Training – QML

Programming Labs

Version 5.5.0

Introduction

This document provides programming lab instructions for QML training class. Course attendees are requested to develop a QML application in nine separate labs. The tenth lab will ceoncentrate on the performance analysis of the app.

The application is estimated to be developed during a three day training course without any previous knowledge of QML or JavaScript. Qt essentials, such as Qt meta-object system, signals and slots, and memory management should be managed by the attendees though. The labs are based on Qt 5.5, but with minor modifications older or newer versions may be used.

Hopefully, you enjoy the training and the programming labs as well. The learning philosophy, used in the training, is

> *You hear you forget. You see you remember. You do you learn.*

Lab 1: QML Essentials

Objectives

- QML Designer
- Basic elements
- Layout and anchoring

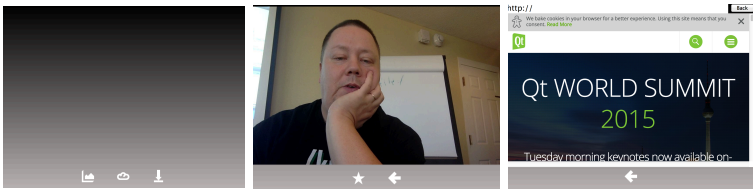
Reference Material

labs/templates/images	Useful images for later use
labs/qmlEssentials	One possible solution

Overview

In the following labs, your task is to implement an image application. The names means the application shows images, but optionally you my also use camera to capture new images and share them in Qt Cloud services. The application uses Qt Quick Controls to navigate between pages to provide a nice user experience on both desktop and mobile platforms. Each page is a separate QML file, providing the content of that page. You are free to implement any additional functionality at any point, if you want.

The screen shots below show the basic functionality of the application. In the main page, the user may navigate to a page, showing images, showing a web view or asking the user for images using a file dialog. Deeper in the navigation, the user may apply graphics effects on the image and play with the particle system.



Let's start the labs by creation a custom item, which we will later use as a notification window to the user. The notification is a simple rectangle with dynamic text. We could have used `MessageDialog` element (as the QML project wizard creates one for us), but the important learning objective is to learn, how to create custom items from the scratch. `MessageDialog` has a native look and feel in Android since Qt 5.4 and iOS in Qt 5.5, but in earlier Qt versions it looks rather terrible on mobile platforms.

Practicalities

1. Create a new "Qt Quick Controls Application", using the project wizard in your Qt Creator IDE.
2. Remove `menubar` property and `Label` element from **main.qml** file as we create about everything ourselves. Add `Rectangle` element type to the file and anchor it to `ApplicationWindow` area.
3. Define a nice background color for the notification rectangle. It could be a gradient, which you may easily create using Qt Quick Toolbar. The toolbar can be opened in the context menu by right-clicking `Rectangle`. Of course, you may use QML Designer as well.
4. Add `Text` element as a child of `Rectangle` and anchor it to the center of `Rectangle`. Define text `color` (so that it can be seen against your background) and large enough font size (e.g., 100 pixels). You may define some initial `text`, but that is not necessary. To have scalable font size, use `fontSizeMode` property of `Text` and define `minimumPixelSize` and `font.pixelSize` (the maximum size). Property `wrapMode` may be useful as well, if your notifications are composed from

several words. Note that you must define explicitly `Text` element dimensions as well. Otherwise, the text does not scale.

5. Notification element will be used in such a way that it will be instantiated in component, who wishes to use the notification. The component also sets dimensions for the notification. To set the notification text, you should provide a JS function, taking a string argument. The argument is assigned to the `text` property of the `Text` element. We will add some functionality to this function later, so that's why we have a function instead of simple alias property, bound to the `text` property of `Text` element.

    a. You may define a function in JavaScript like `function helloWorld(arg1, arg2) { }`

6. Can child items paint outside their parents?

7. Can items overlap?

8. The first lab is now ready. It is actually not a custom item yet, but we'll make it one in the next lab.

Lab 2: Custom Items

Objectives

- Scope and visibility rules
- Signals and signal handlers

References

Starting point:

labs/qmlEssentials

Example solution:

labs/customItems

Overview

Let's refactor the QML file of the previous lab a little bit to make it an actual custom item. It already has a custom function to show the notification message. It could have some custom properties, but instead of them we use the custom function this time. We will also add a custom signal to the item to notify observers that the notification window is closed. The signal is totally obsolete from the application behavior point of view, but we add it to understand how modules communicate using custom signals.

Practical Outline

1. Create a new QML file for your notification message. In Qt Creator, right-click the project name and "Add New…".
2. Copy `Rectangle` element you implemented in **main.qml** in the previous lab to the new QML file. Replace the default `Rectangle` element in that file.
3. The notification could be closed, when the user touches it, but we use a timer instead. Add `Timer` element to the notification message, define an interval, and set the opacity property of Rectangle to 0 or close to 0, when the timer expires.
4. In the custom function you added in the previous lab, set the notification opacity to some initial value (e.g. 0.7, if you want it to be semi-transparent). Start the timer as well in that function. Look at `Timer` documentation to see, how to start/stop the timer.
5. Set the opacity in `Rectangle` to some small value (e.g. 0.2). The notification message should not be visible, if the opacity is not greater than this threshold value. So when the timer expires, we set the custom item opacity to 0, which will set the visibility to false and the item is not painted.
6. Add a custom signal to the notification element. When the timer expires, emit this signal. No arguments are needed.
7. Instantiate the notification message item in **main.qml**. Anchor it to the center of the application window with suitable size. Check you can receive the custom signal from the notification message as well.
    a. You may use `Component.onCompleted` signal handler in the application window to test the notification. Just call your custom function in the notification and provide the message as the function argument. After testing, remove the signal handler from the application window, as we won't need it later.

Time Permitting

1. Add more decorations to your item: background image, border etc.

Lab 3: Qt Quick Controls

Objectives

- Qt Quick Layouts
- Views, especially StackView
- Page-based navigation model
- Control styling

References

Starting point:

labs/customItems

Example solution:

labs/qtQuickControls

Overview

Implement the stack view-based navigation to the application. You are requested to implement the pages and Qt Quick Control buttons to navigate between the pages. The controls are also slightly styled to get an idea, how to do the styling. You may also use your own controls, Qt Quick Enterprise controls or whatever you wish in addition to the requested controls.

Practical Outline

1. Instantiate a new QML element `StackView` in **main.qml**.
2. Create a new QML file and assign its resource URL to `initialItem` property of `StackView`. This initial or main page will be loaded only once. Add other pages to your project as well. We need at least: Effects page to show graphics effects, image viewer page to show images, and web view page.
3. Let's use `ToolButton` objects from Qt Quick Controls to navigate between the pages. Add `ToolBar` element into the initial page. Wait a second! Was it so that `ApplicationWindow` has a toolbar property? So why do we crate a separate `ToolBar` instance on every page? Would not it make sense to use the one in the `ApplicationWindow`? The obvious answer is that we should use the one in `ApplicationWindow`. However, to use it nicely we would need a model, which contains page-specific `ToolButton` icons (or their names) and we have not discussed model/view framework yet. This is the only reason we use page-specific `ToolBar` objects and not the one in `ApplicationWindow` in this programming lab.
4. The tool bar should contain, e.g. `RowLayout`, which contains `ToolButton` objects to navigate to new pages and back to the initial page (normally we would use a model and `Instantiator` to dynamically manage `ToolBar` items – look at **demos/controls/toolbar** for example). Anchor the layout at the bottom of your page. You may also add some margin. Anchor the row layout horizontally in the center of the tool bar and add your tool buttons into the layout. On the initial page, you need two buttons: one to navigate to the image view page and another to navigate to the web view page. From the image view page the user may navigate to the effects page by clicking an image, but we will implement this later.
5. Add signal handlers to the buttons. When a button is clicked, navigate to a new page by pushing the page to the stack (`stackIdentifier.push( "qrc:/PageName.qml")`). Navigate back by just popping the current page. What should be the root item type of your pages?
6. To show the tool buttons, you need to add text or an image to them. Copy the **labs/templates/images/** folder to your project (using file explorer or similar tool). Obviously, you may use any other images you want. In QtCreator, right-click

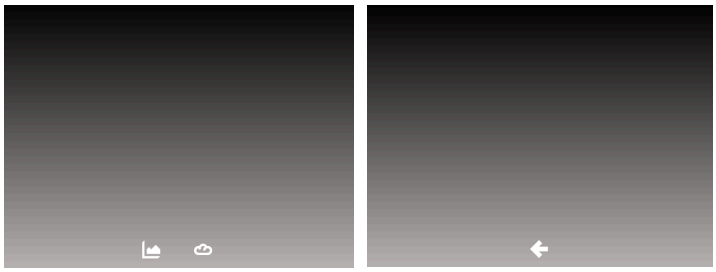
"/" symbol below **qml.qrc** in your project and use "Add Existing Files" to add all image files to your project resource file. Refer now to the image in your QML code by assigning, e.g. "`qrc:/images/info.png`" into `iconSource` property in your `ToolButton`. Note that icons are white, so if your tool bar background is white, you cannot see too much.

   a. You may style the tool bar by assigning a new style into `style` property. To get rid of the background, you may, e.g. assign an empty `Item` object into the `background` property of `ToolBarStyle`.

7. Add buttons containing the left arrow icon and functionality to navigate back to previous pages to all other but the initial page. You may re-use the code you wrote to the initial page.

8. Finally, style the `background` property of `ApplicationWindow` to have a gradient color. Now we do not need to define a background for our pages at all.

When you run the application, you should now have three pages and two buttons in the initial page to navigate between them.



Lab 4: Animations

Objectives

- Behavior animations
- Property animations
- Stack view transitions

Reference Material

labs/qtQuickControls	Starting point
labs/animations	Example solution

Overview

Default page transition animations animate pages from the right to the left, when you push pages to the stack and from the left to the right, when you pop pages. This animation is straightforward to change. Let's also animate the opacity change of the notification message.

Practicalities

1. Use `Behavior` element to animate the `opacity` change in the notification message. Define the animation properties (`duration`, `easing type` etc.) yourself.
2. Add a `delegate` property to `StackView` element in **main.qml**. Instantiate `StackViewDelegate` element. Use the `StackViewDelegate` property `pushTransition` and assign `StackViewTransition` object to the property. `StackViewTransition` inherits from `ParallelAnimation`.
3. Implement two `PropertyAnimation` (or whatever animations you want to use) objects as children of the `StackViewTransition`. You may animate, e.g. page opacity, x-coordinate or scale properties. To refer to the old page, use `exitItem` exposed property and to refer to the new page (which you just push to the stack), use `enterItem`.
4. If you need to reset animated properties after the transition, implement `transitionFinished(args)` function. Use `args.enterItem` and `args.exitItem` to access the page properties.
5. Enjoy the animations.

> Page transitions and the notification message are now animated with custom animations.

Lab 5: Data Models and Views

Objectives

- Model/view framework in QML

Reference Material

labs/animations	Starting point
labs/modelView	Example solution

Overview

The application shows images from a model in a path view. Later we will add functionality to apply graphics effects to the images.

If you have installed Qt Enginio module, you may use that to download images from the cloud and show them in one page. The other approach is to show images from a local file system, which is the default approach. Enginio-based implementation is described in the Time Permitting section.

The solution contains a property `USE_ENGINIO`, which is set by default to false in `main.qml`. Set it to true, if you wish to access Enginio data storage and download images there.

Practicalities

1. Add a basic `ListModel` QML type to your **main.qml**. Add some images to the project resources. You may use tool button icons as well. Add `ListElement` objects with some custom properties to the model. One property should contain the image URL. Otherwise, you may decide to use any properties you wish to show by the view. This time you may just hard code the property values directly in `ListElement` objects.

2. Add a view element, e.g. basic `PathView` first, to the image viewer QML file. For example, a simple V-shape path can be created with the following code:

```
path: Path {
    startX: parent.width * 0.05
    startY: parent.height * 0.05
    PathLine {
        x: parent.width  / 2
        y: parent.height * 0.85
    }
    PathLine {
        x: parent.width * 0.95
        y: parent.height *0.05
     }
 }
```

3. Next we will need a delegate. Your delegate should have at least an `Image` element with the size you prefer. Note that the `Image source` property should be read from model elements' file name property (whatever name you have used – the solution uses `fileId` property). Test you can see your images. If the images are painted on the top of each other, check your delegate root item has some dimensions as well.

4. Add `MouseArea` to your delegate. It should allow the user to navigate to the effects page, when an image is clicked. Note that the skeleton of the effects page is provided to you in the **labs/templates** folder. When you push the page, the existing implementation expects you pass the URL of the file. You may pass properties to pages using the `push()` method in the following way:

```
stackIddentifier.push({item: "qrc:/EffectsPage.qml", properties: {
fileId: model.fileId }});
```

Time Permitting

1. These instructions allow you to use Qt Cloud Service data model to download the images. There are two QML types to be added to your **main.qml** file: `EnginioModel` and `EnginioClient`. The former gets data from the Enginio and stores that into a model available for your views. The latter is used to identify the right data storage in Qt Cloud Services (`backendId`). The client element also provides an API to store, delete, and read image data. As Enginio is not in the scope of this training, you may just copy the following model and client code to your **main.qml**. Remember to import `Enginio 1.0` module as well.

```
EnginioModel {
    id: enginioModel
    client: enginioClient
    query: { // query for all objects of type "objects.image"
             // and include not null references to files
        "objectType": "objects.image",
        "include": {"file": {}},
        "query" : { "file": { "$ne": null } }
    }
}

EnginioClient {
    id: enginioClient
    backendId: "52b173cd5a3d8b15b10342bf"
    onError: console.log("Enginio error: " + reply.errorCode + ":
" + reply.errorString)
}
```

2. Implement `Component.onCompleted` signal handler property to your Image in your delegate. This handler uses JS functions to load the image URL, which is used by Image to download the image itself. Enginio client provides a method `downloadUrl()` to get the URL of the image. The method takes at least one argument, which is file `id`. The `file` property is provided by Enginio model.

3. Let's first download the thumbnails and then whole images as the Enginio stores images in two variant formats: thumbnails of size 100x100 pixels and whole images. Again the implementation details are outside the scope of this training, so you may just copy the code snippet below. Add a new custom property (`imagesUrl`) to your image viewer root element. It is a URL array. The arrays are of `var` type in QML. Allocate an object to store URLs (`new Object`).

```
if (id in imagesUrl) {
    image.source = imagesUrl[id];
} else {
    var data = { "id": file.id,
                 "variant": "thumbnail"};
    var reply = enginioClient.downloadUrl(data);
    reply.finished.connect(function() {
        imagesUrl[id] = reply.data.expiringUrl;
        if (image && reply.data.expiringUrl) {
            // It may be deleted as it is delegate
            image.source = reply.data.expiringUrl;
        }
    })
}
```

4. Add other elements to your delegate too. For example, an image element, containing a mouse area, to delete the image. You can use `enginioModel.remove(index)` or `listModel.remove()` (if you do not use Enginio) to delete the image. Property `index` is obviously provided by the view. You may use `Text` elements to show image file name (property `name`), size (`file.fileSize`), and creation time (`file.createdAt`) or whatever properties use used in your local image model. You may also implement a progress bar (use `Image progress` property). You may hide the progress bar, when `Image status` property is `Image.Ready`.

Now your application is able to show images using hard-coded URLs or/and Enginio URLs.

Lab 6: Scripting and Dynamic QML

Objectives

- Dynamic loading

Reference Material

labs/modelView	Starting point
labs/scripting	Lab solution

Overview

Some of the pages may take a long time to load statically. Or it may take a long time to instantiate all the objects in that page. To keep good user experience, we should load those pages into the memory beforehand, but avoid static loading in the main QML file to minimize the startup time. Camera instantiation is an example of a time-consuming functionality, which we would like to do only once and then keep the camera object in memory.

Soon we are going to add a web view into our application. To keep the user interface smooth, we will create the web view page using incubation.

Practicalities

1. Change the implementation of your main page so that the web view page in incubated.

2. Use `Qt.createComponent()` and `returnObject = incubateObject()` methods.

3. Method `incubateObject()` returns an object(`returnObject` above). Check whether this object `status` property equals to `Component.Ready`. If not, implement a signal handler for `statusChanged()` signal and check again in the signal handler, whether the `status` becomes `Component.Ready`.

4. When the status is ready, you will get the incubated object from the `object` property of the `returnObject`.

5. Define a variant property in the main page and assign the incubated object to that. Use the property to push the new page instead of the hard-coded URL.

6. Check in the web view page that it is dynamically created, but never destroyed, expect when you close the application.

Lab 7: Particles & Effects

Objectives

- Learn using the particle system and ready-made effects based on OpenGL shaders

Reference Material

labs/scripting	Starting point
labs/effects	Lab solution

Overview

Your task is to add a graphical effect and a particle system to the image dialog QML file. Be free to implement any kind of shaders, if you want. However, in the lab we ask you to apply just a ready-made effect.

Practicalities

1. Add at least one button to the effects page to start some graphics effects. Check you have copied the skeleton from **labs/templates**. You may use the button to test any effect you wish and add more buttons, if you want to have several effects.
2. Select any graphical effect from Qt graphical effects. Apply the effect to the image, but toggle the visibility of the effect on/off using your tool button and custom Boolean property.
3. Explode the image using the particle system. Implement an emitter and call its `burst()` method with, e.g. 1000 particles. At the same time, change the opacity of the image to 0.
4. Add a `ParicleSystem`, with `ImageParticle`, `Emitter`, and `Affector` to the effects page.
5. Anchor the `ImageParticle` to the whole area of the parent (root element). The source image (**blueStart.png**) is located in **images** folder. Add some `colorVariation` to have more than a single color for all emitted particles.
6. Anchor the emitter in the middle of the image. Set `emitRate` property to 0, meaning that the emitter does not emit any particles, before it will be asked. Set suitable `lifespan` and `lifeSpanVariation` properties in milliseconds. Set the `velocity` property to `AngleDirection` element, where `angle` property is 0, `angleVariation` is 180, `magnitude` is e.g. 60 and `magnitudeVariation` is 50. You may play with different values to see how they affect the particle emission.
7. Add `Affector`, e.g. `gravity`. The gravity has `magnitude` property. Test how it affects the particles.

You should have learnt that using effects is rather straightforward. However, creating custom effects with OpenGL shaders is complicated art. The particle system allows you to create a game with several emitters and affectors running at the same time. Look at the *affectors* demo in **demos** for further details.

Lab 8: WebView

Objectives

- Basic WebView usage

Reference Material

labs/effects	Starting point
labs/webView	Lab solution

Overview

In this lab, we will add Web View to our QML application. The minimum functionality is to have a QML element to input a URL and a button or corresponding element to navigate back.

Practicalities

1. Design the layout for your web view page. At the bottom, you should have the tool button to pop the page, but otherwise you may decide, where to have other input elements and the view itself. Use `WebEngineView` instead of `WebView`, which is Qt WebKit-based and deprecated.
2. Add an input element (e.g. `TextInput`) to the layout to type the URL. Load the URL, when the input is accepted.
3. Add a button (e.g. `Button`) to navigate to the previous page.
4. Note that you need to initialize the web engine in your **main.cpp** file (`main()` function). This requires you add Qt Webengine module to your application. The `initialize()` function is defined in `<QtWebEngine/qtwebengineglobal.h>`
5. Happy surfing.

Lab 9: C++ Integration

Objectives

- Learn QML/C++ integration

Reference Material

labs/webView	Starting point
labs/imageSharing	Example solution

Overview

Our application is almost complete. What needs to be improved is the model. So far, we have hard-coded the image URLs into the model. In this lab, we will use an existing C++ model class and use a file dialog in QML to add new files to the model. Obviously our view should now show the images in the C++ model.

Practicalities

1. The model class (`ImageModel`) has been implemented for you as this is QML and not C++ training. Copy the model header and C++ files from **labs/templates** to your project folder/subfolder and add them to your project.
2. Your task is to make this model available in QML. Additionally, change your QML code so that your view will use the model you just exposed instead of Enginio model or local `ListModel`.
3. Add `FileDialog` Qt Quick Control to your project. Use its `show()` method to open and show the dialog when a tool button is clicked (yes add another tool button to your main page).
    a. You may use `FileDialog folder` property to define the startup folder, e.g. `shortcust.home`.
4. Implement a signal handler, which is executed when the dialog is accepted. The selected files are in the array in the `fileUrls` property. Go through the URLs in the array and store URLs into the C++ model. Show the notification (the custom item we made in the first lab) to notify the user new images are added.
5. Check in the image view page that new images can be dynamically added to the model.

Time Permitting

1. Implement functionality to remove the images from the model. Note that you need to write the C++ implementation as well.

Lab 10: QML Rendering

Objectives

- looking for rendering bottlenecks

Reference Material

Your app or
labs/imageApp

Overview

It's time to analyze what we did. The application should be rather optimal from the performance point of view, but it is still worth profiling, whether we can find any bottlenecks.

Practical Outline

1. Use QML Profiler to see, if there are any
    a. time consuming object creations
    b. time consuming bindings
    c. time consuming JS functions

2. What is the frame rate of the application on your platform?

3. How many batches (state changes) are caused by the application in each page? What are the badges? Could they be combined?

4. Are there any overpaints in the application? Is something opaque painted behind opaque items? Should any item be invisible in some cases?

5. Any other performance optimizations? Share your thoughts with the class.