Qt in Education

Qt Quick

NOKIA

Introducing Qt Quick

- C++ is great for developing applications
  - Data structures
  - Algorithms
  - Business logic
  - Structured user interfaces
- C++ is not so great for modern device user interfaces
  - Many objects active in parallel
  - Many, potentially overlapping states
  - Timer-driven, fluid changes

Introducing Qt Quick

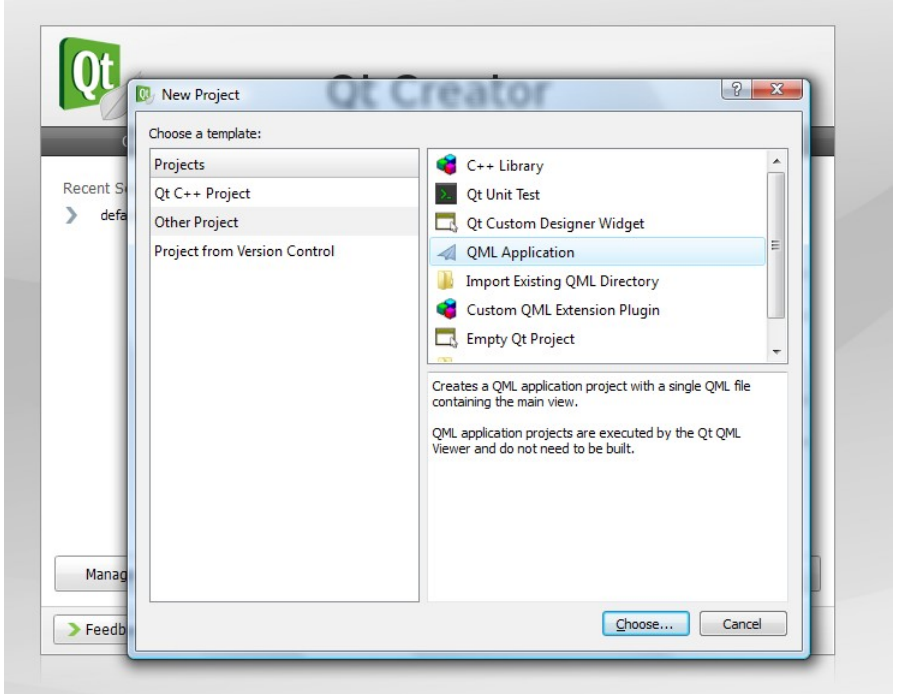
- Using Qt Quick, the business logic and performance critical operations can be implemented in C++
- The user interface can be written using QML
  - *Qt Meta-object Language*
  - Declarative
  - Based on JavaScript

Introducing Qt Quick

- Qt Quick consists of
  - QML – the language
    - Designed for building device user interfaces
    - Can be used in other application too
  - Qt Declarative – the Qt module
    - Contains the QML engine, context and view
    - Qt bindings for QML
    - Mechanisms for integrating C++ and QML
  - Tooling support in Qt Creator (coming)

Working with QML

- Qt Creator 2.0 supports QML
  - Can create QML projects
  - Can run and debug QML



Introducing QML

- QML is a declarative language based on JavaScript

Import QtQuick components

```
import QtQuick 1.0

Rectangle {
    width: 200
    height: 200
    color: "red"
}
```

Declare a Rectangle element – i.e. create an object instance

Bind properties to values

Component names always start with capital letters

Importing Resources

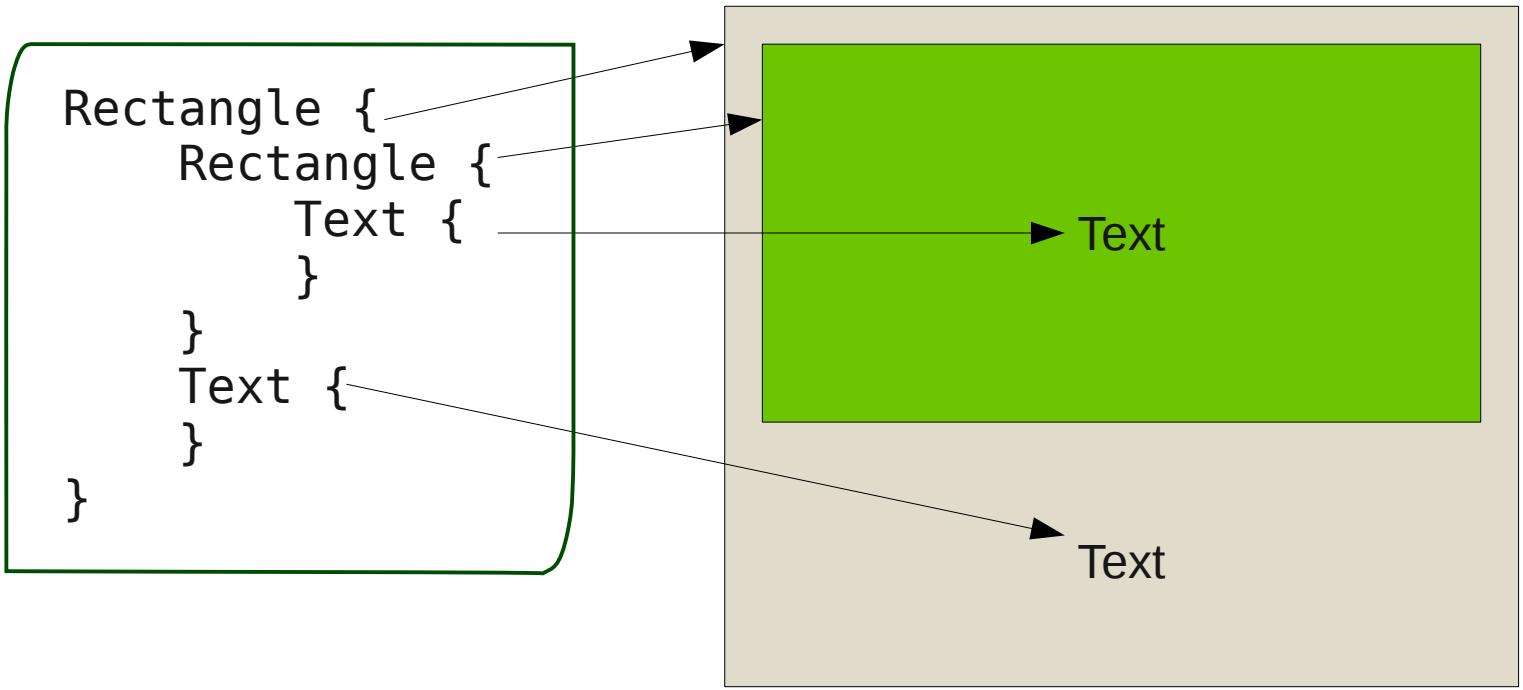
- Importing component definitions
- The import directive imports:
  - Component classes from C++ modules
  - Other QML modules
  - JavaScript files

```
import Qt 4.7
import MyCppClasses 1.2
import "from-qml"
import "scripts.js"
```

- When importing C++ modules, the version must always be specified

Creating Object Hierarchies

- When declaring elements inside other element declarations, you create object hierarchies



Navigating the Objects Tree

- It is possible to refer to the parent object using the parent name

```
Rectangle {
    Rectangle {
        width: parent.width

        Text {
            color: parent.color
        }
    }
    Text {
    }
}
```

Naming Elements

- Using the id property, you can name elements

```
Rectangle {
    id: outerRectangle
    ...
```

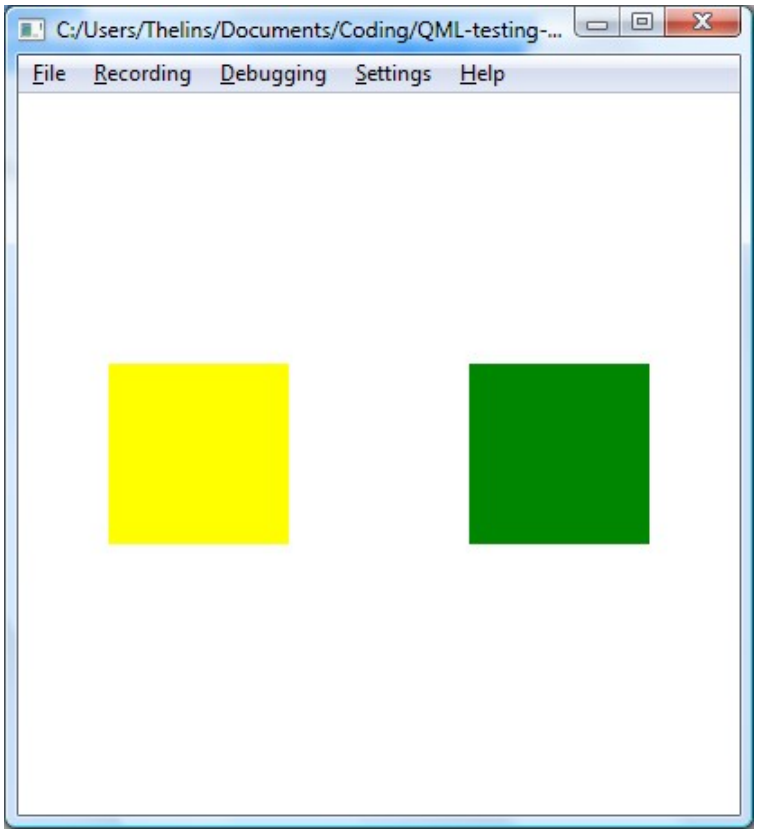
- You can then refer to them by name

```
{
    height: outerRectangle.height
    ...
```

Binding Values

- In QML, values are bound, not assigned
  - Changing input on the right side of the ":" operator updates the left side

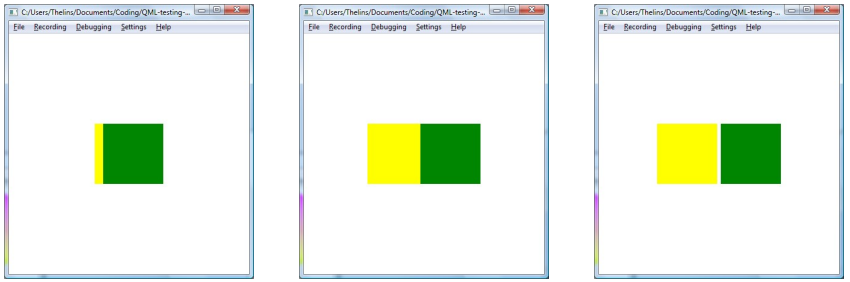
```
Rectangle {
    id: firstRect
    x: 10
    ...
}

Rectangle {
    x: 400 - firstRect.x
    ...
}
```



Animating Values

- Property values can be animated

```
Rectangle {
    id: firstRect
}

Rectangle {
    x: 400 – firstRect.x
    ...
}

SequentialAnimation {
    running: true
    loops: Animation.Infinite
    NumberAnimation { target: firstRect; property: "x"; to: 300 }
    NumberAnimation { target: firstRect; property: "x"; to: 50 }
}
```



Available Components

- Qt provides a range of components
  - Rectangle
  - Text
  - Image
  - BorderImage

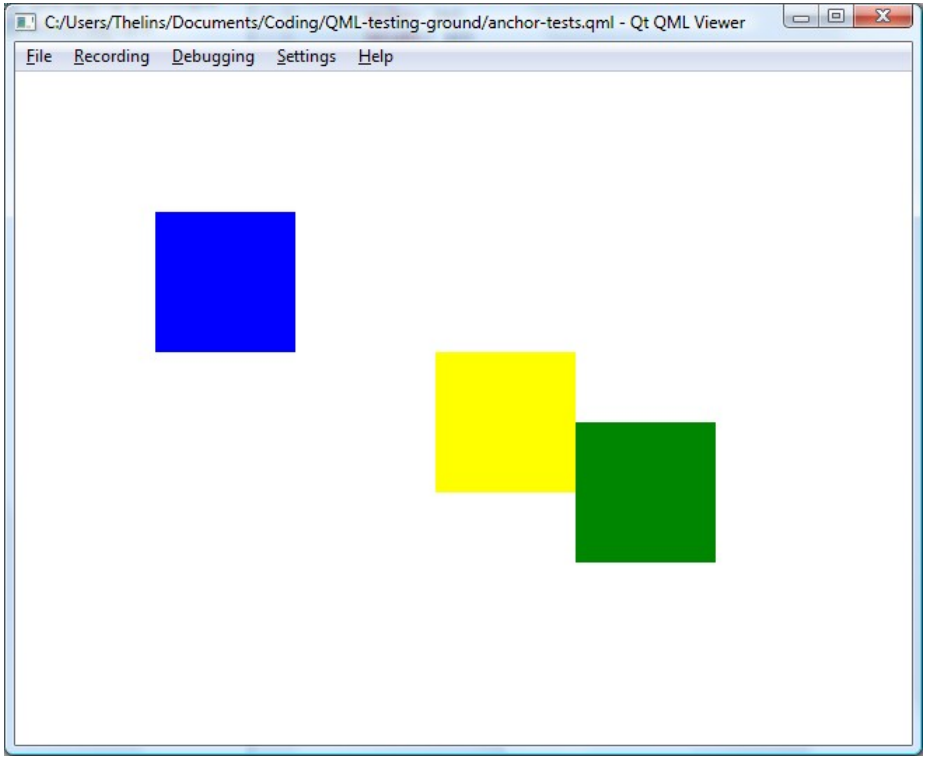
Setting up an Element

- There are a number of common properties for these components
  - x, y, width, height
  - color, opacity
  - visible
  - scale, rotation

Anchor Layouts

- Anchor layouts can be used to anchor elements to each other

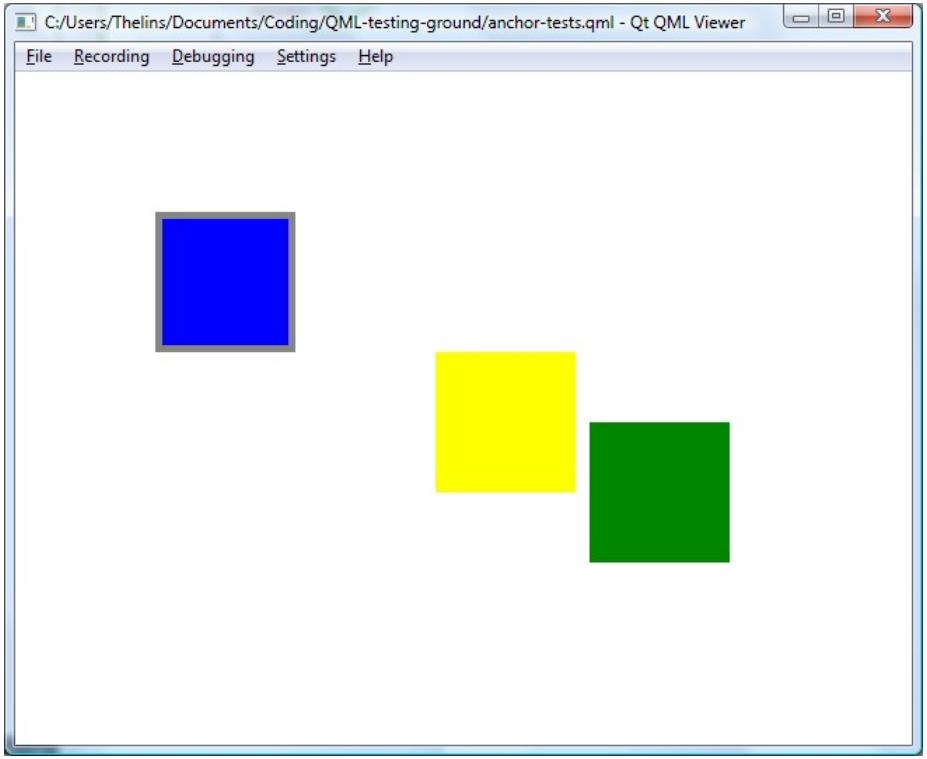
```
Rectangle {
    Rectangle {
        anchors.fill: parent
        ...
    }
    ...
}

Rectangle {
    id: leftRectangle
    ...
}

Rectangle {
    anchors.left: leftRectangle.right
    ...
}
```



Layouts and Margins

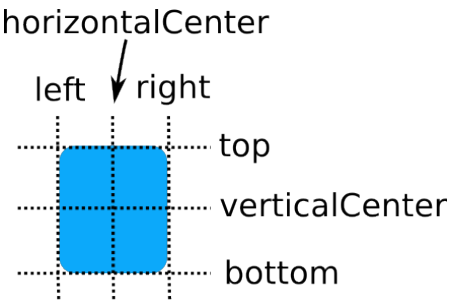
- You can combine anchor layouts with margins

```
Rectangle {
    Rectangle {
        anchors.fill: parent
        anchors.margins: 5
        ...
    }
    ...
}

Rectangle {
    id: leftRectangle
    ...
}

Rectangle {
    anchors.left: leftRectangle.right
    anchors.leftMargin: 10
    ...
}
```

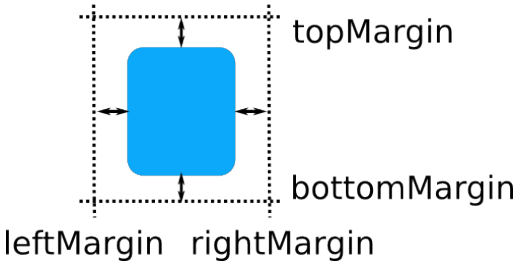


Anchor Layout Properties

- You can anchor items to
  - left, top, right, bottom
  - verticalCenter, horizontalCenter
  - baseline

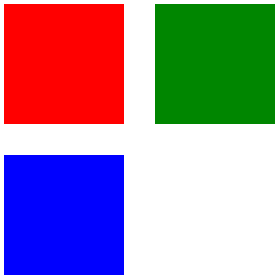


- You can specify individual margins or anchors.margins



Other Layouts

- Using the Grid, Row and Column containers, classic layouts can be built
  - Does not work if x or y are bound
  - The spacing property is available for all
  - The columns property controls the size of grids



```
Grid {
    columns: 2
    spacing: 5

    Rectangle { width: 20; height: 20; color: "red" }
    Rectangle { width: 20; height: 20; color: "green" }
    Rectangle { width: 20; height: 20; color: "blue" }
}
```

Break

Adding Interaction

- Interaction is handled through areas separated from the visuals
  - MouseArea – an area accepting mouse events
  - GestureArea – an area accepting gesture events
    - Requires touch events
    - Single touch devices might only provide mouse events, check your device's documentation
  - Keyboard events are handled through focus

Creating a Button

- You can build a button from a Rectangle, Text and MouseArea

```
Rectangle {
    width: 200; height 100;
    color: "lightBlue"

    Text {
        anchors.fill: parent
        text: "Press me!"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: { parent.color = "green" }
    }
}
```

JavaScript

```
Rectangle {
    width: 200; height 100;
    color: "blue"

    Text {
        anchors.fill: parent
        text: "Press me!"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: { parent.color = "green" }
    }
}
```

What happened here?
We bound an anonymous JavaScript function to a signal.

Building Components

- Having to create each button as a set of three elements is not a feasible solution

- It is possible to create components in QML
- A component can then be instantiated as an element

- Components can be kept in modules that are included into your QML files

A Button Component

- Place the button in the Button.qml file

```
import Qt 4.7

Rectangle {
    width: 200; height: 100;
    color: "lightBlue"
    property alias text: innerText.text

    Text {
        id: innerText
        anchors.fill: parent
    }

    MouseArea {
        anchors.fill: parent
        onClicked: { parent.color = "green" }
    }
}
```

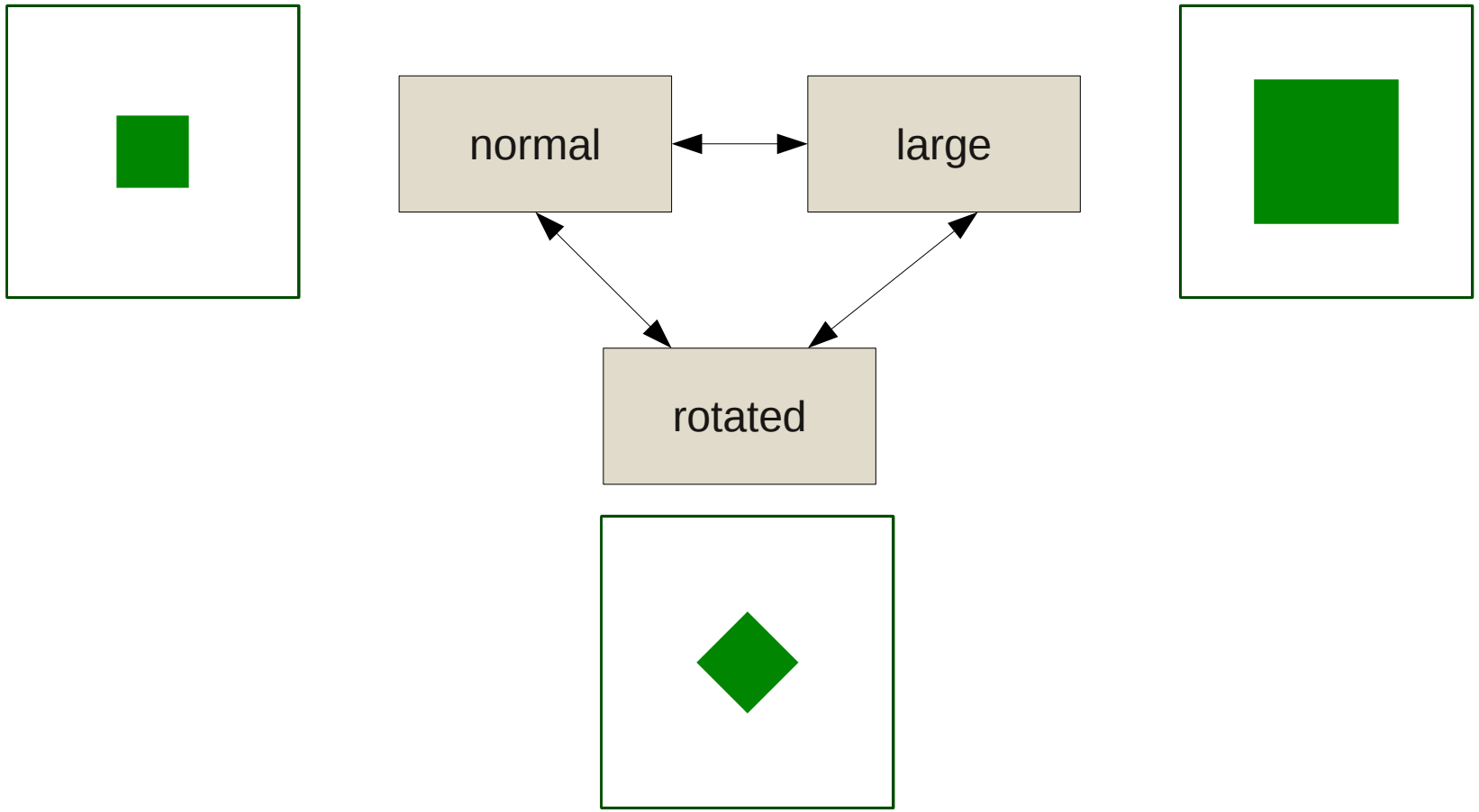
A Button Component

- Instantiate buttons from your main QML file
  - The main QML file must be placed in the same directory as Button.qml
  - If not, you must import the directory containing Button.qml as a Module

```
import Qt 4.7

Row {
    spacing: 10

    Button { text: "Oslo" }
    Button { text: "Copenhagen" }
    Button { text: "Helsinki" }
    Button { text: "Stockholm" }
}
```

States

- Using states, you can easily make smooth transitions between sets of property values



Defining States

- The states property holds the states

```
import Qt 4.7

Rectangle {
    width: 400; height: 400;

    Rectangle {
        id: myRect
        width: 100; height: 100;
        anchors.centerIn: parent
        color: "green";
    }

    states: [
        State { name: "normal" },
        State { name: "large" },
        State { name: "rotated" }
    ]
}
```

Defining States

- Each state contains a set of property changes

```
Rectangle {
    states: [
        State { name: "normal"
            PropertyChanges {
                target: myRect
                width: 100; height: 100;
                rotation: 0
            }
        },
        ...
    ]
}
```

Making Smooth Transitions

- The transitions property defines how to animate properties between states

```
Rectangle {
    transitions: [
        Transition {
            from: "*"; to: "normal"
            NumberAnimation {
                properties: "width, height"
                easing.type: Easing.InOutQuad
                duration: 1000
            }
            NumberAnimation {
                properties: "rotation"
                easing.type: Easing.OutElastic
                duration: 3000
            }
        },
        ...
    ]
}
```

Switching Between States

- Set the state property

```
import Qt 4.7

Rectangle {
    ...
    MouseArea {
        anchors.fill: parent
        onClicked: { if(parent.state == "normal") {
                        parent.state = "rotated";
                    } else if(parent.state == ...
        }
}
```

Switching Between States

- Or bind the state property to a value...

```
import Qt 4.7

Rectangle {
    ...
    state: myState
}
```

- ...which can form the link to C++

Global Variables

- It is possible to bind to values exposed from JavaScript or C++
- By binding to values from C++, the business logic can control the state
- QML only controls the user interface, including transitions and effects

Integrating QML and C++

- QML is executed by an QDeclarativeEngine
- Each component can be created
- The common component is a QGraphicsObject, but can be any QObject

```
QGraphicsScene *scene = myExistingGraphicsScene();

QDeclarativeEngine *engine = new QDeclarativeEngine;

QDeclarativeComponent component(engine, QUrl::fromLocalFile("myqml.qml"));
QGraphicsObject *object =
    qobject_cast<QGraphicsObject *>(component.create());

scene->addItem(object);
```

Integrating QML and C++

- The convenience widget QDeclarativeView can be used
  - Contains an engine
  - Handles the creation of components

```
QDeclarativeView *qmlView = new QDeclarativeView;

qmlView->setSource(QUrl::fromLocalFile("myqml.qml"));
```

Controlling Properties from C++

- The rootContext of an engine can be accessed
- The setContextProperty method can be used to set global variable values

```
QDeclarativeView *qmlView = new QDeclarativeView;

QDeclarativeContext *context = qmlView->rootContext();
context->setContextProperty("myState", QString("normal"));

qmlView->setSource(QUrl::fromLocalFile("myqml.qml"));
```

Bound, not Assigned

- As QML binds values, instead of assigning them, changing a context property from C++ changes the value in QML

```
void Window::rotateClicked()
{
    QDeclarativeContext *context = qmlView->rootContext();
    context->setContextProperty("myState", QString("rotated"));
}

void Window::normalClicked()
{
    QDeclarativeContext *context = qmlView->rootContext();
    context->setContextProperty("myState", QString("normal"));
}

void Window::largeClicked()
{
    QDeclarativeContext *context = qmlView->rootContext();
    context->setContextProperty("myState", QString("large"));
}
```

Exposing QObject

- Exposing a QObject as a context property, exposes slots

```
QDeclarativeView *qmlView = new QDeclarativeView;

QLabel *myLabel = new QLabel;
QDeclarativeContext *context = qmlView->rootContext();
context->setContextProperty("theLabel", myLabel);
```

```
MouseArea {
    anchors.fill: parent
    onClicked: { theLabel.setText("Hello Qt!"); }
}
```