BFH Logo

LOGO

QDE — A visual animation system.

MTE7103

Master-Thesis

Major:	Computer science
Author:	Sven Osterwalder[1]
Advisor:	Prof. Claude Fuhrer[2]
Expert:	Dr. Eric Dubuis[3]
Date:	29.03.2017
Version:	0.1

[1]sven.osterwalder@students.bfh.ch
[2]claude.fuhrer@bfh.ch
[3]eric.dubuis@comet.ch

Versions

Revision	Date	Author(s)	Description
0.1	29.03.2017	SO	Initial creation of the documentation

Todo list

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

1 Introduction

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.1 Purpose and situation

1.1.1 Motivation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.1.2 Objectives and limitations

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.1.3 Preliminary activities

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.2 Related works

Preliminary to this thesis two project works were done: “Volume ray casting — basics & principles” [1], which describes the basics and principles of sphere tracing, a special form of ray tracing, and “QDE — a visual animation system, architecture” [2], which established the ideas and notions of an editor and a player component as well as the basis for a possible software architecture for these components. The latter project work is presented in detail in the chapter about the procedure, the former project work is presented in the chapter about the implementation.

1.3 Document structure

This document is divided into N chapters, the first being this introduction. The second chapter on *administrative aspects* shows the planning of the project, including the involved persons, deliverables and the phases and milestones.

The administrative aspects are followed by a chapter on the *procedure*. The purpose of that chapter is to show the procedure concerning the execution of this thesis. It introduces a concept called literate programming, which builds the foundation for this thesis. Furthermore it establishes a framework for the actual implementation, which is heavily based on the previous project work, “QDE — a visual animation system, architecture” [2] and also includes standards and principles.

The following chapter on the *implementation* shows how the implementation of the editor and the player component as well as how the rendering is done using a special form of ray tracing as described in “Volume ray casting — basics & principles” [1]. As the editor component defines the whole data structure it builds the basis of the thesis and can be seen as main part of the thesis. The player component re-uses concepts established within the editor.

Given that literate programming is very complete and elaborated, as components being developed using this procedure are completely derived from the documentation, the actual implementation is found in the appendix as otherwise this thesis would be simply too extensive.

The last chapter is *discussion and conclusion* and discusses the procedure as well as the implementation. Some further work on the editor and the player components is proposed as well.

After the regular content follows the *appendix*, containing the requirements for building the before mentioned components, the actual source code in form of literal programming as well as test cases for the components.

2 Administrative aspects

Some administrative aspects of this thesis are covered, while they are not required for the understanding of the result.

The whole documentation uses the male form, whereby both genera are equally meant.

2.1 Involved persons

Author	Sven Osterwalder[1]	
Advisor	Prof. Claude Fuhrer[2]	*Supervises the student doing the thesis*
Expert	Dr. Eric Dubuis[3]	*Provides expertise concerning the thesis's subject, monitors and grades the thesis*

Table 2.1: List of the involved persons.

2.2 Deliverables

- **Report**
  Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.
- **Implementation**
  Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.3 Organization of work

2.3.1 Meetings

Various meetings with the supervising professor, Mr. Claude Fuhrer, helped reaching the defined goals and preventing erroneous directions of the thesis. The supervisor supported the author of this thesis by providing suggestions throughout the held meetings. The minutes of the meetings may be found under «Meeting minutes».

[1] sven.osterwalder@students.bfh.ch
[2] claude.fuhrer@bfh.ch
[3] eric.dubuis@comet.ch

2.3.2 Phases of the project and milestones

Phase	Description	Week / 2017
Start of the project		8
Definition of objectives and limitation		8-9
Documentation and development		8-30
Corrections		30-31
Preparation of the thesis' defense		31-32

Table 2.2: Phases of the project.

Phase	Description	End of week / 2017
Project structure is set up		8
Mandatory project goals are reached		30
Hand-in of the thesis		31
Defense of the thesis		32

Table 2.3: Milestones of the project.

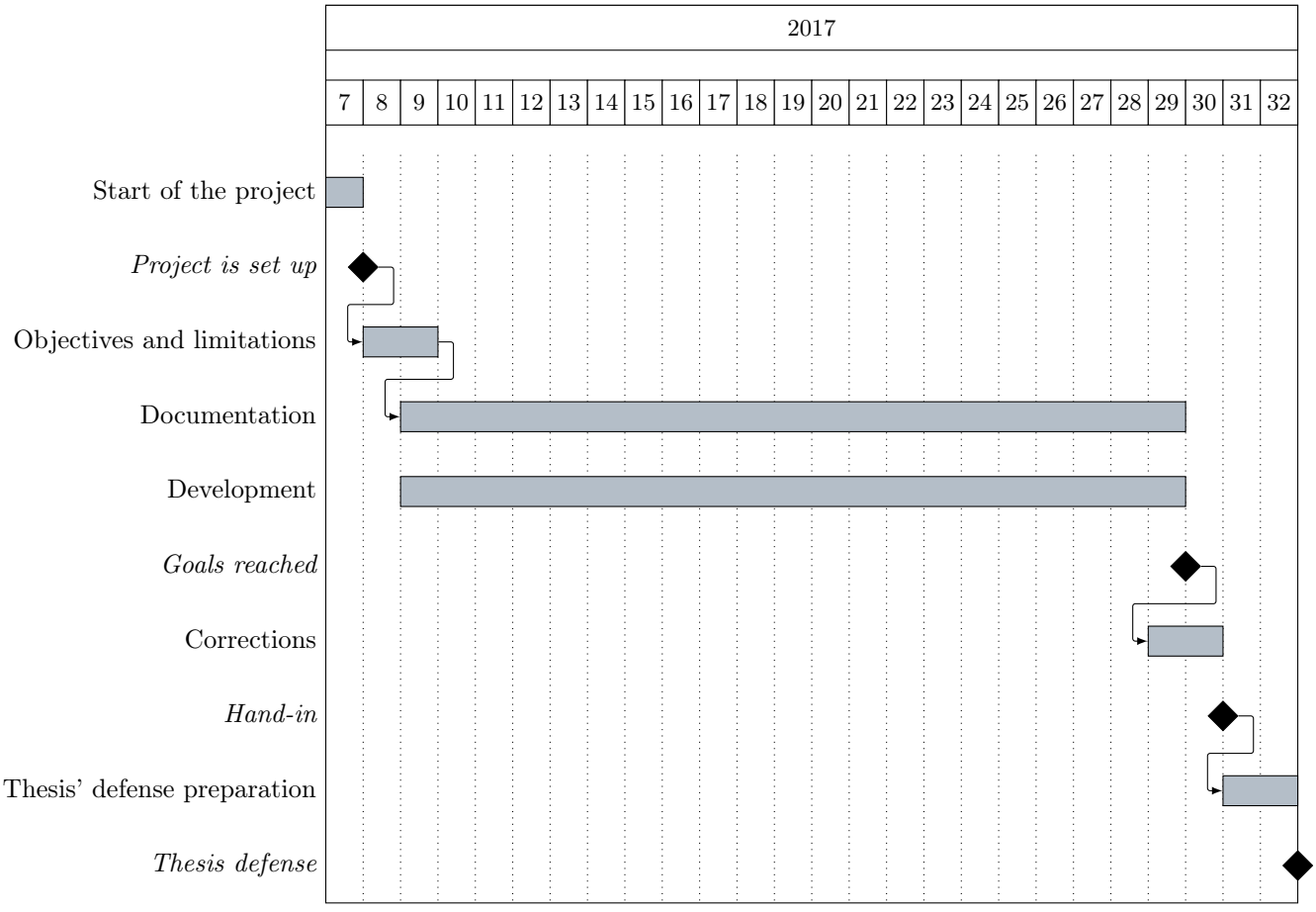

Figure 2.1: Schedule of the project by calendar weeks, including milestones.

3 Procedure

3.0.1 Literate programming

This thesis' implementation is done by a procedure named "literate programming", invented by Donald Knuth. What this means, is that the documentation as well as the code for the resulting program reside in the same file. The documentation is then /weaved/ into a separate document, which may be any by the editor support format. The code of the program is /tangled/ into a run-able computer program.

Provide more information about literate programming. Citations, explain fragments, explain referencing fragments, code structure does not have to be "normal"

Originally it was planned to develop this thesis' application test driven, providing (unit-) test-cases first and implementing the functionality afterwards. Initial trails showed quickly that this method, in company with literate programming, would exaggerate the effort needed. Therefore conventional testing is used. Test are developed after implementing functionality and run separately. A coverage as high as possible is intended. Test cases are /tangled/ too, and may be found in the appendix.

Insert reference/link to test cases here.

3.1 Standards and principles

3.1.1 Requirements

The requirements are defined by the preceding project work, "QDE — a visual animation system, software architecture" [2, p. 8 ff.], and are still valid.

For the editor application however, Python is used as a programming language. This decision is made as the author of the thesis has several years of experience concerning Python and as the performance of the editor is not a critical factor. By performance all aspects are concerned, e.g. the evaluation of the node graph or rendering itself.

As Python provides no direct bindings to Qt, an additional library is needed, which provides those bindings. Currently there exist two Python bindings for Qt: PySide and PyQt. As Qt version 5 is used, the bindings need to provide access to version 5 too. Currently this is only achieved by PyQt5 in a stable and complete way. PySide2 supports Qt version 5 too, is although under heavy development and far from being complete and stable.

Therefore PyQt5 is an additional requirement.

3.1.2 Code

- Classes use camel case.
- Folders / name-spaces use only small letters.
- Methods are all small caps and use underscores as spaces.
- Signals: do_something
- Slots: on_something

- Importing: verb(from Foo import Bar)
  As the naming of the PyQt5 modules prefixes them by /Qt/, it is very unlikely to have naming conflicts with other modules. Therefore the import format verb(from PyQt5 import [QtModuleName]) is used. This still provides a (relatively) unique naming most probably without any conflicts but reduces the effort when writing a bit. The import of system modules is therefore as follows.

Layering

Concerning the architecture, a layered architecture is foreseen, as stated in [2, p. 38 ff.]. A relaxed layered architecture leads to low coupling, reduces dependencies and enhances cohesion as well as clarity.

As the architecture's core components are all graphical, a graphical user interface for those components is developed. As the their data shall be exportable, it would be relatively tedious if the graphical user interface would hold and control that data. Instead models and model-view separation are used. Additionally controllers are introduced which act as workflow objects of the =application= layer and interfere between the model and its view.

Link to com

Model-View-Controller

While models may be instantiated anywhere directly, this would although not contribute to having clean code and sane data structures. Instead controllers, lying within the verb(application) layer, will manage instances of models. The instantiating may either be induced by the graphical user interface or by the player when loading and playing exported animations.

A view may never contain model-data (coming from the verb(domain) layer) directly, instead view models are used [3].

The behavior described above corresponds to the well-known model-view-controller pattern expanded by view models.

As Qt is used as the core for the editor, it may be quite obvious to use Qt's model/view programming practices, as described by [fn:20:http://doc.qt.io/qt-5/model-view-programming.html]. However, Qt combines the controller and the view, meaning the view acts also as a controller while still separating the storage of data. The editor application does not actually store data (in a conventional way, e.g. using a database) but solely exports it. Due to this circumstance the model-view-controller pattern is explicitly used, as also stated in [2, p. 38].

Describe the exact process of communication between ViewModel, Controller and Model.

To avoid coupling and therefore dependencies, signals and slots[fn:16:http://doc.qt.io/qt-5/signalsandslots.html] are used in terms of the observer pattern to allow inter-object and inter-layer communication.

4 Implementation

4.1 Editor

4.2 Player

4.3 Rendering

Bibliography

[1] S. Osterwalder, *Volume ray casting - basics & principles*. Bern University of Applied Sciences, Feb. 14, 2016.

[2] ——, *QDE - a visual animation system. Software-Architektur*. Bern University of Applied Sciences, Aug. 5, 2016.

[3] Martin Fowler. (Jul. 19, 2004). Presentation model, martinfowler.com, [Online]. Available: `https://martinfowler.com/eaaDev/PresentationModel.html` (visited on 03/07/2017).

List of Figures

List of Tables

Listings

5 Appendix

5.1 Implementation

To start the implementation of a project, it is necessary to first think about the goal that one wants to reach and about some basic structures and guidelines which lead to the fulfillment of that goal.

The main goal is to have a visual animation system, which allows the creation and rendering of visually appealing scenes, using a graphical user interface for creation and a ray tracing based algorithm for rendering.

The thoughts to reach this goal were already developed in chapter 3, "Procedure", and will therefore not be repeated again.

As stated in chapter 3, literate programming is used to implement the components. To maintain readability only relevant code fragments are shown in place. The whole code fragments, which are needed for tangling, are found at section 5.5.

First, the implementation of the editor component is described, as it is the basis for the whole project and also contains many concepts, that are re-used by the player component. Before starting with the implementation it is necessary to define requirements and some kind of framework for the implementation.

5.1.1 Requirements

At the current point of time, the requirements for running the components are the following:

- A Unix derivative as operating system (Linux, macOS).
- Python [1] version 3.5.x or above
- PyQt5 [2] version 5.7 or above

Add more requirements? E.g. OpenGL?

5.1.2 Name spaces and project structure

To give the whole project a structure and for being able to stick to the thoughts established in chapter 3, it may be wise to structure the project in analogous way as defined in chapter 3.

Therefore the whole source code shall be placed in the *src* directory underneath the main directory. The creation of the single directories is not explicitly shown, it is done by parts of this documentation which are tangled but not exported.

When dealing with directories and files, Python uses the term *package* for (sub-) directories and *module* for files within directories.[3]

[1] `http://www.python.org`
[2] `https://riverbankcomputing.com/software/pyqt/intro`
[3] https://docs.python.org/3/reference/import.html#packages

To prevent having multiple modules having the same name, name spaces are used.[4] The main name space shall be analogous to the project's name: *qde*. Underneath the source code folder *src*, each sub-folder represents a package and acts therefore also as a name space.

To actually allow a whole package and its modules being imported *as modules*, it needs to have at least a file inside, called *__init__.py*. Those files may be empty or they may contain regular source code such as classes or methods.

5.1.3 Coding style

To stay consistent throughout the implementation of components, a coding style is applied which is defined as follows.

- Classes use camel case, e.g. `class SomeClassName`.
- Folders respectively name-spaces use only small letters, e.g. *foo/bar/baz*.
- Methods are all small caps and use underscores as spaces, e.g. `some_method_name`.
- Signals are methods, which are prefixed by the word "do", e.g. `do_something`.
- Slots are methods, which are prefixed by the word "on", e.g. `on_something`.
- Importing is done by the `from Foo import Bar` syntax, whereas `Foo` is a module and `Bar` is either a module, a class or a method.

Importing of modules

As mentioned at subsection 5.1.1, Python is used. Python has "batteries included", which means that it offers a lot of functionality through various modules, which have to be imported first before using them. The same applies of course for self written modules.

Python offers multiple possibilities concerning imports, for details see `https://docs.python.org/3/tutorial/modules.html`.

Is direct url reference ok or does this need to be citation?

However, PEP number 8 recommends to either import modules directly or to import the needed functionality directly. [5]. As defined by the coding style, subsection 5.1.3, imports are done by the `from Foo import Bar` syntax.

The imported modules are always split up: first the system modules are imported, modules which are provided by Python itself or by external libraries, then project-related modules are imported.

Framework for implementation

For also staying consistent when implementing classes and methods, it make sense to define a rough framework for implementation, which is as follows:

- Define necessary signals.
- Within the constructor,
  - Set up the user interface when it is a class concerning the graphical user interface.
  - Set up class-specific aspects, such as the name, the tile or an icon.
  - Set up other components, used by that class.
  - Initialize the connections, meaning hooking up the defined signals with corresponding methods.

[4] https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces
[5] `https://www.python.org/dev/peps/pep-0020/`

- Implement the remaining functionality in terms of methods and slots.

Now, having defined the requirements, a project structure, a coding style and a framework for the actual implementation, the implementation of the editor may begin.

5.1.4 Editor

Before diving right into the implementation of the editor, it may be good to reconsider what shall actually be implemented, therefore what the main functionality of the editor is and what its components are.

The quintessence of the editor application is to output a structure, be it in the JSON format or even in bytecode, which defines an animation.

An animation is simply a composition of scenes which run in a sequential order within a time span. A scene is at the end of its evaluation nothing else as shader specific code which gets executed on the GPU.

To achieve this overall goal while providing the user a user-friendly experience, several components are needed. These are the following, being defined in *QDE - a visual animation system. Software-Architektur.* pp. 29 ff.

- A scene graph, allowing the creation and deletion of scenes. The scene graph has at least a root scene.
- A node-based graph structure, allowing the composition of scenes using nodes and connections between the nodes. There exists at least a root node at the root scene of the scene graph.
- A parameter window, showing parameters of the currently selected graph node.
- A rendering window, rendering the currently selected node or scene.
- A sequencer, allowing a time-based scheduling of defined scenes.

However, the above list is not quite complete. It is somehow intuitively clear, that there needs to be some main component, which holds all the mentioned components and allows a proper handling of the application. As the whole architecture uses layers and the MVC principle (see subsection 3.1.2 and section 3.1.2), the main component is composed of a view and a controller. A model is (at least at this point) not necessary. The view component shall be called *main window* and its controller shall be called *main application*.

Before implementing any of these components, the editor application needs an entry point, that is a point where the application starts when being called.

Python does this by evaluating a special variable called `__name__`. This value is set to `'__main__'` if the module is "read from standard input, a script, or from an interactive prompt." [6]

All that the entry point needs to do in case of the editor application, is spawning the editor application, execute it and exit again, as can be seen below.

⟨ *Main entry point* 14 ⟩ ≡

```
if __name__ == "__main__":
  app = application.Application(sys.argv)
  status = app.exec()
  sys.exit(status)
◇
```

Fragment referenced in 21a.

[6] `https://docs.python.org/3/library/__main__.html`

But where to place this entry point? A very direct approach would be to implement that main entry point within the main application controller. But when running the editor application by calling it from the command line, calling a controller directly may rather be confusing. Instead it is more intuitive to have only a minimal entry point which is clearly visible as such.

Although a main entry point is defined by now, the editor application cannot be started as there is no such thing as an editor application yet.

As stated in the requirements, see subsection 3.1.1, Qt version 5 is used through the PyQt5 wrapper. Therefore all functionality of Qt 5 may be used. Qt already offers a main application class, which can be used as a controller. The class is called `QApplication`.

But what does such a main application class actually do? What is its functionality? Very roughly sketched, such a type of application initializes resources, enters a main loop where it stays until told to shut down. At the end it frees resources again.

As the main application initializes resources, it act as central node between the various layers of the architecture, initializing them and connecting them using signals.[2, pp. 37 — 38]

Due to the usage of `QApplication` as super class it is not necessary to implement a main (event-) loop, as such is provided by Qt itself [7].

As stated above, the main application acts as entry point and as a central node between the various layers. Therefore it needs to do at least three things: initialize itself, set up components and connect components. This all happens when the main application is being initialized.

⟨ *Main application declarations* 15a ⟩ ≡

```
class Application(QtWidgets.QApplication):
    """Main application for QDE."""

    ⟨ Main application methods 15b ⟩
```

◇

Fragment referenced in 21b.

⟨ *Main application methods* 15b ⟩ ≡

```
    ⟨ Main application constructor 15c, ... ⟩
```

◇

Fragment referenced in 15a.

⟨ *Main application constructor* 15c ⟩ ≡

```
def __init__(self, arguments):
    """Constructor.

    :param arguments: a (variable) list of arguments, that are
                      passed when calling this class.
    :type argv:       list
    """

    ⟨ Set up internals for main application 16a ⟩
    ⟨ Set up components for main application 17d ⟩
    ⟨ Connect components for main application ? ⟩
```

◇

Fragment defined by 15c, 18c.
Fragment referenced in 15b.

[7] http://doc.qt.io/Qt-5/qapplication.html#exec

Setting up the internals is straight forward: Passing any given arguments directly to `QApplication`, setting an application icon, a name as well as a display name.

⟨ *Set up internals for main application* 16a ⟩ ≡

```
super(Application, self).__init__(arguments)
self.setWindowIcon(QtGui.QIcon("assets/icons/im.png"))
self.setApplicationName("QDE")
self.setApplicationDisplayName("QDE")
```

◇

Fragment referenced in 15c.

Having the main application as a very basic implementation, the view component of the main application, the main window, can now be implemented and then be set up by the main application.

The main functionality of the main window is to set up the actual user interface, containing all the views of the components. Qt offers the class `QMainWindow` from which `MainWindow` may inherit.

⟨ *Main window declarations* 16b ⟩ ≡

```
class MainWindow(QtWidgets.QMainWindow):
    """The main window class.
    Acts as main view for the QDE editor application.
    """

    ⟨ Main window signals 16c ⟩

    ⟨ Main window methods 17a, ... ⟩

    ⟨ Main window slots ? ⟩
```

◇

Fragment referenced in 21c.

For being able to shut down the application the main application and therefore the main window need to react to a request for shutting down, either by a keyboard shortcut or a menu command. However, `MainWindow` is not able to force `Application` to quit by itself. It would be possible to pass `MainWindow` a reference to `Application` but that would lead to tight coupling and is therefore not considered as an option. Signals and slots allow exactly such cross-layer communication without coupling components tightly.

To avoid tight coupling a signal within the main window is introduced, which tells the main application to shut down. A fitting name for the signal might be `do_close`.

⟨ *Main window signals* 16c ⟩ ≡

```
do_close = QtCore.pyqtSignal()
```

◇

Fragment referenced in 16b.

Now, that the signal for closing the window and the application is defined, two additional things need to be considered: The emission of the signal by `MainWindow` itself as well as the consumption of the signal by a slot of other classes.

The signal shall be emitted when the escape key on the keyboard is pressed or when the corresponding menu item was selected. As there is no menu at the moment, only the key pressed event is implemented by now.

⟨ *Main window methods* 17a ⟩ ≡

```
⟨ Main window constructor 17b, ... ⟩
⟨ Main window key press event 17c ⟩
◇
```

Fragment defined by 17a, 18a.
Fragment referenced in 16b.

⟨ *Main window constructor* 17b ⟩ ≡

```
def __init__(self):
    """Constructor."""

    super(MainWindow, self).__init__()
◇
```

Fragment defined by 17b, 18b.
Fragment referenced in 17a.

⟨ *Main window key press event* 17c ⟩ ≡

```
def keyPressEvent(self, event):
    """Gets triggered when a key press event is raised.

    :param event: holds the triggered event.
    :type event: QKeyEvent
    """

    if event.key() == QtCore.Qt.Key_Escape:
        self.do_close.emit()
    else:
        super(MainWindow, self).keyPressEvent(event)

◇
```

Fragment referenced in 17a.

The main window can now be set up by the main application controller, which also listens to the `do_close` signal through the inherited `quit` slot.

⟨ *Set up components for main application* 17d ⟩ ≡

```
self.main_window = qde_main_window.MainWindow()
self.main_window.do_close.connect(self.quit)
◇
```

Fragment referenced in 15c.

The used view component for the main window, `QMainWindow`, needs at least a central widget with a layout for being rendered. [8]

[8] http://doc.qt.io/qt-5/qmainwindow.html#creating-main-window-components

As the main window will set up and hold the whole layout for the application through multiple view components, a method `setup_ui` is introduced, which sets up the whole layout.

⟨ *Main window methods* 18a ⟩+ ≡

```
def setup_ui(self):
    """Sets up the user interface specific components."""

    self.setObjectName('MainWindow')
    self.setWindowTitle('QDE')
    self.resize(1024, 768)
    self.move(100, 100)
    # Ensure that the window is not hidden behind other windows
    self.activateWindow()

    central_widget = QtWidgets.QWidget(self)
    central_widget.setObjectName('central_widget')
    grid_layout = QtWidgets.QGridLayout(central_widget)
    central_widget.setLayout(grid_layout)
    self.setCentralWidget(central_widget)
    self.statusBar().showMessage('Ready.')
```

◇

Fragment defined by 17a, 18a.
Fragment referenced in 16b.

⟨ *Main window constructor* 18b ⟩+ ≡

```
self.setup_ui()
```

◇

Fragment defined by 17b, 18b.
Fragment referenced in 17a.

The main window can now be shown by the main application controller.

⟨ *Main application constructor* 18c ⟩+ ≡

```
self.main_window.show()
```

◇

Fragment defined by 15c, 18c.
Fragment referenced in 15b.

5.2 Work log

2017-02-20 Mon
Set up and structure the document initially.

2017-02-21 Tue
Re-structure the document, add first contents of the implementation. Add first tries to tangle the code. he document initially.

2017-02-22 Wed
Provide further content concerning the implementation: Introduce name-spaces/initializers, first steps for a logging facility.

2017-02-23 Thu
Extend logging facility, provide (unit-) tests. Restructure the documentation.

2017-02-24 Fri
Adapt document to output LaTeX code as desired, change styling. Begin development of the applications' main routine.

2017-02-27 Mon
Remove (unit-) tests from main document and put them into appendix instead. Begin explaining literate programming.

2017-02-28 Tue
Provide a first draft for objectives and limitations. Re-structure the document. Correct LaTeX output.

2017-03-01 Wed
Remove split files, re-add everything to index, add objectives.

2017-03-02 Thu
Set up project schedule. Tangle everything instead of doing things manually. Begin changing language to English instead of German. Re-add make targets for cleaning and building the source code.

2017-03-03 Fri
Keep work log up to date. Revise and finish chapter about name-spaces and the project structure for now.

2017-03-04 Sat
Finish translating all already written texts from German to English. Describe the main entry point of the application as well as the main application itself.

2017-03-05 Sun
Finish chapter about the main entry point and the main application for now, start describing the main window and implement its functionality. Keep the work log up to date. Fiddle with references and LaTeX export. Find a bug: main_window needs to be attached to a class, by using the *self* keyword, otherwise the window does not get shown. Introduce new make targets: one to clean Python cache files (*.pyc) and one to run the editor application directly.

2017-03-06 Mon
Update the work log. Add an image of the editor as well as the project schedule. Add the implementation of the main window's layout. Implement the scene domain model. Move keyPressEvent to its own source block instead of expanding the methods of the main window directly. Add a section about (the architecture's) layers to the principles section. Add Dr. Eric Dubuis as an expert to the involved persons. Introduce the 'verb' macro for having nicer verbatim blocks. Use the given image-width for inline images in org-mode when available.

2017-03-07 Tue
Expand the layering principles by adding a section about the model-view-controller pattern and introduce view models. Explain and implement the data- and the view model for scene graph items.

2017-03-08 Wed
Implement the controller for handling the scene graph. Allow the semi-automatic creation of an API documentation by introducing Sphinx. Introduce new make targets for creating the API documentation as RST and as HTML.

2017-03-10 Fri
Implement the scene graph view as widget and integrate it into the application. Update the work log. Fix typing errors. Start to implement missing methods in the scene graph controller for being able to use the scene graph widget.

2017-03-13 Mon
Implement the scene view model. Initialize such a model within the scene graph view model. Implement the =headerData= as well as the =data= methods of the scene graph controller. Update the work log. Add an image of the editor's current state. Continue implementation of the scene graph view model.

2017-03-14 T
Continue the implementation of the scene graph view model. Implement logging. Implement logging. Implement logging. Implement logging functionality. Log whenever a node is added or removed from the scene graph view.

2017-03-15 Wed
Move logging further down in structure. Add connections between scene graph view and controller. Finish implementing the adding and removal of scene graph items. Update the work log.

Next steps: (Re-) Introduce logging. Begin implementing the node graph.

2017-03-16 Thu
Run sphinx apidoc when creating the HTML documentation. Add an illustration about the state of the editor after finishing the implementation of the scene graph. Change width of the images to be 50the text width. Name slots of the scene graph view explicitly to maintain sanity. Re-add logging chapter with a corresponding introduction. Fix display of code listings. Keep work log up to date. Add missing TODO annotations to headings.

Next steps: Continue implementing the node graph.

2017-03-17 Fri
Change verbatim output to be less intrusive, update to do tags, begin adding references do code fragment definitions, begin implement the node graph. Move chapters into separate org files.

2017-03-20 Mon
Re-think how to implement node definitions and revise therefore the chapter about the node graph component, fix various typographic errors, expand and change the Makefile, keep the work log up to date.

2017-03-21 Tue
Re-think how to implement node definitions.

2017-03-22 Wed
Re-think how to implement node definitions and nodes. Begin adding notes about how to implement nodes.

2017-03-23 Thu
Expand notes about the node implementation, begin writing the actual node implementation down, keep the work log up to date.

2017-03-24 Fri
Attend a meeting with Prof. Fuhrer, change and expand the chapter about node implementation according to the before made thoughts, begin implementing the node graph structure, keep the work log up to date.

5.3 Requirements

5.4 Directory structure and name-spaces

This chapter describes the planned directory structure as well as how the usage of name-spaces is intended.

5.5 Code fragments

"../src/editor.py" 21a≡

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

""" Main entry point for the QDE editor application. """

# System imports
import sys

# Project imports
from qde.editor.application import application

⟨ Main entry point 14 ⟩
◇
```

"../src/qde/editor/application/application.py" 21b≡

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""Main application module for the QDE editor."""

# System imports
from PyQt5 import QtGui
from PyQt5 import QtWidgets

# Project imports
from qde.editor.gui import main_window as qde_main_window


⟨ Main application declarations 15a ⟩
◇
```

"../src/qde/editor/gui/main_window.py" 21c≡

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

""" Module holding the main application window. """

# System imports
from PyQt5 import QtCore
from PyQt5 import QtWidgets

# Project imports


⟨ Main window declarations 16b ⟩
◇
```