Qt Quick for Qt Developers

User Interaction

The Qt Company

Based on Qt 5.4 (QtQuick 2.4)

Contents

Objectives

- Knowledge of ways to receive user input
  - Mouse/touch input
  - Keyboard input
- Awareness of different mechanisms to process input
  - Signal handlers
  - Property bindings

Demo: <Qt Examples>/declarative/toys/corkboards

Mouse Input

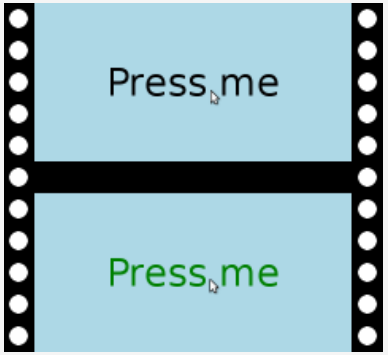
Mouse Areas

- Placed and resized like ordinary items
  - Using anchors if necessary
- Two ways to monitor mouse input:
  - Handle signals
  - Dynamic property bindings

See Documentation: MouseArea Element

Clickable Mouse Area

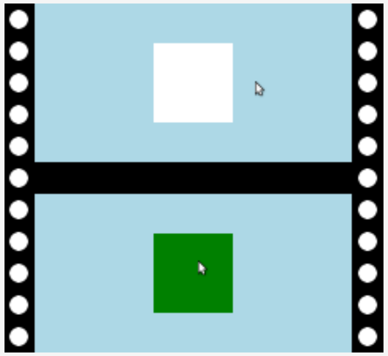
```
Rectangle {
    width: 400; height: 200; color: "lightblue”
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "Press me"; font.pixelSize: 48
        MouseArea {
            anchors.fill: parent
            onPressed: parent.color = "green"
            onReleased: parent.color = "black"
        }
    }
}
```



Demo: qml-user-interaction/ex-mouse-input/mouse-pressed-signals.qml

Mouse Hover and Properties

```
Rectangle {
    width: 400; height: 200; color: "lightblue”
    Rectangle {
        x: 150; y: 50; width: 100; height: 100
        color: mouseArea.containsMouse ? "green" : "white”
        MouseArea {
            id: mouseArea
            anchors.fill: parent
            hoverEnabled: true
        }
    }
}
```



Demo: qml-user-interaction/ex-mouse-input/hover-property.qml

Mouse Area Hints and Tips

- A mouse area only responds to its `acceptedButtons`
  - The handlers are not called for other buttons, but
  - Any click involving an allowed button is reported
  - The `pressedButtons` property contains *all* buttons
  - Even non-allowed buttons, if an allowed button is also pressed
- With `hoverEnabled` set to false
  - Property `containsMouse` can be true if the mouse area is clicked

Signals vs. Property Bindings

- Signals can be easier to use in some cases
  - When a signal only affects one other item
- Property bindings rely on named elements
  - Many items can react to a change by referring to a property
- Use the most intuitive approach for the use case
- Favor simple assignments over complex scripts

Touch Input

Touch Events

- Single-touch (`MouseArea`)
- Multi-touch (`MultiPointTouchArea`)
- Gestures
  - Tap and Hold
  - Swipe
  - Pinch

Multi-Touch Events

```
MultiPointTouchArea {
    anchors.fill: parent
    touchPoints: [
        TouchPoint { id: point1 },
        TouchPoint { id: point2 },
        TouchPoint { id: point3 }
    ]
}
```

- TouchPoint properties:
  - int x
  - int y
  - bool pressed
  - int pointId

MultiPointTouchArea Signals

- onPressed(list<TouchPoint> touchPoints)
- onReleased(...)
    - `touchPoints` is list of *changed* points.

- onUpdated(...)
    - Called when points is updated (moved)
    - `touchPoints` is list of *changed* points.

- onTouchUpdated(...)
    - Called on *any* change
    - `touchPoints` is list of *all* points.

MultiPointTouchArea Signals

- onGestureStarted(GestureEvent gesture)
  - Cancel the gesture using `gesture.cancel()`

- onCanceled(list<TouchPoint> touchPoints)
  - Called when another element takes over touch handling.
  - Useful for undoing what was done on `onPressed`.

Demo: qml-user-interaction/ex-multi-touch/main.qml

Gestures

- Tap and Hold (`MouseArea` signal `onPressAndHold`)
- Swipe (`ListView`)
- Pinch (`PinchArea`)

Swipe Gestures

- Build into `ListView`

- **`snapMode: ListView.SnapOneItem`**
  The view settles no more than one item away from the first visible item at the time the mouse button is released.

- **`orientation: ListView.Horizontal`**

Demo: <Qt Examples>/declarative/toys/corkboards

Pinch Gesture

- Automatic pinch setup using the `target` property:

```
Image {
    source: "qt-logo.jpg”
    PinchArea {
        anchors.fill: parent
        pinch.target: parent
        pinch.minimumScale: 0.5; pinch.maximumScale: 2.0
        pinch.minimumRotation: -3600; pinch.maximumRotation: 3600
        pinch.dragAxis: Pinch.XAxis
    }
}
```

Demo: qml-user-interaction/ex-pinch

Pinch Gestures

- Signals for manual pinch handling
  - onPinchStarted(PinchEventpinch)
  - onPinchUpdated(PinchEventpinch)
  - onPinchFinished()
- PinchEvent properties:
  - point1, point2, center
  - rotation
  - scale
  - accepted
    - set to false in the `onPinchStarted` handler if the gesture should not be handled

Keyboard Input

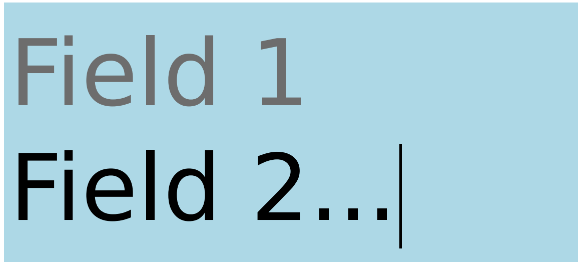
Keyboard Input

- Basic keyboard input is handled in two different use cases:
- Accepting text input
    - Elements `TextInput` and `TextEdit`
- Navigation between elements
    - Changing the focused element
    - directional(arrow keys), tab and backtab

- On Slide 28 we will see how to handle raw keyboard input.

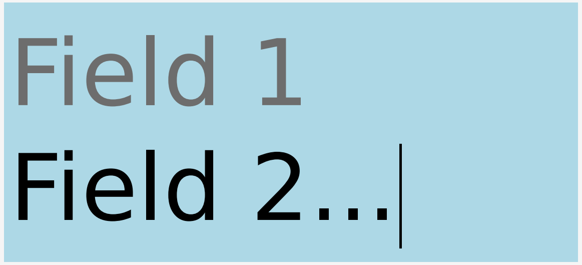
Assigning Focus

- Uis with just one `TextInput`
  - Focus assigned automatically
- More than one `TextInput`
  - Need to change focus by clicking
- What happens if a `TextInput` has no text?
  - No way to click on it
  - Unless it has a `width` or uses anchors
- Set the `focus` property to assign focus



Using TextInputs

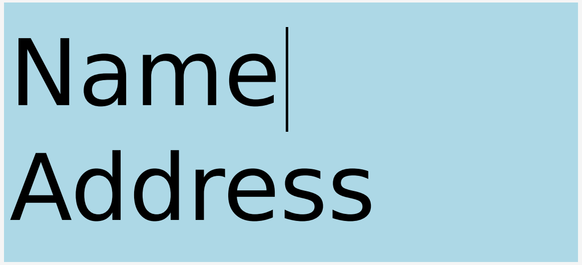
```
TextInput {
    anchors.left: parent.left; y: 16
    anchors.right: parent.right
    text: "Field 1"; font.pixelSize: 32
    color: focus ? "black" : "gray"
    focus: true
}
TextInput {
    anchors.left: parent.left; y: 64
    anchors.right: parent.right
    text: "Field 2"; font.pixelSize: 32
    color: focus ? "black" : "gray"
}
```


Field 1
Field 2...

Demo: qml-user-interaction/ex-key-input/textinputs.qml

Focus Navigation

```
TextInput {
    id: nameField
    focus: true
    KeyNavigation.tab: addressField
}
TextInput {
    id: addressField
    KeyNavigation.backtab: nameField
}
```

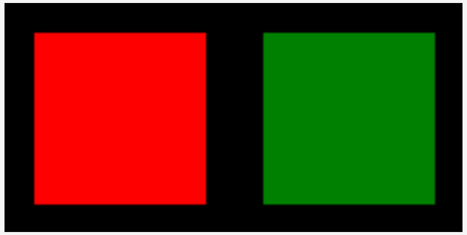


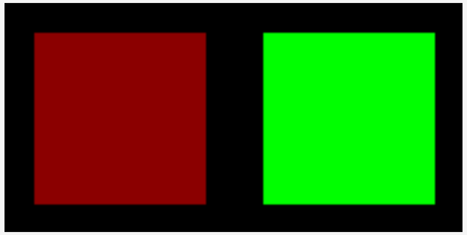
- The `name_field` item defines `KeyNavigation.tab`
  - Pressing Tab moves focus to the address_field item
- The `address_field` item defines `KeyNavigation.backtab`
  - Pressing Shift+Tab moves focus to the name_field item

Demo: qml-user-interaction/ex-key-input/tab-navigation.qml

Key Navigation

```
Rectangle { id: leftRect
            x: 25; y: 25; width: 150; height: 150
            color: focus ? "red" : "darkred"
            KeyNavigation.right: rightRect
            focus: true
}
Rectangle { id: rightRect
            x: 225; y: 25; width: 150; height: 150
            color: focus ? "#00ff00" : "green"
            KeyNavigation.left: leftRect
}
```





- Using cursor keys with non-text items
- Non-text items can have focus, too

Demo: qml-user-interaction/ex-key-input/key-navigation.qml

Summary

Mouse and cursor input handling:

- Element `MouseArea` receives clicks and other events
- Use anchors to fill objects and make them clickable
- Respond to user input:
  - Give the area a name and refer to its properties, or
  - Use handlers in the area and change other named items

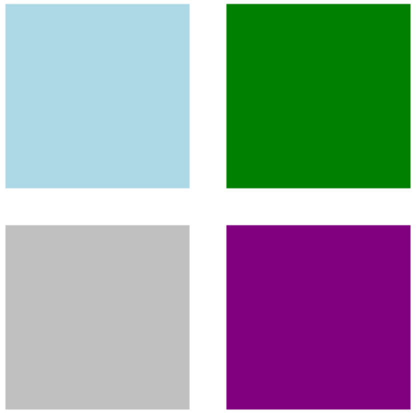
Key handling:

- Elements `TextInput` and `TextEdit` provide text entry features
- Set the `focus` property to start receiving key input
- Use anchors to make items clickable
  - Lets the user set the focus
- Element `KeyNavigation` defines relationships between items
  - Enables focus to be moved
  - Using cursor keys, tab and backtab
  - Works with non-text-input items

Lab – User Input

- Which element is used to receive mouse clicks?
- Name two ways `TextInput` can obtain the input focus.
- How do you define keyboard navigation between items?

Lab – Menu Screen



- Using the partial solution as a starting point, create a user interface similar to the one shown above with these features:
  - Items that change color when they have the focus
  - Clicking an item gives it the focus
  - The current focus can be moved using the cursor keys

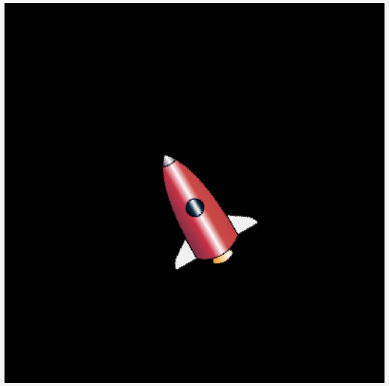
Lab: qml-user-interaction/lab-menu-screen

Raw Keyboard Input

- Raw key input can be handled by item
  - With predefined handlers for commonly used keys
  - Full key event information is also available
- The same focus mechanism is used as for ordinary text input
  - Enabled by setting the `focus` property
- Key handling is not an inherited property of items
  - Enabled using the `Keys` attached property
- Key events can be forwarded to other objects
  - Enabled using the `Keys.forwardTo` attached property
  - Accepts a list of objects

Raw Keyboard Input

```
Rectangle {
    width: 400; height: 400; color: "black"
    Image {
        id: rocket
        x: 150; y: 150
        source: "../images/rocket.svg"
        transformOrigin: Item.Center
    }
    Keys.onLeftPressed: rocket.rotation = (rocket.rotation - 10) % 360
    Keys.onRightPressed: rocket.rotation = (rocket.rotation + 10) % 360
    focus: true
}
```



Raw Keyboard Input

- Can use predefined handlers for arrow keys:

```
Keys.onLeftPressed: rocket.rotation = (rocket.rotation - 10) % 360
Keys.onRightPressed: rocket.rotation = (rocket.rotation + 10) % 360
```

- Or inspect events from all key presses:

```
Keys.onPressed: {
    if (event.key == Qt.Key_Left)
        rocket.rotation = (rocket.rotation - 10) % 360;
    else if (event.key == Qt.Key_Right)
        rocket.rotation = (rocket.rotation + 10) % 360;
}
```

Focus Scopes

- Focus scopes are used to manage focus for items
- Property `FocusScope` delegates focus to one of its children
- When the focus scope loses focus
  - Remembers which one has the focus
- When the focus scope gains focus again
  - Restores focus to the previously active item