parameter (velocity, position or acceleration) is affector by the wander. The `pace` property specifies the maximum of attribute changes per second. The `yVariance` and `yVariance` specifies the influence on x and y component of the particle trajectory.

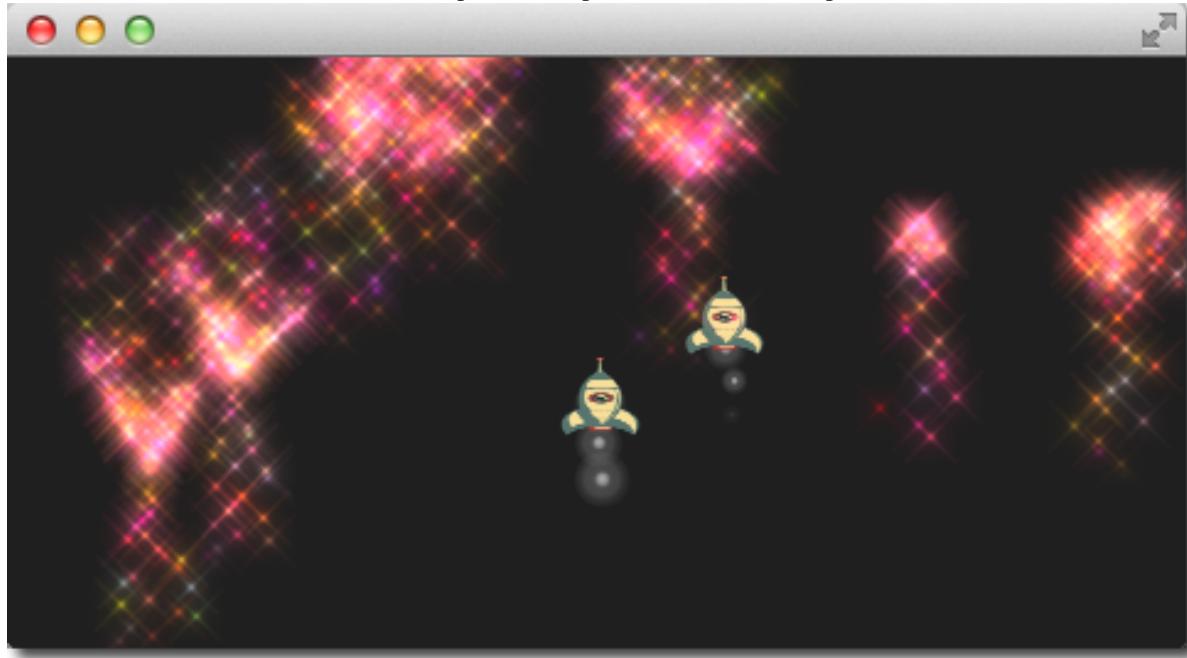
```
Wander {
    anchors.horizontalCenter: parent.horizontalCenter
    width: 240; height: 120
    system: particleSystem
    affectedParameter: Wander.Position
    pace: 200
    yVariance: 240
    Tracer {}
}
```

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.



### 1.8.7 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 2.0
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 pain 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 than 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
}
```

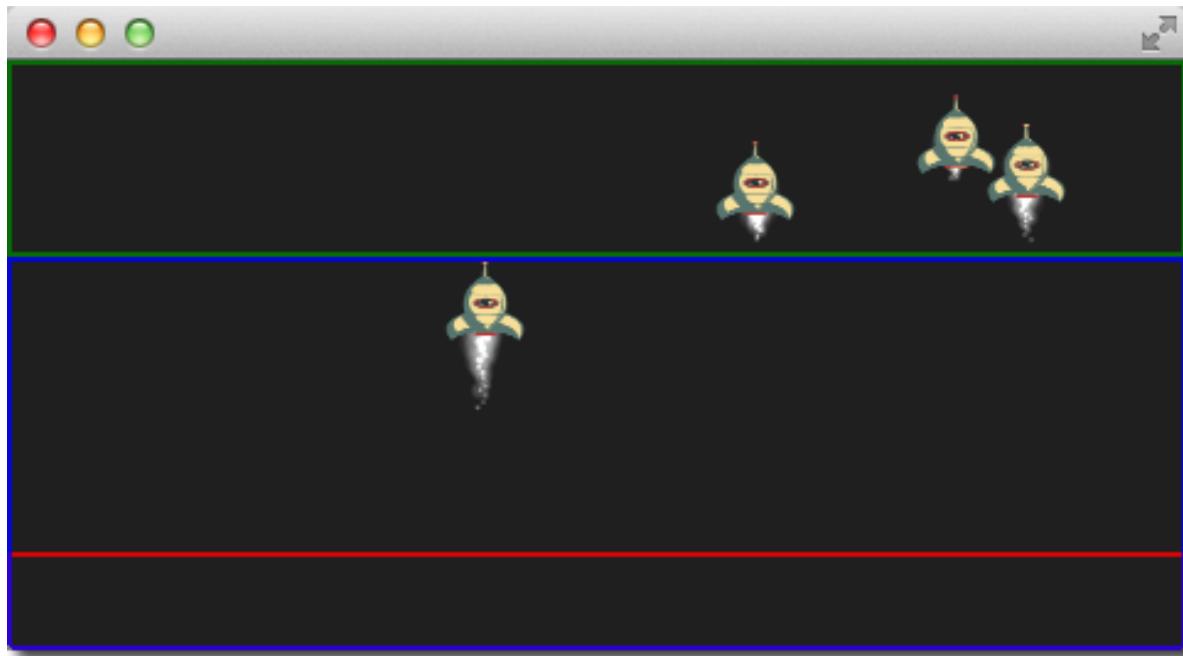
In the friction affector you also need to declare which groups of particles it shall affect. The friction will slow all rockets, which are 80 pixel downwards from the top of the screen down by a factor of 0.9 (try 100 and you will see they almost stop immediately) until they reach a velocity of 5 pixel 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 itself 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 Qt5 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
}
```

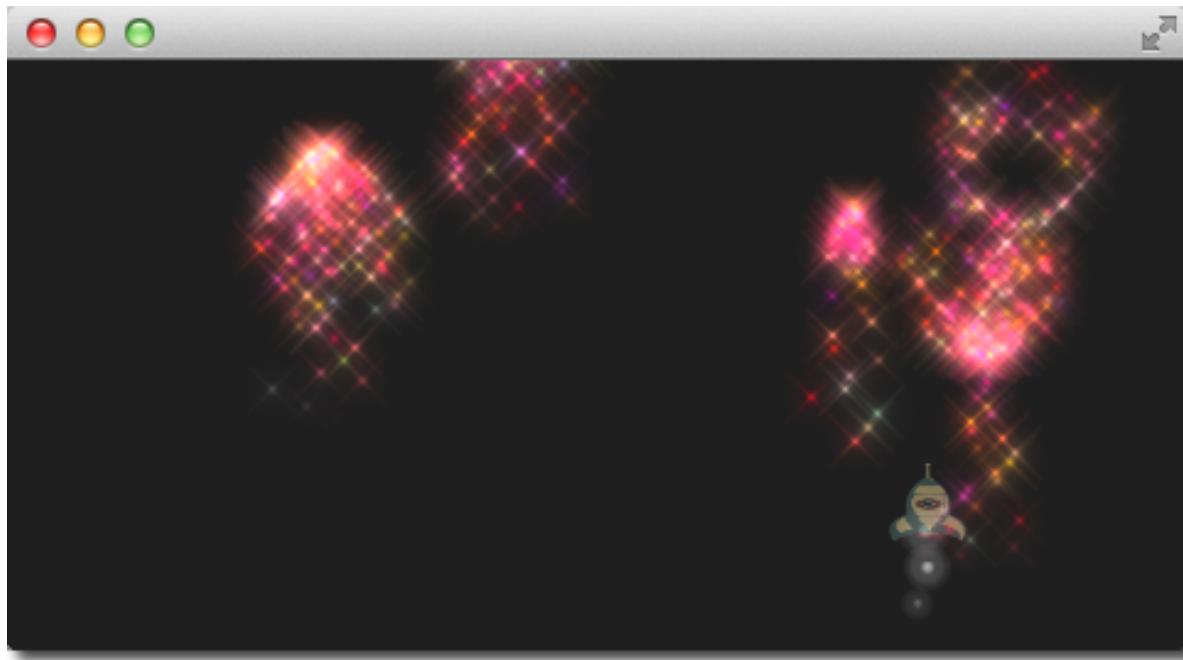
The sparkles of our firework shall be little red stars with a almost transparent color to allow some shine effects.

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 2.0
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: 32
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 }
    }
}
}
}

```

## 1.8.8 Summary

Particles are a very powerful and fun way to express graphical phenomena like smoke, firework, random visual elements. The extended API in Qt5 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 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.

## 1.9 Shader Effects

*Section author: jryannel*

---

**Note:** Last Build: January 20, 2014 at 18:00 CET

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

---

### Objective

- <http://labs.qt.nokia.com/2012/02/02/qt-graphical-effects-in-qt-labs/>
- <http://labs.qt.nokia.com/2011/05/03/qml-shadereffectitem-on-qgraphicsview/>
- <http://qt-project.org/doc/qt-4.8/declarative-shadereffects.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.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](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.

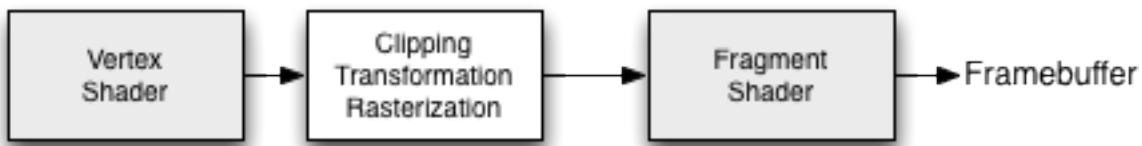
Shaders allows us to create awesome rendering effects on top to the SceneGraph API leveraging directly the power of OpenGL running on the GPU. Shaders are implemented using the ShaderEffect and ShaderEffectSource elements. The shader algorithm itself is implemented using the OpenGL Shading Language.

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.

### 1.9.1 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.

### 1.9.2 Shader Elements

For programming 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 its 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 2.0

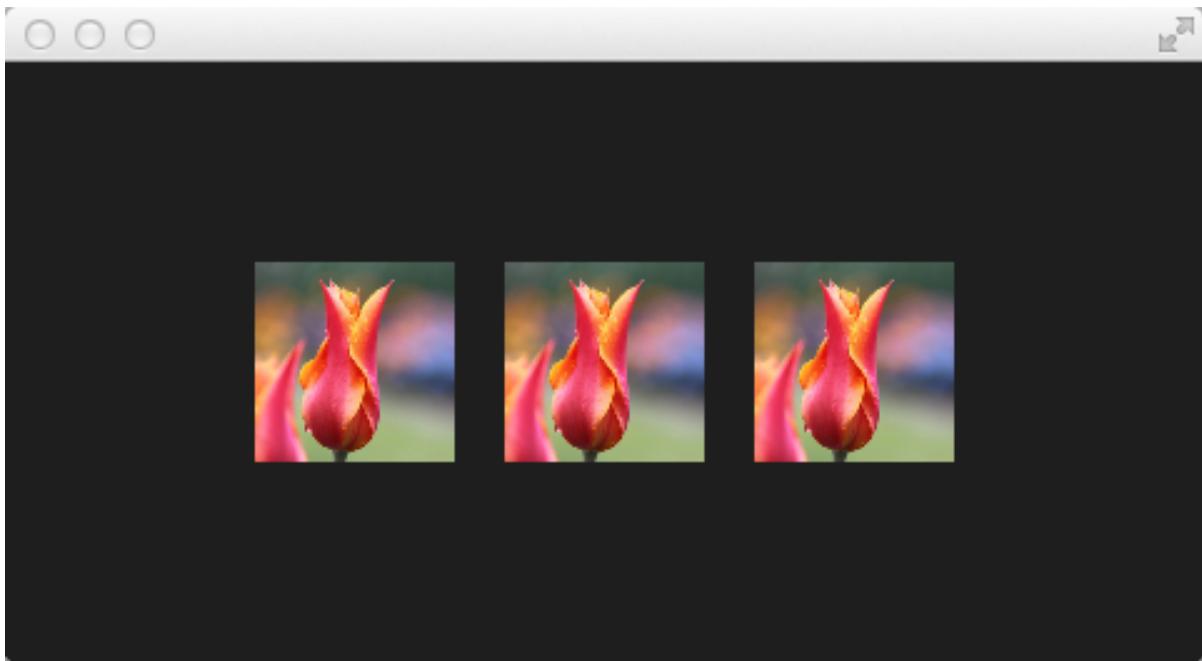
Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    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 Qt5 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.

### 1.9.3 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.

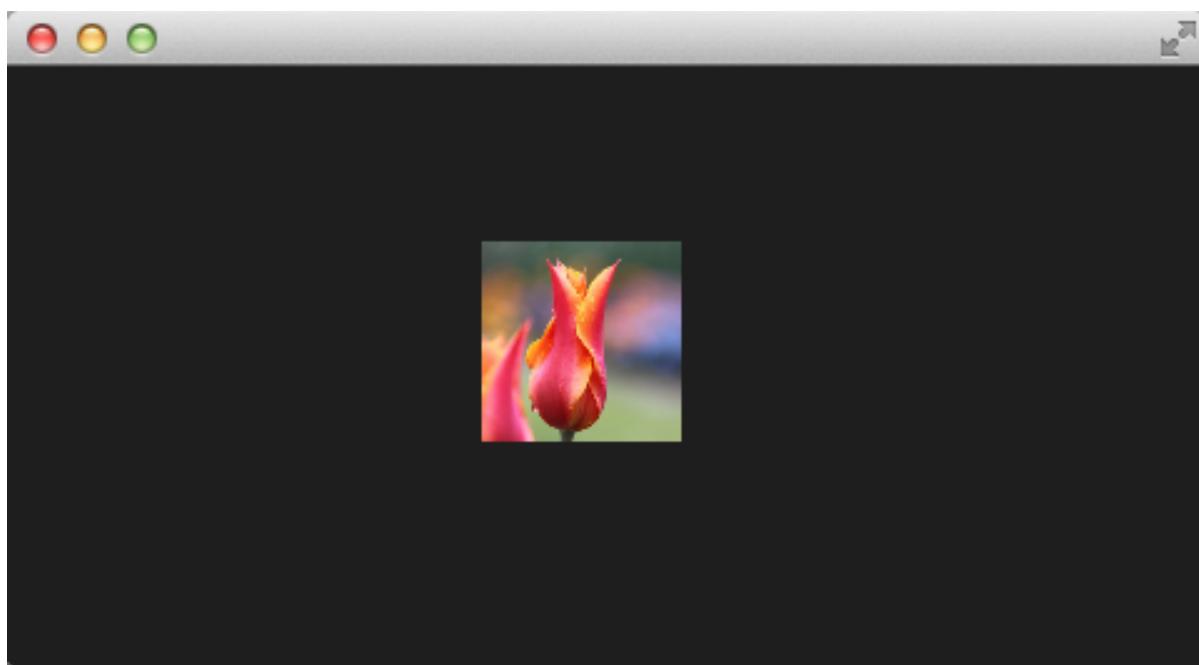
```

import QtQuick 2.0

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

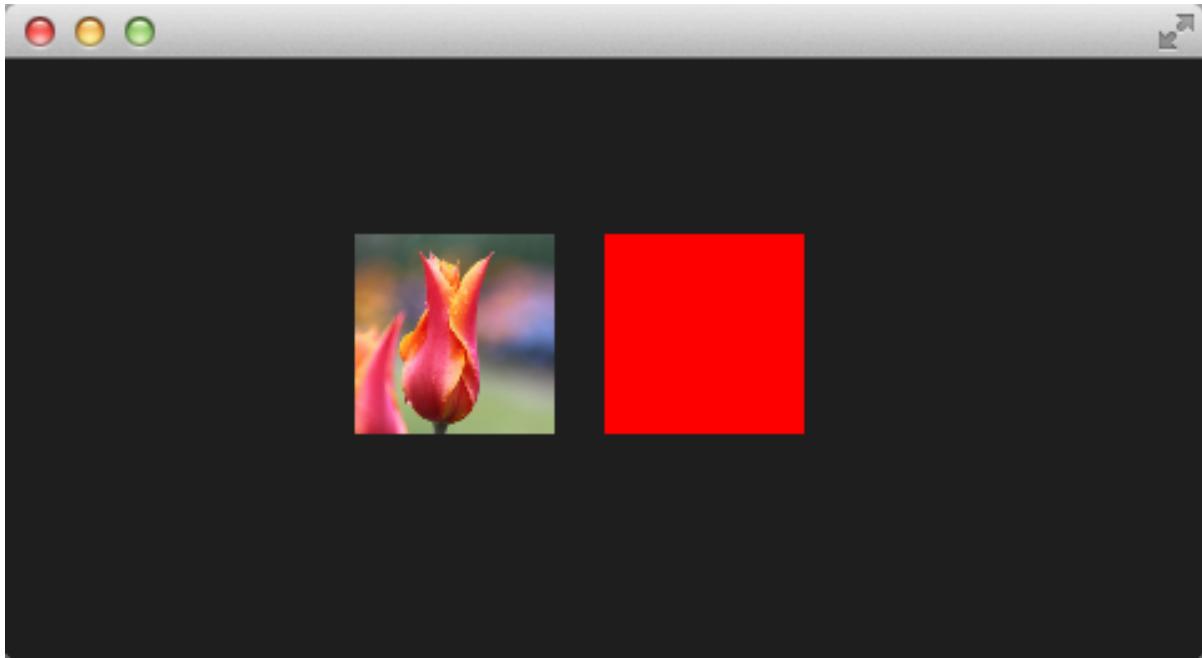
Next we will add a shader, which displays a red rectangle. As we don't need the texture for this we remove the texture from the vertex shader.

```

vertexShader: "
    uniform highp mat4 qt_Matrix;
    attribute highp vec4 qt_Vertex;
    void main() {
        gl_Position = qt_Matrix * qt_Vertex;
    }
"
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 simple 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`.



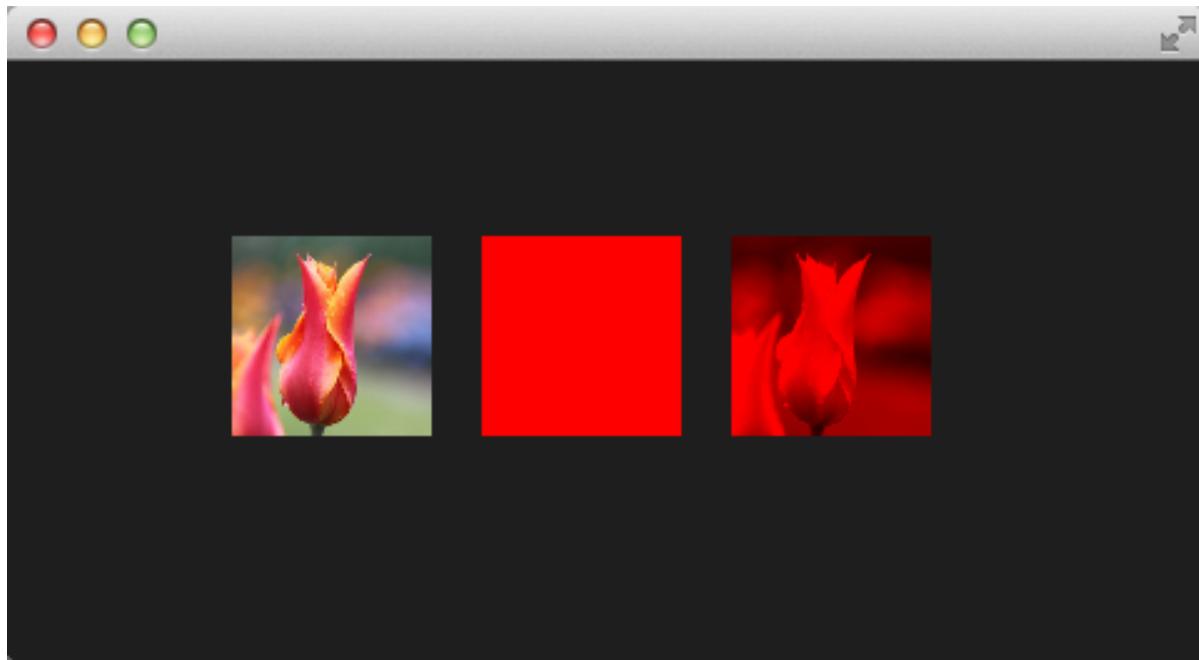
### A red shader with texture

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.

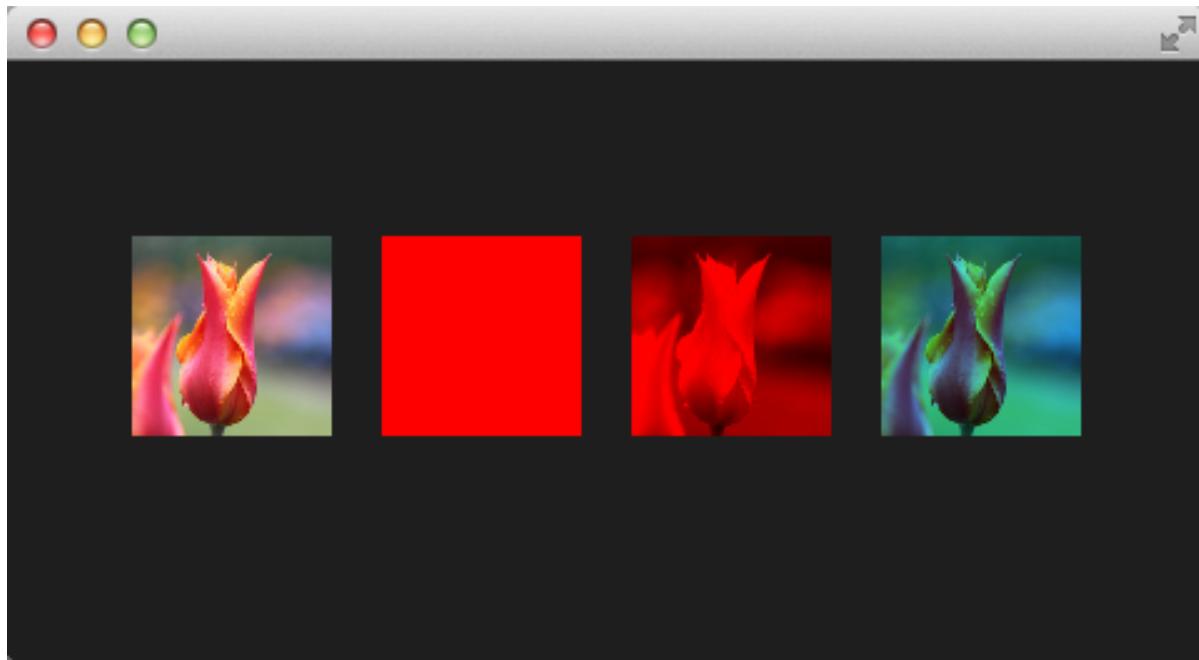


### The red channel property

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);
        }
    "
}
```

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.



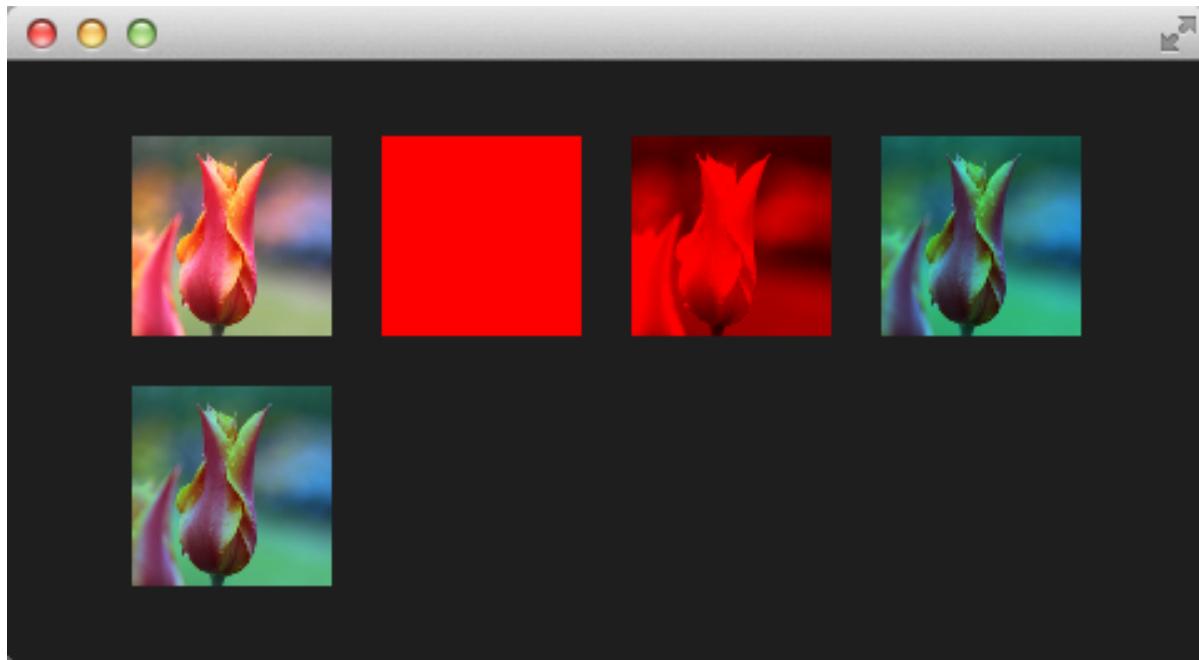
### The red channel animated

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);
        }
    "
}
```

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).

#### 1.9.4 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 2.0

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:

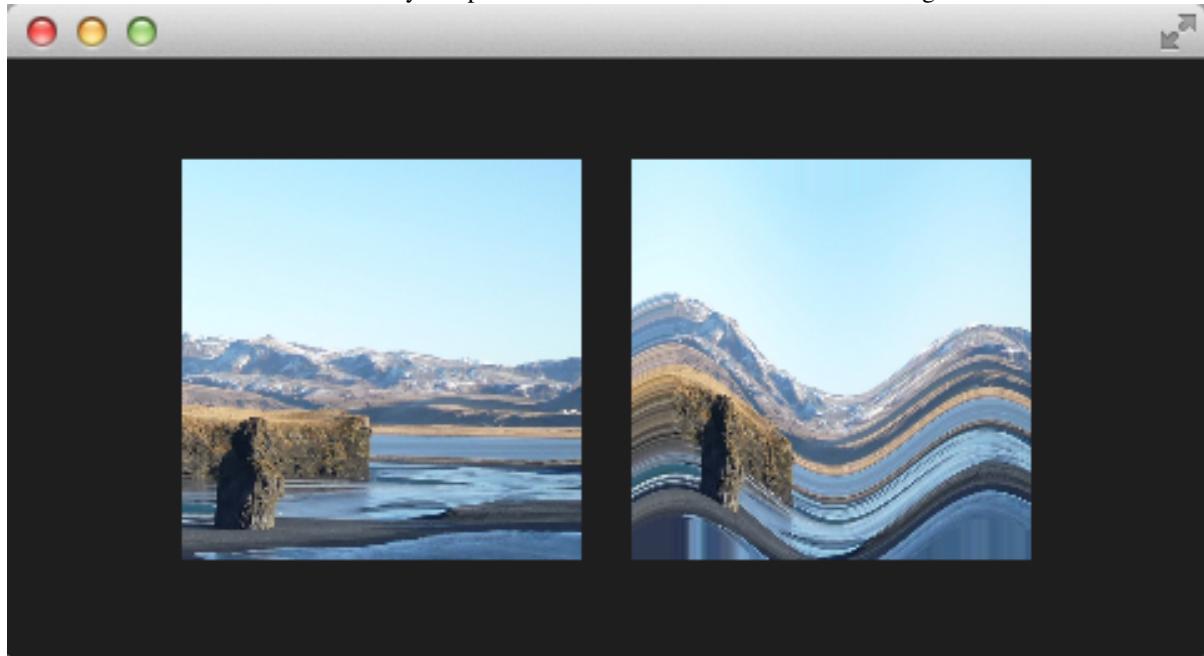
```
highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
```

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);
```

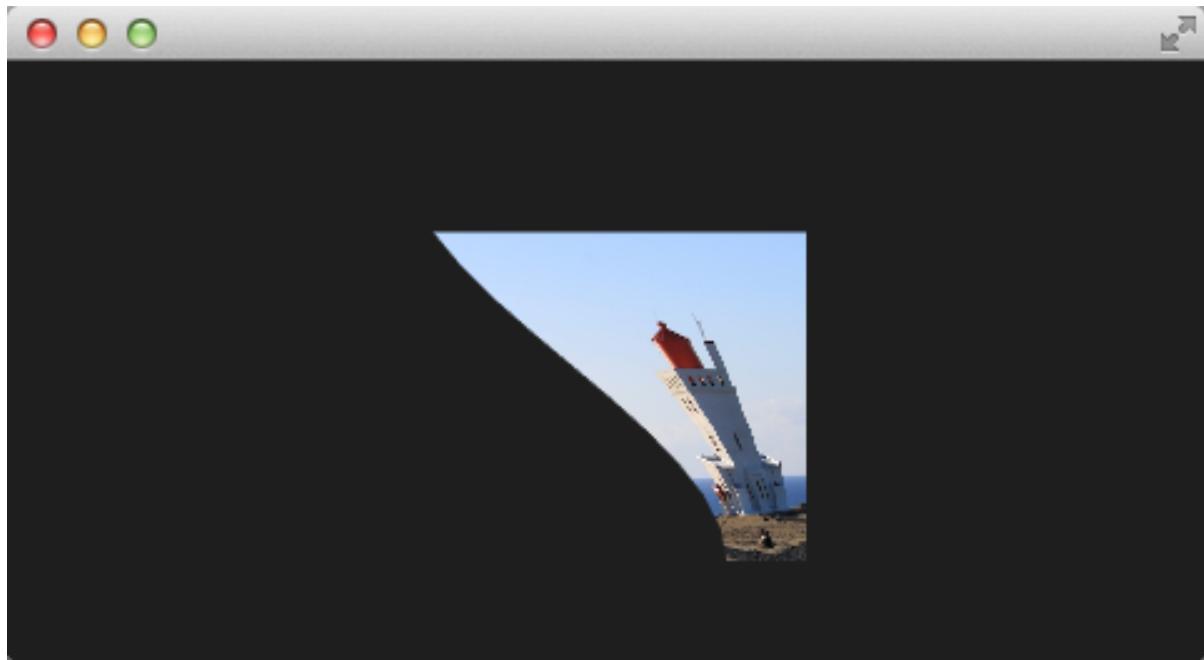
The texture coordinate is influenced by our pulse x-value. The result of this is a moving wave.



Also if we haven't moved pixels in this fragment shader the effect would look at first like a job for a vertex shader.

### 1.9.5 Vertex Shader

The vertex shader can be used to manipulate the vertexes provided by the shader effect. In normal cases the shader effect has 4 vertexes (top-left, top-right, bottom-left and bottom-right). Each vertex reported is of 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

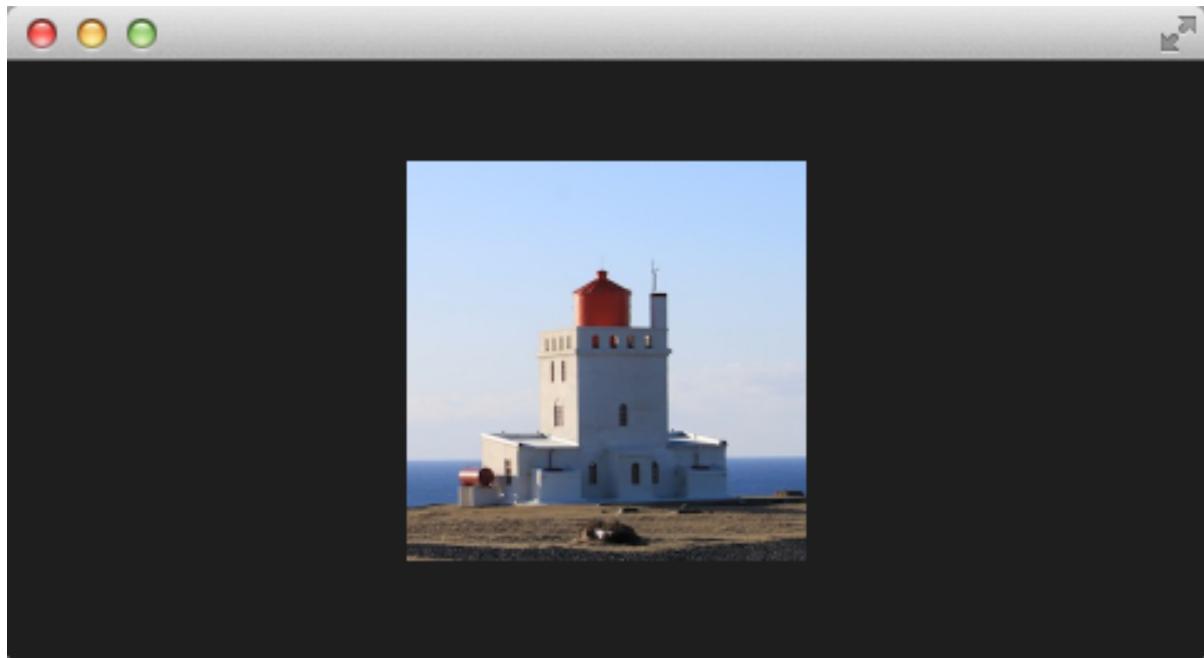
First we will setup our scene again.

```
import QtQuick 2.0

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 us 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;
}

```

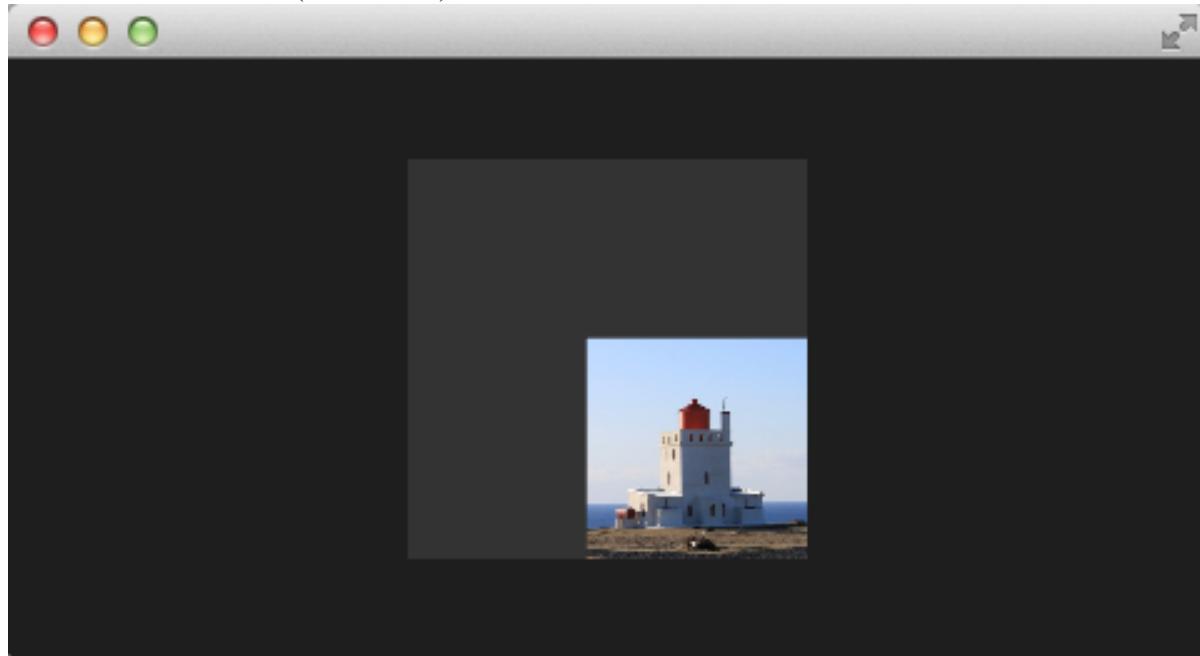
The vertex shader is called for each vertex so four times, in our case. The default qt defined parameters are provided, like *qt\_Matrix*, *qt\_Vertex*, *qt\_MultiTexCoord0*, *qt\_TexCoord0*. We have discussed the variable already earlier. Additional we link the *minimize*, *width* and *height* variables 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:

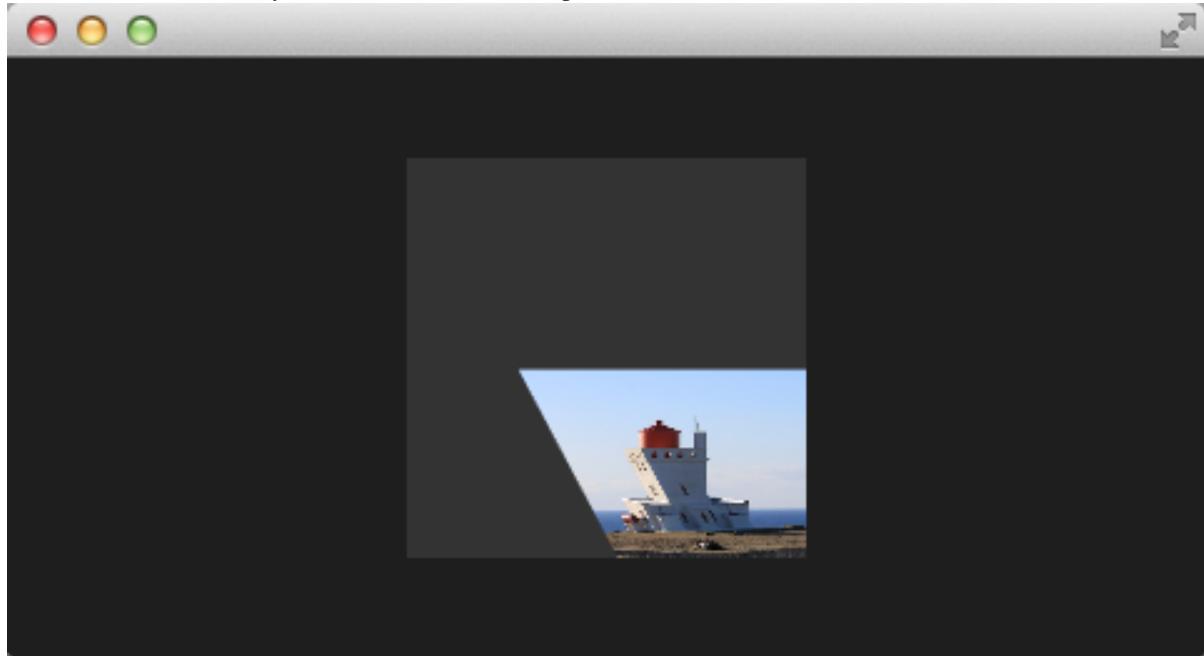
```

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 2.0

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 }
    }
}

```

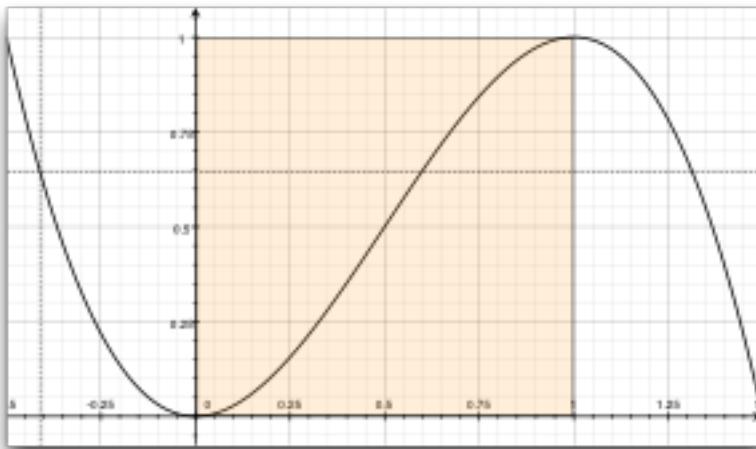
Additional to make the bending a smooth curve the y-effect on the x-position is not modified by a curved function from 0..1 and the `pos.x` depends now on the new bend property animation:

```

highp float t = pos.y / height;
t = (3.0 - 2.0 * t) * t * t;
pos.x = mix(qt_Vertex.x, width, t * bend);

```

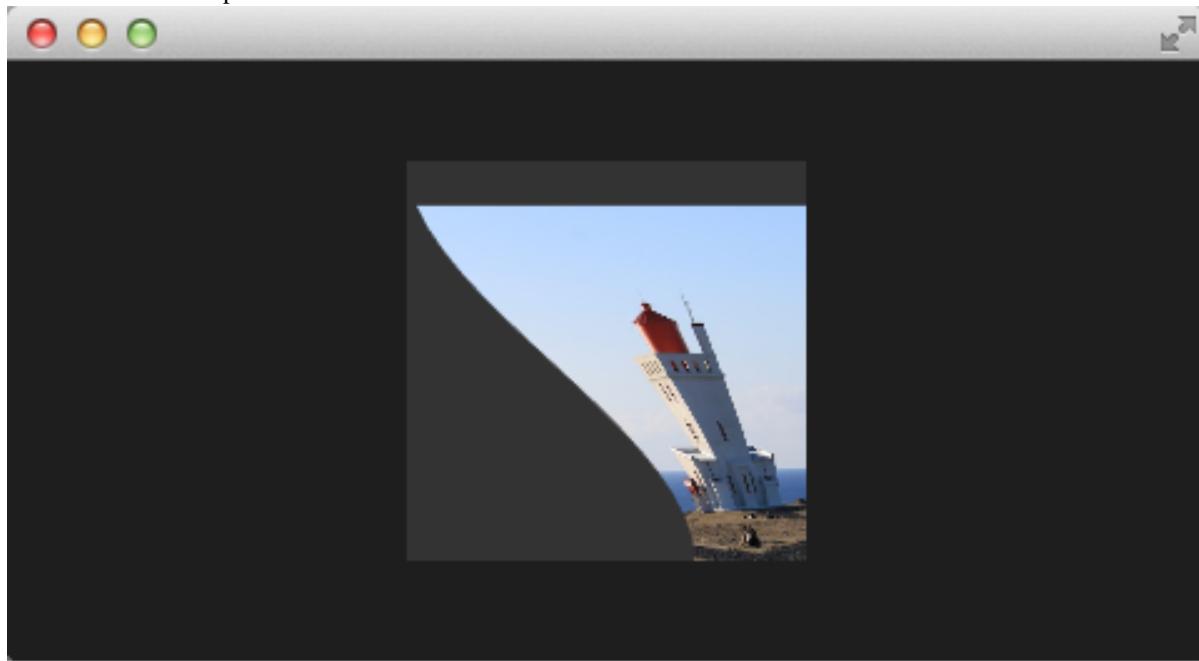
The curve starts smooth at the 0.0 value, grows then and stops smoothly towards the 1.0 value. Here is a plot of the function in the specified range. For us only the range from 0..1 is from interest.



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.

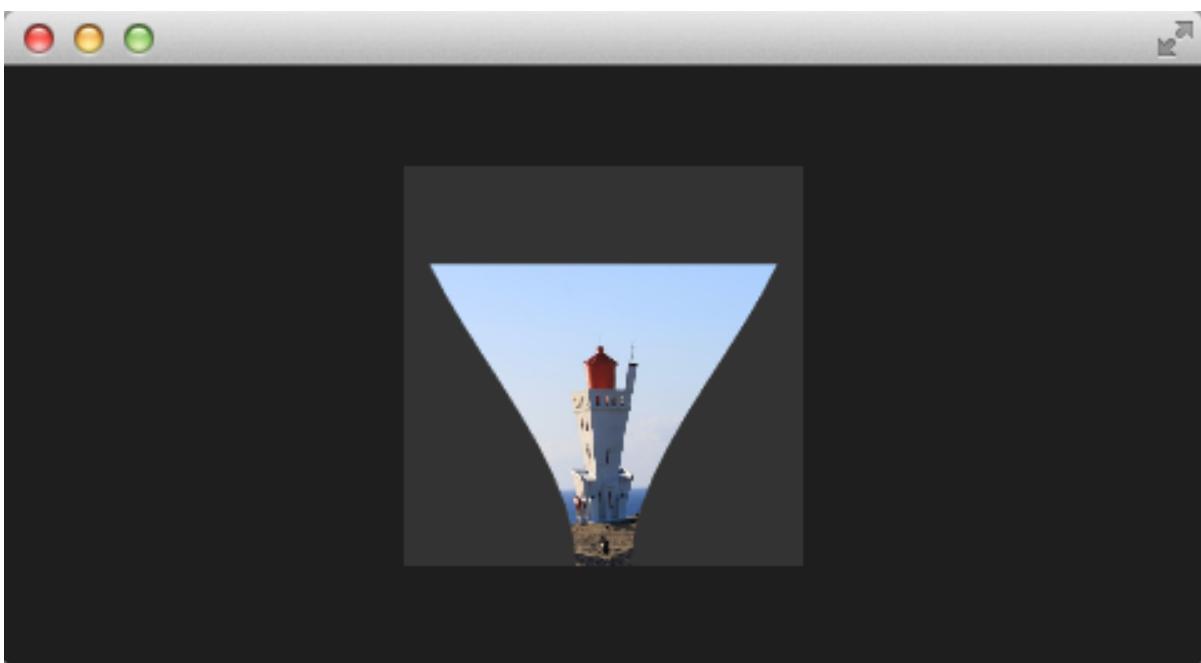
```
ShaderEffect {  
    ...  
}
```

```

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 2.0

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 2.0

Rectangle {
    width: 480; height: 240
    color: '#1e1e1e'

    GenieEffect {
```

```

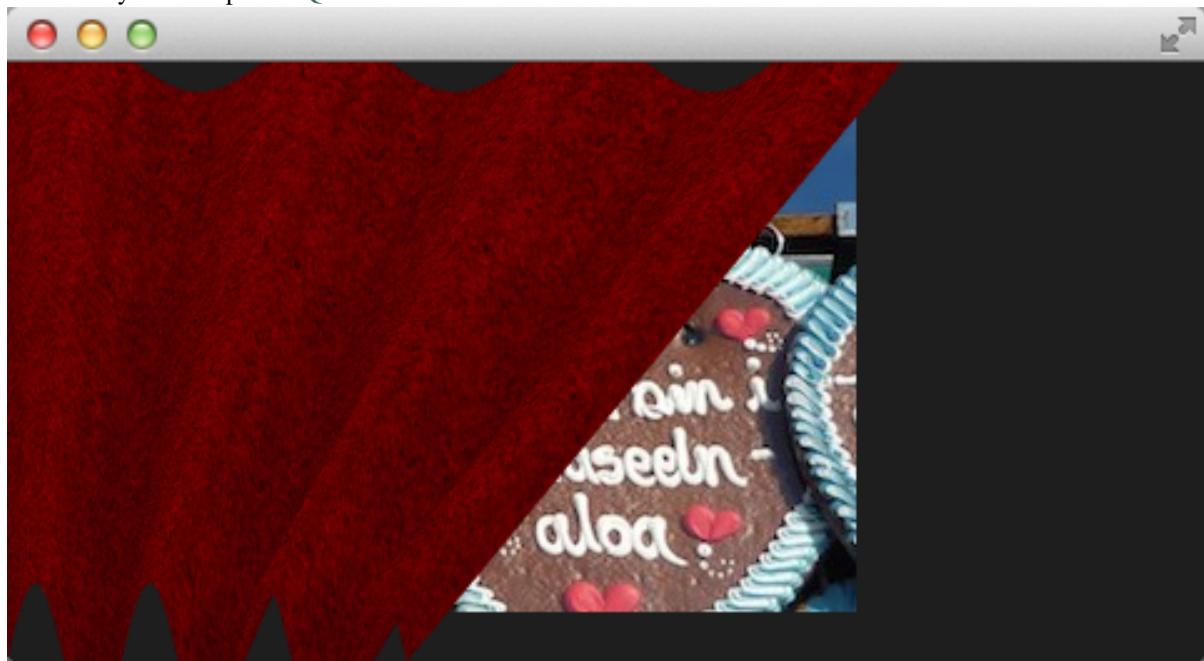
source: Image { source: 'assets/lighthouse.jpg' }
MouseArea {
    anchors.fill: parent
    onClicked: parent.minimized = !parent.minimized
}
}
}
}

```

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.

### 1.9.6 Curtain Effect

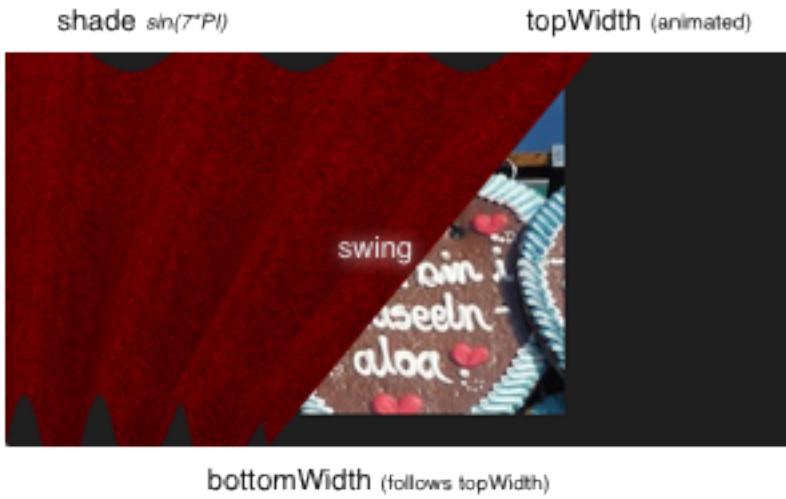
In the last example for custom shader effects I would like to bring you the curtain effect. This effect was published first in May 2011 as part of [Qt labs for shader effects](#).



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 Qt5, 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 ( $7 \times \pi = 21.99\dots$ ) 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 2.0

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.jpg"
    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 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: '#1e1e1e'

    Image {
        anchors.centerIn: parent
        source: 'assets/wiesn.jpg'
    }

    CurtainEffect {
        id: curtain
        anchors.fill: parent
    }
}

```

```
        }

    MouseArea {
        anchors.fill: parent
        onClicked: curtain.open = !curtain.open
    }
}
```

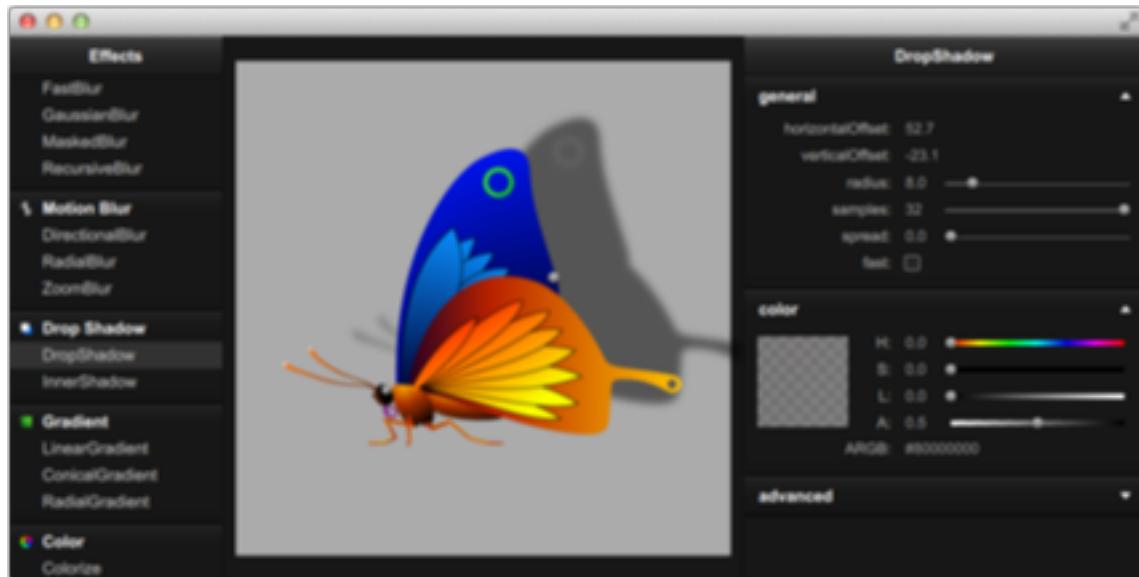
The curtain is opened through a custom *open* property on the curtain effect. We use a *MouseArea* to trigger the opening and closing of the curtain.

### 1.9.7 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 1.1: Graphics Effects List

| Category    | Effect   | Description  |
|-------------|--|--|
| Blend       | <i>Blend</i>   | merges two source items by using a blend mode  |
| Color       | <i>BrightnessContrast</i><br><i>Colorize</i><br><i>ColorOverlay</i><br><i>Desaturate</i><br><i>GammaAdjust</i><br><i>HueSaturation</i><br><i>LevelAdjust</i><br><i>ConicalGradient</i><br><i>LinearGradient</i><br><i>RadialGradient</i> | adjusts brightness and contrast<br>sets color in the HSL color space<br>applies a color layer<br>reduces color saturation<br>adjusts luminance<br>adjusts colors in the HSL color space<br>adjusts colors in the RGB color space<br>draws a conical gradient<br>draws a linear gradient<br>draws a radial gradient |
| Gradient    | <i>Displace</i>  | moves the pixels of the source item according to the specified displacement source   |
| Distortion  | <i>DropShadow</i>  | draws a drop shadow  |
| Drop Shadow | <i>InnerShadow</i>   | draws an inner shadow  |
| Blur        | <i>FastBlur</i><br><i>GaussianBlur</i><br><i>MaskedBlur</i><br><i>RecursiveBlur</i><br><i>DirectionalBlur</i><br><i>RadialBlur</i><br><i>ZoomBlur</i>  | applies a fast blur effect<br>applies a higher quality blur effect<br>applies a varying intensity blur effect<br>blurs repeatedly, providing a strong blur effect<br>applies a directional motion blur effect<br>applies a radial motion blur effect<br>applies a zoom motion blur effect                          |
| Motion Blur | <i>Glow</i><br><i>RectangularGlow</i>  | draws an outer glow effect<br>draws a rectangular outer glow effect  |
| Glow        | <i>OpacityMask</i><br><i>ThresholdMask</i>   | masks the source item with another item<br>masks the source item with another item and applies a threshold value   |
| Mask        |  |  |

Here is an example using the *FastBlur* effect from the *Blur* category:

```
import QtQuick 2.0
import QtGraphicalEffects 1.0

Rectangle {
    width: 480; height: 240
    color: '#e0e0e0'

    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
        }
    }
}
```

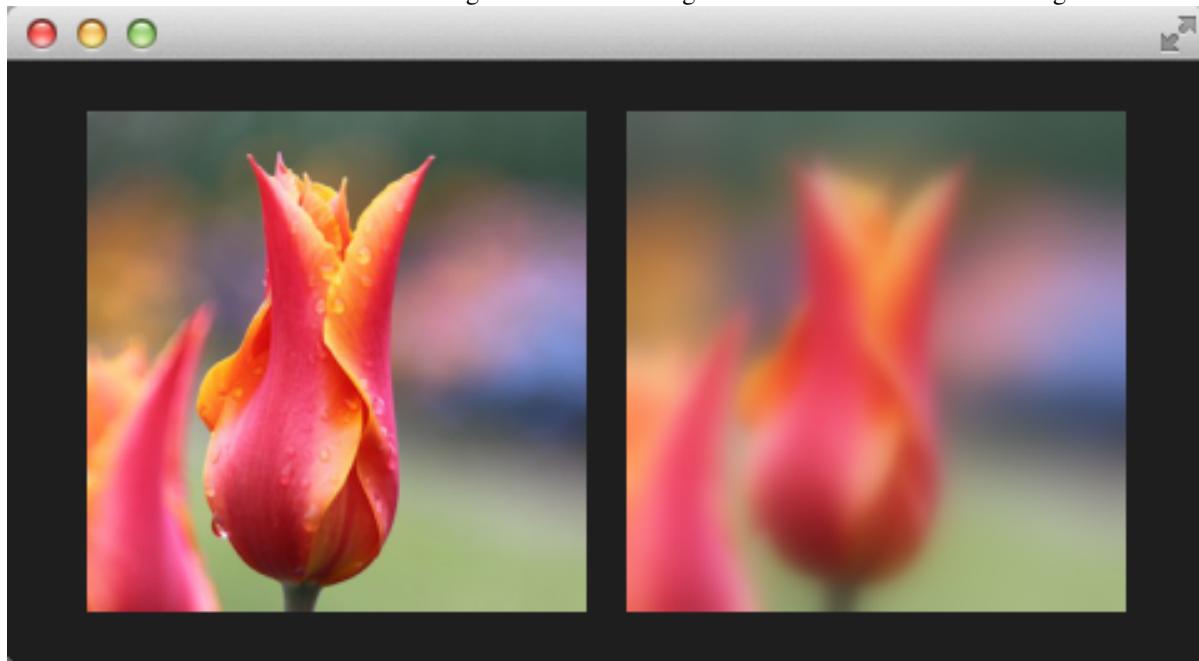
```
property bool blurred: false

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.



## 1.10 Multimedia

*Section author: e8johan*

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 QtQuick core API. Instead, they are provided through a separate API made available by importing QtMultimedia 5.0 as shown below:

```
import QtMultimedia 5.0
```

### 1.10.1 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 2.0
import QtMultimedia 5.0
import QtSystemInfo 5.0

Item {
    width: 1024
    height: 600

    MediaPlayer {
        id: player
        source: "trailer_400p.ogg"
    }

    VideoOutput {
        anchors.fill: parent
        source: player
    }

    Component.onCompleted: {
        player.play();
    }

    ScreenSaver {
        screenSaverEnabled: false;
    }
}
// M1>>
```

In addition to the video playback demonstrated above, the example also contains a small snippet of code for disabling the screen saver. This prevents the video to be interrupted. This is achieved by setting the `screenSaverEnabled` property of the `ScreenSaver` element to `false`. The `ScreenSaver` element is available through the `QtSystemInfo 5.0` API.

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 continuous animations or a looping background song.

## 1.10.2 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.

---

### 1.10.3 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 2.0
import QtMultimedia 5.0

Item {
    width: 1024
    height: 600

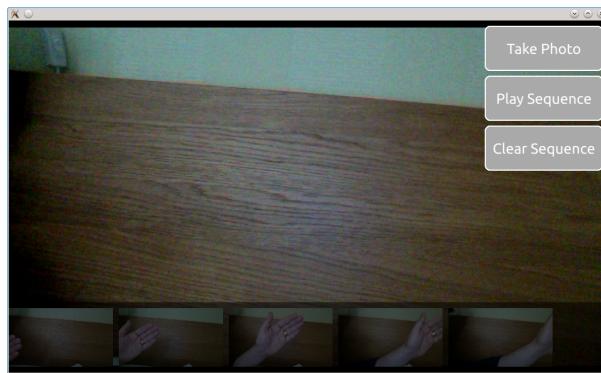
    VideoOutput {
        anchors.fill: parent
        source: camera
    }

    Camera {
        id: camera
    }
}
```

### 1.10.4 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 { source: path; fillMode: Image.PreserveAspectFit; height: 100; }

    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

    width: 200
    height: 75

    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: {
        imagePaths.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 = "";
        }
    }
}
```

### 1.10.5 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`.

```
Playlist {
    id: playlist

    mediaPlayer: player

    items: ListModel {
        ListElement { source: "trailer_400p.ogg" }
        ListElement { source: "trailer_400p.ogg" }
        ListElement { source: "trailer_400p.ogg" }
    }
}

MediaPlayer {
    id: player
}
```

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();
    }
}
}
```

## 1.10.6 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.

# 1.11 Networking

*Section author: jryannel*

Qt5 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 QtQuick 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 QtQuick world.

## 1.11.1 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 2.0

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:8000/main.qml`. You can test it with:

```
$ curl http://localhost:8000/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 simple dynamically load it using QML. We choose the dynamic loading as it work just fine. For this we use a loader element to retrieve for us the remote document.

```
// remote.qml
import QtQuick 2.0

Loader {
    id: root
    source: 'http://localhost:8080/main2.qml'
    onLoaded: {
        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 2.0

Loader {
    id: root
    source: 'https://gist.github.com/jryannel/7983492/raw'
    onLoaded: {
        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 now:

```
import QtQuick 2.0

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 2.0
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.

## 1.11.2 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 doing 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.

## 1.11.3 HTTP Requests

A http request is in Qt typical 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 QtQuick 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 send 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");  
    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 2.0  
  
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) {  
                ...  
            }  
        }  
        xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?format=json&nojsoncallback=1");  
        xhr.send();  
    }  
}
```

```
        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");
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.

#### 1.11.4 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 QtQuick side. To store data back to the source we would need a small REST based HTTP server or a native QtQuick extension for file access.

```
import QtQuick 2.0

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 it 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.

## 1.11.5 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://127.0.0.1:5000/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    | http://localhost:5000/colors/<name> |
| Update Entry | PUT    | http://localhost:5000/colors/<name> |
| Delete Entry | DELETE | 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.

### 1.11.6 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/>

### 1.11.7 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.

### 1.11.8 Web Sockets

WebSockets are not yet part of the Qt delivery. There are some efforts to get WebSockets into Qt/QML. From the author perspective WebSockets have a great potential to add the missing feature to networked HTTP services - notification. HTTP gives us get and post but not yet a notification. Currently the client polls the server bit for real applications the server needs to be able to notify the client also about changes and events. You can compare this to the QML API: properties, functions, signals. Also called get/set/call and notifications.

As it looks like a QML WebSocket plugin will be on the way to Qt5. You can try the new web sockets plugin from the qt playground. For a test run we implement and echo server using an existing web socket server.

First check you are using Qt 5.2.x by querying qmake.

```
$ qmake --version
... Using Qt version 5.2.0 ...
```

Then you need to clone the web socket repository and build it.

```
$ git clone git@gitorious.org:qtplayground/websockets.git
$ cd websockets
$ qmake
$ make
$ make install
```

Now you should be able to use the web socket qml module.

```
import Qt.WebSockets 1.0

WebSocket {
    id: socket
}
```

To test your web socket we will use the echo server from <http://websocket.org>.

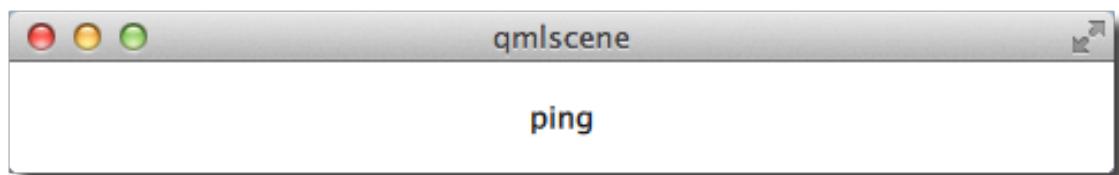
```
import QtQuick 2.0
import Qt.WebSockets 1.0

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.sendTextMessage("ping")
            } else if (socket.status == WebSocket.Closed) {
                text += "\nSocket closed"
            }
        }
    }
}
```

You should see the ping message we send `socket.sendTextMessage("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`).

```
$ 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 sent 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 2.0  
  
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 2.0  
  
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 2.0

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 2.0
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.sendTextMessage(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.

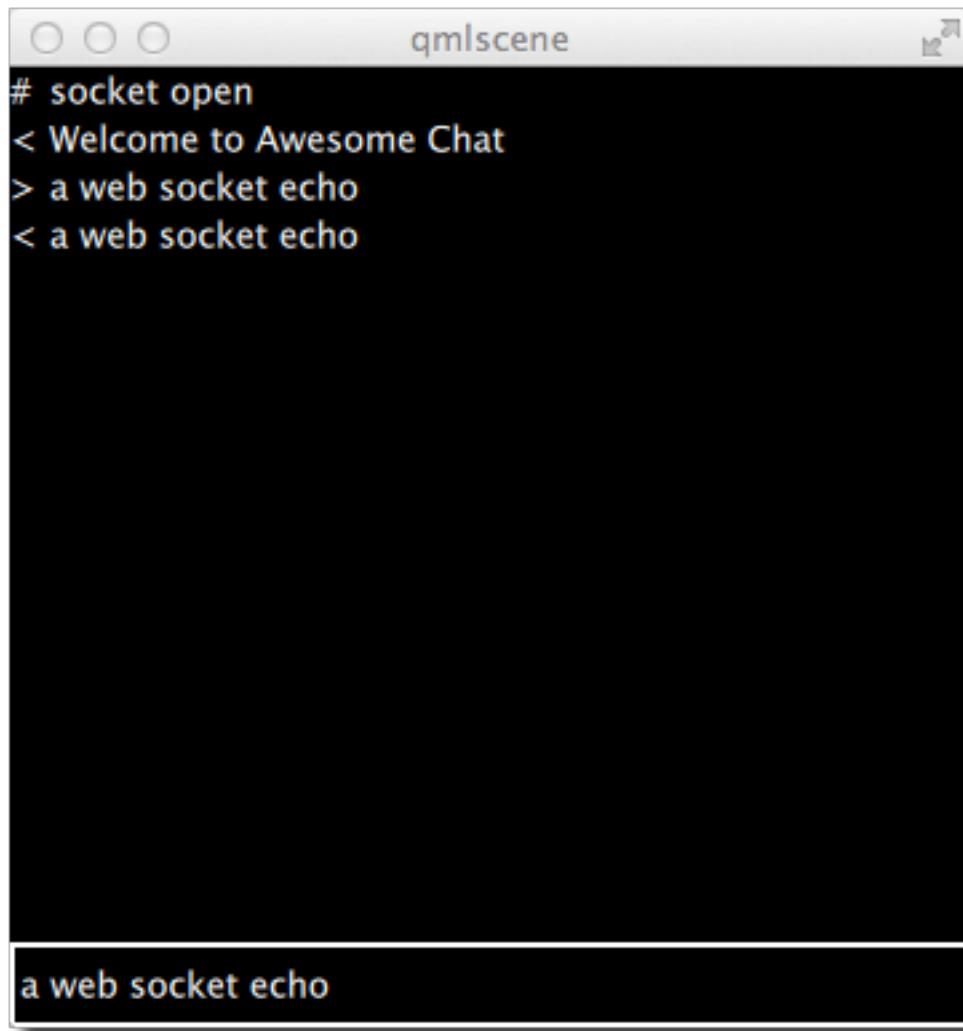
#### Running the server

```
$ 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.



### 1.11.9 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.



---

**CHAPTER  
TWO**

---

**CHAPTER QUEUE**

The chapter queue are the chapters we are working on. They might be almost ready or in an infant state. Give it a try if you like.

- ch12/index
- ch13/index
- ch14/index
- ch15/index
- ch16/index



**A**

anchors, 42  
Animations, 49

**B**

binding, 23  
Bouncing Ball, 49

**C**

ClickableImage, 49  
ClickableImage Helper, 34  
ColorAnimation, 49  
Column, 37  
components, 31

**E**

Easing Curves, 49

**F**

Flow, 37  
focus, 44  
FocusScope, 44

**G**

gradients, 27  
Grid, 37  
GridView, 67  
Grouped Animations, 49

**I**

Image, 27  
Item, 27

**K**

KeyNavigation, 44  
Keys, 44

**L**

ListElement, 67  
ListModel, 67  
ListView, 67

**M**

MouseArea, 27

**N**

NumberAnimation, 49

**P**

ParallelAnimation, 49  
PathView, 67  
properties, 23

**Q**

qmlscene, 23

**R**

Rectangle, 27  
Repeater, 37, 67  
Rotation, 34  
Row, 37

**S**

Scaling, 34  
scripting, 23  
SequentialAnimation, 49  
Square Helper, 37  
Stacking order, 34  
States, 49  
syntax, 23

**T**

Text, 27  
TextEdit, 44  
TextInput, 44  
Transformation, 34  
Transition, 49  
Transitions, 49  
Translation, 34

**X**

XmlListModel, 67