CREATING QML CONTROLS FROM SCRATCH

Chris Cortopassi

ICS

Table of Contents

Introduction

Shrink the overall development effort of your next project with a set of common QML controls.

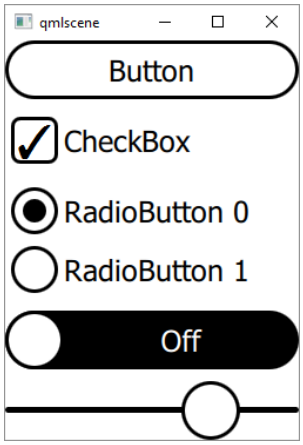
QML provides a powerful and flexible framework for developing user interfaces (UI). The basic elements provided are low level, so you typically need to build up the components of your UI into widget-like controls. Creating a set of common QML controls can greatly reduce the overall development effort of your project.

In this ebook, *Creating QML Controls From Scratch*, we show you how to build 16 bare-bones QML controls to use as a starting point for the controls in a mobile or embedded project. Once created, you can modify these controls to suit your project's specific needs.

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch

Part 0: Getting Started

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch



User interfaces are almost always composed of a set of reusable "controls" such as Button, CheckBox, RadioButton, Switch, Slider, etc. The controls might be designed to be reused only within the same project (e.g. two buttons on a screen), or they can be more general purpose (customizable) controls that can be reused across multiple projects (e.g. Qt Quick Controls).

Qt Quick Controls source code isn't intended to be modified. Rather, the developer writes separate "styling code" to customize. In this series, we will take an alternate "by example" approach where you as the developer have 100% control over the source code, appearance, and behavior of each control.

Rather than writing separate styling code (and learning styling APIs), you can simply modify the source code for our controls directly. Please watch my 10 minute lightning talk from Qt World Summit 2016 for further explanation on the rationale behind this "from scratch" approach.

To reduce them to their essence and keep them clear, simple, and reusable, the controls we create will adhere to the following properties:

1. **No animations:** other than the default scrolling animations provided by Flickable (e.g. ListView and GridView) and unless necessary (e.g. Spinner); however, you can add your own animations.

2. **No states and transitions:** states and transitions may be added and are useful to implement animations.

3. **QML only (no C++):** the focus here is the QML front end; however, a C++ back end can be added.

4. **No image assets (.png, .jpg):** to ensure the controls look good at any size and to avoid any copyright issues; however, you may add icons with the QML Image element. Watch my lightning talk for more information.

5. **Black and white:** to make the controls "style-less" and make it easy for you to add your color scheme (see Styling below) and/or Image .png assets.

6. **Resizable:** each control scales with either the height or width of the root Item so that it looks good at any size (and thus on any screen size and resolution).

Primitives

We will build all controls using only the QML primitives listed below:

1. Item
2. Rectangle
3. Text
4. MouseArea
5. ListView
6. GridView
7. Row
8. Column
9. Grid
10. Repeater
11. Canvas
12. Timer

With the exception of Canvas, all of the above primitives were available in Qt Quick 1.0. So as an added bonus, by replacing “import QtQuick 2.0” with “import QtQuick 1.0”, we also can use our controls in old code using Qt 4 or **qmlviewer**.

Styling

The only primitive items above that actually render pixels on the screen are Rectangle, Text, and Canvas (the rest are for layout or user interaction). Canvas is only used in a couple of cases (PieChart and LineChart) that Rectangle can’t handle, so to “style” (i.e. change colors, fonts, etc.), Rectangle and Text are the only items you need to modify and/or possibly replace with Image .png assets.

Change color by changing Rectangle.color (default ‘white’), Rectangle.border.color (default ‘black’), and Text.color (default ‘black’) where appropriate. If you prefer to change your application’s font in one place instead of setting font.family in every Text element, you can create FontText.qml (below) and use FontText instead of Text in all .qml files.

FontText.qml

```
import QtQuick 2.0

Text {
    font.family: 'Arial'
}
```

Each example will be comprised of:

1. An animated **screen capture** of the control.

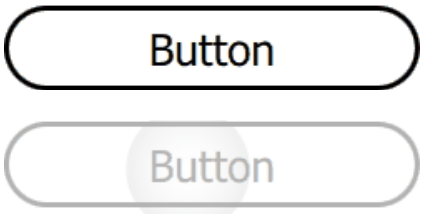
2. A written **description** of how the control is implemented.

3. **Source code** (less than 100 lines) for the control (a .qml file that can be loaded in qmlscene), with public and private sections clearly delimited by comments (since QML doesn’t have public nor private keywords). Note also that the control inherits all public properties, methods, and signals of the root Item, which will at least include those of Item e.g. x, y, width, height, anchors, and enabled.

4. **Example usage** in the file Test.qml (i.e. how to create an instance of the control).

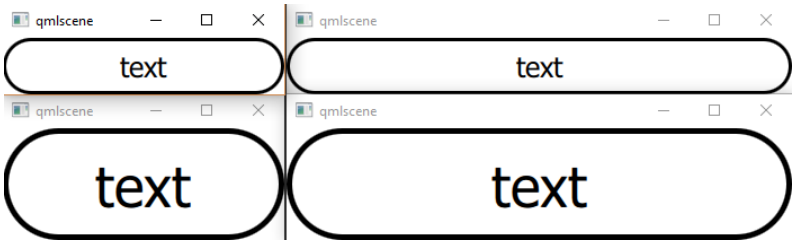
Part 1: Button

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch



Button has a public text property and clicked() signal. It is composed of only a Rectangle, Text, and a MouseArea. The Rectangle's border draws the Button's outline and MouseArea manages the Button's appearance when pressed ("down state") and clicked() signal.

The easiest way to create a down state is to set the root Item's opacity. Other methods include setting Rectangle.color or providing a .png via Image (not recommended for resizing). To allow resizing, the Rectangle's border and font.pixelSize scale with the root Item's height, as seen below.



All QML Items have an enabled property and we exploit it to provide a "disabled state" by setting the root Item's opacity to 0.3 (30%) to create a "faded" look when enabled is set to false.

Button.qml

```
import QtQuick 2.0

Rectangle {
    id: root

// public
    property string text: 'text'

    signal clicked(); // onClicked: print('onClicked')

// private
    width: 500;  height: 100 // default size
    border.color: text? 'black': 'transparent' // Keyboard
    border.width: 0.05 * root.height
    radius:       0.5  * root.height
    opacity:      enabled  &&  !mouseArea.pressed? 1: 0.3 // disabled/pressed state
```

Button (Continued)

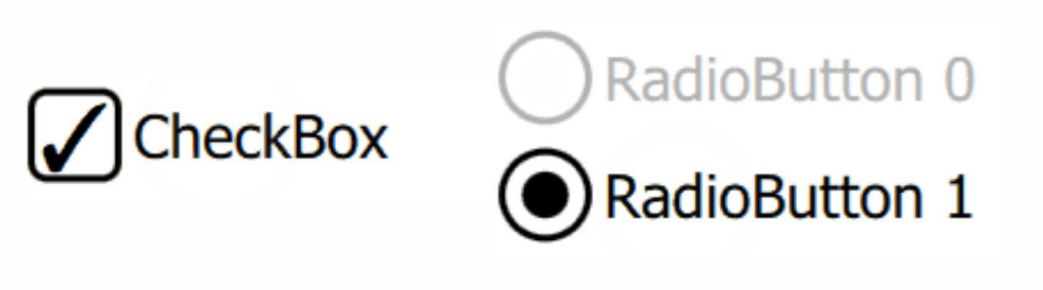
```
    Text {
        text: root.text
        font.pixelSize: 0.5 * root.height
        anchors.centerIn: parent
    }

    MouseArea {
        id: mouseArea

        anchors.fill: parent

        onClicked:  root.clicked() // emit
    }
}
```

Test.qml

```
import QtQuick 2.0

Button {
    text: 'Button'

    onClicked: print('Button onClicked')
}
```

Part 2: CheckBox and RadioButton

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-checkbox-and-radiobutton



This time we'll implement a CheckBox. We'll also get RadioButton almost for free. CheckBox is similar to Button with the exception that it holds checked/unchecked state in the **checked** property. All QML properties have an associated *Changed signal, so the **checkedChanged()** signal causes **onCheckedChanged** to run when the property **checked** changes.

CheckBox also serves as a radio button if a client sets radio: true, which is all RadioButton.qml does. To get multiple radio buttons to act together in a group, put a RadioButton in a ListView delegate with currentIndex indicating the currently selected radio button (see test.qml below).

Since we're not using Image, we used a check ✓ (hexadecimal 2713) unicode glyph to render the check mark as Text, but you might want to replace with an Image .png asset from your designer. The RadioButton dot is implemented with a Rectangle.

CheckBox.qml

```
import QtQuick 2.0

Item {
    id: root

// public
    property string text:    'text'
    property bool   checked: false

    signal clicked(bool checked);   //onClicked:{root.checked = checked;  print('onClicked', checked)}

// private
    property real padding: 0.1    // around rectangle: percent of root.height
    property bool radio:   false  // false: check box, true: radio button

    width: 500;  height: 100                        // default size
    opacity: enabled  &&  !mouseArea.pressed? 1: 0.3 // disabled/pressed state

    Rectangle { // check box (or circle for radio button)
        id: rectangle

        height: root.height * (1 - 2 * padding);  width: height // square
        x: padding * root.height
        anchors.verticalCenter: parent.verticalCenter
        border.width: 0.05 * root.height
        radius: (radio? 0.5: 0.2) * height
```

```
        Text { // check
            visible: checked  &&  !radio
            anchors.centerIn: parent
            text: '\u2713' // CHECK MARK
            font.pixelSize: parent.height
        }

        Rectangle { // radio dot
            visible: checked  &&  radio
            color: 'black'
            width: 0.5 * parent.width;  height: width // square
            anchors.centerIn: parent
            radius: 0.5 * width // circle
        }
    }

    Text {
        id: text

        text: root.text
        anchors {left: rectangle.right;  verticalCenter: rectangle.verticalCenter;  margins: padding * root.height}
        font.pixelSize: 0.5 * root.height
    }

    MouseArea {
        id: mouseArea

        enabled: !(radio  &&  checked) // selected RadioButton isn't selectable
        anchors.fill: parent

        onClicked: root.clicked(!checked) // emit
    }
}
```

RadioButton.qml

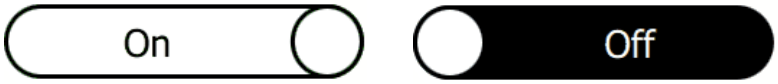
```
import QtQuick 2.0

CheckBox {
    radio: true
}
```

Test.qml

```
import QtQuick 2.0


CheckBox {
    property bool backend: false

    text: 'CheckBox'
    checked:   backend

    onClicked: backend = checked
}

ListView { // RadioButton
    id: radioButtons

    interactive: false

    model: [{text: 'RadioButton 0'}, {text: 'RadioButton 1'}]

    delegate: RadioButton {
        text:      modelData.text
        checked:   radioButtons.currentIndex == index // equality

        onClicked: radioButtons.currentIndex  = index // assignment
    }
}
```

Part 3: Switch

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-switch



Switch is similar to CheckBox with the exception that it has a slidable pill (implemented with a MouseArea and drag property) and no text property. A Switch can be toggled either by tapping or dragging.

Switch.qml

```
import QtQuick 2.0

Rectangle {
    id: root

// public
    property bool checked: false

    signal clicked(bool checked);  // onClicked:{root.checked = checked;  print('onClicked', checked)}

// private
    width: 500;  height: 100 // default size
    border.width: 0.05 * root.height
    radius:       0.5  * root.height
    color:        checked? 'white': 'black' // background
    opacity:      enabled  &&  !mouseArea.pressed? 1: 0.3 // disabled/pressed state

    Text {
        text:  checked?    'On': 'Off'
        color: checked? 'black': 'white'
        x:    (checked? 0: pill.width) + (parent.width - pill.width - width) / 2
        font.pixelSize: 0.5 * root.height
        anchors.verticalCenter: parent.verticalCenter
    }

    Rectangle { // pill
        id: pill

        x: checked? root.width - pill.width: 0 // binding must not be broken with imperative x = ...
        width: root.height;  height: width // square
        border.width: parent.border.width
        radius:       parent.radius
    }

    MouseArea {
        id: mouseArea

        anchors.fill: parent

        drag {
            target:   pill
            axis:     Drag.XAxis
            maximumX: root.width - pill.width
            minimumX: 0
        }

        onReleased: { // releasing at the end of drag
            if( checked  &&  pill.x < root.width - pill.width)  root.clicked(false) // right to left
            if(!checked  &&  pill.x)                            root.clicked(true ) // left  to right
        }

        onClicked: root.clicked(!checked) // emit
    }
}
```

Test.qml

```
import QtQuick 2.0

Switch {
    property bool backend: false

    checked:   backend

    onClicked: backend = checked
}
```

Part 4: Slider

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-slider

The Slider has **value**, **minimum**, and **maximum** public properties. Slider is implemented with a single **MouseArea** that covers the entire control and utilizes drag to handle the user sliding the "pill" (the portion which the user moves) left and right.

The background "tray" (a horizontal line, which can be tapped) is split into left and right pieces so that it doesn't show through the pill when enabled is set to false (since the pill is partially transparent in the disabled state). The only tricky parts about Slider are the equations to compute the value from pixels and pixels from the value.

Slider.qml

```
import QtQuick 2.0

Item {
    id: root

// public
    property double maximum: 10
    property double value:    0
    property double minimum:  0

    signal clicked(double value);  //onClicked:{root.value = value;  print('onClicked', value)}

// private
    width: 500;  height: 100 // default size
    opacity: enabled  &&  !mouseArea.pressed? 1: 0.3 // disabled/pressed state

    Repeater { // left and right trays (so tray doesn't shine through pill in disabled state)
        model: 2

        delegate: Rectangle {
            x:     !index?               0: pill.x + pill.width - radius
            width: !index? pill.x + radius: root.width - x;  height: 0.1 * root.height
            radius: 0.5 * height
            color: 'black'
            anchors.verticalCenter: parent.verticalCenter
        }
    }

    Rectangle { // pill
        id: pill

        x: (value - minimum) / (maximum - minimum) * (root.width - pill.width) // pixels from value
        width: parent.height;  height: width
        border.width: 0.05 * root.height
        radius: 0.5 * height
    }

    MouseArea {
        id: mouseArea

        anchors.fill: parent
```

Slider (Continued)

```
        drag {
            target:   pill
            axis:     Drag.XAxis
            maximumX: root.width - pill.width
            minimumX: 0
        }

        onPositionChanged:  if(drag.active) setPixels(pill.x + 0.5 * pill.width) // drag pill
        onClicked:                          setPixels(mouse.x) // tap tray
    }

    function setPixels(pixels) {
        var value = (maximum - minimum) / (root.width - pill.width) * (pixels - pill.width / 2) + minimum // value from pixels
        clicked(Math.min(Math.max(minimum, value), maximum)) // emit
    }
}
```

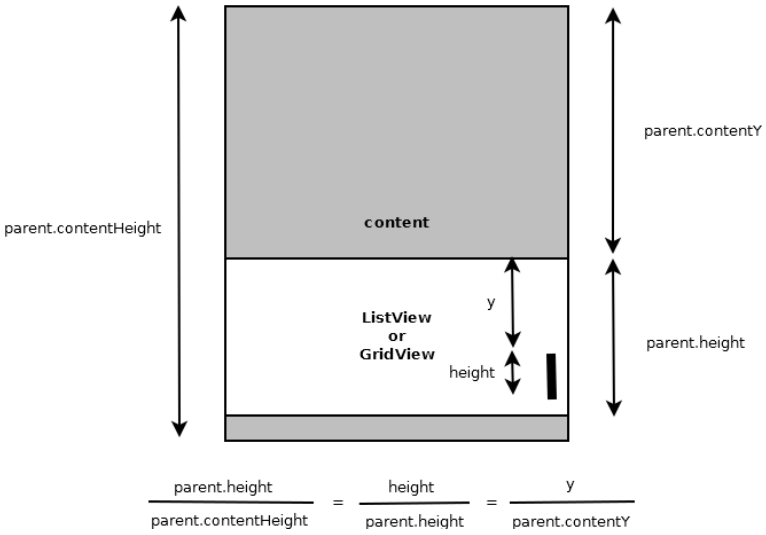
Test.qml

```
import QtQuick 2.0

Slider {
    property double backend: 0

    maximum:  10
    value:    backend
    minimum: -10

    onClicked: backend = value
}
```

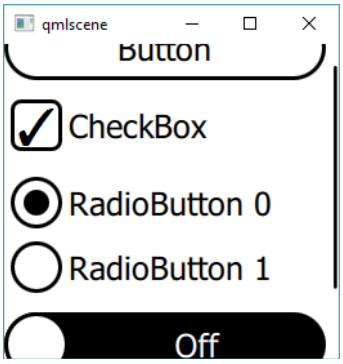
Part 5: Scrollbar

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-scrollbar

A vertical ScrollBar is the vertical line segment you often see on the right of a touch user interface as you scroll vertically through a list of items. **ScrollBar is quite a bit different than the other controls in this series as it can't be run stand-alone in qmlscene.** Rather, it is designed to be a direct child of a ListView or GridView (the ScrollBar's parent). The ScrollBar's position (y) and height are computed with a bit of math, based on proportions of Flickable's contentHeight and contentY, as illustrated below.



You can see an example of ScrollBar on the right side of Test.qml if you resize qmlscene to a short enough height:



Scrollbar.qml

```
import QtQuick 2.0

Rectangle { // ScrollBar to be placed as a direct child of a ListView or GridView (ScrollBar won't run by itself and gives errors)
    color: 'black'
    width: 0.01 * parent.width;  radius: 0.5 * width // size controlled by width
    anchors{right: parent.right;  margins: radius}

    height:  parent.height   / parent.contentHeight * parent.height
    y:       parent.contentY / parent.contentHeight * parent.height
    visible: parent.height   < parent.contentHeight
}
```

Test.qml

```
import QtQuick 2.0

ListView {
    ...

    ScrollBar{}
}
```

Part 6: ProgressBar

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-progressbar

A ProgressBar is often used to indicate the progress of a long-running operation from 0 to 100%. ProgressBar is a read-only control so it's pretty simple, save for the math required to compute its width. ProgressBar is implemented with only two Rectangles. Its public properties are **minimum**, **value**, and **maximum**.

ProgressBar.qml

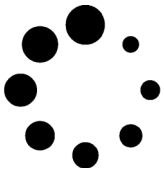
```
import QtQuick 2.0

Rectangle { // background
    id: root

// public
    property double maximum: 10
    property double value:   0
    property double minimum: 0

// private
    width: 500;  height: 100 // default size

    border.width: 0.05 * root.height
    radius: 0.5 * height

    Rectangle { // foreground
        visible: value > minimum
        x: 0.1 * root.height;  y: 0.1 * root.height
        width: Math.max(height,
               Math.min((value - minimum) / (maximum - minimum) * (parent.width - 0.2 * root.height),
                        parent.width - 0.2 * root.height)) // clip
        height: 0.8 * root.height
        color: 'black'
        radius: parent.radius
    }
}
```

Test.qml

```
import QtQuick 2.0

ProgressBar {
    maximum:  10
    value:     0
    minimum: -10
}
```

Part 7: Spinner

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-spinner



A Spinner is also known as a busy indicator. Spinner indicates the progress of a long-running operation when the progress percentage is unknown. (For known percentages, we'd use a ProgressBar.) Spinner uses a Timer to animate rotation. Normally, a designer would provide us with a .png asset to rotate an Image, but for this example we rotate an array of circles, implemented with Rectangle. Spinner has no public properties.

Spinner.qml

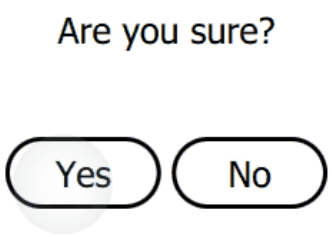
```
import QtQuick 2.0

Item {
    id: root

// public

// private
    width: 500;  height: 100 // default size

    Item { // square
        id: square

        anchors.centerIn: parent
        property double minimum: Math.min(root.width, root.height)
        width: minimum;  height: minimum

        Repeater {
            id: repeater

            model: 8

            delegate: Rectangle{
                color: 'black'

                property double b: 0.1
                property double a: 0.25

                width: ((b - a) / repeater.count * index + a) * square.height;  height: width
                radius: 0.5 * height

                x: 0.5 * square.width  + 0.5 * (square.width  - width )  * Math.cos(2 * Math.PI / repeater.count * index) - 0.5 * width
                y: 0.5 * square.height - 0.5 * (square.height - height)  * Math.sin(2 * Math.PI / repeater.count * index) - 0.5 * height
            }
        }
    }

    Timer {
        interval: 10
        running: true
        repeat:  true

        onTriggered: square.rotation += 2 // degrees
    }
}
```

Test.qml

```
import QtQuick 2.0

Spinner{}
```

Part 8: Dialog

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-dialog



A Dialog supports an arbitrary number of Buttons (and we reuse our Button control).

Its public API includes the **text** to show, an array of strings for the **buttons**, and a **clicked()** signal that provides the index of the button the user clicked. Unlike other controls in the series, Dialog is full screen so we place it at the root level in Test.qml and at the bottom so it is highest in z order (i.e. on top).

The client code (in Test.qml) takes responsibility for managing the visible property of the Dialog. We need to provide a MouseArea to prevent touches on anything but the Buttons from passing through to (hidden) controls beneath the Dialog.

Dialog.qml

```
import QtQuick 2.0

Rectangle { // white background
    id: root

// public
    property string  text:    'text'
    property variant buttons: []//'0', '1']

    signal clicked(int index);  //onClicked: print('onClicked', index)

// private
    width: 500;  height: 500 // default size

    MouseArea{anchors.fill: parent} // don't allow touches to pass to MouseAreas underneath

    Text { // text
        text: root.text
        anchors{centerIn: parent;  verticalCenterOffset: -0.1 * root.height}
        font.pixelSize: 0.1 * root.height
        width: 0.9 * parent.width
        wrapMode: Text.WordWrap
        horizontalAlignment: Text.AlignHCenter
    }

    Row { // buttons
        id: row

        anchors{bottom: parent.bottom;  horizontalCenter: parent.horizontalCenter;  bottomMargin: 0.1 * root.height}
        spacing: 0.03 * root.width

        Repeater {
            id: repeater

            model: buttons

            delegate: Button {
                width: Math.min(0.5 * root.width, (0.9 * root.width - (repeater.count - 1) * row.spacing) / repeater.count)
                height: 0.2 * root.height
                text:  modelData

                onClicked: root.clicked(index)
            }
        }
    }
}
```

Test.qml

```
import QtQuick 2.0

Dialog {
    id: dialog

    text: 'Are you sure?'
    buttons: ['No', 'Yes']

    onClicked: visible = false
}
```

Part 9: PageDots

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-pagedots

PageDots are often used in conjunction with ListView.SnapOneItem to indicate what “page” is currently shown.

The public properties are **page** and **pages** to indicate the current page and total number of pages respectively. The dots are implemented with a Rectangle (configured as a circle) inside a Repeater.

PageDots.qml

```
import QtQuick 2.0

Item {
    id: root

// public
    property int page:  0 // current
    property int pages: 3 // total

// private
    width: 500;  height: 100 // default size

    Row {
        spacing: (root.width - pages * root.height) / (pages - 1)

        Repeater {
            model: pages

            delegate: Rectangle { // circle
                width:  root.height;  height: width
                radius: 0.5 * width
                color:  index == page? 'black': 'white'
                border.width: 0.05 * root.height
            }
        }
    }
}
```

Test.qml

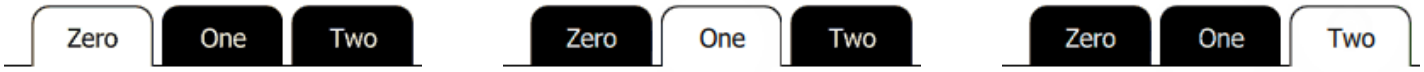
```
import QtQuick 2.0

ListView { // PageDots
    id: listView

    width: 250;  height: 100
    snapMode: ListView.SnapOneItem
    orientation: Qt.Horizontal

    model: 3

    delegate: Item {
        width: listView.width;  height: listView.height

        Text {
            text: index
            font.pixelSize: 0.9 * listView.height
            anchors.centerIn: parent
        }
    }
```

PageDots (Continued)

```
    onMovementEnded: listView.currentIndex = listView.indexAt(contentX, 0)

    PageDots {
        width: 0.25 * parent.width;  height: 0.1 * parent.height
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.bottom: parent.bottom
        page:  listView.currentIndex
        pages: listView.count
    }
}
```

Part 10: Tabs

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-tabs



Tabs are used to expand limited screen real estate by providing two or more “tabs” that divide the user interface into screens (content), only one of which is shown at a time. The Tabs control only renders the tabs themselves. A screen (content) for each tab must be implemented separately.

Tabs has two public properties of interest: model (an array of strings) and currentIndex (indicating the currently selected tab), both of which are from ListView. To implement the content for each tab, simply provide an Item for each tab, and connect each Item’s visible property to currentIndex in a binding. For instance:

```
Item {
    visible: tabs.currentIndex == 0
    ...
```

Here are a couple of our tricks for implementing Tabs:

1. To render a half-rounded Rectangle for each tab, we wrap a rounded Rectangle (twice the tab height) in an Item whose clip property is set true to hide the lower half the the Rectangle (and its two unwanted bottom rounded corners).

2. To render the horizontal line at the bottom of Tabs (implemented with three Rectangles), we draw outside the bounding rectangle of the delegate. This technique is not usually needed nor recommended, but it does come in handy once in a while (as here). The horizontal line is broken into two lines: one left of the selected tab and one right of the selected tab.

Tabs.qml

```
import QtQuick 2.0

ListView {
    id: root

// public
    model:          []//'Zero', 'One', 'Two']
    currentIndex:   0

// private
    width: 500;  height: 100 // default size

    orientation: ListView.Horizontal
    interactive: false
    spacing: 0.1 * height
    clip: true // horizontal line

    header: Item{width: root.width - count * (2 * 0.7 * root.height + spacing)} // left
```

Tabs (Continued)

```
    delegate: Item { // tab
        width: 2 * height;  height: 0.7 * root.height
        y: root.height - height             // align bottom
        opacity: mouseArea.pressed? 0.3: 1 // pressed state

        Item { // don't clip horizontal line
            anchors.fill: parent
            clip: true

            Rectangle { // background
                width: parent.width;  height: 2 * parent.height
                border.width: 0.02 * root.height
                radius: 0.2 * root.height
                color: currentIndex == index? 'transparent': 'black'
            }
        }

        Text {
            text: modelData
            font.pixelSize: 0.3 * root.height
            anchors.centerIn: parent
            color: currentIndex == index? 'black': 'white'
        }

        Rectangle { // horizontal line at bottom left
            visible: currentIndex == index;
            anchors{bottom: parent.bottom;  right: parent.left}
            width: root.width;    height: 0.02 * root.height
            color: 'black'//green'
        }

        Rectangle { // horizontal line at bottom right
            visible: currentIndex == index;
            anchors{bottom: parent.bottom;  left:  parent.right}
            width: root.width;    height: 0.02 * root.height
            color: 'black'//blue'
        }

        MouseArea {
            id: mouseArea

            anchors.fill: parent
            enabled: currentIndex != index

            onClicked:  currentIndex = index
        }
    }
}
```

Test.qml

```
import QtQuick 2.0

Tabs {
    model:          ['Zero', 'One', 'Two']
    currentIndex:   0
}
```

Part 11: Table

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-table

A Table is a two-dimensional matrix of strings supporting an arbitrary number of rows and columns. The Table consists of two main parts: header and data. Consequently, it has two public properties (**headerModel** and **dataModel**) and one public **clicked()** signal, which is emitted when the user taps on a row of data. Both header and data are implemented with ListViews.

The header background implements a half-rounded Rectangle by composing two Rectangles: one for the top two rounded corners, and another un-rounded Rectangle of half height to cover the bottom two round corners. The data ListView has a nested delegate (the second from a Row Repeater) so we must take care to store the index and modelData of the outer delegate. The column widths can be adjusted by setting the **width** property in **headerModel** (the sum of which must add to one). We reuse our ScrollBar control to indicate how far the data has been scrolled.

Table.qml

```
import QtQuick 2.0

Item { // size controlled by width
    id: root

// public
    property variant headerModel: [ // widths must add to 1
        // {text: 'Color',         width: 0.5},
        // {text: 'Hexadecimal',   width: 0.5},
    ]

    property variant dataModel: [
        // ['red',   '#ff0000'],
        // ['green', '#00ff00'],
        // ['blue',  '#0000ff'],
    ]

    signal clicked(int row, variant rowData);  //onClicked: print('onClicked', row, JSON.stringify(rowData))

// private
    width: 500;  height: 200

    Rectangle {
        id: header

        width: parent.width;  height: 0.14 * root.width
        color: 'black'
        radius: 0.03 * root.width

        Rectangle { // half height to cover bottom rounded corners
            width: parent.width;  height: 0.5 * parent.height
            color: parent.color
            anchors.bottom: parent.bottom
        }

        ListView { // header
            anchors.fill: parent
            orientation: ListView.Horizontal
            interactive: false

            model: headerModel
```

Table (Continued)

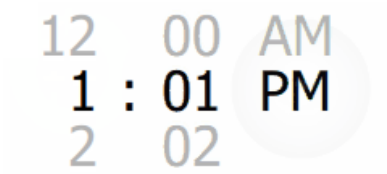
```
            delegate: Item { // cell
                width: modelData.width * root.width;  height: header.height

                Text {
                    x: 0.03 * root.width
                    text: modelData.text
                    anchors.verticalCenter: parent.verticalCenter
                    font.pixelSize: 0.06 * root.width
                    color: 'white'
                }
            }
        }
    }

    ListView { // data
        anchors{fill: parent;  topMargin: header.height}
        interactive: contentHeight > height
        clip: true

        model: dataModel

        delegate: Item { // row
            width: root.width;  height: header.height
            opacity: !mouseArea.pressed? 1: 0.3 // pressed state

            property int     row:     index     // outer index
            property variant rowData: modelData // much faster than listView.model[row]

            Row {
                anchors.fill: parent

                Repeater { // index is column
                    model: rowData // headerModel.length

                    delegate: Item { // cell
                        width: headerModel[index].width * root.width;  height: header.height

                        Text {
                            x: 0.03 * root.width
                            text: modelData
                            anchors.verticalCenter: parent.verticalCenter
                            font.pixelSize: 0.06 * root.width
                        }
                    }
                }
            }

            MouseArea {
                id: mouseArea

                anchors.fill: parent

                onClicked:  root.clicked(row, rowData)
            }
        }

        ScrollBar{}
    }
}
```

Test.qml

```
import QtQuick 2.0

Table {
    width: 0.98 * root.width;  height: 0.4 * root.width // resize

    headerModel: [ // widths must add to 1
        {text: 'Color',         width: 0.5},
        {text: 'Hexadecimal',   width: 0.5},
    ]

    dataModel: [
        ['Red',   '#ff0000'],
        ['Green', '#00ff00'],
        ['Blue',  '#0000ff'],
    ]

    onClicked: print('onClicked', row, JSON.stringify(rowData))
}
```

Part 12: TimePicker

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-timepicker



A TimePicker allows the user to select a time in terms of hours, minutes and am/pm.

TimePicker's public interface consists of a **set()** function, a **clicked()** signal, and an **interval** property (to specify the granularity of minutes e.g. 1, 2, 5...). A function **set()** is used instead of a property since TimePicker also returns a time (via **clicked()**) and we only want to set once (i.e. not a binding).

set() and **clicked()** both pass a JavaScript Date, but use only the time part (hours, minutes) and don't use the date part (year, month, day). TimePicker is implemented with a Row of three ListViews. The ranges of 1-12 hours and 0-59 minutes are each given five repetitions to provide the illusion that they can be scrolled forever (since ListView doesn't support circular lists).

TimePicker.qml

```
import QtQuick 2.0

Item {
    id: root

// public
    function set(date) { // e.g. new Date(0, 0, 0,  0, 0)) // 12:00 AM
        var hour = date.getHours() + (!date.getHours()? 12: date.getHours() <= 12? 0: -12)//24 hour to AM/PM
        repeater.itemAt(0).positionViewAtIndex(12            * (repetitions - 1) / 2 + hour - 1,                    ListView.Center)
        repeater.itemAt(1).positionViewAtIndex(60 / interval * (repetitions - 1) / 2 + date.getMinutes() / interval, ListView.Center)
        repeater.itemAt(2).positionViewAtIndex((rows - 1) / 2 + (date.getHours() < 12? 0: 1),                       ListView.Center)

        for(var column = 0; column < repeater.count; column++)  select(repeater.itemAt(column))
    }

    signal clicked(date date);  //onClicked: print('onClicked', date.toTimeString())

    property int interval: 1 // 30 20 15 10 5 2 1 minutes

// private
    width: 500;  height: 200 // default size
    clip: true

    onHeightChanged: resizeTimer.start() // resize
    Timer {id: resizeTimer;  interval: 1000;  onTriggered: set(get())} // ensure same value is selected after resize

    property int rows:        3 // number of rows on the screen     (must be odd). Also change model ''
    property int repetitions: 5 // number of times data is repeated (must be odd)

    Row {
        Repeater {
            id: repeater

            model: [ 12 * repetitions, 60 / interval * repetitions, ['', 'AM', 'PM', ''] ] // 1-12 hour, 0-59 minute, am/pm

            delegate: ListView { // hours minutes am/pm
                id: view

                property int column: index // outer index
                width: root.width / 3;  height: root.height
                snapMode: ListView.SnapToItem

                model: modelData
```

TimePicker (Continued)

```
            delegate: Item {
                width: root.width / 3;  height: root.height / rows

                Text {
                    text: view.get(index)
                    font.pixelSize: Math.min(0.5 * parent.width, parent.height)
                    anchors{verticalCenter:                parent.verticalCenter
                            right:            column == 0? parent.right:            undefined
                            horizontalCenter: column == 1? parent.horizontalCenter: undefined
                            left:             column == 2? parent.left:             undefined
                            rightMargin:              0.2 * parent.width}
                    opacity: view.currentIndex == index? 1: 0.3
                }
            }

            onMovementEnded: {select(view);  timer.restart()}
            onFlickEnded:    {select(view);  timer.restart()}
            Timer {id: timer;  interval: 1;  onTriggered: clicked(root.get())} // emit only once

            function get(index) { // returns e.g. '00' given row
                     if(column == 0)  return index % 12 + 1                          // hour
                else if(column == 1)  return ('0' + (index * interval) % 60).slice(-2) // minute
                else                  return model[index]                             // AM/PM
            }
        }
    }
}

Text { // colon
    text: ':'
    font.pixelSize: Math.min(0.5 * root.width / 3, root.height / rows)
    anchors{verticalCenter: parent.verticalCenter}
    x: root.width / 3 - width / 4
}

function select(view) {view.currentIndex = view.indexAt(0, view.contentY + 0.5 * view.height)} // index at vertical center

function get() { // returns e.g. '12:00 AM'
    var hour = repeater.itemAt(0).get(repeater.itemAt(0).currentIndex)         // integer
    var am   = repeater.itemAt(2).get(repeater.itemAt(2).currentIndex) == 'AM' // boolean
    return new Date(0, 0, 0,
                    hour == 12? (am? 0: 12): (am? hour: hour + 12),            // hour
                    repeater.itemAt(1).get(repeater.itemAt(1).currentIndex))   // minute
}

// Component.onCompleted: set(new Date(0, 0, 0,  0, 0)) // 12:00 AM otherwise defaults to index 0 selected
}
```

Test.qml

```
import QtQuick 2.0

TimePicker {
    Component.onCompleted: set(new Date(0, 0, 0,  0, 0)) // 12:00 AM
    onClicked:             print('onClicked', Qt.formatTime(date, 'h:mm A'))
}
```

Part 13: DatePicker

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-datepicker

A DatePicker allows the user to select any calendar date (year, month, day). DatePicker's public interface consists of a **set()** function and a **clicked()** signal. A function **set()** is used instead of a property since DatePicker also returns a date (via **clicked()**) and we only want to set once (i.e. not a binding). **set()** and **clicked()** both pass a JavaScript Date, but use only the date part (year, month, day) and don't use the time part (hours, minutes). DatePicker is implemented as a ListView, which can be swiped left or right to change the month.

Note: as of this writing, JavaScript Date contains a bug on Qt 5.12.x and 5.13.x on MinGW on Windows that prevents DatePicker from working correctly.

DatePicker.qml

```
import QtQuick 2.0

ListView {
    id: root

 // public
    function set(date) { // new Date(2019, 10 - 1, 4)
        selectedDate = new Date(date)
        positionViewAtIndex((selectedDate.getFullYear()) * 12 + selectedDate.getMonth(), ListView.Center) // index from month year
    }

    signal clicked(date date);  // onClicked: print('onClicked', date.toDateString())

 // private
    property date selectedDate: new Date()

    width: 500;  height: 100 // default size
    snapMode:    ListView.SnapOneItem
    orientation: Qt.Horizontal
    clip:        true

    model: 3000 * 12 // index == months since January of the year 0

    delegate: Item {
        property int year:      Math.floor(index / 12)
        property int month:     index % 12 // 0 January
        property int firstDay:  new Date(year, month, 1).getDay() // 0 Sunday to 6 Saturday

        width: root.width;  height: root.height

        Column {
            Item { // month year header
                width: root.width;  height: root.height - grid.height

                Text { // month year
                    anchors.centerIn: parent
                    text: ['January', 'February', 'March', 'April', 'May', 'June',
                           'July', 'August', 'September', 'October', 'November', 'December'][month] + ' ' + year
                    font {pixelSize: 0.5 * grid.cellHeight}
                }
            }
```

DatePicker (Continued)

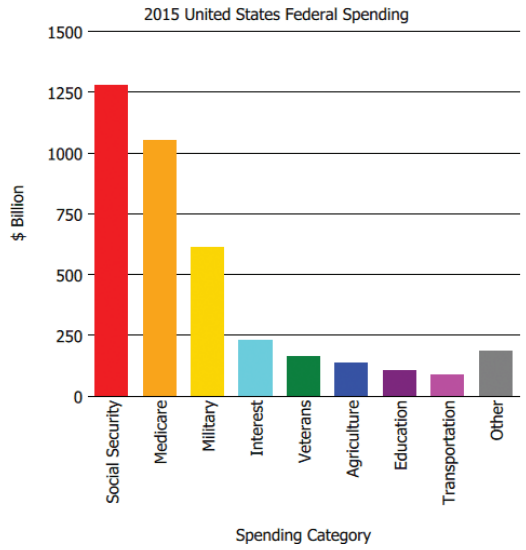
```
        Grid { // 1 month calender
            id: grid

            width: root.width;  height: 0.875 * root.height
            property real cellWidth:  width  / columns;
            property real cellHeight: height / rows // width and height of each cell in the grid.

            columns: 7 // days
            rows:    7

            Repeater {
                model: grid.columns * grid.rows // 49 cells per month

                delegate: Rectangle { // index is 0 to 48
                    property int day:  index - 7 // 0 = top left below Sunday (-7 to 41)
                    property int date: day - firstDay + 1 // 1-31

                    width: grid.cellWidth;  height: grid.cellHeight
                    border.width: 0.3 * radius
                    border.color: new Date(year, month, date).toDateString() == selectedDate.toDateString()  &&  text.text  &&  day >= 0?
                                  'black': 'transparent' // selected
                    radius: 0.02 * root.height
                    opacity: !mouseArea.pressed? 1: 0.3  //  pressed state

                    Text {
                        id: text

                        anchors.centerIn: parent
                        font.pixelSize: 0.5 * parent.height
                        font.bold:      new Date(year, month, date).toDateString() == new Date().toDateString() // today
                        text: {
                            if(day < 0)                                                 ['S', 'M', 'T', 'W', 'T', 'F', 'S'][index] // Su-Sa
                            else if(new Date(year, month, date).getMonth() == month)    date // 1-31
                            else                                                        ''
                        }
                    }

                    MouseArea {
                        id: mouseArea

                        anchors.fill: parent
                        enabled:    text.text  &&  day >= 0

                        onClicked: {
                            selectedDate = new Date(year, month, date)
                            root.clicked(selectedDate)
                        }
                    }
                }
            }
        }
    }

    // Component.onCompleted: set(new Date()) // today (otherwise Jan 0000)
}
```

Test.qml

```
import QtQuick 2.0

DatePicker {
    Component.onCompleted: set(new Date()) // today
    onClicked:             print('onClicked', Qt.formatDate(date, 'M/d/yyyy'))
}
```

Part 14: BarChart

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-barchart



Many people who need charts in their application wonder if they need a charting library, but it turns out to be not that difficult to write a custom autoscaled chart.

The public interface consists of a **title**, **yLabel**, **xLabel**, and a list of **points**, which contains a string for **x**, a number for **y**, and a **color**. Since Rectangle can be used to draw a line, all the pixels are rendered with just Rectangle and Text (as are all the controls so far in this series). The only tricky part is the math necessary to compute the y axis tick lines, which is accomplished via a logarithm.

BarChart.qml

```
import QtQuick 2.0

Item {
    id: root

 // public
    property string title:  'title'
    property string yLabel: 'yLabel'
    property string xLabel: 'xLabel'

    property variant points: []//{x: 'Zero', y: 60, color: 'red'}, {x: 'One', y: 40, color: 'blue' }]

 // private
    property double factor: Math.min(width, height)

    property double yInterval:  1
    property double yMaximum:  10 // set by onPointsChanged
    property double yMinimum:   0
    function toYPixels(y){return plot.height / (yMaximum - yMinimum) * (y - yMinimum)}

    property int    xMaximum:   0 // string length

    onPointsChanged: { // auto scale vertically
        if(!points)  return
        var xMaximum = 0, yMinimum = 0, yMaximum = 0
        for(var i = 0; i < points.length; i++) {
            if(points[i].y > yMaximum)          yMaximum = points[i].y
            if(points[i].y < yMinimum)          yMinimum = points[i].y
            if(points[i].x.length > xMaximum)   xMaximum = points[i].x.length
        }

        var yLog10     = Math.log(yMaximum - yMinimum) / Math.LN10 // take log, convert to integer, and then raise 10 to this power
        root.yInterval = Math.pow(10, Math.floor(yLog10)) / (yLog10 % 1 < 0.7? 4: 2) // distance between ticks
        root.yMaximum  = Math.ceil( yMaximum / yInterval) * yInterval
        root.yMinimum  = Math.floor(yMinimum / yInterval) * yInterval

        root.xMaximum  = xMaximum
    }

    width: 500;  height: 500 // default size
```

BarChart (Continued)

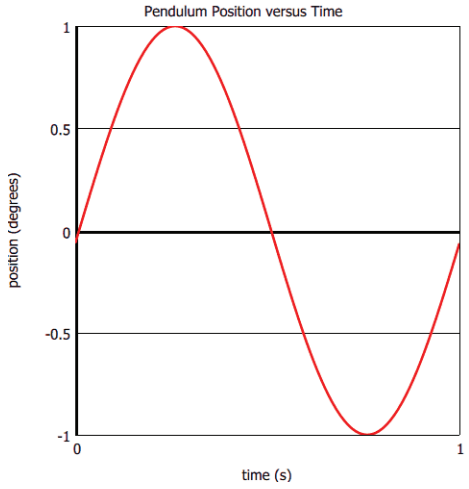
```
    Text { // title
        text: title
        anchors.horizontalCenter: parent.horizontalCenter
        font.pixelSize: 0.03 * factor
    }

    Text { // y label
        text: yLabel
        font.pixelSize: 0.03 * factor
        y: 0.5 * (2 * plot.y + plot.height + width)
        rotation: -90
        transformOrigin: Item.TopLeft
    }

    Text { // x label
        text: xLabel
        font.pixelSize: 0.03 * factor
        anchors{bottom: parent.bottom;  horizontalCenter: plot.horizontalCenter}
    }

    Item { // plot
        id: plot

        anchors{fill: parent;  topMargin: 0.05 * factor;  bottomMargin: (0.015 * xMaximum + 0.05) * factor;
                              leftMargin: 0.15 * factor;  rightMargin: 0.05 * factor}

        Repeater { // y axis tick marks and labels
            model: Math.floor((yMaximum - yMinimum) / yInterval) + 1 // number of tick marks

            delegate: Rectangle {
                property double value: index * yInterval + yMinimum
                y: -toYPixels(value) + plot.height
                width: plot.width;  height: 1
                color: 'black'

                Text {
                    text: parent.value
                    anchors{right: parent.left;  verticalCenter: parent.verticalCenter;  margins: 0.01 * factor}
                    font.pixelSize: 0.03 * factor
                }
            }
        }

        Repeater { // data
            model: points

            delegate: Item { // column
                width: plot.width / points.length;  height: plot.height
                x: width * index

                Rectangle { // bar
                    anchors{horizontalCenter: parent.horizontalCenter
                        bottom: modelData.y > 0? parent.bottom: undefined;  bottomMargin: toYPixels(0)
                        top:    modelData.y < 0? parent.top:    undefined;     topMargin: plot.height - toYPixels(0)}
                    width:  0.7 * parent.width;  height: toYPixels(Math.abs(modelData.y) + yMinimum)
                    color: modelData.color
                }

                Text { // x values (rotated -90 degrees)
                    text: modelData.x
                    x:   (parent.width - height) / 2
                    y:    parent.height + width + 0.5 * height
                    rotation: -90
                    transformOrigin: Item.TopLeft
                    font.pixelSize: 0.03 * factor
                }
            }
        }
    }

   // focus: true
   // Keys.onPressed: { // increase values with 0-9 and decrease with Alt+0-9
       // if(!isNaN(parseInt(event.text))  &&  parseInt(event.text) < root.points.length) { // 0-9 keys
           // var points = root.points
           // points[event.text].y = points[event.text].y + (event.modifiers? -0.1: 0.1) * (yMaximum - yMinimum)
           // root.points = points
       // }
   // }
}
```

Test.qml

```
import QtQuick 2.0

BarChart {
    title:  '2015 United States Federal Spending'
    yLabel: '$ Billion'
    xLabel: 'Spending Category'
    points: [
        {x: 'Social Security',  y: 1275.7, color: 'red'    },
        {x: 'Medicare',         y: 1051.8, color: 'orange' },
        {x: 'Military',         y:  609.3, color: 'gold'   },
        {x: 'Interest',         y:  229.2, color: 'cyan'   },
        {x: 'Veterans',         y:  160.6, color: 'green'  },
        {x: 'Agriculture',      y:  135.7, color: 'blue'   },
        {x: 'Education',        y:  102.3, color: 'purple' },
        {x: 'Transportation',   y:   85.0, color: 'magenta'},
        {x: 'Other',            y:  186.3, color: 'gray'   },
    ]
}
```

Part 15: LineChart

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-linechart



LineChart is similar to BarChart but with two exceptions: (1) it requires x axis tick marks and (2) it uses Canvas to draw the line curve. This is our first control to use Canvas, which is a rectangular area on which to draw with a Context2D. The public interface consists of a **title**, **yLabel**, **xLabel**, a list of **points**, and the **color** of the line.

Note: If using Qt 4 and/or QtQuick 1, replace Canvas either by a custom QDeclarativeItem or an Image fed by a QDeclarativeImageProvider.

LineChart.qml

```
import QtQuick 2.0

Item {
    id: root

 // public
    property string title:  'title'
    property string yLabel: 'yLabel'
    property string xLabel: 'xLabel'

    property variant points: []//{x: 0, y: 0}, {x: 1, y: 2}]
    property string  color: 'red'

 // private
    property double factor: Math.min(width, height)

    property double yInterval:  1 // set by onPointsChanged
    property double yMaximum:  10
    property double yMinimum:   0
    function toYPixels(y){return -plot.height / (yMaximum - yMinimum) * (y - yMinimum) + plot.height}

    property double xInterval:  1 // set by onPointsChanged
    property double xMaximum:  10
    property double xMinimum:   0
    function toXPixels(x){return plot.width  / (xMaximum - xMinimum) * (x - xMinimum)}

    onPointsChanged: { // auto scale
        var xMinimum = 0, xMaximum = 0, yMinimum = 0, yMaximum = 0
        for(var i = 0; i < points.length; i++) {
            if(points[i].y > yMaximum)  yMaximum = points[i].y
            if(points[i].y < yMinimum)  yMinimum = points[i].y
            if(points[i].x > xMaximum)  xMaximum = points[i].x
            if(points[i].x < xMinimum)  xMinimum = points[i].x
        }

        var yLog10     = Math.log(yMaximum - yMinimum) / Math.LN10 //  take log, convert to integer, and then raise 10 to this power
        root.yInterval = Math.pow(10, Math.floor(yLog10)) / 2 // distance between ticks
        root.yMaximum  = Math.ceil( yMaximum / yInterval) * yInterval
        root.yMinimum  = Math.floor(yMinimum / yInterval) * yInterval

        var xLog10     = Math.log(xMaximum - xMinimum) / Math.LN10 //  take log, convert to integer, and then raise 10 to this power
        root.xInterval = Math.pow(10, Math.floor(xLog10)) // distance between ticks
        root.xMaximum  = Math.ceil( xMaximum / xInterval) * xInterval
        root.xMinimum  = Math.floor(xMinimum / xInterval) * xInterval

        canvas.requestPaint()
    }
```

LineChart (Continued)

```
width: 500;  height: 500 // default size

Text { // title
    text: title
    anchors.horizontalCenter: parent.horizontalCenter
    font.pixelSize: 0.03 * factor
}

Text { // y label
    text: yLabel
    font.pixelSize: 0.03 * factor
    y: 0.5 * (2 * plot.y + plot.height + width)
    rotation: -90
    transformOrigin: Item.TopLeft
}

Text { // x label
    text: xLabel
    font.pixelSize: 0.03 * factor
    anchors{bottom: parent.bottom;  horizontalCenter: plot.horizontalCenter}
}

Item { // plot
    id: plot

    anchors{fill: parent;  topMargin: 0.05 * factor;  bottomMargin: 0.1 * factor;  leftMargin: 0.15 * factor; rightMargin: 0.05 * factor}

    Repeater { // y axis tick marks and labels
        model: Math.floor((yMaximum - yMinimum) / yInterval) + 1 // number of tick marks

        delegate: Rectangle {
            property double value: index * yInterval + yMinimum
            y: toYPixels(value)
            width: plot.width;  height: value? 1: 3
            color: 'black'

            Text {
                text: parseFloat(parent.value.toPrecision(9)).toString()
                anchors{right: parent.left;  verticalCenter: parent.verticalCenter;  margins: 0.01 * factor}
                font.pixelSize: 0.03 * factor
            }
        }
    }

    Repeater { // x axis tick marks and labels
        model: Math.floor((xMaximum - xMinimum) / xInterval) + 1 // number of tick marks

        delegate: Rectangle {
            property double value: index * xInterval + xMinimum
            x: toXPixels(value)
            width: value? 1: 3;  height: plot.height;
            color: 'black'

            Text {
                text: parseFloat(parent.value.toPrecision(9)).toString()
                anchors{top: parent.bottom;  horizontalCenter: parent.horizontalCenter;  margins: 0.01 * factor}
                font.pixelSize: 0.03 * factor
            }
        }
    }

    Canvas { // points
        id: canvas

        anchors.fill: parent

        onPaint: {
            var context = getContext("2d")
            context.clearRect(0, 0, width, height) // new points data (animation)
            context.strokeStyle = color
            context.lineWidth   = 0.005 * factor
            context.beginPath()
            for(var i = 0; i < points.length; i++)
                context.lineTo(toXPixels(points[i].x), toYPixels(points[i].y))
            context.stroke()
        }
    }
}
```

LineChart (Continued)

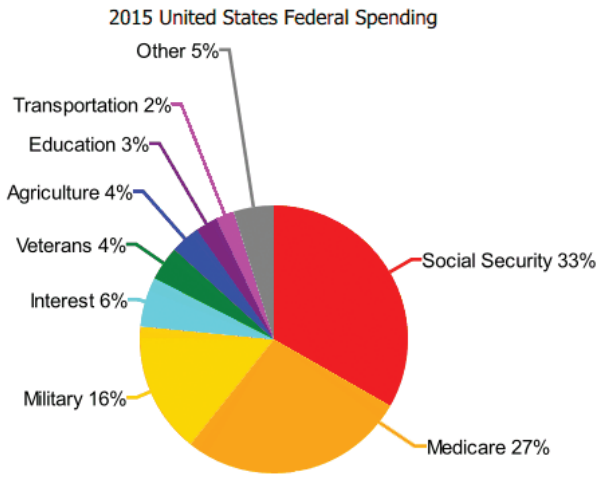
```
//    focus: true
//    Keys.onPressed: { // increase values with 0-9 and decrease with Alt+0-9
//        if(!isNaN(parseInt(event.text))  &&  parseInt(event.text) < root.points.length) { // 0-9 keys
//            var points = root.points
//            points[event.text].y = points[event.text].y + (event.modifiers? -0.1: 0.1) * (yMaximum - yMinimum)
//            root.points = points
//        }
//    }

//    Component.onCompleted: { // sine wave
//        var points = [], N = 100, T = 1
//        for(var i = 0; i <= N; i++)
//            points.push({x: T / N * i , y: Math.sin(2 * Math.PI * i / N)})
//        root.points = points
//    }
}
```

Test.qml

```
import QtQuick 2.0

LineChart {
    width: root.width;  height: root.width // resize

    title:  'Pendulum Position versus Time'
    yLabel: 'position (degrees)'
    xLabel: 'time (s)'
    color:  'red'

    Component.onCompleted: { // sine wave
        var positions = [], N = 100, T = 1
        for(var i = 0; i <= N; i++)
            positions.push({x: T / N * i , y: Math.sin(2 * Math.PI * i / N)})
        points = positions
    }
}
```

Part 16: PieChart

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-piechart



PieChart's public API consists of just a **title** and a list of **points** (whose **x**, **y**, and **color** members are identical to that of BarChart). Since PieChart is a Canvas, it makes heavy use of the Context2D API to draw its pie slices, callout lines, and text. We use a modified cosine to make the callout lines longer on the top/bottom and shorter on the sides so the text doesn't overlap when the pie slices are small and close together.

Note: If using Qt 4 and/or QtQuick 1, replace Canvas either by a custom QDeclarativeItem or an Image fed by a QDeclarativeImageProvider.

PieChart.qml

```
import QtQuick 2.0

Canvas {
    id: root

 // public
    property string title: 'title'

    property variant points: []//{x: 'Zero', y: 60, color: 'red'}, {x: 'One', y: 40, color: 'blue' }] // y values don't need to add to 100

 // private
    onPointsChanged: requestPaint()

    width: 500;  height: 500 // default size
    property double factor: Math.min(width, height)

    Text { // title
        text: title
        anchors.horizontalCenter: parent.horizontalCenter
        font.pixelSize: 0.03 * factor
    }

    onPaint: {
        var context    = getContext("2d")
        var total      = 0 // automatically calculated from points.y
        var start      = -Math.PI / 2 // Start from vertical. 0 is 3 o'clock and positive is clockwise
        var radius     = 0.2  * factor
        var pixelSize  = 0.03 * factor // text
        context.font   = pixelSize + 'px arial'

        for(var i = 0; i < points.length; i++)  total += points[i].y // total

        context.clearRect(0, 0, width, height) // new points data (animation)

        for(var i = 0; i < points.length; i++) {
            var end    = start + 2 * Math.PI * points[i].y / total // radians
            var center = Qt.vector2d(width / 2, height / 2) // center

            // pie
            context.fillStyle = points[i].color
            context.beginPath()
            var midSlice = Qt.vector2d(Math.cos((end + start) / 2), Math.sin((end + start) / 2)).times(radius) // point on edge/middle
            context.arc(center.x, center.y, radius, start, end) // x, y, radius, startingAngle (radians), endingAngle (radians)
            context.lineTo(center.x, center.y) // center
            context.fill()
```

PieChart (Continued)

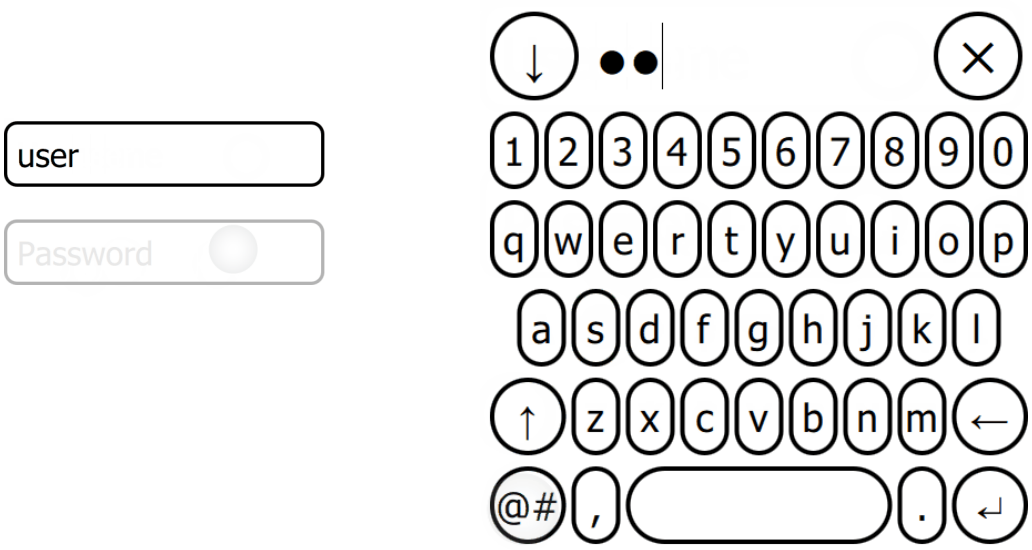
```
            // line
            context.lineWidth   = 0.005 * factor
            context.strokeStyle = points[i].color
            context.beginPath()
            context.moveTo(center.x + midSlice.x, center.y + midSlice.y) // center

            var angle = (start + end) / 2 // of line
            var point = midSlice.times(1 + 1.4 * (1 - Math.abs(Math.cos(angle)))).plus(center) // elbow of line
            context.lineTo(point.x, point.y)
            context.lineTo(point.x + (point.x < center.x? -1: 1) * 0.5 * pixelSize, point.y) // horizontal
            context.stroke()

            // text
            context.fillStyle = 'black'
            var percent   = points[i].y / total * 100
            var text      = points[i].x + ' ' + (percent < 1? '< 1': Math.round(percent)) + '%' // display '< 1%' if < 1
            var textWidth = context.measureText(text).width
            context.fillText(text, (point.x < center.x? -textWidth - 0.5 * pixelSize: 0.5 * pixelSize) + point.x, point.y + 0.4 * pixelSize)

            start = end // radians
        }
    }

    // focus: true
    // Keys.onPressed: { // increase values with 0-9 and decrease with Alt+0-9
        // if(!isNaN(parseInt(event.text))  &&  parseInt(event.text) < root.points.length) { // 0-9 keys
            // var points = root.points
            // points[event.text].y = points[event.text].y + (event.modifiers? -0.1: 0.1) * points[event.text].y
            // root.points = points
        // }
    // }
}
```

Test.qml

```
import QtQuick 2.0

PieChart {
    title:  '2015 United States Federal Spending'
    points: [
        {x: 'Social Security',  y: 1275.7, color: 'red'    },
        {x: 'Medicare',         y: 1051.8, color: 'orange' },
        {x: 'Military',         y:  609.3, color: 'gold'   },
        {x: 'Interest',         y:  229.2, color: 'cyan'   },
        {x: 'Veterans',         y:  160.6, color: 'green'  },
        {x: 'Agriculture',      y:  135.7, color: 'blue'   },
        {x: 'Education',        y:  102.3, color: 'purple' },
        {x: 'Transportation',   y:   85.0, color: 'magenta'},
        {x: 'Other',            y:  186.3, color: 'gray'   },
    ]
}
```

Part 17: Keyboard

Download the code: https://www.ics.com/blog/creating-qml-controls-scratch-keyboard



There are typically three ways to display a virtual keyboard in a QML app:

1. Qt Virtual Keyboard
2. Use the keyboard that ships with the operating system (e.g. on Windows 10 call TabTip.exe in Tablet mode)
3. Roll your own virtual keyboard in QML

If the keyboard must match a designer mockup, 3 is usually the only option, which is the approach we'll take here.

Our Keyboard implementation consists of three QML files:

Keyboard.qml: renders the full-screen keyboard, and reuses our Button control (not used directly by clients)
KeyboardController.qml: non-visual component to show Keyboard.qml and retrieve the text string the user typed in
KeyboardInput.qml: TextInput that brings up Keyboard via KeyboardController

KeyboardController can be used to bring up the Keyboard from anywhere by calling its **show()** method. Internally, it creates and destroys the Keyboard as necessary. While memory-efficient, the Keyboard can be slow to come up on older systems. If there is enough memory, you can make the Keyboard a singleton instead to always keep it in memory (trading memory usage for speed). KeyboardConroller uses a trick to reparent the Keyboard to the application's rootObject() such that the Keyboard is always on top of everything else.

Keyboard.qml

```
import QtQuick 2.0

Item {
    id: root

// public
    property bool password: false

    signal accepted(string text);   // onAccepted: print('onAccepted', text)
    signal rejected();              // onRejected: print('onRejected')

// private
    width: 500;  height: 500 // default size

    property double rowSpacing:     0.01 * width  // horizontal spacing between keyboard
    property double columnSpacing:  0.02 * height // vertical   spacing between keyboard
    property bool   shift:          false
    property bool   symbols:        false
    property double columns:        10
    property double rows:           5

    MouseArea {anchors.fill: parent} // don't allow touches to pass to MouseAreas underneath

    Rectangle { // input
        width: root.width;  height: 0.2 * root.height

        Button { // close v
            id: closeButton

            text: '\u2193' // BLACK DOWN-POINTING TRIANGLE
            width: height;  height: 0.8 * parent.height
            anchors.verticalCenter: parent.verticalCenter
            x: columnSpacing

            onClicked: rejected() // emit
        }

        TextInput {
            id: textInput

            cursorVisible: true
            anchors {left: closeButton.right;  right: clearButton.left;  verticalCenter: parent.verticalCenter;  margins: 0.03 * root.width}
            font.pixelSize: 0.5 * parent.height
            clip: true
            echoMode: password? TextInput.Password: TextInput.Normal

            onAccepted: if(acceptableInput) root.accepted(text) // keyboard Enter key
        }

        Button { // clear x
            id: clearButton

            text: '\u2715' // BLACK DOWN-POINTING TRIANGLE
            width: height;  height: 0.8 * parent.height
            anchors {verticalCenter: parent.verticalCenter;  right: parent.right;  rightMargin: columnSpacing}
            enabled:   textInput.text

            onClicked: textInput.text = ''
        }
    }

    Rectangle {
        width: parent.width;  height: 0.8 * parent.height
        anchors.bottom: parent.bottom

        Item { // keys
            id: keyboard

            anchors {fill: parent; leftMargin: columnSpacing}

            Column {
                spacing: columnSpacing
```

Keyboard (Continued)

```
Row { // 1234567890
    spacing: rowSpacing

    Repeater {
        model: [
             {text: '1', width: 1},
             {text: '2', width: 1},
             {text: '3', width: 1},
             {text: '4', width: 1},
             {text: '5', width: 1},
             {text: '6', width: 1},
             {text: '7', width: 1},
             {text: '8', width: 1},
             {text: '9', width: 1},
             {text: '0', width: 1},
        ]

        delegate: Button {
            text: modelData.text
            width: modelData.width * keyboard.width / columns - rowSpacing
            height: keyboard.height / rows - columnSpacing

            onClicked: root.clicked(text)
        }
    }
}

Row { // qwertyuiop
    spacing: rowSpacing

    Repeater {
        model: [
             {text: 'q', symbol: '+',      width: 1},
             {text: 'w', symbol: '\u00D7', width: 1}, // MULTIPLICATION SIGN
             {text: 'e', symbol: '\u00F7', width: 1}, // DIVISION SIGN
             {text: 'r', symbol:      '=', width: 1},
             {text: 't', symbol:      '/', width: 1},
             {text: 'y', symbol:      '_', width: 1},
             {text: 'u', symbol:      '<', width: 1},
             {text: 'i', symbol:      '>', width: 1},
             {text: 'o', symbol:      '[', width: 1},
             {text: 'p', symbol:      ']', width: 1},
        ]

        delegate: Button {
            text: symbols? modelData.symbol: shift? modelData.text.toUpperCase():  modelData.text
            width: modelData.width * keyboard.width / columns - rowSpacing
            height: keyboard.height / rows - columnSpacing

            onClicked: root.clicked(text)
        }
    }
}

Row { // asdfghjkl
    spacing: rowSpacing

    Repeater {
        model: [
             {text:  '', symbol:  '', width: 0.5},
             {text: 'a', symbol: '!', width: 1},
             {text: 's', symbol: '@', width: 1},
             {text: 'd', symbol: '#', width: 1},
             {text: 'f', symbol: '$', width: 1},
             {text: 'g', symbol: '%', width: 1},
             {text: 'h', symbol: '&', width: 1},
             {text: 'j', symbol: '*', width: 1},
             {text: 'k', symbol: '(', width: 1},
             {text: 'l', symbol: ')', width: 1},
             {text:  '', symbol:  '', width: 0.5},
        ]

        delegate: Button {
            text: symbols? modelData.symbol: shift? modelData.text.toUpperCase():  modelData.text
            width: modelData.width * keyboard.width / columns - rowSpacing
            height: keyboard.height / rows - columnSpacing

            onClicked: root.clicked(text)
        }
    }
}
```

Keyboard (Continued)

```
            Row { // zxcvbnm
                spacing: rowSpacing

                Repeater {
                    model: [
                         {text: '\u2191', symbol:       '', width: 1.5}, // UPWARDS ARROW (shift)
                         {text: 'z',      symbol:      '-', width: 1},
                         {text: 'x',      symbol:     "'", width: 1},
                         {text: 'c',      symbol:     '"', width: 1},
                         {text: 'v',      symbol:      ':', width: 1},
                         {text: 'b',      symbol:      ';', width: 1},
                         {text: 'n',      symbol:      ',', width: 1},
                         {text: 'm',      symbol:      '?', width: 1},
                         {text: '\u2190', symbol: '\u2190', width: 1.5}, // LEFTWARDS ARROW (backspace)
                    ]

                    delegate: Button {
                        text: symbols? modelData.symbol: shift? modelData.text.toUpperCase():  modelData.text
                        width: modelData.width * keyboard.width / columns - rowSpacing
                        height: keyboard.height / rows - columnSpacing
                        enabled: text == '\u2190'? textInput.text: true // LEFTWARDS ARROW (backspace)

                        onClicked: root.clicked(text)
                    }
                }
            }

            Row { // space
                spacing: rowSpacing

                Repeater {
                    model: [
                         {text: symbols? 'AB': '@#', width: 1.5},
                         {text:                 ',', width: 1},
                         {text:                 ' ', width: 5}, // space
                         {text:                 '.', width: 1},
                         {text:            '\u21B5', width: 1.5}, // DOWNWARDS ARROW WITH CORNER LEFTWARDS (enter)
                    ]

                    delegate: Button {
                        text:    modelData.text
                        width: modelData.width * keyboard.width / columns - rowSpacing
                        height: keyboard.height / rows - columnSpacing
                        enabled: text == '\u21B5'? textInput.text: true // DOWNWARDS ARROW WITH CORNER LEFTWARDS (enter)

                        onClicked: root.clicked(text)
                    }
                }
            }
        }
    }

    signal clicked(string text)
    onClicked: {
        if(     text == '\u2190') { // LEFTWARDS ARROW (backspace)
            var position = textInput.cursorPosition
            textInput.text = textInput.text.substring(0, textInput.cursorPosition - 1) +
                             textInput.text.substring(textInput.cursorPosition, textInput.text.length)
            textInput.cursorPosition = position - 1
        }
        else if(text == '\u2191')  shift   = !shift // UPWARDS ARROW (shift)
        else if(text == '@#' )     symbols = true
        else if(text == 'AB'  )    symbols = false
        else if(text == '\u21B5')  accepted(textInput.text) // DOWNWARDS ARROW WITH CORNER LEFTWARDS (enter)
        else { // insert text
            var position = textInput.cursorPosition
            textInput.text = textInput.text.substring(0, textInput.cursorPosition) + text +
                             textInput.text.substring(textInput.cursorPosition, textInput.text.length)
            textInput.cursorPosition = position + 1

            shift = false // momentary
        }
    }
}
```

KeyboardController.qml

```
import QtQuick 2.0

Item {
    id: root

// public
    property bool password: false

    signal accepted(string text);   // onAccepted: print('onAccepted', text)

    function show() {
        keyboard = keyboardComponent.createObject(0,  {password: root.password})

        var rootObject = null, object = parent // search up the parent chain to find QQuickView::rootObject()
        while(object) {
            if(object)  rootObject = object
            object = object.parent
        }

        keyboard.parent  = rootObject
        keyboard.width   = rootObject.width // resize
        keyboard.height  = rootObject.height
    }

// private
    property Item keyboard: null
    Component {id: keyboardComponent;  Keyboard {}}

    Connections {
        target: keyboard

        onAccepted: {
            root.accepted(text) // emit
            keyboard.destroy() // hide
        }

        onRejected: keyboard.destroy() // hide
    }
}
```

KeyboardInput.qml

```
import QtQuick 2.0

Rectangle { // TextInput from virtual keyboard
    id: root

 // public
    property string label:    'label'
    property bool   password: false
    property alias  text:     textInput.text // in/out

    signal accepted(string text);   // onAccepted: print('onAccepted', text)

 // private
    width: 500;  height: 100 // default size
    border.width: 0.05 * root.height
    radius:       0.2 * height
    opacity:      enabled  &&  !mouseArea.pressed? 1: 0.3 // disabled/pressed state

    Text { // label
        visible: !textInput.text
        text: label
        anchors {left: parent.left;  right: parent.right;  verticalCenter: parent.verticalCenter;  margins: parent.radius}
        font.pixelSize: 0.5 * parent.height
        opacity: 0.3
    }

    TextInput {
        id: textInput
```

Keyboard (Continued)

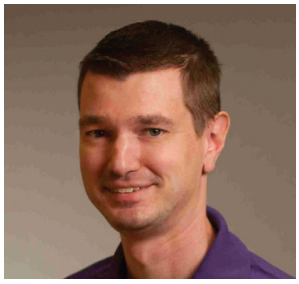
```
        anchors {left: parent.left;  right: parent.right;  verticalCenter: parent.verticalCenter;  margins: parent.radius}
        font.pixelSize: 0.5 * parent.height
        echoMode: password? TextInput.Password: TextInput.Normal
    }

    MouseArea { // comment out to input text via physical keyboard
        id: mouseArea

        anchors.fill: parent

        onClicked:  keyboardController.show()
    }

    KeyboardController {
        id: keyboardController

        password: root.password

        onAccepted: {
            textInput.text = text
            root.accepted(text) // emit
        }
    }
}
```

Test.qml

```
import QtQuick 2.0

KeyboardInput {
    label: 'Username'

    onAccepted: print('onAccepted', text)
}
```

About

Chris Cortopassi



A seasoned engineer with 10+ years experience, Chris has expertise in desktop and embedded systems in larger, real-time multi-threaded applications, as well as experience using Qt-based software for complex projects within the motor control, telecommunications and civil engineering industries.

ICS

Integrated Computer Solutions (ICS) delivers excellence in both user experience (UX) design and custom user interface (UI) software development for IoT and embedded devices, and companion mobile and desktop applications.

If your product is driven by touchscreen or voice, it's in our wheelhouse. We can design the intuitive UX and the UI software, as well as integrate with your platform. That means we help you build better products, with lower risk and cost. We've worked on everything from sensitive medical devices to business-critical industrial equipment to automotive digital cockpits. And we've helped some of the world's most-renowned brands, including Abbott, Boeing, GE, Intel, MilliporeSigma and Thermo Fisher.

We develop on a variety of platforms, including Qt/QML, HTML5, QNX, Linux, Android, iOS and are the largest independent source of Qt expertise in the U.S. Our successful track record stretches back to 1987.