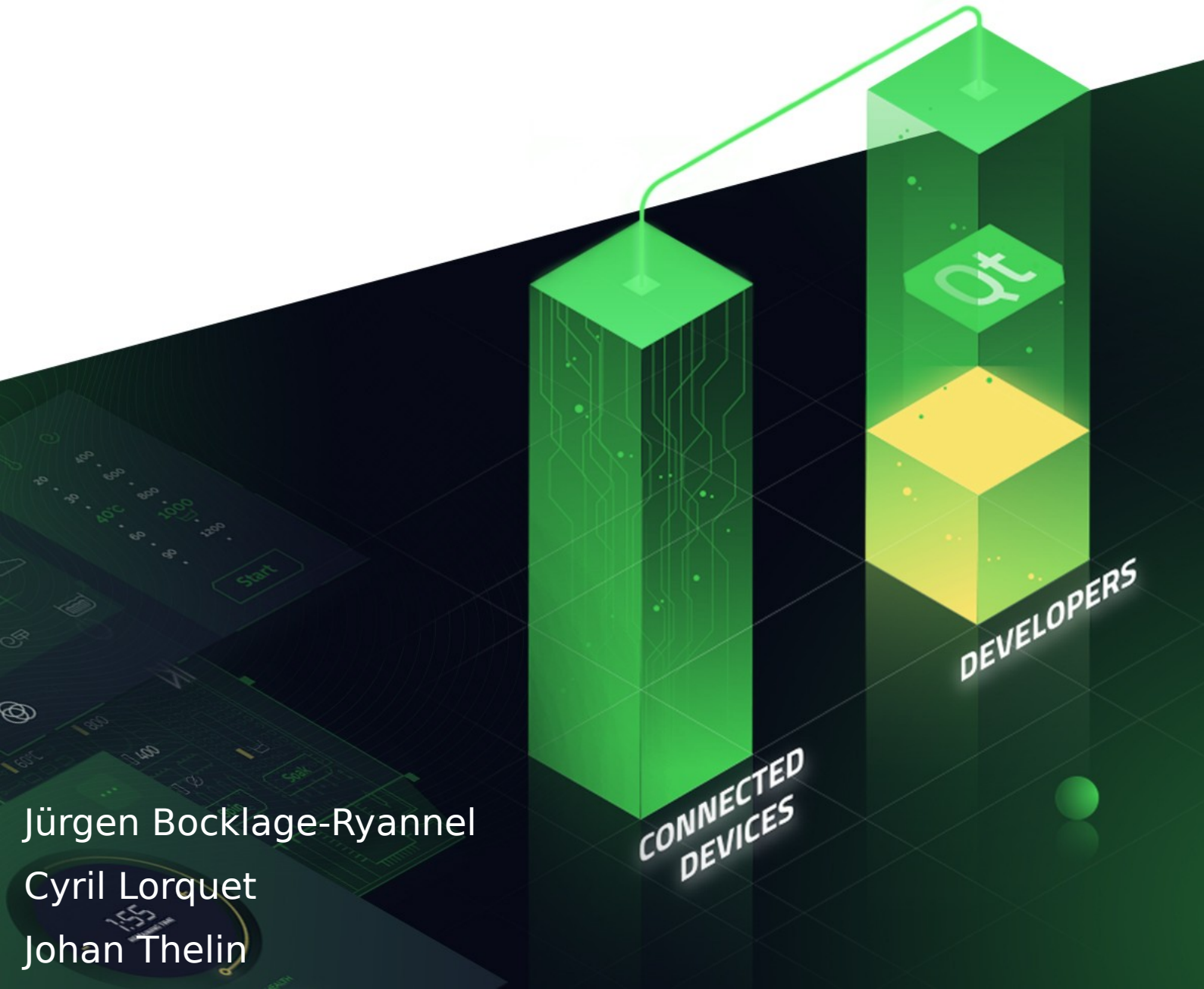
The Qt Company

Qt 6 QML



Jürgen Bocklage-Ryannel
Cyril Lorquet
Johan Thelin

Table of Contents

Welcome! 5
- Acknowledgements 7
- Authors 10

Qt and Qt Quick 14
- Qt Building Blocks 17
- Qt 6 Introduction 21

Quick Start 30
- Installing Qt 6 SDK 31
- Hello World 33
- Application Types 37
- Summary 51

Qt Creator IDE 52
- The User Interface 53
- Registering your Qt Kit 55
- Managing Projects 56
- Using the Editor 58
- Locator 59
- Debugging 61
- Shortcuts 62

Quick Starter 64
- QML Syntax 65
- Core Elements 75
- Components 85
- Simple Transformations 90
- Positioning Elements 95
- Layout Items 103
- Input Elements 107
- Advanced Techniques 116

Fluid Elements 117
- Animations 118
- States and Transitions 145
- Advanced Techniques 152

UI Controls 153
- Introduction to Controls 154
- An Image Viewer 163
- Common Patterns 186
- The Imagine Style 209
- Summary 217

Model-View-Delegate 218
- Concept 219
- Basic Models 221
- Dynamic Views 229
- Delegate 248
- Advanced Techniques 265

Summary 287
Canvas Element 289
Convenience API 293
Gradients 295
Shadows 297
Images 299
Transformation 301
Composition Modes 303
Pixel Buffers 306
Canvas Paint 309
Porting from HTML5 Canvas 312
Shapes 321
A Basic Shape 322
Building Paths 326
Filling Shapes 331
Animating Shapes 339
Summary 342
Effects in QML 343
Particle Concept 345
Simple Simulation 347
Particle Parameters 351
Directed Particles 354
Affecting Particles 361
Particle Groups 368
Particle Painters 380
Graphics Shaders 382
Shader Elements 384
Fragment Shaders 388
Wave Effect 395
Vertex Shader 398
Curtain Effect 412
Summary 418
Multimedia 419
Playing Media 420
Sound Effects 430
Video Streams 435
Capturing Images 437
Summary 444
Qt Quick 3D 445
The Basics 446
Working with Assets 457
Materials and Light 468
Animations 476
Mixing 2D and 3D Contents 483
Summary 486
Networking 488

Serving UI via HTTP 489
Templates 497
HTTP Requests 499
Local files 505
REST API 509
Authentication using OAuth 519
Web Sockets 539
Summary 547
Storage 548
Settings 549
Local Storage - SQL 552
Dynamic QML 558
Loading Components Dynamically 559
Creating and Destroying Objects 568
Tracking Dynamic Objects 575
Summary 580
JavaScript 581
Browser/HTML vs Qt Quick/QML 584
JS Language 586
JS Objects 588
Creating a JS Console 591
Qt and C++ 598
A Boilerplate Application 601
The QObject 609
Build Systems 613
Common Qt Classes 620
Models in C++ 630
Extending QML with C++ 647
Understanding the QML Run-time 648
Plugin Content 654
Creating the plugin 657
FileIO Implementation 660
Using FileIO 663
Summary 674
Qt for Python 676
Introduction 677
Installing 679
Building an Application 681
Limitations 701
Summary 702
Qt for MCUs 703
Setup 704
Hello World - for MCUs 708
Integrating with C++ 716
Working with Models 724
Summary 731

Welcome!

Welcome to The Qt 6 Book - A book about QML. This text will guide you through QML, Qt's language for creating dynamic user interfaces.

I believe that the ability to build declarative, reactive, hardware accelerated user interfaces executing at native performance across *all* major platforms (and some not so major) is a game changer. When starting with Qt, it was almost as if I had my secret weapon to building software quickly. QML takes that to the next level.

How is this book different from the Qt documentation? I hear you ask. The intention is to build a complement. This book is meant as a book that you can read from front to back where each chapter builds on what you've previously learned. But it can also be used as a way for the experienced reader to get oriented in a new topic. Each chapter focuses on a specific topic and introduces the concepts from Qt and QML. However, the Qt documentation will always provide the full picture and is a great reference to look up the details about all elements, properties, enumerations, and more.

I wish you a pleasant read!

Johan Thelin

Structure

The book can be said to be split into three parts. The split is not clear cut enough to motivate a strict division of chapters, but more of a guideline that we've tried to follow when writing it.

The first few chapters, let's say until somewhere around chapter 5 - 7 can be considered an introduction. If you want to learn QML, you should make sure to read these chapters.

The following chapters, 6-14, can be seen as fairly separate chapters introducing independent topics, even though the models from chapter 7 are used in many more places. Feel free to dive into these in the order that you like and learn about the topics that you are curious about.

The remainder of the book focuses on more advanced topics such as details of JavaScript, mixing C++ and QML, and the Qt for Python bindings and QML. These are important topics and I really want you to read them. To build a full application with QML you need to understand these topics, but their main focus is not on QML.

Never Ending Work in Progress

The Qt 6 Book is a never ending work in progress. We welcome contributors and are planning to open up our infrastructure to let you contribute both by reporting issues and by contributing fixes and new content. The end goal is to present you with a printed book when the material has reached a maturity level that we are happy with, but we want to share this with you already now and to learn from your feedback what to improve, and what additional content to add.

Acknowledgements

This book would not have been possible to create without the kind sponsorship from **The Qt Company**. It is a privilege to be able to work on a project such as this, and their help has been invaluable. I would like to give a special mention to (alphabetically):

- empenzes
- Fabian K
- Luca Di sera
- magoldst-qt
- Maurice Kalinowski
- Mitch Curtis
- nezticle
- Tino Pyssysalo
- Ulf Hermann
- Vladimir Minenko

Contributors

This book was made possible by the great community contributors. I would like to thank them all (alphabetically):

- alexshen
- arky
- DavidAdamsPerimeter
- delvianv
- guoci
- nittwitt

- oleksis
- LorenDB
- paulmasri
- QtSCH
- ruudschouten
- task-jp
- topecongiro
- VideoCarp
- wangchunlin5013

History

This book is based on **The QML Book (https://qmlbook.github.io/)** , originally written for Qt 5. I would like to thank all contributors to that book (alphabetically):

- aamirglb
- alexeRadu
- andreabedini
- amura11
- bakku
- cibersheep
- dbelyaev
- danielbaak
- DocWicking
- empyrical
- Ge0
- gillesfernandez
- gitter-badger
- gsantner
- hckr
- iitaka1142

- jiakuan
- justinfx
- maggu2810
- marco-piccolino
- mariopal
- mark-summerfield
- mhubig
- micdoug
- Mihaylov93
- moritzsternemann
- RossRogers
- Swordfish90
- sycy600
- trolley
- 29jm

I would also like to give a special mention to Pelagicore, The Qt Company and Felgo for help the development of The QML Book by sponsoring our work and being generally awesome when it comes to feedback and support.

Authors

The Qt 6 Book has been written by a team of authors. They are:

Johan Thelin

Johan works as a system architect building automotive solutions. He has a background from over twenty years of device creation based on Linux, Qt

and more. He has written for various papers and blogs, presented at numerous conferences, and provides advice on how to build software, and software organizations. As an avid believer in free and open source solutions, he founded and organizes the **foss-north conference (https://foss-north.se)** .

You can find out more about Johan at **LinkedIn (https://www.linkedin.com/in/johanthelin)** , **his blog (http://www.thelins.se/johan/blog/)** , and **his homepage (http://e8johan.se)** .

Jürgen Bocklage-Ryannel

Jürgen is the CEO of ApiGear, which is a collaborative machine interface design tool that enables teams to collaborative design software interfaces with automated monitoring and simulation solutions.

He was the co-founder of Pelagicore AG and was responsible there as Chief User Interface Architect for the early versions of the Daimler MBUX.

He focuses currently on an API driven workflow to design and create the interfaces between the user experience and the underlying services for

different platforms.

You can find out more about Jürgen at **LinkedIn (https://www.linkedin.com/in/jryannel/)** .

Cyril Lorquet

Co-Founder and CEO of the belgian company **Eunoia Studio (https://www.eunoia.be)** , Cyril helps organizations turn their know-how into software products. Since 2009, he has been working on software products in various contexts (construction, healthcare, hydrology, marketing, ...) - several of them involving Qt. Software engineer at heart,

he has a passion for design processes, software development and change management.

You can find out more about Cyril at **LinkedIn (https://www.linkedin.com/in/cyrillorquet)** .

Qt and Qt Quick

This book provides you with a walk through of the different aspects of application development using the new Qt 6. It focuses on the Qt Quick technology, but also provides necessary information about writing C++ back-ends and extensions for Qt Quick.

This chapter shall provide a high-level overview of Qt 6. It shows the different application models available for developers, as well as a showcase application, as a sneak preview of things to come. Additionally, the chapter aims to provide a wide overview of the Qt content and how to get in touch with the makers of Qt the Qt Company.

Qt 6 Focus

Qt 5 was released many years ago and introduced a new declarative way of writing stunning user interfaces. Since then a lot has changed in the world around us.

Qt 6 will be a continuation of what has been done with Qt 5 and should not be disruptive to the majority of users. What makes Qt valuable to its users?

- Its cross-platform nature
- Its scalability
- World class APIs and documentation
- Maintainability, stability and compatibility
- A large developer ecosystem

Qt 6 evolves the Qt product to new markets while keeping close to the users values.

The desktop market is the root of the Qt offering. It is where most users get the first contact with Qt and it forms the base for the Qt tools and its success.

It is expected that Qt 6 will grow most in the embedded and connected devices market from high-end near desktop performing devices to low-end devices like microcontrollers. Touch screens will come to an exponential increasing number to these devices. Many of these devices will have relatively simple functionality but require a polished and smooth user interface.

At the other end of the spectrum there is a demand for more complex and 2D/3D integrated user interfaces. The 3D content with 2D elements based interfaces will be common, as will be the use of augmented and virtual reality.

The growth of connected devices and the higher demand for smooth user interfaces require a simpler workflow to create applications and devices. Integrating UX designers into the development workflow is one of the goals of the Qt 6 series.

Qt 6 brings us:

- Next generation QML
- Next generation graphics
- Unified and consistent tooling
- Enhanced Qts C++ APIs
- Component Marketplace

Qt Building Blocks

Qt 6 consists of a large number of modules. In general, a module is a library for the developer to use. Some modules are mandatory for a Qt-enabled platform and form the set called *Qt Essentials Modules*. Other modules are optional, and form the *Qt Add-On Modules*. The majority of developers may not need to use the latter, but it's good to know about them as they provide invaluable solutions to common challenges.

Qt Modules

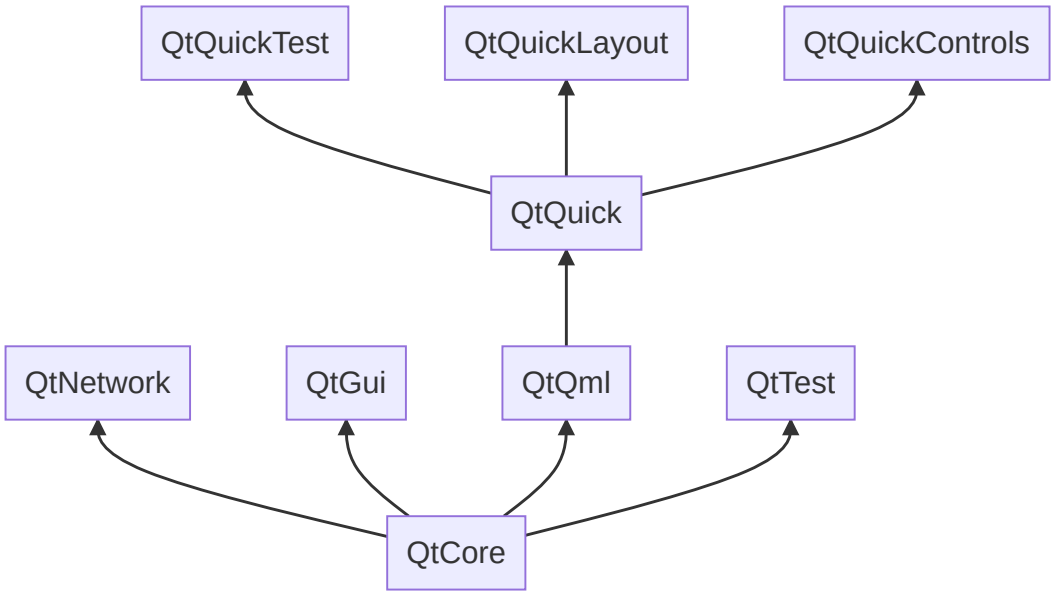
The Qt Essentials modules are mandatory for any Qt-enabled platform. They offer the foundation to develop modern Qt 6 Applications using Qt Quick 2. The full list of modules is available in the **Qt documentation module list (https://doc.qt.io/qt-6/qtmodules.html#qt-essentials)** .

Core-Essential Modules

The minimal set of Qt 6 modules to start QML programming.

- **Qt Core** - Core non-graphical classes used by other modules.
- **Qt D-BUS** - Classes for inter-process communication over the D-Bus protocol on linux.
- **Qt GUI** - Base classes for graphical user interface (GUI) components. Includes OpenGL.
- **Qt Network** - Classes to make network programming easier and more portable.
- **Qt QML** - Classes for QML and JavaScript languages.

- **Qt Quick** - A declarative framework for building highly dynamic applications with custom user interfaces.
- **Qt Quick Controls** - Provides lightweight QML types for creating performant user interfaces for desktop, embedded, and mobile devices. These types employ a simple styling architecture and are very efficient.
- **Qt Quick Layouts** - Layouts are items that are used to arrange Qt Quick 2 based items in the user interface.
- **Qt Quick Test** - A unit test framework for QML applications, where the test cases are written as JavaScript functions.
- **Qt Test** - Classes for unit testing Qt applications and libraries.
- **Qt Widgets** - Classes to extend Qt GUI with C++ widgets.



Qt Add-On Modules

Besides the essential modules, Qt offers additional modules that target specific purposes. Many add-on modules are either feature-complete and exist for backwards compatibility, or are only applicable to certain platforms. Here is a list of some of the available add-on modules, but make sure you familiarize yourself with them all in the **Qt documentation**

add-ons list (https://doc.qt.io/qt-6/qtmodules.html#qt-add-ons) and in the list below.

- **Network**: Qt Bluetooth / Qt Network Authorization
- **UI Components**: Qt Quick 3D / Qt Quick Timeline / Qt Charts / Qt Data Visualization / Qt Lottie Animation / Qt Virtual Keyboard
- **Graphics**: Qt 3D / Qt Image Formats / Qt OpenGL / Qt Shader Tools / Qt SVG / Qt Wayland Compositor
- **Helper**: Qt 5 Core Compatibility APIs / Qt Concurrent / Qt Help / Qt Print Support / Qt Quick Widgets / Qt SCXML / Qt SQL / Qt State Machine / Qt UI Tools / Qt XML

> **TIP**
>
> As these modules are not part of the release, the state of each module may differ depending on how many contributors are active and how well it's tested.

Supported Platforms

Qt supports a variety of platforms including all major desktop and embedded platforms. Through the Qt Platform Abstraction, it's now easier than ever to port Qt to your own platform if required.

Testing Qt 6 on a platform is time-consuming. A subset 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. However, keep in mind that no code is error-free.

Qt Project

From the Qt Wiki (http://wiki.qt.io/) :

> “The Qt Wiki 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 Wiki is a place where Qt users and contributors share their insights. It forms the base for other users to contribute. The biggest contributor is The Qt Company, 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 will not comply with the open-source licenses. Without the commercial aspect, these companies would not be able to use Qt and it would not allow The Qt Company to contribute so much code to the Qt Project.

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

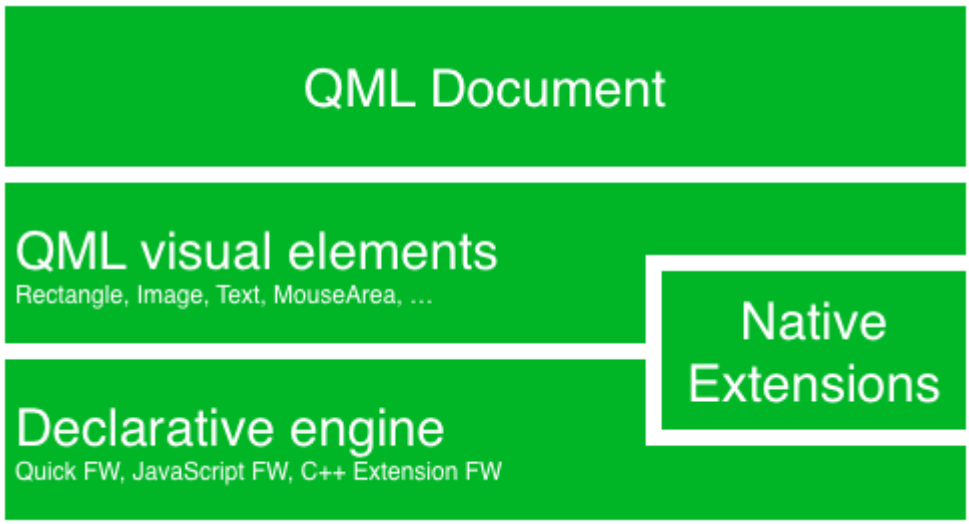
Contribute here: http://wiki.qt.io/

Qt 6 Introduction

Qt Quick

Qt Quick is the umbrella term for the user interface technology used in Qt 6. It was introduced in Qt 4 and now expanded for Qt 6. 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



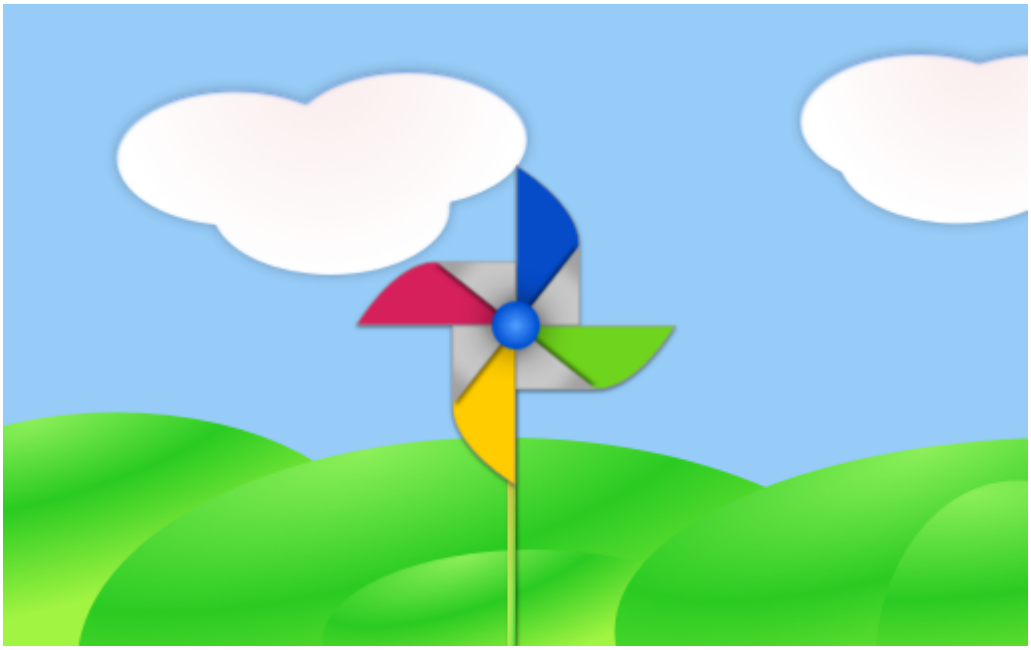
Similar to HTML, QML is a markup language. It is composed of tags, called types in Qt Quick, that are enclosed in curly brackets: `Item {}` . It was designed from the ground up for the creation of user interfaces, speed and easier reading for developers. The user interface can be enhanced further using 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. 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 Test, the Qt unit testing framework, and exported for the front-end developers to use.

Digesting a User Interface

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



We start with an empty document called `main.qml`. All our QML files will have the suffix `.qml`. As a markup language (like HTML), a QML document needs to have one and only one root type. In our case, this is

the `Image` type with a width and height based on the background image geometry:

```
import QtQuick

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

As QML doesn't restrict the choice of type for the root type, we use an `Image` type with the source property set to our background image as the root.

> **TIP**
>
> Each type has properties. For example, an image has the properties `width` and `height`, each holding a count of pixels. It also has other properties, such as `source`. Since the size of the image type is automatically derived from the image size, we don't need to set the `width` and `height` properties ourselves.

The most standard types are located in the `QtQuick` module, which is made available by the import statement at the start of the `.qml` file.

The `id` is a special and optional property that contains an identifier that can be used to reference its associated type elsewhere in the document. Important: An `id` property cannot be changed after it has been set, and it cannot be set during runtime. Using `root` as the id for the root-type is a convention used in this book to make referencing the top-most type predictable in larger QML documents.

The foreground elements, representing the pole and the pinwheel in the user interface, are included as separate images.

We want to place the pole horizontally in the center of the background, but offset vertically towards the bottom. And we want to place the pinwheel in the middle of the background.

Although this beginners example only uses image types, as we progress you will create more sophisticated user interfaces that are composed of many different types.

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

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

To place the pinwheel in the middle, we use a complex property called `anchor`. Anchoring allows you to specify geometric relations between

parent and sibling objects. For example, place me in the center of another type (`anchors.centerIn: parent`). There are left, right, top, bottom, centerIn, fill, verticalCenter and horizontalCenter relations on both ends. Naturally, when two or more anchors are used together, they should complement each other: it wouldn't make sense, for instance, to anchor a type's left side to the top of another type.

For the pinwheel, the anchoring only requires one simple anchor.

> **TIP**
>
> Sometimes you will want to make small adjustments, for example, to nudge a type slightly off-center. This can be done with `anchors.horizontalCenterOffset` or with `anchors.verticalCenterOffset` . Similar adjustment properties are also available for all the other anchors. Refer to the documentation for a full list of anchors properties.

> **TIP**
>
> Placing an image as a child type of our root type (the `Image`) illustrates an important concept of a declarative language. You describe the visual appearance of the user interface in the order of layers and grouping, where the topmost layer (our background image) is drawn first and the child layers are drawn on top of it in the local coordinate system of the containing type.

To make the showcase a bit more interesting, let's make the scene interactive. The idea is to rotate the wheel when the user presses the mouse somewhere in the scene.

We use the `MouseArea` type and make it cover the entire area of our root type.

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

The mouse area emits signals when the user clicks inside the area it covers. You can connect to this signal by overriding the `onClicked` function. When a signal is connected, it means that the function (or functions) it corresponds to are called whenever the signal is emitted. In this case, we say that when there's a mouse click in the mouse area, the type whose `id` is `wheel` (i.e., the pinwheel image) should rotate by +90 degrees.

> **TIP**
>
> This technique works for every signal, with the naming convention being `on` + `SignalName` in title case. Also, all properties emit a signal when their value changes. For these signals, the naming convention is:

```
`on${property}Changed`
```
js

For example, if a `width` property is changed, you can observe it with `onWidthChanged: print(width)`.

The wheel will now rotate whenever the user clicks, but the rotation takes place in one jump, rather than a fluid movement over time. We can achieve smooth movement using animation. An animation defines how a

property change occurs over a period of time. To enable this, we use the `Animation` type's property called `Behavior`. The `Behavior` specifies an animation for a defined property for every change applied to that property. In other words, whenever the property changes, the animation is run. This is only one of many ways of doing animation in QML.

```
Image {
    id: root
    Image {
        id: wheel
        Behavior on rotation {
            NumberAnimation {
                duration: 250
            }
        }
    }
}
```

Now, whenever the wheel's rotation property changes, it will be animated using a `NumberAnimation` with a duration of 250 ms. So each 90-degree turn will take 250 ms, producing a nice smooth turn.

> **TIP**
>
> You will not actually see the wheel blurred. This is just to indicate the rotation. (A blurred wheel is in the assets folder, in case you'd like to experiment with it.)

Now the wheel looks much better and behaves nicely, as well as providing a very brief insight into the basics of how Qt Quick programming works.

Quick Start

This chapter will introduce you to developing with Qt 6. 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.

Installing Qt 6 SDK

The Qt SDK includes the tools you need to build desktop or embedded applications. You can grab the latest version from the **Qt Company (https://qt.io)** 's homepage. There is an offline and online installer. The author personally prefers the online installer package as it allows you to install and update multiple Qt releases. This is the recommended way to start. The SDK itself has a maintenance tool, which allows you to update the SDK to the latest version.

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

When installing the SDK, you should select the default option and ensure that at least Qt 6.2 is enabled. Then you're ready to go.

Update Qt

The Qt SDK comes with an own maintenance tool located under the `${install_dir}` . It allows to add and/or update Qt SDK components.

Build from Source

To build Qt from source you can follow the guide from the **Qt Wiki (https://wiki.qt.io/Building_Qt_6_from_Git)** .

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 ▸ Other Project ▸ Qt Quick UI Prototype`) and name the project `HelloWorld` .

> **TIP**
>
> The Qt Creator IDE allows you to create various types of applications. If not otherwise stated, we always use a Qt Quick UI prototype project. For a production application you would often prefer a `CMake` based project, but for fast prototyping this type is better suited.

> **TIP**
>
> A typical Qt Quick application is made out of a runtime called the QmlEngine which loads the initial QML code. The developer can register C++ types with the runtime to interface with the native code. These C++ types can also be bundled into a plugin and then dynamically loaded using an import statement. The `qml` tool is a pre-made runtime which is used directly. For the beginning, we will not cover the native side of development and focus only on the QML aspects of Qt 6. This is why we start from a prototype project.

Qt Creator creates 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 it yourself.

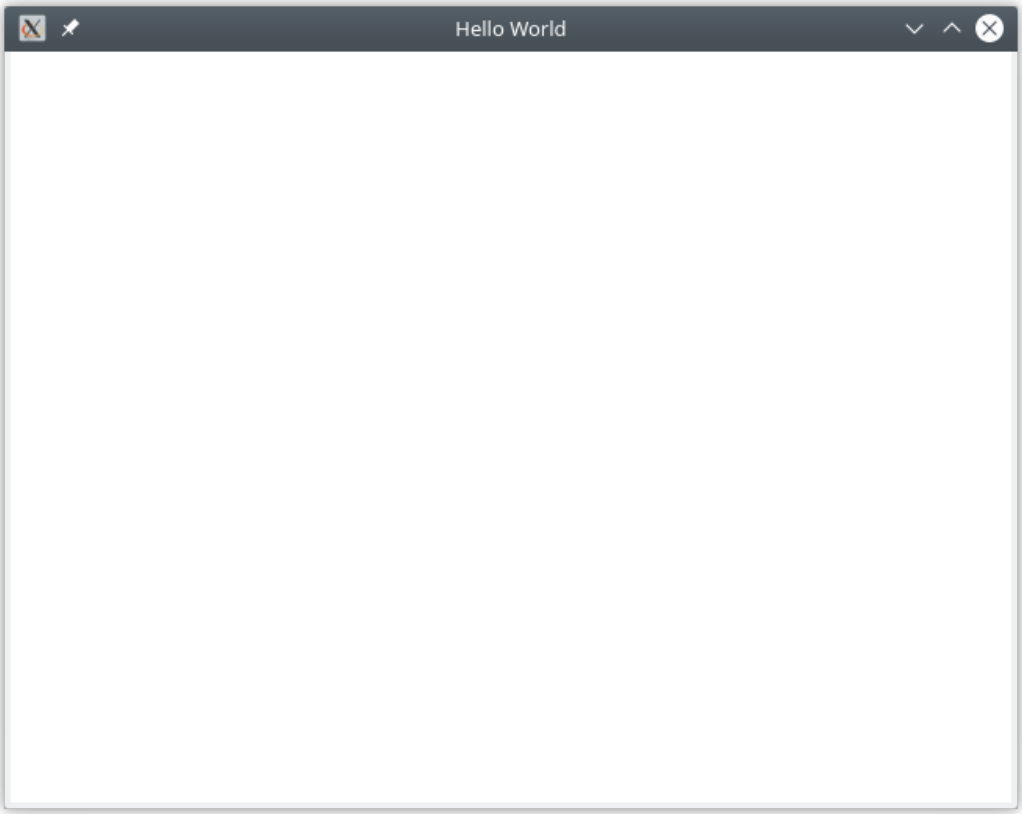
Another file, `HelloWorld.qml`, is our application code. Open it and try to understand what the application does before you read on.

```
// HelloWorld.qml

import QtQuick
import QtQuick.Window

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
}
```

The `HelloWorld.qml` program is written in the QML language. We'll discuss the QML language more in-depth in the next chapter. QML describes the user interface as a tree of hierarchical elements. In this case, a window of 640 x 480 pixels, with a window title "Hello World".

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

In the background, Qt Creator runs `qml` and passes your QML document as the first argument. The `qml` application parses the document, and launches the user interface. You should see something like this:



Qt 6 works! That means we're 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 for your specific device target.

TIP

Build from Scratch

If you'd like to build Qt 6 from the command line, you'll first need to grab a copy of the code repository and build it. Visit Qt's wiki for an up-to-date explanation of how to build Qt from git.

After a successful compilation (and 2 cups of coffee), Qt 6 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, you can now run the example with the default runtime that comes with Qt 6:

```
$ qtbase/bin/qml
```

Application Types

This section is a run through of different application types one can write with Qt 6. It's not limited to the selection presented here, but it will give you a better idea of what you can achieve with Qt 6 in general.

Console Application

A console application does not provide a graphical user interface, and is usually called as part of a system service or from the command line. Qt 6 comes with a series of ready-made components which help you create cross-platform console applications very efficiently. For example, the networking file APIs, string handling, and an efficient command line parser. As Qt is a high-level API on top of C++, you get programming speed paired with execution speed. Don't think of Qt as being *just* a UI toolkit - it has so much more to offer!

String Handling

This first example demonstrates how you could add 2 constant strings. Admittedly, this is not a very useful application, but it gives you an idea of what a native C++ application without an event loop may 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 << Qt::endl;
}
```

Container Classes

This example adds a list, and list iteration, to the application. Qt comes with a large collection of container classes that are easy to use, and has the same API paradigms as other 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 << "!" << Qt::endl;
```

Here is a more advanced list function, that allows 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 the `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 << Qt::endl;
```

File IO

In the next snippet, we read a CSV file from the local directory and loop over the rows to extract the cells from each row. Doing this, we get the table data from the CSV file in ca. 20 lines of code. File reading gives us a byte stream, to be able to convert this into 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 write mode, and pipe the lines into the text stream.

```
QList<QStringList> data;
// file operations
QFile file("sample.csv");
if(file.open(QIODevice::ReadOnly)) {
    QTextStream stream(&file);
    // loop forever macro
    forever {
        QString line = stream.readLine();
        // test for null string 'String()'
        if(line.isNull()) {
            break;
        }
        // test for empty string 'QString("")'
```

```
            if(line.isEmpty()) {
                continue;
            }
            QStringList row;
            // for each loop to iterate over containers
            foreach(const QString& cell, line.split(",")) {
                row.append(cell.trimmed());
            }
            data.append(row);
        }
    }
    // No cleanup necessary.
```

This concludes the section about console based applications with Qt.

C++ Widget Application

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

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

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

```
include <QtGui>

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

Custom Widgets

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

```
#pragma once

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;
};
```

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 and keyboard events change the internal state of the widget and trigger a painting update. We won't go into too much detail about this code, but it is good to know that you have the possibility. Qt comes with a large set of ready-made desktop widgets, so it's likely 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, with a native look on different operating systems. Your job, then, is to arrange these different widgets in a widget container into larger panels. A widget in Qt can also be a container for other widgets. This is accomplished through the parent-child relationship. This means we need to make our ready-made widgets, such as buttons, checkboxes, radio buttons, lists, and grids, children of other widgets. One way to accomplish this is displayed below.

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

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

In the implementation, we use layouts to better arrange our widgets. Layout managers re-layout the widgets according to some size policies

when the container widget is re-sized. In this example, we have a list, a line edit, and a button, which are arranged vertically and allow the user 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*)), th
    connect(m_edit, SIGNAL(editingFinished()), this, SLOT(updat
    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 remotely like geometrical objects, Qt graphics view is a good candidate. A graphics view arranges simple geometrical shapes in 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 is populated with graphics items.

Here is a short example. First the header file with the declaration of the view and scene.

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

};
```

In the implementation, the scene gets attached to the view first. The view is a widget and gets arranged in our container widget. In the end, we add a small rectangle to the scene, which is then 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::No
    rect1->setFlags(QGraphicsItem::ItemIsFocusable|QGraphicsIte
}
```

Adapting Data

Up to now, we have mostly covered basic data types and how to use widgets and graphics views. In your applications, you will often need a larger amount of structured data, which may also need to be stored persistently. Finally, the data also needs to be displayed. For this, Qt uses models. A simple model is the string list model, which gets filled with strings and then attached to a list view.

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

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

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

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

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

```
QT += sql
```

And then we can open our database using C++. First, we need to retrieve a new database object for the specified database engine. With this database object, we open the database. For SQLite, it's enough to specify the path to the database file. Qt provides some high-level database models, one of which is the table model. The table model uses a table identifier and an optional where clause to select the data. The resulting model can be attached to a list view as with 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 a higher level model operations, Qt provides a sorting file proxy model that allows you sort, filter, and transform models.

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

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

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

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

!!! note

```
This has been an overview of the different kind of classic appl
```

Coming next: Qt Quick to the rescue.

Qt Quick Application

There is an inherent conflict in modern software development. The user interface is moving much faster than our back-end services. In a traditional technology, you develop the so-called front-end at the same

pace as the back-end. This results in conflicts when customers want to change the user interface during a project, or develop the idea of a user interface during the project. Agile projects, require agile methods.

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

This is a simple Qt Quick UI below

```
import QtQuick

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

The declaration language is called QML and it needs a runtime to execute it. Qt provides a standard runtime called `qml` . You can also write a custom runtime. For this, we need a quick view and set the main QML document as a source from C++. Then you can show the user interface.

```
#include <QtGui>
#include <QtQml>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine("main.qml");
    return app.exec();
}
```

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

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

```
import QtQuick

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

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

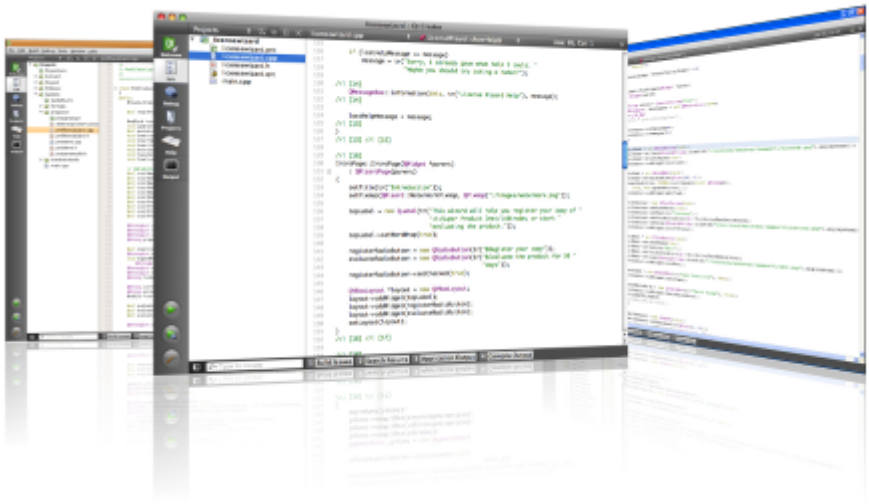
```
m_model = QSqlTableModel(this);
... // some magic code
QHash<int, QByteArray> roles;
roles[Qt::UserRole+1] = "city";
roles[Qt::UserRole+2] = "country";
m_model->setRoleNames(roles);
engine.rootContext()->setContextProperty("cityModel", m_model);
```

Summary

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

Qt Creator IDE

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



WARNING

Update screenshots!

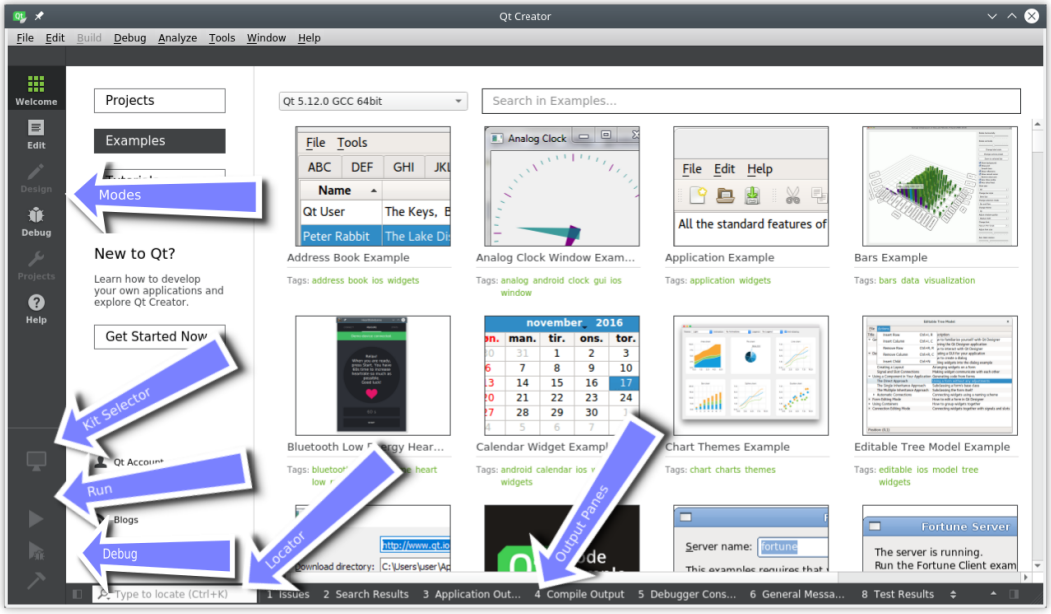
The User Interface

When starting Qt Creator you are greeted by the *Welcome* screen. There you will find the most important hints on how to continue inside Qt Creator and your recently used projects. You will also see the sessions list, which might be empty for you. A session is a collection of projects and configurations stored for fast access. 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 support typical steps from a developer workflow.

- **Welcome mode**: For your orientation.
- **Edit mode**: Focus on the code
- **Design mode**: Focus on the UI design
- **Debug mode**: Retrieve information about a running application
- **Projects mode**: Modify your projects run and build configuration
- **Analyze mode**: For detecting memory leaks and profiling
- **Help mode**: Easy access to the Qt documentation

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



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

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

Registering your Qt Kit

The Qt Kit is probably the most difficult aspect when it comes to 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 C++ compiler and a Qt version (e.g. Qt 6.xx.yy) 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. This is also true for Qt 6 where CMake is the preferred build tool.

Adding a kit and registering a Qt version is done in the `Settings ▸ Kits` entry. There you can also see which compilers are registered.

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

Managing Projects

Qt Creator manages your source code in projects. You can create a new project by using `File ▸ New File or Project` . When you create a project you have many choices of application templates. Qt Creator is capable of creating desktop, embedded, mobile applications and even python projects using Qt for Python. There are templates for applications which uses Widgets or Qt Quick or even bare-bone projects just using a console. For a beginner, it is difficult to choose, so we pick three project types for you.

- **Other Project / QtQuick UI Prototype**: Great for playing around with QML as there is no C++ build step involved. Mostly suitable for prototyping only.
- **Applications (Qt Quick) / Qt Quick Application (Empty)**: Creates a bare C++ project with cmake support and a QML main document to render an empty window. This is the typical default starting point for all native QML application.
- **Libraries / Qt Quick 2.0 Extension Plug-in**: Use this wizard to create a stub for a plug-in for your Qt Quick UI. A plug-in is used to extend Qt Quick with native elements. This is ideally to create a re-usable Qt Quick library.
- **Applications (Qt) / Qt Widgets Application**: Creates a starting point for a desktop application using Qt Widgets. This would be your starting point if you plan to create a traditional C++ widgets based application.

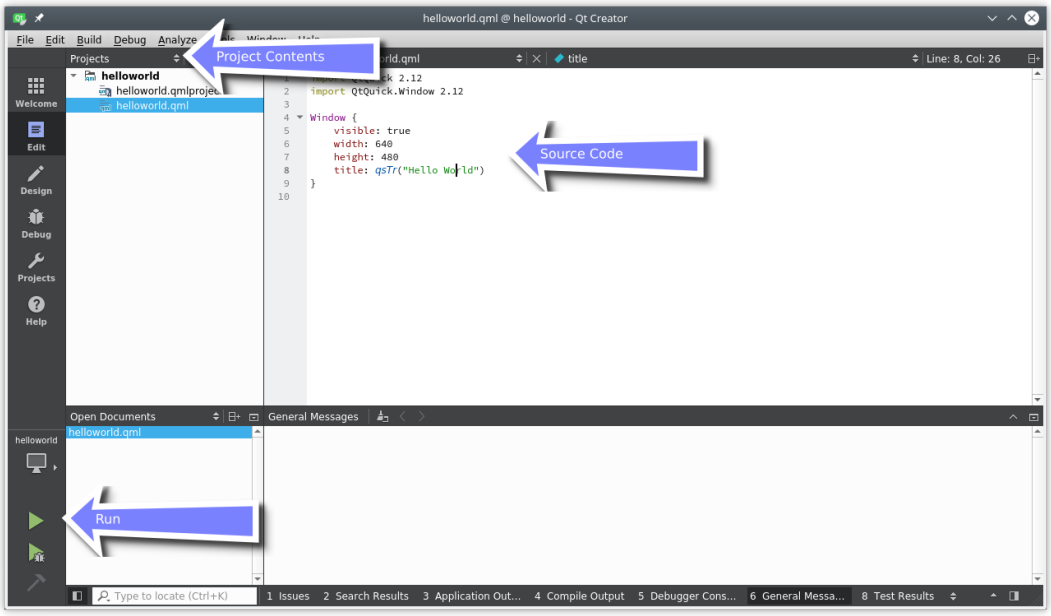
- **Applications (Qt) / Qt Console Application**: Creates a starting point for a desktop application without any user interface. This would be your starting point if you plan to create a traditional C++ command line tool using Qt C++.

> **TIP**
>
> During the first parts of the book, we will mainly use the **QtQuick UI Prototype** type or the **Qt Quick Application**, depending on whether we also use some C++ code with Qt Quick. Later to describe some c++ aspects we will use the **Qt Console Application** type. For extending Qt Quick with our own native plug-ins we will use the *Qt Quick 2.0 Extension Plug-in* wizard type.

Using the Editor

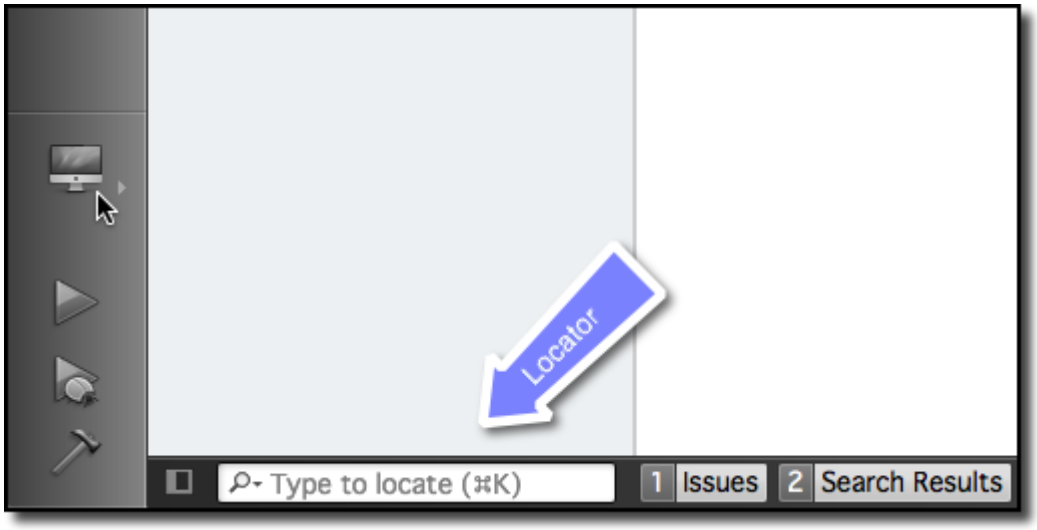
When you open a project or you just created a new project Qt Creator will switch to the edit mode. You should see on the left of your project files and in the center area the code editor. Selecting files on the left will open them in the editor.

The editor provides syntax highlighting, code-completion, and quick-fixes. Also, it supports several commands for code refactoring. When working with the editor you will have the feeling that everything reacts immediately. This is thanks to the developers of Qt Creator which made the tool feel really snappy.

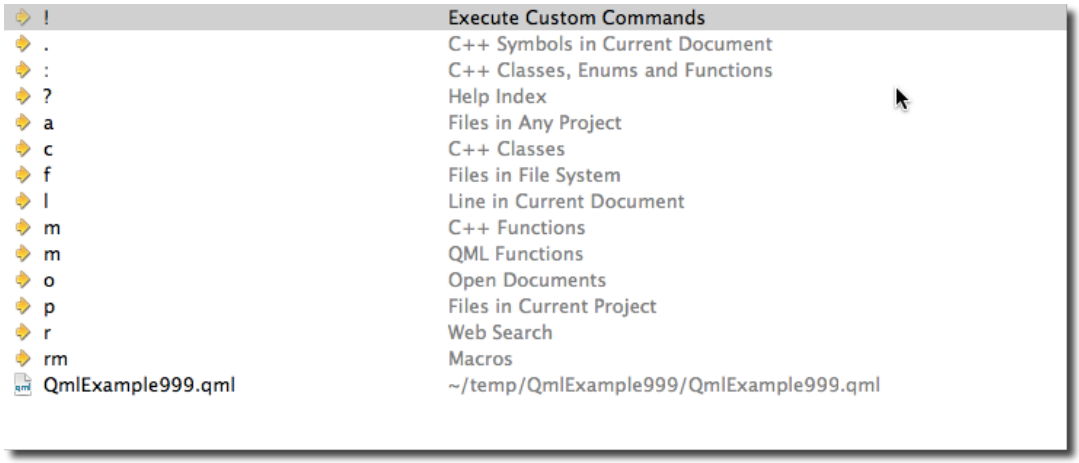


Locator

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



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



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

Debugging

Qt Creator is an easy to use and well designed IDE to code your Qt C++ and QML projects. It has world class `CMake` support and is pre-configured for Qt C++ development. Due to its excellent C++ support it can also be used for any other vanilla C++ projects.

> **TIP**
>
> 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 (http://http://doc.qt.io/qtcreator/index.html)** .

Shortcuts

Shortcuts are the difference between a nice-to-use editor and a professional editor. As a professional you spend hundreds 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.

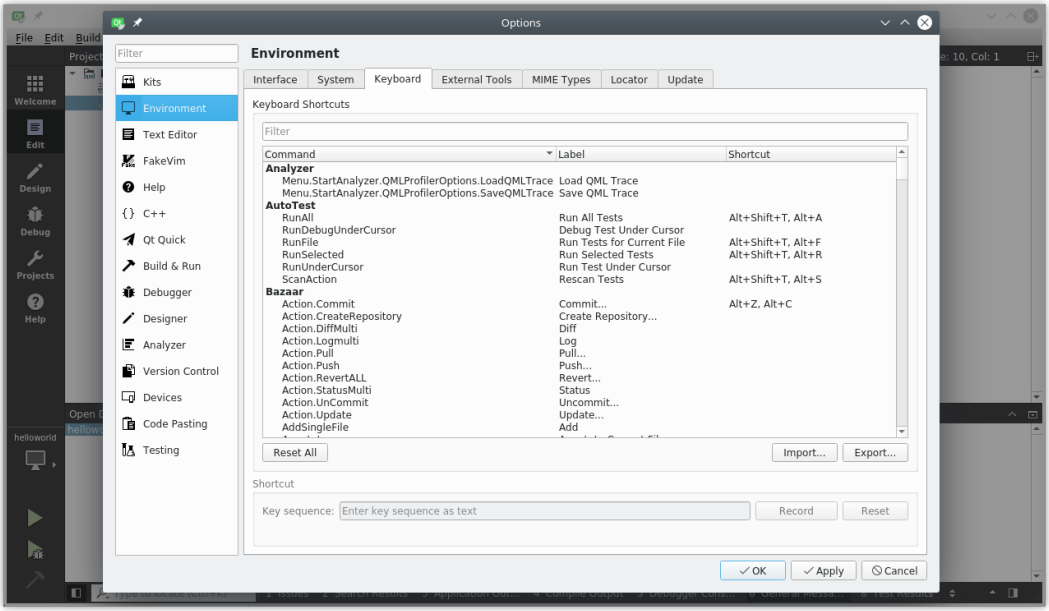
To get started we have collection some basic shortcuts (in Windows notation):

- `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 (http://doc.qt.io/qtcreator/creator-keyboard-shortcuts.html)** from the documentation.

Configure Shortcuts

You can configure the shortcuts from inside creator using the settings dialog.


Options
Filter
Kits
Environment
Text Editor
FakeVim
Help
C++
Qt Quick
Build & Run
Debugger
Designer
Analyzer
Version Control
Devices
Code Pasting
Testing
Environment
Interface
System
Keyboard
External Tools
MIME Types
Locator
Update
Keyboard Shortcuts
Filter
Command
Label
Shortcut
Analyzer
Menu.StartAnalyzer.QMLProfilerOptions.LoadQMLTrace
Load QML Trace
Menu.StartAnalyzer.QMLProfilerOptions.SaveQMLTrace
Save QML Trace
AutoTest
RunAll
Run All Tests
Alt+Shift+T, Alt+A
RunDebugUnderCursor
Debug Test Under Cursor
RunFile
Run Tests for Current File
Alt+Shift+T, Alt+F
RunSelected
Run Selected Tests
Alt+Shift+T, Alt+R
RunUnderCursor
Run Test Under Cursor
ScanAction
Rescan Tests
Alt+Shift+T, Alt+S
Bazaar
Action.Commit
Commit...
Alt+Z, Alt+C
Action.CreateRepository
Create Repository...
Action.DiffMulti
Diff
Action.Logmulti
Log
Action.Pull
Pull...
Action.Push
Push...
Action.RevertALL
Revert...
Action.StatusMulti
Status
Action.UnCommit
Uncommit...
Action.Update
Update...
AddSingleFile
Add
Reset All
Import...
Export...
Shortcut
Key sequence:
Enter key sequence as text
Record
Reset
OK
Apply
Cancel

Quick Starter

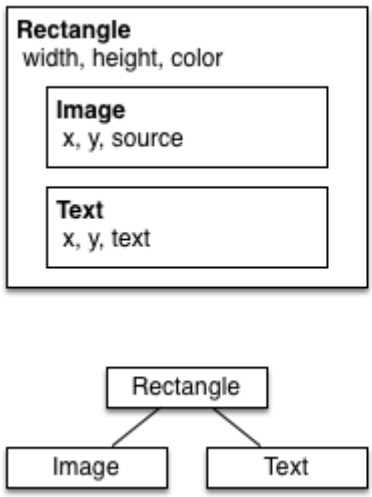
This chapter provides an overview of QML, the declarative user interface language used in Qt 6. We will discuss the QML syntax, which is a tree of elements, followed by an overview of the most important basic elements. Later we will briefly look at how to create our own elements, called components and how to transform elements using property manipulations. Towards the end, we will look at how to arrange elements together in a layout and finally have a look at elements where the user can provide input.

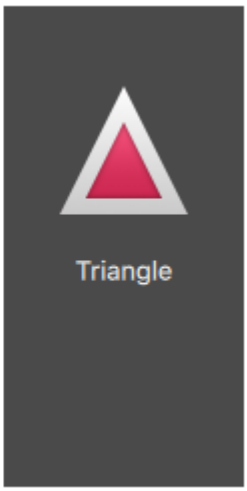
QML Syntax

QML is a declarative language used to describe how objects relate to each other. QtQuick is a framework built on QML for building the user interface of your application. It breaks down the user interface into smaller elements, which can be combined into components. QtQuick 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 and QtQuick are designed from the ground up to describe user interfaces, not text-documents.

In its simplest form, QtQuick lets you create a hierarchy of elements. Child elements inherit the coordinate system from the parent. An `x, y` coordinate is always relative to the parent.

> **TIP**
>
> QtQuick builds on QML. The QML language only knows of elements, properties, signals and bindings. QtQuick is a framework built on QML. Using default properties, the hierarchy of QtQuick elements can be constructed in an elegant way.





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

```
// RectangleExample.qml

import QtQuick

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

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

    // color property
    color: "#4A4A4A"

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

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

        source: 'assets/triangle_red.png'
```

```
    }

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

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

        // reference root element
        width: root.width

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

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

With the `import` statement you import a QML module by name. In Qt5 you had to specify a major and minor version (e.g. `2.15`), this is now optional in Qt6. For the book content we drop this optional version

number as normally you automatically want to choose the newest version available from your selected Qt Kit.

> **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.

> **TIP**
>
> You can run the example using the Qt Quick runtime from the command line from your OS like this:
>
> ```
> $ $QTDIR/bin/qml RectangleExample.qml
> ```
>
> Where you need to replace the `$QTDIR` to the path to your Qt installation. The `qml` executable initializes the Qt Quick runtime and interprets the provided QML file.
>
> In Qt Creator, you can open the corresponding project file and run the document `RectangleExample.qml` .

Properties

Elements are declared by using their element name but are defined by using their properties or by creating custom properties. A property is a simple key-value pair, e.g. `width: 100` , `text: 'Greetings'` , `color:`

`'#FF0000'` . A property has a well-defined type and can have an initial value.

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

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

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

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

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

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

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

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

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

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

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

Let's go through the different features of properties:

- **(1)** `id` is a very special property-like value, it is used to reference elements inside a QML file (called "document" in QML). The `id` is not a string type but rather an identifier and part of the QML syntax. An `id` needs to be unique inside a document and it can't be reset to a different value, nor may it be queried. (It behaves much like a reference 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 new 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 default initial value is chosen.

> **TIP**
>
> You can also declare one property to be the default property using `default` keyword. If another element is created inside the element and not explicitly bound to a property, it is bound to the default property. For instance, This is used 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 in 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 belong to the element class itself. This is done for global settings elements which appear only once in the application (e.g. keyboard input). The writing is `<Element>.<property>: <value>` .

- **(9)** For every property, you can provide a signal handler. This handler is called after the property changes. For example, here we want to be notified whenever the height changes and use the built-in console to log a message to the system.

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

Scripting

QML and JavaScript (also known as ECMAScript) are best friends. In the *JavaScript* chapter we will go into more detail on this symbiosis. Currently, we just want to make you aware of 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. Need to use function to ca
    onTextChanged: function(text) {
```

```
        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 the space bar being pressed. As we use a parameter injected by the signal, we need to use the function syntax here. It's also possible to use an arrow function (`(text) => {}`), but we feel `function(text) {}` is more readable.
- **(2)** When the text element receives the space 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.

Binding

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 on the property or even when a JavaScript value is assigned to the property. For example, a key handler setting the text property to an empty string would destroy our increment display:

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

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

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

Core Elements

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

Currently, we will focus on the fundamental visual elements, such as `Item`, `Rectangle`, `Text`, `Image` and `MouseArea`. However, by using the Qt Quick Controls 2 module, it is possible to create user interfaces built from standard platform components such as buttons, labels and sliders.

Item Element

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

- **Geometry** - `x` and `y` to define the top-left position, `width` and `height` for the expansion of the element, and `z` for the stacking order to lift elements up or down from their natural ordering.
- **Layout handling** - `anchors` (left, right, top, bottom, vertical and horizontal center) to position elements relative to other elements with optional `margins`.
- **Key handling** - attached `Key` and `KeyNavigation` properties to control key handling and the `focus` property to enable key handling in the first place.

- **Transformation** - `scale` and `rotate` transformation and the generic `transform` property list for *x,y,z* transformation, as well as a `transformOrigin` point.
- **Visual** - `opacity` to control transparency, `visible` to show/hide elements, `clip` to restrain paint operations to the element boundary, and `smooth` to enhance the rendering quality.
- **State definition** - `states` list property with the supported list of states, the current `state` property, and the `transitions` list property to animate state changes.

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

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

Rectangle Element

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

```
Rectangle {
    id: rect1
    x: 12; y: 12
    width: 76; height: 96
    color: "lightsteelblue"
}
Rectangle {
```

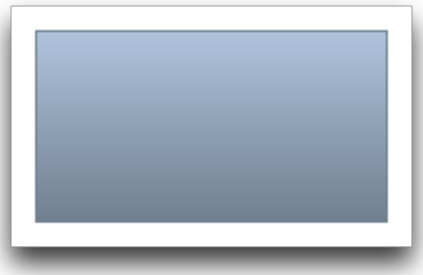
```
    id: rect2
    x: 112; y: 12
    width: 76; height: 96
    border.color: "lightsteelblue"
    border.width: 4
    radius: 8
}
```

> **TIP**
>
> Valid color values are colors from the SVG color names (see **http://www.w3.org/TR/css3-color/#svg-color (http://www.w3.org/TR/css3-color/#svg-color)**). You can provide colors in QML in different ways, but the most common way is an RGB string ('#FF4444') or as a color name (e.g. 'white').
>
> A random color can be created using some JavaScript:
>
> ```
> color: Qt.rgba( Math.random(), Math.random(), Math.rando
> ```

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.

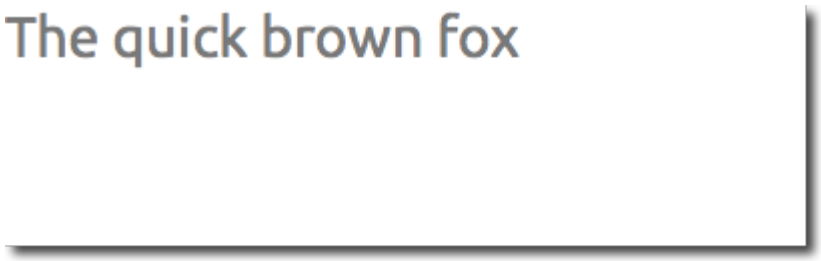
> **TIP**
>
> A rectangle with no *width/height* set will not be visible. This happens often when you have several rectangles width (height) depending on each other and something went wrong in your composition logic. So watch out!

> **TIP**
>
> 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 a rotated rectangle will not change and thus will lead to confusion as the geometry of the element is not the same as the visible area. From the author's perspective, it's really better to use designed gradient images in that case.

Text Element

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

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



Text can be aligned to each side and the center using the `horizontalAlignment` and `verticalAlignment` properties. To further enhance the text rendering you can use the `style` and `styleColor` property, which allows you to 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 the 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, and the remaining space it occupies is transparent. This means it does not render any background decoration, and so it's up to you to provide a sensible background if desired.

> **TIP**
>
> Be aware that the initial width of a `Text` item is dependant on the font and text string that were set. A `Text` element with no width set and no text will not be visible, as the initial width will be 0.

> **TIP**
>
> Often when you want to layout `Text` elements you need to differentiate between aligning the text inside the `Text` element boundary box and aligning the element boundary box itself. In the former, you want to use the `horizontalAlignment` and `verticalAlignment` properties, and in the latter case, you want to manipulate the element geometry or use anchors.

Image Element

An `Image` element is able to display images in various formats (e.g. `PNG`, `JPG`, `GIF`, `BMP`, `WEBP`). *For the full list of supported image formats, please consult the **Qt documentation (https://doc.qt.io/qt-6/qimagereader.html#supportedImageFormats)***. Besides the `source` property to provide the image URL, it contains a `fillMode` which controls the resizing behavior.

```
Image {
    x: 12; y: 12
    // width: 72
    // height: 72
    source: "assets/triangle_red.png"
}
Image {
```

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

TIP

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

TIP

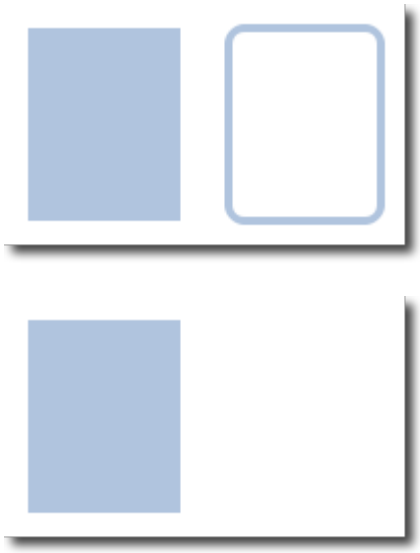
`Image` elements using `PreserveAspectCrop` should also enable clipping to avoid image data being rendered outside the `Image` boundaries. By default clipping is disabled (`clip: false`). You need to enable clipping (`clip: true`) to constrain the painting to the elements bounding rectangle. This can be used on any visual element, but **should be used sparingly (https://doc.qt.io/qt-6/qtquick-performance.html#clipping)** .

> **TIP**
>
> Using C++ you are able to create your own image provider using `QQuickImageProvider`. This allows you to create images on the fly and make use of threaded image loading.

MouseArea Element

To interact with these elements you will often use a `MouseArea`. It's a rectangular invisible item in which 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
}
```



TIP

This is an important aspect of Qt Quick: the input handling is separated from the visual presentation. This allows you to show the user an interface element where the actual interaction area can be larger.

TIP

For more complex interaction, see **Qt Quick Input Handlers (https://doc.qt.io/qt-6/qtquickhandlers-index.html)** . They are intended to be used instead of elements such as `MouseArea` and `Flickable` and offer greater control and flexibility. The idea is to handle one interaction aspect in each handler instance instead of centralizing the handling of all events from a given source in a single element, which was the case before.

Components

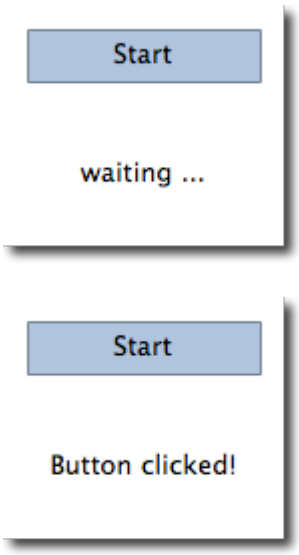
A component is a reusable element. QML provides different ways to create components. Currently, we will look only at the simplest form - a file-based component. A file-based component is created by placing a QML element in a file and giving the file an element name (e.g. `Button.qml`). You can use the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

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

```
Rectangle { // our inlined button ui
    id: button
    x: 12; y: 12
    width: 116; height: 26
    color: "lightsteelblue"
    border.color: "slategrey"
    Text {
        anchors.centerIn: parent
        text: "Start"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            status.text = "Button clicked!"
        }
    }
}
```

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

The UI will look similar to this. In the first image, the UI is in its initial state, and in the second image the button has been clicked.



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

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

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

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

```
// Button.qml

import QtQuick

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

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

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

We have exported the text property and the clicked signal at the root level. Typically we name our root element root to make referencing it

easier. We use the `alias` feature of QML, which is a way to export properties inside nested QML elements to the root level and make this available for the outside world. It is important to know that only the root level properties can be accessed from outside this file by other components.

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

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

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

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

TIP

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

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

    property alias text: label.text
    signal clicked

    Rectangle {
        anchors.fill parent
        color: "lightsteelblue"
        border.color: "slategrey"
    }
    ...
}
```

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

Simple Transformations

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

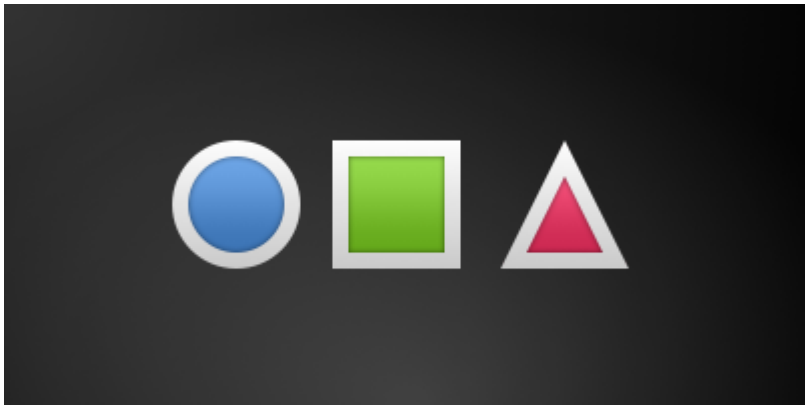
Let's start with the simple transformations. Here is our scene as our starting point.

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

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

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

import QtQuick

Image {
    id: root
    signal clicked
```

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

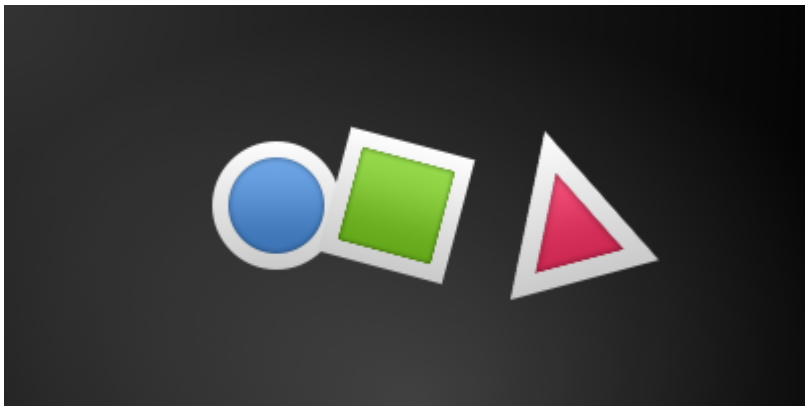


We use our clickable image to present three objects (box, circle, triangle). Each object performs a simple transformation when clicked. Clicking the background will reset the scene.

```
// TransformationExample.qml

import QtQuick

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 e
```

```
    // and consume the mouse events
    anchors.fill: parent
    onClicked: {
        // reset our little scene
        circle.x = 84
        box.rotation = 0
        triangle.rotation = 0
        triangle.scale = 1.0
    }
}

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

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

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

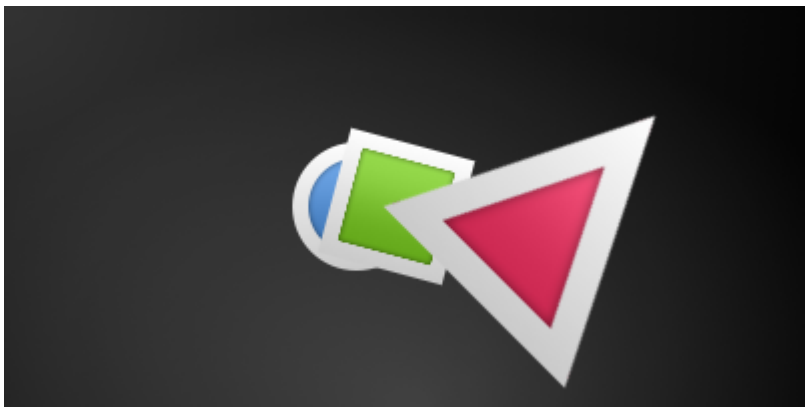
```
            rotation += 15
            scale += 0.05
        }
    }

    // ...
```



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

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

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

> **TIP**
>
> Elements which appear earlier in the code have a lower stacking order (called z-order). If you click long enough on `circle` you will see it moves below `box` . The z-order can also be manipulated by the `z` property of an Item.
>
> 
>
> This is because `box` appears later in the code. The same applies also to mouse areas. A mouse area later in the code will overlap (and thus grab the mouse events) of a mouse area earlier in the code.
>
> Please remember: *the order of elements in the document matters.*

Positioning Elements

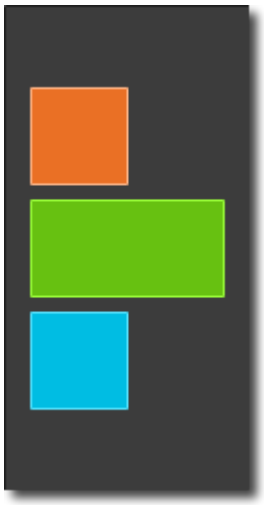
There are a number of QML elements used to position items. These are called positioners, of which the Qt Quick module provides the following: `Row`, `Column`, `Grid` and `Flow`. They can be seen showing the same contents in the illustration below.

> **TIP**
>
> Before we go into details, let me introduce some helper elements: the red, blue, green, lighter and darker squares. Each of these components contains a 48x48 pixel colorized rectangle. As a reference, here is the source code for the `RedSquare`:

```
// RedSquare.qml

import QtQuick

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 readable. Please remember, each rectangle is initially 48x48 pixels.

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



```
// ColumnExample.qml

import QtQuick

DarkSquare {
    id: root
    width: 120
    height: 240

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

The `Row` element places its child items next to each other, either from the left to the right, or from the right to the left, depending on the

`layoutDirection` property. Again, `spacing` is used to separate child items.

```
// RowExample.qml

import QtQuick

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.

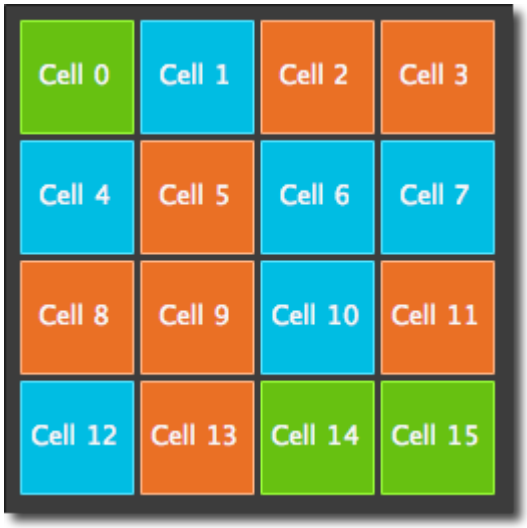
```
// GridExample.qml

import QtQuick

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 though anchor layouts.

```
// FlowExample.qml

import QtQuick

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 number of loops.




```
// RepeaterExample.qml

import QtQuick

DarkSquare {
    id: root
    width: 252
    height: 252
    property variant colorArray: ["#00bde3", "#67c111", "#ea702

    Grid{
        anchors.fill: parent
        anchors.margins: 8
        spacing: 4
        Repeater {
            model: 16
            delegate: Rectangle {
                required property int index
                property int colorIndex: Math.floor(Math.random

                width: 56; height: 56
                color: root.colorArray[colorIndex]
                border.color: Qt.lighter(color)

                Text {
                    anchors.centerIn: parent
```

```
                    color: "#f0f0f0"
                    text: "Cell " + parent.index
                }
            }
        }
    }
}
```

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

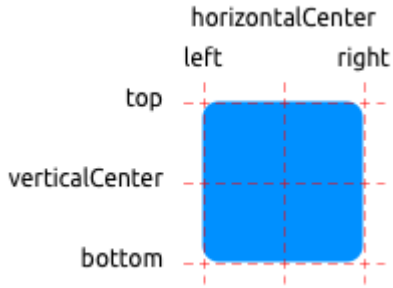
> **TIP**
>
> While the `index` property is dynamically injected into the Rectangle, it is a good practice to declare it as a required property to ease readability and help tooling. This is achieved by the `required property int index` line.

TIP

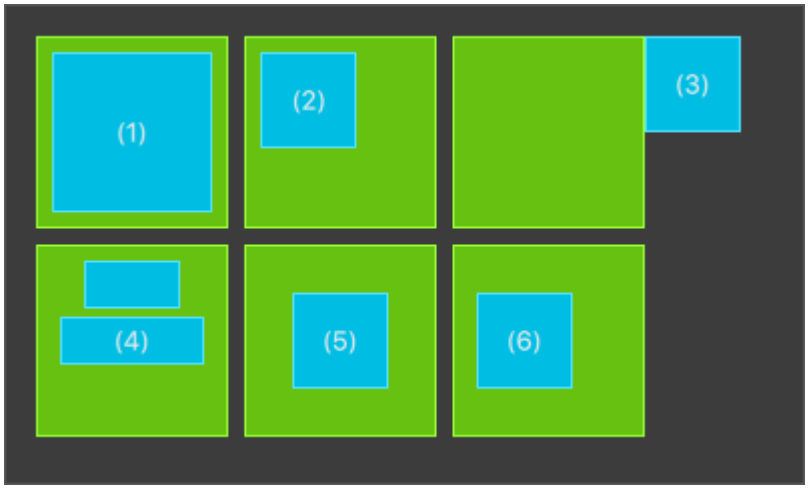
More advanced handling of larger models and kinetic views with dynamic delegates is covered in its own model-view chapter. Repeaters are best used when having a small amount of static data to be presented.

Layout Items

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



An element has 6 major anchor lines (`top`, `bottom`, `left`, `right`, `horizontalCenter`, `verticalCenter`). Additionally, there is the `baseline` anchor for text in `Text` elements. Each anchor line comes with an offset. In the case of the `top`, `bottom`, `left`, and `right` anchors, 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's left side is aligned to the parent's right side.

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

- **(4)** Center-aligned elements. `Blue1` is horizontally centered on the parent. `Blue2` is also horizontally centered, but on `Blue1`, and its top is aligned to the `Blue1` bottom line.

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

- **(5)** An element is centered on a parent element

```
GreenSquare {
    BlueSquare {
        width: 48
        anchors.centerIn: parent
```

```
            text: '(5)'
        }
    }
```

- **(6)** An element is centered with a 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)'
        }
    }
```

Hidden Gems

Our squares have been magically 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 (although 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. The same applies to (3). (4) can only be dragged vertically, as both squares are horizontally centered. (5) is centered on the parent, and as such, can't be dragged. The same applies to (6). Dragging an element means changing its `x,y` position. As anchoring is stronger than setting the `x,y` properties, dragging is restricted by the anchored lines. We will see this effect later when we discuss animations.

Input Elements

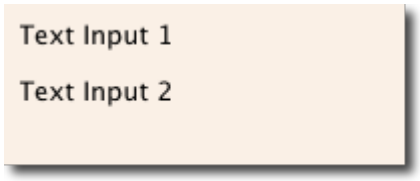
We have already used the `MouseArea` as a mouse input element. Next, we'll focus on keyboard input. We start off with the text editing elements: `TextInput` and `TextEdit`.

TextInput

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

```
// textinput.qml

import QtQuick

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

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 beside a blinking cursor and the entered text. For the user to be able to recognize the element as an input element it needs some visual decoration; for example, a simple rectangle. When placing the `TextInput` inside an element you need make sure you export the major properties you want others to be able to access.

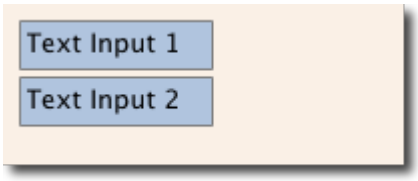
We move this piece of code into our own component called `TLineEditV1` for reuse.

```
// TLineEditV1.qml

import QtQuick

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
    }
}
```

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

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

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



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 is that when 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 when the focus scope receives the focus. So it forwards the focus to the last focus-requesting child element. We will create a second version of our TLineEdit component called TLineEditV2, using a focus scope as the root element.

```
// TLineEditV2.qml

import QtQuick

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

    }

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

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

Our example now looks 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

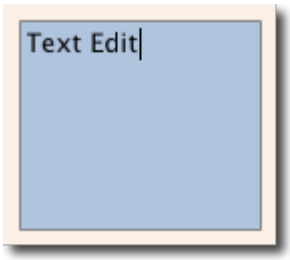
The `TextEdit` is very similar to `TextInput`, and supports a multi-line text edit field. It doesn't have the text constraint properties, as this depends on querying the content size of the text (`contentHeight`, `contentWidth`). We also create our own component called `TTextEdit` to provide an editing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

```
// TTextEdit.qml

import QtQuick

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

Rectangle {
    width: 136
    height: 120
    color: "linen"

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



Keys Element

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

```
// keys.qml

import QtQuick

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: function (event) {
        switch(event.key) {
            case Qt.Key_Plus:
                square.scale += 0.2
                break;
            case Qt.Key_Minus:
                square.scale -= 0.2
```

```
                break;
            }
        }
    }
```

Advanced Techniques

Performance of QML

QML and Javascript are interpreted languages. This means that they do not have to be processed by a compiler before being executed. Instead, they are being run inside an execution engine. However, as interpretation is a costly operation, various techniques are used to improve performance.

The QML engine uses just-in-time (JIT) compilation to improve performance. It also caches the intermediate output to avoid having to recompile. This works seamlessly for you as a developer. The only trace of this is that files ending with `qmlc` and `jsc` can be found next to the source files.

If you want to avoid the initial start-up penalty induced by the initial parsing you can also pre-compile your QML and Javascript. This requires you to put your code into a Qt resource file, and is described in detail in the **Compiling QML Ahead of Time (https://doc.qt.io/qt-6/qtquick-deployment.html#ahead-of-time-compilation)** chapter in the Qt documentation.

Fluid Elements

Until now, we have mostly looked at some simple graphical elements and how to arrange and manipulate them.

This chapter is about how to make these changes more interesting by animating them.

Animations are one of the key foundations for modern, slick user interfaces, and can be employed in your user interface via states, transitions and animations. Each state defines a set of property changes and can be combined with animations on state changes. These changes are described as a transition from one state to another state.

Besides animations being used during transitions, they can also be used as standalone elements triggered by some scripted events.

Animations

Animations are applied to property changes. An animation defines the interpolation curve from one value to another value when a property value changes. These animation curves 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 a duration. All animations in Qt Quick are controlled by the same timer and are therefore synchronized. This improves the performance and visual quality of animations.

> **Animations control how properties change using 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. In the book, you will see this is a spectacular playing field.
>
> You will catch yourself looking at some animations and just admiring their beauty, and your creative genius, too. Please remember then: *animations control property changes and every element has dozens of properties at your disposal*.
>
> **Unlock the power!**

```
// AnimationExample.qml

import QtQuick

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

    property int padding: 40
    property int duration: 4000
    property bool running: false

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

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

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

```
        }
    }
```

The example above shows a simple animation applied on the `x` and `rotation` properties. Each animation has a duration of 4000 milliseconds (msec). The animation on `x` moves the x-coordinate from the object gradually over to 240px. The animation on rotation runs from the current angle to 360 degrees. Both animations run in parallel and are started when the `MouseArea` is clicked.

You can play around with the animation by changing the `to` and `duration` properties, or you could add another animation (for example, on the `opacity` or even the `scale`). **Combining these it could look like the object is disappearing into deep space. Try it out!**

Animation Elements

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

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

Besides these basic and widely used animation elements, Qt Quick also provides 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 alongside a path
- `Vector3dAnimation` - Animates changes in QVector3d values

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

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

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

Applying Animations

Animation can be applied in several ways:

- **Animation on property** - runs automatically after the element is fully loaded

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

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

Clickable Image V2

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

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

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

```
        }
    }

    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 text and image source properties, and a clicked signal. We also wanted the text to be as wide as the image, and for it to wrap. We achieve the latter by using the Text element's `wrapMode` property.

> **Parent/child geometry dependency**
>
> 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.
>
> You should prefer the child's geometry to depend on the parent's geometry if the item is more like a container for other items and should adapt to the parent's geometry.

The objects ascending

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

First object

The first object travels using the `Animation on <property>` strategy. The animation starts immediately.

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

When an object is clicked, its y-position is reset to the start position, and this applies to all of the objects. On the first object, the reset does not

have any effect as long as the animation is running.

This can be visually disturbing, as the y-position is set to a new value for a fraction of a second before the animation starts. *Such competing property changes should be avoided.*

Second object

The second object travels using a `Behavior on` animation. This behavior tells the property it should animate each change in value. The behavior can be disabled by setting `enabled: false` on the `Behavior` element.

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

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

The object will start traveling when you click it (its y-position is then set to 40). Another click has no influence, as the position is already set.

You could try to use a random value (e.g. `40 + (Math.random() \* (205-40)`) for the y-position. You will see that the object will always animate to the new position and adapt its speed to match the 4 seconds to the destination defined by the duration of the animation.

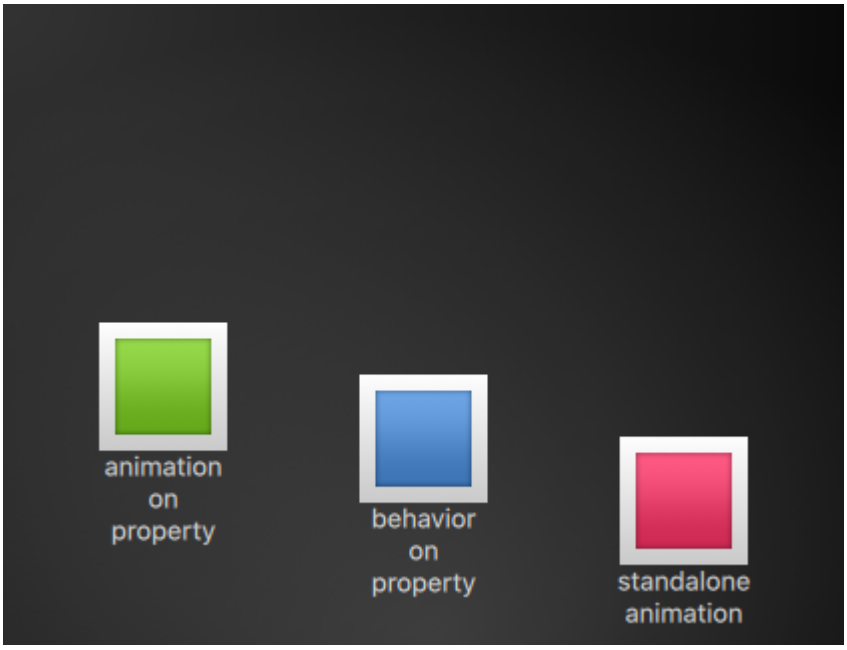
Third object

The third object uses a standalone animation. The animation is defined as its own element and can be almost anywhere in the document.

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

    text: qsTr("standalone animation")

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

The click will start the animation using the animation's `start()` function. Each animation has start(), stop(), resume(), and restart() functions. The animation itself contains much more information than the other animation types earlier.

We need to define the `target`, which is the element to be animated, along with the names of the properties that we want to animate. We also need to define a `to` value, and, in this case, a `from` value, which allows a restart of the animation.



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

> **Other ways to control Animations**
>
> 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

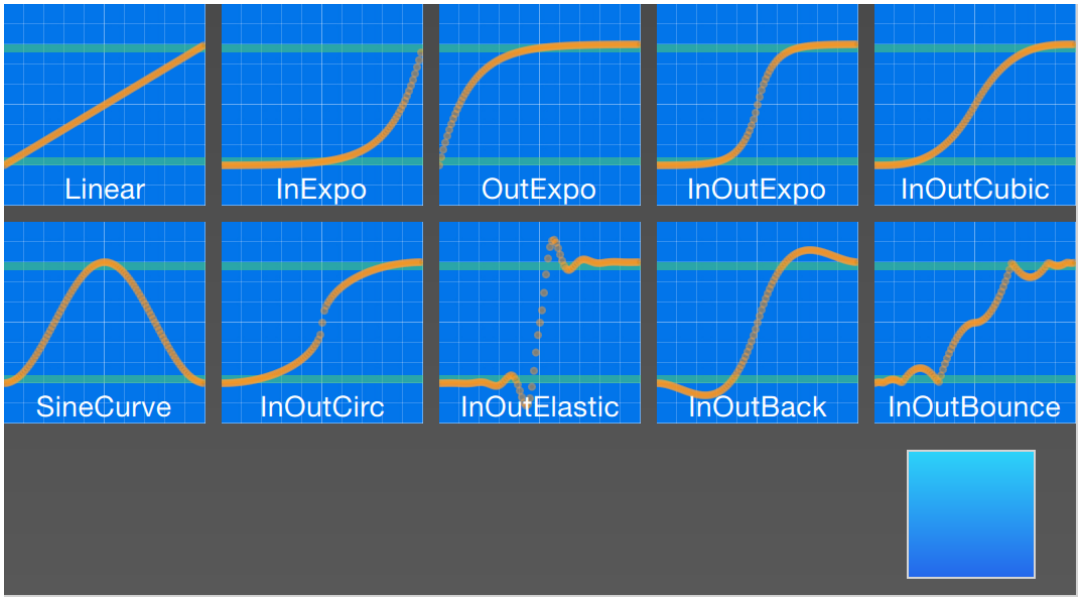
The value change of a property can be controlled by an animation. Easing attributes allow influencing the interpolation curve of a property change.

All animations we have defined by now use a linear interpolation because the initial easing type of an animation is `Easing.Linear`. It's best visualized with a small plot, where the y-axis is the property to be animated and the x-axis is the time (*duration*). A linear interpolation would draw a straight line from the `from` value at the start of the animation to the `to` value at the end of the animation. So the easing type defines the curve of change.

Easing types should be carefully chosen to support a natural fit for a moving object. For example, when a page slides out, the page should initially slide out slowly and then gain momentum to finally slide out at high speed, similar to turning the page of a book.

> **Animations should not be overused.**
>
> As with other aspects of UI design, animations should be designed carefully to support the UI flow, not dominate it. The eye is very sensitive to moving objects and animations can easily distract the user.

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



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

> **TIP**
>
> The internals of the EasingType renders the curve in real time, and the interested reader can look it up in the `EasingCurves` example.

```
// EasingCurves.qml

import QtQuick
import QtQuick.Layouts

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

```
height: childrenRect.height

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

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

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

```
                    animation.easing.type = easingType
                    box.toggle = !box.toggle
                }
            }
            EasingType {
                easingType: Easing.InOutBounce
                title: "InOutBounce"
                onClicked: {
                    animation.easing.type = easingType
                    box.toggle = !box.toggle
                }
            }
        }
        Item {
            height: 80
            Layout.fillWidth: true
            Box {
                id: box
                property bool toggle
                x: toggle ? 20 : root.width - width - 20
                anchors.verticalCenter: parent.verticalCenter
                gradient: Gradient {
                    GradientStop { position: 0.0; color: "#2ed5
                    GradientStop { position: 1.0; color: "#2467
                }
                Behavior on x {
                    NumberAnimation {
                        id: animation
                        duration: 500
                    }
                }
            }
        }
    }
}
```

Please play with the example and 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` type (from which most animations inherit) additionally supports `easing.amplitude`, `easing.overshoot`, and `easing.period` properties, which allow you to fine-tune the behavior of particular easing curves.

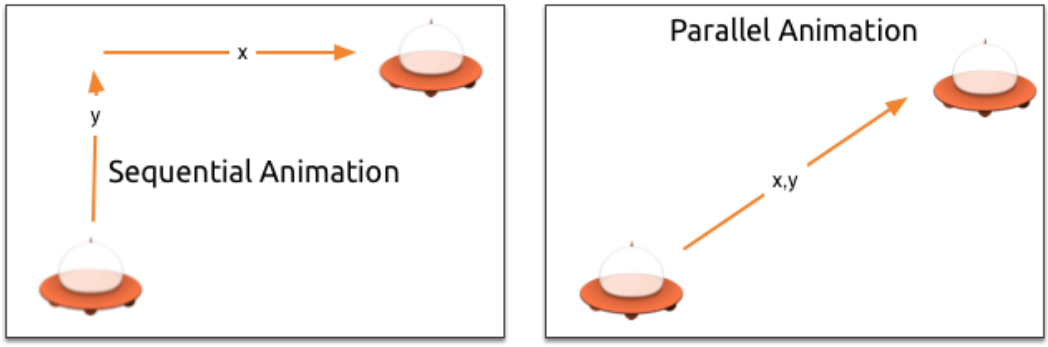
Not all easing curves support these parameters. Please consult the **easing table (http://doc.qt.io/qt-6/qml-qtquick-propertyanimation.html#easing-prop)** from the `PropertyAnimation` documentation to check if an easing parameter has an influence on an easing curve.

> **Choose the right Animation**
>
> 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. You might want to run several animations at the same time or one after another or even execute a script between two animations.

For this, grouped animations can be used. As the name suggests, it's possible to group animations. Grouping can be done in two ways: parallel or sequential. You can use the `SequentialAnimation` or the `ParallelAnimation` element, which act as animation containers for other animation elements. These grouped animations are animations themselves and can be used exactly as such.

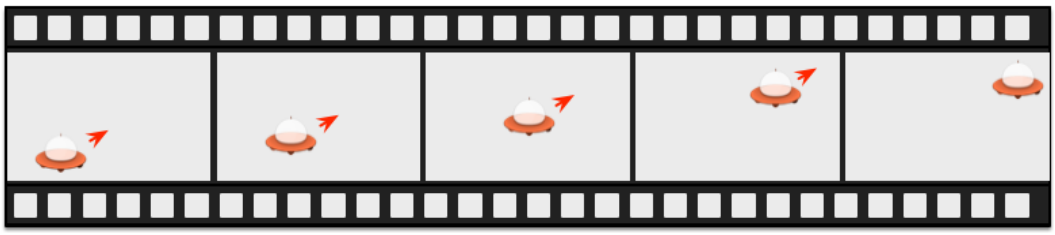


Parallel animations

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

```
// ParallelAnimationExample.qml
import QtQuick

BrightSquare {
    id: root

    property int duration: 3000
    property Item ufo: ufo

    width: 600
    height: 400

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

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

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

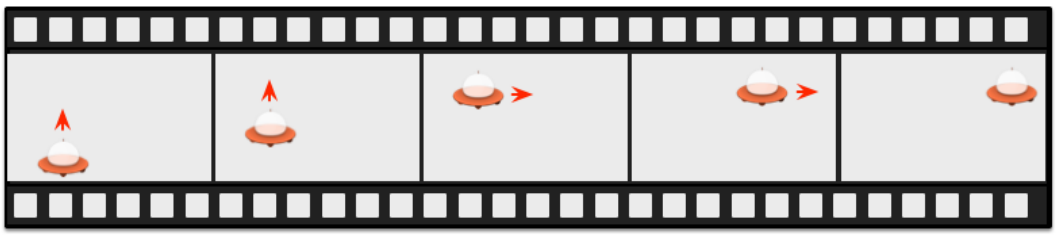


Sequential animations

A sequential animation runs each child animation in the order in which it is declared: top to bottom.

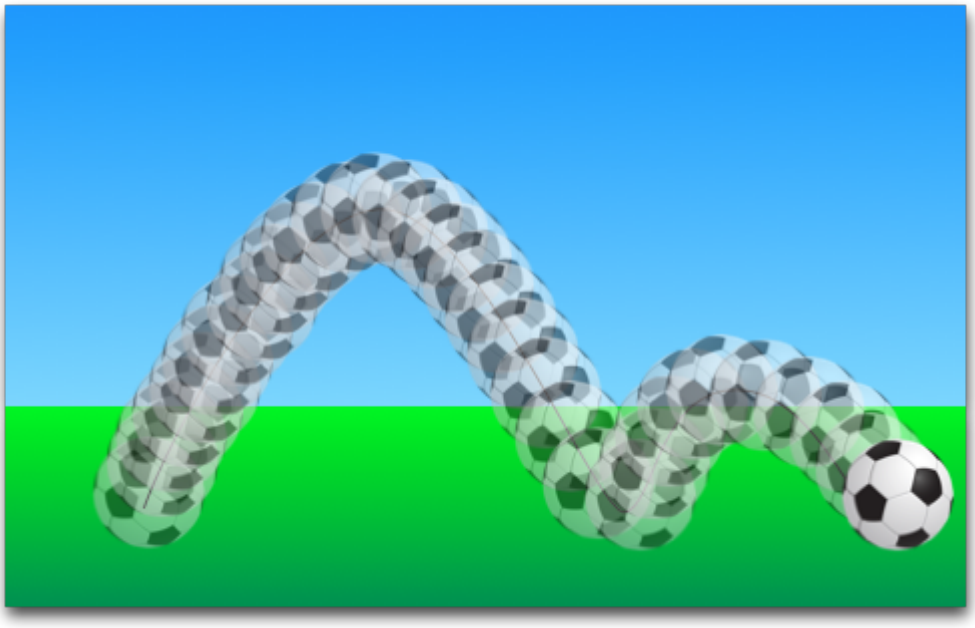
```
// SequentialAnimationExample.qml
import QtQuick

BrightSquare {
    id: root

    property int duration: 3000
    property Item ufo: ufo
```

```
    width: 600
    height: 400

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

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

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



Nested animations

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

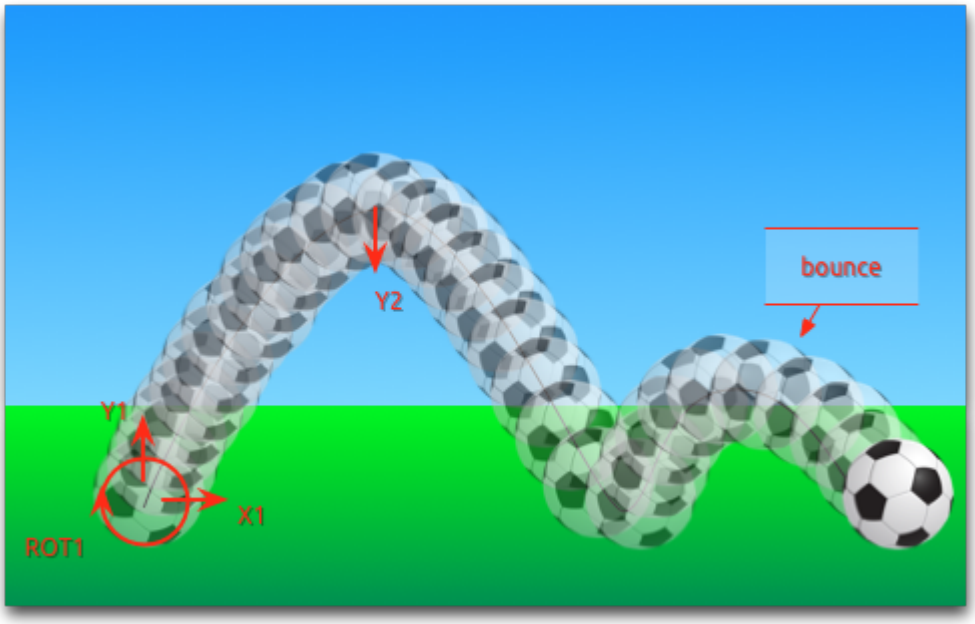


To understand the animation we need to dissect it into the integral transformations of the object. We need to remember that animations animate property changes. Here are the different transformations:

- An x-translation from left-to-right (`X1`)
- A y-translation from bottom to top (`Y1`) followed by a translation from up to down (`Y2`) with some bouncing

- A rotation of 360 degrees over the entire duration of the animation (`ROT1`)

The whole duration of the animation should take three seconds.



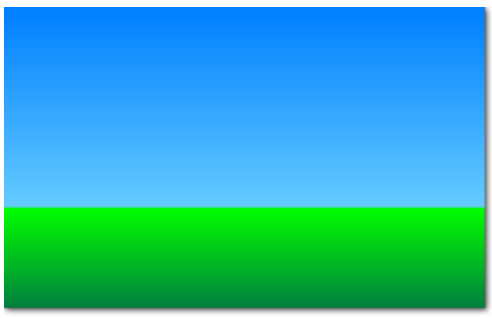
We start with an empty item as the root element of the width of 480 and height of 300.

```
import QtQuick

Item {
    id: root

    property int duration: 3000

    width: 480
    height: 300

    // [...]
}
```

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

The next step is to add the background, which in our case are 2 rectangles with 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 pixels of the height and the lower one is anchored to the bottom of the sky and to the bottom of the root element.

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

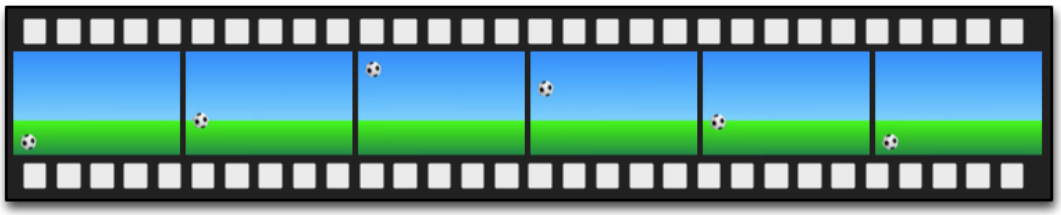
```
Image {
    id: ball
    x: 0; y: root.height-height
    source: "assets/soccer_ball.png"

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

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

Let’s start with a sequential animation for the two y translations first.

```
SequentialAnimation {
    id: anim
    NumberAnimation {
        target: ball
```

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

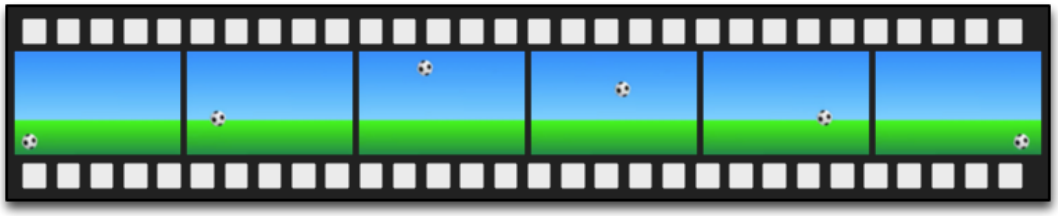


This specifies that 40% of the total animation duration is the up animation and 60% the down animation, with each animation running after the other in sequence. The transformations are animated on a linear path but there is no curving currently. Curves will be added later using the easing curves, at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

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

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

```
            to: 400
            duration: root.duration
        }
    }
```



In the end, we would like the ball to be rotating. For this, we need to add another animation to the parallel animation. We choose `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 that's left is to provide the correct easing curves for the movements of the ball. For the *Y1* animation, we use a `Easing.OutCirc` curve, as this should look more like a circular movement. *Y2* is enhanced using an `Easing.OutBounce` to give the ball its bounce, and the bouncing should happen at the end (try

with `Easing.InBounce` and you will see that the bouncing starts 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: root.height-ball.height
            duration: root.duration * 0.6
            easing.type: Easing.OutBounce
        }
    }
    NumberAnimation {
        target: ball
        properties: "x"
        to: root.width-ball.width
        duration: root.duration
    }
    RotationAnimation {
        target: ball
        properties: "rotation"
        to: 720
        duration: root.duration
    }
}
```

States and Transitions

Often parts of a user interface can be described in states. A state defines a set of property changes and can be triggered by a certain condition.

Additionally, these state switches can have a transition attached which defines how these changes should be animated or any additional actions that 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 in its simplest form, consists of a series of property changes on elements. The default state is defined by the initial properties of the element and is named `""` (an empty string).

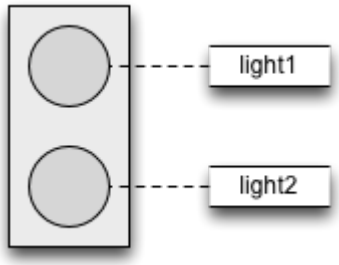
```
Item {
    id: root
    states: [
        State {
            name: "go"
            PropertyChanges { ... }
        },
        State {
            name: "stop"
            PropertyChanges { ... }
        }
```

```
    ]
}
```

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

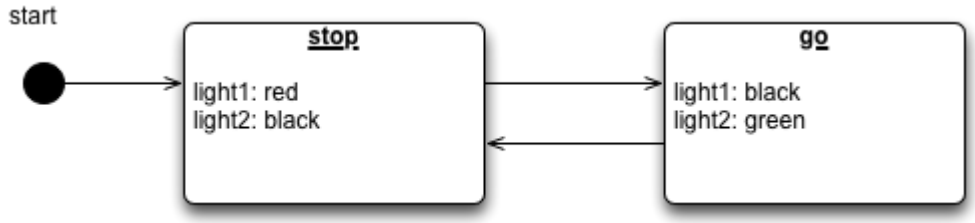
> **Control states using when**
>
> Another way to control states is using the `when` property of the `State` element. The `when` property can be set to an expression that evaluates to true when the state should be applied.

```
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 automatically goes into the stop mode as the default state. The stop state changes `light1` to red and `light2` to black (off).

An external event can now trigger a state switch to the `"go"` state. In the go state, we change the color properties from `light1` to black (off) and `light2` to green to indicate the pedestrians may now cross.

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

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

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

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

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

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

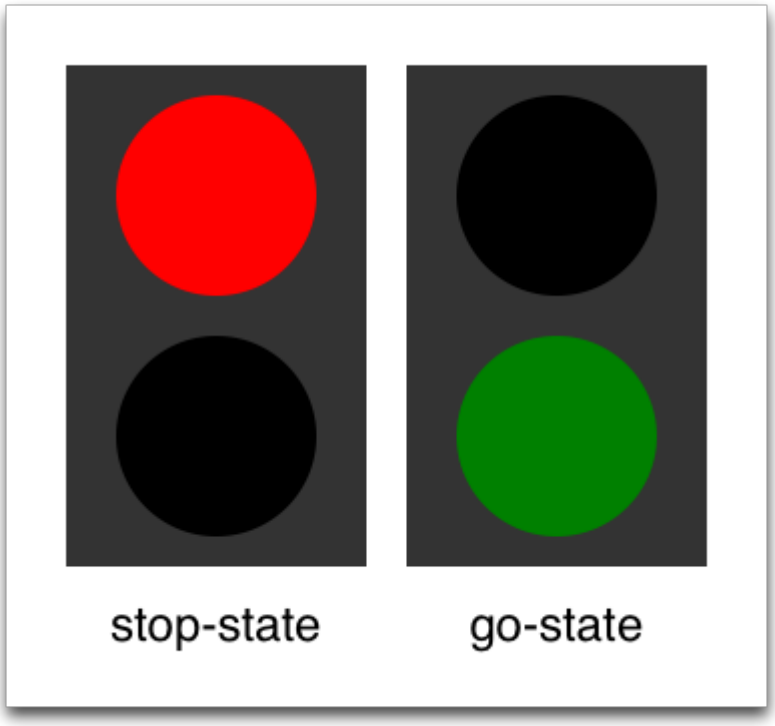
```
    state: "stop"

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

Using `PropertyChanges { target: light2; color: "black" }` is not really required in these 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" :
}
```



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

> **Using scripting**
>
> It's possible to create similar logic using scripting instead of QML states. However, QML is a better language than JavaScript for describing user interfaces. Where possible, aim to write declarative code instead of imperative code.

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 wildcard "*", 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`. This means the transition will be applied to every state switch.

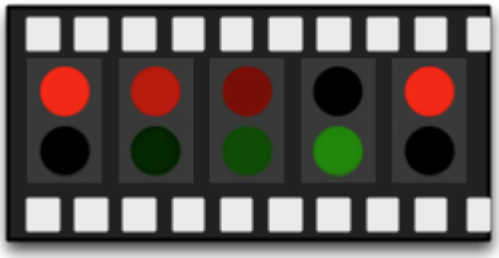
For this example, we would like to animate the color changes when switching state from "go" to "stop". For the other reversed state change ("stop" to "go") we want to keep an immediate color change and don't apply a transition.

We restrict the transition with the `from` and `to` properties to filter only the state change from "go" to "stop". Inside the transition, we add two color animations for each light, which shall animate the property changes defined in the state description.

```
    transitions: [
        Transition {
```

```
            from: "stop"; to: "go"
            // from: "*"; to: "*"
            ColorAnimation { target: light1; properties: "color"; d
            ColorAnimation { target: light2; properties: "color"; d
        }
    ]
```

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 the transition from “stop” to “go”. You will see the change will happen immediately.



You could play around with this UI by, for example, scaling the inactive light down to highlight the active light.

For this, you would need to add another property change for scaling to the states and also handle the animation for the scaling property in the transition.

Another option would be to add an “attention” state where the lights are blinking yellow. For this, you would need to add a sequential animation to the transition for one second going to yellow (“to” property of the animation and one second going to “black”).

Maybe you would also want to change the easing curve to make it more visually appealing.

Advanced Techniques

Nothing advanced here 😃

UI Controls

This chapter shows how to use the Qt Quick Controls module. Qt Quick Controls are used to create advanced user interfaces built from standard components such as buttons, labels, sliders and so on.

Qt Quick Controls can be arranged using the layout module and are easy to style. Also we will look into the various styles for the different platforms before diving into custom styling.

Introduction to Controls

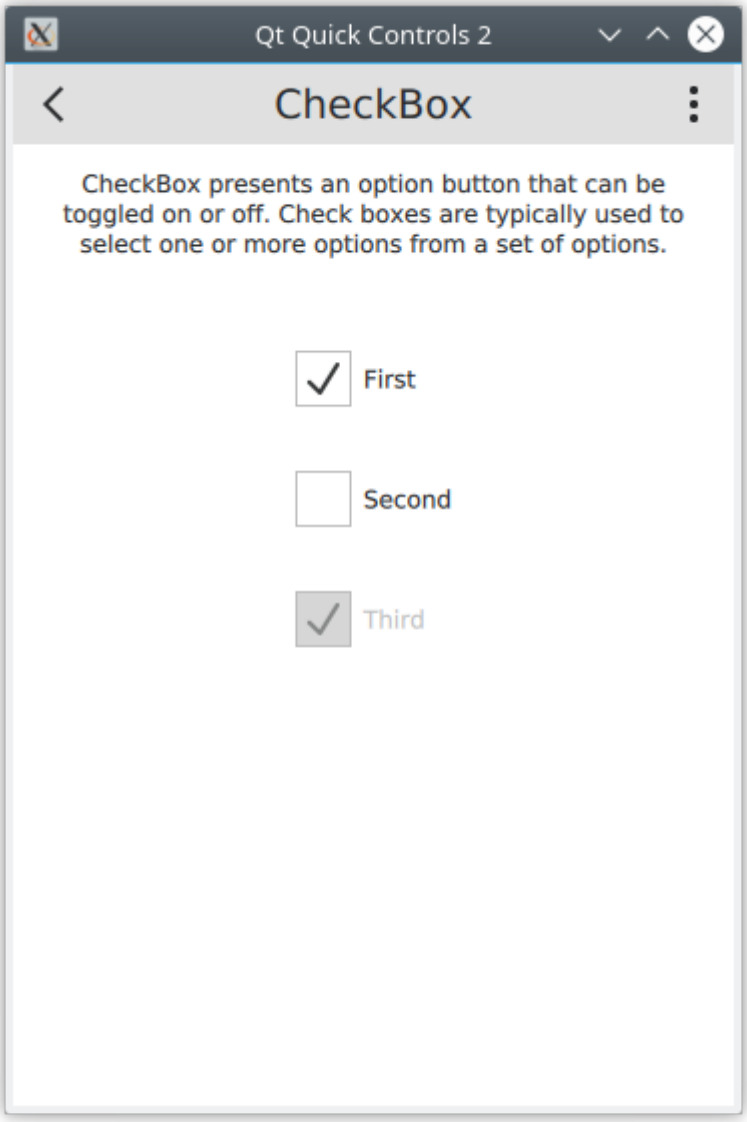
Using Qt Quick from scratch gives you primitive graphical and interaction elements from which you can build your user interfaces. Using Qt Quick Controls you start from a slightly more structured set of controls to build from.

The controls range from simple text labels and buttons to more complex ones such as sliders and dials. These elements are handy if you want to create a user interface based on classic interaction patterns, as they provide a good foundation to stand on.

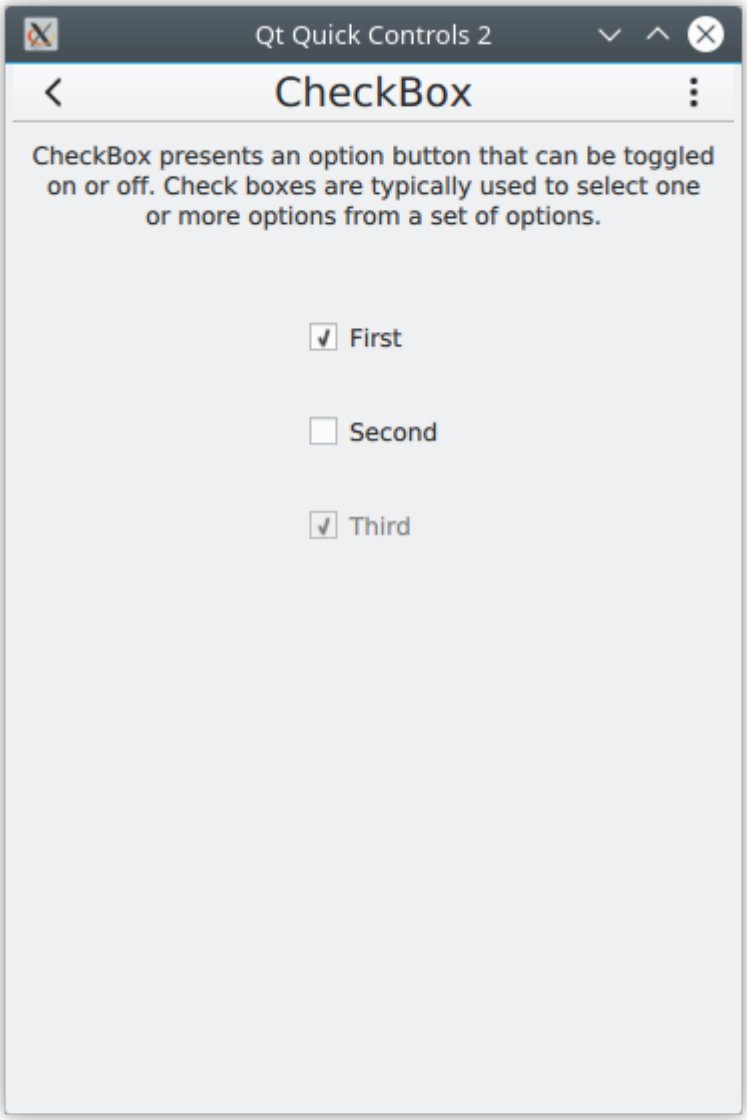
Qt Quick Controls come with a number of styles out of the box that are shown in the table below. The *Basic* style is a basic flat style. The *Universal* style is based on the Microsoft Universal Design Guidelines, while *Material* is based on Google's Material Design Guidelines, and the *Fusion* style is a desktop-oriented style.

Some of the styles can be tweaked by modifying palettes. The *Imagine* style is based on image assets, this allows a graphical designer to create a new style without writing any code at all, not even for palette colour codes.

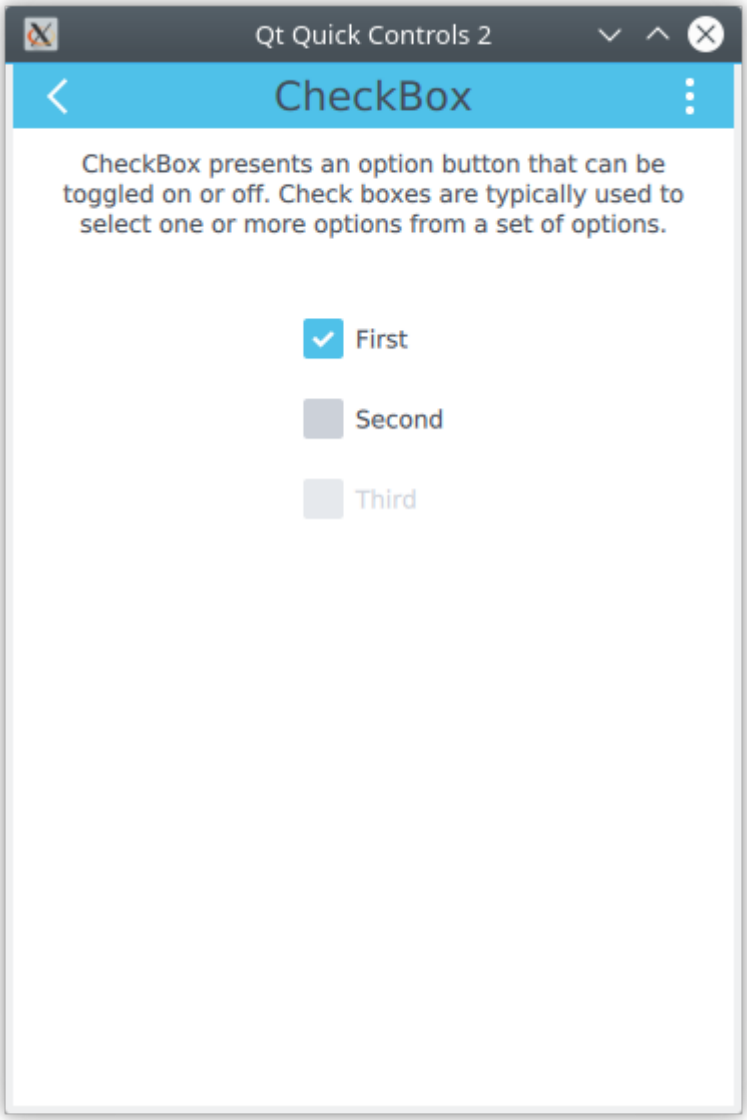
- Basic


Qt Quick Controls 2
CheckBox
CheckBox presents an option button that can be toggled on or off. Check boxes are typically used to select one or more options from a set of options.
First
Second
Third

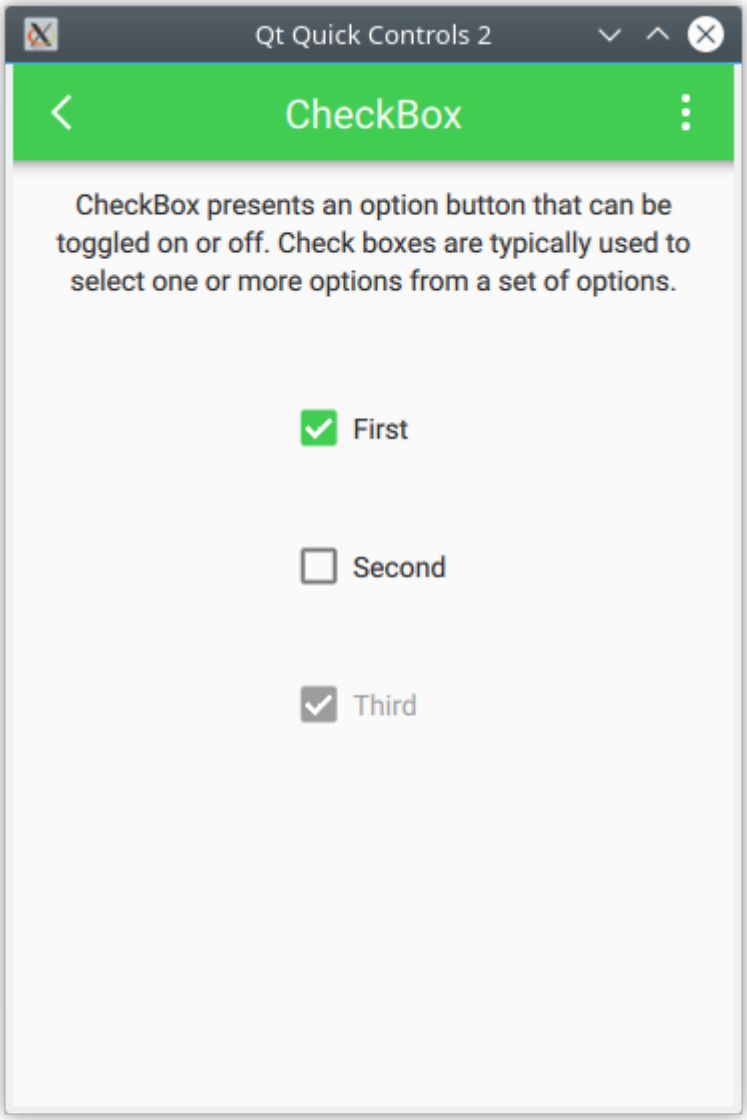
- Fusion


Qt Quick Controls 2
CheckBox
CheckBox presents an option button that can be toggled on or off. Check boxes are typically used to select one or more options from a set of options.
First
Second
Third

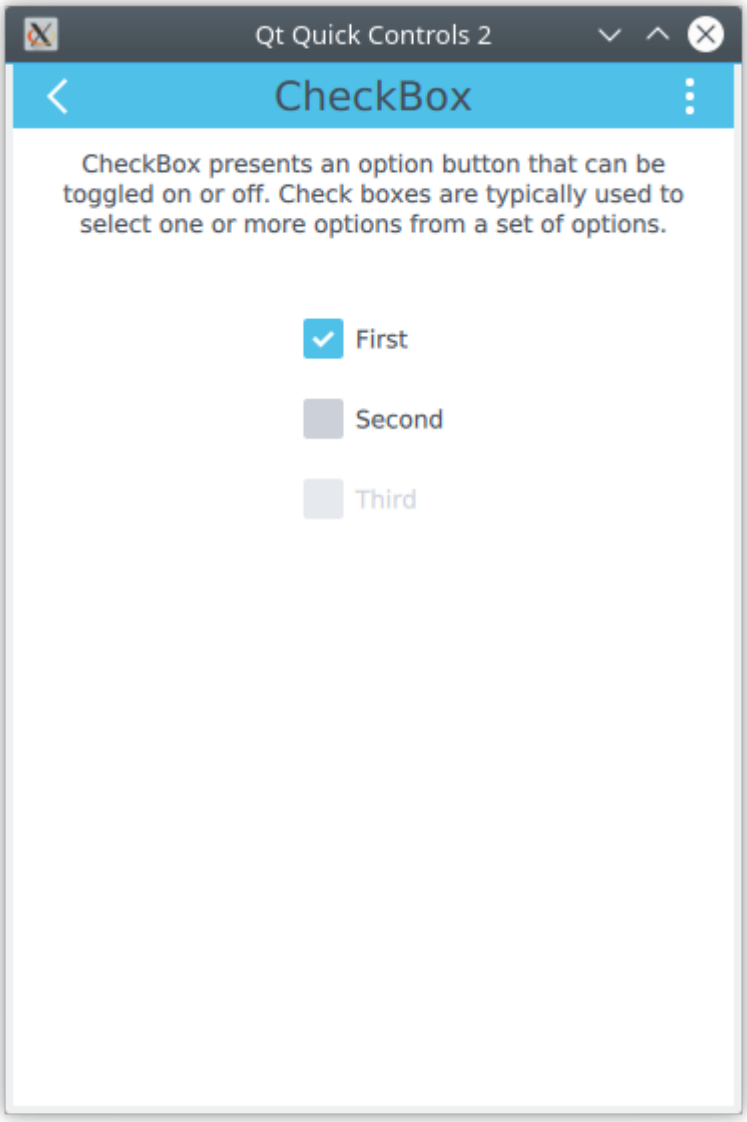
- macOS


Qt Quick Controls 2
CheckBox
CheckBox presents an option button that can be toggled on or off. Check boxes are typically used to select one or more options from a set of options.
First
Second
Third

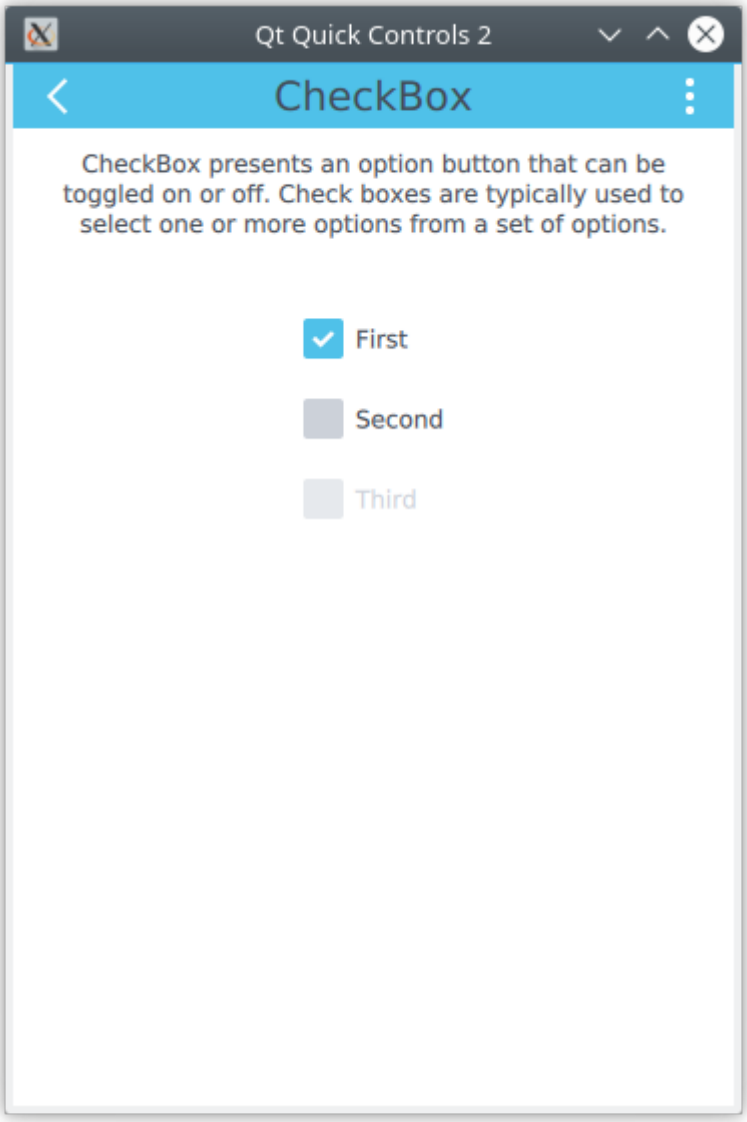
- Material


Qt Quick Controls 2
CheckBox
CheckBox presents an option button that can be toggled on or off. Check boxes are typically used to select one or more options from a set of options.
First
Second
Third

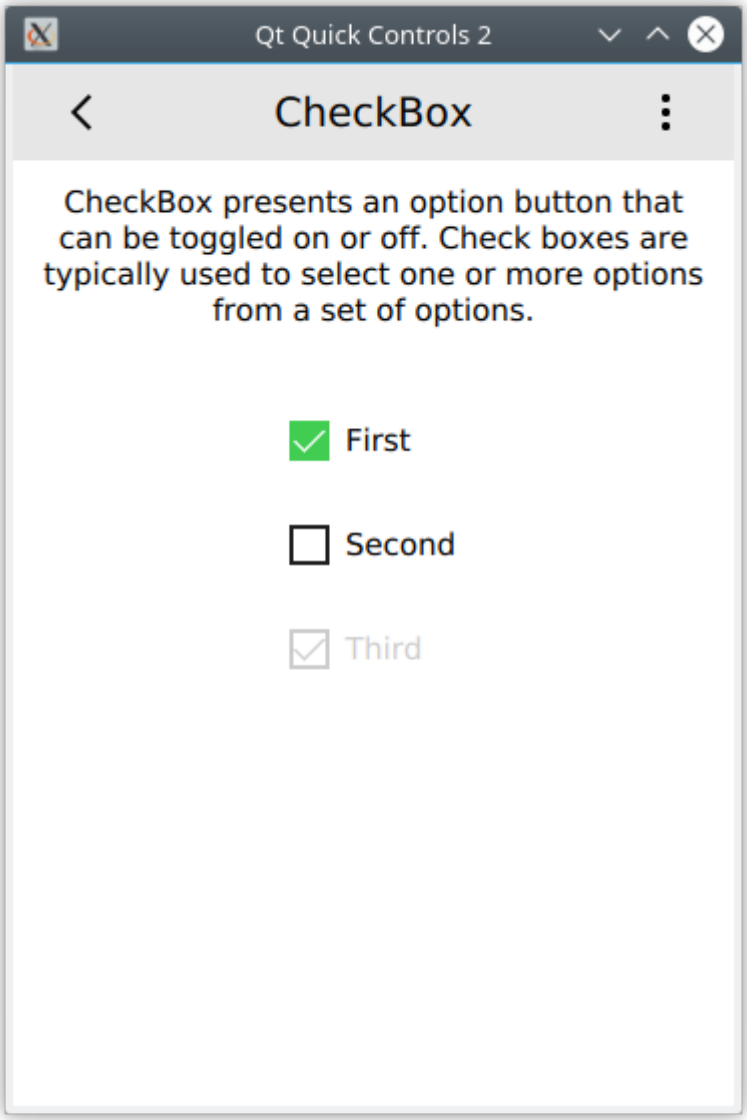
- Imagine


Qt Quick Controls 2
CheckBox
CheckBox presents an option button that can be toggled on or off. Check boxes are typically used to select one or more options from a set of options.
First
Second
Third

- Windows


Qt Quick Controls 2
CheckBox
CheckBox presents an option button that can be toggled on or off. Check boxes are typically used to select one or more options from a set of options.
First
Second
Third

- Universal



Qt Quick Controls 2 is available from the `QtQuick.Controls` import. The following modules are also of interest:

- `QtQuick.Controls` - The basic controls.
- `QtQuick.Templates` - Provides the behavioral, non-visual base types for the controls.
- `QtQuick.Controls.Imagine` - Imagine style theming support.
- `QtQuick.Controls.Material` - Material style theming support.
- `QtQuick.Controls.Universal` - Universal style theming support.

- `Qt.labs.platform` - Support for platform native dialogs for common tasks such as picking files, colours, etc, as well as system tray icons and standard paths.

> **Qt.Labs**
>
> Notice that the `Qt.labs` modules are experimental, meaning that their APIs can have breaking changes between Qt versions.

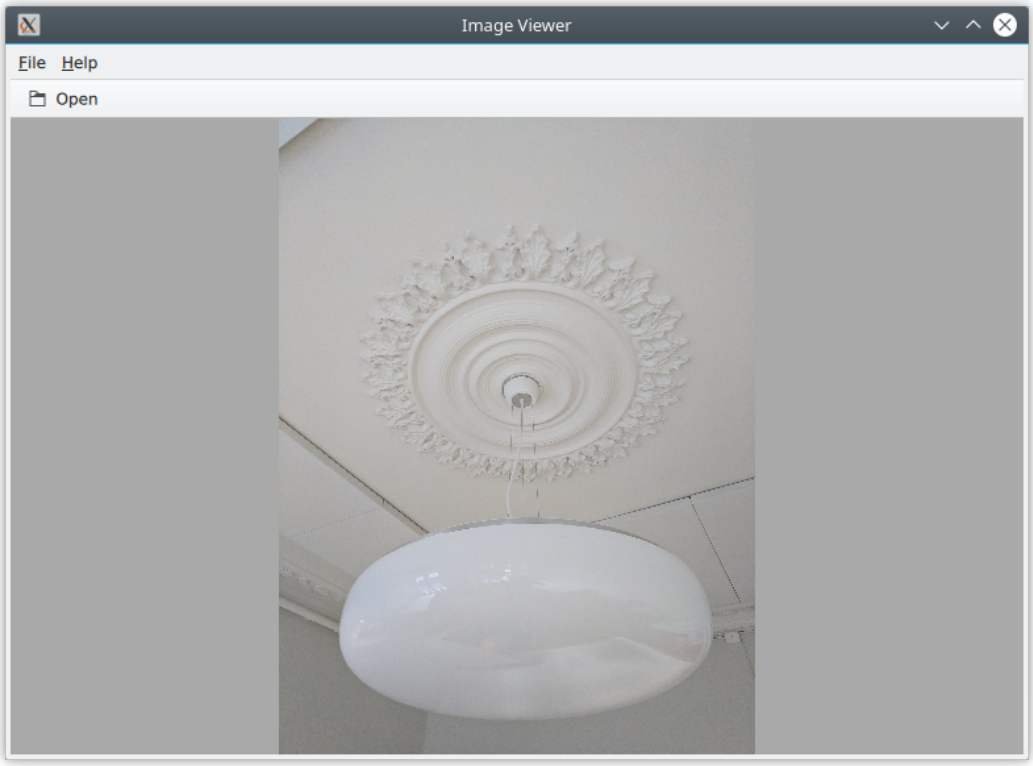
An Image Viewer

Let's look at a larger example of how Qt Quick Controls are used. For this, we will create a simple image viewer.

First, we create it for desktop using the Fusion style, then we will refactor it for a mobile experience before having a look at the final code base.

The Desktop Version

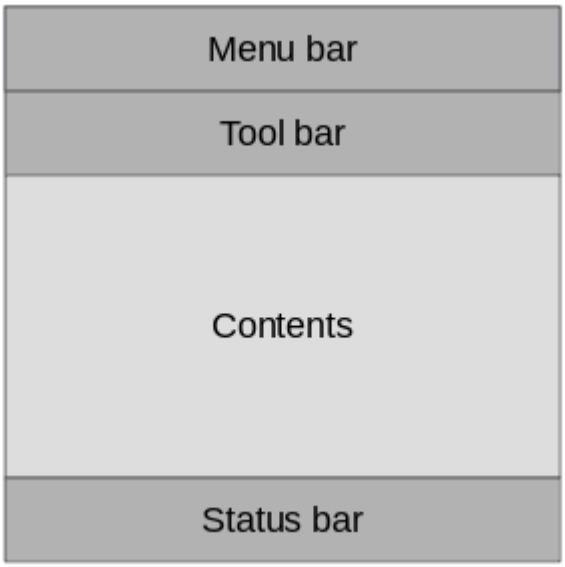
The desktop version is based around a classic application window with a menu bar, a tool bar and a document area. The application can be seen in action below.



We use the Qt Creator project template for an empty Qt Quick application as a starting point. However, we replace the default `Window` element from the template with a `ApplicationWindow` from the `QtQuick.Controls` module. The code below shows `main.qml` where the window itself is created and setup with a default size and title.

```
import QtQuick
import QtQuick.Controls
import Qt.labs.platform as Platform

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    // ...

}
```

The `ApplicationWindow` consists of four main areas as shown below. The menu bar, tool bar and status bar are usually populated by instances of `MenuBar`, `ToolBar` or `TabBar` controls, while the contents area is where the children of the window go. Notice that the image viewer application does not feature a status bar; that is why it is missing from the code show here, as well as from the figure above.



As we are targeting desktop, we enforce the use of the *Fusion* style. This can be done via a configuration file, environment variables, command line arguments, or programmatically in the C++ code. We do it the latter way by adding the following line to `main.cpp`:

```
QQuickStyle::setStyle("Fusion");
```

We then start building the user interface in `main.qml` by adding an `Image` element as the contents. This element will hold the images when the user opens them, so for now it is just a placeholder. The `background` property is used to provide an element to the window to place behind the contents. This will be shown when there is no image loaded, and as borders around the image if the aspect ratio does not let it fill the contents area of the window.

```
ApplicationWindow {

    // ...

    background: Rectangle {
        color: "darkGray"
    }
```

```
        Image {
            id: image
            anchors.fill: parent
            fillMode: Image.PreserveAspectFit
            asynchronous: true
        }

        // ...

    }
```

We then continue by adding the `ToolBar`. This is done using the `toolBar` property of the window. Inside the tool bar we add a `Flow` element which will let the contents fill the width of the control before overflowing to a new row. Inside the flow we place a `ToolButton`.

The `ToolButton` has a couple of interesting properties. The `text` is straight forward. However, the `icon.name` is taken from the **freedesktop.org Icon Naming Specification (https://specifications.freedesktop.org/icon-naming-spec/icon-naming-spec-latest.html)**. In that document, a list of standard icons are listed by name. By referring to such a name, Qt will pick out the correct icon from the current desktop theme.

In the `onClicked` signal handler of the `ToolButton` is the final piece of code. It calls the `open` method on the `fileOpenDialog` element.

```
    ApplicationWindow {

        // ...

        header: ToolBar {
            Flow {
                anchors.fill: parent
                ToolButton {
```

```
                text: qsTr("Open")
                icon.name: "document-open"
                onClicked: fileOpenDialog.open()
            }
        }
    }

    // ...

}
```

The `fileOpenDialog` element is a `FileDialog` control from the `Qt.labs.platform` module. The file dialog can be used to open or save files. We import the `Qt.labs.platform` as `Platform`, to avoid a naming collision with the `QtQuick.Controls` import, hence we refer to it as `Platform.FileDialog`.

In the code we start by assigning a `title`. Then we set the starting folder using the `StandardsPaths` class. The `StandardsPaths` class holds links to common folders such as the user's home, documents, and so on. After that we set a name filter that controls which files the user can see and pick using the dialog.

Finally, we reach the `onAccepted` signal handler where the `Image` element that holds the window contents is set to show the selected file. There is an `onRejected` signal as well, but we do not need to handle it in the image viewer application.

```
ApplicationWindow {

    // ...

    FileDialog {
        id: fileOpenDialog
        title: "Select an image file"
        folder: StandardPaths.writableLocation(StandardPaths.Do
```

```
            nameFilters: [
                "Image files (*.png *.jpeg *.jpg)",
            ]
            onAccepted: {
                image.source = fileOpenDialog.fileUrl
            }
        }

        // ...

    }
```

We then continue with the `MenuBar`. To create a menu, one puts `Menu` elements inside the menu bar, and then populates each `Menu` with `MenuItem` elements.

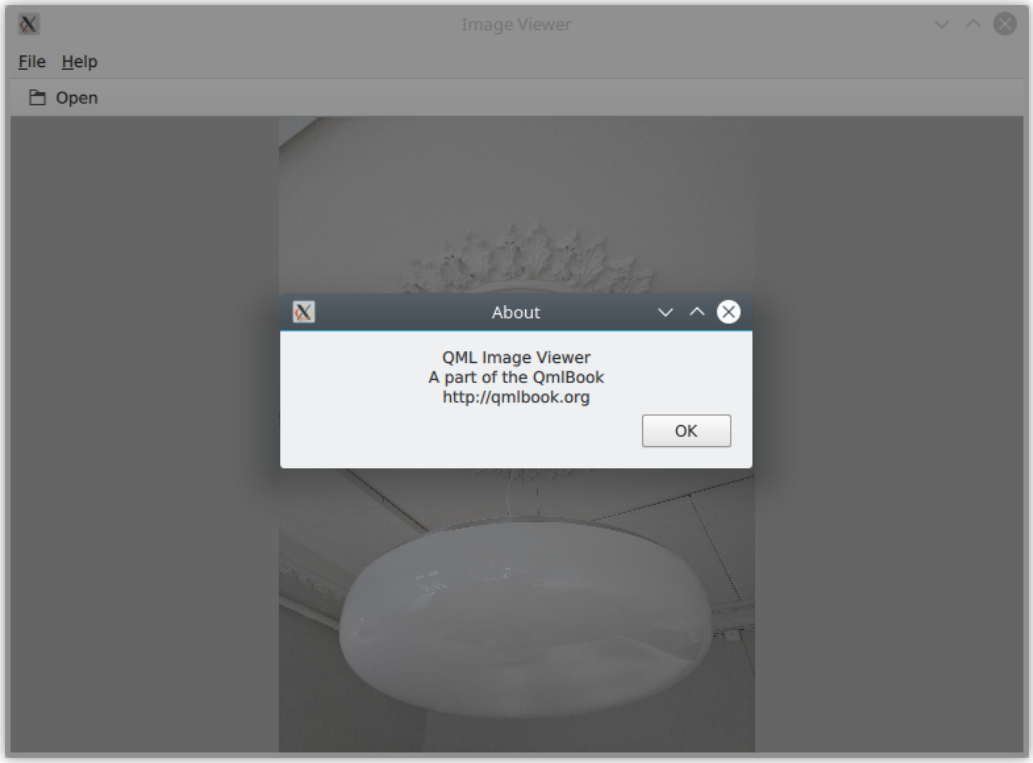
In the code below, we create two menus, *File* and *Help*. Under *File*, we place *Open* using the same icon and action as the tool button in the tool bar. Under *Help*, you find *About* which triggers a call to the `open` method of the `aboutDialog` element.

Notice that the ampersands ("&") in the `title` property of the `Menu` and the `text` property of the `MenuItem` turn the following character into a keyboard shortcut; e.g. you reach the file menu by pressing *Alt+F*, followed by *Alt+O* to trigger the open item.

```
    ApplicationWindow {

        // ...

        menuBar: MenuBar {
            Menu {
                title: qsTr("&File")
                MenuItem {
                    text: qsTr("&Open...")
                    icon.name: "document-open"
```

```
                onTriggered: fileOpenDialog.open()
            }
        }

        Menu {
            title: qsTr("&Help")
            MenuItem {
                text: qsTr("&About...")
                onTriggered: aboutDialog.open()
            }
        }
    }

    // ...

}
```

The `aboutDialog` element is based on the `Dialog` control from the `QtQuick.Controls` module, which is the base for custom dialogs. The dialog we are about to create is shown in the figure below.



The code for the `aboutDialog` can be split into three parts. First, we setup the dialog window with a title. Then, we provide some contents for the dialog – in this case, a `Label` control. Finally, we opt to use a standard *Ok* button to close the dialog.

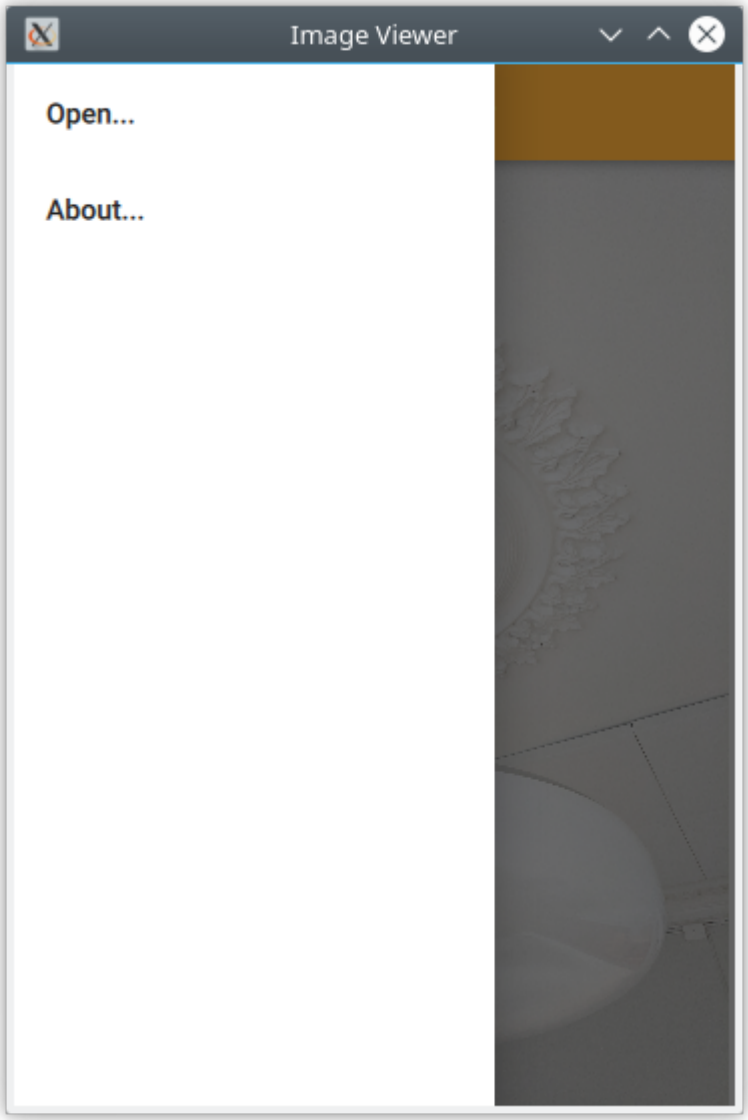
```
ApplicationWindow {

    // ...

    Dialog {
        id: aboutDialog
        title: qsTr("About")
        Label {
            anchors.fill: parent
            text: qsTr("QML Image Viewer\nA part of the QmlBook
            horizontalAlignment: Text.AlignHCenter
        }

        standardButtons: StandardButton.Ok
```

```
        }

        // ...

    }
```

The end result of all this is a functional, albeit simple, desktop application for viewing images.

Moving to Mobile

There are a number of differences in how a user interface is expected to look and behave on a mobile device compared to a desktop application. The biggest difference for our application is how the actions are accessed. Instead of a menu bar and a tool bar, we will use a drawer from which the user can pick the actions. The drawer can be swiped in from the side, but we also offer a hamburger button in the header. The resulting application with the drawer open can be seen below.



First of all, we need to change the style that is set in `main.cpp` from *Fusion* to *Material*:

```
QQuickStyle::setStyle("Material");
```

Then we start adapting the user interface. We start by replacing the menu with a drawer. In the code below, the `Drawer` component is added as a child to the `ApplicationWindow`. Inside the drawer, we put a `ListView` containing `ItemDelegate` instances. It also contains a `ScrollIndicator`

used to show which part of a long list is being shown. As our list only consists of two items, the indicator is not visible in this example.

The drawer's `ListView` is populated from a `ListModel` where each `ListItem` corresponds to a menu item. Each time an item is clicked, in the `onClicked` method, the `triggered` method of the corresponding `ListItem` is called. This way, we can use a single delegate to trigger different actions.

```
ApplicationWindow {

    // ...

    id: window

    Drawer {
        id: drawer

        width: Math.min(window.width, window.height) / 3 * 2
        height: window.height

        ListView {
            focus: true
            currentIndex: -1
            anchors.fill: parent

            delegate: ItemDelegate {
                width: parent.width
                text: model.text
                highlighted: ListView.isCurrentItem
                onClicked: {
                    drawer.close()
                    model.triggered()
                }
            }

            model: ListModel {
                ListElement {
```

```
                    text: qsTr("Open...")
                    triggered: function() { fileOpenDialog.open
                }
                ListElement {
                    text: qsTr("About...")
                    triggered: function() { aboutDialog.open();
                }
            }

            ScrollIndicator.vertical: ScrollIndicator { }
        }
    }

    // ...

}
```

The next change is in the `header` of the `ApplicationWindow`. Instead of a desktop style toolbar, we add a button to open the drawer and a label for the title of our application.

The `ToolBar` contains two child elements: a `ToolButton` and a `Label`.

The `ToolButton` control opens the drawer. The corresponding `close` call can be found in the `ListView` delegate. When an item has been selected, the drawer is closed. The icon used for the `ToolButton` comes from the **Material Design Icons page (https://material.io/tools/icons/?style=baseline)**.

```
ApplicationWindow {
```

```
    // ...

    header: ToolBar {
        ToolButton {
            id: menuButton
            anchors.left: parent.left
            anchors.verticalCenter: parent.verticalCenter
            icon.source: "images/baseline-menu-24px.svg"
            onClicked: drawer.open()
        }
        Label {
            anchors.centerIn: parent
            text: "Image Viewer"
            font.pixelSize: 20
            elide: Label.ElideRight
        }
    }

    // ...

}
```

Finally we make the background of the toolbar pretty — or at least orange. To do this, we alter the `Material.background` attached property. This comes from the `QtQuick.Controls.Material` module and only affects the Material style.

```
import QtQuick.Controls.Material

ApplicationWindow {

    // ...

    header: ToolBar {
        Material.background: Material.Orange

    // ...
```

```
}
```

With these few changes we have converted our desktop image viewer to a mobile-friendly version.

A Shared Codebase

In the past two sections we have looked at an image viewer developed for desktop use and then adapted it to mobile.

Looking at the code base, much of the code is still shared. The parts that are shared are mostly associated with the document of the application, i.e. the image. The changes have accounted for the different interaction patterns of desktop and mobile, respectively. Naturally, we would want to unify these code bases. QML supports this through the use of *file selectors*.

A file selector lets us replace individual files based on which selectors are active. The Qt documentation maintains a list of selectors in the documentation for the `QFileSelector` class (**link (https://doc.qt.io/qt-5/qfileselector.html)**). In our case, we will make the desktop version the default and replace selected files when the *android* selector is encountered. During development you can set the environment variable `QT_FILE_SELECTORS` to `android` to simulate this.

> **File Selector**
>
> File selectors work by replacing files with an alternative when a *selector* is present.
>
> By creating a directory named `+selector` (where `selector` represents the name of a selector) in the same directory as the files that you want to replace, you can then place files with the same name as the file you want to replace inside the directory. When the selector is present, the file in the directory will be picked instead of the original file.
>
> The selectors are based on the platform: e.g. android, ios, osx, linux, qnx, and so on. They can also include the name of the Linux distribution used (if identified), e.g. debian, ubuntu, fedora. Finally, they also include the locale, e.g. en_US, sv_SE, etc.
>
> It is also possible to add your own custom selectors.

The first step to do this change is to isolate the shared code. We do this by creating the `ImageViewerWindow` element which will be used instead of the `ApplicationWindow` for both of our variants. This will consist of the dialogs, the `Image` element and the background. In order to make the open methods of the dialogs available to the platform specific code, we need to expose them through the functions `openFileDialog` and `openAboutDialog` .

```
import QtQuick
import QtQuick.Controls
import Qt.labs.platform as Platform

ApplicationWindow {
    function openFileDialog() { fileOpenDialog.open(); }
    function openAboutDialog() { aboutDialog.open(); }
```

```
        visible: true
        title: qsTr("Image Viewer")

        background: Rectangle {
            color: "darkGray"
        }

        Image {
            id: image
            anchors.fill: parent
            fillMode: Image.PreserveAspectFit
            asynchronous: true
        }

        Platform.FileDialog {
            id: fileOpenDialog

            // ...

        }

        Dialog {
            id: aboutDialog

            // ...

        }
    }
```

Next, we create a new `main.qml` for our default style *Fusion*, i.e. the desktop version of the user interface.

Here, we base the user interface around the `ImageViewerWindow` instead of the `ApplicationWindow`. Then we add the platform specific parts to it, e.g. the `MenuBar` and `ToolBar`. The only changes to these is that the calls to open the respective dialogs are made to the new functions instead of directly to the dialog controls.

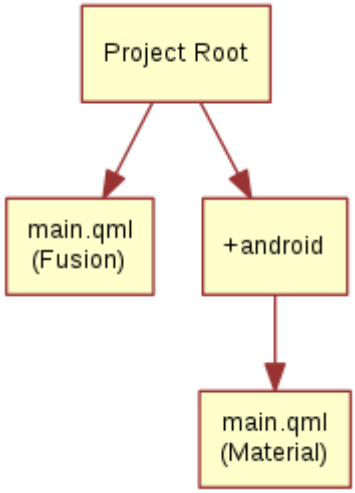
```
import QtQuick
import QtQuick.Controls

ImageViewerWindow {
    id: window

    width: 640
    height: 480

    menuBar: MenuBar {
        Menu {
            title: qsTr("&File")
            MenuItem {
                text: qsTr("&Open...")
                icon.name: "document-open"
                onTriggered: window.openFileDialog()
            }
        }

        Menu {
            title: qsTr("&Help")
            MenuItem {
                text: qsTr("&About...")
                onTriggered: window.openAboutDialog()
            }
        }
    }

    header: ToolBar {
        Flow {
            anchors.fill: parent
            ToolButton {
                text: qsTr("Open")
                icon.name: "document-open"
                onClicked: window.openFileDialog()
            }
        }
    }
}
```

Next, we have to create a mobile specific `main.qml`. This will be based around the *Material* theme. Here, we keep the `Drawer` and the mobile-specific toolbar. Again, the only change is how the dialogs are opened.

```
import QtQuick
import QtQuick.Controls
import QtQuick.Controls.Material

ImageViewerWindow {
    id: window

    width: 360
    height: 520

    Drawer {
        id: drawer

        // ...

        ListView {

            // ...

            model: ListModel {
                ListElement {
                    text: qsTr("Open...")
                    triggered: function(){ window.openFileDialo
                }
                ListElement {
                    text: qsTr("About...")
                    triggered: function(){ window.openAboutDial
                }
            }

            // ...

        }
```

```
        }

        header: ToolBar {

            // ...

        }
    }
```

The two `main.qml` files are placed in the file system as shown below. This lets the file selector that the QML engine automatically creates pick the right file. By default, the *Fusion* `main.qml` is loaded. If the `android` selector is present, then the *Material* `main.qml` is loaded instead.



Until now the style has been set in `main.cpp`. We could continue doing this and use `#ifdef` expressions to set different styles for different platforms. Instead we will use the file selector mechanism again and set the style using a configuration file. Below, you can see the file for the *Material* style, but the *Fusion* file is equally simple.

```
[Controls]
Style=Material
```

These changes have given us a joined codebase where all the document code is shared and only the differences in user interaction patterns differ. There are different ways to do this, e.g. keeping the document in a specific component that is included in the platform specific interfaces, or as in this example, by creating a common base that is extended by each platform. The best approach is best determined when you know how your specific code base looks and can decide how to separate the common from the unique.

Native Dialogs

When using the image viewer you will notice that it uses a non-standard file selector dialog. This makes it look out of place.

The `Qt.labs.platform` module can help us solve this. It provides QML bindings to native dialogs such as the file dialog, font dialog and colour dialog. It also provides APIs to create system tray icons, as well as system global menus that sits on top of the screen (e.g. as in OS X). The cost of this is a dependency on the `QtWidgets` module, as the widget based dialog is used as a fallback where the native support is missing.

In order to integrate a native file dialog into the image viewer, we need to import the `Qt.labs.platform` module. As this module has name clashes with the `QtQuick.Dialogs` module which it replaces, it is important to remove the old import statement.

In the actual file dialog element, we have to change how the `folder` property is set, and ensure that the `onAccepted` handler uses the `file` property instead of the `fileUrl` property. Apart from these details, the usage is identical to the `FileDialog` from `QtQuick.Dialogs`.

```
import QtQuick
import QtQuick.Controls
```

```
import Qt.labs.platform as Platform

ApplicationWindow {

    // ...

    Platform.FileDialog {
        id: fileOpenDialog
        title: "Select an image file"
        folder: StandardPaths.writableLocation(StandardPaths.Do
        nameFilters: [
            "Image files (*.png *.jpeg *.jpg)",
        ]
        onAccepted: {
            image.source = fileOpenDialog.file
        }
    }

    // ...

}
```

In addition to the QML changes, we also need to alter the project file of the image viewer to include the `widgets` module.

```
QT += quick quickcontrols2 widgets
```

And we need to update `main.qml` to instantiate a `QApplication` object instead of a `QGuiApplication` object. This is because the `QGuiApplication` class contains the minimal environment needed for a graphical application, while `QApplication` extends `QGuiApplication` with features needed to support `QtWidgets`.

```
include <QApplication>

// ...
```

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

    // ...

}
```

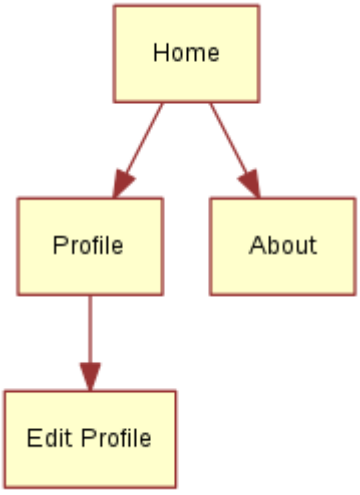
With these changes, the image viewer will now use native dialogs on most platforms. The platforms supported are iOS, Linux (with a GTK+ platform theme), macOS, Windows and WinRT. For Android, it will use a default Qt dialog provided by the `QtWidgets` module.

Common Patterns

There a number of common user interface patterns that can be implemented using Qt Quick Controls. In this section, we try to demonstrate how some of the more common ones can be built.

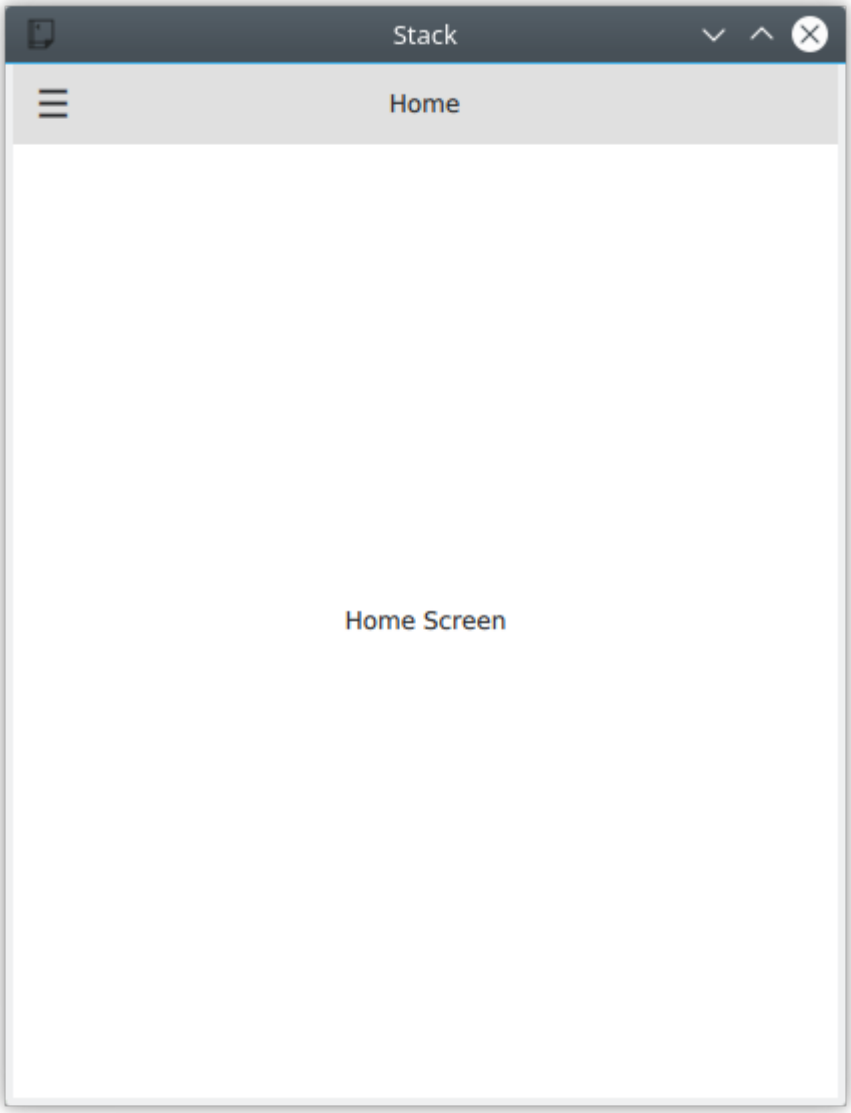
Nested Screens

For this example we will create a tree of pages that can be reached from the previous level of screens. The structure is pictured below.



The key component in this type of user interface is the `StackView`. It allows us to place pages on a stack which then can be popped when the user wants to go back. In the example here, we will show how this can be implemented.

The initial home screen of the application is shown in the figure below.



The application starts in `main.qml`, where we have an `ApplicationWindow` containing a `ToolBar`, a `Drawer`, a `StackView` and a home page element, `Home`. We will look into each of the components below.

```
import QtQuick
import QtQuick.Controls

ApplicationWindow {

    // ...
```

```
        header: ToolBar {

            // ...

        }

        Drawer {

            // ...

        }

        StackView {
            id: stackView
            anchors.fill: parent
            initialItem: Home {}
        }
    }
```

The home page, `Home.qml` consists of a `Page` , which is n control element that support headers and footers. In this example we simply center a `Label` with the text *Home Screen* on the page. This works because the contents of a `StackView` automatically fill the stack view, so the page will have the right size for this to work.

```
    import QtQuick
    import QtQuick.Controls

    Page {
        title: qsTr("Home")

        Label {
            anchors.centerIn: parent
            text: qsTr("Home Screen")
        }
    }
```

Returning to `main.qml`, we now look at the drawer part. This is where the navigation to the pages begin. The active parts of the user interface are the `ÌtemDelegate` items. In the `onClicked` handler, the next page is pushed onto the `stackView`.

As shown in the code below, it is possible to push either a `Component` or a reference to a specific QML file. Either way results in a new instance being created and pushed onto the stack.

```
ApplicationWindow {

    // ...

    Drawer {
        id: drawer
        width: window.width * 0.66
        height: window.height

        Column {
            anchors.fill: parent

            ItemDelegate {
                text: qsTr("Profile")
                width: parent.width
                onClicked: {
                    stackView.push("Profile.qml")
                    drawer.close()
                }
            }
            ItemDelegate {
                text: qsTr("About")
                width: parent.width
                onClicked: {
                    stackView.push(aboutPage)
                    drawer.close()
                }
            }
```

```
        }
    }

    // ...

    Component {
        id: aboutPage

        About {}
    }

    // ...

}
```

The other half of the puzzle is the toolbar. The idea is that a back button is shown when the `stackView` contains more than one page, otherwise a menu button is shown. The logic for this can be seen in the `text` property where the `"\\u..."` strings represents the unicode symbols that we need.

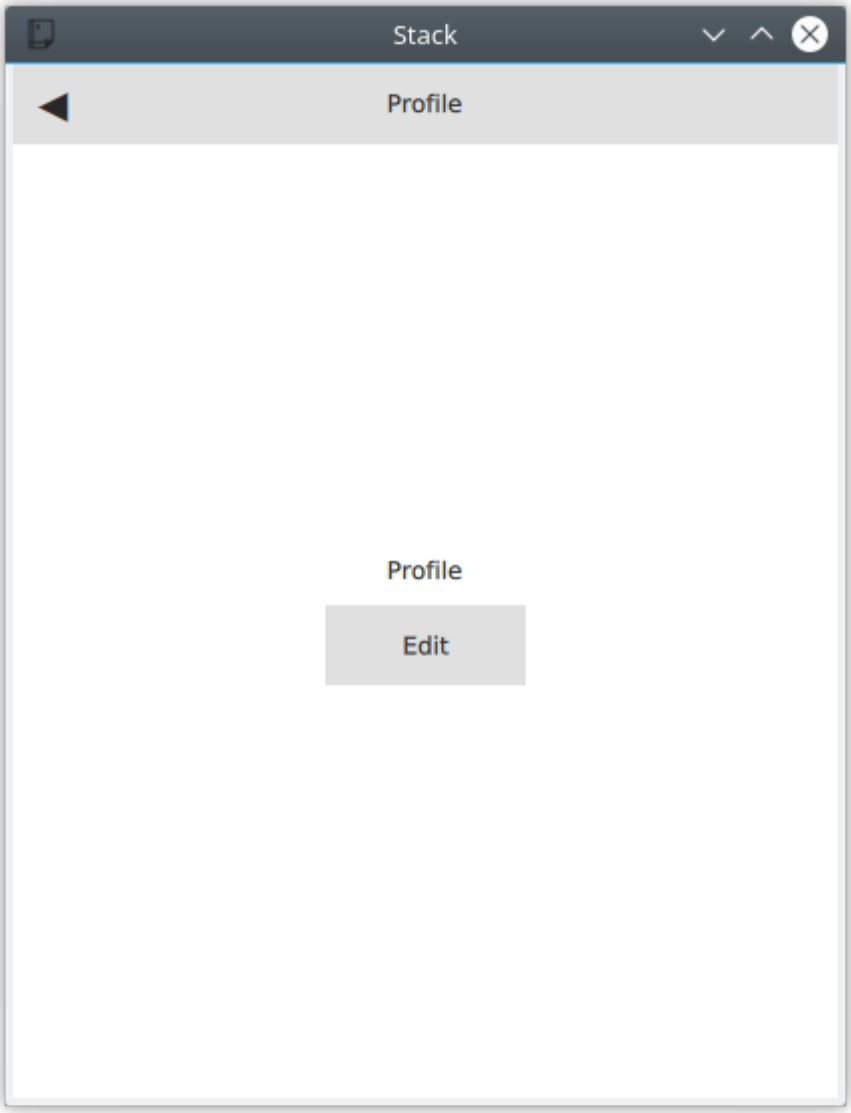
In the `onClicked` handler, we can see that when there is more than one page on the stack, the stack is popped, i.e. the top page is removed. If the stack contains only one item, i.e. the home screen, the drawer is opened.

Below the `ToolBar`, there is a `Label`. This element shows the title of each page in the center of the header.

```
ApplicationWindow {

    // ...

    header: ToolBar {
        contentHeight: toolButton.implicitHeight

        ToolButton {
            id: toolButton
```

```
                text: stackView.depth > 1 ? "\u25C0" : "\u2630"
                font.pixelSize: Qt.application.font.pixelSize * 1.6
                onClicked: {
                    if (stackView.depth > 1) {
                        stackView.pop()
                    } else {
                        drawer.open()
                    }
                }
            }

            Label {
                text: stackView.currentItem.title
                anchors.centerIn: parent
            }
        }

        // ...

    }
```

Now we've seen how to reach the *About* and *Profile* pages, but we also want to make it possible to reach the *Edit Profile* page from the *Profile* page. This is done via the `Button` on the *Profile* page. When the button is clicked, the `EditProfile.qml` file is pushed onto the `StackView`.

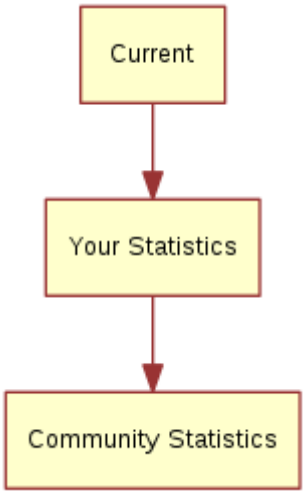



```
import QtQuick
import QtQuick.Controls

Page {
    title: qsTr("Profile")

    Column {
        anchors.centerIn: parent
        spacing: 10
        Label {
            anchors.horizontalCenter: parent.horizontalCenter
```

```
            text: qsTr("Profile")
        }
        Button {
            anchors.horizontalCenter: parent.horizontalCenter
            text: qsTr("Edit");
            onClicked: stackView.push("EditProfile.qml")
        }
    }
}
```

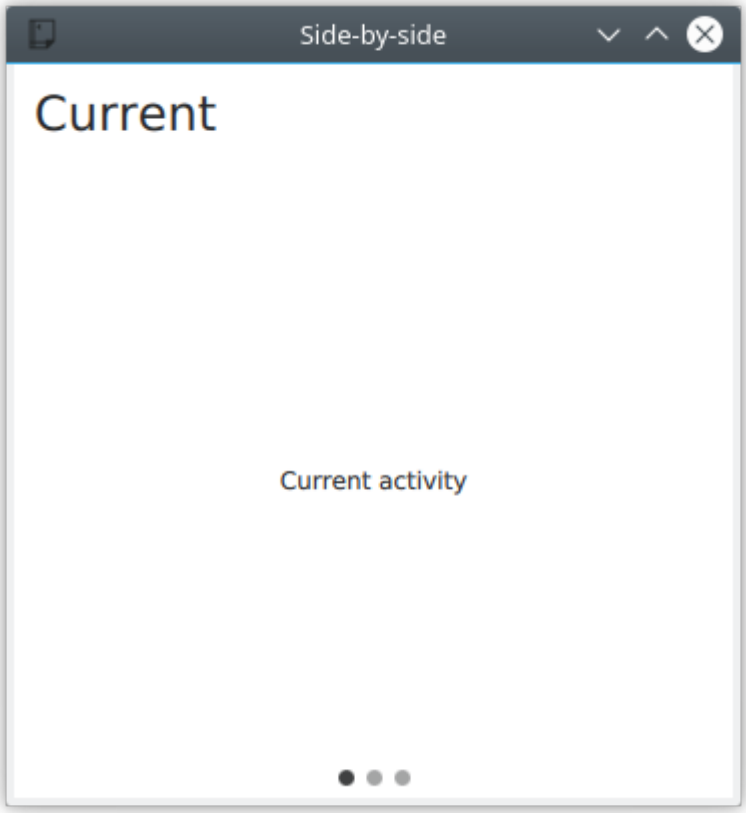
Side by Side Screens

For this example we create a user interface consisting of three pages that the user can shift through. The pages are shown in the diagram below. This could be the interface of a health tracking app, tracking the current state, the user's statistics and the overall statistics.



The illustration below shows how the *Current* page looks in the application. The main part of the screen is managed by a `SwipeView`, which is what enables the side by side screen interaction pattern. The title and text shown in the figure come from the page inside the `SwipeView`, while the `PageIndicator` (the three dots at the bottom) comes from

`main.qml` and sits under the `SwipeView`. The page indicator shows the user which page is currently active, which helps when navigating.



Diving into `main.qml`, it consists of an `ApplicationWindow` with the `SwipeView`.

```
import QtQuick
import QtQuick.Controls

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    title: qsTr("Side-by-side")

    SwipeView {
```

```
            // ...

        }

        // ...

    }
```

Inside the `SwipeView` each of the child pages are instantiated in the order they are to appear. They are `Current` , `UserStats` and `TotalStats` .

```
    ApplicationWindow {

        // ...

        SwipeView {
            id: swipeView
            anchors.fill: parent

            Current {
            }

            UserStats {
            }

            TotalStats {
            }
        }

        // ...

    }
```

Finally, the `count` and `currentIndex` properties of the `SwipeView` are bound to the `PageIndicator` element. This completes the structure

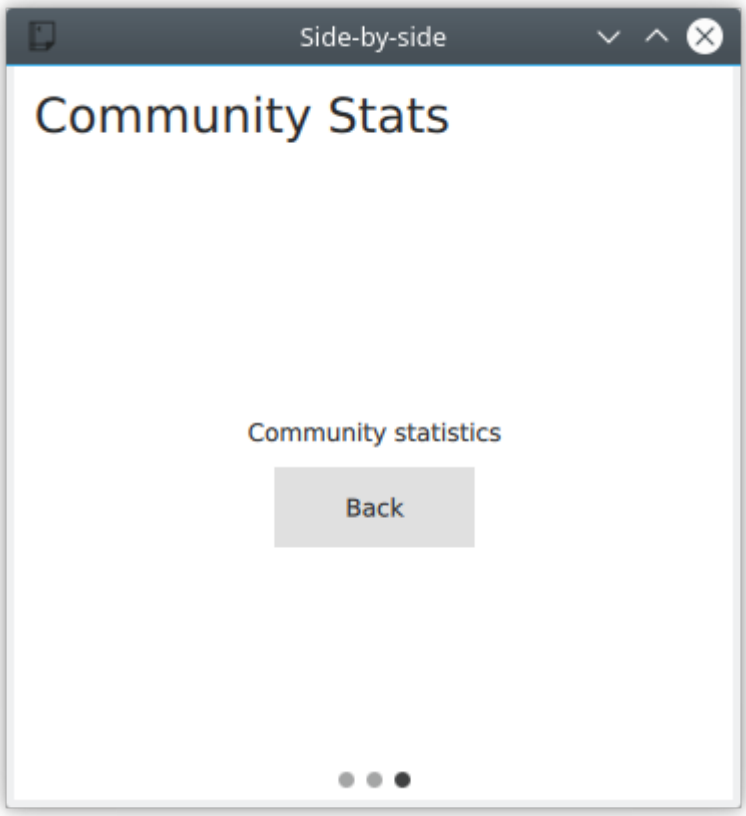
around the pages.

```
ApplicationWindow {

    // ...

    SwipeView {
        id: swipeView

        // ...

    }

    PageIndicator {
        anchors.bottom: parent.bottom
        anchors.horizontalCenter: parent.horizontalCenter

        currentIndex: swipeView.currentIndex
        count: swipeView.count
    }
}
```

Each page consists of a `Page` with a `header` consisting of a `Label` and some contents. For the *Current* and *User Stats* pages the contents consist of a simple `Label`, but for the *Community Stats* page, a back button is included.

```
import QtQuick
import QtQuick.Controls

Page {
    header: Label {
        text: qsTr("Community Stats")
        font.pixelSize: Qt.application.font.pixelSize * 2
        padding: 10
    }
```

```
        // ...

    }
```



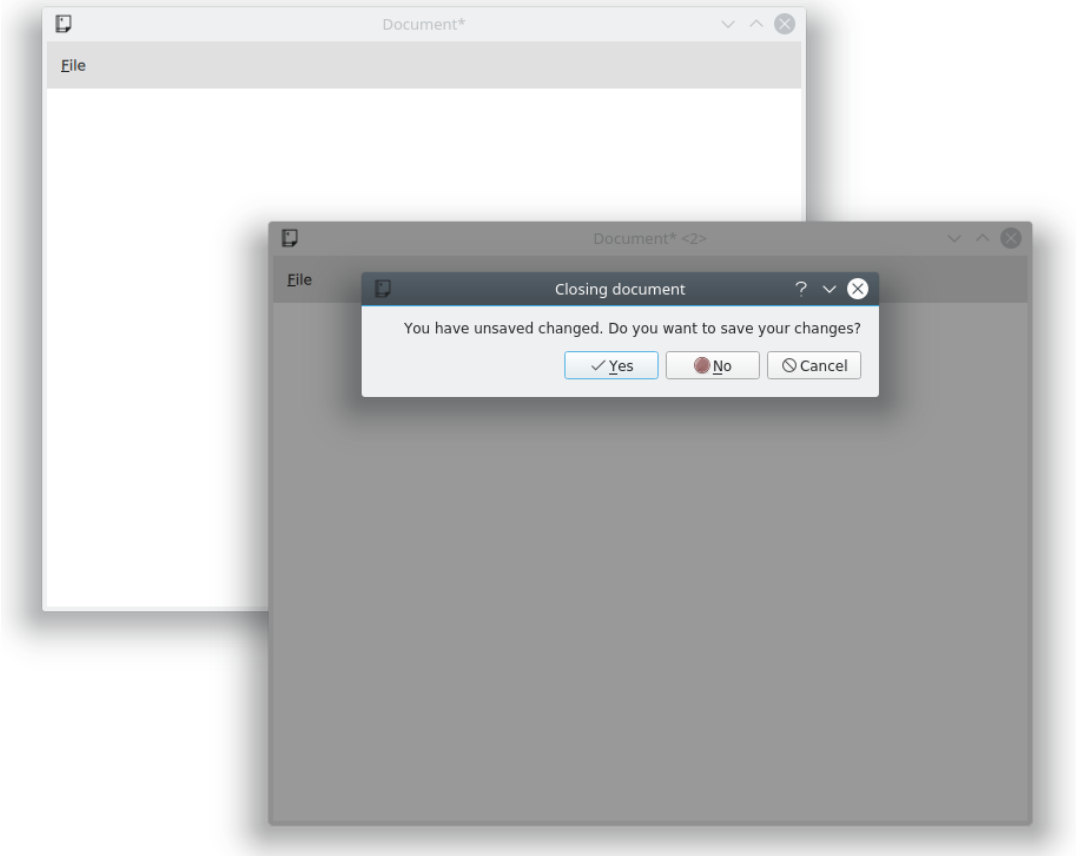
The back button explicitly calls the `setCurrentIndex` of the `SwipeView` to set the index to zero, returning the user directly to the *Current* page. During each transition between pages the `SwipeView` provides a transition, so even when explicitly changing the index the user is given a sense of direction.

> **TIP**
>
> When navigating in a `SwipeView` programatically it is important not to set the `currentIndex` by assignment in JavaScript. This is because doing so will break any QML bindings it overrides. Instead use the methods `setCurrentIndex`, `incrementCurrentIndex`, and `decrementCurrentIndex`. This preserves the QML bindings.

```
Page {

    // ...

    Column {
        anchors.centerIn: parent
        spacing: 10
        Label {
            anchors.horizontalCenter: parent.horizontalCenter
            text: qsTr("Community statistics")
        }
        Button {
            anchors.horizontalCenter: parent.horizontalCenter
            text: qsTr("Back")
            onClicked: swipeView.setCurrentIndex(0);
        }
    }
}
```

Document Windows

This example shows how to implement a desktop-oriented, document-centric user interface. The idea is to have one window per document. When opening a new document, a new window is opened. To the user, each window is a self-contained world with a single document.



The code starts from an `ApplicationWindow` with a *File* menu with the standard operations: *New*, *Open*, *Save* and *Save As*. We put this in the `DocumentWindow.qml`.

We import `Qt.labs.platform` for native dialogs, and have made the subsequent changes to the project file and `main.cpp` as described in the section on native dialogs above.

```
import QtQuick
import QtQuick.Controls
import Qt.labs.platform as NativeDialogs

ApplicationWindow {
    id: root

    // ...
```

```
        menuBar: MenuBar {
            Menu {
                title: qsTr("&File")
                MenuItem {
                    text: qsTr("&New")
                    icon.name: "document-new"
                    onTriggered: root.newDocument()
                }
                MenuSeparator {}
                MenuItem {
                    text: qsTr("&Open")
                    icon.name: "document-open"
                    onTriggered: openDocument()
                }
                MenuItem {
                    text: qsTr("&Save")
                    icon.name: "document-save"
                    onTriggered: saveDocument()
                }
                MenuItem {
                    text: qsTr("Save &As...")
                    icon.name: "document-save-as"
                    onTriggered: saveAsDocument()
                }
            }
        }

        // ...

    }
```

To bootstrap the program, we create the first `DocumentWindow` instance from `main.qml`, which is the entry point of the application.

```
    import QtQuick

    DocumentWindow {
```

```
        visible: true
    }
```

In the example at the beginning of this chapter, each `MenuItem` calls a corresponding function when triggered. Let's start with the *New* item, which calls the `newDocument` function.

The function, in turn, relies on the `createNewDocument` function, which dynamically creates a new element instance from the `DocumentWindow.qml` file, i.e. a new `DocumentWindow` instance. The reason for breaking out this part of the new function is that we use it when opening documents as well.

Notice that we do not provide a parent element when creating the new instance using `createObject`. This way, we create new top level elements. If we had provided the current document as parent to the next, the destruction of the parent window would lead to the destruction of the child windows.

```
    ApplicationWindow {

        // ...

        function createNewDocument()
        {
            var component = Qt.createComponent("DocumentWindow.qml"
            var window = component.createObject();
            return window;
        }

        function newDocument()
        {
            var window = createNewDocument();
            window.show();
        }
```

```
    // ...

}
```

Looking at the *Open* item, we see that it calls the `openDocument` function. The function simply opens the `openDialog`, which lets the user pick a file to open. As we don't have a document format, file extension or anything like that, the dialog has most properties set to their default value. In a real world application, this would be better configured.

In the `onAccepted` handler a new document window is instantiated using the `createNewDocument` method, and a file name is set before the window is shown. In this case, no real loading takes place.

> **TIP**
>
> We imported the `Qt.labs.platform` module as `NativeDialogs`. This is because it provides a `MenuItem` that clashes with the `MenuItem` provided by the `QtQuick.Controls` module.

```
ApplicationWindow {

    // ...

    function openDocument(fileName)
    {
        openDialog.open();
    }

    NativeDialogs.FileDialog {
        id: openDialog
        title: "Open"
        folder: NativeDialogs.StandardPaths.writableLocation(Na
        onAccepted: {
            var window = root.createNewDocument();
```

```
                window.fileName = openDialog.file;
                window.show();
            }
        }

        // ...

    }
```

The file name belongs to a pair of properties describing the document: `fileName` and `isDirty`. The `fileName` holds the file name of the document name and `isDirty` is set when the document has unsaved changes. This is used by the save and save as logic, which is shown below.

When trying to save a document without a name, the `saveAsDocument` is invoked. This results in a round-trip over the `saveAsDialog`, which sets a file name and then tries to save again in the `onAccepted` handler.

Notice that the `saveAsDocument` and `saveDocument` functions correspond to the *Save As* and *Save* menu items.

After having saved the document, in the `saveDocument` function, the `tryingToClose` property is checked. This flag is set if the save is the result of the user wanting to save a document when the window is being closed. As a consequence, the window is closed after the save operation has been performed. Again, no actual saving takes place in this example.

```
    ApplicationWindow {

        // ...

        property bool isDirty: true         // Has the document got
        property string fileName            // The filename of the d
        property bool tryingToClose: false // Is the window trying

        // ...
```

```
    function saveAsDocument()
    {
        saveAsDialog.open();
    }

    function saveDocument()
    {
        if (fileName.length === 0)
        {
            root.saveAsDocument();
        }
        else
        {
            // Save document here
            console.log("Saving document")
            root.isDirty = false;

            if (root.tryingToClose)
                root.close();
        }
    }

    NativeDialogs.FileDialog {
        id: saveAsDialog
        title: "Save As"
        folder: NativeDialogs.StandardPaths.writableLocation(Na
        onAccepted: {
            root.fileName = saveAsDialog.file
            saveDocument();
        }
        onRejected: {
            root.tryingToClose = false;
        }
    }

    // ...

}
```

This leads us to the closing of windows. When a window is being closed, the `onClosing` handler is invoked. Here, the code can choose not to accept the request to close. If the document has unsaved changes, we open the `closeWarningDialog` and reject the request to close.

The `closeWarningDialog` asks the user if the changes should be saved, but the user also has the option to cancel the close operation. The cancelling, handled in `onRejected`, is the easiest case, as we rejected the closing when the dialog was opened.

When the user does not want to save the changes, i.e. in `onNoClicked`, the `isDirty` flag is set to `false` and the window is closed again. This time around, the `onClosing` will accept the closure, as `isDirty` is false.

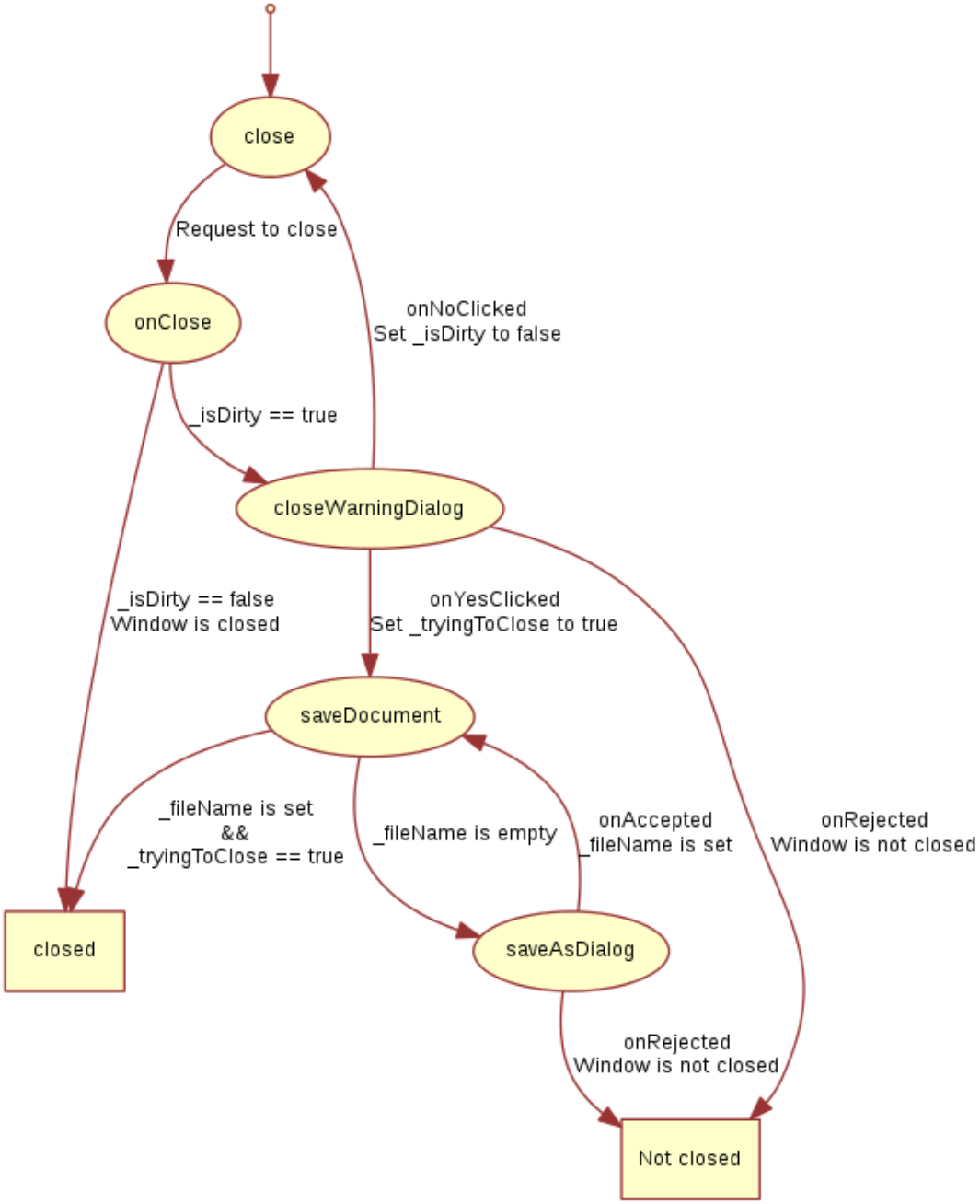
Finally, when the user wants to save the changes, we set the `tryingToClose` flag to true before calling save. This leads us to the save/save as logic.

```
ApplicationWindow {

    // ...

    onClosing: {
        if (root.isDirty) {
            closeWarningDialog.open();
            close.accepted = false;
        }
    }

    NativeDialogs.MessageDialog {
        id: closeWarningDialog
        title: "Closing document"
        text: "You have unsaved changed. Do you want to save yo
        buttons: NativeDialogs.MessageDialog.Yes | NativeDialog
        onYesClicked: {
            // Attempt to save the document
            root.tryingToClose = true;
```

```
                root.saveDocument();
            }
            onNoClicked: {
                // Close the window
                root.isDirty = false;
                root.close()
            }
            onRejected: {
                // Do nothing, aborting the closing of the window
            }
        }
    }
```

The entire flow for the close and save/save as logic is shown below. The system is entered at the *close* state, while the *closed* and *not closed* states are outcomes.

This looks complicated compared to implementing this using `Qt Widgets` and C++. This is because the dialogs are not blocking to QML. This means that we cannot wait for the outcome of a dialog in a `switch` statement. Instead we need to remember the state and continue the operation in the respective `onYesClicked`, `onNoClicked`, `onAccepted`, and `onRejected` handlers.



The final piece of the puzzle is the window title. It is composed of the `fileName` and `isDirty` properties.

```
ApplicationWindow {

    // ...

    title: (fileName.length===0?qsTr("Document"):fileName) + (i
```

```
        // ...

    }
```

This example is far from complete. For instance, the document is never loaded or saved. Another missing piece is handling the case of closing all the windows in one go, i.e. exiting the application. For this function, a singleton maintaining a list of all current `DocumentWindow` instances is needed. However, this would only be another way to trigger the closing of a window, so the logic flow shown here is still valid.

The Imagine Style

One of the goals with Qt Quick Controls is to separate the logic of a control from its appearance. For most of the styles, the implementation of the appearance consists of a mix of QML code and graphical assets. However, using the *Imagine* style, it is possible to customize the appearance of a Qt Quick Controls based application using only graphical assets.

The imagine style is based on **9-patch images (https://developer.android.com/guide/topics/graphics/drawables#nine-patch)** . This allows the images to carry information on how they are stretched and what parts are to be considered as a part of the element and what is outside; e.g. a shadow. For each control, the style supports several elements, and for each element a large number of states are available. By providing assets for certain combinations of these elements and states, you can control the appearance of each control in detail.

The details of 9-patch images, and how each control can be styled is covered in great detail in the **Imagine style documentation (https://doc.qt.io/qt-6/qtquickcontrols2-imagine.html)** . Here, we will create a custom style for an imaginary device interface to demonstrate how the style is used.

The application's style customizes the `ApplicationWindow` and `Button` controls. For the buttons, the normal state, as well as the *pressed* and *checked* states are handled. The demonstration application is shown below.

The code for this uses a `Column` for the clickable buttons, and a `Grid` for the checkable ones. The clickable buttons also stretch with the window width.

```
import QtQuick
import QtQuick.Controls

ApplicationWindow {

    // ...

    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Column {
        anchors.top: parent.top
```

```
    anchors.left: parent.left
    anchors.margins: 10

    width: parent.width/2

    spacing: 10

    // ...

    Repeater {
        model: 5
        delegate: Button {
            width: parent.width
            height: 70
            text: qsTr("Click me!")
        }
    }
}

Grid {
    anchors.top: parent.top
    anchors.right: parent.right
    anchors.margins: 10

    columns: 2

    spacing: 10

    // ...

    Repeater {
        model: 10

        delegate: Button {
            height: 70
            text: qsTr("Check me!")
            checkable: true
        }
    }
```

```
        }
    }
```

As we are using the *Imagine* style, all controls that we want to use need to be styled using a graphical asset. The easiest is the background for the `ApplicationWindow`. This is a single pixel texture defining the background colour. By naming the file `applicationwindow-background.png` and then pointing the style to it using the `qtquickcontrols2.conf` file, the file is picked up.

In the `qtquickcontrols2.conf` file shown below, you can see how we set the `Style` to `Imagine`, and then setup a `Path` for the style where it can look for the assets. Finally we set some palette properties as well. The available palette properties can be found on the **palette QML Basic Type (https://doc.qt.io/qt-6/qml-palette.html#qtquickcontrols2-palette)** page.

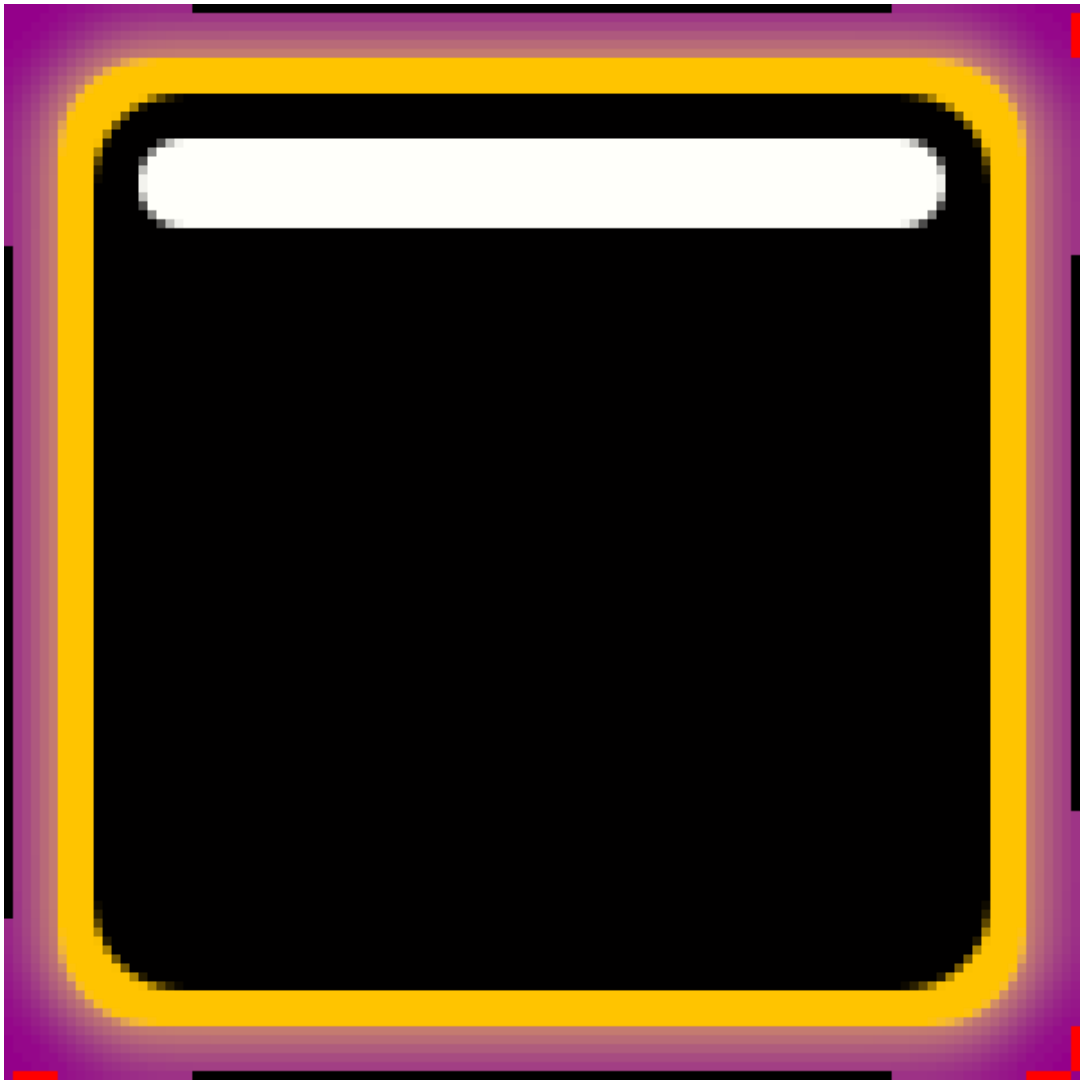
```
[Controls]
Style=Imagine

[Imagine]
Path=:images/imagine

[Imagine\Palette]
Text=#ffffff
ButtonText=#ffffff
BrightText=#ffffff
```

The assets for the `Button` control are `button-background.9.png`, `button-background-pressed.9.png` and `button-background-checked.9.png`. These follow the *control-element-state* pattern. The stateless file, `button-background.9.png` is used for all states without a specific asset. According to the **Imagine style element reference table**

(https://doc.qt.io/qt-6/qtquickcontrols2-imagine.html#element-reference) , a button can have the following states:

- `disabled`
- `pressed`
- `checked`
- `checkable`
- `focused`
- `highlighted`
- `flat`
- `mirrored`
- `hovered`

The states that are needed depend on your user interface. For instance, the hovered style is never used for touch-based interfaces.



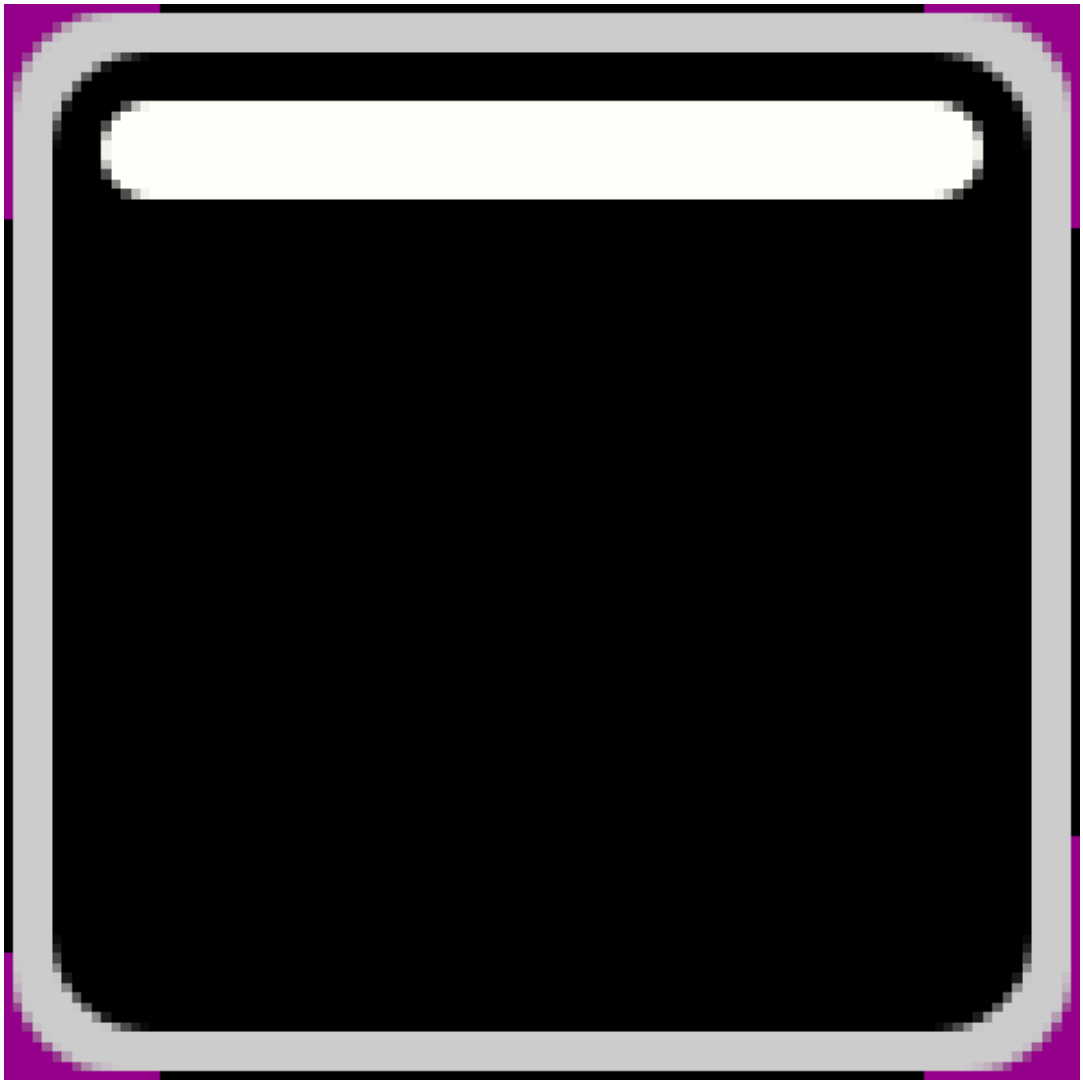
Looking at an enlarged version of `button-background-checked.9.png` above, you can see the 9-patch guide lines along the sides. The purple background has been added for visibility reasons. This area is actually transparent in the asset used in the example.

The pixes along the edges of the image can be either white/transparent, black, or red. These have different meanings that we will go through one by one.

- **Black** lines along the **left** and **top** sides of the asset mark the stretchable parts of the image. This means that the rounded corners and the white marker in the example are not affected when the button is stretched.

- **Black** lines along the **right** and **bottom** sides of the asset mark the area used for the control's contents. That means the part of the button that is used for text in the example.
- **Red** lines along the **right** and **bottom** sides of the asset mark *inset* areas. These areas are a part of the image, but not considered a part of the control. For the checked image above, this is used for a soft halo extending outside the button.

A demonstration of the usage of an *inset* area is shown in `button-background.9.png` (below) and `button-background-checked.9.png` (above): the image seems to light up, but not move.



Summary

In this chapter we have looked at Qt Quick Controls 2. They offer a set of elements that provide more high-level concepts than the basic QML elements. For most scenarios, you will save memory and gain performance by using the Qt Quick Controls 2, as they are based around optimized C++ logic instead of Javascript and QML.

We've demonstrated how different styles can be used, and how a common code base can be developed using file selectors. This way, a single code base can address multiple platforms with different user interactions and visual styles.

Finally, we have looked at the Imagine style, which allows you to completely customize the look of a QML application through the use of graphical assets. In this way, an application can be reskinned without any code change whatsoever.

Model-View-Delegate

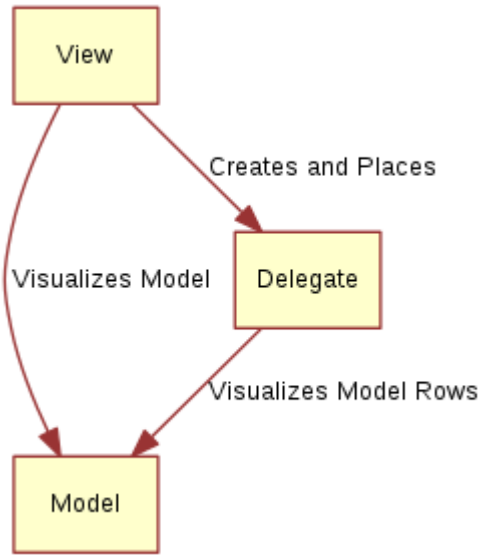
As soon as the amount of data goes beyond the trivial, it is no longer feasible to keep a copy of the data with the presentation. This means that the presentation layer, what is seen by the user, needs to be separated by the data layer, the actual contents. In Qt Quick, data is separated from the presentation through a so called model-view separation. Qt Quick provides a set of premade views in which each data element is the visualization by a delegate. To utilize the system, one must understand these classes and know how to create appropriate delegates to get the right look and feel.

Concept

A common pattern when developing user interfaces is to keep the representation of the data separate from the visualization. This makes it possible to show the same data in different ways depending on what task the user is performing. For instance, a phone book could be arranged as a vertical list of text entries, or as a grid of pictures of the contacts. In both cases, the data is identical: the phone book, 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 phone book 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.

This means that the delegate knows about the contents of the model and how to visualize it. The view knows about the concept of delegates and how to lay them out. The model only knows about the data it is representing.


View
Creates and Places
Visualizes Model
Delegate
Visualizes Model Rows
Model

Basic Models

The most basic way to visualize data from a model 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 full-blown model gathering data from the Internet.

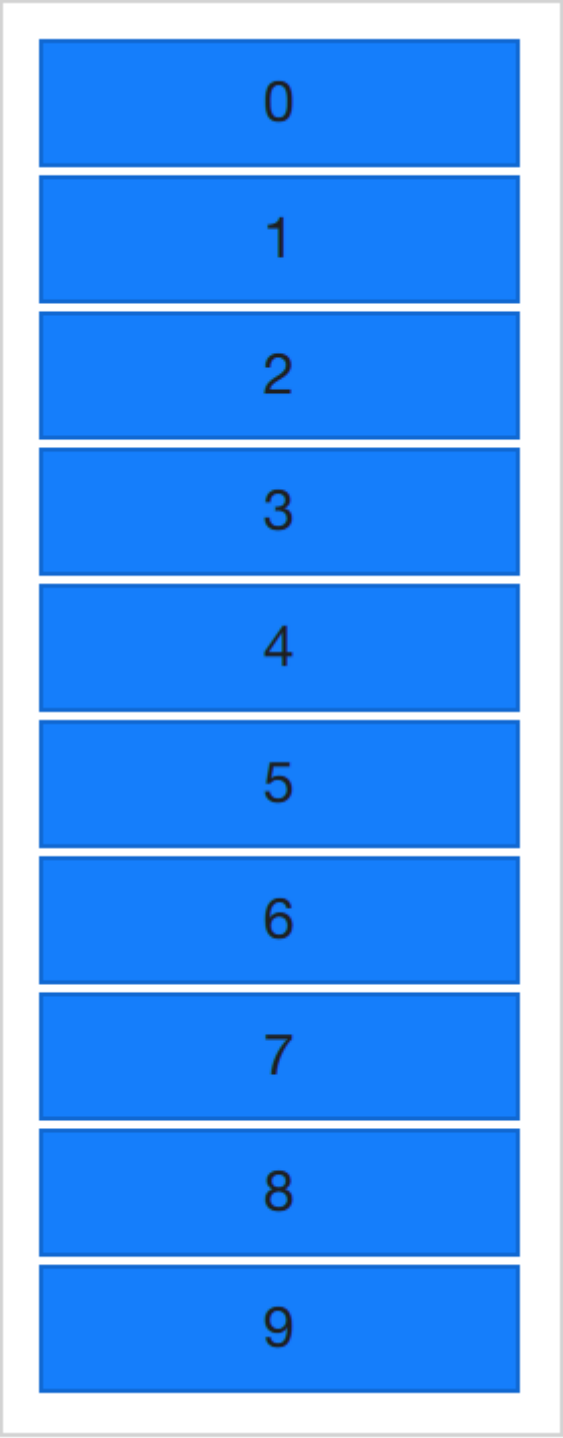
Using a number

In its simplest form, the repeater can be used to instantiate a specified number of items. Each item will have access to an attached property, the variable `index`, that can be used to tell the items apart.

In the example below, a repeater is used to create 10 instances of an item. The number of items is controlled using the `model` property and their visual representation is controlled using the `delegate` property. For each item in the model, a delegate is instantiated (here, the delegate is a `BlueBox`, which is a customized `Rectangle` containing a `Text` element). As you can tell, the `text` property is set to the `index` value, thus the items are numbered from zero to nine.

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

Column {
    spacing: 2

    Repeater {
        model: 10
```

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



Using an array

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
import "../common"

Column {
    spacing: 2

    Repeater {
        model: ["Enterprise", "Columbia", "Challenger", "Discov

        delegate: BlueBox {
            required property var modelData
            required property int index

            width: 100
            height: 32
            radius: 3

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

Using a `ListModel`

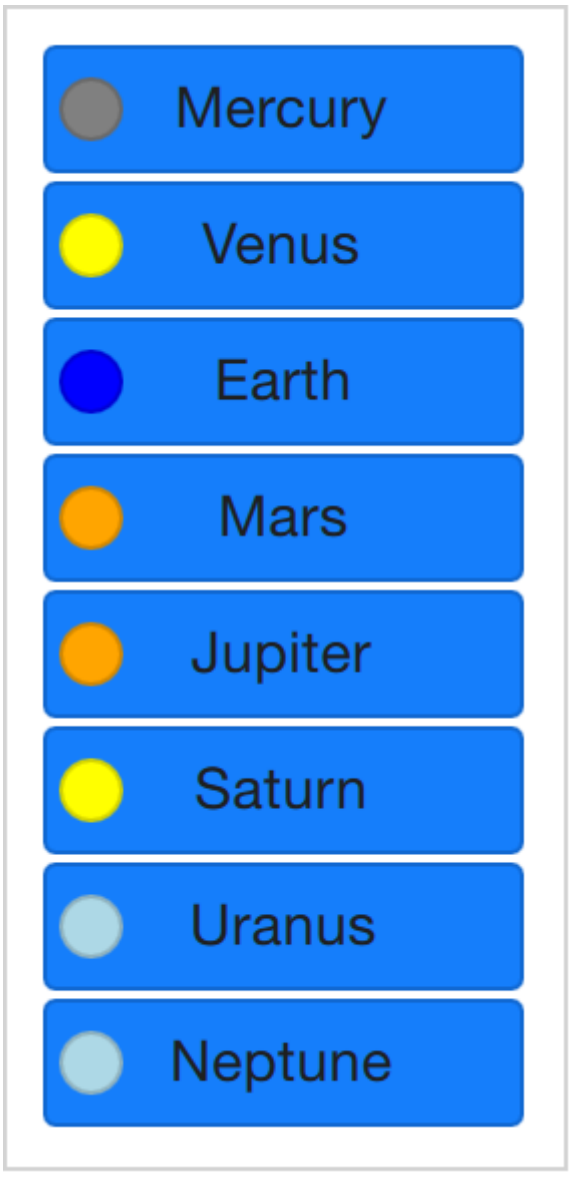
Being able to expose the data of an array, you soon find yourself in a position where you need multiple pieces of data per item in the array. This is where models enter the picture. One of the most trivial models and one of the most commonly used is the `ListModel`. A list model is simply a collection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

Column {
    spacing: 2

    Repeater {
        model: ListModel {
            ListElement { name: "Mercury"; surfaceColor: "gray"
            ListElement { name: "Venus"; surfaceColor: "yellow"
            ListElement { name: "Earth"; surfaceColor: "blue" }
            ListElement { name: "Mars"; surfaceColor: "orange"
            ListElement { name: "Jupiter"; surfaceColor: "orang
            ListElement { name: "Saturn"; surfaceColor: "yellow
            ListElement { name: "Uranus"; surfaceColor: "lightB
            ListElement { name: "Neptune"; surfaceColor: "light
        }

        delegate: BlueBox {
            id: blueBox

            required property string name
            required property color surfaceColor

            width: 120
            height: 32

            radius: 3
            text: name

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

                width: 16
```

```
                height: 16

                radius: 8

                color: blueBox.surfaceColor
            }
        }
    }
}
```



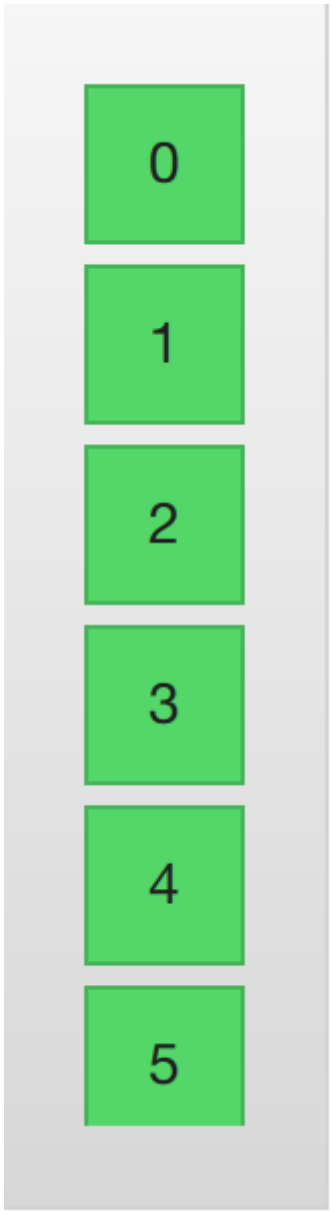
Using a delegate as default property

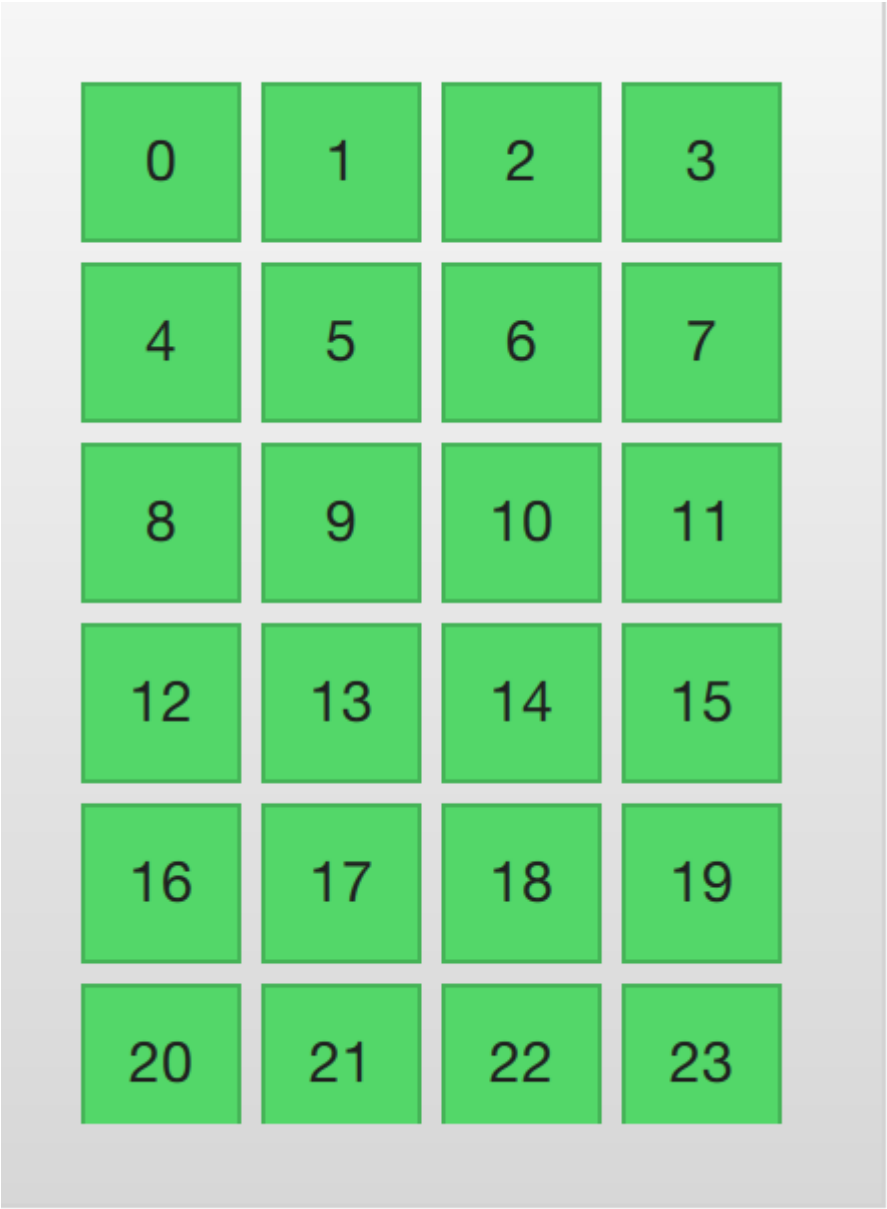
The `delegate` property of the `Repeater` is its default property. This means that it's also possible to write the code of Example 01 as follows:

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

Column {
    spacing: 2

    Repeater {
        model: 10

        BlueBox {
            required property int index
            width: 100
            height: 32
            text: index
        }
    }
}
```

Dynamic Views

Repeaters work well for limited and static sets of data, but in the real world, models are commonly more complex – and larger. Here, a smarter solution is needed. For this, Qt Quick provides the `ListView` and `GridView` elements. These are both based on a `Flickable` area, so the user can move around in a larger dataset. 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.

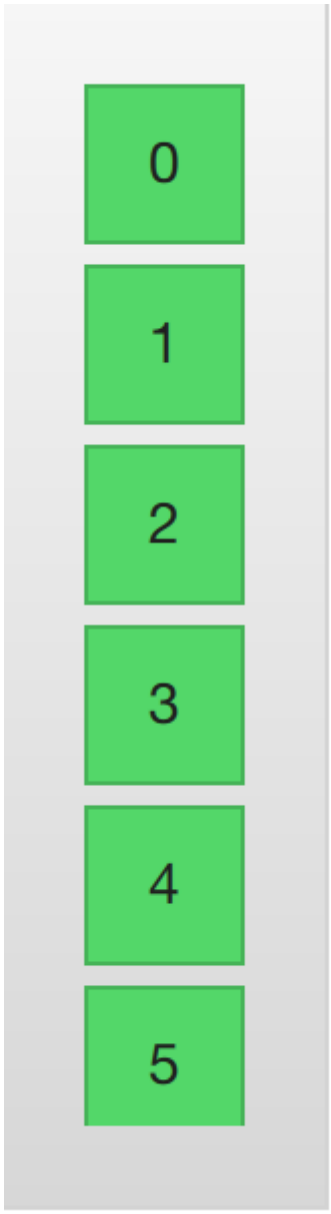

0
1
2
3
4
5



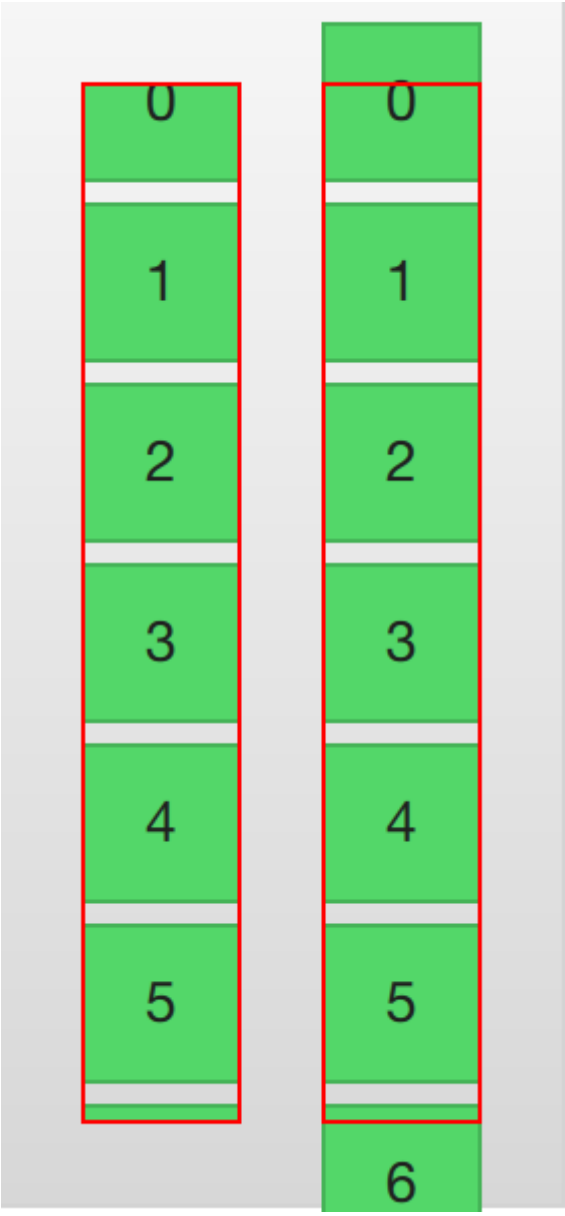
The two elements are similar in their usage. We will begin with the `ListView` and then describe the `GridView` with the former as the starting point of the comparison. Notice that the `GridView` places a list of items into a two-dimensional grid, left-to-right or top-to-bottom. If you want to show a table of data you need to use the `TableView` which is described in the Table Models section.

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
import "../common"

Background {
    width: 80
    height: 300

    ListView {
        anchors.fill: parent
        anchors.margins: 20
        clip: true
        model: 100
        delegate: GreenBox {
            required property int index
            width: 40
            height: 40
            text: index
        }
        spacing: 5
    }
}
```



If the model contains more data than can fit onto the screen, the `ListView` only shows part of the list. However, as a consequence of the default behavior of Qt Quick, the list view does not limit the screen area within which the delegates are shown. This means that delegates may be visible outside the list view and that the dynamic creation and destruction of delegates outside the list view is visible to the user. To prevent this, clipping must be activated on the `ListView` element by setting the `clip` property to `true`. The illustration below shows the result of this (left view), compared to when the `clip` property is `false` (right view).



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 conimaged 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
import "../common"

Background {
    width: 480
    height: 80

    ListView {
        anchors.fill: parent
        anchors.margins: 20
        spacing: 4
        clip: true
```

```
            model: 100
            orientation: ListView.Horizontal
            delegate: GreenBox {
                required property int index
                width: 40
                height: 40
                text: index
            }
        }
    }
```

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

Keyboard Navigation and Highlighting

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

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

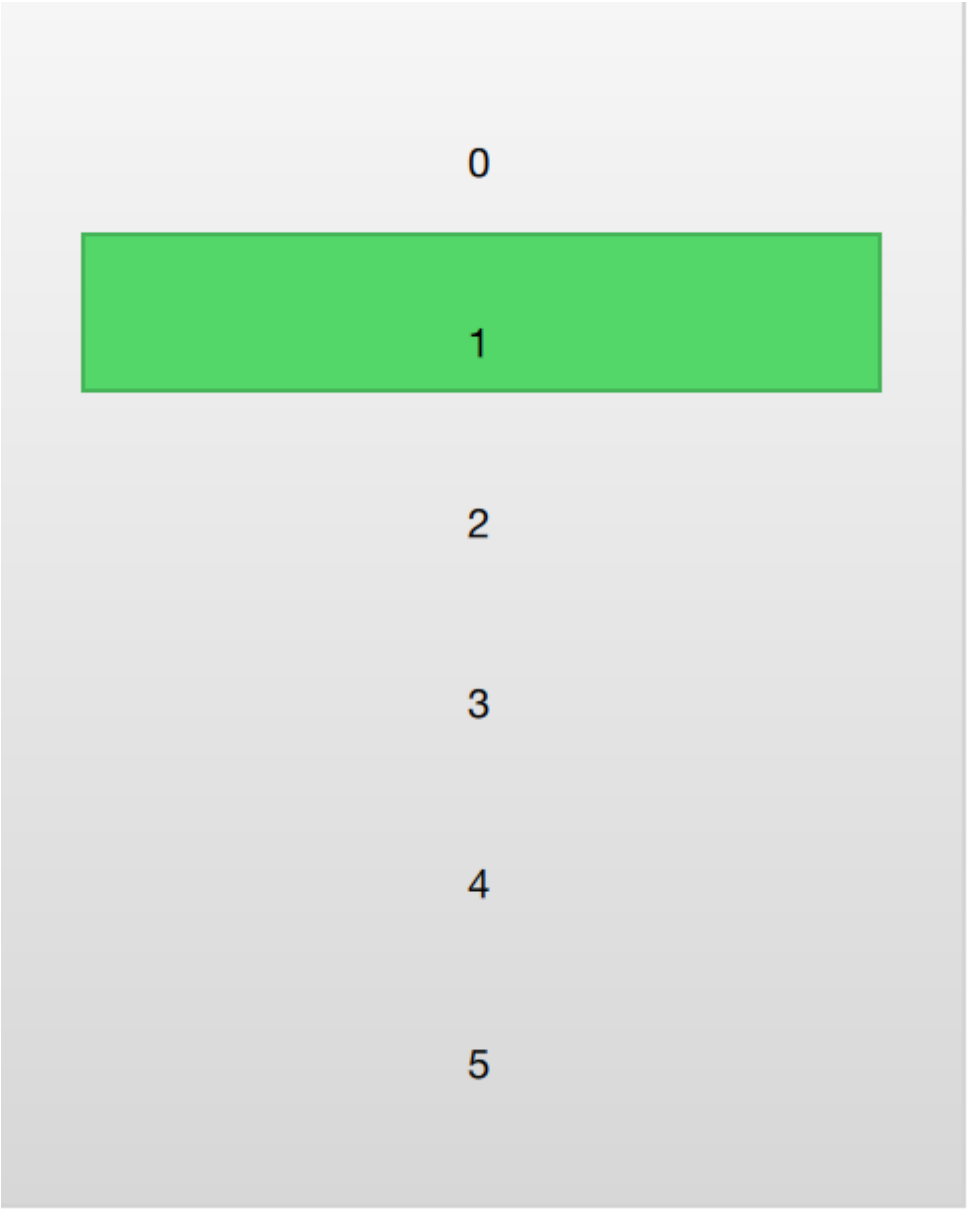
In the example below this is demonstrated. There are two properties involved for this to work. First, the `focus` property is set to true. This

gives the `ListView` the keyboard focus. Second, the `highlight` property is set to point out the highlighting delegate to use. The highlight delegate is given the `x`, `y` and `height` of the current item. If the `width` is not specified, the width of the current item is also used.

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

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

Background {
    width: 240
    height: 300

    ListView {
        id: view

        anchors.fill: parent
        anchors.margins: 20

        focus: true

        model: 100
        delegate: numberDelegate
        highlight: highlightComponent

        spacing: 5
        clip: true
    }

    Component {
        id: highlightComponent

        GreenBox {
            width: ListView.view ? ListView.view.width : 0
```

```
        }
    }

    Component {
        id: numberDelegate

        Item {
            id: wrapper

            required property int index

            width: ListView.view ? ListView.view.width : 0
            height: 40

            Text {
                anchors.centerIn: parent
                font.pixelSize: 10
                text: wrapper.index
            }
        }
    }
}
```



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

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

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

In the default configuration, the view is responsible for moving the highlight into position. The speed of the movement and resizing can be controlled, either as a velocity 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 ? ListView.view.width : 0
            height: ListView.view ? ListView.view.currentItem.heigh

            y: ListView.view ? ListView.view.currentItem.y : 0

            Behavior on y {
                SequentialAnimation {
                    PropertyAnimation { target: highlightRectangle;
                    NumberAnimation { duration: 1 }
                    PropertyAnimation { target: highlightRectangle;
                }
            }

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

Header and Footer

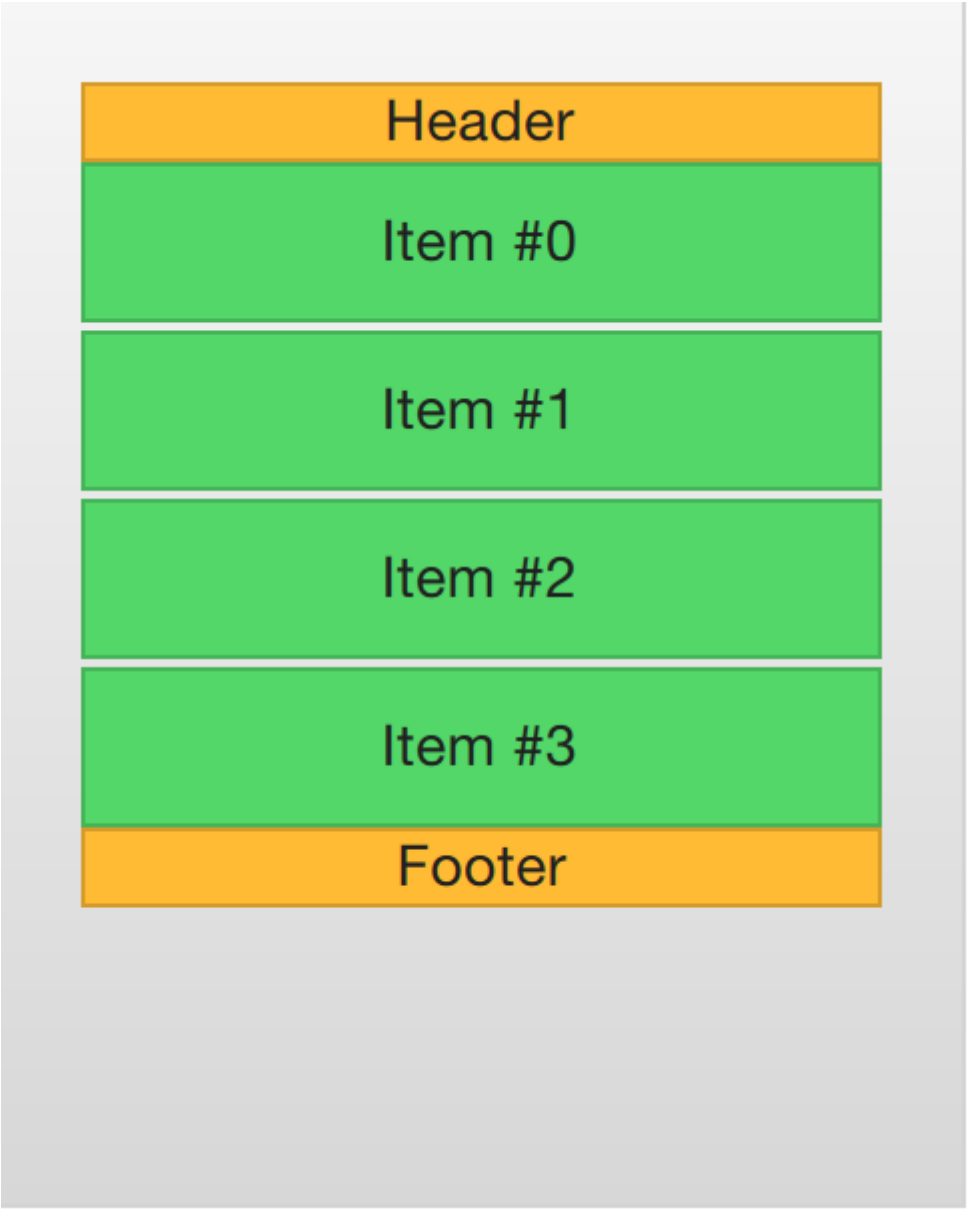
At each end of the `ListView` contents, a `header` and a `footer` element can be inserted. These can be considered special delegates placed at the beginning or end of the list. For a horizontal list, these will not appear at the head or foot, but rather at the beginning or end, depending on the `layoutDirection`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

The example below illustrates how a 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
import "../common"
```

```
Background {
    width: 240
    height: 300

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

        clip: true

        model: 4
        delegate: numberDelegate
        header: headerComponent
        footer: footerComponent

        spacing: 2
    }

    Component {
        id: headerComponent

        YellowBox {
            width: ListView.view ? ListView.view.width : 0
            height: 20
            text: 'Header'

        }
    }

    Component {
        id: footerComponent

        YellowBox {
            width: ListView.view ? ListView.view.width : 0
            height: 20
            text: 'Footer'
        }
    }
```

```
    Component {
        id: numberDelegate

        GreenBox {
            required property int index

            width: ListView.view.width
            height: 40

            text: 'Item #' + index
        }
    }
}
```

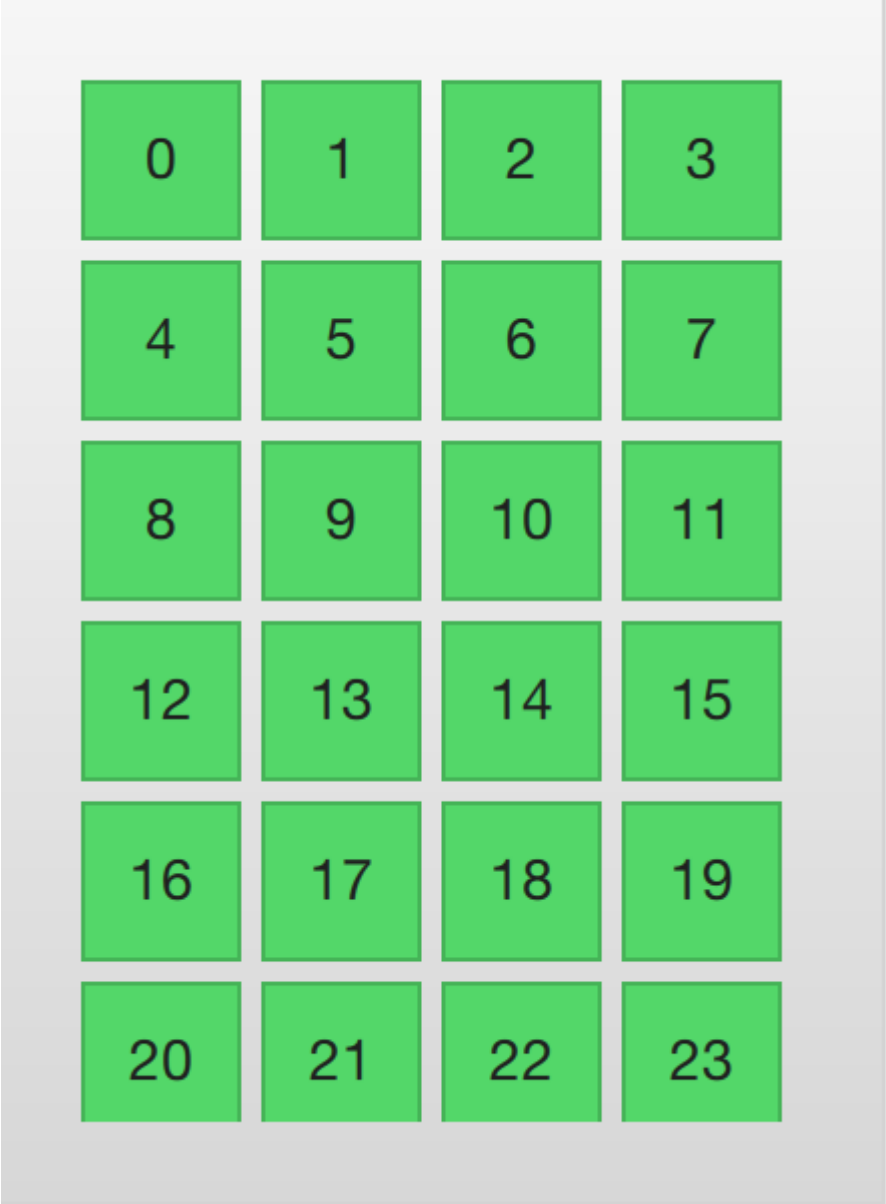


TIP

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 placed in the top left corner of each such cell.

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

Background {
    width: 220
    height: 300

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

        clip: true

        model: 100

        cellWidth: 45
        cellHeight: 45

        delegate: GreenBox {
            required property int index
            width: 40
            height: 40
            text: index
        }
    }
}
```

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

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

adds items from the top to the bottom, filling the view from left to right. The scrolling direction is horizontal in this case.

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

Delegate

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

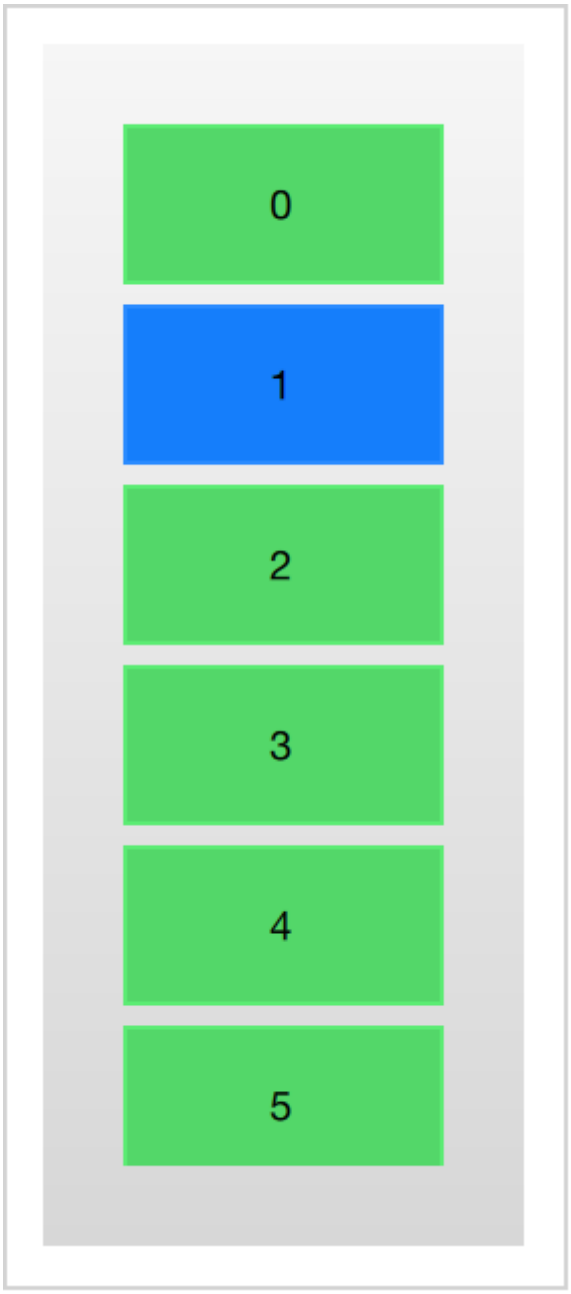
Each delegate gets access to a number of attached properties, some from the data model, others from the view. From the model, the properties convey the data for each item to the delegate. From the view, the properties convey state information related to the delegate within the view. Let's dive into the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

Rectangle {
    width: 120
    height: 300

    gradient: Gradient {
```

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

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

    focus: true

    model: 100
    delegate: numberDelegate

    spacing: 5
    clip: true
}

Component {
    id: numberDelegate

    Rectangle {
        id: wrapper

        required property int index

        width: ListView.view.width
        height: 40

        color: ListView.isCurrentItem ? "#157efb" : "#53d76
        border.color: Qt.lighter(color, 1.1)

        Text {
            anchors.centerIn: parent

            font.pixelSize: 10

            text: wrapper.index
        }
    }
```

```
        }
    }
```



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

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

Animating Added and Removed Items

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

Conveniently enough, QML views attach two signals, `onAdd` and `onRemove`, to each item delegate. By triggering animations from 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` called `addAnimation` is started from the signal causes the item to zoom into view by animating the `scale` property of the delegate.

```
GridView.onAdd: addAnimation.start()

SequentialAnimation {
    id: addAnimation
    NumberAnimation {
        target: wrapper
        property: "scale"
        from: 0
```

```
            to: 1
            duration: 250
            easing.type: Easing.InOutQuad
        }
    }
```

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, starting the `removeAnimation` `SequentialAnimation`. This time, however, the destruction of the delegate must be delayed until the animation has completed. To do this, `PropertyAction` element is 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.

```
    GridView.onRemove: removeAnimation.start()

    SequentialAnimation {
        id: removeAnimation

        PropertyAction { target: wrapper; property: "GridView.delay
        NumberAnimation { target: wrapper; property: "scale"; to: 0
        PropertyAction { target: wrapper; property: "GridView.delay
    }
```

Here is the complete code:

```
    import QtQuick

    Rectangle {
        width: 480
        height: 300

        gradient: Gradient {
```

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

    ListModel {
        id: theModel

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

    Rectangle {
        property int count: 9

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

        height: 40

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

        Text {
            anchors.centerIn: parent

            text: "Add item!"
        }

        MouseArea {
            anchors.fill: parent
```

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

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

        required property int index
        required property int number

        width: 40
        height: 40

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

        Text {
```

```
        anchors.centerIn: parent

        font.pixelSize: 10

        text: wrapper.number
    }

    MouseArea {
        anchors.fill: parent

        onClicked: {
            if (wrapper.index == -1) {
                return
            }
            theModel.remove(wrapper.index)
        }
    }

    GridView.onRemove: removeAnimation.start()

    SequentialAnimation {
        id: removeAnimation

        PropertyAction { target: wrapper; property: "Gr
        NumberAnimation { target: wrapper; property: "s
        PropertyAction { target: wrapper; property: "Gr
    }

    GridView.onAdd: addAnimation.start()

    SequentialAnimation {
        id: addAnimation
        NumberAnimation {
            target: wrapper
            property: "scale"
            from: 0
            to: 1
            duration: 250
            easing.type: Easing.InOutQuad
        }
```

```
                }
            }
        }
    }
```

Shape-Shifting Delegates

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

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

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

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

As the item first is clicked, it enters the `expanded` state, causing the item delegate to fill the `ListView` and the contents to rearrange. When the

close button is clicked, the state is cleared, causing the delegate to return to its previous state and re-enabling the `ListView`.

```
import QtQuick

Item {
    width: 300
    height: 480

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

    ListView {
        id: listView

        anchors.fill: parent

        delegate: detailsDelegate
        model: planets
    }

    ListModel {
        id: planets

        ListElement { name: "Mercury"; imageSource: "images/mer
        ListElement { name: "Venus"; imageSource: "images/venus
        ListElement { name: "Earth"; imageSource: "images/earth
        ListElement { name: "Mars"; imageSource: "images/mars.j
    }

    Component {
        id: detailsDelegate

        Item {
```

```
id: wrapper

required property string name
required property string imageSource
required property string facts

width: listView.width
height: 30

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

    height: 30

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

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

        text: wrapper.name
    }
}


Rectangle {
    id: image

    width: 26
    height: 26

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

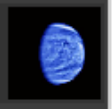
```
        anchors.topMargin: 2

        color: "black"


        Image {
            anchors.fill: parent

            fillMode: Image.PreserveAspectFit

            source: wrapper.imageSource
        }
    }

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

    Item {
        id: factsView

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

        opacity: 0

        Rectangle {
            anchors.fill: parent

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

            Text {
```

```
            anchors.fill: parent
            anchors.margins: 5

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

            font.pixelSize: 12

            text: wrapper.facts
        }
    }
}

Rectangle {
    id: closeButton

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

    width: 26
    height: 26

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

    opacity: 0

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

states: [
    State {
        name: "expanded"
```

```
                PropertyChanges { target: wrapper; height:
                PropertyChanges { target: image; width: lis
                PropertyChanges { target: factsView; opacit
                PropertyChanges { target: closeButton; opac
                PropertyChanges { target: wrapper.ListView.
            }
        ]

        transitions: [
            Transition {
                NumberAnimation {
                    duration: 200;
                    properties: "height,width,anchors.right
                }
            }
        ]
    }
}
}
```

Mercury

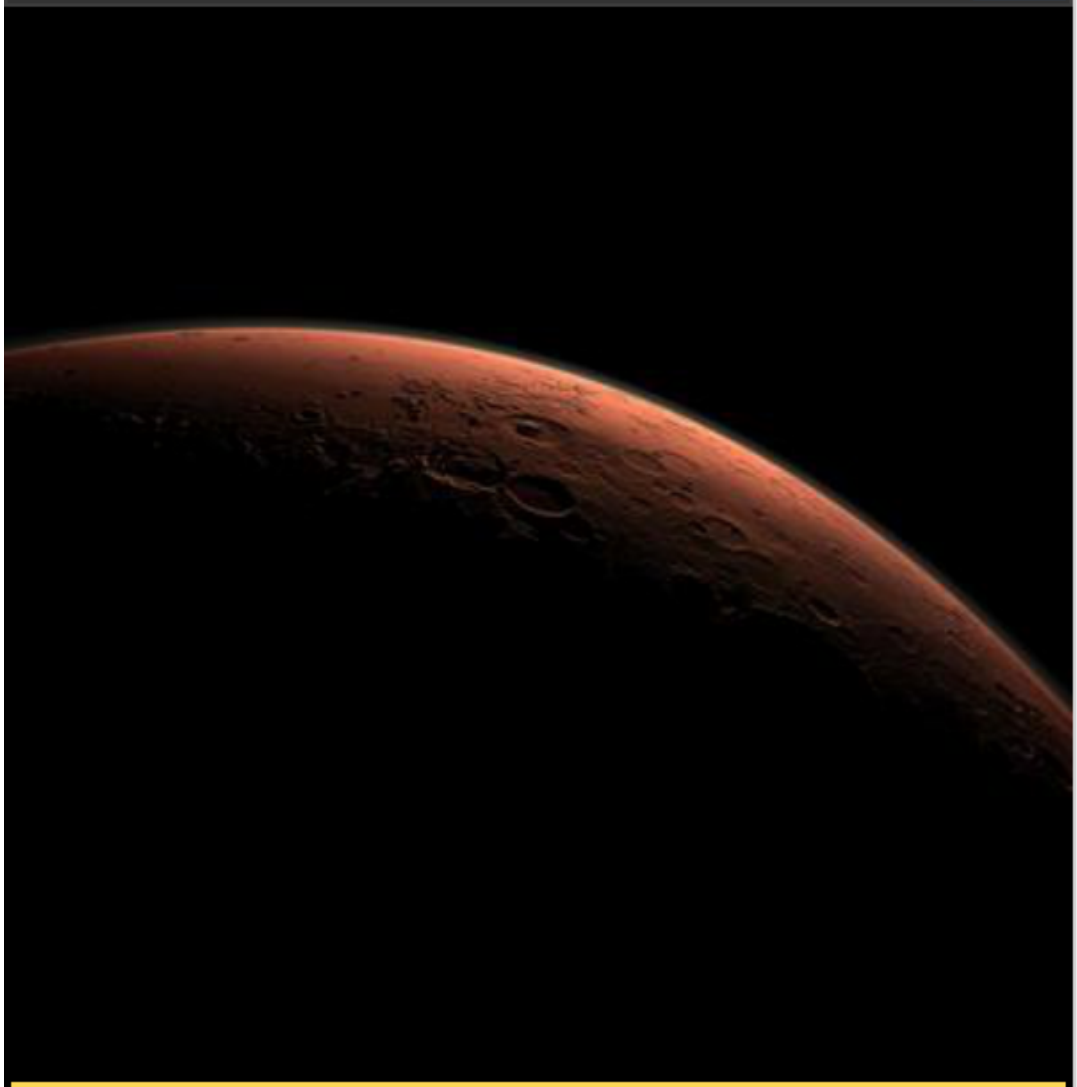
Venus



Earth

Mars

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.

Advanced Techniques

The PathView

The `PathView` element is the most flexible view provided in Qt Quick, but it is also the most complex. 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.

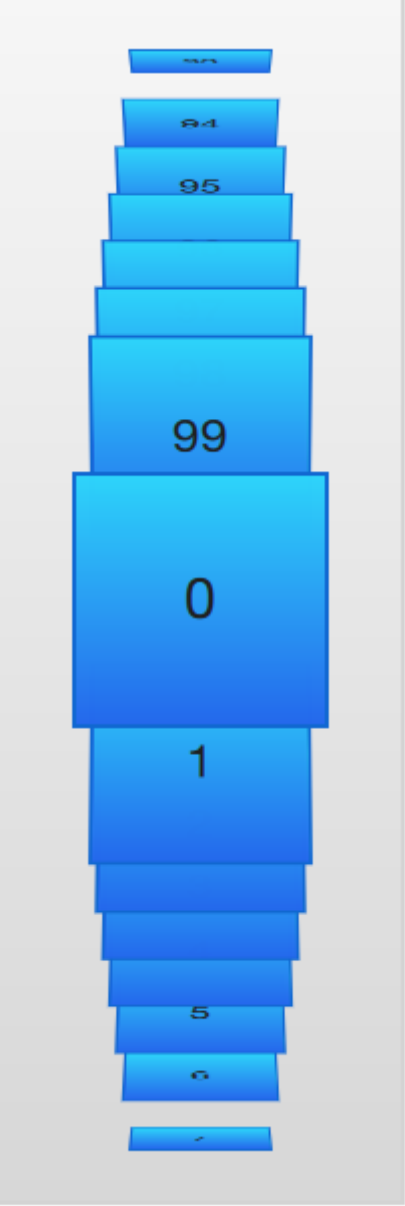
Before looking at the highlight range control properties in depth, we must look at the `path` property. The `path` property expects a `Path` element defining the path that the delegates follow as the `PathView` is being scrolled. The path is defined using the `startX` and `startY` properties in combinations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

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 provide 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 too, 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 a 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 is 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

    model: 100
    delegate: flipCardDelegate

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

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

    pathItemCount: 16

    preferredHighlightBegin: 0.5
    preferredHighlightEnd: 0.5
}
```

The delegate, shown below, utilizes the attached properties `itemZ`, `itemAngle` and `itemScale` from the `PathAttribute` elements. It is

worth noticing that the attached properties of the delegate only are available from the `wrapper`. Thus, the `rotX` property is defined to be able to access the value from within the `Rotation` element.

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

```
Component {
    id: flipCardDelegate

    BlueBox {
        id: wrapper

        required property int index
        property real rotX: PathView.itemAngle

        visible: PathView.onPath

        width: 64
        height: 64
        scale: PathView.itemScale
        z: PathView.itemZ

        antialiasing: true

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

        transform: Rotation {
            axis { x: 1; y: 0; z: 0 }
            angle: wrapper.rotX
```

```
            origin { x: 32; y: 32; }
        }

        text: wrapper.index
    }
}
```

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

> **TIP**
>
> Given the dynamic nature of `PathAttribute`, the qml tooling (in this case: `qmllint`) is not aware of `itemZ`, `itemAngle` nor `itemScale`.

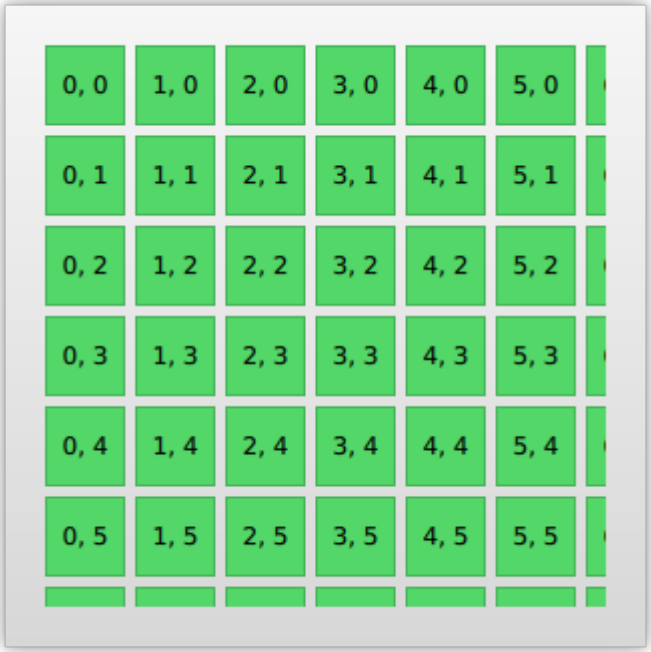
When using the `PathView` and changing the `currentIndex` programmatically you might want to control the direction that the path moves in. You can do this using the `movementDirection` property. It can be set to `PathView.Shortest`, which is the default value. This means that the movement can be either direction, depending on which way is the closest way to move to the target value. The direction can instead be restricted by setting `movementDirection` to `PathView.Negative` or `PathView.Positive`.

Table Models

All views discussed until now present an array of items one way or another. Even the `GridView` expects the model to provide a one dimensional list of items. For two dimensional tables of data you need to use the `TableView` element.

The `TableView` is similar to other views in that it combines a `model` with a `delegate` to form a grid. If given a list oriented model, it displays a single column, making it very similar to the `ListView` element. However, it can also display two-dimensional models that explicitly define both columns and rows.

In the example below, we set up a simple `TableView` with a custom model exposed from C++. At the moment, it is not possible to create table oriented models directly from QML, but in the 'Qt and C++' chapter the concept is explained. The running example is shown in the image below.



In the example below, we create a `TableView` and set the `rowSpacing` and `columnSpacing` to control the horizontal and vertical gaps between delegates. The rest of the properties are set up as for any other type of view.

```
TableView {
    id: view
    anchors.fill: parent
    anchors.margins: 20

    rowSpacing: 5
    columnSpacing: 5

    clip: true

    model: tableModel
    delegate: cellDelegate
}
```

The delegate itself can carry an implicit size through the `implicitWidth` and `implicitHeight`. This is what we do in the example below. The actual data contents, i.e. the data returned from the model's `display` role.

```
Component {
    id: cellDelegate

    GreenBox {
        id: wrapper

        required property string display

        implicitHeight: 40
        implicitWidth: 40

        Text {
            anchors.centerIn: parent
            text: wrapper.display
        }
    }
}
```

It is possible to provide delegates with different sizes depending on the model contents, e.g.:

```
GreenBox {
    implicitHeight: (1 + row) * 10
    // ...
}
```

Notice that both the width and the height must be greater than zero.

When providing an implicit size from the delegate, the tallest delegate of each row and the widest delegate of each column controls the size. This can create interesting behaviour if the width of items depend on the row, or if the height depends on the column. This is because not all delegates are instantiated at all times, so the width of a column might change as the user scrolls through the table.

To avoid the issues with specifying column widths and row heights using implicit delegate sizes, you can provide functions that calculate these sizes. This is done using the `columnWidthProvider` and `rowHeightProvider`. These functions return the size of the width and row respectively as shown below:

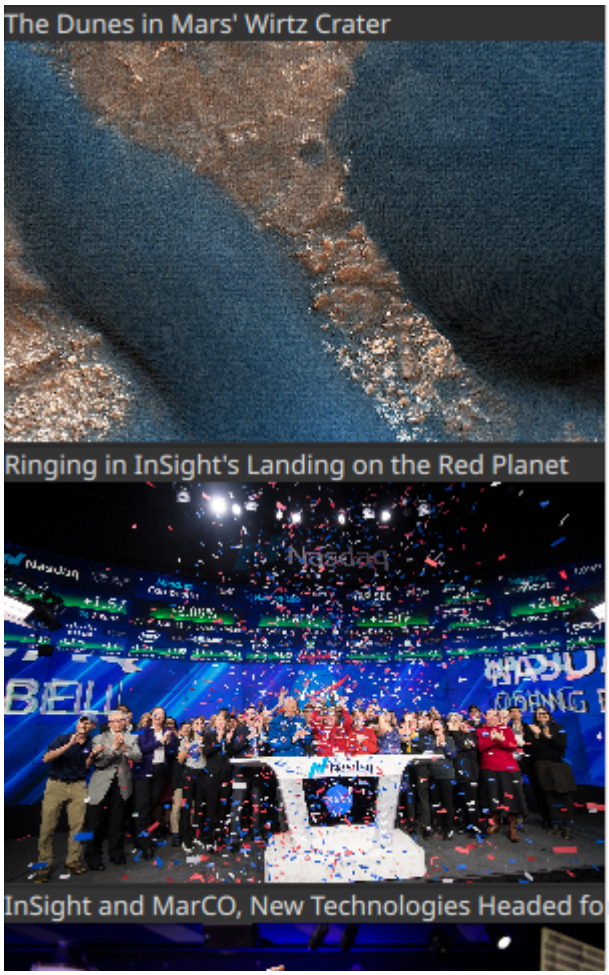
```
TableView {
    columnWidthProvider: function (column) { return 10 * (colum
    // ...
}
```

If you need to dynamically change the column widths or row heights you must notify the view of this by calling the `forceLayout` method. This will make the view re-calculate the size and position of all cells.

A Model from XML

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

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



When the data has been downloaded, it is processed into model items and roles. The `query` property of the `XmlListModel` 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 `XmlListModelRole` 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 `elementName` and (optional) `attributeName` properties for each role. For instance, the `title` property corresponds to the `title` XML element, returning the contents between the `<title>` and `</title>` tags.

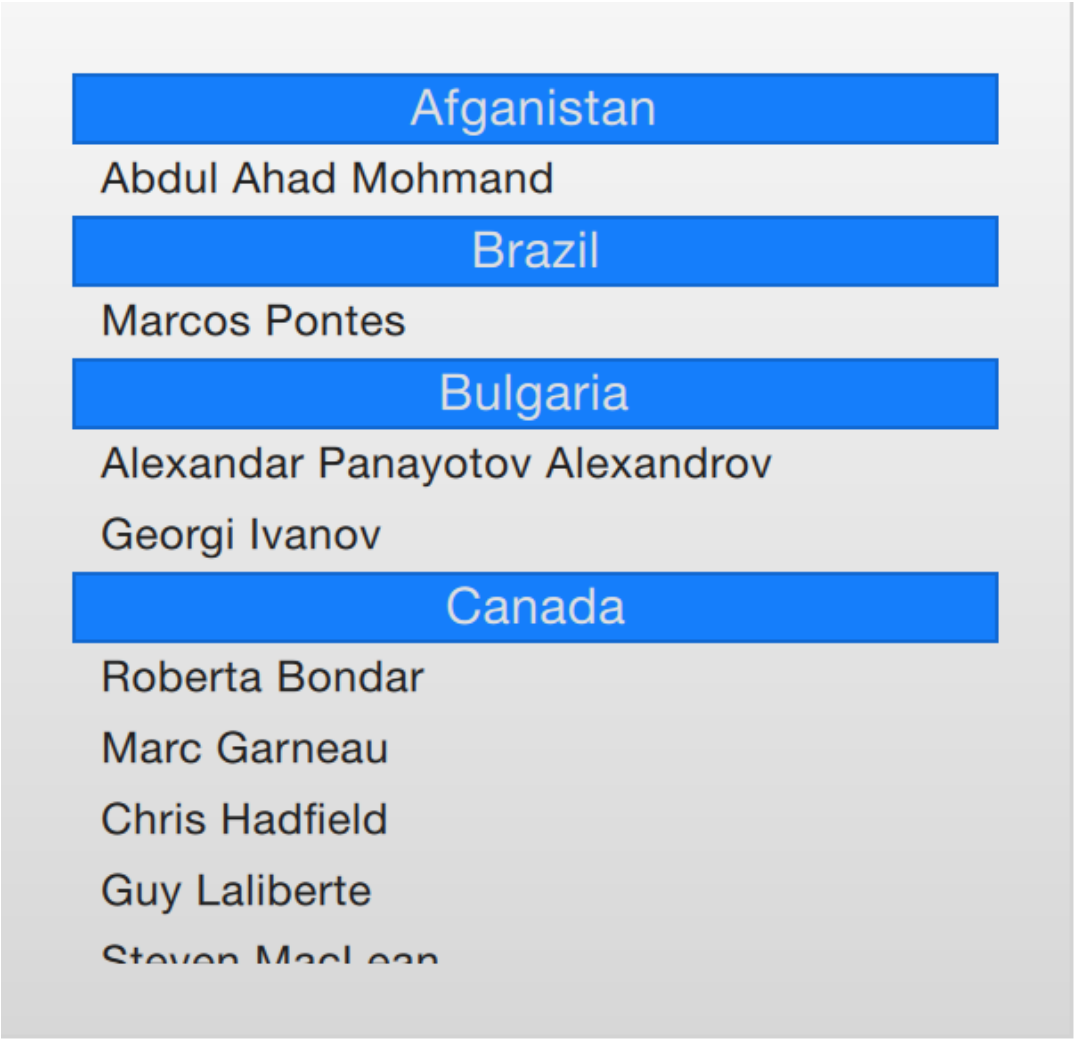
The `imageSource` property extracts the value of an attribute of a tag instead of the contents of the tag. In this case, the `url` attribute of the `enclosure` tag is extracted as a string. The `imageSource` property can then be used directly as the `source` for an `Image` element, which loads the image from the given URL.

```
import QtQuick
import QtQml.XmlListModel
import "../common"

Background {
    width: 300
    height: 480

    Component {
        id: imageDelegate

        Box {
            id: wrapper

            required property string title
            required property string imageSource

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

            Column {
                Text {
```

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

    XmlListModel {
        id: imageModel

        source: "https://www.nasa.gov/rss/dyn/image_of_the_day.
        query: "/rss/channel/item"

        XmlListModelRole { name: "title"; elementName: "title"
        XmlListModelRole { name: "imageSource"; elementName: "e
    }

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

Lists with Sections

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



In order to use sections, the `section.property` and `section.criteria` must be set up. 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 spacemen 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 spacemen are shown using the `spaceManDelegate` component.

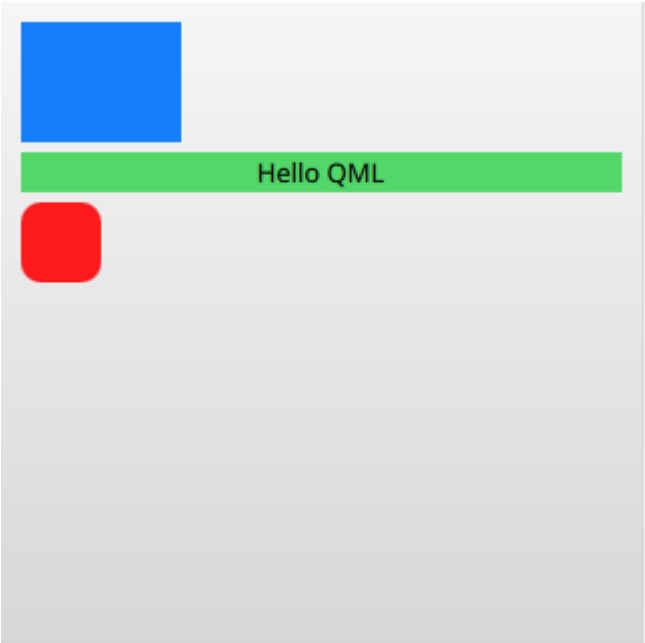
```
import QtQuick
import "../common"

Background {
    width: 300
    height: 290

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

        clip: true

        model: spaceMen

        delegate: spaceManDelegate
```

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

Component {
    id: spaceManDelegate

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

Component {
    id: sectionDelegate

    BlueBox {
        id: sectionWrapper
        required property string section
        width: ListView.view ? ListView.view.width : 0
        height: 20
        text: sectionWrapper.section
        fontColor: '#e0e0e0'
    }
}


ListModel {
    id: spaceMen
```

```
            ListElement { name: "Abdul Ahad Mohmand"; nation: "Afga
            ListElement { name: "Marcos Pontes"; nation: "Brazil";
            ListElement { name: "Alexandar Panayotov Alexandrov"; n
            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: "Canad
            ListElement { name: "Dafydd Williams"; nation: "Canada"
        }
    }
```

The ObjectModel

In some cases you might want to use a list view for a large set of different items. You can solve this using dynamic QML and `Loader`, but another options is to use an `ObjectModel` from the `QtQml.Models` module. The object model is different from other models as it lets you put the actual visual elements side the model. That way, the view does not need any `delegate`.



In the example below we put three `Rectangle` elements into the `ObjectModel`. However, one rectangle has a `Text` element child while the last one has rounded corners. This would have resulted in a table-style model using something like a `ListModel`. It would also have resulted in empty `Text` elements in the model.

```
import QtQuick
import QtQml.Models

Rectangle {
    width: 320
    height: 320

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

    ObjectModel {
        id: itemModel

        Rectangle { height: 60; width: 80; color: "#157efb" }
        Rectangle { height: 20; width: 300; color: "#53d769"
```

```
            Text { anchors.centerIn: parent; color: "black"; te
        }
        Rectangle { height: 40; width: 40; radius: 10; color: "
    }

    ListView {
        anchors.fill: parent
        anchors.margins: 10
        spacing: 5

        model: itemModel
    }
}
```

Another aspect of the `ObjectModel` is that is can be dynamically populated using the `get`, `insert`, `move`, `remove`, and `clear` methods. This way, the contents of the model can be dynamically generated from various sources and still easily shown in a single view.

Models with Actions

The `ListElement` type supports the binding of Javascript functions to properties. This means that you can put functions into a model. This is very useful when building menus with actions and similar constructs.

The example below demonstrates this by having a model of cities that greet you in different ways. The `actionModel` is a model of four cities, but the `hello` property is bound to functions. Each function takes an argument `value`, but you can have any number arguments.

In the delegate `actionDelegate`, the `MouseArea` calls the function `hello` as an ordinary function and this results a call to the corresponding `hello` property in the model.

```
import QtQuick

Rectangle {
    width: 120
    height: 300

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

    ListModel {
        id: actionModel

        ListElement {
            name: "Copenhagen"
            hello: function(value) { console.log(value + ": You
        }
        ListElement {
            name: "Helsinki"
            hello: function(value) { console.log(value + ": Hel
        }
        ListElement {
            name: "Oslo"
            hello: function(value) { console.log(value + ": Hei
        }
        ListElement {
            name: "Stockholm"
            hello: function(value) { console.log(value + ": Sto
        }
    }

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

        focus: true

        model: actionModel
```

```
        delegate: Rectangle {
            id: delegate

            required property int index
            required property string name
            required property var hello

            width: ListView.view.width
            height: 40

            color: "#157efb"

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

            MouseArea {
                anchors.fill: parent
                onClicked: delegate.hello(delegate.index)
            }
        }

        spacing: 5
        clip: true
    }
}
```

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 from 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 take too much time to initialize, it will become apparent to the user as soon as the view is scrolled too quickly.

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

Having more delegates sacrifices memory for a smoother experience and slightly more time to initialize each delegate. This does not solve the problem of complex delegates. Each time a delegate is instantiated, its contents are 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 latter 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 resides outside each delegate. This reduces the time spent compiling JavaScript each time a delegate is created.

> **TIP**
>
> Be aware that using a `Loader` to postpone initialization does just that - it postpones a performance issue. This means that the scrolling performance will be improved, but the actual contents will still take time to appear.

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 a 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 need to be provided by each data item, a `ListModel` populated with `ListElement` items is a better solution.

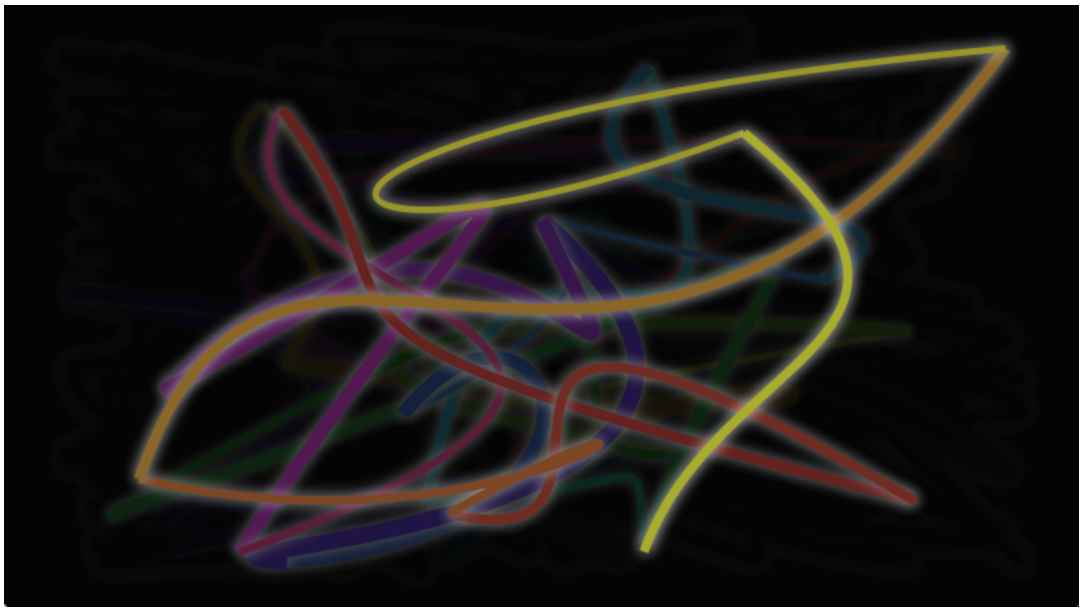
For static models, a `Repeater` can be used as the view. It is easy to combine it with a positioner such as `Row`, `Column`, `Grid` or `Flow` to build user interface parts. For dynamic or large data models, a view such as `ListView`, `GridView`, or `TableView` is more appropriate. These create delegate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

The difference between `GridView` and `TableView` is that the table view expects a table type model with multiple columns of data while the grid view shows a list type model in a grid.

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

Canvas Element

On of the strenghts of QML is its closeness to the Javascript ecosystem. This lets us reuse existing solutions from the web world and combine it with the native performance of QML visuals. However, sometimes we want to reuse graphics solutions from the web space too. That is where the `Canvas` element comes in handy. The canvas element provides an API very closely aligned to the drawing APIs for the identically named HTML element.

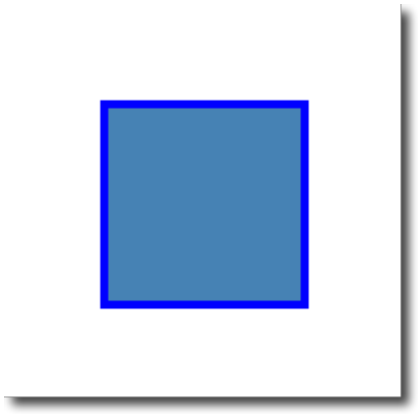


The fundamental idea of the canvas element is to render paths using a context 2D object. The context 2D object, contains the necessary graphics functions, whereas the canvas acts as the drawing canvas. The 2D context supports strokes, fills gradients, text and a different set of path creation commands.

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

```
import QtQuick

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

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



The stroke width is set to 4 and uses a blue color define by `strokeStyle`. The final shape is set up to be filled through the `fillStyle` to a “steel blue” 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()
}
```

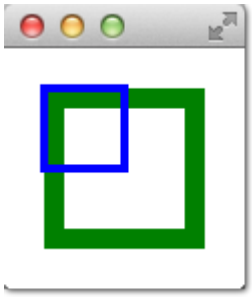
This produces a horizontal stroked line from point `P1(50,50)` to point `P2(150,50)` .

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

Convenience API

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

```
import QtQuick

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

        // draw a filles rectangle
        ctx.fillRect(20, 20, 80, 80)
        // cut our 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)
    }
}
```



TIP

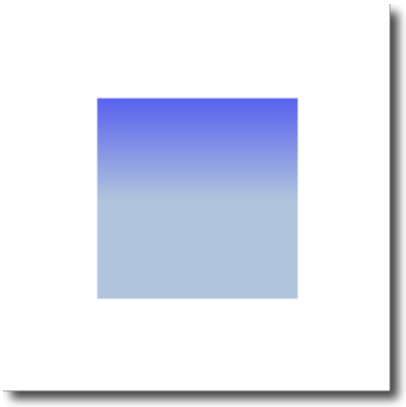
The stroke area extends half of the line width on both sides of the path. A 4 px lineWidth will draw 2 px outside the path and 2 px inside.

Gradients

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

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

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

The gradient in this example is defined along the starting point (100,0) to the end point (100,200), which gives a vertical line in the middle of our canvas. The gradient-stops can be defined as a color from 0.0 (gradient start point) to 1.0 (gradient endpoint). 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 as much larger than the rectangle we want to draw, so the rectangle clips gradient to it's defined the geometry.



TIP

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

Shadows

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

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

First, we draw the dark background:

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

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

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

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

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

Images

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

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

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

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

    }
```

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

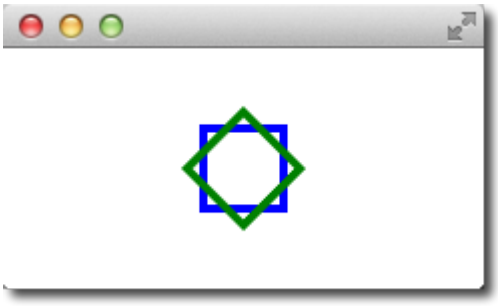
The left shows our ball image painted at the top-left position of 10x10. The right image shows the ball with a clipping path applied. Images and any other path can be clipped using another path. The clipping is applied by defining a path and calling the `clip()` function. All following drawing operations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

Transformation

The canvas allows you to transform the coordinate system in several ways. This is very similar to the transformation offered by QML items. You have the possibility to `scale` , `rotate` , `translate` the coordinate system. Indifference 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.

```
import QtQuick

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

            // translate x/y coordinate system
            ctx.translate(root.width/2, root.height/2);

            // draw path
            ctx.beginPath();
            ctx.rect(-20, -20, 40, 40);
            ctx.stroke();

            // rotate coordinate system
            ctx.rotate(Math.PI/4);
            ctx.strokeStyle = "green";
```

```
            // draw path
            ctx.beginPath();
            ctx.rect(-20, -20, 40, 40);
            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*Math.PI*) and to use a matrix transformation using the `setTransform(m11, m12, m21, m22, dx, dy)` .

> **TIP**
>
> To reset any transformation you can call the `resetTransform()` function to set the transformation matrix back to the identity matrix:
>
> ```
> ctx.resetTransform()
> ```
> js

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)` operations. For instance:

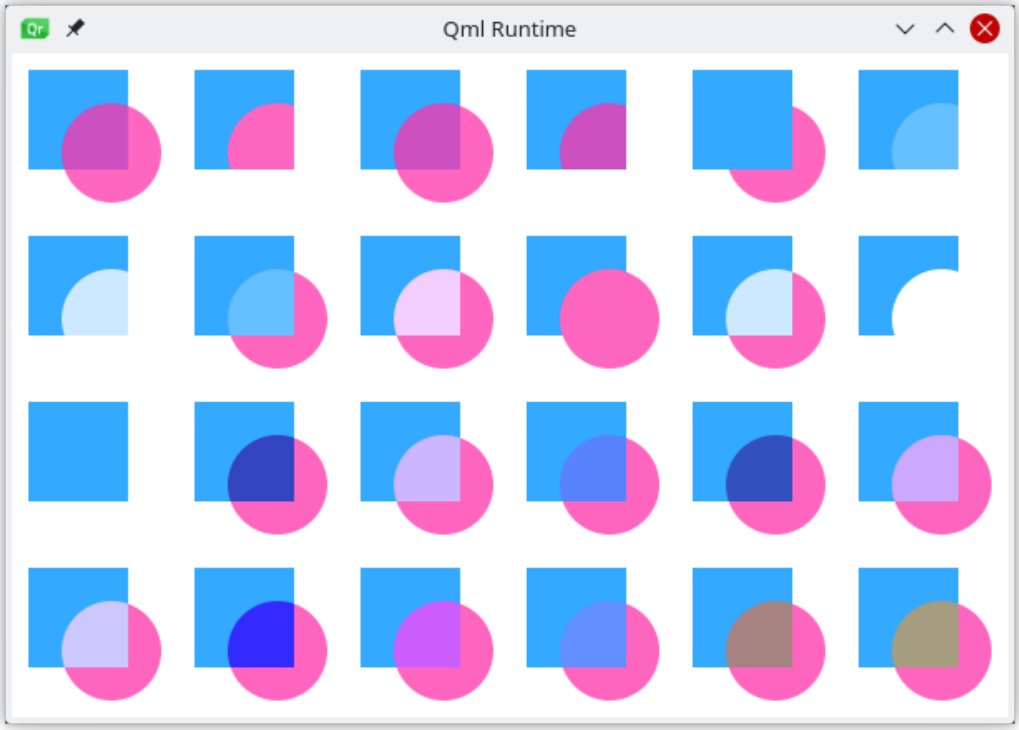
- `source-over`
- `source-in`
- `source-out`
- `source-atop`

Let's begin with a short example demonstrating the exclusive or composition:

```
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*
        ctx.closePath()
        ctx.fill()
    }
}
```

The example below will demonstrate all composition modes by iterating over them and combining a rectangle and a circle. You can find the resulting output below the source code.

```
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)
        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()
    }
}
```


Qml Runtime

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.

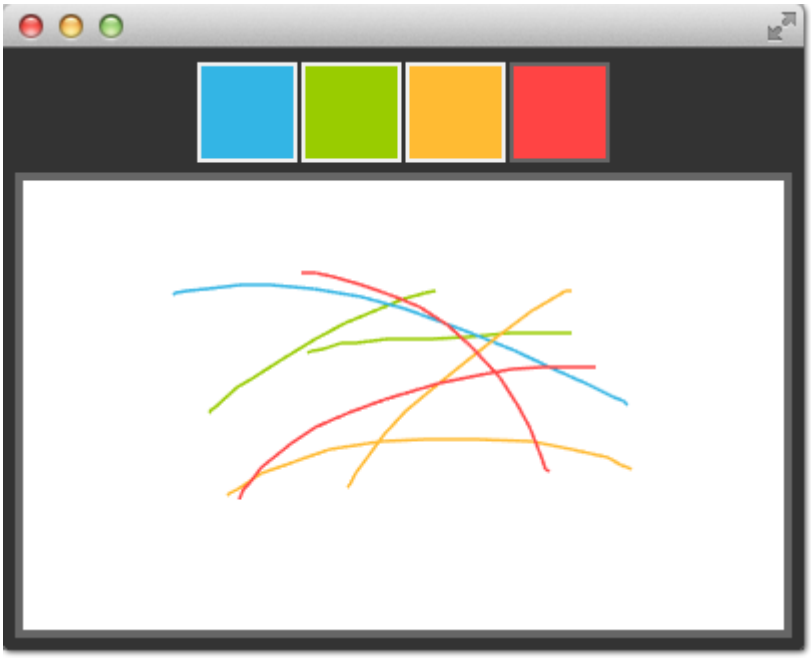
```
import QtQuick

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 on the left canvas. When the user clicks on the mouse area the canvas content is stored and an image URL is retrieved. On the right side of our example, the image is then displayed.

Canvas Paint

In this example, we will 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

    property color paintColor: "#33B5E5"

    anchors {
        horizontalCenter: parent.horizontalCenter
        top: parent.top
        topMargin: 8
```

```
        }
        spacing: 4
        Repeater {
            model: ["#33B5E5", "#99CC00", "#FFBB33", "#FF4444"]
            ColorSquare {
                required property var modelData
                color: modelData
                active: colorTools.paintColor == color
                onClicked: {
                    colorTools.paintColor = color
                }
            }
        }
    }
```

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

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

```
    Canvas {
        id: canvas

        property real lastX: 0
        property real lastY: 0
        property color color: colorTools.paintColor

        anchors {
            left: parent.left
            right: parent.right
            top: colorTools.bottom
            bottom: parent.bottom
            margins: 8
        }
```

```
        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 in calling the `onPaint` handler.

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

Porting from HTML5 Canvas

Porting from an HTML5 canvas to a QML canvas is fairly easy. In this chapter we will look at the example below and do the conversion.

- **https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Transformations**
- **http://en.wikipedia.org/wiki/Spirograph**

Spirograph

We use a **spirograph (http://en.wikipedia.org/wiki/Spirograph)** example from the Mozilla project as our foundation. The original HTML5 was posted as part of the **canvas tutorial (https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Transformations)** .

There were a few lines we needed to change:

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

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

- We adapted the draw method to receive the Context2D object

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

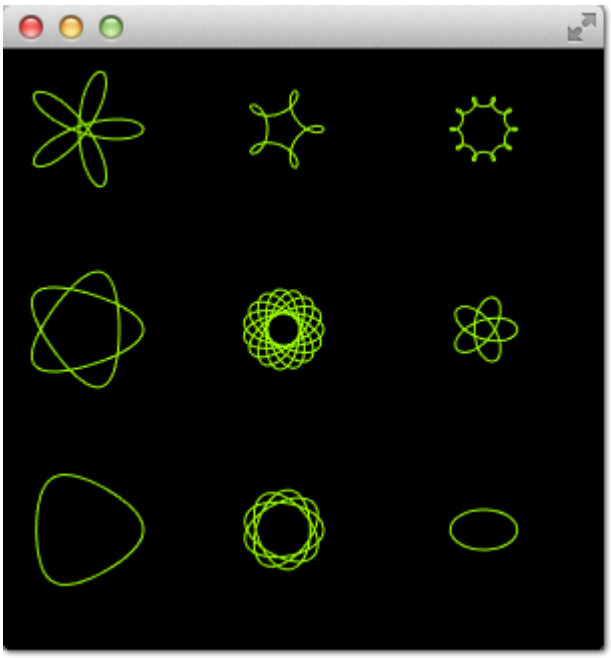
- We needed to adapt the translation for each spiro due to different sizes

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

Finally, we completed 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.



As you can see, with no changes to the actual logic, and relatively few changes to the code itself, a port from HTML5 to QML is possible.

Glowing Lines

Here is another more complicated port from the W3C organization. The original **pretty glowing lines (http://www.w3.org/TR/2dcontext/#examples)** has some pretty nice aspects, which makes the porting more challenging.

html

```
<!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].getCon

// 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.he
    context.scale(0.9, 0.9);
    context.translate(-context.canvas.width/2, -context.canvas.

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

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

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

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

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

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

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

// 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 need 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

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 to trigger we define for each operation a bool flag request an operation and trigger then a repaint request.

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

```
...
property bool requestLine: false

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

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

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

```
Canvas {
    ...
    property real hue: 0
```

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

Now our paint function should look like this:

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

The *line* function was extracted for a canvas as an 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);
```

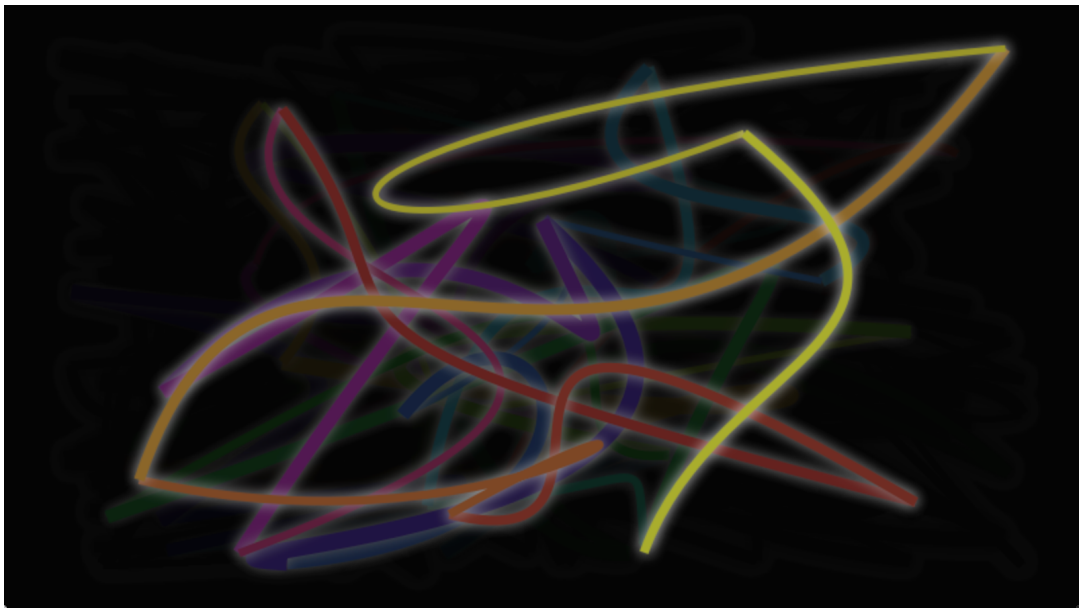
```
        hue += Math.random()*0.1
        if(hue > 1.0) {
            hue -= 1
        }
        context.strokeStyle = Qt.hsla(hue, 0.5, 0.5, 1.0);
        // context.shadowColor = 'white';
        // context.shadowBlur = 10;
        context.stroke();
        context.restore();
    }
```

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

Same applies to the *blank* function.

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

The final result will look similar to this.



Shapes

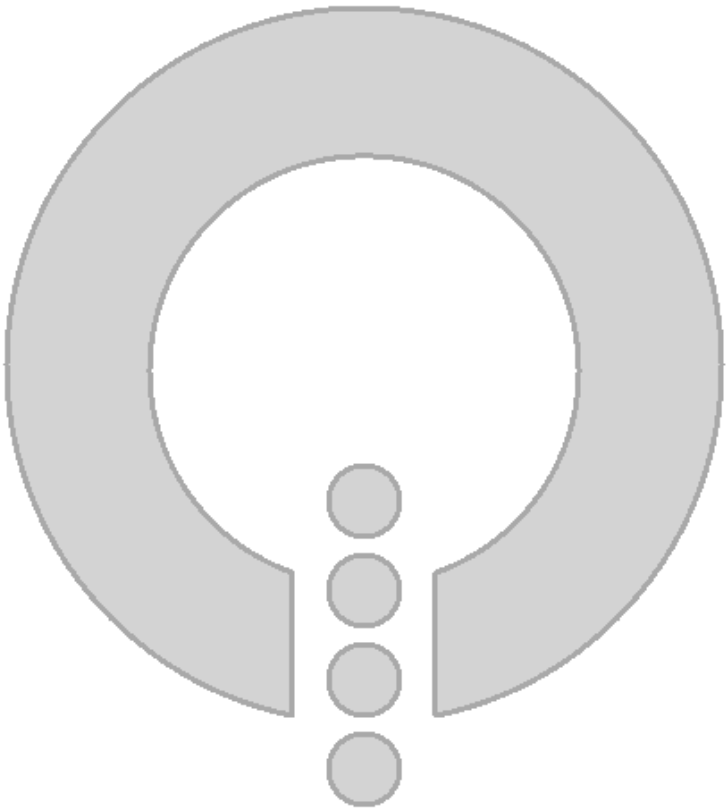
Until now we've used the `Rectangle` element and controls, but for free form shapes, we have to rely on images. Using the Qt Quick Shapes module it is possible to create truly free form shapes. This makes it possible to create visualizations directly from QML in a flexible manner.

In this chapter we will look at how to use shapes, the various path elements available, how shapes can be filled in different ways, and how to combine shapes with the power of QML to smoothly animate shapes.

A Basic Shape

The shape module lets you create arbitrarily paths and then stroke the outline and fill the interior. The definition of the path can be reused in other places where paths are used, e.g. for the `PathView` element used with models. But to paint a path, the `Shape` element is used, and the various path elements are put into a `ShapePath` .

In the example below, the path shown in the screenshot here is created. The entire figure, all five filled areas, are created from a single path which then is stroked and filled.



```
import QtQuick
import QtQuick.Shapes

Rectangle {
    id: root
    width: 600
    height: 600

    Shape {
        anchors.centerIn: parent

        ShapePath {

            strokeWidth: 3
```

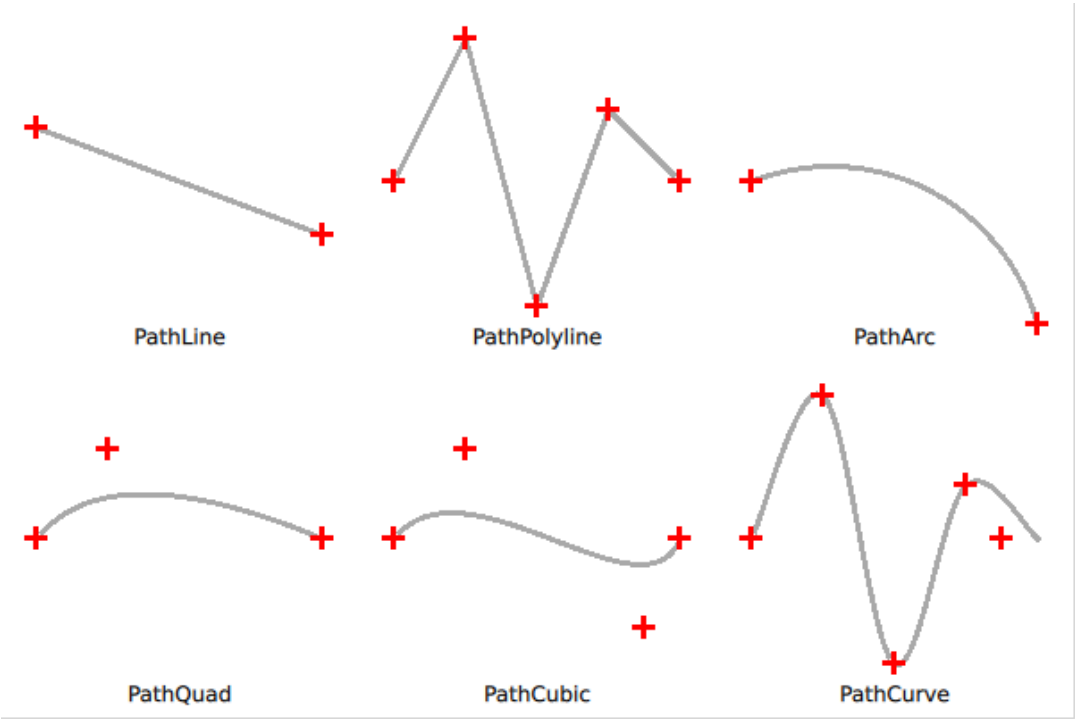
```
                strokeColor: "darkGray"
                fillColor: "lightGray"

                startX: -40; startY: 200

                // The circle

                PathArc { x: 40; y: 200; radiusX: 200; radiusY: 200
                PathLine { x: 40; y: 120 }
                PathArc { x: -40; y: 120; radiusX: 120; radiusY: 12
                PathLine { x: -40; y: 200 }

                // The dots

                PathMove { x: -20; y: 80 }
                PathArc { x: 20; y: 80; radiusX: 20; radiusY: 20; u
                PathArc { x: -20; y: 80; radiusX: 20; radiusY: 20;

                PathMove { x: -20; y: 130 }
                PathArc { x: 20; y: 130; radiusX: 20; radiusY: 20;
                PathArc { x: -20; y: 130; radiusX: 20; radiusY: 20;

                PathMove { x: -20; y: 180 }
                PathArc { x: 20; y: 180; radiusX: 20; radiusY: 20;
                PathArc { x: -20; y: 180; radiusX: 20; radiusY: 20;

                PathMove { x: -20; y: 230 }
                PathArc { x: 20; y: 230; radiusX: 20; radiusY: 20;
                PathArc { x: -20; y: 230; radiusX: 20; radiusY: 20;
            }
        }
    }
```

The path is made up of the children to the `ShapePath`, i.e. the `PathArc`, `PathLine`, and `PathMove` elements in the example above. In the next section, we will have a close look at the building blocks of paths.

Building Paths

As we saw in the last section, shapes are built from paths, which are built from path elements. The most common way to build a path is to close it, i.e. to ensure that it starts and ends in the same point. However, it is possible to create open paths, e.g. only for stroking. When filling an open path, the path is closed by a straight line, basically adding a `PathLine` that is used when filling the path, but not when stroking it.

As shown in the screenshot below, there are a few basic shapes that can be used to build your path. These are: lines, arcs, and various curves. It is also possible to move without drawing using a `PathMove` element. In addition to these elements, the `ShapePath` element also lets you specify a starting point using the `startX` and `startY` properties.



Lines are drawn using the `PathLine` element, as shown below. For creating multiple independent lines, the `PathMultiline` can be used.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        startX: 20; startY: 70

        PathLine {
            x: 180
            y: 130
        }
    }
}
```

When creating a polyline, i.e. a line consisting of several line segments, the `PathPolyline` element can be used. This saves some typing, as the end point of the last line is assumed to be the starting point of the next line.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        PathPolyline {
            path: [
                Qt.point(220, 100),
                Qt.point(260, 20),
                Qt.point(300, 170),
                Qt.point(340, 60),
                Qt.point(380, 100)
            ]
        }
```

```
    }
}
```

For creating arcs, i.e. segments of circles or ellipses, the `PathArc` and `PathAngleArc` elements are used. They provide you with the tools to create arcs, where the `PathArc` is used when you know the coordinates of the starting and ending points, while the `PathAngleArc` is useful when you want to control how many degrees the arc sweeps. Both elements produce the same output, so which one you use comes down to what aspects of the arc are the most important in your application.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        startX: 420; startY: 100

        PathArc {
            x: 580; y: 180
            radiusX: 120; radiusY: 120
        }
    }
}
```

After the lines and arcs follows the various curves. Here, Qt Quick Shapes provides three flavours. First, we have a look a the `PathQuad` which let's you create a quadratic Bezier curve based on the starting and end points (the starting point is implicit) and a single control point.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"
```

```
            startX: 20; startY: 300

            PathQuad {
                x: 180; y: 300
                controlX: 60; controlY: 250
            }
        }
    }
```

The `PathCubic` element creates a cubic Bezier curve from the starting and end points (the starting point is implicit) and two control points.

```
    Shape {
        ShapePath {
            strokeWidth: 3
            strokeColor: "darkgray"

            startX: 220; startY: 300

            PathCubic {
                x: 380; y: 300
                control1X: 260; control1Y: 250
                control2X: 360; control2Y: 350
            }
        }
    }
```

Finally, the `PathCurve` creates a curve passing through a list of provided control points. The curve is created by providing multiple `PathCurve` elements which each contain one control point. The Catmull-Rom spline is used to create a curve passing through the control points.

```
    Shape {
        ShapePath {
            strokeWidth: 3
            strokeColor: "darkgray"
```

```
            startX: 420; startY: 300

            PathCurve { x: 460; y: 220 }
            PathCurve { x: 500; y: 370 }
            PathCurve { x: 540; y: 270 }
            PathCurve { x: 580; y: 300 }
        }
    }
```

There is one more useful path element, the `PathSvg`. This element lets you stroke and fill an SVG path.

> **TIP**
>
> The `PathSvg` element cannot always be combined with other path elements. This depends on the painting backend used, so make sure to use the `PathSvg` element or the other elements for a single path. If you mix `PathSvg` with other path elements, your mileage will vary.

Filling Shapes

A shape can be filled in a number of different ways. In this section we will have a look at the general filling rule, and also the various ways a path can be filled.

Qt Quick Shapes provides two filling rules controlled using the `fillRule` property of the `ShapePath` element. The different results are shown in the screenshot below. It can be set to either `ShapePath.OddEvenFill`, which is the default. This fills each part of the path individually, meaning that you can create a shape with holes in it. The alternative rule is the `ShapePath.WindingFill`, which fills everything between the extreme endpoints on each horizontal line across the shape. Regardless of the filling rule, the shape outline is then drawn using a pen, so even when using the winding fill rule, the outline is drawn inside the shape.

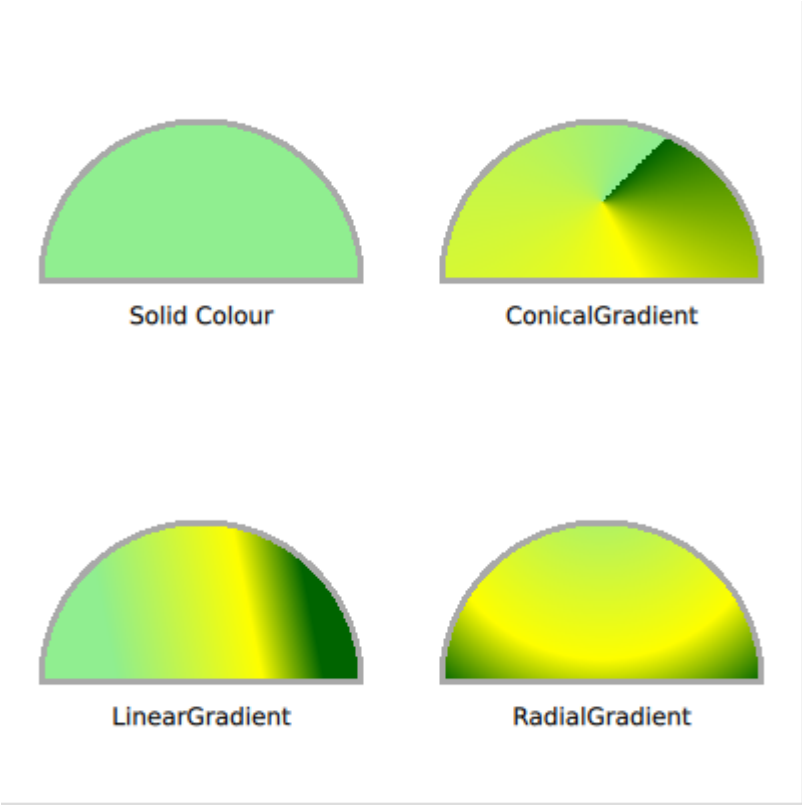
The examples below demonstrate how to use the two fill rules as shown in the screenshot above.

```
Shape {
    ShapePath {
```

```
        strokeWidth: 3
        strokeColor: "darkgray"
        fillColor: "orange"

        fillRule: ShapePath.OddEvenFill

        PathPolyline {
            path: [
                Qt.point(100,  20),
                Qt.point(150, 180),
                Qt.point( 20,  75),
                Qt.point(180,  75),
                Qt.point( 50, 180),
                Qt.point(100,  20),
            ]
        }
    }
}


Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"
        fillColor: "orange"

        fillRule: ShapePath.WindingFill

        PathPolyline {
            path: [
                Qt.point(300,  20),
                Qt.point(350, 180),
                Qt.point(220,  75),
                Qt.point(380,  75),
                Qt.point(250, 180),
                Qt.point(300,  20),
            ]
        }
```

```
        }
    }
```

Once the filling rule has been decided on, there are a number of ways to fill the outline. The various options are shown in the screenshot below. The various options are either a solid color, or one of the three gradients provided by Qt Quick.



To fill a shape using a solid color, the `fillColor` property of the `ShapePath` is used. Set it to a color name or code, and the shape is filled using it.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        fillColor: "lightgreen"
```

```
            startX: 20; startY: 140

            PathLine {
                x: 180
                y: 140
            }
            PathArc {
                x: 20
                y: 140
                radiusX: 80
                radiusY: 80
                direction: PathArc.Counterclockwise
                useLargeArc: true
            }
        }
    }
```

If you do not want to use a solid color, a gradient can be used. The gradient is applied using the `fillGradient` property of the `ShapePath` element.

The first gradient we look at is the `LinearGradient`. It creates a linear gradient between the start and end point. The end points can be positioned anyway you like, to create, for instance, a gradient at an angle. Between the end points, a range of `GradientStop` elements can be inserted. These are put at a `position` from `0.0`, being the `x1, y1` position, to `1.0`, being the `x2, y2` position. For each such stop, a color is specified. The gradient then creates soft transitions between the colors.

> **TIP**
>
> If the shape extends beyond the end points, the first or last color is either continued, or the gradient is repeated or mirrored. This behaviour is specified using the `spread` property of the `LinearGradient` element.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        fillGradient: LinearGradient {
            x1: 50; y1: 300
            x2: 150; y2: 280

            GradientStop { position: 0.0; color: "lightgreen" }
            GradientStop { position: 0.7; color: "yellow" }
            GradientStop { position: 1.0; color: "darkgreen" }
        }

        startX: 20; startY: 340

        PathLine {
            x: 180
            y: 340
        }
        PathArc {
            x: 20
            y: 340
            radiusX: 80
            radiusY: 80
            direction: PathArc.Counterclockwise
            useLargeArc: true
        }
    }
}
```

To create a gradient that spreads around an origin, a bit like a clock, the `ConicalGradient` is used. Here, the center point is specified using the `centerX` and `centerY` properties, and the starting angle is given using the `angle` property. The gradient stops are then spread from the given angle in a clockwise direction for 360 degrees.

```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        fillGradient: ConicalGradient {
            centerX: 300; centerY: 100
            angle: 45

            GradientStop { position: 0.0; color: "lightgreen" }
            GradientStop { position: 0.7; color: "yellow" }
            GradientStop { position: 1.0; color: "darkgreen" }
        }

        startX: 220; startY: 140

        PathLine {
            x: 380
            y: 140
        }
        PathArc {
            x: 220
            y: 140
            radiusX: 80
            radiusY: 80
            direction: PathArc.Counterclockwise
            useLargeArc: true
        }
    }
}
```

To instead create a gradient that forms circles, a bit like rings on the water, the `RadialGradient` is used. For it you specify two circles, the focal circle and the center. The gradient stops go from the focal circle to the center circle, and beyond those circles, the last color continues, is mirrored or repeats, depending on the `spread` property.

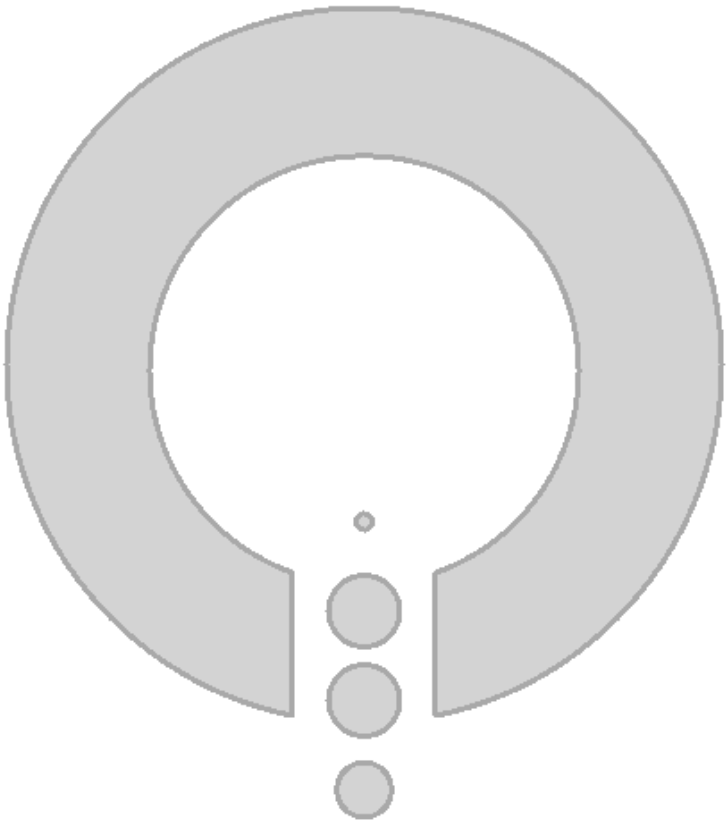
```
Shape {
    ShapePath {
        strokeWidth: 3
        strokeColor: "darkgray"

        fillGradient: RadialGradient {
            centerX: 300; centerY: 250; centerRadius: 120
            focalX: 300; focalY: 220; focalRadius: 10

            GradientStop { position: 0.0; color: "lightgreen" }
            GradientStop { position: 0.7; color: "yellow" }
            GradientStop { position: 1.0; color: "darkgreen" }
        }

        startX: 220; startY: 340

        PathLine {
            x: 380
            y: 340
        }
        PathArc {
            x: 220
            y: 340
            radiusX: 80
            radiusY: 80
            direction: PathArc.Counterclockwise
            useLargeArc: true
        }
    }
}
```

TIP

The advanced user can use a fragment shader to fill a shape. This way, you have full freedom to how the shape is filled. See the Effects chapter for more information on shaders.

Animating Shapes

One of the nice aspects of using Qt Quick Shapes, is that the paths drawn are defined directly in QML. This means that their properties can be bound, transitioned and animated, just like any other property in QML.



In the example below, we reuse the basic shape from the very first section of this chapter, but we introduce a variable, `t`, that we animate from

`0.0` to `1.0` in a loop. We then use this variable to offset the position of the small circles, as well as the size of the top and bottom circle. This creates an animation in which it seems that the circles appear at the top and disappear towards the bottom.

```
import QtQuick
import QtQuick.Shapes

Rectangle {
    id: root
    width: 600
    height: 600

    Shape {
        anchors.centerIn: parent

        ShapePath {
            id: shapepath

            property real t: 0.0

            NumberAnimation on t { from: 0.0; to: 1.0; duration

            strokeWidth: 3
            strokeColor: "darkGray"
            fillColor: "lightGray"

            startX: -40; startY: 200

            // The circle

            PathArc { x: 40; y: 200; radiusX: 200; radiusY: 200
            PathLine { x: 40; y: 120 }
            PathArc { x: -40; y: 120; radiusX: 120; radiusY: 12
            PathLine { x: -40; y: 200 }

            // The dots
```

```
                PathMove { x: -20+(1.0-shapepath.t)*20; y: 80 + sha
                PathArc { x: 20-(1.0-shapepath.t)*20; y: 80 + shape
                PathArc { x: -20+(1.0-shapepath.t)*20; y: 80 + shap

                PathMove { x: -20; y: 130 + shapepath.t*50 }
                PathArc { x: 20; y: 130 + shapepath.t*50; radiusX:
                PathArc { x: -20; y: 130 + shapepath.t*50; radiusX:

                PathMove { x: -20; y: 180 + shapepath.t*50 }
                PathArc { x: 20; y: 180 + shapepath.t*50; radiusX:
                PathArc { x: -20; y: 180 + shapepath.t*50; radiusX:

                PathMove { x: -20+shapepath.t*20; y: 230 + shapepat
                PathArc { x: 20-shapepath.t*20; y: 230 + shapepath.
                PathArc { x: -20+shapepath.t*20; y: 230 + shapepath
            }
        }
    }
```

Notice that instead of using a `NumberAnimation` on `t`, any other binding can be used, e.g. to a slider, an external state, and so on. Your imagination is the limit.

Summary

In this chapter we've a look at what the Qt Quick Shapes module has to offer. Using it we can create arbitrary shapes directly in QML, and leverage the property binding system of QML to create dynamic shapes. We've also had a look at the various path segments that can be used to build shapes from elements such as lines, arcs, and various curves. Finally, we've explored the filling options, where gradients can be used to create exciting visual effects from a path.

Effects in QML

In this chapter, we will look at the tools for various effects in QML. The focus will be on:

- Particle Effects
- Shader Effects

Particle Effects

Particle effects lets us create groups of particles, i.e. instances of a given element. These are generated in a stochastic way and let us work with groups of items rather than individual items. This can be used to create things such as falling leafs, explosions, fire, clouds, and starfields.

Shader Effects

Shader effects are applied in the graphics rendering pipeline and allows us to change both the size and colour of any visible QML element. This can be used to create transitions such as the genie effect, waves and curtains, or filters such as blur, grayscale, and blending.

Shaders are written in a shader language which is then baked and imported into the QML scene, much as other resources. These shaders can then be applied to images or other elements to create advanced visual effects.

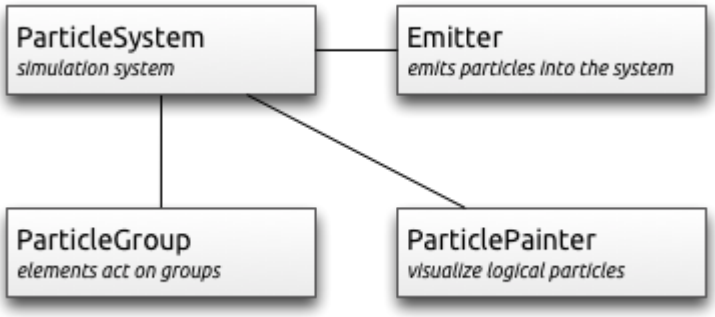
TIP

Working with shader effects is an advanced topic.

Particle Concept

In the heart of the particle simulation is the `ParticleSystem` which controls the shared timeline. 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. Particle ones emitted can't be manipulated by the emitter anymore. The particle module provides the `Affector`, which allows manipulating 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

Simple Simulation

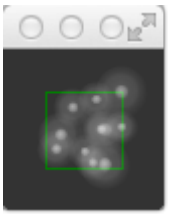
Let us have a look at a very simple simulation to get started. Qt Quick makes it actually very simple to get started with particle rendering. For this we need:

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

```
import QtQuick
import QtQuick.Particles

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 an 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 them 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 lifespan of 1000msec (`lifeSpan : 1000`) and a lifespan 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 its 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 how to change the direction particles.

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

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

Changing these properties can influence the result in a drastical 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 lifespan to 2 seconds the size now starts at 64 pixels and decreases 32 pixels at the end of a particle lifespan.

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

Particle Parameters

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

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

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

The particle shall be 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:

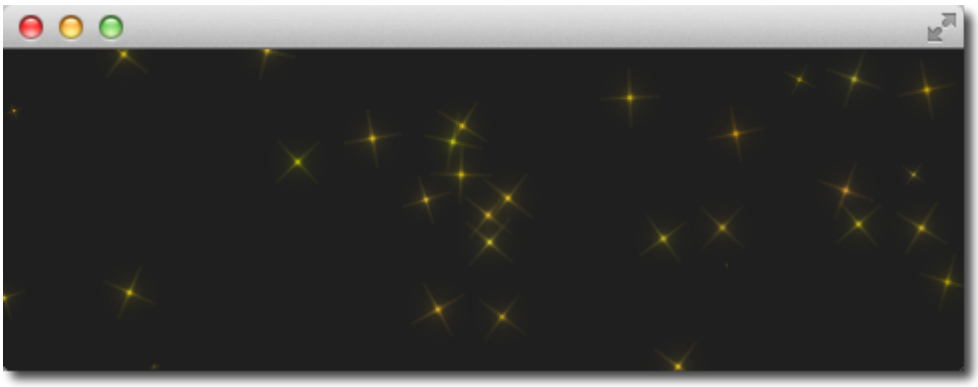
```
rotation: 15
rotationVariation: 5
```

```
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.

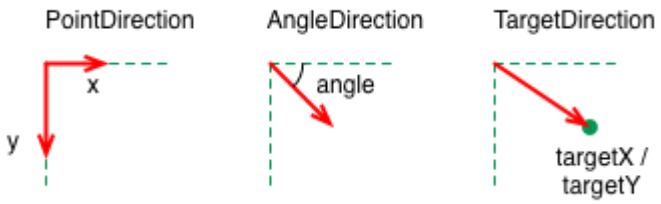
```
ImageParticle {
    source: "assets/star.png"
    system: particleSystem
    color: '#FFD700'
    colorVariation: 0.2
    rotation: 0
    rotationVariation: 45
    rotationVelocity: 15
    rotationVelocityVariation: 15
    entryEffect: ImageParticle.Scale
}
```

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 { }
```

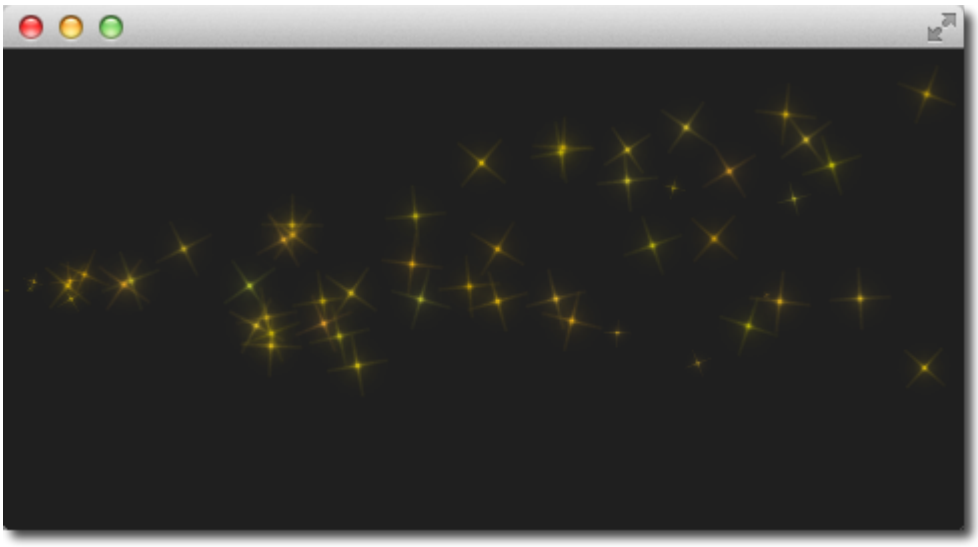
The angle where the particles are emitted is specified using the angle property. The angle is provided as a value between 0..360 degree and 0 points to the right. For our example, we would like the particles to move

to the right so 0 is already the right direction. The particles shall spread by +/- 5 degrees:

```
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 lifetime 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 lifetime 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
    }
}
```

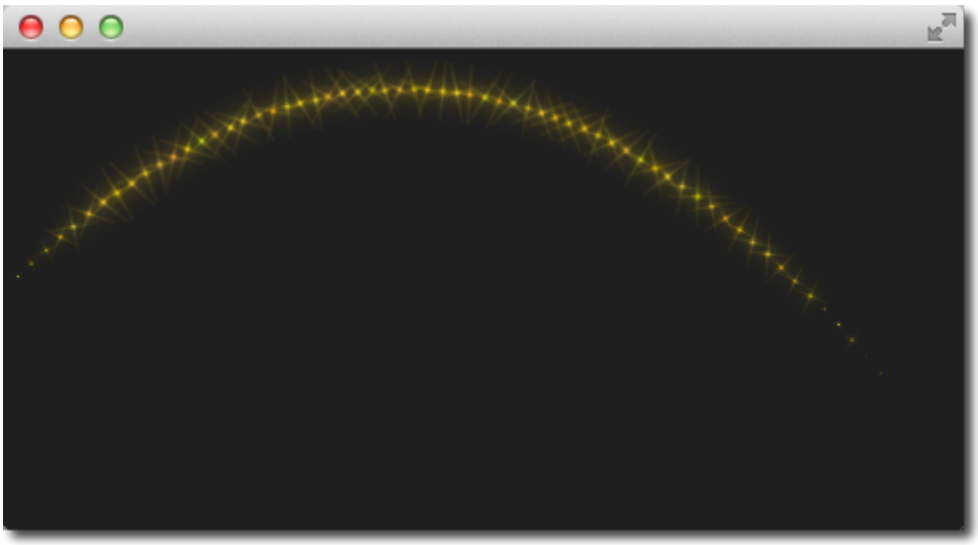
So what is then the acceleration doing? The acceleration adds an acceleration vector to each particle, which changes the velocity vector over time. For example, let's make a trajectory like an arc of stars. For this we change our velocity direction to -45 degree and remove the variations, to better visualize a coherent arc:

```
velocity: AngleDirection {
    angle: -45
    magnitude: 100
}
```

The acceleration direction shall be 90 degrees (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 trial-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 its vector space from an x and y component. For example, if you want the particles to travel in a 45-degree vector, you need to specify the same value for x and y.

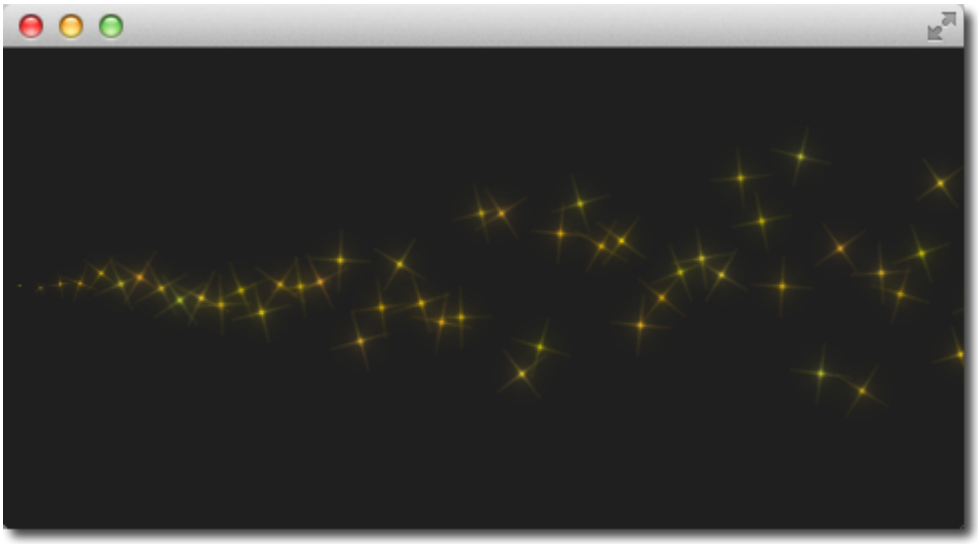
In our case we want the particles to travel from left-to-right building a 15-degree cone. For this we specify a `PointDirection` as our velocity vector space:

```
velocity: PointDirection { }
```

To achieve a traveling velocity of 100 px per seconds we set our x component to 100. For the 15 degrees (which is 1/6th of 90 degrees) we specify any 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.



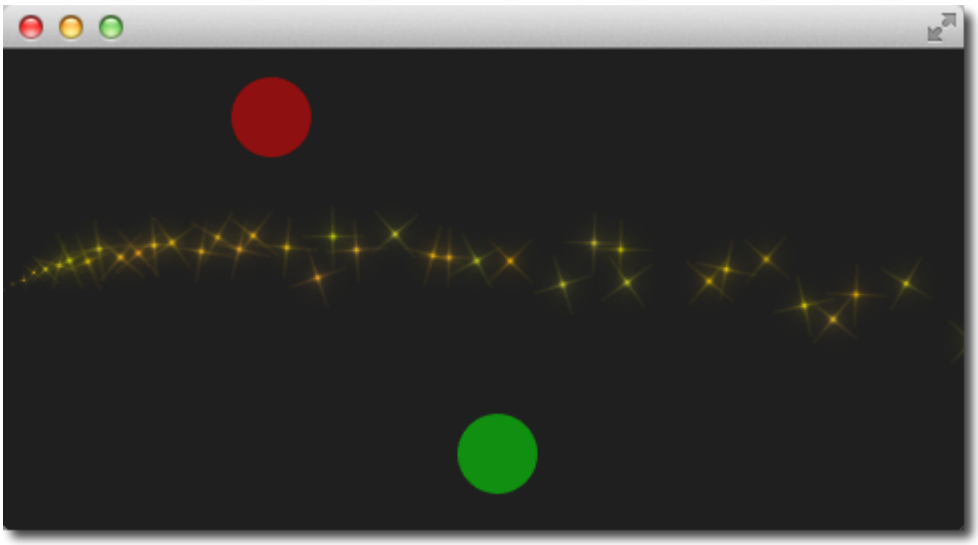
Now coming to our last contender, the `TargetDirection`. The target direction allows us to specify a target point as an x and y coordinate relative to the emitter or an item. When an item has specified the center of the item will become the target point. You can achieve the 15-degree cone by specifying a target variation of 1/6 th of the x target:

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

> **TIP**
>
> 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?



Affecting Particles

Particles are emitted by the emitter. After a particle was emitted it can't be changed any more 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

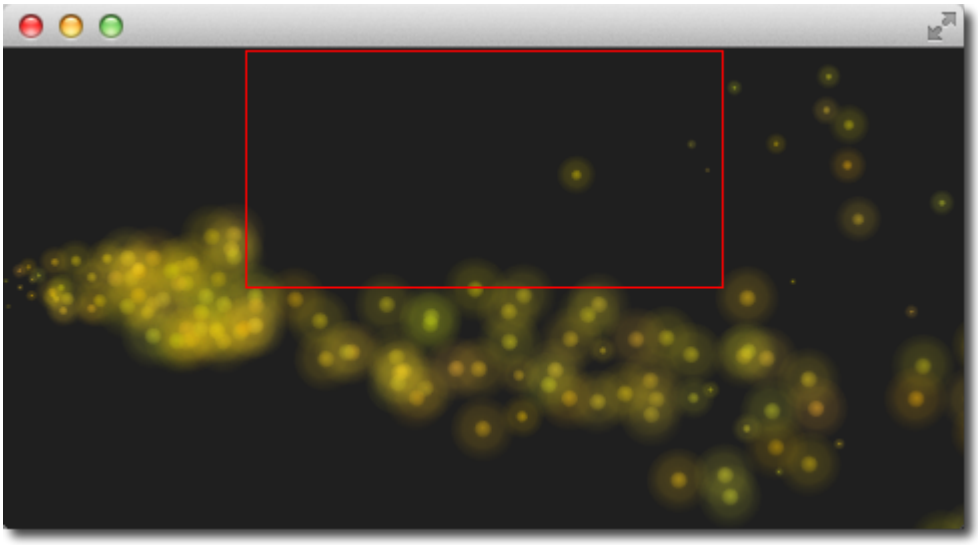
Age

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 of affector to 1200 msec. 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 live.



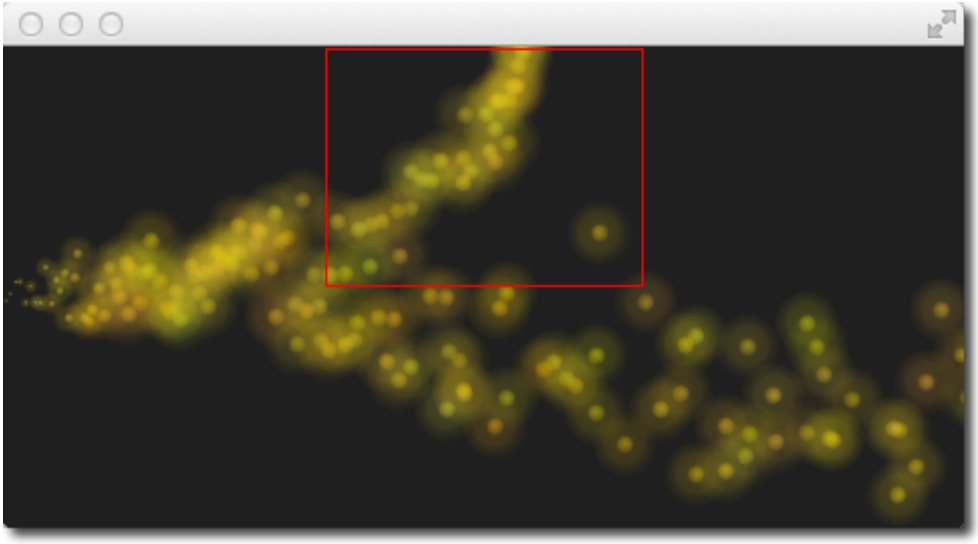
Attractor

The attractor attracts particles towards a specific point. The point is specified using `pointX` and `pointY`, which is relative to the attractor geometry. The strength specifies the force of attraction. In our example we let particles travel from left to right. The attractor is placed on the top and half of the particles travel through the attractor. Affector only affect particles while they are in their bounding box. This split allows us to see the normal stream and the affected stream simultaneous.

```
    Attractor {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 160; height: 120
        system: particleSystem
```

```
    pointX: 0
    pointY: 0
    strength: 1.0
    Tracer {}
}
```

It's easy to see that the upper half of the particles are affected by the attracted to the top. The attraction point is set to top-left (0/0 point) of the attractor with a force of 1.0.

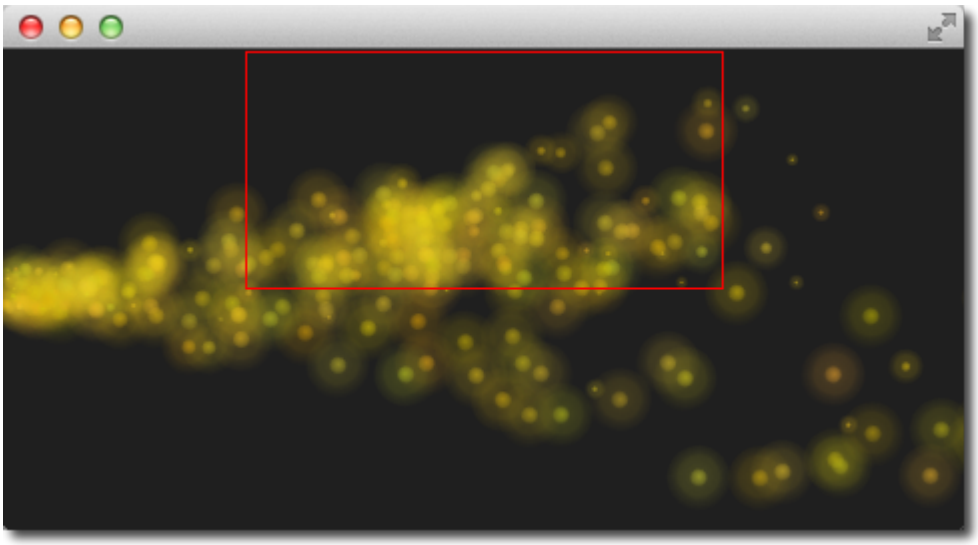


Friction

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 reaches 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.

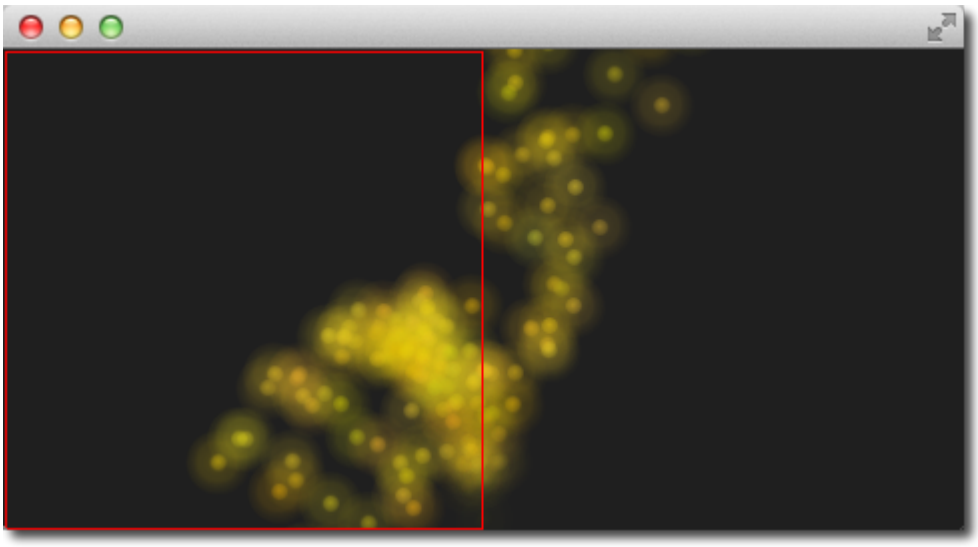


Gravity

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 effect is applied. The gravity is angled to 90 degrees (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.

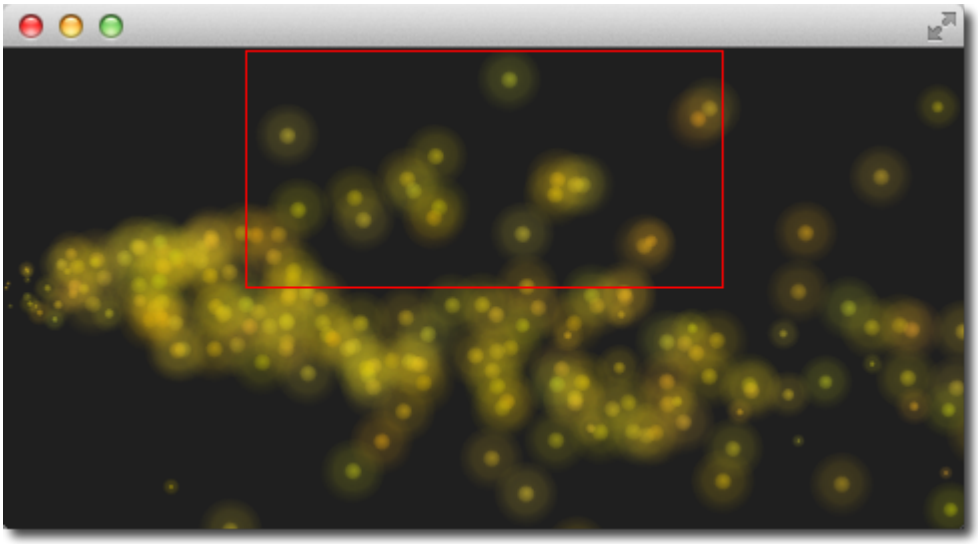


Turbulence

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 defined with the *noiseSource* property. The strength defines how strong the vector will be applied to 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.

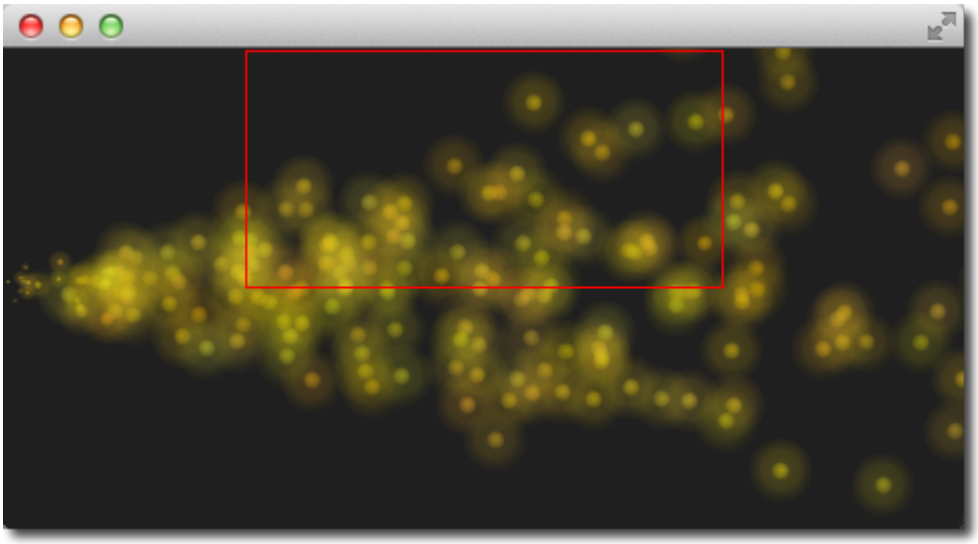


Wander

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 specify the influence on x and y component of the particle trajectory.

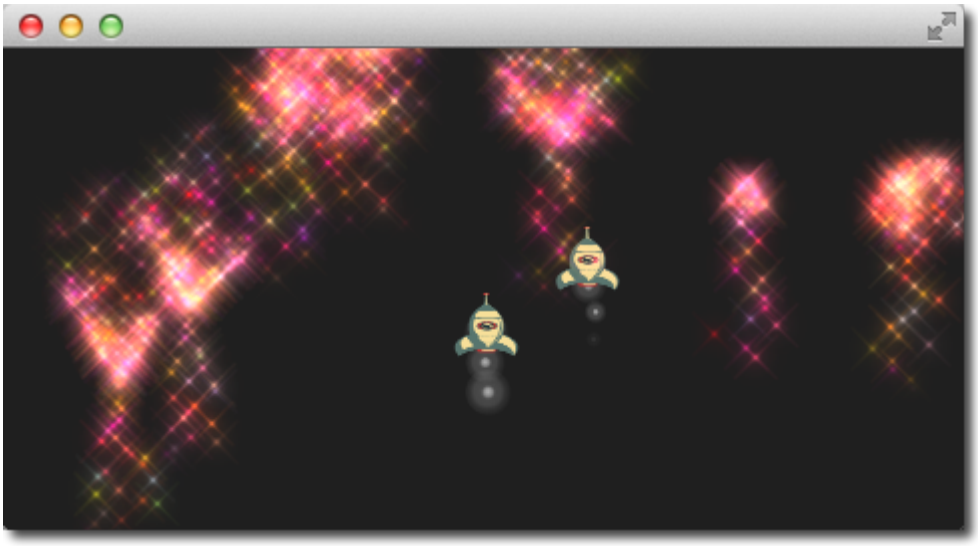
```
Wander {
    anchors.horizontalCenter: parent.horizontalCenter
    width: 240; height: 120
    system: particleSystem
    affectedParameter: Wander.Position
    pace: 200
    yVariance: 240
    Tracer {}
}
```

In the top wander affector particles are shuffled around by random trajectory changes. In this case, the position is changed 200 times per second in the y-direction.



Particle Groups

At the beginning of this chapter, we stated particles are in groups, which is by default the empty group (''). Using the `GroupGoal` affector is it possible to let the particle change groups. To visualize this we would like to create a small firework, where rockets start into space and explode in the air into a spectacular firework.



The example is divided into 2 parts. The 1st part called "Launch Time" is concerned to set up the scene and introduce particle groups and the 2nd part called "Let there be fireworks" focuses on the group changes.

Let's get started!

Launch Time

To get it going we create a typical dark scene:

```
import QtQuick 2.5
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false
}
```

The tracer property will be used to switch the tracer scene wide on and off. The next thing is to declare our particle system:

```
ParticleSystem {
    id: particleSystem
}
```

And our two image particles (one for the rocket and one for the exhaust smoke):

```
ImageParticle {
    id: smokePainter
    system: particleSystem
    groups: ['smoke']
    source: "assets/particle.png"
    alpha: 0.3
    entryEffect: ImageParticle.None
}

ImageParticle {
    id: rocketPainter
    system: particleSystem
    groups: ['rocket']
    source: "assets/rocket.png"
```

```
        entryEffect: ImageParticle.None
    }
```

You can see in on the images, they use the *groups* property to declare to which group the particle belongs. 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 in 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; magn
        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; angle
    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 are around the particle followed from where the smoke particles shall be emitted. If this is not specified then they are 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
}
```

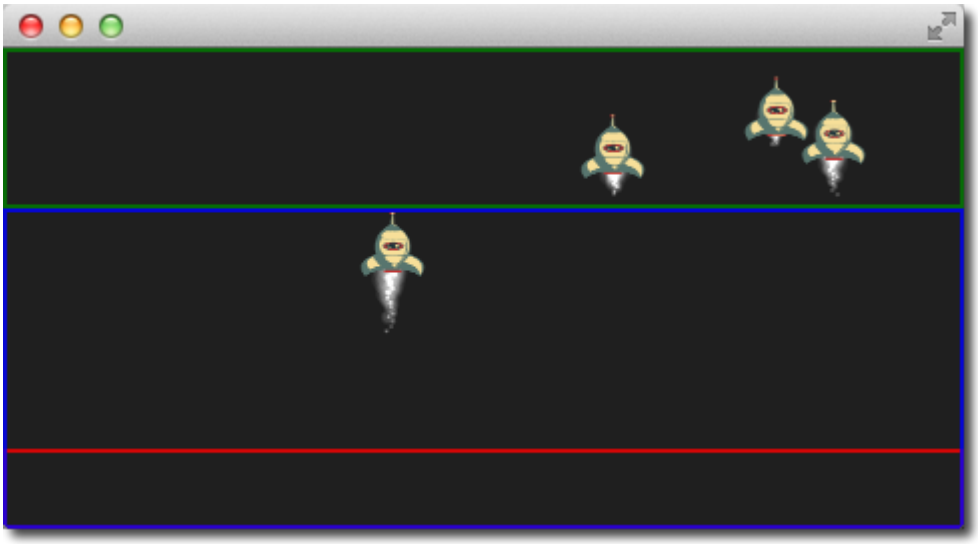
In the friction affector, you also need to declare which groups of particles it shall affect. The friction will slow all rockets, which are 80 pixels downwards from the top of the screen down by a factor of 0.9 (try 100 and you will see they almost stop immediately) until they reach a velocity of 5 pixels 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 needs to declare which groups it shall affect. The turbulence itself reaches from the bottom 160 pixels 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 the ground by the still applied downwards acceleration. The next thing would be to start the firework.



> **TIP**
>
> 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.

> **TIP**
>
> In the Qt 5 alpha release we could the *duration* for the group change not get working. Any ideas?

As the group of the rocket now changes to our ‘explosion’ particle group when the rocket particle enters the group goal area we need to add the firework inside the particle group:

```
// inside particle group
TrailEmitter {
    id: explosionEmitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 750
    emitRatePerParticle: 200
    size: 32
```

```
        velocity: AngleDirection { angle: -90; angleVariation: 180;
    }
```

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 degrees.

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 an almost transparent color to allow some shine effects.

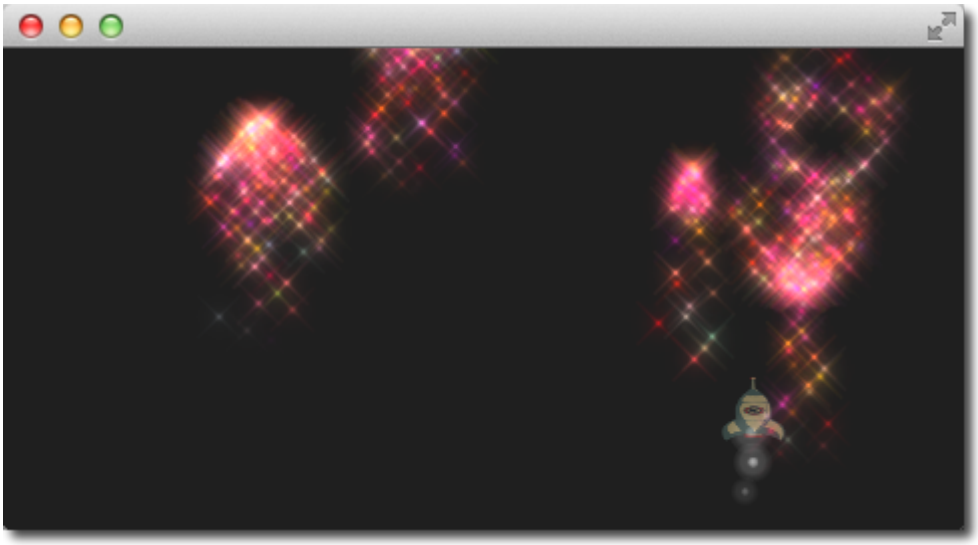
To make the firework more spectacular we also add a second trail emitter to our particle group, which will emit particles in a narrow cone downwards:

```
    // 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; m
    }
```

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
import QtQuick.Particles

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false

    ParticleSystem {
        id: particleSystem
    }
```

```
ImageParticle {
    id: smokePainter
    system: particleSystem
    groups: ['smoke']
    source: "assets/particle.png"
    alpha: 0.3
}

ImageParticle {
    id: rocketPainter
    system: particleSystem
    groups: ['rocket']
    source: "assets/rocket.png"
    entryEffect: ImageParticle.Fade
}

Emitter {
    id: rocketEmitter
    anchors.bottom: parent.bottom
    width: parent.width; height: 40
    system: particleSystem
    group: 'rocket'
    emitRate: 2
    maximumEmitted: 8
    lifeSpan: 4800
    lifeSpanVariation: 400
    size: 128
    velocity: AngleDirection { angle: 270; magnitude: 150;
    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; a
    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; angleVariati
            }

            TrailEmitter {
                id: explosion2Emitter
                anchors.fill: parent
                group: 'sparkle'
                follow: 'rocket'
                lifeSpan: 250
                emitRatePerParticle: 100
                size: 32
                velocity: AngleDirection { angle: 90; angleVariatio
            }
        }
    }
```

Particle Painters

Until now we have only used the image based particle painter to visualize particles. Qt comes also with other particle painters:

- `ItemParticle` : delegate based particle painter
- `CustomParticle` : shader based particle painter

The ItemParticle can be used to emit QML items as particles. For this, you need to specify your own delegate to the particle.

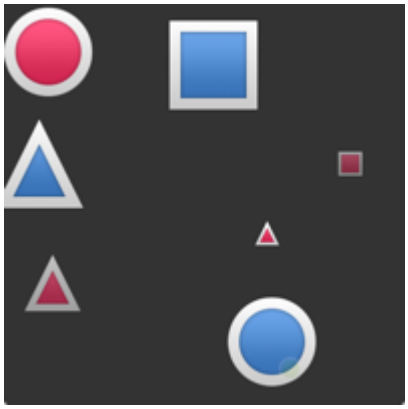
```
ItemParticle {
    id: particle
    system: particleSystem
    delegate: itemDelegate
}
```

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

```
Component {
    id: itemDelegate

    Item {
        id: container
        width: 32 * Math.ceil(Math.random() * 3)
        height: width
        Image {
            anchors.fill: parent
            anchors.margins: 4
            source: 'assets/' + root.images[Math.floor(Math.ran
```

```
            }
        }
    }
```

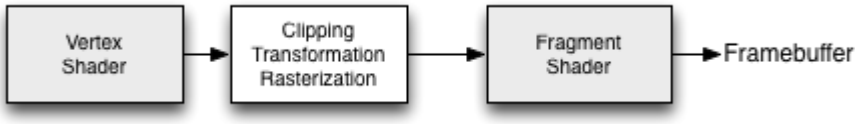
We emit 4 images per second with a lifespan of 4 seconds each. The particles fade automatically in and out.



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)`.

Graphics Shaders

Graphics is rendered using a *rendering pipeline* split into stages. There are multiple APIs to control graphics rendering. Qt supports OpenGL, Metal, Vulcan, and Direct3D. Looking at a simplified OpenGL pipeline, we can spot a vertex and fragment shader. These concepts exist for all other rendering pipelines too.



In the pipeline, the vertex shader receives vertex data, i.e. the location of the corners of each element that makes up the scene, and calculates a `gl_Position` . This means that the vertex shader can *move* graphical elements. In the next stage, the vertexes are clipped, transformed and rasterized for pixel output. Then the pixels, also known as *fragments*,are passed through the fragment shader, which calculates the color of each pixel. The resulting color returned through the `gl_FragColor` variable.

To summarize: the vertex shader is called for each corner point of your polygon (vertex = point in 3D) and is responsible for any 3D manipulation of these points. The fragment (fragment = pixel) shader is called for each pixel and determines the color of that pixel.

As Qt is independent of the underlying rendering API, Qt relies on a standard language for writing shaders. The Qt Shader Tools rely on a *Vulcan-compatible GLSL*. We will look more at this in the examples in this chapter.

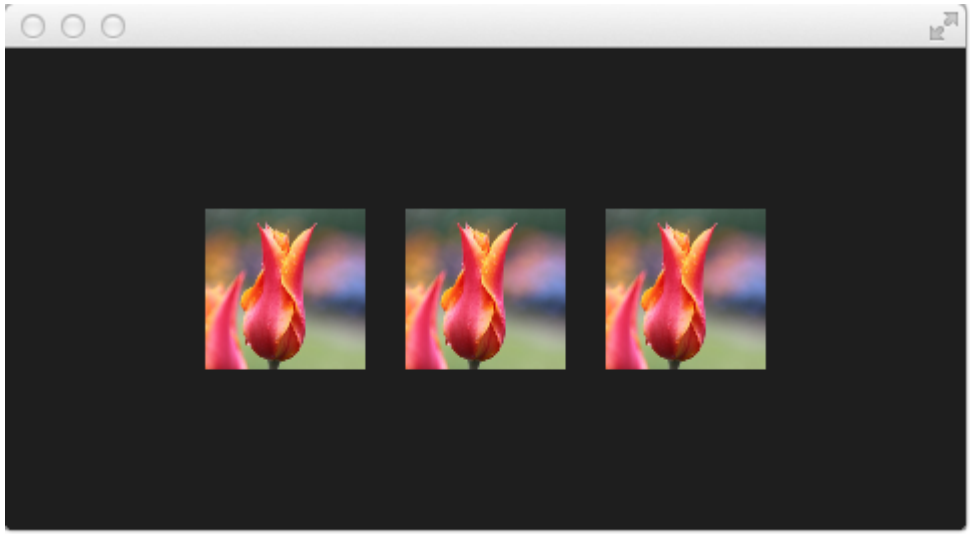
Shader Elements

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 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. Below, we first see the QML file with two `ShaderEffect` elements. One without any shaders specified, and one explicitly specifying default vertex and fragment shaders. We will look at the shaders shortly.

```
import QtQuick

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
            property variant source: sourceImage
            vertexShader: "default.vert.qsb"
            fragmentShader: "default.frag.qsb"
        }
    }
}
```



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 shader code for the fragment and vertex as shown below. Let's have a look at the shaders.

The vertex shader takes the texture coordinate, `qt_MultiTexCoord0`, and propagates it to the `qt_TexCoord0` variable. It also takes the `qt_Vertex` position and multiplies it with Qt's transformation matrix, `ubuf.qt_Matrix`, and returns it through the `gl_Position` variable. This leaves the texture and vertex position on the screen unmodified.

```
#version 440
```

```
layout(location=0) in vec4 qt_Vertex;
layout(location=1) in vec2 qt_MultiTexCoord0;

layout(location=0) out vec2 qt_TexCoord0;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;
} ubuf;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;
    gl_Position = ubuf.qt_Matrix * qt_Vertex;
}
```

The fragment shader takes the texture from the `source` 2D sampler, i.e. the texture, at the coordinate `qt_TexCoord0` and multiplies it with the Qt opacity, `ubuf.qt_Opacity` to calculate the `fragColor` which is the color to be used for the pixel.

```
#version 440

layout(location=0) in vec2 qt_TexCoord0;

layout(location=0) out vec4 fragColor;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;
} ubuf;

layout(binding=1) uniform sampler2D source;

void main() {
```

```
    fragColor = texture(source, qt_TexCoord0) * ubuf.qt_Opacity
}
```

Notice that these two shaders can serve as the boilerplate code for your own shaders. The variables, locations and bindings, are what Qt expects. You can read more about the exact details of this on the **Shader Effect Documentation (https://doc-snapshots.qt.io/qt6-6.2/qml-qtquick-shadereffect.html#details)** .

Before we can use the shaders, they need to be baked. If the shaders are a part of a larger Qt project and included as resources, this can be automated. However, when working with the shaders and a `qml` -file, we need to explicitly bake them by hand. This is done using the following two commands:

```
qsb --glsl 100es,120,150 --hlsl 50 --msl 12    -o default.frag.
qsb --glsl 100es,120,150 --hlsl 50 --msl 12 -b -o default.vert.
```

The `qsb` tool is located in the `bin` directory of your Qt 6 installation.

> **TIP**
>
> 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.

In the next examples, we will be playing around with some simple shader mechanics. First, we concentrate on the fragment shader and then we will come back to the vertex shader.

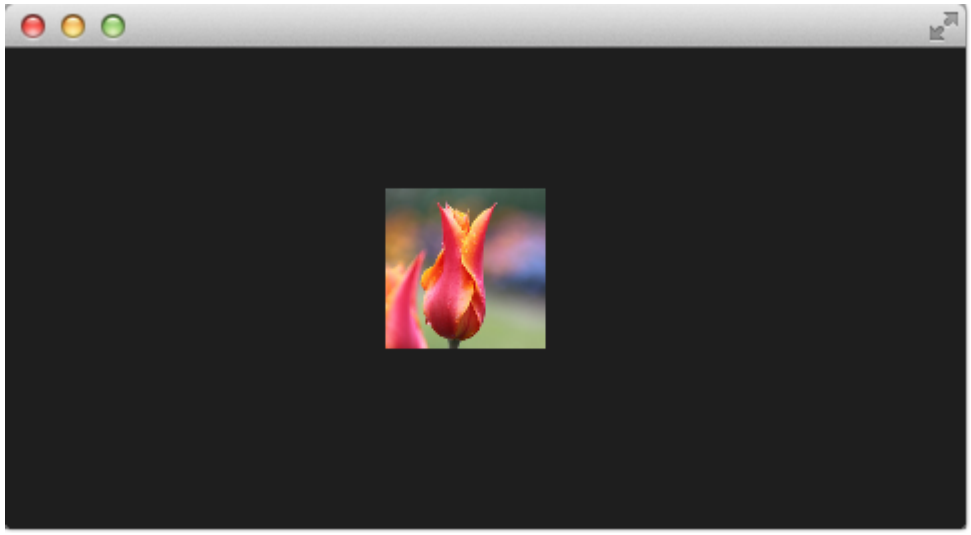
Fragment Shaders

The fragment shader is called for every pixel to be rendered. In this chapter, we will develop a small red lens which will increase the red color channel value of the source.

Setting up the scene

First, we set up our scene, with a grid centered in the field and our source image be displayed.

```
import QtQuick

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

    Grid {
        anchors.centerIn: parent
        spacing: 20
        rows: 2; columns: 4
        Image {
            id: sourceImage
            width: 80; height: width
            source: '../../assets/tulips.jpg'
        }
    }
}
```

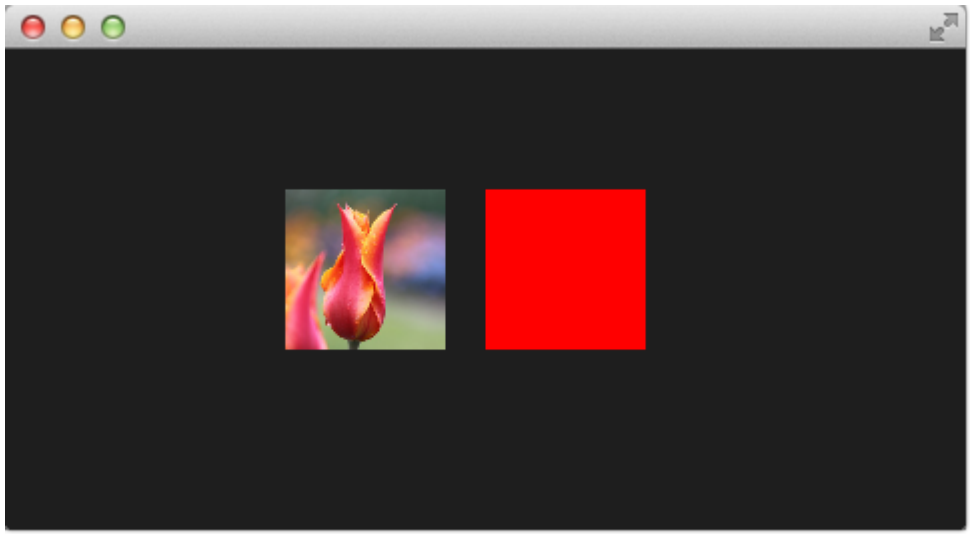


A red shader

Next, we will add a shader, which displays a red rectangle by providing for each fragment a red color value.

```
#version 440

layout(location=0) in vec2 qt_TexCoord0;

layout(location=0) out vec4 fragColor;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;
} ubuf;

layout(binding=1) uniform sampler2D source;

void main() {
    fragColor = vec4(1.0, 0.0, 0.0, 1.0) * ubuf.qt_Opacity;
}
```

In the fragment shader we simply assign a `vec4(1.0, 0.0, 0.0, 1.0)`, representing the color red with full opacity (alpha=1.0), to the `fragColor` for each fragment, turning each pixel to a solid red.



A red shader with texture

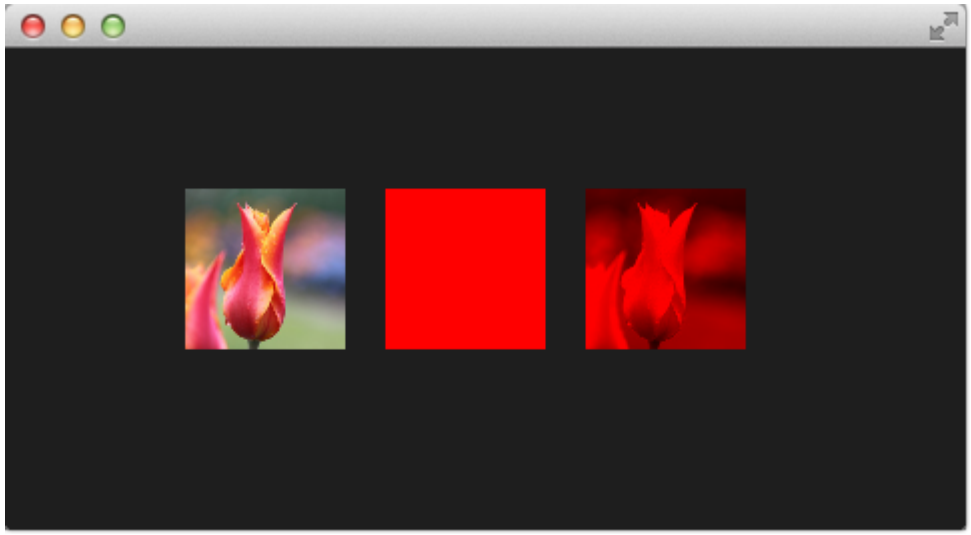
Now we want to apply the red color to each texture pixel. For this, we need the texture back in the vertex shader. As we don't do anything else in the vertex shader the default vertex shader is enough for us. We just need to provide a compatible fragment shader.

```
#version 440

layout(location=0) in vec2 qt_TexCoord0;

layout(location=0) out vec4 fragColor;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;
} ubuf;

layout(binding=1) uniform sampler2D source;
```

```
void main() {
    fragColor = texture(source, qt_TexCoord0) * vec4(1.0, 0.0,
}
```

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 `texture(source, qt_TexCoord0)` and apply the red color to it.



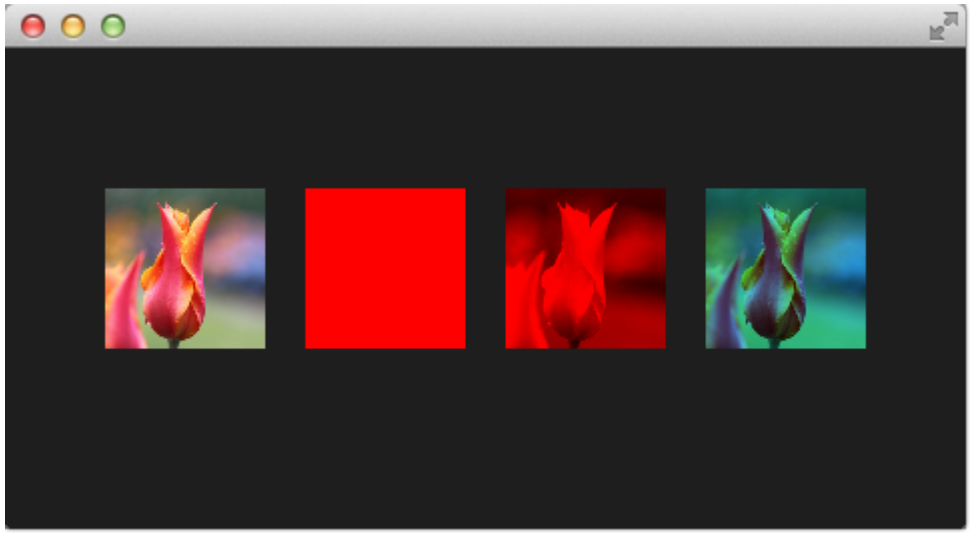
The red channel property

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 `float redChannel` inside the uniform buffer of the fragment shader. That is all that we need to do to make a value from the QML side available to the shader code.

> **TIP**
>
> Notice that the `redChannel` must come after the implicit `qt_Matrix` and `qt_Opacity` in the uniform buffer, `ubuf`. The order of the parameters after the `qt_` parameters is up to you, but `qt_Matrix` and `qt_Opacity` must come first and in that order.

```
#version 440

layout(location=0) in vec2 qt_TexCoord0;

layout(location=0) out vec4 fragColor;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float redChannel;
} ubuf;

layout(binding=1) uniform sampler2D source;

void main() {
    fragColor = texture(source, qt_TexCoord0) * vec4(ubuf.redCh
}
```

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.

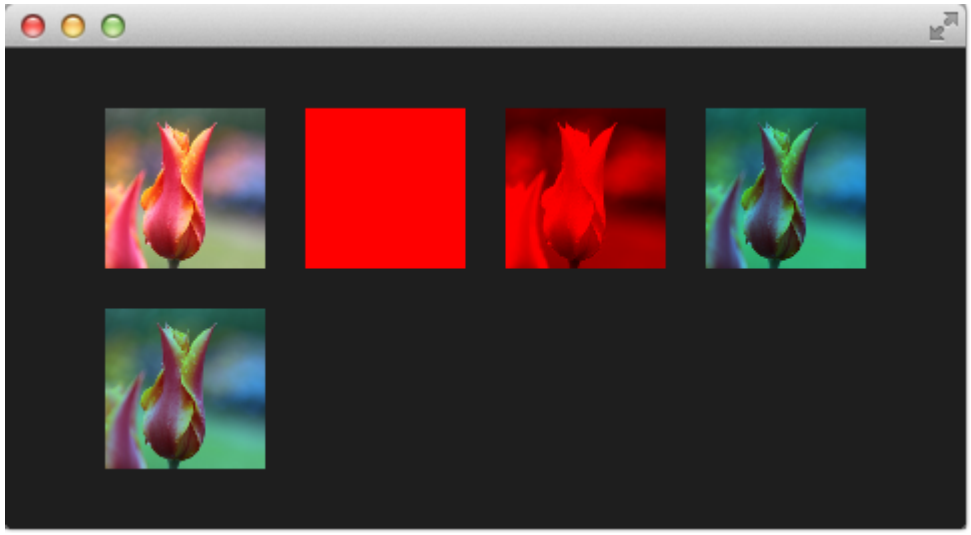


The red channel animated

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
    }

    fragmentShader: "red3.frag.qsb"
}
```

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).

Baking

Again, we need to bake the shaders. The following commands from the command line does that:

```
qsb --glsl 100es,120,150 --hlsl 50 --msl 12 -o red1.frag.qsb re
qsb --glsl 100es,120,150 --hlsl 50 --msl 12 -o red2.frag.qsb re
qsb --glsl 100es,120,150 --hlsl 50 --msl 12 -o red3.frag.qsb re
```

Wave Effect

In this more complex example, we will create a wave effect with the fragment shader. The waveform is based on the sinus curve and it influences the texture coordinates used for the color.

The qml file defines the properties and animation.

```
import QtQuick 2.5

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

    Row {
        anchors.centerIn: parent
        spacing: 20
        Image {
            id: sourceImage
            width: 160; height: width
            source: "../assets/coastline.jpg"
        }
        ShaderEffect {
            width: 160; height: width
            property variant source: sourceImage
            property real frequency: 8
            property real amplitude: 0.1
            property real time: 0.0
            NumberAnimation on time {
                from: 0; to: Math.PI*2; duration: 1000; loops:
            }

            fragmentShader: "wave.frag.qsb"
```

```
        }
    }
}
```

The fragment shader takes the properties and calculates the color of each pixel based on the properties.

```
#version 440

layout(location=0) in vec2 qt_TexCoord0;

layout(location=0) out vec4 fragColor;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float frequency;
    float amplitude;
    float time;
} ubuf;

layout(binding=1) uniform sampler2D source;

void main() {
    vec2 pulse = sin(ubuf.time - ubuf.frequency * qt_TexCoord0)
    vec2 coord = qt_TexCoord0 + ubuf.amplitude * vec2(pulse.x,
    fragColor = texture(source, coord) * ubuf.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:

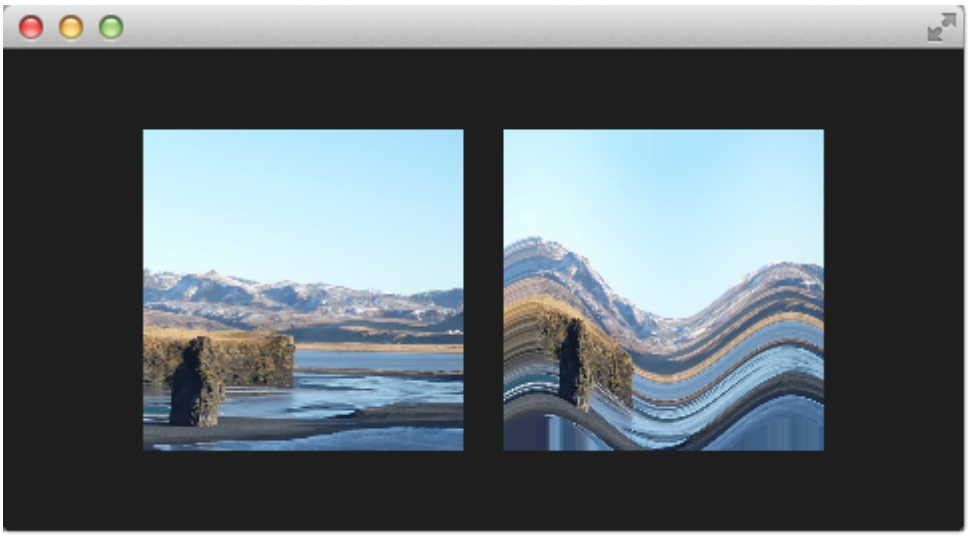
```
vec2 pulse = sin(ubuf.time - ubuf.frequency * qt_TexCoord0);
```

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:

```
vec2 coord = qt_TexCoord0 + ubuf.amplitude * vec2(pulse.x, -pul
```

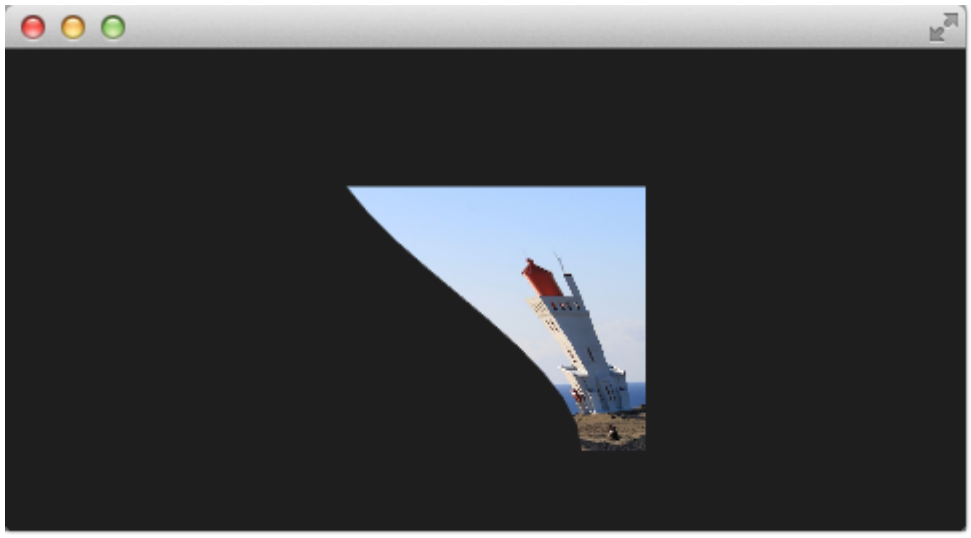
The texture coordinate is influenced by our pulse x-value. The result of this is a moving wave.



In this example we use a fragment shader, meaning that we move the pixels inside the texture of the rectangular item. If we wanted the entire item to move as a wave we would have to use a vertex shader.

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 from type `vec4`. To visualize the vertex shader we will program a genie effect. This effect is used to let a rectangular window area vanish into one point, like a genie disappearing into a lamp.



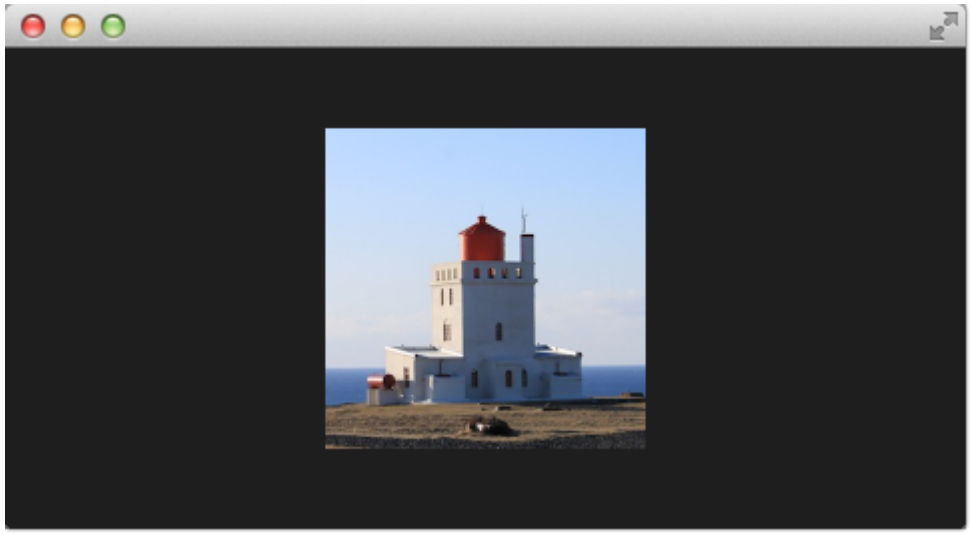
Setting up the scene

First, we will set up our scene with an image and a shader effect.

```
import QtQuick

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.min
            }
        }
    }
```

This provides 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. Additional 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 set up 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
    PauseAnimation { duration: 1000 }
}

SequentialAnimation on minimize {
```

```
        id: animNormalize
        running: !genieEffect.minimized
        NumberAnimation { to: 0; duration: 700; easing.type: Easing
        PauseAnimation { duration: 1300 }
    }
```

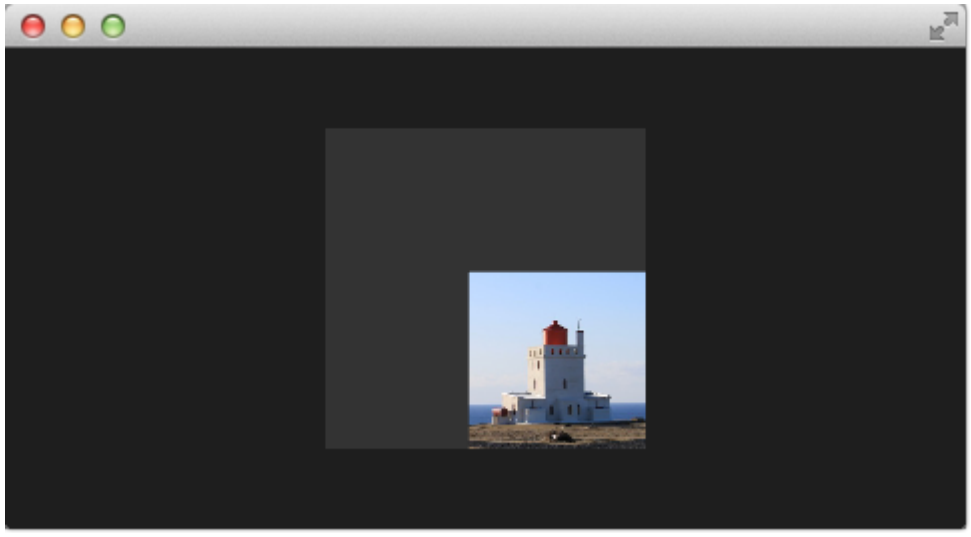
The animation is triggered by the toggling of the *minimized* property. Now that we have set up all our surroundings we finally can look at our vertex shader.

```
#version 440

layout(location=0) in vec4 qt_Vertex;
layout(location=1) in vec2 qt_MultiTexCoord0;

layout(location=0) out vec2 qt_TexCoord0;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float minimize;
    float width;
    float height;
} ubuf;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;
    vec4 pos = qt_Vertex;
    pos.y = mix(qt_Vertex.y, ubuf.height, ubuf.minimize);
    pos.x = mix(qt_Vertex.x, ubuf.width, ubuf.minimize);
    gl_Position = ubuf.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:

```
vec4 pos = qt_Vertex;
pos.y = mix(qt_Vertex.y, ubuf.height, ubuf.minimize);
pos.x = mix(qt_Vertex.x, ubuf.width, ubuf.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 minimized value, similar for x. Bear in mind the minimized value is animated by our sequential animation and travels from 0.0 to 1.0 (or vice versa).



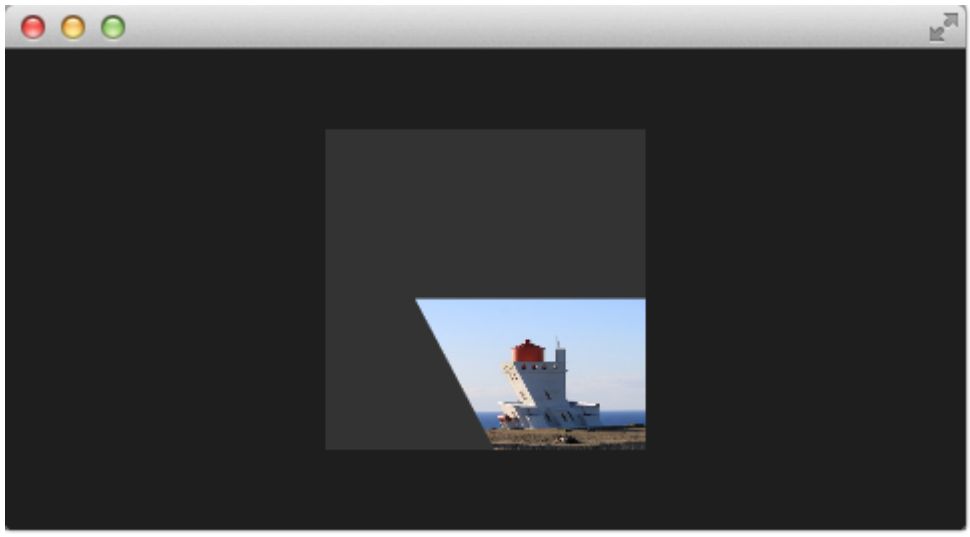
The resulting effect is not really the genie effect but is already a great step towards it.

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 on 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:

```
float t = pos.y / ubuf.height;
pos.x = mix(qt_Vertex.x, ubuf.width, t * minimize);
```

This results in 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 a y-position of 0 and the lower two vertexes x-positions both bend towards the width, so they bend towards the same x-position.



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 animations have in the sum the same duration (300+700+1000 and 700+1300).

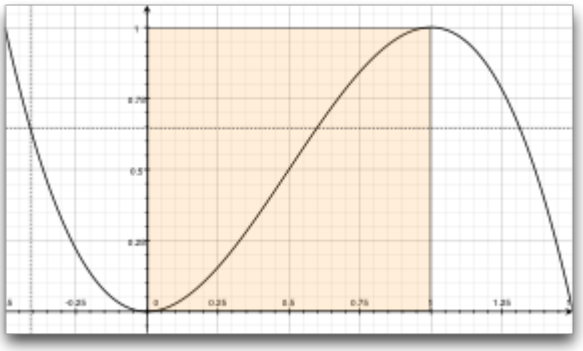
We first add and animate `bend` from QML.

```
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 }
    }
}
```

We then add `bend` to the uniform buffer, `ubuf` and use it in the shader to achieve a smoother bending.

```
#version 440

layout(location=0) in vec4 qt_Vertex;
layout(location=1) in vec2 qt_MultiTexCoord0;

layout(location=0) out vec2 qt_TexCoord0;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float minimize;
    float width;
    float height;
    float bend;
} ubuf;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;
    vec4 pos = qt_Vertex;
    pos.y = mix(qt_Vertex.y, ubuf.height, ubuf.minimize);
    float t = pos.y / ubuf.height;
    t = (3.0 - 2.0 * t) * t * t;
    pos.x = mix(qt_Vertex.x, ubuf.width, t * ubuf.bend);
    gl_Position = ubuf.qt_Matrix * pos;
}
```

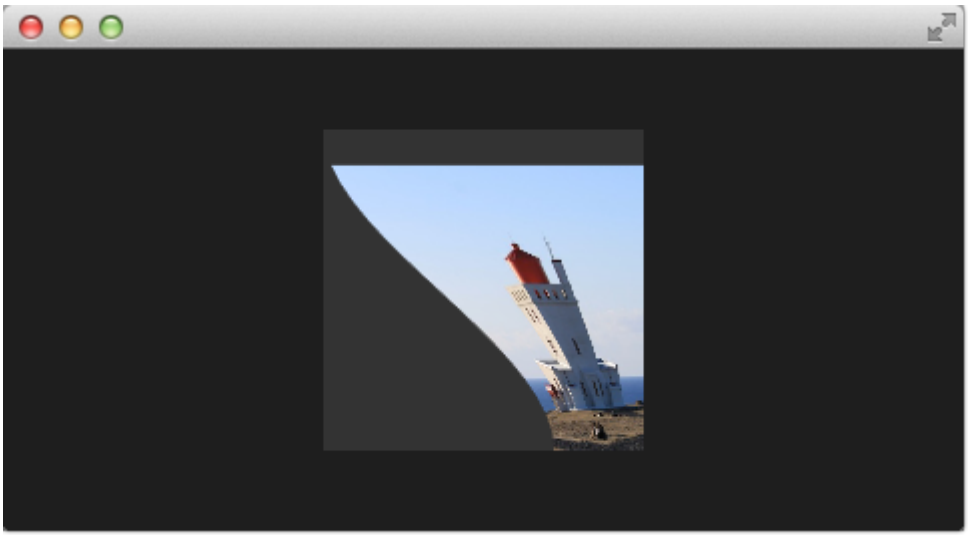
The curve starts smooth at the 0.0 value, grows then and stops smoothly towards the 1.0 value. Here is a plot of the function in the specified range. For us, only the range from 0..1 is from interest.



We also need to increase the number 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.

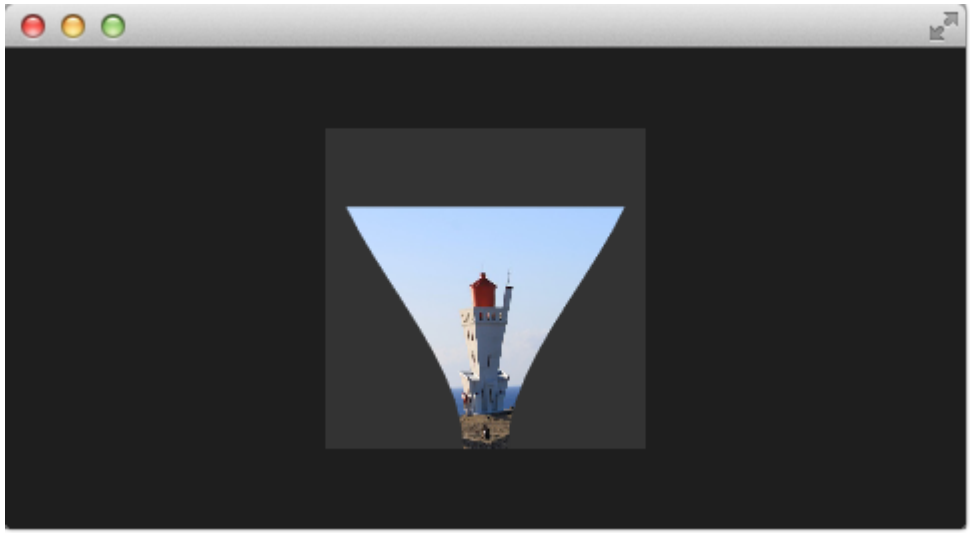
Choosing Sides

As a final enhancement, we want to be able to switch sides. The side is towards which point the genie effect vanishes. Until 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
    ...
}
```

```
#version 440

layout(location=0) in vec4 qt_Vertex;
layout(location=1) in vec2 qt_MultiTexCoord0;

layout(location=0) out vec2 qt_TexCoord0;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float minimize;
    float width;
    float height;
    float bend;
    float side;
} ubuf;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;
    vec4 pos = qt_Vertex;
    pos.y = mix(qt_Vertex.y, ubuf.height, ubuf.minimize);
```

```
    float t = pos.y / ubuf.height;
    t = (3.0 - 2.0 * t) * t * t;
    pos.x = mix(qt_Vertex.x, ubuf.side * ubuf.width, t * ubuf.b
    gl_Position = ubuf.qt_Matrix * pos;
}
```



Packaging

The last thing to-do is packaging 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.

```
// GenieEffect.qml
import QtQuick

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: "genieeffect.vert.qsb"
}
```

```
// genieeffect.vert
#version 440

layout(location=0) in vec4 qt_Vertex;
layout(location=1) in vec2 qt_MultiTexCoord0;

layout(location=0) out vec2 qt_TexCoord0;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float minimize;
    float width;
    float height;
    float bend;
    float side;
} ubuf;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;
    vec4 pos = qt_Vertex;
    pos.y = mix(qt_Vertex.y, ubuf.height, ubuf.minimize);
```

```
        float t = pos.y / ubuf.height;
        t = (3.0 - 2.0 * t) * t * t;
        pos.x = mix(qt_Vertex.x, ubuf.side * ubuf.width, t * ubuf.b
        gl_Position = ubuf.qt_Matrix * pos;
    }
```

You can use now the effect simply like this:

```
import QtQuick

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

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

We have simplified the code by removing our background rectangle and we assigned the image directly to the effect, instead of loading it inside a standalone image element.

Curtain Effect

In the last example for custom shader effects, I would like to bring you the curtain effect. This effect was published first in May 2011 as part of **Qt labs for shader effects (http://labs.qt.nokia.com/2011/05/03/qml-shadereffectitem-on-qgraphicsview/)** .

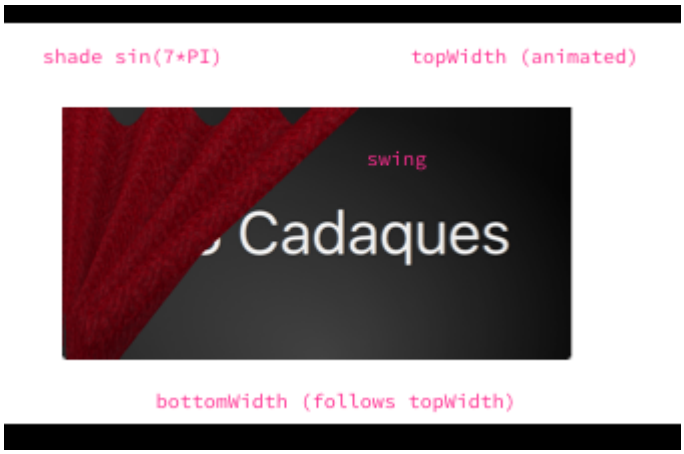
At that time I really loved these effects and the curtain effect was my favorite out of them. I just love how the curtain opens and hide the background object.

For this chapter, the effect has been adapted for Qt 6. It has also been slightly simplified to make it a better showcase.

The curtain image is called `fabric.png` . The effect then uses a vertex shader to swing the curtain forth and back and a fragment shader to apply shadows to show how the fabric folds.

The diagram below shows how the shader works. The waves are computed through a sin curve with 7 periods (7*PI=21.99…). The other

part is the swinging. The `topWidht` of the curtain is animated when the curtain is opened or closed. The `bottomWidth` follows the `topWidth` using a `SpringAnimation`. This creates the effect of the bottom part of the curtain swinging freely. The calculated `swing` component is the strength of the swing based on the y-component of the vertexes.



The curtain effect is implemented in the `CurtainEffect.qml` file where the fabric image act as the texture source. In the QML code, the `mesh` property is adjusted to make sure that the number of vertices is increased to give a smoother result.

```
import QtQuick

ShaderEffect {
    anchors.fill: parent

    mesh: GridMesh {
        resolution: Qt.size(50, 50)
    }

    property real topWidth: open?width:20
    property real bottomWidth: topWidth
    property real amplitude: 0.1
    property bool open: false
    property variant source: effectSource
```

```
        Behavior on bottomWidth {
            SpringAnimation {
                easing.type: Easing.OutElastic;
                velocity: 250; mass: 1.5;
                spring: 0.5; damping: 0.05
            }
        }

        Behavior on topWidth {
            NumberAnimation { duration: 1000 }
        }


        ShaderEffectSource {
            id: effectSource
            sourceItem: effectImage;
            hideSource: true
        }

        Image {
            id: effectImage
            anchors.fill: parent
            source: "../assets/fabric.png"
            fillMode: Image.Tile
        }

        vertexShader: "curtain.vert.qsb"

        fragmentShader: "curtain.frag.qsb"
    }
```

The vertex shader, shown below, reshapes the curtain based on the `topWidth` and `bottomWidth` properties, extrapolating the shift based on the y-coordinate. It also calculates the `shade` value, which is used in the fragment shader. The `shade` property is passed through an additional output in `location` 1.

```
#version 440

layout(location=0) in vec4 qt_Vertex;
layout(location=1) in vec2 qt_MultiTexCoord0;

layout(location=0) out vec2 qt_TexCoord0;
layout(location=1) out float shade;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float topWidth;
    float bottomWidth;
    float width;
    float height;
    float amplitude;
} ubuf;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    qt_TexCoord0 = qt_MultiTexCoord0;

    vec4 shift = vec4(0.0, 0.0, 0.0, 0.0);
    float swing = (ubuf.topWidth - ubuf.bottomWidth) * (qt_Vert
    shift.x = qt_Vertex.x * (ubuf.width - ubuf.topWidth + swing

    shade = sin(21.9911486 * qt_Vertex.x / ubuf.width);
    shift.y = ubuf.amplitude * (ubuf.width - ubuf.topWidth + sw

    gl_Position = ubuf.qt_Matrix * (qt_Vertex - shift);

    shade = 0.2 * (2.0 - shade) * ((ubuf.width - ubuf.topWidth
}
```

In the fragment shader below, the `shade` is picked up as an input in `location` 1 and is then used to calculate the `fragColor`, which is used to draw the pixel in question.

```
#version 440

layout(location=0) in vec2 qt_TexCoord0;
layout(location=1) in float shade;

layout(location=0) out vec4 fragColor;

layout(std140, binding=0) uniform buf {
    mat4 qt_Matrix;
    float qt_Opacity;

    float topWidth;
    float bottomWidth;
    float width;
    float height;
    float amplitude;
} ubuf;

layout(binding=1) uniform sampler2D source;

void main() {
    highp vec4 color = texture(source, qt_TexCoord0);
    color.rgb *= 1.0 - shade;
    fragColor = color;
}
```

The combination of QML animations and passing variables from the vertex shader to the fragment shader demonstrates how QML and shaders can be used together to build complex, animated, effects.

The effect itself is used from the `curtaindemo.qml` file shown below.

```
import QtQuick

Item {
    id: root
    width: background.width; height: background.height


    Image {
        id: background
        anchors.centerIn: parent
        source: '../assets/background.png'
    }

    Text {
        anchors.centerIn: parent
        font.pixelSize: 48
        color: '#efefef'
        text: 'Qt 6 Book'
    }

    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 when the user clicks or taps the area.

Summary

When creating new user interfaces effects can make a difference between a dull interface and a sparkling interface. In this chapter we've looked at particle effects and shaders.

Particles provide a powerful and fun way to express graphical phenomena like smoke. firework, random visual elements. The particles look are playful and have a great potential when used wisely to create some eye catcher in any user interface. Using too many particle effects inside a user interface will definitely lead to the impression towards a game. Creating games is also the real strength of the particles.

Shaders can be used to take the QML scene to the next level. Using vertex shaders it is possible to change the shape of elements, while fragment shaders are used to alter the texture of an element, e.g. changing the colour, or transforming the surface to produce effects such as waves.

In this chapter we've scratched the surface of these two topics. For the interested reader, there are many more possibilities to explore.

Multimedia

The multimedia elements in the Qt Multimedia makes it possible to playback and record media such as sound, video or pictures. Decoding and encoding are handled through platform-specific backends. For instance, the popular GStreamer framework is used on Linux, WMF is used on Windows, AVFramework on OS X and iOS and the Android multimedia APIs are used on Android.

The multimedia elements are not a part of the Qt Quick core API. Instead, they are provided through a separate API made available by importing Qt Multimedia as shown below:

```
import QtMultimedia
```

Playing Media

The most basic case of multimedia integration in a QML application is for it to playback media. The `QtMultimedia` module supports this by providing a dedicated QML component: the `MediaPlayer`.

The `MediaPlayer` component is a non-visual item that connects a media source to one or several output channel(s). Depending on the nature of the media (i.e. audio, image or video) various output channel(s) can be configured.

Playing audio

In the following example, the `MediaPlayer` plays a mp3 sample audio file from a remote URL in an empty window:

```
import QtQuick
import QtMultimedia

Window {
    width: 1024
    height: 768
    visible: true

    MediaPlayer {
        id: player
        source: "https://file-examples-com.github.io/uploads/20
        audioOutput: AudioOutput {}
    }

    Component.onCompleted: {
```

```
            player.play()
        }
    }
```

In this example, the `MediaPlayer` defines two attributes:

- `source` : it contains the URL of the media to play. It can either be embedded (`qrc://`), local (`file://`) or remote (`https://`).
- `audioOutput` : it contains an audio output channel, `AudioOutput` , connected to a physical output device. By default, it will use the default audio output device of the system.

As soon as the main component has been fully initialized, the player's `play` function is called:

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

Playing a video

If you want to play visual media such as pictures or videos, you must also define a `VideoOutput` element to place the resulting image or video in the user interface.

In the following example, the `MediaPlayer` plays a mp4 sample video file from a remote URL and centers the video content in the window:

```
import QtQuick
import QtMultimedia

Window {
    width: 1920
    height: 1080
```

```
        visible: true

        MediaPlayer {
            id: player
            source: "https://file-examples-com.github.io/uploads/20
            audioOutput: AudioOutput {}
            videoOutput: videoOutput
        }

        VideoOutput {
            id: videoOutput
            anchors.fill: parent
            anchors.margins: 20
        }

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

In this example, the `MediaPlayer` defines a third attribute:

- `videoOutput` : it contains the video output channel, `VideoOutput` , representing the visual space reserved to display the video in the user interface.

> **TIP**
>
> Please note that the `VideoOutput` component is a visual item. As such, it's essential that it is created within the visual components hierarchy and not within the `MediaPlayer` itself.

Controlling the playback

The `MediaPlayer` component offers several useful properties. 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.

It's also possible to leverage `AudioOutput` and `VideoOutput` properties to customize the experience and provide, for instance, volume control.

The following example adds custom controls for the playback:

- a volume slider
- a play/pause button
- a progress slider

```
import QtQuick
import QtQuick.Controls
import QtMultimedia

Window {
    id: root
    width: 1920
    height: 1080
    visible: true

    MediaPlayer {
        id: player
        source: Qt.resolvedUrl("sample-5s.mp4")
        audioOutput: audioOutput
        videoOutput: videoOutput
    }

    AudioOutput {
        id: audioOutput
        volume: volumeSlider.value
    }

    VideoOutput {
        id: videoOutput
```

```
        width: videoOutput.sourceRect.width
        height: videoOutput.sourceRect.height
        anchors.horizontalCenter: parent.horizontalCenter
    }

    Slider {
        id: volumeSlider
        anchors.top: parent.top
        anchors.right: parent.right
        anchors.margins: 20
        orientation: Qt.Vertical
        value: 0.5
    }

    Item {
        height: 50
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        anchors.margins: 20

        Button {
            anchors.horizontalCenter: parent.horizontalCenter
            text: player.playbackState ===  MediaPlayer.Playing
            onClicked: {
                switch(player.playbackState) {
                    case MediaPlayer.PlayingState: player.pause
                    case MediaPlayer.PausedState: player.play()
                    case MediaPlayer.StoppedState: player.play(
                }
            }
        }

        Slider {
            id: progressSlider
            width: parent.width
            anchors.bottom: parent.bottom
            enabled: player.seekable
            value: player.duration > 0 ? player.position / play
            background: Rectangle {
```

```
                implicitHeight: 8
                color: "white"
                radius: 3
                Rectangle {
                    width: progressSlider.visualPosition * pare
                    height: parent.height
                    color: "#1D8BF8"
                    radius: 3
                }
            }
            handle: Item {}
            onMoved: function () {
                player.position = player.duration * progressSli
            }
        }
    }

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

The volume slider

A vertical `Slider` component is added on the top right corner of the window, allowing the user to control the volume of the media:

```
Slider {
    id: volumeSlider
    anchors.top: parent.top
    anchors.right: parent.right
    anchors.margins: 20
    orientation: Qt.Vertical
    value: 0.5
}
```

The volume attribute of the `AudioOutput` is then mapped to the value of the slider:

```
AudioOutput {
    id: audioOutput
    volume: volumeSlider.value
}
```

Play / Pause

A `Button` component reflects the playback state of the media and allows the user to control this state:

```
Button {
    anchors.horizontalCenter: parent.horizontalCenter
    text: player.playbackState ===  MediaPlayer.PlayingState ?
    onClicked: {
        switch(player.playbackState) {
            case MediaPlayer.PlayingState: player.pause(); brea
            case MediaPlayer.PausedState: player.play(); break;
            case MediaPlayer.StoppedState: player.play(); break
        }
    }
}
```

Depending on the playback state, a different text will be displayed in the button. When clicked, the corresponding action will be triggered and will either play or pause the media.

> **TIP**
>
> The possible playback states are listed below:
>
> - `MediaPlayer.PlayingState` : The media is currently playing.
> - `MediaPlayer.PausedState` : Playback of the media has been suspended.
> - `MediaPlayer.StoppedState` : Playback of the media is yet to begin.

Interactive progress slider

A `Slider` component is added to reflect the current progress of the playback. It also allows the user to control the current position of the playback.

```
Slider {
    id: progressSlider
    width: parent.width
    anchors.bottom: parent.bottom
    enabled: player.seekable
    value: player.duration > 0 ? player.position / player.durat
    background: Rectangle {
        implicitHeight: 8
        color: "white"
        radius: 3
        Rectangle {
            width: progressSlider.visualPosition * parent.width
            height: parent.height
            color: "#1D8BF8"
            radius: 3
        }
    }
    handle: Item {}
    onMoved: function () {
```

```
            player.position = player.duration * progressSlider.posi
        }
    }
```

A few things to note on this sample:

- This slider will only be enabled when the media is `seekable` (line 5)
- Its value will be set to the current media progress, i.e. `player.position / player.duration` (line 6)
- The media position will be *(also)* updated when the slider is moved by the user (lines 19-21)

The media status

When using `MediaPlayer` to build a media player, it is good to monitor the `status` property of the player. Here is an enumeration of the possible statuses, ranging from `MediaPlayer.Buffered` to `MediaPlayer.InvalidMedia`. The possible values are summarized in the bullets below:

- `MediaPlayer.NoMedia`. No media has been set. Playback is stopped.
- `MediaPlayer.Loading`. The media is currently being loaded.
- `MediaPlayer.Loaded`. The media has been loaded. Playback is stopped.
- `MediaPlayer.Buffering`. The media is buffering data.
- `MediaPlayer.Stalled`. The playback has been interrupted while the media is buffering data.
- `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.

- `MediaPlayer.UnknownStatus` . The status of the media is unknown.

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 an effect. For example, the end can be reached, or it can be invalid, causing playback to stop.

> **TIP**
>
> 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.

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.

```
import QtQuick
import QtMultimedia

Window {
    width: 300
    height: 200
    visible: true

    SoundEffect {
        id: beep
        source: Qt.resolvedUrl("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.

The following example shows how sound effects elements can be used to accompany transition between visual states using animations:

```
import QtQuick
import QtQuick.Controls
import QtMultimedia

Window {
    width: 500
    height: 500
    visible: true

    SoundEffect { id: beep; source: "file:beep.wav"}
    SoundEffect { id: swosh; source: "file:swosh.wav" }

    Rectangle {
        id: rectangle

        anchors.centerIn: parent

        width: 300
        height: width

        color: "red"
        state: "DEFAULT"

        states: [
            State {
```

```
                name: "DEFAULT"
                PropertyChanges { target: rectangle; rotation:
            },
            State {
                name: "REVERSE"
                PropertyChanges { target: rectangle; rotation:
            }
        ]

        transitions: [
            Transition {
                to: "DEFAULT"
                ParallelAnimation {
                    ScriptAction { script: swosh.play(); }
                    PropertyAnimation { properties: "rotation";
                }
            },
            Transition {
                to: "REVERSE"
                ParallelAnimation {
                    ScriptAction { script: beep.play(); }
                    PropertyAnimation { properties: "rotation";
                }
            }
        ]
    }

    Button {
        anchors.centerIn: parent
        text: "Flip!"
        onClicked: rectangle.state = rectangle.state === "DEFAU
    }
}
```

In this example, we want to apply a 180 rotation animation to our `Rectangle` whenever the "Flip!" button is clicked. We also want to play a different sound when the rectangle flips in one direction or the other.

To do so, we first start by loading our effects:

```
SoundEffect { id: beep; source: "file:beep.wav"}
SoundEffect { id: swosh; source: "file:swosh.wav" }
```

Then we define two states for our rectangle, `DEFAULT` and `REVERSE`, specifying the expected rotation angle for each state:

```
states: [
    State {
        name: "DEFAULT"
        PropertyChanges { target: rectangle; rotation: 0; }
    },
    State {
        name: "REVERSE"
        PropertyChanges { target: rectangle; rotation: 180; }
    }
]
```

To provide between-states animation, we define two transitions:

```
transitions: [
    Transition {
        to: "DEFAULT"
        ParallelAnimation {
            ScriptAction { script: swosh.play(); }
            PropertyAnimation { properties: "rotation"; duratio
        }
    },
    Transition {
        to: "REVERSE"
        ParallelAnimation {
            ScriptAction { script: beep.play(); }
            PropertyAnimation { properties: "rotation"; duratio
        }
```

```
        }
    ]
```

Notice the `ScriptAction { script: swosh.play(); }` line. Using the `ScriptAction` component we can run an arbitrary script as part of the animation, which allows us to play the desired sound effect as part of the animation.

> **TIP**
>
> 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.

> **WARNING**
>
> When the PulseAudio backend is used, `stop` will not stop instantaneously, but only prevent further loops. This is due to limitations in the underlying API.

Video Streams

The `VideoOutput` element is not limited to be used in combination with a `MediaPlayer` element. It can also be used with various video sources to display video streams.

For instance, we can use the `VideoOutput` to display the live video stream of the user's `Camera`. To do so, we will combine it with two components: `Camera` and `CaptureSession`.

```
import QtQuick
import QtMultimedia

Window {
    width: 1024
    height: 768
    visible: true

    CaptureSession {
        id: captureSession
        camera: Camera {}
        videoOutput: output
    }

    VideoOutput {
        id: output
        anchors.fill: parent
    }

    Component.onCompleted: captureSession.camera.start()
}
```

The `CaptureSession` component provides a simple way to read a camera stream, capture still images or record videos.

As the `MediaPlayer` component, the `CaptureSession` element provides a `videoOuput` attribute. We can thus use this attribute to configure our own visual component.

Finally, when the application is loaded, we can start the camera recording:

```
Component.onCompleted: captureSession.camera.start()
```

> **TIP**
>
> Depending on your operating system, this application may require sensitive access permission(s). If you run this sample application using the `qml` binary, those permissions will be requested automatically.
>
> However, if you run it as an independant program you may need to request those permissions first (e.g.: under MacOS, you would need a dedicated .plist file bundled with your application).

Capturing Images

One of the key features of the `Camera` element is that is can be used to take pictures. We will use this in a simple stop-motion application. By building the application, you will learn how to show a viewfinder, switch between cameras, snap photos and 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

The viewfinder part of the camera is made using a `VideoOutput` element as video output channel of a `CaptureSession`. The `CaptureSession` in turns uses a `Camera` component to configure the device. This will display a live video stream from the camera.

```
CaptureSession {
    id: captureSession
    videoOutput: output
    camera: Camera {}
    imageCapture: ImageCapture {
        onImageSaved: function (id, path) {
            imagePaths.append({"path": path})
            listView.positionViewAtEnd()
        }
    }
}

VideoOutput {
    id: output
    anchors.fill: parent
}
```

> **TIP**
>
> You can have more control on the camera behaviour by using dedicated `Camera` properties such as `exposureMode`, `whiteBalanceMode` or `zoomFactor`.

The captured images list

The list of photos is a `ListView` oriented horizontally that 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 {
        required property string path
        height: 100
        source: path
        fillMode: Image.PreserveAspectFit
    }

    Rectangle {
        anchors.fill: parent
        anchors.topMargin: -10

        color: "black"
        opacity: 0.5
    }
}
```

For the shooting of images, the `CaptureSession` element contains a set of sub-elements for various tasks. To capture still pictures, the

`CaptureSession.imageCapture` element is used. When you call the `captureToFile` method, a picture is taken and saved in the user's local pictures directory. This results in the `CaptureSession.imageCapture` emitting the `imageSaved` signal.

```
Button {
    id: shotButton

    width: parent.buttonWidth
    height: parent.buttonHeight

    text: qsTr("Take Photo")
    onClicked: {
        captureSession.imageCapture.captureToFile()
    }
}
```

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.

```
CaptureSession {
    id: captureSession
    videoOutput: output
    camera: Camera {}
    imageCapture: ImageCapture {
        onImageSaved: function (id, path) {
            imagePaths.append({"path": path})
            listView.positionViewAtEnd()
        }
    }
}
```

> **TIP**
>
> For showing a preview, connect to the `imageCaptured` signal and use the `preview` signal argument as `source` of an `Image` element. An `id` 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.

Switching between cameras

If the user has multiple cameras, it can be handy to provide a way of switching between those. It's possible to achieve this by using the `MediaDevices` element in conjunction with a `ListView`. In our case, we'll use a `ComboBox` component:

```
MediaDevices {
    id: mediaDevices
}

ComboBox {
    id: cameraComboBox

    width: parent.buttonWidth
    height: parent.buttonHeight

    model: mediaDevices.videoInputs
    textRole: "description"

    displayText: captureSession.camera.cameraDevice.description

    onActivated: function (index) {
```

```
            captureSession.camera.cameraDevice = cameraComboBox.cur
        }
    }
```

The `model` property of the `ComboBox` is set to the `videoInputs` property of our `MediaDevices`. This last property contains the list of usable video inputs. We then set the `displayText` of the control to the description of the camera device (`captureSession.camera.cameraDevice.description`).

Finally, when the user switches the video input, the cameraDevice is updated to reflect that change: `captureSession.camera.cameraDevice = cameraComboBox.currentValue`.

The playback

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"
        root.setImageIndex(0)
        playTimer.start()
    }

    function setImageIndex(i) {
        root._imageIndex = i

        if (root._imageIndex >= 0 && root._imageIndex < imagePaths.
```

```
            image.source = imagePaths.get(root._imageIndex).path
        } else {
            image.source = ""
        }
    }

    Timer {
        id: playTimer

        interval: 200
        repeat: false

        onTriggered: {
            if (root._imageIndex + 1 < imagePaths.count) {
                root.setImageIndex(root._imageIndex + 1)
                playTimer.start()
            } else {
                root.setImageIndex(-1)
                root.state = ""
            }
        }
    }
```

Summary

The media API provided by Qt provides mechanisms for playing and capturing video and audio. Through the `VideoOutput` element, video streams can be rendered 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 recording camera streams, a combination of `CaptureSession` and `Camera` elements can be used.

Qt Quick 3D

The Qt Quick 3D module takes the power of QML to the third dimension. Using Qt Quick 3D you can create three dimensional scenes and use the property bindings, state management, animations, and more from QML to make the scene interactive. You can even mix 2D and 3D contents in various way to create a mixed environment.

Just as Qt provides an abstraction for 2D graphics, Qt Quick 3D relies on an abstraction layer for the various rendering APIs supported. In order to use Qt Quick 3D it is recommended to use a platform provding at least one of the following APIs:

- OpenGL 3.3+ (support from 3.0)
- OpenGL ES 3.0+ (limited support for OpenGL ES 2)
- Direct3D 11.1
- Vulcan 1.0+
- Metal 1.2+

The Qt Quick Software Adaption, i.e. the software only rendering stack, does not support 3D contents.

In this chapter we will take you through the basics of Qt Quick 3D, letting you create interactive 3D scenes based on built in meshes as well as assets created in external tools. We will also look at animations and mixing of 2D and 3D contents.

The Basics

In this section we will walk through the basics of Qt Quick 3D. This includes working with the built in shapes (meshes), using lights, and transformations in 3D.

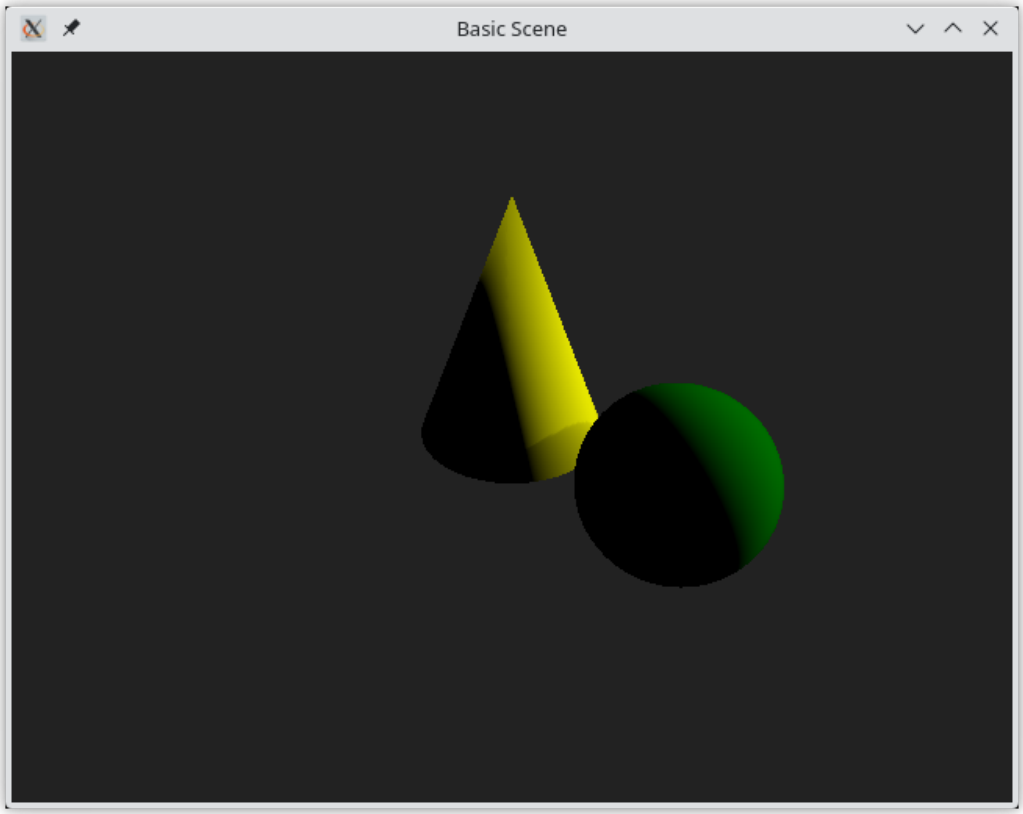
A Basic Scene

A 3D scene consists of a few standard elements:

- `View3D`, which is the top level QML element representing the entire 3D scene.
- `SceneEnvironment`, controls how the scene is rendered, including how the background, or sky box, is rendered.
- `PerspectiveCamera`, the camera in the scene. Can also be a `OrthographicCamera`, or even a custom camera with a custom projection matrix.

In addition to this, the scene usually contains `Model` instances representing objects in the 3D space, and lights.

We will look at how these elements interact by creating the scene shown below.



First of all, the QML code is setup with a `View3D` as the main element, filling the window. We also import the `QtQuick3D` module.

The `View3D` element can be seen as any other Qt Quick element, just that inside of it, the 3D contents will be rendered.

```
import QtQuick
import QtQuick3D

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Basic Scene")

    View3D {
        anchors.fill: parent
```

```
            // ...

        }
    }
```

Then we setup the `SceneEnvironment` with a solid background colour. This is done inside the `View3D` element.

```
    environment: SceneEnvironment {
        clearColor: "#222222"
        backgroundMode: SceneEnvironment.Color
    }
```

The `SceneEnvironment` can be used to control a lot more rendering parameters, but for now, we only use it to set a solid background colour.

The next step is to add *meshes* to the scene. A mesh represents an object in 3D space. Each mesh is created using a `Model` QML element.

A model can be used to load 3D assets, but there are a few built-in meshes allowing us to get started without involving the complexity of 3D assets management. In the code below, we create a `#Cone` and a `#Sphere` .

In addition to the shape of the mesh, we position them in 3D space and provide them with a material with a simple, diffuse base colour. We will discuss materials more in the [Materials and Light]("Materials and Lights") section

When positioning elements in 3D space, coordinates are expressed as `Qt.vector3d(x, y, z)` where the `x` axis controls the horizontal movement, `y` is the vertical movement, and `z` the how close or far away something is.

By default, the positive direction of the `x` axis is to the right, positive `y` points upwards, and positive `z` out of the screen. I say default,

because this depends on the projection matrix of the camera.

```
Model {
    position: Qt.vector3d(0, 0, 0)
    scale: Qt.vector3d(1, 1.25, 1)
    source: "#Cone"
    materials: [ PrincipledMaterial { baseColor: "yellow"; } ]
}

Model {
    position: Qt.vector3d(80, 0, 50)
    source: "#Sphere"
    materials: [ PrincipledMaterial { baseColor: "green"; } ]
}
```

Once we have lights in the scene we add a `DirectionalLight`, which is a light that works much like the sun. It adds an even light in a pre-determined direction. The direction is controlled using the `eulerRotation` property where we can rotate the light direction around the various axes.

By setting the `castsShadow` property to `true` we ensure that the light generates shadows as can be seen on cone, where the shadow from the sphere is visible.

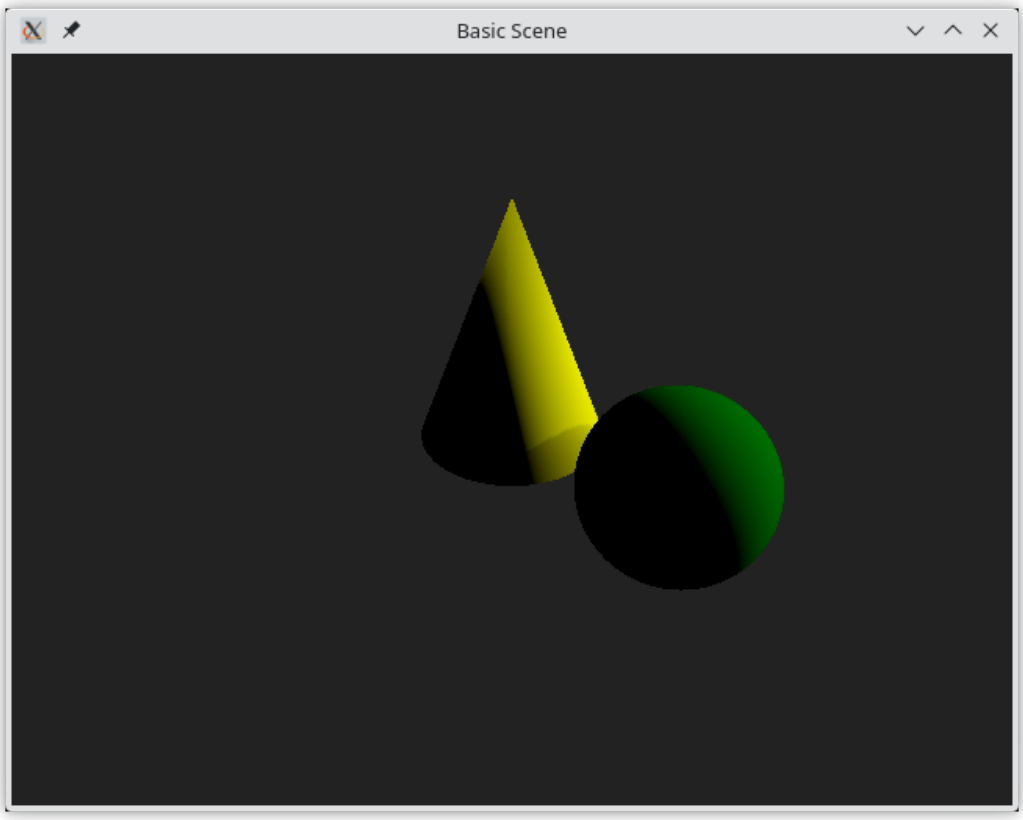
```
DirectionalLight {
    eulerRotation.x: -20
    eulerRotation.y: 110

    castsShadow: true
}
```

The last piece of the puzzle is to add a camera to the scene. There are various cameras for various perspectives, but for a realistic projection, the `ProjectionCamera` is the one to use.

In the code, we place the camera using the `position` property. It is then possible to direct the camera using the `eulerRotation` property, but instead we call the `lookAt` method from the `Component.onCompleted` signal handler. This rotates the camera to look at a specific direction once it has been created and initialized.

```
PerspectiveCamera {
    position: Qt.vector3d(0, 200, 300)
    Component.onCompleted: lookAt(Qt.vector3d(0, 0, 0))
}
```

The resulting scene can be seen below.



So, all in all, a minimal scene consists of a `View3D` with an `SceneEnvironment`, something to look at, e.g. a `Model` with a mesh, a

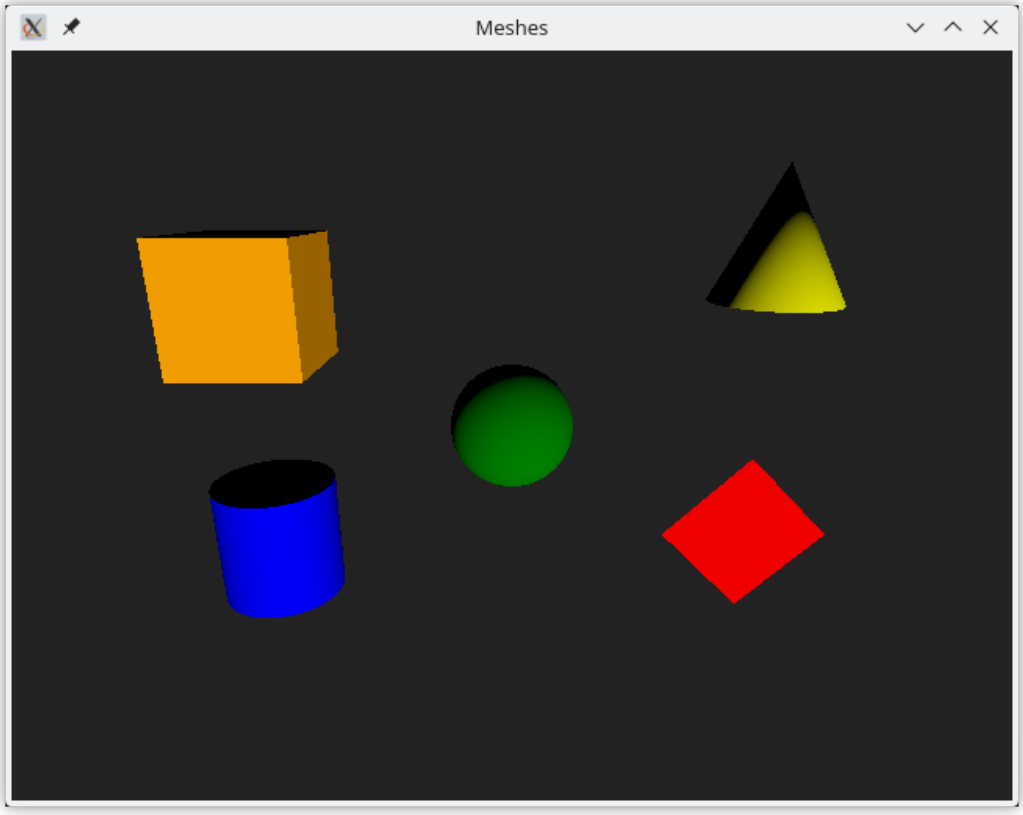
light source, e.g. a `DirectionalLight` , and something to look with, e.g. a `PerspectiveCamera` .

The Built-in Meshes

In the previous example, we used the built-in cone and sphere. Qt Quick 3D comes with the following built in meshes:

- `#Cube`
- `#Cone`
- `#Sphere`
- `#Cylinder`
- `#Rectangle`

These are all shown in the illustration below. (top-left: Cube, top-right: Cone, center: Sphere, bottom-left: Cylinder, bottom-right: Rectangle)



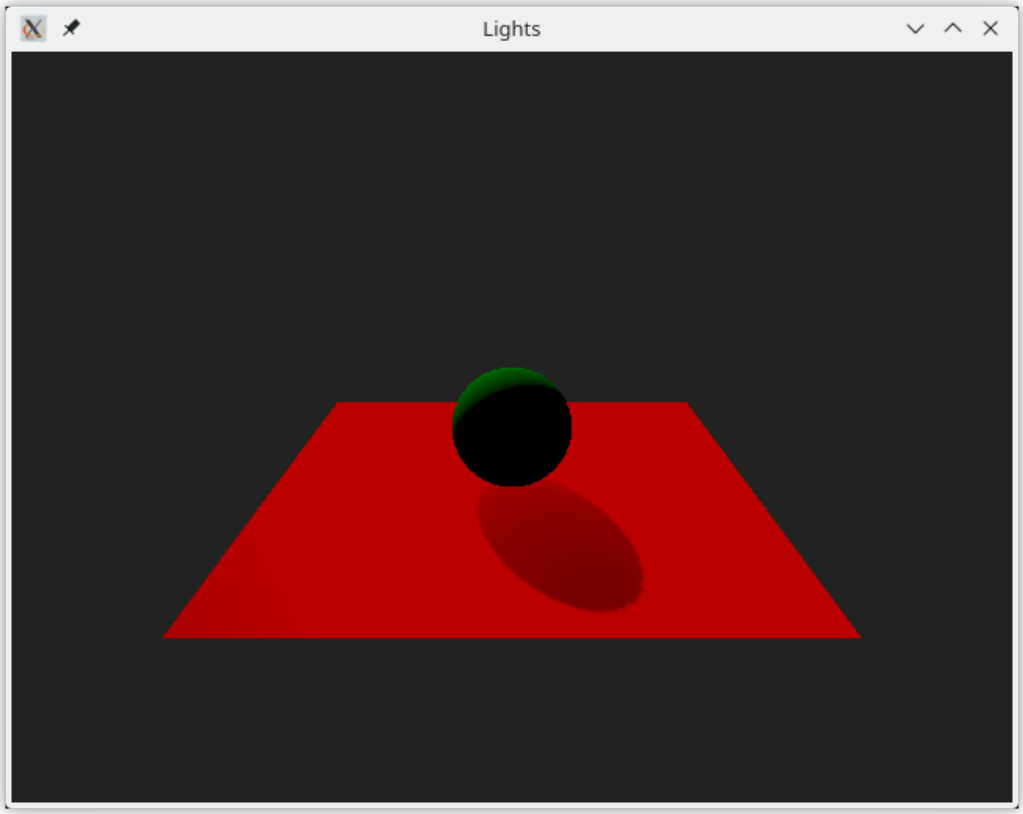
> **Tip**
>
> One caveat is that the `#Rectangle` is one-sided. That means that it is only visible from one direction. This means that the `eulerRotation` property is important.

When working with real scenes, the meshes are exported from a design tool and then imported into the Qt Quick 3D scene. We look at this in more detail in the **Working with Assets (/ch12-qtquick3d/assets.html)** section.

Lights

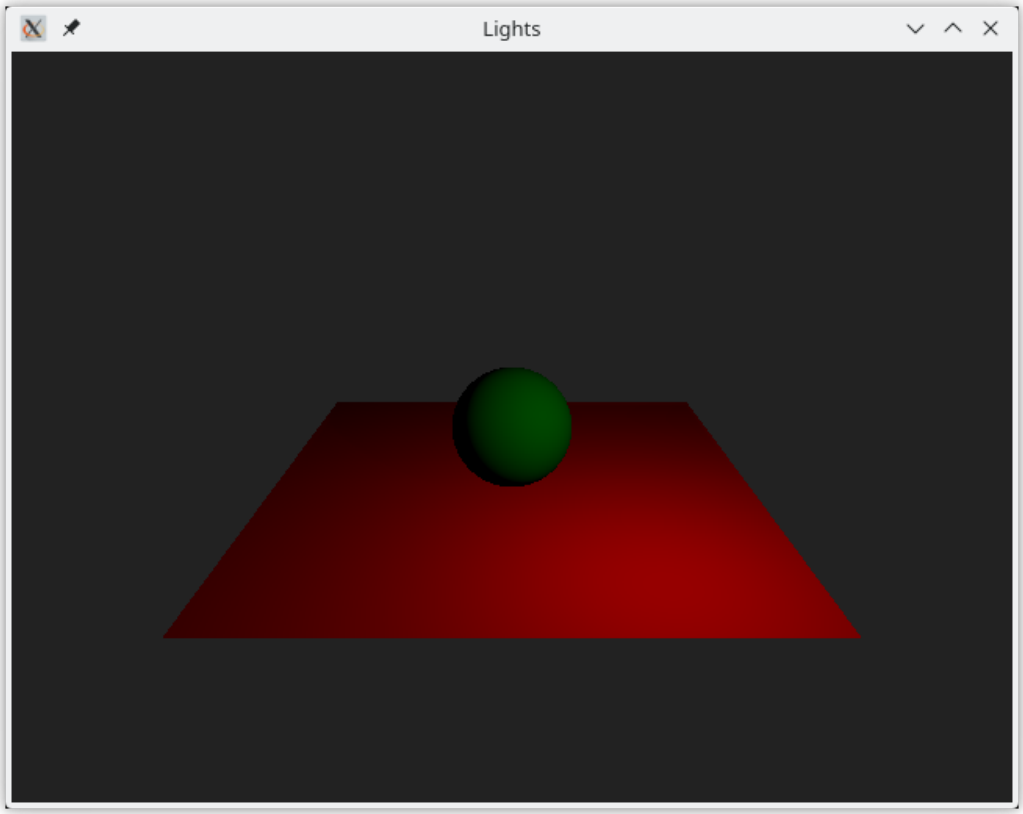
Just as with meshes, Qt Quick 3D comes with a number of pre-defined light sources. These are used to light the scene in different ways.

The first one, `DirectionalLight`, should be familiar from our previous example. It works much as the sun, and casts light uniformly over the scene in a given direction. If the `castsShadow` property is set to `true`, the light will cast shadows, as shown in the illustration below. This property is available for all the light sources.




```
DirectionalLight {
    eulerRotation.x: 210
    eulerRotation.y: 20

    castsShadow: true
}
```

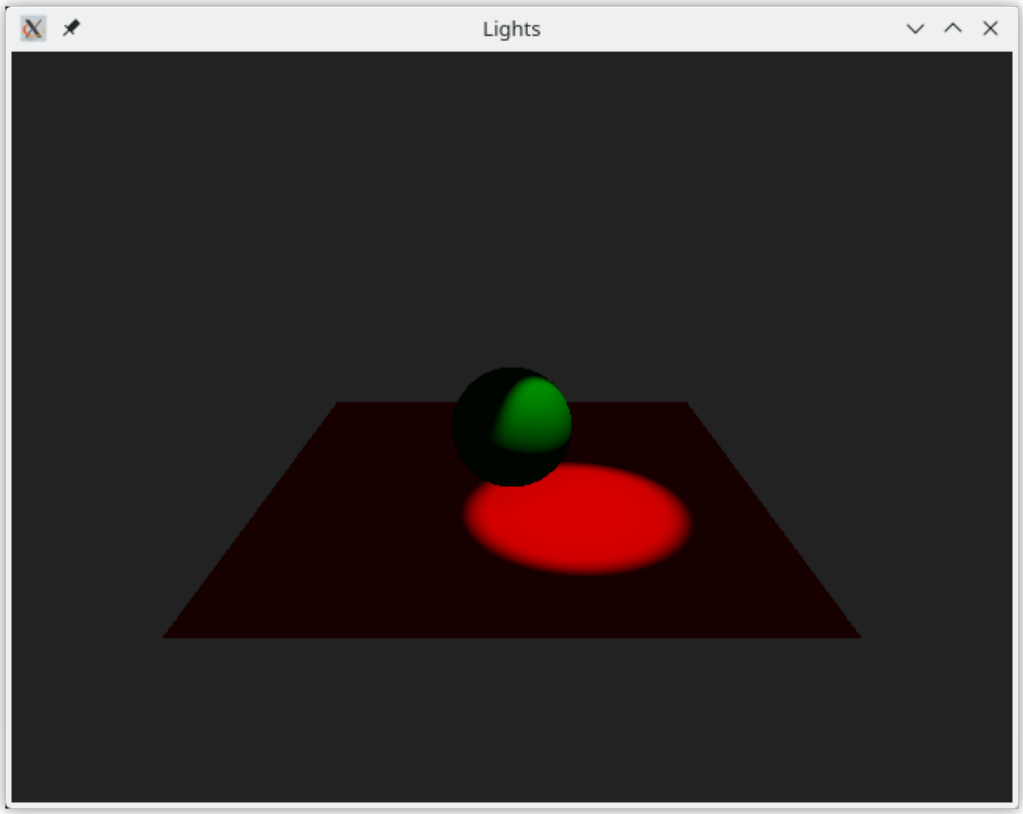
The next light source is the `PointLight`. It is a light that eminates from a given point in space and then falls off towards darkness based on the values of the `constantFade`, `linearFade`, and `quadraticFace` properties, where the light is calculated as `constantFade + distance * (linearFade * 0.01) + distance^2 * (quadraticFade * 0.0001)`. The default values are `1.0` constant and quadratic fade, and `0.0` for the linear fade, meaning that the light falls off according to the inverse square law.




```
PointLight {
    position: Qt.vector3d(100, 100, 150)

    castsShadow: true
}
```

The last of the light sources is the `SpotLight` which emits a cone of light in a given direction, much like a real world spotlight. The cone consists of an inner and an outer cone. The width of these is controlled by the `innerConeAngle` and `coneAngle`, specified in degrees between zero and 180 degrees.

The light in the inner cone behaves much like a `PointLight` and can be controlled using the `constantFade`, `linearFade`, and `quadraticFace` properties. In addition to this, the light fades towards darkness as it approaches the outer cone, controlled by the `coneAngle`.




```
SpotLight {
    position: Qt.vector3d(50, 200, 50)
    eulerRotation.x: -90

    brightness: 5
    ambientColor: Qt.rgba(0.1, 0.1, 0.1, 1.0)

    castsShadow: true
}
```

In addition to the `castsShadow` property, all lights also has the commonly used properties `color` and `brightness` which control the color and intensity of the light emitted. The lights also has an `ambientColor` property defining a base color to be applied to materials, before they are lit by the light source. This property is set to black by default, but can be used to provide a base lighting in a scene.

In the examples this far, we've only used one light source at a time, but it is of course possible to combine multiple light sources in a single scene.

Working with Assets

When working with 3D scenes, the built in meshes quickly grow old. Instead, a good flow from a modelling tool into QML is needed. Qt Quick 3D comes with the Balsam asset import tool, which is used to convert common asset formats into a Qt Quick 3D friendly format.

The purpose of Balsam is to make it easy to take assets created in common tools such as **Blender (https://www.blender.org/)** , Maya or 3ds Max and use them from Qt Quick 3D. Balsam supports the following asset types:

- COLLADA (`*.dae`)
- FBX (`*.fbx`)
- GLTF2 (`*.gltf` , `*.glb`)
- STL (`*.stl`)
- Wavefront (`*.obj`)

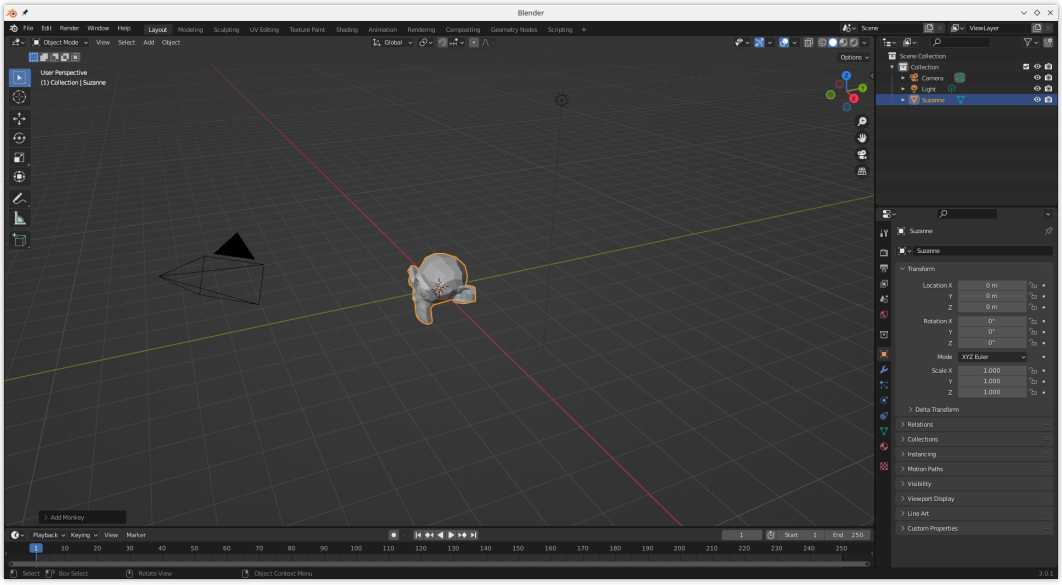
For some format, texture assets can also be exported into a Qt Quick 3D-friendly format, as long as Qt Quick 3D supports the given asset.

Blender

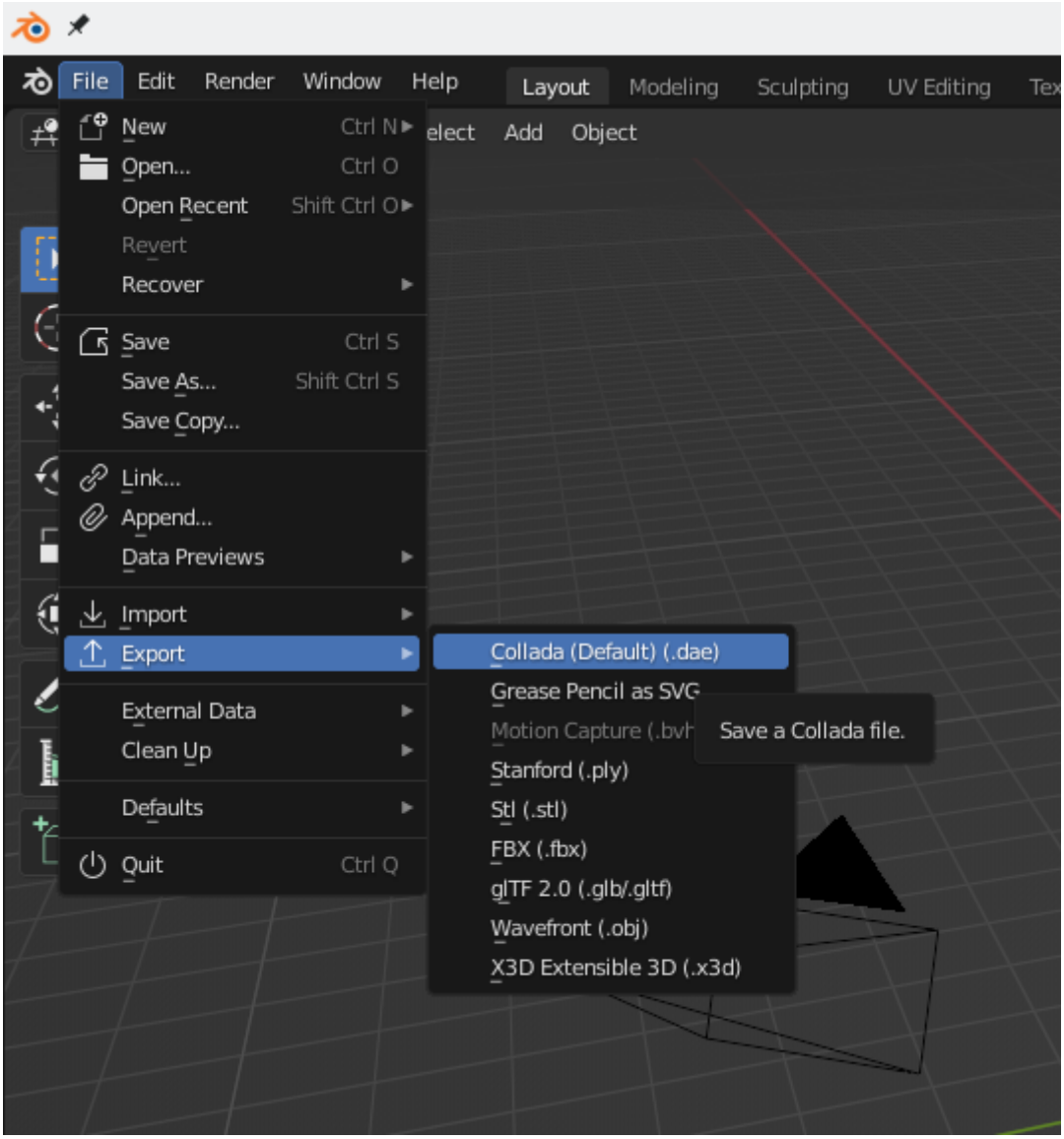
To generate an asset that we can import, we will use Blender to create a scene with a monkey head in it. We will then export this as a **COLLADA (https://en.wikipedia.org/wiki/COLLADA)** file to be able to convert it to a Qt Quick 3D friendly file format using Balsam.

Blender is available from **https://www.blender.org/ (https://www.blender.org/)** , and mastering Blender is a topic for another

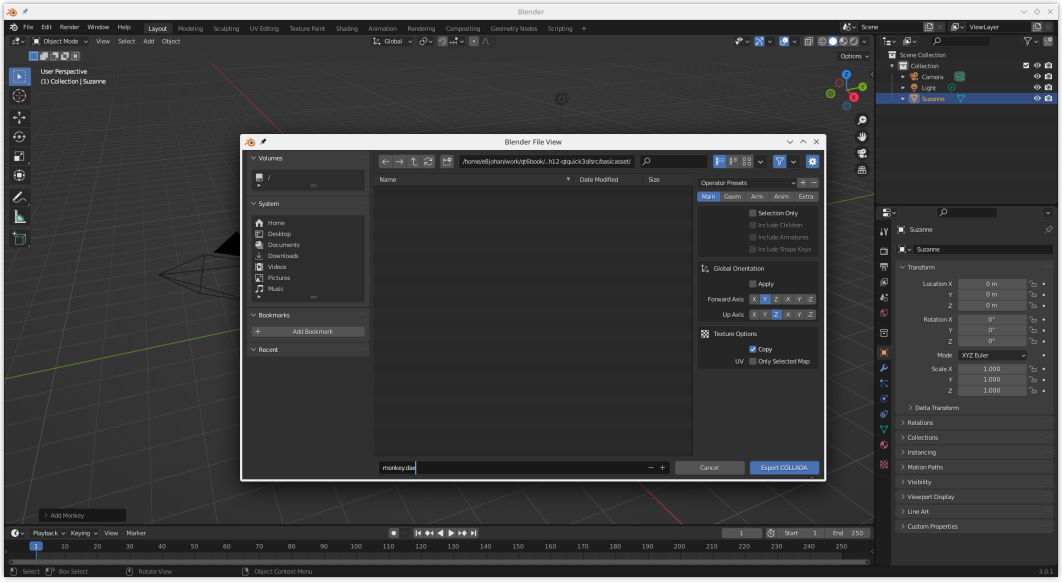
book, so we will do the most basic thing possible. Remove the original cube (select the cube with the left mouse button, press `shift + x`, select *Delete*), add a mesh (from the keyboard `shift + a`, select *Mesh*), and select to add a monkey (select *Monkey* from the list of available meshes). There are a number of video tutorials demonstrating how to do this. The resulting Blender user interface with the monkey head scene can be seen below.



Once the head is in place, go to File -> Export -> COLLADA.



This takes you to the Export COLLADA dialog where you can export the resulting scene.

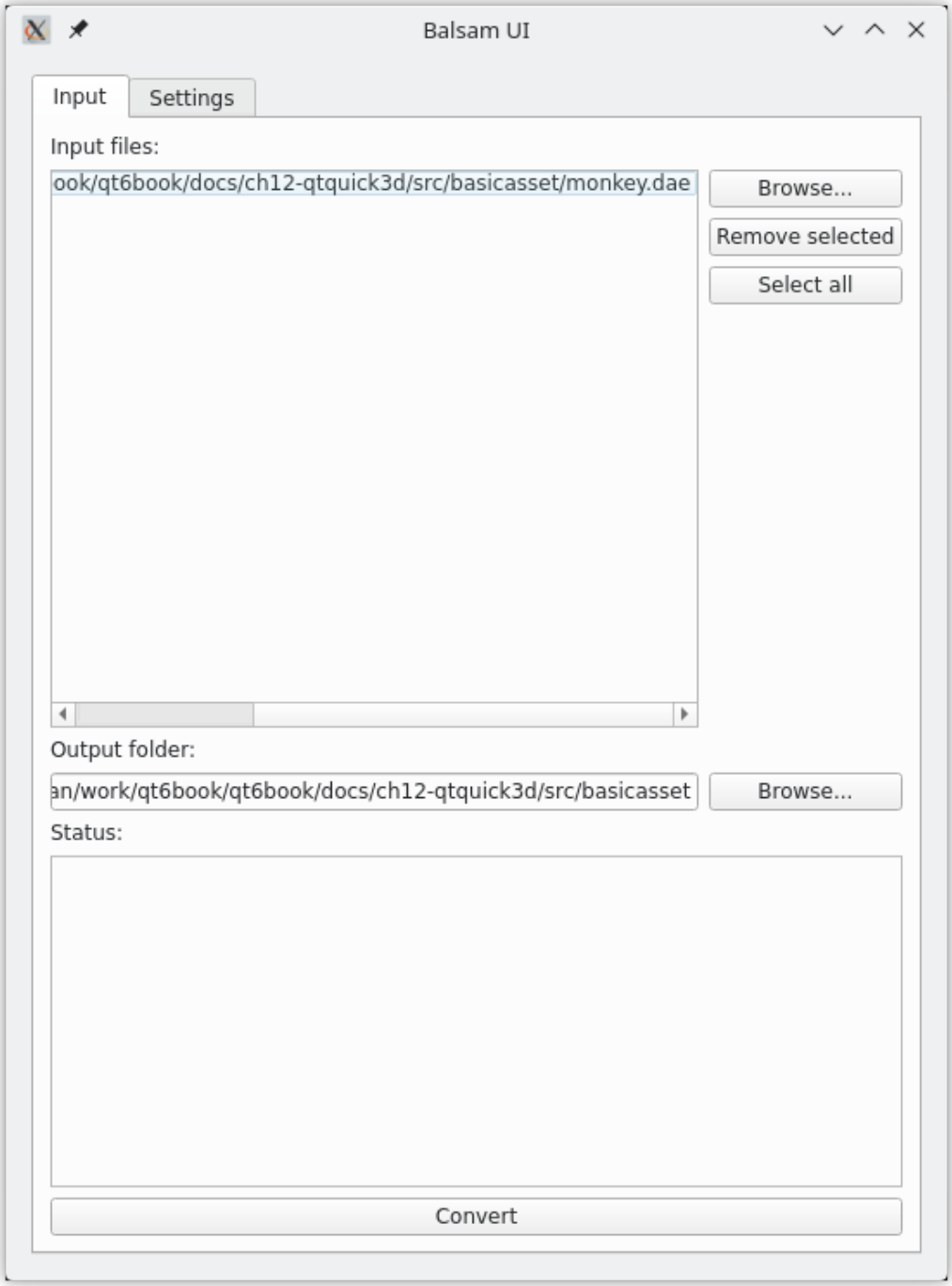


> **Tip**
>
> Both the blender scene and the exported COLLADA file (`*.dae`) can be found among the example files for the chapter.

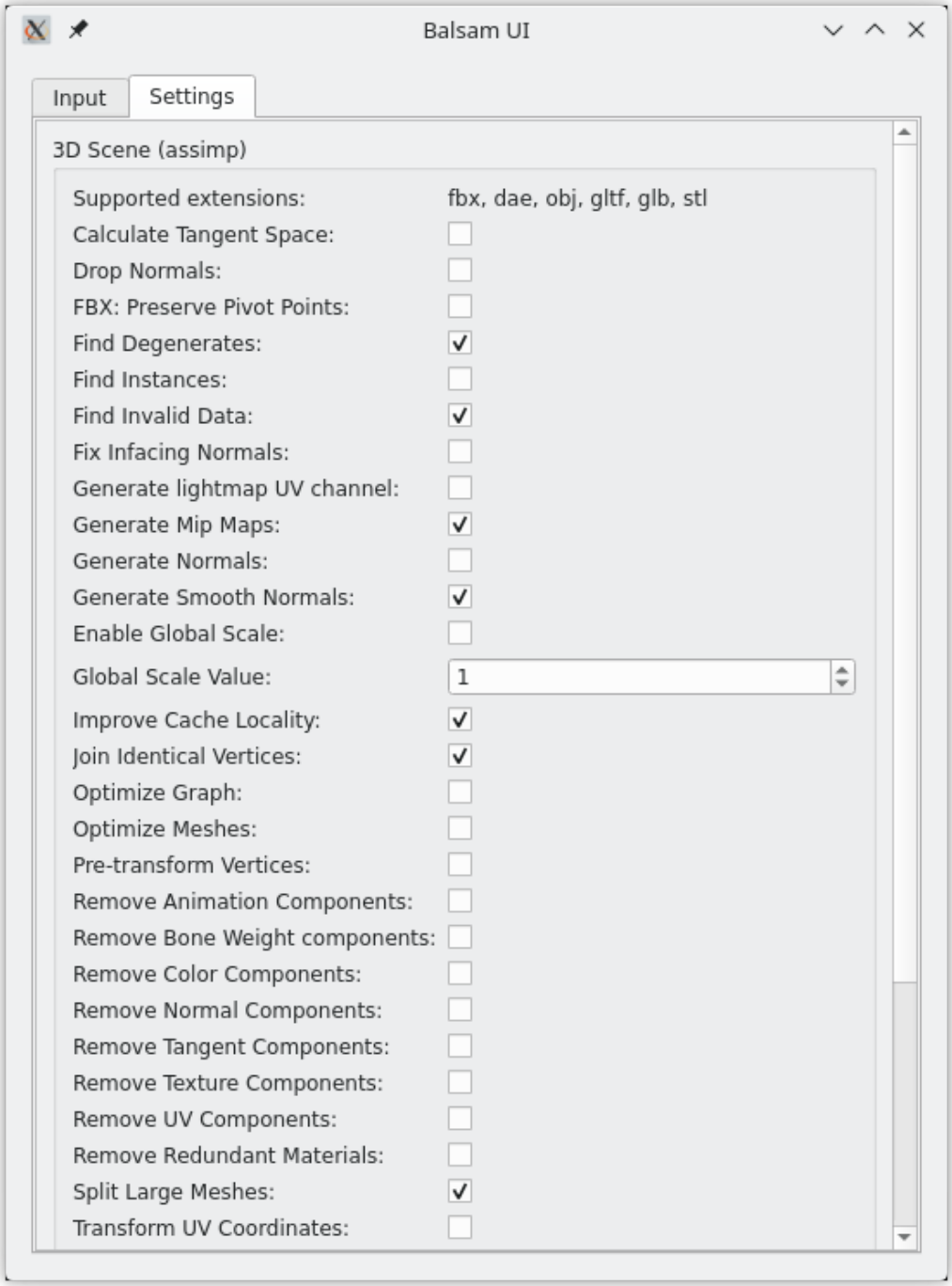
Balsam

Once the COLLADA file has been saved to disk, we can prepare it for use in Qt Quick 3D using Balsam. Balsam is available as a command line tool, or through a graphical user interface, using the `balsamui` tool. The graphical tool is just a thin layer on top of the command line tool, so there is no difference in what you can do with either tool.

We start by adding the `monkey.dae` file to the input files section, and set the output folder to a reasonable path.Your paths will most likely be different from the ones shown in the screenshot.



The *Settings* tab in the `balsamui` includes all the options. These all correspond to a command line option of the `balsam` tool. For now, we will leave all of them with their default values.



Now, go back to the *Input* tab and click *Convert*. This will result in the following output in the status section of the user interface:

```
Converting 1 files...
[1/1] Successfully converted '/home/.../src/basicasset/monkey.d
```

```
Successfully converted all files!
```

If you started `balsamui` from the command line, you will also see the following output there:

```
generated file:  "/home/.../src/basicasset/Monkey.qml"
generated file:  "/home/.../src/basicasset/meshes/suzanne.mesh"
```

This means that Balsam generated a `*.qml` file and a `*.mesh` file.

The Qt Quick 3D Assets

To use the files from a Qt Quick project we need to add them to the project. This is done in the `CMakeLists.txt` file, in the `qt_add_qml_module` macro. Add the files to the `QML_FILES` and `RESOURCES` sections as shown below.

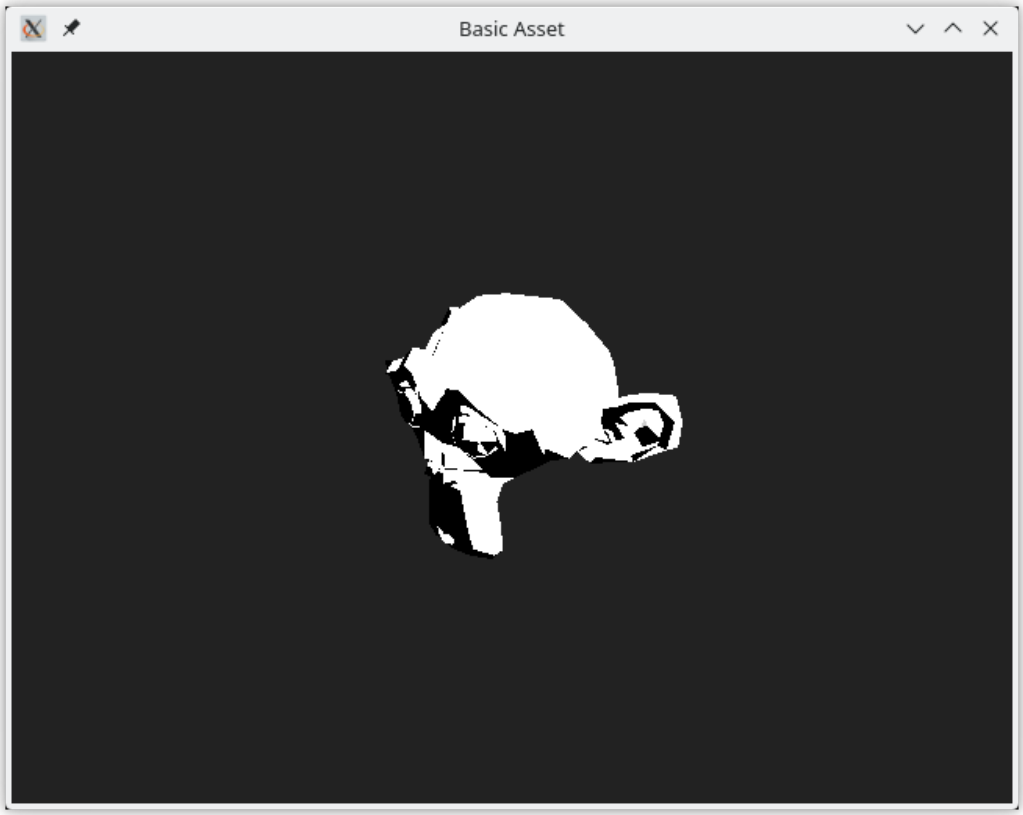
```
qt_add_qml_module(appbasicasset
    URI basicasset
    VERSION 1.0
    QML_FILES main.qml Monkey.qml
    RESOURCES meshes/suzanne.mesh
)
```

Having done this, we can populate a `View3D` with the `Monkey.qml` as shown below.

```
View3D {
    anchors.fill: parent

    environment: SceneEnvironment {
        clearColor: "#222222"
```

```
            backgroundMode: SceneEnvironment.Color
        }

        Monkey {}
    }
```

The `Monkey.qml` contains the entire Blender scene, including the camera and a light, so the result is a complete scene as shown below.



The interested reader is encouraged to explore the `Monkey.qml` file. As you will see, it contains a completely normal Qt Quick 3D scene built from the elements we already have used in this chapter.

> **Tip**
>
> As `Monkey.qml` is generated by a tool, do not modify the file manually. If you do, your changes will be overwritten if you ever have to regenerate the file using Balsam.

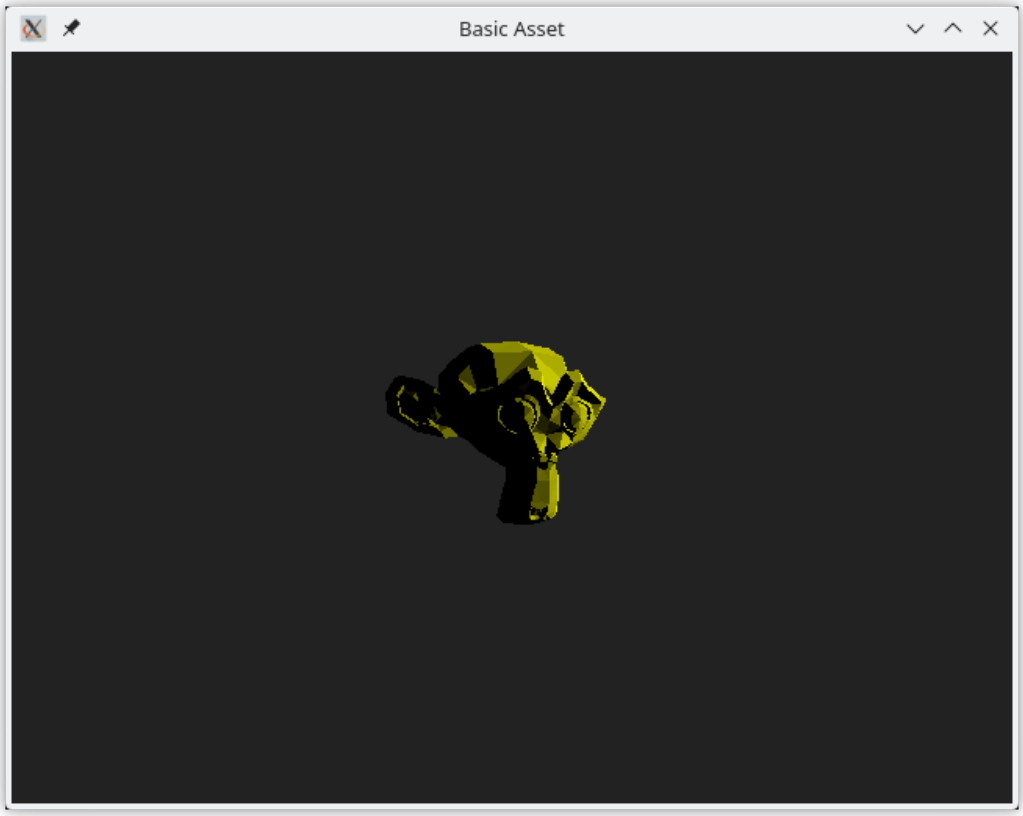
An alternative to using the entire scene from Blender is to use the `*.mesh` file in a Qt Quick 3D scene. This is demonstrated in the code below.

Here, we put a `DirectionalLight` and `PerspectiveCamera` into a `View3D`, and combine it with the mesh using a `Model` element. This way, we can control the positioning, scale, and lighting from QML. We also specify a different, yellow, material for the monkey head.

```
View3D {
    anchors.fill: parent

    environment: SceneEnvironment {
        clearColor: "#222222"
        backgroundMode: SceneEnvironment.Color
    }

    Model {
        source: "meshes/suzanne.mesh"

        scale: Qt.vector3d(2, 2, 2)

        eulerRotation.y: 30
        eulerRotation.x: -80

        materials: [ DefaultMaterial { diffuseColor: "yellow";
    }

    PerspectiveCamera {
        position: Qt.vector3d(0, 0, 15)
```

```
            Component.onCompleted: lookAt(Qt.vector3d(0, 0, 0))
        }

        DirectionalLight {
            eulerRotation.x: -20
            eulerRotation.y: 110

            castsShadow: true
        }
    }
```

The resulting view is shown below.



This demonstrates how a simple mesh can be exported from a 3D design tool such as blender, converted to a Qt Quick 3D format and then used from QML. One thing to think about is that we can import the entire scene as is, i.e. using `Monkey.qml`, or use only the assets, e.g. `suzanne.mesh`. This puts you in control of the trade-off between simple

importing of a scene, and added complexity while gaining flexibility by setting up the scene in QML.

Materials and Light

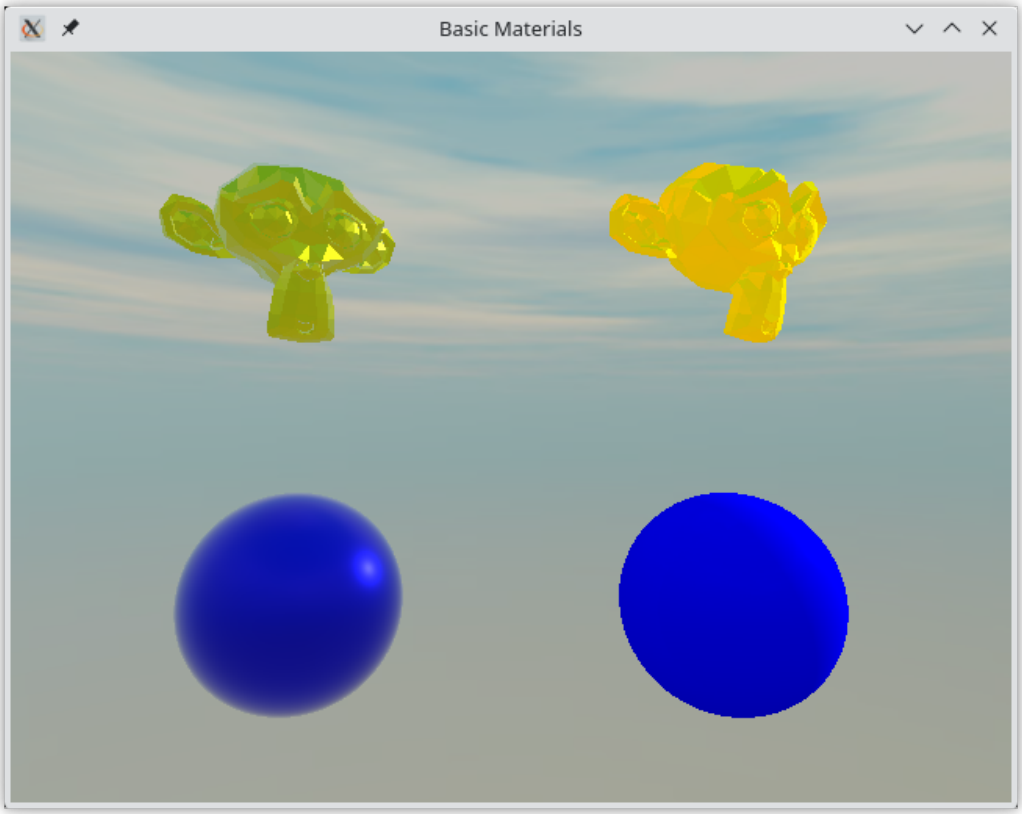
Up until now, we've only worked with basic materials. To create a convincing 3D scene, proper materials and more advanced lighting is needed. Qt Quick 3D supports a number of techniques to achieve this, and in this section we will look at a few of them.

The Built-in Materials

First up, we will look at the built in materials. Qt Quick 3D comes with three material types: `DefaultMaterial`, `PrincipledMaterial`, and `CustomMaterial`. In this chapter we will touch on the two first, while the latter allows you to create truly custom material by providing your own vertex and fragment shaders.

The `DefaultMaterial` lets you control the appearance of the material through the `specular`, `roughness`, and `diffuseColor` properties. The `PrincipledMaterial` lets you control the appearache through the `metalness`, `roughness`, and `baseColor` properties.

Examples of the two material types can be seen below, with the `PrincipledMaterial` to the left, and the `DefaultMaterial` to the right.



Comparing the two Suzannes, we can see how the two materials are set up.

For the `DefaultMaterial`, we use the `diffuseColor`, `specularTint`, and `specularAmount` properties. We will look at how variations of these properties affect the appearance of the objects later in this section.

```
Model {
    source: "meshes/suzanne.mesh"

    position: Qt.vector3d(5, 4, 0)
    scale: Qt.vector3d(2, 2, 2)
    rotation: Quaternion.fromEulerAngles(Qt.vector3d(-80, 30, 0

    materials: [ DefaultMaterial {
            diffuseColor: "yellow";
            specularTint: "red";
            specularAmount: 0.7
```

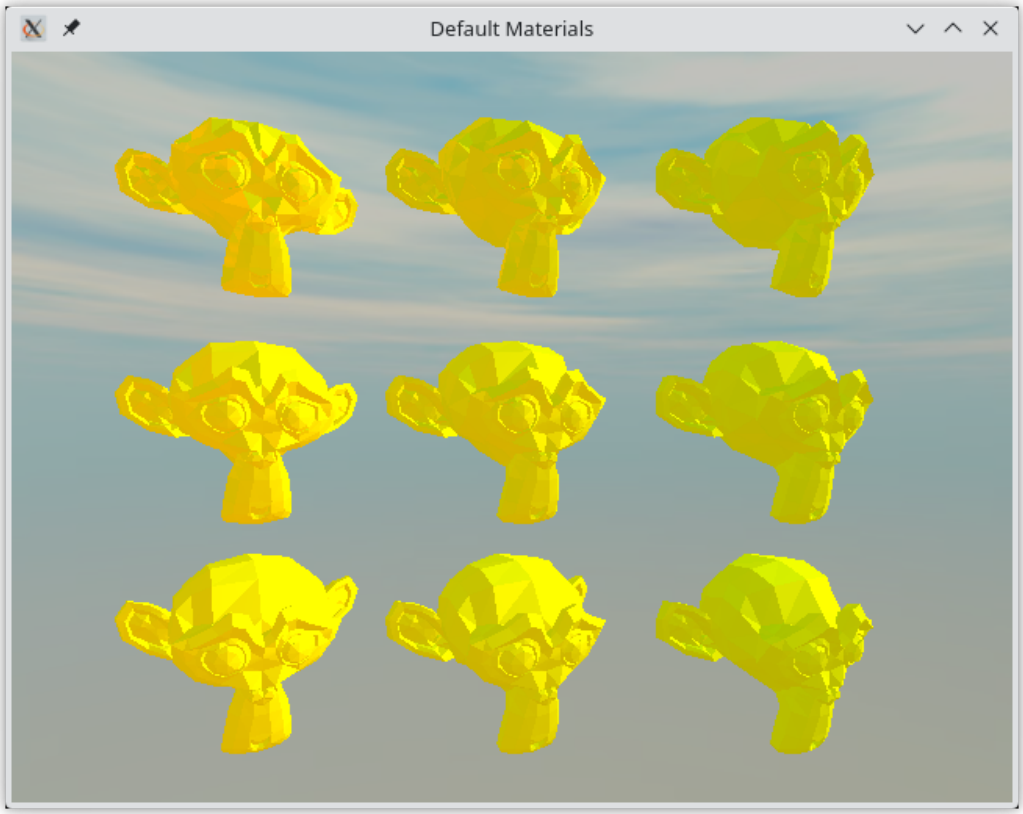
```
        } ]
}
```

For the `PrincipledMaterial`, we tune the `baseColor`, `metalness`, and `roughness` properties. Again, we will look at how variations of these properties affect the appearance later in this section.

```
Model {
    source: "meshes/suzanne.mesh"

    position: Qt.vector3d(-5, 4, 0)
    scale: Qt.vector3d(2, 2, 2)
    rotation: Quaternion.fromEulerAngles(Qt.vector3d(-80, 30, 0

    materials: [ PrincipledMaterial {
            baseColor: "yellow";
            metalness: 0.8
            roughness: 0.3
        } ]
}
```

Default Material Properties

The figure below shows the default material with various values for the `specularAmount` and the `specularRoughness` properties.



The `specularAmount` varies from `0.8` (left-most), through `0.5` (center), to `0.2` (right-most).

The `specularRoughness` varies from `0.0` (top), through `0.4` (middle), to `0.8` (bottom).

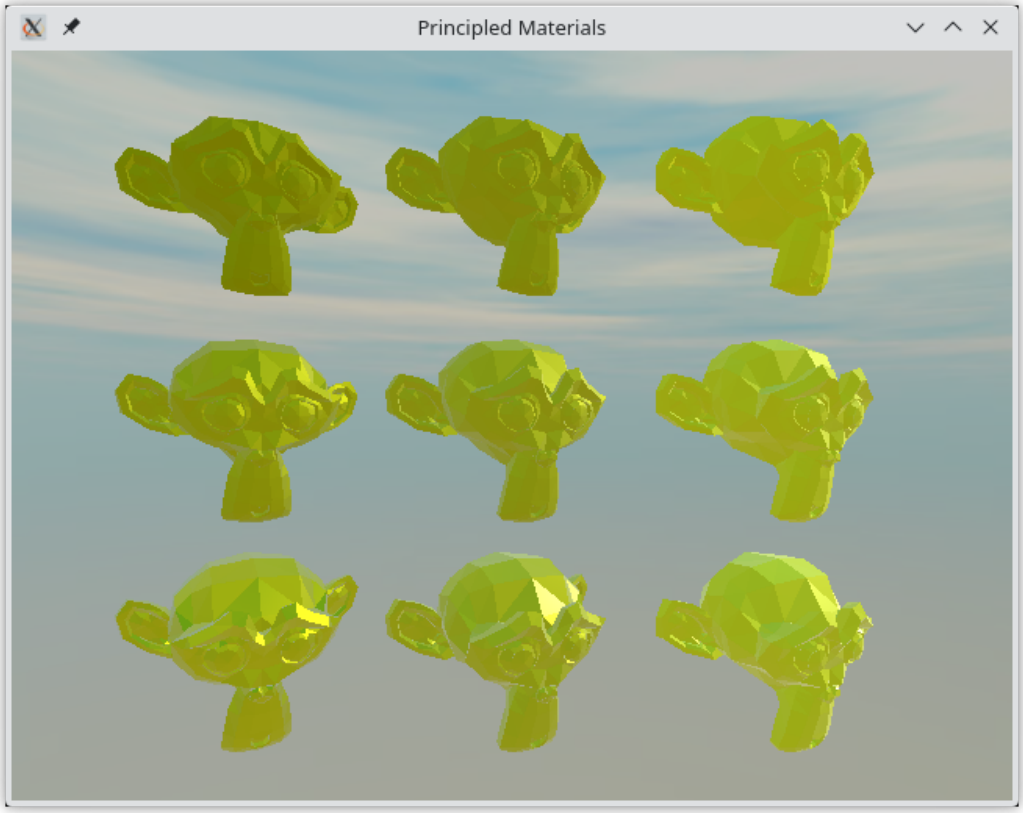
The code for the middle `Model` is shown below.

```
Model {
    source: "meshes/suzanne.mesh"

    position: Qt.vector3d(0, 0, 0)
    scale: Qt.vector3d(2, 2, 2)
    rotation: Quaternion.fromEulerAngles(Qt.vector3d(-80, 30, 0

    materials: [ DefaultMaterial {
            diffuseColor: "yellow";
            specularTint: "red";
```

```
            specularAmount: 0.5
            specularRoughness: 0.4
        } ]
}
```

Principled Material Properties

The figure below shows the principled material with various values for the `metalness` and `roughness` properties.



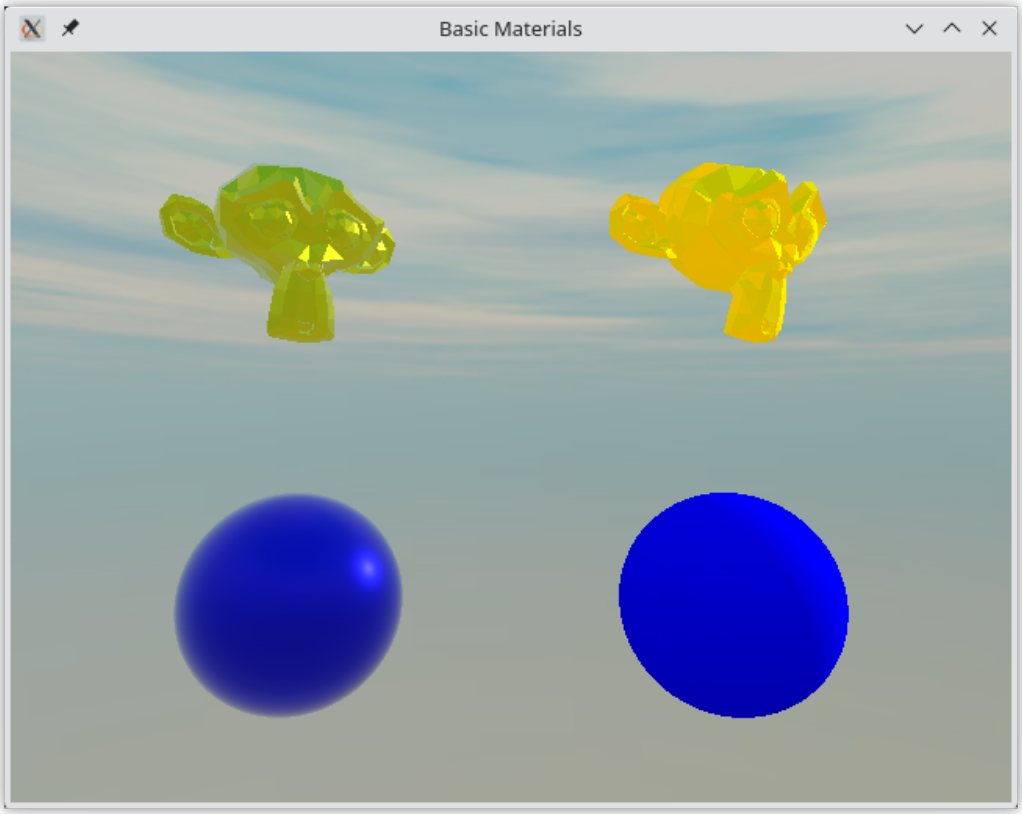
The `metalness` varies from `0.8` (left-most), through `0.5` (center), to `0.2` (right-most).

The `roughness` varies from `0.9` (top), through `0.6` (middle), to `0.3` (bottom).

```
Model {
    source: "meshes/suzanne.mesh"

    position: Qt.vector3d(0, 0, 0)
    scale: Qt.vector3d(2, 2, 2)
    rotation: Quaternion.fromEulerAngles(Qt.vector3d(-80, 30, 0

    materials: [ PrincipledMaterial {
            baseColor: "yellow";
            metalness: 0.5
            roughness: 0.6
        } ]
}
```

Image-based Lighting

One final detail in the main example in this section is the skybox. For this example, we are using an image as skybox, instead of a single colour background.



To provide a skybox, assign a `Texture` to the `lightProbe` property of the `SceneEnvironment` as shown in the code below. This means that the scene receives image-based light, i.e. that the skybox is used to light the scene. We also adjust the `probeExposure` which is used to control how much light is exposed through the probe, i.e. how brightly the scene will be lit. In this scene, we combine the light probe with a `DirectionalLight` for the final lighting.

```
environment: SceneEnvironment {
    clearColor: "#222222"
    backgroundMode: SceneEnvironment.SkyBox
    lightProbe: Texture {
        source: "maps/skybox.jpg"
    }
    probeExposure: 0.75
}
```

In addition to what we show, the orientation of the light probe can be adjusted using the `probeOrientation` vector, and the `probeHorizon` property can be used to darken the bottom half of the environment, simulating that the light comes from above, i.e. from the sky, rather than from all around.

Animations

There are multiple ways to add animations to Qt Quick 3D scenes. The most basic one is to move, rotate, and scale `Model` elements in the scene. However, in many cases, we want to modify the actual meshes. There are two basic approaches to this: morphing animations and skeletal animations.

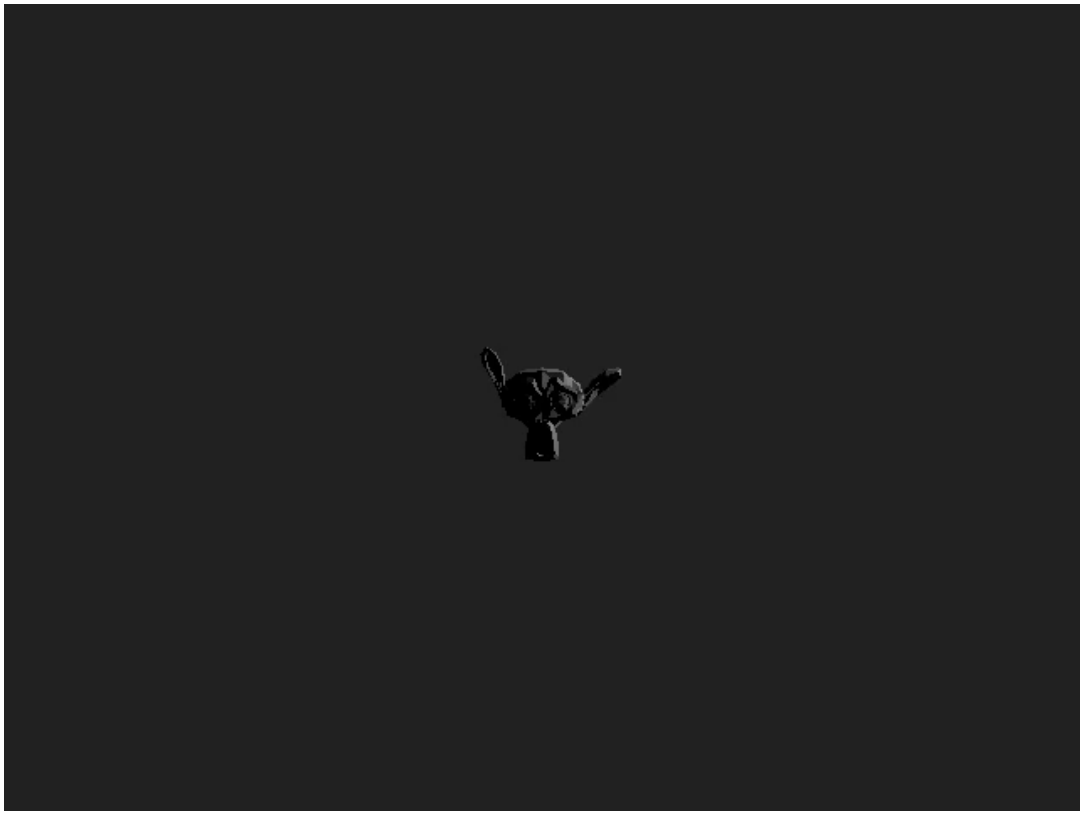
Morphing animations lets you create a number of target shapes to which various weights can be assigned. By combining the target shapes based on the weights, a deformed, i.e. morphed, shape is produced. This is commonly used to simulate deformations of soft materials.

Skeletal animations is used to pose an object, such as a body, based on the positions of a skeleton made of bones. These bones will affect the body, thus deform it into the pose required.

For both types of animations, the most common approach is to define the morphing target shapes and bones in a modelling tool, and then export it to QML using the *Balsam* tool. In this chapter we will do just this for a skeletal animation, but the approach is similar for a morphing animation.

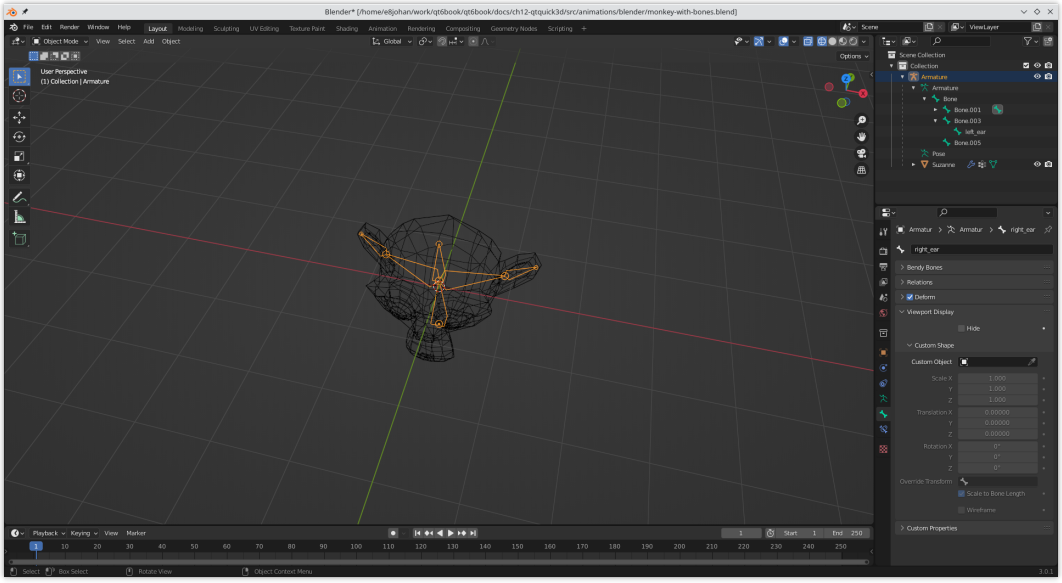
Skeletal Animations

The goal of this example is to make Suzanne, the Blender monkey head, wave with one of her ears.



Skeletal animation is sometimes known as vertex skinning. Basically, a skeleton is put inside of a mesh and the vertexes of the mesh are bound to the skeleton. This way, when moving the skeleton, the mesh is deformed into various poses.

As teaching Blender is beyond the scope of this book, the keywords you are looking for are posing and armatures. Armatures are what Blender calls the bones. There are plenty of video tutorials available explaining how to do this. The screenshot below shows the scene with Suzanne and the armatures in Blender. Notice that the ear armatures have been named, so that we can identify them from QML.



Once the Blender scene is done, we export it as a COLLADA file and convert it to a QML and a mesh, just as in the *Working with Assets* section. The resulting QML file is called `Monkey_with_bones.qml`.

We then have to refer to the files in our `qt_add_qml_module` statement in the `CMakeLists.txt` file:

```
qt_add_qml_module(appanimations
    URI animations
    VERSION 1.0
    QML_FILES main.qml Monkey_with_bones.qml
    RESOURCES meshes/suzanne.mesh
)
```

Exploring the generated QML, we notice that the skeleton is built up from QML elements of the types `Skeleton` and `Joint`. It is possible to work with these elements as code in QML, but it is much more common to create them in a design tool.

```
Node {
    id: armature
    z: -0.874189
```

```
    Skeleton {
        id: qmlskeleton
        Joint {
            id: armature_Bone
            rotation: Qt.quaternion(0.707107, 0.707107, 0, 0)
            index: 0
            skeletonRoot: qmlskeleton
            Joint {
                id: armature_Bone_001
                y: 1
```

The `Skeleton` element is then referred to by the `skeleton` property of the `Model` element, before the `inverseBindPoses` property, linking the joints to the model.

```
Model {
    id: suzanne
    skeleton: qmlskeleton
    inverseBindPoses: [
        Qt.matrix4x4(1, 0, 0, 0, 0, 0, 1, 0.748378, 0, -1, 0, 0
        Qt.matrix4x4(),
        Qt.matrix4x4(0.283576, -0.11343, 0.952218, 1.00072, -0.
        Qt.matrix4x4(),
        Qt.matrix4x4(0.311833, 0.101945, -0.944652, -1.01739, 0
        Qt.matrix4x4()
    ]
    source: "meshes/suzanne.mesh"
```

The next step is to include the newly created `Monkey_with_bones` element into our main `View3D` scene:

```
View3D {
    anchors.fill: parent

    Monkey_with_bones {
```

```
        id: monkey
    }
```

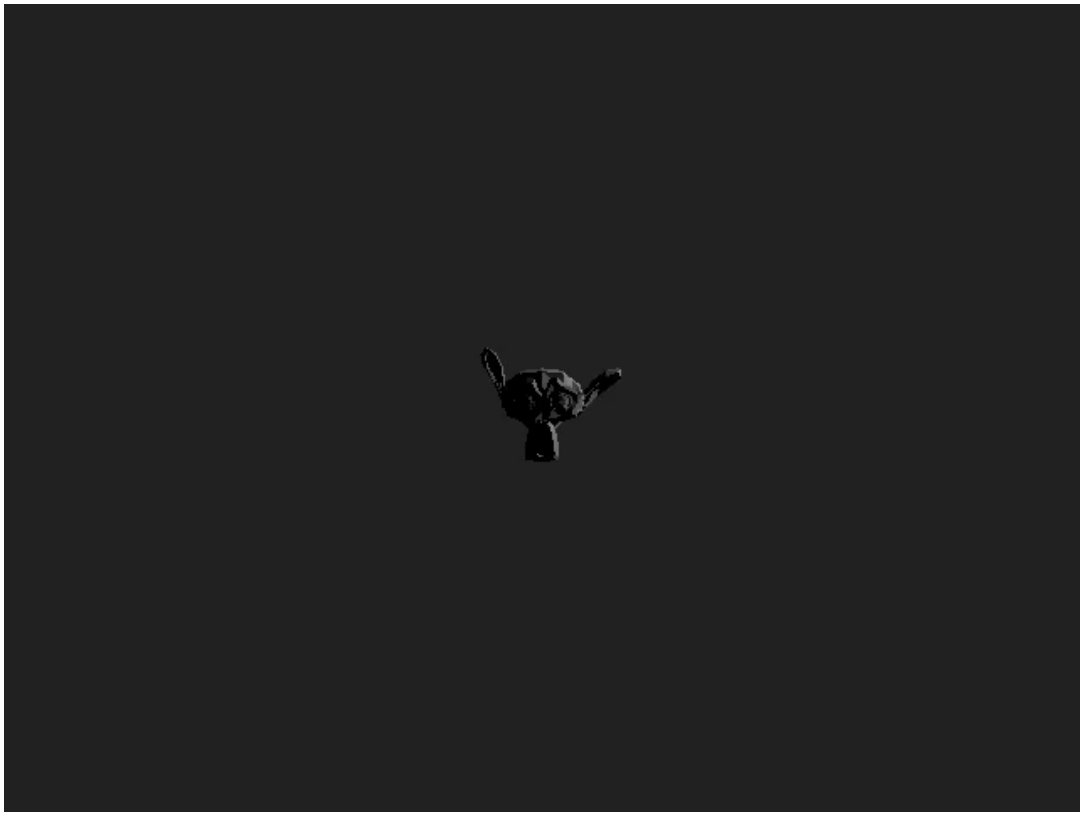
And then we create a `SequentialAnimation` built from two `NumberAnimations` to make the ear flop forth and back.

```
SequentialAnimation {
    NumberAnimation {
        target: monkey
        property: "left_ear_euler"
        duration: 1000
        from: -30
        to: 60
        easing: InOutQuad
    }
    NumberAnimation {
        target: monkey
        property: "left_ear_euler"
        duration: 1000
        from: 60
        to: -30
        easing: InOutQuad
    }
    loops: Animation.Infinite
    running: true
}
```

> **Caveat**
>
> In order to be able to access the `Joint`'s `eulerRotation.y` from outside of the `Monkey_with_bones` file, we need to expose it as a top level property alias. This means modifying a generated file, which is not very nice, but it solves the problem.

```
Node {
    id: scene

    property alias left_ear_euler: armature_left_ear.eul
    property alias right_ear_euler: armature_right_ear.e
```

The resulting floppy ear can be enjoyed below:



As you can see, it is convenient to import and use skeletons created in a design tool. This makes it convenient to animate complex 3D models from

QML.

Mixing 2D and 3D Contents

Qt Quick 3D has been built to integrate nicely into the traditional Qt Quick used to build dynamic 2D contents.

3D Contents in a 2D Scene

It is straight forward to mix 3D contents into a 2D scene, as the `View3D` element represents a 2D surface in the Qt Quick scene.

There are a couple of properties that can be of interest when combining 3D contents into a 2D scene this way.

First, the `renderMode` of `View3D`, which lets you control if the 3D contents is rendered behind, in-front-of, or inline with the 2D contents. It can also be rendered on an off-screen buffer which then is combined with the 2D scene.

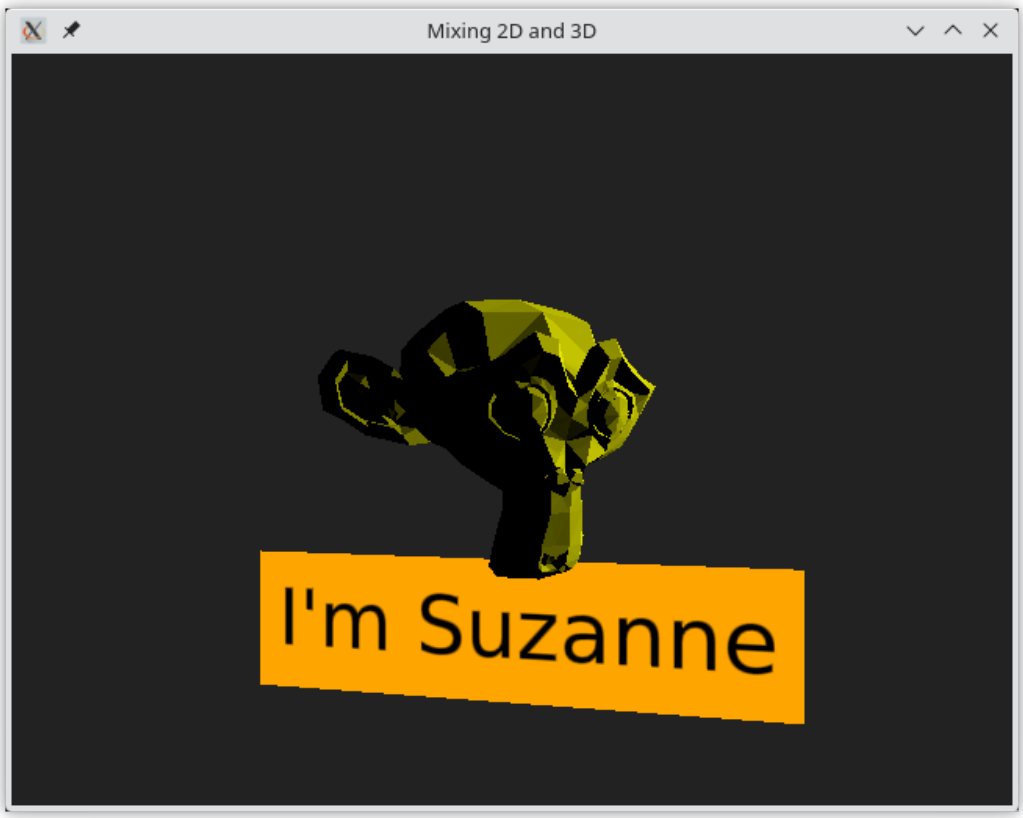
The other property is the `backgroundMode` of the `SceneEnvironment` bound to the `environment` property of the `View3D`. We've seen it set to `Color` or `SkyBox`, but it can also be set to `Transparent` which lets you see any 2D contents behind the `View3D` through the 3D scene.

When building a combined 2D and 3D scene, it is also good to know that it is possible to combine multiple `View3D` elements in a single Qt Quick scene. For instance, if you want to have multiple 3D models in a single scene, but they are separate parts of the 2D interface, then you can put them in separate `View3D` elements and handle the layout from the 2D Qt Quick side.

2D Contents in a 3D Scene

To put 2D contents into a 3D scene, it needs to be placed on a 3D surface. The Qt Quick 3D `Texture` element has a `sourceItem` property that allows you to integrate a 2D Qt Quick scene as a texture for an arbitrary 3D surface.

Another apporoach is to put the 2D Qt Quick elements directly into the scene. This is the approach used in the example below where we provide a name badge for Suzanne.



What we do here is that we instantiate a `Node` that serves as an anchor point in the 3D scene. We then place a `Rectangle` and a `Text` element inside the `Node` . These two are 2D Qt Quick elements. We can then

control the 3D position, rotation, and scale through the corresponding properties of the `Node` element.

```
Node {
    y: -30
    eulerRotation.y: -10
    Rectangle {
        anchors.horizontalCenter: parent.horizontalCenter
        color: "orange"
        width: text.width+10
        height: text.height+10
        Text {
            anchors.centerIn: parent
            id: text
            text: "I'm Suzanne"
            font.pointSize: 14
            color: "black"
        }
    }
}
```

Summary

Qt Quick 3D offers a rich way of integrating 3D contents into a Qt Quick scene, allowing a tight integration through QML.

When working with 3D contents, the most common approach is to work with assets created in other tools such as Blender, Maya, or 3ds Max. Using the *Balsam* tool it is possible to import meshes, materials, as well as animation skeletons, from these models into QML. This can then be used to render, as well as interacting with the models.

QML is still used to setup the scene, as well as instantiating models. This means that a scene can be built in an external tool, or be instantiated dynamically from QML using elements created using external tool. In the most basic cases, scenes can also be created from the built in meshed that come with Qt Quick 3D.

By allowing the tight integration of Qt Quick's 2D contents, and Qt Quick 3D, it is possible to create modern and intuit user interfaces. With QML's ability to bind C++ properties to QML properties, this makes it easy to connect 3D model state to underlying C++ state.

In this chapter we've only scratched the surface of what is possible using Qt Quick 3D. There are more advanced concepts ranging from custom filters and shaders, to generating meshes dynamically from C++. There is also a large set of optimization techniques that can be used to ensure good rendering performance of complex 3D contents. You can read more about this in the **Qt Quick 3D Reference Documentation (https://doc.qt.io/qt-6/qtquick3d-index.html)** .

Networking

Qt 6 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 exist to manage proxies, network cache and also the systems network configuration.

This chapter will not be about C++ networking, this chapter is about Qt Quick and networking. So how can I connect my QML/JS user interface directly with a network service or how can I serve my user interface via a network service? There are good books and references out there to cover network programming with Qt/C++. Then it is just a manner to read the chapter about C++ integration to come up with an integration layer to feed your data into the Qt Quick world.

Serving UI via HTTP

To load a simple user interface via HTTP we need to have a web-server, which serves the UI documents. We start off 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 `RemoteComponent.qml` file in our project folder and create a red rectangle inside.

```
// RemoteComponent.qml
import QtQuick

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'
}
```

To serve this file we can start a small python script:

```sh
cd <PROJECT>
python -m http.server 8080
```

Now our file should be reachable via `http://localhost:8080/RemoteComponent.qml`. You can test it with:

```sh
curl http://localhost:8080/RemoteComponent.qml
```

Or just point your browser to the location. Unfortunately, your browser does not understand QML and will not be able to render the document. Fortunately, a QML web browser does exist. It's called **Canonic (https://www.canonic.com)** . Canonic is itself built with QML and runs in your web browser via WebAssembly. However, if you are using the WebAssembly version of Canonic, you won't be able to view files served from `localhost` ; in a bit, you'll see how to make your QML files available to use with the WebAssembly version of Canonic. If you want, you can download the code to run Canonic as an app on your desktop, but there are security concerns related to doing so (see **here (https://docs.page/canonic/canonic)** and **here (https://doc.qt.io/qt-6/qtqml-documents-networktransparency.html#implications-for-application-security)** for more details).

Furthermore, Qt 6 provides the `qml` binary, which can be used like a web browser. You can directly load a remote QML document by using the following command:

```sh
qml http://localhost:8080/RemoteComponent.qml
```

Sweet and simple.

> **TIP**
>
> If the `qml` program is not in your path, you can find it in the Qt binaries: `<qt-install-path>/<qt-version>/<your-os>/bin/qml` .

Another way of importing a remote QML document is to dynamically load it using QML ! For this, we use a `Loader` element to retrieve for us the remote document.

```
// LocalHostExample.qml
import QtQuick

Loader {
    id: root
    source: 'http://localhost:8080/RemoteComponent.qml'
    onLoaded: {
        root.width = root.item.width  // qmllint disable
        root.height = root.item.height  // qmllint disable
    }
}
```

Now we can ask the `qml` executable to load the local `LocalHostExample.qml` loader document.

```sh
qml LocalHostExample.qml
```

TIP

If you do not want to run a local server you can also use the gist service from GitHub. The gist is a clipboard like online services like Pastebin and others. It is available under **https://gist.github.com (https://gist.github.com)** . For this example, I created a small gist under the URL **https://gist.github.com/jryannel/7983492 (https://gist.github.com/jryannel/7983492)** . This will reveal a green rectangle. As the gist URL will provide the website as HTML code we need to attach a `/raw` to the URL to retrieve the raw file and not the HTML code.

Since this content is hosted on a web server with a public web address, you can now use the web-based version of Canonic to view it. To do so, simply point your web browser to **https://app.canonic.com/#http://gist.github.com/jryannel/7983492 (https://app.canonic.com/#http://gist.github.com/jryannel/7983492)** . Of course, you'll need to change the part after the `#` to view your own files.

```
// GistExample.qml
import QtQuick

Loader {
    id: root
    source: 'https://gist.github.com/jryannel/7983492/raw'
    onLoaded: {
        root.width = root.item.width  // qmllint disable
        root.height = root.item.height  // qmllint disable
    }
}
```

To load another file over the network from `RemoteComponent.qml`, you will need to create a dedicated `qmldir` file in the same directory on the server. Once done, you will be able to reference the component by its name.

Networked Components

Let us create a small experiment. We add to our remote side a small button as a reusable component.

Here's the directory structure that we will use:

```sh
./src/SimpleExample.qml
./src/remote/qmldir
./src/remote/Button.qml
./src/remote/RemoteComponent.qml
```

Our `SimpleExample.qml` is the same as our previous `main.qml` example:

```
import QtQuick

Loader {
    id: root
    anchors.fill: parent
    source: 'http://localhost:8080/RemoteComponent.qml'
    onLoaded: {
        root.width = root.item.width  // qmllint disable
        root.height = root.item.height  // qmllint disable
    }
}
```

In the `remote` directory, we will update the `RemoteComponent.qml` file so that it uses a custom `Button` component:

```
// remote/RemoteComponent.qml
import QtQuick

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Button {
        anchors.centerIn: parent
        text: qsTr('Click Me')
        onClicked: Qt.quit()
    }
}
```

As our components are hosted remotely, the QML engine needs to know what other components are available remotely. To do so, we define the content of our remote directory within a `qmldir` file:

```
# qmldir
Button 1.0 Button.qml
```

And finally we will create our dummy `Button.qml` file:

```
// remote/Button.qml
import QtQuick.Controls

Button {

}
```

We can now launch our web-server (keep in mind that we now have a `remote` subdirectory):

```sh
cd src/serve-qml-networked-components/
python -m http.server --directory ./remote 8080
```

And remote QML loader:

```sh
qml SimpleExample.qml
```

Importing a QML components directory

By defining a `qmldir` file, it's also possible to directly import a library of components from a remote repository. To do so, a classical import works:

```
import QtQuick
import "http://localhost:8080" as Remote

Rectangle {
    width: 320
    height: 320
    color: 'blue'

    Remote.Button {
        anchors.centerIn: parent
        text: qsTr('Quit')
        onClicked: Qt.quit()
    }
}
```

TIP

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.

WARNING

Be very cautious about loading QML components from the Internet. By doing so, you introduce the risk of accidentally downloading malicious components that will do evil things to your computer. These security risks have been **documented (https://doc.qt.io/qt-6/qtqml-documents-networktransparency.html#implications-for-application-security)** by Qt. The Qt page was already linked to on this page, but the warning is worth repeating.

Templates

When working with HTML projects they often use template driven development. A small HTML stub is expanded on the server side with code generated by the server using a template mechanism. For example, for a photo list, the list header would be coded in HTML and the dynamic image list would be dynamically generated using a template mechanism. In general, this can also be done using QML but there are some issues with it.

First, it is not necessary. The reason HTML developers are doing this is to overcome limitations on the HTML backend. There is no component model yet in HTML so dynamic aspects have to be covered using these mechanisms 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 (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 milliseconds or a random number.

```
Loader {
    source: 'http://localhost:8080/main.qml#' + new Date().getT
}
```

In summary templating is possible but not really recommended and does not play to the strength of QML. A better approach is to use web-services which serve JSON or XML data.

HTTP Requests

An HTTP request is in Qt typically done using `QNetworkRequest` and `QNetworkReply` from the c++ site and then the response would be pushed using the Qt/C++ integration into the QML space. So we try to push the envelope here a little bit to use the current tools Qt Quick gives us to communicate with a network endpoint. For this, we use a helper object to make an HTTP request, response cycle. It comes in the form of the javascript `XMLHttpRequest` object.

The `XMLHttpRequest` object allows the user to register a response handler function and a URL. A request can be sent using one of the HTTP verbs (get, post, put, delete, …) to make the request. When the response arrives the handler function is called. The handler 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:

```js
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)` .

```js
/* ... */
} 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).

> **TIP**
>
> 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 newly 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.

> **TIP**
>
> 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:

```sh
curl "http://api.flickr.com/services/feeds/photos_public.gne?fo
```

The response will be something like this:

```json
{
    "title": "Recent Uploads tagged munich",
    ...
    "items": [
        {
        "title": "Candle lit dinner in Munich",
        "media": {"m":"http://farm8.staticflickr.com/7313/11444
        ...
        },{
        "title": "Munich after sunset: a train full of \"must h
        "media": {"m":"http://farm8.staticflickr.com/7394/11443
        ...
        }
    ]
    ...
}
```

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
// 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() {
    const xhr = new XMLHttpRequest()
    xhr.onreadystatechange = function() {
        if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED)
            print('HEADERS_RECEIVED')
        } else if(xhr.readyState === XMLHttpRequest.DONE) {
            print('DONE')
            const response = JSON.parse(xhr.responseText.toStri
            // Set JS object as model for listview
            view.model = response.items
        }
    }
    xhr.open("GET", "http://api.flickr.com/services/feeds/photo
    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 the model for our simple list view.

```
import QtQuick

Rectangle {
    id: root

    width: 320
    height: 480

    ListView {
        id: view
        anchors.fill: parent
        delegate: Thumbnail {
            required property var modelData
            width: view.width
            text: modelData.title
            iconSource: modelData.media.m
        }
    }

    function request() {
        const xhr = new XMLHttpRequest()
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEI
                print('HEADERS_RECEIVED')
            } else if(xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE')
                const response = JSON.parse(xhr.responseText.to
                // Set JS object as model for listview
                view.model = response.items
            }
        }
        xhr.open("GET", "http://api.flickr.com/services/feeds/p
        xhr.send()
    }
```

```
        Component.onCompleted: {
            root.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.

The other option would be to have a placeholder `ListModel` and append each item onto the list model. To support larger models it is required to support pagination (e.g. page 1 of 10) and lazy content retrieval.

Local files

Is it also possible to load local (XML/JSON) files using the XMLHttpRequest. For example a local file named “colors.json” can be loaded using:

```js
xhr.open("GET", "colors.json")
```

We use this to read a color table and display it as a grid. It is not possible to modify the file from the Qt Quick side. To store data back to the source we would need a small REST based HTTP server or a native Qt Quick extension for file access.

```
import QtQuick

Rectangle {
    width: 360
    height: 360
    color: '#000'

    GridView {
        id: view
        anchors.fill: parent
        cellWidth: width / 4
        cellHeight: cellWidth
        delegate: Rectangle {
            required property var modelData
            width: view.cellWidth
            height: view.cellHeight
            color: modelData.value
        }
```

```
    }

    function request() {
        const xhr = new XMLHttpRequest()
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEI
                print('HEADERS_RECEIVED')
            } else if(xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE')
                const response = JSON.parse(xhr.responseText.to
                view.model = response.colors
            }
        }
        xhr.open("GET", "colors.json")
        xhr.send()
    }

    Component.onCompleted: {
        request()
    }
}
```

TIP

By default, using GET on a local file is disabled by the QML engine. To overcome this limitation, you can set the `QML_XHR_ALLOW_FILE_READ` environment variable to `1`:

```sh
QML_XHR_ALLOW_FILE_READ=1 qml localfiles.qml
```

The issue is when allowing a QML application to read local files through an `XMLHttpRequest`, hence `XHR`, this opens up the entire file system for reading, which is a potential security issue. Qt will allow you to read local files only if the environment variable is set, so that this is a concious decision.

Instead of using the `XMLHttpRequest` it is also possible to use the XmlListModel to access local files.

```
import QtQuick
import QtQml.XmlListModel

Rectangle {
    width: 360
    height: 360
    color: '#000'

    GridView {
        id: view
        anchors.fill: parent
        cellWidth: width / 4
        cellHeight: cellWidth
        model: xmlModel
        delegate: Rectangle {
            id: delegate
            required property var model
            width: view.cellWidth
            height: view.cellHeight
            color: model.value
            Text {
                anchors.centerIn: parent
                text: delegate.model.name
            }
        }
    }

    XmlListModel {
        id: xmlModel
        source: "colors.xml"
        query: "/colors/color"
        XmlListModelRole { name: 'name'; elementName: 'name' }
        XmlListModelRole { name: 'value'; elementName: 'value'
    }
}
```

With the XmlListModel it is only possible to read XML files and not JSON files.

REST API

To use a web-service, we first need to create one. We will use Flask (**https://flask.palletsprojects.com** **(https://flask.palletsprojects.com)**) 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** **(https://flask.palletsprojects.com/en/2.0.x/quickstart/)** documentation.

py

```
from flask import Flask, jsonify, request
import json

with open('colors.json', 'r') as file:
    colors = json.load(file)

app = Flask(__name__)
```

py

```
# Services registration & implementation...
```

py

```
if __name__ == '__main__':
    app.run(debug = True)
```

When you run this script, it will provide a web-server at **http://localhost:5000 (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.

```py
@app.route('/colors', methods = ['GET'])
def get_colors():
    return jsonify({ "data" : colors })
```

This will return the colors under the '/colors' endpoint. To test this we can use curl to create an HTTP request.

```sh
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.

```py
@app.route('/colors/<name>', methods = ['GET'])
def get_color(name):
```

```
    for color in colors:
        if color["name"] == name:
            return jsonify(color)
    return jsonify({ 'error' : True })
```

And we can test it with using curl again. For example to get the red color entry.

```sh
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.

```py
@app.route('/colors', methods= ['POST'])
def create_color():
    print('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.

```sh
curl -i -H "Content-Type: application/json" -X POST -d '{"name"
```

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.

```py
@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['v
            return jsonify(color)
    return jsonify({ 'error' : True })
```

In the curl request, we only provide the values to be updated as JSON data and then a named endpoint to identify the color to be updated.

```sh
curl -i -H "Content-Type: application/json" -X PUT -d '{"value"
```

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.

```py
@app.route('/colors/<name>', methods=['DELETE'])
def delete_color(name):
    for color in colors:
        if color["name"] == name:
```

```
            colors.remove(color)
            return jsonify(color)
    return jsonify({ 'error' : True })
```

This request looks similar to the GET request for an individual color.

```sh
curl -i -X DELETE http://localhost:5000/colors/red
```

HTTP Verbs

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.

```sh
# 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 a color list
- Create color
- Read the last color
- Update last color
- Delete the 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 (`colorservice.js`), we create a helper function to make the HTTP requests for us:

```js
function request(verb, endpoint, obj, cb) {
    print('request: ' + verb + ' ' + BASE + (endpoint ? '/' + e
    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 a postfix to the BASE address (e.g. '**http://localhost:5000/colors (http://localhost:5000/colors)** '). The third parameter is the optional obj, to be sent 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). This code resides in the service implementation:

```js
function getColors(cb) {
    // GET http://localhost:5000/colors
    request('GET', null, null, cb)
}

function createColor(entry, cb) {
    // POST http://localhost:5000/colors
    request('POST', null, entry, cb)
}

function getColor(name, cb) {
    // GET http://localhost:5000/colors/${name}
    request('GET', name, null, cb)
}

function updateColor(name, entry, cb) {
    // PUT http://localhost:5000/colors/${name}
    request('PUT', name, entry, cb)
}

function deleteColor(name, cb) {
    // DELETE http://localhost:5000/colors/${name}
    request('DELETE', name, null, cb)
}
```

In the UI we use the service to implement our commands. We have a `ListModel` with the id `gridModel` as a data provider for the `GridView`. The commands are indicated using a `Button` UI element.

Importing our service library is pretty straightforward:

```
import "colorservice.js" as Service
```

Reading the color list from the server:

```
Button {
    text: 'Read Colors'
    onClicked: {
        Service.getColors(function(response) {
            print('handle get colors response: ' + JSON.stringi
            gridModel.clear()
            const entries = response.data
            for(let i=0; i<entries.length; i++) {
                gridModel.append(entries[i])
            }
        })
    }
}
```

Create a new color entry on the server:

```
Button {
    text: 'Create New'
    onClicked: {
        const index = gridModel.count - 1
        const entry = {
            name: 'color-' + index,
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toStri
        }
        Service.createColor(entry, function(response) {
```

```
            print('handle create color response: ' + JSON.strin
            gridModel.append(response)
        })
    }
}
```

Reading a color based on its name:

```
Button {
    text: 'Read Last Color'
    onClicked: {
        const index = gridModel.count - 1
        const name = gridModel.get(index).name
        Service.getColor(name, function(response) {
            print('handle get color response:' + JSON.stringify
            message.text = response.value
        })
    }
}
```

Update a color entry on the server based on the color name:

```
Button {
    text: 'Update Last Color'
    onClicked: {
        const index = gridModel.count - 1
        const name = gridModel.get(index).name
        const entry = {
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toStri
        }
        Service.updateColor(name, entry, function(response) {
            print('handle update color response: ' + JSON.strin
            gridModel.setProperty(gridModel.count - 1, 'value',
        })
    }
}
```

Delete a color by the color name:

```
Button {
    text: 'Delete Last Color'
    onClicked: {
        const index = gridModel.count - 1
        const name = gridModel.get(index).name
        Service.deleteColor(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 one endpoint. And the API could be defined using JSON RPC (**http://www.jsonrpc.org/ (http://www.jsonrpc.org/)**). Sure also XML based API is possible, but the JSON approach has great advantages as the parsing is built into the QML/JS as part of JavaScript.

Authentication using OAuth

OAuth is an open protocol to allow secure authorization in a simple and standard method from web, mobile, and desktop applications. OAuth is used to authenticate a client against common web-services such as Google, Facebook, and Twitter.

> **TIP**
>
> 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/**

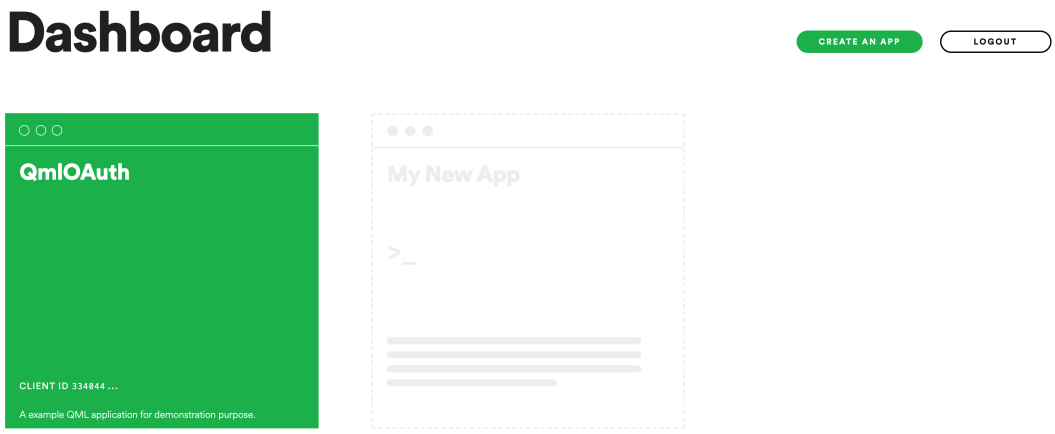
Integration example

In this section, we will go through an example of OAuth integration using the **Spotify API (https://developer.spotify.com/documentation/web-api/)** . This example uses a combination of C++ classes and QML/JS. To discover more on this integration, please refer to Chapter 16.

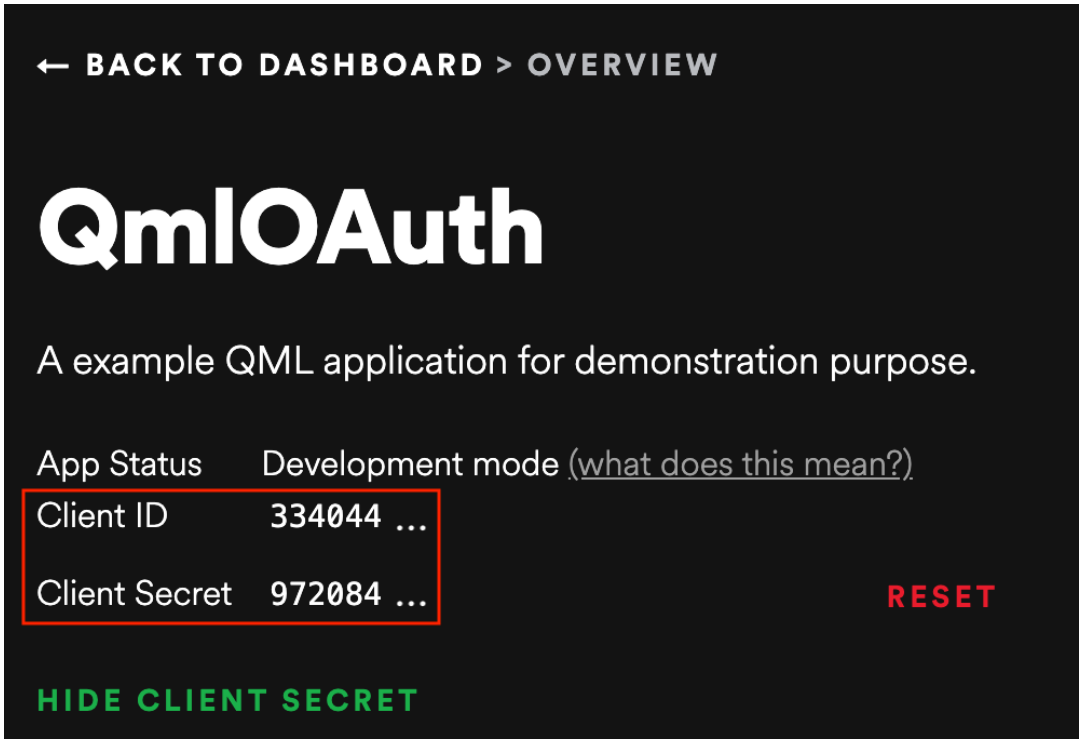
This application's goal is to retrieve the top ten favourite artists of the authenticated user.

Creating the App

First, you will need to create a dedicated app on the **Spotify Developer's portal (https://developer.spotify.com/dashboard/applications)** .



Once your app is created, you'll receive two keys: a `client id` and a `client secret` .



The QML file

The process is divided in two phases:

1. The application connects to the Spotify API, which in turns requests the user to authorize it;
2. If authorized, the application displays the list of the top ten favourite artists of the user.

Authorizing the app

Let's start with the first step:

```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

import Spotify
```

When the application starts, we will first import a custom library, `Spotify`, that defines a `SpotifyAPI` component (we'll come to that later). This component will then be instantiated:

```
ApplicationWindow {
    width: 320
    height: 568
    visible: true
    title: qsTr("Spotify OAuth2")

    BusyIndicator {
        visible: !spotifyApi.isAuthenticated
        anchors.centerIn: parent
    }

    SpotifyAPI {
        id: spotifyApi
        onIsAuthenticatedChanged: if(isAuthenticated) spotifyMo
    }
```

Once the application has been loaded, the `SpotifyAPI` component will request an authorization to Spotify:

```
Component.onCompleted: {
    spotifyApi.setCredentials("CLIENT_ID", "CLIENT_SECRET")
    spotifyApi.authorize()
}
```

Until the authorization is provided, a busy indicator will be displayed in the center of the app.

> **TIP**
>
> Please note that for security reasons, the API credentials should never be put directly into a QML file!

Listing the user's favorite artists

The next step happens when the authorization has been granted. To display the list of artists, we will use the Model/View/Delegate pattern:

```
SpotifyModel {
    id: spotifyModel
    spotifyApi: spotifyApi
}

ListView {
    visible: spotifyApi.isAuthenticated
    width: parent.width
    height: parent.height
    model: spotifyModel
    delegate: Pane {
        id: delegate
        required property var model
        topPadding: 0
        Column {
            width: 300
            spacing: 10

            Rectangle {
                height: 1
                width: parent.width
                color: delegate.model.index > 0 ? "#3d3d3d" : "
            }

            Row {
                spacing: 10
```

```
                Item {
                    width: 20
                    height: width

                    Rectangle {
                        width: 20
                        height: 20
                        anchors.top: parent.top
                        anchors.right: parent.right
                        color: "black"

                        Label {
                            anchors.centerIn: parent
                            font.pointSize: 16
                            text: delegate.model.index + 1
                            color: "white"
                        }
                    }
                }

                Image {
                    width: 80
                    height: width
                    source: delegate.model.imageURL
                    fillMode: Image.PreserveAspectFit
                }

                Column {
                    Label {
                        text: delegate.model.name
                        font.pointSize: 16
                        font.bold: true
                    }
                    Label { text: "Followers: " + delegate.mode
                }
            }
        }
    }
}
```

The model `SpotifyModel` is defined in the `Spotify` library. To work properly, it needs a `SpotifyAPI` .

The ListView displays a vertical list of artists. An artist is represented by a name, an image and the total count of followers.

SpotifyAPI

Let's now get a bit deeper into the authentication flow. We'll focus on the `SpotifyAPI` class, a `QML_ELEMENT` defined on the C++ side.

```
#ifndef SPOTIFYAPI_H
#define SPOTIFYAPI_H

#include <QtCore>
#include <QtNetwork>
#include <QtQml/qqml.h>

#include <QOAuth2AuthorizationCodeFlow>

class SpotifyAPI: public QObject
{
    Q_OBJECT
    QML_ELEMENT

    Q_PROPERTY(bool isAuthenticated READ isAuthenticated WRITE

public:
    SpotifyAPI(QObject *parent = nullptr);

    void setAuthenticated(bool isAuthenticated) {
        if (m_isAuthenticated != isAuthenticated) {
            m_isAuthenticated = isAuthenticated;
            emit isAuthenticatedChanged();
        }
    }
```

```
    bool isAuthenticated() const {
        return m_isAuthenticated;
    }

    QNetworkReply* getTopArtists();

public slots:
    void setCredentials(const QString& clientId, const QString&
    void authorize();

signals:
    void isAuthenticatedChanged();

private:
    QOAuth2AuthorizationCodeFlow m_oauth2;
    bool m_isAuthenticated;
};

#endif // SPOTIFYAPI_H
```

First, we'll import the `<QOAuth2AuthorizationCodeFlow>` class. This class is a part of the `QtNetworkAuth` module, which contains various implementations of `OAuth` .

```
#include <QOAuth2AuthorizationCodeFlow>
```

Our class, `SpotifyAPI` , will define a `isAuthenticated` property:

```
Q_PROPERTY(bool isAuthenticated READ isAuthenticated WRITE setA
```

The two public slots that we used in the QML files:

```
void setCredentials(const QString& clientId, const QString& cli
void authorize();
```

And a private member representing the authentication flow:

```
QOAuth2AuthorizationCodeFlow m_oauth2;
```

On the implementation side, we have the following code:

```
#include "spotifyapi.h"

#include <QtGui>
#include <QtCore>
#include <QtNetworkAuth>

SpotifyAPI::SpotifyAPI(QObject *parent): QObject(parent), m_isA
    m_oauth2.setAuthorizationUrl(QUrl("https://accounts.spotify
    m_oauth2.setAccessTokenUrl(QUrl("https://accounts.spotify.c
    m_oauth2.setScope("user-top-read");

    m_oauth2.setReplyHandler(new QOAuthHttpServerReplyHandler(8
    m_oauth2.setModifyParametersFunction([&](QAbstractOAuth::St
        if(stage == QAbstractOAuth::Stage::RequestingAuthorizat
            parameters->insert("duration", "permanent");
        }
    });

    connect(&m_oauth2, &QOAuth2AuthorizationCodeFlow::authorize
    connect(&m_oauth2, &QOAuth2AuthorizationCodeFlow::statusCha
        if (status == QAbstractOAuth::Status::Granted) {
            setAuthenticated(true);
        } else {
            setAuthenticated(false);
        }
    });
}

void SpotifyAPI::setCredentials(const QString& clientId, const
    m_oauth2.setClientIdentifier(clientId);
```

```
    m_oauth2.setClientIdentifierSharedKey(clientSecret);
}

void SpotifyAPI::authorize() {
    m_oauth2.grant();
}

QNetworkReply* SpotifyAPI::getTopArtists() {
    return m_oauth2.get(QUrl("https://api.spotify.com/v1/me/top
}
```

The constructor task mainly consists in configuring the authentication flow. First, we define the Spotify API routes that will serve as authenticators.

```
m_oauth2.setAuthorizationUrl(QUrl("https://accounts.spotify.com
m_oauth2.setAccessTokenUrl(QUrl("https://accounts.spotify.com/a
```

We then select the scope (= the Spotify authorizations) that we want to use:

```
m_oauth2.setScope("user-top-read");
```

Since OAuth is a two-way communication process, we instantiate a dedicated local server to handle the replies:

```
m_oauth2.setReplyHandler(new QOAuthHttpServerReplyHandler(8000,
```

Finally, we configure two signals and slots.

```
connect(&m_oauth2, &QOAuth2AuthorizationCodeFlow::authorizeWith
connect(&m_oauth2, &QOAuth2AuthorizationCodeFlow::statusChanged
```

The first one configures the authorization to happen within a web-browser (through `&QDesktopServices::openUrl`), while the second makes sure that we are notified when the authorization process has been completed.

The `authorize()` method is only a placeholder for calling the underlying `grant()` method of the authentication flow. This is the method that triggers the process.

```
void SpotifyAPI::authorize() {
    m_oauth2.grant();
}
```

Finally, the `getTopArtists()` calls the web api using the authorization context provided by the `m_oauth2` network access manager.

```
QNetworkReply* SpotifyAPI::getTopArtists() {
    return m_oauth2.get(QUrl("https://api.spotify.com/v1/me/top
}
```

The Spotify model

This class is a `QML_ELEMENT` that subclasses `QAbstractListModel` to represent our list of artists. It relies on `SpotifyAPI` to gather the artists from the remote endpoint.

```
#ifndef SPOTIFYMODEL_H
#define SPOTIFYMODEL_H

#include <QtCore>

#include "spotifyapi.h"
```

```
QT_FORWARD_DECLARE_CLASS(QNetworkReply)

class SpotifyModel : public QAbstractListModel
{
    Q_OBJECT
    QML_ELEMENT

    Q_PROPERTY(SpotifyAPI* spotifyApi READ spotifyApi WRITE set

public:
    SpotifyModel(QObject *parent = nullptr);

    void setSpotifyApi(SpotifyAPI* spotifyApi) {
        if (m_spotifyApi != spotifyApi) {
            m_spotifyApi = spotifyApi;
            emit spotifyApiChanged();
        }
    }

    SpotifyAPI* spotifyApi() const {
        return m_spotifyApi;
    }

    enum {
        NameRole = Qt::UserRole + 1,
        ImageURLRole,
        FollowersCountRole,
        HrefRole,
    };

    QHash<int, QByteArray> roleNames() const override;

    int rowCount(const QModelIndex &parent) const override;
    int columnCount(const QModelIndex &parent) const override;
    QVariant data(const QModelIndex &index, int role) const ove

signals:
    void spotifyApiChanged();
    void error(const QString &errorString);
```

```
public slots:
    void update();

private:
    QPointer<SpotifyAPI> m_spotifyApi;
    QList<QJsonObject> m_artists;
};

#endif // SPOTIFYMODEL_H
```

This class defines a `spotifyApi` property:

```
Q_PROPERTY(SpotifyAPI* spotifyApi READ spotifyApi WRITE setSpot
```

An enumeration of Roles (as per `QAbstractListModel`):

```
enum {
    NameRole = Qt::UserRole + 1,     // The artist's name
    ImageURLRole,                    // The artist's image
    FollowersCountRole,              // The artist's followers c
    HrefRole,                        // The link to the artist's
};
```

A slot to trigger the refresh of the artists list:

```
public slots:
    void update();
```

And, of course, the list of artists, represented as JSON objects:

```
public slots:
    QList<QJsonObject> m_artists;
```

On the implementation side, we have:

```
#include "spotifymodel.h"

#include <QtCore>
#include <QtNetwork>

SpotifyModel::SpotifyModel(QObject *parent): QAbstractListModel

QHash<int, QByteArray> SpotifyModel::roleNames() const {
    static const QHash<int, QByteArray> names {
        { NameRole, "name" },
        { ImageURLRole, "imageURL" },
        { FollowersCountRole, "followersCount" },
        { HrefRole, "href" },
    };
    return names;
}

int SpotifyModel::rowCount(const QModelIndex &parent) const {
    Q_UNUSED(parent);
    return m_artists.size();
}

int SpotifyModel::columnCount(const QModelIndex &parent) const
    Q_UNUSED(parent);
    return m_artists.size() ? 1 : 0;
}

QVariant SpotifyModel::data(const QModelIndex &index, int role)
    Q_UNUSED(role);
    if (!index.isValid())
        return QVariant();

    if (role == Qt::DisplayRole || role == NameRole) {
        return m_artists.at(index.row()).value("name").toString
    }

    if (role == ImageURLRole) {
```

```
            const auto artistObject = m_artists.at(index.row());
            const auto imagesValue = artistObject.value("images");

            Q_ASSERT(imagesValue.isArray());
            const auto imagesArray = imagesValue.toArray();
            if (imagesArray.isEmpty())
                return "";

            const auto imageValue = imagesArray.at(0).toObject();
            return imageValue.value("url").toString();
        }

        if (role == FollowersCountRole) {
            const auto artistObject = m_artists.at(index.row());
            const auto followersValue = artistObject.value("followe
            return followersValue.value("total").toInt();
        }

        if (role == HrefRole) {
            return m_artists.at(index.row()).value("href").toString
        }

        return QVariant();
    }

    void SpotifyModel::update() {
        if (m_spotifyApi == nullptr) {
            emit error("SpotifyModel::error: SpotifyApi is not set.
            return;
        }

        auto reply = m_spotifyApi->getTopArtists();

        connect(reply, &QNetworkReply::finished, [=]() {
            reply->deleteLater();
            if (reply->error() != QNetworkReply::NoError) {
                emit error(reply->errorString());
                return;
            }
```

```
        const auto json = reply->readAll();
        const auto document = QJsonDocument::fromJson(json);

        Q_ASSERT(document.isObject());
        const auto rootObject = document.object();
        const auto artistsValue = rootObject.value("items");

        Q_ASSERT(artistsValue.isArray());
        const auto artistsArray = artistsValue.toArray();
        if (artistsArray.isEmpty())
            return;

        beginResetModel();
        m_artists.clear();
        for (const auto artistValue : qAsConst(artistsArray)) {
            Q_ASSERT(artistValue.isObject());
            m_artists.append(artistValue.toObject());
        }
        endResetModel();
    });
}
```

The `update()` method calls the `getTopArtists()` method and handle its reply by extracting the individual items from the JSON document and refreshing the list of artists within the model.

```
auto reply = m_spotifyApi->getTopArtists();

connect(reply, &QNetworkReply::finished, [=]() {
    reply->deleteLater();
    if (reply->error() != QNetworkReply::NoError) {
        emit error(reply->errorString());
        return;
    }

    const auto json = reply->readAll();
    const auto document = QJsonDocument::fromJson(json);
```

```
        Q_ASSERT(document.isObject());
        const auto rootObject = document.object();
        const auto artistsValue = rootObject.value("items");

        Q_ASSERT(artistsValue.isArray());
        const auto artistsArray = artistsValue.toArray();
        if (artistsArray.isEmpty())
            return;

        beginResetModel();
        m_artists.clear();
        for (const auto artistValue : qAsConst(artistsArray)) {
            Q_ASSERT(artistValue.isObject());
            m_artists.append(artistValue.toObject());
        }
        endResetModel();
    });
```

The `data()` method extracts, depending on the requested model role, the relevant attributes of an Artist and returns as a `QVariant` :

```
        if (role == Qt::DisplayRole || role == NameRole) {
            return m_artists.at(index.row()).value("name").toString
        }

        if (role == ImageURLRole) {
            const auto artistObject = m_artists.at(index.row());
            const auto imagesValue = artistObject.value("images");

            Q_ASSERT(imagesValue.isArray());
            const auto imagesArray = imagesValue.toArray();
            if (imagesArray.isEmpty())
                return "";

            const auto imageValue = imagesArray.at(0).toObject();
            return imageValue.value("url").toString();
        }
```

```
if (role == FollowersCountRole) {
    const auto artistObject = m_artists.at(index.row());
    const auto followersValue = artistObject.value("followe
    return followersValue.value("total").toInt();
}

if (role == HrefRole) {
    return m_artists.at(index.row()).value("href").toString
}
```

Spotify OAuth2

1

Hans Zimmer
Followers: 2630192

2

Gavin Greenaway
Followers: 9932

3

John Powell
Followers: 148515

4

The Lumineers
Followers: 4389936

5

Howard Shore
Followers: 361362

6

The Who
Followers: 4121795

Web Sockets

The WebSockets module provides an implementation of the WebSockets protocol for WebSockets clients and servers. It mirrors the Qt CPP module. It allows sending a string and binary messages using a full duplex communication channel. A WebSocket is normally established by making an HTTP connection to the server and the server then “upgrades” the connection to a WebSocket connection.

In Qt/QML you can also simply use the WebSocket and WebSocketServer objects to creates direct WebSocket connection. The WebSocket protocol uses the “ws” URL schema or “wss” for a secure connection.

You can use the web socket qml module by importing it first.

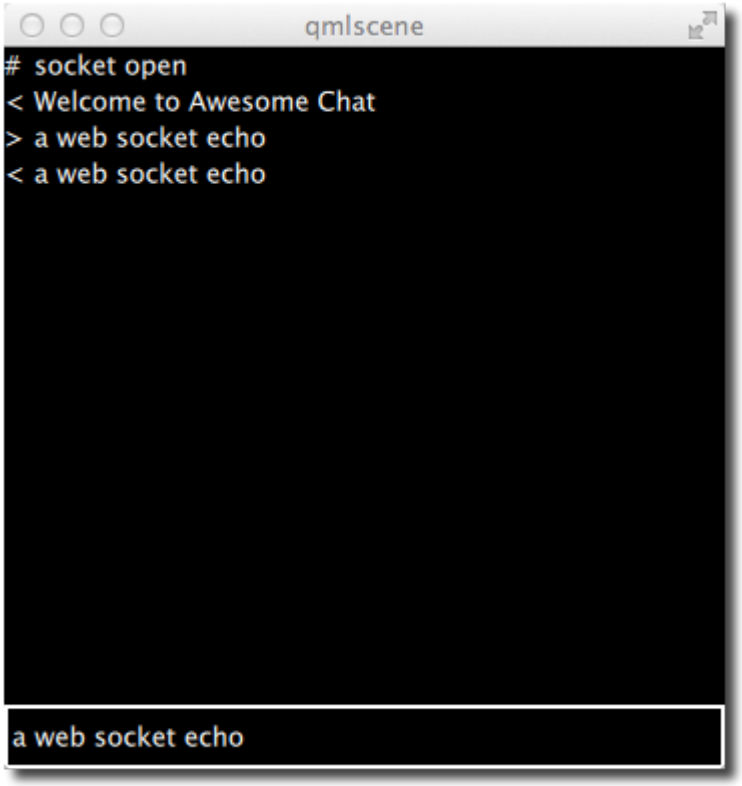
```
import QtWebSockets

WebSocket {
    id: socket
}
```

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 connecting 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 (https://npmjs.org/package/ws)** module. For this, you first

need to install **node js (http://nodejs.org/)** . Then, create a `ws_server` folder and install the ws package using the node package manager (npm).

The code shall create a simple echo server in NodeJS to echo our messages back to our QML client.




```sh
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.

```js
const WebSocketServer = require('ws').Server

const 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.toString())
    });
    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

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 with into 24 columns.

```
// ChatView.qml
import QtQuick

ListView {
    id: root
    width: 100
    height: 62

    model: ListModel {}

    function append(prefix, message) {
        model.append({prefix: prefix, message: message})
    }

    delegate: Row {
        id: delegate

        required property var model
        property real cw: width / 24

        width: root.width
        height: 18

        Label {
            width: delegate.cw * 1
            height: parent.height
            text: delegate.model.prefix
        }

        Label {
            width: delegate.cw * 23
```

```
            height: parent.height
            text: delegate.model.message
        }
    }
}
```

The chat input is just a simple text input wrapped with a colored border.

```
// ChatInput.qml
import QtQuick

FocusScope {
    id: root

    property alias text: input.text
    signal accepted(string text)

    width: 240
    height: 32

    Rectangle {
        anchors.fill: parent
        color: '#000'
        border.color: '#fff'
        border.width: 2
    }

    TextInput {
        id: input
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.verticalCenter: parent.verticalCenter
        anchors.leftMargin: 4
        anchors.rightMargin: 4
        color: '#fff'
        focus: true
        onAccepted: function () {
            root.accepted(text)
```

```
        }
    }
}
```

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 sent to the server.

```
// ws_client.qml
import QtQuick
import QtWebSockets

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: function(text) {
            print('send message: ' + text)
            socket.sendTextMessage(text)
            box.append('>', text)
            text = ''
```

```
        }
    }

    WebSocket {
        id: socket

        url: "ws://localhost:3000"
        active: true

        onTextMessageReceived: function (message) {
            box.append('<', message)
        }

        onStatusChanged: {
            if (socket.status == WebSocket.Error) {
                box.append('#', 'socket error ' + socket.errorS
            } 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.

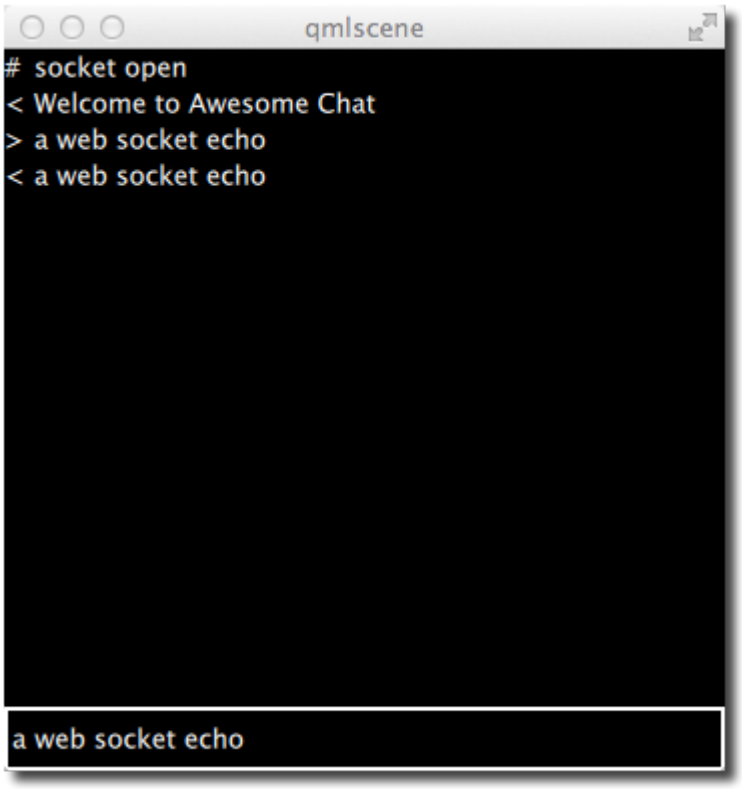
Running the server

```sh
cd ws_server
node server.js
```

Running the client

```sh
cd ws_client
qml ws_client.qml
```

When entering text and pressing enter you should see something like this.



Summary

This concludes our chapter about QML networking. Please bear in mind Qt has on the native side a much richer networking API as on the QML side currently. But the idea of the chapter is to push the boundaries of QML networking and how to integrate with cloud-based services.

Storage

In this chapter we discuss how to store and retrieve data from Qt Quick. Qt Quick offers only limited ways of storing local data directly. In this sense, it acts more like a browser. In many projects storing data is handled by the C++ backend and the required functionality is exported to the Qt Quick frontend side. Qt Quick does not provide you with access to the host file system to read and write files as you are used from the Qt C++ side. So it would be the task of the backend engineer to write such a plugin or maybe use a network channel to communicate with a local server, which provides these capabilities.

Every application needs to store smaller and larger information persistently. This can be done locally on the file system or remote on a server. Some information will be structured and simple (e.g. settings), some will be large and complicated for example documentation files and some will be large and structured and will require some sort of database connection. Here we will mainly cover the built-in capabilities of Qt Quick to store data as also the networked ways.

Settings

Qt comes with a `Settings` element for loading and storing settings. The is still in the lab's module, which means the API may break in the future. So be aware.

Here is a small example which applies a color value to a base rectangle. Every time the user clicks on the window a new random color is generated. When the application is closed and relaunched again you should see your last color. The default color should be the color initially set on the root rectangle.

```
import QtQuick
import Qt.labs.settings 1.0

Rectangle {
    id: root

    width: 320
    height: 240
    color: '#fff' // default color
    Settings {
        property alias color: root.color
    }
    MouseArea {
        anchors.fill: parent
        // random color
        onClicked: root.color = Qt.hsla(Math.random(), 0.5, 0.5
    }
}
```

The settings value are stored every time the value changes. This might be not always what you want. To store the settings only when required you can use standard properties combined with a function that alters the setting when called.

```
Rectangle {
    id: root
    color: settings.color
    Settings {
        id: settings
        property color color: '#000000'
    }
    function storeSettings() { // executed maybe on destruction
        settings.color = root.color
    }
}
```

It is also possible to group settings into different categories using the `category` property.

```
Settings {
    category: 'window'
    property alias x: window.x
    property alias y: window.x
    property alias width: window.width
    property alias height: window.height
}
```

The settings are stored according to your application name, organization, and domain. This information is normally set in the main function of your C++ code.

```
int main(int argc, char** argv) {
    ...
```

```
    QCoreApplication::setApplicationName("Awesome Application")
    QCoreApplication::setOrganizationName("Awesome Company");
    QCoreApplication::setOrganizationDomain("org.awesome");
    ...
}
```

If you are writing a pure QML application, you can set the same attributed using the global properties `Qt.application.name`, `Qt.application.organization`, and `Qt.application.domain`.

Local Storage - SQL

Qt Quick supports a local storage API known from the web browsers the local storage API. the API is available under “import QtQuick.LocalStorage 2.0”.

In general, it stores the content into an SQLite database in a system-specific location in a unique ID based file based on the given database name and version. It is not possible to list or delete existing databases. You can find the storage location from `QQmlEngine::offlineStoragePath()` .

You use the API by first creating a database object and then creating transactions on the database. Each transaction can contain one or more SQL queries. The transaction will roll-back when a SQL query will fail inside the transaction.

For example, to read from a simple notes table with a text column you could use the local storage like this:

```
import QtQuick
import QtQuick.LocalStorage 2.0

Item {
    Component.onCompleted: {
        const db = LocalStorage.openDatabaseSync("MyExample", "
        db.transaction( function(tx) {
            const result = tx.executeSql('select * from notes')
            for(let i = 0; i < result.rows.length; i++) {
                print(result.rows[i].text)
            }
        })
```

```
    }
}
```

Crazy Rectangle

As an example assume we would like to store the position of a rectangle on our scene.

Here is the base of the example. It contains a rectangle called `crazy` that is draggable and shows its current `x` and `y` position as text.

```
Item {
    width: 400
    height: 400

    Rectangle {
```

```
        id: crazy
        objectName: 'crazy'
        width: 100
        height: 100
        x: 50
        y: 50
        color: "#53d769"
        border.color: Qt.lighter(color, 1.1)
        Text {
            anchors.centerIn: parent
            text: Math.round(parent.x) + '/' + Math.round(paren
        }
        MouseArea {
            anchors.fill: parent
            drag.target: parent
        }
    }
    // ...
```

You can drag the rectangle freely around. When you close the application and launch it again the rectangle is at the same position.

Now we would like to add that the x/y position of the rectangle is stored inside the SQL DB. For this, we need to add an `init`, `read` and `store` database function. These functions are called when on component completed and on component destruction.

```
import QtQuick
import QtQuick.LocalStorage 2.0

Item {
    // reference to the database object
    property var db

    function initDatabase() {
        // initialize the database object
    }
```

```
    function storeData() {
        // stores data to DB
    }

    function readData() {
        // reads and applies data from DB
    }

    Component.onCompleted: {
        initDatabase()
        readData()
    }

    Component.onDestruction: {
        storeData()
    }
}
```

You could also extract the DB code in an own JS library, which does all the logic. This would be the preferred way if the logic gets more complicated.

In the database initialization function, we create the DB object and ensure the SQL table is created. Notice that the database functions are quite talkative so that you can follow along on the console.

```
function initDatabase() {
    // initialize the database object
    print('initDatabase()')
    db = LocalStorage.openDatabaseSync("CrazyBox", "1.0", "A bo
    db.transaction( function(tx) {
        print('... create table')
        tx.executeSql('CREATE TABLE IF NOT EXISTS data(name TEX
    })
}
```

The application next calls the read function to read existing data back from the database. Here we need to differentiate if there is already data in the table. To check we look into how many rows the select clause has returned.

```
function readData() {
    // reads and applies data from DB
    print('readData()')
    if(!db) { return }
    db.transaction(function(tx) {
        print('... read crazy object')
        const result = tx.executeSql('select * from data where
        if(result.rows.length === 1) {
            print('... update crazy geometry')
            // get the value column
            const value = result.rows[0].value
            // convert to JS object
            const obj = JSON.parse(value)
            // apply to object
            crazy.x = obj.x
            crazy.y = obj.y
        }
    })
}
```

We expect the data is stored in a JSON string inside the value column. This is not typical SQL like, but works nicely with JS code. So instead of storing the x,y as properties in the table, we store them as a complete JS object using the JSON stringify/parse methods. In the end, we get a valid JS object with x and y properties, which we can apply on our crazy rectangle.

To store the data, we need to differentiate the update and insert cases. We use update when a record already exists and insert if no record under the name “crazy” exists.

```
function storeData() {
    // stores data to DB
    print('storeData()')
    if(!db) { return }
    db.transaction(function(tx) {
        print('... check if a crazy object exists')
        var result = tx.executeSql('SELECT * from data where na
        // prepare object to be stored as JSON
        var obj = { x: crazy.x, y: crazy.y }
        if(result.rows.length === 1) { // use update
            print('... crazy exists, update it')
            result = tx.executeSql('UPDATE data set value=? whe
        } else { // use insert
            print('... crazy does not exists, create it')
            result = tx.executeSql('INSERT INTO data VALUES (?,
        }
    })
}
```

Instead of selecting the whole recordset we could also use the SQLite count function like this: `SELECT COUNT(*) from data where name = "crazy"` which would return use one row with the number of rows affected by the select query. Otherwise, this is common SQL code. As an additional feature, we use the SQL value binding using the `?` in the query.

Now you can drag the rectangle and when you quit the application the database stores the x/y position and applies it on the next application run.

Dynamic QML

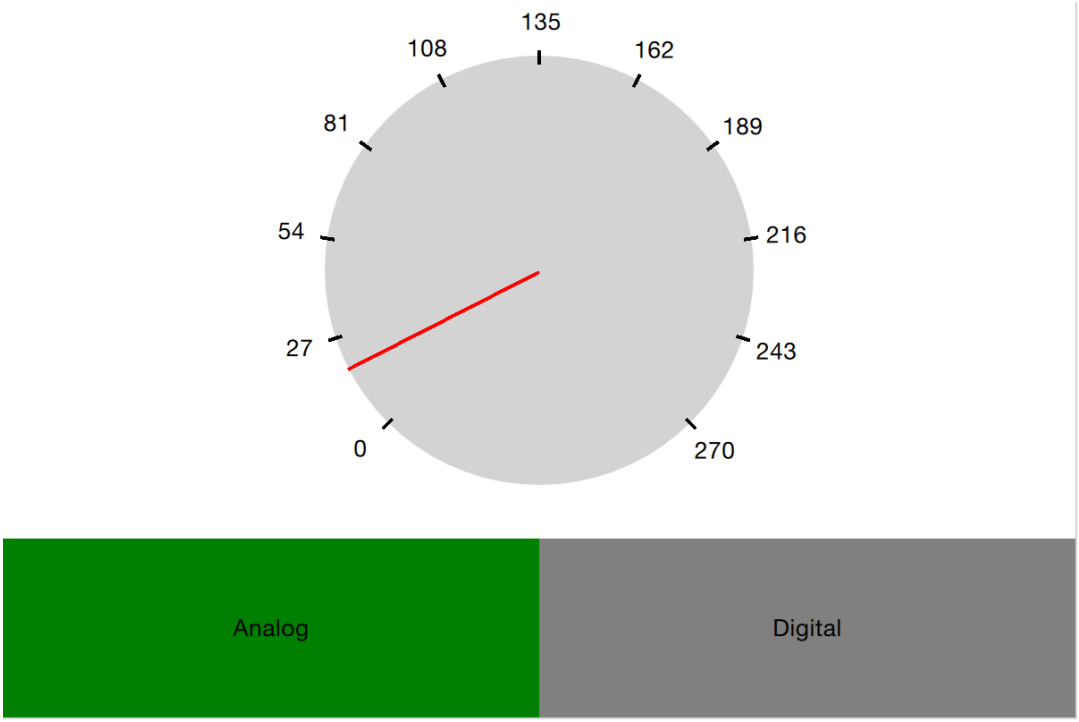
Until now, we have treated QML as a tool for constructing a static set of scenes and navigating between them. Depending on various states and logic rules, a living and dynamic user interface is constructed. By working with QML and JavaScript in a more dynamic manner, the flexibility and possibilities expand even further. Components can be loaded and instantiated at run-time, elements can be destroyed. Dynamically created user interfaces can be saved to disk and later restored.

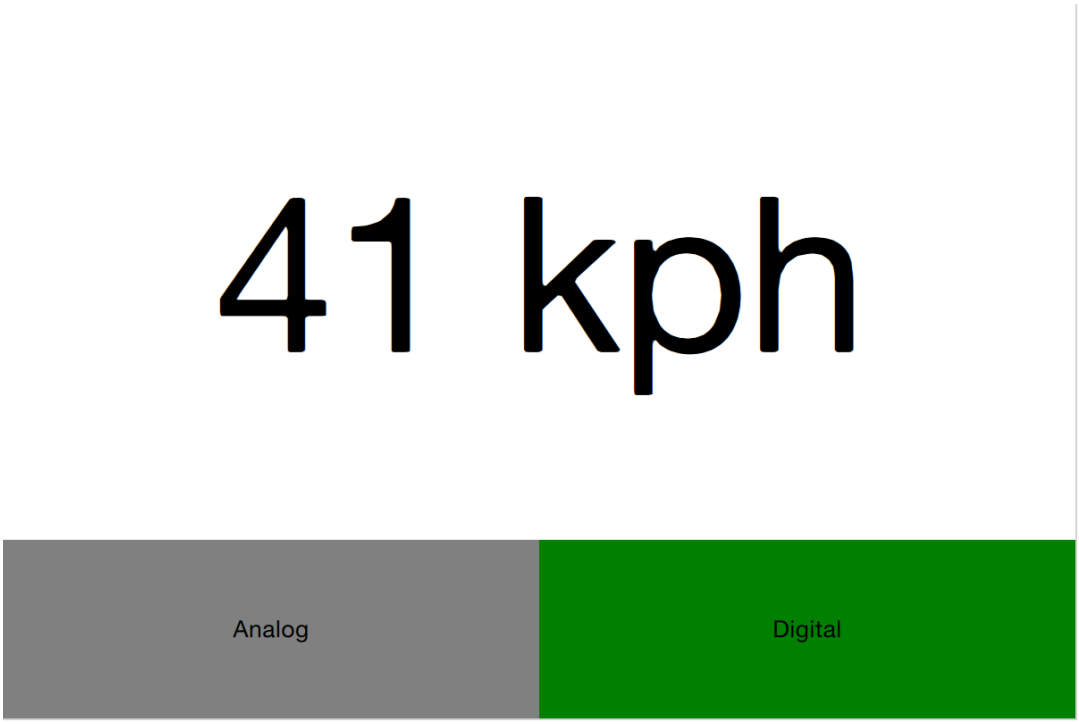
Loading Components Dynamically

The easiest way to dynamically load different parts of QML is to use the `Loader` element. It serves as a placeholder to the item that is being loaded. The item to load is controlled through either the `source` property or the `sourceComponent` property. The former loads the item from a given URL, while the latter instantiates a `Component` .

As the loader serves as a placeholder for the item being loaded, its size depends on the size of the item, and vice versa. If the `Loader` element has a size, either by having set `width` and `height` or through anchoring, the loaded item will be given the loader's size. If the `Loader` has no size, it is resized in accordance to the size of the item being loaded.

The example described below demonstrates how two separate user interface parts can be loaded into the same space using a `Loader` element. The idea is to have a speed dial that can be either digital or analog, as shown in the illustration below. The code surrounding the dial is unaffected by which item that is loaded for the moment.





The first step in the application is to declare a `Loader` element. Notice that the `source` property is left out. This is because the `source` depends on which state the user interface is in.

```
Loader {
    id: dialLoader

    anchors.fill: parent
}
```

In the `states` property of the parent of `dialLoader` a set of `PropertyChanges` elements drives the loading of different QML files depending on the `state`. The `source` property happens to be a relative file path in this example, but it can just as well be a full URL, fetching the item over the web.

```
states: [
    State {
        name: "analog"
        PropertyChanges { target: analogButton; color: "green";
        PropertyChanges { target: dialLoader; source: "Analog.q
    },
    State {
        name: "digital"
        PropertyChanges { target: digitalButton; color: "green"
        PropertyChanges { target: dialLoader; source: "Digital.
    }
]
```

In order to make the loaded item come alive, it is `speed` property must be bound to the root `speed` property. This cannot be done as a direct binding as the item not always is loaded and changes over time. Instead, a `Binding` element must be used. The `target` property of the binding is changed every time the `Loader` triggers the `onLoaded` signal.

```
Loader {
    id: dialLoader
```

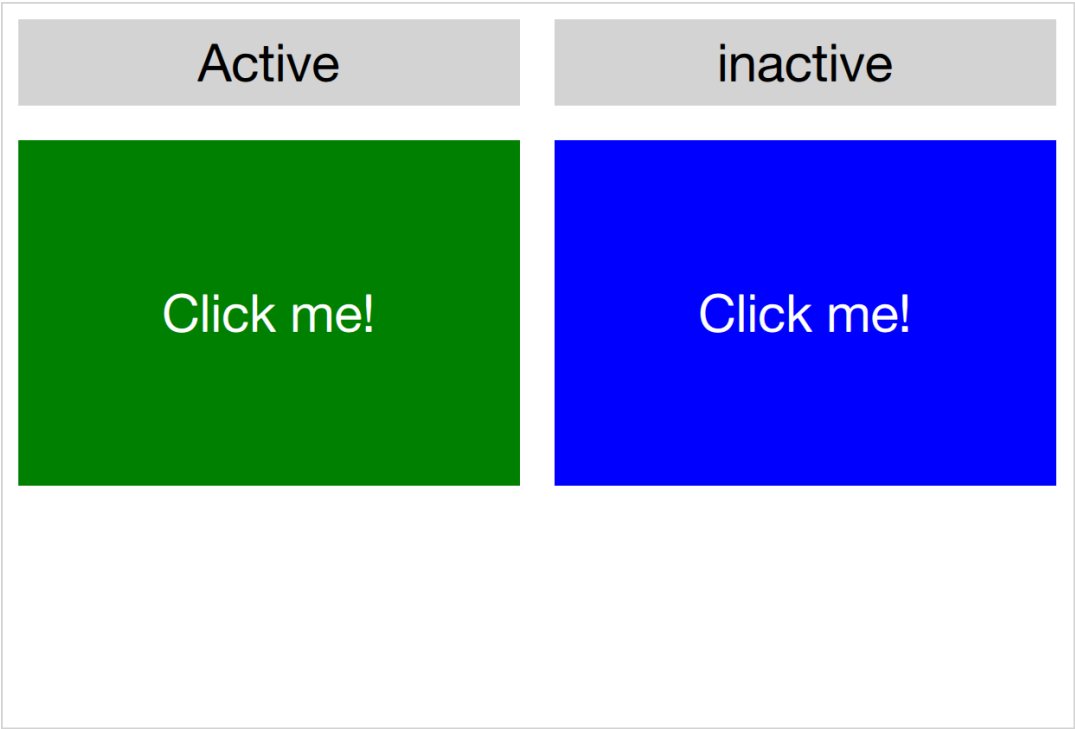
```
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: analogButton.top

        onLoaded: {
            binder.target = dialLoader.item;
        }
    }

    Binding {
        id: binder

        property: "speed"
        value: root.speed
    }
```

The `onLoaded` signal lets the loading QML act when the item has been loaded. In a similar fashion, the QML being loaded can rely on the `Component.onCompleted` signal. This signal is actually available for all components, regardless of how they are loaded. For instance, the root component of an entire application can use it to kick-start itself when the entire user interface has been loaded.

Connecting Indirectly

When creating QML elements dynamically, you cannot connect to signals using the `onSignalName` approach used for static setup. Instead, the `Connections` element must be used. It connects to any number of signals of a `target` element.

Having set the `target` property of a `Connections` element, the signals can be connected, as usual, that is, using the `onSignalName` approach. However, by altering the `target` property, different elements can be monitored at different times.



In the example shown above, a user interface consisting of two clickable areas is presented to the user. When either area is clicked, it is flashed using an animation. The left area is shown in the code snippet below. In the `MouseArea` , the `leftClickedAnimation` is triggered, causing the area to flash.

```
Rectangle {
    id: leftRectangle

    width: 290
    height: 200

    color: "green"

    MouseArea {
        id: leftMouseArea
        anchors.fill: parent
        onClicked: leftClickedAnimation.start()
    }

    Text {
```

```
            anchors.centerIn: parent
            font.pixelSize: 30
            color: "white"
            text: "Click me!"
        }
    }
```

In addition to the two clickable areas, a `Connections` element is used. This triggers the third animation when the active, i.e. the `target` of the element, is clicked.

```
    Connections {
        id: connections
        function onClicked() { activeClickedAnimation.start() }
    }
```

To determine which `MouseArea` to target, two states are defined. Notice that we cannot set the `target` property using a `PropertyChanges` element, as it already contains a `target` property. Instead a `StateChangeScript` is utilized.

```
    states: [
        State {
            name: "left"
            StateChangeScript {
                script: connections.target = leftMouseArea
            }
        },
        State {
            name: "right"
            StateChangeScript {
                script: connections.target = rightMouseArea
            }
        }
    ]
```

When trying out the example, it is worth noticing that when multiple signal handlers are used, all are invoked. The execution order of these is, however, undefined.

When creating a `Connections` element without setting the `target` property, the property defaults to `parent`. This means that it has to be explicitly set to `null` to avoid catching signals from the `parent` until the `target` is set. This behavior does make it possible to create custom signal handler components based on a `Connections` element. This way, the code reacting to the signals can be encapsulated and re-used.

In the example below, the `Flasher` component can be put inside any `MouseArea`. When clicked, it triggers an animation, causing the parent to flash. In the same `MouseArea` the actual task being triggered can also be carried out. This separates the standardized user feedback, i.e. the flashing, from the actual action.

```
import QtQuick

Connections {
    function onClicked() {
        // Automatically targets the parent
    }
}
```

To use the `Flasher`, simply instantiate a Flasher within each MouseArea, and it all works.

```
import QtQuick

Item {
    // A background flasher that flashes the background of any
}
```

When using a `Connections` element to monitor the signals of multiple types of `target` elements, you sometimes find yourself in a situation where the available signals vary between the targets. This results in the `Connections` element outputting run-time errors as signals are missed. To avoid this, the `ignoreUnknownSignal` property can be set to `true`. This ignores all such errors.

> **TIP**
>
> It is usually a bad idea to suppress error messages, and if you do, make sure to document why in a comment.

Binding Indirectly

Just as it is not possible to connect to signals of dynamically created elements directly, nor it is possible to bind properties of a dynamically created element without working with a bridge element. To bind a property of any element, including dynamically created elements, the `Binding` element is used.

The `Binding` element lets you specify a `target` element, a `property` to bind and a `value` to bind it to. Through using a `Binding` element, it is, for instance, possible to bind properties of a dynamically loaded element. This was demonstrated in the introductory example in this chapter, as shown below.

```
Loader {
    id: dialLoader

    anchors.left: parent.left
    anchors.right: parent.right
    anchors.top: parent.top
    anchors.bottom: analogButton.top
```

```
        onLoaded: {
            binder.target = dialLoader.item;
        }
    }

    Binding {
        id: binder

        property: "speed"
        value: root.speed
    }
```

As the `target` element of a `Binding` not always is set, and perhaps not always has a given property, the `when` property of the `Binding` element can be used to limit the time when the binding is active. For instance, it can be limited to specific modes in the user interface.

The `Binding` element also comes with a `delayed` property. When this property is set to `true` the binding is not propagated to the `target` until the event queue has been emptied. In high load situations this can serve as an optimization as intermediate values are not pushed to the `target` .

Creating and Destroying Objects

The `Loader` element makes it possible to populate part of a user interface dynamically. However, the overall structure of the interface is still static. Through JavaScript, it is possible to take one more step and to instantiate QML elements completely dynamically.

Before we dive into the details of creating elements dynamically, we need to understand the workflow. When loading a piece of QML from a file or even over the Internet, a component is created. The component encapsulates the interpreted QML code and can be used to create items. This means that loading a piece of QML code and instantiating items from it is a two-stage process. First, the QML code is parsed into a component. Then the component is used to instantiate actual item objects.

In addition to creating elements from QML code stored in files or on servers, it is also possible to create QML objects directly from text strings containing QML code. The dynamically created items are then treated in a similar fashion once instantiated.

Dynamically Loading and Instantiating Items

When loading a piece of QML, it is first interpreted as a component. This includes loading dependencies and validating the code. The location of the QML being loaded can be either a local file, a Qt resource, or even a distance network location specified by a URL. This means that the loading time can be everything from instant, for instance, a Qt resource located in RAM without any non-loaded dependencies, to very long, meaning a piece

of code located on a slow server with multiple dependencies that need to be loaded.

The status of a component being created can be tracked by it is `status` property. The available values are `Component.Null`, `Component.Loading`, `Component.Ready` and `Component.Error`. The `Null` to `Loading` to `Ready` is the usual flow. At any stage, the `status` can change to `Error`. In that case, the component cannot be used to create new object instances. The `Component.errorString()` function can be used to retrieve a user-readable error description.

When loading components over slow connections, the `progress` property can be of use. It ranges from `0.0`, meaning nothing has been loaded, to `1.0` indicating that all have been loaded. When the component's `status` changes to `Ready`, the component can be used to instantiate objects. The code below demonstrates how that can be achieved, taking into account the event of the component becoming ready or failing to be created directly, as well as the case where a component is ready slightly later.

```js
var component;

function createImageObject() {
    component = Qt.createComponent("dynamic-image.qml");
    if (component.status === Component.Ready || component.statu
        finishCreation();
    } else {
        component.statusChanged.connect(finishCreation);
    }
}

function finishCreation() {
    if (component.status === Component.Ready) {
        var image = component.createObject(root, {"x": 100, "y"
        if (image === null) {
            console.log("Error creating image");
```

```
        }
    } else if (component.status === Component.Error) {
        console.log("Error loading component:", component.error
    }
}
```

The code above is kept in a separate JavaScript source file, referenced from the main QML file.

```
import QtQuick
import "create-component.js" as ImageCreator

Item {
    id: root

    width: 1024
    height: 600

    Component.onCompleted: ImageCreator.createImageObject();
}
```

The `createObject` function of a component is used to create object instances, as shown above. This not only applies to dynamically loaded components but also `Component` elements inlined in the QML code. The resulting object can be used in the QML scene like any other object. The only difference is that it does not have an `id` .

The `createObject` function takes two arguments. The first is a `parent` object of the type `Item` . The second is a list of properties and values on the format `{"name": value, "name": value}` . This is demonstrated in the example below. Notice that the properties argument is optional.

```
var image = component.createObject(root, {"x": 100, "y": 100});
```

> **TIP**
>
> A dynamically created component instance is not different to an in-line `Component` element. The in-line `Component` element also provides functions to instantiate objects dynamically.

Incubating Components

When components are created using `createObject` the creation of the object component is blocking. This means that the instantiation of a complex element may block the main thread, causing a visible glitch. Alternatively, complex components may have to be broken down and loaded in stages using `Loader` elements.

To resolve this problem, a component can be instantiated using the `incubateObject` method. This might work just as `createObject` and return an instance immediately, or it may call back when the component is ready. Depending on your exact setup, this may or may not be a good way to solve instantiation related animation glitches.

To use an incubator, simply use it as `createComponent` . However, the returned object is an incubator and not the object instance itself. When the incubator's status is `Component.Ready` , the object is available through the `object` property of the incubator. All this is shown in the example below:

```js
function finishCreate() {
    if (component.status === Component.Ready) {
        var incubator = component.incubateObject(root, {"x": 10
        if (incubator.status === Component.Ready) {
            var image = incubator.object; // Created at once
        } else {
            incubator.onStatusChanged = function(status) {
                if (status === Component.Ready) {
```

```
                    var image = incubator.object; // Created as
                }
            };
        }
    }
}
```

Dynamically Instantiating Items from Text

Sometimes, it is convenient to be able to instantiate an object from a text string of QML. If nothing else, it is quicker than putting the code in a separate source file. For this, the `Qt.createQmlObject` function is used.

The function takes three arguments: `qml`, `parent` and `filepath`. The `qml` argument contains the string of QML code to instantiate. The `parent` argument provides a parent object to the newly created object. The `filepath` argument is used when reporting any errors from the creation of the object. The result returned from the function is either a new object or `null`.

> **WARNING**
>
> The `createQmlObject` function always returns immediately. For the function to succeed, all the dependencies of the call must be loaded. This means that if the code passed to the function refers to a non-loaded component, the call will fail and return `null`. To better handle this, the `createComponent` / `createObject` approach must be used.

The objects created using the `Qt.createQmlObject` function resembles any other dynamically created object. That means that it is identical to every other QML object, apart from not having an `id`. In the example

below, a new `Rectangle` element is instantiated from in-line QML code when the `root` element has been created.

```
import QtQuick

Item {
    id: root

    width: 1024
    height: 600

    function createItem() {
        Qt.createQmlObject("import QtQuick 2.5; Rectangle { x:
    }

    Component.onCompleted: root.createItem()
}
```

Managing Dynamically Created Elements

Dynamically created objects can be treated as any other object in a QML scene. However, there are some pitfalls that we need to be aware of. The most important is the concept of the creation contexts.

The creation context of a dynamically created object is the context within it is being created. This is not necessarily the same context as the parent exists in. When the creation context is destroyed, so are the bindings concerning the object. This means that it is important to implement the creation of dynamic objects in a place in the code which will be instantiated during the entire lifetime of the objects.

Dynamically created objects can also be dynamically destroyed. When doing this, there is a rule of thumb: never attempt to destroy an object that you have not created. This also includes elements that you have

created, but not using a dynamic mechanism such as `Component.createObject` or `createQmlObject`.

An object is destroyed by calling its `destroy` function. The function takes an optional argument which is an integer specifying how many milliseconds the objects shall exist before being destroyed. This is useful too, for instance, let the object complete a final transition.

```js
item = Qt.createQmlObject(...)
...
item.destroy()
```

> **TIP**
>
> It is possible to destroy an object from within, making it possible to create self-destroying popup windows for instance.

Tracking Dynamic Objects

Working with dynamic objects, it is often necessary to track the created objects. Another common feature is to be able to store and restore the state of the dynamic objects. Both these tasks are easily handled using an `XmlListModel` that is dynamically populated.

In the example shown below two types of elements, rockets and UFOs can be created and moved around by the user. In order to be able to manipulate the entire scene of dynamically created elements, we use a model to track the items.

The model, a `XmlListModel`, is populated as the items are created. The object reference is tracked alongside the source URL used when instantiating it. The latter is not strictly needed for tracking the objects but will come in handy later.

```
import QtQuick
import "create-object.js" as CreateObject

Item {
    id: root

    ListModel {
        id: objectsModel
    }

    function addUfo() {
        CreateObject.create("ufo.qml", root, itemAdded)
    }

    function addRocket() {
```

```
        CreateObject.create("rocket.qml", root, itemAdded)
    }

    function itemAdded(obj, source) {
        objectsModel.append({"obj": obj, "source": source})
    }
```

As you can tell from the example above, the `create-object.js` is a more generalized form of the JavaScript introduced earlier. The `create` method uses three arguments: a source URL, a root element, and a callback to invoke when finished. The callback gets called with two arguments: a reference to the newly created object and the source URL used.

This means that each time `addUfo` or `addRocket` functions are called, the `itemAdded` function will be called when the new object has been created. The latter will append the object reference and source URL to the `objectsModel` model.

The `objectsModel` can be used in many ways. In the example in question, the `clearItems` function relies on it. This function demonstrates two things. First, how to iterate over the model and perform a task, i.e. calling the `destroy` function for each item to remove it. Secondly, it highlights the fact that the model is not updated as objects are destroyed. Instead of removing the model item connected to the object in question, the `obj` property of that model item is set to `null`. To remedy this, the code explicitly has to clear the model item as the objects are removed.

```
function clearItems() {
    while(objectsModel.count > 0) {
        objectsModel.get(0).obj.destroy()
        objectsModel.remove(0)
    }
}
```

Having a model representing all dynamically created items, it is easy to create a function that serializes the items. In the example code, the serialized information consists of the source URL of each object along its `x` and `y` properties. These are the properties that can be altered by the user. The information is used to build an XML document string.

```
function serialize() {
    var res = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n<sce

    for(var ii=0; ii < objectsModel.count; ++ii) {
        var i = objectsModel.get(ii)
        res += "  <item>\n    <source>" + i.source + "</source>
    }

    res += "</scene>"

    return res
}
```

> **TIP**
>
> Currently, the `XmlListModel` of Qt 6 lacks the `xml` property and `get()` function needed to make serialization and deserialization work.

The XML document string can be used with an `XmlListModel` by setting the `xml` property of the model. In the code below, the model is shown along the `deserialize` function. The `deserialize` function kickstarts the deserialization by setting the `dsIndex` to refer to the first item of the model and then invoking the creation of that item. The callback, `dsItemAdded` then sets that `x` and `y` properties of the newly created object. It then updates the index and creates the next object, if any.

```
XmlListModel {
    id: xmlModel
    query: "/scene/item"
    XmlListModelRole { name: "source"; elementName: "source" }
    XmlListModelRole { name: "x"; elementName: "x" }
    XmlListModelRole { name: "y"; elementName: "y" }
}

function deserialize() {
    dsIndex = 0
    CreateObject.create(xmlModel.get(dsIndex).source, root, dsI
}

function dsItemAdded(obj, source) {
    itemAdded(obj, source)
    obj.x = xmlModel.get(dsIndex).x
    obj.y = xmlModel.get(dsIndex).y

    dsIndex++

    if (dsIndex < xmlModel.count) {
        CreateObject.create(xmlModel.get(dsIndex).source, root,
    }
}

property int dsIndex
```

The example demonstrates how a model can be used to track created items, and how easy it is to serialize and deserialize such information. This can be used to store a dynamically populated scene such as a set of widgets. In the example, a model was used to track each item.

An alternate solution would be to use the `children` property of the root of a scene to track items. This, however, requires the items themselves to know the source URL to use to re-create them. It also requires us to implement a way to be able to tell dynamically created items apart from

the items that are a part of the original scene, so that we can avoid attempting to serialize and later deserialize any of the original items.

Summary

In this chapter, we have looked at creating QML elements dynamically. This lets us create QML scenes freely, opening the door for user configurable and plug-in based architectures.

The easiest way to dynamically load a QML element is to use a `Loader` element. This acts as a placeholder for the contents being loaded.

For a more dynamic approach, the `Qt.createQmlObject` function can be used to instantiate a string of QML. This approach does, however, have limitations. The full-blown solution is to dynamically create a `Component` using the `Qt.createComponent` function. Objects are then created by calling the `createObject` function of a `Component` .

As bindings and signal connections rely on the existence of an object `id` , or access to the object instantiation, an alternate approach must be used for dynamically created objects. To create a binding, the `Binding` element is used. The `Connections` element makes it possible to connect to signals of a dynamically created object.

One of the challenges of working with dynamically created items is to keep track of them. This can be done using a model. By having a model tracking the dynamically created items, it is possible to implement functions for serialization and deserialization, making it possible to store and restore dynamically created scenes.

JavaScript

JavaScript is the lingua-franca on web client development. It also starts to get traction on web server development mainly by node js. As such it is a well-suited addition as an imperative language onto the side of declarative QML language. QML itself as a declarative language is used to express the user interface hierarchy but is limited to express operational code. Sometimes you need a way to express operations, here JavaScript comes into play.

> **TIP**
>
> There is an open question in the Qt community about the right mixture about QML/JS/Qt C++ in a modern Qt application. The commonly agreed recommended mixture is to limit the JS part of your application to a minimum and do your business logic inside Qt C++ and the UI logic inside QML/JS.

This book pushes the boundaries, which is not always the right mix for a product development and not for everyone. It is important to follow your team skills and your personal taste. In doubt follow the recommendation.

Here a short example of how JS used in QML looks like:

```
Button {
  width: 200
  height: 300
  property bool checked: false
  text: "Click to toggle"
```

```
    // JS function
    function doToggle() {
      checked = !checked
    }

    onClicked: {
      // this is also JavaScript
      doToggle();
      console.log('checked: ' + checked)
    }
  }
```

So JavaScript can come in many places inside QML as a standalone JS function, as a JS module and it can be on every right side of a property binding.

```
import "util.js" as Util // import a pure JS module

Button {
  width: 200
  height: width*2 // JS on the right side of property binding

  // standalone function (not really useful)
  function log(msg) {
    console.log("Button> " + msg);
  }

  onClicked: {
    // this is JavaScript
    log();
    Qt.quit();
  }
}
```

Within QML you declare the user interface, with JavaScript you make it functional. So how much JavaScript should you write? It depends on your

style and how familiar you are with JS development. JS is a loosely typed language, which makes it difficult to spot type defects. Also, functions expect all argument variations, which can be a very nasty bug to spot. The way to spot defects is rigorous unit testing or acceptance testing. So if you develop real logic (not some glue lines of code) in JS you should really start using the test-first approach. In generally mixed teams (Qt/C++ and QML/JS) are very successful when they minimize the amount of JS in the frontend as the domain logic and do the heavy lifting in Qt C++ in the backend. The backend should then be rigorous unit tested so that the frontend developers can trust the code and focus on all these little user interface requirements.

> **TIP**
>
> In general: backend developers are functional driven and frontend developers are user story driven.

Browser/HTML vs Qt Quick/QML

The browser is the runtime to render HTML and execute the Javascript associated with the HTML. Nowadays modern web applications contain much more JavaScript than HTML. The Javascript inside the browser is a standard ECMAScript environment with some additional browser APIs. A typical JS environment inside the browser has a global object named `window` which is used to interact with the browser window (title, location URL, DOM tree etc.) Browsers provide functions to access DOM nodes by their id, class etc. (which were used by jQuery to provide the CSS selectors) and recently also by CSS selectors (`querySelector` , `querySelectorAll`). Additionally, there is a possibility to call a function after a certain amount of time (`setTimeout`) and to call it repeatedly (`setInterval`). Besides these (and other browser APIs), the environment is similar to QML/JS.

Another difference is how JS can appear inside HTML and QML. In HTML, you can execute JS only during the initial page load or in event handlers (e.g. page loaded, mouse pressed). For example, your JS initializes normally on page load, which is comparable to `Component.onCompleted` in QML. By default, you cannot use JS for property bindings in a browser (AngularJS enhances the DOM tree to allow these, but this is far away from standard HTML).

In QML, JS is a much more of a first-class citizen and is much deeper integrated into the QML render tree. Which makes the syntax much more readable. Besides these differences, people who have developed HTML/JS applications should feel at home using QML/JS.

JS Language

This chapter will not give you a general introduction to JavaScript. There are other books out there for a general introduction to JavaScript, please visit this great side on **Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript)** .

On the surface JavaScript is a very common language and does not differ a lot from other languages:

```js
function countDown() {
  for(var i=0; i<10; i++) {
    console.log('index: ' + i)
  }
}

function countDown2() {
  var i=10;
  while( i>0 ) {
    i--;
  }
}
```

But be warned JS has function scope and not block scope as in C++ (see **Functions and function scope (https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Functions_and_function_scope)**).

The statements `if ... else` , `break` , `continue` also work as expected. The switch case can also compare other types and not just integer values:

```js
function getAge(name) {
  // switch over a string
  switch(name) {
  case "father":
    return 58;
  case "mother":
    return 56;
  }
  return unknown;
}
```

JS knows several values which can be false, e.g. `false` , `0` , `""` , `undefined` , `null`). For example, a function returns by default `undefined` . To test for false use the `===` identity operator. The `==` equality operator will do type conversion to test for equality. If possible use the faster and better `===` strict equality operator which will test for identity (see **Comparison operators (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators)**).

Under the hood, javascript has its own ways of doing things. For example arrays:

```js
function doIt() {
  var a = [] // empty arrays
  a.push(10) // addend number on arrays
  a.push("Monkey") // append string on arrays
  console.log(a.length) // prints 2
  a[0] // returns 10
  a[1] // returns Monkey
  a[2] // returns undefined
  a[99] = "String" // a valid assignment
  console.log(a.length) // prints 100
  a[98] // contains the value undefined
}
```

Also for people coming from C++ or Java which are used to an OO language JS just works differently. JS is not purely an OO language it is a so-called prototype based language. Each object has a prototype object. An object is created based on his prototype object. Please read more about this in the book **Javascript the Good Parts by Douglas Crockford (http://javascript.crockford.com)** .

To test some small JS snippets you can use the online **JS Console (http://jsconsole.com)** or just build a little piece of QML code:

```
import QtQuick 2.5

Item {
  function runJS() {
    console.log("Your JS code goes here");
  }
  Component.onCompleted: {
    runJS();
  }
}
```

JS Objects

While working with JS there are some objects and methods which are more frequently used. This is a small collection of them.

- `Math.floor(v)` , `Math.ceil(v)` , `Math.round(v)` - largest, smallest, rounded integer from float
- `Math.random()` - create a random number between 0 and 1
- `Object.keys(o)` - get keys from object (including QObject)
- `JSON.parse(s)` , `JSON.stringify(o)` - conversion between JS object and JSON string
- `Number.toFixed(p)` - fixed precision float
- `Date` - Date manipulation

You can find them also at: **JavaScript reference (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference)**

Here some small and limited examples of how to use JS with QML. They should give you an idea how you can use JS inside QML

Print all keys from QML Item

```
Item {
    id: root
    Component.onCompleted: {
        var keys = Object.keys(root);
        for(var i=0; i<keys.length; i++) {
```

```
            var key = keys[i];
            // prints all properties, signals, functions from o
            console.log(key + ' : ' + root[key]);
        }
    }
}
```

Parse an object to a JSON string and back

```
Item {
    property var obj: {
        key: 'value'
    }

    Component.onCompleted: {
        var data = JSON.stringify(obj);
        console.log(data);
        var obj = JSON.parse(data);
        console.log(obj.key); // > 'value'
    }
}
```

Current Date

```
Item {
    Timer {
        id: timeUpdater
        interval: 100
        running: true
        repeat: true
        onTriggered: {
            var d = new Date();
            console.log(d.getSeconds());
        }
```

```
    }
}
```

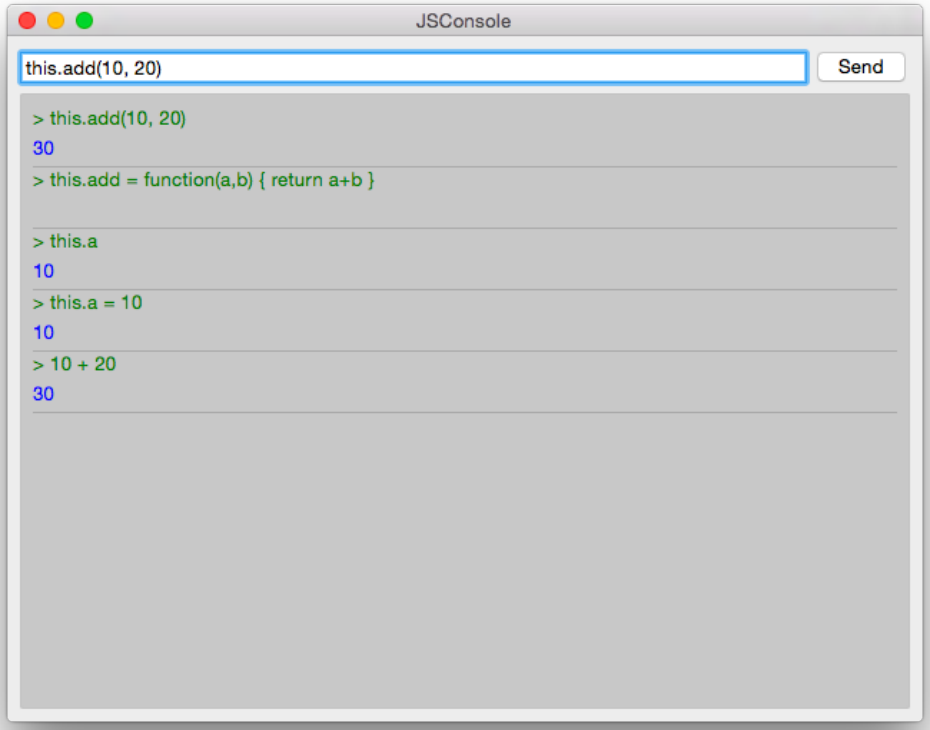
Call a function by name

```
Item {
    id: root

    function doIt() {
        console.log("doIt()")
    }

    Component.onCompleted: {
        // Call using function execution
        root["doIt"]();
        var fn = root["doIt"];
        // Call using JS call method (could pass in a custom th
        fn.call()
    }
}
```

Creating a JS Console

As a little example, we will create a JS console. We need an input field where the user can enter his JS expressions and ideally there should be a list of output results. As this should more look like a desktop application we use the Qt Quick Controls module.

> **TIP**
>
> A JS console inside your next project can be really beneficial for testing. Enhanced with a Quake-Terminal effect it is also good to impress customers. To use it wisely you need to control the scope the JS console evaluates in, e.g. the currently visible screen, the main data model, a singleton core object or all together.



We use Qt Creator to create a Qt Quick UI project using Qt Quick controls. We call the project JSConsole. After the wizard has finished we have already a basic structure for the application with an application window and a menu to exit the application.

For the input, we use a TextField and a Button to send the input for evaluation. The result of the expression evaluation is displayed using a ListView with a ListModel as the model and two labels to display the expression and the evaluated result.

Our application will be split in two files:

- `JSConsole.qml` : the main view of the app
- `jsconsole.js` : the javascript library responsible for evaluating user statements

JSConsole.qml

Application window

```
// JSConsole.qml
import QtQuick
import QtQuick.Controls
import QtQuick.Layouts
import QtQuick.Window
import "jsconsole.js" as Util

ApplicationWindow {
    id: root

    title: qsTr("JSConsole")
    width: 640
    height: 480
    visible: true

    menuBar: MenuBar {
        Menu {
            title: qsTr("File")
            MenuItem {
                text: qsTr("Exit")
                onTriggered: Qt.quit()
            }
        }
    }
```

Form

```
ColumnLayout {
    anchors.fill: parent
    anchors.margins: 9
    RowLayout {
        Layout.fillWidth: true
        TextField {
            id: input
            Layout.fillWidth: true
            focus: true
            onAccepted: {
                // call our evaluation function on root
                root.jsCall(input.text)
            }
        }
        Button {
            text: qsTr("Send")
            onClicked: {
                // call our evaluation function on root
                root.jsCall(input.text)
            }
        }
    }
    Item {
        Layout.fillWidth: true
        Layout.fillHeight: true
        Rectangle {
            anchors.fill: parent
            color: '#333'
            border.color: Qt.darker(color)
            opacity: 0.2
            radius: 2
        }

        ScrollView {
            id: scrollView
            anchors.fill: parent
            anchors.margins: 9
            ListView {
                id: resultView
```

```
                model: ListModel {
                    id: outputModel
                }
                delegate: ColumnLayout {
                    id: delegate
                    required property var model
                    width: ListView.view.width
                    Label {
                        Layout.fillWidth: true
                        color: 'green'
                        text: "> " + delegate.model.expression
                    }
                    Label {
                        Layout.fillWidth: true
                        color: delegate.model.error === "" ? 'b
                        text: delegate.model.error === "" ? ""
                    }
                    Rectangle {
                        height: 1
                        Layout.fillWidth: true
                        color: '#333'
                        opacity: 0.2
                    }
                }
            }
        }
    }
}
```

Calling the library

The evaluation function `jsCall` does the evaluation not by itself this has been moved to a JS module (`jsconsole.js`) for clearer separation.

```
import "jsconsole.js" as Util
```

```
function jsCall(exp) {
    const data = Util.call(exp)
    // insert the result at the beginning of the list
    outputModel.insert(0, data)
}
```

> **TIP**
>
> For safety, we do not use the `eval` function from JS as this would allow the user to modify the local scope. We use the Function constructor to create a JS function on runtime and pass in our scope as this variable. As the function is created every time it does not act as a closure and stores its own scope, we need to use `this.a = 10` to store the value inside this scope of the function. This scope is set by the script to the scope variable.

jsconsole.js

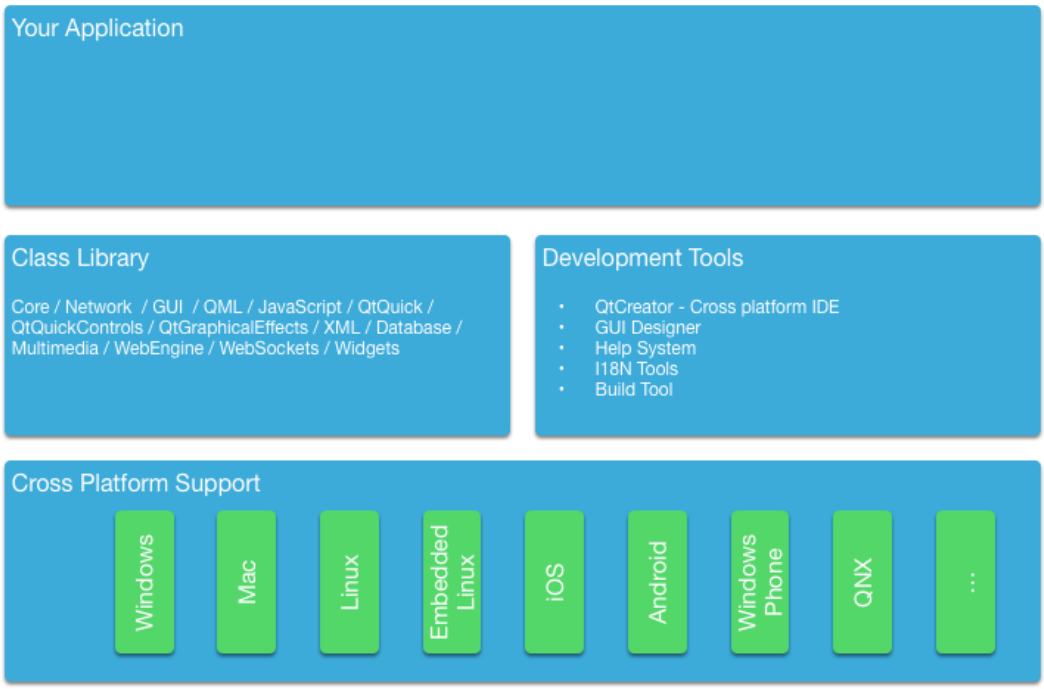
```js
// jsconsole.js
.pragma library

const scope = {
    // our custom scope injected into our function evaluation
}

function call(msg) {
    const exp = msg.toString()
    console.log(exp)
    const data = {
        expression : msg,
        result: "",
        error: ""
    }
```

```
        try {
            const fun = new Function('return (' + exp + ')')
            data.result = JSON.stringify(fun.call(scope), null, 2)
            console.log('scope: ' + JSON.stringify(scope, null, 2),
        } catch(e) {
            console.log(e.toString())
            data.error = e.toString()
        }
        return data
    }
```

The data return from the call function is a JS object with a result, expression and error property: `data: { expression: "", result: "", error: "" }` . We can use this JS object directly inside the ListModel and access it then from the delegate, e.g. `delegate.model.expression` gives us the input expression.

Qt and C++

Qt is a C++ toolkit with an extension for QML and Javascript. There exist many language bindings for Qt, but as Qt itself is developed in C++. The spirit of C++ can be found throughout the classes. In this section, we will look at Qt from a C++ perspective to build a better understanding of how to extend QML with native plugins developed using C++. Through C++, it is possible to extend and control the execution environment provided to QML.



This chapter will, just as Qt, require the reader to have some basic knowledge of C++. Qt does not rely on advanced C++ features and I generally consider the Qt style of C++ to be very readable, so do not worry if you feel that your C++ knowledge is shaky.

Qt C++

Approaching Qt from a C++ direction, you will find that Qt enriches C++ with a number of modern language features enabled through making introspection data available. This is made possible through the use of the `QObject` base class. Introspection data, or metadata, maintains information of the classes at run-time, something that ordinary C++ does not do. This makes it possible to dynamically probe objects for information about such details as their properties and available methods.

Qt uses this meta information to enable a very loosely bound callback concept using signals and slots. Each signal can be connected to any number of slots or even other signals. When a signal is emitted from an object instance, the connected slots are invoked. As the signal emitting object does not need to know anything about the object owning the slot and vice versa, this mechanism is used to create very reusable components with very few inter-component dependencies.

Qt for Python

The introspection features are also used to create dynamic language bindings, making it possible to expose a C++ object instance to QML and making C++ functions callable from Javascript. Other bindings for Qt C++ exist and besides the standard Javascript binding, the official one is the Python binding called **PySide6 (https://www.qt.io/qt-for-python)** .

Cross Platform

In addition to this central concept, Qt makes it possible to develop cross-platform applications using C++. Qt C++ provides a platform abstraction on the different operating systems, which allows the developer to concentrate on the task at hand and not the details of how you open a file on different operating systems. This means you can re-compile the same source code for Windows, OS X, and Linux and Qt takes care of the different OS ways of handling certain things. The end result is natively built applications with the look and feel of the target platform. As the mobile is the new desktop, newer Qt versions can also target a number of mobile platforms using the same source code, e.g. iOS, Android, Jolla, BlackBerry, Ubuntu Phone, Tizen.

When it comes to re-using, not only can source code be re-used but developer skills are also reusable. A team knowing Qt can reach out to far more platforms than a team just focusing on a single platform specific technology and as Qt is so flexible the team can create different system components using the same technology.

For all platform, Qt offers a set of basic types, e.g. strings with full Unicode support, lists, vectors, buffers. It also provides a common abstraction to the target platform's main loop, and cross-platform threading and networking support. The general philosophy is that for an application developer Qt comes with all required functionality included. For domain-specific tasks such as to interface to your native libraries, Qt comes with several helper classes to make this easier.

A Boilerplate Application

The best way to understand Qt is to start from a small example. This application creates a simple `"Hello World!"` string and writes it into a file using Unicode characters.

```
#include <QCoreApplication>
#include <QString>
#include <QFile>
#include <QDir>
#include <QTextStream>
#include <QDebug>


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

    // prepare the message
    QString message("Hello World!");

    // prepare a file in the users home directory named out.txt
    QFile file(QDir::home().absoluteFilePath("out.txt"));
    // try to open the file in write mode
    if(!file.open(QIODevice::WriteOnly)) {
        qWarning() << "Can not open file with write access";
        return -1;
    }
    // as we handle text we need to use proper text codecs
    QTextStream stream(&file);
    // write message to file via the text stream
    stream << message;
```

```
    // do not start the eventloop as this would wait for extern
    // app.exec();

    // no need to close file, closes automatically when scope e
    return 0;
}
```

The example demonstrates the use of file access and the how to write text to a file using text codecs using a text stream. For binary data, there is a cross-platform binary stream called `QDataStream` that takes care of endianess and other details. The different classes we use are included using their class name at the top of the file. You can also include classes using the module and class name e.g. `#include <QtCore/QFile>` . For the lazy, there is also the possibility to include all the classes from a module using `#include <QtCore>` . For instance, in `QtCore` you have the most common classes used for an application that are not UI related. Have a look at the **QtCore class list (http://doc.qt.io/qt-5/qtcore-module.html)** or the **QtCore overview (http://doc.qt.io/qt-5/qtcore-index.html)** .

You build the application using CMake and make. CMake reads a project file, `CMakeLists.txt` and generates a Makefile which is used to build the application. CMake supports other build systems too, for example ninja. The project file is platform independent and CMake has some rules to apply the platform specific settings to the generated makefile. The project can also contain platform scopes for platform-specific rules, which are required in some specific cases.

Here is an example of a simple project file generated by Qt Creator. Notice that Qt attempts to create a file that is compatible with both Qt 5 and Qt 6, as well as various platforms such as Android, OS X and such.

```sh
cmake_minimum_required(VERSION 3.14)
```

```
project(projectname VERSION 0.1 LANGUAGES CXX)

set(CMAKE_INCLUDE_CURRENT_DIR ON)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# QtCreator supports the following variables for Android, which
# Check https://doc.qt.io/qt/deployment-android.html for more i
# They need to be set before the find_package(...) calls below.

#if(ANDROID)
#    set(ANDROID_PACKAGE_SOURCE_DIR "${CMAKE_CURRENT_SOURCE_DIR
#    if (ANDROID_ABI STREQUAL "armeabi-v7a")
#        set(ANDROID_EXTRA_LIBS
#            ${CMAKE_CURRENT_SOURCE_DIR}/path/to/libcrypto.so
#            ${CMAKE_CURRENT_SOURCE_DIR}/path/to/libssl.so)
#    endif()
#endif()

find_package(QT NAMES Qt6 Qt5 COMPONENTS Core Quick REQUIRED)
find_package(Qt${QT_VERSION_MAJOR} COMPONENTS Core Quick REQUIR

set(PROJECT_SOURCES
        main.cpp
        qml.qrc
)

if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
    qt_add_executable(projectname
        MANUAL_FINALIZATION
        ${PROJECT_SOURCES}
    )
else()
    if(ANDROID)
        add_library(projectname SHARED
```

```
            ${PROJECT_SOURCES}
        )
    else()
        add_executable(projectname
          ${PROJECT_SOURCES}
        )
    endif()
endif()

target_compile_definitions(projectname
  PRIVATE $<$<OR:$<CONFIG:Debug>,$<CONFIG:RelWithDebInfo>>:QT_Q
target_link_libraries(projectname
  PRIVATE Qt${QT_VERSION_MAJOR}::Core Qt${QT_VERSION_MAJOR}::Qu

set_target_properties(projectname PROPERTIES
    MACOSX_BUNDLE_GUI_IDENTIFIER my.example.com
    MACOSX_BUNDLE_BUNDLE_VERSION ${PROJECT_VERSION}
    MACOSX_BUNDLE_SHORT_VERSION_STRING ${PROJECT_VERSION_MAJOR}
)

if(QT_VERSION_MAJOR EQUAL 6)
    qt_import_qml_plugins(projectname)
    qt_finalize_executable(projectname)
endif()
```

We will not go into the depths of this file. Just remember Qt uses CMake's `CMakeLists.txt` files to generate platform-specific makefiles, that are then used to build a project. In the build system section we will have a look at more basic, handwritten, CMake files.

The simple code example above just writes the text and exits the application. For a command line tool, this is good enough. For a user interface you need an event loop which waits for user input and somehow schedules draw operations. Here follows the same example now uses a button to trigger the writing.

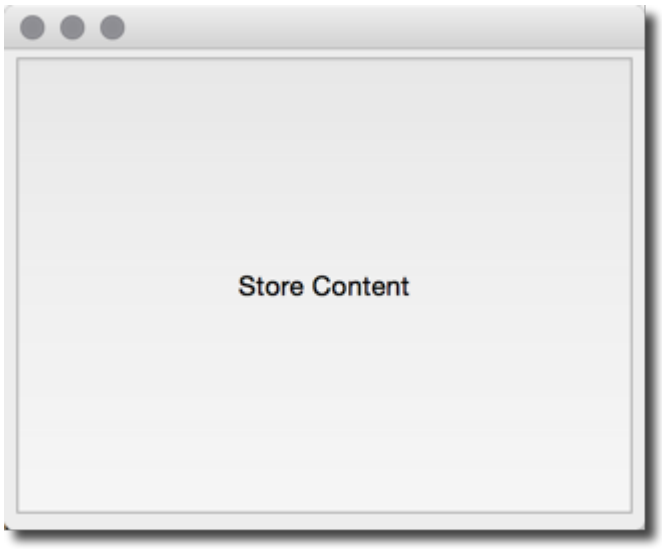
Our `main.cpp` surprisingly got smaller. We moved code into an own class to be able to use Qt's signal and slots for the user input, i.e. to handle the button click. The signal and slot mechanism normally needs an object instance as you will see shortly, but it can also be used with C++ lambdas.

```
#include <QtCore>
#include <QtGui>
#include <QtWidgets>
#include "mainwindow.h"


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

    MainWindow win;
    win.resize(320, 240);
    win.setVisible(true);

    return app.exec();
}
```

In the `main` function we create the application object, a window, and then start the event loop using `exec()`. For now, the application sits in the event loop and waits for user input.

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv); // init application

    // create the ui

    return app.exec(); // execute event loop
}
```

Using Qt, you can build user interfaces in both QML and Widgets. In this book we focus on QML, but in this chapter, we will look at Widgets. This lets us create the program only C++.



The main window itself is a widget. It becomes a top-level window as it does not have any parent. This comes from how Qt sees a user interface as a tree of UI elements. In this case, the main window is the root element, thus becomes a window, while the push button, that is a child of the main window, becomes a widget inside the window.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtWidgets>

class MainWindow : public QMainWindow
{
public:
    MainWindow(QWidget* parent=0);
    ~MainWindow();
public slots:
    void storeContent();
private:
    QPushButton *m_button;
```

```
};

#endif // MAINWINDOW_H
```

Additionally, we define a public slot called `storeContent()` in a custom section in the header file. Slots can be public, protected, or private, and can be called just like any other class method. You may also encounter a `signals` section with a set of signal signatures. These methods should not be called and must not be implemented. Both signals and slots are handled by the Qt meta information system and can be introspected and called dynamically at run-time.

The purpose of the `storeContent()` slot is that it is called when the button is clicked. Let's make that happen!

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    m_button = new QPushButton("Store Content", this);

    setCentralWidget(m_button);
    connect(m_button, &QPushButton::clicked, this, &MainWindow:
}

MainWindow::~MainWindow()
{

}

void MainWindow::storeContent()
{
    qDebug() << "... store content";
    QString message("Hello World!");
    QFile file(QDir::home().absoluteFilePath("out.txt"));
    if(!file.open(QIODevice::WriteOnly)) {
```

```
            qWarning() << "Can not open file with write access";
            return;
        }
        QTextStream stream(&file);
        stream << message;
    }
```

In the main window, we first create the push button and then register the signal `clicked()` with the slot `storeContent()` using the connect method. Every time the signal `clicked` is emitted the slot `storeContent()` is called. And now, the two objects communicate via signal and slots despite not being aware of each other. This is called loose coupling and is made possible using the `QObject` base class which most Qt classes derive from.

The QObject

As described in the introduction, the `QObject` is what enables many of Qt's core functions such as signals and slots. This is implemented through introspection, which is what `QObject` provides. `QObject` is the base class of almost all classes in Qt. Exceptions are value types such as `QColor`, `QString` and `QList`.

A Qt object is a standard C++ object, but with more abilities. These can be divided into two groups: introspection and memory management. The first means that a Qt object is aware of its class name, its relationship to other classes, as well as its methods and properties. The memory management concept means that each Qt object can be the parent of child objects. The parent *owns* the children, and when the parent is destroyed, it is responsible for destroying its children.

The best way of understanding how the `QObject` abilities affect a class is to take a standard C++ class and Qt enables it. The class shown below represents an ordinary such class.

The person class is a data class with a name and gender properties. The person class uses Qt's object system to add meta information to the c++ class. It allows users of a person object to connect to the slots and get notified when the properties get changed.

```
class Person : public QObject
{
    Q_OBJECT // enabled meta object abilities

    // property declarations required for QML
```

```
    Q_PROPERTY(QString name READ name WRITE setName NOTIFY name
    Q_PROPERTY(Gender gender READ gender WRITE setGender NOTIFY

    // enables enum introspections
    Q_ENUM(Gender)

    // makes the type creatable in QML
    QML_ELEMENT

public:
    // standard Qt constructor with parent for memory managemen
    Person(QObject *parent = 0);

    enum Gender { Unknown, Male, Female, Other };

    QString name() const;
    Gender gender() const;

public slots: // slots can be connected to signals, or called
    void setName(const QString &);
    void setGender(Gender);

signals: // signals can be emitted
    void nameChanged(const QString &name);
    void genderChanged(Gender gender);

private:
    // data members
    QString m_name;
    Gender m_gender;
};
```

The constructor passes the parent to the superclass and initializes the members. Qt's value classes are automatically initialized. In this case `QString` will initialize to a null string (`QString::isNull()`) and the gender member will explicitly initialize to the unknown gender.

```
Person::Person(QObject *parent)
    : QObject(parent)
    , m_gender(Person::Unknown)
{
}
```

The getter function is named after the property and is normally a basic `const` function. The setter emits the changed signal when the property has changed. To ensure that the value actually has changed, we insert a guard to compare the current value with the new value. Only when the value differs we assign it to the member variable and emit the changed signal.

```
QString Person::name() const
{
    return m_name;
}

void Person::setName(const QString &name)
{
    if (m_name != name) // guard
    {
        m_name = name;
        emit nameChanged(m_name);
    }
}
```

Having a class derived from `QObject`, we have gained more meta object abilities we can explore using the `metaObject()` method. For example, retrieving the class name from the object.

```
Person* person = new Person();
person->metaObject()->className(); // "Person"
Person::staticMetaObject.className(); // "Person"
```

There are many more features which can be accessed by the `QObject` base class and the meta object. Please check out the `QMetaObject` documentation.

> **TIP**
>
> `QObject`, and the `Q_OBJECT` macro has a lightweight sibling: `Q_GADGET`. The `Q_GADGET` macro can be inserted in the private section of non-`QObject`-derived classes to expose properties and invokable methods. Beware that a `Q_GADGET` object cannot have signals, so the properties cannot provide a change notification signal. Still, this can be useful to provide a QML-like interface to data structures exposed from C++ to QML without invoking the cost of a fully fledged `QObject`.

Build Systems

Building software reliably across different platforms can be a complex task. You will encounter different environments with different compilers, paths, and library variations. The purpose of Qt is to shield the application developer from these cross-platform issues. Qt relies on **CMake (https://cmake.org/)** to convert `CMakeLists.txt` project files to platform specific make files, which then can be built using the platform specific tooling.

> **TIP**
>
> Qt comes with three different build systems. The original Qt build system was called `qmake`. Another Qt specific build system is `QBS` which uses a declarative approach to describing the build sequence. Since version 6, Qt has shifted from `qmake` to CMake as the official build system.

A typical build flow in Qt under Unix would be:

```sh
vim CMakeLists.txt
cmake . // generates Makefile
make
```

With Qt you are encouraged to use shadow builds. A shadow build is a build outside of your source code location. Assume we have a myproject folder with a `CMakeLists.txt` file. The flow would be like this:

```sh
mkdir build
cd build
cmake ..
```

We create a build folder and then call cmake from inside the build folder with the location of our project folder. This will set up the makefile in a way that all build artifacts are stored under the build folder instead of inside our source code folder. This allows us to create builds for different qt versions and build configurations at the same time and also it does not clutter our source code folder which is always a good thing.

When you are using Qt Creator it does these things behind the scenes for you and you do not have to worry about these steps in most cases. For larger projects and for a deeper understanding of the flow, it is recommended that you learn to build your Qt project from the command line to ensure that you have full control over what is happening.

CMake

CMake is a tool created by Kitware. Kitware is very well known for their 3D visualization software VTK and also CMake, the cross-platform makefile generator. It uses a series of `CMakeLists.txt` files to generate platform-specific makefiles. CMake is used by the KDE project and as such has a special relationship with the Qt community and since version 6, it is the preferred way to build Qt projects.

The `CMakeLists.txt` is the file used to store the project configuration. For a simple hello world using Qt Core the project file would look like this:

```
// ensure cmake version is at least 3.16.0
cmake_minimum_required(VERSION 3.16.0)
```

```
// defines a project with a version
project(foundation_tests VERSION 1.0.0 LANGUAGES CXX)

// pick the C++ standard to use, in this case C++17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

// tell CMake to run the Qt tools moc, rcc, and uic automatical
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)

// configure the Qt 6 modules core and test
find_package(Qt6 COMPONENTS Core REQUIRED)
find_package(Qt6 COMPONENTS Test REQUIRED)

// define an executable built from a source file
add_executable(foundation_tests
    tst_foundation.cpp
)

// tell cmake to link the executable to the Qt 6 core and test
target_link_libraries(foundation_tests PRIVATE Qt6::Core Qt6::T
```

This will build a `foundations_tests` executable using `tst_foundation.cpp` and link against the Core and Test libraries from Qt 6. You will find more examples of CMake files in this book, as we use CMake for all C++ based examples.

CMake is a powerful, a complex, tool and it takes some time to get used to the syntax. CMake is very flexible and really shines in large and complex projects.

References

- **CMake Help (http://www.cmake.org/documentation/)** - available online but also as Qt Help format
- **Running CMake (http://www.cmake.org/runningcmake/)**
- **KDE CMake Tutorial (https://techbase.kde.org/Development/Tutorials/CMake)**
- **CMake Book (http://www.kitware.com/products/books/CMakeBook.html)**
- **CMake and Qt (http://www.cmake.org/cmake/help/v3.0/manual/cmake-qt.7.html)**

QMake

QMake is the tool which reads your project file and generates the build file. A project file is a simplified write-down of your project configuration, external dependencies, and your source files. The simplest project file is probably this:

```js
// myproject.pro

SOURCES += main.cpp
```

Here we build an executable application which will have the name `myproject` based on the project file name. The build will only contain the `main.cpp` source file. And by default, we will use the `QtCore` and `QtGui` module for this project. If our project were a QML application we would need to add the `QtQuick` and `QtQml` module to the list:

```js
// myproject.pro

QT += qml quick
```

```
SOURCES += main.cpp
```

Now the build file knows to link against the `QtQml` and `QtQuick` Qt modules. QMake uses the concept of `=`, `+=` and `-=` to assign, add, remove elements from a list of options, respectively. For a pure console build without UI dependencies you would remove the `QtGui` module:

```js
// myproject.pro

QT -= gui

SOURCES += main.cpp
```

When you want to build a library instead of an application, you need to change the build template:

```js
// myproject.pro
TEMPLATE = lib

QT -= gui

HEADERS += utils.h
SOURCES += utils.cpp
```

Now the project will build as a library without UI dependencies and used the `utils.h` header and the `utils.cpp` source file. The format of the library will depend on the OS you are building the project.

Often you will have more complicated setups and need to build a set of projects. For this, qmake offers the `subdirs` template. Assume we would have a mylib and a myapp project. Then our setup could be like this:

```
my.pro
mylib/mylib.pro
mylib/utils.h
mylib/utils.cpp
myapp/myapp.pro
myapp/main.cpp
```

We know already how the mylib.pro and myapp.pro would look like. The my.pro as the overarching project file would look like this:

```
// my.pro
TEMPLATE = subdirs

subdirs = mylib \
    myapp

myapp.depends = mylib
```

This declares a project with two subprojects: `mylib` and `myapp`, where `myapp` depends on `mylib`. When you run qmake on this project file it will generate file a build file for each project in a corresponding folder. When you run the makefile for `my.pro`, all subprojects are also built.

Sometimes you need to do one thing on one platform and another thing on other platforms based on your configuration. For this qmake introduces the concept of scopes. A scope is applied when a configuration option is set to true.

For example, to use a Unix specific utils implementation you could use:

```
unix {
    SOURCES += utils_unix.cpp
} else {
```

```
    SOURCES += utils.cpp
}
```

What it says is if the CONFIG variable contains a Unix option then apply this scope otherwise use the else path. A typical one is to remove the application bundling under mac:

```js
macx {
    CONFIG -= app_bundle
}
```

This will create your application as a plain executable under mac and not as a `.app` folder which is used for application installation.

QMake based projects are normally the number one choice when you start programming Qt applications. There are also other options out there. All have their benefits and drawbacks. We will shortly discuss these other options next.

References

- **QMake Manual (http://doc.qt.io/qt-5//qmake-manual.html)** - Table of contents of the qmake manual
- **QMake Language (http://doc.qt.io/qt-5//qmake-language.html)** - Value assignment, scopes and so like
- **QMake Variables (http://doc.qt.io/qt-5//qmake-variable-reference.html)** - Variables like TEMPLATE, CONFIG, QT is explained here

Common Qt Classes

Most Qt classes are derived from the `QObject` class. It encapsulates the central concepts of Qt. But there are many more classes in the framework. Before we continue looking at QML and how to extend it, we will look at some basic Qt classes that are useful to know about.

The code examples shown in this section are written using the Qt Test library. This way, we can ensure that the code works, without constructing entire programs around it. The `QVERIFY` and `QCOMPARE` functions from the test library to assert a certain condition. We will use `{}` scopes to avoid name collisions. Don't let this confuse you.

QString

In general, text handling in Qt is Unicode based. For this, you use the `QString` class. It comes with a variety of great functions which you would expect from a modern framework. For 8-bit data, you would use normally the `QByteArray` class and for ASCII identifiers the `QLatin1String` to preserve memory. For a list of strings you can use a `QList<QString>` or simply the `QStringList` class (which is derived from `QList<QString>`).

Below are some examples of how to use the `QString` class. QString can be created on the stack but it stores its data on the heap. Also when assigning one string to another, the data will not be copied - only a reference to the data. So this is really cheap and lets the developer concentrate on the code and not on the memory handling. `QString` uses reference counters to know when the data can be safely deleted. This

feature is called **Implicit Sharing (http://doc.qt.io/qt-6/implicit-sharing.html)** and it is used in many Qt classes.

```
QString data("A,B,C,D"); // create a simple string
// split it into parts
QStringList list = data.split(",");
// create a new string out of the parts
QString out = list.join(",");
// verify both are the same
QVERIFY(data == out);
// change the first character to upper case
QVERIFY(QString("A") == out[0].toUpper());
```

Below you can see how to convert a number to a string and back. There are also conversion functions for float or double and other types. Just look for the function in the Qt documentation used here and you will find the others.

```
// create some variables
int v = 10;
int base = 10;
// convert an int to a string
QString a = QString::number(v, base);
// and back using and sets ok to true on success
bool ok(false);
int v2 = a.toInt(&ok, base);
// verify our results
QVERIFY(ok == true);
QVERIFY(v = v2);
```

Often in a text, you need to have parameterized text. One option could be to use `QString("Hello" + name)` but a more flexible method is the `arg` marker approach. It preserves the order also during translation when the order might change.

```
// create a name
QString name("Joe");
// get the day of the week as string
QString weekday = QDate::currentDate().toString("dddd");
// format a text using paramters (%1, %2)
QString hello = QString("Hello %1. Today is %2.").arg(name).arg
// This worked on Monday. Promise!
if(Qt::Monday == QDate::currentDate().dayOfWeek()) {
    QCOMPARE(QString("Hello Joe. Today is Monday."), hello);
} else {
    QVERIFY(QString("Hello Joe. Today is Monday.") !=  hello);
}
```

Sometimes you want to use Unicode characters directly in your code. For this, you need to remember how to mark them for the `QChar` and `QString` classes.

```
// Create a unicode character using the unicode for smile :-)
QChar smile(0x263A);
// you should see a :-) on you console
qDebug() << smile;
// Use a unicode in a string
QChar smile2 = QString("\u263A").at(0);
QVERIFY(smile == smile2);
// Create 12 smiles in a vector
QVector<QChar> smilies(12);
smilies.fill(smile);
// Can you see the smiles
qDebug() << smilies;
```

This gives you some examples of how to easily treat Unicode aware text in Qt. For non-Unicode, the `QByteArray` class also has many helper functions for conversion. Please read the Qt documentation for `QString` as it contains tons of good examples.

Sequential Containers

A list, queue, vector or linked-list is a sequential container. The mostly used sequential container is the `QList` class. It is a template based class and needs to be initialized with a type. It is also implicit shared and stores the data internally on the heap. All container classes should be created on the stack. Normally you never want to use `new QList<T>()` , which means never use `new` with a container.

The `QList` is as versatile as the `QString` class and offers a great API to explore your data. Below is a small example of how to use and iterate over a list using some new C++ 11 features.

```
// Create a simple list of ints using the new C++11 initializat
// for this you need to add "CONFIG += c++11" to your pro file.
QList<int> list{1,2};

// append another int
list << 3;

// We are using scopes to avoid variable name clashes

{ // iterate through list using Qt for each
    int sum(0);
    foreach (int v, list) {
        sum += v;
    }
    QVERIFY(sum == 6);
}
{ // iterate through list using C++ 11 range based loop
    int sum = 0;
    for(int v : list) {
        sum+= v;
```

```
    }
    QVERIFY(sum == 6);
}

{ // iterate through list using JAVA style iterators
    int sum = 0;
    QListIterator<int> i(list);

    while (i.hasNext()) {
        sum += i.next();
    }
    QVERIFY(sum == 6);
}

{ // iterate through list using STL style iterator
    int sum = 0;
    QList<int>::iterator i;
    for (i = list.begin(); i != list.end(); ++i) {
        sum += *i;
    }
    QVERIFY(sum == 6);
}


// using std::sort with mutable iterator using C++11
// list will be sorted in descending order
std::sort(list.begin(), list.end(), [](int a, int b) { return a
QVERIFY(list == QList<int>({3,2,1}));


int value = 3;
{ // using std::find with const iterator
    QList<int>::const_iterator result = std::find(list.constBeg
    QVERIFY(*result == value);
}

{ // using std::find using C++ lambda and C++ 11 auto variable
    auto result = std::find_if(list.constBegin(), list.constBeg
    QVERIFY(*result == value);
}
```

Associative Containers

A map, a dictionary, or a set are examples of associative containers. They store a value using a key. They are known for their fast lookup. We demonstrate the use of the most used associative container the `QHash` also demonstrating some new C++ 11 features.

```
QHash<QString, int> hash({{"b",2},{"c",3},{"a",1}});
qDebug() << hash.keys(); // a,b,c - unordered
qDebug() << hash.values(); // 1,2,3 - unordered but same as ord

QVERIFY(hash["a"] == 1);
QVERIFY(hash.value("a") == 1);
QVERIFY(hash.contains("c") == true);

{ // JAVA iterator
    int sum =0;
    QHashIterator<QString, int> i(hash);
    while (i.hasNext()) {
        i.next();
        sum+= i.value();
        qDebug() << i.key() << " = " << i.value();
    }
    QVERIFY(sum == 6);
}

{ // STL iterator
    int sum = 0;
    QHash<QString, int>::const_iterator i = hash.constBegin();
    while (i != hash.constEnd()) {
        sum += i.value();
        qDebug() << i.key() << " = " << i.value();
        i++;
    }
```

```
    QVERIFY(sum == 6);
}

hash.insert("d", 4);
QVERIFY(hash.contains("d") == true);
hash.remove("d");
QVERIFY(hash.contains("d") == false);

{ // hash find not successfull
    QHash<QString, int>::const_iterator i = hash.find("e");
    QVERIFY(i == hash.end());
}

{ // hash find successfull
    QHash<QString, int>::const_iterator i = hash.find("c");
    while (i != hash.end()) {
        qDebug() << i.value() << " = " << i.key();
        i++;
    }
}

// QMap
QMap<QString, int> map({{"b",2},{"c",2},{"a",1}});
qDebug() << map.keys(); // a,b,c - ordered ascending

QVERIFY(map["a"] == 1);
QVERIFY(map.value("a") == 1);
QVERIFY(map.contains("c") == true);

// JAVA and STL iterator work same as QHash
```

File IO

It is often required to read and write from files. `QFile` is actually a `QObject` but in most cases, it is created on the stack. `QFile` contains signals to inform the user when data can be read. This allows reading chunks of data asynchronously until the whole file is read. For

convenience, it also allows reading data in blocking mode. This should only be used for smaller amounts of data and not large files. Luckily we only use small amounts of data in these examples.

Besides reading raw data from a file into a `QByteArray` you can also read data types using the `QDataStream` and Unicode string using the `QTextStream`. We will show you how.

```
QStringList data({"a", "b", "c"});
{ // write binary files
    QFile file("out.bin");
    if(file.open(QIODevice::WriteOnly)) {
        QDataStream stream(&file);
        stream << data;
    }
}
{ // read binary file
    QFile file("out.bin");
    if(file.open(QIODevice::ReadOnly)) {
        QDataStream stream(&file);
        QStringList data2;
        stream >> data2;
        QCOMPARE(data, data2);
    }
}
{ // write text file
    QFile file("out.txt");
    if(file.open(QIODevice::WriteOnly)) {
        QTextStream stream(&file);
        QString sdata = data.join(",");
        stream << sdata;
    }
}
{ // read text file
    QFile file("out.txt");
    if(file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        QStringList data2;
```

```
            QString sdata;
            stream >> sdata;
            data2 = sdata.split(",");
            QCOMPARE(data, data2);
        }
    }
```

More Classes

Qt is a rich application framework. As such it has thousands of classes. It takes some time to get used to all of these classes and how to use them. Luckily Qt has a very good documentation with many useful examples includes. Most of the time you search for a class and the most common use cases are already provided as snippets. Which means you just copy and adapt these snippets. Also, Qt's examples in the Qt source code are a great help. Make sure you have them available and searchable to make your life more productive. Do not waste time. The Qt community is always helpful. When you ask, it is very helpful to ask exact questions and provide a simple example which displays your needs. This will drastically improve the response time of others. So invest a little bit of time to make the life of others who want to help you easier 😃.

Here some classes whose documentation the author thinks are a must read:

- **QObject**, **QString**, **QByteArray**
- **QFile**, **QDir**, **QFileInfo**, **QIODevice**
- **QTextStream**, **QDataStream**
- **QDebug**, **QLoggingCategory**
- **QTcpServer**, **QTcpSocket**, **QNetworkRequest**, **QNetworkReply**
- **QAbstractItemModel**, **QRegularExpression**
- **QList**, **QHash**

- **QThread** , **QProcess**
- **QJsonDocument** , **QJSValue**

That should be enough for the beginning.

Models in C++

One of the most common ways to integrate C++ and QML is through *models*. A *model* provides data to a *view* such as `ListViews`, `GridView`, `PathViews`, and other views which take a model and create an instance of a delegate for each entry in the model. The view is smart enough to only create these instances which are visible or in the cache range. This makes it possible to have large models with tens of thousands of entries but still have a very slick user interface. The delegate acts like a template to be rendered with the model entries data. So in summary: a view renders entries from the model using a delegate as a template. The model is a data provider for views.

When you do not want to use C++ you can also define models in pure QML. You have several ways to provide a model for the view. For handling of data coming from C++ or a large amount of data, the C++ model is more suitable than this pure QML approach. But often you only need a few entries then these QML models are well suited.

```
ListView {
    // using a integer as model
    model: 5
    delegate: Text { text: 'index: ' + index }
}

ListView {
    // using a JS array as model
    model: ['A', 'B', 'C', 'D', 'E']
    delegate: Text { 'Char['+ index +']: ' + modelData }
}
```

```
ListView {
    // using a dynamic QML ListModel as model
    model: ListModel {
        ListElement { char: 'A' }
        ListElement { char: 'B' }
        ListElement { char: 'C' }
        ListElement { char: 'D' }
        ListElement { char: 'E' }
    }
    delegate: Text { 'Char['+ index +']: ' + model.char }
}
```

The QML views know how to handle these different types of models. For models coming from the C++ world, the view expects a specific protocol to be followed. This protocol is defined in the API defined in `QAbstractItemModel`, together with documentation describing the dynamic behavior. The API was developed for driving views in the desktop widget world and is flexible enough to act as a base for trees, or multi-column tables as well as lists. In QML, we usually use either the list variant of the API, `QAbstractListModel` or, for the `TableView` element, the table variant of the API, `QAbstractTableModel`. The API contains some functions that have to be implemented and some optional ones that extend the capabilities of the model. The optional parts mostly handle the dynamic use cases for changing, adding or deleting data.

A simple model

A typical QML C++ model derives from `QAbstractListModel` and implements at least the `data` and `rowCount` function. In the example below, we will use a series of SVG color names provided by the `QColor` class and display them using our model. The data is stored into a `QList<QString>` data container.

Our `DataEntryModel` is derived form `QAbstractListModel` and implements the mandatory functions. We can ignore the parent in `rowCount` as this is only used in a tree model. The `QModelIndex` class provides the row and column information for the cell, for which the view wants to retrieve data. The view is pulling information from the model on a row/column and role-based. The `QAbstractListModel` is defined in `QtCore` but `QColor` in `QtGui`. That is why we have the additional `QtGui` dependency. For QML applications it is okay to depend on `QtGui` but it should normally not depend on `QtWidgets`.

```
#ifndef DATAENTRYMODEL_H
#define DATAENTRYMODEL_H

#include <QtCore>
#include <QtGui>

class DataEntryModel : public QAbstractListModel
{
    Q_OBJECT
public:
    explicit DataEntryModel(QObject *parent = 0);
    ~DataEntryModel();

public: // QAbstractItemModel interface
    virtual int rowCount(const QModelIndex &parent) const;
    virtual QVariant data(const QModelIndex &index, int role) c
private:
    QList<QString> m_data;
};

#endif // DATAENTRYMODEL_H
```

On the implementation side, the most complex part is the data function. We first need to make a range check to ensure that we've been provided a valid index. Then we check that the display role is supported. Each item in

a model can have multiple display roles defining various aspects of the data contained. The `Qt::DisplayRole` is the default text role a view will ask for. There is a small set of default roles defined in Qt which can be used, but normally a model will define its own roles for clarity. In the example, all calls which do not contain the display role are ignored at the moment and the default value `QVariant()` is returned.

```
#include "dataentrymodel.h"

DataEntryModel::DataEntryModel(QObject *parent)
    : QAbstractListModel(parent)
{
    // initialize our data (QList<QString>) with a list of colo
    m_data = QColor::colorNames();
}

DataEntryModel::~DataEntryModel()
{
}

int DataEntryModel::rowCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    // return our data count
    return m_data.count();
}

QVariant DataEntryModel::data(const QModelIndex &index, int rol
{
    // the index returns the requested row and column informati
    // we ignore the column and only use the row information
    int row = index.row();

    // boundary check for the row
    if(row < 0 || row >= m_data.count()) {
        return QVariant();
    }
```

```
    // A model can return data for different roles.
    // The default role is the display role.
    // it can be accesses in QML with "model.display"
    switch(role) {
        case Qt::DisplayRole:
            // Return the color name for the particular row
            // Qt automatically converts it to the QVariant typ
            return m_data.value(row);
    }

    // The view asked for other data, just return an empty QVar
    return QVariant();
}
```

The next step would be to register the model with QML using the `qmlRegisterType` call. This is done inside the `main.cpp` before the QML file was loaded.

```
#include <QtGui>
#include <QtQml>

#include "dataentrymodel.h"

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

    // register the type DataEntryModel
    // under the url "org.example" in version 1.0
    // under the name "DataEntryModel"
    qmlRegisterType<DataEntryModel>("org.example", 1, 0, "DataE

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}
```

Now you can access the `DataEntryModel` using the QML import statement `import org.example 1.0` and use it just like other QML item `DataEntryModel {}` .

We use this in this example to display a simple list of color entries.

```
import org.example 1.0

ListView {
    id: view
    anchors.fill: parent
    model: DataEntryModel {}
    delegate: ListDelegate {
        // use the defined model role "display"
        text: model.display
    }
    highlight: ListHighlight { }
}
```

The `ListDelegate` is a custom type to display some text. The `ListHighlight` is just a rectangle. The code was extracted to keep the example compact.

The view can now display a list of strings using the C++ model and the display property of the model. It is still very simple, but already usable in QML. Normally the data is provided from outside the model and the model would act as an interface to the view.

> **TIP**
>
> To expose a table of data instead of a list, the `QAbstractTableModel` is used. The only difference compared to implementing a `QAbstractListModel` is that you must also provide the `columnCount` method.

More Complex Data

In reality, the model data is often much more complex. So there is a need to define custom roles so that the view can query other data via properties. For example, the model could provide not only the color as a hex string but maybe also the hue, saturation, and brightness from the HSV color model as “model.hue”, “model.saturation” and “model.brightness” in QML.

```
#ifndef ROLEENTRYMODEL_H
#define ROLEENTRYMODEL_H

#include <QtCore>
#include <QtGui>

class RoleEntryModel : public QAbstractListModel
{
    Q_OBJECT
public:
    // Define the role names to be used
    enum RoleNames {
        NameRole = Qt::UserRole,
        HueRole = Qt::UserRole+2,
        SaturationRole = Qt::UserRole+3,
        BrightnessRole = Qt::UserRole+4
    };

    explicit RoleEntryModel(QObject *parent = 0);
    ~RoleEntryModel();

    // QAbstractItemModel interface
public:
    virtual int rowCount(const QModelIndex &parent) const overr
    virtual QVariant data(const QModelIndex &index, int role) c
```

```
protected:
    // return the roles mapping to be used by QML
    virtual QHash<int, QByteArray> roleNames() const override;
private:
    QList<QColor> m_data;
    QHash<int, QByteArray> m_roleNames;
};

#endif // ROLEENTRYMODEL_H
```

In the header, we added the role mapping to be used for QML. When QML tries now to access a property from the model (e.g. “model.name”) the listview will lookup the mapping for “name” and ask the model for data using the `NameRole`. User-defined roles should start with `Qt::UserRole` and need to be unique for each model.

```
#include "roleentrymodel.h"

RoleEntryModel::RoleEntryModel(QObject *parent)
    : QAbstractListModel(parent)
{
    // Set names to the role name hash container (QHash<int, QB
    // model.name, model.hue, model.saturation, model.brightnes
    m_roleNames[NameRole] = "name";
    m_roleNames[HueRole] = "hue";
    m_roleNames[SaturationRole] = "saturation";
    m_roleNames[BrightnessRole] = "brightness";

    // Append the color names as QColor to the data list (QList
    for(const QString& name : QColor::colorNames()) {
        m_data.append(QColor(name));
    }

}

RoleEntryModel::~RoleEntryModel()
{
```

```
}

int RoleEntryModel::rowCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return m_data.count();
}

QVariant RoleEntryModel::data(const QModelIndex &index, int rol
{
    int row = index.row();
    if(row < 0 || row >= m_data.count()) {
        return QVariant();
    }
    const QColor& color = m_data.at(row);
    qDebug() << row << role << color;
    switch(role) {
    case NameRole:
        // return the color name as hex string (model.name)
        return color.name();
    case HueRole:
        // return the hue of the color (model.hue)
        return color.hueF();
    case SaturationRole:
        // return the saturation of the color (model.saturation
        return color.saturationF();
    case BrightnessRole:
        // return the brightness of the color (model.brightness
        return color.lightnessF();
    }
    return QVariant();
}

QHash<int, QByteArray> RoleEntryModel::roleNames() const
{
    return m_roleNames;
}
```

The implementation now has changed only in two places. First in the initialization. We now initialize the data list with QColor data types. Additionally, we define our role name map to be accessible for QML. This map is returned later in the `::roleNames` function.

The second change is in the `::data` function. We extend the switch to cover the other roles (e.g hue, saturation, brightness). There is no way to return an SVG name from a color, as a color can take any color and SVG names are limited. So we skip this. Storing the names would require to create a structure `struct { QColor, QString }` to be able to identify the named color.

After registering the type we can use the model and its entries in our user interface.

```
ListView {
    id: view
    anchors.fill: parent
    model: RoleEntryModel {}
    focus: true
    delegate: ListDelegate {
        text: 'hsv(' +
              Number(model.hue).toFixed(2) + ',' +
              Number(model.saturation).toFixed() + ',' +
              Number(model.brightness).toFixed() + ')'
        color: model.name
    }
    highlight: ListHighlight { }
}
```

We convert the returned type to a JS number type to be able to format the number using fixed-point notation. The code would also work without the Number call (e.g. plain `model.saturation.toFixed(2)`). Which format to choose, depends how much you trust the incoming data.

Dynamic Data

Dynamic data covers the aspects of inserting, removing and clearing the data from the model. The `QAbstractListModel` expect a certain behavior when entries are removed or inserted. The behavior is expressed in signals which need to be called before and after the manipulation. For example to insert a row into a model you need first to emit the signal `beginInsertRows`, then manipulate the data and then finally emit `endInsertRows`.

We will add the following functions to our headers. These functions are declared using `Q_INVOKABLE` to be able to call them from QML. Another way would be to declare them as public slots.

```
// inserts a color at the index (0 at begining, count-1 at end)
Q_INVOKABLE void insert(int index, const QString& colorValue);
// uses insert to insert a color at the end
Q_INVOKABLE void append(const QString& colorValue);
// removes a color from the index
Q_INVOKABLE void remove(int index);
// clear the whole model (e.g. reset)
Q_INVOKABLE void clear();
```

Additionally, we define a `count` property to get the size of the model and a `get` method to get a color at the given index. This is useful when you would like to iterate over the model content from QML.

```
// gives the size of the model
Q_PROPERTY(int count READ count NOTIFY countChanged)
// gets a color at the index
Q_INVOKABLE QColor get(int index);
```

The implementation for insert checks first the boundaries and if the given value is valid. Only then do we begin inserting the data.

```
void DynamicEntryModel::insert(int index, const QString &colorV
{
    if(index < 0 || index > m_data.count()) {
        return;
    }
    QColor color(colorValue);
    if(!color.isValid()) {
        return;
    }
    // view protocol (begin => manipulate => end]
    emit beginInsertRows(QModelIndex(), index, index);
    m_data.insert(index, color);
    emit endInsertRows();
    // update our count property
    emit countChanged(m_data.count());
}
```

Append is very simple. We reuse the insert function with the size of the model.

```
void DynamicEntryModel::append(const QString &colorValue)
{
    insert(count(), colorValue);
}
```

Remove is similar to insert but it calls according to the remove operation protocol.

```
void DynamicEntryModel::remove(int index)
{
    if(index < 0 || index >= m_data.count()) {
```

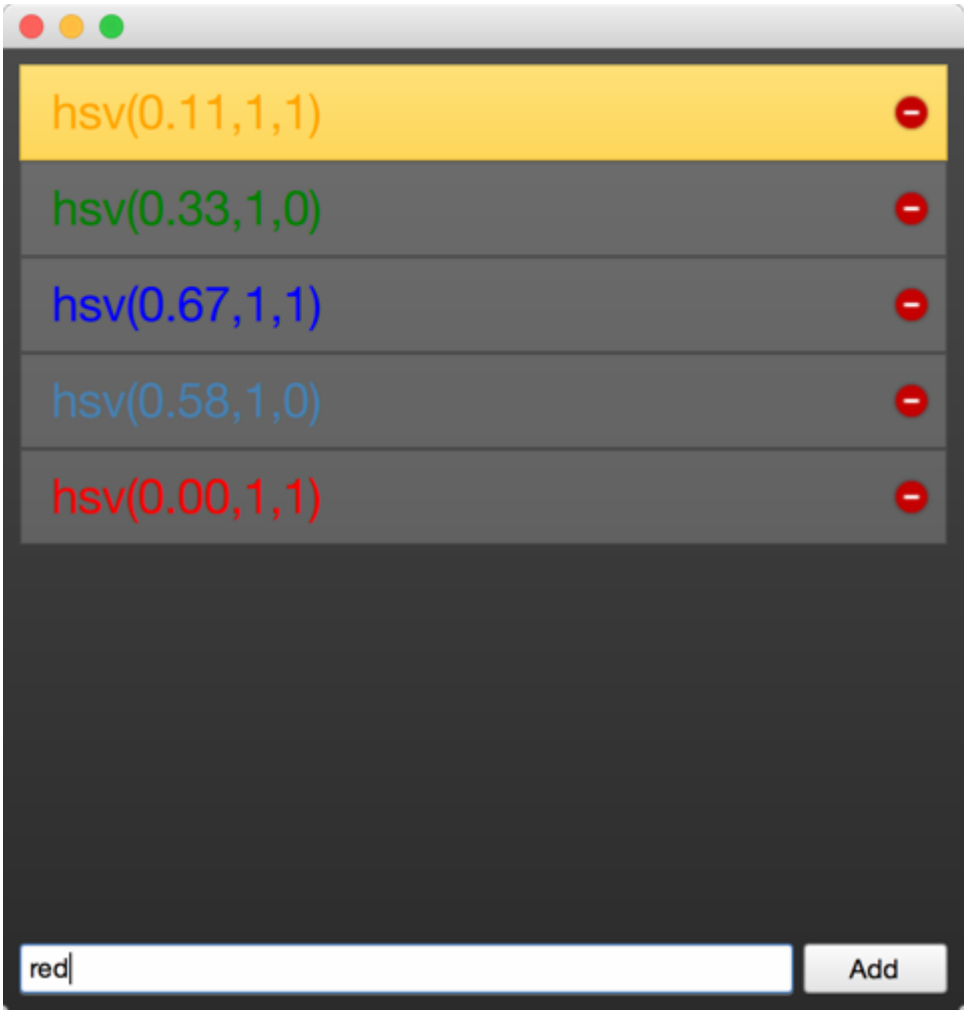
```
        return;
    }
    emit beginRemoveRows(QModelIndex(), index, index);
    m_data.removeAt(index);
    emit endRemoveRows();
    // do not forget to update our count property
    emit countChanged(m_data.count());
}
```

The helper function `count` is trivial. It just returns the data count. The `get` function is also quite simple.

```
QColor DynamicEntryModel::get(int index)
{
    if(index < 0 || index >= m_data.count()) {
        return QColor();
    }
    return m_data.at(index);
}
```

You need to be careful that you only return a value which QML understands. If it is not one of the basic QML types or types known to QML you need to register the type first with `qmlRegisterType` or `qmlRegisterUncreatableType`. You use `qmlRegisterUncreatableType` if the user shall not be able to instantiate its own object in QML.

Now you can use the model in QML and insert, append, remove entries from the model. Here is a small example which allows the user to enter a color name or color hex value and the color is then appended onto the model and shown in the list view. The red circle on the delegate allows the user to remove this entry from the model. After the entry is to remove the list view is notified by the model and updates its content.



And here is the QML code. You find the full source code also in the assets for this chapter. The example uses the `QtQuick.Controls` and `QtQuick.Layout` module to make the code more compact. These controls module provides a set of desktop related UI elements in Qt Quick and the layouts module provides some very useful layout managers.

```
import QtQuick
import QtQuick.Window
import QtQuick.Controls
import QtQuick.Layouts

// our module
import org.example 1.0
```

```
Window {
    visible: true
    width: 480
    height: 480

    Background { // a dark background
        id: background
    }

    // our dyanmic model
    DynamicEntryModel {
        id: dynamic
        onCountChanged: {
            // we print out count and the last entry when count
            print('new count: ' + dynamic.count)
            print('last entry: ' + dynamic.get(dynamic.count -
        }
    }

    ColumnLayout {
        anchors.fill: parent
        anchors.margins: 8
        ScrollView {
            Layout.fillHeight: true
            Layout.fillWidth: true
            ListView {
                id: view
                // set our dynamic model to the views model pro
                model: dynamic
                delegate: ListDelegate {
                    required property var model
                    width: ListView.view.width
                    // construct a string based on the models p
                    text: 'hsv(' +
                          Number(model.hue).toFixed(2) + ',' +
                          Number(model.saturation).toFixed() +
                          Number(model.brightness).toFixed() +
                    // sets the font color of our custom delega
                    color: model.name
```

```
            onClicked: {
                // make this delegate the current item
                view.currentIndex = model.index
                view.focus = true
            }
            onRemove: {
                // remove the current entry from the mo
                dynamic.remove(model.index)
            }
        }
        highlight: ListHighlight { }
        // some fun with transitions :-)
        add: Transition {
            // applied when entry is added
            NumberAnimation {
                properties: "x"; from: -view.width;
                duration: 250; easing.type: Easing.InCi
            }
            NumberAnimation { properties: "y"; from: vi
                duration: 250; easing.type: Easing.InCi
            }
        }
        remove: Transition {
            // applied when entry is removed
            NumberAnimation {
                properties: "x"; to: view.width;
                duration: 250; easing.type: Easing.InBo
            }
        }
        displaced: Transition {
            // applied when entry is moved
            // (e.g because another element was removed
            SequentialAnimation {
                // wait until remove has finished
                PauseAnimation { duration: 250 }
                NumberAnimation { properties: "y"; dura
                }
            }
        }
    }
```

```
            }
            TextEntry {
                id: textEntry
                onAppend: function (color) {
                    // called when the user presses return on the t
                    // or clicks the add button
                    dynamic.append(color)
                }

                onUp: {
                    // called when the user presses up while the te
                    view.decrementCurrentIndex()
                }
                onDown: {
                    // same for down
                    view.incrementCurrentIndex()
                }
            }
        }
    }
```

Model view programming is one of the more complex development tasks in Qt. It is one of the very few classes where you have to implement an interface as a normal application developer. All other classes you just use normally. The sketching of models should always start on the QML side. You should envision how your users would use your model inside QML. For this, it is often a good idea to create a prototype first using the `ListModel` to see how this best works in QML. This is also true when it comes to defining QML APIs. Making data available from C++ to QML is not only a technology boundary it is also a programming paradigm change from imperative to declarative style programming. So be prepared for some setbacks and aha moments:-).

Extending QML with C++

Creating application using only QML can sometimes be limiting. The QML run-time is developed using C++, and the run-time environment can be extended, making it possible to make use of the full performance and freedom of the surrounding system.

Understanding the QML Run-time

When running QML, it is being executed inside of a run-time environment. The run-time is implemented in C++ in the `QtQml` module. It consists of an engine, responsible for the execution of QML, contexts, holding global properties accessible for each component, and components - QML elements that can be instantiated from QML.

```
#include <QtGui>
#include <QtQml>

int main(int argc, char **argv)
{
    QGuiApplication app(argc, argv);
    QUrl source(QStringLiteral("qrc:/main.qml"));
    QQmlApplicationEngine engine;
    engine.load(source);
    return app.exec();
}
```

In the example, the `QGuiApplication` encapsulates all that is related to the application instance (e.g. application name, command line arguments and managing the event loop). The `QQmlApplicationEngine` manages the hierarchical order of contexts and components. It requires typical a QML file to be loaded as the starting point of your application. In this case, it is a `main.qml` containing a window and a text type.

> **TIP**
>
> Loading a `main.qml` with a simple `Item` as the root type through the `QmlApplicationEngine` will not show anything on your display, as it requires a window to manage a surface for rendering. The engine is capable of loading QML code which does not contain any user interface (e.g. plain objects). Because of this, it does not create a window for you by default. The `qml` runtime will internally first check if the main QML file contains a window as a root item and if not create one for you and set the root item as a child to the newly created window.

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 512
    height: 300

    Text {
        anchors.centerIn: parent
        text: "Hello World!"
    }
}
```

In the QML file we declare our dependencies here it is `QtQuick` and `QtQuick.Window` . These declarations will trigger a lookup for these modules in the import paths and on success will load the required plugins by the engine. The newly loaded types will then be made available to the QML environment through a declaration in a a `qmldir` file representing the report.

It is also possible to shortcut the plugin creation by adding our types directly to the engine in our `main.cpp`. Here we assume we have a `CurrentTime`, which is a class based on the `QObject` base class.

```
QQmlApplicationEngine engine();

qmlRegisterType<CurrentTime>("org.example", 1, 0, "CurrentTime"

engine.load(source);
```

Now we can also use the `CurrentTime` type within our QML file.

```
import org.example 1.0

CurrentTime {
    // access properties, functions, signals
}
```

If we don't need to be able to instantiate the new class from QML, we can use context properties to expose C++ objects into QML, e.g.

```
QScopedPointer<CurrentTime> current(new CurrentTime());

QQmlApplicationEngine engine();

engine.rootContext().setContextProperty("current", current.valu

engine.load(source);
```

> **TIP**
>
> Do not mix up `setContextProperty()` and `setProperty()`. The first one sets a context property on a qml context, and `setProperty()` sets a dynamic property value on a `QObject` and will not help you.

Now you can use the current property everywhere in your application. It is available everywhere in the QML code thanks to context inheritance. The `current` object is registered in the outermost root context, which is inherited everywhere.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 512
    height: 300

    Component.onCompleted: {
        console.log('current: ' + current)
    }
}
```

Here are the different ways you can extend QML in general:

- Context properties - `setContextProperty()`
- Register type with engine - calling `qmlRegisterType` in your `main.cpp`
- QML extension plugins - maximum flexibility, to be discussed next

Context properties are easy to use for small applications. They do not require any effort you just expose your system API with kind of global

objects. It is helpful to ensure there will be no naming conflicts (e.g. by using a special character for this (`$`) for example `$.currentTime`). `$` is a valid character for JS variables.

Registering QML types allows the user to control the lifecycle of a C++ object from QML. This is not possible with the context properties. Also, it does not pollute the global namespace. Still all types need to be registered first and by this, all libraries need to be linked on application start, which in most cases is not really a problem.

The most flexible system is provided by the **QML extension plugins**. They allow you to register types in a plugin which is loaded when the first QML file calls the import identifier. Also by using a QML singleton, there is no need to pollute the global namespace anymore. Plugins allow you to reuse modules across projects, which comes quite handy when you do more than one project with Qt.

Going back to our simple example `main.qml` file:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 512
    height: 300

    Text {
        anchors.centerIn: parent
        text: "Hello World!"
    }
}
```

When we import the `QtQuick` and `QtQuick.Window` , what we do is that we tell the QML run-time to find the corresponding QML extension

plugins and load them. This is done by the QML engine by looking for these modules in the QML import paths. The newly loaded types will then be made available to the QML environment.

For the remainder of this chapter will focus on the QML extension plugins. As they provide the greatest flexibility and reuse.

Plugin Content

A plugin is a library with a defined interface, which is loaded on demand. This differs from a library as a library is linked and loaded on startup of the application. In the QML case, the interface is called `QQmlExtensionPlugin`. There are two methods interesting for us `initializeEngine()` and `registerTypes()`. When the plugin is loaded first the `initializeEngine()` is called, which allows us to access the engine to expose plugin objects to the root context. In the majority, you will only use the `registerTypes()` method. This allows you to register your custom QML types with the engine on the provided URL.

Let us explore by building a small `FileIO` utility class. It would let you read and write text files from QML. A first iteration could look like this in a mocked QML implementation.

```
// FileIO.qml (good)
QtObject {
    function write(path, text) {};
    function read(path) { return "TEXT" }
}
```

This is a pure QML implementation of a possible C++ based QML API. We use this to explore the API. We see we need a `read` and a `write` function. We also see that the `write` function takes a `path` and a `text`, while the `read` function takes a `path` and returns a `text`. As it looks, `path` and `text` are common parameters and maybe we can extract them as properties to make the API easier to use in a declarative context.

```
// FileIO.qml (better)
QtObject {
    property url source
    property string text
    function write() {} // open file and write text
    function read() {} // read file and assign to text
}
```

Yes, this looks more like a QML API. We use properties to allow our environment to bind to our properties and react to changes.

To create this API in C++ we would need to create a Qt C++ interface looking like this.

```
class FileIO : public QObject {
    ...
    Q_PROPERTY(QUrl source READ source WRITE setSource NOTIFY s
    Q_PROPERTY(QString text READ text WRITE setText NOTIFY text
    ...
public:
    Q_INVOKABLE void read();
    Q_INVOKABLE void write();
    ...
}
```

The `FileIO` type need to be registered with the QML engine. We want to use it under the “org.example.io” module

```
import org.example.io 1.0

FileIO {
}
```

A plugin could expose several types with the same module. But it can not expose several modules from one plugin. So there is a one to one relationship between modules and plugins. This relationship is expressed by the module identifier.

Creating the plugin

Qt Creator contains a wizard to create a **QtQuick 2 QML Extension Plugin**, found under **Library** when creating a new project. We use it to create a plugin called `fileio` with a `FileIO` object to start within the module `org.example.io` .

> **TIP**
>
> The current wizard generates a QMake based project. Please use the example from this chapter as a starting point to build a CMake based project.

The project should consist of the `fileio.h` and `fileio.cpp` , that declare and implement the `FileIO` type, and a `fileio_plugin.cpp` that contains the actual plugin class that allows the QML engine to discover out extension.

The plugin class is derived from the `QQmlEngineExtensionPlugin` class, and contains a the `Q_OBJECT` and `Q_PLUGIN_METADATA` macros. The entire file can be seen below.

```
#include <QQmlEngineExtensionPlugin>

class FileioPlugin : public QQmlEngineExtensionPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID QQmlEngineExtensionInterface_iid)
};
```

```
#include "fileio_plugin.moc"
```

The extension will automatically discover and register all types marked with `QML_ELEMENT` and `QML_NAMED_ELEMENT`. We will see how this is done in the FileIO Implementation section below.

For the import of the module to work, the user also needs to specify a URI, e.g. `import org.example.io`. Interestingly we cannot see the module URI anywhere. This is set from the outside using a `qmldir` file, alternatively in the `CMakeLists.txt` file of your project.

The `qmldir` file specifies the content of your QML plugin or better the QML side of your plugin. A hand-written `qmldir` file for our plugin should look something like this:

```
module org.example.io
plugin fileio
```

The module is the URI that the user imports, and after it you name which plugin to load for the said URI. The plugin line must be identical with your plugin file name (under mac this would be *libfileio_debug.dylib* on the file system and *fileio* in the *qmldir*, for a Linux system, the same line would look for *libfileio.so*). These files are created by Qt Creator based on the given information.

The easier way to create a correct `qmldir` file is in the `CMakeLists.txt` for your project, in the `qt_add_qml_module` macro. Here, the `URI` parameter is used to specify the URI of the plugin, e.g. `org.example.io`. This way, the `qmldir` file is generated when the project is built. For the example here, the `qt_add_qml_module` macro looks as follows:

```
qt_add_qml_module(fileio PLUGIN_TARGET
    VERSION 1.0.0
    URI "org.example.io"
    OUTPUT_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}/imports/org/e
    SOURCES
        fileio.cpp
        fileio.h
        fileio_plugin.cpp
)
```

When importing a module called `org.example.io`, the QML engine will look in one of the import paths, e.g. the `QML2_IMPORT_PATH` environment variable, and try to locate the `org/example/io` path with a `qmldir` file. The `qmldir` file will tell the engine which library to load as a QML extension plugin using which module URI. Two modules with the same URI will override each other. For the example above, the module can be imported with the following command:

```
QML2_IMPORT_PATH=/home/.../ch18-extensions/src/fileio/imports \
./.../ch18-extensions/src/CityUI/CityUI
```

Notice that the `QML2_IMPORT_PATH` points to the `imports` directory, and that the `org/example/io` sub-path is found via the `org.example.io` part of the import statement.

FileIO Implementation

Remember the `FileIO` API we want to create should look like this.

```
class FileIO : public QObject {
    ...
    Q_PROPERTY(QUrl source READ source WRITE setSource NOTIFY s
    Q_PROPERTY(QString text READ text WRITE setText NOTIFY text
    ...
public:
    Q_INVOKABLE void read();
    Q_INVOKABLE void write();
    ...
}
```

We will leave out the properties, as they are simple setters and getters.

The read method opens a file in reading mode and reads the data using a text stream.

```
void FileIO::read()
{
    if(m_source.isEmpty()) {
        return;
    }
    QFile file(m_source.toLocalFile());
    if(!file.exists()) {
        qWarning() << "Does not exist: " << m_source.toLocalFil
        return;
    }
    if(file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
```

```
            m_text = stream.readAll();
            emit textChanged(m_text);
        }
    }
```

When the text is changed it is necessary to inform others about the change using `emit textChanged(m_text)`. Otherwise, property binding will not work.

The write method does the same but opens the file in write mode and uses the stream to write the contents of the `text` property.

```
    void FileIO::read()
    {
        if(m_source.isEmpty()) {
            return;
        }
        QFile file(m_source.toLocalFile());
        if(!file.exists()) {
            qWarning() << "Does not exist: " << m_source.toLocalFil
            return;
        }
        if(file.open(QIODevice::ReadOnly)) {
            QTextStream stream(&file);
            m_text = stream.readAll();
            emit textChanged(m_text);
        }
    }
```

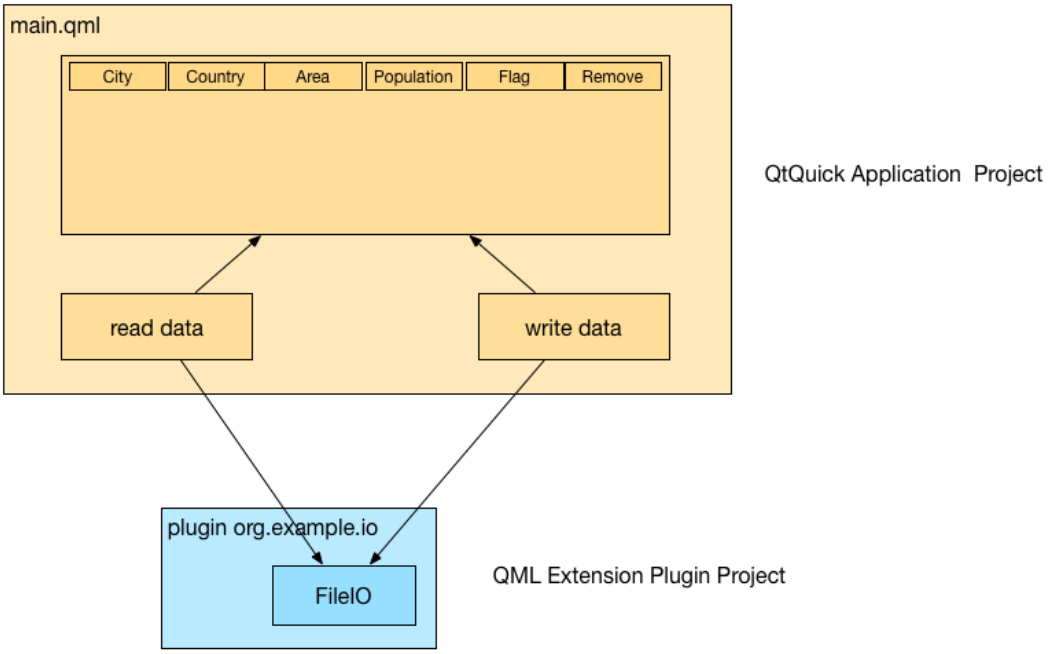
To make the type visible to QML, we add the `QML_ELEMENT` macro just after the `Q_PROPERTY` lines. This tells Qt that the type should be made available to QML. If you want to provide a different name than the C++ class, you can use the `QML_NAMED_ELEMENT` macro.

TIP

As the reading and writing are blocking function calls you should only use this `FileIO` for small texts, otherwise, you will block the UI thread of Qt. Be warned!

Using FileIO

Now we can use our newly created file to access some data. In this example, we will some city data in a JSON format and display it in a table. We build this as two projects: one for the extension plugin (called `fileio`) which provides us a way to read and write text from a file, and the other, which displays the data in a table, (`CityUI`). The `CityUI` uses the `fileio` extension for reading and writing files.



JSON data is just text that is formatted in such a way that it can be converted into a valid JS object/array and back to text. We use our `FileIO` to read the JSON formatted data and convert it into a JS object using the built-in Javascript function `JSON.parse()`. The data is later used as a model for the table view. This is implemented in the read document and write document functions shown below.

```
FileIO {
    id: io
}

function readDocument() {
    io.source = openDialog.fileUrl
    io.read()
    view.model = JSON.parse(io.text)
}

function saveDocument() {
    var data = view.model
    io.text = JSON.stringify(data, null, 4)
    io.write()
}
```

The JSON data used in this example is in the `cities.json` file. It contains a list of city data entries, where each entry contains interesting data about the city such as what is shown below.

```json
[
    {
        "area": "1928",
        "city": "Shanghai",
        "country": "China",
        "flag": "22px-Flag_of_the_People's_Republic_of_China.sv
        "population": "13831900"
    },
    ...
]
```

The Application Window

We use the Qt Creator `QtQuick Application` wizard to create a Qt Quick Controls 2 based application. We will not use the new QML forms as this is difficult to explain in a book, although the new forms approach with a *ui.qml* file is much more usable than previous. So you can remove/delete the forms file for now.

The basic setup is an `ApplicationWindow` which can contain a toolbar, menubar, and status bar. We will only use the menubar to create some standard menu entries for opening and saving the document. The basic setup will just display an empty window.

```
import QtQuick 2.5
import QtQuick.Controls 1.3
import QtQuick.Window 2.2
import QtQuick.Dialogs 1.2

ApplicationWindow {
    id: root
    title: qsTr("City UI")
    width: 640
    height: 480
    visible: true
}
```

Using Actions

To better use/reuse our commands we use the QML `Action` type. This will allow us later to use the same action also for a potential toolbar. The open and save and exit actions are quite standard. The open and save action do not contain any logic yet, this we will come later. The menubar is created with a file menu and these three action entries. Additional we prepare already a file dialog, which will allow us to pick our city document

later. A dialog is not visible when declared, you need to use the `open()` method to show it.

```
Action {
    id: save
    text: qsTr("&Save")
    shortcut: StandardKey.Save
    onTriggered: {
        saveDocument()
    }
}

Action {
    id: open
    text: qsTr("&Open")
    shortcut: StandardKey.Open
    onTriggered: openDialog.open()
}

Action {
    id: exit
    text: qsTr("E&xit")
    onTriggered: Qt.quit();
}

menuBar: MenuBar {
    Menu {
        title: qsTr("&File")
        MenuItem { action: open }
        MenuItem { action: save }
        MenuSeparator {}
        MenuItem { action: exit }
    }
}

FileDialog {
    id: openDialog
    onAccepted: {
        root.readDocument()
```

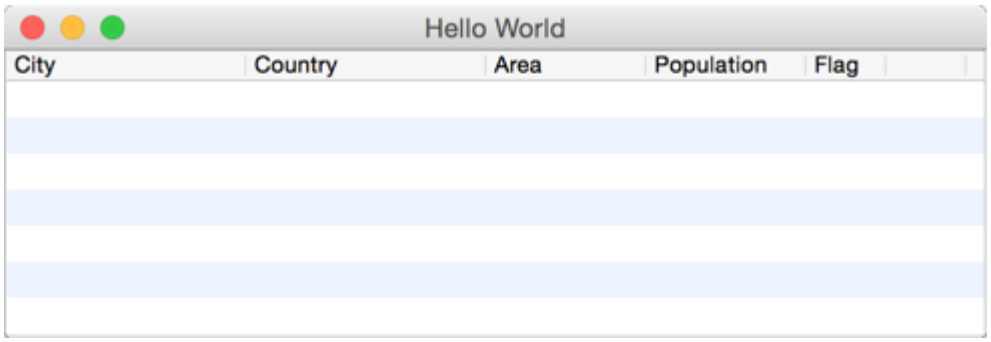
```
    }
}
```

Formatting the Table

The content of the city data shall be displayed in a table. For this, we use the `TableView` control and declare 4 columns: city, country, area, population. Each column is a standard `TableViewColumn`. Later we will add columns for the flag and remove operation which will require a custom column delegate.

```
TableView {
    id: view
    anchors.fill: parent
    TableViewColumn {
        role: 'city'
        title: "City"
        width: 120
    }
    TableViewColumn {
        role: 'country'
        title: "Country"
        width: 120
    }
    TableViewColumn {
        role: 'area'
        title: "Area"
        width: 80
    }
    TableViewColumn {
        role: 'population'
        title: "Population"
        width: 80
    }
}
```

Now the application should show you a menubar with a file menu and an empty table with 4 table headers. The next step will be to populate the table with useful data using our *FileIO* extension.



The `cities.json` document is an array of city entries. Here is an example.

```json
[
    {
        "area": "1928",
        "city": "Shanghai",
        "country": "China",
        "flag": "22px-Flag_of_the_People's_Republic_of_China.sv
        "population": "13831900"
    },
    ...
]
```

Our job is it to allow the user to select the file, read it, convert it and set it onto the table view.

Reading Data

For this we let the open action open the file dialog. When the user has selected a file the `onAccepted` method is called on the file dialog. There we call the `readDocument()` function. The `readDocument()` function sets the URL from the file dialog to our `FileIO` object and calls the

`read()` method. The loaded text from `FileIO` is then parsed using the `JSON.parse()` method and the resulting object is directly set onto the table view as a model. How convenient is that?

```
Action {
    id: open
    ...
    onTriggered: {
        openDialog.open()
    }
}

...

FileDialog {
    id: openDialog
    onAccepted: {
        root.readDocument()
    }
}

function readDocument() {
    io.source = openDialog.fileUrl
    io.read()
    view.model = JSON.parse(io.text)
}


FileIO {
    id: io
}
```

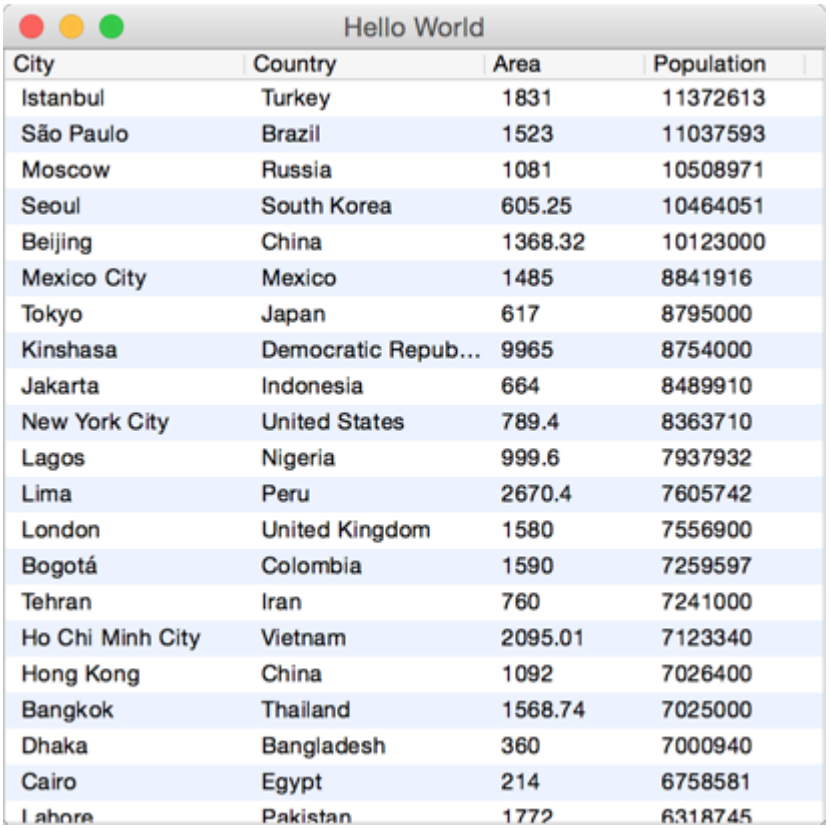
Writing Data

For saving the document, we hook up the “save” action to the `saveDocument()` function. The save document function takes the model

from the view, which is a JS object and converts it into a string using the `JSON.stringify()` function. The resulting string is set to the text property of our `FileIO` object and we call `write()` to save the data to disk. The "null" and "4" parameters on the `stringify` function will format the resulting JSON data using indentation with 4 spaces. This is just for better reading of the saved document.

```
Action {
    id: save
    ...
    onTriggered: {
        saveDocument()
    }
}

function saveDocument() {
    var data = view.model
    io.text = JSON.stringify(data, null, 4)
    io.write()
}

FileIO {
    id: io
}
```

This is basically the application with reading, writing and displaying a JSON document. Think about all the time spend by writing XML readers and writers. With JSON all you need is a way to read and write a text file or send receive a text buffer.



Finishing Touch

The application is not fully ready yet. We still want to show the flags and allow the user to modify the document by removing cities from the model.

In this example, the flag files are stored relative to the `main.qml` document in a *flags* folder. To be able to show them the table column needs to define a custom delegate for rendering the flag image.

```
TableViewColumn {
    delegate: Item {
        Image {
            anchors.centerIn: parent
            source: 'flags/' + styleData.value
        }
    }
```

```
        role: 'flag'
        title: "Flag"
        width: 40
    }
```

That is all that is needed to show the flag. It exposes the flag property from the JS model as `styleData.value` to the delegate. The delegate then adjusts the image path to pre-pend `'flags/'` and displays it as an `Image` element.

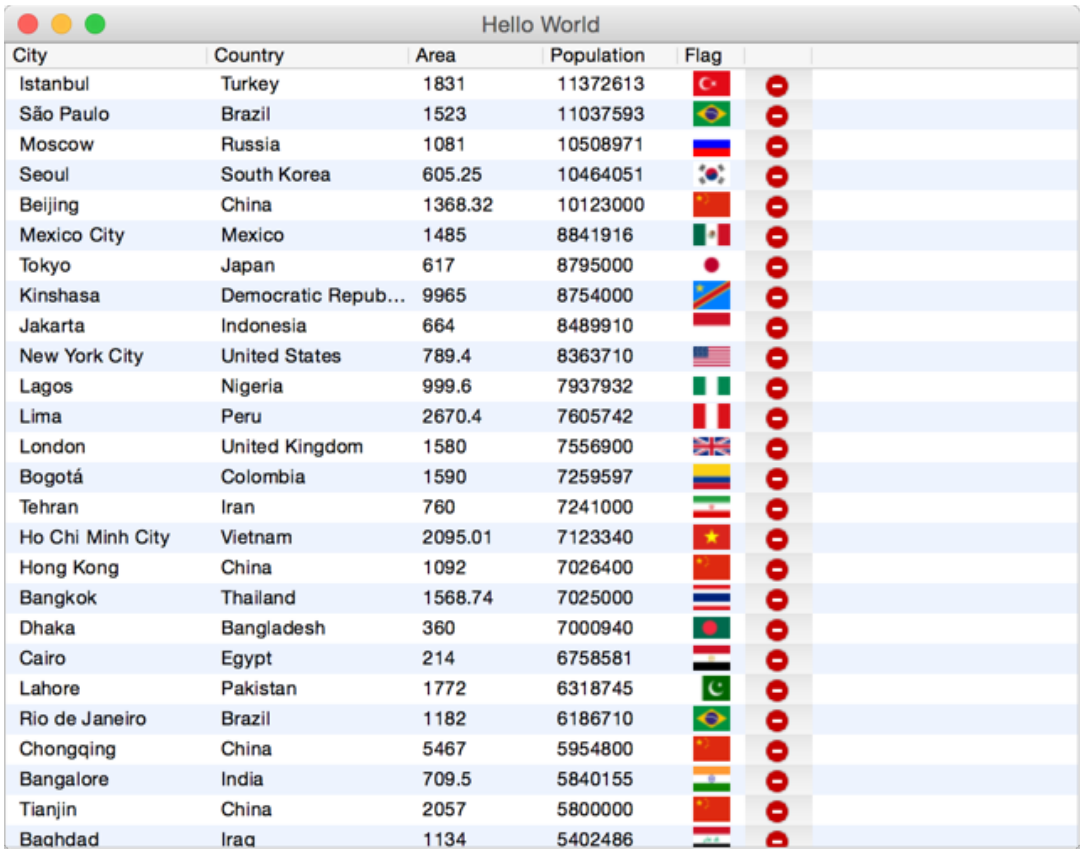
For removing we use a similar technique to display a remove button.

```
    TableViewColumn {
        delegate: Button {
            iconSource: "remove.png"
            onClicked: {
                var data = view.model
                data.splice(styleData.row, 1)
                view.model = data
            }
        }
        width: 40
    }
```

For the data removal operation, we get a hold on the view model and then remove one entry using the JS `splice` function. This method is available to us as the model is from the type JS array. The splice method changes the content of an array by removing existing elements and/or adding new elements.

A JS array is unfortunately not so smart as a Qt model like the `QAbstractItemModel`, which will notify the view about row changes or data changes. The view will not show any updated data by now as it is never notified of any changes. Only when we set the data back to the view, the view recognizes there is new data and refreshes the view

content. Setting the model again using `view.model = data` is a way to let the view know there was a data change.



Summary

The plugin created in this chapter is a very simple plugin. but it can be re-used and extended by other types for different applications. Using plugins creates a very flexible solution. For example, you can now start the UI by just using the `qml`. Open the folder where your `CityUI` project is a start the UI with `qml main.qml`. The extension is readily available to the QML engine from any project and can be imported anywhere.

You are encouraged to write your applications in a way so that they work with a `qml`. This has a tremendous increase in turnaround time for the UI developer and it is also a good habit to keep a clear separation of the logic and the presentation of an application.

The only drawback with using plugins is that the deployment gets more difficult. This becomes more apparent the simpler the application is (as the overhead of creating and deploying the plugin stays the same). You need now to deploy your plugin with your application. If this is a problem for you, you can still use the same `FileIO` class and register it directly in your `main.cpp` using `qmlRegisterType`. The QML code would stay the same.

In larger projects, you do not use an application as such. You have a simple qml runtime similar to the `qml` command provided with Qt, and require all native functionality to come as plugins. And your projects are simple pure qml projects using these qml extension plugins. This provides greater flexibility and removes the compilation step for UI changes. After editing a QML file you just need to run the UI. This allows the user

interface writers to stay flexible and agile to make all these little changes to push pixels.

Plugins provide a nice and clean separation between C++ backend development and QML frontend development. When developing QML plugins always have the QML side in mind and do not hesitate to start with a QML only mockup first to validate your API before you implement it in C++. If an API is written in C++ people often hesitate to change it or not to speak of to rewrite it. Mocking an API in QML provides much more flexibility and less initial investment. When using plugins the switch between a mocked API and the real API is just changing the import path for the qml runtime.

Qt for Python

This chapter describes the PySide6 module from the Qt for Python project. You will learn how to install it and how to leverage QML together with Python.

Introduction

The Qt for Python project provides the tooling to bind C++ and Qt to Python, and a complete Python API to Qt. This means that everything that you can do with Qt and C++, you can also do with Qt and Python. This ranges from headless services to widget based user interfaces. In this chapter, we will focus on how to integrate QML and Python.

Currently, Qt for Python is available for all desktop platforms, but not for mobile. Depending on which platform you use, the setup of Python is slightly different, but as soon as you have a **Python (https://www.python.org/)** and **PyPA (https://www.pypa.io/en/latest/)** environment setup, you can install Qt for Python using `pip`. This is discussed in more detail further down.

As the Qt for Python project provides an entirely new language binding for Qt, it also comes with a new set of documentation. The following resources are good to know about when exploring this module.

- Reference documentation: **https://doc.qt.io/qtforpython/**
- Qt for Python wiki: **https://wiki.qt.io/Qt_for_Python**
- Caveats: **https://wiki.qt.io/Qt_for_Python/Considerations**

The Qt for Python bindings are generated using the Shiboken tool. At times, it might be of interest to read about it as well to understand what is going on. The preferred point for finding information about Shiboken is the **reference documentation (https://doc.qt.io/qtforpython/shiboken6/index.html)**. If you want to mix your own C++ code with Python and QML, Shiboken is the tool that you need.

TIP

Through-out this chapter we will use Python 3.7.

Installing

Qt for Python is available through PyPA using `pip` under the name `pyside6`. In the example below we setup a `venv` environment in which we will install the latest version of Qt for Python:

```sh
mkdir qt-for-python
cd qt-for-python
python3 -m venv .
source bin/activate
(qt-for-python) $ python --version
Python 3.9.6
```

When the environment is setup, we continue to install `pyside6` using `pip`:

```sh
(qt-for-python) $ pip install pyside6
Collecting pyside6
Downloading [ ... ] (60.7 MB)
Collecting shiboken6==6.1.2
Downloading [ ... ] (1.0 MB)
Installing collected packages: shiboken6, pyside6
Successfully installed pyside6-6.1.2 shiboken6-6.1.2
```

After the installation, we can test it by running a *Hello World* example from the interactive Python prompt:

```sh
(qt-for-python) $ python
Python 3.9.6 (default, Jun 28 2021, 06:20:32)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more infor
>>> from PySide6 import QtWidgets
>>> import sys
>>> app = QtWidgets.QApplication(sys.argv)
>>> widget = QtWidgets.QLabel("Hello World!")
>>> widget.show()
>>> app.exec()
0
>>>
```

The example results in a window such as the one shown below. To end the program, close the window.

Building an Application

In this chapter we will look at how you can combine Python and QML. The most natural way to combine the two worlds is to do as with C++ and QML, i.e. implement the logic in Python and the presentation in QML.

To do this, we need to understand how to combine QML and Python into a single program, and then how to implement interfaces between the two worlds. In the sub-sections below, we will look at how this is done. We will start simple and progress to an example exposing the capabilities of a Python module to QML through a Qt item model.

Running QML from Python

The very first step is to create a Python program that can host the *Hello World* QML program shown below.

```
import QtQuick
import QtQuick.Window

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello Python World!")
}
```

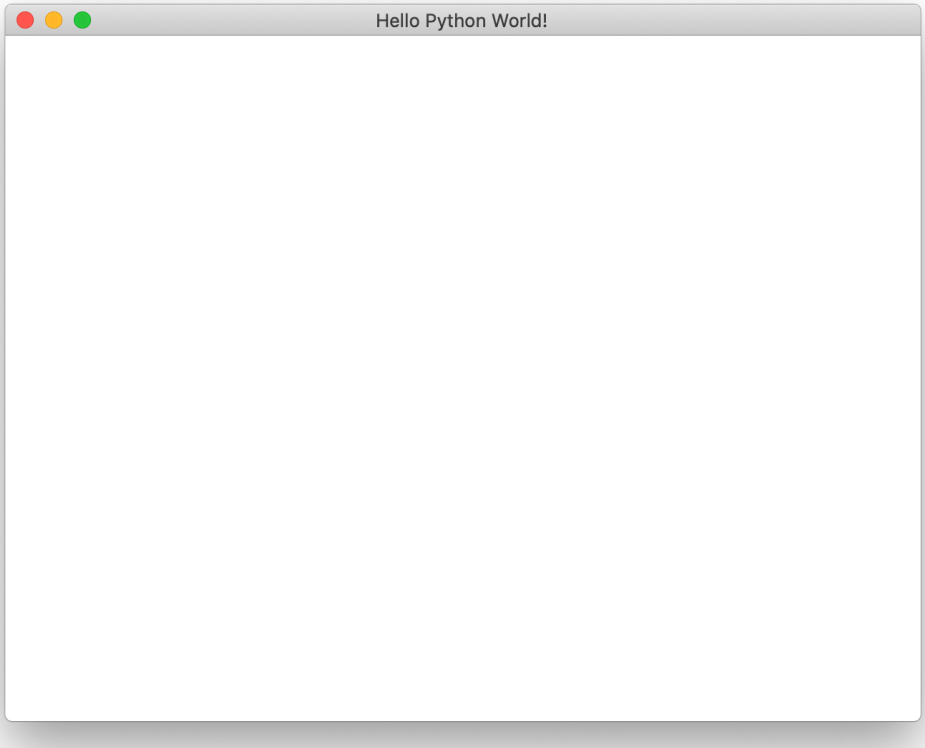
To do this, we need a Qt mainloop provided by `QGuiApplication` from the `QtGui` module. We also need a `QQmlApplicationEngine` from the

`QtQml` module. In order to pass the reference to the source file to the QML application engine, we also need the `QUrl` class from the `QtCore` module.

In the code below we emulate the functionality of the boilerplate C++ code generated by Qt Creator for QML projects. It instantiates the application object, and creates a QML application engine. It then loads the QML and then ensures that the QML was loaded by checking if a root object was created. Finally, it exits and returns the value returned by the `exec` method of the application object.

```py
import sys
from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine
from PySide6.QtCore import QUrl

if __name__ == '__main__':
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()
    engine.load(QUrl("main.qml"))

    if not engine.rootObjects():
        sys.exit(-1)

    sys.exit(app.exec())
```

Executing the example results in a window with the title *Hello Python World*.



> **TIP**
>
> The example assumes that it is executed from the directory containing the `main.qml` source file. You can determine the location of the Python file being executed using the `__file__` variable. This can be used to locate the QML files relative to the Python file as shown in this **blog post (http://blog.qt.io/blog/2018/05/14/qml-qt-python/)** .

Exposing Python Objects to QML

The easiest way to share information between Python and QML is to expose a Python object to QML. This is done by registering a *context*

property through the `QQmlApplicationEngine`. Before we can do that, we need to define a class so that we have an object to expose.

Qt classes come with a number of features that we want to be able to use. These are: signals, slots and properties. In this first example, we will restrict ourselves to a basic pair of signal and slot. The rest will be covered in the examples further on.

Signals and Slots

We start with the class `NumberGenerator`. It has a constructor, a method called `updateNumber` and a signal called `nextNumber`. The idea is that when you call `updateNumber`, the signal `nextNumber` is emitted with a new random number. You can see the code for the class below, but first we will look at the details.

First of all we make sure to call `QObject.__init__` from our constructor. This is very important, as the example will not work without it.

Then we declare a signal by creating an instance of the `Signal` class from the `PySide6.QtCore` module. In this case, the signal carries an integer value, hence the `int`. The signal parameter name, `number`, is defined in the `arguments` parameter.

Finally, we *decorate* the `updateNumber` method with the `@Slot()` decorator, thus turning it into a slot. There is no concept of *invokables* in Qt for Python, so all callable methods must be slots.

In the `updateNumber` method we emit the `nextNumber` signal using the `emit` method. This is a bit different than the syntax for doing so from QML or C++ as the signal is represented by an object instead of being a callable function.

```py
import random

from PySide6.QtCore import QObject, Signal, Slot

class NumberGenerator(QObject):
    def __init__(self):
        QObject.__init__(self)

    nextNumber = Signal(int, arguments=['number'])

    @Slot()
    def giveNumber(self):
        self.nextNumber.emit(random.randint(0, 99))
```

Next up is to combine the class we just created with the boilerplate code for combining QML and Python from earlier. This gives us the following entry-point code.

The interesting lines are the one where we first instantiate a `NumberGenerator`. This object is then exposed to QML using the `setContextProperty` method of the `rootContext` of the QML engine. This exposes the object to QML as a global variable under the name `numberGenerator`.

```py
import sys

from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine
from PySide6.QtCore import QUrl

if __name__ == '__main__':
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()

    number_generator = NumberGenerator()
    engine.rootContext().setContextProperty("numberGenerator",
```

```
    engine.load(QUrl("main.qml"))

    if not engine.rootObjects():
        sys.exit(-1)

    sys.exit(app.exec())
```

Continuing to the QML code, we can see that we've created a Qt Quick Controls 2 user interface consisting of a `Button` and a `Label`. In the button's `onClicked` handler, the `numberGenerator.updateNumber()` function is called. This is the slot of the object instantiated on the Python side.

To receive a signal from an object that has been instantiated outside of QML we need to use a `Connections` element. This allows us to attach a signal handler to an existing target.

```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

Window {
    id: root

    width: 640
    height: 480
    visible: true
    title: qsTr("Hello Python World!")

    Flow {
        Button {
            text: qsTr("Give me a number!")
            onClicked: numberGenerator.giveNumber()
        }
        Label {
            id: numberLabel
```

```
            text: qsTr("no number")
        }
    }

    Connections {
        target: numberGenerator
        function onNextNumber(number) {
            numberLabel.text = number
        }
    }
}
```

Properties

Instead of relying solely on signals and slots, the common way to expose state to QML is through properties. A property is a combination of a setter, getter and notification signal. The setter is optional, as we can also have read-only properties.

To try this out we will update the `NumberGenerator` from the last example to a property based version. It will have two properties: `number`, a read-only property holding the last random number, and `maxNumber`, a read-write property holding the maximum value that can be returned. It will also have a slot, `updateNumber` that updates the random number.

Before we dive into the details of properties, we create a basic Python class for this. It consists of the relevant getters and setters, but not Qt signalling. As a matter of fact, the only Qt part here is the inheritance from `QObject`. Even the names of the methods are Python style, i.e. using underscores instead of camelCase.

Take notice of the underscores (“ `__` ”) at the beginning of the `__set_number` method. This implies that it is a private method. So even when the `number` property is read-only, we provide a setter. We just

don’t make it public. This allows us to take actions when changing the value (e.g. emitting the notification signal).

```py
class NumberGenerator(QObject):
    def __init__(self):
        QObject.__init__(self)
        self.__number = 42
        self.__max_number = 99

    def set_max_number(self, val):
        if val < 0:
            val = 0

        if self.__max_number != val:
            self.__max_number = val

        if self.__number > self.__max_number:
            self.__set_number(self.__max_number)

    def get_max_number(self):
        return self.__max_number

    def __set_number(self, val):
        if self.__number != val:
            self.__number = val

    def get_number(self):
        return self.__number
```

In order to define properties, we need to import the concepts of `Signal`, `Slot`, and `Property` from `PySide2.QtCore`. In the full example, there are more imports, but these are the ones relevant to the properties.

```py
from PySide6.QtCore import QObject, Signal, Slot, Property
```

Now we are ready to define the first property, `number`. We start off by declaring the signal `numberChanged`, which we then invoke in the `__set_number` method so that the signal is emitted when the value is changed.

After that, all that is left is to instantiate the `Property` object. The `Property` constructor takes three arguments in this case: the type (`int`), the getter (`get_number`) and the notification signal which is passed as a named argument (`notify=numberChanged`). Notice that the getter has a Python name, i.e. using underscore rather than camelCase, as it is used to read the value from Python. For QML, the property name, `number`, is used.

```py
class NumberGenerator(QObject):

    # ...

    # number

    numberChanged = Signal(int)

    def __set_number(self, val):
        if self.__number != val:
            self.__number = val
            self.numberChanged.emit(self.__number)

    def get_number(self):
        return self.__number

    number = Property(int, get_number, notify=numberChanged)
```

This leads us to the next property, `maxNumber`. This is a read-write property, so we need to provide a setter, as well as everything that we did for the `number` property.

First up we declare the `maxNumberChanged` signal. This time, using the `@Signal` decorator instead of instantiating a `Signal` object. We also provide a setter slot, `setMaxNumber` with a Qt name (camelCase) that simply calls the Python method `set_max_number` alongside a getter with a Python name. Again, the setter emits the change signal when the value is updated.

Finally we put the pieces together into a read-write property by instantiating a `Property` object taking the type, getter, setter and notification signal as arguments.

```py
class NumberGenerator(QObject):

    # ...

    # maxNumber

    @Signal
    def maxNumberChanged(self):
        pass

    @Slot(int)
    def setMaxNumber(self, val):
        self.set_max_number(val)

    def set_max_number(self, val):
        if val < 0:
            val = 0

        if self.__max_number != val:
            self.__max_number = val
            self.maxNumberChanged.emit()

        if self.__number > self.__max_number:
            self.__set_number(self.__max_number)

    def get_max_number(self):
```

```
        return self.__max_number

    maxNumber = Property(int, get_max_number, set_max_number, n
```

Now we have properties for the current random number, `number`, and the maximum random number, `maxNumber`. All that is left is a slot to produce a new random number. It is called `updateNumber` and simply sets a new random number.

```py
class NumberGenerator(QObject):

    # ...

    @Slot()
    def updateNumber(self):
        self.__set_number(random.randint(0, self.__max_number))
```

Finally, the number generator is exposed to QML through a root context property.

```py
if __name__ == '__main__':
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()

    number_generator = NumberGenerator()
    engine.rootContext().setContextProperty("numberGenerator",

    engine.load(QUrl("main.qml"))

    if not engine.rootObjects():
        sys.exit(-1)

    sys.exit(app.exec())
```

In QML, we can bind to the `number` as well as the `maxNumber` properties of the `numberGenerator` object. In the `onClicked` handler of the `Button` we call the `updateNumber` method to generate a new random number and in the `onValueChanged` handler of the `Slider` we set the `maxNumber` property using the `setMaxNumber` method. This is because altering the property directly through Javascript would destroy the bindings to the property. By using the setter method explicitly, this is avoided.

```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

Window {
    id: root

    width: 640
    height: 480
    visible: true
    title: qsTr("Hello Python World!")

    Column {
        Flow {
            Button {
                text: qsTr("Give me a number!")
                onClicked: numberGenerator.updateNumber()
            }
            Label {
                id: numberLabel
                text: numberGenerator.number
            }
        }
        Flow {
            Slider {
                from: 0
                to: 99
                value: numberGenerator.maxNumber
```

```
                onValueChanged: numberGenerator.setMaxNumber(va
            }
        }
    }
}
```

Exposing a Python class to QML

Up until now, we've instantiated an object Python and used the `setContextProperty` method of the `rootContext` to make it available to QML. Being able to instantiate the object from QML allows better control over object life-cycles from QML. To enable this, we need to expose the *class*, instead of the *object*, to QML.

The class that is being exposed to QML is not affected by where it is instantiated. No change is needed to the class definition. However, instead of calling `setContextProperty`, the `qmlRegisterType` function is used. This function comes from the `PySide2.QtQml` module and takes five arguments:

- A reference to the class, `NumberGenerator` in the example below.
- A module name, `'Generators'`.
- A module version consisting of a major and minor number, `1` and `0` meaning `1.0`.
- The QML name of the class, `'NumberGenerator'`

```py
import random
import sys

from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine, qmlRegisterTyp
from PySide6.QtCore import QUrl, QObject, Signal, Slot

class NumberGenerator(QObject):
```

```
    def __init__(self):
        QObject.__init__(self)

    nextNumber = Signal(int, arguments=['number'])

    @Slot()
    def giveNumber(self):
        self.nextNumber.emit(random.randint(0, 99))


if __name__ == '__main__':
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()

    qmlRegisterType(NumberGenerator, 'Generators', 1, 0, 'Numbe

    engine.load(QUrl("main.qml"))

    if not engine.rootObjects():
        sys.exit(-1)

    sys.exit(app.exec())
```

In QML, we need to import the module, e.g. `Generators 1.0` and then instantiate the class as `NumberGenerator { ... }`. The instance now works like any other QML element.

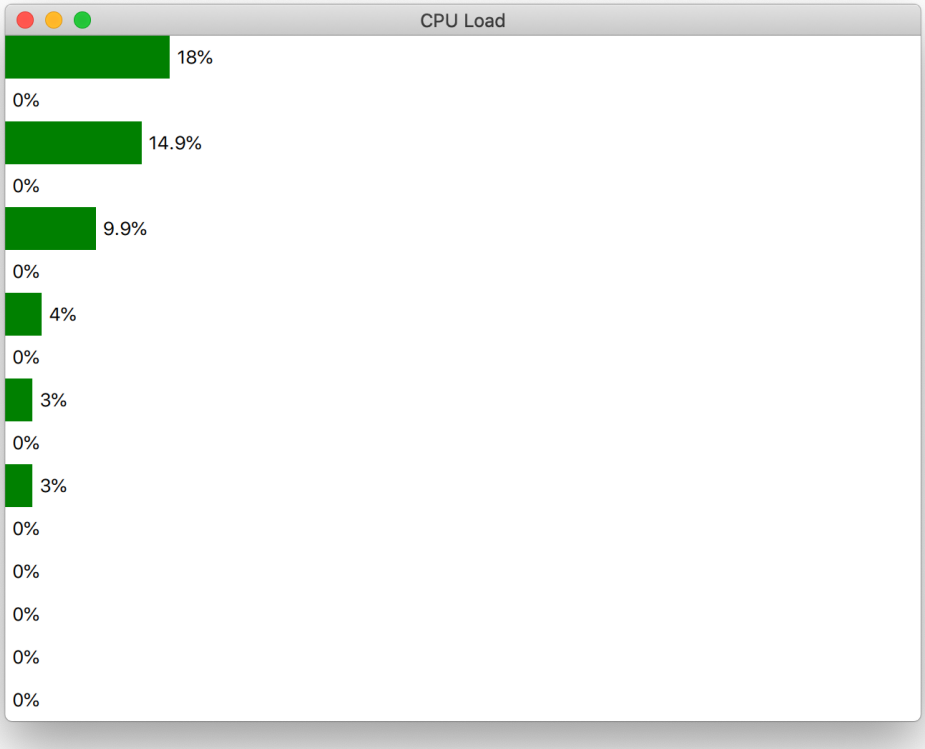
```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

import Generators

Window {
    id: root

    width: 640
    height: 480
```

```
        visible: true
        title: qsTr("Hello Python World!")

        Flow {
            Button {
                text: qsTr("Give me a number!")
                onClicked: numberGenerator.giveNumber()
            }
            Label {
                id: numberLabel
                text: qsTr("no number")
            }
        }

        NumberGenerator {
            id: numberGenerator
        }

        Connections {
            target: numberGenerator
            function onNextNumber(number) {
                numberLabel.text = number
            }
        }
    }
```

A Model from Python

One of the more interesting types of objects or classes to expose from Python to QML are item models. These are used with various views or the `Repeater` element to dynamically build a user interface from the model contents.

In this section we will take an existing python utility for monitoring CPU load (and more), `psutil`, and expose it to QML via a custom made item model called `CpuLoadModel`. You can see the program in action below:



> **TIP**
>
> The psutil library can be found at **https://pypi.org/project/psutil/ (https://pypi.org/project/psutil/)** .

"psutil (process and system utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python."

You can install psutil using `pip install psutil` .

We will use the `psutil.cpu_percent` function (**documentation (https://psutil.readthedocs.io/en/latest/#psutil.cpu_percent)**) to sample the CPU load per core every second. To drive the sampling we use a `QTimer` . All of this is exposed through the `CpuLoadModel` which is a `QAbstractListModel` .

Item models are interesting. They allow you to represent a two dimensional data set, or even nested data sets, if using the `QAbstractItemModel`. The `QAbstractListModel` that we use allow us to represent a list of items, so a one dimensional set of data. It is possible to implement a nested set of lists, creating a tree, but we will only create one level.

To implement a `QAbstractListModel`, it is necessary to implement the methods `rowCount` and `data`. The `rowCount` returns the number of CPU cores which we get using the `psutil.cpu_count` method. The `data` method returns data for different *roles*. We only support the `Qt.DisplayRole`, which corresponds to what you get when you refer to `display` inside the delegate item from QML.

Looking at the code for the model, you can see that the actual data is stored in the `__cpu_load` list. If a valid request is made to `data`, i.e. the row, column and role is correct, we return the right element from the `__cpu_load` list. Otherwise we return `None` which corresponds to an uninitialized `QVariant` on the Qt side.

Every time the update timer (`__update_timer`) times out, the `__update` method is triggered. Here, the `__cpu_load` list is updated, but we also emit the `dataChanged` signal, indicating that all data was changed. We do not do a `modelReset` as that also implies that the number of items might have changed.

Finally, the `CpuLoadModel` is exposed to QML are a registered type in the `PsUtils` module.

```py
import psutil
import sys

from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine, qmlRegisterTyp
from PySide6.QtCore import Qt, QUrl, QTimer, QAbstractListModel
```

```
class CpuLoadModel(QAbstractListModel):
    def __init__(self):
        QAbstractListModel.__init__(self)

        self.__cpu_count = psutil.cpu_count()
        self.__cpu_load = [0] * self.__cpu_count

        self.__update_timer = QTimer(self)
        self.__update_timer.setInterval(1000)
        self.__update_timer.timeout.connect(self.__update)
        self.__update_timer.start()

        # The first call returns invalid data
        psutil.cpu_percent(percpu=True)

    def __update(self):
        self.__cpu_load = psutil.cpu_percent(percpu=True)
        self.dataChanged.emit(self.index(0,0), self.index(self.

    def rowCount(self, parent):
        return self.__cpu_count

    def data(self, index, role):
        if (role == Qt.DisplayRole and
            index.row() >= 0 and
            index.row() < len(self.__cpu_load) and
            index.column() == 0):
            return self.__cpu_load[index.row()]
        else:
            return None

if __name__ == '__main__':
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()

    qmlRegisterType(CpuLoadModel, 'PsUtils', 1, 0, 'CpuLoadMode

    engine.load(QUrl("main.qml"))
```

```
    if not engine.rootObjects():
        sys.exit(-1)

    sys.exit(app.exec())
```

On the QML side we use a `ListView` to show the CPU load. The model is bound to the `model` property. For each item in the model a `delegate` item will be instantiated. In this case that means a `Rectangle` with a green bar (another `Rectangle`) and a `Text` element displaying the current load.

```
import QtQuick
import QtQuick.Window

import PsUtils

Window {
    id: root

    width: 640
    height: 480
    visible: true
    title: qsTr("CPU Load")

    ListView {
        anchors.fill: parent
        model: CpuLoadModel { }
        delegate: Rectangle {
            id: delegate

            required property int display

            width: parent.width
            height: 30
            color: "white"

            Rectangle {
```

```
                id: bar
                width: parent.width * delegate.display / 100.0
                height: 30
                color: "green"
            }

            Text {
                anchors.verticalCenter: parent.verticalCenter
                x: Math.min(bar.x + bar.width + 5, parent.width
                text: delegate.display + "%"
            }
        }
    }
}
```

Limitations

At the moment, there are some things that are not easily available. One of them is that you cannot easily create QML plugins using Python. Instead you need to import the Python QML “modules” into your Python program and then use qmlRegisterType to make it possible to import them from QML.

Summary

In this chapter we have looked at the PySide6 module from the Qt for Python project. After a brief look at installation, we focused on how Qt concepts are used from Python. This included slots, signals and properties. We also looked at a basic list model and how to expose both Python objects and classes from Python to QML.

Qt for MCUs

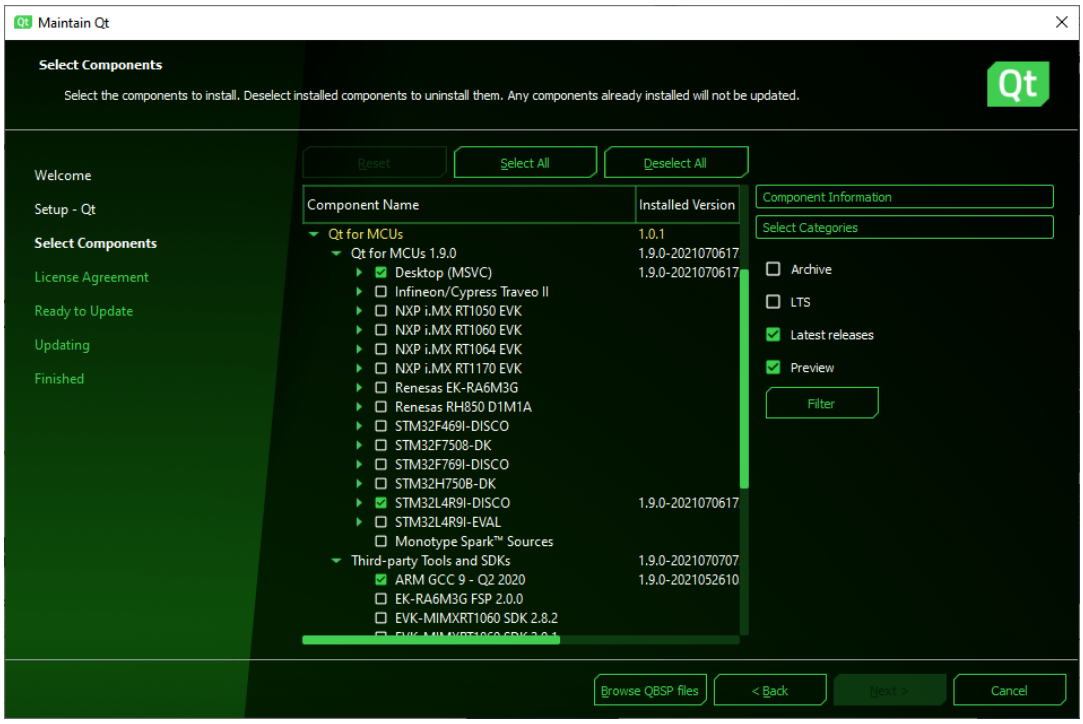
> **Notice**
>
> Qt for MCUs is not a part of the open source Qt distribution, but as a commercial add-on.

Qt for MCUs is a Qt version takes Qt for platforms that are too small to run Linux. Instead, Qt for MCUs can run on top of FreeRTOS, or even on the bare metal, i.e. without any operating system involved. As this book focuses on QML, we will have a deeper look at Qt Quick Ultralite and compare it to the full-size Qt offering.

Using Qt for MCUs you can build beautiful, fluid graphical user interfaces for your micro controller-based systems. Qt for MCUs is focused on the graphical front-end, so instead of the traditional Qt modules, common C++ types are used. This means that some interfaces change. Most notably how models are exposed to QML. In this chapter we will dive into this, and more.

Setup

Qt for MCUs (https://doc.qt.io/QtForMCUs/index.html) comes with support for a range of evaluation boards from companies such as NCP, Renesas, ST, and Infinion/Cypress. These are good for getting started and helps you try out the integration to the specific MCU. In the end, you will most likely end up tuning a specific platform definition to your specific hardware, e.g. to configure the amount of RAM, FLASH and screen configuration.

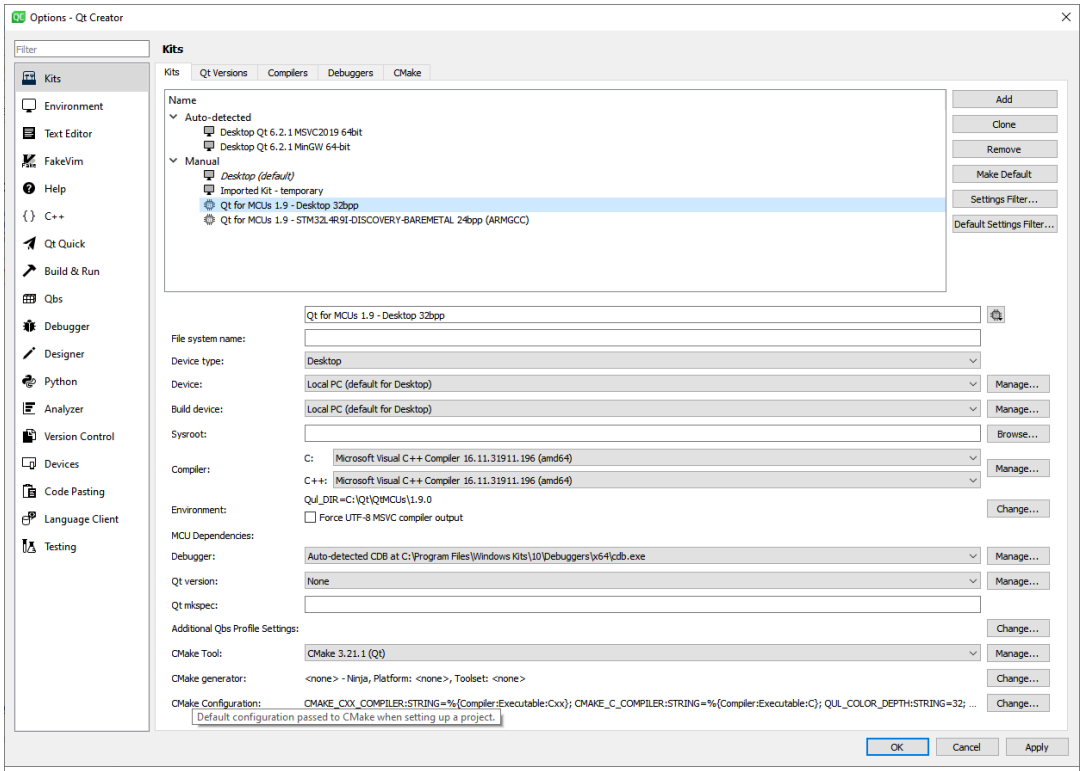


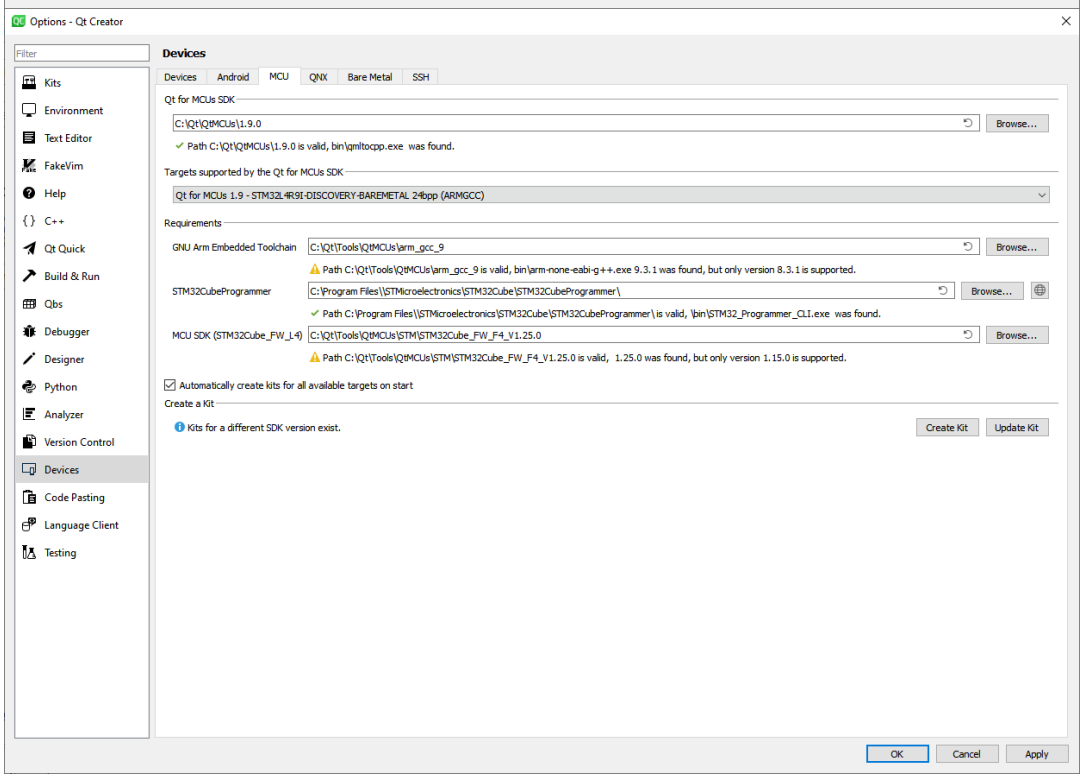
In addition to supporting multiple MCUs out of the box, Qt for MCUs also support running either on FreeRTOS or directly on the bare metal, i.e. without an operating system. As Qt for MCUs focuses on the graphical front-end part of things, there are no classes for filesystems and such. All

this has to come from the underlying system. Hence, if you need support for more complex feature, FreeRTOS is one option.

When it comes to the development environment, various boards come with various compilers, so the Qt for MCUs setup will look a bit different depending on which MCU you target, as well as which compiler you choose. For instance, for the boards from ST, both GCC and IAR are supported, while for some other boards Green Hills MULTI Compiler is used. The officially supported development hosts from Qt's point of view are Linux (Ubuntu 20.04 LTS on x86_64) or Windows (Windows 10 on x86_64). For Windows, please notice that the MSVC compilers supported are the 2017 and 2019 editions - not the very latest. Make sure to follow the latest **setup instructions on qt.io (https://doc.qt.io/QtForMCUs/qtul-setup-development-host.html)** to get a working environment.

Once you have setup your environment, you can find the supported boards as *Kits* as well as under *Devices - MCU* under the *Tools - Options...* menu item in Qt Creator.


Options - Qt Creator
Filter
Kits
Environment
Text Editor
FakeVim
Help
C++
Qt Quick
Build & Run
Qbs
Debugger
Designer
Python
Analyzer
Version Control
Devices
Code Pasting
Language Client
Testing
Kits
Kits
Qt Versions
Compilers
Debuggers
CMake
Name
Auto-detected
Desktop Qt 6.2.1 MSVC2019 64bit
Desktop Qt 6.2.1 MinGW 64-bit
Manual
Desktop (default)
Imported Kit - temporary
Qt for MCUs 1.9 - Desktop 32bpp
Qt for MCUs 1.9 - STM32L4R9I-DISCOVERY-BAREMETAL 24bpp (ARMGCC)
Add
Clone
Remove
Make Default
Settings Filter...
Default Settings Filter...
Qt for MCUs 1.9 - Desktop 32bpp
File system name:
Device type:
Desktop
Device:
Local PC (default for Desktop)
Build device:
Local PC (default for Desktop)
Sysroot:
Compiler:
C:
Microsoft Visual C++ Compiler 16.11.31911.196 (amd64)
C++:
Microsoft Visual C++ Compiler 16.11.31911.196 (amd64)
Environment:
QUL_DIR=C:\Qt\QtMCUs\1.9.0
Force UTF-8 MSVC compiler output
MCU Dependencies:
Debugger:
Auto-detected CDB at C:\Program Files\Windows Kits\10\Debuggers\x64\cdb.exe
Qt version:
None
Qt mkspec:
Additional Qbs Profile Settings:
CMake Tool:
CMake 3.21.1 (Qt)
CMake generator:
<none> - Ninja, Platform: <none>, Toolset: <none>
CMake Configuration:
CMAKE_CXX_COMPILER:STRING=%{Compiler:Executable:Cxx}; CMAKE_C_COMPILER:STRING=%{Compiler:Executable:C}; QUL_COLOR_DEPTH:STRING=32; ...
Default configuration passed to CMake when setting up a project.
Manage...
Browse...
Change...
OK
Cancel
Apply


Options - Qt Creator
Filter
Kits
Environment
Text Editor
FakeVim
Help
C++
Qt Quick
Build & Run
Qbs
Debugger
Designer
Python
Analyzer
Version Control
Devices
Code Pasting
Language Client
Testing
Devices
Devices
Android
MCU
QNX
Bare Metal
SSH
Qt for MCUs SDK
C:\Qt\QtMCUs\1.9.0
Path C:\Qt\QtMCUs\1.9.0 is valid, bin\qmltocpp.exe was found.
Targets supported by the Qt for MCUs SDK
Qt for MCUs 1.9 - STM32L4R9I-DISCOVERY-BAREMETAL 24bpp (ARMGCC)
Requirements
GNU Arm Embedded Toolchain
C:\Qt\Tools\QtMCUs\arm_gcc_9
Path C:\Qt\Tools\QtMCUs\arm_gcc_9 is valid, bin\arm-none-eabi-g++.exe 9.3.1 was found, but only version 8.3.1 is supported.
STM32CubeProgrammer
C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\
Path C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\ is valid, \bin\STM32_Programmer_CLI.exe was found.
MCU SDK (STM32Cube_FW_L4)
C:\Qt\Tools\QtMCUs\STM\STM32Cube_FW_F4_V1.25.0
Path C:\Qt\Tools\QtMCUs\STM\STM32Cube_FW_F4_V1.25.0 is valid, 1.25.0 was found, but only version 1.15.0 is supported.
Automatically create kits for all available targets on start
Create a Kit
Kits for a different SDK version exist.
Create Kit
Update Kit
Browse...
OK
Cancel
Apply

TIP

If you do not find the MCUs tab under Tools, make sure that the Qt for MCUs plugins (McuSupport and BaremetalSupport) are available and activated under *Help - About Plugins...*.

Links

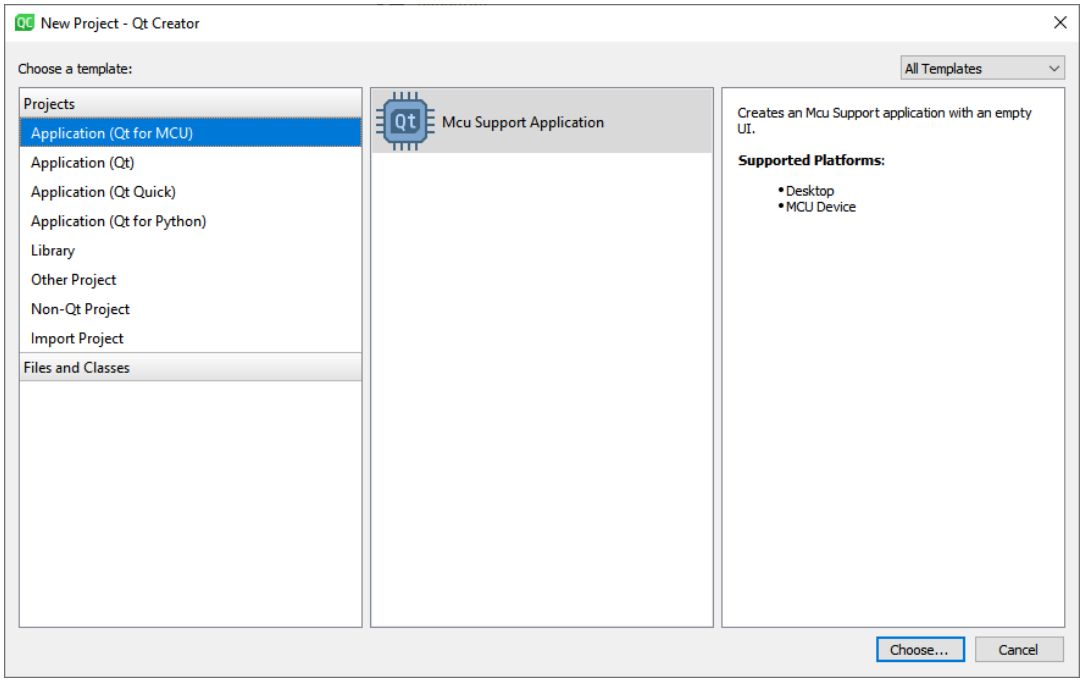
Further reading at qt.io:

- **Supported boards**
- **Platform porting guide**

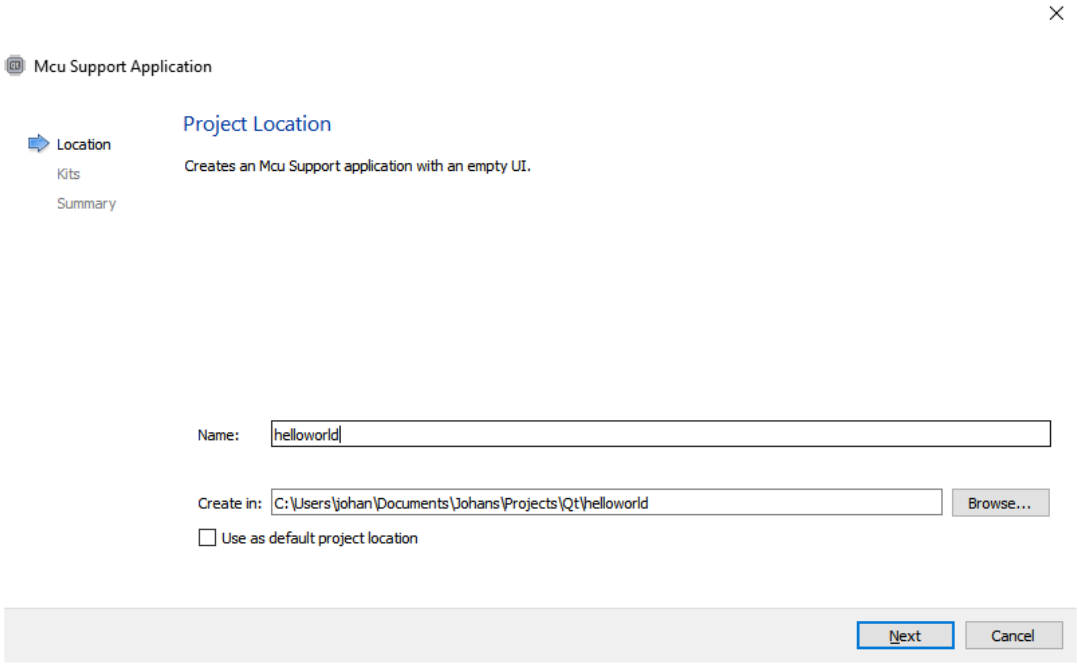
Hello World - for MCUs

As the setup of Qt for MCU can be a bit tricky, we will start with a *Hello World* like example to ensure that the toolchain works, and so that we can discuss the basic differences between Qt Quick Ultralite and standard Qt Quick.

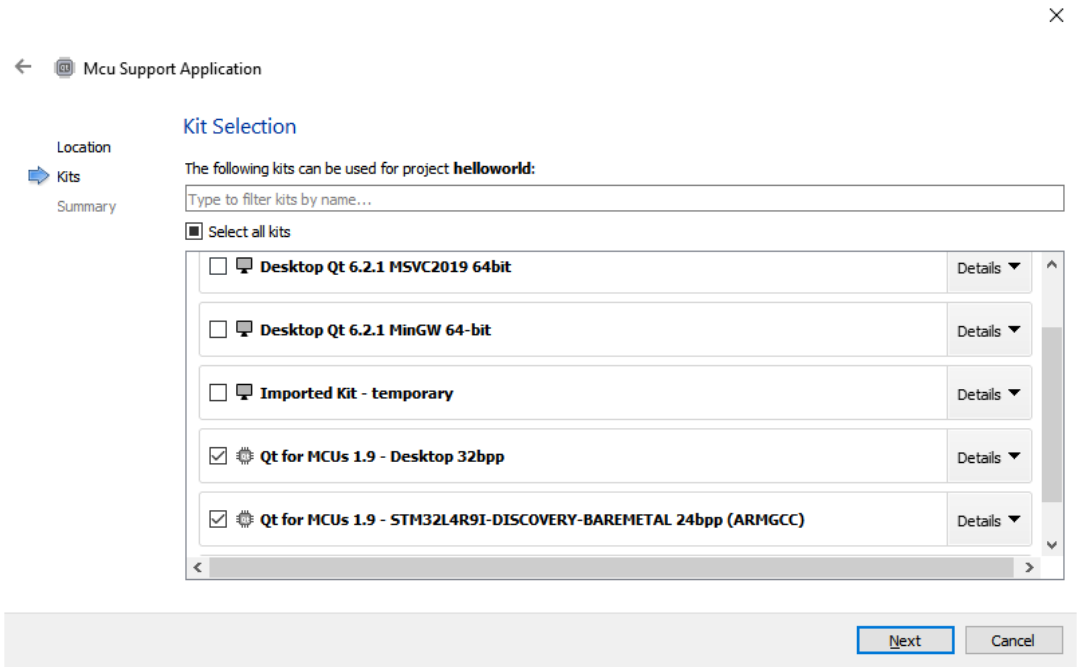
First up, we need to start by creating a Qt for MCUs project in Qt Creator to get a C++ entry point into the system. When working with Qt Quick Ultralite, we cannot use a common runtime such as `qml`. This is because Qt Quick Ultralite is translated into C++ together with optimized versions of all the assets. These are then built into a target executable. This means that there is no support for dynamic loading of QML and such - as there is no interpreter running on the target.



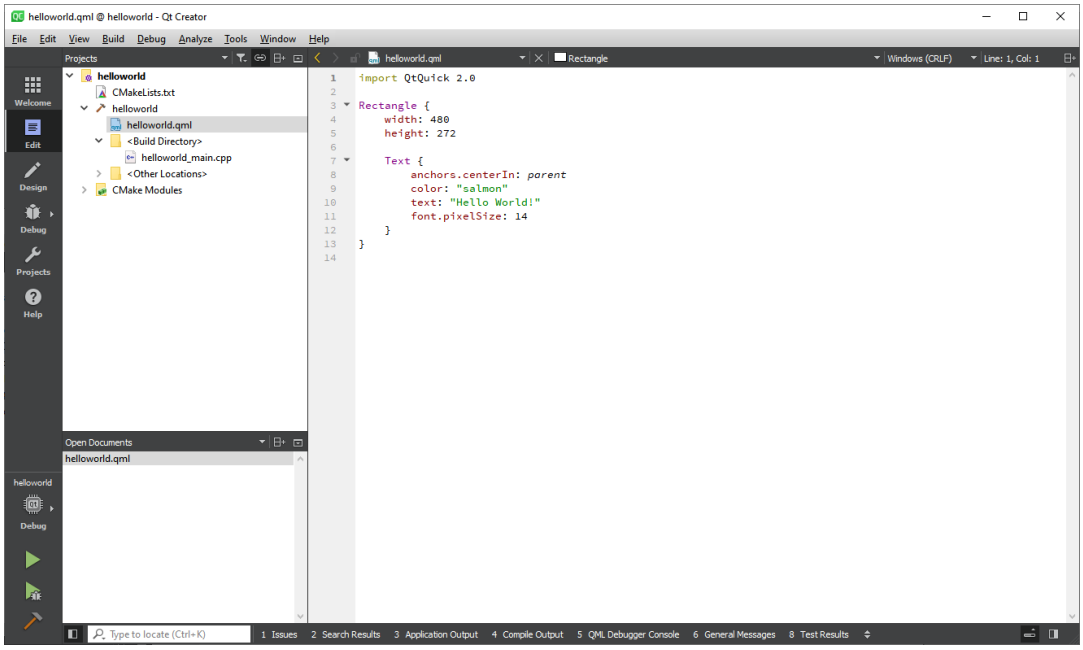
I call the project `helloworld`. Feel free to pick a name of your own. The only thing changing is the name of the entry-point QML-file of the project.



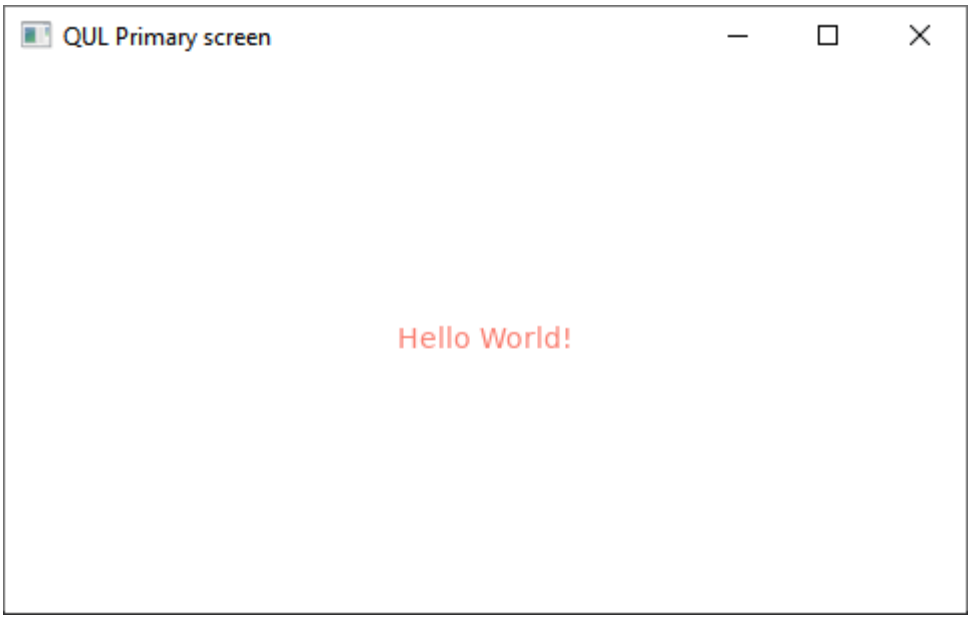
Also, make sure to pick the Qt for MCUs kits when creating your project.



After a few more configuration pages, you will end up with a project as shown below.

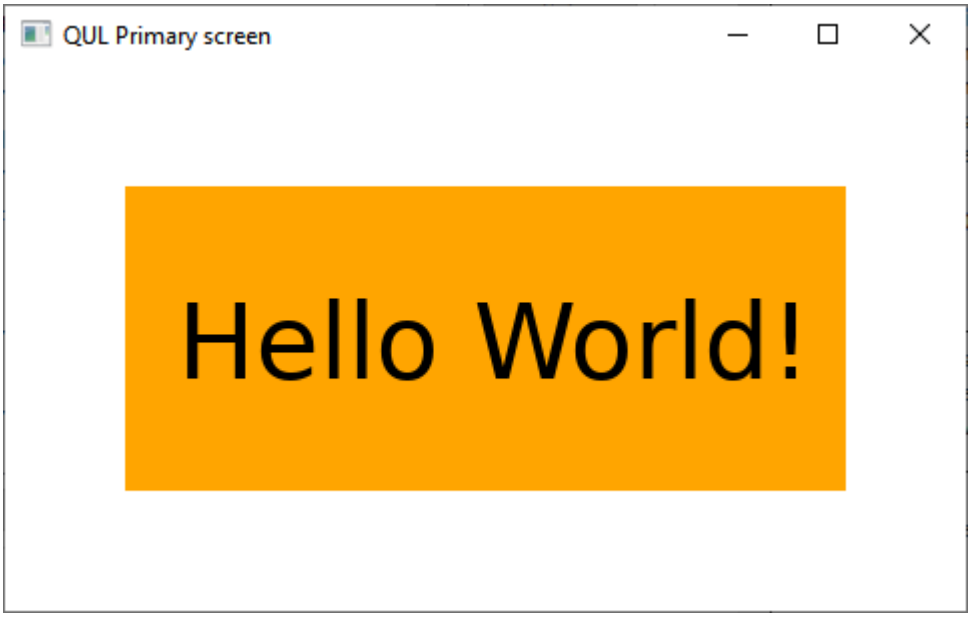


Once the basic project is setup, run the project on your desktop and ensure that you get a window like the one shown below.



Now that we know that the installation works, replace the QML in `helloworld.qml` with the code shown below. We will walk through this example line by line below, but first, build and run it for your *Qt for MCU Desktop* target. This should result in a window looking like the screenshot below the code.

```
import QtQuick
import QtQuickUltralite.Extras

Rectangle {
    width: 480
    height: 272

    Rectangle {
        id: rect
        anchors.fill: parent
        anchors.margins: 60

        color: "orange"
        Behavior on color {
            ColorAnimation { duration: 400 }
        }

        MouseArea {
            anchors.fill: parent

            onClicked: {
                if (rect.color == "red")
                    rect.color = "orange";
                else
                    rect.color = "red";
            }
        }
    }

    StaticText {
        anchors.centerIn: parent
        color: "black"
        text: "Hello World!"
        font.pixelSize: 52
    }
}
```



Click the orange rectangle, and it fades to red. Click it again and it fades back to orange.

Now, let's have a look at the source code from a Qt Quick perspective and compare.

First up, Qt Quick Ultralight ignores the version numbers after import statements. This is supported in Qt Quick since Qt 6 too by leaving out the

version number, so if you can manage without it and need compatibility, make sure to leave out the version number.

```
import QtQuick
import QtQuickUltralite.Extras
```

In the root of our scene, we place a `Rectangle`. This is because Qt Quick Ultralite does not provide a default, white, background. By using a `Rectangle` as root, we ensure that we control the background color of the scene.

```
Rectangle {
    width: 480
    height: 272
```

The next part, the clickable `Rectangle`, is straight forward QML, with some Javascript bound to the `onClicked` event. Qt for MCUs has limited support for Javascript, so ensure to keep such scripts simple. You can read more about the specific limitations in the links at the end of this section.

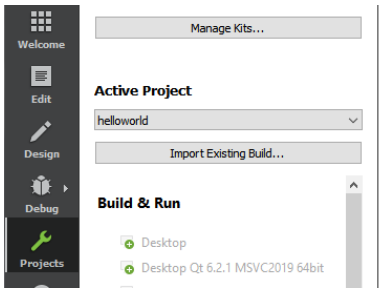
```
    Rectangle {
        id: rect
        anchors.fill: parent
        anchors.margins: 60

        color: "orange"
        Behavior on color {
            ColorAnimation { duration: 400 }
        }

        MouseArea {
            anchors.fill: parent

            onClicked: {
```

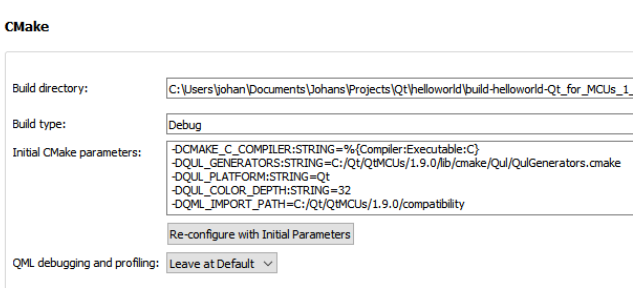
```
                if (rect.color == "red")
                    rect.color = "orange";
                else
                    rect.color = "red";
            }
        }
    }
```

Finally, the text is rendered using a `StaticText` element, which is a version of the `Text` element for static texts. This means that the text can be rendered once, or even pre-rendered, which can save a lot of resources on a small, MCU-based, system.

```
    StaticText {
        anchors.centerIn: parent
        color: "black"
        text: "Hello World!"
        font.pixelSize: 52
    }
}
```

In Qt Creator, you will notice that you get warnings around the `StaticText` element. This is because Qt Creator assumes that you are working with Qt Quick. To make Qt Creator aware of Qt Quick Ultralite, you need to set the `QML_IMPORT_PATH` to the path of your Qt for MCUs compatibility module. You can do this in your CMakeLists.txt file, or in your project settings. The project settings for a standard Windows 10 install is shown below.





In addition to what has been stated above, there are more differences. For instance, the Qt Quick Ultralite `Item` class, and hence the `Rectangle` class, lacks a lot of properties that could be found in Qt Quick. For instance, the `scale` and `rotation` properties are missing. These are only available for specific elements such as `Image`, and there it is made available through the `Rotation` and `Scale` types instead of properties.

Going beyond the example above, there are fewer QML elements in general in Qt Quick Ultralite, but the supported types is continuously growing. The intention is that the provided types cover the use-cases of the intended target devices. You can read more about this and general compatibility issues in the links provided below.

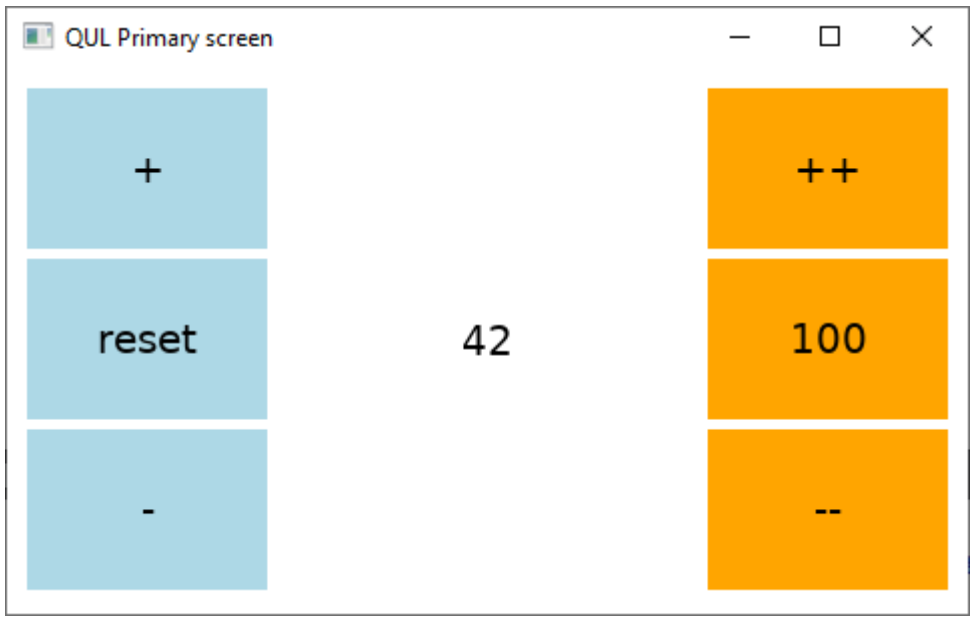
> **Links**
>
> Further reading at qt.io:
>
> - **Qt Quick Ultralite vs Qt Quick**
> - **Differences between Qt Quick Ultralite Controls and Qt Quick Controls**
> - **Known Issues and Limitations**

Integrating with C++

The C++

In order to demonstrate the connection between C++ and QML in Qt for MCUs, we will create a simple `Counter` singleton holding an integer value. Notice that we start from a `struct` and not a `class`. This is common practice in Qt Quick Ultralite.

The singleton will be used from a small UI as shown below.



The `Counter` struct provides a property, `value`, as well as methods for changing the value, `increase` and `decrease`, as well as a `reset` method. It also provides a signal, `hasBeenReset`.

```
#ifndef COUNTER_H
#define COUNTER_H
```

```
#include <qul/singleton.h>
#include <qul/property.h>
#include <qul/signal.h>

class Counter : public Qul::Singleton<Counter>
{
public:
    Counter();

    Qul::Property<int> value;

    void increase();
    void decrease();

    void reset();
    Qul::Signal<void(void)> hasBeenReset;

};

#endif // COUNTER_H
```

Coming from Qt, this looks odd. This is where Qt for MCUs shows the main differences. There is no `QObject` base class or `Q_OBJECT` macro, instead a new set of classes from the `Qul` is used. In this particular case, the base class is the `Qul::Singleton` class, creating a globally accessible singleton in the QML world. We also use the `Qul::Signal` class to create a signal and the `Qul::Property` class to create a property. All public, non-overloaded member functions are exposed to QML automatically.

> **TIP**
>
> To create an element that can be instantiated from QML, instead of a singleton, use the `Qul::Object` base class.

The struct is then exposed to QML using the CMake macro `qul_target_generate_interfaces`. Below you can see the `CMakeLists.txt`, based on the file generated by Qt Creator, with the `counter.h` and `counter.cpp` files added.

```
qul_target_generate_interfaces(cppintegration counter.h)
```

Now, let's continue with the implementation of the `Counter` struct. First up, for the `value` property, we use the `value` and `setValue` functions to access and modify the actual value. In our case, the property holds and `int`, but just as for the ordinary QML engine, types are **mapped between C++ and QML (https://doc.qt.io/QtForMCUs/qtul-integratecppqml.html#type-mapping)**.

This is used in the constructor, shown below, that sets the initial value to zero.

```
Counter::Counter()
{
    value.setValue(0);
}
```

The `increase` and `decrease` functions look similar. They use the getter and setter instead of interacting directly with the value.

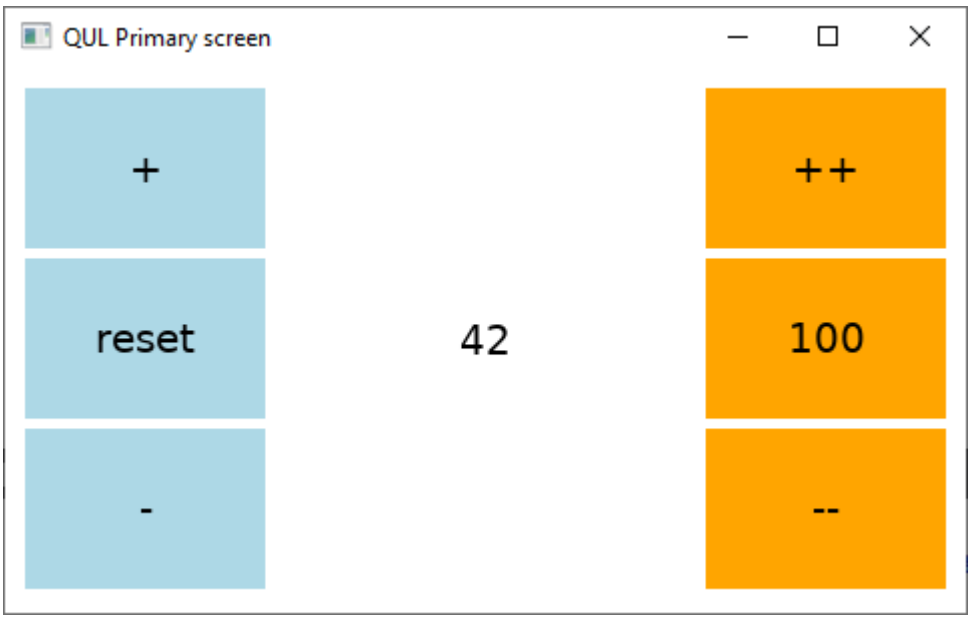
```
void Counter::increase()
{
    value.setValue(value.value()+1);
}

void Counter::decrease()
{
    value.setValue(value.value()-1);
}
```

`Counter` also has a signal. The signal is represented by the `Qul::Signal` instance named `hasReset`. The signal takes a function signature as template argument, so to create a signal carrying an integer, create a `Qul::Signal<void(int)>`. In this case, the signal does not carry any values, so it is defined as a `void(void)`. To emit the signal, we simply call it as if it was an ordinary function as shown in the `reset` function below.

```
void Counter::reset()
{
    std::cout << "Resetting from " << value.value() << std::end
    value.setValue(0);
    hasBeenReset();
}
```

The QML

The QML code produces the simple user interface shown below.



We will look at the UI in three parts. First, the basic structure, and bindings to `Counter.value`:

```
import QtQuick

Rectangle {
    width: 480
    height: 272

    Column {
        // Left buttons goes here
    }

    Column {
        // Right buttons goes here
    }

    Text {
        anchors.centerIn: parent
        text: Counter.value;
    }
}
```

As you can tell, the `Text` element's `text` property is bound to the `Counter.value` as in all QML.

Now, let's look at the left side buttons. These are used to invoke the C++ methods provided via the `Counter` singleton. The `PlainButton` is a QML element that we use to create these simple buttons. It lets you set the text, background color and a handler for the `clicked` signal. As you can tell, each button calls the corresponding method on the `Counter` singleton.

```
Column {
    x: 10
    y: 10
```

```
        spacing: 5
        PlainButton {
            text: "+"
            onClicked: Counter.increase()
        }
        PlainButton {
            text: "reset"
            onClicked: Counter.reset()
        }
        PlainButton {
            text: "-"
            onClicked: Counter.decrease()
        }
    }
```

The buttons on the right modify the `Counter.value` directly from QML. This is possible to do, but invisible to C++. There is no simple way for C++ to monitor if a property has changed, so if a C++ reaction is needed, it is recommended to use a setter method, rather than directly modifying the property value.

```
    Column {
        x: 350
        y: 10
        spacing: 5
        PlainButton {
            color: "orange"
            text: "++"
            onClicked: Counter.value += 5;
        }
        PlainButton {
            color: "orange"
            text: "100"
            onClicked: Counter.value = 100;
        }
        PlainButton {
            color: "orange"
```

```
            text: "--"
            onClicked: Counter.value -= 5;
        }
    }
```

This shows how to provide a singleton from C++ and how to make function calls, emit signals, and share state (properties) between C++ and QML.

Revisiting the CMake file

The `CMakeLists.txt` file may look familiar to you, but there are some tips and tricks that we need to discuss.

First of all, in order to expose a C++ class to QML, use the `qul_target_generate_interfaces`, e.g:

```
qul_target_generate_interfaces(cppintegration counter.h)
```

The other half, the QML files, are added using the `qul_target_qml_sources` macro. If you have multiple QML files, simply list them one by one as shown below:

```
qul_target_qml_sources(cppintegration cppintegration.qml PlainB
```

Another interesting aspect is that we are building a C++ project without writing a `main` function. This is taken care of by the `app_target_default_main` macro that adds a reference main implementation to the project. You can of course replace this with a custom `main` function if you need more control.

```
app_target_default_main(cppintegration cppintegration)
```

Finally, the libraries linked to are not the standard Qt ones, but the `Qul::` ones, e.g:

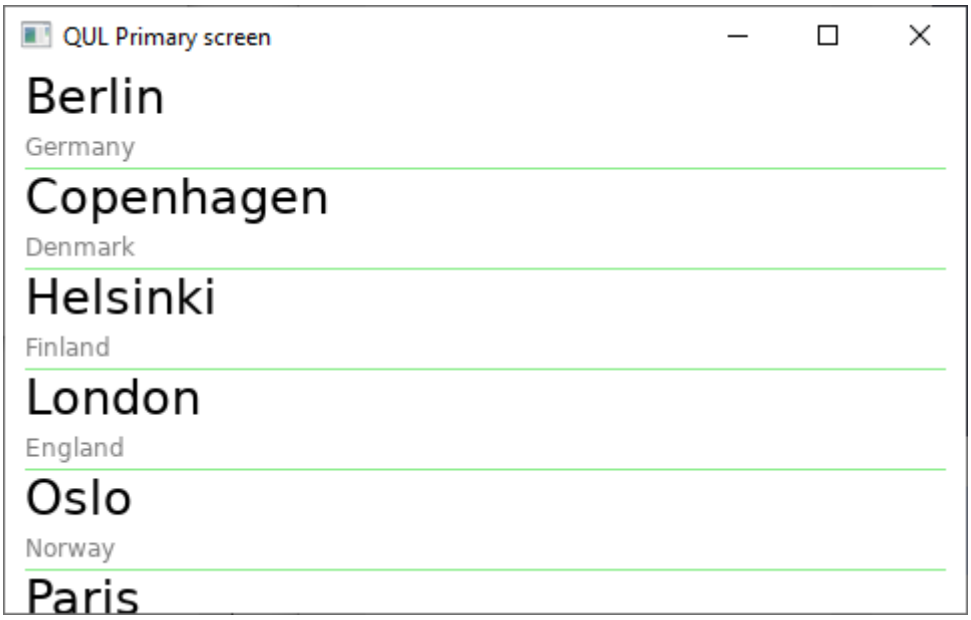
```
target_link_libraries(cppintegration
    Qul::QuickUltralite
    Qul::QuickUltralitePlatform)
```

> **Links**
>
> Further reading at qt.io:
>
> - **Integrate C++ and QML**

Working with Models

In Qt Quick Ultralite, it is possible to create models in QML using the `ListModel` element. It is also possible, and a bit more interesting, to create models from C++. This lets you expose lists of data from C++ to QML and to instantiate user interface elements for each item of the list. The setup is very similar to the ordinary Qt Quick, but the base classes and interfaces are more limited.

In this chapter we will create a list of cities in Europe, listing the name of the city and the country in which the city is located. The cities will be shown in a `ListView` as shown below:



The C++

To create a model in Qt Quick Ultralite, the first thing we need to do is to define a `struct` with the data of each list item. For this struct, we also

need to provide a `==` operator. This is what we do with the `CityData` struct shown below. Notice that we use `std::string` rather than `QString` in Qt Quick Ultralite. The assumption is that UTF-8 encoding is used.

```
#include <string>

struct CityData
{
    std::string name;
    std::string country;
};

inline bool operator==(const CityData &l, const CityData &r)
{
    return l.name == r.name && l.country == r.country;
}
```

Once the data type has been prepared, we declare the `CityModel` struct, inheriting from `Qul::ListModel`. This lets us define a model that can be accessed from QML. We must implement the `count` and `data` methods, which are similar, but not identical to, the corresponding methods from the `QAbstractListModel` class. We also use the `CMake macro `qul_target_generate_interfaces` to make the types available to QML.

```
#include <qul/model.h>
#include <platforminterface/allocator.h>

struct CityModel : Qul::ListModel<CityData>
{
private:
    Qul::PlatformInterface::Vector<CityData> m_data;

public:
```

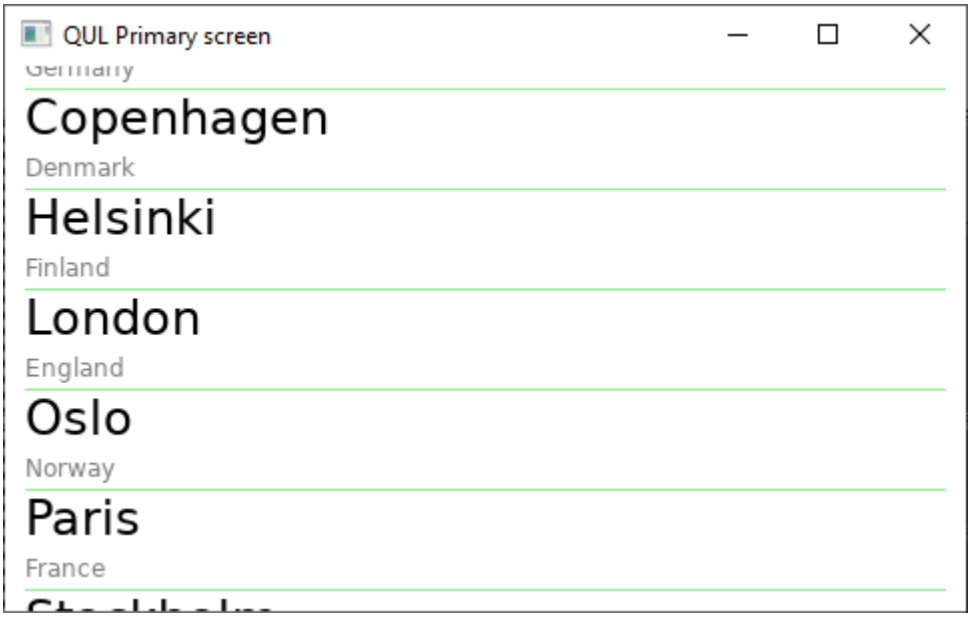
```
    CityModel();

    int count() const override { return m_data.size(); }
    CityData data(int index) const override { return m_data[ind
};
```

We also implement a constructor for the `CityModel` struct that populates the `m_data` vector with data.

```
#include "citymodel.h"

CityModel::CityModel()
{
    m_data.push_back(CityData {"Berlin", "Germany"});
    m_data.push_back(CityData {"Copenhagen", "Denmark"});
    m_data.push_back(CityData {"Helsinki", "Finland"});
    m_data.push_back(CityData {"London", "England"});
    m_data.push_back(CityData {"Oslo", "Norway"});
    m_data.push_back(CityData {"Paris", "France"});
    m_data.push_back(CityData {"Stockholm", "Sweden"});
}
```

The QML

In the example, we show the model as a scrollable list, as shown below.



The QML code is shown in its entirety below:

```
import QtQuick 2.0

Rectangle {
    width: 480
    height: 272

    CityModel {
        id: cityModel
    }

    Component {
        id: cityDelegate

        Item {
            width: 480
            height: 45

            Column {
                spacing: 2
                Text {
                    text: model.name
                    x: 10
```

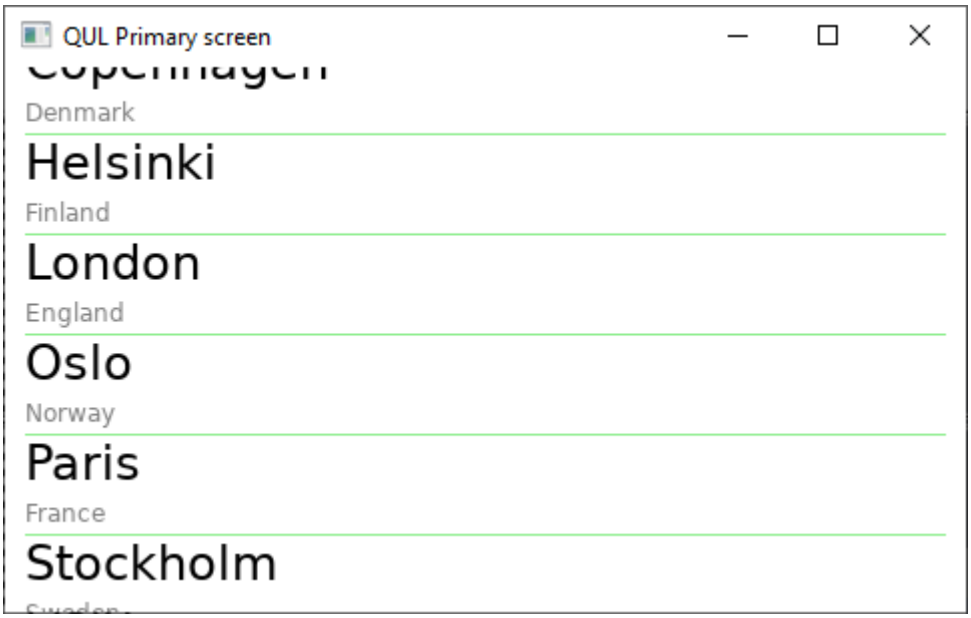
```
                    font: Qt.font({
                            pixelSize: 24,
                            unicodeCoverage: [Font.UnicodeBlock
                        })
                }
                Text {
                    text: model.country
                    x: 10
                    color: "gray"
                    font: Qt.font({
                            pixelSize: 12,
                            unicodeCoverage: [Font.UnicodeBlock
                        })
                }
                Rectangle {
                    color: "lightGreen"
                    x: 10
                    width: 460
                    height: 1
                }
            }
        }
    }

    ListView {
        anchors.fill: parent
        model: cityModel
        delegate: cityDelegate
        spacing: 5
    }
}
```

The example starts by instantiating the `cityModel`. As the model is not a singleton, it has to be instantiated from QML.

Then the delegate, `cityDelegate` is implemented as a `Component`. This means that it can be instantiated multiple times from QML. The model

data is accessed via the `model.name` and `model.country` attached properties.

Finally, the `ListView` element joins the model and the delegate, resulting in the list shown in the screenshots in this chapter.



An interesting aspect of the QML is how the font of the `Text` elements is configured. The `unicodeCoverage` property lets us tell the Qt Quick Ultralite compiler what characters we would like to be able to render. When specifying fixed strings, the Qt Quick Ultralite tooling generates minimal fonts containing exactly the glyphs that we intend to use. However, since the model will provide us with dynamic data, we need to tell the font what characters we expect to use.

When rendering a complete font, sometimes you encounter the following style of warnings:

```
[2/7 8.8/sec] Generating CMakeFiles/cppmodel.dir/qul_font_engin
Warning: Glyph not found for character "\u0000"
Warning: Glyph not found for character "\u0001"
Warning: Glyph not found for character "\u0002"
Warning: Glyph not found for character "\u0003"
```

```
Warning: Glyph not found for character "\u0004"
Warning: Glyph not found for character "\u0005"
Warning: Glyph not found for character "\u0006"
Warning: Glyph not found for character "\u0007"
...
```

These can safely be disregarded, unless you expect to show the character in question.

Summary

In this chapter we've scratched the surface of Qt for MCUs and Qt Quick Ultralite. These technologies bring Qt to much smaller platforms and make it truly embeddable. Through-out this chapter, we've used the virtual desktop target, which allows for quick prototyping. Targeting a specific board is just as easy, but requires access to hardware and the associated tools.

Key take-aways from using Qt Quick Ultralite is that there are fewer built in elements, and that some APIs are slightly more restricted. But given the intended target systems, this is usually not a hindrance. Qt Quick Ultralite also turns QML into a compiled language, which actually is nice. It allows you to catch more errors during compile time, instead of having to test more during run time.