QDE.

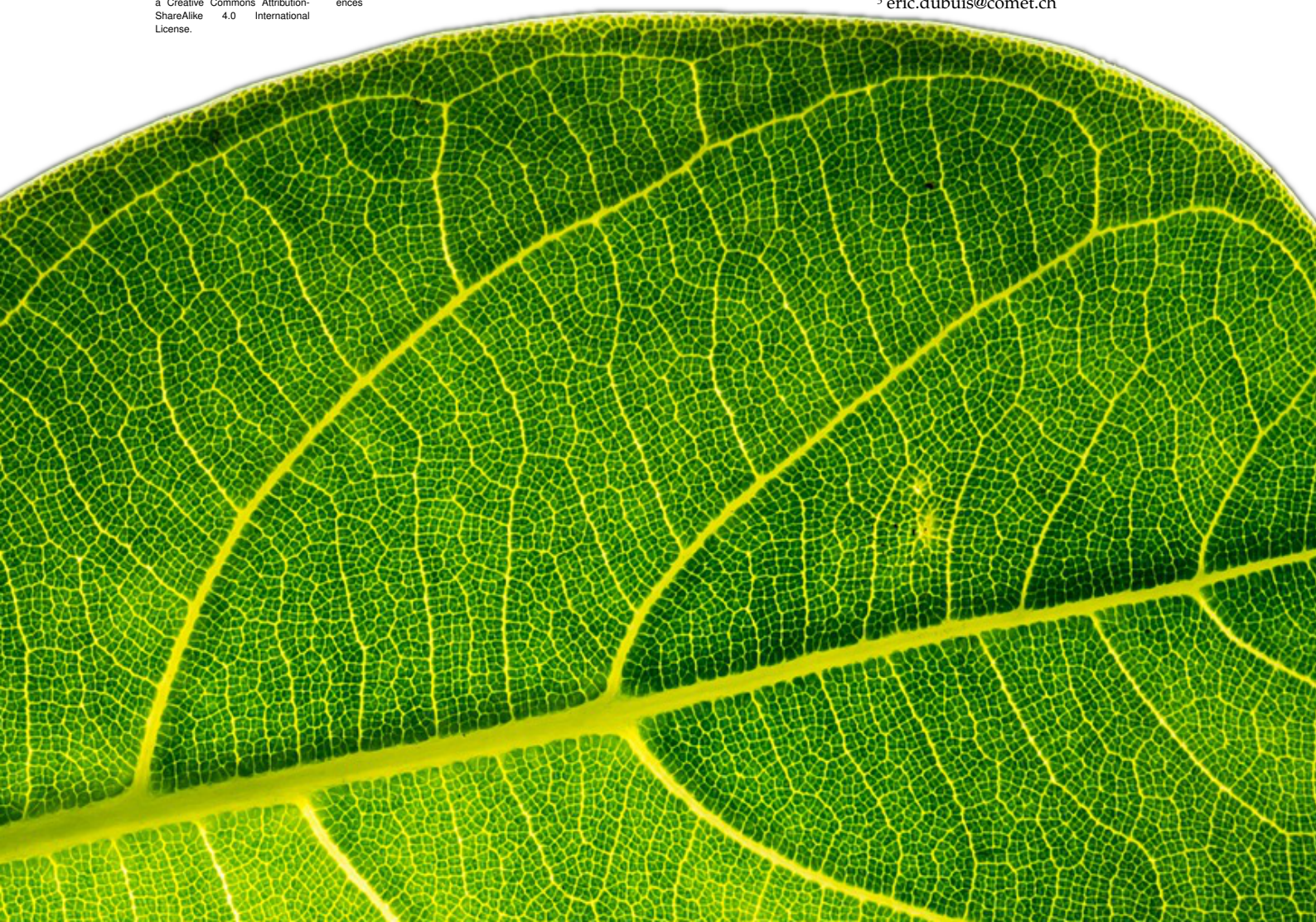
A SYSTEM FOR COMPOSING REAL TIME COMPUTER GRAPHICS.

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

Introduction 10

Administrative aspects 14

Fundamentals 17

Methodologies 25

Implementation 36

Discussion and conclusion 43

Implementation 45

Editor 48

Scene graph 56

List of Figures

1 Schedule of the project. The subtitle displays calendar weeks. 16

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. 19

3 Domain model of the editor component. 19

4 Domain model of the player component. 20

5 Class diagram of the editor component. 21

6 Class diagram of the player component. 22

7 The rendering equation as defined by James "Jim" Kajiya. 22

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*. 23

9 An abstract implementation of the sphere tracing algorithm. Algorithm in pseudo code, after [3][S. 531, Fig. 1] 24

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. 24

11 Illustration showing the processes of *weaving* and *tangling* documents from a input document. [12] 26

12 Declaration of the node definition class. 29

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. 30

14 Head of the method that loads node definitions from external JSON files. 31

15 Check whether the path containing the node definition files exist or not. 31

16 When the directory containing the node definitions exists, files matching the pattern **.node* are searched. 31

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. 32

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. 32
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. 33
20 Output a warning when the path containing the node definition files does not exist. 33
21 Output a warning when no node definitions are being found. 34
22 Phases of the water fall methodology. [17, p. 16] 34
23 Phases of iterative development. [17, p. 16] 34
24 Iterations in the extreme programming methodology and phases of an interation. [17, p. 18] 35

25 Components of the used pattern and their communication. 37
26 Qt's model/view pattern. [21] 38
27 The observer pattern. [18] 40
28 An example of an observer being registered to a signal. 40
29 An example of emitting a signal including a value. 41
30 An example of emitting a signal including a value and a reference to an object. 41

31 Main entry point of the editor application.

Editor → Main entry point 50
32 Main application class of the editor application.

Editor → Application 51
33 Constructor of the editor application class.

Editor → Application → Constructor 51
34 Setting up the internals for the main application class.

Editor → Application → Constructor 52
35 Main window class of the editor application.

Editor → Main window 52
36 Definition of the `do_close` signal of the main window class.

Editor → Main window → Signals 52
37 Definition of methods for the main window class.

Editor → Main window → Methods 53
38 Setting up of components for the main application class.

Editor → Main application → Constructor 53

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

Editor → Main application → Constructor 54

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

Editor → Main window → Methods 55

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

Editor → Scene model 56

42 The constructor of the scene model.

Editor → Scene model → Constructor 57

List of Tables

2 List of the involved persons. 14
3 List of deliverables. 15
4 Phases of the project. 15
5 Milestones of the project. 15

6 Description of the components of the envisaged software. 18
7 Layers of the envisaged software. 20

8 The major commands of nuweb. [13, p. 3] 27
9 Ways of defining scraps in nuweb. [13, p. 3] 27
10 Properties/attributes of the node class. 29

11 Description of the components of the used software design pattern. [19], [20] 37
12 Layers of the developed software. 39

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

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.

Table 2: List of the involved persons.

[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

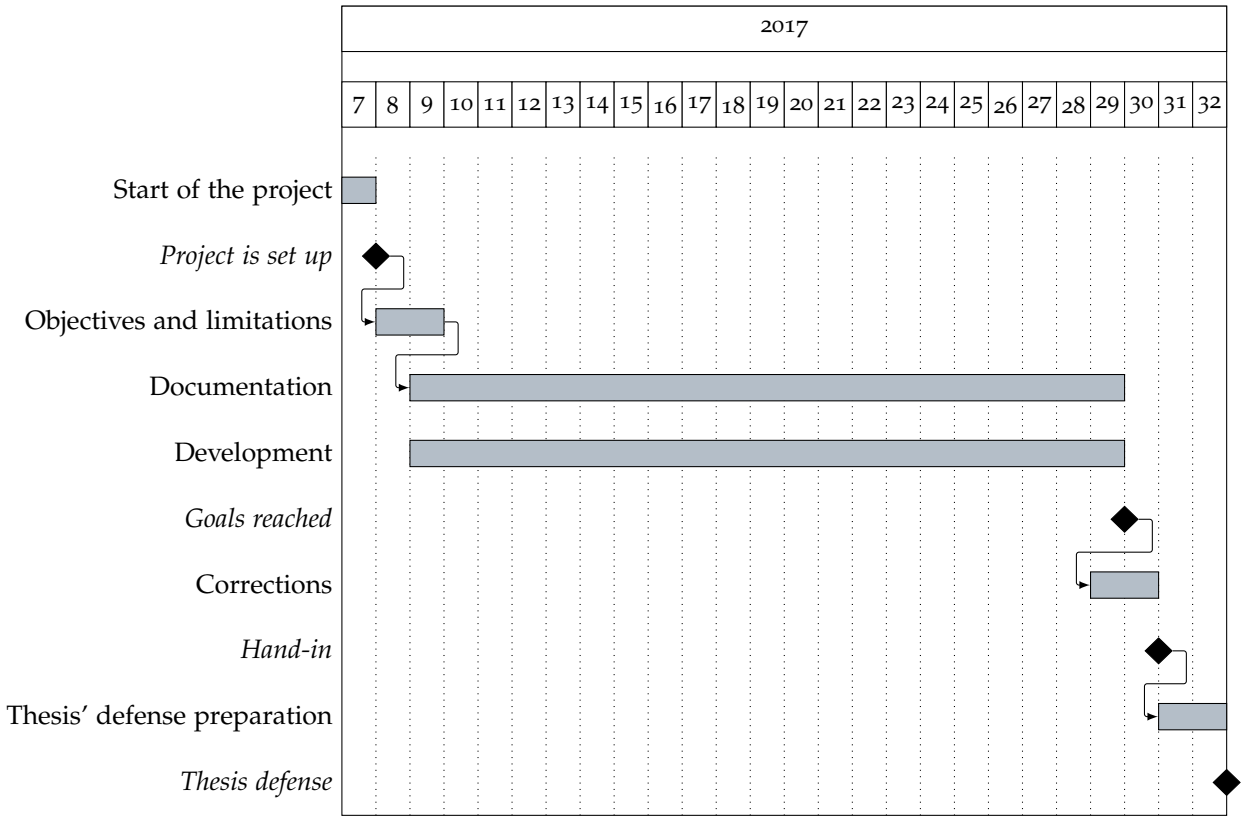


Figure 1: Schedule of the project. The subtitle displays calendar weeks.

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 impor-

tant. [...] 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-

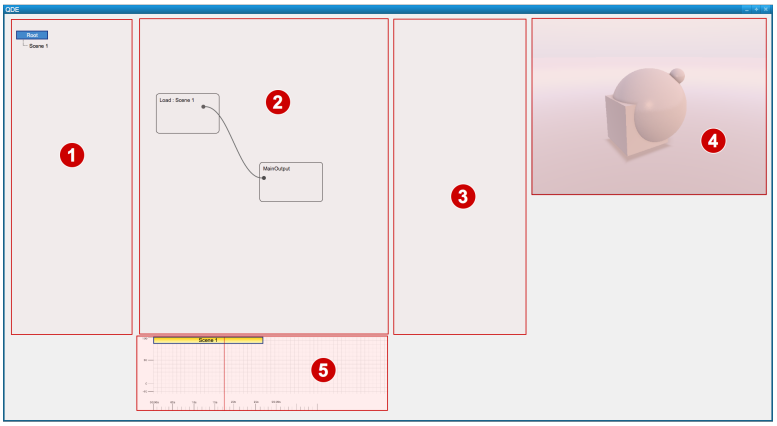


Figure 2: A mock up of the editor application showing its components.
1: Scene graph.
2: Node graph.
3: Parameter view.
4: Rendering view.
5: Time line.

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.

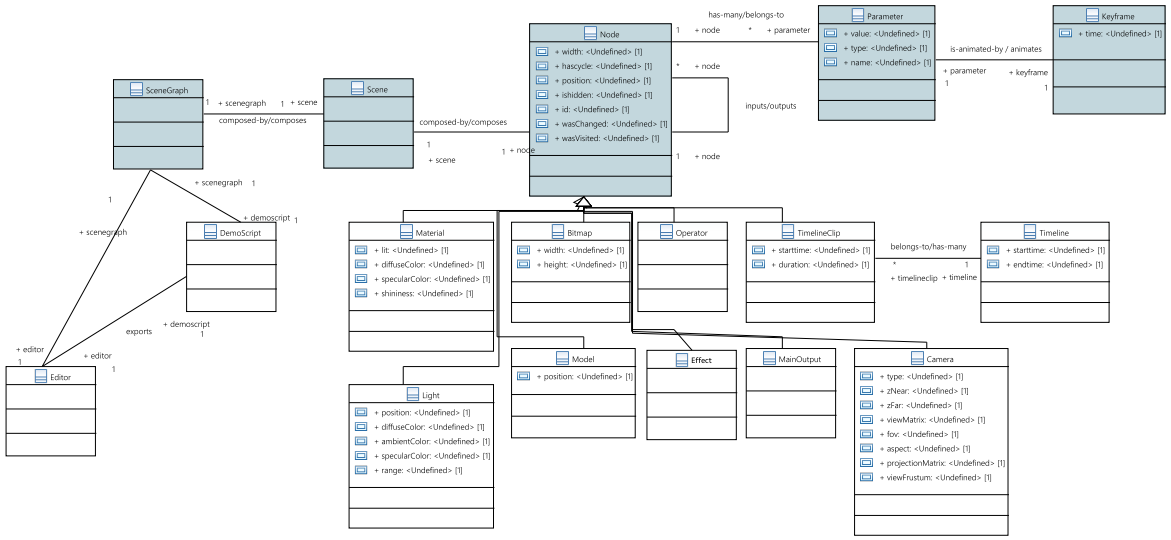


Figure 3: Domain model of the editor component.

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.

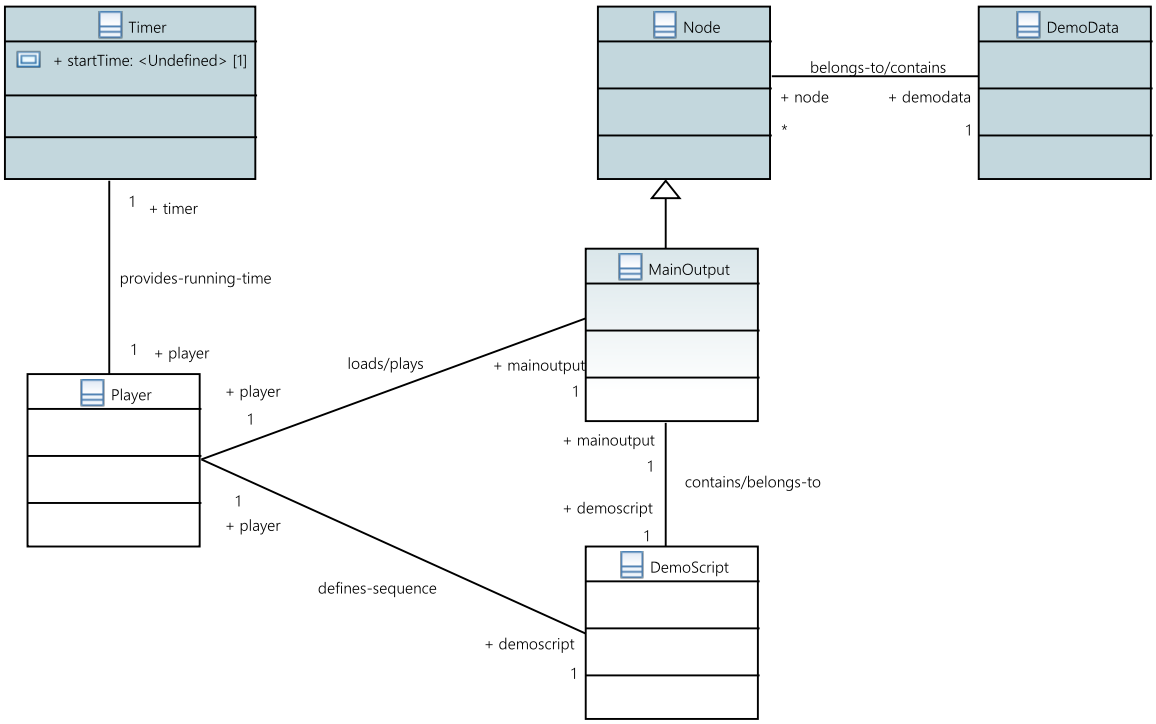


Figure 4: Domain model of the player component.

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.

Table 7: Layers of the envisaged software.

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-

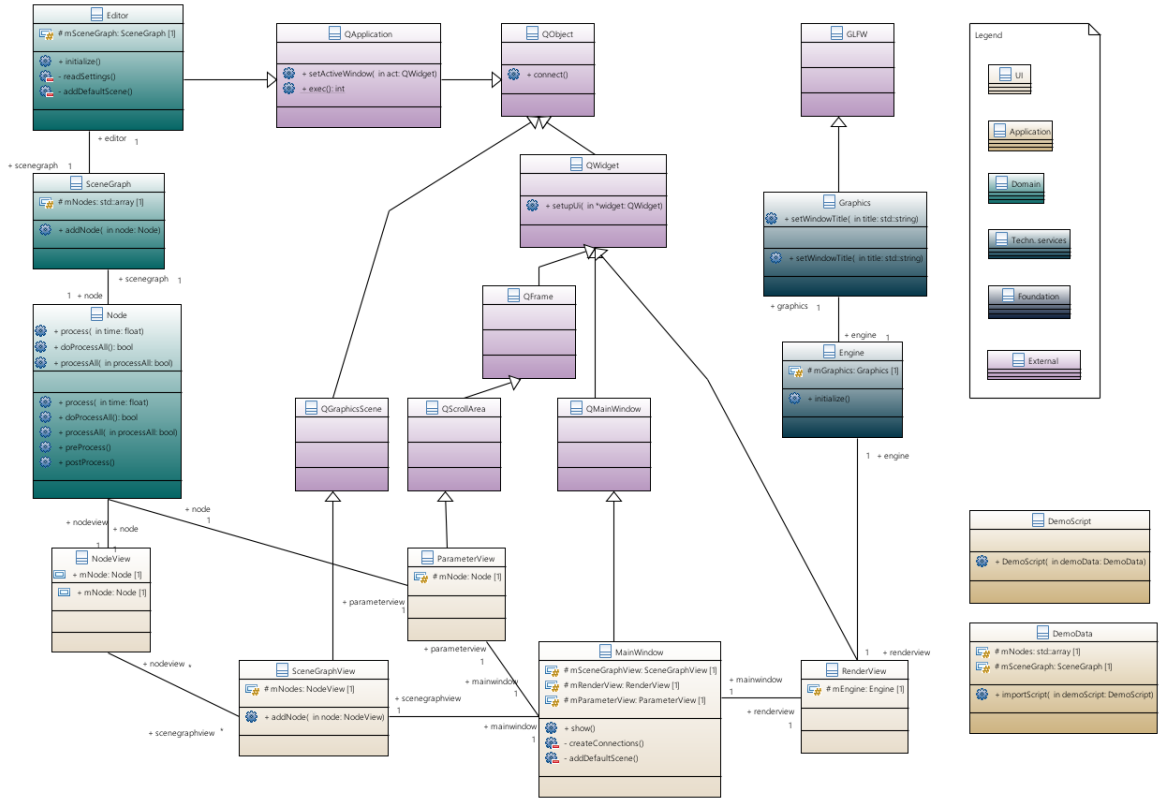


Figure 5: Class diagram of the editor component.

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.

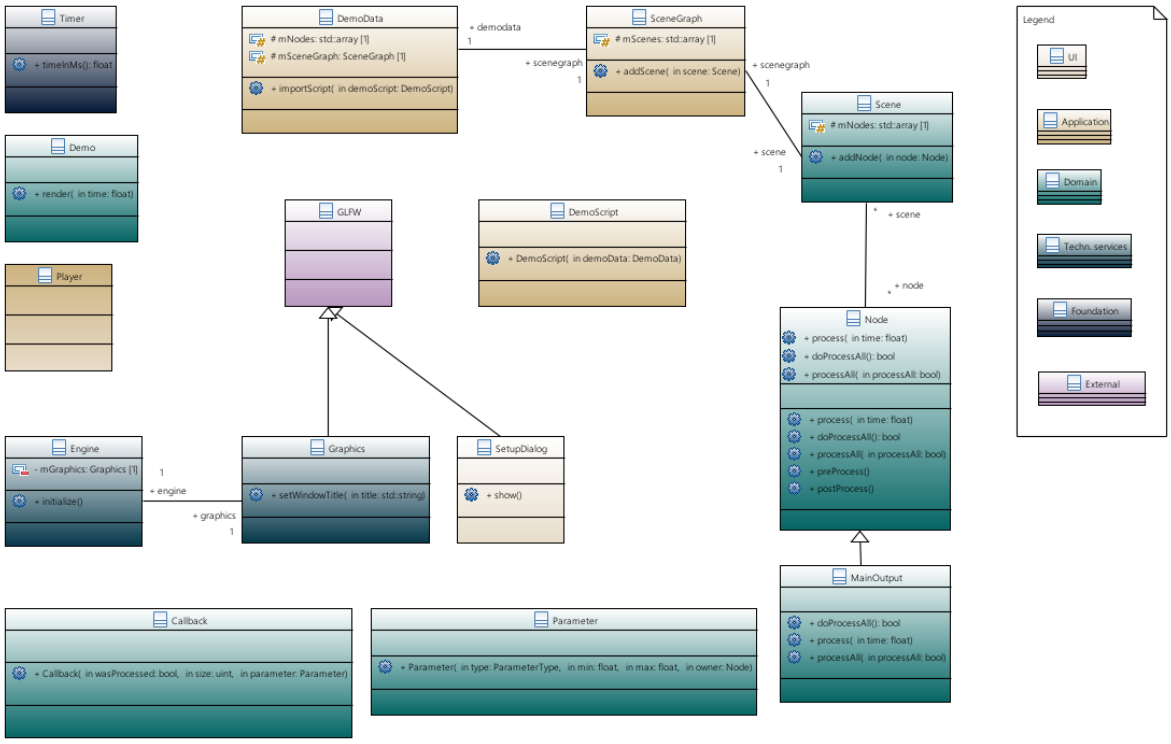


Figure 6: Class diagram of the player component.

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.

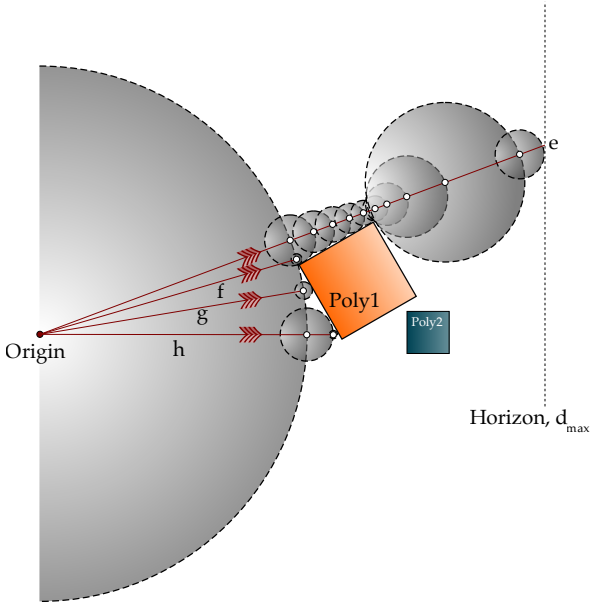


Figure 8: Illustration of the sphere tracing algorithm. Ray e hits no objects until reaching the horizon at d_{max}. Rays f, g and h hit polygon *poly1*.

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.

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

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.

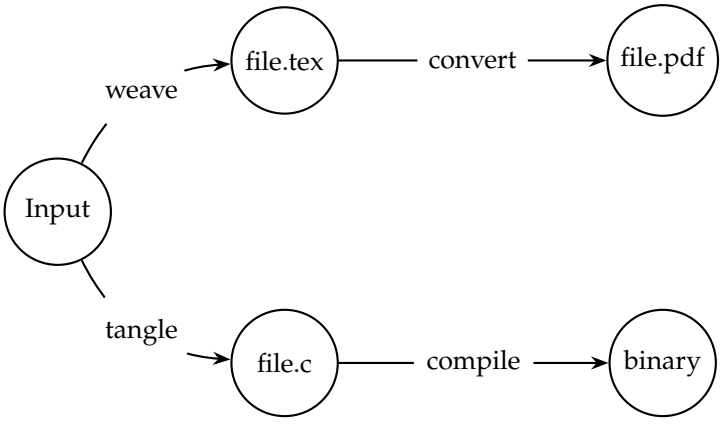


Figure 11: Illustration showing the processes of *weaving* and *tangling* documents from a input document. [12]

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 ?* ⟩ ≡

```
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◇
```

Fragment referenced in ?.

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.

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 is logged.

In the remaining 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.
    """
◇
```

Fragment defined by ?, ?.
Fragment never referenced.

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

⟨ *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 ?⟩◇
```

Fragment defined by ?, ?.
Fragment never referenced.

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

⟨ *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)◇
```

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

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

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 ?⟩◇
```

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

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.

⟨ *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
    )◇
```

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

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.

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)◇
```

Fragment referenced in ?.

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

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.

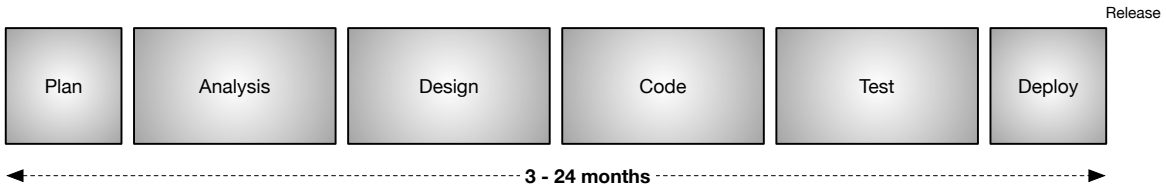


Figure 22: Phases of the water fall methodology. [17, p. 16]

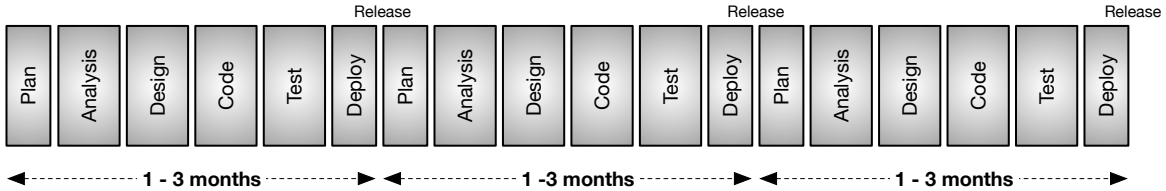


Figure 23: Phases of iterative development. [17, p. 16]

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.

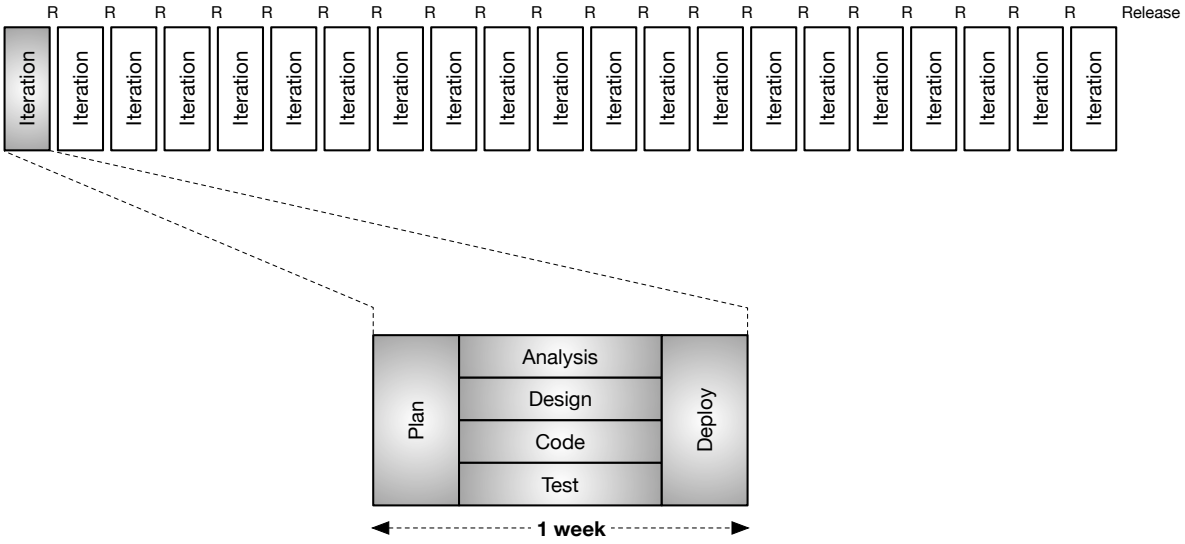


Figure 24: Iterations in the extreme programming methodology and phases of an interation. [17, p. 18]

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. [20] [22]

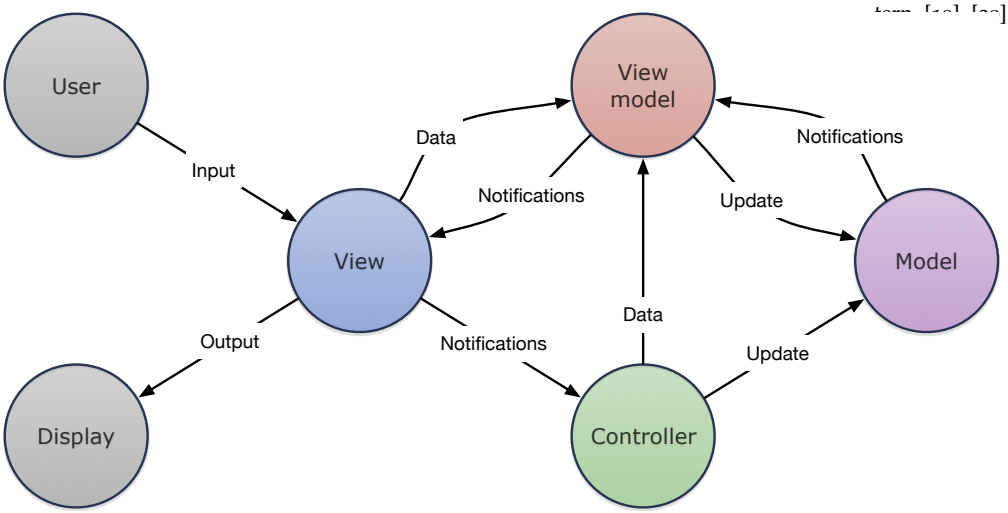


Figure 25: Components of the used pattern and their communication.

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.

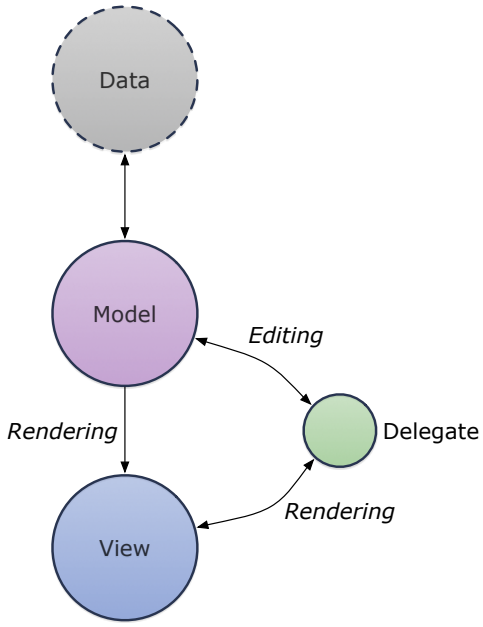


Figure 26: Qt's model/view pattern. [21]

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.

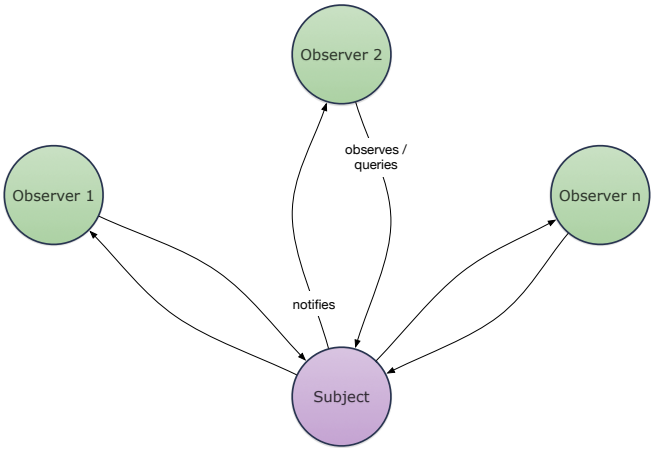


Figure 27: The observer pattern. [18]

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.

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

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

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

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.

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

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

```
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 `'__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)
```

◇

Fragment never referenced.

Figure 31: Main entry point of the editor application.

Editor → Main entry point

rather be confusing. Instead it is more intuitive to have only a minimal entry point which is clearly visible as such. Therefore the main entry point will be put in a file called `editor.py` which is at the top level of the `src` directory.

Main application

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].

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

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

⟨ *Main application declarations ?* ⟩ ≡

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

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

Fragment never referenced.

Figure 32: Main application class of the editor application.

Editor → Application

⟨ *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()◇
```

Fragment referenced in ?.

Figure 33: Constructor of the editor application class.

Editor → Application → Constructor

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 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)
◇
```

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

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

Editor → Main window → Methods

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  ?⟩◇
```

Fragment referenced in ?.

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

Editor → Main application → Constructor

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

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 not provide enough possibilities. Instead a horizontal box layout in combination with splitters is used.

Recalling the components, the following layout is approached:

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

⟨ *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 ?⟩◇
```

Fragment referenced in ?.

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

Editor → Main application → Constructor

- 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 ?* ⟩+ ≡

```
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  ?⟩◇
```

Fragment never referenced.

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

Editor → Scene model

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.

⟨ *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◇
```

Fragment referenced in ?.

Figure 42: The constructor of the scene model.

Editor → Scene model → Constructor

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