QtQuick for Complex Applications

Concepts & Best Practices

Andreas Cord-Landwehr

February 1, 2015

FOSDEM, Bruxelles

About Me

And About the Talk

Me = Andreas <CoLa> Cord-Landwehr

- KDE developer since ≈ 4 years
- doing stuff with QtQuick for ≈ 2 years
- PhD student, doing "strange things" with networks and algorithmic game theory
  ... and this fall/after my PhD, one can hire me :)

This Talk...

1. will tell you/remind you what QtQuick is and why it is useful
2. gives walk-through important topics & techniques when working with QtQuick
3. has no live-coding, but an example:
   https://github.com/cordlandwehr/example-fosdem-2015
4. is focused on people with ≈basic QtQuick knowledge
5. will cover:
   - the interplay of C++ code and QtQuick
   - the basic design patterns needed for using QtQuick
   - best practices for hybrid C++/QtQuick applications
6. is available on the FOSDEM website

About Me

And About the Talk

Me = Andreas <CoLa> Cord-Landwehr

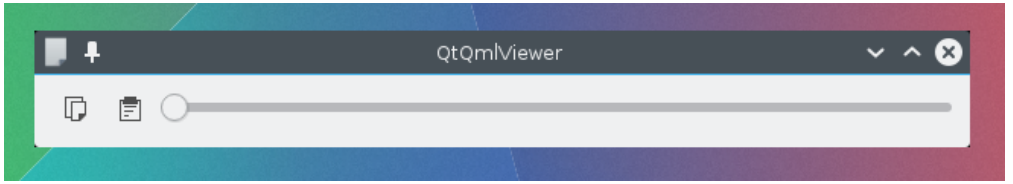
- KDE developer since ≈ 4 years
- doing stuff with QtQuick for ≈ 2 years
- PhD student, doing "strange things" with networks and algorithmic game theory . . . and this fall/after my PhD, one can hire me :)

This Talk. . .

1. will tell you/remind you what QtQuick is and why it is useful
2. gives walk-through important topics & techniques when working with QtQuick
3. has no live-coding, but an example: `https://github.com/cordlandwehr/example-fosdem-2015`
4. is focused on people with ≈basic QtQuick knowledge
5. will cover:
   - the interplay of C++ code and QtQuick
   - the basic design patterns needed for using QtQuick
   - best practices for hybrid C++/QtQuick applications
6. is available on the FOSDEM website

QtQuick ≠ QML

Basic Notions



1. **QML** = name of a declarative language that also allows imperative JavaScript expressions
2. **QtQuick** = toolkit for QML, allowing to develop graphical interfaces

Further reading: https://doc.qt.io/qt-5/qmlapplications.html

Simple Code Example of QML Code

```
1  import QtQuick 2.1
2  import QtQuick.Controls 1.3
3  import QtQuick.Layouts 1.0
4
5  ApplicationWindow {
6  ToolBar {
7      RowLayout {
8          anchors.fill: parent
9          spacing: 2
10         ToolButton { iconName: "edit-copy" }
11         ToolButton { iconName: "edit-paste" }
12         Slider { Layout.fillWidth: true }
13     }
14 } }
```

QtWidgets or QtQuick?

Two Different Concepts

QtWidgets: rock stable and sufficient for many use cases

- discrete forms/pages, each containing static controls (known as widgets)
- presenting new forms to user = new window/dialog that replaces current one
- created in C++ (resp. language bindings) or by compiling XML files (UI-files)
- common sets of widgets (along with theme) gives good graphical consistency between apps with same toolkit

QtQuick: declarative and powerful

- create interfaces by describing them (= declarative language)
- make changes easier to understand for humans (= animations instead of discrete changes)
- do interface once for different form factors: touch, desktop
- QtQuick enforces clear separation of UI and data

QtWidgets or QtQuick?

Two Different Concepts

QtWidgets: rock stable and sufficient for many use cases

- discrete forms/pages, each containing static controls (known as widgets)
- presenting new forms to user = new window/dialog that replaces current one
- created in C++ (resp. language bindings) or by compiling XML files (UI-files)
- common sets of widgets (along with theme) gives good graphical consistency between apps with same toolkit

QtQuick: declarative and powerful

- create interfaces by describing them (= declarative language)
- make changes easier to understand for humans
  (= animations instead of discrete changes)
- do interface once for different form factors: touch, desktop
- QtQuick enforces clear separation of UI and data

How to Use QtQuick in a (complex) Application

My Notion of "Complex"

This Talk's Definition: Complex application ≈ something complex enough to spend time setting up a build system.

Typical non-complex applications

- plasmoids/desktop applets
- mobile applications

Benefits of hybrid C++/QML applications

1. compared to QtWidget based UIs, allows much more flexibility
2. results in code that can be unit tested
3. much better performance (no need for tools like qtquickcompiler to speed-up JS)
4. your (modern) compiler helps discovering tons of issues

This Talk's Example

Yeah, this example is actually non-complex...

Example: A Visual Box Placement Editor

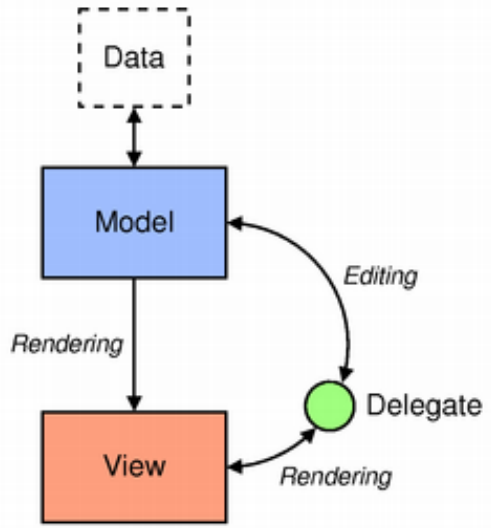
- (blue) whiteboard where we can place (yellow) boxes
- list-view that displays all box-coordinates

Topics for the rest of this talk:

1. How to expose data to the QML engine?
2. How to write data back to the model?
3. How to combine QtQuick with QtWidgets?
4. Some Best-Practices for complex UIs?

Models and Views in QtQuick

Implementation of the Proxy Pattern



Model contains the data and its structure

View a container that displays the data

Delegate dictates how the data should appear in the view

In Our Example

- Data = Boxes + BoxManager (C++)
- Model = BoxModel (C++ with QML interfaces)
- View = ListView (QML)
- Delegate = Rectangles and Labels (QML)

Further Reading:
`https://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html`

Models and Views in Qt Quick

How it is done in code: a Box (boring) (1/4)

```
class Box : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QPointF position READ position WRITE setPosition
               NOTIFY positionChanged)

public:
    explicit Box(QObject *parent = Q_NULLPTR);
    ~Box();
    QPointF position() const;
    void setPosition(const QPointF &position);

Q_SIGNALS:
    void positionChanged();

private:
    Q_DISABLE_COPY(Box)
    QPointF m_position;
};
```

Models and Views in QtQuick

How it is done in code: the BoxManager (mostly boring) (2/4)

```
class BoxManager : public QObject
{
    Q_OBJECT

public:
    explicit BoxManager(QObject *parent = Q_NULLPTR);
    ~BoxManager();
    Box * createBox();
    void removeBox(Box *box);
    QList<Box*> boxes() const;

Q_SIGNALS:
    void boxAdded();
    void boxAboutToBeAdded(Box*,int);
    void boxRemoved();
    void boxAboutToBeRemoved(int);

private:
    Q_DISABLE_COPY(BoxManager);
    QList<Box*> m_boxes;
};
```

Models and Views in QtQuick

How it is done in code: the BoxModel (important!) (3/4)

```
class BoxModel : public QAbstractListModel
{
    Q_OBJECT
public:
    enum boxRoles {
        PositionRole = Qt::UserRole + 1,
        DataRole
    };
    [...]
    virtual int rowCount(const QModelIndex &parent = QModelIndex())
        const Q_DECL_OVERRIDE;
    virtual QVariant data(const QModelIndex &index,
        int role = Qt::DisplayRole) const Q_DECL_OVERRIDE;
    virtual QHash<int,QByteArray> roleNames() const Q_DECL_OVERRIDE;
    [...]
private Q_SLOTS:
    void boxAboutToBeAdded(Box *box, int index);
    void onBoxAdded();
    void onBoxAboutToBeRemoved(int index);
    void emitBoxChanged(int row);
[...]
```

Caution: important (yet boring) code was removed!

Models and Views in QtQuick

How it is done in code: the UI code for the whiteboard (4/4)

```
[...]
Repeater {
    model: BoxModel {
        boxManager: globalBoxManager
    }
    Rectangle {
        width: 20; height: 20
        color: "yellow"
        border.width: 2
        property Box box: model.dataRole
        x: box.position.x
        y: box.position.y
    }
}
```

Remaining Questions

1. how can we tell the engine about the globalBoxManager?
2. how can we tell the box manager when a new box shall be created?

Models and Views in QtQuick

How it is done in code: the UI code for the whiteboard (4/4)

```
[...]
Repeater {
    model: BoxModel {
        boxManager: globalBoxManager
    }
    Rectangle {
        width: 20; height: 20
        color: "yellow"
        border.width: 2
        property Box box: model.dataRole
        x: box.position.x
        y: box.position.y
    }
}
```

Remaining Questions

1. how can we tell the engine about the globalBoxManager?
2. how can we tell the box manager when a new box shall be created?

Structure of a Hybrid C++/QML App

QtQuick Engine Lifting

```
qmlRegisterType<BoxManager>("org.kde.fosdemexample", 1, 0, "BoxManager");
qmlRegisterType<BoxModel>("org.kde.fosdemexample", 1, 0, "BoxModel");
qmlRegisterType<Box>("org.kde.fosdemexample", 1, 0, "Box");
```

```
m_widget = new QQuickWidget;
m_boxManager = new BoxManager;

// register box manager globally in QML Context
// set this before loading the scene
m_widget->rootContext()->setContextProperty("globalBoxManager", m_boxManager
    );
m_widget->setSource(QUrl::fromLocalFile("path/to/my/qml/Scene.qml"));

// listen to context signals
connect(m_widget->rootObject(), SIGNAL(createBox(qreal,qreal)),
    this, SLOT(createBox(qreal,qreal)));
```

Hence: root context must provide signal createBox(x,y)

Live Demo of Example Application

Keep your Fingers Crossed

. . . hopefully :)

Working with Complex Objects

Data Access → Bypassing the Proxy Pattern

There are two ways for accessing your (Q)Objects:

1. add an access role to your model for every property
2. provide "raw" access role and equip object with Q_PROPERTIES (← my choice)

Announce roles:

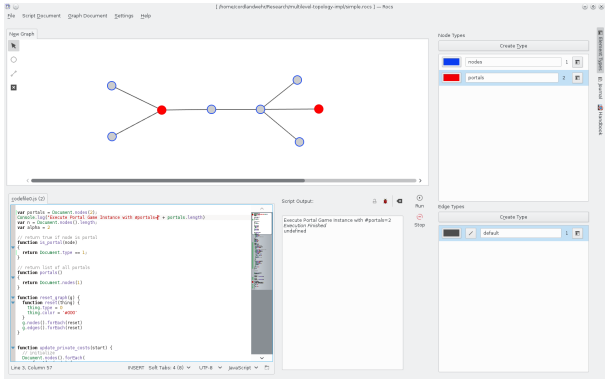
```
QHash< int , QByteArray > BoxModel::roleNames() const {
    QHash<int , QByteArray> roles;
    roles[DataRole] = "dataRole";
    return roles;
}
```

Provide access:

```
QVariant BoxModel::data(const QModelIndex &index, int role) const
{   [...]
    if (!index.isValid() || index.row() >= m_boxManager->boxes().count())
        return QVariant();
    Box * const box = m_boxManager->boxes().at(index.row());
    switch(role) {
    case DataRole:
        return QVariant::fromValue<QObject*>(box);
    case [...]
    }
}
```

Structure of a Hybrid C++/QML App

Mix with QtWidgets



Since Qt 5.3: QtQuickWidgets

- provides a widget for displaying a QtQuick user interface → you can add QtQuick in a QtWidget based UI
- convenience wrapper for QQuickWindow which will automatically load and display a QML scene

Note:

- QQuickWidget disables the threaded render loop on all platforms
- Qt 5.4 fixes *a lot* of issues

Further reading: `http://doc.qt.io/qt-5/qquickwidget.html`

Wrapping-Up the Talk

Keep QML files maintainable

QtQuick allows a lot

- fancy interfaces
- better UI design experience
- clear separation of logic and UI

. . . but requires discipline

1. Keep logic in C++ and unit test it
2. Keep file sizes moderate (rule of thumb: < 500 LOC)
3. Keep a common coding and naming style
   $\rightarrow$ "private" variable naming space, root ids. . .
4. for complex interactions: think about using state machines
   $\rightarrow$ without them, the editor functionality would not be maintainable in Rocs

Thank you for your attention!

Andreas Cord-Landwehr
E-mail: cordlandwehr@kde.org