QDE.

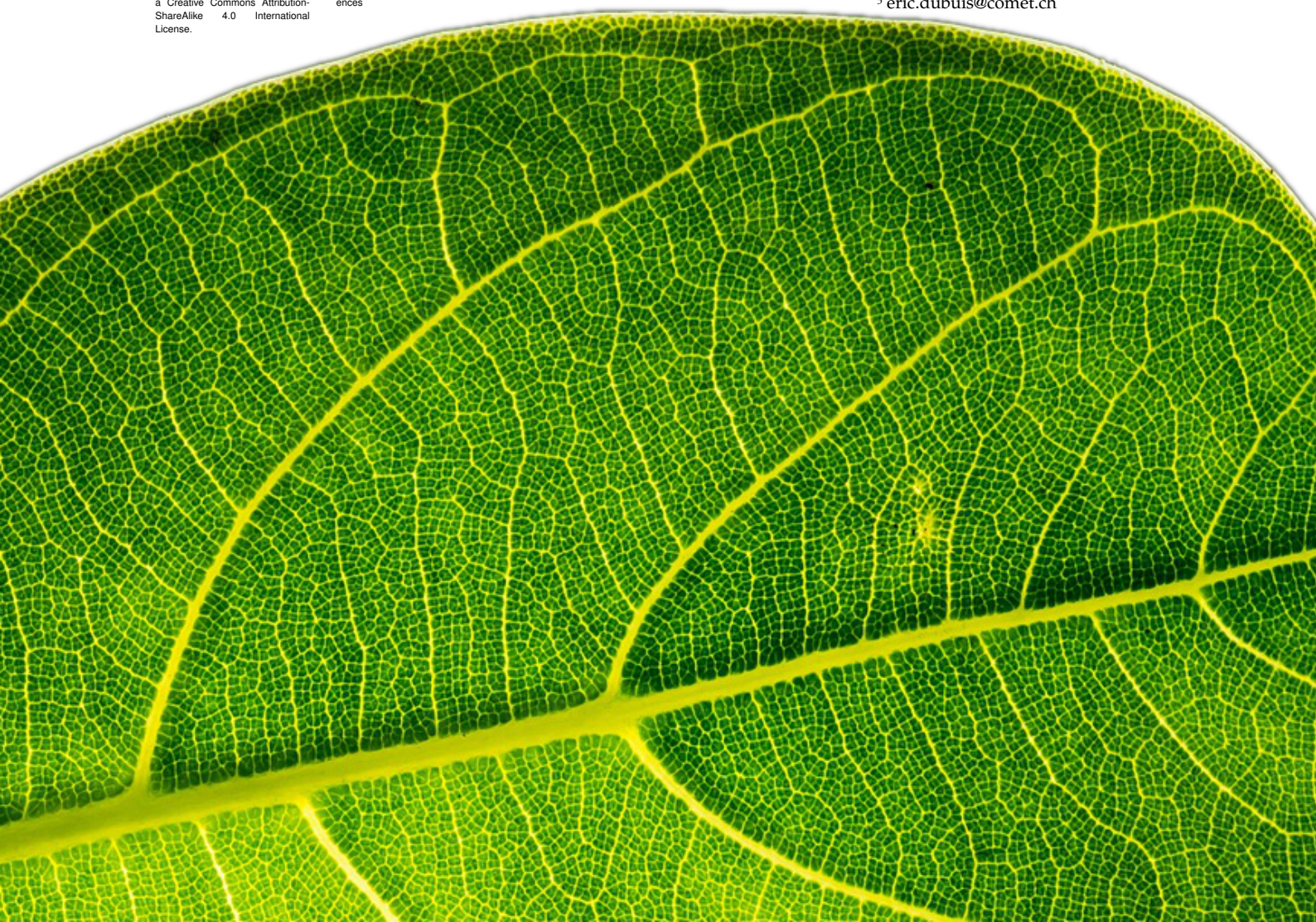
A SYSTEM FOR COMPOSING REAL TIME COMPUTER GRAPHICS.

MTE7103 — MASTER THESIS

Major:	Computer science
Author:	Sven Osterwalder[1]
Advisor:	Prof. Claude Fuhrer[2]
Expert:	Dr. Eric Dubuis[3]
Date:	2017-05-29 09:10:48
Version:	399669d

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

Abstract

Provide correct abstract.

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

Contents

List of Figures

List of Tables

Introduction

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

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

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

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

Purpose and situation

Motivation

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

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

Objectives and limitations

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

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

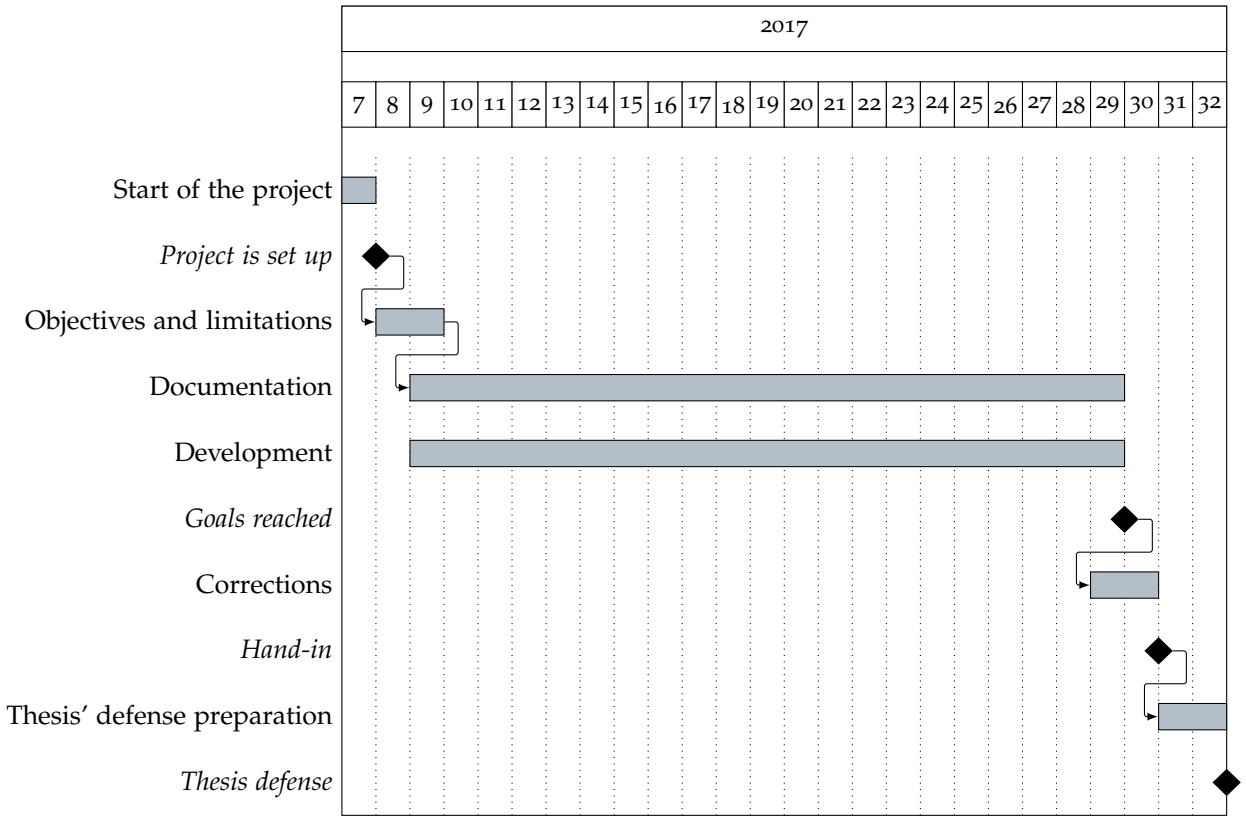


Figure 1: Schedule of the project. The subtitle displays calendar weeks.

Fundamentals

The last 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-

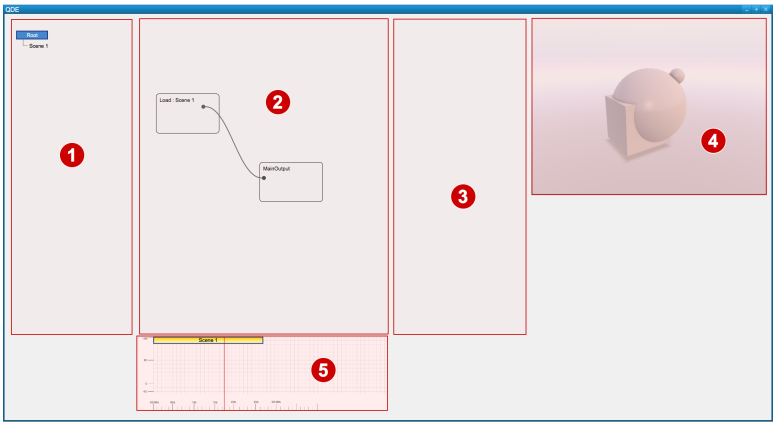


Figure 2: A mock up of the editor application showing its components.
1: Scene graph.
2: Node graph.
3: Parameter view.
4: Rendering view.
5: Time line.

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.

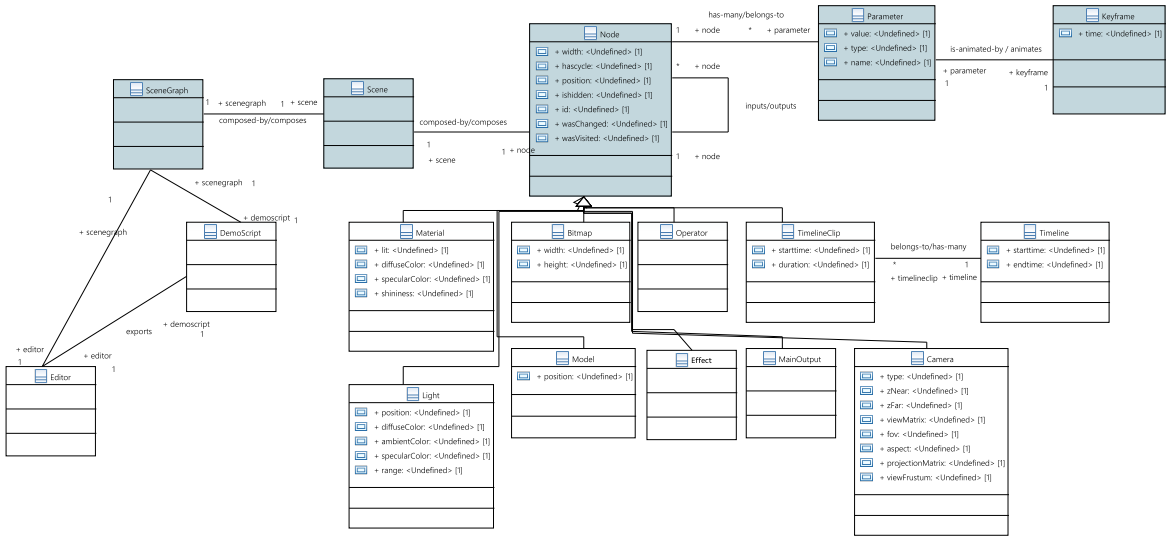


Figure 3: Domain model of the editor component.

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.

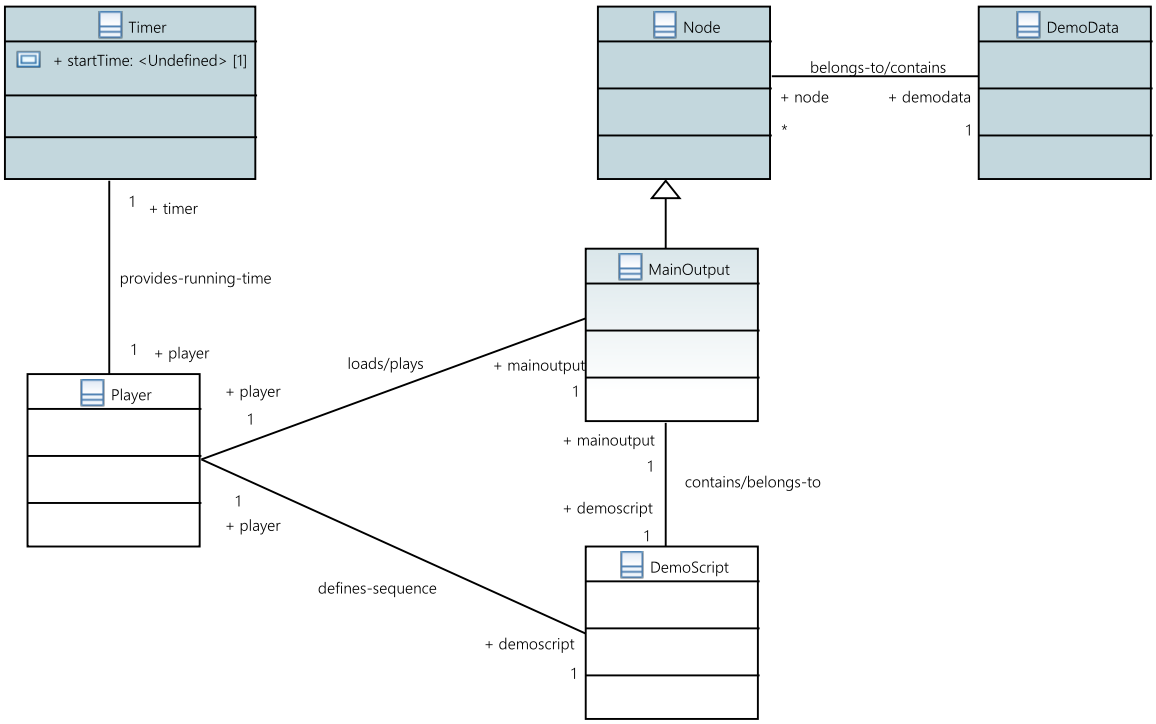


Figure 4: Domain model of the player component.

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-

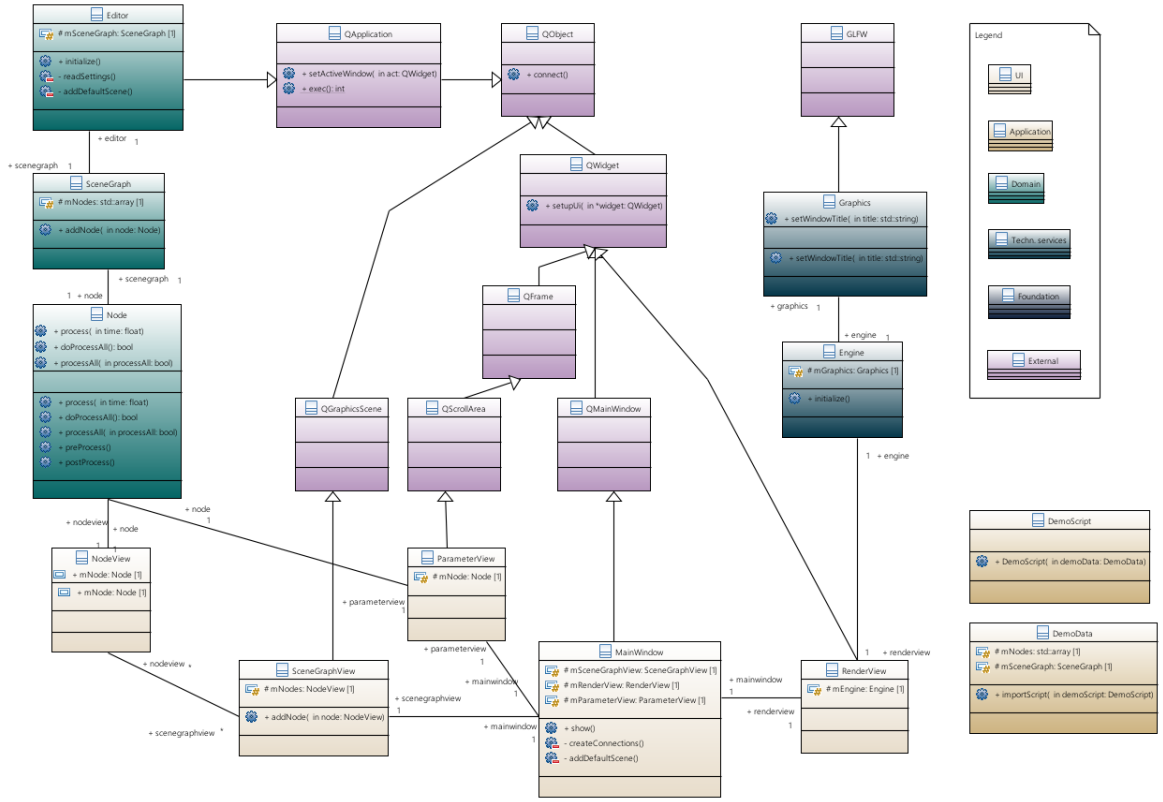


Figure 5: Class diagram of the editor component.

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.

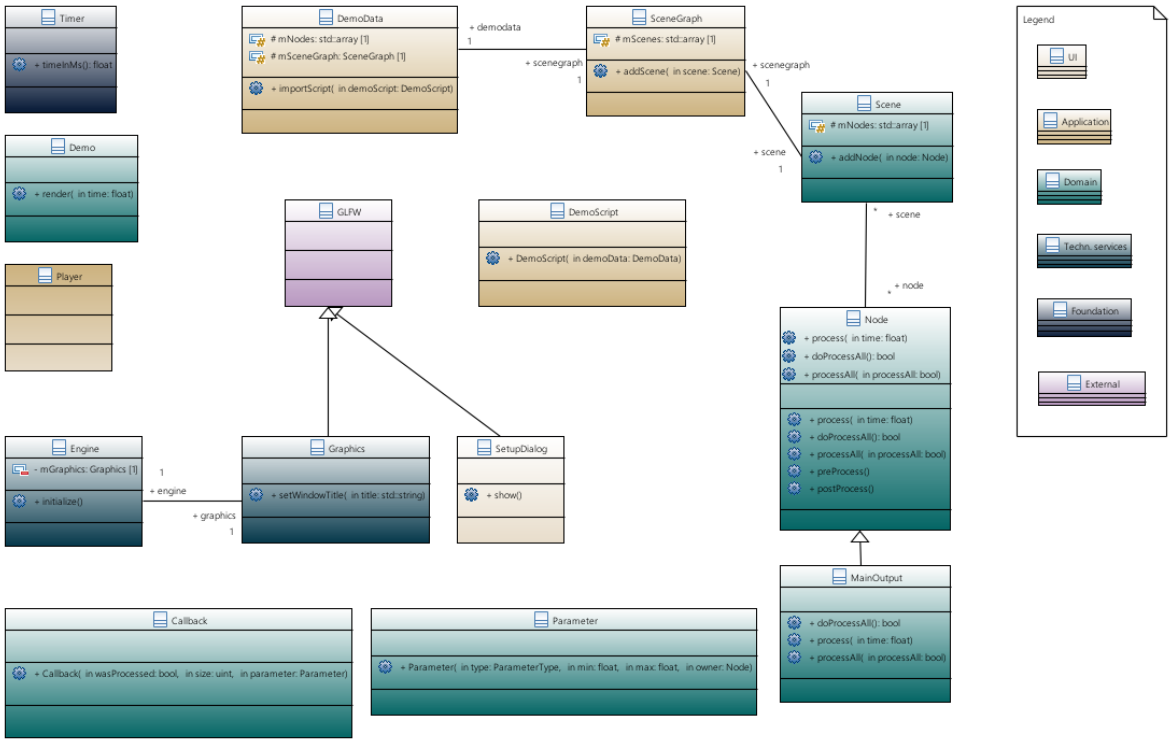


Figure 6: Class diagram of the player component.

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.

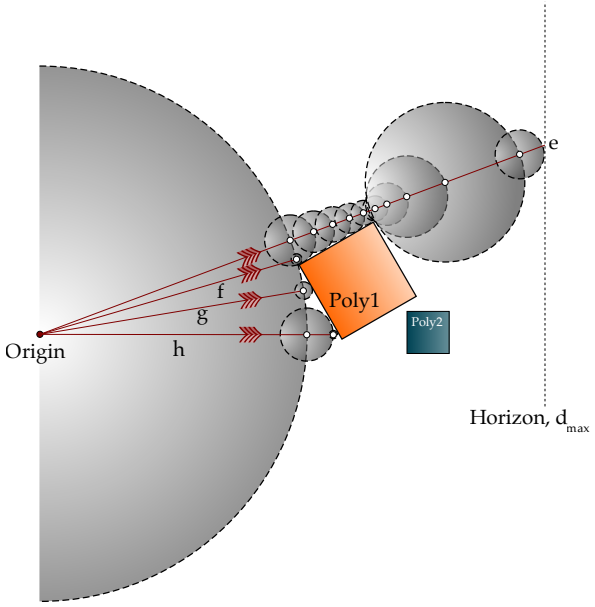


Figure 8: Illustration of the sphere tracing algorithm. Ray e hits no objects until reaching the horizon at d_{max}. Rays f, g and h hit polygon *poly1*.

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 last 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.

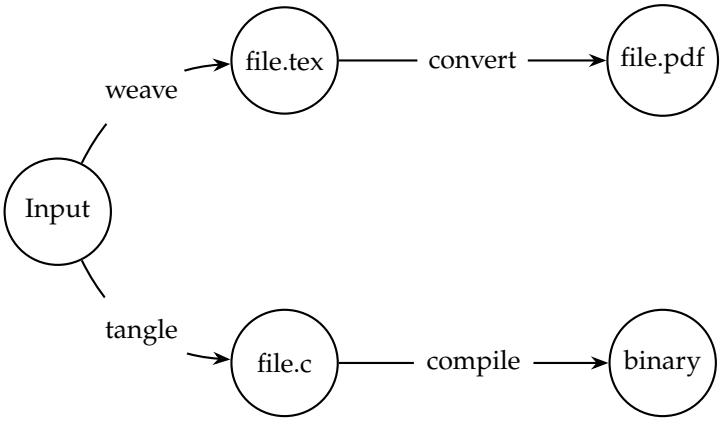


Figure 11: Illustration showing the processes of *weaving* and *tangling* documents from a input document. [12]

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 ?⟩◇
```

Fragment never referenced.

Figure 12: Declaration of the node definition class.

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.

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.

⟨ *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.

When loading the node definitions, the first thing to do is to check whether the path containing the files with the node definitions exist or not. Is this not the case a warning message will be logged.

Literate programming allows great verbosity as can be seen

⟨ *Load node definitions ?* ⟩+ ≡

```
    if os.path.exists(self.nodes_path):
        ⟨ Load existing node definitions ?⟩
    else:
        ⟨ Output warning when no node definitions are found 29⟩◇
```

Fragment defined by ?, ?.
Fragment never referenced.

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

in fig. 15. Logging the warning message is simply a matter of preparing a corresponding message and calling the `fatal` method of the logging interface.

⟨ *Output warning when no node definitions are found* 29 ⟩ ≡

```
    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 16: Output a warning when the path containing the node definition files does not exist.

Agile software development

TBD.

Results

Write chapter.

Discussion and conclusion

Write chapter.

fix appendix

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.

Fix glossaries

Print index