Exercises Lecture 7 – Qt Quick

Aim: This exercise will let you try some of the features of Qt Quick.

Duration: 1h

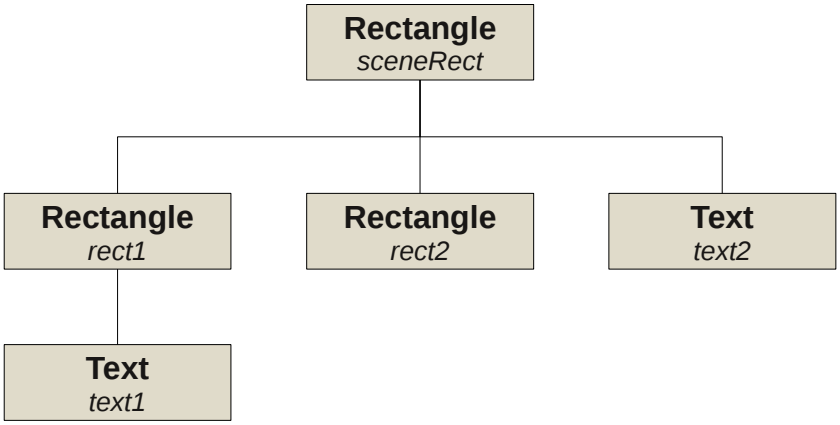
Populating a Scene

As a starting point for this exercise, you will write a QML application, populating a scene with a set of items. The scene will consist of two rectangles with text.

Start by creating a QML Application project in Qt Creator. It will be initialized with a single `.qml` file and a `.qmlproject` file. Start by opening the `.qml` file.

In that file, create a set of elements corresponding to the ownership tree shown below. i.e. the `sceneRect` contains all elements, while `text1` is contained within `rect1`.

To name the elements, use the `id` property. Notice that you will have to name the `sceneRect` as well.



Set the `width` and `height` of the `sceneRect` to 400x400 and make the other two rectangles 100x100 pixels large.

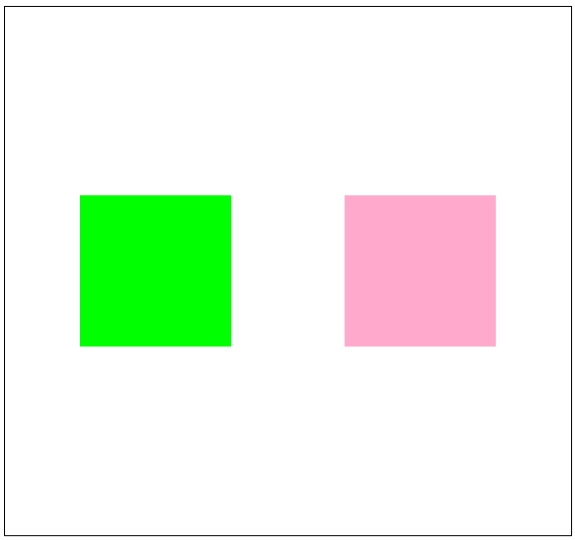
Set the text property of `text1` and `text2` to “Hello Qt” and “Quicker Qt” respectively.

Set the color properties of the rectangles to a named color and a color that you specify using a hex code, e.g. “green” and “#ffaacc”. The text defaults to black, but if you choose to have dark colored rectangles, you might want to change them to white.

If you try to run this file in the qml viewer, i.e. by pressing the run button in Qt Creator, all you will see is a bunch of overlapping items in the top left corner of the scene.

Placing Items

In order to arrange the items visually, you will use anchor layouts. The end result will look something like the illustration below, but feel free to add your own personal touch.



Start by making `text1` fill its parent by binding the `anchors.fill` property to parent.

Then center `text2` in `rect2` by using the `anchors.centerIn` property. Finally, make the `y` value of `rect2` mirror and invert the `y`-value of `rect1`.

I.e. the `y` value of `rect2` is bound to the following expression:

```
sceneRect.height - rect1.y - height
```

Finally, place `rect1` and `rect2` in a visually pleasing way, next to each other by binding the `x` values of both and the `y` value of `rect1`.

Running the qml script, it becomes obvious that the text of `text1` needs to be centered in both the vertical and horizontal direction to be correctly placed in `rect1`. This is achieved by binding the `verticalAlignment` and `horizontalAlignment` to `Text.AlignVCenter` and `Text.AlignHCenter` respectively.

Adding Movement

To understand the dynamics of the scene, you will now add an animation for the `y` value of `rect1`. This, in turn, will cause the `y` value of `rect2` to change too, as it is bound to the animated value.

Start by adding a single `NumberAnimation` for the `y` value of `rect1`. Simply replace the binding of the `y` property with the following line.

```
NumberAnimation on y {
       to: 300; duration: 5000; easing.type: Easing.InOutQuad;
}
```

In the animation, the target value (`to`), duration and easing curve are set. The `duration` is expressed in milliseconds (1s = 1000ms), so the animation lasts five seconds. The easing curve describes how the value is changed. In this case a quadratic curve is used to make the start and end softer. You can see all the available easing curves at *http://doc.qt.nokia.com/latest/qml-propertyanimation.html#easing.type-prop* .

Having a single number animation moves the rectangles once. The next task will be for you to set up two animations in a sequential group, i.e. make them run after each other. One animation will move the rectangles in one direction and the other animation will move them in the opposite direction. By looping the animation group, an infinite loop of continuous movement will be created.

Start by replacing the `NumberAnimation` by a `SequentialAnimation` on `y`. Within the sequential animation, create two `NumberAnimation` instances.

The first number animation has a destination (`to`) value of 300 while the second goes to zero. Now set the duration and easing type according to your own liking. Letting the rectangles bounce on impact (`Easing.OutBounce`) can produce a good effect, as can different combinations of `InQuad`, `OutQuad` or `InOutQuad`.

To make the sequential animation loop forever, bind the property `loops` like this:

```
loops: Animation.Infinite
```

Interacting with States

Hardcoded animations is one way to achieve movement, but usually the target positions are different for different animations and continuous movement is not the main objective. Instead, smooth transitions between states is what is requested.

Qt Quick addresses this using states and transitions, where states define the targets, and transitions how to get to the target. You will now introduce this to the scene from the previous exercise steps.

First, remove the animations from step 3 and resume to the static scene that you had at the end of step 2.

Now add the two states first and second in the `sceneRect` using the following lines:

```
states: [
    State { name: "first" },
    State { name: "second" }
]
```

For each of the states, a set of `PropertyChanges` elements can be added. A `PropertyChanges` element changes a set of properties for a given target element. For instance, the following line changes the `y` position of `rect1` to zero:

```
PropertyChanges { target: rect1; y: 0 }
```

Now, set up the states so that the `y` value of `rect1` is zero for the first state, and 300 for the second state.

You can now test the two states by binding the state property of the `sceneRect` to first or second.

```
state: "first"
```

This is, however, a bit dull. Instead, add a `MouseArea` in the `sceneRect` using the following lines:

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        if(parent.state == "first")
            parent.state = "second";
        else
```

```
            parent.state = "first";
    }
}
```

The result of this should be that mouse clicks anywhere within the scene switch the state. As you can see from testing the script, the state changes are instantaneous. Not pretty, nor smooth.

The solution to this is transitions. The `transitions` property contains a list of animations that are applied to given properties. This way, the state changes can be controlled.

To add transitions to your scene, start by adding a `transitions` property to the `sceneRect`, with a `NumberAnimation` for the `y` property to `rect1`.

```
transitions: [
    Transition {
        NumberAnimation {
            target: rect1
            property: "y"
            duration: 500
        }
    }
]
```

Experiment by tuning the duration and easing type of the `NumberAnimation` to get a smooth transition effect.

In the example used here, the same transition effect is applied to all transitions. Using the `to` and `from` properties of of each `Transition` object, you can control the behavior of different properties for different transition changes.

Try using different easing curves and durations depending on the destination (`to`) state. Then try to add changes to the `scale` of the two rectangles by adding more `PropertyChanges` to the states.

Solution Tips

Step 1

You set a property using a binding expression, e.g. binding the `width` of an element to the value 200, type the following within the `{` and `}` of the element declaration.

```
width: 200
```

The dimensions of a rectangle are controlled by the `width` and `height` properties.

To name an element, simply bind the `id` property to the name, without quotation marks.

```
id: sceneRect
```

To set a color, ensure to contain the name of the color or hex-code prefixed by a hash character within quotes.

```
color: "red"
```

Controlling the text of a `Text` element is simply a question of binding the `text` property to a quoted string.

```
text: "Qt is cute"
```

Step 2

Controlling the location of an item is simply a question of binding the `x` and `y` properties.

```
x: 10
y: 20
```

To ensure that the binding of the `y` value of the rectangles match up, try setting the `y` property of `rect1` to zero. This should place `rect1` at the top of the scene and `rect2` at the bottom.

Step 3

Do not forget to remove the binding of `y` to a static value when animating the property.

One example of a nice looking animation sequence looks like this:

```
SequentialAnimation on y {
    NumberAnimation { to: 300; duration: 1000;
        easing.type: Easing.OutBounce; }
    NumberAnimation { to: 0; duration: 3000;
        easing.type: Easing.InOutQuad; }
    loops: Animation.Infinite
}
```

Step 4

Adding a `PropertyChanges` to a `State` is a matter of declaring it within the `State` as shown below. Notice that you can add multiple `PropertyChanges` to a single `State`. When adding `State` elements to the states property, do not forget the comma between the states. This is because the `states` property is a list.

```
states: [
    State {
        PropertyChanges { ... }
        PropertyChanges { ... }
    },
    State {
...
```

To declare transitions for a specific `transition` with multiple `NumberAnimation` elements, you set the to and from properties of the `Transition` and declare the `NumberAnimation` elements within the `Transition` element.

```
transitions: [
    Transition {
        from: "state-name"
        NumberAnimation { ... }
        NumberAnimation { ... }
    },
    Transition {
...
```

The scale of an element is controlled by the `scale` property. A scale of 1.0 means that the element is not scaled, while a larger scale value means a larger object.

Other interesting properties to play with can be `rotation`, `opacity` and `color`.