QDE.

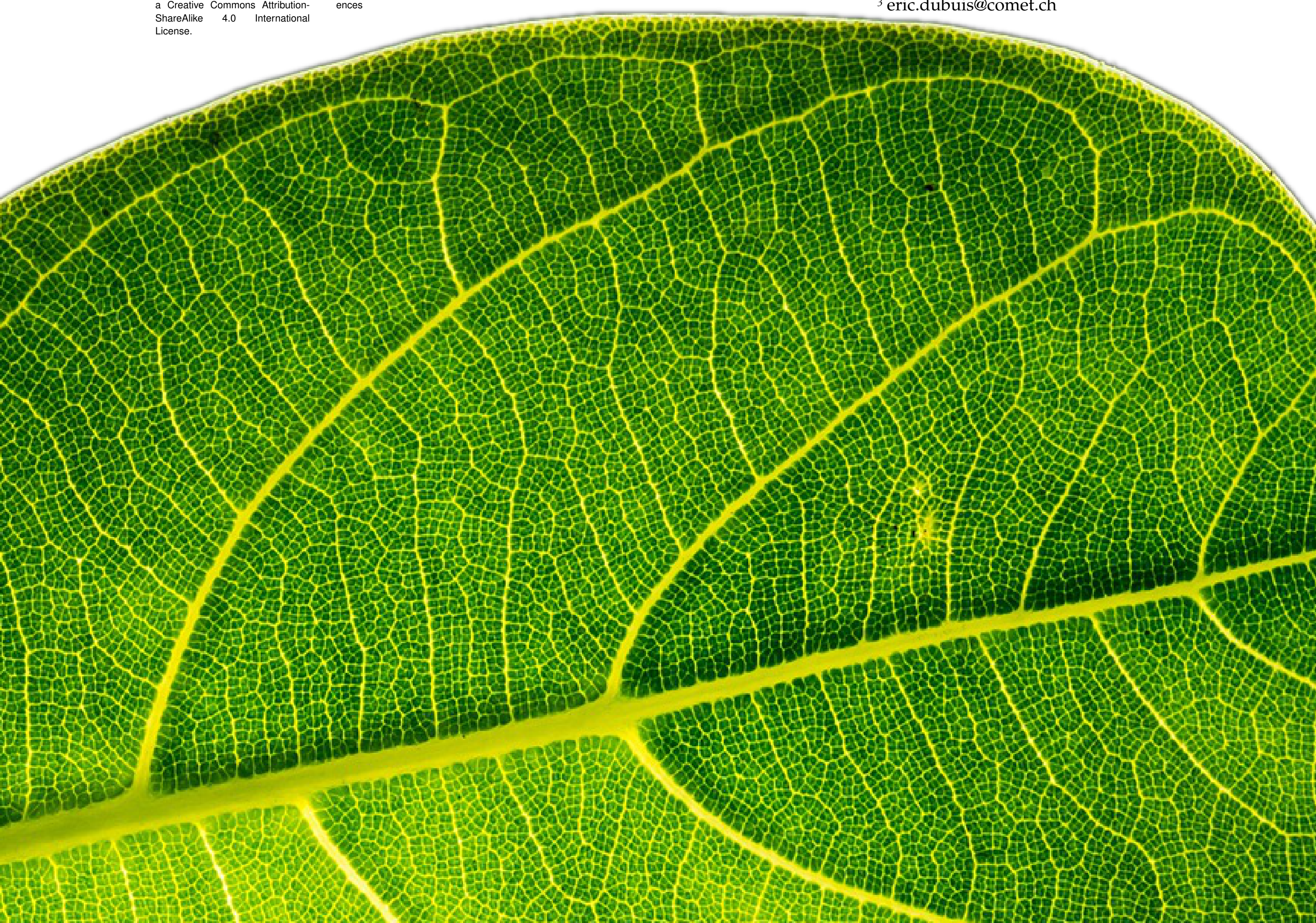
A SYSTEM FOR COMPOSING REAL TIME COMPUTER GRAPHICS.

MTE7103 — MASTER THESIS

Major:	Computer science
Author:	Sven Osterwalder[1]
Advisor:	Prof. Claude Fuhrer[2]
Expert:	Dr. Eric Dubuis[3]
Date:	2017-06-11 22:41:25
Version:	66f4421

Berne University of Applied Sciences

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



Revision History

Revision	Date	Author(s)	Description
89c7544	2017-02-28 17:09:43	SO	Set up initial project structure, provide first content
10192d8	2017-03-02 22:37:17	SO	Set up project schedule
54f4b23	2017-03-05 22:53:15	SO	Set up project structure, implement main entry point and main window
ffda0e5	2017-03-15 10:58:51	SO	Scene graph, logging, adapt project schedule
34b09b7	2017-03-24 17:08:22	SO	Update meeting minutes, thoughts about node implementation
06fe268	2017-04-03 23:00:35	SO	Add requirements, node graph implemenetation
d264175	2017-04-10 16:07:01	SO	Conversion from Org-Mode to Nuweb, revise editor implementation
f8b524b	2017-04-30 23:42:57	SO	Approach node graph and nodes
2f46832	2017-05-04 21:13:41	SO	Impel node definitions further
ae34fc5	2017-05-24 13:56:31	SO	Change class to tufte-book, title page, introduction, further implementation
27c549c	2017-05-24 22:28:41	SO	Change document structure, re-work admin. aspects
6772ef9	2017-05-27 14:18:09	SO	Adapt document structure, fundamentals: introduction and rendering
399669d	2017-05-29 09:10:48	SO	Finish fundamentals
66f4421	2017-06-11 22:41:25	SO	Add methodologies, begin adding results

Abstract

Provide correct abstract.

A highly optimized rendering algorithm based on ray tracing is presented. It outperforms the classical ray tracing methods and allows the rendering of ray traced scenes in real-time on the GPU. The classical approach for modelling scenes using triangulated meshes is replaced by mathematical descriptions based on signed distance functions. The effectiveness of the algorithm is demonstrated using a prototype application which renders a simple scene in real-time.

Contents

List of Figures

List of Tables

Introduction

The subject of computer graphics exists since the beginning of modern computing. Ever since the subject of computer graphics has strived to create realistic depictions of the observable reality. Over time various approaches for creating artificial images (the so called rendering) evolved. One of those approaches is ray tracing. It was introduced in 1968 by Appel in the work "Some Techniques for Shading Machine Renderings of Solids" [1]. In 1980 it was improved by Whitted in his work "An Improved Illumination Model for Shaded Display" [2].

Ray tracing captivates through simplicity while providing a very high image quality including perfect refractions and reflections. For a long time although, the approach was not performant enough to deliver images in real time. Real time means being able to render at least 25 rendered images (frames) within a second. Otherwise, due to the human anatomy, the output is perceived as either still images or as a too slow animation.

Sphere tracing is a ray tracing approach introduced in 1994 by Hart in his work "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces" [3]. This approach is faster than the classical ray tracing approaches in finding intersections between rays and objects. The speed up is achieved by using signed distance functions for modeling the objects to be rendered and by expanding volumes for finding intersections.

Graphics processing units (GPUs) have evolved over time and have gotten more powerful in processing power. Since around 2009 GPUs are able to produce real time computer graphics using sphere tracing. While allowing ray tracing in real time on modern GPUs, sphere tracing has also a clear disadvantage. The de facto way of representing objects, using triangle based meshes, cannot be used directly. Instead distance fields defined by implicit functions build the basis for sphere tracing.

Purpose and situation

Motivation

To this point in time there are no solutions (at least none are known to the author), that provide a convenient way for modeling, animating and rendering objects and scenes using signed distance functions for modeling and sphere tracing for rendering. Most of the solutions using sphere tracing implement it by having one or multiple big fragment shaders containing everything from modeling to lighting. Other solutions provide node based approaches, but they allow either no sphere tracing at all, meaning they use rasterization, or they provide nodes containing (fragment-) shader code, which leads again to a single big fragment shader.

This thesis aims at designing and developing a software which provides both: a node based approach for modeling and animating objects using signed distance functions as well as allowing the composition of scenes while rendering objects, or scenes respectively, in real time on the GPU using sphere tracing.

Objectives and limitations

The objective of this thesis is the design and development of a software for *modeling*, *composing* and *rendering* real time computer graphics through a graphical toolbox.

Modeling is done by composing single nodes to objects using a node based graph structure.

Compositing includes two aspects: the composition of objects into scenes and the composition of an animation which is defined by multiple scenes which follow a chronological order. The first aspect is realized by a scene graph structure, which contains at least a root scene. Each scene may contain nodes. The second aspect is realized by a time line, which allows a chronological organization of scenes.

For rendering a highly optimized algorithm based on ray tracing is used. The algorithm is called sphere tracing and allows the rendering of ray traced scenes in real time on the GPU. Contingent upon the used rendering algorithm all models are modeled using implicit surfaces. In addition mesh-based models and corresponding rendering algorithms may be implemented.

Required objectives are the following:

- Development of an editor for creating and editing real time rendered scenes, containing the following features.
  - A scene graph, allowing management (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.
  - Nodes for the node-based graph structure.
    * Simple objects defined by signed distance functions: Cube and sphere
    * Simple operations: Merge/Union, Intersection, Difference
    * Transformations: Rotate, Translate and Scale
    * Camera
    * Renderer (ray traced rendering using sphere tracing)
    * Lights

Optional objectives are the following:

- Additional features for the editor, as follows.
  - A sequencer, allowing a time-based scheduling of defined scenes.
  - Additional nodes, such as operations (e.g. replication of objects) or post-processing effects (glow/glare, color grading and so on).
- Development of a standalone player application. The player allows the playback of animations (time-based, compounded scenes in sequential order) created with the editor.

Related works

Preliminary to this thesis two project works were done: "Volume ray casting — basics & principles" [4], which describes the basics and principles of sphere tracing, a special form of ray tracing, and "QDE — a visual animation system, architecture" [5], 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.

Document structure

This document is divided into six 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 *fundamentals*. The purpose of that chapter is to present the fundamentals, that this thesis is built upon. One aspect is a framework for the implementation of the intended software, which is heavily based on the previous project work, "QDE — a visual animation system, architecture" [5]. Another aspect is the rendering, which is using a special form of ray tracing as described in "Volume ray casting — basics & principles" [4].

The next chapter on the *methodologies* introduces a concept called literate programming and elaborates some details of the implementation using literate programming. Additionally it introduces standards and principles concerning the implementation of the intended software.

The following chapter on the *results* concludes on the implementation of the editor and the player components.

The last chapter is *discussion and conclusion* and discusses the methodologies as well as the results. 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.

Administrative aspects

THE LAST CHAPTER provided an introduction to this thesis by outlining the purpose and situation, the related works and the document structure.

THIS CHAPTER covers some administrative aspects of this thesis, they are although not required for understanding of the result.

THE FIRST SECTION defines the involved persons and their role during this thesis. Afterwards the deliverable items are shown and described. The last section elaborates on the organization of work including meetings, the phases and milestones as well as the thesis's schedule.

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

Involved persons

Table 2: List of the involved persons.

Role	**Name**	**Task**
Author	Sven Osterwalder[1]	Author of the thesis.
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.

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

Deliverables

Deliverable	Description
Report	The report contains the theoretical and technical details for implementing a system for composing real time computer graphics.
Implementation	The implementation of a system for composing real time computer graphics, which was developped during this thesis.

Table 3: List of deliverables.

Organization of work

Meetings

Various meetings with the supervisor and the expert helped reaching the defined goals and preventing erroneous directions of the thesis. The supervisor and the expert supported the author of this thesis by providing suggestions throughout the held meetings. The minutes of the meetings may be found under meeting minutes.

Add correct reference

Phases and milestones

Phase	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 4: Phases of the project.

Milestone	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 5: Milestones of the project.

Schedule

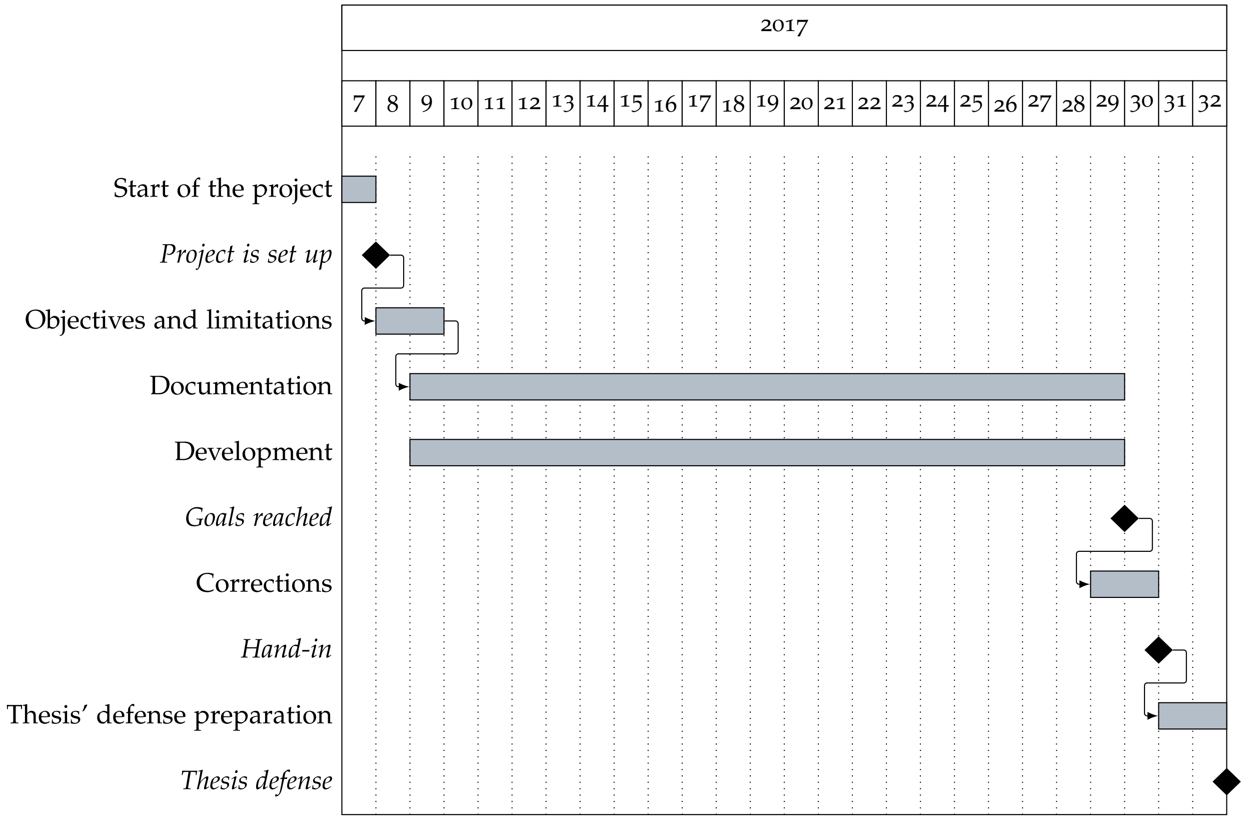


Figure 1: Schedule of the project. The subtitle displays calendar weeks.

Fundamentals

THE PREVIOUS CHAPTER covered some administrative aspects including the involved persons, the phases and milestones of the thesis as well as its schedule.

THIS CHAPTER presents the fundamentals which are required for understanding of the result of this thesis.

THE FIRST SECTION OF THIS CHAPTER defines the software architecture that is used for the implementation of the intended software. It is mainly a summary of the previous project work, "QDE — a visual animation system, architecture" [5]. The second section shows the algorithm which is used for rendering. It is a summary of a previous project work, "Volume ray casting — basics & principles" [4].

Software architecture

THIS SECTION is a summary of the previous project work of the author, "QDE — a visual animation system, architecture" [5]. It describes the fundamentals for the architecture for the intended software of this thesis.

SOFTWARE ARCHITECTURE is inherent to software engineering and software development. It may be done implicitly, for example when developing a smaller software where the concepts are somewhat intuitively clear and the decisions forming the design are worked out in one's head. But it may also be done explicitly, when developing a larger software for example. But what is software architecture? Kruchten defines software architecture as follows.

"AN ARCHITECTURE IS THE *set of significant decisions* about the organization of a software system, the selection of *structural elements* and their interfaces by which the system is composed, together with their *behavior* as specified in the collaborations among those elements, the *composition* of these elements into progressively larger subsystems, and the *architectural style* that guides this organization – these elements and their interfaces, their collaborations, and their composition." [6]

Or as Fowler puts it: "Whether something is part of the architecture is entirely based on whether the developers think it is important. [...] So, this makes it hard to tell people how to describe their architecture. 'Tell us what is important.' Architecture is about the important stuff. Whatever that is." [7]

THE ENVISAGED IDEA OF THIS THESIS, using a node based graph for modeling objects and scenes and rendering them using sphere tracing, was developed ahead of this thesis. To ensure that this idea is really feasible, a prototype was developed during the former project work *Volume ray casting - basics & principles*. This prototype acted as a proof of concept. For this prototype an implicitly defined architecture was used, which led to an architecture which is hard to maintain and extend by providing no clear segregation between the data model and its representation.

WITH THE PREVIOUS PROJECT WORK, *QDE - a visual animation system. Software-Architektur.* a software architecture was developed to prevent this circumstance. The software architecture is based on the unified process, what leads to an iterative approach.

BASED UPON THE VISION actors are defined. The actors in turn are used in use cases, which define functional requirements for the behavior of a system. The definition of use cases shows the extent of the software and define its functionality and therefore the requirements. Based on the these requirements, the components shown in Table 6 are established.

Table 6: Description of the components of the envisaged software.

Component	Description
Player	Reads objects and scenes defined by the editor component and plays them back in the defined chronological order.
Editor	Allows *modeling* and *composing* of objects and scenes using a node based graphical user interface. *Renders* objects and scenes in real time using sphere tracing.
Scene graph	Holds scenes in a tree like structure and has at least a root node.
Node graph	Contains all nodes which define a single scene.
Parameter	Holds the parameters of a node from the node graph.
Rendering	Renders a node.
Time line	Depicts temporal events in terms of scenes which follow a chronological order.

IDENTIFYING THE COMPONENTS helps finding the noteworthy concepts or objects. Decomposing a domain into noteworthy concepts or objects is "the quintessential object-oriented analysis step" [8]. "The domain model is a visual representation of conceptual classes or real-

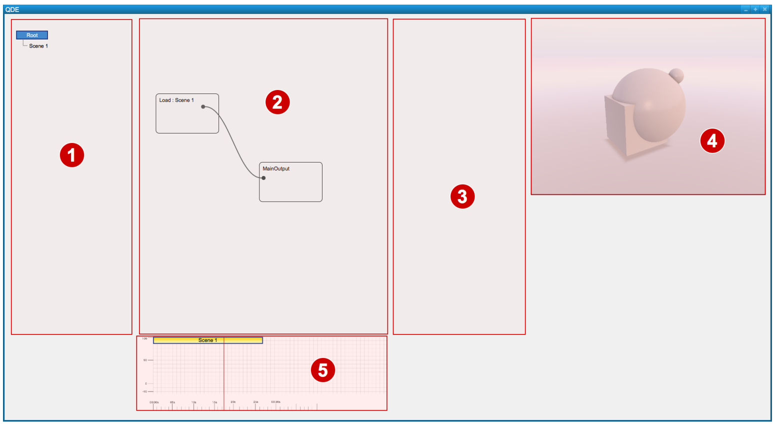


Figure 2: A mock up of the editor application showing its components.
1: Scene graph.
2: Node graph.
3: Parameter view.
4: Rendering view.
5: Time line.

situation objects in a domain." [8] The domain models for the editor and the player component are shown in Figure 3 and in Figure 4 respectively.

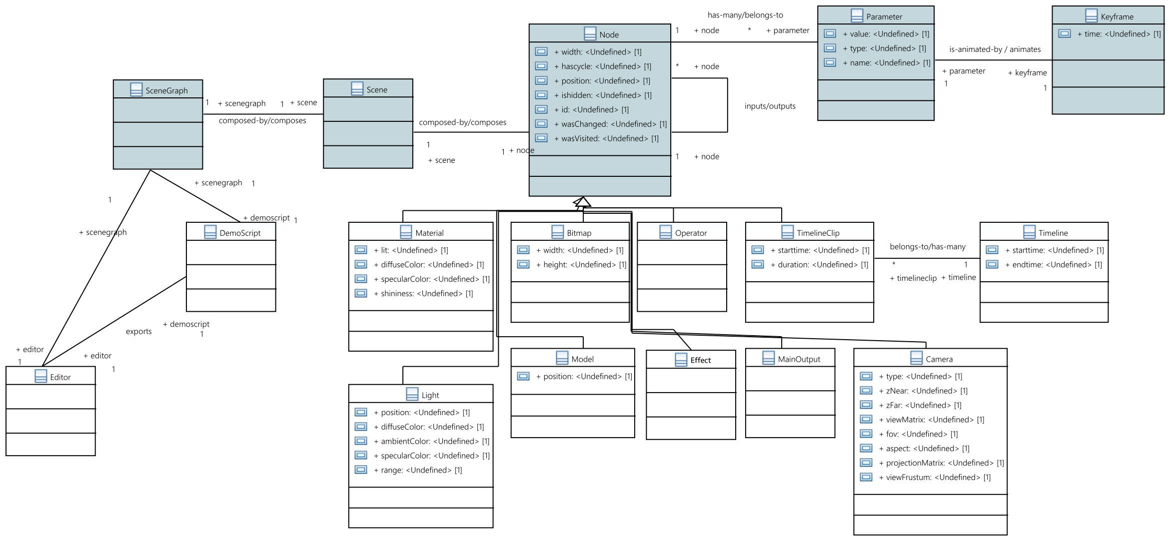


Figure 3: Domain model of the editor component.

Identifying the noteworthy concepts or objects allows the definition of the logical architecture, which shows the overall image of (software) classes in form of packets, subsystems and layers.

To reduce coupling and dependencies a relaxed layered architecture is used. In contrast to a strict layered architecture, which allows any layer calling only services or interfaces from the layer below, the relaxed layered architecture allows higher layers to communicate with any lower layer. To ensure low coupling and dependencies also for the graphical user interface, the models and their views are segregated using the model-view separation principle. This principle states that domain objects should have no direct knowledge about objects of the graphical user interface. In addition controllers are used, which represent workflow objects of the application layer.

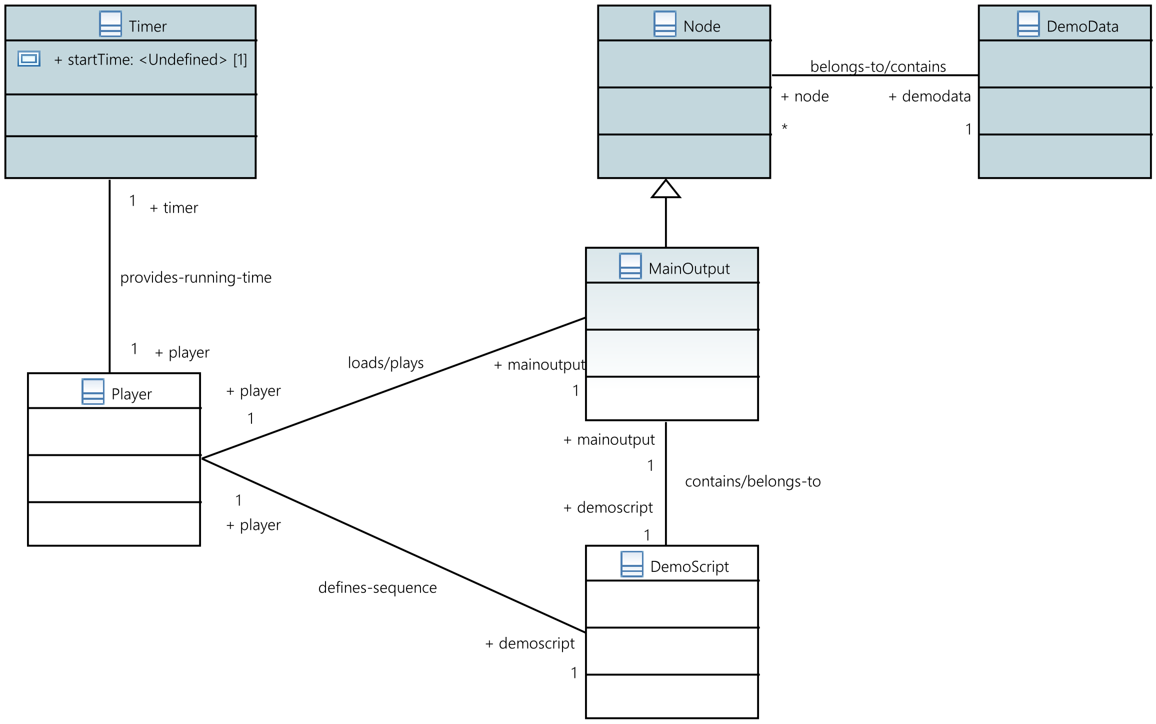


Figure 4: Domain model of the player component.

Table 7: Layers of the envisaged software.

Layer	Description
UI	All elements of the graphical user interface.
Application	Controller/workflow objects.
Domain	Models respectively logic of the application.
Technical services	Technical infrastructure, such as graphics, window creation and so on.
Foundation	Basic elements and low level services, such as a timer, arrays or other data classes.

Class diagrams provide a software point of view whereas domain models provide rather a conceptual point of view. A class diagram shows classes, interfaces and their relationships. Figure 5 shows the class diagram of the editor component whereas Figure 6 shows the class diagram for the player component.

Rendering

This section is a summary of a previous project work of the author, “Volume ray casting — basics & principles” [4]. It describes the fundamentals for the rendering algorithm that is used for the intended software of this thesis.

Rendering is one of the main aspects of this thesis, as the main objective of the thesis is the design and development of a software for modeling, composing and *rendering* real time computer graphics through a graphical user interface. Foley describes rendering as a “process of creating images from models” [9]. The basic idea of ren-

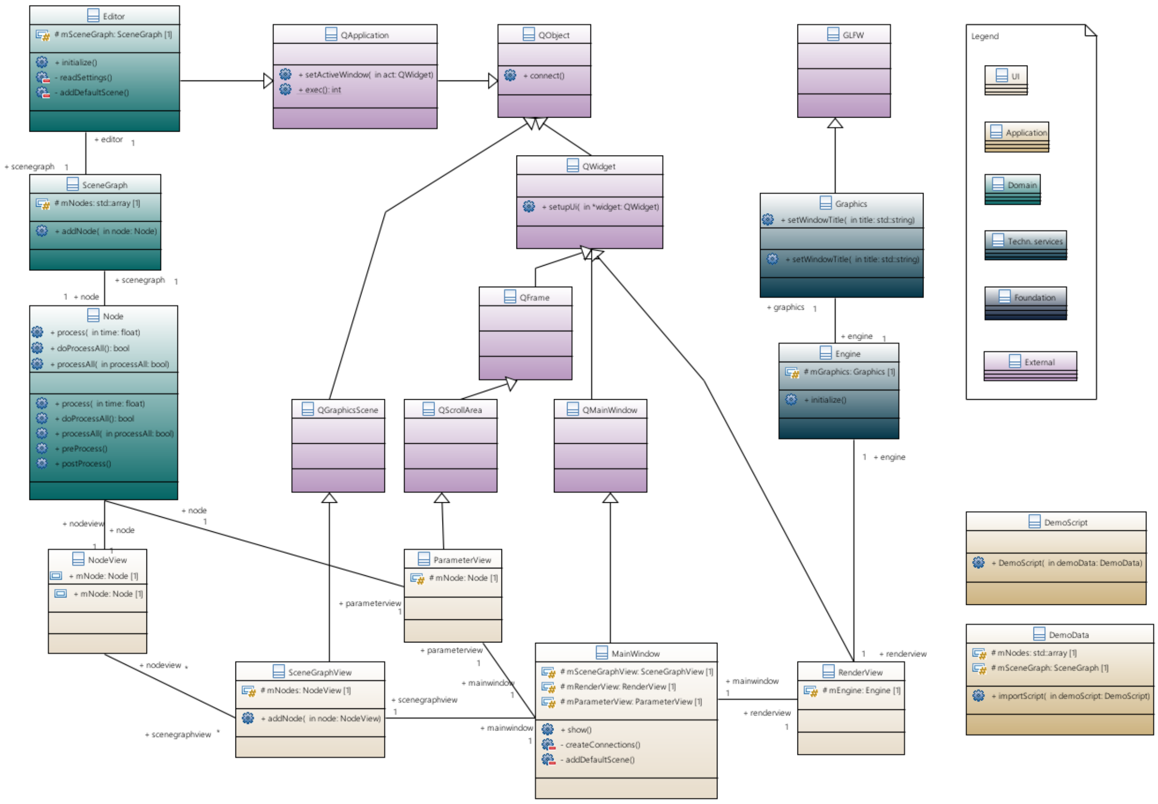


Figure 5: Class diagram of the editor component.

dering is to determine the color of a surface at a certain point. For this task two concepts have evolved: *illumination models* and *shading models*.

Shading models define when to use which illumination model and the parameters for the illumination model.

Illumination models describe the amount of light that is transmitted from a point on a surface to a viewer. There exist two kinds of illumination models: local illumination models and global illumination models. Whereas local illumination models aggregate local data from adjacent surfaces and directly incoming light, global illumination models consider also indirect light. The algorithm used for rendering in the intended software is an algorithm using a *global illumination model*.

Global illumination models "express the light being transferred from one point to another in terms of the intensity of the light emitted from the first point to the second" [9, pp. 775 and 776]. Additionally to this direct intensity the indirect intensity is considered, therefore "the intensity of light emitted from all other points that reaches the first and is reflected from the first to the second" [9, pp. 775 and 776] point is added.

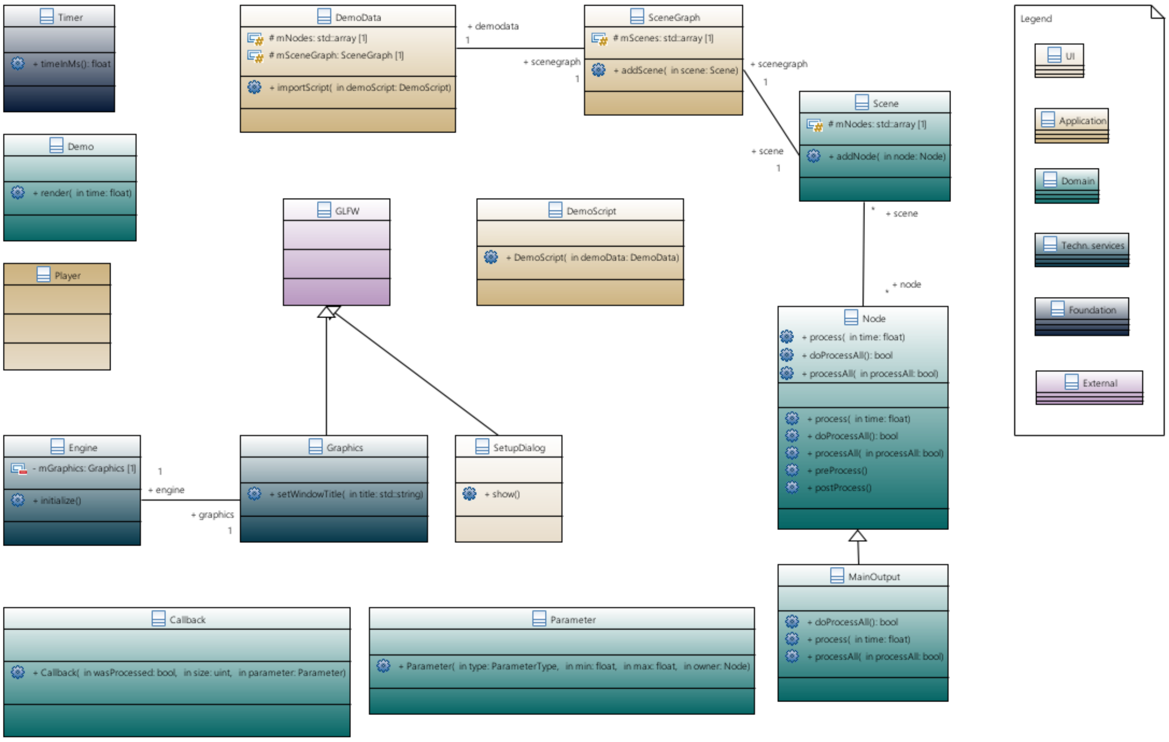


Figure 6: Class diagram of the player component.

In 1986 James "Jim" Kajiya set up the so called rendering equation, which expresses this behavior. [10, 9, p. 776]

$$I(x,x') = g(x,x')[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''] \qquad (1)$$

Figure 7: The rendering equation as defined by James "Jim" Kajiya.

x, x' *and* x'' Points in space.

$I(x,x')$ Intensity of the light going from point x' to point x.

$g(x,x')$ A geometrical term.

- 0 x and x' are occluded by each other.
- $\frac{1}{r^2}$ x and x' are visible to one other, r being the distance between the two points.

$\varepsilon(x,x')$ Intensity of the light being emitted from point x' to point x.

$\rho(x,x',x'')$ Intensity of the light going from x'' to x, being scattered on the surface of point x'.

$\int_S$ Integral over the union of all surfaces, hence $S = \bigcup_{i=0}^{n} S_i$, n being the number of surfaces. All points x, x' and x'' brush all surfaces of all objects within the scene. S_0 being an additional surface in form of a hemisphere which spans the whole scene and acts as background.

Implementing a global illumination model or the rendering equation directly for rendering images in viable or even real time is not really feasible, even on modern hardware. The procedure is computationally complex and very time demanding.

A simplified approach to implement global illumination models (or the rendering equation) is ray tracing. Ray tracing is able to produce high quality, realistic looking images. Although it is still demanding in terms of time and computations, the time complexity is reasonable for producing still images. For producing images in real time however, the procedure is still too demanding. This is where a special form of ray tracing comes in.

Sphere tracing is a ray tracing approach for implicit surfaces introduced in 1994 by Hart in his work "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces" [3]. Sphere tracing is faster than the classical ray tracing approaches in finding intersections between rays and objects. In contrast to the classical ray tracing approaches, the marching distance on rays is not defined by an absolute or a relative distance, instead distance functions are used. The distance functions are used to expand unbounding volumes (in this concrete case spheres, hence the name) along rays. Figure 8 illustrates this procedure.

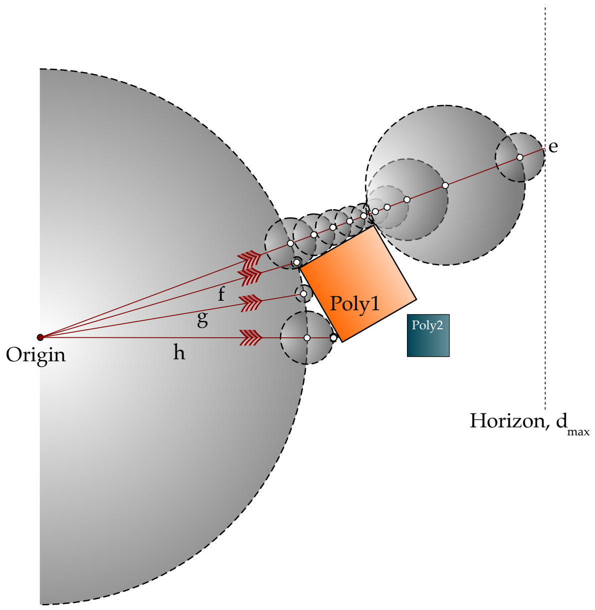


Figure 8: Illustration of the sphere tracing algorithm. Ray e hits no objects until reaching the horizon at d_{max}. Rays f, g and h hit polygon *poly1*.

Unbounding volumes contrast with bounding volumes, which enclose a solid. Unbounding volumes enclose a part of space without including certain objects (whereas including means touching). For calculating a unbounding volume, the distance between an object and the origin is being searched. Is this distance known, it can be taken as a radius of a sphere. Sphere tracing defines objects as implicit surfaces using distance functions. Therefore the distance from every point in space to every other point in space and to every surface of every object is known. These distances build a so called distance field.

The sphere tracing algorithm is as follows. A ray is being shot from a viewer (an eye or a pinhole camera) through the image plane into a scene. The radius of an unbounding volume in form of a sphere is being calculated at the origin, as described above. This radius builds an intersection with the ray and represents the distance, that the ray will travel in a first step. From this intersection the next unbounding volume is being expanded and its radius is being calculated, which gives the next intersection with the ray. This procedure continues until an object is being hit or until a predefined maximum distance of the ray d_{max} is being reached. An object is being hit, whenever the returned radius of the distance function is below a predefined constant ϵ. A possible implementation of the sphere tracing algorithm is shown in Figure 9. This Figure 9 is although only showing the distance estimation. Shading is done outside, for example in a render method which calls the sphere trace method. Shading means in this context the determination of a surface's respectively a pixel's color.

```
def sphere_trace():
    ray_distance          = 0
    estimated_distance    = 0
    max_distance          = 9001
    max_steps             = 100
    convergence_precision = 0.000001

    while ray_distance < max_distance:
        # sd_sphere is a signed distance function defining the implicit surface.
        # cast_ray defines the ray equation given the current traveled /
        # marched distance of the ray.
        estimated_distance = sd_sphere(cast_ray(ray_distance))

        if estimated_distance < convergence_precision:
            # the estimated distance is already smaller than the desired
            # precision of the convergence, so return the distance the ray has
            # travelled as we have an intersection
            return ray_distance

        ray_distance = ray_distance + estimated_distance

    # When we reach this point, there was no intersection between the ray and a
    # implicit surface, so simply return 0
    return 0
```

Figure 9: An abstract implementation of the sphere tracing algorithm. Algorithm in pseudo code, after [3][S. 531, Fig. 1]

Shading is done as proposed by Whitted in "An Improved Illumination Model for Shaded Display" [2]. This means, that the sphere tracing algorithm needs to return which object was hit and the material of this object. Depending on the objects material, three cases can occur: (1) the material is reflective and refractive, (2) the material is only reflective or (3) the material is diffuse. For simplicity only the last case is being taken into account. For the actual shading a local illumination method is used: *phong shading*.

The phong illumination model describes (reflected) light intensity I as a composition of the ambient, the diffuse and the perfect specular reflection of a surface.

$$I(\vec{V}) = k_a \cdot L_a + k_d \sum_{i=0}^{n-1} L_i \cdot (\vec{S_i} \cdot \vec{N}) + k_s \sum_{i=0}^{n-1} L_i \cdot (\vec{R_i} \cdot \vec{V})^{k_e} \tag{2}$$

Figure 10: The phong illumination model as defined by Phong Bui-Tuong. Note that the emissive term was left out intentionally as it is mainly used to achieve special effects.

Methodologies

The previous chapter provided the fundamentals that are required for understanding the results of this thesis.

This chapter presents the methodologies that are used to implement this thesis.

The first section of this chapter shows a principle called literate programming, which is used to generate this documentation and the practical implementation in terms of a software. The second section describes the agile methodologies, that are used to implement this thesis.

Literate programming

Software may be documented in different ways. It may be in terms of a preceding documentation, e.g. in form of a software architecture, which describes the software conceptually and hints at its implementation. It may be in terms of documenting the software inline through inline comments. Frequently both methodologies are used, in independent order. However, all too frequently the documentation is not done properly and is even neglected as it can be quite costly with seemingly little benefit.

Documenting software is crucial. Whenever software is written, decisions are made. In the moment a decision is made, it may seem intuitively clear as it evolved by thought. This seemingly clearness of the decision is most of the time deceptive. Is a decision still clear when some time has passed by since making that decision? What were the facts that led to it? Is the decision also clear for other, may be less involved persons? All these concerns show that documenting software is crucial. No documentation at all, outdated or irrelevant documentation can lead to unforeseen efforts concerning time and costs.

Hoare states 1973 in his work *Hints on Programming Language Design* that "documentation must be regarded as an integral part of the process of design and coding" [11, p. 195]: "The purpose of program documentation is to explain to a human reader the way in which a program works so that it can be successfully adapted after it goes into service, to meet the changing requirements of its users, or to improve it in the light of increased knowledge, or just to remove latent errors and oversights. The view that documentation is something that is added to a program after it has been commissioned seems to be wrong in principle and counter-productive in practice. Instead, documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear self-documenting code, and even perhaps to develop and display a pleasant style of writing. The readability of programs is immeasurably more important than their writeability." [11, p. 195]

LITERATE PROGRAMMING, a paradigm proposed in 1984 by Knuth, goes even further. Knuth believes that "significantly better documentation of programs" can be best achieved "by considering programs to be works of literature" [12, p. 1]. Knuth proposes to change the "traditional attitude to the construction of programs" [12, p. 1]. Instead of imagining that the main task is to instruct a computer what to do, one shall concentrate on explaining to human beings what the computer shall do. [12, p. 1]

THE IDEAS OF LITERATE PROGRAMMING have been embodied in several software systems, the first being *WEB*, introduced by Knuth himself. These systems are a combination of two languages: (1) a document formatting language and (2) a programming language. Such a software system uses a single document as input (which can be split up in multiple files) and generates two outputs: (1) a document in a document formatting language, such as LaTeX which, may then be converted in a platform independent binary description, such as PDF. (2) a compilable program in a programming language, such as Python or C which may then be compiled into an executable program. [12] The first is called *weaving* and the latter *tangling*. This process is illustrated in Figure 11.

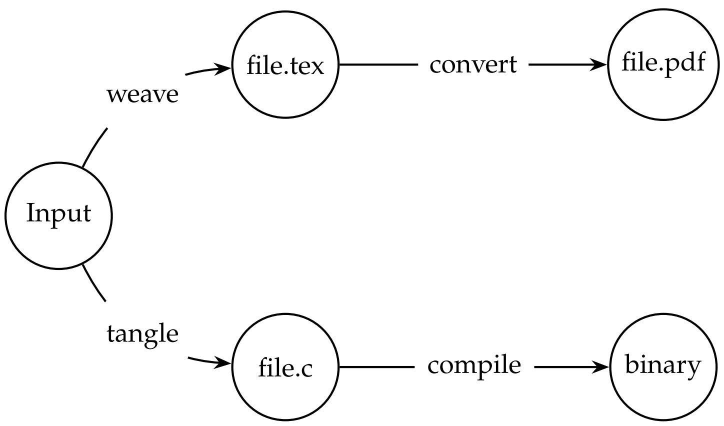


Figure 11: Illustration showing the processes of *weaving* and *tangling* documents from a input document. [12]

SEVERAL LITERATE PROGRAMMING SYSTEMS WERE EVALUATED during the first phases of this thesis: CWEB [1], Noweb [2], lit [3], PyLiter-

[1] http://www-cs-faculty.stanford.edu/~uno/cweb.html
[2] https://www.cs.tufts.edu/~nr/noweb/
[3] http://cdosborn.github.io/lit/lit/root.html

ate [4], pyWeb [5] and Babel [6] (which is part of org mode of Emacs). All of these tools have their strengths and weaknesses. However, none of these systems fulfill all the needed requirements: (1) Provide pretty printing of the program parts. (2) Provide automatic references between the definition of program parts and their usage. (3) Expand program parts having the same name instead of redefining them. (4) Support Python as programming language. (5) Allow the inclusion of files for both parts, the document formatting language and the programming language.

[4] https://github.com/bslatkin/pyliterate
[5] http://pywebtool.sourceforge.net/
[6] http://orgmode.org/worg/org-contrib/babel/

Ultimately nuweb [7] was chosen as it fulfills all these requirements. It has adapted and simplified the ideas FunnelWeb [8]. It is independent of the programming language for the source code. As document formatting language it uses LaTeX. Although the documentation of nuweb states, that it has no pretty printing of source code, it provides an option to display source code as listings. This method was modified to support visualizing the expansion of parts as well as to use specific syntax highlighting and code output within LaTeX.

[7] http://nuweb.sourceforge.net/
[8] http://www.ross.net/funnelweb/

nuweb provides several commands to process files. All commands begin with an at sign (@). Whenever a file does not contain any commands the file is copied unprocessed. The same applies for parts of files which contain no commands. nuweb provides a single binary, which processes the input files and generates the output files (in document formatting language and as source code respectively). The commands are used to *specify output files*, *define fragments* and to *delimit scraps*.

Command	Description
`@o file-name flags scrap`	*Outputs* the given scrap to the defined *file* using the provided flags.
`@d fragment-name scrap`	Defines a *fragment* which refers to / holds the given scrap.
`@q fragment-name scrap`	Defines a *quoted fragment* which refers to / holds the given scrap. Inside a quoted fragment referred fragments are not expanded.

Table 8: The major commands of nuweb. [13, p. 3]

Note that fragment names may be abbreviated, either during invocation or definition. nuweb simply preserves the longest version of a fragment name. [13, p. 4]

Scraps define content in form of source code. They "have specific markers to allow precise control over the contents and layout." [13] There are three ways of defining scraps, which can be seen in Table 9. They all include everything between the specific markers but they differ when being typeset.

Scrap	Typesetting
`@{ Content of scrap here @}`	Verbatim mode.
`@[ Content of scrap here @]`	Paragraph mode.
`@( Content of scrap here @)`	Math mode.

Table 9: Ways of defining scraps in nuweb. [13, p. 3]

A FRAGMENT IS BEING INVOKED by using `@<fragment-name@>`. “It causes the fragment fragment-name to be expanded inline as the code is written out to a file. It is an error to specify recursive fragment invocations.” [13, p. 3] There are various other commands and details, but mentioning them would go beyond the scope of this thesis. They can be found at [13].

LITERATE PROGRAMMING CAN BE VERY EXPRESSIVE as all thoughts are laid down before implementing something. Knuth sees this expressiveness an advantage as one is forced to clarify his thoughts before programming [12, p. 13]. This is surely very true for rather small software and partly also for larger software. The problem with larger software is, that using literate programming, the documentation tends to be rather large too. *To overcome this aspect* the actual implementation of the intended software is moved to the appendix .

insert reference to appendix here

ANOTHER PROBLEMATIC ASPECT IS THE IMPLEMENTATION OF TECHNICAL DETAILS such as imports for example or plain getter and setter methods, which may recur and may often be very similar. While this might be interesting for software developers or technically oriented readers, who want to grasp all the details, this might not be interesting for other readers. *This aspect can be overcome*, by moving recurring or seemingly uninteresting parts to a separate file, see , which holds these code fragments.

add reference to code fragments

TO SHOW THE PRINCIPLES OF LITERATE PROGRAMMING nevertheless, without annoying the reader, only an excerpt of some details is given at this place. One of the more interesting things of the intended software might be the definition of a node and the loading of node definitions from external files. These two aspects are shown below. However, not all of the details are shown as this would go beyond the scope.

add reference to the node concept within appendix

SOME ESSENTIAL THOUGHTS ABOUT CLASSES AND OBJECTS may help to stay consistent when developing the software, before implementing the node class. Each class should at least have

(1) Signals — to inform other components about events.

(2) A constructor.

(3) Various methods.

(4) Slots — to get informed about events from other components.

This pattern is applied to the declaration of the node class.

Implementing the node class means simply defining a *scrap* called "*Node definition declaration*" using the above pattern. The *scrap* does not have any content at the moment, except references to other scraps, which build the body of the scrap and which will be defined later on.

⟨*Node definition declaration ?*⟩ ≡

```
class NodeDefinition(object):
    """Represents a definition of a node."""

    # Signals
    ⟨ Node definition signals  ?⟩
    ⟨ Node definition constructor  ?⟩
    ⟨ Node definition methods  ?⟩

    # Slots
    ⟨ Node definition methods  ?⟩◇
```

Figure 12: Declaration of the node definition class.

Fragment never referenced.

The constructor might be the first thing to implement, following the developed pattern. In Python the constructor defines the properties of a class [9], therefore it defines what a class actually is or represents — the concept. After some thinking, and in context of the intended software, one might come up with the properties in Table 10 defining a node definition.

[9] Properties do not need to be defined in the constructor, they may be defined anywhere within the class. However, this can lead to confusion and it is therefore considered as good practice to define the properties of a class in its constructor.

Property	**Description**
ID	A globally unique identifier for the node definition.
Name	The name of the definition.
Description	The description of the definition. What does that definition provide?
Parent	The parent object of the current node definition.
Inputs	Inputs of the node definition. This may be distinct types or references to other nodes.
Outputs	The same as for inputs.
Invocation	A list of the node's invocations or calls respectively.
Parts	Defines parts that may be processed when evaluating the node. Contains code which can be interpreted directly.
Connections	A list of connections of the node's inputs and outputs. Each connection is composed by two parts: (1) a reference to another node and (2) 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.
Instances	A list of node instances from a certain node definition.
Was changed	Flag, which indicates whether a definition was changed or not.

Table 10: Properties/attributes of the node class.

Implementing the constructor of the node definition may now follow from the properties defined in Table 10. As the name of the constructor definition was already given, by using it within Figure 12

(`@<Node definition constructor@>`), the very same name will be used for actually defining the scrap itself.

⟨*Node definition constructor ?*⟩ ≡

```python
def __init__(self, id_):
    """Constructor.

    :param id_: the globally unique identifier of the node.
    :type  id_: uuid.uuid4
    """

    self.id_          = id_

    self.name         = ""
    self.description = ""
    self.parent       = None
    self.inputs       = []
    self.outputs      = []
    self.invocations = []
    self.parts        = []
    self.nodes        = []
    self.connections = []
    self.instances    = []
    self.was_changed = False◇
```

Figure 13: Constructor of the node definition class. Note that the identifier is given by a corresponding parameter. Identifiers have to be generated when defining a node using an external file.

Fragment referenced in ?.

One of the problems mentioned before can be seen in fig. 13: it shows a rather dull constructor without any logic which is not interesting. Additionally importing of modules would be needed, e.g. PyQt or system modules. This was left out deliberately. At this point the implementation of node definitions will not be shown further, as this is beyond scope. Further implementation can be seen at .

insert reference(s) to node domain model here

Node definitions will be loaded from external files in JSON format. This happens within the node controller component, which will not be shown here as this would go beyond the scope. Required attributes will be mentioned explicitly although. The method for loading the nodes, `load_node_definitions`, defined in fig. 14, does not have any arguments. Everything needed for loading nodes is encapsulated in the node controller.

When loading the node definitions, there are two cases (and consequences) at the first instance: (1) the directory containing the node definitions exists, the load definitions may be loaded or (2) the directory does not exit. In the first case the directory possibly containing node definitions is being searched for such files, in the second case a warning message is logged.

In the first case, when the directory containing the node definitions exists, files containing node definitions are searched. The files are searched by wildcard pattern matching the extension: `*.node`.

⟨*Load node definitions ?*⟩ ≡

```
def load_node_definitions(self):
    """Loads all files with the ending NODES_EXTENSION
    within the NODES_PATH directory, relative to
    the current working directory.
    """
◇
```

Figure 14: Head of the method that loads node definitions from external JSON files.

Fragment defined by ?, ?.
Fragment never referenced.

⟨*Load node definitions ?*⟩+ ≡

```
if os.path.exists(self.nodes_path):
    ⟨ Find and load node definition files  ?, ... ⟩
else:
    ⟨ Output warning when directory with node definitions does not exist  ?⟩◇
```

Figure 15: Check whether the path containing the node definition files exist or not.

Fragment defined by ?, ?.
Fragment never referenced.

⟨*Find and load node definition files ?*⟩ ≡

```
node_definition_files = glob.glob("{path}{sep}*.{ext}".format(
    path=self.nodes_path,
    sep=os.sep,
    ext=self.nodes_extension
))
num_node_definitions = len(node_definition_files)◇
```

Figure 16: When the directory containing the node definitions exists, files matching the pattern **.node* are searched.

Fragment defined by ?, ?.
Fragment referenced in ?.

Having searched for node definition files, there are again two cases, similar as before: (1) files (possibly) containing node definitions exist or (2) no files with the ending `.node` exist within the source directory. Again, as before, in the first case the node definitions will be loaded, in the second case a warning message will be logged.

Given that node definitions are present, they are loaded from the file system, parsed and then stored internally as domain model. To maintain readability, all this is encapsulated in a method, `load_node_definition_from_file_name`, which is deliberately not shown here as this would go beyond scope. If the node definition cannot be loaded or parsed `None` is being returned.

⟨*Find and load node definition files* ?⟩+ ≡

```
if num_node_definitions > 0:
    ⟨ Load found node definitions ?, ... ⟩
else:
    ⟨ Output warning when no node definitions are found ?⟩◇
```

Figure 17: When files (possibly) containing node definition files are found, they are tried being loaded. When no such files are found, a warning message is being logged.

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨*Load found node definitions* ?⟩ ≡

```
self.logger.info(
    "Found %d node definition(s), loading.",
    num_node_definitions
)
t0 = time.perf_counter()
for file_name in node_definition_files:
    self.logger.debug(
        "Found node definition %s, trying to load",
        file_name
    )
    node_definition = self.load_node_definition_from_file_name(
        file_name
    )◇
```

Figure 18: Loading and parsing of the node definitions found within the folder containing (possibly) node definition files. If a node definition cannot be loaded or parsed, *None* is being returned.

Fragment defined by ?, ?.
Fragment referenced in ?.

When a node definition could be loaded, a view model based on the domain model is being created. Both models are then stored internally and a signal about the loaded node definition is being emitted, to inform other components which are interested in this event.

⟨*Load found node definitions* ?⟩+ ≡

```
    if node_definition is not None:
        node_definition_view_model = node_view_model.NodeViewModel(
            id_=node_definition.id_,
            domain_object=node_definition
        )
        self.node_definitions[node_definition.id_] = (
            node_definition,
            node_definition_view_model
        )
        ⟨ Node controller load node definition emit  ?⟩

t1 = time.perf_counter()
self.logger.info(
    "Loading node definitions took %.10f seconds",
    (t1 - t0)
)◇
```

Figure 19: A view model, based on the domain model, for the node definition is being created. Both models are then stored internally and the signal, that a node definition was loaded is being emitted.

Fragment defined by ?, ?.
Fragment referenced in ?.

The implementation of the edge cases is still remaining at this point. When such an edge case happens, a corresponding message is logged. The edge cases are:

(1) the directory holding the node definitions does not exist

⟨*Output warning when directory with node definitions does not exist* ?⟩ ≡

```
    message = QtCore.QCoreApplication.translate(
        __class__.__name__,
        "The directory holding the node definitions, %s, does not exist." % self.nodes_path
    )
    self.logger.fatal(message)◇
```

Figure 20: Output a warning when the path containing the node definition files does not exist.

Fragment referenced in ?.

or

(2) no files containing node definitions are found.

Agile software development

Software engineering invokes always a methodology, be it wittingly or unwittingly. For a (very) small project the methodology may follow intuitively, by experience and it may be a mixture

⟨*Output warning when no node definitions are found ?*⟩ ≡

```
    message = QtCore.QCoreApplication.translate(
        __class__.__name__,
        "No files with node definitions found at %s." % self.nodes_path
    )
    self.logger.fatal(message)◇
```

Figure 21: Output a warning when no node definitions are being found.

Fragment referenced in ?.

of methodologies. For medium to large projects however, using certain methodologies or principles becomes inevitable for being able to evaluate (the success of) a project.

Every commonly used software engineering methodology has advantages but buries also certain risks. Be it a traditional method like the waterfall model, incremental development, the v-model, the spiral model or a more recent method like agile development. It depends largely on the project what methodology fits best and buries the least risks. [14], [15]

Risk is the basic problem of software development. [16] Examples of risks are: schedule slips, canceled projects, increased defect rates, misunderstood domain/business, changes, false feature rich. [16]

Traditional software engineering methodologies, such as the waterfall model or incremental development, struggle with change. In case of the waterfall model they embrace change not at all or, in the case of incremental development, the phases are rather long what allows only slow reaction.

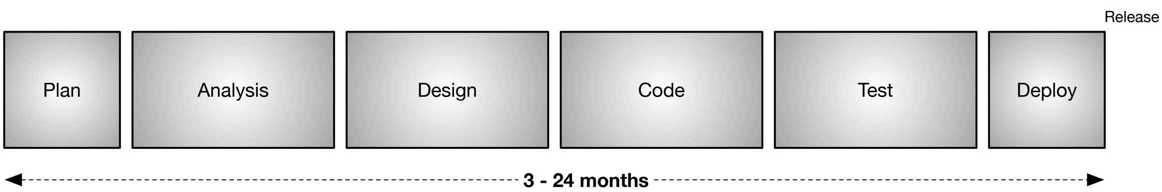


Figure 22: Phases of the water fall methodology. [17, p. 16]

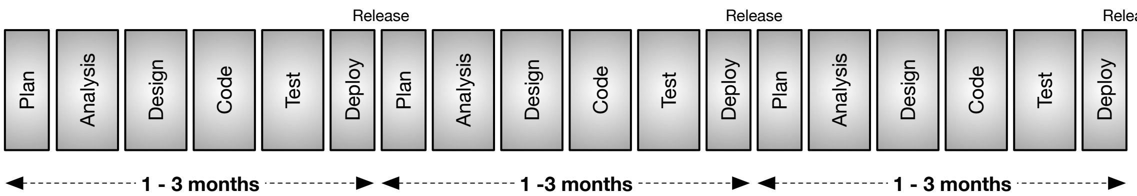


Figure 23: Phases of iterative development. [17, p. 16]

By applying basic principles, agile development methodologies try to overcome this problem. These principles may vary depending on the used methodology, but the fundamental principles are: (1) rapid feedback, (2) assume simplicity, (3) incremental change, (4) embracing change and (5) quality work. [16] Further details can be found at [16], [17].

An adapted version of extreme programming is used for this thesis. This methodology was chosen as after the preceding project work, *QDE - a visual animation system. Software-Architektur.* several things were still subject to change and therefore an exact planning, analysis and design, as traditional methodologies require it, would not be very practical.

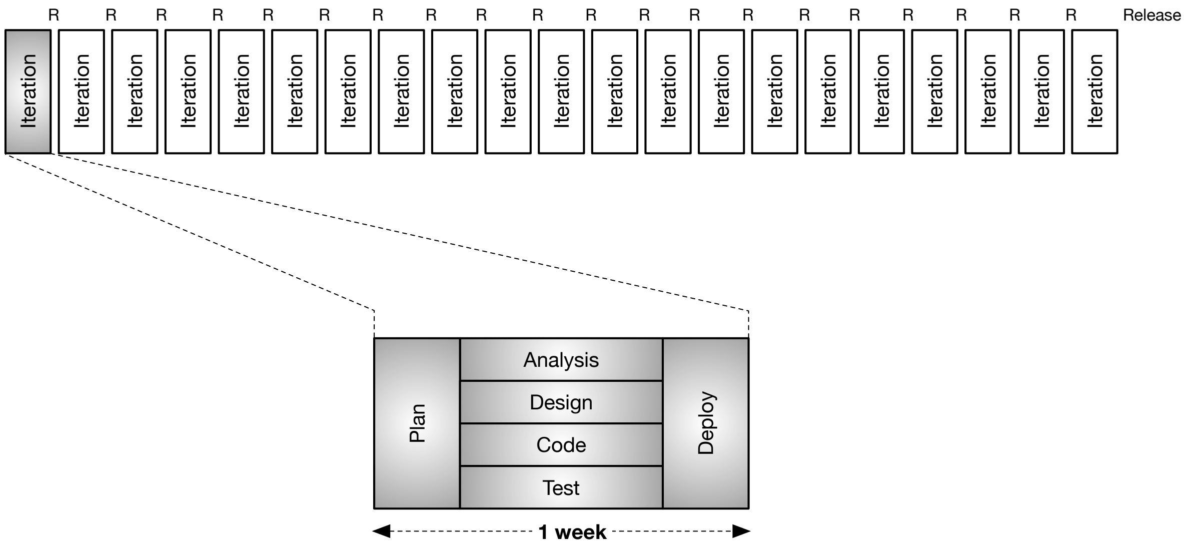


Figure 24: Iterations in the extreme programming methodology and phases of an interation. [17, p. 18]

Implementation

THE PREVIOUS CHAPTER introduced the methodologies that are required for understanding the following results of this thesis.

THIS CHAPTER presents the achieved results by means of three sections. The first section shows the software architecture, that was developed and that is used for the developed software. Aspects of the developed literate program are shown in the second section. The main concepts and the components of the developed software are shown in the third section.

Software architecture

THE SOFTWARE ARCHITECTURE holds the significant decisions of the envisaged software, the selection of structural elements, their behavior and their interfaces. [6] It is derived from the experiences based on the former project works, *Volume ray casting - basics & principles* and *QDE - a visual animation system. Software-Architektur.* which are condensed to build the fundamentals, see .

THREE ASPECTS define the software architecture: (1) an architectural software design pattern, (2) layers and (3) signals, allowing communication between components.

Software design

A [SOFTWARE] DESIGN PATTERN "names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use." [18, p. 16]

TO SEPARATE DATA FROM ITS REPRESENTATION and to ensure a coherent design, a combination of the model-view-controller (MVC) and the model-view-view model pattern (MVVM) is used as architectural software design pattern. [19], [20] This decision is based on experiences from the previous project works and allows to modify and reuse individual parts. This is especially necessary as the data created in the editor component will be reused by the player component.

Four kinds of components build the basis of the used pattern. table 11 provides a description of the components. fig. 25 shows an overview of the components (the colored items) including their communication. Additionally the user as well as the display is shown (in gray color).

Component	Description	Examples
Model	Represents the data or the business logic, completely independent from the user interface. It stores the state and does the processing of the problem domain.	Scene, Node Parameter
View	Consists of the visual elements.	Scene graph view, Scene view
View model	"Model of a view", abstraction of the view, provides a specialization of the model that the view can use for data-binding, stores the state and may provide complex operations.	Scene graph view model, Scene view model, Node view model
Controller	Holds the data in terms of models. models. Acts as an interface between the components.	Scene graph controller, scene controller, node controller

Table 11: Description of the components of the used software design pattern [10], [20]

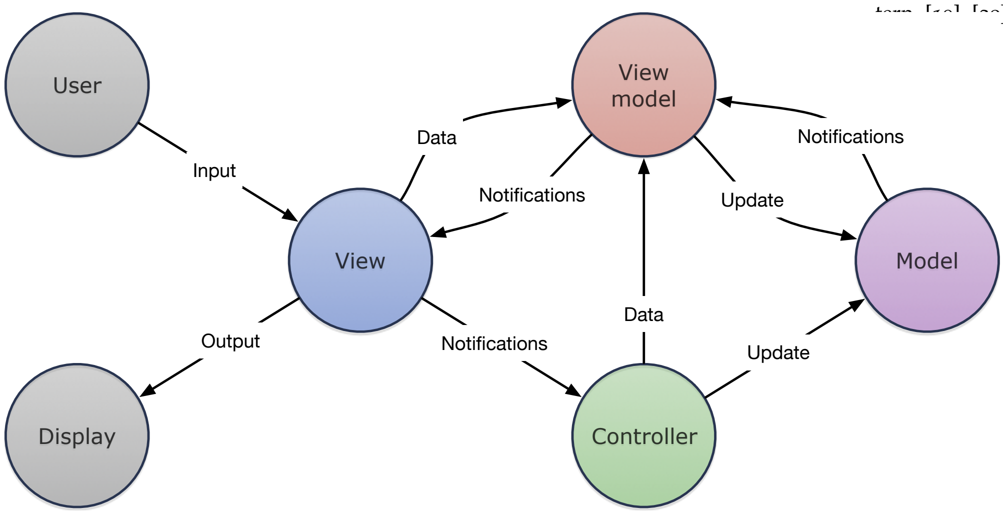


Figure 25: Components of the used pattern and their communication.

The used Qt framework provides a very similar pattern respectively concept called "model/view pattern". It combines the view and the controller into a single object. The pattern introduces a delegate between view and model, similar to a view model. The delegate allows editing the model and communicates with the view. The communication is done by so called model indices coming from the model. Model indices are references to items of data. [21] "By supplying model indexes to the model, the view can retrieve items of data from the data source. In standard views, a delegate renders the items of data. When an item is edited, the delegate communicates with the model directly using model indexes." [21] fig. 26 shows the model/view pattern.

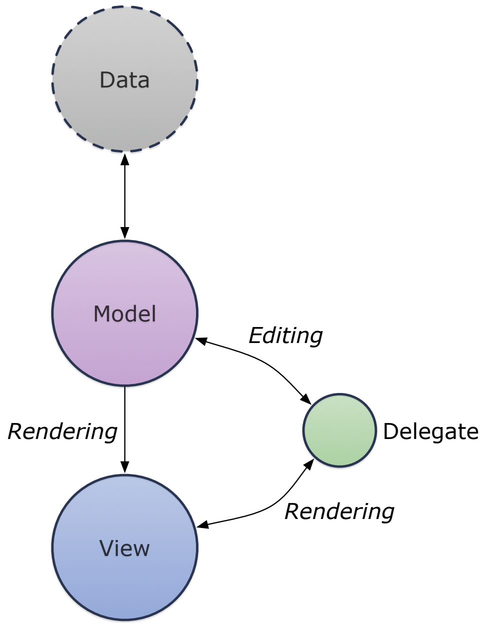


Figure 26: Qt's model/view pattern. [21]

Although offering advantages, such as to customize the presentation of items or the usage of a wide range of data sources, the model/view pattern was not used in general. This is mainly due to two reasons: (1) the developed and intended components use no data source except external files and (2) the concept of using model indices may add flexibility but introduces also overhead. The scene graph component of the editor was developed using Qt's abstract item model class which uses the model/view pattern. This showed, that the usage of the pattern can introduce unnecessary overhead, in terms of being more effort to implement, while not using the features of the pattern. Therefore the decision was taken against the usage of the pattern.

Layers

To reduce coupling and dependencies a relaxed layered architecture is used, as written in **??** . In contrast to a strict layered architecture, which allows any layer calling only services or interfaces from the layer below, the relaxed layered architecture allows higher layers to communicate with any lower layer. table 12 provides a graphical overview as well as a description of the layers. The colors have no meaning except to distinguish the layers visually.

Layer	Description	Examples
Graphical user interface (GUI)	All elements of the graphical user interface, views.	Scene graph view, scene view, render view
Graphical user interface domain (GUI domain)	View models.	Scene graph view model, node view model
Application	Controller/workflow objects.	Main application, scene graph controller, scene controller, node controller
Domain	Models respectively logic of the application.	Scene model, parameter model, node definition model, node domain model
Technical	Technical infrastructure, such as graphics, window creation and so on.	JSON parser, camera, culling, graphics, renderer
Foundation	Basic elements and low level services, such as a timer, arrays or other data classes.	Colors, common, constants, flags

Table 12: Layers of the developed software.

Signals and slots

Whenever designing and developing software, coupling and cohesion can occur and may pose a problem if not considered early enough and properly. *Coupling* measures how strongly a component is connected, has knowledge of or depends on other components. High coupling impedes the readability and maintainability of software. Therefore low coupling ought to be strived. Larman states, that the principle of low coupling applies to many dimensions of software development and that it is one of the cardinal goals in building software. [8] *Cohesion* is a measurement of "how functionally related the operations of a software element are, and also measures how much work a software element is doing". [8] Or put otherwise, "a measure of the strength of association of the elements within a module". [22, p. 52] Low (or bad) cohesion does not imply, that a component does work only by itself, indeed it probably collaborates with many other objects. But low cohesion tends to create high (bad) coupling. It is therefore strived to keep objects focused, understandable and manageable while supporting low coupling. [8]

To overcome the problems of high coupling and low cohesion *signals and slots* are used. Signals and slots are a generalized implementation of the observer pattern, which can be seen in fig. 27.

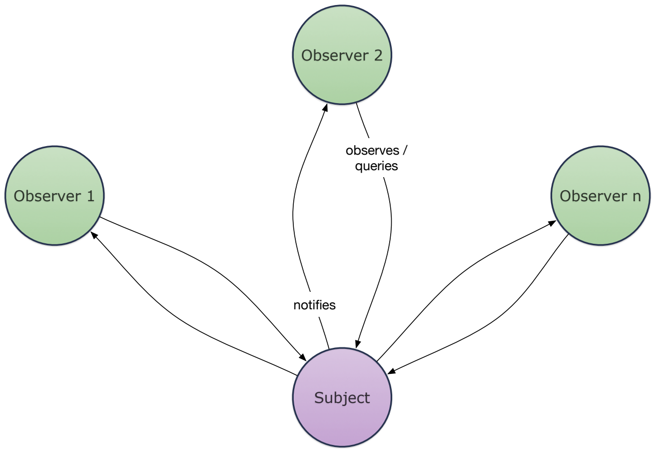


Figure 27: The observer pattern. [18]

A signal is an observable event. A slot is a potential observer, typically a function. Slots are registered as observers to signals. Whenever a signal is emitted, the emitting class must call all the registered observers for that signal. Signals an slots have a many-to-many relationship. One signal may be connected to any number of slots and a slot may listen to any number of signals.

Figure 28: An example of an observer being registered to a signal.

```
sender     = Sender()
observer_1 = Observer()

sender.emit_some_signal.connect(
    observer_1.some_slot
)
```

Signals can hold additional information, such as single values or even references to objects. A simple example is loading nodes from files containing node definitions. The node controller, which loads node definitions from the file system, could emit two signals to inform other components, for example components of the GUI layer. (1) The total amount of node definitions to load and (2) the index of the last loaded node definition including a reference to the node definition This information could for example be used by a dialog showing the progress of loading node definitions from the file system.

```python
self.total_node_definitions.emit(num_node_definitions)
```

Figure 29: An example of emitting a signal including a value.

```python
for index, definition_file in enumerate(node_definition_files):
    node_definition = self.load_node_definition_from_file(
        definition_file
    )
    self.node_definition_loaded(index, node_definition)
```

Figure 30: An example of emitting a signal including a value and a reference to an object.

Literate programming

Documentation is crucial to any software project. However, all too frequently the documentation is not done properly or is even neglected as it can be quite effortful with seemingly little benefit. No documentation at all, outdated or irrelevant documentation can cause unforeseen cost- and time-wise efforts. Using the literate programming paradigm prevents these problems, as the software emanates from the documentation. For this thesis literate programming was used as described in .

Mention usage of nuweb here, again?

Another train of thought is required when using literate programming to develop software than when using traditional methodologies. This is due to the fact, that the approach is completely different. Traditional methodologies focus on instructing the computer what to do by writing program code. Literate programming focuses on explaining to human beings what the computer shall do by combining the documentation with code fragments in a single document. From this single document a program which can be compiled or run directly is extracted. The order of the code fragments matters only indirectly. They may appear in any order throughout the text. The code fragments are put into the right order for being compiled or run by defining the output files containing the needed code fragments in the right order.

The need to include every detail makes literate programming very expressive and verbose. While this expressiveness may be an advantage for small software and partly also for larger software, it can also be a problem, especially for larger software: the documentation can get lengthy and hard to read, especially when including the implementation of technical details.

These aspects were overcome by moving the implementation into the appendix and by outsourcing similar and very technical parts

Insert reference to appendix here.

and output file definitions into a separate file.

Insert reference to code fragments here.

Software

USING THE INTRODUCED METHODOLOGIES (see) and the developed software architecture (see) the intended software was developed.

Discussion and conclusion

Write chapter.

Appendix

Implementation

To begin with 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.

Adapt goal to current state.

The thoughts to reach this goal were already developed in Fundamentals and Methodologies and will therefore not be repeated again.

As stated in Methodologies, the literate programming paradigm 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 **??**.

The editor component is described first 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.

Requirements

The requirements for running the implementation are currently the following:

- A Unix derivative as operating system (Linux, macOS).
- Python [10] version 3.5.x or above
- PyQt5 [11] version 5.7 or above
- OpenGL [12] version 3.3 or above

[10] `http://www.python.org`

[11] `https://riverbankcomputing.com/software/pyqt/intro`

[12] `https://www.opengl.org/`

Name spaces and project structure

To provide a structure for the whole project and for being able to stick to the thoughts established in Fundamentals and Methodologies, it may be wise to structure the project a certain way.

The 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.[13]

[13] https://docs.python.org/3/reference/import.html#pa

To prevent having multiple modules having the same name, name spaces are used.[14] 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.

[14] https://docs.python.org/3/tutorial/classes.html#pyth scopes-and-namespaces

To 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.

Coding style

To stay consistent throughout 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

For the implementation python is used, as mentioned in section "Requirements". 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. [15]. As defined by the coding style, **??**, imports are done by the `from Foo import Bar` syntax.

[15] `https://www.python.org/dev/peps/pep-0020/`

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

To stay consistent when implementing classes an methods, it makes sense to define a rough framework for their implementation, which is as follows: (1) Define necessary signals, (2) define the constructor and (3) implement the remaining functionality in terms of methods and slots. Concerning the constructor, the following pattern may be applied: (1) Set up the user interface when it is a class concerning the graphical user interface, (2) set up class-specific aspects, such as the name, the tile or an icon, (3) set up other components used by that class and (4) initialize the connections, meaning hooking up the defined signals with corresponding methods.

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

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 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. As this definition is rather abstract, it may be easier to define what shall be achieved in terms of content and then work towards this definition.

A very basic definition of what shall be achieved is the following. It shall be possible to create an animated scene using the editor application. The scene shall be composed of two objects, a sphere and a cube. Additionally it shall have a camera as well as a point light.

The camera shall be placed 5 units in height and 10 units in front of the center of the scene. The cube shall be placed in the middle of the scene, the sphere shall have an offset of 5 units to the right and 2 units in depth. The point light shall be placed 10 units above the center.

Both objects shall have different materials: the cube shall have a dull surface of any color whereas the sphere shall have a glossy surface of any color.

There shall be an animation of ten seconds duration. During this animation the sphere shall move towards the cube and they shall merge into a blob-like object. The camera shall move 5 units towards the two objects during this time.

Scene: Composition of nodes. Define scene already here.

To achieve this overall goal, while providing an user-friendly experience, several components are needed. These are the following, being defined in *QDE - a visual animation system. Software-Architektur.* pp. 29 ff. [5]

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).

THE MAIN COMPONENT is composed of a view and a controller, as the whole architecture uses layers and the MVVMC principle, see section "Software architecture". 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*.

TO PRESERVE CLARITY all components described in discrete chapters. Although the implementation of the components is very specific, in terms of the programming language, their logic may be reused later on when developing the player component.

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

Main entry point

AN ENTRY POINT is a point where an application starts when being called. Python does this by evaluating a special variable within a module, called `__name__+. Its value is set to~\verb+'__main__'` if the module is "read from standard input, a script, or from an interactive prompt." [16]

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

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.

BUT WHERE TO PLACE THE MAIN 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

⟨*Main entry point ?*⟩ ≡

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

Figure 31: Main entry point of the editor application.

Editor → Main entry point

Fragment never referenced.

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~\verb+src` directory.

Main application

THE EDITOR APPLICATION CANNOT BE STARTED YET, although a main entry point is defined by now. This is due the fact that there is no such thing as an editor application yet. Therefore a main application needs to implemented.

QT VERSION 5 IS USED through the PyQt5 wrapper, as stated in the section "Requirements". 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 [17].

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

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.[5, pp. 37 — 38]

Therefore it needs to do at least three things: (1) initialize itself, (2) set up components and (3) connect components. This all happens when the main application is being initialized through its constructor.

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.

The other two steps, setting up the components and connecting them can however not be done at this point, as there simply are

⟨*Main application declarations* ?⟩ ≡

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

    ⟨ Main application constructor  ?⟩
    ⟨ Main application methods  ?⟩◇
```

Figure 32: Main application class of the editor application.

Editor → Application

Fragment never referenced.

⟨*Main application constructor* ?⟩ ≡

```
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  ?, ... ⟩
    ⟨ Set up components for main application  ?⟩
    ⟨ Add root node for main application  ?⟩
    ⟨ Set model for scene graph view  ?⟩
    ⟨ Load nodes  ?⟩
    self.main_window.show()◇
```

Figure 33: Constructor of the editor application class.

Editor → Application → Constructor

Fragment referenced in ?.

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.

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

⟨ *Set up internals for main application ?* ⟩ ≡

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

Fragment defined by ?, ?.
Fragment referenced in ?.

Figure 34: Setting up the internals for the main application class.

Editor → Application → Constructor

⟨ *Main window declarations ?* ⟩ ≡

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

    ⟨ Main window signals  ?⟩

    ⟨ Main window methods  ?, ... ⟩
◇
```

Fragment never referenced.

Figure 35: Main window class of the editor application.

Editor → Main window

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 ?* ⟩ ≡

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

Fragment referenced in ?.

Figure 36: Definition of the `do_close` signal of the main window class.

Editor → Main window → Signals

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 ?* ⟩ ≡

```
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)
◇
```

Figure 37: Definition of methods for the main window class.

Editor → Main window → Methods

Fragment defined by ?, ?.
Fragment referenced in ?.

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 ?* ⟩ ≡

```
⟨ Set up controllers for main application  ?, ... ⟩
⟨ Connect controllers for main application  ?, ... ⟩
⟨ Set up main window for main application  ?⟩◇
```

Figure 38: Setting up of components for the main application class.

Editor → Main application → Constructor

Fragment referenced in ?.

The used view component for the main window, `QMainWindow`, needs at least a central widget with a layout for being rendered. [18]

[18] 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. The method creates a central widget containing a grid layout.

Targeting a look as proposed in *QDE - a visual animation system. Software-Architektur.* p. 9, a simple grid layout does however

⟨ *Set up main window for main application ?* ⟩ ≡

```
self.main_window = qde_main_window.MainWindow()
self.main_window.do_close.connect(self.quit)
⟨ Connect main window components  ?, ... ⟩◇
```

Figure 39: Set up of the editor main window and its signals from within the main application.

Editor → Main application → Constructor

Fragment referenced in ?.

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.

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.

A GOOD STARTING POINT for the implementation of the remaining components might be the scene graph, as it might be the most straight-forward component to implement.

⟨*Main window methods* ?⟩+ ≡

```python
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_view = guiscene.SceneGraphView()
    self.scene_graph_view.setObjectName('scene_graph_view')
    self.scene_graph_view.setMaximumWidth(300)
    horizontal_layout.addWidget(self.scene_graph_view)

    ⟨ Set up scene 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 scene view to vertical splitter in main window  ?⟩

    horizontal_layout.addWidget(vertical_splitter)
◇
```

Fragment defined by ?, ?.
Fragment referenced in ?.

Figure 40: Set up of the user interface of the editor's main window.

Editor → Main window → Methods

Scene graph

The scene graph component has two aspects to consider, as mentioned in chapter "Editor": (1) a graphical aspect as well as (2) its data structure.

Define what a scene is by prose and code.

As described in subsection "Software design", 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 ?* ⟩ ≡

```
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  ?⟩
    ⟨ Scene model slots  ?⟩◇
```

Figure 41: Definition of the scene model class, which acts as a base class for scene instances within the whole application.

Editor → Scene model

Fragment never referenced.

The only known fact at this point is, that a scene is a composition of nodes and therefore holds its nodes as a list. Additionally it holds a reference to its parent.

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 this are actually two components: the scene graph view as well as the scene view.

⟨ *Scene model methods ?* ⟩ ≡

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

    :param parent: the parent scene of this scene. The parent is
                   None if the current scene is the root scene.
    :type parent:  SceneModel
    """

    self.id_ = uuid.uuid4()
    self.nodes = []
    self.parent = parent◇
```

Figure 42: The constructor of the scene model.

Editor → Scene model → Constructor

Fragment referenced in ?.

It would therefore make sense the use one view model for both components, but this is not possible as the view model of the scene view, `QGraphicsScene`, uses its own data model.

Therefore `QObject` will be used for the scene graph view model and `QGraphicsScene` will be used for the scene view model.

⟨ *Scene graph view model declarations ?* ⟩ ≡

```
class SceneGraphViewModel(Qt.QObject):
    """View model representing scene graph items.

    The SceneGraphViewModel 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 graph view model signals  ?⟩
    ⟨ Scene graph view model constructor  ?, ... ⟩
    ⟨ Scene graph view model methods  ?⟩
    ⟨ Scene graph view model slots  ?⟩
◇
```

Figure 43: Definition of the scene graph view model class, which corresponds to an entry within the scene graph.

Editor → Scene graph view model

Fragment never referenced.

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.

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

⟨ *Scene graph view model constructor ?* ⟩ ≡

```
def __init__(
        self,
        row,
        domain_object,
        name=QtCore.QCoreApplication.translate(
            'SceneGraphViewModel', '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.
                          SceneGraphViewModel
    """

    super(SceneGraphViewModel, self).__init__(parent)

    self.id_  = domain_object.id_
    self.row  = row
    self.domain_object = domain_object
    self.name = name
◇
```

Figure 44: The constructor of the scene graph view model.

Editor → Scene graph view model → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

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."[19]

[19] http://doc.qt.io/qt-5/qtreewidget.html#details

SUCH A STANDARD ITEM MODEL is `QAbstractItemModel`[20], which is used as a base class for the scene graph controller.

[20] http://doc.qt.io/qt-5/qabstractitemmodel.html

⟨ *Scene graph controller declarations ?* ⟩ ≡

```
common.with_logger
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  ?⟩
    ⟨ Scene graph controller constructor  ?, ... ⟩
    ⟨ Scene graph controller methods  ?, ... ⟩
    ⟨ Scene graph controller slots  ?⟩
◇
```

Figure 45: The scene graph controller, inheriting from `QAbstractItemModel`.

Editor → Scene graph controller

Fragment never referenced.

AS AT THIS POINT THE FUNCTIONALITY of the scene graph controller is not fully known, the constructor simply initializes its parent class and an empty list of scenes.

⟨ *Scene graph controller constructor ?* ⟩ ≡

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

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

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

Figure 46: Constructor of the scene graph controller.

Editor → Scene graph controller → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

THE SCENE GRAPH CONTROLLER HOLDS AND MANAGES SCENE DATA.

Therefore 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.

Due to the dependencies of other components this cannot be done within the constructor, as components depending on the scene graph controller may not be listening to its signals at this point. Therefore this is done in a separate method called `add_root_node`.

⟨ *Scene graph controller add root node ?* ⟩ ≡

```
def add_root_node(self):
    """Add a root node to the data structure.
    """

    if self.root_node is None:
        root_node = domain_scene.SceneModel()
        self.view_root_node = guidomain_scene.SceneGraphViewModel(
            row=0,
            domain_object=root_node,
            name=QtCore.QCoreApplication.translate(
                __class__.__name__, 'Root scene'
            )
        )
        self.do_add_scene.emit(root_node)
        self.layoutChanged.emit()
        self.logger.debug("Added root node")
    else:
        self.logger.warn((
            "Not (re-) adding root node, already"
            "present!"
        ))
◇
```

Figure 47: A method to add the root node from within the scene graph controller.

Editor → Scene graph controller → Methods

Fragment referenced in ?.

⟨ *Scene graph controller methods ?* ⟩ ≡

```
⟨ Scene graph controller add root node ?⟩
◇
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

The root scene can now be added by the main application, as all necessary components are set up.

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

Add reference to Qt's view model

⟨*Add root node for main application ?*⟩ ≡

```
self.scene_graph_controller.add_root_node()
◇
```

Figure 48: The root node of the scene graph being added by the main application.

Editor → Main application → Constructor

Fragment referenced in ?.

displayed.

⟨*Scene graph controller constructor ?*⟩+ ≡

```
    self.header_data = [
        QtCore.QCoreApplication.translate(
            __class__.__name__, 'Name'
        ),
        QtCore.QCoreApplication.translate(
            __class__.__name__, '# Nodes'
        )
    ]
    self.root_node = None
    self.view_root_node = None◇
```

Figure 49: Initialization of the header data and the root node of the scene graph.

Editor → Scene graph controller → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

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."[20]

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

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.

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.

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

The last method that has to be implemented due to the usage of `QAbstractItemModel`, is the `data` method. It returns the data for an

⟨*Scene graph controller methods ?*⟩+ ≡

```
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 not parent.isValid():
        self.logger.debug((
            "Getting index for row {0}, col {1}, root node"
        ).format(row, column))
        return self.createIndex(row, column, self.view_root_node)

    parent_node = parent.internalPointer()
    self.logger.debug((
        "Getting index for row {0}, col {1}, "
        "parent {2}. Children: {3}"
    ).format(
        row, column, parent_node, len(parent_node.children())
    ))
    child_nodes = parent_node.children()

    # It may happen, that the index is called at the same time as
    # a node is being deleted respectively was deleted. In this
    # case an invalid index is returned.
    try:
        child_node  = child_nodes[row]
        return self.createIndex(row, column, child_node)

    except IndexError:
        return QtCore.QModelIndex()◇
```

Figure 50: Implementation of QAbstractItemModel's index method for the scene graph controller.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Scene graph controller methods ?* ⟩+ ≡

```
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
    """

    # self.logger.debug("Getting parent")

    if not model_index.isValid():
        # self.logger.debug("No valid index for parent")
        return QtCore.QModelIndex()

    # The internal pointer of the the model index returns a
    # scene graph view model.
    node = model_index.internalPointer()
    if node and node.parent() is not None:
        # self.logger.debug("Index for parent")
        return self.createIndex(
            node.parent().row, 0, node.parent()
        )
    else:
        # self.logger.debug("Index for root")
        return QtCore.QModelIndex()
◇
```

Figure 51: Implementation of QAbstractItemModel's parent method for the scene graph controller.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨*Scene graph controller methods ?*⟩+ ≡

```
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
    """

    column_count = len(self.header_data) - 1
    self.logger.debug("Getting column count: %s", column_count)

    return column_count
◇
```

Figure 52: Implementation of QAbstractItemModel's columnCount method for the scene graph controller.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

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.

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."[21]

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

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 graph view model*.

The *scene graph 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. Therefore it is necessary, that the scene view model keeps track of its nodes separately.

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

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

⟨ *Scene graph controller methods ?* ⟩+ ≡

```
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():
        self.logger.debug("Parent is not valid")
        row_count = 1
    else:
        # Get the actual object stored by the parent. In this case
        # it is a SceneGraphViewModel.
        node = parent.internalPointer()

        if node is None:
            self.logger.debug("Parent (node) is not valid")
            row_count = 1
        else:
            row_count = len(node.children())

    self.logger.debug("Getting row count: %s", row_count)
    return row_count
◇
```

Figure 53: Implementation of QAbstractItemModel's rowCount method for the scene graph controller.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨*Scene graph controller methods ?*⟩+ ≡

```
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():
        self.logger.debug("Model index is not valid")
        return None

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

    if node is None:
        self.logger.debug("Node is not valid")
        return None

    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
◇
```

Figure 54: Implementation of QAbstractItemModel's data method for the scene graph controller.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨*Scene graph controller methods ?*⟩+ ≡

```
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]
◇
```

Figure 55: Implementation of QAbstractItemModel's headerData method for the scene graph controller.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨*Scene graph view model constructor ?*⟩+ ≡

```
    self.nodes = []
◇
```

Figure 56: Scene graph view models hold references to the nodes they contain.

Editor → Scene graph view model → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨ *Scene graph view model methods ?* ⟩ ≡

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

    return len(self.nodes)
◇
```

Figure 57: The number of (graphical) nodes which a scene graph view model contains implemented as a property.

Editor → Scene graph view model → Methods

Fragment referenced in ?.

⟨ *Set up controllers for main application ?* ⟩ ≡

```
self.scene_graph_controller = scene.SceneGraphController(self)◇
```

Figure 58: The scene graph controller gets initialized within the main application.

Editor → Main application → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

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." [22] Therefore `QTreeView` is used as basis for the scene graph view.

[22] fn:f3778826acb87691:http://doc.qt.io/qt-5/qtreeview.html#details

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

For being able to display something, 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. This model may although not be set too early, as otherwise problems arise. It may only then be added, when the depending components are properly initialized, e.g. when the root node has been added.

But scenes shall not only be displayed, instead it shall be possible to work with them. What shall be achieved, are three things: (1) Adding and removing scenes, (2) renaming scenes and (3) switching between scenes.

⟨ *Scene graph view declarations ?* ⟩ ≡

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

    ⟨ Scene graph view signals  ?, ... ⟩
    ⟨ Scene graph view constructor  ?, ... ⟩
    ⟨ Scene graph view methods  ?⟩
    ⟨ Scene graph view slots  ?, ... ⟩
◇
```

Figure 59: Scene graph view, based on Qt's QTreeView.

Editor → Scene graph view

Fragment never referenced.

⟨ *Scene graph view constructor ?* ⟩ ≡

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

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

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

Figure 60: Constructor of the scene graph view.

Editor → Scene graph view → Constructor

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Set model for scene graph view ?* ⟩ ≡

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

Figure 61: The scene graph controller is being set as the scene graph view's model.

Editor → Main application → Constructor

Fragment referenced in ?.

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. To emit the selected index of the currently selected scene directly, the slot `on_tree_item_selected` is introduced.

⟨*Scene graph view slots* ? ⟩ ≡

```
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 by the selected
    paramater.

    This method emits the selected scene graph item as scene
    graph 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]
    self.do_select_item.emit(selected_index)
    self.logger.debug(
        "Tree item was selected: %s" % selected_index
    )◇
```

Figure 62: Slot which is called when the selection within the scene graph view is changed.

Editor → Scene graph view → Slots

Fragment defined by ?, ?.
Fragment referenced in ?.

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 `selectionChanged` signal. The `selectionChanged` signal is however not directly accessible, it is only accessible through the selection model of the 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.

As stated in the above code fragment, `on_tree_item_selected`

⟨ *Scene graph view methods ?* ⟩ ≡

```
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 graph 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))
    self.logger.debug("Root node selected")◇
```

Figure 63: The setModel method, provided by QTreeView's interface, which is begin overwritten for being able to trigger the on_tree_item_selected slot whenever the selection in the scene graph view has changed.

Editor → Scene graph view → Methods

Fragment referenced in ?.

emits another signal containing a reference to the currently selected scene, which needs to be implemented as well.

⟨*Scene graph view signals* ? ⟩ ≡

```
do_select_item = QtCore.pyqtSignal(QtCore.QModelIndex)
◇
```

Figure 64: The signal that is being emitted when a scene within the scene graph view was selected. Note that the signal includes the model index of the selected item.

Editor → Scene graph view → Signals

Fragment defined by ?, ?.
Fragment referenced in ?.

Adding and removing of a scene are implemented in a similar manner as the selection of an item was 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* ? ⟩+ ≡

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

Figure 65: Signals that get emitted whenever a scene is added or removed.

Editor → Scene graph view → Signals

Fragment defined by ?, ?.
Fragment referenced in ?.

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.

The removal of a selected node shall be triggered upon the press of the `delete+ and the~\verb+backspace` key on the keyboard.

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 graph view model, which may be used by other components, instead of a model index.

One of the mentioned other components is the scene graph controller. He needs to be informed whenever a scene was added, removed or selected, so that he is able to manage his data model correspondingly.

which exactly?

⟨ *Scene graph view constructor ?* ⟩+ ≡

```
    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)
◇
```

Figure 66: Introduction of an action for adding a new scene, which reacts upon the "A" key being pressed on the keyboard.

Editor → Scene graph view → Constructor

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Scene graph view constructor ?* ⟩+ ≡

```
    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.setShortcut(Qt.QKeySequence('Backspace'))
    remove_action.setShortcutContext(QtCore.Qt.WidgetShortcut)
    remove_action.triggered.connect(self.on_tree_item_removed)
    self.addAction(remove_action)
◇
```

Figure 67: Introduction of an action for removing a new scene, which reacts upon the "delete" key being pressed on the keyboard.

Editor → Scene graph view → Constructor

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

Figure 68: Slots which emit themselves a signal whenever a scene is added from the scene graph or removed respectively.

Editor → Scene graph view → Slots

⟨*Scene graph view slots ?*⟩+ ≡

```
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 graph 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  ?⟩

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  ?⟩
        else:
            self.logger.warn("Root scene cannot be deleted")
    else:
        self.logger.warn('No item selected for removal')
◇
```

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨*Scene graph controller slots* ?⟩ ≡

```
QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_added(self, selected_item):
    # TODO: Document method.

    self.insertRows(0, 1, selected_item)
    self.logger.debug("Added new scene")

QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_removed(self, selected_item):
    # TODO: Document method.

    if not selected_item.isValid():
        self.logger.warn(
            "Selected scene is not valid, not removing"
        )
        return False

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

QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_selected(self, selected_item):
    # TODO: Document method.

    if not selected_item.isValid():
        self.logger.warn("Selected scene is not valid")
        return False

    selected_scene_view_model = selected_item.internalPointer()
    selected_scene_domain_model  = selected_scene_view_model.domain_object
    self.do_select_scene.emit(selected_scene_domain_model)◇
```

Figure 69: Slots to handle adding, removing and selecting of tree items within the scene graph. The slots take a model index as argument (coming from QAbstractItemModel). This is analogous to the scene graph view.

Editor → Scene graph controller → Slots

Fragment referenced in ?.

Despite having the slots for adding, removing and selecting scene graph items implemented, the actual methods for adding and removing scenes, `on_tree_item_added+ and~\verb+on_tree_item_removed`, 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 keep track of the domain model.

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

⟨*Scene graph controller methods ?*⟩+ ≡

```
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(parent_node.domain_object)
    view_model = guidomain_scene.SceneGraphViewModel(
        row=row,
        domain_object=domain_model,
        parent=parent_node
    )
    self.endInsertRows()

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

    return True
◇
```

Figure 70: Method for adding new scenes in terms of a domain model as well as a scene graph view model.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

The same logic applies when removing a scene.

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

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

At this point it is possible to manage scenes in terms of adding and removing them. The scenes are added to (or removed

⟨ *Scene graph controller methods ?* ⟩+ ≡

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

    if not parent.isValid():
        self.logger.warn("Cannot remove rows, parent is invalid")
        return False

    self.beginRemoveRows(parent, row, row + count - 1)
    parent_node = parent.internalPointer()
    node_index = parent.child(row, parent.column())
    node       = node_index.internalPointer()
    node.setParent(None)
    # TODO: parent_node.child_nodes.remove(node)
    self.endRemoveRows()
    self.logger.debug(
        "Removed {0} rows starting from {1} for parent {2}. Children: {3}".format(
            count, row, parent_node, len(parent_node.children())
        )
    )

    self.layoutChanged.emit()
    self.do_remove_scene.emit(node.domain_object)

    return True
◇
```

Figure 71: Method for removing scenes. Note that this is mainly done by getting the object related to the given model index and setting the parent of that object to a nil object.

Editor → Scene graph controller → Methods

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Connect main window components ?* ⟩ ≡

```
self.main_window.scene_graph_view.do_add_item.connect(
    self.scene_graph_controller.on_tree_item_added
)
self.main_window.scene_graph_view.do_remove_item.connect(
    self.scene_graph_controller.on_tree_item_removed
)
self.main_window.scene_graph_view.do_select_item.connect(
    self.scene_graph_controller.on_tree_item_selected
)◇
```

Figure 72: The scene graph view's signals for adding, removing and selecting a scene are connected to the corresponding slots from the scene graph controller. Or, in other words, the controller/data reacts to actions invoked by the user interface.

Editor → Main application → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨ *Scene graph controller signals ?* ⟩ ≡

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

Figure 73: Signals emitted by the scene graph controller, in terms of domain models, whenever a scene is added, removed or selected.

Editor → Scene graph controller → Signals

Fragment referenced in ?.

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.

Check if the paragraph is still correct.

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

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.

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`.

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.

⟨ *Main application methods ?* ⟩ ≡

```
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)◇
```

Figure 74: A method for setting up the logging, provided by the main application. If there exists an external configuration file for logging, this file is used for configuring the logging facility. Otherwise the standard configuration is used.

Editor → Main application → Methods

Fragment referenced in ?.

⟨ *Set up internals for main application ?* ⟩+ ≡

```
self.setup_logging()◇
```

Figure 75: Set up of the logging from within the main application class.

Editor → Main application → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

THE CONSEQUENCE OF PROVIDING logging on a class basis, as stated before, is, 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 [23].

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

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* 87a ⟩ ≡

```
        logger_name = "{module_name}.{class_name}".format(
    module_name=cls.__module__,
    class_name=cls.__name__
)◇
```

Fragment never referenced.

- Provide an easy to use interface for logging.

⟨ *Logger interface* 87b ⟩ ≡

```
        cls.logger = logging.getLogger(logger_name)

return cls◇
```

Fragment never referenced.

THE USAGE OF THE DECORATOR `with_logger` is shown in the example in the following listing.

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.

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.

SCENES BUILD THE BASIS for the scene graph and the node graph as well. This is a good point to begin with the implementation of the node graph.

⟨ *With logger example ?* ⟩ ≡

```
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."))

◇
```

Figure 76: An example of how to use the logging decorator in a class.

Fragment never referenced.

⟨ *Scene graph view log tree item added ?* ⟩ ≡

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

Figure 77: The scene graph view logs a corresponding message whenever an item is added to or removed from the scene graph. Note, that this logging only happens in *debug* mode.

Editor → Scene graph view → Methods

Fragment referenced in ?.

⟨ *Scene graph view log tree item removed ?* ⟩ ≡

```
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 ?.

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

WHAT ARE NODES AND NODE DEFINITIONS? As mentioned before, they 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.

MULTIPLE KINDS OF NODE REPRESENTATIONS shall be supported by the system: 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.

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.

The external definition of nodes is chosen considering these aspects. 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, it is important to define what a node actually is. As a node may be referenced 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 consists also of a definition. In terms of implicit surfaces this section contains the actual definition of a node in terms of the implicit function. In terms of triangle based meshes this is the part where the mesh and all its prerequisites as vertex array buffers and vertex array objects are set up or used from a given context.

In addition to a definition, a node contains an invocation part, which is the call of its defining function (coming from the definition mentioned just before) while respecing the parameters.

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. This part of the node is mainly about evaluating inputs and passing them on to a shader.

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 and an output plus a reference to a part. 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* ?⟩ ≡

```
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"
◇
```

Figure 78: Connections between nodes in EBNF notation.

Fragment never referenced.

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

[1] https://docs.python.org/3/library/uuid.html

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 instances of atomic types, there must be classes defining them.

Component	Description
ID	A global unique identifier (UUID[1])
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.
Definitions	A list of the node's definitions. This may be an actual definition by a (shader-) function in terms of an implicit surface or prerequisites as vertex array buffers in terms of a triangle based mesh.
Invocation	A list of the node's invocations or calls respectively.
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.

Table 13: Components a node is composed of.

For identification of the atomic types, an enumerator is used. Python provides the `enum` module, which provides a convenient interface for using enumerations[24].

[24] 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 at least 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 indicates, these types are atomic, meaning there only exists one explicit instance per type, which is therefore static. As can be seen in the code fragment below, the atomic types are parts of node definitions themselves. Only the creation of the generict atomic type is shown, the rest is omitted and can be found at

Add reference to code fragments.

Having the atomic types defined, nodes may now be defined.

⟨ *Node type declarations ?* ⟩ ≡

```
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
    IMPLICIT = 7
◇
```

Figure 79: Types of a node wrapped in a class, implemented as an enumerator.

Editor → Types → Node type

Fragment never referenced.

⟨ *Parameter declarations ?* ⟩ ≡

```
class AtomicType(object):
    """Represents an atomic type and is the basis for each node."""

    def __init__(self, id_, type_):
        """Constructor.

        :param id_: the globally unique identifier of the atomic type.
        :type  id_: uuid.uuid4
        :param type_: the type of the atomic type, e.g. "float".
        :type  type_: types.NodeType
        """

        self.id_   = id_
        self.type_ = type_
◇
```

Figure 80: The atomic type class which builds the basis for node parameters. Note that the type of an atomic type is defined by the before implemented node type.

Editor → Parameters → Atomic type

Fragment defined by ?, ?.
Fragment never referenced.

⟨*Parameter declarations* ?⟩+ ≡

```
class AtomicTypes(object):
    """Creates and holds all atomic types of the system."""

    staticmethod
    def create_node_definition_part(id_, type_):
        """Creates a node definition part based on the given identifier and
        type.

        :param id_: the identifiert to use for the part.
        :type  id_: uuid.uuid4
        :param type_: the type of the part.
        :tpye type_: qde.editor.domain.parameter.AtomicType

        :return: a node definition part.
        :rtype: qde.editor.domain.node.NodeDefinitionPart
        """

        def create_func(id_, default_function, name, type_):
            node_part = node.NodePart(id_, default_function)
            node_part.type_ = type_
            node_part.name = name
            return node_part

        node_definition_part = node.NodeDefinitionPart(id_)
        node_definition_part.type_ = type_
        node_definition_part.creator_function = create_func

        return node_definition_part

    Generic = create_node_definition_part.__func__(
        id_="54b20acc-5867-4535-861e-f461bdbf3bf3",
        type_=types.NodeType.GENERIC
    )
◇
```

Figure 81: A class which creates and holds all atomic types of the editor. Note that at this point only an atomic type for generic nodes is being created.

Editor → Parameters → Atomic types

Fragment defined by ?, ?.
Fragment never referenced.

⟨*Node domain model declarations* ?⟩ ≡

```
class NodeModel(object):
    """Represents a node."""

    # Signals
    ⟨ Node domain model signals  ?⟩

    ⟨ Node domain model constructor  ?, ... ⟩

    ⟨ Node domain model methods  ?⟩
◇
```

Figure 82: Definition of the node (domain) model.

Editor → Node model

Fragment never referenced.

⟨*Node domain model constructor* ?⟩ ≡

```
def __init__(self, id_, name="New node"):
    """Constructor.

    :param id_: the globally unique identifier of the node.
    :type  id_: uuid.uuid4
    :param name: the name of the node.
    :type  name: str
    """

    self.id_   = id_
    self.name = name

    self.definition = None
    self.description = ""
    self.parent = None
    self.inputs = []
    self.outputs = []
    self.parts = []
    self.nodes = []
    self.connections = []
◇
```

Figure 83: Constructor of the node (domain) model.

Editor → Node model → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

While the details of a node are rather unclear at the moment, it is clear that a node needs to have a view model, which renders a node within a scene of the node graph.

Qt does not offer a graph view by default, therefore it is necessary to implement such a graph view.

The most obvious choice for this implementation is the `QGraphicsView` component, which displays the contents of a `QGraphicsScene`, whereas `QGraphicsScene` manages `QGraphicsObject` components.

It is therefore obvious to use the `QGraphicsObject` component for representing graph nodes through a view model.

⟨*Node view model declarations ?*⟩ ≡

```
class NodeViewModel(Qt.QGraphicsObject):
    """Class representing a single node within GUI."""

    # Constants
    WIDTH = 20
    HEIGHT = 17

    # Signals
    ⟨ Node view model signals  ?⟩

    ⟨ Node view model constructor  ?, ... ⟩

    ⟨ Node view model methods  ?, ... ⟩
◇
```

Figure 84: Definition of the node view model.

Editor → Node view model

Fragment never referenced.

To distinguish nodes, the name and the type of a node is used. It makes sense to access both attributes directly via the domain model instead of duplicating them.

The domain model does not provide access to its type at the moment however. The type is directly derived from the primary output of a node. If a node has no outputs at all, its type is assumed to be generic.

Concerning the drawing of nodes (or painting, as Qt calls it) , each node type may be used multiple times. But instead of re-creating the same image representation over and over again, it makes sense to create it only once per node type. Qt provides `QPixmap+ and~\verb+QPixmapCache` for this use case.

Each node has a cache key assigned, which is used to identify that node.

⟨ *Node view model constructor ?* ⟩ ≡

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

    :param id_: the globally unique identifier of the atomic type.
    :type  id_: uuid.uuid4
    :param domain_object: Reference to a scene model.
    :type  domain_object: qde.editor.domain.scene.SceneModel
    :param parent: The parent of the current view widget.
    :type parent:  QtCore.QObject
    """

    super(NodeViewModel, self).__init__(parent)
    self.id_ = id_
    self.domain_object = domain_object

    self.position = QtCore.QPoint(0, 0)
    self.width = 4
◇
```

Figure 85: Constructor of the node view model.

Editor → Node view model → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

The cache key is composed of the type of the node, its status and whether it is selected or not.

As can be seen in the above code fragment, the status property of the node is used to create a cache key, but currently nodes do not have a status.

It may make sense although to provide a status for each node, which allows to output eventual problems like not having required connections and so on.

This status is added to the constructor of the domain model of a node.

Concerning the view model, again the status of the domain model is used as otherwise different states between user interface and domain model would be possible in the worst case.

Therefore it can now be checked, whether a node has a cache key or not. If it has no cache key, a new cache key is created.

The cache key itself is then used to find a corresponding pixmap.

If no pixmap with the given cache key exists, a new pixmap is being created and added to the cache using the cache key created before.

For actually displaying the nodes, another component is necessary: the scene view which is a graph consisting the nodes and edges.

⟨ *Node view model methods ?* ⟩ ≡

```
property
def type_(self):
    """Return the type of the node, determined by its domain model.

    :return: the type of the node.
    :rtype: types.NodeType
    """

    return self.domain_model.type_
◇
```

Figure 86: The type and name attributes of the node view model as properties.

Editor → Node view model → Methods

Fragment defined by ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Node view model methods ?* ⟩+ ≡

```
property
def name(self):
    """Return the name of the node, determined by its domain model.

    :return: the name of the node.
    :rtype: str
    """

    return self.domain_model.name
◇
```

Fragment defined by ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Node domain model methods ?* ⟩ ≡

```
    property
    def type_(self):
        """Return the type of the node, determined by its primary output.
        If no primary output is given, it is assumed that the node is of
        generic type."""

        type_ = types.NodeType.GENERIC

        if len(self.outputs) > 0:
            type_ = self.outputs[0].type_

        return type_
◇
```

Figure 87: The type attributes of the node domain model as property.

Editor → Node (domain) model → Methods

Fragment referenced in ?.

⟨*Node view model methods* ?⟩+ ≡

```
def paint(self, painter, option, widget):
    """Paint the node.

    First a pixmap is loaded from cache if available, otherwise
    a new pixmap gets created. If the current node is selected a
    rectangle gets additionally drawn on it. Finally the name, the type
    as well as the subtype gets written on the node.
    """

    ⟨ Node view model methods paint  ?, ... ⟩
◇
```

Figure 88: The paint method of the node view model. When a pixmap is being created, it gets cached immediately, based on its type, status and its selection status. If a pixmap already existing for a given tripe, type, status and selection, that pixmap is used.

Editor → Node view model → Methods

Fragment defined by ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨*Node view model constructor* ?⟩+ ≡

```
    self.cache_key = None
◇
```

Figure 89: The cache key is being initialized within a node's constructor.

Editor → Node view model → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨*Node view model methods* ?⟩+ ≡

```
def create_cache_key(self):
    """Create an attribute based cache key for finding and creating
    pixmaps."""

    return "{type_name}{status}{selected}".format(
        type_name=self.type_,
        status=self.status,
        selected=self.isSelected(),
    )
◇
```

Figure 90: A method which creates a cache key based on the type, the status and the state of selection of a node.

Editor → Node view model → Methods

Fragment defined by ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨*Node domain model constructor* ?⟩+ ≡

```
    self.status = flag.NodeStatus.OK
◇
```

Figure 91: The status of the node is being initialized within the node's constructor.

Editor → Node domain model → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨ *Node view model methods ?* ⟩+ ≡

```
property
def status(self):
    """Return the current status of the node.

    :return: the current status of the node.
    :rtype: flag.NodeStatus
    """

    return self.domain_object.status
◇
```

Figure 92: The status of a node view model is obtained by accessing the domain model's status.

Editor → Node domain model → Methods

Fragment defined by ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Node view model methods paint ?* ⟩ ≡

```
if self.cache_key is None:
    self.cache_key = self.create_cache_key()
◇
```

Figure 93: A cache key is being created when no cache key for the given attributes is found.

Editor → Node view model → Methods → Paint

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Node view model methods paint ?* ⟩+ ≡

```
    pixmap = Qt.QPixMapCache.find(self.cache_key)
◇
```

Figure 94: Based on the created or retrieved cache key a pixmap is being searched for.

Editor → Node view model → Methods → Paint

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Node view model methods paint ?* ⟩+ ≡

```
    if pixmap is None:
        pixmap = self.create_pixmap()
        Qt.QPixmapCache.insert(self.cache_key, pixmap)
◇
```

Figure 95: If no pixmap is found, a new pixmap is being created for the provided key and stored.

Editor → Node view model → Methods → Paint

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

Scene view

For implementing the scene view the `QGraphicsView` component is used as basis, as before with the node graph component. The graphics view displays the contents of scene, therefore a `QGraphicsScene`, whereas `QGraphicsScene` manages nodes in form of `QGraphicsObject` components.

⟨ *Scene view declarations ?* ⟩ ≡

```
common.with_logger
class SceneView(Qt.QGraphicsView):
    """Scene view widget.
    A widget for displaying and managing scenes including their nodes and
    connections between nodes."""

    # Signals
    ⟨ Scene view signals  ?⟩

    ⟨ Scene view constructor  ?⟩
    ⟨ Scene view methods  ?⟩
    ⟨ Scene view slots  ?⟩
◇
```

Figure 96: Definition of the scene view component, derived from the QGraphicsView component.

Editor → Scene view

Fragment never referenced.

⟨ *Scene view constructor ?* ⟩ ≡

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

    :param parent: the parent of this scene view.
    :type parent: Qt.QObject
    """

    super(SceneView, self).__init__(parent)
◇
```

Figure 97: Constructor of the scene view component.

Editor → Scene view → Constructor

Fragment referenced in ?.

The scene view can now be set up by the main window and is then added to its vertical splitter.

⟨ *Set up scene view in main window ?* ⟩ ≡

```
self.scene_view = guiscene.SceneView()
self.scene_view.setObjectName('scene_view')
size_policy = QtWidgets.QSizePolicy(
    QtWidgets.QSizePolicy.Expanding,
    QtWidgets.QSizePolicy.Expanding
)
size_policy.setHorizontalStretch(2)
size_policy.setVerticalStretch(0)
size_policy.setHeightForWidth(self.scene_view.sizePolicy().hasHeightForWidth())
self.scene_view.setSizePolicy(size_policy)
self.scene_view.setMinimumSize(Qt.QSize(0, 0))
self.scene_view.setAutoFillBackground(False)
self.scene_view.setFrameShape(QtWidgets.QFrame.StyledPanel)
self.scene_view.setFrameShadow(QtWidgets.QFrame.Sunken)
self.scene_view.setLineWidth(1)
self.scene_view.setVerticalScrollBarPolicy(QtCore.Qt.ScrollBarAsNeeded)
self.scene_view.setHorizontalScrollBarPolicy(QtCore.Qt.ScrollBarAsNeeded)
brush = QtGui.QBrush(Qt.QColor(0, 0, 0, 255))
brush.setStyle(QtCore.Qt.NoBrush)
self.scene_view.setBackgroundBrush(brush)
self.scene_view.setAlignment(QtCore.Qt.AlignLeadingQtCore.Qt.AlignLeftQtCore.Qt.AlignTop)
self.scene_view.setDragMode(QtWidgets.QGraphicsView.RubberBandDrag)
self.scene_view.setTransformationAnchor(QtWidgets.QGraphicsView.AnchorUnderMouse)
self.scene_view.setOptimizationFlags(QtWidgets.QGraphicsView.DontAdjustForAntialiag)
◇
```

Figure 98: The scene view component is being set up by the main window.

Editor → Main window → Methods → Setup UI

Fragment referenced in ?.

⟨ *Add scene view to vertical splitter in main window ?* ⟩ ≡

```
vertical_splitter.addWidget(self.scene_view)
◇
```

Figure 99: The scene view component is being added to the main window's vertical splitter.

Editor → Main window → Methods → Setup UI

Fragment referenced in ?.

At this point the scene view does not react whenever the scene is changed by the scene graph view. As before, the main application needs connect the components.

Connecting the view models of the scene graph view and the scene view directly would not make much sense, as they both use different view models. Instead it makes sense to connect the `do_select_scene` signal of the scene graph controller with the `on_scene_changed` slot of the scene controller as they both use the domain model of the scene.

⟨*Connect controllers for main application* ?⟩ ≡

```
self.scene_graph_controller.do_select_scene.connect(
    self.scene_controller.on_scene_changed
)◇
```

Figure 100: Whenever a scene is selected in the scene graph, the scene graph controller informs the scene controller about that selection.

Editor → Main application → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

The scene controller does not manage scene models directly, as the scene graph controller does. Instead it reacts on signals sent by the latter and manages its own scene view models.

⟨*Connect controllers for main application* ?⟩+ ≡

```
self.scene_graph_controller.do_add_scene.connect(
    self.scene_controller.on_scene_added
)
self.scene_graph_controller.do_remove_scene.connect(
    self.scene_controller.on_scene_removed
)◇
```

Figure 101: Whenever a scene is added to or removed from the scene graph, the scene graph controller informs the scene controller about those actions.

Editor → Main application → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

Loose some words about the scene controller?

The scene view models represent a certain scene of the scene graph and hold the nodes of a specific scene. A scene view model is of type `QGraphicsScene`.

For being able to distinguish different scenes, their identifier will be drawn at the top left position.

The scene controller does not directly manage scenes. It has to react upon the signals sent by the scene graph controller. Additionally it needs to keep track of the currently selected scene, by holding a reference to that. The common identifier is the identifier of the domain model.

Whenever a new scene is created, the scene controller needs to create a scene of type `QGraphicsScene` and needs to keep track of that scene.

Whenever a scene is deleted, it needs to delete the scene from

⟨*Scene controller declarations ?*⟩ ≡

Figure 102: Definition of the scene controller.

Editor → Scene controller

```
common.with_logger
class SceneController(Qt.QObject):
    """The scene controller.

    A controller for switching scenes and managing the nodes of a scene by
    adding, editing and removing nodes to / from a scene.
    """

    # Signals
    ⟨ Scene controller signals  ?⟩

    ⟨ Scene controller constructor  ?⟩
    ⟨ Scene controller methods  ?⟩

    ⟨ Scene controller slots  ?, ... ⟩
◇
```

Fragment never referenced.

⟨*Set up controllers for main application ?*⟩+ ≡

Figure 103: The scene controller being set up by the main application.

Editor → Main application → Constructor

```
self.scene_controller = scene.SceneController(self)◇
```

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨*Scene view model declarations ?*⟩ ≡

Figure 104: Definition of the scene view model.

Editor → Scene view model

```
common.with_logger
class SceneViewModel(Qt.QGraphicsScene):
    """Scene view model.
    Represents a certain scene from the scene graph and is used to manage the
    nodes of that scene."""

    # Constants
    WIDTH = 15
    HEIGHT = 15

    # Signals
    ⟨ Scene view model signals  ?⟩

    ⟨ Scene view model constructor  ?⟩
    ⟨ Scene view model methods  ?⟩
◇
```

Fragment never referenced.

⟨*Scene view model constructor ?*⟩ ≡

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

   :param domain_object: Reference to a scene model.
   :type  domain_object: qde.editor.domain.scene.SceneModel
   :param parent:        The parent of the current view model.
   :type parent:         qde.editor.gui_domain.scene.SceneViewModel
   """

   super(SceneViewModel, self).__init__(parent)

   self.id_              = domain_object.id_
   self.nodes            = []
   self.insert_at        = QtCore.QPoint(0, 0)
   self.insert_at_colour = Qt.QColor(self.palette().highlight().color())

   self.width            = SceneViewModel.WIDTH * 20
   self.height           = SceneViewModel.HEIGHT * 17

   self.setSceneRect(0, 0, self.width, self.height)
   self.setItemIndexMethod(self.NoIndex)
◇
```

Figure 105: Constructor of the scene view model component.

Editor → Scene view model → Constructor

Fragment referenced in ?.

its known scenes as well.

To actually change the scene, the scene controller needs to react whenever the scene was changed. It does that by reacting on the `do_select_scene` signal sent by the scene graph controller.

As can be seen in Figure 111, the scene controller emits a signal that the scene was changed, containing the view model of the new scene.

The emitted signal, `do_change_scene`, is in turn consumed by the `on_scene_changed` slot of the scene view for actually changing the displayed scene.

At this point scenes can be managed and displayed but they still cannot be rendered as nodes cannot be added yet. First of all as there are no nodes yet and second as there exists no possibility to add nodes.

⟨ *Scene view model methods ?* ⟩ ≡

```
def drawBackground(self, painter, rect):
    # io = Qt.QGraphicsTextItem()
    # io.setPos(0, 0)
    # io.setDefaultTextColor(Qt.QColor(102, 102, 102))
    # io.setPlainText(
    #     "Scene: {0}".format(str(self))
    # )
    # self.addItem(io)

    scene_rect = self.sceneRect()
    text_rect = QtCore.QRectF(scene_rect.left()   + 4,
                              scene_rect.top()    + 4,
                              scene_rect.width()  - 4,
                              scene_rect.height() - 4)
    message = str(self)
    font = painter.font()
    font.setBold(True)
    font.setPointSize(14)
    painter.setFont(font)
    painter.setPen(QtCore.Qt.lightGray)
    painter.drawText(text_rect.translated(2, 2), message)
    painter.setPen(QtCore.Qt.black)
    painter.drawText(text_rect, message)◇
```

Figure 106: The method to draw the background of a scene. It is used to draw the identifier of a scene at the top left position of it.

Editor → Scene view model → Methods

Fragment referenced in ?.

⟨ *Scene controller constructor ?* ⟩ ≡

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

    :param parent: the parent of this scene controller.
    :type parent: Qt.QObject
    """

    super(SceneController, self).__init__(parent)

    self.scenes = {}
    self.current_scene = None
◇
```

Figure 107: Constructor of the scene controller. As can be seen, the scene controller holds all scenes (as a dictionary) and keeps track of the currently active scene.

Editor → Scene controller → Constructor

Fragment referenced in ?.

⟨*Scene controller slots ?*⟩ ≡

```
QtCore.pyqtSlot(domain_scene.SceneModel)
def on_scene_added(self, scene_domain_model):
    """React when a scene was added.

    :param scene_domain_model: the scene that was added.
    :type scene_domain_model:  qde.domain.scene.SceneModel
    """

    if scene_domain_model.id_ not in self.scenes:
        scene_view_model = guidomain_scene.SceneViewModel(
            domain_object=scene_domain_model
        )
        self.scenes[scene_domain_model.id_] = scene_view_model
        self.logger.debug("Scene '%s' was added" % scene_view_model)
    else:
        self.logger.debug("Scene '%s' already known" % scene)
◇
```

Figure 108: The slot which gets triggered whenever a new scene is added via the scene graph.

Editor → Scene controller → Slots

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨*Scene controller slots ?*⟩+ ≡

```
QtCore.pyqtSlot(domain_scene.SceneModel)
def on_scene_removed(self, scene_domain_model):
    """React when a scene was removed/deleted.

    :param scene_domain_model: the scene that was removed.
    :type scene_domain_model:  qde.domain.scene.SceneModel
    """

    if scene_domain_model.id_ in self.scenes:
        del(self.scenes[scene_domain_model.id_])
        self.logger.debug("Scene '%s' was removed" % scene_domain_model)
    else:
        self.logger.warn((
            "Scene '%s' should be removed, "
            "but is not known"
        ) % scene_domain_model)
◇
```

Figure 109: The slot which gets triggered whenever a scene is removed via the scene graph.

Editor → Scene controller → Slots

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Scene controller slots ?* ⟩+ ≡

```
QtCore.pyqtSlot(domain_scene.SceneModel)
def on_scene_changed(self, scene_domain_model):
    """Gets triggered when the scene was changed by the view.

    :param scene_domain_model: The currently selected scene.
    :type  scene_domain_model: qde.editor.domain.scene.SceneModel
    """

    if scene_domain_model.id_ in self.scenes:
        self.current_scene = self.scenes[scene_domain_model.id_]
        self.do_change_scene.emit(self.current_scene)
        self.logger.debug("Scene changed: %s", self.current_scene)
    else:
        self.logger.warn((
            "Should change to scene '%s', "
            "but that scene is not known"
        ) % scene_domain_model)
◇
```

Figure 110:

Editor → Scene controller → Slots

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Scene controller signals ?* ⟩ ≡

```
do_change_scene = QtCore.pyqtSignal(guidomain_scene.SceneViewModel)
◇
```

Figure 111: The signal which is emitted when the scene has been changed by the scene graph controller and that scene is known to the scene controller.

Editor → Scene controller → Signals

Fragment referenced in ?.

⟨ *Scene view slots ?* ⟩ ≡

```
QtCore.pyqtSlot(scene.SceneViewModel)
def on_scene_changed(self, scene_view_model):
    # TODO: Document method

    self.setScene(scene_view_model)
    # TODO: self.scrollTo(scene_view_model.view_position)
    self.scene().invalidate()
    self.logger.debug("Scene has changed: %s", scene_view_model)◇
```

Figure 112: The slot of the scene view, which gets triggered whenever the scene changes. The scene interface, provided by QGraphicsView, is then invalidated to trigger the rendering of the scene view.

Editor → Scene view → Slots

Fragment referenced in ?.

⟨*Connect main window components* ?⟩+ ≡

```
self.scene_controller.do_change_scene.connect(
    self.main_window.scene_view.on_scene_changed
)◇
```

Figure 113: The main application connects the scene view's signal that the scene was changed with the corresponding slot of the scene controller.

Editor → Main application → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

Nodes

Thinking of the definition of what shall be achieved, as defined at Appendix , a node defining a sphere is implemented.

⟨ *Implicit sphere node ?* ⟩ ≡

```
{
    "name": "Implicit sphere",
    "id_": "16d90b34-a728-4caa-b07d-a3244ecc87e3",
    "description": "Definition of a sphere by using implicit surfaces",
    "inputs": [
        ⟨ Implicit sphere node inputs  ?⟩
    ],
    "outputs": [
        ⟨ Implicit sphere node outputs  ?⟩
    ],
    "definitions": [
        ⟨ Implicit sphere node definitions  ?⟩
    ],
    "invocations": [
        ⟨ Implicit sphere node invocations  ?⟩
    ],
    "parts": [
        ⟨ Implicit sphere node parts  ?⟩
    ],
    "nodes": [
        ⟨ Implicit sphere node nodes  ?⟩
    ],
    "connections": [
        ⟨ Implicit sphere node connections  ?⟩
    ]
}◇
```

Figure 114: Definition of a node for an implicitly defined sphere.

Implicit sphere node

Fragment never referenced.

At the current point the sphere node will only have one input: the radius of the sphere. The positition of the sphere will be at the center (meaning the X-, the Y- and the Z-position are all 0).

For being able to change the position, another node will be introduced.

The output of the sphere node is of type implicit as the node

⟨ *Implicit sphere node inputs ?* ⟩ ≡

```
{
    "name": "radius",
    "atomic_id": "468aea9e-0a03-4e63-b6b4-8a7a76775a1a",
    "default_value": {
        "type_": "float",
        "value": "1"
    },
    "id_": "f5c6a538-1dbc-4add-a15d-ddc4a5e553da",
    "description": "The radius of the sphere",
    "min_value": "-1000",
    "max_value": "1000"
}◇
```

Figure 115: Radius of the implicit sphere node as input.

Implicit sphere node → Inputs

Fragment referenced in ?.

represents an implicit surface.

⟨ *Implicit sphere node outputs ?* ⟩ ≡

```
{
    "name": "output",
    "id_": "a3ac68e5-5afe-4779-9e9f-5b619e041ae6",
    "atomic_id": "c019271c-35b6-425c-9ff2-a1d893111adb"
}◇
```

Figure 116: The output of the implicit sphere node, which is of the atomic type implicit.

Implicit sphere node → Outputs

Fragment referenced in ?.

The definition of the node is the actual implementation of a sphere as a implicit surface.

The invocation of the node is simply calling the above definition using the parameters of the node, which is in this case the radius.

The parameters are in case of implicit surfaces uniform variables of the type of the parameter, as implicit surfaces are rendered by the fragment shader. The uniform variables are defined by a type and an identifier, whereas in the case of paramaters their identifier is used.

The position of the node is an indirect parameter, which is not defined by the node's inputs. It will be setup by the node's parts.

The parts of the node, in this case it is only one part, contain the body of the node. The body is about evaluating the inputs and passing them on to a shader.

⟨ *Implicit sphere node definitions ?* ⟩ ≡

Figure 117: Implementation of the sphere in the OpenGL Shading Language (GLSL).

Implicit sphere node → Definitions

```
{
    "id_": "99d20a26-f233-4310-adb2-5e540726d079",
    "script": [
        "// Returns the signed distance to a sphere with given radius for the",
        "// given position.",
        "float sphere(vec3 position, float radius)",
        "{",
        "    return length(position) - radius;",
        "}"
    ]
}◇
```

Fragment referenced in ?.

⟨ *Implicit sphere node invocations ?* ⟩ ≡

Figure 118: The position of the implicit sphere node as invocation.

Implicit sphere node → Invocations

```
{
    "id_": "4cd369d2-c245-49d8-9388-6b9387af8376",
    "type": "implicit",
    "script": [
        "float s = sphere(",
        "    16d90b34-a728-4caa-b07d-a3244ecc87e3-position,",
        "    5c6a538-1dbc-4add-a15d-ddc4a5e553da",
        ");"
    ]
}◇
```

Fragment referenced in ?.

Change this to C and use CFFI.

⟨ *Implicit sphere node parts ?* ⟩ ≡

```
{
    "id_": "74b73ce7-8c9d-4202-a533-c77aba9035a6",
    "name": "Implicit sphere node function",
    "type_": "implicit",
    "script": [
        "# -*- coding: utf-8 -*-",
        "",
        "from PyQt5 import QtGui",
        "",
        "",
        "class Class_ImplicitSphere(object):",
        "    def __init__(self):",
        "        self.position = QtGui.QVector3D()",
        "",
        "    def process(self, context, inputs):",
        "        shader = context.current_shader.program",
        "        ",
        "        radius = inputs[0].process(context).value",
        "        shader_radius_location = shader.uniformLocation(\"f5c6a538-1dbc-4add-a15d-ddc4a5e553da\")",
        "        shader.setUniformValue(shader_radius_location, radius)",
        "        ",
        "        position = self.position",
        "        shader_position_location = shader.uniformLocation(",
        "            \"16d90b34-a728-4caa-b07d-a3244ecc87e3-position\"",
        "        )",
        "        shader.setUniformValue(shader_position_location, position)",
        "        ",
        "        return context"
    ]
}◇
```

Figure 119: The "body" of the implicit sphere node as node part.

Implicit sphere node → Parts

Fragment referenced in ?.

Connections are composed of an input and an output plus a reference to a part, as stated in . In this case there is exactly one input, the radius, and one output, an object defined by implicit functions.

Add reference

The radius is being defined by an input, which is therefore being referenced as source. There is although no external node being referenced, as the radius is of the atomic type float. Therefore the source node is 0, meaning it is an internal reference. The input itself is used as part for the input.

The very same applies for the output of that connection. The radius is being consumed by the first part of the node's part (which has only this part). As this definition is within the same node, the target node is also 0. The part is then being referenced by its identifier.

⟨ *Implicit sphere node connections ?* ⟩ ≡

```
{
    "source_node": "00000000-0000-0000-0000-000000000000",
    "source_part": "f5c6a538-1dbc-4add-a15d-ddc4a5e553da",
    "target_node": "00000000-0000-0000-0000-000000000000",
    "target_part": "74b73ce7-8c9d-4202-a533-c77aba9035a6"
}◇
```

Figure 120: Mapping of the connections of the implicit sphere node. Note that the inputs and outputs are internal, therefore the node references are 0.

Implicit sphere node → Connections

Fragment referenced in ?.

Now a very basic node is availble, but the node does not get recognized by the application yet. As nodes are defined by external files, they need to be searched, loaded and registered to make them available to the application.

Therefore the node controller is introduced, which will manage the node definitions.

⟨ *Node controller declarations ?* ⟩ ≡

```
common.with_logger
class NodeController(QtCore.QObject):
    """The node controller.

    A controller managing nodes.
    """

    # Constants
    NODES_PATH = "nodes"
    NODES_EXTENSION = "node"
    ROOT_NODE_ID = uuid.UUID("026c04d0-36d2-49d5-ad15-f4fb87fe8eeb")
    ROOT_NODE_OUTPUT_ID = uuid.UUID("a8fadcfc-4e19-4862-90cf-a262eef2219b")

    # Signals
    ⟨ Node controller signals  ?⟩

    ⟨ Node controller constructor  ?, ... ⟩
    ⟨ Node controller methods  ?⟩

    ⟨ Node controller slots  ?⟩
◇
```

Figure 121: Definition of the node controller.

Editor → Node controller

Fragment never referenced.

The node controller assumes, that all node definitions are placed within the `nodes` subdirectory of the application's working directory. Further it assumes, that node definition files use the `node` extension.

⟨ *Node controller constructor ?* ⟩ ≡

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

    :param parent: the parent of this node controller.
    :type  parent: QtCore.QObject
    """

    super(NodeController, self).__init__(parent)

    self.nodes_path = "{current_dir}{sep}{nodes_path}".format(
        current_dir=os.getcwd(),
        sep=os.sep,
        nodes_path=NodeController.NODES_PATH
    )
    self.nodes_extension = NodeController.NODES_EXTENSION◇
```

Figure 122: Constructor of the node controller.

Editor → Node controller → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

THE NODE CONTROLLER WILL THEN SCAN that directory containing the node definitions and load each one.

⟨ *Node controller methods ?* ⟩ ≡

```
def load_nodes(self):
    """Loads all files with the ending NodeController.NODES_EXTENSION
    within the NodeController.NODES_PATH directory, relative to the current
    working directory.
    """

    ⟨ Node controller load nodes method  ?⟩◇
```

Figure 123: A method that loads node definitions from external files from within the node controller.

Editor → Node controller → Methods

Fragment referenced in ?.

NODE DEFINITONS WILL CONTAIN PARTS. The parts within a node definition are used to create corresponding parts within instances of themselves. The parts are able to create values based on the atomic types through functions.

THE PART OF A NODE DEFINITION holds an identifier as well as an expression to create a function for creating and handling values which will be used when evaluating a node. Further it provides a function which allows to instantiate itself as part of a node (instance).

THE NODE CONTROLLER NEEDS TO KEEP TRACK of node defintion parts, as they are a central aspect and may be reused.

⟨*Node definition part domain model declarations ?*⟩ ≡

```
class NodeDefinitionPart(object):
    """Represents a part of the definition of a node."""

    # Signals
    ⟨ Node definition part domain model signals  ?⟩

    ⟨ Node definition part domain model constructor  ?⟩
    ⟨ Node definition part domain model methods  ?⟩◇
```

Figure 124: Definition of a part of a node definition.

Editor → Node definition part

Fragment never referenced.

⟨*Node definition part domain model constructor ?*⟩ ≡

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

    :param id_: the globally unique identifier of the part of the node
                definition.
    :type  id_: uuid.uuid4
    """

    self.id_     = id_
    self.type_   = None
    self.name    = None
    self.parent = None

    # This property is used when evaluating node instances using this node
    # definition
    self.function_creator = lambda: create_value_function(
        parameter.FloatValue(0)
    )

    # This property will be used to create/instantiate a part of a node
    # instance
    self.creator_function = None
    ◇
```

Figure 125: Constructor of the node definition part.

Editor → Node definition part

Fragment referenced in ?.

⟨*Node controller constructor ?*⟩+ ≡

```
    self.node_definition_parts = {}
◇
```

Figure 126: The node controller keeps track of node definition parts.

Editor → Node controller → Constructor

Fragment defined by ?, ?.
Fragment referenced in ?.

The code snippet defining the constructor of a node definition part, Figure 125, uses a function called `create_value_function` of the `functions` module.

⟨*Node domain module methods ?*⟩ ≡

```
def create_value_function(value):
    """Creates a new value function using the provided value.

    :param value: the value which the function shall have.
    :type  value: qde.editor.domain.parameter.Value
    """

    value_function = NodePart.ValueFunction()
    value_function.value = value.clone()

    return value_function
◇
```

Figure 127: Helper function which creates a value function from the given value.

Editor → Node domain model → Module methods

Fragment never referenced.

That brings up the concept of value functions. Value functions are one of the building blocks of a node. They are used to evaluate a node value-wise through its inputs.

The value function of a node may not be clear during the initialization of the node or it may be simply be subject to change. Therefore it makes sense to provide a default value function which gets used by default.

The value function relies strongly on the conecpt of node parts, which is not defined yet. A part of a node is actually an instance of an atomic type (which is usually an input) within an instance of a node definition.

⟨ *Node part domain model value function declarations ?* ⟩ ≡

```
class ValueFunction(Function):
    """Class representing a value function for nodes."""

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

        super(NodePart.ValueFunction, self).__init__()
        self.value = None

    def clone(self):
        """Clones the currently set value function.

        :return: a clone of the currently set value function.
        :rtype: qde.editor.domain.node.NodePart.Function
        """

        new_function = create_value_function(self.value)
        new_function.node_part = self.node_part

        return new_function

    def process(self, context, inputs, output_index):
        """Processes the value function for the given context, the given inputs
        and the given index of the output.

        :param context: the context of the processing
        :type  context: qde.editor.domain.node.NodePartContext
        :param inputs: a list of inputs to process
        :type inputs: list
        :param output_index: the index of the output which shall be used
        :type output_index: int

        :return: the context
        :rtype:  qde.editor.domain.node.NodePartContext
        """

        if not self.value.is_cachable or self.has_changed:
            if len(inputs) > 0:
                inputs[0].process(context, self.processing_index)
                value.set_value_from_context(context)
            else:
                self.value.set_value_in_context(context)

            self.has_changed = False
        else:
            self.value.set_value_in_context(context)

        # TODO: Handle events

        return context◇
```

Figure 128: Definition of the value function class which is used within nodes.

Editor → Value function

Fragment referenced in ?.

⟨ *Node part domain model default value function declarations ?* ⟩ ≡

```
class DefaultValueFunction(ValueFunction):
    """The default value function of a node part."""

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

        super(NodePart.DefaultValueFunction, self).__init__()

    def clone(self):
        """Returns itself as a default value function may not be cloned.

        :return: a self-reference.
        :rtype: DefaultValueFunction
        """

        return self

    def process(self, context, inputs, output_index):
        """Processes the default value function for the given context, the given inputs
        and the given index of the output.

        :param context: the context of the processing
        :type  context: qde.editor.domain.node.NodePartContext
        :param inputs: a list of inputs to process
        :type inputs: list
        :param output_index: the index of the output which shall be used
        :type output_index: int

        :return: the context
        :rtype:  qde.editor.domain.node.NodePartContext
        """

        self.value.set_value_in_context(context)
        self.has_changed = False

        return context◇
```

Figure 129: Definition of the default value function class, which is derived from the value function class.

Editor → Default value function

Fragment referenced in ?.

⟨ *Node part domain model declarations ?* ⟩ ≡

Figure 130: The node part class.

Editor → Node part

```
class NodePart(object):
    """Represents a part of a node."""

    ⟨ Node part domain model function declarations  ?⟩
    ⟨ Node part domain model value function declarations  ?⟩
    ⟨ Node part domain model default value function declarations  ?⟩

    # Signals
    ⟨ Node part domain model signals  ?⟩

    ⟨ Node part domain model constructor  ?⟩
    ⟨ Node part domain model methods  ?⟩
◇
```

Fragment never referenced.

⟨ *Node part domain model constructor ?* ⟩ ≡

Figure 131: Constructor of the node part class.

Editor → Node part

```
def __init__(self, id_, default_function):
    """Constructor.

    :param id_: the identifier of the node part.
    :type  id_: uuid.uuid4
    :param default_function: the default function of the part
    :type default_function: Function
    """

    self.id_               = id_
    self.function_         = default_function
    self.default_function  = default_function
    self.type_             = types.NodeType.GENERIC◇
```

Fragment referenced in ?.

Bibliography

[1] A. Appel, "Some Techniques for Shading Machine Renderings of Solids", in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring), New York, NY, USA: ACM, 1968, pp. 37–45. DOI: `10.1145/1468075.1468082`.

[2] T. Whitted, "An Improved Illumination Model for Shaded Display", *Commun. ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980, ISSN: 0001-0782. DOI: `10.1145/358876.358882`.

[3] J. C. Hart, "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces", *The Visual Computer*, vol. 12, pp. 527–545, 1994.

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

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

[6] P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN: 0321197704.

[7] M. Fowler, "Who needs an architect?", *IEEE Softw.*, vol. 20, no. 5, pp. 11–13, Sep. 2003, ISSN: 0740-7459. DOI: `10.1109/MS.2003.1231144`.

[8] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004, ISBN: 978-0-13-148906-6.

[9] J. Foley, *Computer Graphics: Principles and Practice*, ser. Addison-Wesley systems programming series. Addison-Wesley, 1996, ISBN: 978-0-201-84840-3.

[10] J. T. Kajiya, "The Rendering Equation", *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 143–150, Aug. 1986, ISSN: 0097-8930. DOI: `10.1145/15886.15902`.

[11] C. A. R. Hoare, "Hints on programming language design", Stanford, CA, USA, Tech. Rep., 1973.

[12] D. E. Knuth, "Literate programming", *Comput. J.*, vol. 27, no. 2, pp. 97–111, May 1984, ISSN: 0010-4620. DOI: `10.1093/comjnl/27.2.97`.

[13] P. Briggs, "Nuweb, A simple literate programming tool", Rice University, Houston, TX, Tech. Rep., 1993.

[14] H. Hijazi, T. Khdour, and A. Alarabeyyat, "Article: A review of risk management in different software development methodologies", *International Journal of Computer Applications*, vol. 45, no. 7, pp. 8–12, Apr. 2012, Full text available.

[15] T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer Publishing Company, Incorporated, 2008, ISBN: 9783540764397.

[16] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004, ISBN: 0321278658.

[17] J. Shore and S. Warden, *The Art of Agile Development*, First. O'Reilly, 2007, ISBN: 9780596527679.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2.

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

[20] J. Gossman. (). Introduction to model/view/ViewModel pattern for building WPF apps, Tales from the Smart Client, [Online]. Available: `https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/` (visited on 06/07/2017).

[21] The Qt Company Ltd., *Model/View Programming | Qt Widgets 5.9*, en, 2017.

[22] I. C. Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014, ISBN: 9780769551661.

Fix glossaries

Print index