ICS

QML Programming Fundamentals and Beyond

QML Item Placement

Material based on Qt 5.12

Course Outline

Session 1: April 28, Introduction to QML
- About QML
- Properties
- Basic Types

Session 2: May 1, QML Item Placement
- How to correctly size and place items
- When to use Anchors, Layouts and Positioners

Session 3: May 5, Touch Interaction
- QML Signals
- Touch Events
- Single and Multi-Touch
- Swipe and Pinch Gestures

Session 4: May 8, States & Transitions
- Creating and defining states
- Using Transitions

Session 5: May 15, Custom Items & Components
- Creating your own Components
- Creating a Module

Session 6: May 19, Model / View
- Model / View
- QML Models
- QML Views

Session 7: May 22, C++ Integration
- Why expose C++ to QML
- Exposing C++ Objects
- Exposing C++ Classes

About ICS

ICS Designs User Experiences and Develops Software for Connected Devices

- Largest source of independent Qt expertise in North America since 2002
- Headquartered in Waltham, MA with offices in California, Canada, Europe
- Includes Boston UX, ICS' UX design division

- Embedded, touchscreen, mobile and desktop applications
- Exclusive Open Enrollment Training Partner in North America

Qt Service Partner

BOSTON UX

UX/UI Design and Development for Connected Devices Across Many Industries

Agenda

- How to place QML Items onto the screen with the use of
  - Anchors
  - Absolute x, y and sizing
  - Positioners
  - Layouts

Placing QML Items

Several strategies for item placement

- Set fixed x, y, height and width
  - Also called “Dead Reckoning”
- Define anchors
- Place the items inside a positioning item
- Make use of layouts

Dead Reckoning Layout

```
Item {
  width: 800; height: 400
  Rectangle {
    id:rectangleA
    color: "red"
    height: 50 ; width: 70
    x: 0; y: 0
  }
  Rectangle {
    id:rectangleB
    color: "blue"
    height: rectangleA.height * 2; width: rectangleA.width * 2
    x: 0; y: 100
  }
}
```

Why is dead reckoning bad?

The good:

- It resizes correctly
- It uses bindings so it's "declarative"

The bad:

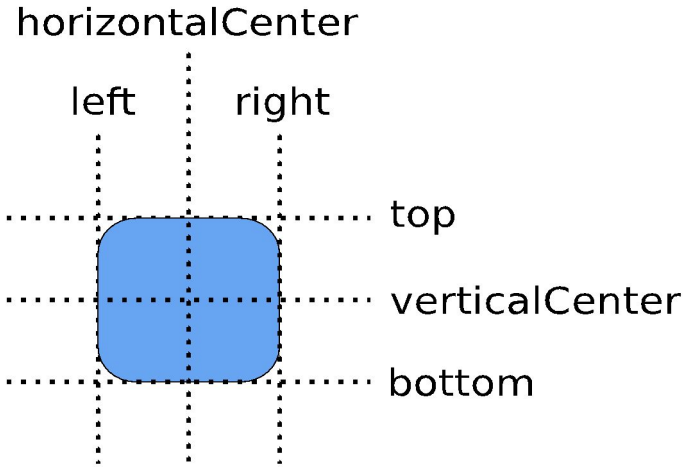
- There are a lot of binding re-calculations
  - Each recalculation is run in JavaScript
- Cascading bindings cause intermediate states

Binding Recalculation

- This example has ~40 items
- If each item needs 2 bindings
  - 80 Recalculations on resize
- Not including intermediate states

Anchors

- Used to position and align items
- Line up the edges or central lines of items
- Anchors refer to
  - Other items (`centerIn`, `fill`)
  - Anchors of other items (`left`,`top`, `etc`)
- Can only anchor to the parent or siblings



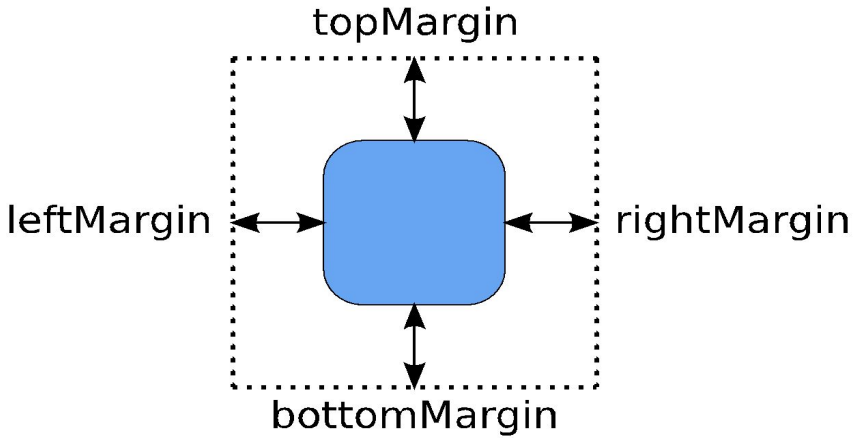
Anchors

```
Text {
    id:textLabel
    y: 34
    text: qsTr("Right-aligned text”)
    color: "green"
    font { family: "Helvetica"; pixelSize: … }
    anchors.right: parent.right
}
```

- Connecting anchors together
- Anchors of other items are referred to directly
  - Use `parent.right`
  - Not `parent.anchors.right`

Anchor Margins

- Used with anchors to add space
- Specifies distances in pixels
- Between items connected with anchors



Margins Example

```
Rectangle {
    width: 400; height: 200; color: "lightblue"
    Image {
        id: book
        source: "../images/book.svg"
        anchors.left: parent.left
        anchors.leftMargin: parent.width / 16
        anchors.verticalCenter: parent.verticalCenter
    }
    Text {
        text: qsTr("Writing"); font.pixelSize: 32
        anchors.left: book.right
        anchors.leftMargin: 32
        anchors.baseline: book.verticalCenter
    }
}
```

Complex Anchors

Set multiple anchors at once

anchors.fill: anotherItemID

- Sets the left, right, top and bottom anchors
- Can use all outer margins

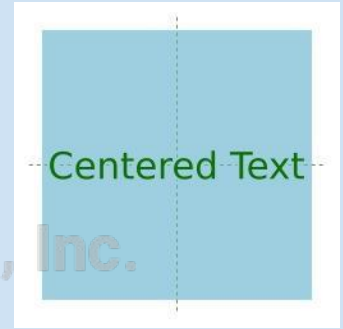
anchors.centerIn: anotherItemID

- Sets both the horizontalCenter and verticalCenter
- Can use horizontal and vertical offsets

Anchors

```
Rectangle {
    id: rectangle1
    width: 400; height: 400
    color: "lightblue"
    Text {
        id: textLabel
        text: qsTr("Centered text"); color: "green"
        font { family: "Helvetica"; pixelSize: … }
        anchors.centerIn: parent

        // SAME AS :
        // anchors.horizontalCenter: parent.horizontalCenter
        // anchors.verticalCenter: parent.verticalCenter
    }
}
```



- Each item can refer to its parent item with the keyword "parent"
- Can refer to ancestors and named children of ancestors

Anchors

```
Rectangle {
    id: rectangle1
    width: 400; height: 400
    color: "lightblue"
    Rectangle {
        id: redRectangle
        color: "red"
        anchors.fill: parent

        // SAME AS
        // anchors.left : parent.left
        // anchors.right : parent.right
        // anchors.top : parent.top
        // anchors.bottom : parent.bottom
    }
}
```

Strategy for Using Anchors

Identify items with different roles in the user interface:

- Fixed items
  - Make sure these have the `id` property defined, unless these items can easily be referenced as parent items
- Items that dominate the user interface
  - Make sure these items have the `id` property defined
  - Other items should react to the size changes of the dominant item
    - These items should be given anchors that refer to a dominant item

Hints and Tips

- Anchors can only be used with the parent and sibling items
- Anchors work on constraints
  - Some items need to have well-defined positions and sizes
  - Items without default sizes should be anchored to fixed or well-defined Items
- Anchors create dependencies on geometries of other items
  - Creates an order in which geometries are calculated
  - Avoid creating circular dependencies
    - e.g., parent → child → parent
- Margins are only applicable if the corresponding anchors are used
  - e.g., `leftMargin` needs the `left` anchor to be defined

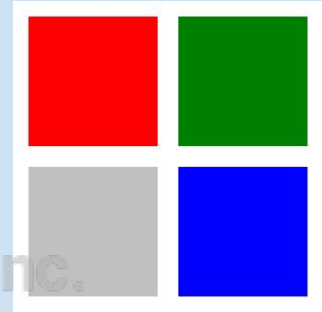
Positioners

Positioners make it easier to work with many items

- Positioners arrange items in standard layouts
  - In a column: `Column`
  - In a row: `Row`
  - In a grid: `Grid`
  - Like words on a page: `Flow`
- Combining these make it easy to layout the lots of items
- Anchors used in Positioners will be applied to all children

Positioning Items

```
Grid {
    x: 15; y: 15; width: 300; height: 300
    columns: 2; rows: 2; spacing: 20
    Rectangle { width: 125; height: 125; color: "red" }
    Rectangle { width: 125; height: 125; color: "green" }
    Rectangle { width: 125; height: 125; color: "silver" }
    Rectangle { width: 125; height: 125; color: "blue" }
}
```



- Items inside a positioner are automatically arranged
  - In a 2 by 2 `Grid`
  - With horizontal/vertical spacing of 20 pixels
- `x,y` is the position of the first item
- Like layouts in Qt

Layouts

- Default behavior is similar to positioners
  - **GridLayout**, **RowLayout**, **ColumnLayout**
- However, can be used in the same way as **`QLayout`** works for widgets
  - The layout automatically defines the size of the items – no anchors or explicit width/height needed
- Just set the **`Layout.fillHeight`** or **`Layout.fillWidth`** to
  - `false` – if you do not want the layout to use all extra space for the item
  - `true` – if you want the extra space to be used to expand the item
  - Compare to **`QSizePolicy::Expanding`**
- Can specify a preferred width and height with **`Layout.preferredHeight`** or **`Layout.preferredWidth`**

Layouts Example

```
GridLayout {
    columns: 3
    …
    Button {
        text: qsTr("Btn 2")
        Layout.fillHeight: true
        … }
    Button {
        text: qsTr("Btn 4")
        Layout.fillWidth: true
        … }
}
```

- Two buttons expand vertically and horizontally
- Other button dimensions are based on the text and font properties

Q&A Session

If you have additional questions or feedback, please contact us at QtTraining@ics.com