Qt day

Luca Ottaviano

# Develop Qt UIs on Raspberry Pi

# Bio



Senior developer @Develer

Working with Qt on embedded systems since 2008

Specialized in training and mentoring

# Introduction to QtQuick

# What is QtQuick



QtQuick is a set of technologies to develop fluid UI interfaces

It allows to interact with the system thanks to the integration with C++

It’s developed by The Qt Company and it is released with two licenses, open source and commercial

# Why QtQuick



QtQuick is a “batteries included” framework

It is a de-facto standard for embedded Linux industrial projects

It’s a cross platform solution which scales from microcontrollers to desktop applications

# Today's goal



Car dashboard

# Agenda



- Creation of a UI starting from graphical assets
- Integration with C++ code
- Cross compilation overview and compilation on Raspberry targets
- Visualization on browsers (if time allows)

# QtQuick: overview



It’s a set of technologies for rapid application development

- QML: it’s a declarative language to write UI components
- Runtime: executes QML code and provides the Javascript engine to execute the code
- C++: it provides the UI backend, it integrates with the machine

# QML

It describes the components that are part of the UI

It handles user’s input through mouse, touchscreen or keyboard

It executes in a sandbox

It communicates with QML through some “blessed” objects

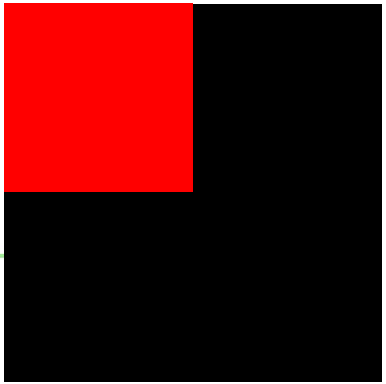
It interacts and controls the machine

Unrestricted code

# QML: basic concepts

# QML: basic concepts





```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



## Import block

```
import QtQuick 2.11
```

## Object creation

```
Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



This code creates two objects of type Rectangle

The objects are in a scene

The istantiation position defines also the position in a hierarchy

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



Beware, these are not drawing commands, they are instantiations

A QML file is interpreted to create a scene

The scene is then drawn when needed

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



An object may have at most one parent

The object without a parent is the root object

1 QML file = 1 root object

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



Unique identifier of a QML object

Starts with a lowercase letter or _

It must be unique inside a QML file

It's needed to refer to other objects inside a file

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



Each QML type exposes properties

Properties:

- have a type
- control the aspect and the behaviour of an object

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts



You can bind properties together

A binding describes the relationship between properties

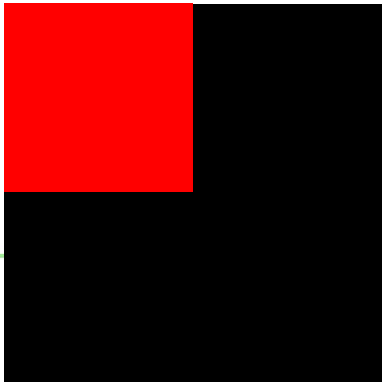
Each time any property mentioned on the right changes, the expression is evaluated again

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# QML: basic concepts

18/11/2020



```
import QtQuick 2.11

Rectangle {
    id: root
    color: "black"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

# Property types



Provided by Javascript

- int, bool, real, double
- string, url, list, var

Provided by QtQuick

- color, font, date, time, point, size, rect
- matrix4x4, vector2d, vector3d

# Types of QML objects



Visual types

- Item: it has an area but no visual aspect
- Rectangle, Text, Image
- Row, Column: they don’t have a visual aspect

Input types

- MouseArea

# Positioning of elements

# Positioning



You can create complex designs by assembling together simpler objects

There are three main methods to set the position of an object:

- Containers (Row, Column)
- Anchors
- ...and even absolute positioning (x, y)

# Containers




```
import QtQuick 2.11

Row {
    spacing: 10
    Image {
        source: "images/D.png"
    }
    Image {
        source: "images/N.png”
    }
    Image {
        source: "images/P.png"
    }
}
```

# Containers



They control the position of children elements

They have no authority on the size

Containers are “logic” elements, they don’t have a visual aspect

```
import QtQuick 2.11

Row {
    spacing: 10
    Image {
        source: "images/D.png"
    }
    Image {
        source: "images/N.png”
    }
    Image {
        source: "images/P.png"
    }
}
```

# Anchors



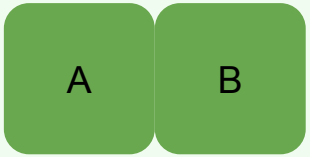
Anchors are an alternative positioning method

They control element's position and size

They rely on **property bindings**

# Anchors



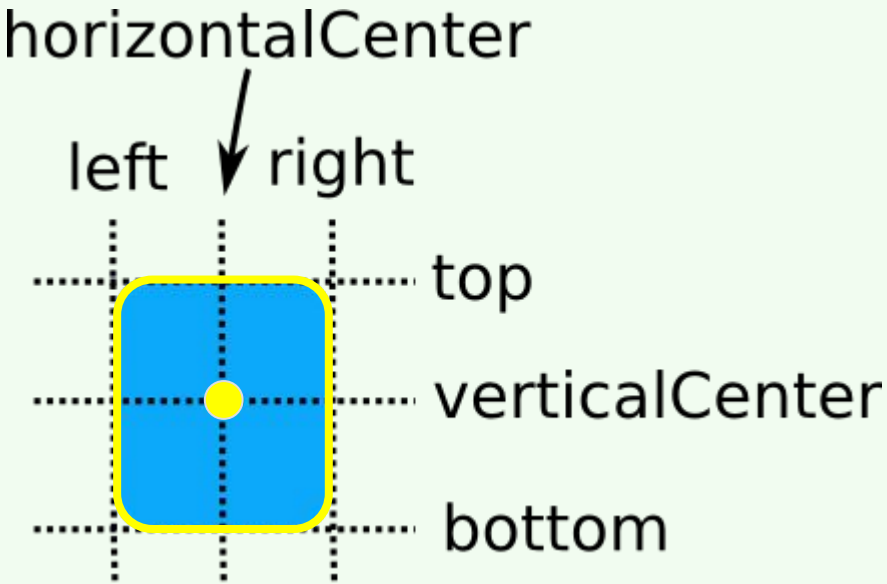



```
Item {
    id: b
    anchors.left: a.right
}
```

```
Item {
    id: small
    anchors.centerIn: big
}
```

# Anchors






You can apply margins, eg: topMargin, leftMargin
fill and centerIn are just shortcuts

# Anchors



```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

# Anchors



Anchors lines are a specific type, they can bind only with a compatible anchor line

Horizontal lines bind with horizontal ones; vertical lines bind with vertical ones

```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

# Anchors



Margins are active only if the respective anchor line is set

```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

# Components

# Components



You can use Components to assemble together simpler objects

They are the main method of code reuse in QML

# Components



Let's see how we can make a reusable component for the battery

```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

# Components



Let's add a property on the root element

Properties on the root object are the Component's interface

Let's save the file with an uppercase first letter (Battery.qml)

```
import QtQuick 2.11

Item {
    id: batt

    property int battery: 0
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: batt.battery + "%"
    }
}
```

# Components



```
Item {
    id: mainWindow
    Battery {
        battery: control_unit.battery
    }
}
```

main.qml

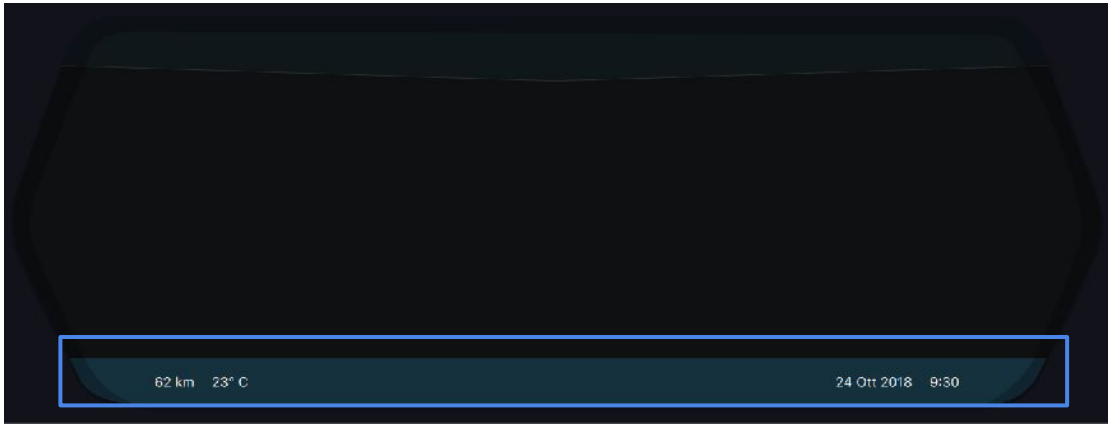
```
import QtQuick 2.11

Item {
    id: batt

    property int battery: 0
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: batt.battery + "%"
    }
}
```

Battery.qml

# Let’s build the first component together



# Repository structure



There is a tag for each step we are going to do

Don’t worry if you fall behind, there are checkpoints

My suggestion: start with tag “step0” and create your branch there

- git checkout step0
- git checkout -b mybranch

# First component: steps



1. Create a new file called “StatusBar.qml”
2. Create 4 Text elements and position them with anchors
   a. Don’t worry about the size of the root item for now
3. Use StatusBar{} inside MainDashboard.qml
   a. Now we set the size and the position of the item

# Next component: odometer

# Checkpoint



In case you didn't make it in time...

Current checkpoint: step1

- git commit -am "wip" #in case you want to save your work
- git checkout step1
- git checkout -b mybranch_step1

# Odometer: steps



1. The odometer is composed of multiple images (contagiri_back.png, contagiri_top.png, ghiera+numeri.png, centrale.png)

2. These images should be stacked.
   a. Look at the images, which one is at the bottom of the stack? Which at the top?

3. Put the element at the bottom of the stack directly in MainDashboard.qml, just above StatusBar{}

# Next: battery and speed

# Checkpoint



In case you didn't make it in time...

Current checkpoint: step2

- git commit -am "wip" #in case you want to save your work
- git checkout step2
- git checkout -b mybranch_step2

# Battery: steps



1. There is a battery image in the repo, but let's start with text

2. Create a Cruise.qml file, add a "battery" property on the root item

3. Add Cruise{} in MainDashboard.qml on top of the latest image of the previous step

4. Use control_unit.battery to read the battery value from C++. Set the binding in MainDashboard

# Speed: steps



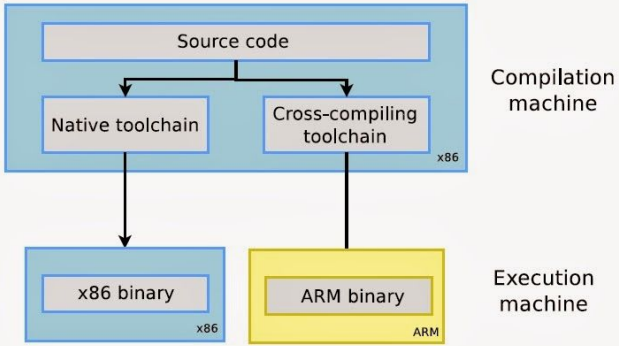
1. In Cruise.qml, add a “speed” property on the root item
2. Add a Text{} to render the speed
3. Use control_unit.speed to read the current speed from C++. Set the binding in MainDashboard

# Cross compilation

# Cross compilation



When doing cross compilation, you use a compiler on your PC (HOST) which generates code for the embedded board (TARGET)

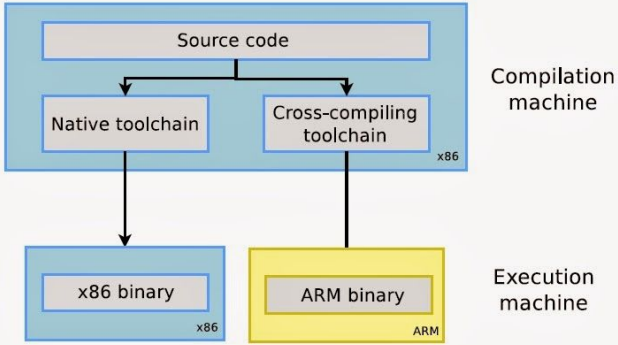



We use it because:

- The HOST machine is generally much more powerful to compile code
- We don't want to ship a compiler on the TARGET
- We don't need development tools on the TARGET

# Cross compilation






However for this workshop I will not use proper cross compilation because:

- The Raspberry TARGET requires a Linux HOST (this is true basically for every Linux TARGET)
- We have little C++ code, we can compile directly on TARGET

# Compiling on Raspberry



Let’s copy the project on the target, then let’s compile:

```
host $ rsync -avz workshop-qt-webgl pi@raspberrypi.local:/home/pi --exclude .git
host $ ssh pi@raspberrypi.local

target $ cd /home/pi/workshop-qt-webgl
target $ qmake
target $ make
```

# Executing on the target



This step depends on the root file system you have
On my machine it’s just:

```
target $ export DISPLAY=:0 # may not be required
target $ ./dashboard
```

# WebGL streaming plugin

# What is the WebGL streaming plugin?



The Qt Quick WebGL it’s a platform plugin to expose Qt Quick UIs to the browser

The UI is rendered on the browser using WebGL

Access is single user only, in case of more than one simultaneous connection only the first user can interact with the application

# How to use the WebGL plugin



Launch the application with the plugin:

target $ ./dashboard -platform webgl

Connect with the browser to:

http://raspberrypi.local:8080


Thanks for attending

Powered by

develer