《Qt5 Cadaques》中文版

Samuel Tseng

Published with GitBook

Table of Contents

《Qt5 Cadaques》in Chinese

中文版《Qt5 Cadaques》

github上的《The Swift Programming Language》 in Chinese 的共享方式让我觉得很不错，参照这个方式我翻译了《Qt5 Cadaques》。

QML的中文资料一直比较少，希望大家能喜欢。

在线阅读

使用Gitbook制作，可以直接在线阅读。

PDF下载

点我下载

当前阶段

Qt5 Cadaques上发布的课程已全部由我一个人翻译完成，但是没有校对过，希望大家可以帮忙校对。

本人渣英语，很多术语可能翻译不准确，如果有什么错误希望广大Qt爱好者谅解。

课程目录

    - QML语法（QML Syntax）
    - 基本元素（Basic Elements）
    - 组件（Compontents）
    - 简单的转换（Simple Transformations）
    - 定位元素（Positioning Element）
    - 布局元素（Layout items）
    - 输入元素（Input Element）
    - 高级用法（Advanced Techniques）
- Fluid Elements
    - 动画（Animations)
    - 状态与过渡（States and Transitions）
    - 高级用法（Advanced Techniques）
- Model-View-Delegate
    - 概念（Concept）
    - 基础模型（Basic Model）
    - 动态视图（Dynamic Views）
    - 代理（Delegate）
    - 高级用法（Advanced Techniques）
    - 总结（Summary）
- Canvas Element
    - 便捷的接口（Convenient API）
    - 渐变（Gradients）
    - 阴影（Shadows）
    - 图片（Images）
    - 转换（Transformation）
    - 组合模式（Composition Mode）
    - 像素缓冲（Pixels Buffer）
    - 画布绘制（Canvas Paint）
    - HTML5画布移植（Porting from HTML5 Canvas）
- Particle Simulations
    - 概念（Concept）
    - 简单的模拟（Simple Simulation）
    - 粒子参数（Particle Parameters）
    - 粒子方向（Directed Particle）
    - 粒子画笔（Particle Painter）
    - 粒子控制（Affecting Particles）
    - 粒子组（Particle Group）
    - 总结（Summary）
- Shader Effect
    - OpenGL着色器（OpenGL Shader）
    - 着色器元素（Shader Elements）
    - 片段着色器（Fragment Shader）
    - 波浪效果（Wave Effect）
    - 顶点着色器（Vertex Shader）
    - 剧幕效果（Curtain Effect）
    - Qt图像效果库（Qt GraphicsEffect Library）

- Multimedia
  - 媒体播放（Playing Media）
  - 声音效果（Sounds Effects）
  - 视频流（Video Streams）
  - 捕捉图像（Capturing Images）
  - 高级用法（Advanced Techniques）
  - 总结（Summary）
- Networking
  - 通过HTTP服务UI（Serving UI via HTTP）
  - 模板（Templating）
  - HTTP请求（HTTP Requests）
  - 本地文件（Local files）
  - REST接口（REST API）
  - Engine IO
  - Web Sockets
  - 总结（Summary）

原作者

感谢原作者Juergen Bocklage-Ryannel和Johan Thelin的分享，Qt5 Cadaques地址http://qmlbook.org/

开源协议

问题与建议

帮忙校对可以参考《The Swift Programming Language》 in Chinese的流程，我会及时合并，有任何建议可以在项目issue中提出，或者email我：cwc1987@163.com

Meet Qt 5

注意

这章的源代码能够在**assetts folder**找到。

在这一章是对于Qt5的一个概述，它展示了开发者可以使用的不同开发模型和Qt5程序的预演。另外本章的内容提供一个广泛的Qt5概述和如何与Qt的开发者联系。

这本书向你提供了使用Qt开发不同的应用程序。它主要关注了新的Qt Quick的技术，但也提供了如何使用C++渲染后端的方法和Qt Quick的扩展方法。

序（Preface）

历史

Qt4自2005年发布以来向成千上万的应用程序提供了开发框架，甚至是完整的桌面与移动系统。在最近几年计算机的使用模式发生了改变。从PC机向便携式设备和移动电脑发展。传统的桌面设备被越来越多的基于触摸屏的手机设备取代。桌面用户的体验模式也在发生改变。在过去，Windows UI占据了我们的世界，但现在我们会花更多的时间在其它的UI语言上。

Qt4设计用于满足在大多数主流平台的桌面上有一个可以使用的UI窗口部件。如今对于Qt的开发者面临新的问题，它将提供更多的基于用户触摸驱动的用户界面并且适用于大多数主流桌面与移动系统。Qt4.7开始引进了QtQuick技术，允许用户创建一个满足客户需求的，从简单的元素来实现一个完整的新的用户界面。

1.1.1 Qt5关注方面（Qt5 Focus）

Qt5是Qt4版本完整的更新，到Qt4.8版本，Qt4已经发布了7年。是时候让这个令人惊奇的工具更加惊奇了。

Qt5主要关注以下方面：

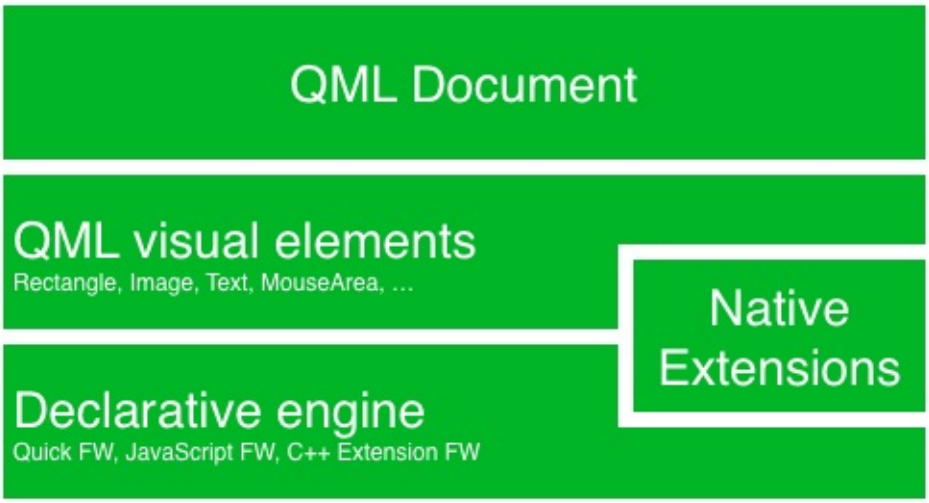
- 杰出的图形绘制：Qt Quick2是基于OpenGL(ES)场景的实现。重组的图形堆栈可以得到更加好的图形效果与更加简单的使用方法，在这一领域是之前是从未实现的。
- 开发者生产率：QML和JavaScript语言是主要用于创建UI的方法。后端将有C++来完成绘制。将JavaScript与C++分开能够快速的迭代开发，让前端的开发人员专注于创建漂亮的用户界面，后端的C++开发人员专注于稳定，性能和扩展。
- 跨平台移植性：基于Qt平台的统一抽象概念，现在可以更加容易和快速的将Qt移植到更多的平台上。Qt5是一个围绕Qt必要组件和附加组件的概念，操作系统开发者只需要专注于必要模块的实现，可以使程序更加效率的运行。
- 开放的开发：Qt是由Qt-Porject(qt-project.org)主持的开放管理的项目，它的开发是开放的，由Qt社区驱动的。

Qt5介绍（Qt5 Introduction）

1.2.1 Qt Quick

Qt Quick是Qt5中用户界面技术的涵盖。Qt Quick自身包含了以下几种技术：

- QML-使用于用户界面的标识语言
- JavaScript-动态脚本语言
- Qt C++-具有高度可移植性的C++库.



类似HTML语言，QML是一个标识语言。它由QtQuick封装在Item {}的元素的标识组成。它从头设计了用户界面的创建，并且可以让开发人员快速，简单的理解。用户界面可以使用JavaScript代码来提供和加强更多的功能。Qt Quick可以使用你自己本地已有的Qt C++轻松快速的扩展它的能力。简单声明的UI被称作前端，本地部分被称作后端。这样你可以将程序的计算密集部分与来自应用程序用户界面操作部分分开。

在典型的项目中前端开发使用QML/JaveScript，后端代码开发使用Qt C++来完成系统接口和繁重的计算工作。这样就很自然的将设计界面的开发者和功能开发者分开了。后端开发测试使用Qt自有的单元测试框架后，导出给前端开发者使用。

1.2.2 一个用户界面（Digesting an User Interface）

让我们来使用QtQuick来创建一个简单的用户界面，展示QML语言某些方面的特性。最后我们将获得一个旋转的风车。

我们开始创建一个空的main.qml文档。所有的QML文件都已.qml作为后缀。作为一个标识语言（类似HTML）一个QML文档需要并且只有一个根元素，在我们的案例中是一个基于background的图像高度与宽度的几何图形元素：

```
import QtQuick 2.0
```

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

QML不会对根元素设置任何限制，我们使用一个backgournd图像作为资源的图像元素来作为我们的根元素。

注意

每一个元素都有属性，比如一个图像有宽度，高度但是也有一些其它的属性例如资源。图像元素的大小能够自动的从图像大小上得出。否则我们应该设置宽度和高度属性来显示有效的像素。

大多数典型的元素都放置在**QtQuick2.0**模块中，我们首先应该在第一行作这个重要的声明。

id是这个特殊的属性是可选的，包含了一个标识符，在文档后面的地方可以直接引用。

重要提示：一个**id**属性无法在它被设置后改变，并且在程序执行期间无法被设置。使用**root**作为根元素**id**仅仅是作者的习惯，可以在比较大的**QML**文档中方便的引用最顶层元素。

风车作为前景元素使用图像的方式放置在我们的用户界面上。

正常情况下你的用户界面应该有不同类型的元素构成，而不是像我们的例子一样只有图像元素。

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

为了把风车放在中间的位置，我们使用了一个复杂的属性，称之为锚。锚定允许你指定几何对象与父对象或者同级对象之间的位置关系。比如放置我在另一个元素中间（anchors.centerIn:parent）.有左边（left），右边（right），顶部（top），底部（bottom），中央（cengerIn），填充（fill），垂直中央（verticalCenter）和水平中央（horizontalCenter）来表示元素之间的关系。确保他们能够匹配，锚定一个对象的左侧顶部的一个元素这样的做法是没有意义的。所以我们设置风车在父对象background的中央。

注意

有时你需要进行一些微小的调整。使用**anchors.horizontalCenterOffset**或者**anchors.verticalCenterOffset**可以帮你实现这个功能。类似的调整属性也可以用于其他所有的锚。查阅**Qt**的帮助文档可以知道完整的锚属性列表。

注意

将一个图像作为根矩形元素的子元素放置展示了一种声明式语言的重要概念。你描述了用户界面的层和分组的顺序，最顶部的一层（根矩形框）先绘制，然后子层按照包含它的元素局部坐标绘制在包含它的元素上。

为了让我们的展示更加有趣一点，我们应该让程序有一些交互功能。当用户点击场景上某个位置时，让我们的风车转动起来。

我们使用mouseArea元素，并且让它与我们的根元素大小一样。

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

当用户点击覆盖区域时，鼠标区域会发出一个信号。你可以重写onClicked函数来链接这个信号。在这个案例中引用了风车的图像并且让他旋转增加90度。

注意

对于每个工作的信号，命名方式都是**on + SignalName**的标题。当属性的值发生改变时也会发出一个信

号。它们的命名方式是：**on + PropertyName + Chagned**。 如果一个宽度（**width**）属性改变了，你可以使用**onWidthChanged: print(width)**来得到这个监控这个新的宽度值。

现在风车将会旋转，但是还不够流畅。风车的旋转角度属性被直接改变了。我们应该怎样让90度的旋转可以持续一段时间呢。现在是动画效果发挥作用的时候了。一个动画定义了一个属性的在一段时间内的变化过程。为了实现这个效果，我们使用一个动画类型叫做属性行为。这个行为指定了一个动画来定义属性的每一次改变并赋值给属性。每次属性改变，动画都会运行。这是QML中声明动画的几种方式中的一种方式。

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

现在每当风车旋转角度发生改变时都会使用NumberAnimation来实现250毫秒的旋转动画效果。每一次90度的转变都需要花费250ms。

现在风车看起来好多了，我希望以上这些能够让你能够对Qt Quick编程有一些了解。

Qt构建模块（Qt Building Blocks）

Qt5是由大量的模块组成的。一个模块通常情况下是一个库，提供给开发者使用。一些模块是强制性用来支持Qt平台的，它们分成一组叫做Qt基础模块。许多模块是可选的，它们分成一组叫做Qt附加模块，预计大多数得到开发人员将不会使用它们，但是最好知道它们可以对一些通用的问题提供非常有价值的解决方案。

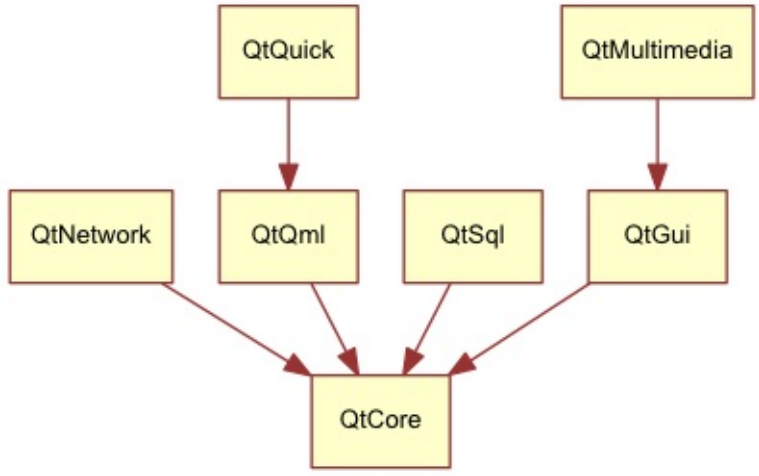
1.3.1 Qt模块（Qt Modules）

Qt基础模块是对Qt一台的必要支持。它们使用Qt Quick 2开发Qt 5应用程序的基础。

核心基础模块

以下这些是启动QML程序最小的模块集合。

模块名	描述
Qt Core	核心的非图形类，供其它模块使用。
Qt GUI	图形用户界面（GUI）组件的基类，包括OpenGL。
Qt Multimedia	音频，视频，电台，摄像头的功能类。
Qt Network	简化方便的网络编程的类。
Qt QML	QML类与JavaScript语言的支持。
Qt Quick	可高度动态构建的自定义应用程序用户界面框架。
Qt SQL	集成SQL数据库类。
Qt Test	Qt应用程序与库的单元测试类。
Qt WebKit	集成WebKit2的基础实现并且提供了新的QML应用程序接口。在附件模块中查看Qt WebKit Widgets可以获取更多的信息。
Qt WebKit Widgets	Widgets 来自Qt4中集成WebKit1的窗口基础类。
Qt Widgets	扩展Qt GUI模块的C++窗口类。



Qt附加模块

除了必不可少的基础模块，Qt提供了附加模块供软件开发者使用，这部分不一定包含在发布的版本中。以下简短的列出了一些可用的附加模块列表。

- Qt 3D - 一组使3D编程更加方便的应用程序接口和声明。
- Qt Bluetooth - 在多平台上使用无线蓝牙技术的C++和QML应用程序接口。
- Qt Contacts - 提供访问联系人与联系人数据库的C++和QML应用程序接口。
- Qt Location - 提供了定位，地图，导航和位置搜索的C++与QML接口。使用NMEA在后端进行定位。（NMEA缩写，同时也是数据传输标准工业协会，在这里，实际上应为NMEA 0183。它是一套定义接收机输出的标准信息，有几种不同的格式，每种都是独立相关的ASCII格式，逗点隔开数据流，数据流长度从30-100字符不等，通常以每秒间隔选择输出，最常用的格式为"GGA"，它包含了定位时间，纬度，经度，高度，定位所用的卫星数，DOP值,差分状态和校正时段等，其他的有速度，跟踪，日期等。NMEA实际上已成为所有的GPS接收机和最通用的数据输出格式，同时它也被用于与GPS接收机接口的大多数的软件包里。）
- Qt Organizer - 提供了组织事件（任务清单，事件等等）的C++和QML应用程序接口。
- Qt Publish and SubScribe - Qt发布与订阅
- Qt Sensors - 访问传感器的QML与C++接口。
- Qt Service Framework - 允许应用程序读取，操纵和订阅来改变通知信息。
- Qt System Info - 发布系统相关的信息和功能。
- Qt Versit - 支持电子名片与日历数据格式（iCalendar）。（iCalendar是“日历数据交换”的标准（RFC 2445）。 此标准有时指的是“iCal”，即苹果公司的出品的一款同名日历软件，这个软件也是此标准的一种实现方式。）
- Qt Wayland - 只用于Linux系统。包含了Qt合成器应用程序接口（server），和Wayland平台插件（clients）。
- Qt Feedback - 反馈用户的触摸和声音操作。
- Qt JSON DB - 对于Qt的一个不使用SQL对象存储。

注意

这些模块一部分还没有发布，这依赖于有多少贡献者，并且它们能够获得更好的测试。

1.3.2 支持的平台（Supported Platforms）

Qt支持各种不同的平台。大多数主流的桌面与嵌入式平台都能够支持。通过Qt应用程序抽象，现在可以更容易的将Qt移植到你自己的平台上。在一个平台上测试Qt5是非常花费时间的。选择测试的平台子集可以参考qt-project构件的平台设置。这些平台需要完全通过系统的测试才能确保最好的质量。友情提醒：任何代码都可能会有Bug的。

Qt项目（Qt Project）

来自qt-project百科：Qt-Project是由Qt社区上对Qt感兴趣的人达成共识的地方。任何人都可以在社区上分享它感兴趣的东西，参与它的开发，并且向Qt的开发做出贡献。

Qt-Project是一个为Qt未来开发开源部分的组织。它基于使用者的贡献。最大的贡献者是DIGIA，它可以提供Qt的商业授权。 Qt对于公司分为开源方向和商业方向。商业方向的公司不需要遵守开源协议。没有商业方向的许可的公司不能使用Qt，并且它也不允许DIGIA向Qt项目贡献太多的代码。

在全球有很多公司，他们在不同的平台上使用Qt开发产品，提供咨询。同样也有很多开源项目和开源开发者，它们使用Qt作为它们的开发库。成为这样开发活泼的社区的一部分，并且使用这个很棒的工具盒库让人感觉很好。它能让你成为一个更好的人吗？也许:-)。

Get Start

这一章介绍了如何使用Qt5进行开发。我们将告诉你如何安装Qt软件开发工具包（Qt SDK）和如何使用Qt Creator集成开发环境（Qt Creator IDE）创建并运行一个简单的hello word应用程序。

注意

这章的源代码能够在**assetts folder**找到。

安装Qt5软件工具包（Installing Qt 5 SDK）

Qt软件工具包包含了编译桌面或者嵌入式应用程序的工具。最新的版本可以从Qt-Project下载。我们将使用这种方法开始。

软件工具包自身包含了一个维护工具允许你更新到最新版本的软件工具包。

Qt软件工具包非常容易安装，并且附带了一个它自身的快速集成开发环境叫做Qt Creator。这个集成开发环境可以让你高效的使用Qt进行开发，我们推荐给所有的读者使用。在任何情况下Qt都可以通过命令的方式来编译，你可以自由的选择你的代码编辑器。

当你安装软件工具包时，你最好选择默认的选项确保Qt 5.x可以被使用。然后一切准备就绪。

你好世界（**Hello World**）

为了测试你的安装，我们创建一个简单的应用程序hello world.打开Qt Creator并且创建一个Qt Quick UI Project（File->New File 或者 Project-> Qt Quick Project -> Qt Quick UI）并且给项目取名 HelloWorld。

注意

Qt Creator集成开发环境允许你创建不同类型的应用程序。如果没有另外说明，我们都创建**Qt Quick UI Project**。

提示

一个典型的**Qt Quick**应用程序在运行时解释，与本地插件或者本地代码在运行时解释代码一样。对于才开始的我们不需要关注本地端的解释开发，只需要把注意力集中在**Qt5**运行时的方面。

Qt Creator将会为我们创建几个文件。HellWorld.qmlproject文件是项目文件，保存了项目的配置信息。这个文件由Qt Creator管理，我们不需要编辑它。

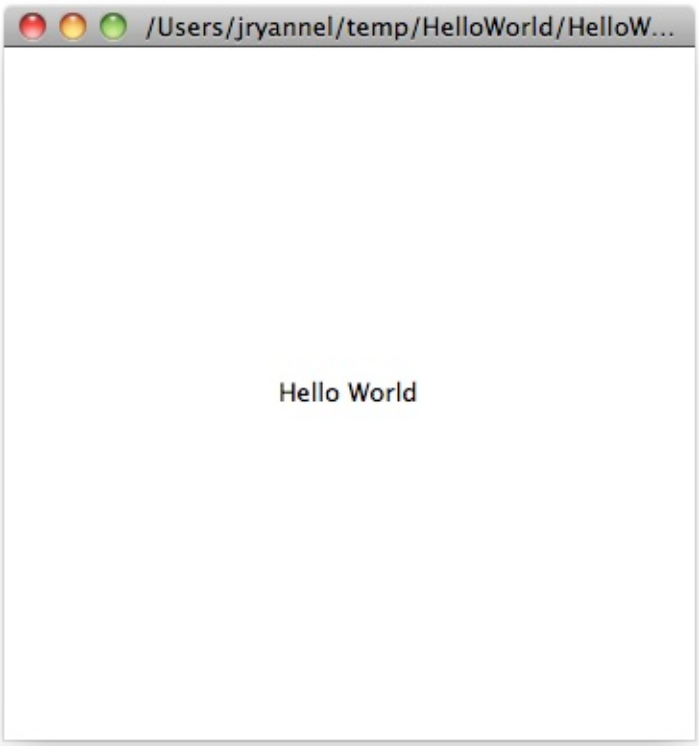
另一个文件HelloWorld.qml保存我们应用程序的代码。打开它，并且尝试想想这个应用程序要做什么，然后再继续读下去。

```
// HelloWorld.qml

import QtQuick 2.0

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

HelloWorld.qml使用QML语言来编写。我们将在下一章更深入的讨论QML语言，现在只需要知道它描述了一系列有层次的用户界面。这个代码指定了显示一个360乘以360像素的一个矩形，矩形中间有一个“Hello World"的文本。鼠标区域覆盖了整个矩形，当用户点击它时，程序就会退出。

你自己可以运行这个应用程序，点击左边的运行或者从菜单选择select Bulid->Run。

如果一切顺利，你将看到下面这个窗口：



Qt 5似乎已经可以工作了，我们接着继续。

建议

如果你是一个名系统集成人员，你会想要安装最新稳定的**Qt**版本，将这个**Qt**版本的源代码编译到你特定的目标机器上。

从头开始构建

如果你想使用命令行的方式构建Qt5，你首先需要拷贝一个代码库并构建他。

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

等待两杯咖啡的时间编译完成后，qtbase文件夹中将会出现可以使用的Qt5。任何饮料都好，不过我喜欢喝着咖啡等待最好的结果。

如果你想测试你的编译，只需简单的启动qtbase/bin/qmlscene并且选择一个QtQuick的例子运行，或者跟着我们进入下一章。

为了测试你的安装，我们创建了一个简单的hello world应用程序。创建一个空的qml文件example1.qml，使用你最喜爱的文本编辑器并且粘贴一下内容：

```
// HelloWorld.qml

import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Greetings from Qt5"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

你现在使用来自Qt5的默认运行环境来可以运行这个例子：

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

应用程序类型（Application Types）

这一节贯穿了可能使用Qt5编写的不同类型的应用程序。没有任何建议的选择，只是想告诉读者Qt5通常情况下能做些什么。

2.3.1 控制台应用程序

一个控制台应用程序不需要提供任何人机交互图形界面通常被称作系统服务，或者通过命令行来运行。Qt5附带了一系列现成的组件来帮助你非常有效的创建跨平台的控制台应用程序。例如网络应用程序编程接口或者文件应用程序编程接口，字符串的处理，自Qt5.1发布的高效的命令解析器。由于Qt是基于C++的高级应用程序接口，你能够快速的编程并且程序拥有快速的执行速度。不要认为Qt仅仅只是用户界面工具，它也提供了许多其它的功能。

字符串处理

在第一个例子中我们展示了怎样简单的增加两个字符串常量。这不是一个有用的应用程序，但能让你了解本地端C++应用程序没有事件循环时是什么样的。

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

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

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

容器类

这个例子在应用程序中增加了一个链表和一个链表迭代器。Qt自带大量方便使用的容器类，并且其中的元素使用相同的应用程序接口模式。

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

```
    cout << iter.next();
    if(iter.hasNext()) {
        cout << " ";
    }
}
cout << "!" << endl;
```

这里我们展示了一些高级的链表函数，允许你在一个字符串中加入一个链表的字符串。当你需要持续的文本输入时非常的方便。使用QString::split()函数可以将这个操作逆向（将字符串转换为字符串链表）。

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

文件**IO**

下一个代码片段我们从本地读取了一个CSV文件并且遍历提取每一行的每一个单元的数据。我们从CSV文件中获取大约20行的编码。文件读取仅仅给了我们一个比特流，为了有效的将它转换为可以使用的Unicode文本，我们需要使用这个文件作为文本流的底层流数据。编写CSV文件，你只需要以写入的方式打开一个文件并且一行一行的输入到文件流中。

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

现在我们结束Qt关于基于控制台应用程序小节。

2.3.2 窗口应用程序

基于控制台的应用程序非常方便，但是有时候你需要有一些用户界面。但是基于用户界面的应用程序需要后端来写入/读取文件，使用网络进行通讯或者保存数据到一个容器中。

在第一个基于窗口的应用程序代码片段，我们仅仅只创建了一个窗口并显示它。没有父对象的窗口部件是Qt世界中的一个窗口。我们使用智能指针来确保当智能指针指向范围外时窗口会被删除掉。

这个应用程序对象封装了Qt的运行，调用exec开始我们的事件循环。从这里开始我们的应用程序只响应由鼠标或者键盘或者其它的例如网络或者文件IO的事件触发。应用程序也只有在事件循环退出时才退出，在应用程序中调用"quit()"或者关掉窗口来退出。 当你运行这段代码的时候你可以看到一个240乘以120像素的窗口。

```
#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();
}
```

自定义窗口部件

当你使用用户界面时你需要创建一个自定义的窗口部件。典型的窗口是一个窗口部件区域的绘制调用。附加一些窗口部件内部如何处理外部触发的键盘或者鼠标输入。为此我们需要继承QWidget并且重写几个函数来绘制和处理事件。

```
#ifndef CUSTOMWIDGET_H
#define CUSTOMWIDGET_H

#include <QtWidgets>

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

#endif // CUSTOMWIDGET_H
```

在实现中我们绘制了窗口的边界并在鼠标最后的位置上绘制了一个小的矩形框。这是一个非常典型的低层

次的自定义窗口部件。鼠标或者键盘事件会改变窗口的内部状态并触发重新绘制。我们不需要更加详细的分析这个代码，你应该有能力分析它。Qt自带了大量现成的桌面窗口部件，你有很大的几率不需要再做这些工作。

```
#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();
}
```

桌面窗口

Qt的开发者们已经为你做好大量现成的桌面窗口部件，在不同的操作系统中他们看起来都像是本地的窗口部件。你的工作只需要在一个打的窗口容器中安排不同的的窗口部件。在Qt中一个窗口部件能够包含其它的窗口部件。这个操作由分配父子关系来完成。这意味着我们需要准备类似按钮（button），复选框（check box），单选按钮（radio button）的窗口部件并且对它们进行布局。下面展示了一种完成的方法。

这里有一个头文件就是所谓的窗口部件容器。

```
class CustomWidget : public QWidget
{
    Q_OBJECT
public:
    explicit CustomWidgetQWidget *parent = 0);
private slots:
```

```
    void itemClicked(QListWidgetItem* item);
    void updateItem();
private:
    QListWidget *m_widget;
    QLineEdit *m_edit;
    QPushButton *m_button;
};
```

在实现中我们使用布局来更好的安排我们的窗口部件。当容器窗口部件大小被改变后它会按照窗口部件的大小策略进行重新布局。在这个例子中我们有一个链表窗口部件，行编辑器与按钮垂直排列来编辑一个城市的链表。我们使用Qt的信号与槽来连接发送和接收对象。

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

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

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

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

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

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

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

绘制图形

有一些问题最好用可视化的方式表达。如果手边的问题看起来有点像几何对象，qt graphics view是一个很

好的选择。一个图形视窗（graphics view）能够在一个场景（scene）排列简单的几何图形。用户可以与这些图形交互，它们使用一定的算法放置在场景（scene）上。填充一个图形视图你需要一个图形窗口（graphics view）和一个图形场景（graphics scene）。一个图形场景（scene）连接在一个图形窗口（view）上，图形对象（graphics item）是被放在图形场景（scene）上的。这里有一个简单的例子，首先头文件定义了图形窗口（view）与图形场景（scene）。

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

};
```

在实现中首先将图形场景（scene）与图形窗口（view）连接。图形窗口（view）是一个窗口部件，能够被我们的窗口部件容器包含。最后我们添加一个小的矩形框在图形场景（scene）中。然后它会被渲染到我们的图形窗口（view）上。

```
#include "customwidgetv2.h"

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

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

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

2.3.3 数据适配

到现在我们已经知道了大多数的基本数据类型，并且知道如何使用窗口部件和图形视图（graphics views）。通常在应用程序中你需要处理大量的结构体数据，也可能需要不停的储存它们，或者这些数据需要被用来显示。对于这些Qt使用了模型的概念。下面一个简单的模型是字符串链表模型，它被一大堆字符串填满然后与一个链表视图（list view）连接。

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

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

另一个比较普遍的用法是使用SQL（结构化数据查询语言）来存储和读取数据。Qt自身附带了嵌入式版的SQLLite并且也支持其它的数据引擎（比如MySQL，PostgresSQL，等等）。首先你需要使用一个模式来创建你的数据库，比如像这样：

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

为了能够在使用sql，我们需要在我们的项目文件（*.pro）中加入sql模块。

```
QT += sql
```

然后我们需要c++来打开我们的数据库。首先我们需要获取一个指定的数据库引擎的数据对象。使用这个数据库对象我们可以打开数据库。对于SQLLite这样的数据库我们可以指定一个数据库文件的路径。Qt提供了一些高级的数据库模型，其中有一种表格模型（table model）使用表格标示符和一个选项分支语句（where clause）来选择数据。这个模型的结果能够与一个链表视图连接，就像之前连接其它数据模型一样。

```
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();
```

对高级的模型操作，Qt提供了一种分类文件代理模型，允许你使用基础的分类排序和数据过滤来操作其它的模型。

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

数据过滤基于列号与一个字符串参数完成。

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

过滤代理模型比这里演示的要强大的多，现在我们只需要知道有它的存在就够了。

注意

这里是综述了你可以在**Qt5**中开发的不同类型的经典应用程序。桌面应用程序正在发生着改变，不久之后移动设备将会为占据我们的世界。移动设备的用户界面设计非常不同。它们相对于桌面应用程序更加简洁，只需要专注的做一件事情。动画效果是一个非常重要的部分，用户界面需要生动活泼。传统的**Qt**技术已经不适于这些市场了。

接下来：**Qt Quick**将会解决这个问题。

2.3.4 Qt Quick应用程序

在现代的软件开发中有一个内在的冲突，用户界面的改变速度远远高于我们的后端服务。在传统的技术中我们开发的前端需要与后端保持相同的步调。当一个项目在开发时用户想要改变用户界面，或者在一个项目中开发一个用户界面的想法就会引发这个冲突。敏捷项目需要敏捷的方法。

Qt Quick 提供了一个类似HTML声明语言的环境应用程序作为你的用户界面前端（the front-end），在你的后端使用本地的c++代码。这样允许你在两端都游刃有余。

下面是一个简单的Qt Quick UI的例子。

```
import QtQuick 2.0

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

这种声明语言被称作QML，它需要在运行时启动。Qt提供了一个典型的运行环境叫做qmlscene，但是想要写一个自定义的允许环境也不是很困难，我们需要一个快速视图（quick view）并且将QML文档作为它的资源。剩下的事情就只是展示我们的用户界面了。

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

回到我们之前的例子，在一个例子中我们使用了一个c++的城市数据模型。如果我们能够在QML代码中使用它将会更加的好。

为了实现它我们首先要编写前端代码怎样展示我们需要使用的城市数据模型。在这一个例子中前端指定了一个对象叫做cityModel，我们可以在链表视图（list view）中使用它。

```
import QtQuick 2.0

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

为了使用cityModel，我们通常需要重复使用我们以前的数据模型，给我们的根环境（root context）加上一个内容属性（context property）。（root context是在另一个文档的根元素中）。

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

警告

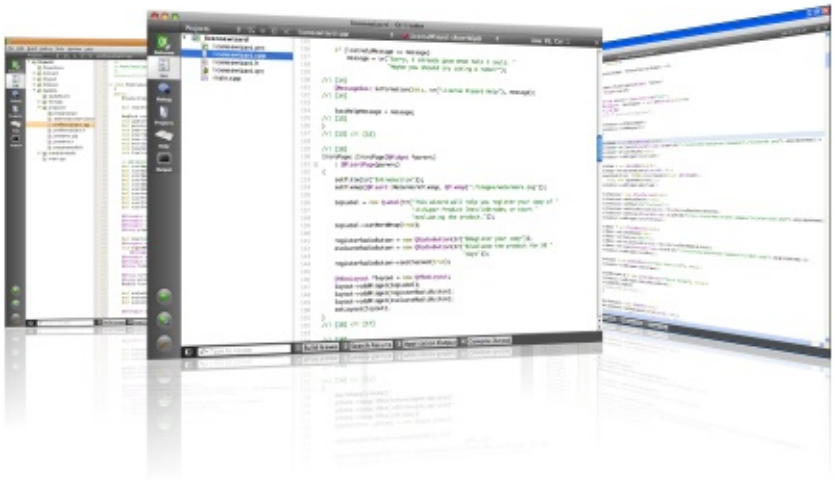
这不是完全正确的用法，作为包含在**SQL**表格模型列中的数据，一个**QML**模型的任务是来表达这些数据。所以需要做一个在列和任务之间的映射关系。请查看来**QML and QSqlTableModel**获得更多的信息。

总结（ Summary）

我们已经知道了如何安装Qt软件开发工具包，并且知道如何创建我们的应用。我们向你展示和概述了使用Qt开发不同类型的应用程序。展示Qt可以给你的应用程序开发提供的一些功能。我希望你对Qt留下一个好的印象，Qt是一个非常好的用户界面开发工具并且尽可能的提供了一个应用开发者期望的东西。当前你也不必一直锁定使用Qt，你也可以使用其它的库或者自己来扩展Qt。Qt对于不同类型的应用程序开发支持非常丰富：包括控制台程序，经典的桌面用户界面程序和触摸式用户界面程序。

Qt Creator IDE

Qt Creator是Qt默认的集成开发环境。它由Qt的开发者们编写提供的。这个集成开发环境能够在大多数的桌面开发平台上使用，例如 Windows/Mac/Linux。我们也已经看到有些用户在嵌入式设备上使用Qt Creator。Qt Creator有着精简的用户界面，可以帮助开发者们高效的完成开发生产。Qt Creator 能够启动你的QtQuick用户界面，也可以用来编译c++代码到你的主机系统或者使用交叉编译到你的设备系统上。



注意

这章的源代码能够在**assetts folder**找到。

用户界面（The User Interface）

当你启动Qt Creator时，你可以看到一个欢迎画面。在这里你可以找到怎样在Qt Creator中继续的重要提示，或者你最近使用的项目。你可以看到一个会话列表，你可以看到是一个空的。一个会话是供你参考使用的一堆项目的集合。当你在同时拥有几个客户的大项目时，这个功能非常有用。

你可以在左边看到模式选择。模式选择包含了你典型的工作步骤。

- 欢迎模式：你目前所在的位置。
- 编辑模式：专注于编码。
- 设计模式：专注于用户界面设计。
- 调试模式：获取当前运行程序的相关信息。
- 项目模式：修改你的项目编译运行配置。
- 分析模式：检查内存泄露并剖析。
- 帮助模式：阅读Qt的帮助文档。

在模式选择下面你可以找到项目配置选择与执行/调试。

你应该大多数时间都处于编辑模式的中央面板中的代码编辑器编辑你的代码。当你需要配置你的项目时，你将不时的访问项目模式。当你点击Run（运行）。Qt Creator会先确保充分的构建你的项目后再运行它。

在最下面的输出窗是错误信息，应用程序信息，编译信息和其它的信息。

注册你的Qt工具箱（Registering your Qt Kit）

最开始使用Qt Creator时最困难的部分可能是Qt Kit。一个Qt Kit由Qt的版本，编译系统和设备等等其它设置来配置它。它使用唯一标识的工具组合来构建你的项目。一个典型的桌面kit（工具箱）可能包含一个GCC编译程序，一个Qt版本库（比如Qt5.1.1）和一个设备（”桌面“）。在你创建好你的项目后你需要为项目指定一个kit（工具箱）来构建项目。在你创建一个kit（工具箱）之前你需要先安装一个编译程序并注册一个Qt版本。Qt版本的注册由指定qmake的执行路径完成。Qt Creator通过查询qmake的信息来获取Qt的版本标识。

添加kit（工具箱）与注册Qt版本在Settings->Bulild & Run entry中完成，在这里你也可以查看有哪些编译程序已经被注册了的。

注意

请首先确保你的**Qt Creator**中已经注册了正确的**Qt**版本，并且确保一个**Kit**（工具箱）指定了一个编译程序与**Qt**版本和设备的组合。你无法离开**Kit**（工具箱）来构建一个项目。

项目管理（**Managing Projects**）

Qt Creator在项目中管理你的源代码。你可以使用File->New File或者Project来创建一个新项目。当你创建一个项目时，你可以选择多种应用程序模板。Qt Creator 能够创建桌面，手机应用程序。这些应用程序使用窗口部件（Widgets）或者QtQuick或者控制台，甚至可以是更加简单的项目。当然也支持HTML5与python的项目。对于一个新手是很难选择的，所以我们为你选择了三种类型的项目。

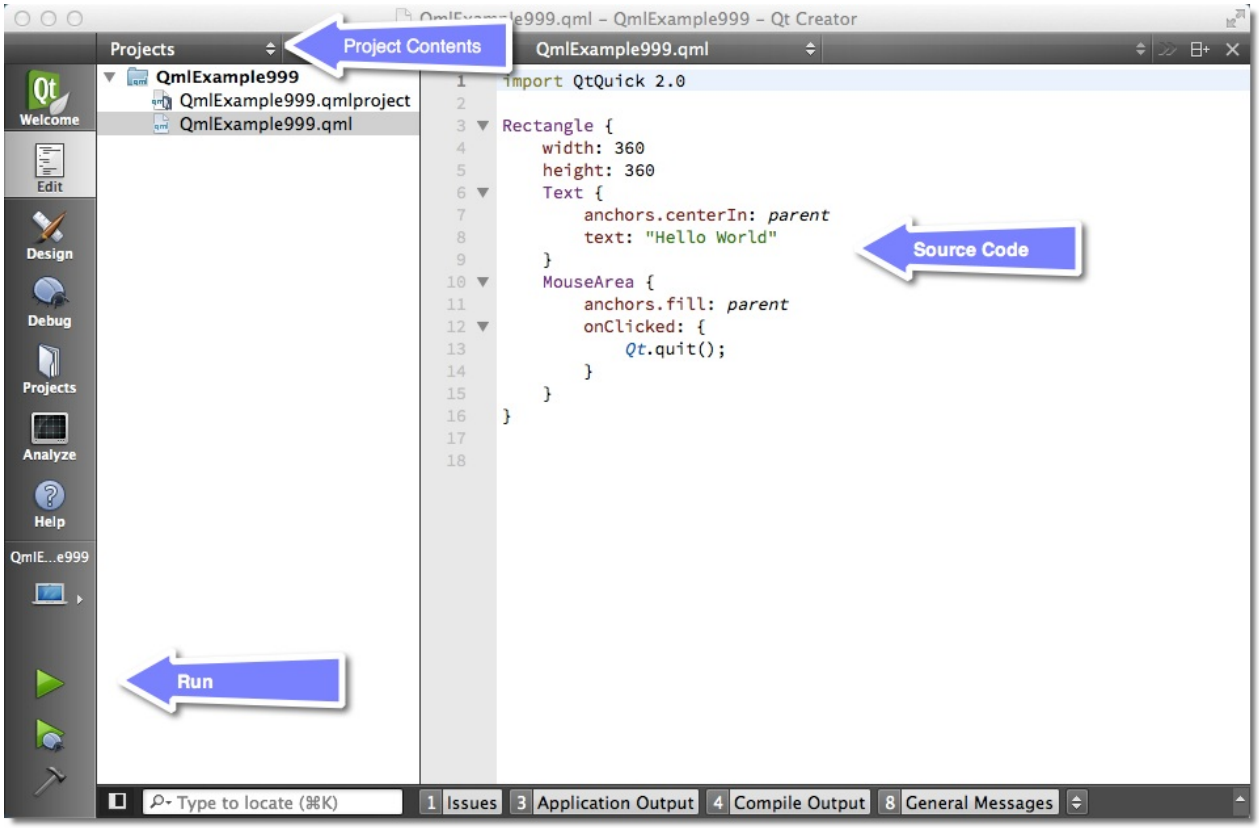
- 应用程序/QtQuick2.0用户界面：这将会为你创建一个QML/JS的项目，不需要使用任何的C++代码。使用这个你可以迅速的创建一个新的用户界面或者计划创建一个基于本地插件的现代的用户界面应用程序。
- 库/Qt Quick2.0扩展插件：使用这个安装引导能够创建一个你自己的Qt Quick用户界面插件。这个插件被用来扩展Qt Quick的本地元素。
- 其它项目/空的Qt项目：只是一个项目的骨架。如果你想从头使用C++来编写你的应用程序，你可以使用这种方式。你需要知道你在这里能做什么。

注意

在这本书的前面部分我们主要使用**QtQuick 2.0**用户界面项目。在后面我们会使用空的**Qt**项目或者类似的项目描述一些**C++**方面的使用。为了使用我们自己的本地插件来扩展**QtQuick**，我们将会使用**Qt Quick2.0**扩展插件安装引导项目。

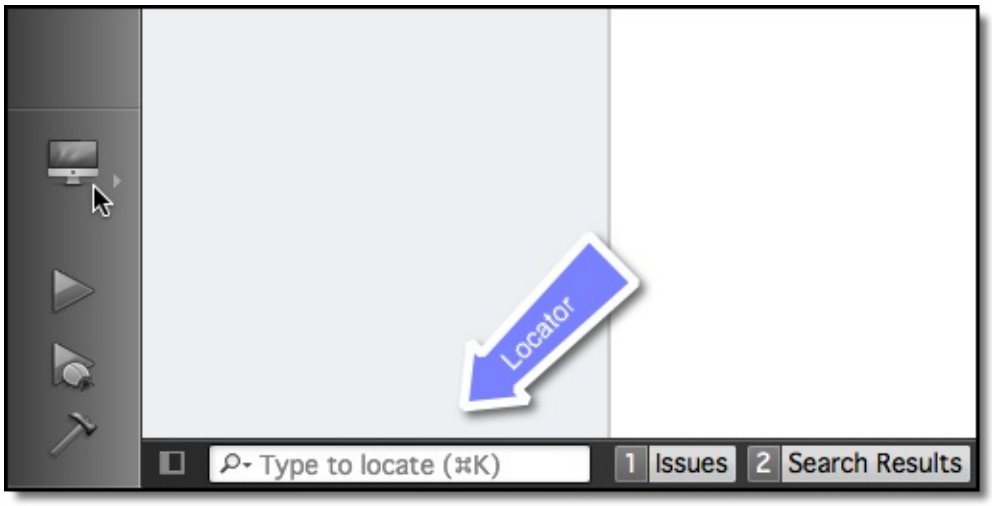
使用编辑器（Using the Editor）

当你打开一个项目或者创建一个新的项目后，Qt Creator将会转换到编辑模式下。你应该可以在左边看到你的项目文件，在中央区域看到代码编辑器。左边选中的文件将会被编辑器打开。编辑器提供了语法高亮，代码补全和智能纠错的功能。也提供几种代码重构的命令。当你使用这个编辑器工作时你会觉得它的响应非常的迅速。这感谢与Qt Creaotor的开发者将这个工具做的如此杰出。

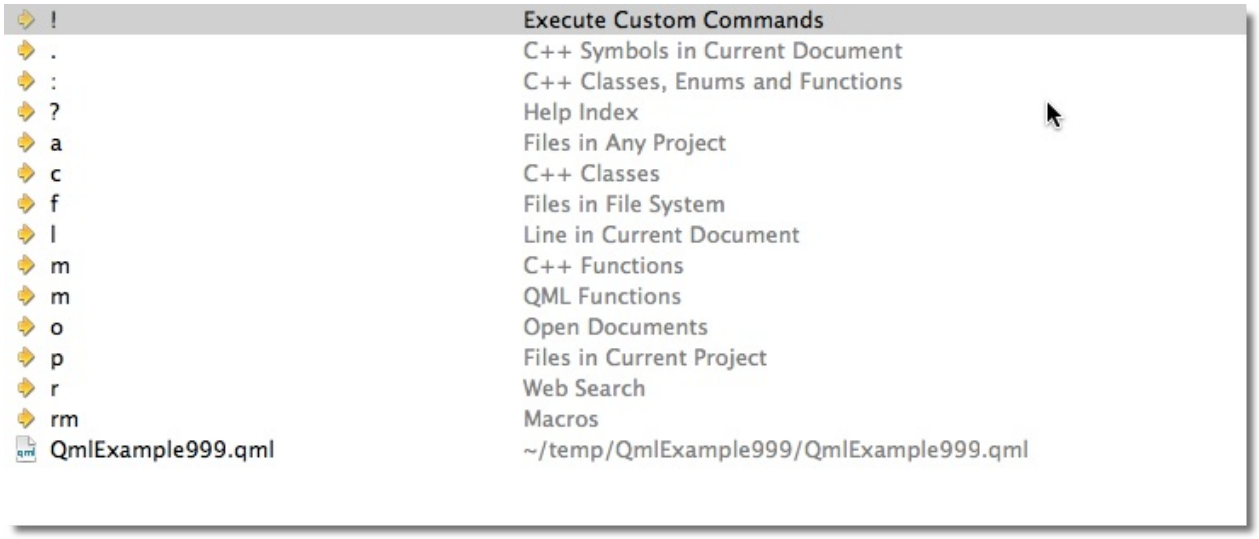


定位器（Locator）

定位器是Qt Creaotor中心的一个组件。它可以让开发者迅速的找到指定代码的位置，或者获得帮助。使用Ctrl+K来打开定位器。



左边底部可以显示弹出一系列的选项。如果你只是想搜索你项目中的一个文件，你只需要给出文件第一个字母提示就可以了。定位器也接收通配符，比如*main.qml也可以查找。你也可以通过前缀搜索来搜索指定内容的类型。



试试它，例如寻找一个QML矩形框的帮助，输入?rectangle。定位器会不停的更新它的建议直到你找到你想要的参考文档。

调试（Debugging）

Qt Creator支持C++与QML代码调试。

注意

嗯，我才意识到我还没有使用过调试。这是一个好的现象。我需要有人对此提出问题，查看**Qt Creator documentation**来获得更多的帮助吧。

快捷键（Shortcuts）

在好使用的系统中和专业系统中，快捷键是不同的。作为专业的开发人员，你也许会在你的应用程序上花很多时间，每一个快捷键都能使你的工作效率得到提高。Qt Creator的开发者也这样想，并且在应用程序中加入了许许多多的快捷键。

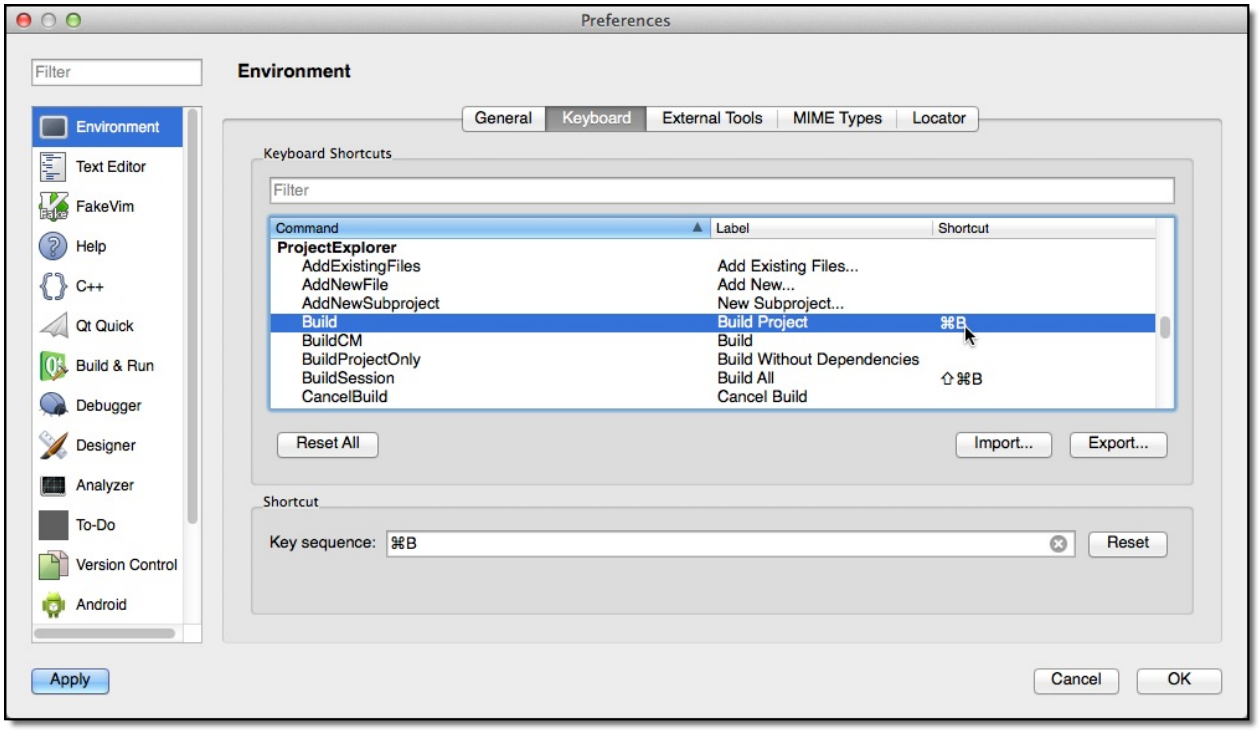
我们列出了一些基本的快捷键操作：

- Ctrl+B - 构建项目
- Ctrl+R - 运行项目
- Ctrl+Tab - 切换已打开的文档
- Ctrl+k - 打开定位器
- Esc - 返回
- F2 - 查找对应的符号解释。
- F4 - 在头文件与源文件之间切换（只对c++代码有效）

这些快捷键的定义来自Qt Creator shortcuts这个文档。

注意

你可以使用设置窗口来编辑你的快捷键。



Quick Starter

Quick Starter

注意

最后一次构建：**2014**年**1**月**20**日下午**18:00**。

这章的源代码能够在**assetts folder**找到。

这章概述了QML语言，Qt5中大量使用了这种声明用户界面的语言。我们将会讨论QML语言，一个树形结构的元素，跟着是一些最基本的元素概述。然后我们会简短的介绍怎样创建我们自己的元素，这些元素被叫做组件，并如何使用属性操作来转换元素。最后我们会介绍如何对元素进行布局，如何向用户提供输入。

QML语法（QML Syntax）

QML是一种描述用户界面的声明式语言。它将用户界面分解成一些更小的元素，这些元素能够结合成一个组件。QML语言描述了用户界面元素的形状和行为。用户界面能够使用JavaScript来提供修饰，或者增加更加复杂的逻辑。从这个角度来看它遵循HTML-JavaScript模式，但QML是被设计用来描述用户界面的，而不是文本文档。

从QML元素的层次结构来理解是最简单的学习方式。子元素从父元素上继承了坐标系统，它的x,y坐标总是相对应于它的父元素坐标系统。

让我们开始用一个简单的QML文件例子来解释这个语法。

```
// rectangle.qml

import QtQuick 2.0

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

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

    // color property
    color: "#D8D8D8"

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

```
        id: rocket

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

        source: 'assets/rocket.png'
    }

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

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

        // reference root element
        width: root.width

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

- import声明导入了一个指定的模块版本。一般来说会导入QtQuick2.0来作为初始元素的引用。
- 使用//可以单行注释，使用/**/可以多行注释，就像C/C++和JavaScript一样。
- 每一个QML文件都需要一个根元素，就像HTML一样。
- 一个元素使用它的类型声明，然后使用{}进行包含。
- 元素拥有属性，他们按照name:value的格式来赋值。
- 任何在QML文档中的元素都可以使用它们的id进行访问（id是一个任意的标识符）。
- 元素可以嵌套，这意味着一个父元素可以拥有多个子元素。子元素可以通过访问parent关键字来访问它们的父元素。

建议

你会经常使用**id**或者关键字**parent**来访问你的父对象。有一个比较好的方法是命名你的根元素对象**id**为**root**（**id:root**），这样就不用去思考你的**QML**文档中的根元素应该用什么方式命名了。

提示

你可以在你的操作系统命令行模式下使用**QtQuick**运行环境来运行这个例子，比如像下面这样：

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

将$QTDIR替换为你的Qt的安装路径。qmlscene会执行Qt Quick运行环境初始化，并且解释这个QML文件。

在Qt Creator中你可以打开对应的项目文件然后运行rectangle.qml文档。

4.1.1 属性（Properties）

元素使用他们的元素类型名进行声明，使用它们的属性或者创建自定义属性来定义。一个属性对应一个值，例如 width:100，text: 'Greeting', color: '#FF0000'。一个属性有一个类型定义并且需要一个初始值。

```
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 neeed to receive key events
    focus: true

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

让我们来看看不同属性的特点：

1. id是一个非常特殊的属性值，它在一个QML文件中被用来引用元素。id不是一个字符串，而是一个标识符和QML语法的一部分。一个id在一个QML文档中是唯一的，并且不能被设置为其它值，也无法被查询（它的行为更像C++世界里的指针）。

2. 一个属性能够设置一个值，这个值依赖于它的类型。如果没有对一个属性赋值，那么它将会被初始化为一个默认值。你可以查看特定的元素的文档来获得这些初始值的信息。

3. 一个属性能够依赖一个或多个其它的属性，这种操作称作属性绑定。当它依赖的属性改变时，它的值也会更新。这就像订了一个协议，在这个例子中height始终是width的两倍。

4. 添加自己定义的属性需要使用property修饰符，然后跟上类型，名字和可选择的初始化值（property : ）。如果没有初始值将会给定一个系统初始值作为初始值。注意如果属性名与已定义的默认属性名不重复，使用**default**关键字你可以将一个属性定义为默认属性。这在你添加子元素时用得着，如果他们是可视化的元素，子元素会自动的添加默认属性的子类型链表（**children property list**）。

5. 另一个重要的声明属性的方法是使用alias关键字（property alias : ）。alias关键字允许我们转发一个属性或者转发一个属性对象自身到另一个作用域。我们将在后面定义组件导出内部属性或者引用根级元素id会使用到这个技术。一个属性别名不需要类型，它使用引用的属性类型或者对象类型。

6. text属性依赖于自定义的timers（int整型数据类型）属性。int整型数据会自动的转换为string字符串类型数据。这样的表达方式本身也是另一种属性绑定的例子，文本结果会在times属性每次改变时刷新。

7. 一些属性是按组分配的属性。当一个属性需要结构化并且相关的属性需要联系在一起时，我们可以这样使用它。另一个组属性的编码方式是 font{family: "UBuntu"; pixelSize: 24 }。

8. 一些属性是元素自身的附加属性。这样做是为了全局的相关元素在应用程序中只出现一次（例如键盘输入）。编码方式.: 。

9. 对于每个元素你都可以提供一个信号操作。这个操作在属性值改变时被调用。例如这里我们完成了当height（高度）改变时会使用控制台输出一个信息。

警告

一个元素**id**应该只在当前文档中被引用。**QML**提供了动态作用域的机制，后加载的文档会覆盖之前加载文档的元素**id**号，这样就可以引用已加载并且没有被覆盖的元素**id**，这有点类似创建全局变量。但不幸的是这样的代码阅读性很差。目前这个还没有办法解决这个问题，所以你使用这个机制的时候最好仔细一些甚至不要使用这种机制。如果你想向文档外提供元素的调用，你可以在根元素上使用属性导出的方式来提供。

4.1.2 脚本（Scripting）

QML与JavaScript是最好的配合。在JavaScrpit的章节中我们将会更加详细的介绍这种关系，现在我们只需要了解这种关系就可以了。

```
    Text {
        id: label

        x: 24; y: 24

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

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

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

        // need focus to receive key events
        focus: true

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

```
            increment()
        }

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

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

1. 文本改变操作onTextChanged会将每次空格键按下导致的文本改变输出到控制台。
2. 当文本元素接收到空格键操作（用户在键盘上点击空格键），会调用JavaScript函数increment()。
3. 定义一个JavaScript函数使用这种格式function (){....}，在这个例子中是增加spacePressed的计数。每次spacePressed的增加都会导致它绑定的属性更新。

注意

QML的：（属性绑定）与**JavaScript**的**=**（赋值）是不同的。绑定是一个协议，并且存在于整个生命周期。然而**JavaScript**赋值（**=**）只会产生一次效果。当一个新的绑定生效或者使用**JavaScript**赋值给属性时，绑定的生命周期就会结束。例如一个按键的操作设置文本属性为一个空的字符串将会销**毁**我们的增值显示：

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

在点击取消（**ESC**）后，再次点击空格键（**space-bar**）将不会更新我们的显示，之前的**text**属性绑定（**text: "Space pressed:" + spacePresses + "times")**被销**毁**。

当你对改变属性的策略有冲突时（文本的改变基于一个增值的绑定并且可以被**JavaScript**赋值清零），类似于这个例子，你最好不要使用绑定属性。你需要使用赋值的方式来改变属性，属性绑定会在赋值操作后被销**毁**（销**毁**协议！）。

基本元素（Basic Elements）

元素可以被分为可视化元素与非可视化元素。一个可视化元素（例如矩形框Rectangle）有着几何形状并且可以在屏幕上显示。一个非可视化元素（例如计时器Timer）提供了常用的功能，通常用于操作可视化元素。

现在我们将专注于几个基础的可视化元素，例如Item（基础元素对象），Rectangle（矩形框），Text（文本），Image（图像）和MouseArea（鼠标区域）。

4.2.1 基础元素对象（Item Element）

Item（基础元素对象）是所有可视化元素的基础对象，所有其它的可视化元素都继承自Item。它自身不会有任何绘制操作，但是定义了所有可视化元素共有的属性：

Group（分组）	**Properties**（属性）
Geometry（几何属性）	x,y（坐标）定义了元素左上角的位置，width，height（长和宽）定义元素的显示范围，z（堆叠次序）定义元素之间的重叠顺序。
Layout handling（布局操作）	anchors（锚定），包括左（left），右（right），上（top），下（bottom），水平与垂直居中（vertical center，horizontal center），与margins（间距）一起定义了元素与其它元素之间的位置关系。
Key handlikng（按键操作）	附加属性key（按键）和keyNavigation（按键定位）属性来控制按键操作，处理输入焦点（focus）可用操作。
Transformation（转换）	缩放（scale）和rotate（旋转）转换，通用的x,y,z属性列表转换（transform），旋转基点设置（transformOrigin）。
Visual（可视化）	不透明度（opacity）控制透明度，visible（是否可见）控制元素是否显示，clip（裁剪）用来限制元素边界的绘制，smooth（平滑）用来提高渲染质量。
State definition（状态定义）	states（状态列表属性）提供了元素当前所支持的状态列表，当前属性的改变也可以使用transitions（转变）属性列表来定义状态转变动画。

为了更好的理解不同的属性，我们将会在这章中尽量的介绍这些元素的显示效果。请记住这些基本的属性在所有可视化元素中都是可以使用的，并且在这些元素中的工作方式都是相同的。

注意

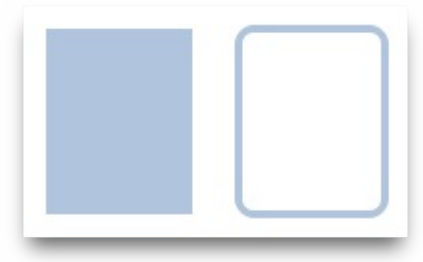
Item（基本元素对象）通常被用来作为其它元素的容器使用，类似**HTML**语言中的**div**元素（**div element**）。

4.2.2 矩形框元素（Rectangle Element）

Rectangle（矩形框）是基本元素对象的一个扩展，增加了一个颜色来填充它。它还支持边界的定义，使用border.color（边界颜色），border.width（边界宽度）来自定义边界。你可以使用radius（半径）属性来创建一个圆角矩形。

```
    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
    }
```



注意

颜色的命名是来自**SVG**颜色的名称（查看**http://www.w3.org/TR/css3-color/#svg-color**可以获取更多的颜色名称）。你也可以使用其它的方法来指定颜色，比如**RGB**字符串（**'#FF4444'**），或者一个颜色名字（例如**'white'**）。

此外，填充的颜色与矩形的边框也支持自定义的渐变色。

```
    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"
    }
```

一个渐变色是由一系列的梯度值定义的。每一个值定义了一个位置与颜色。位置标记了y轴上的位置（0 = 顶，1 = 底）。GradientStop（倾斜点）的颜色标记了颜色的位置。

注意

一个矩形框如果没有**width/height**（宽度与高度）将不可见。如果你有几个相互关联**width/height**（宽度与高度）的矩形框，在你组合逻辑中出了错后可能就会发生矩形框不可见，请注意这一点。

注意

这个函数无法创建一个梯形，最好使用一个已有的图像来创建梯形。有一种可能是在旋转梯形时，旋转的矩形几何结构不会发生改变，但是这会导致几何元素相同的可见区域的混淆。从作者的观点来看类似的情况下最好使用设计好的梯形图形来完成绘制。

4.2.3 文本元素（Text Element）

显示文本你需要使用Text元素（Text Element）。它最值得注意的属性时字符串类型的text属性。这个元素会使用给出的text（文本）与font（字体）来计算初始化的宽度与高度。可以使用字体属性组来（font property group）来改变当前的字体，例如font.family，font.pixelSize，等等。改变文本的颜色值只需要改变颜色属性就可以了。

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

文本可以使用horizontalAlignment与verticalAlignment属性来设置它的对齐效果。为了提高文本的渲染效果，你可以使用style和styleColor属性来配置文字的外框效果，浮雕效果或者凹陷效果。对于过长的文本，你可能需要使用省略号来表示，例如A very ... long text，你可以使用elide属性来完成这个操作。elide属性允许你设置文本左边，右边或者中间的省略位置。如果你不想'....'省略号出现，并且希望使用文字换行的方式显示所有的文本，你可以使用wrapMode属性（这个属性只在明确设置了宽度后才生效）：

```
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
}
```

一个text元素（text element）只显示的文本，它不会渲染任何背景修饰。除了显示的文本，text元素背景是透明的。为一个文本元素提供背景是你自己需要考虑的问题。

注意

知道一个文本元素（**Text Element**）的初始宽度与高度是依赖于文本字符串和设置的字体这一点很重要。一个没有设置宽度或者文本的文本元素（**Text Element**）将不可见，默认的初始宽度是**0**。

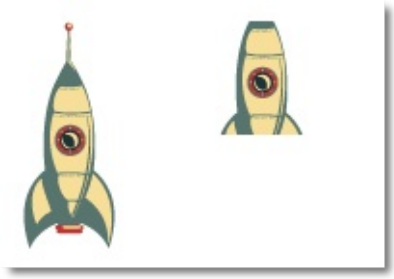
注意

通常你想要对文本元素布局时，你需要区分文本在文本元素内部的边界对齐和由元素边界自动对齐。前一种情况你需要使用**horizontalAlignment**和**verticalAlignment**属性来完成，后一种情况你需要操作元素的几何形状或者使用**anchors**（锚定）来完成。

4.2.4 图像元素（Image Element）

一个图像元素（Image Element）能够显示不同格式的图像（例如PNG,JPG,GIF,BMP）。想要知道更加详细的图像格式支持信息，可以查看Qt的相关文档。source属性（source property）提供了图像文件的链接信息，fillMode（文件模式）属性能够控制元素对象的大小调整行为。

```
    Image {
        x: 12; y: 12
        // width: 48
        // height: 118
        source: "assets/rocket.png"
    }
    Image {
        x: 112; y: 12
        width: 48
        height: 118/2
        source: "assets/rocket.png"
        fillMode: Image.PreserveAspectCrop
        clip: true
    }
```



注意

一个**URL**可以是使用**'/'**语法的本地路径（**"./images/home.png"**）或者一个网络链接（**"http://example.org/home.png"**）。

注意

图像元素（**Image element**）使用**PreserveAspectCrop**可以避免裁剪图像数据被渲染到图像边界外。默认情况下裁剪是被禁用的（**clip:false**）。你需要打开裁剪（**clip:true**）来约束边界矩形的绘制。这对任何可视化元素都是有效的。

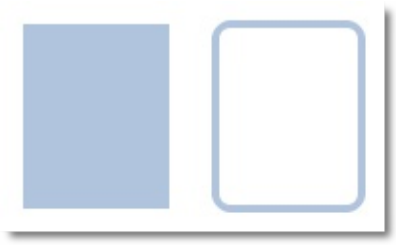
建议

使用**QQmlImageProvider**你可以通过**C++**代码来创建自己的图像提供器，这允许你动态创建图像并且使用线程加载。

4.2.5 鼠标区域元素（MouseArea Element）

为了与不同的元素交互，你通常需要使用MouseArea（鼠标区域）元素。这是一个矩形的非可视化元素对象，你可以通过它来捕捉鼠标事件。当用户与可视化端口交互时，mouseArea通常被用来与可视化元素对象一起执行命令。

```
    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
    }
```



注意

这是**QtQuick**中非常重要的概念，输入处理与可视化显示分开。这样你的交互区域可以比你显示的区域大很多。

组件（Compontents）

一个组件是一个可以重复使用的元素，QML提供几种不同的方法来创建组件。但是目前我们只对其中一种方法进行讲解：一个文件就是一个基础组件。一个以文件为基础的组件在文件中创建了一个QML元素，并且将文件以元素类型来命名（例如Button.qml）。你可以像任何其它的QtQuick模块中使用元素一样来使用这个组件。在我们下面的例子中，你将会使用你的代码作为一个Button（按钮）来使用。

让我们来看看这个例子，我们创建了一个包含文本和鼠标区域的矩形框。它类似于一个简单的按钮，我们的目标就是让它足够简单。

```
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
}
```

用户界面将会看起来像下面这样。左边是初始化的状态，右边是按钮点击后的效果。

我们的目标是提取这个按钮作为一个可重复使用的组件。我们可以简单的考虑一下我们的按钮会有的哪些API（应用程序接口），你可以自己考虑一下你的按钮应该有些什么。下面是我考虑的结果：

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

我想要使用text属性来设置文本，然后实现我们自己的点击操作。我也期望这个按钮有一个比较合适的初始化大小（例如width:240）。 为了完成我们的目标，我创建了一个Button.qml文件，并且将我们的代码拷贝了进去。我们在根级添加一个属性导出方便使用者修改它。

我们在根级导出了文本和点击信号。通常我们命名根元素为root让引用更加方便。我们使用了QML的alias（别名）的功能，它可以将内部嵌套的QML元素的属性导出到外面使用。有一点很重要，只有根级目录的属性才能够被其它文件的组件访问。

```
// Button.qml

import QtQuick 2.0

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

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

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

使用我们新的Button元素只需要在我们的文件中简单的声明一下就可以了，之前的例子将会被简化。

```
    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
    }
```

现在你可以在你的用户界面代码中随意的使用Button{ ...}来作为按钮了。一个真正的按钮将更加复杂，比如提供按键反馈或者添加一些装饰。

注意

就个人而言，可以更进一步的使用基础元素对象（**Item**）作为根元素。这样可以防止用户改变我们设计的按钮的颜色，并且可以提供出更多相关控制的**API**（应用程序接口）。我们的目标是导出一个最小的**API**（应用程序接口）。实际上我们可以将根矩形框（**Rectangle**）替换为一个基础元素（**Item**），然后将一个矩形框（**Rectangle**）嵌套在这个根元素（**root item**）就可以完成了。

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

使用这项技术可以很简单的创建一系列可重用的组件。

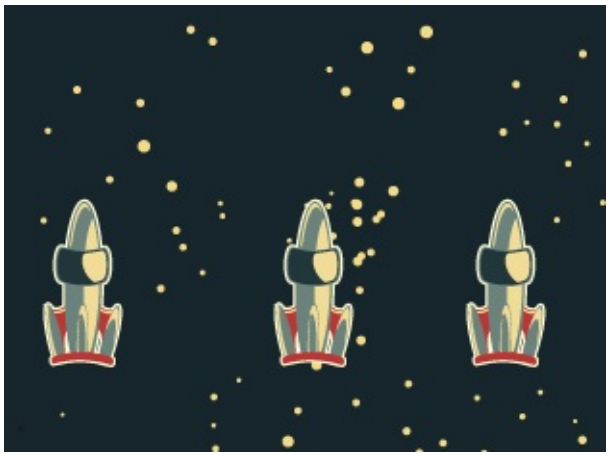
简单的转换（Simple Transformations）

转换操作改变了一个对象的几何状态。QML元素对象通常能够被平移，旋转，缩放。下面我们将讲解这些简单的操作和一些更高级的用法。 我们先从一个简单的转换开始。用下面的场景作为我们学习的开始。

简单的位移是通过改变x,y坐标来完成的。旋转是改变rotation（旋转）属性来完成的，这个值使用角度作为单位（0~360）。缩放是通过改变scale（比例）的属性来完成的，小于1意味着缩小，大于1意味着放大。旋转与缩放不会改变对象的几何形状，对象的x,y（坐标）与width/height（宽/高）也类似。只有绘制指令是被转换的对象。

在我们展示例子之前我想要介绍一些东西：ClickableImage元素（ClickableImage element），ClickableImage仅仅是一个包含鼠标区域的图像元素。我们遵循一个简单的原则，三次使用相同的代码描述一个用户界面最好可以抽象为一个组件。

```
// ClickableImage.qml

// Simple image which can be clicked

import QtQuick 2.0

Image {
    id: root
    signal clicked

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



我们使用我们可点击图片元素来显示了三个火箭。当点击时，每个火箭执行一种简单的转换。点击背景将会重置场景。

```
// transformation.qml
```

```
import QtQuick 2.0

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

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

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

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

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

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

```
}
```

火箭1在每次点击后X轴坐标增加5像素，火箭2每次点击后会旋转。火箭3每次点击后会缩小。对于缩放和旋转操作我们都设置了smooth:true来增加反锯齿，由于性能的原因通常是被关闭的（与剪裁属性clip类似）。当你看到你的图形中出现锯齿时，你可能就需要打开平滑（smooth）。

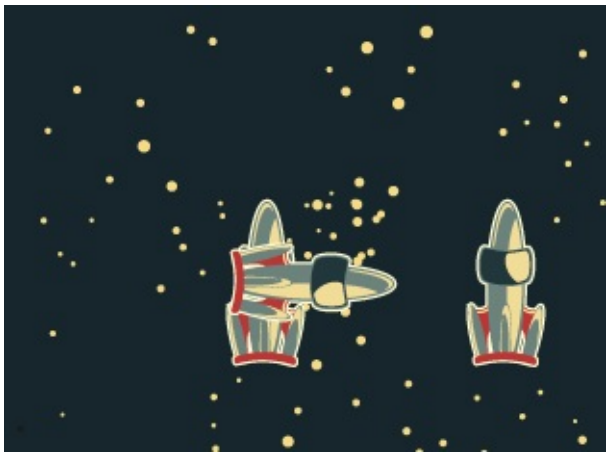
注意

为了获得更好的显示效果，当缩放图片时推荐使用已缩放的图片来替代，过量的放大可能会导致图片模糊不清。当你在缩放图片时你最好考虑使用**smooth:true**来提高图片显示质量。

使用MouseArea来覆盖整个背景，点击背景可以初始化火箭的值。

注意

在代码中先出现的元素有更低的堆叠顺序（叫做**z**顺序值**z-order**），如果你点击火箭**1**足够多次，你会看见火箭**1**移动到了火箭**2**下面。**z**轴顺序也可以使用元素对象的**z-property**来控制。



由于火箭**2**后出现在代码中，火箭**2**将会放在火箭**1**上面。这同样适用于**MouseArea**（鼠标区域），一个后出现在代码中的鼠标区域将会与之前的鼠标区域重叠，后出现的鼠标区域才能捕捉到鼠标事件。

请记住：文档中元素的顺序很重要。

定位元素（Positioning Element）

有一些QML元素被用于放置元素对象，它们被称作定位器，QtQuick模块提供了Row，Column，Grid，Flow用来作为定位器。你可以在下面的插图中看到它们使用相同内容的显示效果。

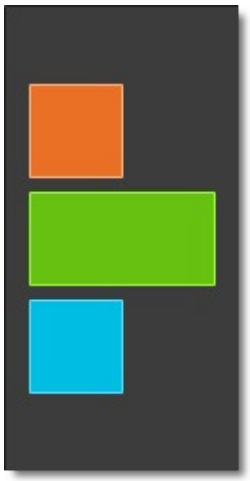
注意

在我们详细介绍前，我们先介绍一些相关的元素，红色（**red**），蓝色（**blue**），绿色（**green**），高亮（**lighter**）与黑暗（**darker**）方块，每一个组件都包含了一个**48**乘**48**的着色区域。下面是关于**RedSquare**（红色方块）的代码：

```
// RedSquare.qml

import QtQuick 2.0

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

请注意使用了**Qt.lighter**（**color**）来指定了基于填充色的边界高亮色。我们将会在后面的例子中使用到这些元素，希望后面的代码能够容易读懂。请记住每一个矩形框的初始化大小都是**48**乘**48**像素大小。

Column（列）元素将它的子对象通过顶部对齐的列方式进行排列。spacing属性用来设置每个元素之间的间隔大小。



```
// column.qml

import QtQuick 2.0

DarkSquare {
    id: root
    width: 120
    height: 240
```

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

// M1<<
```

Row（行）元素将它的子对象从左到右，或者从右到左依次排列，排列方式取决于layoutDirection属性。spacing属性用来设置每个元素之间的间隔大小。

```
// row.qml

import QtQuick 2.0

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

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

Grid（栅格）元素通过设置rows（行数）和columns（列数）将子对象排列在一个栅格中。可以只限制行数或者列数。如果没有设置它们中的任意一个，栅格元素会自动计算子项目总数来获得配置，例如，设置rows（行数）为3，添加了6个子项目到元素中，那么会自动计算columns（列数）为2。属性flow（流）与layoutDirection（布局方向）用来控制子元素的加入顺序。spacing属性用来控制所有元素之间的间隔。

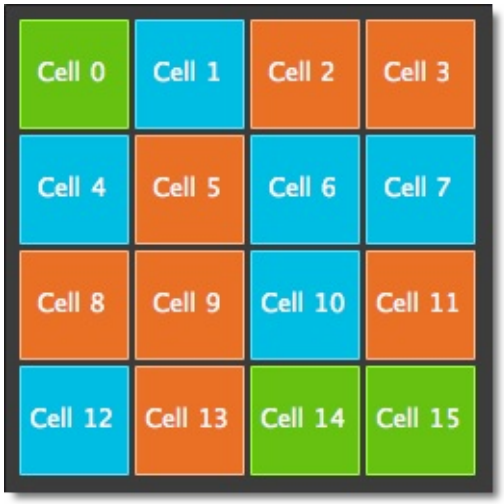
```
// grid.qml

import QtQuick 2.0

BrightSquare {
    id: root
    width: 160
    height: 160

    Grid {
        id: grid
        rows: 2
        columns: 2
        anchors.centerIn: parent
        spacing: 8
        RedSquare { }
        RedSquare { }
        RedSquare { }
        RedSquare { }
    }

}
```

最后一个定位器是Flow（流）。通过flow（流）属性和layoutDirection（布局方向）属性来控制流的方向。它能够从头到底的横向布局，也可以从左到右或者从右到左进行布局。作为加入流中的子对象，它们在需要时可以被包装成新的行或者列。为了让一个流可以工作，必须指定一个宽度或者高度，可以通过属性直接设定，或者通过anchor（锚定）布局设置。

```
// flow.qml

import QtQuick 2.0

BrightSquare {
```

```
    id: root
    width: 160
    height: 160

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

通常Repeater（重复元素）与定位器一起使用。它的工作方式就像for循环与迭代器的模式一样。在这个最简单的例子中，仅仅提供了一个循环的例子。




```
// repeater.qml

import QtQuick 2.0

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


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

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

在这个重复元素的例子中，我们使用了一些新的方法。我们使用一个颜色数组定义了一组颜色属性。重复元素能够创建一连串的矩形框（16个，就像模型中定义的那样）。每一次的循环都会创建一个矩形框作为repeater的子对象。在矩形框中，我们使用了JS数学函数Math.floor(Math.random()*3)来选择颜色。这个函数会给我们生成一个0~2的随机数，我们使用这个数在我们的颜色数组中选择颜色。注意之前我们说过JavaScript是QtQuick中的一部分，所以这些典型的库函数我们都可以使用。

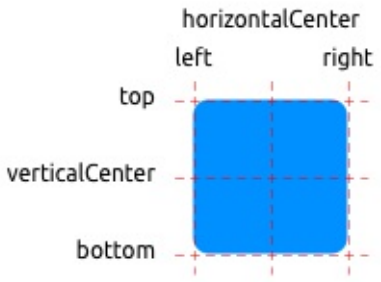
一个重复元素循环时有一个index（索引）属性值。当前的循环索引（0,1,2,....15）。我们可以使用这个索引值来做一些操作，例如在我们这个例子中使用Text（文本）显示当前索引值。

注意

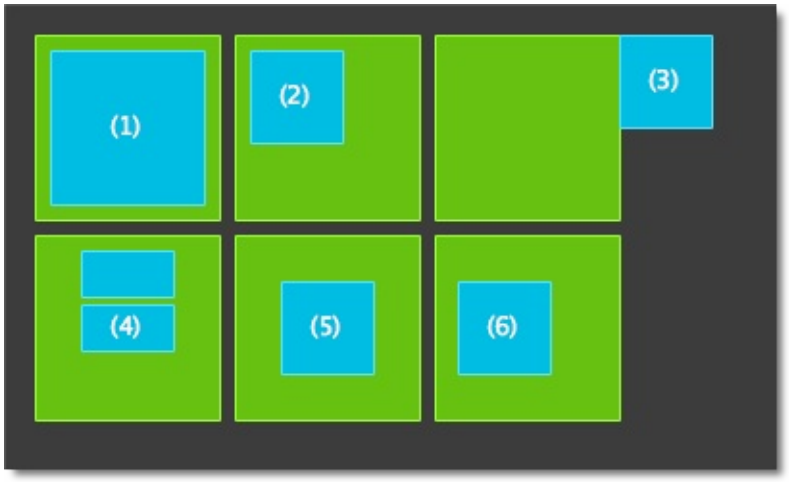
高级的大数据模型处理和使用动态代理的动态视图会在模型与视图（**model-view**）章节中讲解。当有一小部分的静态数据需要显示时，使用重复元素是最好的方式。

布局元素（Layout Items）

QML使用anchors（锚）对元素进行布局。anchoring（锚定）是基础元素对象的基本属性，可以被所有的可视化QML元素使用。一个anchors（锚）就像一个协议，并且比几何变化更加强大。Anchors（锚）是相对关系的表达式，你通常需要与其它元素搭配使用。



一个元素有6条锚定线（top顶，bottom底，left左，right右，horizontalCenter水平中，verticalCenter垂直中）。在文本元素（Text Element）中有一条文本的锚定基线（baseline）。每一条锚定线都有一个偏移（offset）值，在top（顶），bottom（底），left（左），right（右）的锚定线中它们也被称作边距。对于horizontalCenter（水平中）与verticalCenter（垂直中）与baseline（文本基线）中被称作偏移值。



1. 元素填充它的父元素。

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

2. 元素左对齐它的父元素。

```
GreenSquare {
```

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

3. 元素的左边与它父元素的右边对齐。

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

4. 元素中间对齐。Blue1与它的父元素水平中间对齐。Blue2与Blue1中间对齐，并且它的顶部对齐Blue1的底部。

```
    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. 元素在它的父元素中居中。

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

6. 元素水平方向居中对齐父元素并向后偏移12像素，垂直方向居中对齐。

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

注意

我们的方格都打开了拖拽。试着拖放几个方格。你可以发现第一个方格无法被拖拽因为它每个边都被固定了，当然第一个方格的父元素能够被拖拽是因为它的父元素没有被固定。第二个方格能够在垂直方向上拖拽是因为它只有左边被固定了。类似的第三个和第四个方格也只能在垂直方向上拖拽是因为它们都使用水平居中对齐。第五个方格使用居中布局，它也无法被移动，第六个方格与第五个方格类似。拖拽一个元素意味着会改变它的**x,y**坐标。**anchoring**（锚定）比几何变化（例如**x,y**坐标变化）更强大是因为锚定线（**anchored lines**）的限制，我们将在后面讨论动画时看到这些功能的强大。

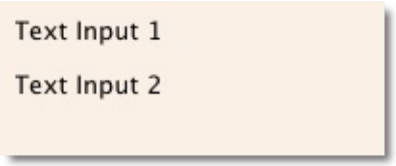
输入元素（Input Element）

我们已经使用过MouseArea（鼠标区域）作为鼠标输入元素。这里我们将更多的介绍关于键盘输入的一些东西。我们开始介绍文本编辑的元素：TextInput（文本输入）和TextEdit（文本编辑）。

4.7.1 文本输入（TextInput）

文本输入允许用户输入一行文本。这个元素支持使用正则表达式验证器来限制输入和输入掩码的模式设置。

```
// textinput.qml

import QtQuick 2.0

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

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

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



用户可以通过点击TextInput来改变焦点。为了支持键盘改变焦点，我们可以使用KeyNavigation（按键向导）这个附加属性。

```
// textinput2.qml

import QtQuick 2.0

Rectangle {
    width: 200
    height: 80
```

```
    color: "linen"

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

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

KeyNavigation（按键向导）附加属性可以预先设置一个元素id绑定切换焦点的按键。

一个文本输入元素（text input element）只显示一个闪烁符和已经输入的文本。用户需要一些可见的修饰来鉴别这是一个输入元素，例如一个简单的矩形框。当你放置一个TextInput（文本输入）在一个元素中时，你需要确保其它的元素能够访问它导出的大多数属性。

我们提取这一段代码作为我们自己的组件，称作TLineEditV1用来重复使用。

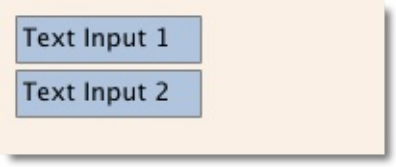
```
// TLineEditV1.qml

import QtQuick 2.0

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

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

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

注意

如果你想要完整的导出**TextInput**元素，你可以使用**property alias input: input**来导出这个元素。第一个**input**是属性名字，第二个**input**是元素**id**。

我们使用TLineEditV1组件重写了我们的KeyNavigation（按键向导）的例子。

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



尝试使用Tab按键来导航，你会发现焦点无法切换到input2上。这个例子中使用focus:true的方法不正确，这个问题是因为焦点被转移到input2元素时，包含TlineEditV1的顶部元素接收了这个焦点并且没有将焦点转发给TextInput（文本输入）。为了防止这个问题，QML提供了FocusScope（焦点区域）。

4.7.2 焦点区域（FocusScope）

一个焦点区域（focus scope）定义了如果焦点区域接收到焦点，它的最后一个使用focus:true的子元素接收焦点，它将会把焦点传递给最后申请焦点的子元素。我们创建了第二个版本的TLineEdit组件，称作TLineEditV2，使用焦点区域（focus scope）作为根元素。

```
// TLineEditV2.qml

import QtQuick 2.0

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

    }

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

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

现在我们的例子将像下面这样：

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

按下Tab按键可以成功的在两个组件之间切换焦点，并且能够正确的将焦点锁定在组件内部的子元素中。

4.7.3 文本编辑（TextEdit）

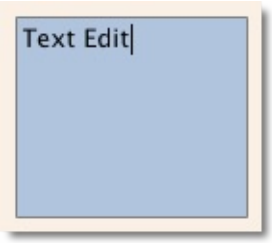
文本编辑（TextEdit）元素与文本输入（TextInput）非常类似，它支持多行文本编辑。它不再支持文本输入的限制，但是提供了已绘制文本的大小查询（paintedHeight，paintedWidth）。我们也创建了一个我们自己的组件TTextEdit，可以编辑它的背景，使用focus scope（焦点区域）来更好的切换焦点。

```
// TTextEdit.qml

import QtQuick 2.0

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

    }

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

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

你可以像下面这样使用这个组件：

```
// textedit.qml

import QtQuick 2.0
```

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

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



4.7.4 按键元素（Key Element）

附加属性key允许你基于某个按键的点击来执行代码。例如使用up，down按键来移动一个方块，left，right按键来旋转一个元素，plus，minus按键来缩放一个元素。

```
// keys.qml

import QtQuick 2.0

DarkSquare {
    width: 400; height: 200

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

    }
}
```

高级用法（Advanced Techniques）

后续添加。

Fluid Elements

注意

最后一次构建：**2014年1月20日下午18:00**。

这章的源代码能够在**assetts folder**找到。

到目前为止，我们已经介绍了简单的图形元素和怎样布局，怎样操作它们。这一章介绍如何控制属性值的变化，通过动画的方式在一段时间内来改变属性值。这项技术是建立一个现代化的平滑界面的基础，通过使用状态和过渡来扩展你的用户界面。每一种状态定义了属性的改变，与动画联系起来的状态改变称作过渡。

动画（Animations）

动画被用于属性的改变。一个动画定义了属性值改变的曲线，将一个属性值变化从一个值过渡到另一个值。动画是由一连串的目标属性活动定义的，平缓的曲线算法能够引发一个定义时间内属性的持续变化。所有在QtQuick中的动画都由同一个计时器来控制，因此它们始终都保持同步，这也提高了动画的性能和显示效果。

注意

动画控制了属性的改变，也就是值的插入。这是一个基本的概念，**QML**是基于元素，属性与脚本的。每一个元素都提供了许多的属性，每一个属性都在等待使用动画。在这本书中你将会看到这是一个壮阔的场景，你会发现你自己在看一些动画时欣赏它们的美丽并且肯定自己的创造性想法。然后请记住：动画控制了属性的改变，每个元素都有大量的属性供你任意使用。

```
// animation.qml

import QtQuick 2.0

Image {
    source: "assets/background.png"

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

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

上面这个例子在x坐标和旋转属性上应用了一个简单的动画。每一次动画持续4000毫秒并且永久循环。x轴坐标动画展示了火箭的x坐标逐渐移至240，旋转动画展示了当前角度到360度的旋转。两个动画同时运行，并且在加载用户界面完成后开始。

现在你可以通过to属性和duration属性来实现动画效果。或者你可以在opacity或者scale上添加动画作为例子，集成这两个参数，你可以实现火箭逐渐消失在太空中，试试吧!

5.1.1 动画元素（Animation Elements）

有几种类型的动画，每一种都在特定情况下都有最佳的效果，下面列出了一些常用的动画：

- PropertyAnimation（属性动画）- 使用属性值改变播放的动画
- NumberAnimation（数字动画）- 使用数字改变播放的动画
- ColorAnimation（颜色动画）- 使用颜色改变播放的动画
- RotationAnimation（旋转动画）- 使用旋转改变播放的动画

除了上面这些基本和通常使用的动画元素，QtQuick还提供了一切特殊场景下使用的动画：

- PauseAnimation（停止动画）- 运行暂停一个动画
- SequentialAnimation（继续动画）- 允许动画继续播放
- ParallelAnimation（平行动画）- 允许动画平行播放
- AnchorAnimation（锚定动画）- 使用锚定改变播放的动画
- ParentAnimation（父元素动画）- 使用父对象改变播放的动画
- SmotthedAnimation（平滑动画）- 跟踪一个平滑值播放的动画
- SpringAnimation（弹簧动画）- 跟踪一个弹簧变换的值播放的动画
- PathAnimation（路径动画）- 跟踪一个元素对象的路径的动画
- Vector3dAnimation（3D容器动画）- 使用QVector3d值改变播放的动画

我们将在后面学习怎样创建一连串的动画。当使用更加复杂的动画时，我们可能需要在播放一个动画时中改变一个属性或者运行一个脚本。对于这个问题，QtQuick提供了一个动作元素：

- PropertyAction（属性动作）- 在播放动画时改变属性
- ScriptAction（脚本动作）- 在播放动画时运行脚本

在这一章中我们将会使用一些小的例子来讨论大多数类型的动画。

5.1.2 应用动画（Applying Animations）

动画可以通过以下几种方式来应用：

- 属性动画 - 在元素完整加载后自动运行
- 属性动作 - 当属性值改变时自动运行

- 独立运行动画 - 使用start()函数明确指定运行或者running属性被设置为true（比如通过属性绑定）

后面我们会谈论如何在状态变换时播放动画。

扩展可点击图像元素版本**2**（**ClickableImage Version2**）

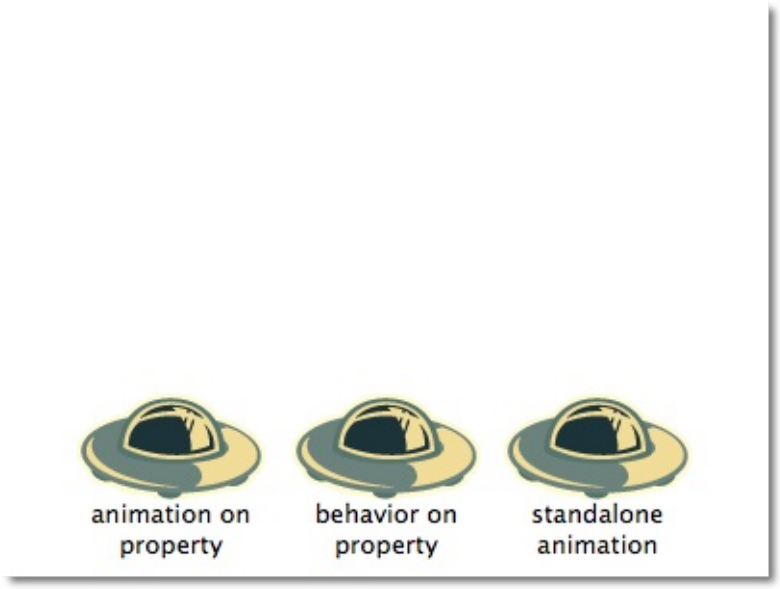
为了演示动画的使用方法，我们重新实现了ClickableImage组件并且使用了一个文本元素（Text Element）来扩展它。

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

import QtQuick 2.0

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

    Column {
        id: container
        Image {
            id: image
        }
        Text {
            id: label
            width: image.width
            horizontalAlignment: Text.AlignHCenter
            wrapMode: Text.WordWrap
            color: "#111111"
        }
    }

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

为了给图片下面的元素定位，我们使用了Column（列）定位器，并且使用基于列的子矩形（childRect）属性来计算它的宽度和高度（width and height）。我们导出了文本（text）和图形源（source）属性，一个点击信号（clicked signal）。我们使用文本元素的wrapMode属性来设置文本与图像一样宽并且可以自动换行。

注意

由于几何依赖关系的反向（父几何对象依赖于子几何对象）我们不能对**ClickableImageV2**设置宽度**/**高度（**width/height**），因为这样将会破坏我们已经做好的属性绑定。这是我们内部设计的限制，作为一个设计组件的人你需要明白这一点。通常我们更喜欢内部几何图像依赖于父几何对象。



三个火箭位于相同的y轴坐标（y = 200）。它们都需要移动到y = 40。每一个火箭都使用了一种的方法来完成这个功能。

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

第一个火箭

第一个火箭使用了Animation on 属性变化的策略来完成。动画会在加载完成后立即播放。点击火箭可以重置它回到开始的位置。在动画播放时重置第一个火箭不会有任何影响。在动画开始前的几分之一秒设置一个新的y轴坐标让人感觉挺不安全的，应当避免这样的属性值竞争的变化。

```
ClickableImageV3 {
    id: rocket2
    x: 152; y: 200
    source: "assets/rocket2.png"
    text: "behavior on property"
    Behavior on y {
        NumberAnimation { duration: 4000 }
    }

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

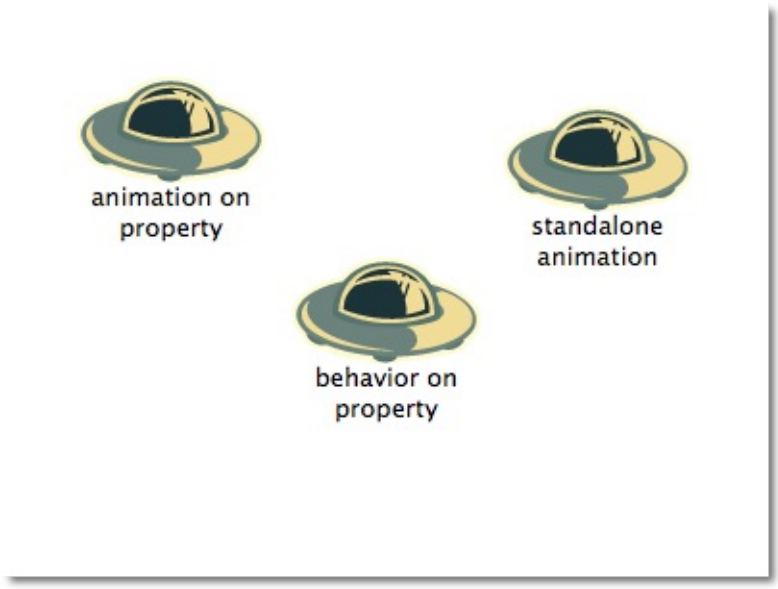
第二个火箭

第二个火箭使用了behavior on 属性行为策略的动画。这个行为告诉属性值每时每刻都在变化，通过动画的方式来改变这个值。可以使用行为元素的enabled : false来设置行为失效。当你点击这个火箭时它将会开始运行（y轴坐标逐渐移至40）。然后其它的点击对于位置的改变没有任何的影响。你可以试着使用一个随机值（例如 40+(Math.random()*(205-40)）来设置y轴坐标。你可以发现动画始终会将移动到新位置的时间匹配在4秒内完成。

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

        text: "standalone animation"

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

第三个火箭

第三个火箭使用standalone animation独立动画策略。这个动画由一个私有的元素定义并且可以写在文档的任何地方。点击火箭调用动画函数start()来启动动画。每一个动画都有start()，stop()，resume()，restart()函数。这个动画自身可以比其他类型的动画更早的获取到更多的相关信息。我们只需要定义目标和目标元素的属性需要怎样改变的一个动画。我们定义一个to属性的值，在这个例子中我们也定义了一个from属性的值允许动画可以重复运行。



点击背景能够重新设置所有的火箭回到它们的初始位置。第一个火箭无法被重置，只有重启程序重新加载元素才能重置它。

注意

另一个启动/停止一个动画的方法是绑定一个动画的**running**属性。当需要用户输入控制属性时这种方法非常有用：

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

5.1.3 缓冲曲线（Easing Curves）

属性值的改变能够通过一个动画来控制，缓冲曲线属性影响了一个属性值改变的插值算法。我们现在已经定义的动画都使用了一种线性的插值算法，因为一个动画的默认缓冲类型是Easing.Linear。在一个小场景下的x轴与y轴坐标改变可以得到最好的视觉效果。一个线性插值算法将会在动画开始时使用from的值到动画结束时使用的to值绘制一条直线，所以缓冲类型定义了曲线的变化情况。精心为一个移动的对象挑选一个合适的缓冲类型将会使界面更加自然，例如一个页面的滑出，最初使用缓慢的速度滑出，然后在最后滑出时使用高速滑出，类似翻书一样的效果。

注意

不要过度的使用动画。用户界面动画的设计应该尽量小心，动画是让界面更加生动而不是充满整个界面。眼睛对于移动的东西非常敏感，很容易干扰用户的使用。

在下面的例子中我们将会使用不同的缓冲曲线，每一种缓冲曲线都都使用了一个可点击图片来展示，点击将会在动画中设置一个新的缓冲类型并且使用这种曲线重新启动动画。

扩展可点击图像**V3**（**ClickableImage V3**）

我们给图片和文本添加了一个小的外框来增强我们的ClickableImage。添加一个属性property bool framed: false来作为我们的API，基于framed的值我们能够设置这个框是否可见，并且不破坏之前用户的使用。下面是我们做的修改。

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

import QtQuick 2.0

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

```
    signal clicked

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

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

这个例子的代码非常简洁。我们使用了一连串的缓冲曲线的名称（property variant easings）并且在一个Repeater（重复元素）中将它们分配给一个ClickableImage。图片的源路径通过一个命名方案来定义，一个叫做“InQuad”的缓冲曲线在“curves/InQuad.png”中有一个对应的图片。如果你点击一个曲线图，这个点击将会分配一个缓冲类型给动画然后重新启动动画。动画自身是用来设置方块的x坐标属性在2秒内变化的独立动画。

```
// easingtypes.qml

import QtQuick 2.0

DarkSquare {
    id: root
    width: 600
    height: 340

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


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

```
            }
        }
    }

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

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

当你运行这个例子时，请注意观察动画的改变速度。一些动画对于这个对象看起来很自然，一些看起来非常恼火。

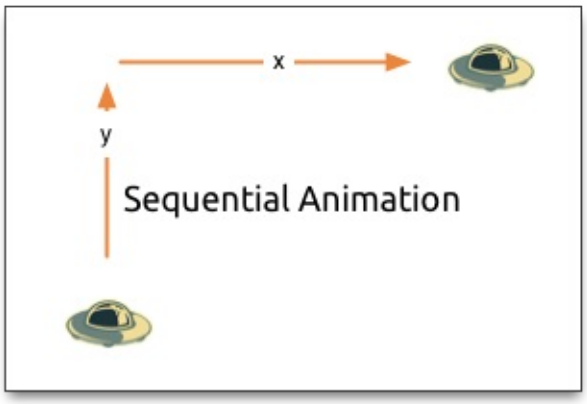
除了duration属性与easing.type属性，你也可以对动画进行微调。例如PropertyAnimation属性，大多数动画都支持附加的easing.amplitude（缓冲振幅），easing.overshoot（缓冲溢出），easing.period（缓冲周期），这些属性允许你对个别的缓冲曲线进行微调。不是所有的缓冲曲线都支持这些参数。可以查看Qt PropertyAnimation文档中的缓冲列表（easing table）来查看一个缓冲曲线的相关参数。

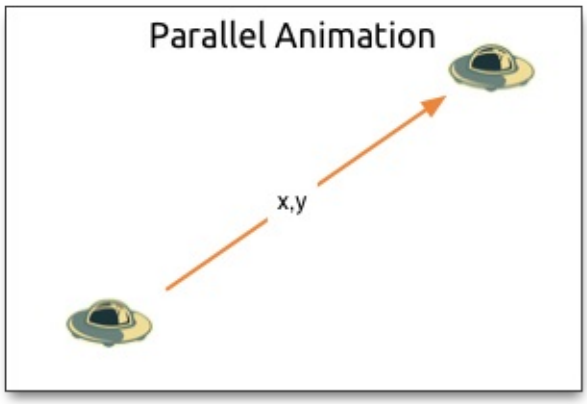
注意

对于用户界面正确的动画非常重要。请记住动画是帮助用户界面更加生动而不是刺激用户的眼睛。

5.1.4 动画分组（Grouped Animations）

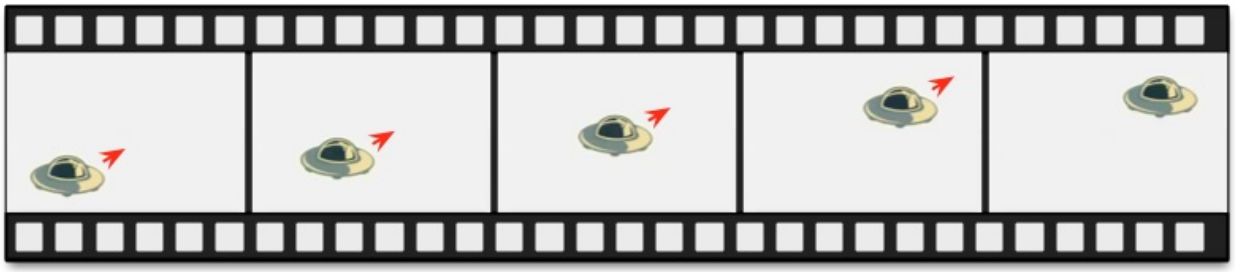
通常使用的动画比一个属性的动画更加复杂。例如你想同时运行几个动画并把他们连接起来，或者在一个一个的运行，或者在两个动画之间执行一个脚本。动画分组提供了很好的帮助，作为命名建议可以叫做一组动画。有两种方法来分组：平行与连续。你可以使用SequentialAnimation（连续动画）和ParallelAnimation（平行动画）来实现它们，它们作为动画的容器来包含其它的动画元素。





当开始时，平行元素的所有子动画都会平行运行，它允许你在同一时间使用不同的属性来播放动画。

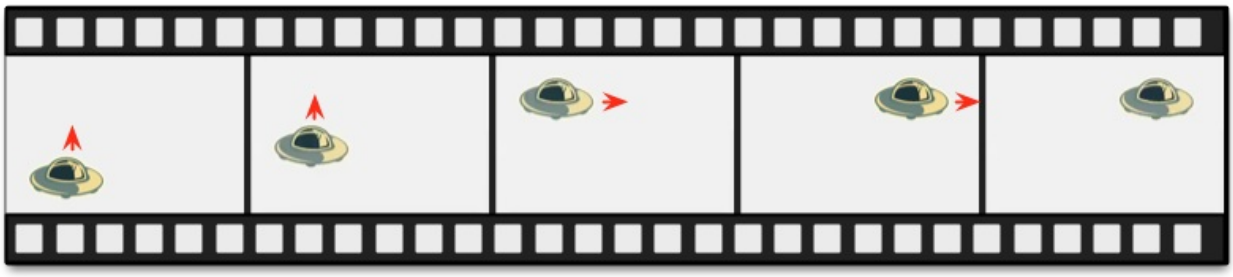
```
// parallelanimation.qml
import QtQuick 2.0

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

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

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



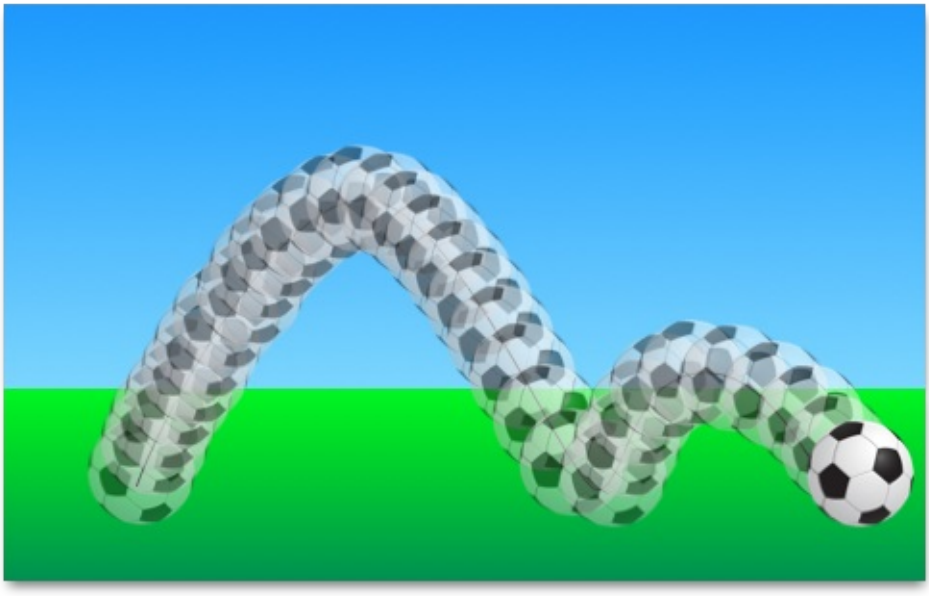
一个连续的动画将会一个一个的运行子动画。

```
// sequentialanimation.qml
import QtQuick 2.0

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

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

    SequentialAnimation {
        id: anim
        NumberAnimation {
            target: rocket
            properties: "y"
            to: 20
            // 60% of time to travel up
            duration: root.duration*0.6
        }
        NumberAnimation {
            target: rocket
            properties: "x"
            to: 160
            // 40% of time to travel sideways
            duration: root.duration*0.4
        }
    }
}
```



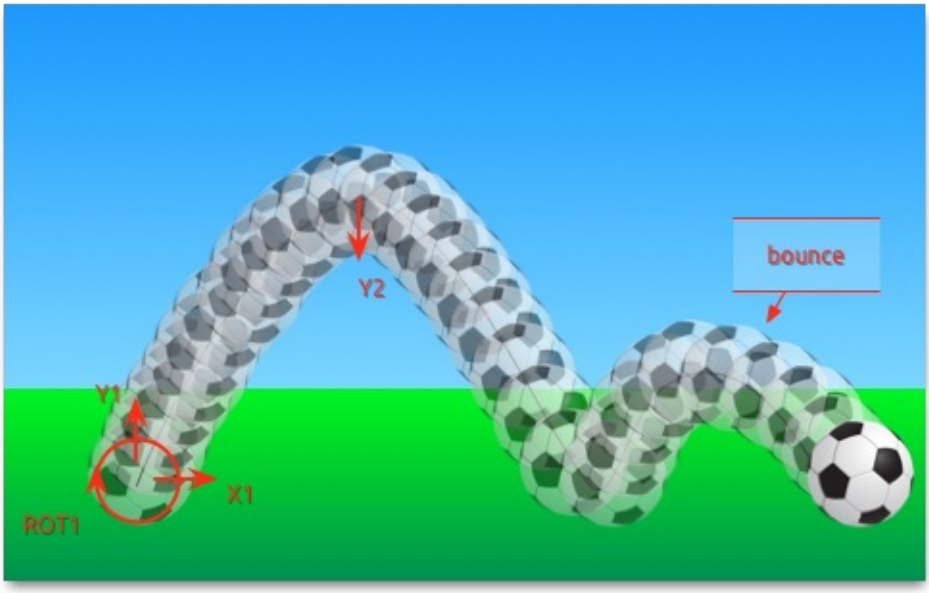
分组动画也可以被嵌套，例如一个连续动画可以拥有两个平行动画作为子动画。我们来看看这个足球的例子。这个动画描述了一个从左向右扔一个球的行为：



要弄明白这个动画我们需要剖析这个目标的运动过程。我们需要记住这个动画是通过属性变化来实现的动画，下面是不同部分的转换：

- 从左向右的x坐标转换（X1）。
- 从下往上的y坐标转换（Y1）然后跟着一个从上往下的Y坐标转换（Y2）。
- 整个动画过程中360度旋转。

这个动画将会花掉3秒钟的时间。



我们使用一个空的基本元素对象（Item）作为根元素，它的宽度为480，高度为300。

```
import QtQuick 1.1

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

    ...
}
```

我们定义动画的总持续时间作为参考，以便更好的同步各部分的动画。

下一步我们需需要添加一个背景，在我们这个例子中有两个矩形框分别使用了绿色渐变和蓝色渐变填充。

```
    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" }
        }
    }
```

上面部分的蓝色区域高度为200像素，下面部分的区域使用上面的蓝色区域的底作为锚定的顶，使用根元素的底作为底。

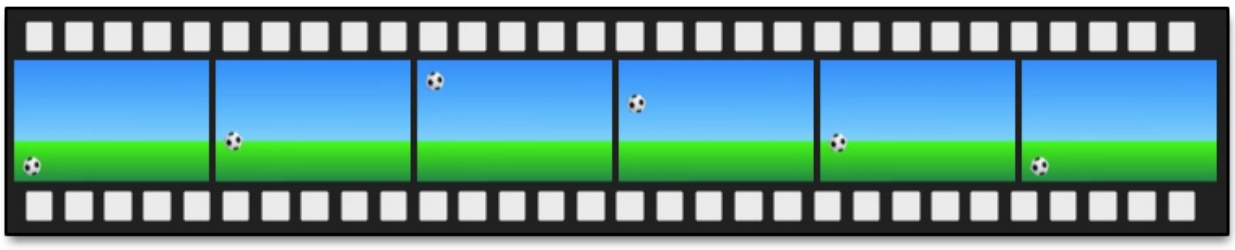
让我们将足球加入到屏幕上，足球是一个图片，位于路径“assets/soccer_ball.png”。首先我们需要将它放置在左下角接近边界处。

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

        MouseArea {
            anchors.fill: parent
            onClicked: {
                ball.x = 20; ball.y = 240
                anim.restart()
            }
        }
    }
```

图片与鼠标区域连接，点击球将会重置球的状态，并且动画重新开始。

首先使用一个连续的动画来播放两次的y轴变换。

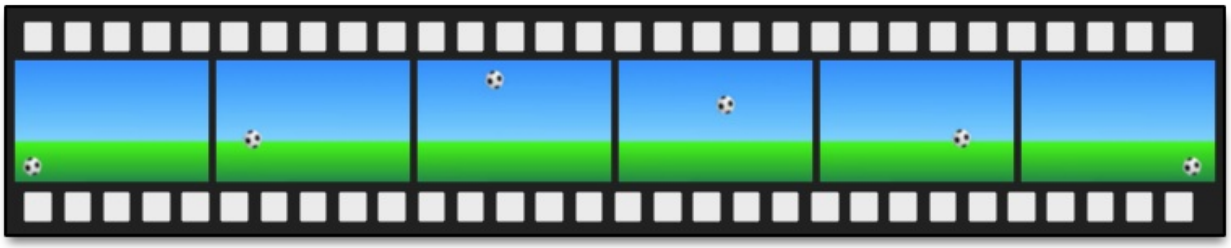
```
SequentialAnimation {
    id: anim
    NumberAnimation {
        target: ball
        properties: "y"
        to: 20
        duration: root.duration * 0.4
    }
    NumberAnimation {
        target: ball
        properties: "y"
        to: 240
        duration: root.duration * 0.6
    }
}
```



在动画总时间的40%的时间里完成上升部分，在动画总时间的60%的时间里完成下降部分，一个动画完成后播放下一个动画。目前还没有使用任何缓冲曲线。缓冲曲线将在后面使用easing curves来添加，现在我们只关心如何使用动画来完成过渡。

现在我们需要添加x轴坐标转换。x轴坐标转换需要与y轴坐标转换同时进行，所以我们需要将y轴坐标转换的连续动画和x轴坐标转换一起压缩进一个平行动画中。

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



最后我们想要旋转这个球，我们需要向平行动画中添加一个新的动画，我们选择RotationAnimation来实现旋转。

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

我们已经完成了整个动画链表，然后我们需要给动画提供一个正确的缓冲曲线来描述一个移动的球。对于Y1动画我们使用Easing.OutCirc缓冲曲线，它看起来更像是一个圆周运动。Y2使用了Easing.OutBounce缓冲曲线，因为在最后球会发生反弹。（试试使用Easing.InBounce，你会发现反弹将会立刻开始。）。X1和ROT1动画都使用线性曲线。

下面是这个动画最后的代码，提供给你作为参考：

```
ParallelAnimation {
    id: anim
    SequentialAnimation {
        NumberAnimation {
            target: ball
            properties: "y"
            to: 20
            duration: root.duration * 0.4
            easing.type: Easing.OutCirc
        }
        NumberAnimation {
            target: ball
            properties: "y"
            to: 240
            duration: root.duration * 0.6
            easing.type: Easing.OutBounce
        }
    }
    NumberAnimation {
        target: ball
```

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

状态与过渡（States and Transitions）

通常我们将用户界面描述为一种状态。一个状态定义了一组属性的改变，并且会在一定的条件下被触发。另外在这些状态转化的过程中可以有一个过渡，定义了这些属性的动画或者一些附加的动作。当进入一个新的状态时，动作也可以被执行。

5.2.1 状态（States）

在QML中，使用State元素来定义状态，需要与基础元素对象（Item）的states序列属性连接。状态通过它的状态名来鉴别，由组成它的一系列简单的属性来改变元素。默认的状态在初始化元素属性时定义，并命名为""（一个空的字符串）。

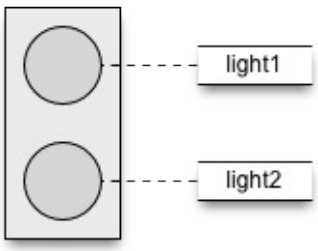
```
Item {
    id: root
    states: [
        State {
            name: "go"
            PropertyChanges { ... }
        },
        State {
            name: "stop"
            PropertyChanges { ... }
        }
    ]
}
```

状态的改变由分配一个元素新的状态属性名来完成。

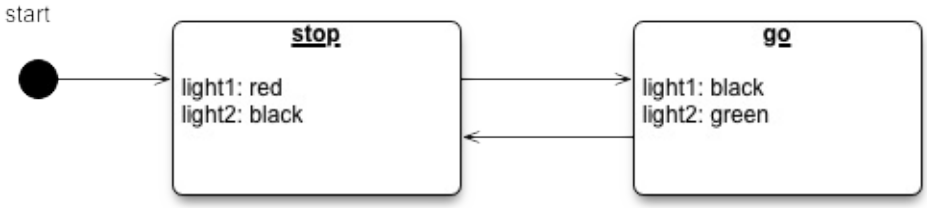
注意

另一种切换属性的方法是使用状态元素的**when**属性。**when**属性能够被设置为一个表达式的结果，当结果为**true**时，状态被使用。

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

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



例如一个交通信号灯有两个信号灯。上面的一个信号灯使用红色，下面的信号灯使用绿色。在这个例子中，两个信号灯不会同时发光。让我们看看状态图。



当系统启动时，它会自动切换到停止模式作为默认状态。停止状态改变了light1为红色并且light2为黑色（关闭）。一个外部的事件能够触发现在的状态变换为“go”状态。在go状态下，我们改变颜色属性，light1变为黑色（关闭），light2变为绿色。

为了实现这个方案，我们给这两个灯绘制一个用户界面的草图，为了简单起见，我们使用两个包含园边的矩形框，设置园半径为宽度的一半（宽度与高度相同）。

```
    Rectangle {
        id: light1
        x: 25; y: 15
        width: 100; height: width
        radius: width/2
        color: "black"
    }

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

就像在状态图中定义的一样，我们有一个“go”状态和一个“stop”状态，它们将会分别将交通灯改变为红色和绿色。我们设置state属性到stop来确保初始化状态为stop状态。

注意

我们可以只使用**“go”**状态来达到同样的效果，设置颜色**light1**为红色，颜色**light2**为黑色。初始化状态**“”**（空字符串）定义初始化属性，并且扮演类似**“stop”**状态的角色。

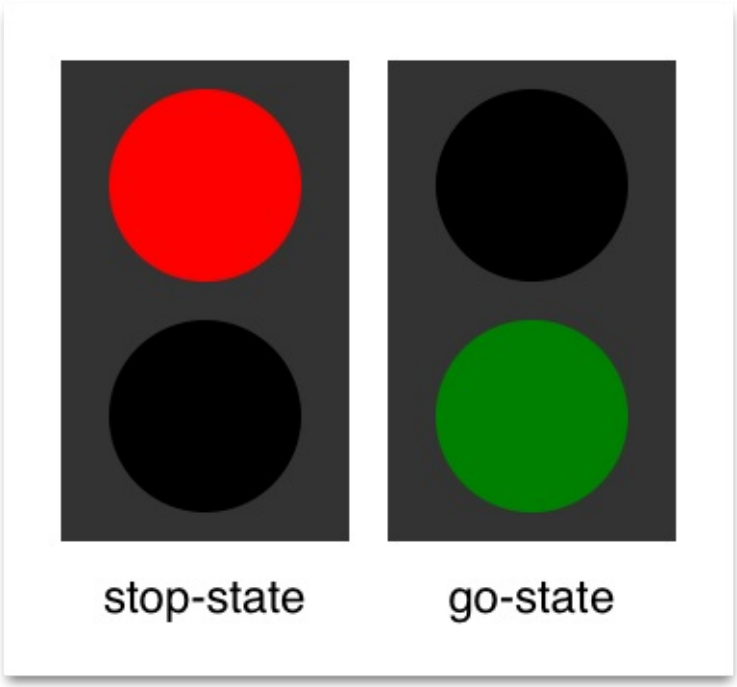
```
    state: "stop"
```

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

PropertyChanges{ target: light2; color: "black" }在这个例子中不是必要的，因为light2初始化颜色已经是黑色了。在一个状态中，只需要描述属性如何从它们的默认状态改变（而不是前一个状态的改变）。

使用鼠标区域覆盖整个交通灯，并且绑定在点击时切换go和stop状态。

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



我们现在已经成功实现了交通灯的状态切换。为了让用户界面看起来更加自然，我们需要使用动画效果来增加一些过渡。一个过渡能够被状态的改变触发。

注意

可以使用一个简单逻辑的脚本来替换**QML**状态。开发人员很容易落入这种陷阱，写的代码更像一个**JavaScript**程序而不是一个**QML**程序。

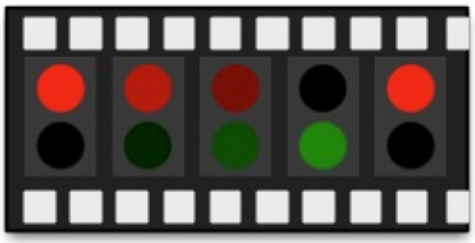
5.2.2 过渡（Transitions）

一系列的过渡能够被加入任何元素，一个过渡由状态的改变触发执行。你可以使用属性的from:和to:来定义状态改变的指定过渡。这两个属性就像一个过滤器，当过滤器为true时，过渡生效。你也可以使用“”来表示任何状态。例如*from:""*; to:"*"表示从任一状态到另一个任一状态的默认值，这意味着过渡用于每个状态的切换。

在这个例子中，我们期望从状态“go”到“stop”转换时实现一个颜色改变的动画。对于从“stop”到“go”状态的改变，我们期望保持颜色的直接改变，不使用过渡。我们使用from和to来限制过渡只在从“go”到“stop”时生效。在过渡中我们给每个灯添加两个颜色的动画，这个动画将按照状态的描述来改变属性。

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

你可以点击用户界面来改变状态。试试点击用户界面，当状态从“stop”到“go”时，你将会发现改变立刻发生了。



接下来，你可以修改下这个例子，例如缩小未点亮的等来突出点亮的等。为此，你需要在状态中添加一个属性用来缩放，并且操作一个动画来播放缩放属性的过渡。另一个选择是可以添加一个“attention”状态，灯会出现黄色闪烁，为此你需要添加为这个过渡添加一个一秒连续的动画来显示黄色（使用“to”属性来实现，一秒后变为黑色）。也许你也可以改变缓冲曲线来使这个例子更加生动。

高级用法（Advanced Techniques）

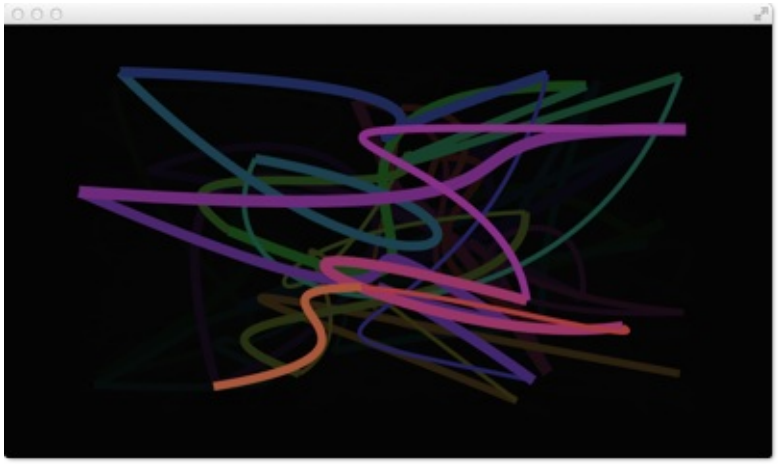
后续添加。

Canvas Element

注意

最后一次构建：**2014年1月20日下午18:00**。

这章的源代码能够在**assetts folder**找到。



在早些时候的Qt4中加入QML时，一些开发者讨论如何在QtQuick中绘制一个圆形。类似圆形的问题，一些开发者也对于其它的形状的支持进行了讨论。在QtQuick中没有圆形，只有矩形。在Qt4中，如果你需要一个除了矩形外的形状，你需要使用图片或者使用你自己写的C++圆形元素。

Qt5中引进了画布元素（canvas element），允许脚本绘制。画布元素（canvas element）提供了一个依赖于分辨率的位图画布，你可以使用JavaScript脚本来绘制图形，制作游戏或者其它的动态图像。画布元素（canvas element）是基于HTML5的画布元素来完成的。

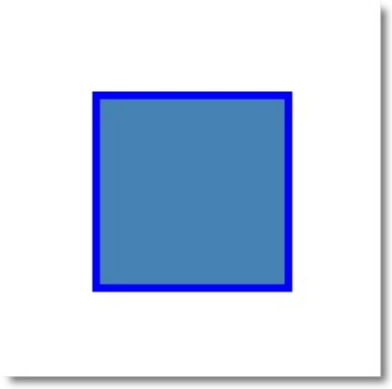
画布元素（canvas element）的基本思想是使用一个2D对象来渲染路径。这个2D对象包括了必要的绘图函数，画布元素（canvas element）充当绘制画布。2D对象支持画笔，填充，渐变，文本和绘制路径创建命令。

让我们看看一个简单的路径绘制的例子：

```
import QtQuick 2.0

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

```
        // begin a new path to draw
        ctx.beginPath()
        // top-left start point
        ctx.moveTo(50,50)
        // upper line
        ctx.lineTo(150,50)
        // right line
        ctx.lineTo(150,150)
        // bottom line
        ctx.lineTo(50,150)
        // left line through path closing
        ctx.closePath()
        // fill using fill style
        ctx.fill()
        // stroke using line width and stroke style
        ctx.stroke()
    }
}
```

这个例子产生了一个在坐标（50,50），高宽为100的填充矩形框，并且使用了画笔来修饰边界。



画笔的宽度被设置为4个像素，并且定义strokeStyle（画笔样式）为蓝色。最后的形状由设置填充样式（fillStyle）为steelblue颜色，然后填充完成的。只有调用stroke或者fill函数，创建的路径才会绘制，它们与其它的函数使用是相互独立的。调用stroke或者fill将会绘制当前的路径，创建的路径是不可重用的，只有绘制状态能够被存储和恢复。

在QML中，画布元素（canvas element）充当了绘制的容器。2D绘制对象提供了实际绘制的方法。绘制需要在onPaint事件中完成。

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

画布自身提供了典型的二维笛卡尔坐标系统，左上角是（0,0）坐标。Y轴坐标轴向下，X轴坐标轴向右。

典型绘制命令调用如下：

1. 装载画笔或者填充模式
2. 创建绘制路径
3. 使用画笔或者填充绘制路径

```
    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()
    }
```

这将产生一个从P1（50，50）到P2（150,50）水平线。

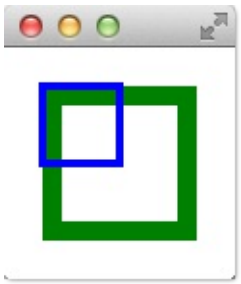
注意

通常在你重置了路径后你将会设置一个开始点，所以，在**beginPath()**这个操作后，你需要使用**moveTo**来设置开始点。

便捷的接口（Convenient API）

在绘制矩形时，我们提供了一个便捷的接口，而不需要调用stroke或者fill来完成。

```
// convenient.qml

import QtQuick 2.0

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

        // draw a 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)
    }
}
```



注意

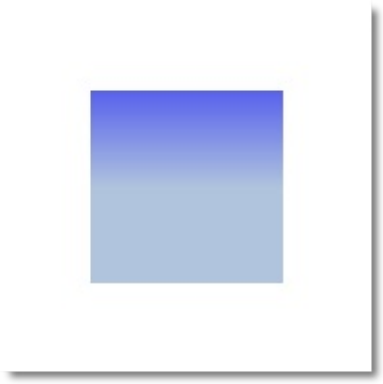
画笔的绘制区域由中间向两边延展。一个宽度为**4**像素的画笔将会在绘制路径的里面绘制**2**个像素，外面绘制**2**个像素。

渐变（Gradients）

画布中可以使用颜色填充也可以使用渐变或者图像来填充。

```
    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)
    }
```

在这个例子中，渐变色定义在开始点（100,0）到结束点（100,200）。在我们画布中是一个中间垂直的线。渐变色在停止点定义一个颜色，范围从0.0到1.0。这里我们使用一个蓝色作为0.0（100,0），一个高亮刚蓝色作为0.5（100,200）。渐变色的定义比我们想要绘制的矩形更大，所以矩形在它定义的范围内对渐变进行了裁剪。



注意

渐变色是在画布坐标下定义的，而不是在绘制路径相对坐标下定义的。画布中没有相对坐标的概念。

阴影（Shadows）

注意

在**Qt5**的**alpha**版本中，我们使用阴影遇到了一些问题。

2D对象的路径可以使用阴影增强显示效果。阴影是一个区域的轮廓线使用偏移量，颜色和模糊来实现的。所以你需要指定一个阴影颜色（shadowColor），阴影X轴偏移值（shadowOffsetX），阴影Y轴偏移值（shadowOffsetY）和阴影模糊（shadowBlur）。这些参数的定义都使用2D context来定义。2D context是唯一的绘制操作接口。

阴影也可以用来创建发光的效果。在下面的例子中我们使用白色的光创建了一个“Earth”的文本。在一个黑色的背景上可以有更加好的显示效果。

首先我们绘制黑色背景：

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

然后定义我们的阴影配置：

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

最后我们使用加粗的，80像素宽度的Ubuntu字体来绘制“Earth”文本：

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

图片（Images）

QML画布支持多种资源的图片绘制。在画布中使用一个图片需要先加载图片资源。在我们的例子中我们使用Component.onCompleted操作来加载图片。

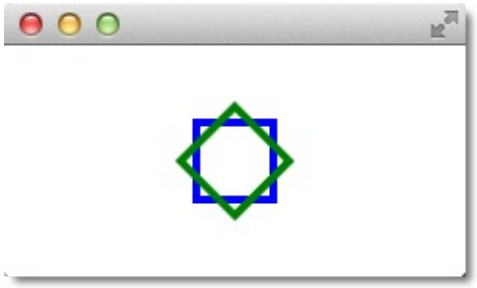
```
    onPaint: {
        var ctx = getContext("2d")


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

        // store current context setup
        ctx.save()
        ctx.strokeStyle = 'red'
        // create a triangle as clip region
        ctx.beginPath()
        ctx.moveTo(10,10)
        ctx.lineTo(55,10)
        ctx.lineTo(35,55)
        ctx.closePath()
        // translate coordinate system
        ctx.translate(100,0)
        ctx.clip()  // create clip from triangle path
        // draw image with clip applied
        ctx.drawImage('assets/ball.png', 10, 10)
        // draw stroke around path
        ctx.stroke()
        // restore previous setup
        ctx.restore()

    }

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

在左边，足球图片使用10×10的大小绘制在左上方的位置。在右边我们对足球图片进行了裁剪。图片或者轮廓路径都可以使用一个路径来裁剪。裁剪需要定义一个裁剪路径，然后调用clip()函数来实现裁剪。在clip()之前所有的绘制操作都会用来进行裁剪。如果还原了之前的状态或者定义裁剪区域为整个画布时，裁剪是无效的。

转换（Transformation）

画布有多种方式来转换坐标系。这些操作非常类似于QML元素的转换。你可以通过缩放（scale），旋转（rotate），translate（移动）来转换坐标系。与QML元素的转换不同的是，转换原点通常就是画布原点。例如，从中心点放大一个封闭的路径，你需要先将画布原点移动到整个封闭的路径的中心点上。使用这些转换的方法你可以创建一些更加复杂的转换。

```
// transform.qml

import QtQuick 2.0

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

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

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



除了移动画布外，也可以使用scale(x,y)来缩放x,y坐标轴。旋转使用rotate(angle)，angle是角度（360度=2*Math.PI）。使用setTransform(m11,m12,m21,m22,dx,dy)来完成矩阵转换。

警告

QML画布中的转换与**HTML5**画布中的机制有些不同。不确定这是不是一个**Bug**。

注意

重置矩阵你可以调用**resetTransform()**函数来完成，这个函数会将转换矩阵还原为单位矩阵。

组合模式（Composition Mode）

组合允许你绘制一个形状然后与已有的像素点集合混合。画布提供了多种组合模式，使用globalCompositeOperation(mode)来设置。

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

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

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

下面这个例子遍历了列表中的组合模式，使用对应的组合模式生成了一个矩形与圆形的组合。

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

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

        for(var i=0; i<operation.length; i++) {
            var dx = Math.floor(i%6)*100
            var dy = Math.floor(i/6)*100
            ctx.save()
            ctx.fillStyle = '#33a9ff'
            ctx.fillRect(10+dx,10+dy,60,60)
            // TODO: does not work yet
            ctx.globalCompositeOperation = root.operation[i]
            ctx.fillStyle = '#ff33a9'
            ctx.globalAlpha = 0.75
```

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

像素缓冲（Pixels Buffer）

当你使用画布时，你可以检索读取画布上的像素数据，或者操作画布上的像素。读取图像数据使用createImageData(sw,sh)或者getImageData(sx,sy,sw,sh)。这两个函数都会返回一个包含宽度（width），高度（height）和数据（data）的图像数据（ImageData）对象。图像数据包含了一维数组像素数据，使用RGBA格式进行检索。每个数据的数据范围在0到255之间。设置画布的像素数据你可以使用putImageData(imagedata,dx,dy)函数来完成。

另一种检索画布内容的方法是将画布的数据存储进一张图片中。可以使用画布的函数save(path)或者toDataURL(mimeType)来完成，toDataURL(mimeType)会返回一个图片的地址，这个链接可以直接用Image元素来读取。

```
import QtQuick 2.0

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

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

    Timer {
        interval: 1000
        running: true
```

```
        triggeredOnStart: true
        repeat: true
        onTriggered: canvas.requestPaint()
    }
}
```

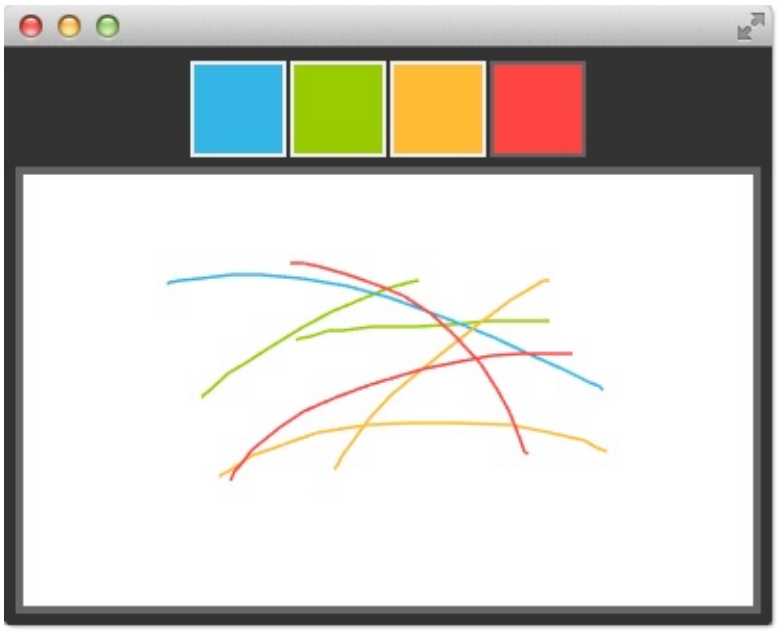
在我们这个例子中，我们每秒在左边的画布中绘制一个的圆形。当使用鼠标点击画布内容时，会将内容存储为一个图片链接。在右边将会展示这个存储的图片。

注意

在**Qt5**的**Alpha**版本中，检索图像数据似乎不能工作。

画布绘制（Canvas Paint）

在这个例子中我们将使用画布（Canvas）创建一个简单的绘制程序。



在我们场景的顶部我们使用行定位器排列四个方形的颜色块。一个颜色块是一个简单的矩形，使用鼠标区域来检测点击。

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

颜色存储在一个数组中，作为绘制颜色使用。当用户点击一个矩形时，矩形内的颜色被设置为colorTools的paintColor属性。

为了在画布上跟踪鼠标事件，我们使用鼠标区域（MouseArea）覆盖画布元素，并连接点击和移动操作。

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

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

鼠标点击存储在laxstX与lastY属性中。每次鼠标位置的改变会触发画布的重绘，这将会调用onPaint操作。

最后绘制用户的笔划，在onPaint操作中，我们绘制从最近改变的点上开始绘制一条新的路径，然后我们从鼠标区域采集新的点，使用选择的颜色绘制线段到新的点上。鼠标位置被存储为新改变的位置。

HTML5画布移植（Porting from HTML5 Canvas）

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

移植一个HTML5画布图像到QML画布非常简单。在成百上千的例子中，我们选择了一个来移植。

螺旋图形（**Spiro Graph**）

我们使用一个来自Mozila项目的螺旋图形例子来作为我们的基础示例。原始的HTML5代码被作为画布教程发布。

下面是我们需要修改的代码：

- Qt Quick要求定义变量使用，所以我们需要添加var的定义：

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

- 修改绘制方法接收Context2D对象：

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

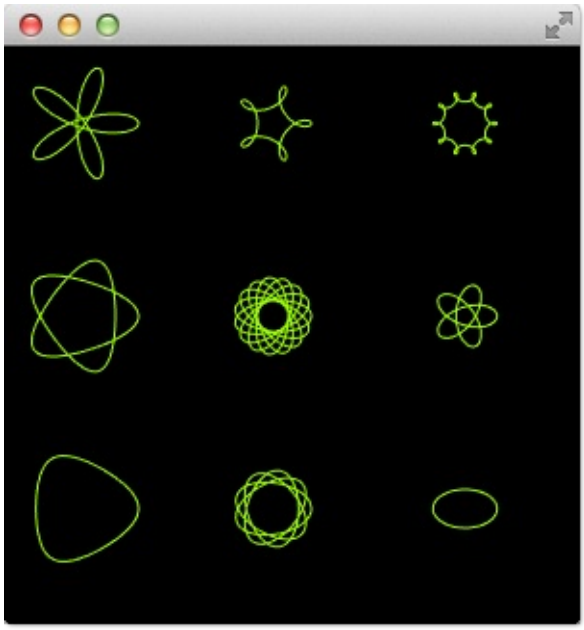
- 由于不同的大小，我们需要对每个螺旋适配转换：

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

最后我们实现onPaint操作。在onPaint中我们请求一个context，并且调用我们的绘制方法。

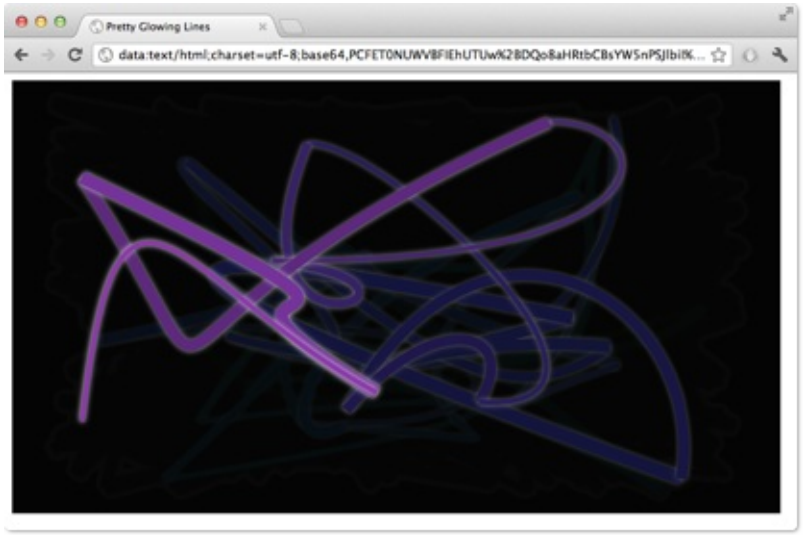
```
onPaint: {
    var ctx = getContext("2d");
    draw(ctx);
}
```

下面这个结果就是我们使用QML画布移植的螺旋图形。



发光线（**Glowing Lines**）

下面有一个更加复杂的移植来自W3C组织。原始的发光线有些很不错的地方，这使得移植更加具有挑战性。




```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <title>Pretty Glowing Lines</title>
</head>
<body>

<canvas width="800" height="450"></canvas>
<script>
var context = document.getElementsByTagName('canvas')[0].getContext('2d');

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

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

    context.save();

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

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

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

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

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

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

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

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

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

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

在HTML5中，context2D对象可以随意在画布上绘制。在QML中，只能在onPaint操作中绘制。在HTML5中，通常调用setInterval使用计时器触发线段的绘制或者清屏。由于QML中不同的操作方法，仅仅只是调用

这些函数不能实现我们想要的结果，因为我们需要通过onPaint操作来实现。我们也需要修改颜色的格式。让我们看看需要改变哪些东西。

修改从画布元素开始。为了简单，我们使用画布元素（Canvas）作为我们QML文件的根元素。

```
import QtQuick 2.0

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

   ...
}
```

代替直接调用的setInterval函数，我们使用两个计时器来请求重新绘制。一个计时器触发间隔较短，允许我们可以执行一些代码。我们无法告诉绘制函数哪个操作是我想触发的，我们为每个操作定义一个布尔标识，当重新绘制请求时，我们请求一个操作并且触发它。

下面是线段绘制的代码，清屏操作类似。

```
...
property bool requestLine: false

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

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

现在我们已经有了告诉onPaint操作中我们需要执行哪个操作的指示。当我们进入onPaint处理每个绘制请求时，我们需要提取画布元素中的初始化变量。

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

现在我们的绘制函数应该像这样：

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

线段绘制函数提取画布作为一个参数。

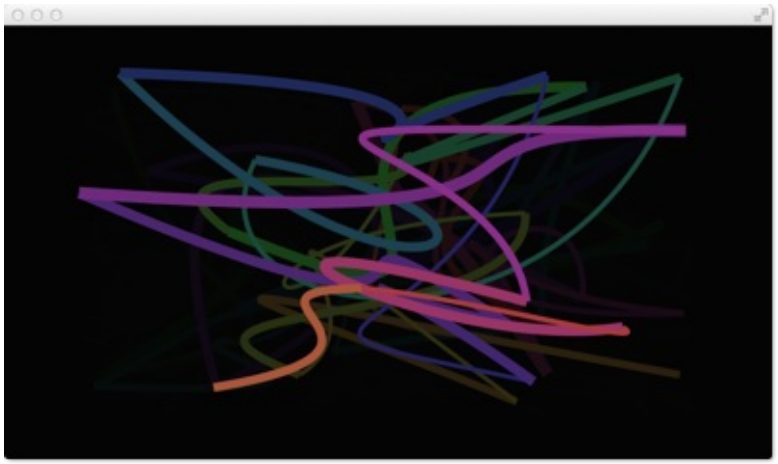
```
function line(context) {
    context.save();
    context.translate(canvas.width/2, canvas.height/2);
    context.scale(0.9, 0.9);
    context.translate(-canvas.width/2, -canvas.height/2);
    context.beginPath();
    context.lineWidth = 5 + Math.random() * 10;
    context.moveTo(lastX, lastY);
 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

最大的变化是使用QML的Qt.rgba()和Qt.hsla()。在QML中需要把变量值适配在0.0到1.0之间。

同样应用在清屏函数中。

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

下面是最终结果（目前没有阴影）类似下面这样。



查看下面的链接获得更多的信息：

- W3C HTML Canvas 2D Context Specification
- Mozilla Canvas Documentation
- HTML5 Canvas Tutorial

Particle Simulations

注意

最后一次构建：**2014年1月20日下午18:00**。

这章的源代码能够在**assetts folder**找到。

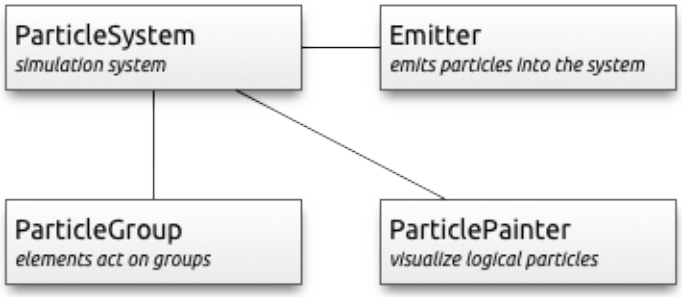
粒子模拟是计算机图形技术的可视化图形效果。典型的效果有：落叶，火焰，爆炸，流星，云等等。

它不同于其它图形渲染，粒子是基于模糊来渲染。它的结果在基于像素下是不可预测的。粒子系统的参数描述了随机模拟的边界。传统的渲染技术实现粒子渲染效果很困难。有一个好消息是你可以使用QML元素与粒子系统交互。同时参数也可以看做是属性，这些参数可以使用传统的动画技术来实现动态效果。

概念（Concept）

粒子模拟的核心是粒子系统（ParticleSystem），它控制了共享时间线。一个场景下可以有多个粒子系统，每个都有自己独立的时间线。一个粒子使用发射器元素（Emitter）发射，使用粒子画笔（ParticlePainter）实现可视化，它可以是一张图片，一个QML项或者一个着色项（shader item）。一个发射器元素（Emitter）也提供向量来控制粒子方向。一个粒子被发送后就再也无法控制。粒子模型提供粒子控制器（Affector），它可以控制已发射粒子的参数。

在一个系统中，粒子可以使用粒子群元素（ParticleGroup）来共享移动时间。默认下，每个例子都属于空（""）组。



- 粒子系统（ParticleSystem）- 管理发射器之间的共享时间线。
- 发射器（Emitter）- 向系统中发射逻辑粒子。
- 粒子画笔（ParticlePainter）- 实现粒子可视化。
- 方向（Direction）- 已发射粒子的向量空间。
- 粒子组（ParticleGroup）- 每个粒子是一个粒子组的成员。
- 粒子控制器（Affector）- 控制已发射粒子。

简单的模拟（Simple Simulation）

让我们从一个简单的模拟开始学习。Qt Quick使用简单的粒子渲染非常简单。下面是我们需要的：

- 绑定所有元素到一个模拟的粒子系统（ParticleSystem）。
- 一个向系统发射粒子的发射器（Emitter）。
- 一个ParticlePainter派生元素，用来实现粒子的可视化。

```
import QtQuick 2.0
import QtQuick.Particles 2.0

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

    ParticleSystem {
        id: particleSystem
    }

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

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

例子的运行结果如下所示：

我们使用一个80x80的黑色矩形框作为我们的根元素和背景。然后我们定义一个粒子系统（ParticleSystem）。这通常是粒子系统绑定所有元素的第一步。下一个元素是发射器（Emitter），它定义了基于矩形框的发射区域和发射粒子的基础属性。发射器使用system属性与粒子系统进行绑定。

在这个例子中，发射器每秒发射10个粒子（emitRate:10）到发射器的区域，每个粒子的生命周期是1000毫秒（lifeSpan:1000），一个已发射粒子的生命周期变化是500毫秒（lifeSpanVariation:500）。一个粒子开始的大小是16个像素（size:16），生命周期结束时的大小是32个像素（endSize:32）。

绿色边框的矩形是一个跟踪元素，用来显示发射器的几何形状。这个可视化展示了粒子在发射器矩形框内发射，但是渲染效果不被限制在发射器的矩形框内。渲染位置依赖于粒子的寿命和方向。这将帮助我们更加清楚的知道如何改变粒子的方向。

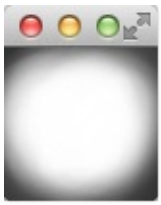
发射器发射逻辑粒子。一个逻辑粒子的可视化使用粒子画笔（ParticlePainter）来实现，在这个例子中我们使用了图像粒子（ImageParticle），使用一个图片链接作为源属性。图像粒子也有其它的属性用来控制粒子的外观。

- 发射频率（emitRate）- 每秒粒子发射数（默认为10个）。
- 生命周期（lifeSpan）- 粒子持续时间（单位毫秒，默认为1000毫秒）。
- 初始大小（size），结束大小（endSize）- 粒子在它的生命周期的开始和结束时的大小（默认为16像素）。

改变这些属性将会彻底改变显示结果：

```
    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' }
    }
```

增加发射频率为40，生命周期增加到2秒，开始大小为64像素，结束大小减少到32像素。



增加结束大小（endSize）可能会导致白色的背景出现。请注意粒子只有发射被限制在发射器定义的区域内，而粒子渲染是不会考虑这个参数的。

粒子参数（Particle Parameters）

我们已经知道通过改变发射器的行为就可以改变我们的粒子模拟。粒子画笔被用来绘制每一个粒子。 回到我们之前的粒子中，我们更新一下我们的图片粒子画笔（ImageParticle）。首先我们改变粒子图片为一个小的星形图片：

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

粒子使用金色来进行初始化，不同的粒子颜色变化范围为+/- 20%。

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

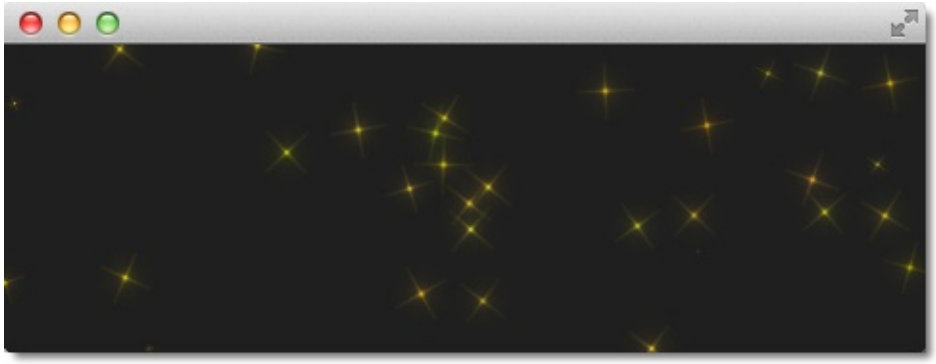
为了让场景更加生动，我们需要旋转粒子。每个粒子首先按顺时针旋转15度，不同的粒子在+/-5度之间变化。每个例子会不断的以每秒45度旋转。每个粒子的旋转速度在+/-15度之间变化：

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

最后，我们改变粒子的入场效果。 这个效果是粒子产生时的效果，在这个例子中，我们希望使用一个缩放效果：

```
entryEffect: ImageParticle.Scale
```

现在我们可以看到旋转的星星出现在我们的屏幕上。



下面是我们如何改变图片粒子画笔的代码段。

```
    ImageParticle {
```

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

粒子方向（Directed Particle）

我们已经看到了粒子的旋转，但是我们的粒子需要一个轨迹。轨迹由速度或者粒子随机方向的加速度指定，也可以叫做矢量空间。

有多种可用矢量空间用来定义粒子的速度或加速度：

- 角度方向（AngleDirection）- 使用角度的方向变化。
- 点方向（PointDirection）- 使用x,y组件组成的方向变化。
- 目标方向（TargetDirection）- 朝着目标点的方向变化。

让我们在场景下试着用速度方向将粒子从左边移动到右边。

首先使用角度方向（AngleDirection）。我们使用AngleDirection元素作为我们的发射器（Emitter）的速度属性：

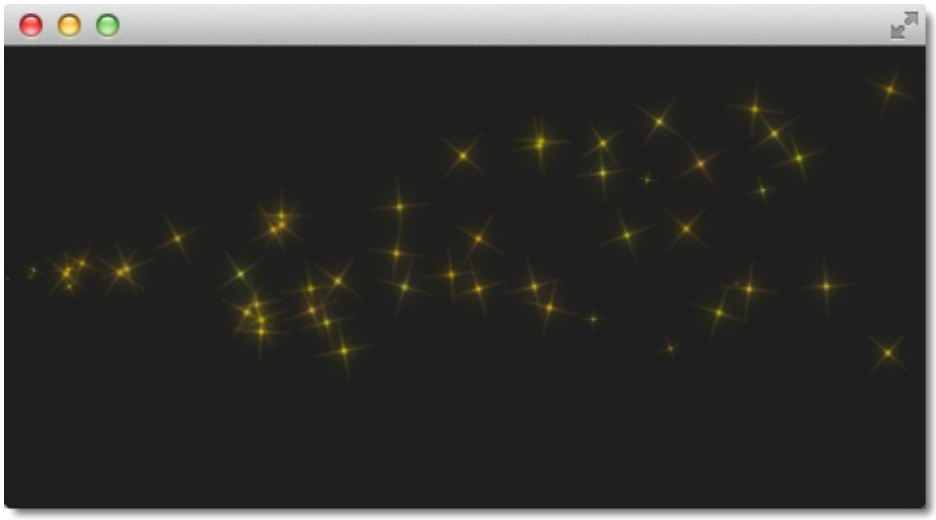
```
velocity: AngleDirection { }
```

粒子的发射将会使用指定的角度属性。角度值在0到360度之间，0度代表指向右边。在我们的例子中，例子将会移动到右边，所以0度已经指向右边方向。粒子的角度变化在+/-15度之间：

```
velocity: AngleDirection {
    angle: 0
    angleVariation: 15
}
```

现在我们已经设置了方向，下面是指定粒子的速度。它由一个梯度值定义，这个梯度值定义了每秒像素的变化。正如我们设置大约640像素，梯度值为100，看起来是一个不错的值。这意味着平均一个6.4秒生命周期的粒子可以穿越我们看到的区域。为了让粒子的穿越看起来更加有趣，我们使用magnitudeVariation来设置梯度值的变化，这个值是我们的梯度值的一半：

```
velocity: AngleDirection {
    ...
    magnitude: 100
    magnitudeVariation: 50
}
```



下面是完整的源码，平均的生命周期被设置为6..4秒。我们设置发射器的宽度和高度为1个像素，这意味着所有的粒子都从相同的位置发射出去，然后基于我们给定的轨迹运动。

```
    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
        }
    }
```

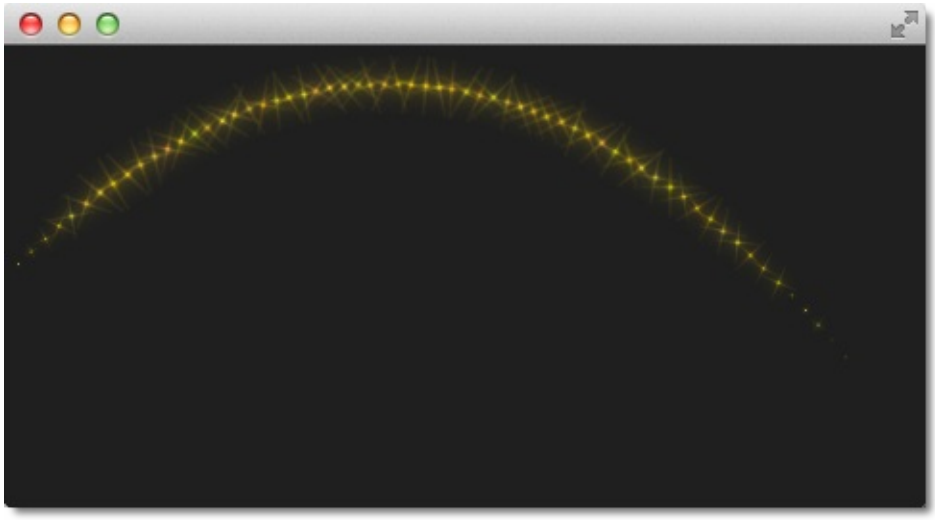
那么加速度做些什么？加速度是每个粒子加速度矢量，它会在运动的时间中改变速度矢量。例如我们做一个星星按照弧形运动的轨迹。我们将会改变我们的速度方向为-45度，然后移除变量，可以得到一个更连贯的弧形轨迹：

```
velocity: AngleDirection {
    angle: -45
    magnitude: 100
}
```

加速度的方向为90度（向下），加速度为速度的四分之一：

```
acceleration: AngleDirection {
    angle: 90
    magnitude: 25
}
```

结果是中间左方到右下的一个弧。



这个值是在不断的尝试与错误中发现的。

下面是发射器完整的代码。

```
    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
        }
    }
```

在下一个例子中，我们将使用点方向（PointDirection）矢量空间来再一次演示粒子从左到右的运动。

一个点方向（PointDirection）是由x和y组件组成的矢量空间。例如，如果你想粒子以45度的矢量运动，你需要为x，y指定相同的值。

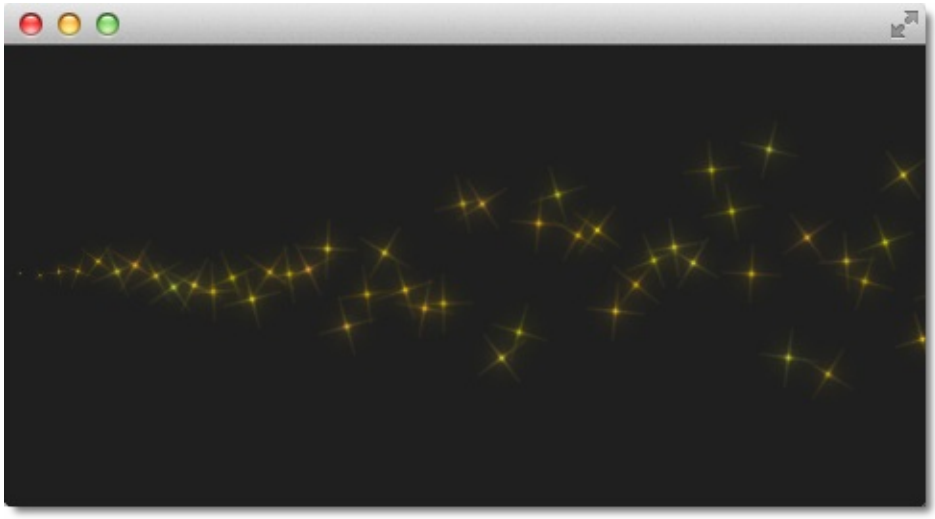
在我们的例子中，我们希望粒子在从左到右的例子中建立一个15度的圆锥。我们指定一个坐标方向（PointDirection）作为我们速度矢量空间：

```
velocity: PointDirection { }
```

为了达到运动速度每秒100个像素，我们设置x为100,。15度角（90度的1/6）,我们指定y变量为100/6：

```
velocity: PointDirection {
    x: 100
    y: 0
    xVariation: 0
    yVariation: 100/6
}
```

结果是粒子的运动从左到右构成了一个15度的圆锥。



现在是最后一个方案，目标方向（TargetDirection）。目标方向允许我们指定发射器或者一个QML项的x,y坐标值。当一个QML项的中心点成为一个目标点时，你可以指定目标变化值是x目标值的1/6来完成一个15度的圆锥：

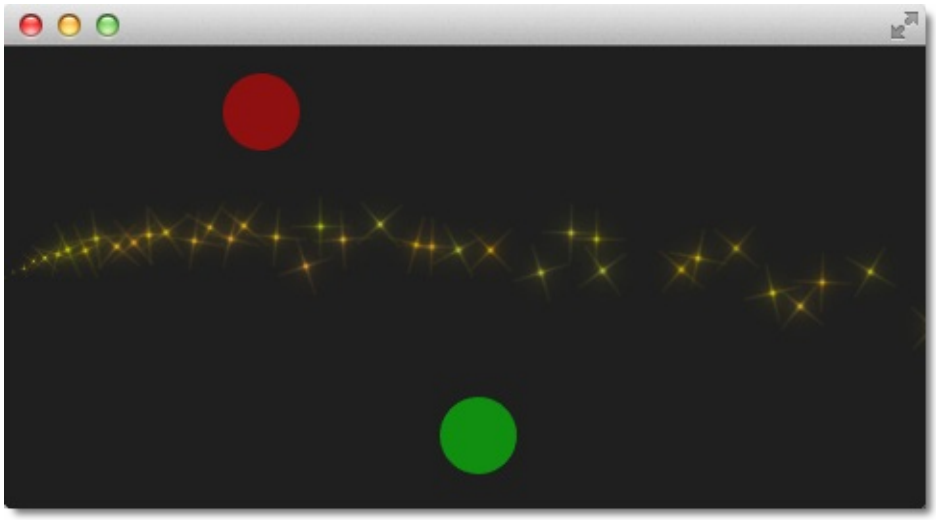
```
velocity: TargetDirection {
    targetX: 100
    targetY: 0
    targetVariation: 100/6
    magnitude: 100
}
```

注意

当你期望发射粒子朝着指定的**x,y**坐标值流动时，目标方向是非常好的方案。

我没有再贴出结果图，因为它与前一个结果相同，取而代之的有一个问题留给你。

在下图的红色和绿色圆指定每个目标项的目标方向速度的加速属性。每个目标方向有相同的参数。那么哪一个负责速度，哪一个负责加速度？



粒子画笔（Particle Painter）

到目前为止我们只使用了基于粒子画笔的图像来实现粒子可视化。Qt也提供了一些其它的粒子画笔：

- 粒子项（ItemParticle）：基于粒子画笔的代理
- 自定义粒子（CustomParticle）：基于粒子画笔的着色器

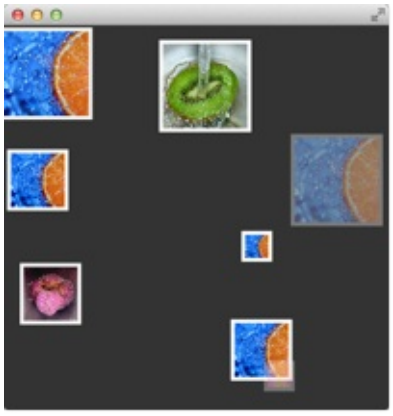
粒子项可以将QML元素项作为粒子发射。你需要制定自己的粒子代理。

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

在这个例子中，我们的代理是一个随机图片（使用Math.random()完成），有着白色边框和随机大小。

```
Component {
    id: itemDelegate
    Rectangle {
        id: container
        width: 32*Math.ceil(Math.random()*3); height: width
        color: 'white'
        Image {
            anchors.fill: parent
            anchors.margins: 4
            source: 'assets/fruits'+Math.ceil(Math.random()*10)+'.jpg'
        }
    }
}
```

每秒发出四个粒子，每个粒子拥有4秒的生命周期。粒子自动淡入淡出。



对于更多的动态情况，也可以由你自己创建一个子项，让粒子系统来控制它，使用take(item, priority)来完成。粒子系统控制你的粒子就像控制普通的粒子一样。你可以使用give(item)来拿回子项的控制权。你也可以操作子项粒子，甚至可以使用freeze(item)来停止它，使用unfreeze(item)来恢复它。

粒子控制（**Affecting Particles**）

粒子由粒子发射器发出。在粒子发射出后，发射器无法再改变粒子。粒子控制器允许你控制发射后的粒子参数。

控制器的每个类型使用不同的方法来影响粒子：

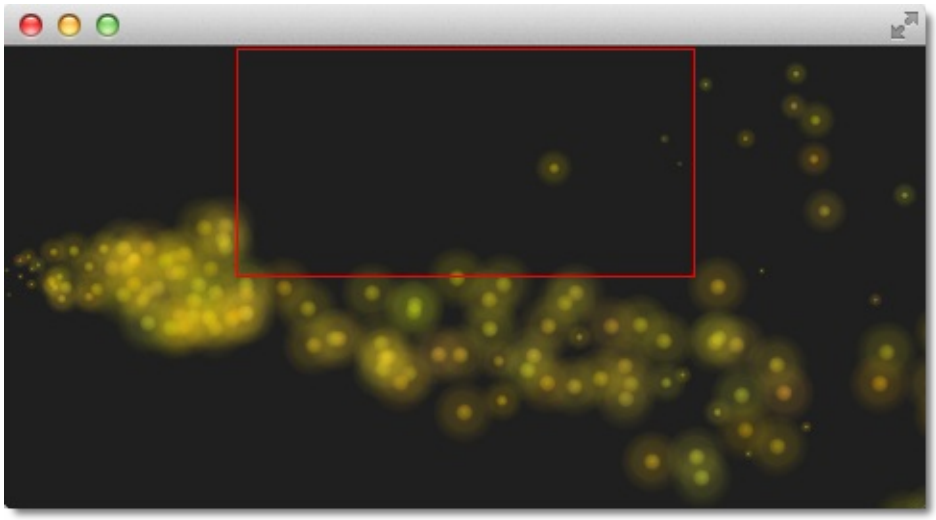
- 生命周期（Age）- 修改粒子的生命周期
- 吸引（Attractor）- 吸引粒子朝向指定点
- 摩擦（Friction）- 按当前粒子速度成正比减慢运动
- 重力（Gravity）- 设置一个角度的加速度
- 紊流（Turbulence）- 强制基于噪声图像方式的流动
- 漂移（Wander）- 随机变化的轨迹
- 组目标（GroupGoal）- 改变一组粒子群的状态
- 子粒子（SpriteGoal）- 改变一个子粒子的状态

生命周期（**Age**）

允许粒子老得更快，lifeLeft属性指定了粒子的有多少的生命周期。

```
    Age {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        advancePosition: true
        lifeLeft: 1200
        once: true
        Tracer {}
    }
```

在这个例子中，当粒子的生命周期达到1200毫秒后，我们将会缩短上方的粒子的生命周期一次。由于我们设置了advancePosition为true，当粒子的生命周期到达1200毫秒后，我们将会再一次在这个位置看到粒子出现。

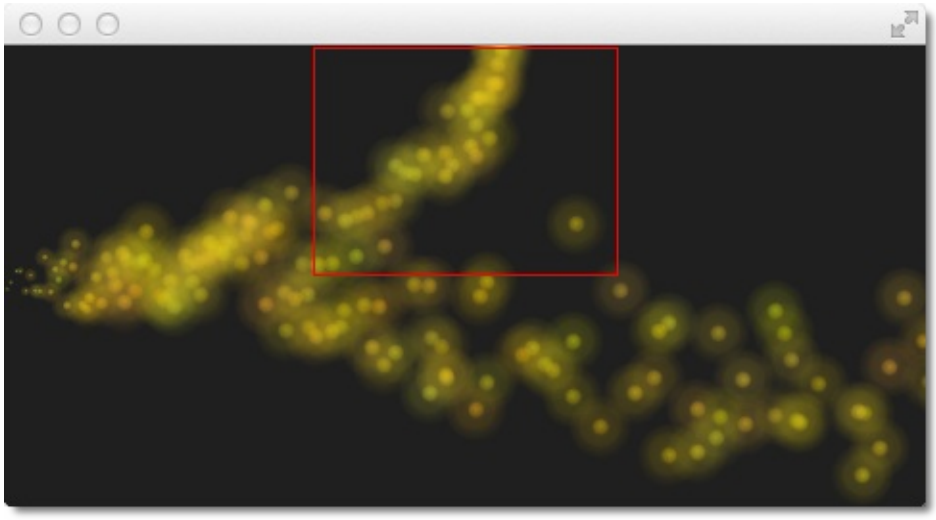


吸引（**Attractor**）

吸引会将粒子朝指定的点上吸引。这个点使用pointX与pointY来指定，它是与吸引区域的几何形状相对的。strength指定了吸引的力度。在我们的例子中，我们让粒子从左向右运动，吸引放在顶部，有一半运动的粒子会穿过吸引区域。控制器只会影响在它们几何形状内的粒子。这种分离让我们可以同步看到正常的流动与受影响的流动。

```
    Attractor {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 160; height: 120
        system: particleSystem
        pointX: 0
        pointY: 0
        strength: 1.0
        Tracer {}
    }
```

很容易看出上半部分粒子受到吸引。吸引点被设置为吸引区域的左上角（0/0点），吸引力为1.0。

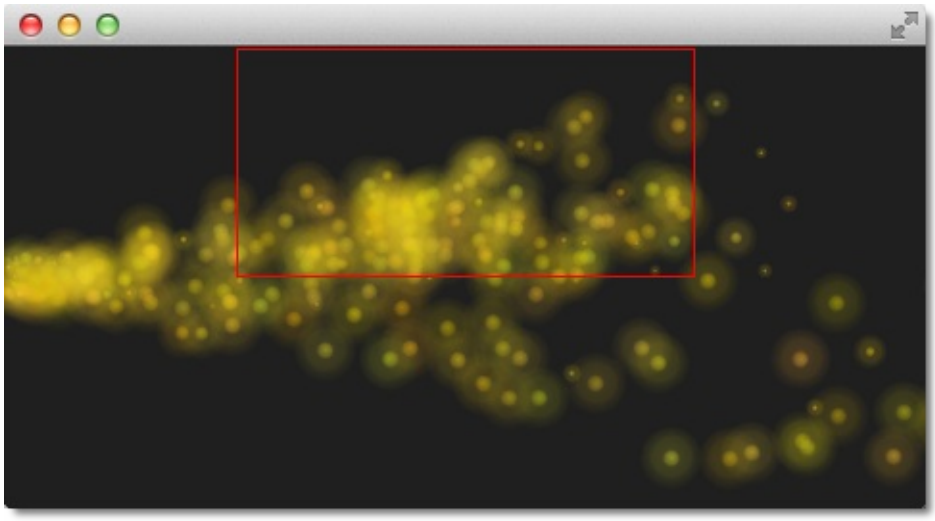


摩擦（**Friction**）

摩擦控制器使用一个参数（factor）减慢粒子运动，直到达到一个阈值。

```
    Friction {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        factor : 0.8
        threshold: 25
        Tracer {}
    }
```

在上部的摩擦区域，粒子被按照0.8的参数（factor）减慢，直到粒子的速度达到25像素每秒。这个阈值像一个过滤器。粒子运动速度高于阈值将会按照给定的参数来减慢它。

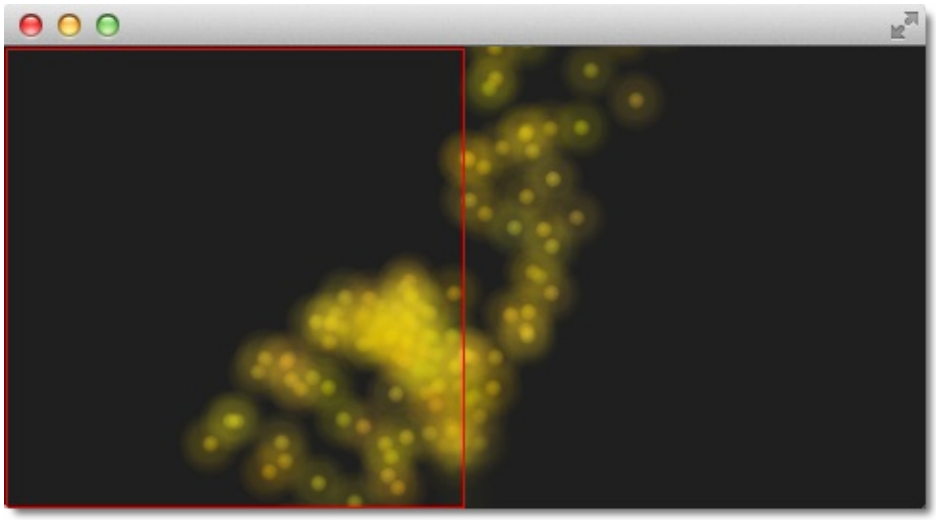


重力（**Gravity**）

重力控制器应用在加速度上，在我们的例子中，我们使用一个角度方向将粒子从底部发射到顶部。右边是为控制区域，左边使用重力控制器控制，重力方向为90度方向（垂直向下），梯度值为50。

```
    Gravity {
        width: 240; height: 240
        system: particleSystem
        magnitude: 50
        angle: 90
        Tracer {}
    }
```

左边的粒子试图爬上去，但是稳定向下的加速度将它们按照重力的方向拖拽下来。

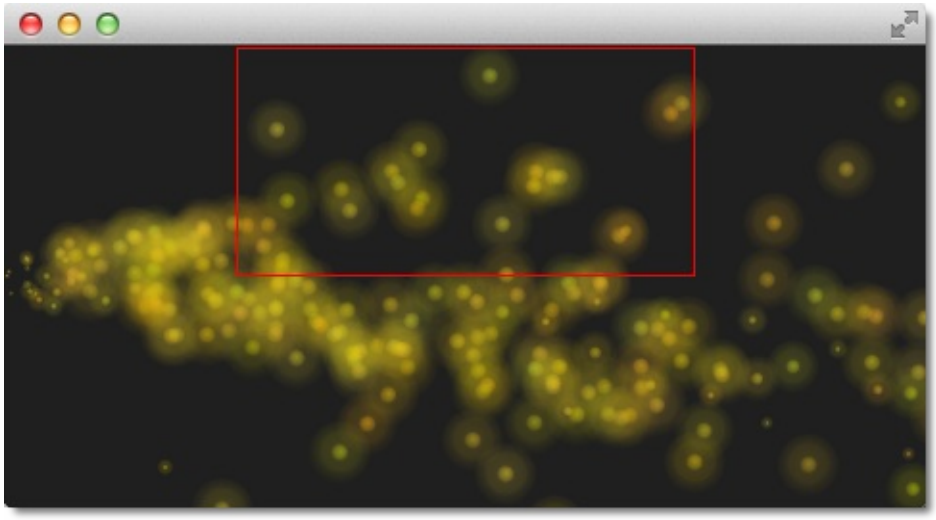


紊流（**Turbulence**）

紊流控制器，对粒子应用了一个混乱映射方向力的矢量。这个混乱映射是由一个噪声图像定义的。可以使用noiseSource属性来定义噪声图像。strength定义了矢量对于粒子运动的影响有多大。

```
    Turbulence {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        strength: 100
        Tracer {}
    }
```

在这个例子中，上部区域被紊流影响。它们的运动看起来是不稳定的。不稳定的粒子偏差值来自原路径定义的strength。

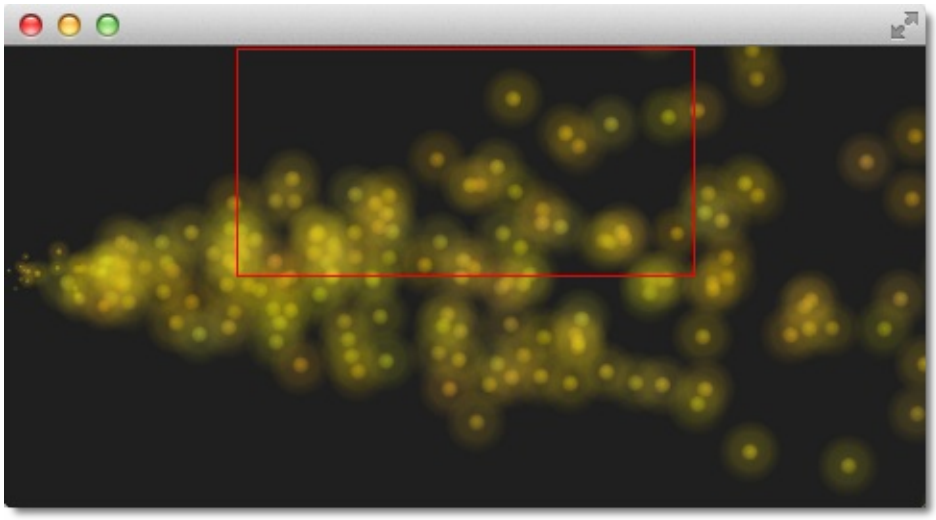


漂移（**Wander**）

漂移控制器控制了轨迹。affectedParameter属性可以指定哪个参数控制了漂移（速度，位置或者加速度）。pace属性制定了每秒最多改变的属性。yVariance指定了y组件对粒子轨迹的影响。

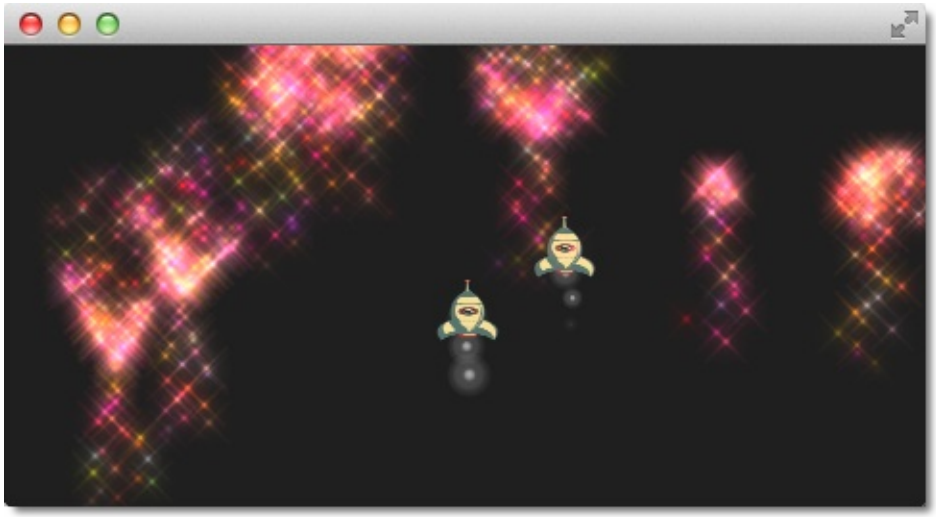
```
    Wander {
        anchors.horizontalCenter: parent.horizontalCenter
        width: 240; height: 120
        system: particleSystem
        affectedParameter: Wander.Position
        pace: 200
        yVariance: 240
        Tracer {}
    }
```

在顶部漂移控制器的粒子被随机的轨迹改变。在这种情境下，每秒改变粒子y方向的位置200次。



粒子组（Particle Group）

在本章开始时，我们已经介绍过粒子组了，默认下，粒子都属于空组（""）。使用GroupGoal控制器可以改变粒子组。为了实现可视化，我们创建了一个烟花示例，火箭进入，在空中爆炸形成壮观的烟火。



这个例子分为两部分。第一部分叫做“发射时间（Launch Time）”连接场景，加入粒子组，第二部分叫做“爆炸烟花（Let there be firework）”，专注于粒子组的变化。

让我们看看这两部分。

发射时间（**Launch Time**）

首先我们创建一个典型的黑色场景：

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false
}
```

tracer使用被用作场景追踪的开关，然后定义我们的粒子系统：

```
ParticleSystem {
    id: particleSystem
}
```

我们添加两种粒子图片画笔（一个用于火箭，一个用于火箭喷射烟雾）：

```
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
}
```

你可以看到在这些画笔定义中，它们使用groups属性来定义粒子的归属。只需要定义一个名字，Qt Quick将会隐式的创建这个分组。

现在我们需要将这些火箭发射到空中。我们在场景底部创建一个粒子发射器，将速度设置为朝上的方向。为了模拟重力，我们设置一个向下的加速度：

```
Emitter {
    id: rocketEmitter
    anchors.bottom: parent.bottom
    width: parent.width; height: 40
    system: particleSystem
    group: 'rocket'
    emitRate: 2
    maximumEmitted: 4
    lifeSpan: 4800
    lifeSpanVariation: 400
    size: 32
    velocity: AngleDirection { angle: 270; magnitude: 150; magnitudeVariation: 10 }
    acceleration: AngleDirection { angle: 90; magnitude: 50 }
    Tracer { color: 'red'; visible: root.tracer }
}
```

发射器属于'rocket'粒子组，与我们的火箭粒子画笔相同。通过粒子组将它们联系在一起。发射器将粒子发射到'rocket'粒子组中，火箭画笔将会绘制它们。

对于烟雾，我们使用一个追踪发射器，它将会跟在火箭的后面。我们定义'smoke'组，并且它会跟在'rocket'粒子组后面：

```
TrailEmitter {
    id: smokeEmitter
    system: particleSystem
    emitHeight: 1
    emitWidth: 4
    group: 'smoke'
    follow: 'rocket'
    emitRatePerParticle: 96
    velocity: AngleDirection { angle: 90; magnitude: 100; angleVariation: 5 }
```

```
    lifeSpan: 200
    size: 16
    sizeVariation: 4
    endSize: 0
}
```

向下模拟从火箭里面喷射出的烟。emitHeight与emitWidth指定了围绕跟随在烟雾粒子发射后的粒子。如果不指定这个值，跟随的粒子将会被拿掉，但是对于这个例子，我们想要提升显示效果，粒子流从一个接近于火箭尾部的中间点发射出。

如果你运行这个例子，你会发现一些火箭正常飞起，一些火箭却飞出场景。这不是我们想要的，我们需要在它们离开场景前让他们慢下来，这里可以使用摩擦控制器来设置一个最小阈值：

```
Friction {
    groups: ['rocket']
    anchors.top: parent.top
    width: parent.width; height: 80
    system: particleSystem
    threshold: 5
    factor: 0.9
}
```

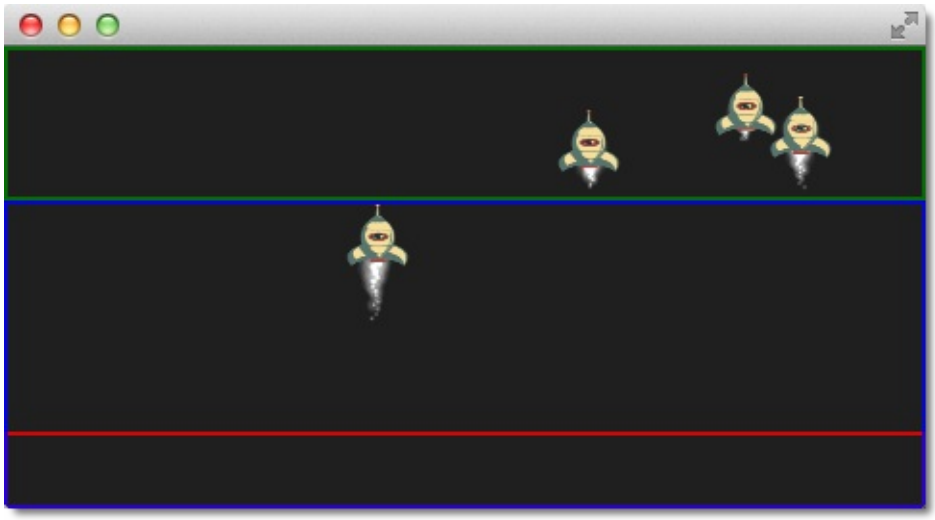
在摩擦控制器中，你也需要定义哪个粒子组受控制器影响。当火箭经过从顶部向下80像素的区域时，所有的火箭将会以0.9的factor减慢（你可以试试100，你会发现它们立即停止了），直到它们的速度达到每秒5个像素。随着火箭粒子向下的加速度继续生效，火箭开始向地面下沉，直到它们的生命周期结束。

由于在空气中向上运动是非常困难的，并且非常不稳定，我们在火箭上升时模拟一些紊流：

```
Turbulence {
    groups: ['rocket']
    anchors.bottom: parent.bottom
    width: parent.width; height: 160
    system: particleSystem
    strength: 25
    Tracer { color: 'green'; visible: root.tracer }
}
```

当然，紊流控制器也需要定义它会影响哪些粒子组。紊流控制器的区域从底部向上160像素（直到摩擦控制器边界上），它们也可以相互覆盖。

当你运行程序时，你可以看到火箭开始上升，然后在摩擦控制器区域开始减速，向下的加速度仍然生效，火箭开始后退。下一步我们开始制作爆炸烟花。



注意

使用**tracers**跟踪区域可以显示场景中的不同区域。火箭粒子发射的红色区域，蓝色区域是紊流控制器区域，最后在绿色的摩擦控制器区域减速，并且再次下降是由于向下的加速度仍然生效。

爆炸烟花（**Let there be fireworks**）

让火箭变成美丽的烟花，我们需要添加一个粒子组来封装这个变化：

```
ParticleGroup {
    name: 'explosion'
    system: particleSystem
}
```

我们使用GroupGoal控制器来改变粒子组。这个组控制器被放置在屏幕中间垂直线附近，它将会影响'rocket'粒子组。使用groupGoal属性，我们设置目标组改变为我们之前定义的'explosion'组：

```
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 }
}
```

jump属性定义了粒子组的变化是立即变化而不是在某个时间段后变化。

注意

在**Qt5**的**alpha**发布版中，粒子组的持续改变无法工作，有好的建议吗？

由于火箭粒子变为我们的爆炸粒子，当火箭粒子进入GroupGoal控制器区域时，我们需要在粒子组中添加一个烟花：

```
// inside particle group
TrailEmitter {
    id: explosionEmitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 750
    emitRatePerParticle: 200
    size: 32
    velocity: AngleDirection { angle: -90; angleVariation: 180; magnitude: 50 }
}
```

爆炸释放粒子到'sparkle'粒子组。我们稍后会定义这个组的粒子画笔。轨迹发射器跟随火箭粒子每秒发射200个火箭爆炸粒子。粒子的方向向上，并改变180度。

由于向'sparkle'粒子组发射粒子，我们需要定义一个粒子画笔用于绘制这个组的粒子：

```
ImageParticle {
    id: sparklePainter
    system: particleSystem
    groups: ['sparkle']
    color: 'red'
    colorVariation: 0.6
    source: "assets/star.png"
    alpha: 0.3
}
```

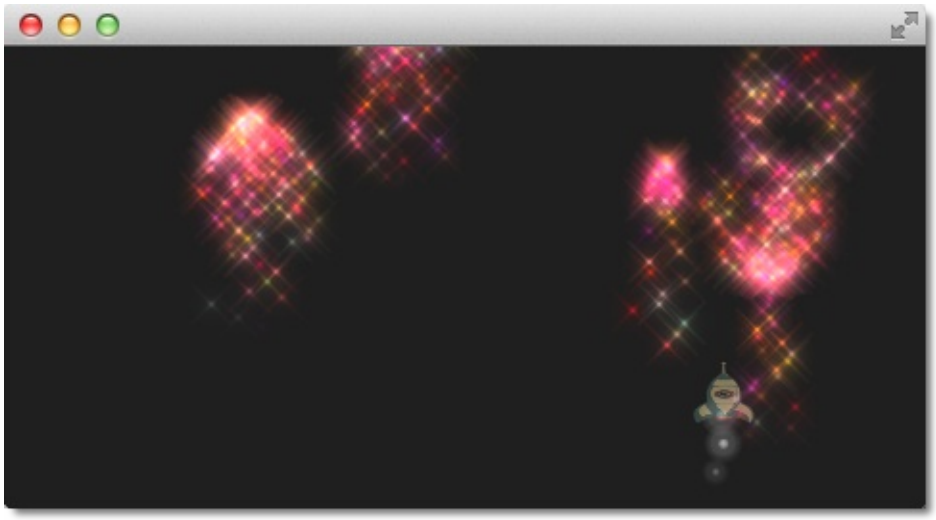
闪烁的烟花是红色的星星，使用接近透明的颜色来渲染出发光的效果。

为了使烟花更加壮观，我们也需要添加给我们的粒子组添加第二个轨迹发射器，它向下发射锥形粒子：

```
// inside particle group
TrailEmitter {
    id: explosion2Emitter
    anchors.fill: parent
    group: 'sparkle'
    follow: 'rocket'
    lifeSpan: 250
    emitRatePerParticle: 100
    size: 32
    velocity: AngleDirection { angle: 90; angleVariation: 15; magnitude: 400 }
}
```

其它的爆炸轨迹发射器与这个设置类似，就这样。

下面是最终结果。



下面是火箭烟花的所有代码。

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {
    id: root
    width: 480; height: 240
    color: "#1F1F1F"
    property bool tracer: false

    ParticleSystem {
        id: particleSystem
    }

    ImageParticle {
        id: smokePainter
        system: particleSystem
        groups: ['smoke']
        source: "assets/particle.png"
        alpha: 0.3
    }

    ImageParticle {
        id: rocketPainter
        system: particleSystem
        groups: ['rocket']
        source: "assets/rocket.png"
        entryEffect: ImageParticle.Fade
    }

    Emitter {
        id: rocketEmitter
        anchors.bottom: parent.bottom
        width: parent.width; height: 40
        system: particleSystem
        group: 'rocket'
        emitRate: 2
        maximumEmitted: 8
        lifeSpan: 4800
        lifeSpanVariation: 400
```

```
        size: 32
        velocity: AngleDirection { angle: 270; magnitude: 150; magnitudeVariation: 10 }
        acceleration: AngleDirection { angle: 90; magnitude: 50 }
        Tracer { color: 'red'; visible: root.tracer }
    }

    TrailEmitter {
        id: smokeEmitter
        system: particleSystem
        group: 'smoke'
        follow: 'rocket'
        size: 16
        sizeVariation: 8
        emitRatePerParticle: 16
        velocity: AngleDirection { angle: 90; magnitude: 100; angleVariation: 15 }
        lifeSpan: 200
        Tracer { color: 'blue'; visible: root.tracer }
    }

    Friction {
        groups: ['rocket']
        anchors.top: parent.top
        width: parent.width; height: 80
        system: particleSystem
        threshold: 5
        factor: 0.9

    }

    Turbulence {
        groups: ['rocket']
        anchors.bottom: parent.bottom
        width: parent.width; height: 160
        system: particleSystem
        strength:25
        Tracer { color: 'green'; visible: root.tracer }
    }


    ImageParticle {
        id: sparklePainter
        system: particleSystem
        groups: ['sparkle']
        color: 'red'
        colorVariation: 0.6
        source: "assets/star.png"
        alpha: 0.3
    }

    GroupGoal {
        id: rocketChanger
        anchors.top: parent.top
        width: parent.width; height: 80
        system: particleSystem
        groups: ['rocket']
        goalState: 'explosion'
        jump: true
        Tracer { color: 'blue'; visible: root.tracer }
    }
```

```
    ParticleGroup {
        name: 'explosion'
        system: particleSystem

        TrailEmitter {
            id: explosionEmitter
            anchors.fill: parent
            group: 'sparkle'
            follow: 'rocket'
            lifeSpan: 750
            emitRatePerParticle: 200
            size: 32
            velocity: AngleDirection { angle: -90; angleVariation: 180; magnitude: 50 }
        }

        TrailEmitter {
            id: explosion2Emitter
            anchors.fill: parent
            group: 'sparkle'
            follow: 'rocket'
            lifeSpan: 250
            emitRatePerParticle: 100
            size: 32
            velocity: AngleDirection { angle: 90; angleVariation: 15; magnitude: 400 }
        }
    }
}
```

总结（Summary）

粒子是一个非常强大且有趣的方法，用来表达图像现象的一种方式，比如烟， 火花，随机可视元素。Qt5的扩展API非常强大，我们仅仅只使用了一些浅显的。有一些元素我们还没有使用过，比如精灵（spirites），尺寸表（size tables），颜色表（color tables）。粒子看起来非常有趣，它在界面上创建引人注目的东西是非常有潜力的。在一个用户界面中使用非常多的粒子效果将会导致用户对它产生这是一个游戏的印象。粒子的真正力量也是用来创建游戏。

Shader Effect

注意

最后一次构建：**2014年1月20日下午18:00**。

这章的源代码能够在**assetts folder**找到。

- http://labs.qt.nokia.com/2012/02/02/qt-graphical-effects-in-qt-labs/
- http://labs.qt.nokia.com/2011/05/03/qml-shadereffectitem-on-qgraphicsview/
- http://qt-project.org/doc/qt-4.8/declarative-shadereffects.html
- http://www.opengl.org/registry/doc/GLSLangSpec.4.20.6.clean.pdf
- http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf
- http://www.lighthouse3d.com/opengl/glsl/
- http://wiki.delphigl.com/index.php/Tutorial_glsl
- Qt5Doc qtquick-shaders

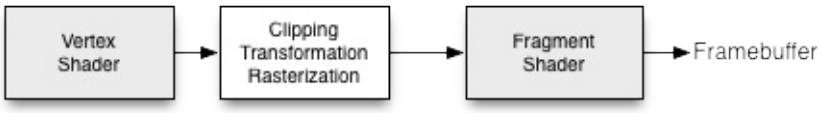
着色器允许我们利用SceneGraph的接口直接调用在强大的GPU上运行的OpenGL来创建渲染效果。着色器使用ShaderEffect与ShaderEffectSource元素来实现。着色器本身的算法使用OpenGL Shading Language（OpenGL着色语言）来实现。

实际上这意味着你需要混合使用QML代码与着色器代码。执行时，会将着色器代码发送到GPU，并在GPU上编译执行。QML着色器元素（Shader QML Elements）允许你与OpenGL着色器程序的属性交互。

让我们首先来看看OpenGL着色器。

OpenGL着色器（OpenGL Shader）

OpenGL的渲染管线分为几个步骤。一个简单的OpenGL渲染管线将包含一个顶点着色器和一个片段着色器。



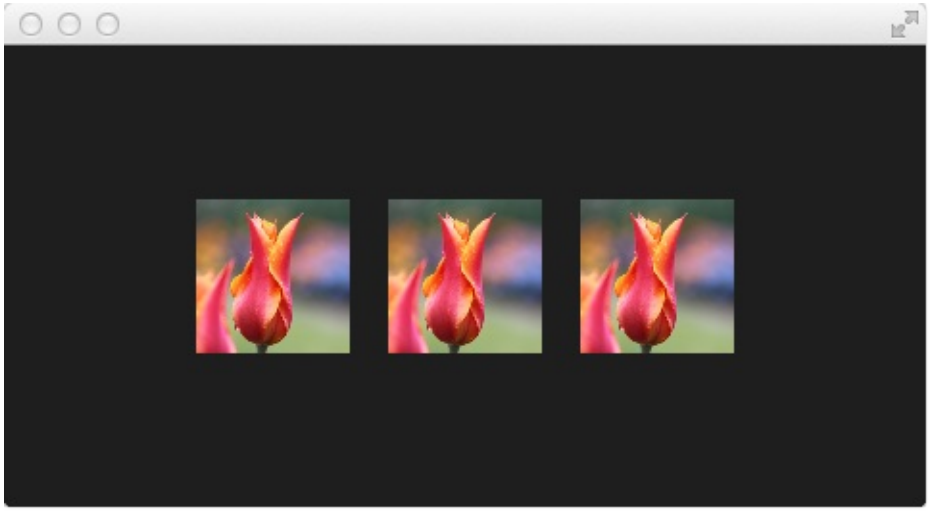
顶点着色器接收顶点数据，并且在程序最后赋值给gl_Position。然后，顶点将会被裁剪，转换和栅格化后作为像素输出。 片段（像素）进入片段着色器，进一步对片段操作并将结果的颜色赋值给gl_FragColor。顶点着色器调用多边形每个角的点（顶点=3D中的点），负责这些点的3D处理。片段（片度=像素）着色器调用每个像素并决定这个像素的颜色。

着色器元素（Shader Elements）

为了对着色器编程，Qt Quick提供了两个元素。ShaderEffectSource与ShaderEffect。ShaderEffect将会使用自定义的着色器，ShaderEffectSource可以将一个QML元素渲染为一个纹理然后再渲染这个纹理。由于ShaderEffect能够应用自定义的着色器到它的矩形几何形状，并且能够使用在着色器中操作资源。一个资源可以是一个图片，它被作为一个纹理或者着色器资源。

默认下着色器使用这个资源并且不作任何改变进行渲染。

```
import QtQuick 2.0

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

    Row {
        anchors.centerIn: parent
        spacing: 20
        Image {
            id: sourceImage
            width: 80; height: width
            source: 'assets/tulips.jpg'
        }
        ShaderEffect {
            id: effect
            width: 80; height: width
            property variant source: sourceImage
        }
        ShaderEffect {
            id: effect2
            width: 80; height: width
            // the source where the effect shall be applied to
            property variant source: sourceImage
            // default vertex shader code
            vertexShader: "
                uniform highp mat4 qt_Matrix;
                attribute highp vec4 qt_Vertex;
                attribute highp vec2 qt_MultiTexCoord0;
                varying highp vec2 qt_TexCoord0;
                void main() {
                    qt_TexCoord0 = qt_MultiTexCoord0;
                    gl_Position = qt_Matrix * qt_Vertex;
                }"
            // default fragment shader code
            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * qt_Opacity;
                }"
        }
    }
}
```



在上边这个例子中，我们在一行中显示了3张图片，第一张是原始图片，第二张使用默认的着色器渲染出来的图片，第三张使用了Qt5源码中默认的顶点与片段着色器的代码进行渲染的图片。

注意

如果你不想看到原始图片，而只想看到被着色器渲染后的图片，你可以设置**Image**为不可见（**visible:false**）。着色器仍然会使用图片数据，但是图像元素（**Image Element**）将不会被渲染。

让我们仔细看看着色器代码。

```
vertexShader: "
    uniform highp mat4 qt_Matrix;
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    varying highp vec2 qt_TexCoord0;
    void main() {
        qt_TexCoord0 = qt_MultiTexCoord0;
        gl_Position = qt_Matrix * qt_Vertex;
    }"
```

着色器代码来自Qt这边的一个字符串，绑定了顶点着色器（vertexShader）与片段着色器（fragmentShader）属性。每个着色器代码必须有一个main(){....}函数，它将被GPU执行。Qt已经默认提供了以qt_开头的变量。

下面是这些变量简短的介绍：

- uniform-在处理过程中不能够改变的值。
- attribute-连接外部数据
- varying-着色器之间的共享数据
- highp-高精度值
- lowp-低精度值

- mat4-4x4浮点数（float）矩阵
- vec2-包含两个浮点数的向量
- sampler2D-2D纹理
- float-浮点数

可以查看OpenGL ES 2.0 API Quick Reference Card获得更多信息。

现在我们可以更好的理解下面这些变量：

- qt_Matrix：model-view-projection（模型-视图-投影）矩阵
- qt_Vertex：当前顶点坐标
- qt_MultiTexCoord0：纹理坐标
- qt_TexCoord0：共享纹理坐标

我们已经有可以使用的投影矩阵（projection matrix），当前顶点与纹理坐标。纹理坐标与作为资源（source）的纹理相关。在main()函数中，我们保存纹理坐标，留在后面的片段着色器中使用。每个顶点着色器都需要赋值给gl_Postion，在这里使用项目矩阵乘以顶点，得到我们3D坐标系中的点。

片段着色器从顶点着色器中接收我们的纹理坐标，这个纹理仍然来自我们的QML资源属性（source property）。在着色器代码与QML之间传递变量是如此的简单。此外我们的透明值，在着色器中也可以使用，变量是qt_Opacity。每 个片段着色器需要给gl_FragColor变量赋值，在这里默认着色器代码使用资源纹理（source texture）的像素颜色与透明值相乘。

```
fragmentShader: "
    varying highp vec2 qt_TexCoord0;
    uniform sampler2D source;
    uniform lowp float qt_Opacity;
    void main() {
        gl_FragColor = texture2D(source, qt_TexCoord0) * qt_Opacity;
    }"
```

在后面的例子中，我们将会展示一些简单的着色器例子。首先我们会集中在片段着色器上，然后在回到顶点着色器上。

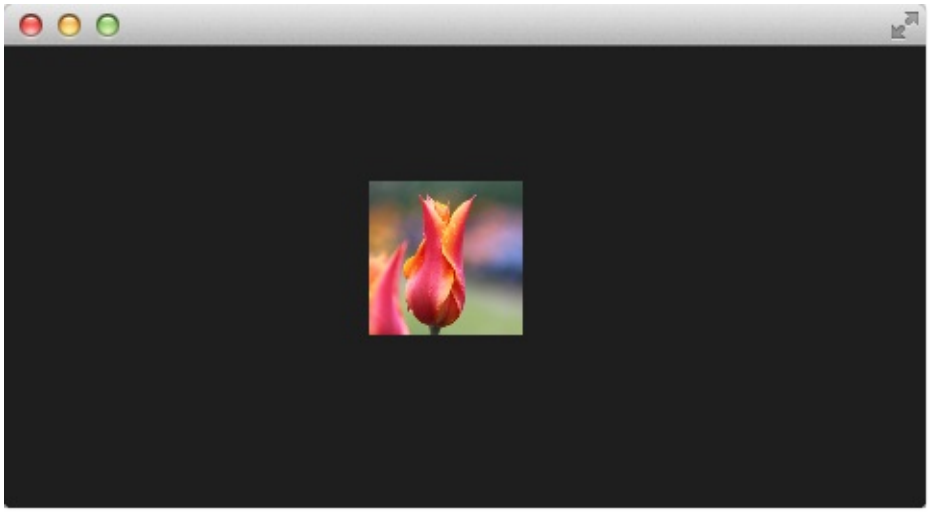
片段着色器（Fragement Shader）

片段着色器调用每个需要渲染的像素。我们将开发一个红色透镜，它将会增加图片的红色通道的值。

配置场景（Setting up the scene）

首先我们配置我们的场景，在区域中央使用一个网格显示我们的源图片（source image）。

```
import QtQuick 2.0

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

    Grid {
        anchors.centerIn: parent
        spacing: 20
        rows: 2; columns: 4
        Image {
            id: sourceImage
            width: 80; height: width
            source: 'assets/tulips.jpg'
        }
    }
}
```



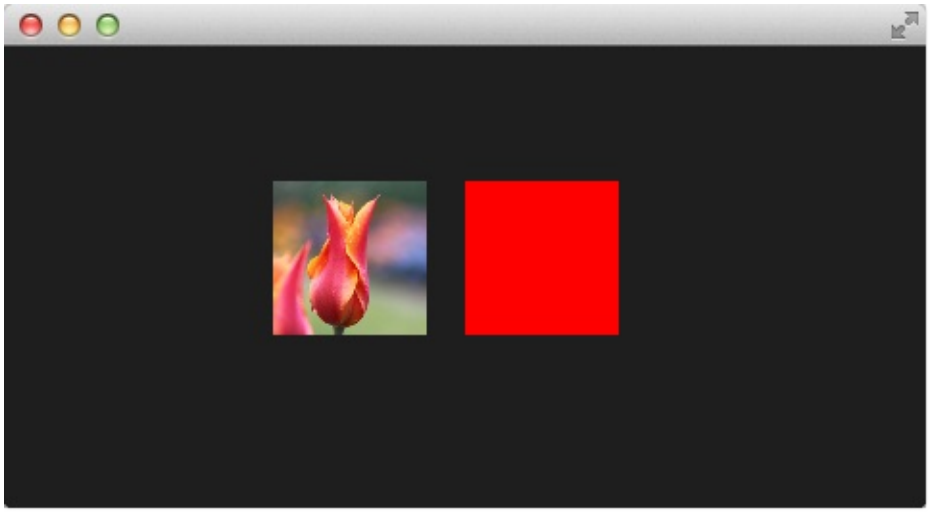
红色着色器（A red Shader）

下一步我们添加一个着色器，显示一个红色矩形框。由于我们不需要纹理，我们从顶点着色器中移除纹理。

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

```
            }"
        fragmentShader: "
            uniform lowp float qt_Opacity;
            void main() {
                gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0) * qt_Opacity;
            }"
```

在片段着色器中，我们简单的给gl_FragColor赋值为vec4(1.0, 0.0, 0.0, 1.0)，它代表红色， 并且不透明（alpha=1.0）。



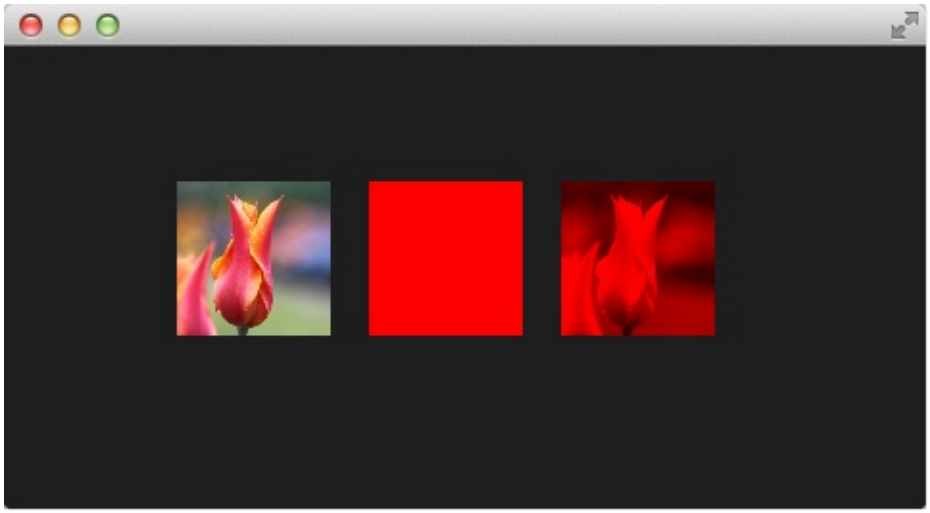
使用纹理的红色着色器（**A red shader with texture**）

现在我们想要将这个红色应用在纹理的每个像素上。我们需要将纹理加回顶点着色器。由于我们不再在顶点着色器中做任何其它的事情，所以默认的顶点着色器已经满足我们的要求。

```
    ShaderEffect {
        id: effect2
        width: 80; height: width
        property variant source: sourceImage
        visible: root.step>1
        fragmentShader: "
            varying highp vec2 qt_TexCoord0;
            uniform sampler2D source;
            uniform lowp float qt_Opacity;
            void main() {
                gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(1.0, 0.0, 0.0,
            }"
    }
```

完整的着色器重新包含我们的源图片作为属性，由于我们没有特殊指定，使用默认的顶点着色器，我没有重写顶点着色器。

在片段着色器中，我们提取纹理片段texture2D(source,qt_TexCoord0)，并且与红色一起应用。

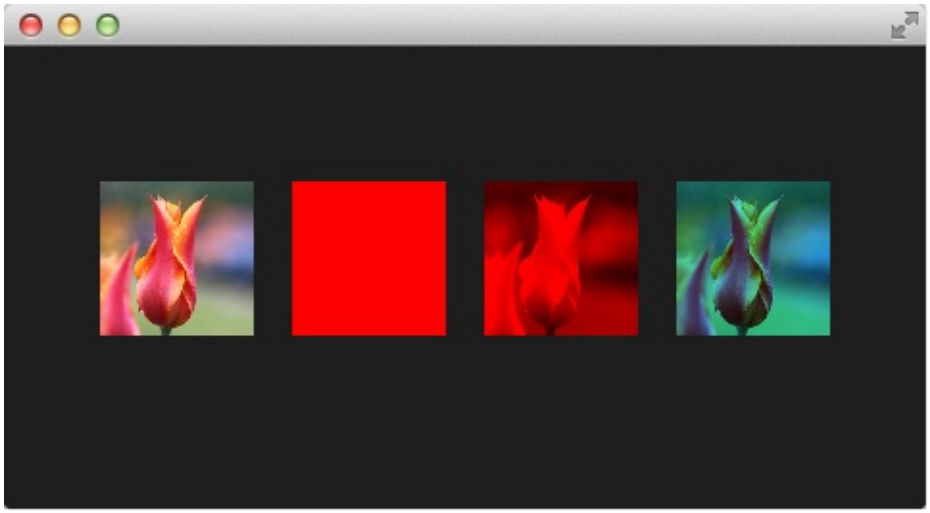


红色通道属性（**The red channel property**）

这样的代码用来修改红色通道的值看起来不是很好，所以我们想要将这个值包含在QML这边。我们在ShaderEffect中增加一个redChannel属性，并在我们的片段着色器中申明一个uniform lowpfloat redChannel。这就是从一个着色器代码中标记一个值到QML这边的方法，非常简单。

```
        ShaderEffect {
            id: effect3
            width: 80; height: width
            property variant source: sourceImage
            property real redChannel: 0.3
            visible: root.step>2
            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                uniform lowp float redChannel;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(redChannel, 1.0
                }"
        }
```

为了让这个透镜更真实，我们改变vec4颜色为vec4(redChannel, 1.0, 1.0, 1.0)，这样其它颜色与1.0相乘，只有红色部分使用我们的redChannel变量。

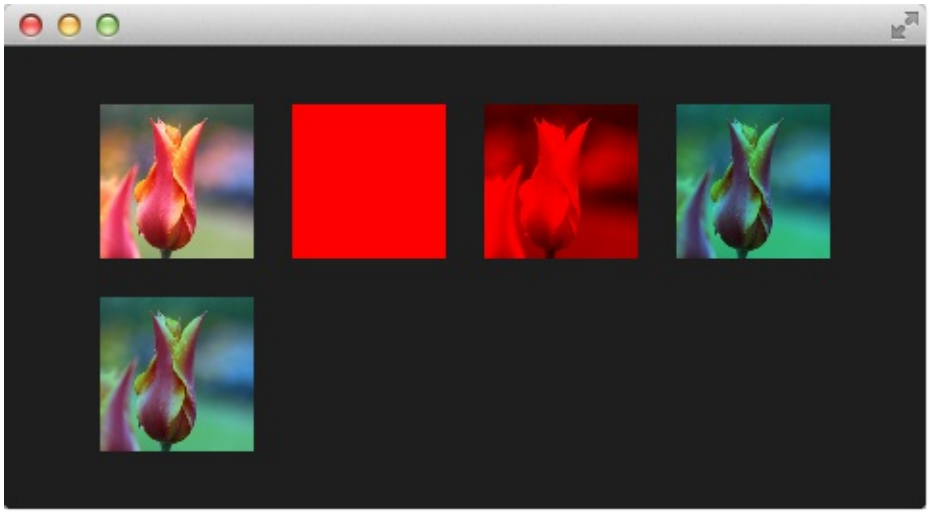


红色通道的动画（**The red channel animated**）

由于redChannel属性仅仅是一个正常的属性，我们也可以像其它QML中的属性一样使用动画。我们使用QML属性在GPU上改变这个值，来影响我们的着色器，这真酷！

```
        ShaderEffect {
            id: effect4
            width: 80; height: width
            property variant source: sourceImage
            property real redChannel: 0.3
            visible: root.step>3
            NumberAnimation on redChannel {
                from: 0.0; to: 1.0; loops: Animation.Infinite; duration: 4000
            }

            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                uniform lowp float redChannel;
                void main() {
                    gl_FragColor = texture2D(source, qt_TexCoord0) * vec4(redChannel, 1.0
                }"
        }
```

下面是最后的结果。



在这4秒内，第二排的着色器红色通道的值从0.0到1.0。图片从没有红色信息（0.0 red）到一个正常的图片（1.0 red）。

波浪效果（Wave Effect）

在这个更加复杂的例子中，我们使用片段着色器创建一个波浪效果。波浪的形成是基于sin曲线，并且它影响了使用的纹理坐标的颜色。

```
import QtQuick 2.0

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

    Row {
        anchors.centerIn: parent
        spacing: 20
        Image {
            id: sourceImage
            width: 160; height: width
            source: "assets/coastline.jpg"
        }
        ShaderEffect {
            width: 160; height: width
            property variant source: sourceImage
            property real frequency: 8
            property real amplitude: 0.1
            property real time: 0.0
            NumberAnimation on time {
                from: 0; to: Math.PI*2; duration: 1000; loops: Animation.Infinite
            }

            fragmentShader: "
                varying highp vec2 qt_TexCoord0;
                uniform sampler2D source;
                uniform lowp float qt_Opacity;
                uniform highp float frequency;
                uniform highp float amplitude;
                uniform highp float time;
                void main() {
                    highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
                    highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -pulse.x)
                    gl_FragColor = texture2D(source, coord) * qt_Opacity;
                }"
        }
    }
}
```

波浪的计算是基于一个脉冲与纹理坐标的操作。我们使用一个基于当前时间与使用的纹理坐标的sin波浪方程式来实现脉冲。

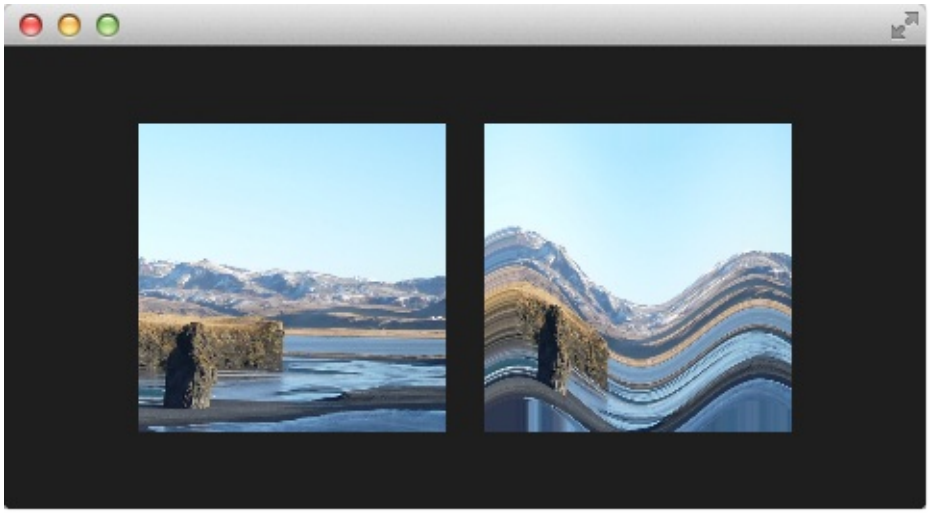
```
highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
```

离开了时间的因素，我们仅仅只有扭曲，而不是像波浪一样运动的扭曲。

我们使用不同的纹理坐标作为颜色。

```
highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -pulse.x);
```

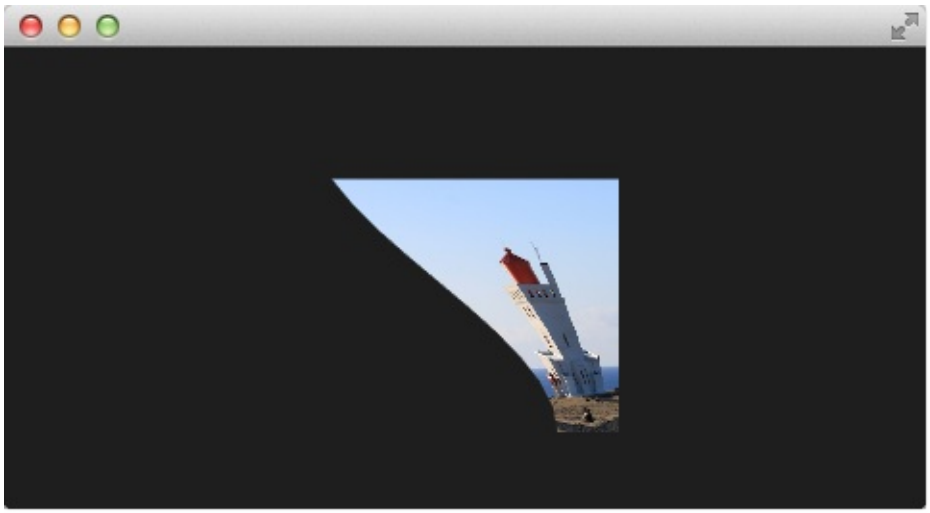
纹理坐标受我们的x脉冲值影响，结果就像一个移动的波浪。



如果我们没有在片段着色器中使用像素的移动，这个效果可以首先考虑使用顶点着色器来完成。

顶点着色器（Vertex Shader）

顶点着色器用来操作ShaderEffect提供的顶点。正常情况下，ShaderEffect有4个顶点（左上top-left，右上top-right，左下bottom-left，右下bottom-right）。每个顶点使用vec4类型记录。为了实现顶点着色器的可视化，我们将编写一个吸收的效果。这个效果通常被用来让一个矩形窗口消失为一个点。



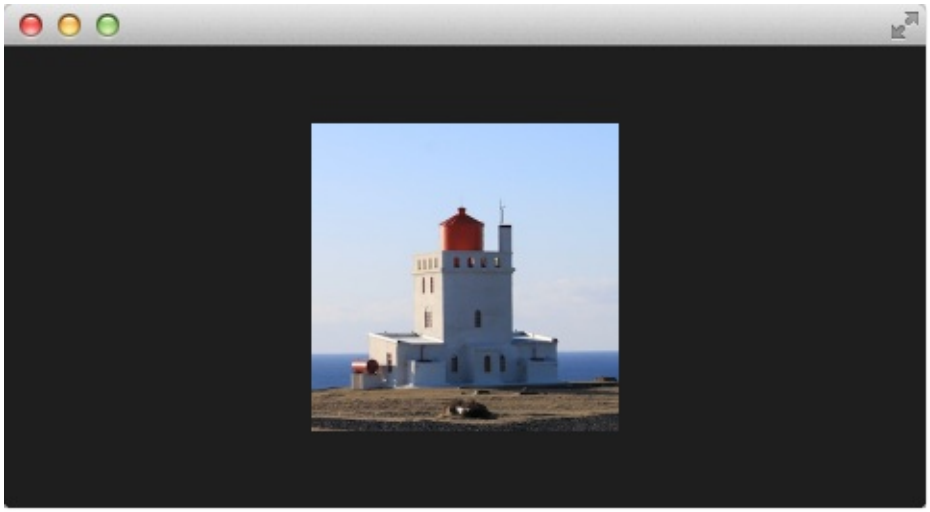
配置场景（**Setting up the scene**）

首先我们再一次配置场景。

```
import QtQuick 2.0

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

    Image {
        id: sourceImage
        width: 160; height: width
        source: "assets/lighthouse.jpg"
        visible: false
    }
    Rectangle {
        width: 160; height: width
        anchors.centerIn: parent
        color: '#333333'
    }
    ShaderEffect {
        id: genieEffect
        width: 160; height: width
        anchors.centerIn: parent
        property variant source: sourceImage
        property bool minimized: false
        MouseArea {
            anchors.fill: parent
            onClicked: genieEffect.minimized = !genieEffect.minimized
        }
    }
```

```
}
```

这个场景使用了一个黑色背景，并且提供了一个使用图片作为资源纹理的ShaderEffect。使用image元素的原图片是不可见的，只是给我们的吸收效果提供资源。此外我们在ShaderEffect的位置添加了一个同样大小的黑色矩形框，这样我们可以更加明确的知道我们需要点击哪里来重置效果。



点击图片将会触发效果，MouseArea覆盖了ShaderEffect。在onClicked操作中，我们绑定了自定义的布尔变量属性minimized。我们稍后使用这个属性来触发效果。

最小化与正常化（**Minimize and normalize**）

在我们配置好场景后，我们定义一个real类型的属性，叫做minimize，这个属性包含了我们当前最小化的值。这个值在0.0到1.0之间，由一个连续的动画来控制它。

```
        property real minimize: 0.0

        SequentialAnimation on minimize {
            id: animMinimize
            running: genieEffect.minimized
            PauseAnimation { duration: 300 }
            NumberAnimation { to: 1; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }

        SequentialAnimation on minimize {
            id: animNormalize
            running: !genieEffect.minimized
            NumberAnimation { to: 0; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }
```

这个动画绑定了由minimized属性触发。现在我们已经配置好我们的环境，最后让我们看看顶点着色器的代码。

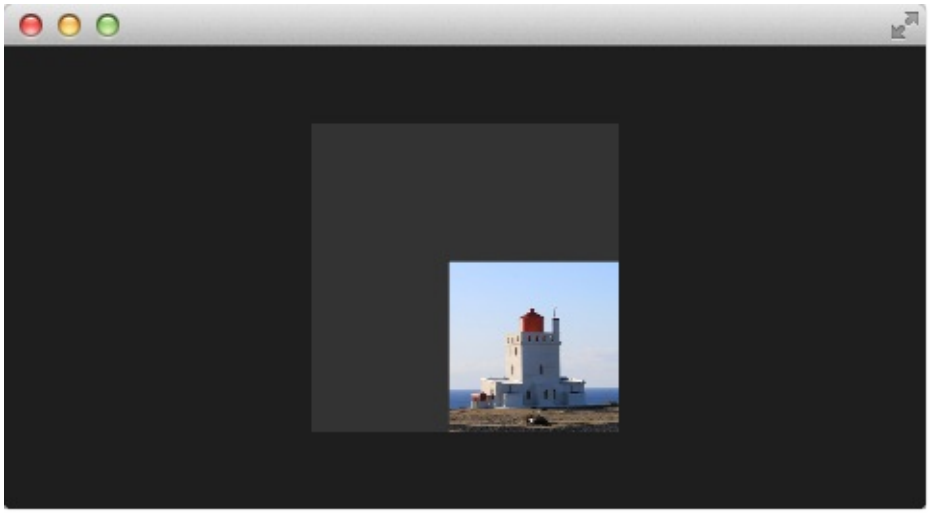
```
        vertexShader: "
            uniform highp mat4 qt_Matrix;
```

```
            attribute highp vec4 qt_Vertex;
            attribute highp vec2 qt_MultiTexCoord0;
            varying highp vec2 qt_TexCoord0;
            uniform highp float minimize;
            uniform highp float width;
            uniform highp float height;
            void main() {
                qt_TexCoord0 = qt_MultiTexCoord0;
                highp vec4 pos = qt_Vertex;
                pos.y = mix(qt_Vertex.y, height, minimize);
                pos.x = mix(qt_Vertex.x, width, minimize);
                gl_Position = qt_Matrix * pos;
            }"
```

顶点着色器被每个顶点调用，在我们这个例子中，一共调用了四次。默认下提供qt已定义的参数，如qt_Matrix，qt_Vertex，qt_MultiTexCoord0，qt_TexCoord0。我们在之前已经讨论过这些变量。此外我们从ShaderEffect中链接minimize，width与height的值到我们的顶点着色器代码中。在main函数中，我们将当前纹理值保存在qt_TexCoord()中，让它在片段着色器中可用。现在我们拷贝当前位置，并修改顶点的x,y的位置。

```
highp vec4 pos = qt_Vertex;
pos.y = mix(qt_Vertex.y, height, minimize);
pos.x = mix(qt_Vertex.x, width, minimize);
```

mix(...)函数提供了一种在两个参数之间（0.0到1.0）的线性插值的算法。在我们的例子中，在当前y值与高度值之间基于minimize的值插值获得y值，x的值获取类似。记住minimize的值是由我们的连续动画控制，并且在0.0到1.0之间（反之亦然）。



这个结果的效果不是真正吸收效果，但是已经能朝着这个目标完成了一大步。

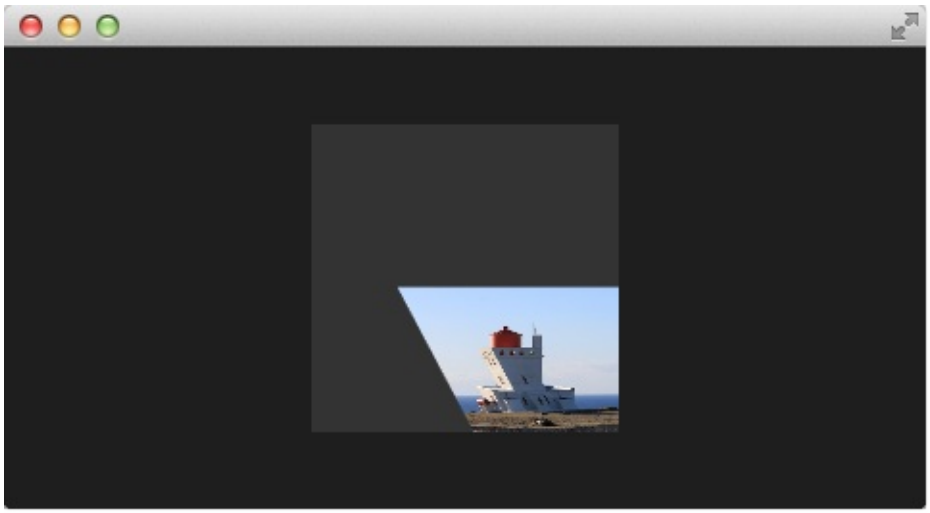
基础弯曲（**Primitive Bending**）

我们已经完成了最小化我们的坐标。现在我们想要修改一下对x值的操作，让它依赖当前的y值。这个改变很简单。y值计算在前。x值的插值基于当前顶点的y坐标。

```
highp float t = pos.y / height;
```

```
pos.x = mix(qt_Vertex.x, width, t * minimize);
```

这个结果造成当y值比较大时，x的位置更靠近width的值。也就是说上面2个顶点根本不受影响，它们的y值始终为0，下面两个顶点的x坐标值更靠近width的值，它们最后转向同一个x值。



```
import QtQuick 2.0

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

    Image {
        id: sourceImage
        width: 160; height: width
        source: "assets/lighthouse.jpg"
        visible: false
    }
    Rectangle {
        width: 160; height: width
        anchors.centerIn: parent
        color: '#333333'
    }
    ShaderEffect {
        id: genieEffect
        width: 160; height: width
        anchors.centerIn: parent
        property variant source: sourceImage
        property real minimize: 0.0
        property bool minimized: false


        SequentialAnimation on minimize {
            id: animMinimize
            running: genieEffect.minimized
            PauseAnimation { duration: 300 }
            NumberAnimation { to: 1; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }

        SequentialAnimation on minimize {
```

```
            id: animNormalize
            running: !genieEffect.minimized
            NumberAnimation { to: 0; duration: 700; easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }


        vertexShader: "
            uniform highp mat4 qt_Matrix;
            uniform highp float minimize;
            uniform highp float height;
            uniform highp float width;
            attribute highp vec4 qt_Vertex;
            attribute highp vec2 qt_MultiTexCoord0;
            varying highp vec2 qt_TexCoord0;
            void main() {
                qt_TexCoord0 = qt_MultiTexCoord0;
                // M1>>
                highp vec4 pos = qt_Vertex;
                pos.y = mix(qt_Vertex.y, height, minimize);
                highp float t = pos.y / height;
                pos.x = mix(qt_Vertex.x, width, t * minimize);
                gl_Position = qt_Matrix * pos;
```

更好的弯曲（**Better Bending**）

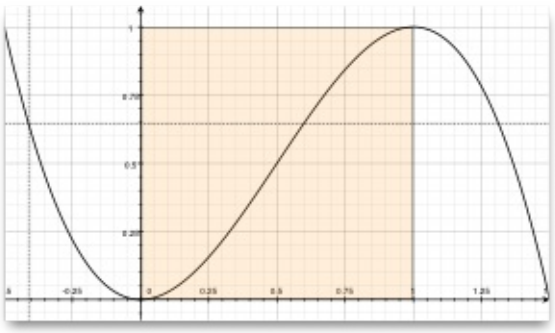
现在简单的弯曲并不能真正的满足我们的要求，我们将添加几个部件来提升它的效果。首先我们增加动画，支持一个自定义的弯曲属性。这是非常必要的，由于弯曲立即发生，y值的最小化需要被推迟。两个动画在同一持续时间计算总和（300+700+100与700+1300）。

```
        property real bend: 0.0
        property bool minimized: false


        // change to parallel animation
        ParallelAnimation {
            id: animMinimize
            running: genieEffect.minimized
            SequentialAnimation {
                PauseAnimation { duration: 300 }
                NumberAnimation {
                    target: genieEffect; property: 'minimize';
                    to: 1; duration: 700;
                    easing.type: Easing.InOutSine
                }
                PauseAnimation { duration: 1000 }
            }
            // adding bend animation
            SequentialAnimation {
                NumberAnimation {
                    target: genieEffect; property: 'bend'
                    to: 1; duration: 700;
                    easing.type: Easing.InOutSine }
                PauseAnimation { duration: 1300 }
            }
        }
```

此外，为了使弯曲更加平滑，不再使用y值影响x值的弯曲函数，pos.x现在依赖新的弯曲属性动画：

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

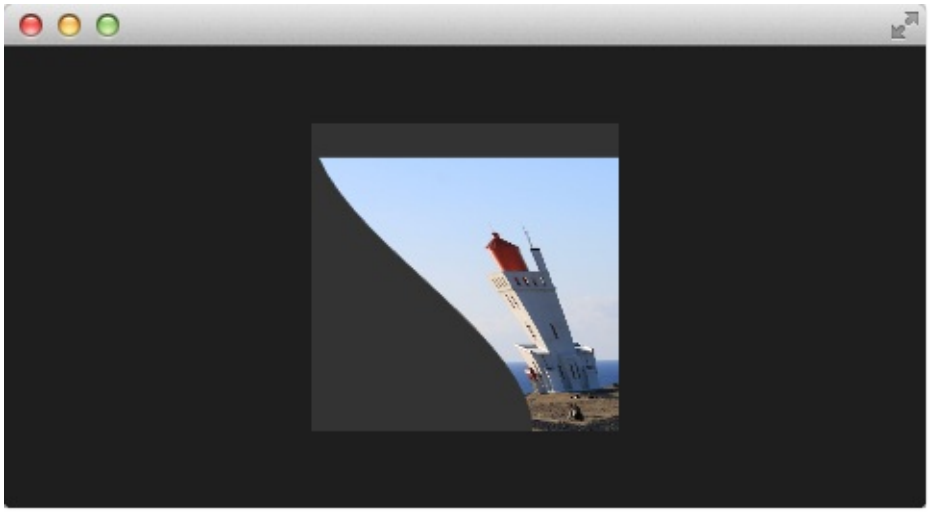
弯曲从0.0平滑开始，逐渐加快，在1.0时逐渐平滑。下面是这个函数在指定范围内的曲线图。对于我们，只需要关注0到1的区间。



想要获得最大化的视觉改变，需要增加我们的顶点数量。可以使用网眼（mesh）来增加顶点：

```
mesh: GridMesh { resolution: Qt.size(16, 16) }
```

现在ShaderEffect被分布为16x16顶点的网格，替换了之前2x2的顶点。这样顶点之间的插值将会看起来更加平滑。

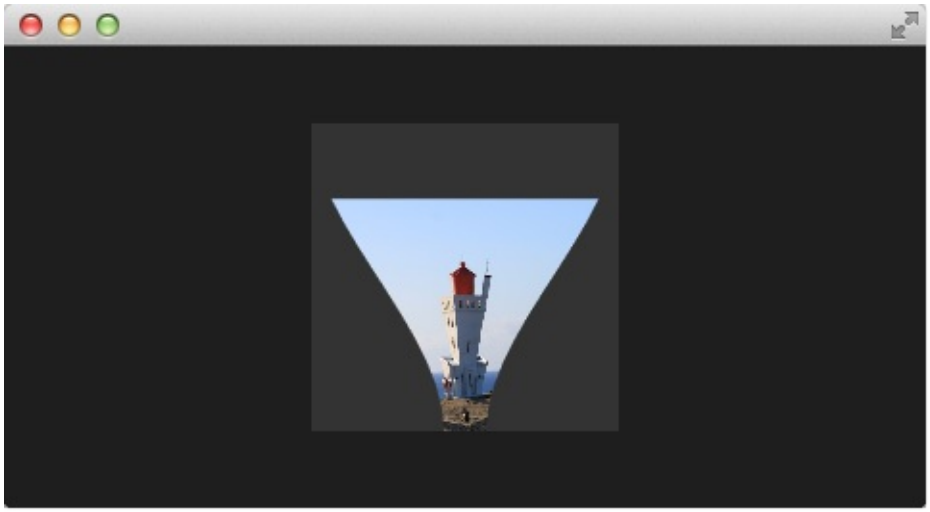


你可以看见曲线的变化，在最后让弯曲变得非常平滑。这让弯曲有了更加强大的效果。

侧面收缩（**Choosing Sides**）

最后一个增强，我们希望能够收缩边界。边界朝着吸收的点消失。直到现在它总是在朝着width值的点消失。添加一个边界属性，我们能够修改这个点在0到width之间。

```
ShaderEffect {
    ...
    property real side: 0.5

    vertexShader: "
        ...
        uniform highp float side;
        ...
        pos.x = mix(qt_Vertex.x, side * width, t * bend);
    "
}
```



包装（**Packing**）

最后将我们的效果包装起来。将我们吸收效果的代码提取到一个叫做GenieEffect的自定义组件中。它使用ShaderEffect作为根元素。移除掉MouseArea，这不应该放在组件中。绑定minimized属性来触发效果。

```
import QtQuick 2.0

ShaderEffect {
    id: genieEffect
    width: 160; height: width
    anchors.centerIn: parent
    property variant source
    mesh: GridMesh { resolution: Qt.size(10, 10) }
    property real minimize: 0.0
    property real bend: 0.0
    property bool minimized: false
    property real side: 1.0


    ParallelAnimation {
        id: animMinimize
        running: genieEffect.minimized
        SequentialAnimation {
            PauseAnimation { duration: 300 }
            NumberAnimation {
                target: genieEffect; property: 'minimize';
                to: 1; duration: 700;
```

```
                easing.type: Easing.InOutSine
            }
            PauseAnimation { duration: 1000 }
        }
        SequentialAnimation {
            NumberAnimation {
                target: genieEffect; property: 'bend'
                to: 1; duration: 700;
                easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1300 }
        }
    }

    ParallelAnimation {
        id: animNormalize
        running: !genieEffect.minimized
        SequentialAnimation {
            NumberAnimation {
                target: genieEffect; property: 'minimize';
                to: 0; duration: 700;
                easing.type: Easing.InOutSine
            }
            PauseAnimation { duration: 1300 }
        }
        SequentialAnimation {
            PauseAnimation { duration: 300 }
            NumberAnimation {
                target: genieEffect; property: 'bend'
                to: 0; duration: 700;
                easing.type: Easing.InOutSine }
            PauseAnimation { duration: 1000 }
        }
    }

    vertexShader: "
        uniform highp mat4 qt_Matrix;
        attribute highp vec4 qt_Vertex;
        attribute highp vec2 qt_MultiTexCoord0;
        uniform highp float height;
        uniform highp float width;
        uniform highp float minimize;
        uniform highp float bend;
        uniform highp float side;
        varying highp vec2 qt_TexCoord0;
        void main() {
            qt_TexCoord0 = qt_MultiTexCoord0;
            highp vec4 pos = qt_Vertex;
            pos.y = mix(qt_Vertex.y, height, minimize);
            highp float t = pos.y / height;
            t = (3.0 - 2.0 * t) * t * t;
            pos.x = mix(qt_Vertex.x, side * width, t * bend);
            gl_Position = qt_Matrix * pos;
        }"
}
```

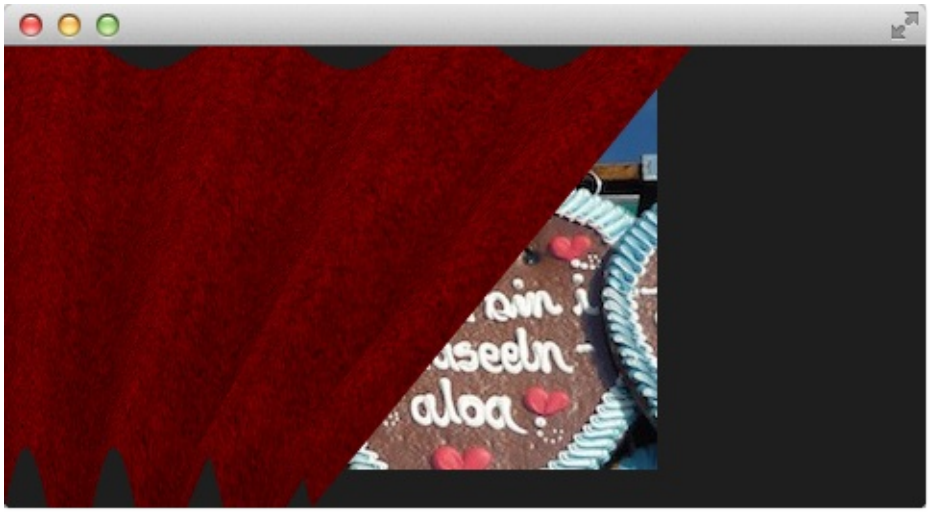
你现在可以像这样简单的使用这个效果：

```
import QtQuick 2.0

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

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

我们简化了代码，移除了背景矩形框，直接使用图片完成效果，替换了在一个单独的图像元素中加载它。

剧幕效果（Curtain Effect）

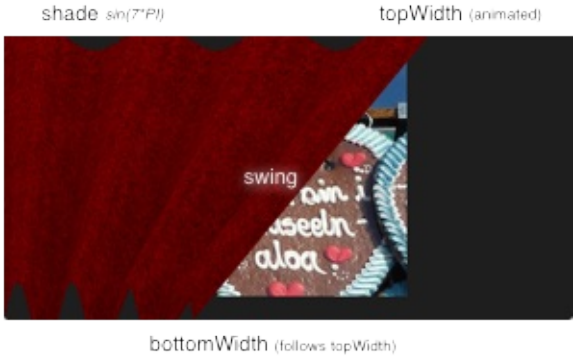
在最后的自定义效果例子中，我们将带来一个剧幕效果。这个效果是2011年5月Qt实验室发布的着色器效果中的一部分。目前网址已经转到blog.qt.digia.com，不知道还能不能找到。



当时我非常喜欢这些效果，剧幕效果是我最喜爱的一个。我喜欢剧幕打开然后遮挡后面的背景对象。

我将代码移植适配到Qt5上，这非常简单。同时我做了一些简化让它能够更好的展示。如果你对整个例子有兴趣，可以访问Qt实验室的博客。

只有一个小组件作为背景，剧幕实际上是一张图片，叫做fabric.jpg，它是ShaderEffect的资源。整个效果使用顶点着色器来摆动剧幕，使用片段着色器提供阴影的效果。下面是一个简单的图片，让你更加容易理解代码。



剧幕的波形阴影通过一个在剧幕宽度上的sin曲线使用7的振幅来计算（7*PI=221.99..）另一个重要的部分是摆动，当剧幕打开或者关闭时，使用动画来播放剧幕的topWidth。bottomWidth使用SpringAnimation来跟随topWidth变化。这样我们就能创建出底部摆动的剧幕效果。计算得到的swing提供了摇摆的强度，用来对顶点的y值进行插值。

剧幕效果放在CurtainEffect.qml组件中，fabric图像作为纹理资源。在阴影的使用上没有新的东西加入，唯一不同的是在顶点着色器中操作gl_Postion和片段着色器中操作gl_FragColor。

```
import QtQuick 2.0
```

```
ShaderEffect {
    anchors.fill: parent

    mesh: GridMesh {
        resolution: Qt.size(50, 50)
    }

    property real topWidth: open?width:20
    property real bottomWidth: topWidth
    property real amplitude: 0.1
    property bool open: false
    property variant source: effectSource

    Behavior on bottomWidth {
        SpringAnimation {
            easing.type: Easing.OutElastic;
            velocity: 250; mass: 1.5;
            spring: 0.5; damping: 0.05
        }
    }

    Behavior on topWidth {
        NumberAnimation { duration: 1000 }
    }


    ShaderEffectSource {
        id: effectSource
        sourceItem: effectImage;
        hideSource: true
    }

    Image {
        id: effectImage
        anchors.fill: parent
        source: "assets/fabric.jpg"
        fillMode: Image.Tile
    }

    vertexShader: "
        attribute highp vec4 qt_Vertex;
        attribute highp vec2 qt_MultiTexCoord0;
        uniform highp mat4 qt_Matrix;
        varying highp vec2 qt_TexCoord0;
        varying lowp float shade;

        uniform highp float topWidth;
        uniform highp float bottomWidth;
        uniform highp float width;
        uniform highp float height;
        uniform highp float amplitude;

        void main() {
            qt_TexCoord0 = qt_MultiTexCoord0;

            highp vec4 shift = vec4(0.0, 0.0, 0.0, 0.0);
            highp float swing = (topWidth - bottomWidth) * (qt_Vertex.y / height);
            shift.x = qt_Vertex.x * (width - topWidth + swing) / width;
```

```
            shade = sin(21.9911486 * qt_Vertex.x / width);
            shift.y = amplitude * (width - topWidth + swing) * shade;

            gl_Position = qt_Matrix * (qt_Vertex - shift);

            shade = 0.2 * (2.0 - shade ) * ((width - topWidth + swing) / width);
        }"

    fragmentShader: "
        uniform sampler2D source;
        varying highp vec2 qt_TexCoord0;
        varying lowp float shade;
        void main() {
            highp vec4 color = texture2D(source, qt_TexCoord0);
            color.rgb *= 1.0 - shade;
            gl_FragColor = color;
        }"
}
```

这个效果在curtaindemo.qml文件中使用。

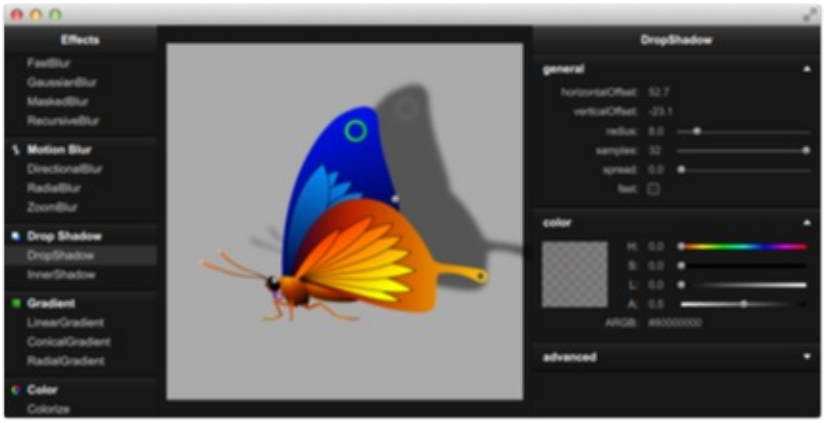
```
import QtQuick 2.0

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

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

    CurtainEffect {
        id: curtain
        anchors.fill: parent
    }

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

剧幕效果通过自定义的open属性打开。我们使用了一个MouseArea来触发打开和关闭剧幕。

Qt图像效果库（Qt GraphicsEffect Library）

图像效果库是一个着色器效果的集合，是由Qt开发者提供制作的。它是一个很好的工具，你可以将它应用在你的程序中，它也是一个学习如何创建着色器的例子。

图像效果库附带了一个手动测试平台，这个工具可以帮助你测试发现不同的效果 测试工具在$QTDIR/qtgraphicaleffects/tests/manual/testbed下。



效果库包含了大约20种效果，下面是效果列表和一些简短的描述。

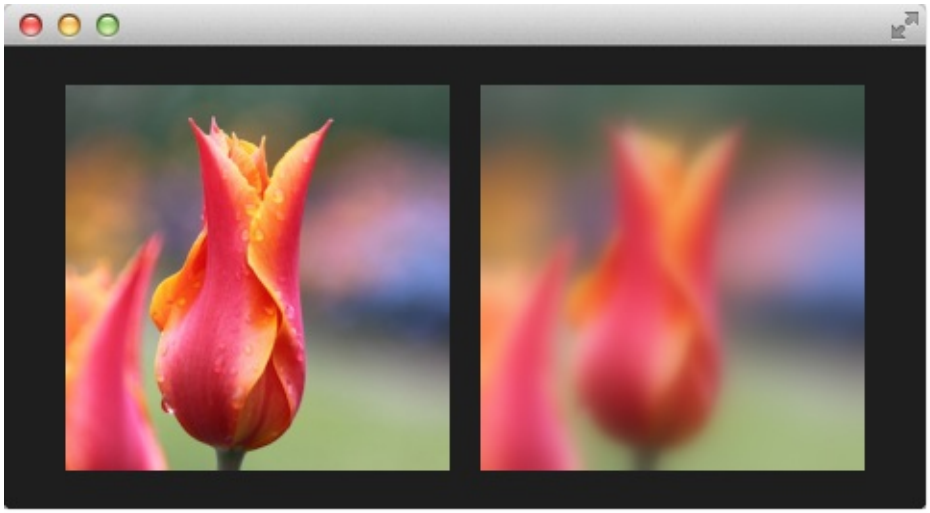
种类	效果	描述
混合（Blend）	混合（Blend）	使用混合模式合并两个资源项
颜色（Color）	亮度与对比度（BrightnessContrast）	调整亮度与对比度
	着色（Colorize）	设置HSL颜色空间颜色
	颜色叠加（ColorOverlay）	应用一个颜色层
	降低饱和度（Desaturate）	减少颜色饱和度
	伽马调整（GammaAdjust）	调整发光度
	色调饱和度（HueSaturation）	调整HSL颜色空间颜色
	色阶调整（LevelAdjust）	调整RGB颜色空间颜色
渐变（Gradient）	圆锥渐变（ConicalGradient）	绘制一个圆锥渐变
	线性渐变（LinearGradient）	绘制一个线性渐变
	射线渐变（RadialGradient）	绘制一个射线渐变
失真（Distortion）	置换（Displace）	按照指定的置换源移动源项的像素
阴影（Drop Shadow）	阴影 （DropShadow）	绘制一个阴影
	内阴影（InnerShadow）	绘制一个内阴影
模糊 （Blur）	快速模糊（FastBlur）	应用一个快速模糊效果
	高斯模糊（GaussianBlur）	应用一个高质量模糊效果
	蒙版模糊（MaskedBlur）	应用一个多种强度的模糊效果
	递归模糊（RecursiveBlur）	重复模糊，提供一个更强的模糊效果
运动模糊（Motion Blur）	方向模糊（DirectionalBlur）	应用一个方向的运动模糊效果

	放射模糊（RadialBlur）	应用一个放射运动模糊效果
	变焦模糊（ZoomBlur）	应用一个变焦运动模糊效果
发光（Glow）	发光（Glow）	绘制一个外发光效果
	矩形发光（RectangularGlow）	绘制一个矩形外发光效果
蒙版（Mask）	透明蒙版（OpacityMask）	使用一个源项遮挡另一个源项
	阈值蒙版（ThresholdMask）	使用一个阈值，一个源项遮挡另一个源项

下面是一个使用快速模糊效果的例子：

```
import QtQuick 2.0
import QtGraphicalEffects 1.0

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

    Row {
        anchors.centerIn: parent
        spacing: 16

        Image {
            id: sourceImage
            source: "assets/tulips.jpg"
            width: 200; height: width
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
        }

        FastBlur {
            width: 200; height: width
            source: sourceImage
            radius: blurred?32:0
            property bool blurred: false

            Behavior on radius {
                NumberAnimation { duration: 1000 }
            }

            MouseArea {
                id: area
                anchors.fill: parent
                onClicked: parent.blurred = !parent.blurred
            }
        }
    }
}
```

左边是原图片。点击右边的图片将会触发blurred属性，模糊在1秒内从0到32。左边显示模糊后的图片。



Multimedia

在QtMultimedia模块中的multimedia元素可以播放和记录媒体资源，例如声音，视频，或者图片。解码和编码的操作由特定的后台完成。例如在Linux上的gstreamer框架，Windows上的DirectShow，和OS X上的QuickTime。 multimedia元素不是QtQuick核心的接口。它的接口通过导入QtMultimedia 5.0来加入，如下所示：

```
import QtMultimedia 5.0
```

媒体播放（Playing Media）

在QML应用程序中，最基本的媒体应用是播放媒体。使用MediaPlayer元素可以完成它，如果源是一个图片或者视频，可以选择结合VideoOutput元素。MediaPlayer元素有一个source属性指向需要播放的媒体。当媒体源被绑定后，简单的调用play函数就可以开始播放。

如果你想播放一个可视化的媒体，例如图片或者视频等，你需要配置一个VideoOutput元素。MediaPlayer播放通过source属性与视频输出绑定。

在下面的例子中，给MediaPlayer元素一个视频文件作为source。一个VideoOutput被创建和绑定到媒体播放器上。一旦主要部件完全初始化，例如在Component.onCompleted中，播放器的play函数被调用。

```
import QtQuick 2.0
import QtMultimedia 5.0
import QtSystemInfo 5.0

Item {
    width: 1024
    height: 600

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

    VideoOutput {
        anchors.fill: parent
        source: player
    }

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

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

除了上面介绍的视频播放，这个例子也包括了一小段代码用于禁止屏幕保护。这将阻止视频被中断。通过设置ScreenSaver元素的screenSaverEnabled属性为false来完成。通过导入QtSystemInfo 5.0可以使用ScreenSaver元素。

基础操作例如当播放媒体时可以通过MediaPlayer元素的volume属性来控制音量。还有一些其它有用的属性。例如，duration与position属性可以用来创建一个进度条。如果seekable属性为true，当拨动进度条时可以更新position属性。下面这个例子展示了在上面的例子基础上如何添加基础播放。

```
    Rectangle {
        id: progressBar
```

```
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        anchors.margins: 100

        height: 30

        color: "lightGray"

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

            width: player.duration>0?parent.width*player.position/player.duration:0

            color: "darkGray"
        }

        MouseArea {
            anchors.fill: parent

            onClicked: {
                if (player.seekable)
                    player.position = player.duration * mouse.x/width;
            }
        }
    }
```

默认情况下position属性每秒更新一次。这意味着进度条将只会在大跨度下的时间周期下才会更新，需要媒体持续时间足够长，进度条像素足够宽。然而，这个可以通过mediaObject属性的notifyInterval属性改变。它可以设置每个position之间更新的毫秒数，增加用户界面的平滑度。

```
    Connections {
        target: player
        onMediaObjectChanged: {
            if (player.mediaObject)
                player.mediaObject.notifyInterval = 50;
        }
    }
```

当使用MediaPlayer创建一个媒体播放器时，最好使用status属性来监听播放器。这个属性是一个枚举，它枚举了播放器可能出现的状态，从MediaPlayer.Buffered到MediaPlayer.InvalidMedia。下面是这些状态值的总结：

- MediaPlayer.UnknownStatus - 未知状态
- MediaPlayer.NoMedia - 播放器没有指定媒体资源，播放停止
- MediaPlayer.Loading - 播放器正在加载媒体
- MediaPlayer.Loaded - 媒体已经加载完毕，播放停止

- MediaPlayer.Stalled - 加载媒体已经停止
- MediaPlayer.Buffering - 媒体正在缓冲
- MediaPlayer.Buffered - 媒体缓冲完成
- MediaPlayer.EndOfMedia - 媒体播放完毕，播放停止
- MediaPlayer.InvalidMedia - 无法播放媒体，播放停止

正如上面提到的这些枚举项，播放状态会随着时间变化。调用play，pause或者stop将会切换状态，但由于媒体的原因也会影响这些状态。例如，媒体播放完毕，它将会无效，导致播放停止。当前的播放状态可以使用playbackState属性跟踪。这个值可能是MediaPlayer.PlayingState，MediaPlayer.PasuedState或者MediaPlayer.StoppedState。

使用autoPlay属性，MediaPlayer在source属性改变时将会尝试进入播放状态。类似的属性autoLoad将会导致播放器在source属性改变时尝试加载媒体。默认下autoLoad是被允许的。

当然也可以让MediaPlayer循环播放一个媒体项。loops属性控制source将会被重复播放多少次。设置属性为MediaPlayer.Infinite将会导致不停的重播。非常适合持续的动画或者一个重复的背景音乐。

声音效果（Sounds Effects）

当播放声音效果时，从请求播放到真实响应播放的响应时间非常重要。在这种情况下，SoundEffect元素将会派上用场。设置source属性，一个简单调用play函数会直接开始播放。

当敲击屏幕时，可以使用它来完成音效反馈，如下所示：

```
SoundEffect {
    id: beep
    source: "beep.wav"
}

Rectangle {
    id: button

    anchors.centerIn: parent

    width: 200
    height: 100

    color: "red"

    MouseArea {
        anchors.fill: parent
        onClicked: beep.play()
    }
}
```

这个元素也可以用来完成一个配有音效的转换。为了从转换触发，使用ScriptAction元素。

```
SoundEffect {
    id: swosh
    source: "swosh.wav"
}

transitions: [
    Transition {
        ParallelAnimation {
            ScriptAction { script: swosh.play(); }
            PropertyAnimation { properties: "rotation"; duration: 200; }
        }
    }
]
```

除了调用play函数，在MediaPlayer中类似属性也可以使用。比如volume和loops。loops可以设置为SoundEffect.Infinite来提供无限重复播放。停止播放调用stop函数。

注意

当后台使用**PulseAudio**时，**stop**将不会立即停止，但会阻止继续循环。这是由于底层**API**的限制造成的。

视频流（Video Streams）

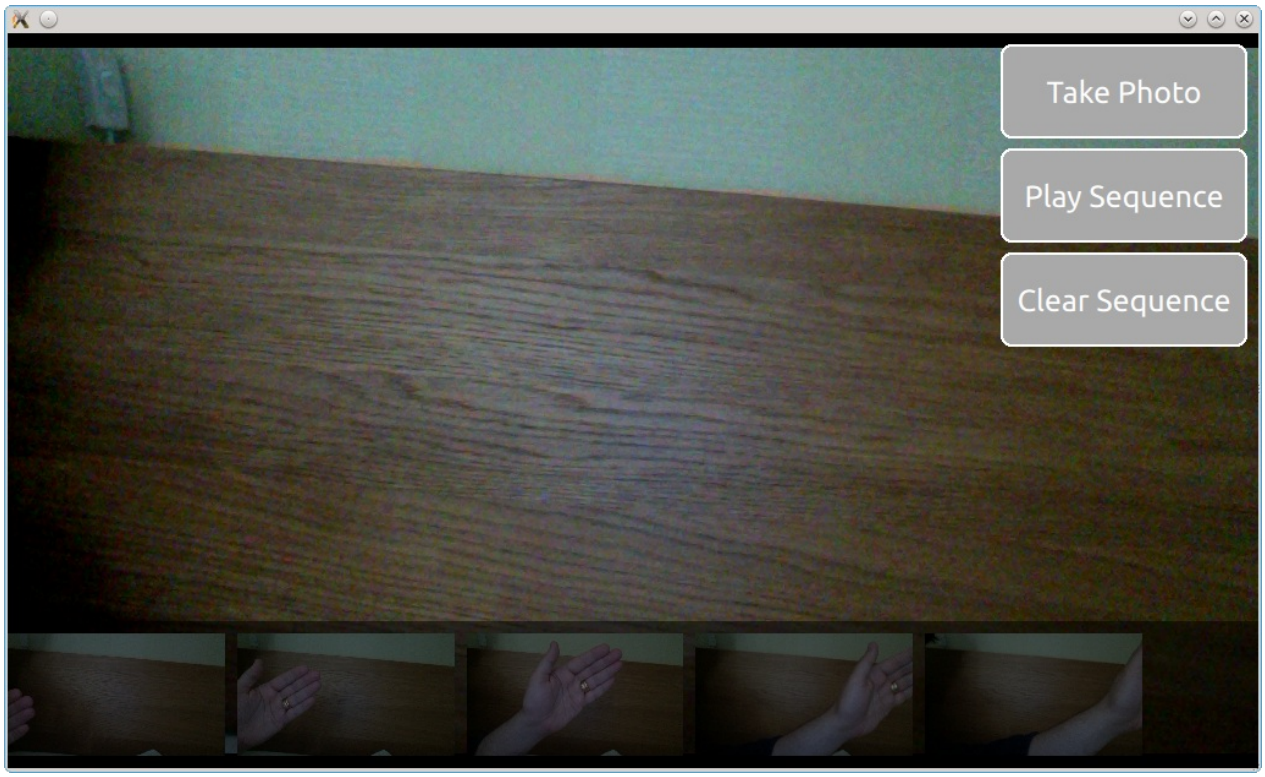
VideoOutput元素不被限制与MediaPlayer元素绑定使用的。它也可以直接用来加载实时视频资源显示一个流媒体。应用程序使用Camera元素作为资源。来自Camera的视频流给用户提供了一个实时流媒体。

```
import QtQuick 2.0
import QtMultimedia 5.0

Item {
    width: 1024
    height: 600

    VideoOutput {
        anchors.fill: parent
        source: camera
    }

    Camera {
        id: camera
    }
}
```

捕捉图像（Capturing Images）

Camera元素一个关键特性就是可以用来拍照。我们将在一个简单的定格动画程序中使用到它。在这章中，你将学习如何显示一个视图查找器，截图和追踪拍摄的图片。

用户界面如下所示。它由三部分组成，背景是一个视图查找器，右边有一列按钮，底部有一连串拍摄的图片。我们想要拍摄一系列的图片，然后点击Play Sequence按钮。这将回放图片，并创建一个简单的定格电影。



相机的视图查找器部分是在VideoOutput中使用一个简单的Camera元素作为资源。这将给用户显示一个来自相机的流媒体视频。

```
    VideoOutput {
        anchors.fill: parent
        source: camera
    }

    Camera {
        id: camera
    }
```

使用一个水平放置的ListView显示来自ListModel的图片，这个部件叫做imagePaths。在背景中使用一个半透明的Rectangle。

```
    ListModel {
        id: imagePaths
    }
```

```
    ListView {
        id: listView

        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        anchors.bottomMargin: 10

        height: 100

        orientation: ListView.Horizontal
        spacing: 10

        model: imagePaths

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

        Rectangle {
            anchors.fill: parent
            anchors.topMargin: -10

            color: "black"
            opacity: 0.5
        }
    }
```

为了拍摄图像，你需要知道Camera元素包含了一组子对象用来完成各种工作。使用Camera.imageCapture用来捕捉图像。当你调用capture方法时，一张图片就被拍摄下来了。Camera.imageCapture的结果将会发送imageCaptured信号，接着发送imageSaved信号。

```
        Button {
            id: shotButton

            width: 200
            height: 75

            text: "Take Photo"
            onClicked: {
                camera.imageCapture.capture();
            }
        }
```

为了拦截子元素的信号，需要一个Connections元素。在这个例子中，我们不需要显示预览图片，仅仅只是将结果图片加入底部的ListView中。就如下面的例子展示的一样，图片保存的路径由信号的path参数提供。

```
    Connections {
        target: camera.imageCapture

        onImageSaved: {
            imagePaths.append({"path": path})
            listView.positionViewAtEnd();
        }
```

```
    }
```

为了显示预览，连接imageCaptured信号，并且使用preview信号参数作为Image元素的source。requestId信号参数与imageCaptured和imageSaved一起发送。这个值由capture方法返回。这样，就可以完整的跟踪拍摄的图片了。预览的图片首先被使用，然后替换为保存的图片。然而在这个例子中我们不需要这样做。

最后是自动回放的部分。使用Timer元素来驱动它，并且加上一些JavaScript。_imageIndex变量被用来跟踪当前显示的图片。当最后一张图片被显示时，回放停止。在例子中，当播放序列时，root.state被用来隐藏用户界面。

```
    property int _imageIndex: -1

    function startPlayback()
    {
        root.state = "playing";
        setImageIndex(0);
        playTimer.start();
    }

    function setImageIndex(i)
    {
        _imageIndex = i;

        if (_imageIndex >= 0 && _imageIndex < imagePaths.count)
            image.source = imagePaths.get(_imageIndex).path;
        else
            image.source = "";
    }

    Timer {
        id: playTimer

        interval: 200
        repeat: false

        onTriggered: {
            if (_imageIndex + 1 < imagePaths.count)
            {
                setImageIndex(_imageIndex + 1);
                playTimer.start();
            }
            else
            {
                setImageIndex(-1);
                root.state = "";
            }
        }
    }
```

高级用法（Advanced Techniques）

10.5.1 实现一个播放列表（Implementing a Playlist）

Qt 5 multimedia接口没有提供播放列表。幸好，它非常容易实现。通过设置模型子项与MediaPlayer元素可以实现它，如下所示。当playstate通过player控制时，Playlist元素负责设置MediaPlayer的source。

```
    Playlist {
        id: playlist

        mediaPlayer: player

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

    MediaPlayer {
        id: player
    }
```

Playlist元素的第一部分如下，注意使用setIndex函数来设置source元素的索引值。我们也实现了next与previous函数来操作链表。

```
Item {
    id: root

    property int index: 0
    property MediaPlayer mediaPlayer
    property ListModel items: ListModel {}

    function setIndex(i)
    {
        console.log("setting index to: " + i);

        index = i;

        if (index < 0 || index >= items.count)
        {
            index = -1;
            mediaPlayer.source = "";
        }
        else
            mediaPlayer.source = items.get(index).source;
    }

    function next()
    {
        setIndex(index + 1);
    }
```

```
function previous()
{
    setIndex(index + 1);
}
```

让播放列表自动播放下一个元素的诀窍是使用MediaPlayer的status属性。当得到MediaPlayer.EndOfMedia状态时，索引值增加，恢复播放，或者当列表达到最后时，停止播放。

```
Connections {
    target: root.mediaPlayer

    onStopped: {
        if (root.mediaPlayer.status == MediaPlayer.EndOfMedia)
        {
            root.next();
            if (root.index == -1)
                root.mediaPlayer.stop();
            else
                root.mediaPlayer.play();
        }
    }
}
```

总结（Summary）

Qt的媒体应用程序接口提供了播放和捕捉视频和音频的机制。通过VideoOutput元素，视频源能够在我们的用户界面上显示。通过MediaPlayer元素，可以操作大多数的播放，SoundEffect被用于低延迟的声音。Camera元素被用来截图或者显示一个实时的视频流。

Networking

Qt5在C++中有丰富的网络相关的类。例如在http协议层上使用请求回答方式的高级封装类如QNetworkRequest，QNetworkReply，QNetworkAccessManageer。也有在TCP/IP或者UDP协议层封装的低级类如QTcpSocket，QTcpServer和QUdpSocket。还有一些额外的类用来管理代理，网络缓冲和系统网络配置。

这章将不再阐述关于C++网络方面的知识，这章是关于QtQuick与网络的知识。我们应该怎样连接QML/JS用户界面与网络服务，或者如何通过网络服务来为我们用户界面提供服务。已经有很好的教材和示例覆盖了关于Qt/C++的网络编程。然后你只需要阅读这章相关的C++集成来满足你的QtQuick就可以了。

通过HTTP服务UI（Serving UI via HTTP）

通过HTTP加载一个简单的用户界面，我们需要一个web服务器，它为UI文件服务。但是首先我们需要有用户界面，我们在项目里创建一个创建了红色矩形框的main.qml。

```
// main.qml
import QtQuick 2.0

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'
}
```

我们加载一段python脚本来提供这个文件：

```
$ cd <PROJECT>
# python -m SimpleHTTPServer 8080
```

现在我们可以通过http://localhost:8000/main.qml来访问，你可以像下面这样测试：

```
$ curl http://localhost:8000/main.qml
```

或者你可以用浏览器来访问。浏览器无法识别QML，并且无法通过文档来渲染。我们需要创建一个可以浏览QML文档的浏览器。为了渲染文档，我们需要指出qmlscene的位置。不幸的是qmlscene只能读取本地文件。我们为了突破这个限制，我们可以使用自己写的qmlscene或者使用QML动态加载。我们选择动态加载的方式。我们选择一个加载元素来加载远程的文档。

```
// remote.qml
import QtQuick 2.0

Loader {
    id: root
    source: 'http://localhost:8080/main2.qml'
    onLoaded: {
        root.width = item.width
        root.height = item.height
    }
}
```

我们现在可以使用qmlscene来加载remote.qml文档。这里仍然有一个小问题。加载器将会调整加载项的大小。我们的qmlscene需要适配大小。可以使用--resize-to-root选项来运行qmlscene。

```
$ qmlscene --resize-to-root remote.qml
```

按照root元素调整大小，告诉qmlscene按照root元素的大小调它的窗口大小。remote现在从本地服务器加载main.qml，并且可以自动调整加载的用户界面。方便且简单。

注意

如果你不想使用一个本地服务器，你可以使用来自**GitHub**的**gist**服务。**Gist**是一个在线剪切板服务，就像**PasteBin**等等。可以在**https://gist.github.com**下使用。我创建了一个简单的**gist**例子，地址是**https://gist.github.com/jryannel/7983492**。这将会返回一个绿色矩形框。由于**gist**连接提供的是**HTML**代码，我们需要连接一个**/raw**来读取原始文件而不是**HTML**代码。

```
// remote.qml
import QtQuick 2.0

Loader {
    id: root
    source: 'https://gist.github.com/jryannel/7983492/raw'
    onLoaded: {
        root.width = item.width
        root.height = item.height
    }
}
```

从网络加载另一个文件，你只需要引用组件名。例如一个Button.qml，只要它们在同一个远程文件夹下就能够像正常一样访问。

11.1.1 网络组件（Networked Components）

我们做了一个小实验。我们在远程端添加一个按钮作为可以复用的组件。

```
- src/main.qml
- src/Button.qml
```

我们修改main.qml来使用button：

```
import QtQuick 2.0

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Button {
        anchors.centerIn: parent
        text: 'Click Me'
        onClicked: Qt.quit()
    }
}
```

再次加载我们的web服务器：

```
$ cd src
# python -m SimpleHTTPServer 8080
```

再次使用http加载远mainQML文件：

```
$ qmlscene --resize-to-root remote.qml
```

我们看到一个错误：

```
http://localhost:8080/main2.qml:11:5: Button is not a type
```

所以，在远程加载时，QML无法解决Button组件的问题。如果代码使用本地加载qmlscene src/main.qml，将不会有问题。Qt能够直接解析本地文件，并且检测哪些组件可用，但是使用http的远程访问没有“list-dir”函数。我们可以在main.qml中使用import声明来强制QML加载元素：

```
import "http://localhost:8080" as Remote

...

Remote.Button { ... }
```

再次运行qmlscene后，它将正常工作：

```
$ qmlscene --resize-to-root remote.qml
```

这是完整的代码：

```
// main2.qml
import QtQuick 2.0
import "http://localhost:8080" 1.0 as Remote

Rectangle {
    width: 320
    height: 320
    color: '#ff0000'

    Remote.Button {
        anchors.centerIn: parent
        text: 'Click Me'
        onClicked: Qt.quit()
    }
}
```

一个更好的选择是在服务器端使用qmldir文件来控制输出：

```
// qmldir
Button 1.0 Button.qml
```

然后更新main.qml：

```
import "http://localhost:8080" 1.0 as Remote

...

Remote.Button { ... }
```

当从本地文件系统使用组件时，它们的创建没有延迟。当组件通过网络加载时，它们的创建是异步的。创建时间的影响是未知的，当其它组件已经完成时，一个组件可能还没有完成加载。当通过网络加载组件时，需要考虑这些。

模板（Templating）

当使用HTML项目时，通常需要使用模板驱动开发。服务器使用模板机制生成代码在服务器端对一个HTML根进行扩展。例如一个照片列表的列表头将使用HTML编码，动态图片链表将会使用模板机制动态生成。通常这也可以使用QML解决，但是仍然有一些问题。

首先，HTML开发者这样做的原因是为了克服HTML后端的限制。在HTML中没有组件模型，动态机制方面不得不使用这些机制或者在客户端边使用javascript编程。很多的JS框架产生（jQuery，dojo，backbone，angular，...）可以用来解决这个问题，把更多的逻辑问题放在使用网络服务连接的客户端浏览器。客户端使用一个web服务的接口（例如JSON服务，或者XML数据服务）与服务器通信。这也适用于QML。

第二个问题是来自QML的组件缓冲。当QML访问一个组件时，缓冲渲染树（render-tree），并且只加载缓冲版本来渲染。磁盘上的修改版本或者远程的修改在没有重新启动客户端时不会被检测到。为了克服这个问题，我们需要跟踪。我们使用URL后缀来加载链接（例如 http://localhost:8080/main.qml#1234），“#1234”就是后缀标识。HTTP服务器总是为相同的文档服务，但是QML将使用完整的链接来保存这个文档，包括链接标识。每次我们访问的这个链接的标识获得改变，QML缓冲无法获得这个信息。这个后缀标识可以是当前时间的毫秒或者一个随机数。

```
Loader {
    source: 'http://localhost:8080/main.qml#' + new Date().getTime()
}
```

总之，模板可以实现，但是不推荐，无法完整发挥QML的长处。一个更好的方法是使用web服务提供JSON或者XML数据服务。

HTTP请求（HTTP Requests）

从c++方面来看，Qt中完成http请求通常是使用QNetworkRequest和QNetworkReply，然后使用Qt/C++将响应推送到集成的QML。所以我们尝试使用QtQuick的工具给我们的网络信息尾部封装了小段信息，然后推送这些信息。为此我们使用一个帮助对象来构造http请求，和循环响应。它使用java脚本的XMLHttpRequest对象的格式。

XMLHttpRequest对象允许用户注册一个响应操作函数和一个链接。一个请求能够使用http动作来发送（如get，post，put，delete，等等）。当响应到达时，会调用注册的操作函数。操作函数会被调用多次。每次调用请求的状态都已经改变（例如信息头部已接收，或者响应完成）。

下面是一个简短的例子：

```
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();
}
```

从一个响应中你可以获取XML格式的数据或者是原始文本。可以遍历XML结果但是通常使用原始文本来匹配JSON格式响应。使用JSON.parse(text）可以JSON文档将转换为JS对象使用。

```
...
} else if(xhr.readyState === XMLHttpRequest.DONE) {
    var object = JSON.parse(xhr.responseText.toString());
    print(JSON.stringify(object, null, 2));
}
```

在响应操作中，我们访问原始响应文本并且将它转换为一个javascript对象。JSON对象是一个可以使用的JS对象（在javascript中，一个对象可以是对象或者一个数组）。

注意

toString()转换似乎让代码更加稳定。在不使用显式的转换下我有几次都解析错误。不确定是什么问题引起的。

11.3.1 Flickr调用（Flickr Call）

让我们看看更加真实的例子。一个典型的例子是使用网络相册服务来取得公共订阅中新上传的图片。我们可以使用http://api.flicker.com/services/feeds/photos_public.gne链接。不幸的是它默认返回XML流格式的

数据，在qml中可以很方便的使用XmlListModel来解析。为了达到只关注JSON数据的目的，我们需要在请求中附加一些参数可以得到JSON响应：http://api.flickr.com/services/feeds/photo_public.gne?format=json&nojsoncallback=1。这将会返回一个没有JSON回调的JSON响应。

注意 一个**JSON**回调将**JSON**响应包装在一个函数调用中。这是一个**HTML**编程中的快捷方式，使用脚本标记来创建一个**JSON**请求。响应将触发本地定义的回调函数。在**QML**中没有**JSON**回调的工作机制。

使用curl来查看响应：

```
curl "http://api.flickr.com/services/feeds/photos_public.gne?format=json&nojsoncallback=1
```

响应如下：

```
{
    "title": "Recent Uploads tagged munich",
    ...
    "items": [
        {
        "title": "Candle lit dinner in Munich",
        "media": {"m":"http://farm8.staticflickr.com/7313/11444882743_2f5f87169f_m.jpg"},
        ...
        },{
        "title": "Munich after sunset: a train full of \"must haves\" =",
        "media": {"m":"http://farm8.staticflickr.com/7394/11443414206_a462c80e83_m.jpg"},
        ...
        }
    ]
    ...
}
```

JSON文档已经定义了结构体。一个对象包含一个标题和子项的属性。标题是一个字符串，子项是一组对象。当转换文本为一个JSON文档后，你可以单独访问这些条目，它们都是可用的JS对象或者结构体数组。

```
// 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
}
```

我们可以使用obj.items数组将JS数组作为链表视图的模型，试着完成这个操作。首先我们需要取得响应并且将它转换为可用的JS对象。然后设置response.items属性作为链表视图的模型。

```
function request() {
    var xhr = new XMLHttpRequest();
```

```
    xhr.onreadystatechange = function() {
        if(...) {
            ...
        } else if(xhr.readyState === XMLHttpRequest.DONE) {
            var response = JSON.parse(xhr.responseText.toString());
            // set JS object as model for listview
            view.model = response.items;
        }
    }
    xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?format=json&n
    xhr.send();
}
```

下面是完整的源代码，当组件加载完成后，我们创建请求。然后使用请求的响应作为我们链表视图的模型。

```
import QtQuick 2.0

Rectangle {
    width: 320
    height: 480
    ListView {
        id: view
        anchors.fill: parent
        delegate: Thumbnail {
            width: view.width
            text: modelData.title
            iconSource: modelData.media.m
        }
    }

    function request() {
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
                print('HEADERS_RECEIVED')
            } else if(xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE')
                var json = JSON.parse(xhr.responseText.toString())
                view.model = json.items
            }
        }
        xhr.open("GET", "http://api.flickr.com/services/feeds/photos_public.gne?format=js
        xhr.send();
    }

    Component.onCompleted: {
        request()
    }
}
```

当文档完整加载后（Component.onCompleted）,我们从Flickr请求最新的订阅内容。我们解析JSON的响应并且设置item数组作为我们视图的模型。链表视图有一个代理可以在一行中显示图标缩略图和标题文本。

另一种方法是添加一个ListModel，并且将每个子项添加到链表模型中。为了支持更大的模型，需要支持分页和懒加载。

本地文件（Local files）

使用XMLHttpRequest也可以加载本地文件（XML/JSON）。例如加载一个本地名为“colors.json”的文件可以这样使用：

```
xhr.open("GET", "colors.json");
```

我们使用它读取一个颜色表并且使用表格来显示。从QtQuick这边无法修改文件。为了将源数据存储回去，我们需要一个基于HTTP服务器的REST服务支持或者一个用来访问文件的QtQuick扩展。

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
    color: '#000'

    GridView {
        id: view
        anchors.fill: parent
        cellWidth: width/4
        cellHeight: cellWidth
        delegate: Rectangle {
            width: view.cellWidth
            height: view.cellHeight
            color: modelData.value
        }
    }

    function request() {
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function() {
            if (xhr.readyState === XMLHttpRequest.HEADERS_RECEIVED) {
                print('HEADERS_RECEIVED')
            } else if(xhr.readyState === XMLHttpRequest.DONE) {
                print('DONE');
                var obj = JSON.parse(xhr.responseText.toString());
                view.model = obj.colors
            }
        }
        xhr.open("GET", "colors.json");
        xhr.send();
    }

    Component.onCompleted: {
        request()
    }
}
```

也可以使用XmlListModel来替代XMLHttpRequest访问本地文件。

```
import QtQuick.XmlListModel 2.0

XmlListModel {
    source: "http://localhost:8080/colors.xml"
    query: "/colors"
    XmlRole { name: 'color'; query: 'name/string()' }
    XmlRole { name: 'value'; query: 'value/string()' }
}
```

XmlListModel只能用来读取XML文件，不能读取JSON文件。

REST接口（REST API）

为了使用web服务，我们首先需要创建它。我们使用Flask（http://flask.pocoo.org），一个基于python创建简单的颜色web服务的HTTP服务器应用。你也可以使用其它的web服务器，只要它接收和返回JSON数据。通过web服务来管理一组已经命名的颜色。在这个例子中，管理意味着CRUD（创建-读取-更新-删除）。

在Flask中一个简单的web服务可以写入一个文件。我们使用一个空的服务器.py文件开始，在这个文件中我们创建一些规则并且从额外的JSON文件中加载初始颜色。你可以查看Flask文档获取更多的帮助。

```
from flask import Flask, jsonify, request
import json

colors = json.load(file('colors.json', 'r'))

app = Flask(__name__)

# ... service calls go here

if __name__ == '__main__':
    app.run(debug = True)
```

当你运行这个脚本后，它会在http://localhost:5000。

我们开始添加我们的CRUD（创建，读取，更新，删除）到我们的web服务。

11.5.1 读取请求（Read Request）

从web服务读取数据，我们提供GET方法来读取所有的颜色。

```
@app.route('/colors', methods = ['GET'])
def get_colors():
    return jsonify( { "colors" :  colors })
```

这将会返回'/colors'下的颜色。我们使用curl来创建一个http请求测试。

```
curl -i -GET http://localhost:5000/colors
```

这将会返回给我们JSON数据的颜色链表。

11.5.2 读取接口（Read Entry）

为了通过名字读取颜色，我们提供更加详细的后缀，定位在'/colors/'下。名称是后缀的参数，用来识别一个独立的颜色。

```
@app.route('/colors/<name>', methods = ['GET'])
def get_color(name):
    for color in colors:
        if color["name"] == name:
            return jsonify( color )
```

我们再次使用curl测试，例如获取一个红色的接口。

```
curl -i -GET http://localhost:5000/colors/red
```

这将返回一个JSON数据的颜色。

11.5.3 创建接口（Create Entry）

目前我们仅仅使用了HTTP GET方法。为了在服务器端创建一个接口，我们使用POST方法，并且将新的颜色信息发使用POST数据发送。后缀与获取所有颜色相同，但是我们需要使用一个POST请求。

```
@app.route('/colors', methods= ['POST'])
def create_color():
    color = {
        'name': request.json['name'],
        'value': request.json['value']
    }
    colors.append(color)
    return jsonify( color ), 201
```

curl非常灵活，允许我们使用JSON数据作为新的接口包含在POST请求中。

```
curl -i -H "Content-Type: application/json" -X POST -d '{"name":"gray1","value":"#333"}'
```

11.5.4 更新接口（Update Entry）

我们使用PUT HTTP方法来添加新的update接口。后缀与取得一个颜色接口相同。当颜色更新后，我们获取更新后JSON数据的颜色。

```
@app.route('/colors/<name>', methods= ['PUT'])
def update_color(name):
    for color in colors:
        if color["name"] == name:
            color['value'] = request.json.get('value', color['value'])
            return jsonify( color )
```

在curl请求中，我们用JSON数据来定义更新值，后缀名用来识别哪个颜色需要更新。

```
curl -i -H "Content-Type: application/json" -X PUT -d '{"value":"#666"}' http://localhost
```

11.5.5 删除接口（Delete Entry）

使用DELETE HTTP来完成删除接口。使用与颜色相同的后缀，但是使用DELETE HTTP方法。

```
@app.route('/colors/<name>', methods=['DELETE'])
def delete_color(name):
    success = False
    for color in colors:
        if color["name"] == name:
            colors.remove(color)
            success = True
    return jsonify( { 'result' : success } )
```

这个请求看起来与GET请求一个颜色类似。

```
curl -i -X DELETE http://localhost:5000/colors/red
```

现在我们能够读取所有颜色，读取指定颜色，创建新的颜色，更新颜色和删除颜色。我们知道使用HTTP后缀来访问我们的接口。

动作	**HTTP**协议	后缀
读取所有	GET	http://localhost:5000/colors
创建接口	POST	http://localhost:5000/colors
读取接口	GET	http://localhost:5000/colors/name
更新接口	PUT	http://localhost:5000/colors/name
删除接口	DELETE	http://localhost:500/colors/name

REST服务已经完成，我们现在只需要关注QML和客户端。为了创建一个简单好用的接口，我们需要映射每个动作为一个独立的HTTP请求，并且给我们的用户提供一个简单的接口。

11.5.6 REST客户端（REST Client）

为了展示REST客户端，我们写了一个小的颜色表格。这个颜色表格显示了通过HTTP请求从web服务取得的颜色。我们的用户界面提供以下命令：

- 获取颜色链表
- 创建颜色
- 读取最后的颜色
- 更新最后的颜色

- 删除最后的颜色

我们将我们的接口包装在一个JS文件中，叫做colorservice.js，并将它导入到我们的UI中作为服务（Service）。在服务模块中，我们创建了帮助函数来为我们构造HTTP请求：

```
// colorservice.js
function request(verb, endpoint, obj, cb) {
    print('request: ' + verb + ' ' + BASE + (endpoint?'/' + endpoint:''))
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        print('xhr: on ready state change: ' + xhr.readyState)
        if(xhr.readyState === XMLHttpRequest.DONE) {
            if(cb) {
                var res = JSON.parse(xhr.responseText.toString())
                cb(res);
            }
        }
    }
    xhr.open(verb, BASE + (endpoint?'/' + endpoint:''));
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.setRequestHeader('Accept', 'application/json');
    var data = obj?JSON.stringify(obj):''
    xhr.send(data)
}
```

包含四个参数。verb，定义了使用HTTP的动作（GET，POST，PUT，DELETE）。第二个参数是作为基础地址的后缀（例如'http://localhost:5000/colors'）。第三个参数是可选对象，作为JSON数据发送给服务的数据。最后一个选项是定义当响应返回时的回调。回调接收一个响应数据的响应对象。

在我们发送请求前，我们需要明确我们发送和接收的JSON数据修改的请求头。

```
// colorservice.js
function get_colors(cb) {
    // GET http://localhost:5000/colors
    request('GET', null, null, cb)
}

function create_color(entry, cb) {
    // POST http://localhost:5000/colors
    request('POST', null, entry, cb)
}

function get_color(name, cb) {
    // GET http://localhost:5000/colors/<name>
    request('GET', name, null, cb)
}

function update_color(name, entry, cb) {
    // PUT http://localhost:5000/colors/<name>
    request('PUT', name, entry, cb)
}

function delete_color(name, cb) {
    // DELETE http://localhost:5000/colors/<name>
    request('DELETE', name, null, cb)
```

```
}
```

这些代码在服务实现中。在UI中我们使用服务来实现我们的命令。我们有一个存储id的ListModel和存储数据的gridModel为GridView提供数据。命令使用Button元素来发送。

读取服务器颜色链表。

```
// rest.qml
import "colorservice.js" as Service
...
// read colors command
Button {
    text: 'Read Colors';
    onClicked: {
        Service.get_colors( function(resp) {
            print('handle get colors resp: ' + JSON.stringify(resp));
            gridModel.clear();
            var entries = resp.data;
            for(var i=0; i<entries.length; i++) {
                gridModel.append(entries[i]);
            }
        });
    }
}
```

在服务器上创建一个新的颜色。

```
// rest.qml
import "colorservice.js" as Service
...
// create new color command
Button {
    text: 'Create New';
    onClicked: {
        var index = gridModel.count-1
        var entry = {
            name: 'color-' + index,
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toString()
        }
        Service.create_color(entry, function(resp) {
            print('handle create color resp: ' + JSON.stringify(resp))
            gridModel.append(resp)
        });
    }
}
```

基于名称读取一个颜色。

```
// rest.qml
import "colorservice.js" as Service
...
// read last color command
Button {
```

```
    text: 'Read Last Color';
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        Service.get_color(name, function(resp) {
            print('handle get color resp:' + JSON.stringify(resp))
            message.text = resp.value
        });
    }
}
```

基于颜色名称更新服务器上的一个颜色。

```
// rest.qml
import "colorservice.js" as Service
...
// update color command
Button {
    text: 'Update Last Color'
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        var entry = {
            value: Qt.hsla(Math.random(), 0.5, 0.5, 1.0).toString()
        }
        Service.update_color(name, entry, function(resp) {
            print('handle update color resp: ' + JSON.stringify(resp))
            var index = gridModel.count-1
            gridModel.setProperty(index, 'value', resp.value)
        });
    }
}
```

基于颜色名称删除一个颜色。

```
// rest.qml
import "colorservice.js" as Service
...
// delete color command
Button {
    text: 'Delete Last Color'
    onClicked: {
        var index = gridModel.count-1
        var name = gridModel.get(index).name
        Service.delete_color(name)
        gridModel.remove(index, 1)
    }
}
```

在CRUD（创建，读取，更新，删除）操作使用REST接口。也可以使用其它的方法来创建web服务接口。可以基于模块，每个模块都有自己的后缀。可以使用JSON RPC（http://www.jsonrpc.org/）来定义接口。当然基于XML的接口也可以使用，但是JSON在作为JavaScript部分解析进QML/JS中更有优势。

使用开放授权登陆验证（Authentication using OAuth）

OAuth是一个开放协议，允许简单的安全验证，是来自web的典型方法，用于移动和桌面应用程序。使用OAuth对通常的web服务的客户端进行身份验证，例如Google，Facebook和Twitter。

注意

对于自定义的**web**服务，你也可以使用典型的**HTTP**身份验证，例如使用**XMLHttpRequest**的用户名和密码的获取方法（比如**xhr.open(verb,url,true,username,password**））。

Auth目前不是QML/JS的接口，你需要写一些C++代码并且将身份验证导入到QML/JS中。另一个问题是安全的存储访问密码。

下面这些是我找到的有用的连接：

- http://oauth.net
- http://hueniverse.com/oauth/
- https://github.com/pipacs/o2
- http://www.johanpaul.com/blog/2011/05/oauth2-explained-with-qt-quick/

Engine IO

Engine IO是DIGIA运行的一个web服务。它允许Qt/QML应用程序访问来自Engin.IO的NoSQL存储。这是一个基于云存储对象的Qt/QML接口和一个管理平台。如果你想存储一个QML应用程序的数据到云存储中，它可以提供非常方便的QML/JS的接口。

查看EnginIO的文档获得更多的帮助。

Web Sockets

webSockets不是Qt提供的。将WebSockets加入到Qt/QML中需要花费一些工作。从作者的角度来看WebSockets有巨大的潜力来添加HTTP服务缺少的功能-通知。HTTP给了我们get和post的功能，但是post还不是一个通知。目前客户端轮询服务器来获得应用程序的服务，服务器也需要能通知客户端变化和事件。你可以与QML接口比较：属性，函数，信号。也可以叫做获取/设置/调用和通知。

QML WebSocket插件将会在Qt5中加入。你可以试试来自qt playground的web sockets插件。为了测试，我们使用一个现有的web socket服务实现了echo server。

首先确保你使用的Qt5.2.x。

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

然后你需要克隆web socket的代码库，并且编译它。

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

现在你可以在qml模块中使用web socket。

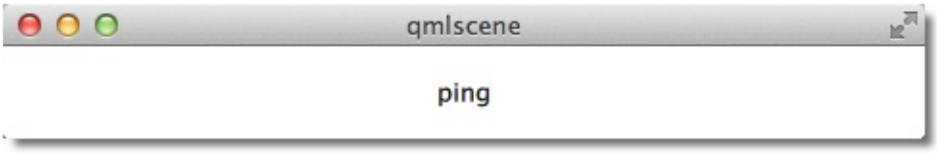
```
import Qt.WebSockets 1.0

WebSocket {
    id: socket
}
```

测试你的web socket，我们使用来自http://websocket.org的echo server 。

```
import QtQuick 2.0
import Qt.WebSockets 1.0

Text {
    width: 480
    height: 48

    horizontalAlignment: Text.AlignHCenter
    verticalAlignment: Text.AlignVCenter

    WebSocket {
        id: socket
        url: "ws://echo.websocket.org"
        active: true
        onTextMessageReceived: {
```

```
            text = message
        }
        onStatusChanged: {
            if (socket.status == WebSocket.Error) {
                console.log("Error: " + socket.errorString)
            } else if (socket.status == WebSocket.Open) {
                socket.sendTextMessage("ping")
            } else if (socket.status == WebSocket.Closed) {
                text += "\nSocket closed"
            }
        }
    }
}
```

你可以看到我们使用socket.sendTextMessage("ping")作为响应在文本区域中。



11.8.1 WS Server

你可以使用Qt WebSocket的C++部分来创建你自己的WS Server或者使用一个不同的WS实现。它非常有趣，是因为它允许连接使用大量扩展的web应用程序服务的高质量渲染的QML。在这个例子中，我们将使用基于web socket的ws模块的Node JS。你首先需要安装node.js。然后创建一个ws_server文件夹，使用node package manager（npm）安装ws包。

```
$ cd ws_server
$ npm install ws
```

npm工具下载并安装了ws包到你的本地依赖文件夹中。

一个server.js文件是我们服务器的实现。服务器代码将在端口3000创建一个web socket服务并监听连接。在一个连接加入后，它将会发送一个欢迎并等待客户端信息。每个客户端发送到socket信息都会发送回客户端。

```
var WebSocketServer = require('ws').Server;

var server = new WebSocketServer({ port : 3000 });

server.on('connection', function(socket) {
    console.log('client connected');
    socket.on('message', function(msg) {
        console.log('Message: %s', msg);
        socket.send(msg);
    });
    socket.send('Welcome to Awesome Chat');
});
```

```
console.log('listening on port ' + server.options.port);
```

你需要获取使用的JavaScript标记和回调函数。

11.8.2 WS Client

在客户端我们需要一个链表视图来显示信息，和一个文本输入来输入新的聊天信息。

在例子中我们使用一个白色的标签。

```
// Label.qml
import QtQuick 2.0

Text {
    color: '#fff'
    horizontalAlignment: Text.AlignLeft
    verticalAlignment: Text.AlignVCenter
}
```

我们的聊天视图是一个链表视图，文本被加入到链表模型中。每个条目显示使用行前缀和信息标签。我们使用单元将它分为24列。

```
// ChatView.qml
import QtQuick 2.0

ListView {
    id: root
    width: 100
    height: 62

    model: ListModel {}

    function append(prefix, message) {
        model.append({prefix: prefix, message: message})
    }

    delegate: Row {
        width: root.width
        height: 18
        property real cw: width/24
        Label {
            width: cw*1
            height: parent.height
            text: model.prefix
        }
        Label {
            width: cw*23
            height: parent.height
            text: model.message
        }
    }
}
```

聊天输入框是一个简单的使用颜色包裹边界的文本输入。

```
// ChatInput.qml
import QtQuick 2.0

FocusScope {
    id: root
    width: 240
    height: 32
    Rectangle {
        anchors.fill: parent
        color: '#000'
        border.color: '#fff'
        border.width: 2
    }

    property alias text: input.text

    signal accepted(string text)

    TextInput {
        id: input
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.verticalCenter: parent.verticalCenter
        anchors.leftMargin: 4
        anchors.rightMargin: 4
        onAccepted: root.accepted(text)
        color: '#fff'
        focus: true
    }
}
```

当web socket返回一个信息后，它将会把信息添加到聊天视图中。这也同样适用于状态改变。也可以当用户输入一个聊天信息，将聊天信息拷贝添加到客户端的聊天视图中，并将信息发送给服务器。

```
// ws_client.qml
import QtQuick 2.0
import Qt.WebSockets 1.0

Rectangle {
    width: 360
    height: 360
    color: '#000'

    ChatView {
        id: box
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: input.top
    }
    ChatInput {
        id: input
```

```
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        focus: true
        onAccepted: {
            print('send message: ' + text)
            socket.sendTextMessage(text)
            box.append('>', text)
            text = ''
        }
    }
    WebSocket {
        id: socket

        url: "ws://localhost:3000"
        active: true
        onTextMessageReceived: {
            box.append('<', message)
        }
        onStatusChanged: {
            if (socket.status == WebSocket.Error) {
                box.append('#', 'socket error ' + socket.errorString)
            } else if (socket.status == WebSocket.Open) {
                box.append('#', 'socket open')
            } else if (socket.status == WebSocket.Closed) {
                box.append('#', 'socket closed')
            }
        }
    }
}
```

你首先需要运行服务器，然后是客户端。在我们简单例子中没有客户端重连的机制。

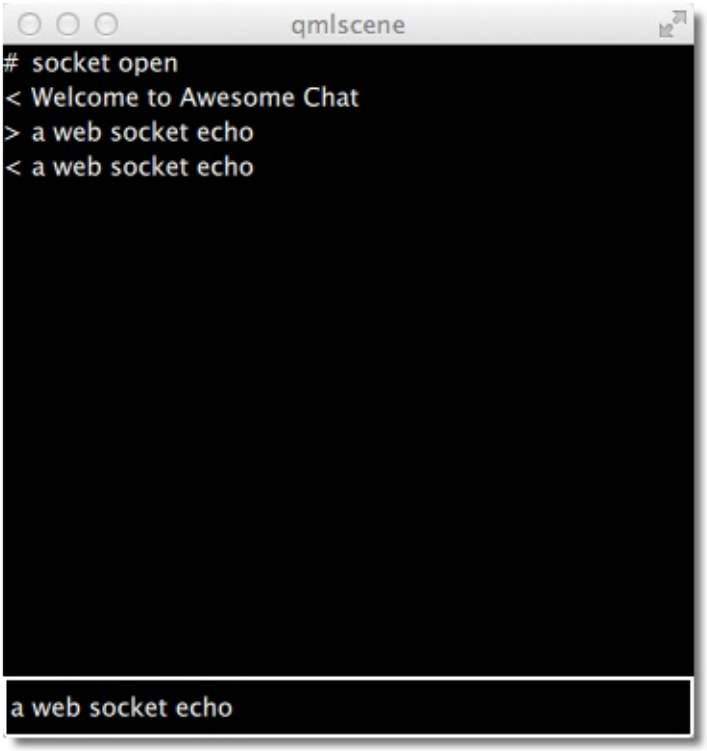
运行服务器

```
$ cd ws_server
$ node server.js
```

运行客户端

```
$ cd ws_client
$ qmlscene ws_client.qml
```

当输入文本并点击发送后，你可以看到类似下面这样。


qmlscene
socket open
< Welcome to Awesome Chat
> a web socket echo
< a web socket echo
a web socket echo

总结（Summary）

这章我们讨论了关于QML的网络应用。请记住Qt已在本地端提供了丰富的网络接口可以在QML中使用。但是这一章的我们是想推动QML的网络运用和如何与云服务集成。