|  |
| --- |
| Programming Labs Handbook |
| QML Edition |
| Version 5.10 |

You hear you forget.

You see you remember.

You do you learn.

TABLE OF CONTENTS

Lab 1: Hello World 3

Lab 2: QML Essentials 5

Lab 3: Custom Items 8

Lab 4: Qt Quick Controls 10

Lab 5: Animations 13

Lab 6: Data Models and Views 15

Lab 7: Scripting and Dynamic QML 17

Lab 8: Particles & Effects 19

Lab 9: C++ Integration 21

Lab 10: QML Rendering 23

# Lab 1: Hello World

Objectives

* Basic QML file structure
* User interactions
* Qt Quick Designer

References

* labs/solutions/helloWorld One possible solution

Overview

Let’s create a simple QML application.

Practicalities

1. Launch QtCreator and use File menu to create a new QML project. Choose the project template “Qt Quick UI Prototype” from “Other Project” templates.
2. In “Define Project Details” page, check “With ui.qml file”. This will create a form, which can be edited with QML Designer. Other settings may have the default values.
3. Your project should contain now two QML files. MainForm.ui.qml and <project name>.qml. Open the <project name>.qml file. The file contains Window and MainForm objects.
4. Try changing the Window width and height properties. You may also change, for example, flags property to Qt.FramelessWindowHint.
5. Try changing Window color property. The color value may be defined in many ways: e.g. with SVG color names like “green” or hexadecimal strings like "#008000" or with Qt.rgba(r, g, b, a), where the arguments are given as red, green, blue, and alpha values in the range *[0, 1]*. What happens? Can you explain why? Hint! You can test MainForm visible and opacity properties.
6. Open the form. On the bottom left, you can see a Navigator pane. Click on Rectangle and change its color property. Run the application to see the rectangle color changes.
7. Drag a new Rectangle element to the UI. Make sure the new rectangle is a child of the root rectangle. You may use Navigator arrow buttons to change item hierarchy. Increase the new rectangle’s border width and radius and define a color for the border and the rectangle itself.
8. Drag Text into the center of the rectangle you just added. Now we would like to show the text in TextEdit also in Text element inside our Rectangle. Let’s use a binding for that. Select Text element and modify Binding so that Text.text is bound to TextEdit.text (use TextEdit id and change it in Navigator, if you wish).
9. TextEdit will get the focus if it is clicked on (look at TextEdit properties in QML designer). If we click on the window, the application will print out a log message. Add the signal handler to MainForm in <project name>.qml file so that quit() is called if the mouse is double clicked. If you run the application, possibly you do not see the rectangle you added, because it is not in the window area. In that case, remove the frameless window hint flag, so that you may resize the window to see your rectangle.
10. Run and test your UI. Click on text edit and type some text. Does the text in Text element change?
11. If time permits, add other elements, bindings, and interactions to the UI.

# Lab 2: QML Essentials

Objectives

* QML Designer
* Basic QML types
* Layout and anchoring

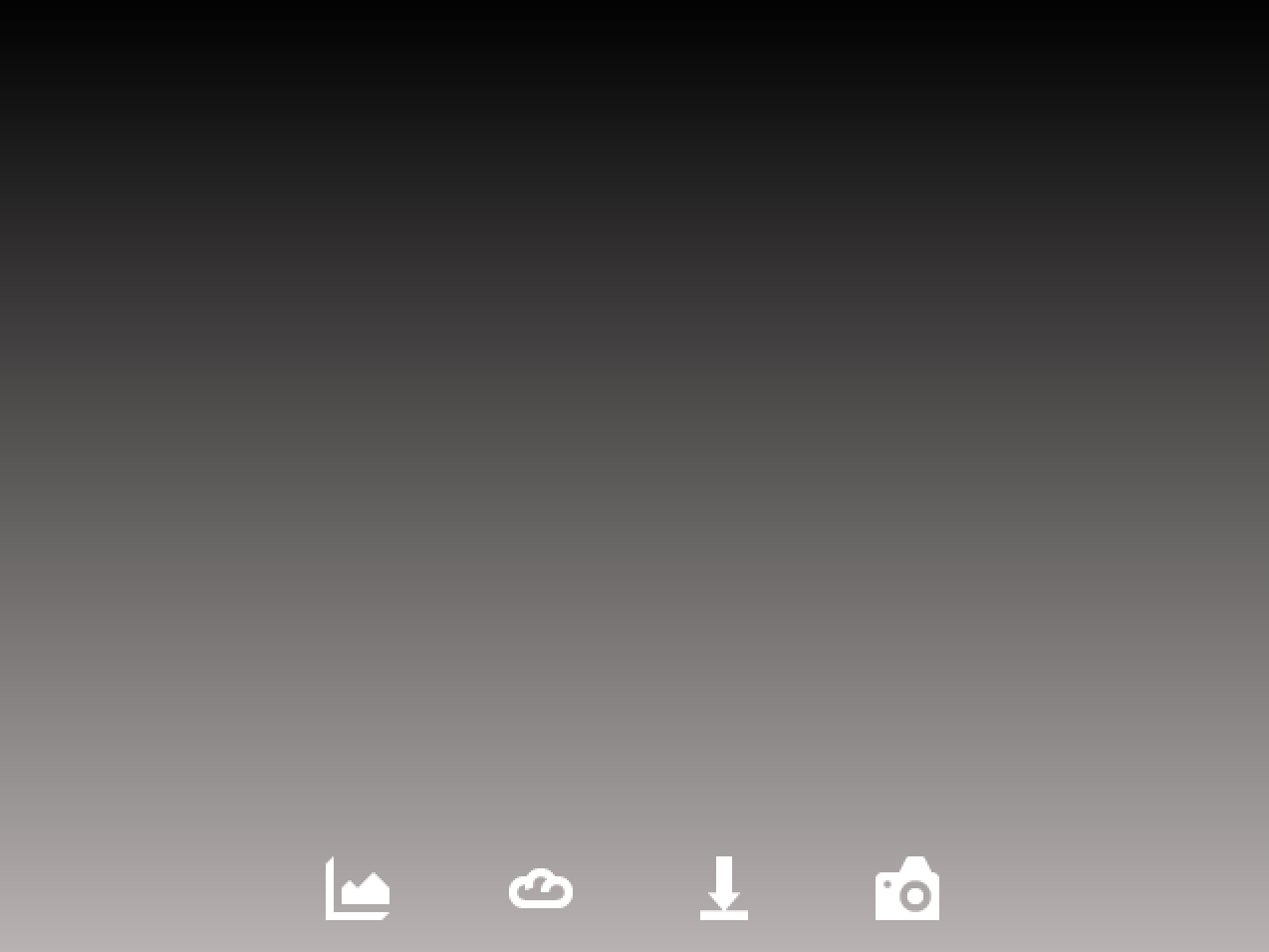
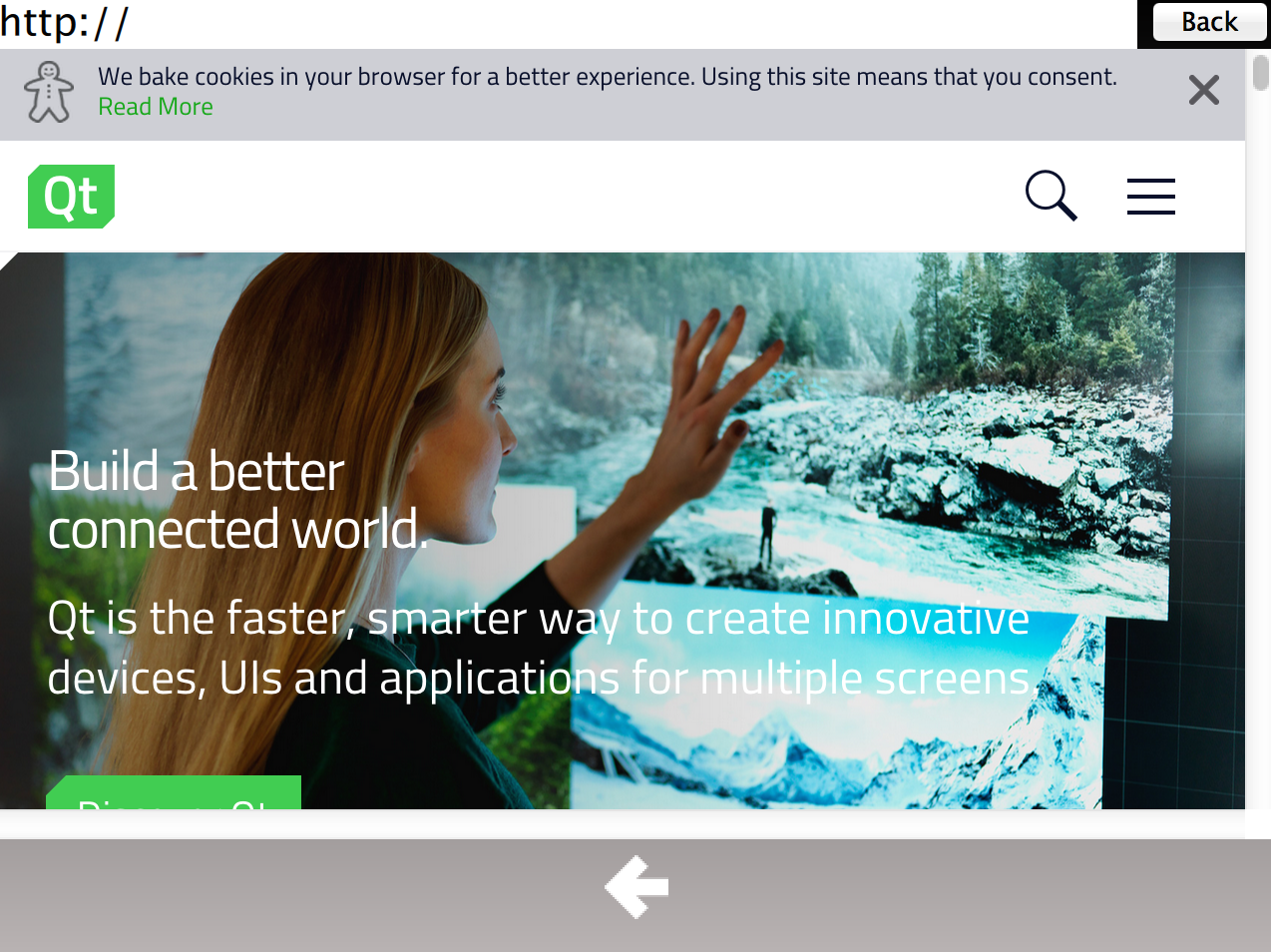
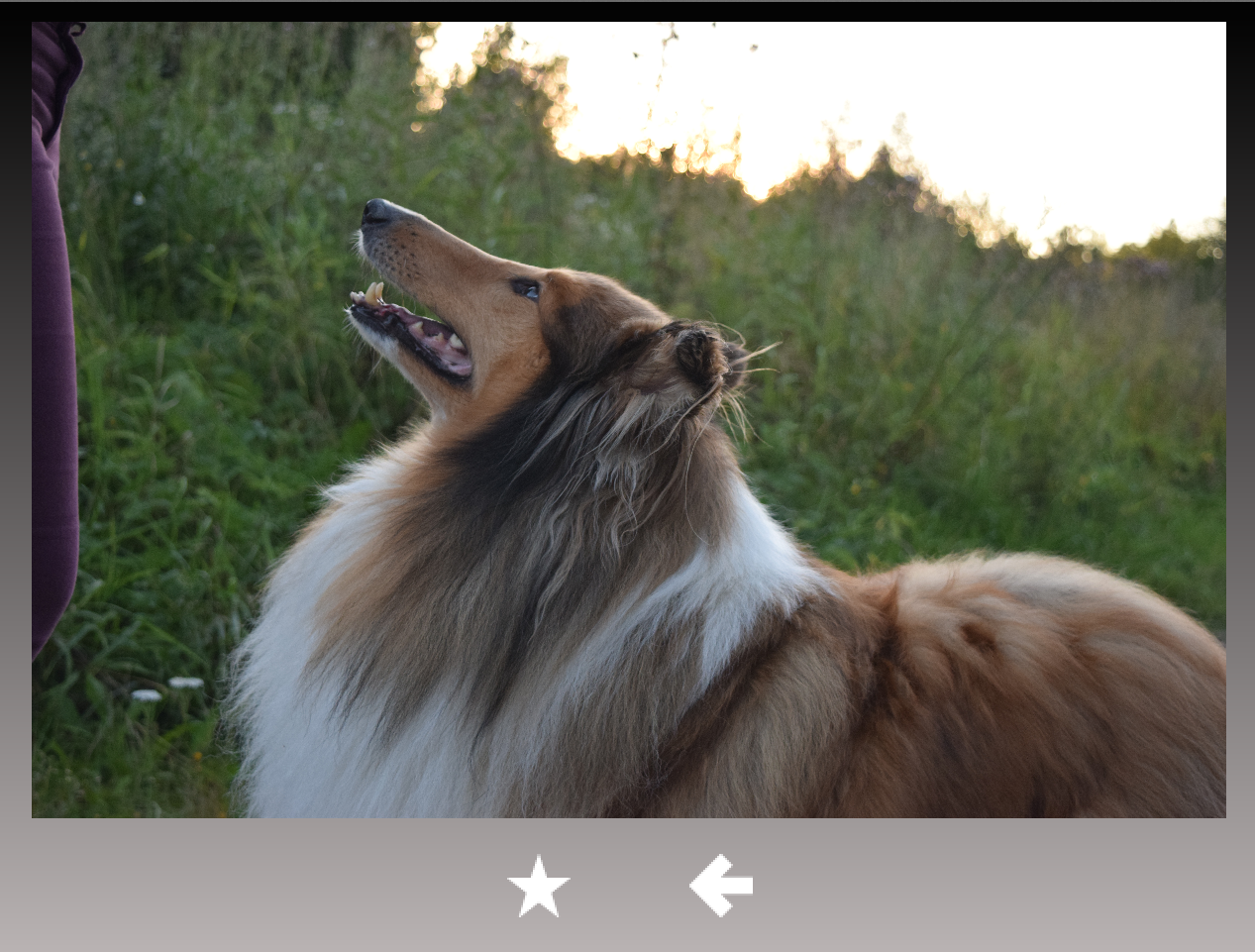
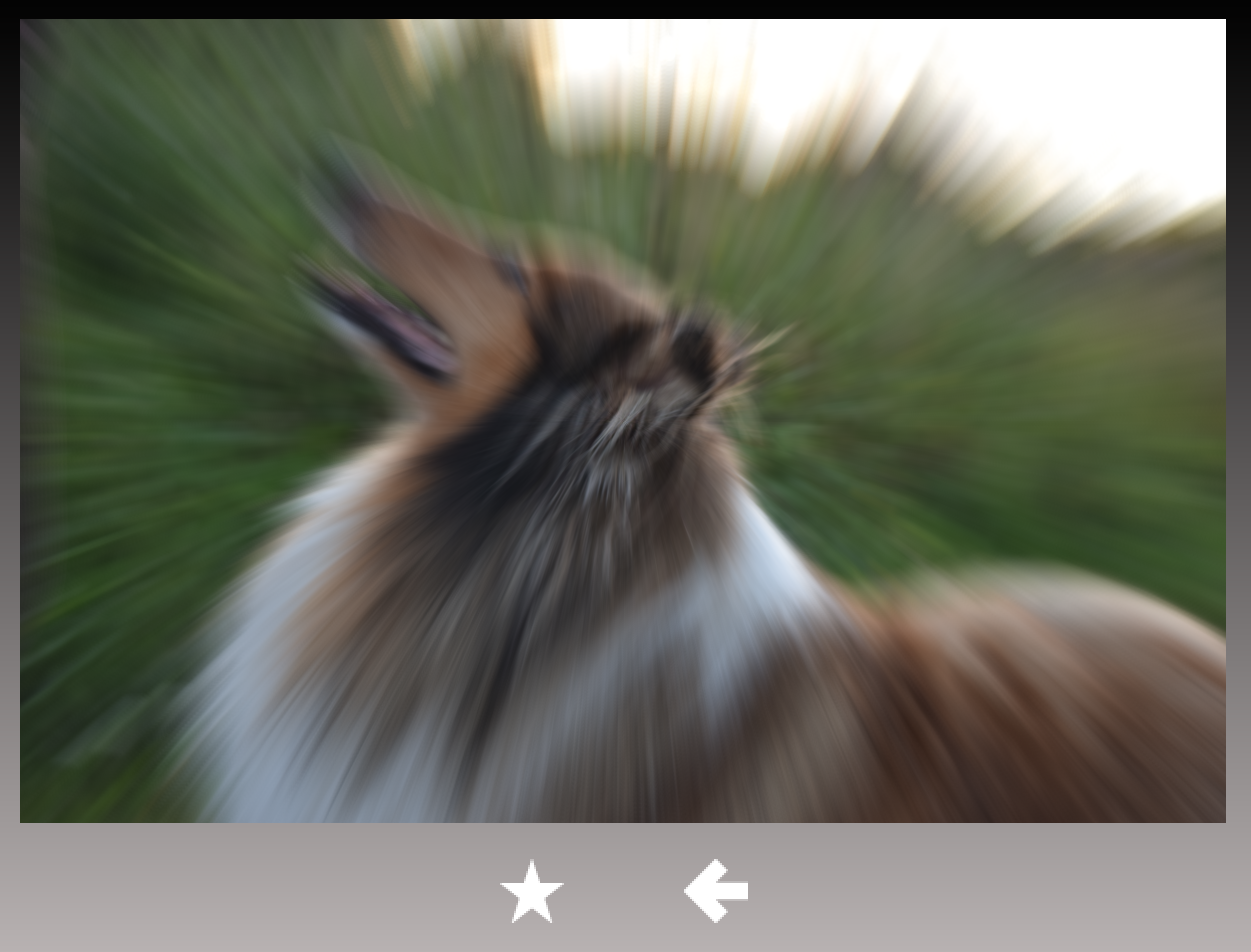
References

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

Overview

In the following labs, your task is to implement an image application. The application uses a QML view to show images, loaded from the file system or captured using a camera. 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, capturing images, showing a web view or asking the user to add new 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 creating 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 QML type (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.

Practicalities

1. Create a new “Qt Quick Application”, using the project wizard in your Qt Creator IDE. In “Define Project Details” page, **uncheck** “With ui.qml file”.
2. Change Window to ApplicationWindow. Change also import from import QtQuick.Window 2.2 to import QtQuick.Controls 2.0. Remove MouseArea and TextEdit QML types from main.qml file as we create about everything ourselves. Add Rectangle QML type to main.qml 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 QML type 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 QML type dimensions as well. Otherwise, the text does not scale.



1. Notification item 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 QML type. 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 QML type.
   1. You may define a function in JavaScript like function helloWorld(arg1, arg2) { }
2. Can child items paint outside their parents?
3. Can items overlap?
4. 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 3: Custom Items

Objectives

* Scope and visibility rules
* Signals and signal handlers

References

* Your own solution to Lab 2 Lab starting point
* Or labs/solutions/qmlEssentials
* labs/solutions/customItems One possible solution

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.

Practicalities

1. Create a new QML file for your notification message. In Qt Creator, right-click the project name and “Add New…”.
2. Copy Rectangle QML type you implemented in main.qml in the previous lab to the new QML file. Replace the default Rectangle QML type in that file.
3. The notification could be closed, when the user touches it, but we use a timer instead. Add Timer QML type 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 item. 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.
   1. 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 4: Qt Quick Controls

Objectives

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

References

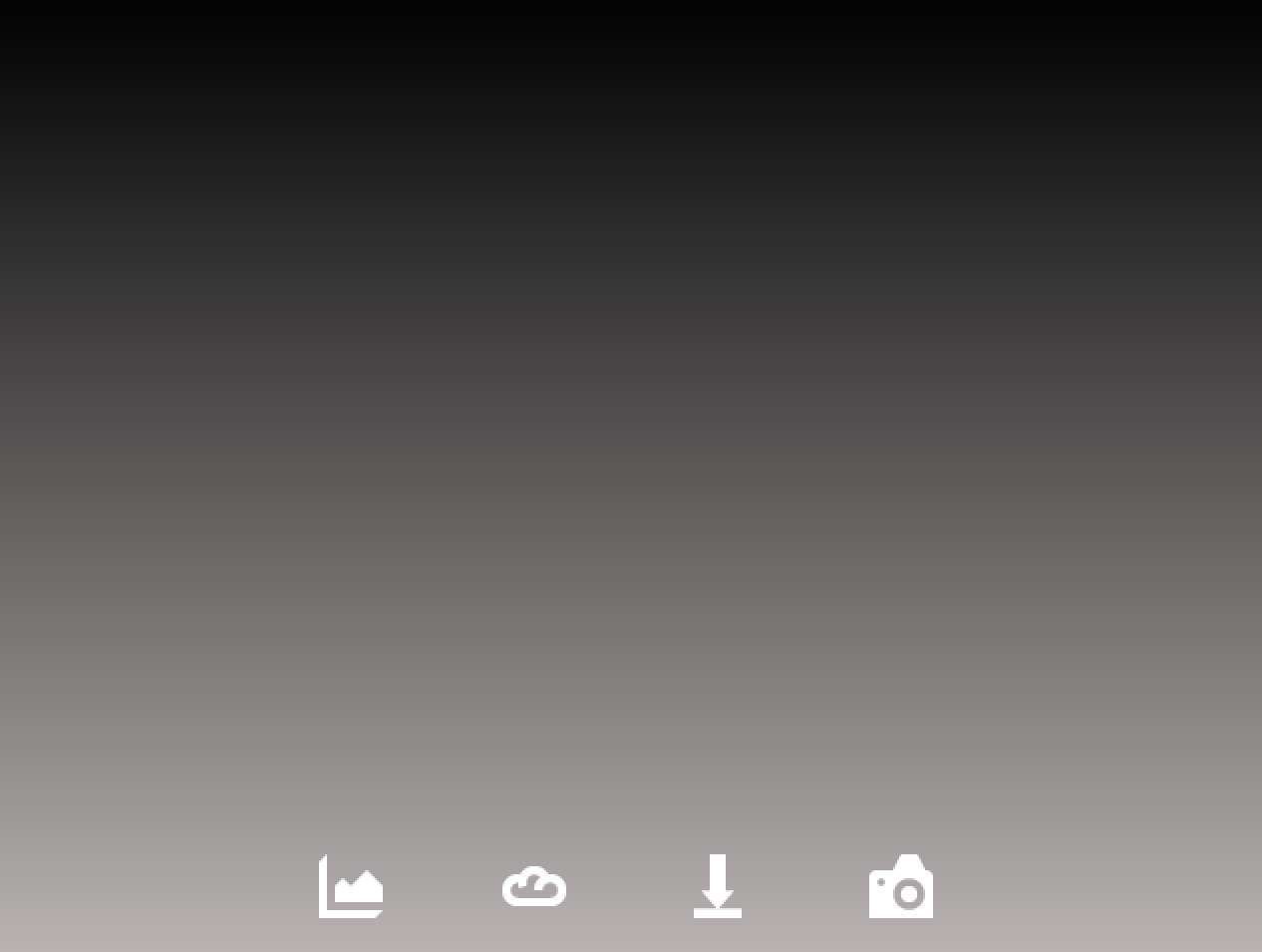
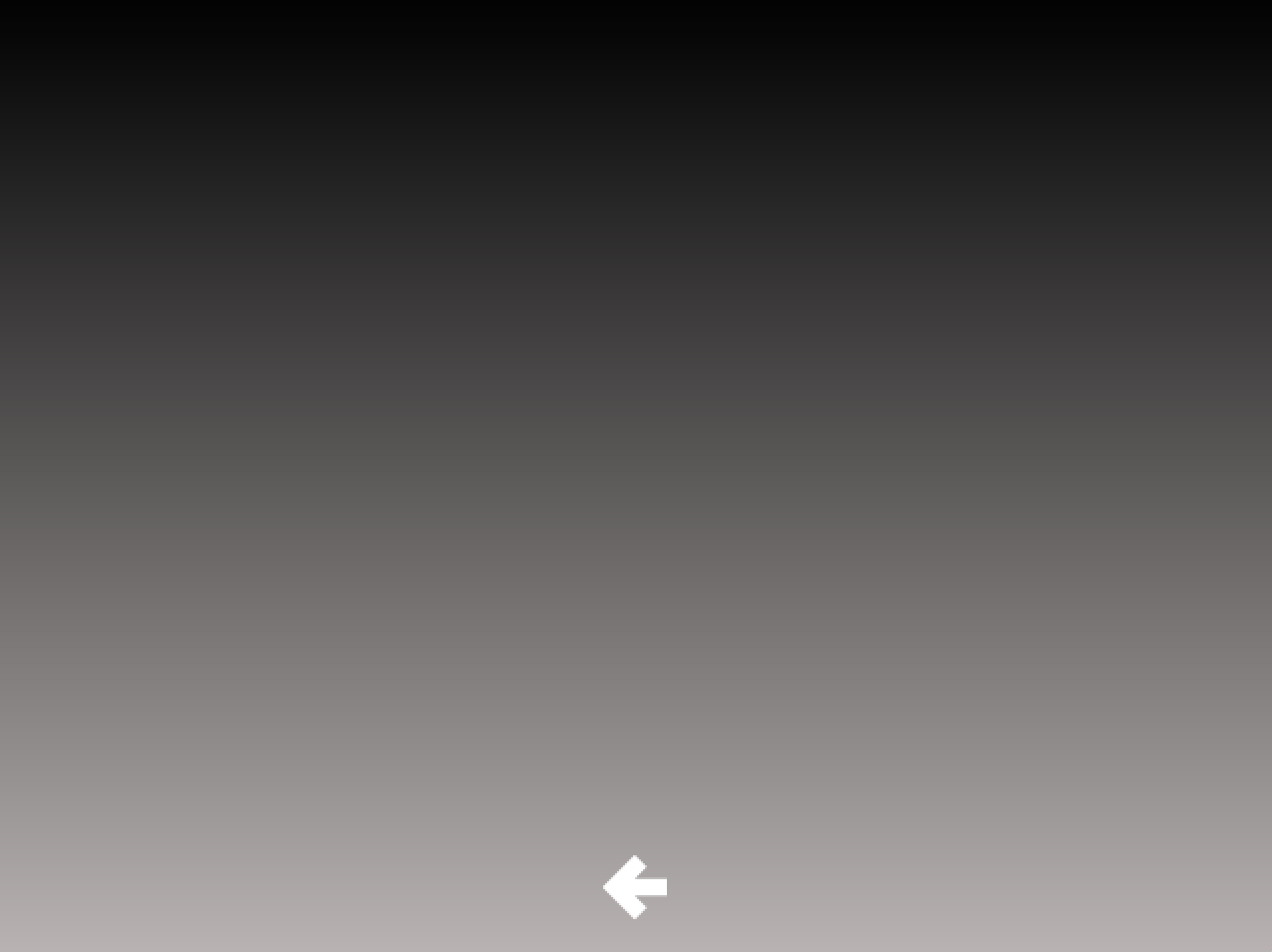
* Your own solution to Lab 3 Lab starting point
* Or labs/solutions/customItems
* labs/solutions/qtQuickControls One possible solution
* labs/solutions/qtQuickControls2 Quick Controls 2 solution for reference

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.

Practicalities

1. Instantiate a new QML type 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, image capture page, and web view page.
3. Let’s use ToolButton objects from Qt Quick Controls to navigate between the pages. Add ToolBar QML type 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 four buttons: one to navigate to the image view page, one for image capture page, one for web view page, and one for file dialog. 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.
   1. 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 four pages and four buttons in the initial page to navigate between them.

Time Permitting

1. Compare differences between Quick Controls and Quick Controls 2 StackView and ToolButton by taking a look at labs/solutions/qtQuickControls2 implementation.

# Lab 5: Animations

Objectives

* Behavior animations
* Property animations
* Stack view transitions

References

* Your own solution to Lab 4 Lab starting point
* Or labs/solutions/qtQuickControls
* labs/solutions/animations One possible 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 QML type 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 QML type in main.qml. Instantiate StackViewDelegate QML type. 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 6: Data Models and Views

Objectives

* Model/view framework in QML

References

* Your own solution to Lab 5 Lab starting point
* Or labs/solutions/animations
* labs/solutions/modelView One possible solution

Overview

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

Images are added statically in the code to the model. Optionally, a camera may be used, if you have a camera in your laptop.

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, 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

}

}

1. Next we will need a delegate. Your delegate should have at least an Image QML type with the size you prefer. Note that the Image source property should be read from model’s 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.
2. 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. If you have access to a camera in your computer, you may use camera to add images to the model.
2. Add a Camera QML type (in QtMultimedia module) to ImageCapturePage. Camera provides imageCapture property, in which you may implement s signal handler for imageCaptured(requestId, preview) signal. The preview parameter is the captured image URL. You may store the URL into the image model using the ListModel.append({“propertyName”: value}) function.
3. Camera itself does not render anything. We need VideoOutput for that. It is a normal visual item, so define the layout for the item using, e.g. anchors. VideoOutput source property defines, where the actual images come from. Assign the Camera object identifier to source property.
4. Add ToolButton to capture an image. You may use the function imageCapture.capture(). You may also specify the location yourself using captureToLocation(path), if the default location does not work.

# Lab 7: Scripting and Dynamic QML

Objectives

* Dynamic loading

References

* Your own solution to Lab 6 Lab starting point
* Or labs/solutions/modelView
* labs/solutions/scripting One possible 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, except when you close the application.

# Lab 8: Particles & Effects

Objectives

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

References

* Your own solution to Lab 7 Lab starting point
* Or labs/solutions/scripting
* labs/solutions/effects One possible 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 ParticleSystem, with ImageParticle, Emitter, and Affector to the effects page.
5. Anchor the ImageParticle to the whole area of the parent (root item). 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 QML type, 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 9: C++ Integration

Objectives

* Learn QML/C++ integration

References

* Your own solution to Lab 8 Lab starting point
* Or labs/solutions/effects
* labs/solutions/imageApp One possible 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. 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 the local ListModel, we used earlier.
3. Add FileDialog Qt Quick Control to your project. Use FileDialog show() method to open and show the dialog when the corresponding tool button is clicked. If you do not have an extra tool button, add a new one.
4. You may use FileDialog folder property to define the startup folder, e.g. shortcuts.home.
5. 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.
6. Check in the image view page that new images can be dynamically added to the model.
7. Our web view page is still empty. You may copy one implementation from labs/templates/WebViewPage.qml. Note that we have to initialize the web engine, before it can be used. Add webengine module to your project .pro file and initialize the webengine using the QtWebEngine::initialize() function in the main() function. The initialize() function is defined in <QtWebEngine/qtwebengineglobal.h>. Navigate to the web view page and test, whether you may navigate to web pages.

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

References

* Your own QML app Lab starting point
* Or labs/solutions/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.

Practicalities

1. Use QML Profiler to see, if there are any
   1. time consuming object creations
   2. time consuming bindings
   3. 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?

1. Are there any overdraws in the application? Is something opaque painted behind opaque items? Should any item be invisible in some cases?
2. Any other performance optimizations? Share your thoughts with the class.