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.

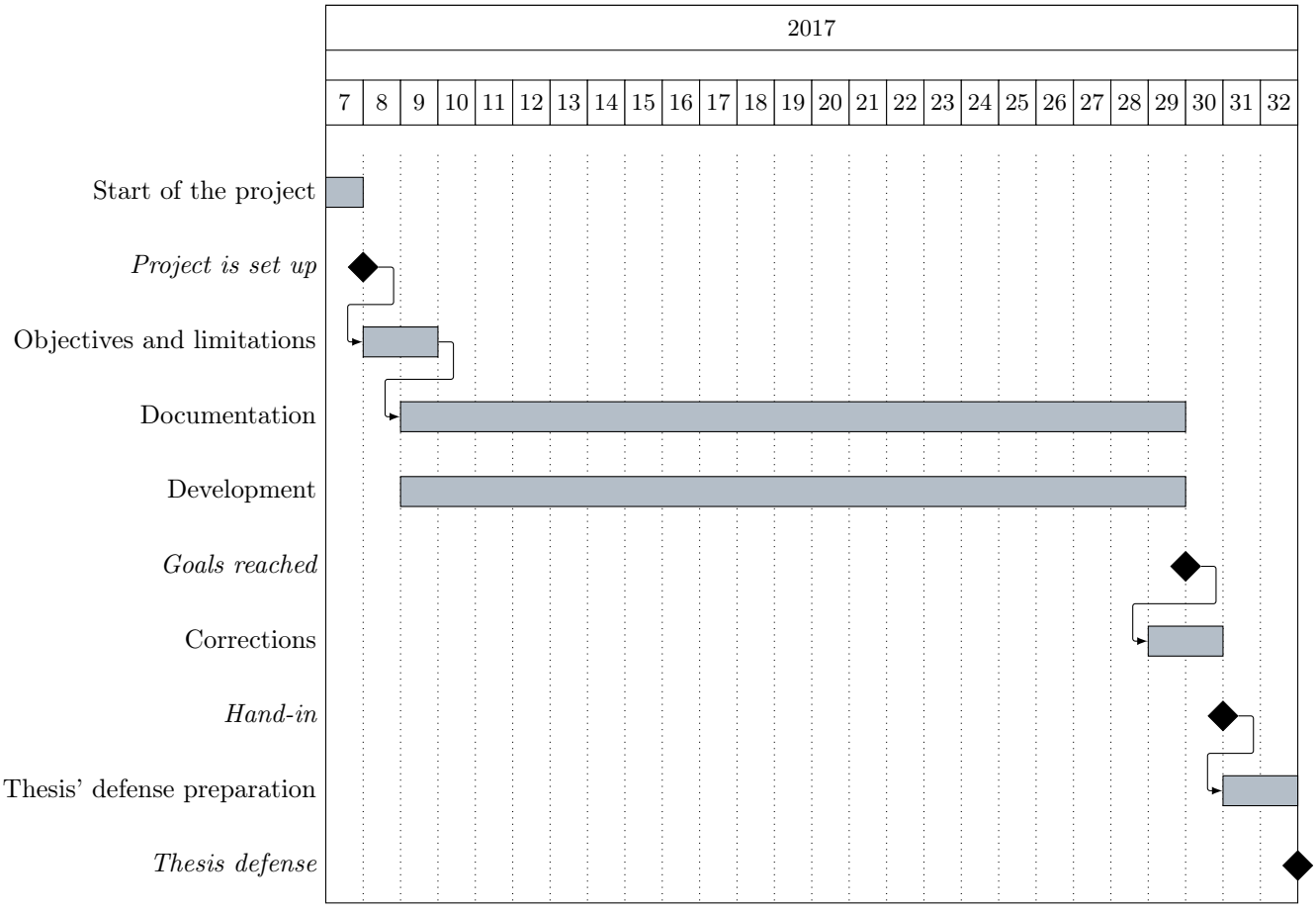

Figure 2.1: Schedule of the project by calendar weeks, including milestones.

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 5.6.

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 then a composition of nodes, which are at the end of their evaluation nothing else as shader specific code which gets executed on the GPU.

Scene: Composition of nodes. Define scene already here.

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 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 (like managing resources, shutting down properly and so on).

As the whole architecture uses layers and the MVC principle (see section 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*.

Fix references to subsection (they are displaying section atm)

Main entry point

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 within a module, called `__name__`. Its 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.

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

⟨ *Main entry point* 15a ⟩ ≡

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

◇

Fragment referenced in 44.

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. Therefore the main entry point will be put in a file called *editor.py* which is at the top level of the *src* directory.

Main application

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. Therefore a main application needs to be implemented.

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, and at the end it frees the allocated resources again.

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 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]

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* 15b ⟩ ≡

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

    ⟨ Main application methods 16a, ... ⟩
```

◇

Fragment referenced in 45a.

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

⟨ *Main application methods* 16a ⟩ ≡

```
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 16b ⟩
    ⟨ Set up components for main application 17c, ... ⟩
    ⟨ Connect components for main application 30a, ... ⟩

    self.main_window.show()
```

◇

Fragment defined by 16a, 37a.
Fragment referenced in 15b.

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* 16b ⟩ ≡

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

◇

Fragment referenced in 16a.

The other two steps, setting up the components and connecting them can however not be done at this point, as there simply are no components available. A component to start with is the view component of the main application, the main window.

Main window

Having a very basic implementation of the main application, its view component, 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* 16c ⟩ ≡

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

    ⟨ Main window signals 17a ⟩

    ⟨ Main window methods 17b, ... ⟩
```

◇

Fragment referenced in 45b.

For being able to shut down the main application and therefore the main window, they need to react to a request for shutting down, either by a keyboard shortcut or a menu command. However, the main window is not able to force the main application to quit by itself. It would be possible to pass the main window a reference to the 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* 17a ⟩ ≡

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

Fragment referenced in 16c.

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 the main window 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* 17b ⟩ ≡

```
def __init__(self, parent=None):
    """Constructor."""

    super(MainWindow, self).__init__(parent)
    self.setup_ui()

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 defined by 17b, 19.
Fragment referenced in 16c.

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* 17c ⟩ ≡

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

Fragment defined by 17c, 29a.
Fragment referenced in 16a.

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

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. The method creates a central widget containing a grid layout.

As the main window holds all other view components and a look as proposed in *QDE - a visual animation system. Software-Architektur.* p. 9 is targeted, a simple grid layout does not provide enough possibilities. Instead a horizontal box layout in combination with splitters is used.

Recalling the components, the following layout is approached:

- A scene graph, on the left of the window, covering the whole height.
- A node graph on the right of the scene graph, covering as much height as possible.
- A view for showing the properties (and therefore parameters) of the selected node on the right of the node graph, covering as much height as possible.
- A display for rendering the selected node, on the right of the properties view, covering as much height as possible
- A sequencer at the right of the scene graph and below the other components at the bottom of the window, covering as much width as possible

Provide a picture of the layout here.

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

⟨ *Main window methods* 19 ⟩+ ≡

```
    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.')

        horizontal_layout_widget = QtWidgets.QWidget(central_widget)
        horizontal_layout_widget.setObjectName('horizontal_layout_widget')
        horizontal_layout_widget.setGeometry(QtCore.QRect(12, 12, 781, 541))
        horizontal_layout_widget.setSizePolicy(QtWidgets.QSizePolicy.MinimumExpanding,
        QtWidgets.QSizePolicy.MinimumExpanding)
        grid_layout.addWidget(horizontal_layout_widget, 0, 0)

        horizontal_layout = QtWidgets.QHBoxLayout(horizontal_layout_widget)
        horizontal_layout.setObjectName('horizontal_layout')
        horizontal_layout.setContentsMargins(0, 0, 0, 0)

        self.scene_graph_widget = guiscene.SceneGraphView()
        self.scene_graph_widget.setObjectName('scene_graph')
        self.scene_graph_widget.setMaximumWidth(300)
        horizontal_layout.addWidget(self.scene_graph_widget)

    ⟨ Set up node graph view in main window ? ⟩
    ⟨ Set up parameter view in main window ? ⟩
    ⟨ Set up render view in main window ? ⟩

        horizontal_splitter = QtWidgets.QSplitter()
    ⟨ Add render view to horizontal splitter in main window ? ⟩
    ⟨ Add parameter view to horizontal splitter in main window ? ⟩

        vertical_splitter = QtWidgets.QSplitter()
        vertical_splitter.setOrientation(QtCore.Qt.Vertical)
        vertical_splitter.addWidget(horizontal_splitter)
    ⟨ Add node graph to vertical splitter in main window ? ⟩

        horizontal_layout.addWidget(vertical_splitter)
    ◇
```

Fragment defined by 17b, 19.
Fragment referenced in 16c.

All the above taken actions to lay out the main window change nothing in the window's yet plain appearance. This is quite obvious, as none of the actual components are implemented yet.

The most straight-forward component to implement may be scene graph, so this is a good starting point for the implementation of the remaining components.

Scene graph

As mentioned in subsection 5.1.4, the scene graph has also two aspects to consider: a graphical aspect as well as its data structure.

Define what a scene is by prose and code.

As described in chapter 3, two kinds of models are used. A domain model, containing the actual data and a view model, which holds a reference to its corresponding domain model.

As the domain model builds the basis for the whole (data-) structure, it is implemented first.

⟨ *Scene model declarations* 20a ⟩ ≡

```
class SceneModel(object):
    """The scene model.
    It is used as a base class for scene instances within the whole system.
    """

    ⟨ Scene model signals ? ⟩

    ⟨ Scene model methods 20b ⟩

    ⟨ Scene model slots ? ⟩
```

◇

Fragment referenced in 45c.

At this point the only known fact is, that a scene is a composition of nodes, and therefore it holds its nodes as a list.

⟨ *Scene model methods* 20b ⟩ ≡

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

    self.nodes = []
```

◇

Fragment referenced in 20a.

The counter part of the domain model is the view model. View models are used to visually represent something within the graphical user interface and they provide an interface to the `domain` layer. To this point, a simple reference in terms of an attribute is used as interface, which may be changed later on.

Concerning the user interface, a view model must fulfill the requirements posed by the user interface's corresponding component. In this case, the scene graph view model inherits from `QObject` as this base class already provides a tree structure, which fits the structure of the scene graph perfectly and therefore fulfills (part of) the requirements posed by the view.

⟨ *Scene view model declarations* 21a ⟩ ≡

```
class SceneViewModel(QtCore.QObject):
    """View model representing scene graph items.

    The SceneViewModel corresponds to an entry within the scene graph. It
    is used by the QAbstractItemModel class and must therefore at least provide
    a name and a row.
    """

    ⟨ Scene view model signals ? ⟩

    ⟨ Scene view model constructor 21b ⟩

    ⟨ Scene view model methods 28b ⟩

    ⟨ Scene view model slots ? ⟩
```

◇

Fragment referenced in 46a.

In terms of the scene graph, the view model must provide at least a name and a row. In addition, as written above, it holds a reference to the domain model.

⟨ *Scene view model constructor* 21b ⟩ ≡

```
def __init__(
        self,
        row,
        domain_object,
        name=QtCore.QCoreApplication.translate('SceneViewModel', 'New scene'),
        parent=None
):
    """Constructor.

    :param row:          The row the view model is in.
    :type  row:          int
    :param domain_object: Reference to a scene model.
    :type  domain_object: qde.editor.domain.scene.SceneModel
    :param name:         The name of the view model, which will be displayed in
                         the scene graph.
    :type  name:         str
    :param parent:       The parent of the current view model within the scene
                         graph.
    :type  parent:       qde.editor.gui_domain.scene.SceneViewModel
    """

    super(SceneViewModel, self).__init__(parent)
    self.row = row
    self.domain_object = domain_object
    self.name = name
```

◇

Fragment referenced in 21a.

Scenes may now be instantiated, it is although necessary to manage scenes in a controlled manner. Therefore the class `SceneGraphController` will now be implemented, for being able to manage scenes.

As the scene graph shall be built as a tree structure, an appropriate data structure is needed. Qt provides the `QTreeWidget` class, but that class is in this case not suitable, as it does not separate the data from

its representation, as stated by Qt: “Developers who do not need the flexibility of the Model/View framework can use this class to create simple hierarchical lists very easily. A more flexible approach involves combining a QTreeView with a standard item model. This allows the storage of data to be separated from its representation.”[9]

Such a standard item model is `QAbstractItemModel`[10], which is used as a base class for the scene graph controller.

⟨ *Scene graph controller declarations* 22a ⟩ ≡

```
class SceneGraphController(QtCore.QAbstractItemModel):
    """The scene graph controller.
    A controller for managing the scene graph by adding, editing and removing
    scenes.
    """

    ⟨ Scene graph controller signals 35c ⟩

    ⟨ Scene graph controller methods 22b, ... ⟩

    ⟨ Scene graph controller slots 34a ⟩
```

◇

Fragment referenced in 46b.

As at this point the functionality of the scene graph controller is not fully known, the constructor simply initializes its parent class.

⟨ *Scene graph controller methods* 22b ⟩ ≡

```
def __init__(self, parent=None):
    """Constructor.

    :param parent: The parent of the current view model within the scene
                   graph.
    :type parent: qde.editor.gui_domain.scene.SceneViewModel
    """

    super(SceneGraphController, self).__init__(parent)
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

As the scene graph controller holds and manages the data, it needs to have at least a root node. As the controller manages both, domain models and the view models, it needs to create both models.

⟨ *Scene graph controller methods* 22c ⟩+ ≡

```
    self.root_node = domain_scene.SceneModel()
    self.view_root_node = guidomain_scene.SceneViewModel(
        row=0,
        domain_object=self.root_node,
        name=QtCore.QCoreApplication.translate(__class__.__name__, 'Root scene')
    )
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

[9] http://doc.qt.io/qt-5/qtreewidget.html#details
[10] http://doc.qt.io/qt-5/qabstractitemmodel.html

Whenever a scene is added, the item model needs to be informed for updating the view. This happens by emitting the `rowsInserted` signal, which is already given by the `QAbstractItemModel` class. Therefore the signal must also be emitted when the root node is added.

⟨ *Scene graph controller methods* 23a ⟩ ≡

```
        self.rowsInserted.emit(QtCore.QModelIndex(), 0, 1)
    ◇
```

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

The scene graph controller must also provide the header data, which is used to display the header within the view (due to the usage of the Qt view model). As header data the name of the scenes as well as the number of nodes a scene contains shall be displayed.

Add referenc

⟨ *Scene graph controller methods* 23b ⟩ ≡

```
        self.header_data = [
            QtCore.QCoreApplication.translate(__class__.__name__, 'Name'),
            QtCore.QCoreApplication.translate(__class__.__name__, '# Nodes')
        ]
    ◇
```

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

As `QAbstractItemModel` is used as a basis for the scene graph controller, some methods must be implemented at very least: "When subclassing QAbstractItemModel, at the very least you must implement index(), parent(), rowCount(), columnCount(), and data(). These functions are used in all read-only models, and form the basis of editable models."[10]

The method `index` returns the position of an item in the (data-) model for a given row and column below a parent item.

⟨ *Scene graph controller methods* 24 ⟩ ≡

```
def index(self, row, column, parent=QtCore.QModelIndex()):
    """Return the index of the item in the model specified by the given row,
    column and parent index.

    :param row: The row for which the index shall be returned.
    :type row: int
    :param column: The column for which the index shall be returned.
    :type column: int
    :param parent: The parent index of the item in the model. An invalid model
                   index is given as the default parameter.
    :type parent: QtQore.QModelIndex

    :return: the model index based on the given row, column and the parent
            index.
    :rtype: QtCore.QModelIndex
    """

    # If the given parent (index) is not valid, create a new index based on the
    # currently set root node
    if not parent.isValid():
        return self.createIndex(row, column, self.view_root_node)

    # The internal pointer of the the parent (index) returns a scene graph view
    # model
    parent_node = parent.internalPointer()
    child_nodes = parent_node.children()

    return self.createIndex(row, column, child_nodes[row])
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

The method `parent` returns the parent item of an item identified by a provided index. If that index is invalid, an invalid index is returned as well.

⟨ *Scene graph controller methods* 25a ⟩ ≡

```
def parent(self, model_index):
    """Return the parent of the model item with the given index. If the item has
    no parent, an invalid QModelIndex is returned.

    :param model_index: The model index which the parent model index shall be
                        derived for.
    :type model_index: int

    :return: the model index of the parent model item for the given model index.
    :rtype: QtCore.QModelIndex
    """

    if not model_index.isValid():
        return QtCore.QModelIndex()

    # The internal pointer of the the model index returns a scene graph view
    # model.
    node = model_index.internalPointer()
    if node.parent() is None:
        return QtCore.QModelIndex()
    else:
        return self.createIndex(node.parent().row, 0, node.parent())
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

Implementing the `columnCount` and `rowCount` methods is straight forward. The former returns simply the number of columns, in this case the number of headers, therefore 2.

⟨ *Scene graph controller methods* 25b ⟩ ≡

```
def columnCount(self, parent):
    """Return the number of columns for the children of the given parent.

    :param parent: The index of the item in the scene graph, which the
                   column count shall be returned for.
    :type parent: QtCore.QModelIndex

    :return: the number of columns for the children of the given parent.
    :rtype: int
    """

    return len(self.header_data)
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

The method `rowCount` returns the number of children for a given parent item (identified by its index within the data model).

⟨ *Scene graph controller methods* 26 ⟩ ≡

```
def rowCount(self, parent):
    """Return the number of rows for the children of the given parent.

    :param parent: The index of the item in the scene graph, which the
                   row count shall be returned for.
    :type parent: QtCore.QModelIndex

    :return: the number of rows for the children of the given parent.
    :rtype: int
    """

    if not parent.isValid():
        return 1

    # Get the actual object stored by the parent. In this case it is a
    # SceneViewModel.
    node = parent.internalPointer()

    return len(node.children())
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

The last method, that has to be implemented due to the usage of `QAbstractItemModel`, is the `data` method. It returns the data for an item identified by the given index for the given role.

A role indicates what type of data is provided. Currently the only role considered is the display of models (further information may be found at `http://doc.qt.io/qt-5/qt.html#ItemDataRole-enum`).

Depending on the column of the model index, the method returns either the name of the scene graph node or the number of nodes a scene contains.

⟨ *Scene graph controller methods* 27 ⟩ ≡

```
    def data(self, model_index, role=QtCore.Qt.DisplayRole):
        """Return the data stored under the given role for the item referred by the
        index.

        :param model_index: The (data-) model index of the item.
        :type model_index: int
        :param role: The role which shall be used for representing the data. The
                     default (and currently only supported) is displaying the data.
        :type role: QtCore.Qt.DisplayRole

        :return: the data stored under the given role for the item referred by the
                 given index.
        :rtype: str
        """

        if not model_index.isValid():
            return None

        # The internal pointer of the model index returns a scene view model.
        node = model_index.internalPointer()

        if role == QtCore.Qt.DisplayRole:
            # Return either the name of the scene or its number of nodes.
            column = model_index.column()

            if column == 0:
                return node.name
            elif column == 1:
                return node.node_count
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

In addition to the above mentioned methods, the `QAbstractItemModel` offers the method `headerData`, which "returns the data for the given role and section in the header with the specified orientation."[11]

[11] http://doc.qt.io/qt-5/qabstractitemmodel.html#headerData

⟨ *Scene graph controller methods* 28a ⟩ ≡

```
def headerData(self, section, orientation=QtCore.Qt.Horizontal,
               role=QtCore.Qt.DisplayRole):
    """Return the data for the given role and section in the header with the
    specified orientation.

    Currently vertical is the only supported orientation. The only supported
    role is DisplayRole. As the sections correspond to the header, there are
    only two supported sections: 0 and 1. If one of those parameters is not
    within the described values, None is returned.

    :param section: the section in the header. Currently only 0 and 1 are
                    supported.
    :type section: int
    :param orientation: the orientation of the display. Currently only
                        Horizontal is supported.
    :type orientation: QtCore.Qt.Orientation
    :param role: The role which shall be used for representing the data. The
                 default (and currently only supported) is displaying the data.
    :type role: QtCore.Qt.DisplayRole

    :return: the header data for the given section using the given role and
             orientation.
    :rtype: str
    """

    if (
            orientation == QtCore.Qt.Horizontal and
            role        == QtCore.Qt.DisplayRole and
            section     in [0, 1]
    ):
        return self.header_data[section]
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

One thing, that may stand out, is, that the above defined `data` method returns the number of graph nodes within a scene by accessing the `node_count` property of the *scene view model*.

The *scene view model* does therefore need to keep track of the nodes it contains, in form of a list, analogous to the domain model.

It does not make sense however to use the list of nodes from the domain model, as the view model will hold references to graphical objects where as the domain model holds only pure data objects.

Due to the inheritance from `QObject`, it is not necessary to explicitly implement the nodes of a scene as a list, instead it is already given by the `children` method.

The method `node_count` then simply returns the length of the node list.

⟨ *Scene view model methods* 28b ⟩ ≡

```
@property
def node_count(self):
    """Return the number of nodes, that this scene contains."""

    return len(self.children())
```

◇

Fragment referenced in 21a.

The scene graph controller can now be set up by the main application controller.

⟨ *Set up components for main application* 29a ⟩+ ≡

```
self.scene_graph_controller = scene_graph.SceneGraphController(self)
```

◇

Fragment defined by 17c, 29a.
Fragment referenced in 16a.

At this point data structures in terms of a (data-) model and a view model concerning the scene graph are implemented. Further a controller for handling the flow of the data for both models is implemented. What is still missing, is the actual representation of the scene graph in terms of a view.

Qt offers a plethora of widgets for implementing views. One such widget is `QTreeView`, which "implements a tree representation of items from a model. This class is used to provide standard hierarchical lists that were previously provided by the QListView class, but using the more flexible approach provided by Qt's model/view architecture." [12] Therefore `QTreeView` is used as basis for the scene graph view.

⟨ *Scene graph view declarations* 29b ⟩ ≡

```
⟨ Scene graph view decorators 48b ⟩
class SceneGraphView(QtWidgets.QTreeView):
    """The scene graph view widget.
    A widget for displaying and managing the scene graph.
    """

⟨ Scene graph view signals 31b, ... ⟩

⟨ Scene graph view constructor 29c, ... ⟩

⟨ Scene graph view methods 31a ⟩

⟨ Scene graph view slots 30b, ... ⟩
```

◇

Fragment referenced in 46c.

As at this point the functionality of the scene graph view is not fully known, the constructor simply initializes its parent class.

⟨ *Scene graph view constructor* 29c ⟩ ≡

```
def __init__(self, parent=None):
    """Constructor.

    :paramparent:       The parent of the current view widget.
    :type parent:       QtCore.QObject
    """

    super(SceneGraphView, self).__init__(parent)
```

◇

Fragment defined by 29c, 32ab.
Fragment referenced in 29b.

[12] fn:f377826acb87691:http://doc.qt.io/qt-5/qtreeview.html#details

For being able to display anything, the scene graph view needs a controller to work with. In terms of Qt, the controller is called a model, as due its model/view architecture.

⟨ *Connect components for main application* 30a ⟩ ≡

```
self.main_window.scene_graph_widget.setModel(
    self.scene_graph_controller
)
```
◇

Fragment defined by 30a, 35b.
Fragment referenced in 16a.

But scenes shall not only be displayed, instead it shall be possible to work with them. What shall be achieved, are three things: Adding and removing scenes, renaming scenes and switching scenes.

To switch between scenes it is necessary to emit what scene was selected. This is needed to tell the other components, such as the node graph for example, that the scene has changed.

Through the `selectionChanged` signal the scene graph view already provides a possibility to detect if another scene was selected. This signal emits an item selection in terms of model indices although. As this is very view- and model-specific, it would be easier for other components if the selected scene is emitted directly in terms of a view model.

⟨ *Scene graph view slots* 30b ⟩ ≡

```
@QtCore.pyqtSlot(QtCore.QItemSelection, QtCore.QItemSelection)
def on_tree_item_selected(self, selected, deselected):
    """Slot which is called when the selection within the scene graph view is
    changed.

    The previous selection (which may be empty) is specified by the deselected
    parameter, the new selection is specified by the selected paramater.

    This method emits the selected scene graph item as scene view model.

    :param selected: The new selection of scenes.
    :type selected: QtCore.QModelIndex
    :param deselected: The previous selected scenes.
    :type deselected: QtCore.QModelIndex
    """

    selected_item = selected.first()
    selected_index = selected_item.indexes()[0]
    selected_scene_graph_view_model = selected_index.internalPointer()
    self.tree_item_selected.emit(selected_scene_graph_view_model)
```
◇

Fragment defined by 30b, 33.
Fragment referenced in 29b.

But the `on_tree_item_selected` slot needs to be triggered as soon as the selection is changed. This is done by connecting the slot with the signal.

The `selectionChanged` signal is however not directly accessible, it is only accessible through the selection model of scene graph view (which is given by the usage of `QTreeView`). The selection model can although only be accessed when setting the data model of the view, which needs therefore to be expanded.

⟨ *Scene graph view methods* 31a ⟩ ≡

```
def setModel(self, model):
    """Set the model for the view to present.

    This method is only used for being able to use the selection model's
    selectionChanged method and setting the current selection to the root node.

    :param model: The item model which the view shall present.
    :type model: QtCore.QAbstractItemModel
    """

    super(SceneGraphView, self).setModel(model)

    # Use a slot to emit the selected scene view model upon the selection of a
    # tree item
    selection_model = self.selectionModel()
    selection_model.selectionChanged.connect(
        self.on_tree_item_selected
    )

    # Set the index to the first node of the model
    self.setCurrentIndex(model.index(0, 0))
```

◇

Fragment referenced in 29b.

As stated above, `on_tree_item_selected` emits another signal containing a reference to a scene view model.

⟨ *Scene graph view signals* 31b ⟩ ≡

```
tree_item_selected = QtCore.pyqtSignal(scene.SceneViewModel)
```

◇

Fragment defined by 31bc.
Fragment referenced in 29b.

In the same manner as the selection of an item was implemented, the adding and removal of a scene are implemented. However, the tree widget does not provide direct signals for those cases as it is the case when selecting a tree item, instead own signals, slots and actions have to be used.

⟨ *Scene graph view signals* 31c ⟩+ ≡

```
do_add_item = QtCore.pyqtSignal(QtCore.QModelIndex)
do_remove_item = QtCore.pyqtSignal(QtCore.QModelIndex)
```

◇

Fragment defined by 31bc.
Fragment referenced in 29b.

An action gets triggered, typically by hovering over some item (in terms of a context menu for example) or by pressing a defined keyboard shortcut. For the adding and the removal, a keyboard shortcut will be used.

Adding of a scene item shall happen when pressing the `a` key on the keyboard.

⟨ *Scene graph view constructor* 32a ⟩+ ≡

```
        new_action_label = QtCore.QCoreApplication.translate(
            __class__.__name__, 'New scene'
        )
        new_action = QtWidgets.QAction(new_action_label, self)
        new_action.setShortcut(Qt.QKeySequence('a'))
        new_action.setShortcutContext(QtCore.Qt.WidgetShortcut)
        new_action.triggered.connect(self.on_new_tree_item)
        self.addAction(new_action)
```

◇

Fragment defined by 29c, 32ab.
Fragment referenced in 29b.

Removal of a selected node shall be triggered upon the press of the `delete` key on the keyboard.

⟨ *Scene graph view constructor* 32b ⟩ ≡

```
        remove_action_label = QtCore.QCoreApplication.translate(
            __class__.__name__, 'Remove selected scene(s)'
        )
        remove_action = QtWidgets.QAction(remove_action_label, self)
        remove_action.setShortcut(Qt.QKeySequence('Delete'))
        remove_action.setShortcutContext(QtCore.Qt.WidgetShortcut)
        remove_action.triggered.connect(self.on_tree_item_removed)
        self.addAction(remove_action)
```

◇

Fragment defined by 29c, 32ab.
Fragment referenced in 29b.

As can be seen in the two above listings, the `triggered` signals are connected with a corresponding slot. All these slots do is emitting another signal, but this time it contains a scene view model, which may be used by other components, instead of a model index.

⟨ *Scene graph view slots* 33 ⟩+ ≡

```
@QtCore.pyqtSlot()
def on_new_tree_item(self):
    """Slot which is called when a new tree item was added by the scene graph
    view.

    This method emits the selected scene graph item as new tree item in form of
    a scene view model.
    """

    selected_indexes = self.selectedIndexes()

    # Sanity check: is actually an item selected?
    if len(selected_indexes) > 0:
        selected_item = selected_indexes[0]
        self.do_add_item.emit(selected_item)
```

⟨ *Scene graph view log tree item added* 38b ⟩

```
@QtCore.pyqtSlot()
def on_tree_item_removed(self):
    """Slot which is called when a one or multiple tree items were removed by
    the scene graph view.

    This method emits the removed scene graph item in form of scene graph view
    models.
    """

    selected_indexes = self.selectedIndexes()

    # Sanity check: is actually an item selected? And has that item a parent?
    # We only allow removal of items with a valid parent, as we do not want to
    # have the root item removed.
    if len(selected_indexes) > 0:
        selected_item = selected_indexes[0]
        if selected_item.parent().isValid():
            self.do_remove_item.emit(selected_item)
```

⟨ *Scene graph view log tree item removed* 38c ⟩

```
        else:
            print('Not removing root scene')
    else:
        print('No item selected')
```

◇

Fragment defined by 30b, 33.
Fragment referenced in 29b.

One of the mentioned other components is the scene graph controller. He needs to be informed, so that he is able to manage the data model correspondingly.

⟨ *Scene graph controller slots* 34a ⟩ ≡

```
@QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_added(self, selected_item):
    self.insertRows(0, 1, selected_item)

@QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_removed(self, selected_item):
    if not selected_item.isValid():
        print('selected scene not valid, not removing')
        return False

    row = selected_item.row()
    parent = selected_item.parent()
    self.removeRows(row, 1, parent)
```

◇

Fragment referenced in 22a.

Having the slots for adding and removing scene graph items implemented, the actual methods for these actions are still missing.

When inserting a new scene graph item, actually a row must be inserted, as the data model (Qt's) is using rows to represent the data. At the same time the controller has to create and keep track of the domain model.

As can be seen in the implementation below, it is not necessary to add the created view model instance to a list of nodes, the usage of `QAbstractItemModel` keeps already track of this.

⟨ *Scene graph controller methods* 34b ⟩ ≡

```
def insertRows(self, row, count, parent=QtCore.QModelIndex()):
    # TODO: Document method.

    if not parent.isValid():
        return False

    parent_node = parent.internalPointer()
    self.beginInsertRows(parent, row, row + count - 1)
    domain_model = domain_scene.SceneModel()
    view_model = guidomain_scene.SceneViewModel(
        row=row,
        domain_object=domain_model,
        parent=parent_node
    )
    self.endInsertRows()

    self.layoutChanged.emit()
    self.do_add_scene.emit(domain_model)

    return True
```

◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

The same logic applies when removing a scene.

⟨ *Scene graph controller methods* 35a ⟩ ≡

```
def removeRows(self, row, count, parent=QtCore.QModelIndex()):
    # TODO: Document method.

    if not parent.isValid():
        return False

    self.beginRemoveRows(parent, row, row + count - 1)
    node_index = parent.child(row, parent.column())
    node       = node_index.internalPointer()
    node.setParent(None)
    # TODO: Still needed? parent_node.child_nodes.remove(node)
    self.endRemoveRows()
    print('Remove rows')

    self.layoutChanged.emit()
    self.do_remove_scene.emit(domain_model)

    return True
```
◇

Fragment defined by 22bc, 23ab, 24, 25ab, 26, 27, 28a, 34b, 35a.
Fragment referenced in 22a.

As before, the main application needs connect the components, in this case the scene graph view with the controller.

⟨ *Connect components for main application* 35b ⟩+ ≡

```
self.main_window.scene_graph_widget.do_add_item.connect(
    self.scene_graph_controller.on_tree_item_added
)
self.main_window.scene_graph_widget.do_remove_item.connect(
    self.scene_graph_controller.on_tree_item_removed
)
```
◇

Fragment defined by 30a, 35b.
Fragment referenced in 16a.

To inform other components, such as the node graph for example, the scene graph controller emits signals when a scene is being added or removed respectively.

⟨ *Scene graph controller signals* 35c ⟩ ≡

```
do_add_scene   = QtCore.pyqtSignal(domain_scene.SceneModel)
do_remove_scene = QtCore.pyqtSignal(domain_scene.SceneModel)
```
◇

Fragment referenced in 22a.

At this point it is possible to manage scenes in terms of adding and removing them. The scenes are added to (or removed from respectively) the graphical user interface as well as the data structure.

So far the application (or rather the scene graph) seems to be working as intended. But how does one ensure, that it really does? Without a doubt, unit and integration tests are one of the best instruments to ensure functionality of code.

As stated before, in subsection 3.0.1, it was an intention of this project to develop the application test driven. Due to required amount of work for developing test driven, it was abstained from this intention and regular unit tests are written instead, which can be found in appendix, section 5.3.

But nevertheless, it would be very handy to have at least some idea what the code is doing at certain places and at certain times.

One of the simplest approaches to achieve this, is a verbose output at various places of the application, which may be as simple as using Python's `print` function. Using the `print` function may allow printing something immediately, but it lacks of flexibility and demands each time a bit of effort to format the output accordingly (e.g. adding the class and the function name and so on).

Python's logging facility provides much more functionality while being able to keep things simple as well — if needed. The usage of the logging facility to log messages throughout the application may later even be used to implement a widget which outputs those messages. So logging using Python's logging facility will be implemented and applied for being able to have feedback when needed.

Logging

It is always very useful to have a facility which allows tracing of errors or even just the flow of an application. Logging does allow such aspects by outputting text messages to a defined output, such as STDERR, STDOUT, streams or files.

Logging shall be provided on a class-basis, meaning that each class (which wants to log something) needs to instantiate a logger and use a corresponding handler.

As logging is a very central aspect of the application, it is the task of the main application to set up the logging facility which may then be used by other classes through a decorator.

The main application shall therefore set up the logging facility as follows:

- Use either an external logging configuration or the default logging configuration.
- When using an external logging configuration
  - The location of the external logging configuration may be set by the environment variable `QDE_LOG_CFG`.
  - Is no such environment variable set, the configuration file is assumed to be named `logging.json` and to reside in the application's main directory.
- When using no external logging configuration, the default logging configuration defined by `basicConfig` is used.
- Always set a level when using no external logging configuration, the default being `INFO`.

⟨ *Main application methods* 37a ⟩+ ≡

```
def setup_logging(self,
                  default_path='logging.json',
                  default_level=logging.INFO):
    """Setup logging configuration"""

    env_key = 'QDE_LOG_CFG'
    env_path = os.getenv(env_key, None)
    path     = env_path or default_path

    if os.path.exists(path):
        with open(path, 'rt') as f:
            config = json.load(f)
            logging.config.dictConfig(config)
    else:
        logging.basicConfig(level=default_level)
```
◇

Fragment defined by 16a, 37a.
Fragment referenced in 15b.

For not having only basic logging available, a logging configuration is defined. The logging configuration provides three handlers: a console handler, which logs debug messages to STDOUT, a info file handler, which logs informational messages to a file named `info.log`, and a error file handler, which logs errors to a file named `error.log`. The default level is set to debug and all handlers are used.

This configuration allows to get an arbitrarily named logger which uses that configuration.

As stated before, logging shall be provided on a class basis. This has the consequence, that each class has to instantiate a logging instance. To prevent the repetition of the same code fragment over and over, Python's decorator pattern is used [13].

The decorator will be available as a method named `with_logger`. The method has the following functionality.

- Provide a name based on the current module and class.

  ⟨ *Set logger name* 37b ⟩ ≡
  ```
  logger_name = "{module_name}.{class_name}".format(
      module_name=cls.__module__,
      class_name=cls.__name__
  )
  ```
  ◇

  Fragment referenced in 48a.

- Provide an easy to use interface for logging.

  ⟨ *Logger interface* 37c ⟩ ≡
  ```
  cls.logger = logging.getLogger(logger_name)

  return cls
  ```
  ◇

  Fragment referenced in 48a.

The implementation of the `with_logger` method allows the usage of the logging facility as a decorator, as shown in the example in the following listing.

[13] https://www.python.org/dev/peps/pep-0318/

⟨ *With logger example* 38a ⟩ ≡

```
from qde.editor.foundation import common

@common.with_logger
def SomeClass(object):
    """This class provides literally nothing and is used only to demonstrate the
    usage of the logging decorator."""

    def some_method():
        """This method does literally nothing and is used only to demonstrate the
        usage of the logging decorator."""

        self.logger.debug(("I am some logging entry used for"
                           "demonstration purposes only."))
```

◇

Fragment never referenced.

The logging facility may now be used wherever it is useful to log something. Such a place is for example the adding and removal of scenes in the scene graph view.

⟨ *Scene graph view log tree item added* 38b ⟩ ≡

```
self.logger.debug("A new scene graph item was added.")
```

◇

Fragment referenced in 33.

⟨ *Scene graph view log tree item removed* 38c ⟩ ≡

```
self.logger.debug((
    "The scene graph item at row {row} "
    "and column {column} was removed."
).format(
    row=selected_item.row(),
    column=selected_item.column()
))
```

◇

Fragment referenced in 33.

Whenever the *a* or the *delete* key is being pressed now, when the scene graph view is focused, the corresponding log messages appear in the standard output, hence the console.

Now, having the scene graph component as well as an interface to log messages throughout the application implemented, the next component may be approached. A very interesting aspect to face would be the rendering. But for being able to render something, there actually needs to exist something to render: nodes. Nodes are being represented within the node graph. So this is a good point to begin with the implementation of the node graph.

5.1.5 Node graph

The functionality of the node graph is, as its name states, to represent a data structure composed of nodes and edges. Each scene from the scene graph is represented within the node graph as such a data structure.

The nodes are the building blocks of a real time animation. They represent different aspects, such as scenes themselves, time line clips, models, cameras, lights, materials, generic operators and effects. These aspects are only examples (coming from *QDE - a visual animation system. Software-Architektur.* p. 30 and 31) as the node structure will be expandable for allowing the addition of new nodes.

The implementation of the scene graph component was relatively straightforward partly due to its structure and partly due to the used data model and representation. The node graph component however, seems to be a bit more complex.

To get a first overview and to manage its complexity, it might be good to identify its sub components first before implementing them. When thinking about the implementation of the node graph, one may identify the following sub components:

Nodes Building blocks of a real time animation.

- **Domain model** Holds data of a node, like its definition, its inputs and so on.
- **Definitions** Represents a domain model as JSON data structure.
- **Controller** Handles the loading of node definitions as well as the creation of node instances.
- **View model** Represents a node within the graphical user interface.

Scenes A composition of nodes, connected by edges.

- **Domain model** Holds the data of a scene, e.g. its nodes.
- **Controller** Handles scene related actions, like when a node is added to a scene, when the scene was changed or when a node within a scene was selected.
- **View model** Defines the graphical representation of scene which can be represented by the corresponding view. Basically the scene view model is a canvas consisting of nodes.
- **View** Represents scenes in terms of scene view models within the graphical user interface.

Nodes

As mentioned before, nodes are the building blocks of a real time animation. But what are those definitions actually? What do they actually define? There is not only one answer to this question, it is simply a matter of how the implementation is being done and therefore a set of decisions.

The whole (rendering) system shall not be bound to only one representation of nodes, e.g. triangle based meshes. Instead it shall let the user decide, what representation is the most fitting for the goal he wants to achieve.

Therefore the system shall be able to support multiple kinds of node representations: Images, triangle based meshes and solid modeling through function modeling (using signed distance functions for modeling implicit surfaces). Whereas triangle based meshes may either be loaded from externally defined files (e.g. in the Filmbox (FBX), the Alembic (ABC) or the Object file format (OBJ)) or directly be generated using procedural mesh generation.

The nodes are always part of a graph, hence the name node graph, and are therefore typically connected by edges. This means that the graph gets evaluated recursively by its nodes, starting with the root node within the root scene. However, the goal is to have OpenGL shading language (GLSL) code at the end, independent of the node types.

From this point of view it would make sense to let the user define shader code directly within a node (definition) and to simply evaluate this code, which adds a lot of (creative) freedom. The problem with this approach is though, that image and triangle based mesh nodes are not fully implementable by using shader code only. Instead they have specific requirements, which are only perform-able on the CPU (e.g. allocating buffer objects).

When thinking of nodes used for solid modeling however, it may appear, that they may be evaluated directly, without the need for pre-processing, as they are fully implementable using shader code only. This is kind of misleading however as each node has its own definition which has to be added to shader

and this definition is then used in a mapping function to compose the scene. This would mean to add a definition of a node over and over again, when spawning multiple instances of the same node type, which results in overhead bloating the shader. It is therefore necessary to pre-process solid modeling nodes too, exactly as triangle mesh based and image nodes, for being able to use multiple instances of the same node type within a scene while having the definition added only once.

All of these thoughts sum up in one central question for the implementation: Shall objects be predefined within the code (and therefore only nodes accepted whose type and sub type match those of predefined nodes) or shall all objects be defined externally using files?

This is a question which is not that easy to answer. Both methods have their advantages and disadvantages. Pre-defining nodes within the code minimizes unexpected behavior of the application. Only known and well-defined nodes are processed.

But what if someone would like to have a new node type which is not yet defined? The node type has to be implemented first. As Python is used for the editor application, this is not really a problem as the code is interpreted each time and is therefore not being compiled. Nevertheless such changes follow a certain process, such as making the actual changes within the code, reviewing and checking-in the code and so on, which the user normally does not want to be bothered with. Furthermore, when thinking about the player application, the problem of the necessity to recompile the code is definitively given. The player will be implemented in C, as there is the need for performance, which Python may not fulfill satisfactorily.

Considering these aspects, the external definition of nodes is chosen. This may result in nodes which cannot be evaluated or which have unwanted effects. As it is (most likely) in the users best interest to create (for his taste) appealing real time animations, it can be assumed, that the user will try avoiding to create such nodes or quickly correct faulty nodes or simply does not use such nodes.

Now, having chosen how to implement nodes, let us define what a node actually is. As a node may be reference by other nodes, it must be uniquely identifiable and must therefore have a globally unique identifier. Concerning the visual representation, a node shall have a name as well as a description.

Each node can have multiple inputs and at least one output. The inputs may be either be atomic types (which have to be defined) or references to other nodes. The same applies to the outputs.

A node shall be able to have one or more parts. A part typically contains the "body" of the node in terms of code and represents therefore the code-wise implementation of the node. A part can be processed when evaluating the node.

Furthermore a node may contain children, child-nodes, which are actually references to other nodes combined with properties such as a name, states and so on.

Each node can have multiple connections. A connection is composed of an input plus a reference to a part of that input as well as an output and a reference to a part of that output. The input respectively the output may be zero, what means that the part of the input or output is internal. Or, a bit more formal:

⟨ *Connections between nodes in EBNF notation* 40 ⟩ ≡

```
input = internal input | external input
internal input = zero reference, part reference
external input = node reference, part reference
zero reference = "0"
node reference = "uuid4"
part reference = "uuid4"
◇
```

Fragment never referenced.

Component	Description
ID	A global unique identifier (UUID [14])
Name	The name of the node, e.g. "Cube".
Description	A description of the node's purpose.
Inputs	A list of the node's inputs. The inputs may either be parameters (which are atomic types such as float values or text input) or references to other nodes.
Outputs	A list of the node's outputs. The outputs may also either be parameters or references to other nodes.
Parts	Defines parts that may be processed when evaluating the node. Contains code which can be interpreted directly.
Nodes	The children a node has (child nodes). These entries are references to other nodes only.
Connections	A list of connections of the node's inputs and outputs. Each connection is composed by two parts: A reference to another node and a reference to an input or an output of that node. Is the reference not set, that is, its value is zero, this means that the connection is internal.

Recapitulating the above made thoughts, a node is essentially composed by the following elements:

The inputs and outputs may be parameters of an atomic type, as stated above. This seems like a good point to define the atomic types the system will have:

- Generic
- Float
- Text
- Scene
- Image
- Dynamic
- Mesh

As these atomic types are the foundation of all other nodes, the system must ensure, that they are initialized before all other nodes. Before being able to create the atomic types there must be classes defining them.

For identification of the atomic types, an enumerator is used. Python provides the `enum` module, which provides a convenient interface for using enumerations[15].

⟨ *Node type declarations* 41 ⟩ ≡

```
class NodeType(enum.Enum):
    """Atomic types which a parameter may be made of."""

    GENERIC = 0
    FLOAT   = 1
    TEXT    = 2
    SCENE   = 3
    IMAGE   = 4
    DYNAMIC = 5
    MESH    = 6
```

◇

Fragment referenced in 48c.

[15] https://docs.python.org/3/library/enum.html

Now, having identifiers for the atomic types available, the atomic types themselves can be implemented. The atomic types will be used for defining various properties of a node and are therefore its parameters.

Each node may contain one or more parameters as inputs and one parameter as output. Each parameter will lead back to its atomic type by referencing the unique identifier of the atomic type. For being able to distinguish multiple parameters using the same atomic type, it is necessary that each instance of an atomic type has its own identifier in form of an instance identifier (instance ID).

As the word atomic in atomic type indicates, these types are atomic, meaning there only exists one explicit instance per atomic type.

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 Tue
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 Test cases

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.

5.4 Requirements

5.5 Directory structure and name-spaces

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

5.6 Code fragments

"../src/editor.py" 44≡

```
#!/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 15a ⟩
◇
```

"../src/qde/editor/application/application.py" 45a≡

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

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

# System imports
import logging
import logging.config
import os
import json
from PyQt5 import QtGui
from PyQt5 import QtWidgets

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


⟨ Main application declarations 15b ⟩
◇
```

"../src/qde/editor/gui/main_window.py" 45b≡

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

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

# System imports
from PyQt5 import QtCore
from PyQt5 import QtWidgets

# Project imports
from qde.editor.gui import scene as guiscene


⟨ Main window declarations 16c ⟩
◇
```

"../src/qde/editor/domain/scene.py" 45c≡

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

""" Module holding scene related aspects concerning the domain layer. """

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore

# Project imports


⟨ Scene model declarations 20a ⟩
◇
```

"../src/qde/editor/gui_domain/scene.py" 46a≡

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

""" Module holding scene related aspects concerning the gui_domain layer. """

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore

# Project imports

⟨ Scene view model declarations 21a ⟩
⟨ Scene graph view model declarations ? ⟩
◇
```

"../src/qde/editor/application/scene_graph.py" 46b≡

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

""" Module holding scene graph related aspects concerning the application layer.
"""

# System imports
from PyQt5 import QtCore

# Project imports
from qde.editor.domain  import scene as domain_scene
from qde.editor.gui_domain import scene as guidomain_scene

⟨ Scene graph controller declarations 22a ⟩
◇
```

"../src/qde/editor/gui/scene.py" 46c≡

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

""" Module holding scene related aspects concerning the graphical user interface layer.
"""

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore
from PyQt5 import QtWidgets

# Project imports
from qde.editor.foundation import common
from qde.editor.gui_domain import scene

⟨ Scene graph view declarations 29b ⟩
◇
```

"../logging.json" 47≡

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(asctime)s - %(levelname)-7s - %(name)s.%(funcName)s::%(lineno)s: %(message)s"
        }
    },

    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "level": "DEBUG",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        },

        "info_file_handler": {
            "class": "logging.handlers.RotatingFileHandler",
            "level": "INFO",
            "formatter": "simple",
            "filename": "info.log",
            "maxBytes": 10485760,
            "backupCount": 20,
            "encoding": "utf8"
        },

        "error_file_handler": {
            "class": "logging.handlers.RotatingFileHandler",
            "level": "ERROR",
            "formatter": "simple",
            "filename": "errors.log",
            "maxBytes": 10485760,
            "backupCount": 20,
            "encoding": "utf8"
        }
    },

    "root": {
        "level": "DEBUG",
        "handlers": ["console", "info_file_handler", "error_file_handler"],
        "propagate": "no"
    }
}◇
```

"../src/qde/editor/foundation/common.py" 48a≡

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

"""Module holding common helper methods."""

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore
from PyQt5 import QtWidgets

# Project imports
from qde.editor.gui_domain import scene


def with_logger(cls):
    """Add a logger instance (using a stream handler) to the given class.

    :param cls: the class which the logger shall be added to.
    :type cls: a class of type cls.

    :return: the class with the logger instance added.
    :rtype: a class of type cls.
    """
```

⟨ *Set logger name* 37b ⟩
⟨ *Logger interface* 37c ⟩
◇

⟨ *Scene graph view decorators* 48b ⟩ ≡

```
@common.with_logger
```

◇

Fragment referenced in 29b.

"../src/qde/editor/foundation/type.py" 48c≡

```
# -*- coding: utf-8 -*-
"""Module for type-specific aspects."""

# System imports
import enum

# Project imports


```

⟨ *Node type declarations* 41 ⟩
◇