

THE PHOTO VIEWER APPLICATION

Andrey Koptev
Bachelor's thesis
Spring 2011
Degree Programme in Information Technology
Oulu University of Applied Sciences

PREFACE

This Bachelor's thesis was commissioned by Digia Finland Oy, Helsinki, Finland.

Digia is one of the leading ICT companies in Finland with around 1,600 employees all over the world. Digia's offering includes ERP systems, and mobile and user experience services and solutions. The customers of the company are businesses and organizations from various industries, with an emphasis on public administration, industry, mobile industry, retail, services, banking and insurance.

ACKNOWLEDGEMENTS

I would like to thank Digia Finland Oy in general and Sami Koivumäki in person for the great opportunity of completing my Bachelor's thesis work. I would also like to express my gratitude to Leo Ilkko, my thesis supervisor, for his excellent tutoring and organizing skills, to all my university teachers who helped me to develop my competence, and to all my colleagues in Digia Mobile Solutions department for their advice and willingness to help.

Special thanks go to my friends who believed in me no matter what the cost was.

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Mobile Technology

Author: Andrey Koptev
Title of Bachelor's thesis: The Photo Viewer Application
Supervisor: Leo Ilkko
Term and year of completion: Spring 2011
Number of pages: 39 + appendices 7

The aim of this Bachelor's thesis was to prove the concept of building flexible and dynamic user interfaces for Symbian^3 mobile phone platform by applying declarative programming techniques and combining them together with the modern powerful programming language – Qt.

To achieve that aim it was decided to construct a photo viewer application that would be run on Symbian^3 devices. The key feature of that application would be a dynamic user interface with multiple views for representing pictures located on the phone's memory.

The design was implemented using the UML language. This approach allowed realizing important milestones and it also helped to drive the development process in the right direction. The structure of the application was split into two major parts – Engine logic and UI. The engine's code was implemented using the Qt programming language. The UI was done by applying a declarative programming approach with the help of the QML language and Qt Declarative module. The Nokia N8 smartphone was chosen to be a target device for testing the application.

Although the project was internal competence development work during my induction period as a software developer in Digia Finland Oy company, the application still carries significant potential for future upgrading and a wide utilization on the open market. It may especially be useful for developers who are interested in producing different dynamic UIs for their own applications and who consider applying declarative programming techniques for this purpose.

Keywords:

Qt, QML, QtQuick, Symbian, user interface, pictures, views

TIIVISTELMÄ

Oulun seudun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, Mobiiliteknologia

Tekijä: Andrey Koptev
Opinnäytetyön nimi: Valokuvien katseluohjelma
Työn ohjaaja: Leo Ilkko
Työn valmistumislukukausi ja -vuosi: Kevät 2011
Sivumäärä: 39 + liitteet 7

Tämän opinnäytetyön tavoite oli konseptoida joustavia ja dynaamisia käyttöliittymiä Symbian^3-matkapuhelinalustalle. Deklaratiivisia ohjelmointitekniikoita yhdisteltiin tehokkaassa ja modernissa Qt-ohjelmointikehyksessä.

Tavoitteen saavuttamiseksi suunniteltiin valokuvien katseluohjelma, joka toimii Symbian^3-laitteissa. Dynaaminen käyttöliittymä, jossa on useita eriselausnäkyviä, ja jotka näyttävät kuvia puhelimen muistista, on ohjelmassa avainasemassa.

Suunnitelma toteutettiin UML-kielellä. Se mahdollisti tärkeiden virstanpylväiden saavuttamisen ja ohjasi kehitysprosessia oikeaan suuntaan. Ohjelman rakenne on jaettu kahteen tärkeään osaan: ohjelmalogiikka ja käyttöliittymä. Ohjelmalogiikka toteutettiin Qt-ohjelmointikielellä. Käyttöliittymä tehtiin deklarativisten ohjelmointitekniikoiden, QML-kielen ja muiden deklarativisten Qt-komponenttien avulla. Ohjelman testaamista varten valittiin Nokia N8-älypuhelin.

Ohjelma on sisäistä kompetenssinkehittämistä varten tehty ollessani ohjelmistokehittäjänä Digia Finland Oy yhtiössä. Sillä on silti merkittävää potentiaalia tulevaisuuden laajennuksille ja laajalle hyödyntämiselle vapilla markkinoilla. Se voi olla erityisen käytännöllinen kehittäjille, jotka ovat kiinnostuneita tekemään erilaisia dynaamisia käyttöliittymiä omiin ohjelmiinsa, ja jotka harkitsevat deklarativisten ohjelmointitekniikoiden käyttämistä.

Asiasanat:

Qt, QML, QtQuick, Symbian, käyttöliittymä, kuvat, näkymät

TABLE OF CONTENTS

PREFACE	1
ACKNOWLEDGEMENTS	1
TABLE OF CONTENTS	4
SYMBOLS AND ABBREVIATIONS	6
1. INTRODUCTION	7
1.1 Research problems and methods	8
2. WORKING ENVIRONMENT	9
3. DEFINITION	10
3.1 User's perception	11
3.2 Technologies	12
3.2.1 Symbian platform	12
3.2.2 Qt technology	13
3.2.3 Declarative programming	15
3.2.4 Qt Quick	16
3.2.5 QML	16
3.2.6 MVC design pattern	17
3.2.7 Image file formats	19
4. IMPLEMENTATION	20
4.1 Architecture	20
4.2 Engine	21
4.3 Model and delegates	23
4.4 UI Controller	24
4.5 User Interface	26
4.5.1 Grid view	27
4.5.2 3D view	28
4.5.3 List view	29
4.5.4 Settings screen	30
5. TESTING	31

6.	FUTURE DEVELOPMENT	32
	6.1 Functionality	32
	6.2 Video playback	32
	6.3 Metadata	33
	6.4 Location	33
	6.5 Connectivity	34
	6.6 Social sharing	35
7.	CONCLUSION	36
8.	LIST OF REFERENCES	37
9.	APPENDICES	39

SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
EXIF	Exchangeable Image File Format
GIF	Graphics Interchange Format
GPS	Global Positioning System
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
JEIDA	Japan Electronic Industries Development Association
JPEG	Joint Photographic Experts Group
MMS	Multimedia Messaging Service
MOC	Meta Object Compiler
MVC	Model-View-Controller
OS	Operating System
PNG	Portable Network Graphics
POI	Point Of Interest
QML	Qt Meta-Object Language
Qt QUICK	Qt User Interface Creation Kit
REST	Representational State Transfer
SDK	Software Development Kit
SMS	Short Messaging Service
SQL	Structured Query Language
TIFF	Tagged Image File Format
UI	User Interface
UML	Unified Modeling Language
XML	Extensible Markup Language
Wi-Fi	Wireless Fidelity
WWW	World Wide Web

1. INTRODUCTION

Traditionally all desktop and mobile applications were developed using imperative programming languages and techniques - by describing computation in terms of statements that change a program state. This approach is simple and understandable. It is much easier to describe a sequence of actions that are needed to be done to complete a task, then to present a task in a form fully understandable to a machine. However, in terms of user interface development this approach is prone to failures.

On the contrary, web developers are used to describing web applications by defining *what* the program should accomplish without prescribing *how* to do it in terms of sequences of actions to be taken. This approach is called a declarative programming and, unfortunately, in traditional desktop or mobile applications it is still rarely used, compared to an imperative programming.

In order to break that tendency and drive a higher utilization of the declarative approach in the mobile segment, Qt Software has recently came up with a new project called Qt Declarative User Interface and introduced a new markup language - QML.

“QML is a declarative language designed to describe the user interface of a program: both what it looks like, and how it behaves. In QML, a user interface is specified as a tree of objects with properties”. (Nokia 2010, date of retrieval 21.3.2011)

The user interface is basically a certain area where the user interacts with the program flow. The main goal of that interaction between the user and the program is to control the effective execution of the latter.

UIs designed by the QML language are often referred to as fluid or dynamic. This description comes from a certain type of behaviour a program expresses when interrupted by a user's action. The behaviour could be e.g. flicking, flipping, dimming, bouncing or another sort of animation. The secret of the UI's fluid behaviour lies in animated transitions between the sets of QML properties. Thus, for example, by changing the value of opacity one can force objects to become fully or partially transparent.

“The Qt Declarative module implements the interface between the QML language and the elements available to it. It also provides a C++ API that can be used to load and interact with QML files from within Qt applications”. (Nokia 2010, date of retrieval 21.3.2011)

The QML language could be fully extended by a C++ code via the Qt Declarative module.

Both the QML language and the Qt Declarative module are parts of Qt Quick – a powerful framework that contains a rich set of user interface elements, a declarative language for describing user interfaces and a language runtime environment.

1.1 Research problems and methods

During the development stage the following problems are required to be resolved:

1. Building an application that would be responsible for locating and displaying images
2. Defining the user experience for a dynamic and fluid multi-view data representation
3. Deploying the application on a target device

Images could be of various data types and must be scaled to a certain degree depending on geometry or the specific content layout of the current view.

Qt was chosen as the programming language for the engine's code mainly because of its flexibility and support of the target platform – Symbian^3. The QML language support was necessary for running the application on the mobile phone. Unfortunately, at the early stages of development that support did not exist. QML became available for mobile phones only with the release of Qt 4.7.1 for Symbian libraries which happened approximately at the end of the 4th quarter of 2010. (Nokia 2010, date of retrieval 16.12.2010)

The research process consisted of exploring various sources of information for gathering, combining and processing data and realizing possible activities in pursuit of effective solutions for problems described above.

Internet articles, technical literature and open source communities were widely used as the main available sources of information in this project.

2. WORKING ENVIRONMENT

In order to achieve the major goal of the project and to find an effective solution for the defined research problems, Qt SDK 1.1 Technology Preview was used as a main working environment. The reason for that choice is that it is a simple and easy way to design, debug and deploy applications.

The SDK contains a powerful IDE – Qt Creator 2.1 which includes the first iteration of tooling support for Qt Quick. Additionally the SDK contains Qt 4.7.1 for Symbian libraries that are used in compiling and are available as a sis packages for installing on a target device.

These are used for running the application on the phone.

At the early stages of development, before the Qt SDK had been officially released, Symbian^3 SDK served as a major development framework.

The Qml viewer tool was used for development and testing purposes. It was useful for testing and debugging the QML based user interfaces.

The GCCE toolchain was used for compiling the application.

The Symbian AppTRK tool was used for on-device debugging.

StarUML 5.0 was used for modelling UML diagrams.

3. DEFINITION

The system is regarded as a software demonstrator for proving the concept of building flexible and dynamic user interfaces for Symbian^3 mobile phone platform, by applying declarative programming techniques and methods.

The Photo Viewer application (which later in this document may be referred to simply as “the application”) is a software program that will aid the research process and serve as a base for concept demonstration purposes.

The application will be responsible for showing images contained on a host’s memory.

By the term “host” one shall consider a mobile device on which the application will be run.

In order to support all required features the host must be running the latest Symbian Operating System – Symbian^3.

Major emphasis will be given to the way how images would be presented to the user. Multiple graphical views would be used to demonstrate it.

The development process will be split into three stages:

- Building the functionality for harvesting the image data
- Designing the graphical user interface
- Combining both previous parts into one package

3.1 User's perception

From the user's point of view the application must be responsive and entertaining.

The UML diagram on the Figure 1 explains the most relevant user stories that are valid for the system:

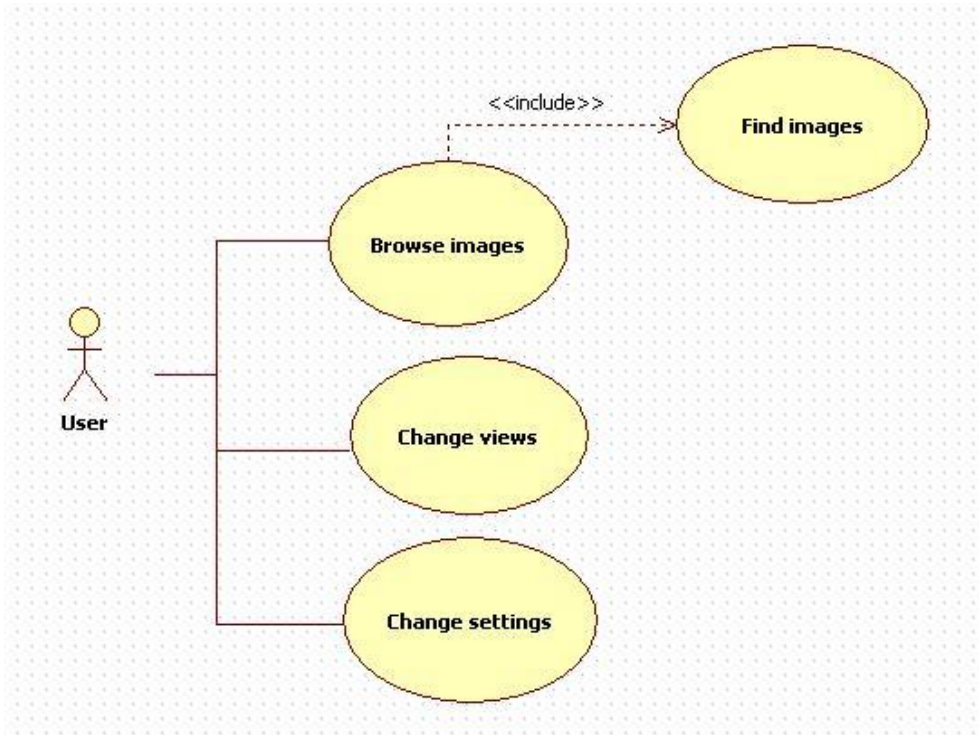


FIGURE 1. The system overview.

The application must allow a user to browse images, located on the phone's memory. In order to achieve that, the application must provide functionality for locating image files on the memory. To help the application find the desired images, the list of settings is introduced.

The list of settings is a dialogue-based screen, where the user will be specifying the proposed location of image files, and also filtering image file formats.

In addition to all above, the user also must be able to change the visual representation of the desired image files. Adding multiple different layouts will provide the possibility of browsing images in different views, therefore making the whole application look entertaining.

3.2 Technologies

3.2.1 Symbian platform

Symbian is an open source operating system and a software platform designed to be deployed on smartphones i.e. mobile phones that offer a more advanced computing ability and connectivity than legacy mobile devices.

“Symbian is the world’s most popular smartphone platform. It’s implemented in a diverse range of devices and provides app and media developers with a consistent set of technologies. The flexibility of Symbian means it can offer users classic mobile devices, utilising a standard keypad and QVGA screen, through to high-end smartphones that offer nHD touch screens with tactile feedback, full keyboards, and device sensors in innovative flip and slide form factors. Equally at home delivering advanced enterprise apps, games, or music, Symbian gives developers unparalleled opportunities in the mobile space”.
(Nokia 2010, date of retrieval 24.3.2011)

The Symbian operating system sees a wide utilization among major digital mobile device manufacturers, including its producer – Nokia (Figure 2). For several years until the current moment Symbian is still the leading operating system available on the mobile phone market, having about 36,6 % of its share.

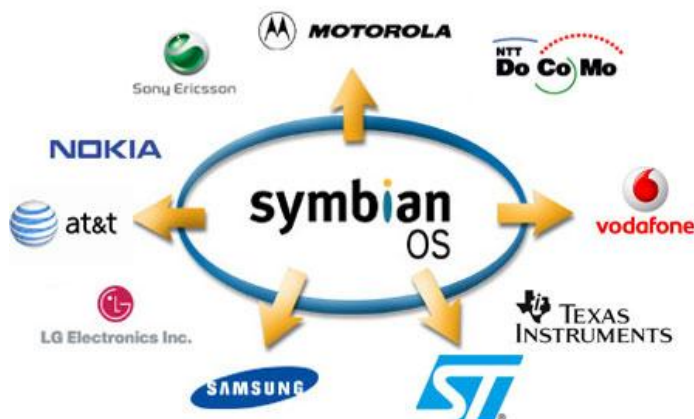


FIGURE 2. Symbian platform utilization.

The Symbian platform is based on the EPOC software architecture originally developed by Psion.

“EPOC is one of the most exciting (if not the most exciting) C++ programming systems available today. The operating system, with a solid object-oriented design, is combined with particularities that are a necessity for hand-held devices. This includes power management built within the kernel, sophisticated memory management, event handling mechanisms, and effective multitasking”.
(Nokia Mobile Phones 2000, date of retrieval 24.3.2011)

The latest and the most advanced version of the Symbian platform, called Symbian^3, was released in the fourth quarter of 2010.

3.2.2 Qt technology

Qt is a cross-platform application framework used for developing a software with GUI as well as console applications, UI forms and scripts. It is popular among many developers due to its rich toolkit with ready-made visual components, called widgets.

“Qt allows open source and commercial software developers to code less, create more and deploy everywhere. With Qt, developers can build innovative applications and touch-enabled user interfaces once and then deploy across all major mobile, desktop, consumer electronic and embedded platforms without rewriting the code.” (Nokia 2010, date of retrieval 23.3.2011)

The cross-platform capability allows users to save development time for porting their applications to different platforms. With the exception of some platform specific libraries, general Qt core elements and features are fully compatible with the following operating systems:

- Windows
- Linux/X11
- Mac OS
- Embedded Linux
- Symbian
- Maemo/MeeGo
- Windows CE

In addition to the standard C++ language, Qt extensively uses a special code generator - Meta Object Compiler, or MOC. Although Qt is most popular among C++ developers, it also contains bindings to other famous programming languages, such as:

- Java
- Ruby
- Python
- C#
- Perl
- BASIC
- PHP

Qt features include 2D/3D drawing and hardware accelerated graphics support, SQL database access, XML parsing, thread management, inter-object communication, network connectivity, low-level multimedia functionality, a web browser environment with real-time web content and services, and scripting.

All these features are split into separate modules and libraries (Figure 3).

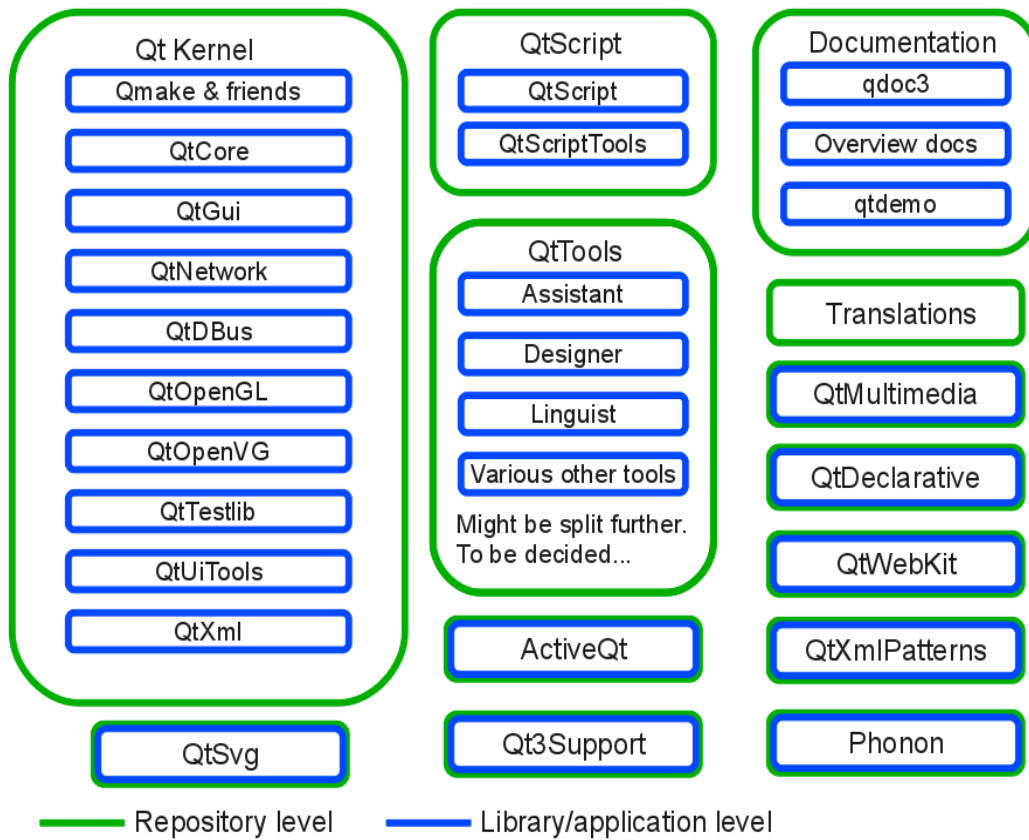


FIGURE 3. Qt Modularization.

Qt is currently produced by Nokia's Qt Development Frameworks division, which came into being after Nokia's decision to acquire the original founder, Norwegian company Trolltech, in January 2008.

3.2.3 Declarative programming

In computer science, declarative programming is a paradigm of expressing the logic of computation. The key approach is to model the results that the user expects without explicitly describing the control flow.

“Declarative programming is a way of specifying what a program should do, rather than specifying how to do it. Most computer languages are based on the steps needed to solve a problem, but some languages only indicate the essential characteristics of the problem and leave it to the computer to determine the best way to solve the problem. The former languages are said to support imperative programming whereas the latter support declarative programming”. (Archana Khambekar, C. Wilborn 2010, date of retrieval 24.3.2011)

Declarative programming is an umbrella term under which a number of other famous programming paradigms may be referenced. Those branches include:

- Constraint programming
- Domain-specific languages
- Functional programming
- Hybrid languages
- Logic programming

“Declarative programs are context-independent. Because they only declare what the ultimate goal is, but not the intermediary steps to reach that goal, the same program can be used in different contexts”. (Jörg W. Mittag 2008, date of retrieval 24.3.2011)

There are two main advantages of using declarative programming languages over the imperative ones. The first one is that the program becomes concise and easy to understand.

“The second advantage of the declarative programming model is that repetitive imperative code that indicates how to solve things is provided in the computer system behind the scenes. Such code can be made highly efficient and can incorporate the best ideas from computing. It can take advantage of parallelism.” (Wisageek 2010, date of retrieval 24.3.2011)

There are not currently so many languages that utilize declarative programming techniques, but the most famous of them are SQL and QML.

3.2.4 Qt Quick

Qt Quick, as Qt User Interface Creation Kit, is a powerful framework which contains a rich set of user interface elements, a declarative language for describing user interfaces and a language runtime environment.

“Qt Quick is a framework that provides a declarative way of building custom, highly dynamic user interfaces with fluid transitions and effects, which are becoming more and more common especially in mobile devices”. (Ryan Paul 2010, date of retrieval 25.3.2011)

Generally, Qt Quick can be described as a collection of the following technologies:

- QtDeclarative module
- QML language
- QtCreator IDE support

Qt Quick is officially supported starting from Qt 4.7.

3.2.5 QML

QML, as Qt Meta-Object Language, is a flexible script-like, declarative language for designing applications with fluid and dynamic user interfaces.

“QML is a declarative language designed to describe the user interface of a program: both what it looks like, and how it behaves. In QML, a user interface is specified as a tree of objects with properties”. (Nokia 2010, date of retrieval 21.3.2011)

Originally, QML was considered to be an extension to ECMAScript (cf. JavaScript). The QML language provides a mechanism to construct an object tree of QML elements, and enables the interaction between those elements and Qt C++ objects.

QML is a part of Qt Quick. Generally, QML is aimed to mobile applications where a good user experience is an important issue.

3.2.6 MVC design pattern

Model-View-Controller, in short MVC, is a classic software design pattern often utilized by applications that require the using of multiple views for representing the same data. Originally MVC was introduced in the Smalltalk-80 programming language, which uses it for building user interfaces.

“MVC consists of three kinds of objects. The Model is the application object, the View is its screen representation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse”. (Gamma, Helm, Johnson, and Vlissides 1994, 4)

The structure of the application that was built by using a MVC pattern is shown on Figure 4.

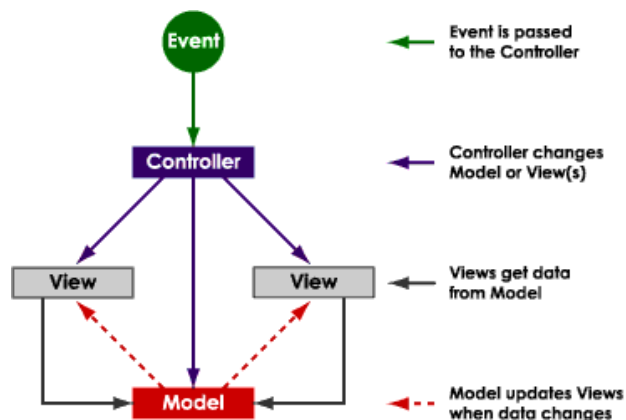


FIGURE 4. Application structure and the main processes inside the MVC pattern.

The control flow inside an MVC structured application will start from an event that was triggered by the user's action.

MVC allows the users to change the way the view responds to an input without changing its visual representation. MVC encapsulates the response mechanism in a Controller object. (Gamma et al. 1994, 5)

The controller's responsibility is to handle events that affect the model or views. The controller receives the user input and instructs the model and views what actions should be taken in order to react on that input.

The model maintains the data and manages the behaviour of the application. The model can be represented by various data structures, such as database tables, XML documents, or lists of abstract objects.

The view object renders the model into a form suitable for interaction. Multiple views can be linked to a single model. A constant bidirectional communication between the model and views is maintained for the purpose of monitoring and applying data changes.

By combining the functionality of both View and Controller objects, one shall be utilizing the simplified Model/View architecture. (Figure 5)

“This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles. This separation makes it possible to display the same data in several different views, and to implement new types of views, without changing the underlying data structures.” (Nokia 2010, date of retrieval 24.3.2011)

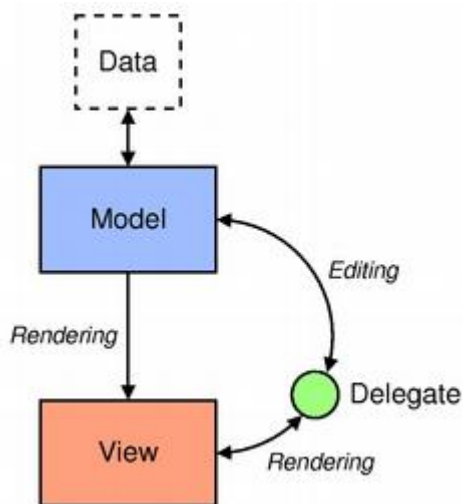


FIGURE 5. Model/View architecture.

For the flexible handling of the user input, delegates are introduced in the architecture. A delegate is a component that describes a prototype item of each piece of data in the model. The delegates are used for creating the instances of items in the model for the view.

The key relationships inside the Model/View architecture could be grouped as follows:

- The model communicates with a source of data, thus providing an interface for other components in the architecture.
- The view communicates with the model in order to obtain model indexes i.e unique values referenced to items of data.
- A delegate communicates with both the model and the view for rendering the items of data.

Usually, all three groups of elements - models, views and delegates are split into separate classes. Splitting allows replacing and adding new instances to them easily, thus extending the overall architecture.

3.2.7 Image file formats

The list of image file formats supported by the photo viewer application contains:

- JPEG

“As Joint Photographic Experts Group, the abbreviated name of the committee that created the JPEG standard.

JPEG is the most common image format used by digital cameras. It is also the most common format for storing and transmitting photographic images on the World Wide Web”. (JPEG 2007, date of retrieval 25.3.2011)

- TIFF

“Tagged Image File Format is a file format for storing images. TIFF describes image data that typically comes from scanners, frame grabbers, and paint- and photo-retouching programs. TIFF is not a printer language or page description language. The purpose of TIFF is to describe and store raster image data.” (G. Parsons, J. Rafferty 2002, date of retrieval 25.3.2011)

- GIF

“The Graphics Interchange Format (GIF) is a bitmap image format that was introduced by CompuServe in 1987 and has since come into widespread usage in the internet due to its wide support and portability. GIF images are compressed using the Lempel-Ziv-Welch (LZW) lossless data compression technique to reduce the file size without degrading the visual quality”. (Steve Olsen 2003, date of retrieval 25.3.2011)

- PNG

“PNG (Portable Network Graphics), an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF and can also replace many common uses of TIFF. PNG is designed to work well in online viewing applications”. (T. Boutell 1997, date of retrieval 25.3.2011)

4. IMPLEMENTATION

4.1 Architecture

The application's architecture is displayed on Figure 6.

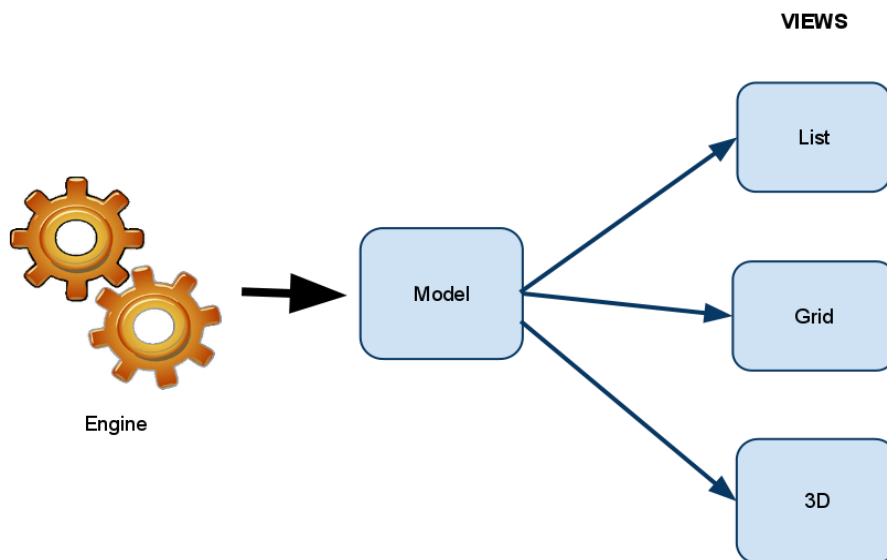


FIGURE 6. The application's architecture.

The entire system consists of three major parts: engine, model, and multiple views.

The Engine is taking care of searching pictures and maintaining the model.

The Model represents a list of absolute file paths to found pictures. The Model is responsible for storing the data and updating the views.

The views are separate QML documents used for rendering the contents of the Model. The Model data is represented in three forms:

- List – pictures are allocated in one row.
- Grid – pictures are placed in six columns.
- 3D – pictures are displayed in a neat carousel-like layout.

Special QML components, called delegates, belong to the View group. They serve as view prototypes. All user interactions among the views are handled in the designated element – UI Controller, which also belongs to the View group.

4.2 Engine

The engine is the most important part of the system. It is the main area where different logical operations, including communication with the file system, locating image files, handling search filters, and updating the model list, take place. Figure 7 shows what kinds of use cases are relevant to the Engine: `

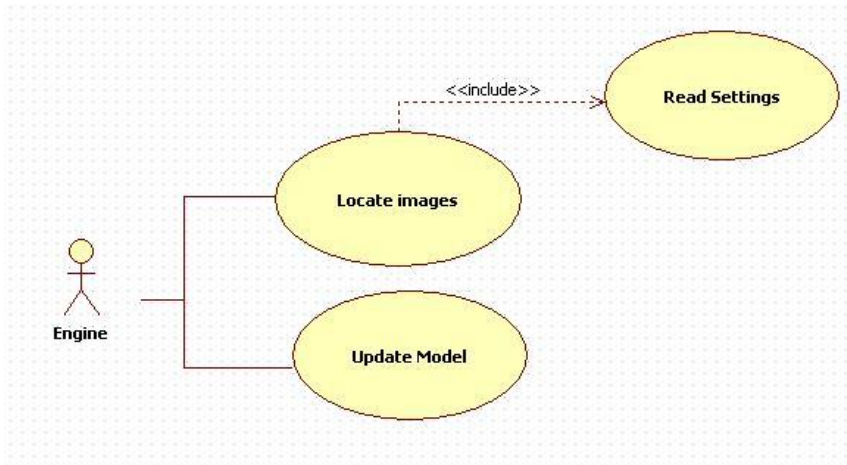


FIGURE 7. Engine's use cases.

Searching images through the file system can be a time consuming task because the file system's structure can be too wide, containing hundreds of directories and subdirectories. The search mechanism will have to explicitly check the contents of each directory and its subdirectories, thus causing an enduring freeze of the user interface. In order to prevent that the worker thread is introduced, it keeps the user interface responsive, while at the same time executing the search algorithm in the background. This functionality is implemented by inheriting the Engine class from QThread:

```
#include <QThread>
class GalleryEngine : public QThread
```

“A QThread represents a separate thread of control within the program; it shares data with all the other threads within the process but executes independently in the way that a separate program does on a multitasking operating system. Instead of starting in main(), QThreads begin executing in run()”. (Nokia 2010, date of retrieval 3.4.2011)

All possible time consuming operations in the Engine, such as searching for image files and storing them in the model, are implemented inside the run method (see Appendix 1).

The worker thread is triggered using the start method. Once the thread has completed its operation, it emits the finished signal. The connecting to that signal asynchronously allows the updating of the user interface:

```
iEngine->start();
connect (iEngine, SIGNAL(finished()), ctxt, SIGNAL(modelChanged()));
```

The Engine's operations are displayed in the diagram on Figure 8.

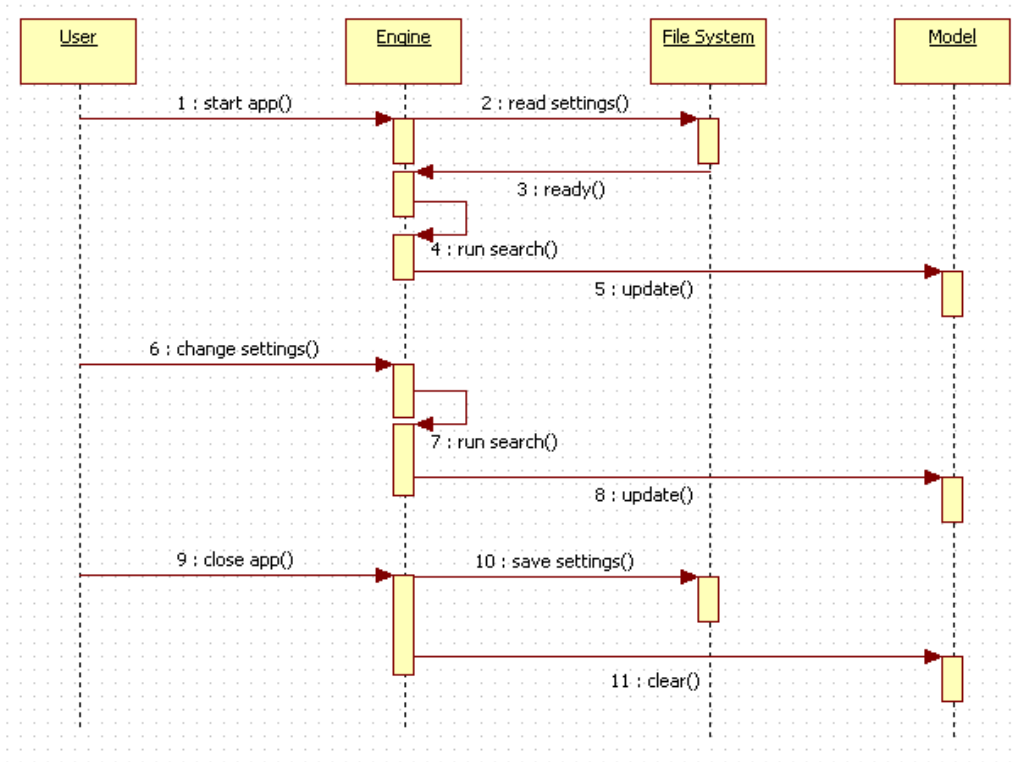


FIGURE 8. Sequence diagram shows the operations that happen in the Engine.

Once the user has started the application, the Engine reads the settings from the special data file “settings.dat”, the place where properties such as the working directory and chosen file filters are stored when the application is not running. Methods `readProperties` and `saveProperties` are used for that purpose (see Appendix 2).

When the properties are ready and the worker thread has been started, then the search mechanism is executed:

```

//find files specified by filters
files = iDir.entryList(iFilters, QDir::Files | QDir::NoSymLinks);
  
```

When the search has finished its execution, the model is updated:

```

for(int i=0; i<files.length(); i++)
{
    iFiles.append(iDir.absoluteFilePath(files[i]));
}
  
```

Every time the user changes the settings, the same operational sequence is repeated. When the application is closed, the properties are saved and the model is cleared of items.

Additionally, the Engine provides an API for retrieving and updating the working directory, applying filters, and navigating through items in the model (see Appendix 3).

4.3 Model and delegates

The model is a place where the application stores the file paths of the found images during the runtime. It is presented in the form of a list of text strings. Qt's standard type `QStringList`, which is a dynamic array of a `QString` type (`QList<QString>`), was used to implement the model:

```
#include <QStringList>
...
QStringList iFiles;
```

Each data item in the model is represented as a string value:

```
"E:/Images/someimage.jpg"
```

Binding the Qt C++ model to QML views is done using the context property:

```
QDeclarativeView view;
QDeclarativeContext *ctxt = view.rootContext();
ctxt->setContextProperty("model", QVariant::fromValue(iEngine->model()));
```

Engine's model method is used to access the data.

Every QML based view in the application has its own delegate component – a prototype item that defines how the model data is rendered inside the view.

All delegates in the application are described as a separate components and split into different QML modules. In order to achieve that, all delegates contain a root item of a similar type:

```
Component {
    ...
}
```

The idea behind this separation is that each delegate, if needed, can be easily shared among multiple views. For example, an application can have two list views that render model data in different ways: vertically and horizontally. In this case, a single list delegate is reused.

There are three delegates in the application, each corresponding to its own view:

- `GridDelegate` (see Appendix 4 for a detailed description and code)
- `ThreeDDelegate` (see Appendix 5 for a detailed description and code)
- `ListDelegate` (see Appendix 6 for a detailed description and code)

4.4 UI Controller

UI Controller is a special QML document used to handle the user interactions among the different views of the application. Figure 9 shows what kinds of use cases are relevant to the UI Controller:

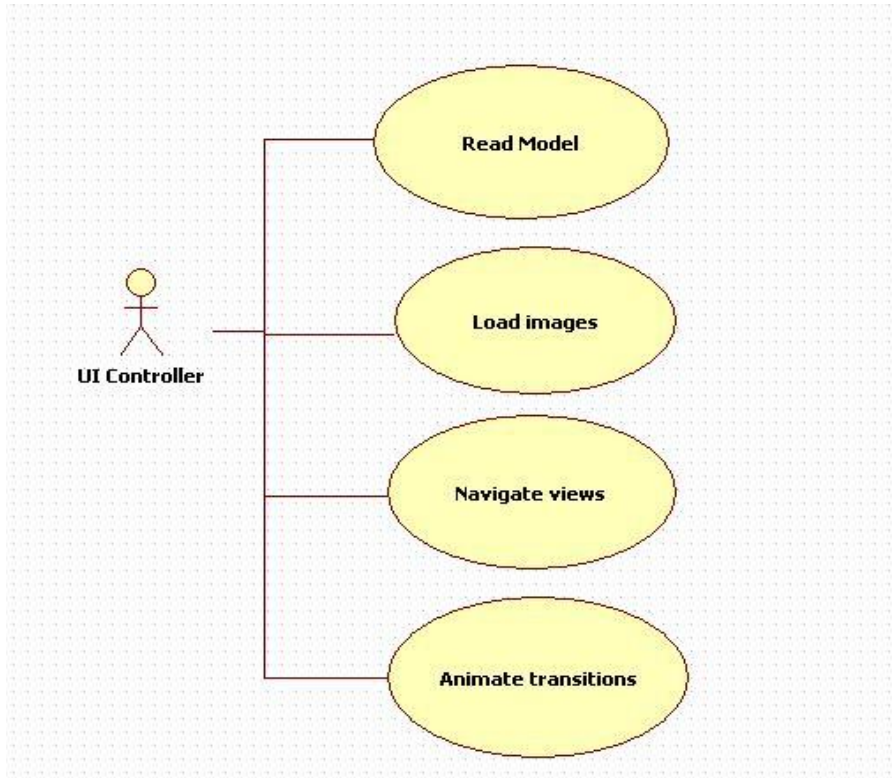


FIGURE 9. UI Controller's use cases.

The UI Controller is also the main entry point for a QML based user interface. An instance of the `QmlApplicationViewer` class, which is a QML runtime environment, sets the main QML file for the application as follows:

```
QmlApplicationViewer viewer;  
viewer.setOrientation(QmlApplicationViewer::ScreenOrientationLockLandscape  
);  
viewer.setMainQmlFile(QLatin1String("qml/photoviewer/UIController.qml"));
```

Hence the runtime environment locks the screen orientation to a landscape mode.

In order to be displayed in a view, independent user interface components must be initialized by the UI Controller e.g. as follows:

```
Components.TitleBar { id: titleBar; width: parent.width; height: 40; opacity: 0.9 }
```

Important property values, related to the component, are explicitly initialized, too. The original property values could be changed during the runtime depending on user actions.

The UI Controller is responsible for handling multiple different operations as a response to user actions (Figure 10).

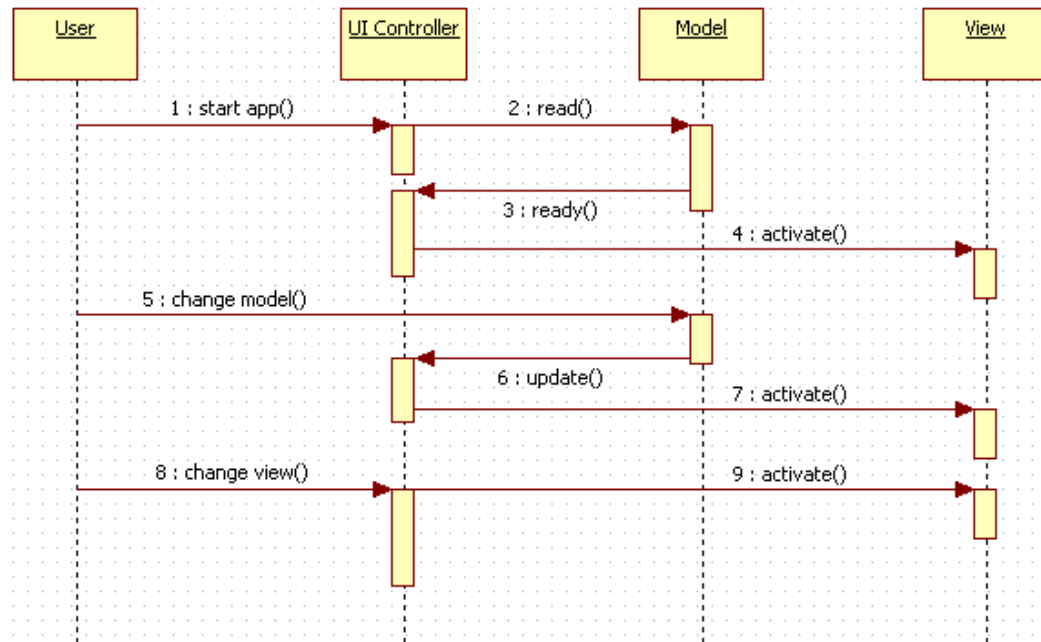


FIGURE 10. Sequence diagram shows the operations that happen in the UI Controller.

When the UI Controller initializes the views, they are linked to the model data as well as the delegate prototype through special properties:

```

GridView {
  model: model
  delegate: gridDelegate
  ...
}
  
```

Navigating through the views is done by handling the clicked signals of the toolbar button:

```

Components.Toolbar {
  ...
  onMiddleButtonClicked: if (controller.inListView == true) screen.inListView =
false;
  else controller.inListView = true
}
  
```

Transitions between the views as well as the events, such as loading the image, are emphasized using multiple types of animations e.g. NumberAnimation:

```

NumberAnimation { properties: "x"; duration: 500; easing.type:
Easing.InOutQuad }
  
```

The UI Controller is described in more detail in Appendix 7.

4.5 User Interface

The Application's user interface consists of four major views:

- Grid view
- 3D view
- List view
- Settings view

Most of the above views, except for the last one, were implemented using the QML language and feature dynamic transitions.

In order to preserve the development time, it was considered to borrow the custom UI style and colour scheme from the Flickr demo application. Some custom components, such as title bar, tool bar, and buttons are also borrowed from the same Flickr demo application with small modifications. Figure 11 displays an overview of the application's user interface:



FIGURE 11. User Interface overview.

The orientation of the application's user interface is permanently locked in the landscape mode. On the top of the screen, there is a title bar, which contains the application's title and exit button. Pressing that button will close the application.

On the bottom of the screen, there is a toolbar, which contains three buttons. The left button opens the 3D view. The right one opens the Settings screen. The one in the center swaps the Grid view with the List view and vice versa. Once the view is swapped, the label in the center button will be changed, too.

At the startup, the Grid view is set as a default one.

4.5.1 Grid view

The Grid view is a default user interface view of the application, meaning that it will be displayed every time the program is initialized. The Grid view is displayed in more detail on Figure 12.

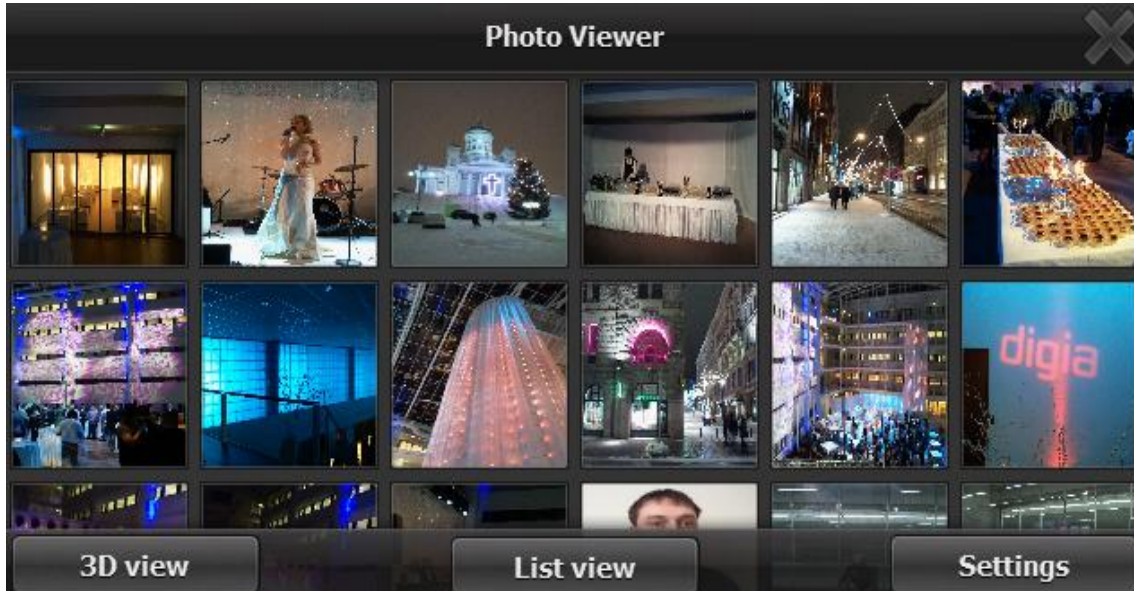


FIGURE 12. Grid view.

The Grid view renders images placed in six columns. The view is vertically scrollable.

The code snippet responsible for creating the Grid view looks as follows:

```
GridView {  
  id: gridView  
  width: parent.width; height: parent.height - 1  
  x: (width/6-99)/2; y: x  
  model: model  
  delegate: gridDelegate  
  cacheBuffer: 100  
  cellWidth: (parent.width-2)/6; cellHeight: cellWidth;  
}
```

Above properties determine:

- Special unique identifier
- Size policy of the view and its coordinates.
- Size policy of the single cell in the grid.
- Model and delegate
- Buffer size for retaining delegates outside the visible area of the view

4.5.2 3D view

The 3D view represents the screen where images are arranged in a carousel-like layout (Figure 13).

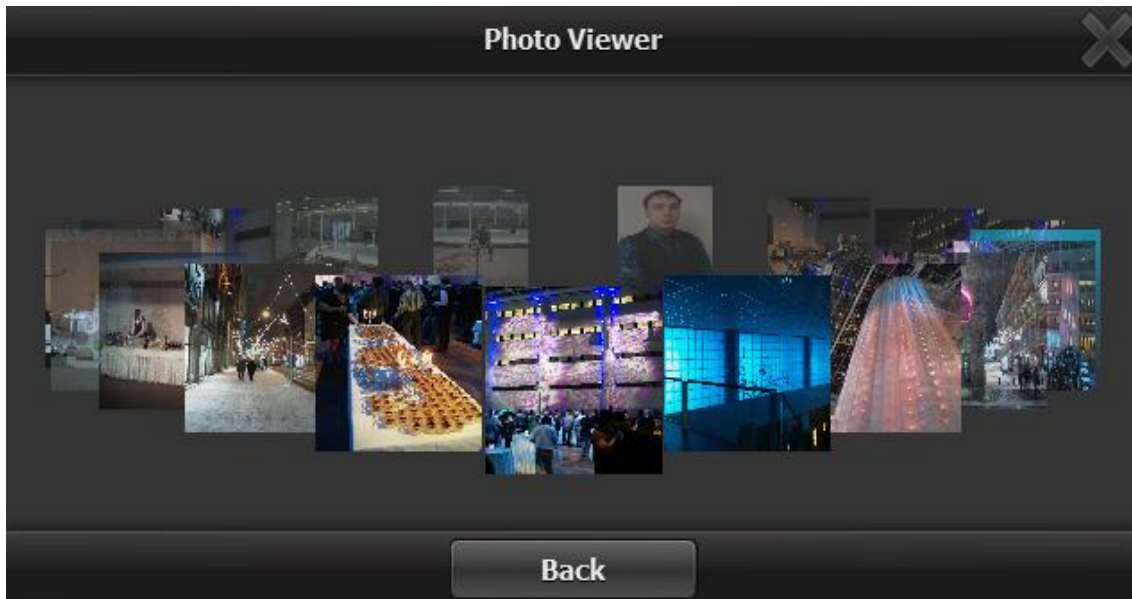


FIGURE 13. 3D view.

The view is horizontally scrollable. The further images go to background, the more their opacity and scale properties are decreased. The code snippet responsible for creating the 3D view looks as follows:

```
PathView {  
  id: threeDView; anchors.fill: parent  
  model: model; delegate: threeDDelegate  
  path: Path {  
    startX: views.width/2; startY: 2* views.height / 3;  
    PathAttribute { name: "opacity"; value: 1 }  
    PathAttribute { name: "scale"; value: 1 }  
    PathAttribute { name: "z"; value: 100 }  
    PathQuad { x: views.width/2; y: views.height / 3  
      controlX: views.width+200; controlY: views.height/2 }  
    PathAttribute { name: "opacity"; value: 0.3 }  
    PathAttribute { name: "scale"; value: 0.5 }  
    PathAttribute { name: "z"; value: 0 }  
    PathQuad { x: views.width/2; y: 2*views.height / 3  
      controlX: -200; controlY: views.height/2 }  
  }  
}
```

The 3D view is implemented utilizing QML's PathView component. The idea is that the delegates follow the predefined path. Each section of the path has its own attributes that define how the delegate is changed. The path can be a straight line or a Bezier curve.

Pressing the Back button will change the screen to the Grid view.

4.5.3 List view

The List view represents a simple layout where images are arranged in a straight line in the centre of the screen (Figure 14).

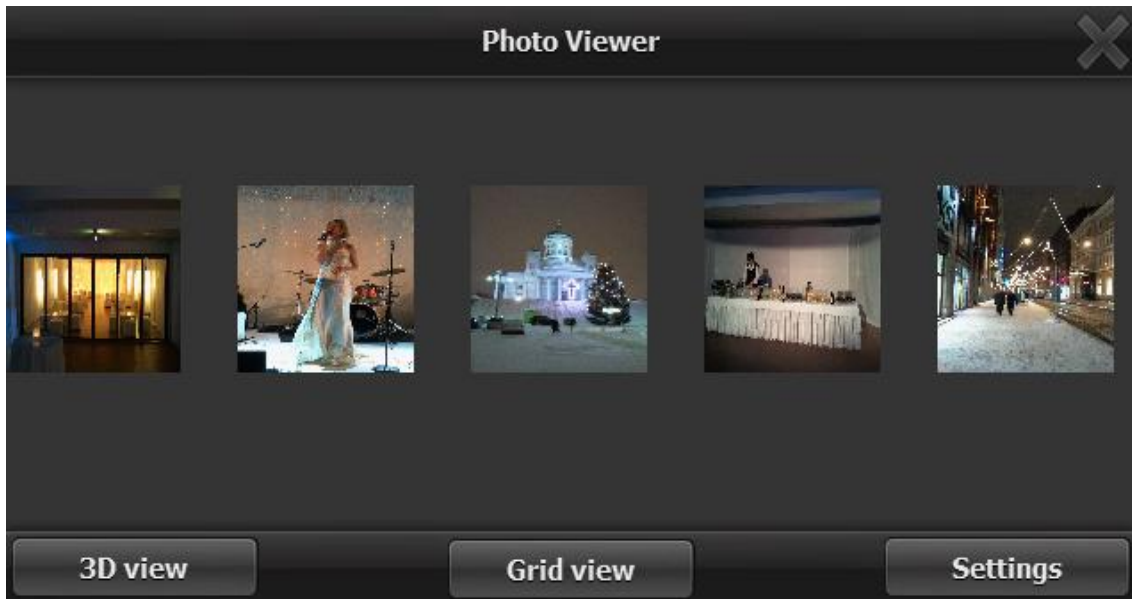


FIGURE 14. List view.

The view is horizontally scrollable. The code snippet responsible for creating the List view looks as follows:

```
ListView {  
  id: itemsList  
  model: model  
  delegate: listDelegate  
  width: parent.width  
  spacing: 32  
  height: (2*(itemsList.width / 5) - 32) * 9 / 16  
  anchors.verticalCenter: parent.verticalCenter  
  orientation: ListView.Horizontal  
  snapMode: ListView.SnapToItem  
  cacheBuffer: parent.width * 3  
}
```

The cache buffer property is used to preload images that are outside the visible area of the screen. It improves the smoothness of the scrolling behaviour at the expense of the additional memory usage.

The snap mode property defines how the view scrolling will settle. In this case, it will settle the view with an image aligned to the beginning of the view.

4.5.4 Settings screen

In order to change the search parameters the user must be able to modify the application settings. The settings screen allows the user to change the values of search filters and set a new search location (Figure 15).

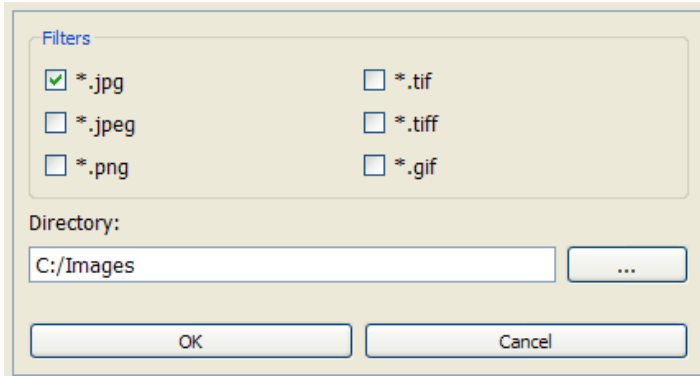


FIGURE 15. Settings screen overview.

The settings screen is the only UI element that was implemented using native Qt C++ widgets, such as QCheckBox, QLineEdit, QLabel, and QPushButton, as well as native layouts: QGroupBox, QScrollArea, QVBoxLayout, and QHBoxLayout. Unfortunately, the QML language doesn't still have such elements as a checkbox and a groupbox. Although, of course, it is possible to implement those missing QML elements, it was considered to be a subject of future development.

By pressing the browse button, for short labeled with three dots, the user triggers the native Symbian file dialogue, where the search directory can be selected:

```
QString dir = iEngine->getWorkingDir();  
QString newDir = QFileDialog::getExistingDirectory(this, tr("Directory"), dir);
```

Once the dialogue is dismissed, a new directory location is prompted on the text field and updated in the Engine.


The Settings screen can be closed with or without applying modifications. Therefore, when the user presses the "OK" button, the settings are updated calling updateFilters and updateDirectory methods. The example of updating file format filters looks as follows:

```
if (ui->jpegCBox->isChecked())  
{  
    iEngine->jpegFilter = true;  
    iEngine->addFilter(ui->jpegCBox->text());  
}
```

On the contrary, pressing the "Cancel" button will cause all the changes to be discarded.

5. TESTING

The application was tested on the following hardware for usability and performance issues:

Device name	Device picture	Specification
Nokia N8		<p>Display : 3.5 inch 16:9 nHD AMOLED Capacitive Touchscreen Display</p> <p>Resolutions : 640 x 360 pixels</p> <p>Colors : 16.7 Million Colors</p> <p>Dimensions : 113.5 x 59 x 12.9 mm</p> <p>Weight : 135 g</p> <p>Camera : 12 MP camera with Carl Zeiss optics</p> <p>Video : 16 :9 video recording in HD Video Capture in 720p 25fps with codes H.264, MPEG-4</p> <p>Video Playback : HD 720p Video playback</p> <p>Music : Music Player</p> <p>Music Player Formats : MP3, WMA, AAC, eAAC, eAAC+, AMR-NB, AMR-WB, E-AC-3, AC-3</p> <p>Memory : build-in Memory : 16 GB Expandable Memory : 32 GB with MicroSD card</p> <p>Connectivity : Bluetooth v3.0 HDMI Micro USB connector and charging High-Speed USB 2.0 3.5 mm Audio Jack</p>

6. FUTURE DEVELOPMENT

Due to the fact, that this thesis project is a proof of concept for utilizing declarative programming techniques when building dynamic user interfaces, the application lacks many functional features that are typical for the most similar applications. Although those features are outside the scope of my thesis work, it is still important to realize them. Therefore, I have researched and compiled a list of possible improvement scenarios for the application that must be considered during the future development process. Out of all, I have picked six most relevant topics and described them explicitly in the following sections.

6.1 Functionality

An additional functionality should definitely be added to the application. A full screen view needs to be implemented carrying a set of tools for customizing the image representation. Such tools are image scaling, e.g. zoom-in and zoom-out functionality, image rotation to both directions, image deletion, image brightness and transparency modification, and, possibly, slide show functionality.

Setting the current image as a home screen wallpaper could also be considered as an additional functionality.

6.2 Video playback

A video playback support could be added to the application, thus allowing not only browsing through images but also watching video files. At this moment the Engine is already capable of finding video files on the phone's memory. The only thing that needs to be modified in order to support searching for video files are additional checkboxes with file extensions on the Settings screen.

The QML language assigns a special component for creating a video playback functionality via the QtMultimediaKit module of QtMobility API. It is called a QML Video element. Modern Symbian^3 smartphones, such as the Nokia N8, support QtMobility by default, while the older phones require installing QtMobility libraries on the phone manually.

This improvement will transform the application from just a simple photo viewer into a full gallery.

6.3 Metadata

One of the possible improvement activities would be adding functionality for reading the EXIF metadata in the application.

“EXIF, the Exchangeable Image File Format, is a specification standard for storing information within digital image files. EXIF was created by the Japan Electronic Industries Development Association (JEIDA)”. (Fred Zahradnik 2010, date of retrieval 22.3.2011)

EXIF data is captured by digital cameras and stored into the EXIF file. This file itself is embedded within each digital image file. Normally this data is hidden from average users, but with the help of certain tools or methods it is possible to extract it.

There are many EXIF file tags available, and the most commonly used are: camera model, date and time, exposure, focal length.

This improvement could be interesting to a certain group of users such as professional photographers who pay attention to image details.

6.4 Location

Another great idea for a future upgrade of the application is to implement a geographical position binding with Nokia’s own OviMaps services. As it was already mentioned in the previous section, most modern digital cameras are capable of embedding the metadata information directly into a digital image file. In addition to that mobile phone cameras are able to obtain GPS data and embed it into an image file. These latitude and longitude values could be extracted and used for setting the POI – point of interest, a specific point location on a digital map that someone may find useful.

“A GPS point of interest specifies, at minimum, the latitude and longitude of the POI, assuming a certain map datum. A name or description for the POI is usually included and other information such as altitude or a telephone number may also be attached. GPS applications typically use icons to represent different categories of POI on a map graphically”. (Garmin 2011, date of retrieval 22.3.2011)

So by clicking on that POI in the OviMaps application the user could actually track where each picture was taken. This idea is not unique. It was somehow already implemented using the Google Maps services and Picasa web albums (Figure 16).

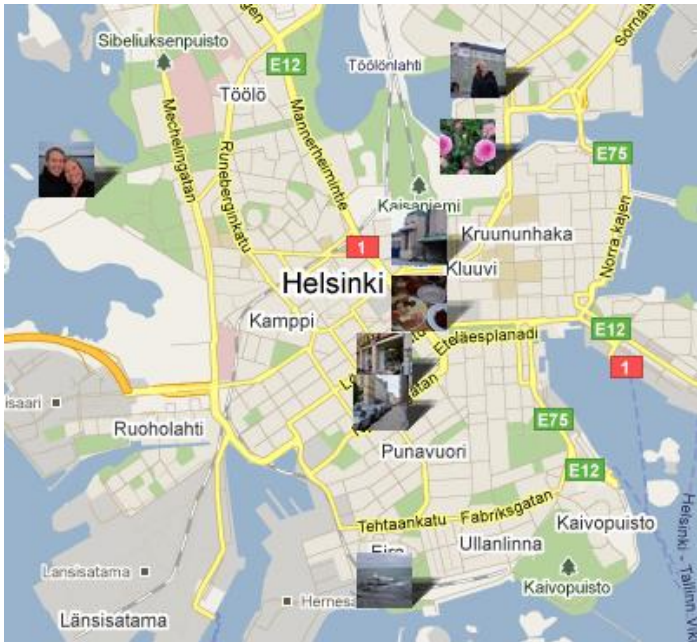


FIGURE 16. Picasa Web's album map overview.

This improvement could be especially useful for tourists and photographers who travel around the world and take pictures in different locations.

6.5 Connectivity

One of the possible future development scenarios would be implementing the functionality of a data transfer over mobile networks. It will provide users with the option of whether to send selected images via Bluetooth, an MMS message or available Wi-Fi networks.

“Bluetooth technology is the global short-range wireless standard enabling connectivity for a broad range of electronic devices”. (Bluetooth SIG 2011, date of retrieval 22.3.2011)

“Multimedia Messaging Service (MMS) is an upgraded version of the SMS (Short Messaging Service) through which you can send and receive multimedia messages such as texts, pictures, video clips, audio clips, etc., with any other compatible cell phone”. (Tech-faq 2010, date of retrieval 22.3.2011)

“In computer networking, wireless technology, or Wi-Fi, is a modern alternative to networks that use cables. A wireless network transmits data by microwave and other radio signals”. (Bradley Mitchell 2010, date of retrieval 22.3.2011)

This functionality is also typical to many similar applications.

6.6 Social sharing

In the past few years various social networks and internet services became extremely popular among people of different ages world-wide (Figure 17). Many of those networks, such as Facebook, MySpace, Twitter, LiveJournal and internet services such as Flickr, Picasa Web Album and YouTube, allow registered users to upload their own images and/or videos onto remote servers and thus, share it among communities. Other users at the same time can browse through the uploaded content and give feedback or rate those materials as favorites. REST API is the functionality behind this type of data transfer.



FIGURE 17. Popular social networks and internet services.

“REST (Representational State Transfer) is a style of software architecture for distributed hypermedia systems such as the WWW. REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages”. (Alex Rodriguez 2008; Roy Thomas Fielding 2000, date of retrieval 22.3.2011)

Upgrading the application to support the REST APIs of some popular web services would allow sharing images over social networks and therefore, attracting more potential users.

7. CONCLUSION

The aim of my thesis was to prove the concept that applying declarative programming techniques over the imperative ones, will lead to a better understanding of the advantages of the former. In the development process of my project I chose to utilize the QML declarative programming language for building dynamic user interfaces. This decision resulted to a significantly faster design and implementation and allowed me to concentrate more on auxiliary features such as animations.

In the aftermath of development activities I have succeeded in creating the Photo Viewer – a mobile software application that serves as a concept demonstrator for my thesis project.

Currently, the Photo Viewer application allows the user to browse through images located on the phone's memory, and to change the visual representation of those images via multiple different views. In addition to that, the application features nice fluid effects and animations, therefore making the user experience look entertaining.

Unfortunately, some of the functionality was sacrificed due to the fact that the scope of my project has been emphasized generally on random research tasks.

At this moment the application lacks many functional features that are typical for most of the similar applications. The few cases demanding a future development are the video playback and connectivity – the functionality responsible for the data transfer over mobile networks. Implementing both of them will drive the usability of the application to the same level among other competitors currently available on the market.

The research and development process was interesting and challenging. I have explored many articles on the subject and tested out different mobile applications with similar functionality. However, finding the solution was never a trivial task.

The project was internal competence development work during my induction period as a software developer in Digia Finland Oy company. It allowed me to familiarize myself with the modern trend in mobile software development. I have significantly improved my professional skills in Qt programming language and developed a new competence in QML language. I believe it will have a positive effect on my future career.

8. LIST OF REFERENCES

1. Alex Rodriguez 2008. RESTful Web services: The basics
<https://www.ibm.com/developerworks/webservices/library/ws-restful/>
Last access date: 22 March 2011
2. Archana Khambekar, C. Wilborn 2010. Declarative programming definition
<http://www.wisegeek.com/what-is-declarative-programming.htm>
Last access date: 24 March 2011
3. Bluetooth SIG 2011. Bluetooth definition
https://www.bluetooth.org/About/bluetooth_sig.htm
Last access date: 22 March 2011
4. Bradley Mitchell 2010. Wi-Fi
<http://compnetworking.about.com/cs/wireless/f/whatiswireless.htm>
Last access date: 22 March 2011
5. Fred Zahradnik 2010. EXIF standard
<http://gps.about.com/od/glossary/g/exif.htm>
Last access date: 22 March 2011
6. Gamma, Helm, Johnson, and Vlissides 1994. Design Patterns - Elements of Reusable Object-Oriented Software, ISBN 0-201-63361-2
Last access date: 24 March 2011
7. Garmin 2011. POI specification
<http://www8.garmin.com/products/poiloader/>
Last access date: 22 March 2011
8. G. Parsons, J. Rafferty 2002. TIFF specification
<http://tools.ietf.org/html/rfc3302>
Last access date: 25 March 2011
9. JPEG 2007. JPEG format specification
<http://www.jpeg.org/jpeg/index.html>
Last access date: 25 March 2011
10. Jörg W Mittag 2008. Characteristics of a declarative program.
<http://stackoverflow.com/questions/129628/what-is-declarative-programming>
Last access date: 24 March 2011
11. Nokia Mobile Phones 2000. EPOC specification
[http://www.nokia.com/NOKIA_COM_1/About Nokia/Press/White Papers/pdf files/dec002_net.pdf](http://www.nokia.com/NOKIA_COM_1/About%20Nokia/Press/White%20Papers/pdf_files/dec002_net.pdf)
Last access date: 24 March 2011

12. Nokia 2010. Model/View architecture
<http://doc.qt.nokia.com/latest/model-view-programming.html>
Last access date: 24 March 2011
13. Nokia 2010. QML language
<http://doc.qt.nokia.com/4.7/qdeclarativeintroduction.html>
Last access date: 21 March 2011
14. Nokia 2010. QML support on mobile phones
<http://labs.qt.nokia.com/2010/12/12/start-with-qt-4-7-for-symbian-today/>
Last access date: 16 December 2010
15. Nokia 2010. Qt Technology
<http://qt.nokia.com/about>
Last access date: 23 March 2011
16. Nokia 2010. Qt Declarative module
<http://doc.qt.nokia.com/4.7/qtquick.html#qml-elements-and-the-qt-declarative-module>
Last access date: 21 March 2011
17. Nokia 2010. QThread class reference
<http://doc.qt.nokia.com/4.7/qthread.html#details>
Last access date: 3 April 2011
18. Nokia 2010. Symbian platform definition
<http://www.forum.nokia.com/Devices/Symbian/>
Last access date: 24 March 2011
19. Roy Thomas Fielding 2000. Architectural Styles and the Design of Network-based Software Architectures
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
Last access date: 22 March 2011
20. Ryan Paul 2010. Nokia releases Qt 4.7 with new mobile UI framework
<http://arstechnica.com/open-source/news/2010/09/nokias-cross-platform-development-strategy-evolves-with-qt-47.ars>
Last access date: 25 March 2011
21. Steve Olsen 2003. GIF specification
<http://www.olsenhome.com/gif/>
Last access date: 25 March 2011
22. T. Boutell 1997. PNG specification
<http://tools.ietf.org/html/rfc2083>
Last access date: 25 March 2011
23. Tech-faq 2010. MMS definition
<http://www.tech-faq.com/mms.html>
Last access date: 22 March 2011

9. APPENDICES

APPENDIX 1	THE SOURCE CODE FOR ENGINE'S RUN METHOD
APPENDIX 2	THE SOURCE CODE FOR SETTINGS INPUT/OUTPUT
APPENDIX 3	THE SOURCE CODE FOR ENGINE'S API
APPENDIX 4	THE SOURCE CODE FOR GRID DELEGATE
APPENDIX 5	THE SOURCE CODE FOR 3D DELEGATE
APPENDIX 6	THE SOURCE CODE FOR LIST DELEGATE
APPENDIX 7	THE SOURCE CODE FOR UI CONTROLLER

APPENDIX 1

THE SOURCE CODE FOR ENGINE'S RUN METHOD

```
void GalleryEngine::run()
{
    //clear list
    iFiles->clear();

    //If no directory - skip looking for files
    if(!iDir.exists())
        return;

    //find media files specified by filters
    QStringList files;

    //null filter
    if(iFilters.isEmpty())
        iFilters << "";

    //find files specified by filters
    files = iDir.entryList(iFilters, QDir::Files | QDir::NoSymLinks);

    //store found files in the list
    for(int i=0; i<files.length(); i++)
    {
        iFiles->append(iDir.absoluteFilePath(files[i]));
    }

    //set the current file index
    if(files.length() == 0)
    {
        iIndex = -1;
        qDebug() << "No files found!";
    }
    else
    {
        iIndex = 0;
        qDebug() << "Files found!";
    }
}
```

APPENDIX 2

THE SOURCE CODE FOR SETTINGS INPUT/OUTPUT

```
/* Read properties from data file. */
void GalleryEngine::readProperties()
{
    QFile file("settings.dat");
    QDataStream in;
    file.open(QIODevice::ReadOnly);
    in.setDevice(&file);
    if(file.exists()) {
        QString dir;
        //input and set the working directory
        in >> dir;
        setWorkingDir(dir);
        //input filter settings
        in >> jpgFilter >> jpegFilter >> pngFilter >> tiffFilter >> gifFilter >> tiffFilter;
        //input filters
        in >> iFilters;
    }
    else {
        //set defaults
        setWorkingDir("/");
        jpgFilter = false;
        jpegFilter = false;
        pngFilter = false;
        tiffFilter = false;
        gifFilter = false;
        tiffFilter = false;
    }
    file.close();
}

/* Save properties to data file. */
void GalleryEngine::saveProperties()
{
    QFile file("settings.dat");
    QDataStream out;
    file.open(QIODevice::ReadWrite);
    out.setDevice(&file);
    //output working directory
    out << getWorkingDir();
    //output filter settings
    out << jpgFilter << jpegFilter << pngFilter << tiffFilter << gifFilter << tiffFilter;
    //output filters
    out << iFilters;
    file.close();
}
```

APPENDIX 3

THE SOURCE CODE FOR ENGINE'S API

```
/* Get index of the current file. */
int GalleryEngine::getCurrentIndex()
{
    return iIndex;
}
/* Get working directory path. */
QString GalleryEngine::getWorkingDir()
{
    return iDir.path();
}

/* Set new working directory. */
void GalleryEngine::setWorkingDir(QString aDir)
{
    iDir = QDir(aDir);
}

/* Reset file extension search filters. */
void GalleryEngine::resetFilters()
{
    iFilters.clear();

    jpgFilter = false;
    jpegFilter = false;
    pngFilter = false;
    tiffFilter = false;
    gifFilter = false;
    tiffFilter = false;
}

/* Add file search filter. */
void GalleryEngine::addFilter(QString aText)
{
    iFilters.append(aText);
}
```

APPENDIX 4

THE SOURCE CODE FOR GRID DELEGATE

```
Component {
  id: gridDelegate
  Item {
    id: wrapper; width: 99; height: 99

    Item {
      anchors.centerIn: parent
      scale: 0.0
      Behavior on scale { NumberAnimation { easing.type: Easing.InOutQuad} }
      id: scaleMe

      Rectangle { id: frame; height: 99; width: 99;
        anchors.centerIn: parent; color: "black"; smooth: true }
      Rectangle { id: whiteRect; width: 97; height: 97;
        anchors.centerIn: parent; color: "#dddddd"; smooth: true
      Image { id: thumb; source: image; anchors.fill: parent; smooth: true}
      }
      states: [
        State {
          name: "Show"; when: thumb.status == Image.Ready
          PropertyChanges { target: scaleMe; scale: 1 }
        }
      ]
      transitions: [
        Transition {
          from: "Show"; to: "*"
          ParentAnimation {
            NumberAnimation { properties: "x,y"; duration: 500;
              easing.type: Easing.InOutQuad } }
        },
        Transition {
          from: "*"; to: "Show"
          SequentialAnimation {
            ParentAnimation {
              NumberAnimation { properties: "x,y"; duration: 500;
                easing.type: Easing.InOutQuad }
            }
            PropertyAction { targets: wrapper; properties: "z" }
          }
        }
      ]
    }
  }
}
```

APPENDIX 5

THE SOURCE CODE FOR 3D DELEGATE

```
Component {
  id: threed_delegate
  Item { id: wrapper; width: 99; height: 99
    opacity: PathView.opacity
    scale: PathView.scale
    z: PathView.z

    Image {
      id: photo
      source: image
      width: 99
      height: 99

    }
    states: [
      State {
        name: "small"
      },
      State {
        name: "big"
        PropertyChanges {
          target: wrapper
          height: 130; z: 200; scale: 2
          x: (views.width - wrapper.width) / 2
          y: (views.height - wrapper.height) / 2
        }
        PropertyChanges {
          target: photo
          fillMode: Image.PreserveAspectRatio
        }
      }
    ]
    state: "small"
    transitions: [
      Transition {
        from: "*" to: "*"
        PropertyAnimation {
          target: wrapper; duration: 700
          properties: "scale, x, y"
          easing.type: "OutElastic"
        }
      }
    ]
  }
}
```

APPENDIX 6

THE SOURCE CODE FOR LIST DELEGATE

```
Component {
  id: listDelegate
  Item { id: wrapper; width: 99; height: 99
  Item {
    id: thumbcontainer
    width: parent.width
    height: parent.height
    opacity: 0.0
    scale: 0.0
    Behavior on opacity { NumberAnimation { easing.type:
Easing.InOutQuad;} }
    Behavior on scale { NumberAnimation { easing.type:
Easing.InOutQuad;} }

    Rectangle {
      id: thumbbg
      color: "white"
      width: parent.width
      height: parent.height
      anchors.centerIn: parent

      Image {
        id: thumb;
        anchors.centerIn: parent
        //smooth: true
        //sourceSize.width: GridView.view.cellWidth
        //sourceSize.height: GridView.view.cellHeight
        source: image
        width: parent.width
        height: parent.height
      }
    }
  }
  states: [
    State {
      name: "Show"; when: thumb.status == Image.Ready ||
thumb.status == Image.Error
      PropertyChanges { target: thumbcontainer; opacity: 1.0 }
      PropertyChanges { target: thumbcontainer; scale: 1.0 }
    }
  ]
}
}
```

APPENDIX 7

THE SOURCE CODE FOR UI CONTROLLER

```
Item { id: controller; width: 640; height: 320
  Rectangle { id: background; color: "#343434"; anchors.fill: parent
    Item { id: views; x: 2; width: parent.width - 4
      anchors.top: titleBar.bottom; anchors.bottom: toolBar.top
      Components.GridDelegate { id: gridDelegate }
      GridView { id: gridView
        model: model; delegate: gridDelegate; cacheBuffer: 100
        cellWidth: (parent.width-2)/6; cellHeight: cellWidth;
        width: parent.width; height: parent.height - 1
        x: (width/6-99)/2; y: x
      }
      Components.ThreeDDelegate { id: threeDDelegate }
      PathView { id: threeDView; anchors.fill: parent
        model: model; delegate: threeDDelegate
        path: Path {
          startX: views.width/2; startY: 2* views.height / 3;
          PathAttribute { name: "opacity"; value: 1 }
          PathAttribute { name: "scale"; value: 1 }
          PathAttribute { name: "z"; value: 100 }
          PathQuad { x: views.width/2; y: views.height / 3
            controlX: views.width+200; controlY: views.height/2}
          PathAttribute { name: "opacity"; value: 0.3 }
          PathAttribute { name: "scale"; value: 0.5 }
          PathAttribute { name: "z"; value: 0 }
          PathQuad { x: views.width/2; y: 2*views.height / 3
            controlX: -200; controlY: views.height/2}
        }
      }
      Components.ListDelegate { id: listDelegate }
      ListView { id: itemsList; width: parent.width; spacing: 32
        height: (2*(itemsList.width / 5) - 32) * 9 / 16
        anchors.verticalCenter: parent.verticalCenter
        orientation: ListView.Horizontal
        model: model; delegate: listDelegate
        snapMode: ListView.SnapToItem
        cacheBuffer: parent.width * 3
      }
    }
  }
  Components.TitleBar { id: titleBar; width: parent.width; height: 40; }
  Components.ToolBar {
    id: toolBar; height: 40; width: parent.width; opacity: 0.9
    anchors.bottom: parent.bottom;
    leftButtonLabel: "3D view"
    middleButtonLabel: "Grid view"
    rightButtonLabel: "Settings"
  }
}
```