Best Practices in Qt Quick/QML

Langston Ball

Senior Qt Developer

ICS, Inc.

Topics

- Introduction
- Integrating C++ and QML
- Anchors Are Your Friend
- Keyboard and Input Handling
- Closing

Introduction

(Delete if not used)

Integrating C++ and QML

- Rationale
- Integration techniques
- Using a controller between C++ & QML
  - Exporting controller objects
  - Using controller objects
- Comparison of approaches
- Design differences

Integrating C++ and QML

The goal is maximum functionality for the application.

- Integration of specialized hardware components and controls into the GUI
- Advanced data processing and long running operations.
- Providing clean, maintainable interfaces and extendable applications.

Any QObject-derived class can be registered as a qml type or injected into the QML context

Two well documented and efficient techniques:

- Injection of a c++ object into the qml context, making it a global object accessible to the qml engine
- Registering your c++ object as a type in QML
  - qmlRegisterType()
  - qmlRegisterInterface()
  - qmlRegisterUncreatableType()
  - qmlRegisterSingletonType()

Integrating C++ and QML

- QML , signals, and slots.
  - QML can utilize the signal of your c++ objects and its properties
  - It can also utilize slots inside those objects
    - public slots as well as ones marked with Q_INVOKABLE macro
  - These features make using c++ backends ideal for controlling the UI and allow for a clean seperation of UI and logic.

Integrating C++ and QML

- Writing a QML plug-in
  - QQmlExtensionPlugin
  -
  - Using plug-ins

Data Models and Views

- What is a model? What is a view?
  - Kinds of models
  - Kinds of views
- ListModels
  - Static
  - Dynamic
  - Delegate
    - Special delegate properties
  - C++ Models with QML Views
  - What about trees?

Closing

- *(Fill in)*

Anchors are Your Friend

- Principles of layout
- Property bindings
- Issues with bindings
- Complex property bindings
- Anchor lines
- Introduction to anchors
  - Margins

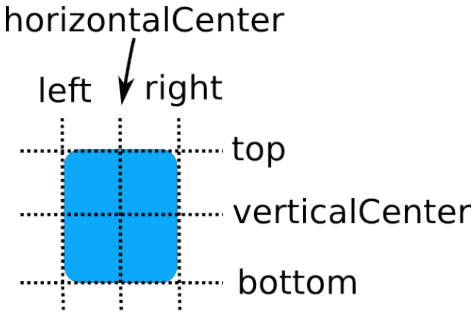
Implementation of Property Bindings

- Each QML Item is actually a QObject under the covers.
- It emits signals in response to property changes.
- Signals, when emitted, are processed by Qt's event loop and propagated to all connected objects.
- QML binding expressions are implemented as pre-compiled JavaScript closures, evaluated at runtime.
- Even though Qt's signal/slots are really efficient, each of those steps requires several cycles to be completed.
- Complex JavaScript expressions also need to be re-evaluated on value change.

Anchor Lines

- Each Item has seven invisible **anchor lines,**
- implemented privately in C++ as QDeclarativeAnchors in Qt Quick 1 and QQuickAnchors in Qt Quick 2.
- AnchorLine is a non-creatable type in QML.
- Anchor lines contain no value, only a reference to an Item and an enum value for which line it is referring to.
- Six anchor lines for an Item are shown in Figure 1.2.

Figure 1.2. Anchor Lines



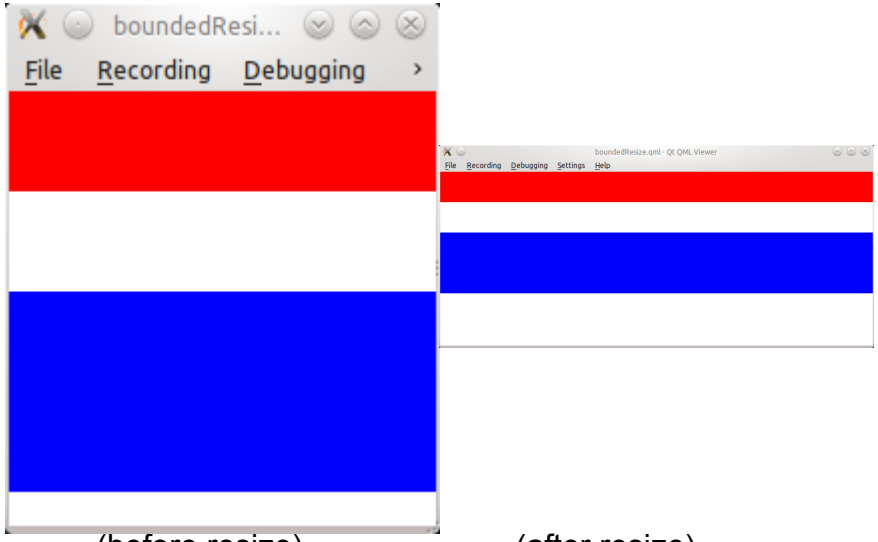
The seventh line is called the baseline. It is either:

- The line on which text would sit in that Item
- The same line as top, for an Item with no text

Introduction to Anchors

- The anchors attached property is used to establish unidirectional relationships between anchor lines of different Items.
- This provides a convenient way to position an item.
- Figure 1.3 shows two rectangles that grow as the parent window grows.

Figure 1.3. A Rescalable Screen with Two Rectangles



(before resize) (after resize)

Keyboard and Input Handling

- The overall picture
- Tab and arrow key navigation
- Focus scopes and object hierarchy
  - Focus scopes with dynamic items
  - Debugging keyboard focus
- Keyboard event forwarding
- Keyboard input simulation

The Overall Picture

QML Item behavior related to keyboard handling may not be obvious from the Qt documentation:

- **focus** property is false by default on all built-in items.
- When the scene loads, an item with statically set focus, that should also have the active focus, actually acquires it after the **onCompleted()** handlers run.
- **activeFocus** being true guarantees that an item will be the first to receive keyboard events.
- If, for example, you would like to bind to a property of a text input item in some complex scene to know for sure whether it is "active" (i.e. has a blinking cursor, etc), you should bind to **activeFocus**, instead of focus.

The Overall Picture

- Invisible items can have active focus (and focus), but are skipped during keyboard-driven navigation and key event forwarding.
- Disabled items (i.e. those with **enabled** set to false), on the other hand, are not excluded from the key event forwarding chain (but are still skipped during keyboard-driven navigation).
- A disabled item can not have active focus.
- If the user switches to another window, and that window starts getting keyboard input, **activeFocus** value changes to false.

The Overall Picture

```
Item {
    width: 200; height: 100
    Rectangle {
        anchors { fill: parent; margins: 25 }
        color: activeFocus? "red": "black"

        focus: true
        onFocusChanged: console.log("focus changed to: " + focus)
        onActiveFocusChanged: console.log("active focus changed to: " +
activeFocus)

        Component.onCompleted: console.log("onCompleted(), focus: " + focus
                                           + " activeFocus: " + activeFocus)
    }
}
```

Tab and Arrow Key Navigation

Things to note about the built-in means of keyboard navigation in QML:

- both disabled and invisible items are excluded from the focus chain
- invisible items still get keyboard input and can have active focus
- navigation chains don't wrap by themselves

Tab and Arrow Key Navigation

- when an item with active focus becomes invisible or disabled, active focus does not move to the next item in the navigation chain automatically.
- if at some point the item becomes visible and enabled again, active focus returns (but only if no other item in the scene acquired active focus between these events).
- the navigation chain may be dynamically changed.
- the enabled property, introduced in Qt 5, is handy when implementing "navigable" controls.

Tab and Arrow Key Navigation

```
Item {
    width: row.width + 100; height: row.height + 100

    Row {
        id: row
        anchors.centerIn: parent
        spacing: 25

        Item {                      1
            width: first.width; height: first.height
            FocusableButton {
                id: first
                KeyNavigation.tab: second 2
                text: "First"
                onClicked: console.log("First")
            }
        }

        FocusableButton {
            id: second
            KeyNavigation.tab: third
            text: "Second"
            onClicked: console.log("Second")
        }
```

continued...

Tab and Arrow Key Navigation

```
    FocusableButton {
      id: third
      KeyNavigation.tab: first     3
      text: "Third"
      onClicked: console.log("Third")
    }
    FocusableButton {              4
      focusOnClick: false          5
      text: "Make First " +  (first.visible? "Invisible": "Visible")
      onClicked: {
        first.visible = !first.visible
        console.log("first.visible set to: " + first.visible)
      }
    }
    FocusableButton {              6
      focusOnClick: false          7
      text: (first.enabled? "Disable": "Enable") + " First"
      onClicked: {
        first.enabled = !first.enabled
        console.log("first.enabled set to: " + first.enabled)
      }
    }
  }
}
```

Focus Scopes and Item Hierarchy

Important things to know about keyboard scope in QML:

- From the application's point of view, the state of **focus** and **activeFocus** properties of all items in a QML scene are always consistent.
- Even when something changes, the QML runtime first makes all the necessary updates, and only then emits the notification signals.
- Even in **onFocusChanged** or **onActiveFocusChanged**, there can never be any item with two focus scope children having focus at the same time, or two items (anywhere within the scene) with active focus.

Focus Scopes and Item Hierarchy

```
FocusScope {
    width: row.width + 100; height: row.height + 100
    focus: true
    onActiveFocusChanged: assert()
    Rectangle {
        anchors.fill: parent; color: "grey"
    }
    Row {
        id: row
        anchors.centerIn: parent
        spacing: 50
        Input {
            id: input1
            width: 100
            focus: true
            onFocusChanged: assert()
            onActiveFocusChanged: assert()
        }
        Input {
            id: input2
            width: 100
            onFocusChanged: assert()
            onActiveFocusChanged: assert()
        }
```

Focus Scopes and Item Hierarchy

```
    }
    property bool noFocus: !input1.focus && !input2.focus
    property bool excessiveFocus: input1.focus && input2.focus
    property bool noActiveFocus: activeFocus &&
                            (!input1.activeFocus && !input2.activeFocus)
    function assert() {
        if (noFocus || excessiveFocus || noActiveFocus)
            console.log("Inconsistent focus!")
    }
    onNoFocusChanged: if (noFocus) console.log("no focus")
    onExcessiveFocusChanged: if (excessiveFocus) console.log("excessive focus")
    onNoActiveFocusChanged: if (noActiveFocus) console.log("no active focus")
}
```

Focus Scopes and Item Hierarchy

The focus-related properties of a scene item cannot be brought into an inconsistent state, even if there are changes to the item with active focus, or to the window's active state.

Knowing that items' focus properties are always in a consistent state can be extremely useful in several situations, for example:

- If an item loses focus, you can always obtain the state of other items from the signal handler
- Sometimes it is necessary to differentiate between focus acquisition by another item and the whole window becoming inactive, and not only bind to the root focus scope's **activeFocus**, but also make some changes to **onFocusChanged** or **onActiveFocusChanged** code.

Focus Scopes and Item Hierarchy

Other things to note about focus scopes:

- Not every focus scope is a **FocusScope**. Actually, on the C++ side, for an item to behave like a focus scope it is enough to have the **QQuickItem::ItemIsFocusScope** flag set.
- **QQuickFocusScope** (known as FocusScope on the QML side) differs from the plain **QQuickItem** it inherits only by setting this flag in the constructor.
- The built-in view elements (**ListView**, **GridView** and **PathView**) are also focus scopes which inherit from a "private" **QQuickItemView** class that sets the flag.
- Loader's implementation (**QQuickLoader**) also sets the flag.

Focus in Dynamic Item Hierarchies

- When items change focus, QML focus behavior is quite complicated.
- It becomes really obscure when you attempt to add items with focus.
- For dynamically created items, several ways to give focus (and active focus) to an instantiated item may seem obvious, but not all of them appear to actually work.

Focus in Dynamic Item Hierarchies

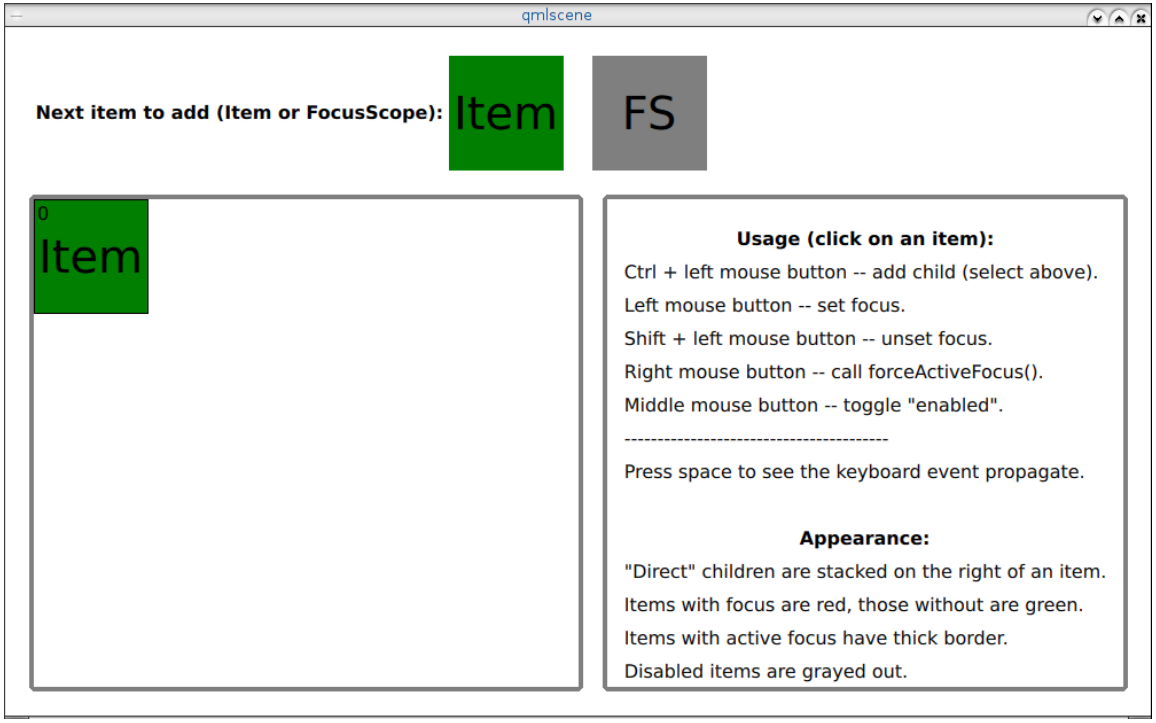
```
FocusRect {
    id: root
    width: 300; height: 300
    focus: true
    Component {
        id: focusedComponent
        FocusRect { focus: true }
    }
    Component {
        id: noFocusComponent
        FocusRect {}
    }
    function createAddItem(component, createWithFocus, number) {
        var nextXY = children.length*25 + 25
        var properties = createWithFocus
                ?{ "x": nextXY, "y": nextXY, "text": number, "focus": true }
                :{ "x": nextXY, "y": nextXY, "text": number }
        return component.createObject(root, properties)
    }
```

Focus in Dynamic Item Hierarchies

```
    Keys.onDigit1Pressed: createAddItem(focusedComponent, false, 1)
    Keys.onDigit2Pressed: createAddItem(focusedComponent, true, 2)
    Keys.onDigit3Pressed: createAddItem(noFocusComponent, false, 3)
    Keys.onDigit4Pressed: createAddItem(noFocusComponent, true, 4)
    Keys.onDigit5Pressed: createAddItem(noFocusComponent, false, 5).focus = true
}
```

More on Focus Scopes

- Nested focus scope behavior is not always obvious.
- QML comes with a simple visual tool for testing assumptions about various focus situations.



Handling Focus and Keyboard Input in Standard Items

The **onAccepted** handler conveniently handles both Enter and Return key presses in **TextInput**, but has the following peculiarities:

- **onAccepted** will not be called if the relevant key press event (Enter or Return) is accepted by the item (this is currently not documented explicitly).
- **onAccepted** itself is not a keyboard event handler: there is no event parameter supplied, and thus no way to accept the event and stop it from propagating to the parent item.

Debugging Keyboard Focus

- Since QML keyboard focus obeys somewhat complicated rules, and due to the "invisible" nature of focus itself, simpler debugging means (like outputting state changes to the console, or stepping through code and watching variable values) may prove ineffective in non-trivial cases, when several-level deep scope hierarchies mix with keyboard navigation and event propagation.
- Things can get especially troublesome when some focus changes trigger further ones via **onFocusChanged()** and **onActiveFocusChanged()** handlers, leading to mysterious binding loop warnings.

Debugging Keyboard Focus

An interesting solution to the problem of debugging focus is using visual debug elements to display the state of items' **focus** and **activeFocus** properties:

- Focus debug items can display the state of non-visual items as well as visual ones.
- There's no need to devote extra screen estate to the debug items, as they can be small (conveying just two bits of information per item) and naturally placed on top of the items they relate to.
- Having all necessary focus state displayed on top of the "debugged" items allows to track it with ease, comfortably checking one's assumptions about the application focus state and behavior and catching (otherwise tricky) errors early.

Keyboard Event Forwarding

Qt documentation on keyboard event forwarding is only one paragraph long (plus a small code example), but the forwarding logic has important nuances. Getting to know the not yet documented peculiarities of key event forwarding may be helpful to:

- Avoid possible traps it introduces even in simple cases.
- Implement complex keyboard event handling logic with much less code.
- Achieve what is otherwise impossible, like sending input key event sequence to a **TextInput** or an alike element with it correctly handling special keys like backspace and arrows.

Keyboard Event Forwarding

Forwarded key event processing follows these rules:

- If the event is not accepted by the item it is forwarded to, it propagates up the parent chain until it is accepted, or the next item in chain is the one with active focus.
- If there are no more items in the forward chain, an event which is not accepted yet is processed by the handlers of the item with active focus.
- The forwarded event may go through the same handler multiple times, if the items in the **Keys.forwardTo** list share a common ancestor.

Keyboard Event Forwarding

- Items without active focus still forward key events, which may highly complicate event handling (unless forwarding is disabled in the absence of active focus).
- Disabled items still get forwarded keyboard input, but invisible items don't.
- Built-in text editing items can be fed forwarded input, **TextInput**'s **onAccepted()** will be triggered when a forwarded Enter or Return press is received.

Keyboard Input Simulation

Sometimes it is necessary to send a keyboard event to an item without the user actually pressing keys. For testing scenarios, Qt offers the following well-documented options:

- On the QML side, TestCase item (http://qt-project.org/doc/qt-5.1/qtquick/qml-testcase.html, part of QtTest module), provides means to emulate both keyboard and mouse input.

- On C++ side, QTest namespace (http://qt-project.org/doc/qt-5.0/qttestlib/qtest.html) gives similar possibilities.

Keyboard Input Simulation

- However, for the production setup, there are no pure-QML means to simulate keyboard input, and on the C++ side of things, some of seemingly natural approaches like using **QApplication:: sendEvent()** do not work (though, strangely, work with mouse input).

- A working C++ side solution involves using **bool QQuickWindow:: sendEvent(QQuickItem * item, QEvent * e)** to an item obtained (after a cast) from **QGuiApplication::focusObject()**.

Summary

1. Understanding keyboard focus handling requires knowledge of focus scopes.
2. Enabled and visible properties affect item's focus, as well as keyboard navigation and key event forwarding.
3. Using some of the built-in items, like **ListView** or **TextInput**, may require keeping in mind slight peculiarities of their keyboard handling behavior.
4. Sometimes, the best way to debug keyboard focus is doing it "visually".
5. Key events can easily be simulated with C++.

Back-up Slides